Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. Bernhard Mitschang
Betreuer:	Dipl.-Inform. Aiko Frank
begonnen am:	17.11.2000
beendet am:	16.5.2001
CR-Klassifikation:	H.4.1, H.5.3, J.6

Diplomarbeit Nr. 1896

Erstellung eines Constraints-basierten Aktivitätenmanagers für CASSY

Frank Wagner

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Computer setzen sich in der Arbeitswelt immer mehr durch. Angefangen bei zentralen Datenbanksystemen über die dezentrale Erstellung von Dokumenten wird nun die Vernetzung der einzelnen Systeme immer wichtiger.

Auch beim Entwurf von komplexen Systemen wie z.B. einem Auto oder Software bestand die Unterstützung der Entwerfer früher aus einer Menge einzelner Werkzeuge, die jeweils einen abgeschlossenen Entwurfsschritt realisierten. Durch Design-Management-Systeme kann mittlerweile aber der Ablauf des Entwurfs vordefiniert und vom Computer entsprechend unterstützt werden.

Diese Systeme unterstützen aber im wesentlichen den Teil der Arbeit eines Entwerfers, der direkt mit dem Entwurf und dem Entwurfsobjekt zu tun hat. Darüber hinausgehende Arbeiten wie z.B. Besprechungen werden von diesen Systemen nicht betrachtet. Solche Arbeiten werden entweder ohne Computerunterstützung durchgeführt oder es werden eigenständige Groupware-Systeme verwendet.

Für eine umfassende Unterstützung von verteilten Teams bei der Lösung ihrer Aufgaben ist eine Integration sowohl des koordinativen Aspekts der Ablaufsteuerung als auch des kooperativen Aspekts von Groupware durchaus sinnvoll. Bisherige Ansätze sind dabei immer von einer der beiden Seiten ausgegangen und haben ein existierendes System dahingehend erweitert. Im Projekt ASCEND (Activity Support in Co-operative ENvironments for Design issues) wird nun versucht beide Aspekte möglichst gleichwertig in ein System zu integrieren.

1.1 Aufgabenstellung

Im Rahmen dieser Diplomarbeit soll ein Scheduling und Management von Aktivitäten implementiert werden, das auf Constraints basiert (z.B. Aktivität A darf nur nach Aktivität B ausgeführt werden). Dafür ist die Entwicklung einer Menge von geeigneten Constraints notwendig. Beim Aktivitäten-Management ist die Initialisierung und Versorgung der Aktivitäten für einen Start zu berücksichtigen.

1.2 Überblick

Die vorliegende Arbeit beschreibt zunächst das Einsatzgebiet des Aktivitäten-Managements und baut dieses dann von unten, einem formalen Modell, nach oben auf.

In **Kapitel 2** wird zunächst das Forschungsgebiet Computer Supported Cooperative Work (CSCW) beschrieben. In **Kapitel 3** wird dann gezeigt, wie mit dem im Projekt ASCEND entwickelten System CASSY Teilaspekte von CSCW unterstützt werden können. In **Kapitel 4** wird das formale Modell beschrieben, auf dem aufbauend dann in **Kapitel 5** die Implementierung der EventEngine beschrieben wird. Wie mit Hilfe dieser EventEngine dann ein Constraints-basiertes Aktivitäten-Management realisiert werden kann wird in **Kapitel 6** beschrieben.

Den Schluss der Arbeit bildet eine Zusammenfassung und Bewertung der Arbeit, der Implementierung in CASSY und schließlich ein Ausblick.

Kapitel 2

Computer Supported Cooperative Work

Computer Supported Cooperative Work (CSCW) ist ein Forschungsgebiet mit dem Ziel, die Kooperation zwischen Menschen möglichst effizient durch Computer zu unterstützen. In [Bur97] wird diese Unterstützung in drei Aspekte gegliedert, die auch in Abbildung 2.1 dargestellt sind:

Kommunikationsunterstützung: Die Kommunikationsunterstützung behandelt den Transport von Informationen. Die Informationen können z.B. asynchron per elektronischer Post oder synchron per HTTP ausgetauscht werden. Weitere Verfahren sind gemeinsame Arbeitsbereiche oder das in CASSY verwendete CORBA (siehe Abschnitt 3.6).

Koordinationsunterstützung: Dieser Aspekt betrachtet die Koordination von Menschen und Abläufen. Eine Möglichkeit der Unterstützung sind hier elektronische Kalender, die durch selbstständige Aushandlung mit anderen Kalendern die Termine ihrer Eigentümer koordinieren. Sie stimmen also die einzelnen Tätigkeiten noch vor ihrer Ausführung aufeinander ab. Während ihrer Ausführung können nebenläufige Tätigkeiten durch Workflow-Systeme (siehe in Abschnitt 2.2) koordiniert werden.

Kooperationsunterstützung: Hier wird die gemeinsame, parallele Arbeit mehrerer Personen unterstützt. Dabei muss die Kooperation nicht synchron wie z.B. bei den Konferenzsystemen geschehen, sondern sie kann auch asynchron erfolgen. Beispiele dafür sind Mehrbenutzereditoren, die die gemeinsame Erstellung z.B. eines Buches unterstützen, oder auch Versionsverwaltungssysteme wie CVS oder Microsoft Visual SourceSafe.

Wie schon erwähnt wird als Basismechanismus zur Kommunikation in ASCEND CORBA verwendet. Darauf aufbauend geschieht die eigentliche Kommunikation dann aber entweder durch Nachrichtenaustausch zwischen einzelnen Komponenten oder über gemeinsame Arbeitsbereiche (workspaces). Für die Koordination und Kooperation werden in ASCEND bereits existierende

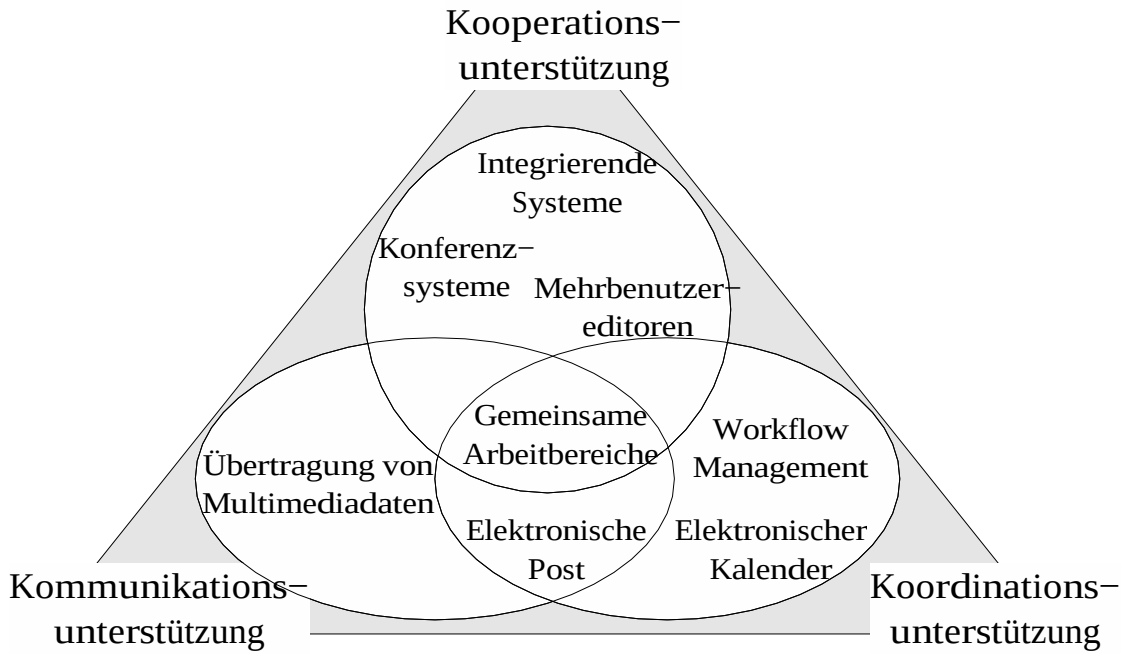

Abbildung 2.1: Klassifikation von Groupware nach [Bur97]

Dienste (facilities) eingesetzt, die mit Hilfe von Designflows zu größeren Abläufen zusammengesetzt werden können. Diese drei Dienste werden in den folgenden Abschnitten einzeln beschrieben.

2.1 Groupware-Systeme

Der Begriff "Groupware" ist hier enger gefasst als in [Bur97]. Burger bezeichnet mit Groupware die Soft- und Hardware, mit der CSCW, also alle Aspekte des Dreiecks in Abbildung 2.1, unterstützt werden. In ASCEND wird unter Groupware speziell der kooperationsunterstützende Aspekt von CSCW betrachtet.

Mit der Groupware-Facility wurde von Muchitsch ein Grundgerüst (Framework) erstellt, das als Basis für die Entwicklung von Groupware-Software verwendet werden kann. Der Fokus lag dabei auf einem möglichst allgemeinen Modell, durch das die darauf aufbauenden Anwendungen in ihrer Kooperationsform möglichst nicht eingeschränkt werden.

Wie in Abbildung 2.2 zu sehen ist, sind nur die grundlegenden Klassen und Beziehungen zwischen diesen Klassen vorgegeben. Dieses Modell wurde in CASSY im wesentlichen übernommen. Auch hier gibt es Benutzer (*User*), die zum einen zu Gruppen (*Group*) zusammengefasst und zum anderen unterschiedliche Rollen (*Role*) einnehmen können. Einer Gruppe ist ein gemeinsamer Arbeitsbereich (*Workspace*) zugeordnet, in dem die von der Gruppe für die Erledigung ihrer Arbeit benötigten Ressourcen (*Resources*) und Aufgaben (*Task*) verwaltet werden.

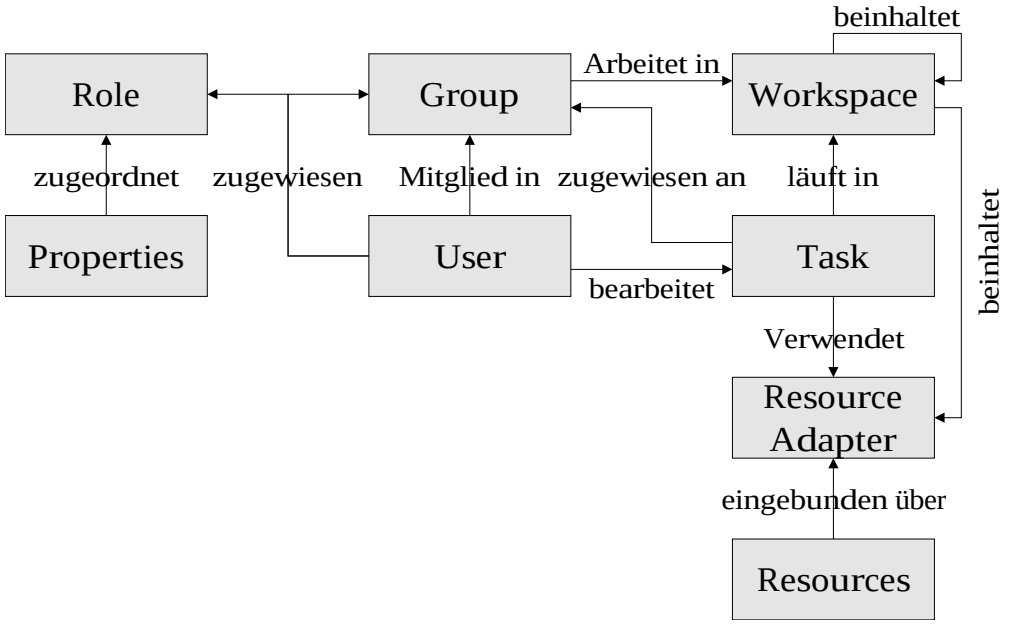

Abbildung 2.2: Klassen und Beziehungen in der Groupware-Facility nach [Muc98]

Einen Unterschied gibt es bei der Klasse *Task*, die mit den Aktivtäten im ASCEND-Modell verglichen werden kann. Aktivitäten werden in der Regel nur ein Mal von einem Benutzer gestartet, wohingegen die Benutzer mehrfach die Arbeit an einem Task aufnehmen können. Dabei wird vom System jedes Mal die zugehörige Anwendung neu gestartet. Die Anwendungen werden in zwei Klassen eingeteilt: in die Groupware-Facility-bewussten Anwendungen, die direkt auf dem System aufgebaut wurden (white-box-integration), und die nicht-Groupware-Facility-bewussten Anwendungen, die über einen Proxy angebunden werden (black-box-integration).

Als Anwendungsszenarien beschreibt Muchitsch einen Mehrbenutzereditor, ein Videokonferenzsystem und ein E-Mail System. Dadurch, dass die eigentliche Kooperation im Modell offen gelassen wurde, sollten hier aber beliebige Kooperationsformen eingebunden werden können.

2.2 Workflow-Management

Auch mit Workflow-Management-Systemen (WfMS) soll die Zusammenarbeit von mehreren Benutzern an einer gemeinsamen Aufgabe unterstützt werden. Allerdings steht hier nicht der Austausch bzw. die Erstellung von Informationen sondern der koordinierte Ablauf der einzelnen Schritte im Vordergrund. Die Abläufe müssen dazu vor der Ausführung modelliert werden, was das Einsatzgebiet von WfMS auf Aktivitäten mit starker Strukturierung und geringer

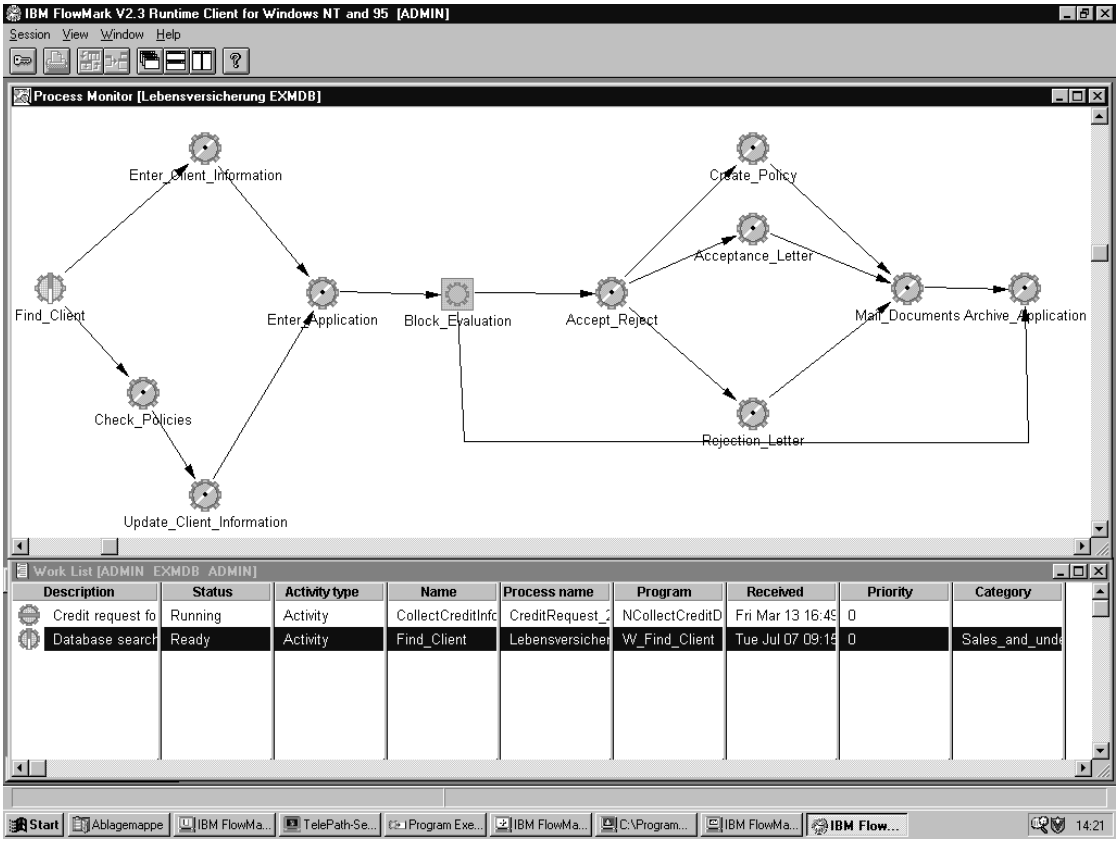

Abbildung 2.3: Der Client von IBM FlowMark

Veränderung einschränkt. Mögliche alternative Abläufe müssen explizit modelliert werden.

Als Beispiel für ein WfMS ist in Abbildung 2.3 eine Aufnahme des Runtime-Clients von IBM FlowMark zu sehen. Im unteren Teil des Bildes wird die Worklist des Benutzers angezeigt, in der die Aktivitäten eingetragen sind, die der Benutzer ausführen kann. Im oberen Teil des Fensters wird ein Workflow grafisch dargestellt. Die Zahnräder repräsentieren einzelne Sub-Aktivitäten und die Pfeile legen den Ablauf zwischen diesen Aktivitäten fest.

Die Voraussetzung für den Einsatz eines WfMS ist eine Analyse und Modellierung der Abläufe im Unternehmen. Dabei werden die Prozesse in der Regel auch überarbeitet, um sie an die neuen Bedingungen anzupassen und ihre Effizienz zu steigern (Business Process Reengineering). Mit einer einfachen, grafischen Notation können die Abläufe auch von geschulten Experten im Anwendungsbereich (Management, Wirtschaftswissenschaftlern, Kaufleuten, Bankern) modelliert werden. Das WfMS treibt (pusht) dann die Aufgabe entsprechend dem Modell durch das Unternehmen und stellt den jeweils zuständigen Bearbeitern die von ihnen zu erledigenden Aufgaben mit allen notwendigen Daten zu. Diese Abläufe müssen nun also nicht mehr von der IT-Abteilung implementiert werden.

Die einzelnen Schritte in einem Workflow werden entweder durch Sub-Workflows oder durch Anwendungsfunktionen realisiert. Diese Anwendungsfunktionen werden in der Regel wie gewöhnliche Anwendungen von Programmierern implementiert. Über Wrapper können auch alte Systeme (legacy systems) weiter verwendet werden. Dabei werden in der Regel die einzelnen Geschäftsfunktionen einer Anwendung jeweils über einen eigenen Wrapper für die Verwendung in Workflows zur Verfügung gestellt.

Der Kontroll- und Datenfluss, der über einzelne Geschäftsfunktionen hinausgeht, muss nun nicht mehr direkt in die Programme einprogrammiert werden. Durch die Trennung von Fluss und Funktion kann das System auch einfach und schnell an geänderte Anforderungen angepasst werden. Im Idealfall muss nur das Modell von den Fachleuten für die Abläufe modifiziert werden und die Änderungen müssen damit nicht mehr an die IT-Abteilung weitergegeben werden, die oft die eigentliche Abläufe und deren Randbedingungen gar nicht kennen.

Die Workflow Management Coalition (WfMC) hat einen Standard [WfM95] definiert, mit dem die Iteroperabilität von WfMS unterschiedlicher Hersteller und damit die Verbreitung von WfMS gefördert werden soll. Darauf aufbauend hat die OMG eine CORBA-Schnittstelle [OMG00] spezifiziert.

Die im Rahmen des Projektes ASCEND entwickelte Workflow-Facility basiert nicht auf diesem Standard sondern noch auf einer Vorab-Version (jFlow, [OMG97]). Sie leitet die Anfragen weiter an das WfMS IBM FlowMark, einen Vorgänger von IBM MQSeries Workflow.

2.3 Designflow-Management

Das Designflow-Management soll Entwerfer durch eine integrierte Entwurfsumgebung bei ihrer Arbeit unterstützen. Dabei beschränkt man sich nicht auf ein spezielles Einsatzgebiet. Für diese allgemeine Entwurfsunterstützung hat sich die Abkürzung CAX für Computer Aided X eingebürgert, wobei X ein Platzhalter für die Einsatzgebiete ist. Zwei mögliche Einsatzgebiete sind Design (CAD) und Software Engineering (CASE). Beispiele für Entwurfsobjekte in diesen Einsatzgebieten sind Computerchips, Motoren, Fahrräder und Software.

Abbildung 2.4 zeigt den Aufbau einer CAX-Entwurfsumgebung. Das Framework bietet den Werkzeugen, die selbst nicht zum Framework gezählt werden, unterschiedliche Dienste an:

- Die Benutzungsoberfläche sorgt für ein einheitliches Look-and-Feel der Werkzeuge. Dazu gehört unter anderem das Aussehen der Bedienelemente, die Bedienung dieser Elemente, die Anordnung von Menüpunkten und die Belegung von Tastenkombinationen mit Funktionen.

- Das Design-Management koordiniert die Abläufe zwischen den einzelnen Werkzeugen. Auf diese Komponente konzentriert sich das Projekt ASCEND.

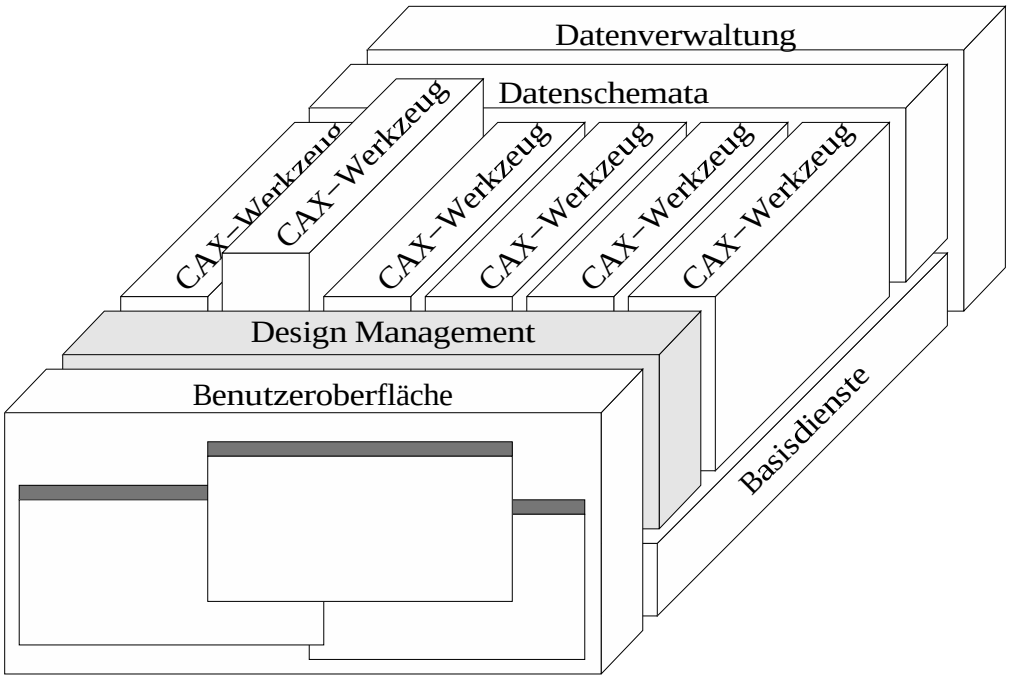

Abbildung 2.4: CAX Framework Structure nach [Bai98]

- Die Datenverwaltung behandelt die Speicherung der Daten. Sie legt die Daten z.B. in einer Datenbank oder in Dateien ab.

- Mit den Datenschemata werden die Daten des Einsatzgebietes modelliert. Hier wird ein konzeptionelles Modell für das Entwurfsobjekt aufgestellt. Den einzelnen Werkzeugen werden die jeweils von ihnen benötigten Teile des Entwurfsobjektes zur Verfügung gestellt.

- Die Basisdienste bilden eine Abstraktionsschicht über dem Betriebssystem und bieten z.B. Prozessverwaltung und Kommunikation zwischen Prozessen. Durch diese Schicht wird auch die Portabilität der Entwurfsumgebung zwischen verschiedenen Betriebssystemen ermöglicht.

Durch das Design-Management bietet ein CAX-Framework für den Entwerfer eine Unterstützung, die über einzelne Werkzeuge hinausgeht. Es soll den Entwerfer entsprechend der gewählten oder vorgegebenen Entwurfsmethodik führen. Dazu präsentiert es dem Bearbeiter die anstehenden Aufgaben und ruft dann das jeweilige Werkzeug mit den aktuellen Daten auf. Das sieht aus wie bei Workflow-Systemen, durch den Einsatz im Entwurf ergeben sich aber Unterschiede:

- Während der Ausführung des Projektes können zusätzliche Teil-Projekte notwendig werden, die anfangs nicht vorgesehen waren. Diese sind daher auch nicht im anfangs definierten Modell enthalten. Zu einer Aktivi-

tät müssen also dynamisch weitere Sub-Aktivitäten hinzugefügt werden können.

- Auch wenn ein Arbeitsschritt schon von Anfang an feststeht, kann die Art der Umsetzung dennoch von der aktuellen Situation abhängig sein. Die Wahl einer konkreten Aktivität zur Realisierung einer Aufgabe muss also aufschiebbar sein.

- Entwerfer mit viel Erfahrung sind eine teure Ressource für ein Unternehmen. Diese Entwerfer wollen aber oft in ihrer Kreativität nicht durch ein System eingeschränkt werden. Daher soll ein Design-Management mehr eine führende als eine kontrollierende Rolle spielen. In [RM97] wird dies die Assistenzfunktion von Designflows genannt.

- Ein Ablaufgraph ist im Allgemeinen zu restriktiv um Entwurfsabläufe zu modellieren. Es bestehen oft nur einige wenig Restriktionen zwischen den Aktivitäten.

Neben der Flexibilität auf Modellierungsebene durch alternative Ausführungspfade wie bei Workflow-Systemen muss hier also auch eine Flexibilität auf Instanzenebene möglich sein. Zudem werden entsprechend dem letzten Punkt zur Modellierung eines Designablaufs eher deskriptive als präskriptive Kontrollflusskonstrukte verwendet.

Kapitel 3

ASCEND und CASSY

ASCEND ist ein Projekt, das von Professor Mitschang und dem Betreuer dieser Arbeit Aiko Frank an der Technischen Universität München initiiert wurde und nach deren Wechsel zur Universität Stuttgart dort fortgesetzt wird.

Das Ziel des Projektes ist die Entwicklung eines Modells in dem Goupware- und Workflow-Aktivitäten zusammengefasst werden. Letztendlich soll ein System (CASSY) implementiert werden, das ein verteiltes Team von Entwerfern möglichst effizient und umfassend bei seiner Arbeit unterstützt.

In Abbildung 3.1 ist der grobe Aufbau von ASCEND zu sehen. Im Zentrum ist der Informationsraum, in dem alle Objekte für alle Benutzer und Aktivitäten zugreifbar abgelegt sind. Solche Informationsräume haben ihren Ursprung bei Groupware-Systemen. In ASCEND können sie zusätzlich auch von Worflow- und Designflow-Aktivitäten verwendet werden. Durch den Informationsaustausch über die Informationsräume werden kooperierende Aktivitäten realisierbar. Bei Bedarf dürfen die Aktivitäten jederzeit auf Informationen in diesen Informationsräumen zugreifen. Dabei müssen sie sich aber in der Regel an Protokolle halten, die von Erol Bozak in seiner Diplomarbeit entwickelt werden.

Um den Informationsraum herum sind die drei Dienste dargestellt, die im ASCEND-Modell integriert werden sollen. Oben die Groupware-Aktivitäten, über die die in Abschnitt 2.1 beschriebenen Groupware-Systeme in ASCEND eingebunden werden, links unten die Workflow-Aktivitäten, über die die in Abschnitt 2.2 beschriebenen Workflow-Management-Systeme eingebunden werden und schließlich rechts unten die Designflow-Aktivitäten, mit denen das Designflow-Management aus Abschnitt 2.3 realisiert wird.

Die ursprüngliche Ausrichtung von ASCEND ist die Unterstützung von Entwurfsanwendungen, es lässt sich aber auch in anderen Bereichen anwenden. Mariucci hat in seiner Diplomarbeit [Mar98] auch Abläufe in einem Krankenhaus modelliert und Frank beschreibt in [Fra00] die Abläufe bei der Suche nach dem Standort für eine neue Filiale.

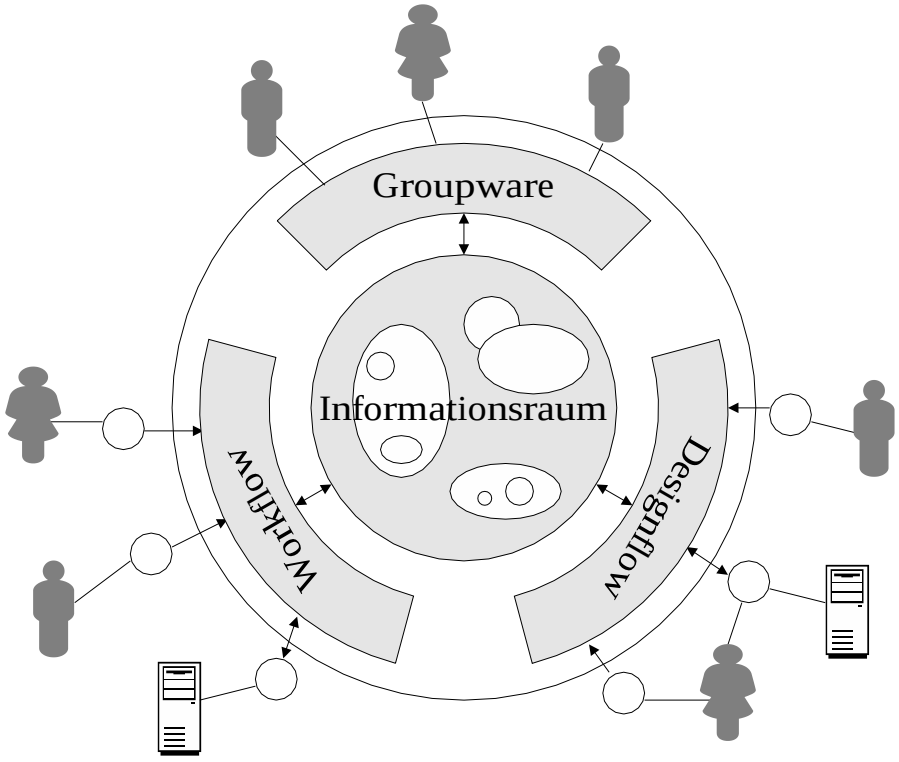

Abbildung 3.1: Aufbau von ASCEND nach [Fra00]

3.1 Das Aktivitätenmodell

Mit dem ASCEND Designflow Model (ADM) kann eine große Aktivität (z.B. die gesamte Entwicklung eines Produktes) in mehrere Sub-Aktivitäten gegliedert werden. Die Aktivitäten werden in ASCEND unterteilt in die folgenden Aktivitätenarten (siehe auch Abbildung 3.2):

Activity: Die *Activity* ist eine Oberklasse aller Aktivitäten und legt damit das grundsätzliche Verhalten von Aktivitäten fest. Ihr können ein Akteur und zusätzliche Ressourcen zugeordnet werden.

PrimitiveActivity: Diese Aktivitäten repräsentieren einzelne, atomare Arbeitsschritte. Diese Schritte werden in der Regel mit Hilfe von externen Programmen erledigt. Die *PrimitiveActivity* fungiert in diesem Fall als Wrapper, der bei einem Start die Daten in temporäre Dateien speichert und dann das zugehörige Programm aufruft. Wenn das Programm beendet wurde werden gegebenenfalls seine Ergebnisse wieder im Informationsraum abgelegt. Es können aber auch Aktivitäten als Sub-Klasse der *PrimitivtActivity* implementiert werden, die selbst direkt auf den Informationsraum zugreifen.

BasicActivity: Diese Aktivitäten-Klasse ist eine gemeinsame Oberklasse für alle Aktivitäten mit Sub-Aktivitäten. Sie unterscheiden sich daher von der *Activity* durch Methoden, die den Zugriff auf eine Menge von Aktivitäten erlauben. Diese Semantik wurde aus der Implementierung abgeleitet und stimmt auch mit der Beschreibung in [Rös99] überein. In allen bisherigen Arbeiten hat die *BasicActivity* die Position der *Activity* und rechtfertigt dort auch ihren Namen. Ein konkreter Grund für diesen Namens-Konflikt ist jedoch nicht ersichtlich.

ToplevelActivity: In einem laufenden System gibt es genau eine Instanz dieser Aktivität und alle anderen Aktivitäten sind Kinder, Enkel, ... dieser Aktivität. Im Prinzip könnte als oberste Aktivität einfach eine Instanz einer *BasicActivity* verwendet werden. Hier wurde dafür aber eine eigene Klasse implementiert. Diese Klasse hat neben den von der *BasicActivity* geerbten Methoden zum Zugriff auf Sub-Aktivitäten noch eine eigene Menge von Methoden, die in der Implementierung auch auf eine zweite Liste zugreifen. Dies ist eigentlich nicht nötig und kann sogar zu Inkonsistenzen führen.

GroupwareActivity: Dies sind Aktivitäten, bei denen mehrere Benutzer gemeinsam an der Erledigung einer Aufgabe arbeiten. Über diese Aktivitäten soll die von Muchitsch entwickelte CORBA Groupware Facility in das System CASSY integriert werden und damit die gemeinsame Erstellung eines Dokuments, Tele-Konferenzen und ähnliche Anwendungen integrieren. Beim Entwurf wurde leider ein Fehler gemacht, da Groupware-Aktivitäten eigentlich keine Sub-Aktivitäten haben. Sie hätten damit als direkte Sub-Klasse der *Activity* und nicht der *BasicActivity* implementiert werden müssen. Allerdings ist die Implementierung nur rudimentär.

WorkflowActivity: Über diese Aktivitäten werden die in Abschnitt 2.2 beschriebenen Workflows in CASSY verfügbar gemacht. Diese Aktivitäten wurden von Kuhn [Kuh00] mit einem Wrapper-Ansatz implementiert. Dieser Wrapper leitet die Anfragen über eine von Heiss [Hei98] implementierte CORBA-Schnittstelle (jFlow) an das Workflowsystem Flowmark von IBM weiter.

Als Sub-Klasse der *BasicActivity* könnten diesen Aktivitäten auch neue Sub-Aktivitäten hinzugefügt werden. Dies wird aber von WfMS in der Regel nicht unterstützt und sollte deshalb nicht gemacht werden.

DesignflowActivity: Mit dieser Aktivitätenart können die komplexeren Entwurfsabläufe modelliert werden. Im Gegensatz zu Workflows, wo wirklich Flüsse modelliert werden, werden hier aber nur einzelne Restriktionen zwischen den Sub-Aktivitäten definiert. Diese Restriktionen sind die Constraints, die in Abschnitt 6.3 behandelt werden.

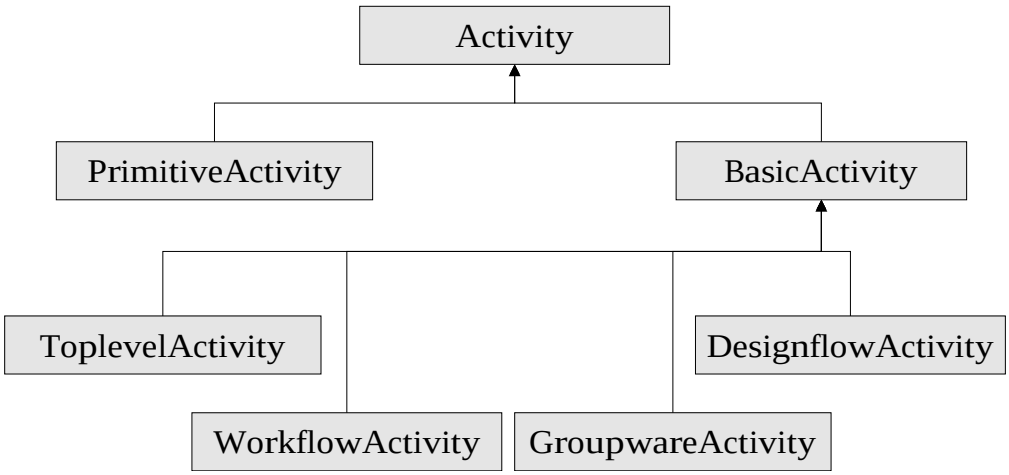

Abbildung 3.2: Vererbung der Aktivitäten-Klassen

3.2 Akteure und Ressourcen

Neben den Aktivitäten und den Informationsräumen gibt es in ASCEND noch eine weiter Gruppe von Objekten: die Akteure. Zu den Akteuren gehören hier mehr Objekte als üblicherweise. Sie werden untergliedert in:

User: Diese Objekte repräsentieren Menschen aus der realen Welt. User-Objekte haben eine Worklist auf der die vom Benutzer bearbeitbaren Aktivitäten abgelegt sind. Allerdings wurde diese Klasse aufbauend auf der User-Klasse der CSCW-Facility nur unzureichend implementiert, so dass sie kaum verwendet werden kann. Wo ein User benötigt wird kann aushilfsweise ein Aktor verwendet werden, der allerdings keine Worklist hat.

Group: Eine Gruppe ist eine Menge von Akteuren und kann selbst wieder wie ein Akteur verwendet werden. Eine solche Gruppe kann als Aktor einer Groupware-Aktivität zugeordnet werden.

BasicResourceAdapter: Gedacht ist hierbei z.B. an Maschinen oder Programme, die selbstständig eine Aktivität ausführen können. So kann kann z.B. ein Drucker als Aktor für eine Druck-Aktivität gesetzt werden, der sich dann selbstständig um den Ausdruck kümmert. Auch diese Objekte haben eine Worklist, deren Syntax sich interessanter Weise von der User-Klasse unterscheidet.

In der Arbeit von Rösner [Rös99] werden als *BasicResourceAdapter* (nur) Dokumente, Nachrichtenkanäle und Wrapper erwähnt. Weshalb diese Arten von Ressourcen als Spezialfall der Akteure modelliert wurde ist unklar.

3.3 Negotiation, Delegation und Usage

Neben dem unkoordinierten Austausch von Informationen über die gemeinsamen Daten im Informationsraum werden in ASCEND noch drei Kooperationsbeziehungen betrachtet:

Usage: Diese Beziehung besteht zwischen einer Aktivität und einer Ressource. Sie zeigt aber nicht nur an, dass die Aktivität die Ressource verwendet, sondern über diese Beziehung kann auch verhandelt werden, wie bei gleichzeitigem Zugriff auf die Ressource verfahren werden soll. Insbesondere ist diese Beziehung dazu gedacht, die vorzeitige Weitergabe von noch nicht fertiggestellten Dokumenten zu ermöglichen.

Negotiation: Diese Beziehung repräsentiert eine Verhandlung zwischen zwei Aktivitäten. In einer solchen Verhandlung können die Aktivitäten z.B. über die Veränderung von gemeinsamen Ressourcen verhandeln. Eine Ressource kann hier z.B. auch ein Dokument mit einer Aufgabenbeschreibung sein. Per Negotiation können sich die beteiligten Aktivitäten dann auch über die Aufgabenteilung verständigen.

Delegation: Die Delegation geht von einer Aktivität zu einer anderen. Die beiden Aktivitäten sind in der Regel Sub-Aktivitäten der gleichen Aktivität. Die Eltern-Aktivität kann noch während die Sub-Aktivität arbeitet die Aufgabe an globale Veränderungen anpassen. Bevor die Sub-Aktivität beendet wird, muss das Ergebnis von der delegierenden Aktivität angenommen werden. Ist sie damit nicht einverstanden kann sie die Sub-Aktivität zum weiterarbeiten veranlassen.

Ansätze für Protokolle, mit denen diese Beziehungen realisiert werden können, wurden von Bozak schon in seiner Studienarbeit [Boz99] betrachtet. In seiner zur Zeit laufenden Diplomarbeit sucht er eine Möglichkeit, wie diese Protokolle und ihre Auswirkungen modelliert werden können.

3.4 Ein Beispiel

Abbildung 3.3 ist angelehnt an ein Beispiel aus [Fra00] und soll zeigen, wie Aktivitäten im ASCEND-Modell in Sub-Aktivitäten aufgeteilt werden können. Die gesamte Aktivität “Standortplanung” wird von einer Groupware-Aktivität “Geschäftsführung” kontrolliert, die die anderen Aktivitäten delegiert. Im Rahmen dieser Aktivität trifft sich das Management des Unternehmens, kontrolliert den Fortschritt der Standortplanung und greift gegebenenfalls durch Modifikation der Dokumente im Informationsraum oder durch Erzeugen weiterer Aktivitäten ein.

In der Workflow-Aktivität “Marktanalyse” werden allgemeine Randbedingungen für neue Filialen bestimmt. Die Designflow-Aktivität “Standortsuche”

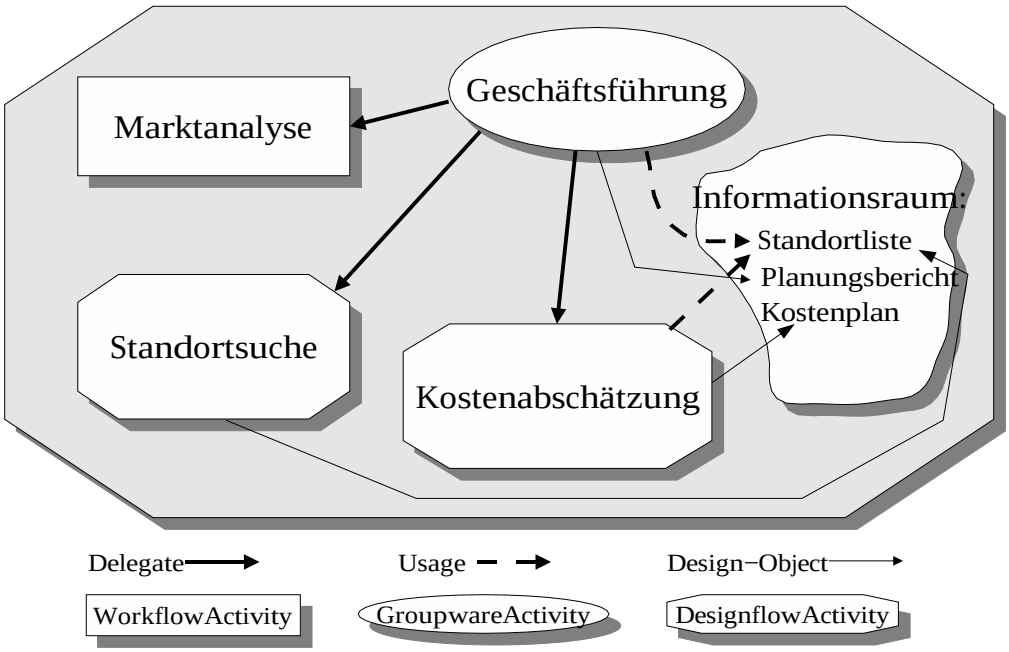

Abbildung 3.3: Beispiel eines Designflows zur Standortplanung

sucht nach möglichen Standorten, die dann von der Designflow-Aktivität "Kostenabschätzung" bewertet werden. Alle diese Aktivitäten greifen auf Dokumente im gemeinsamen Informationsraum zu.

3.5 Betrachtete Use Cases

Die in diesem Abschnitt aufgeführten Anwendungsfälle sind im wesentlich auf Designflow-Aktivitäten bezogen, können soweit sinnvoll aber auch für andere Aktivitäten angewendet werden.

- Erzeugen einer leeren Designflow-Aktivität. Dies geschieht wie das Erzeugen jeder anderen Aktivität auch: Auswahl der Factory, die die gewünschte Aktivität erzeugt, durch den Benutzer und einhängen der erzeugten Aktivität unter der gewünschten Instanz einer Sub-Klasse der *BasicActivity*, gegebenenfalls unter der *TopLevelActivity*. Dann können interaktiv Ressourcen, Sub-Aktivitäten und Constraints hinzugefügt werden.

- Erzeugen einer Designflow-Aktivität entsprechend einer Vorlage. Diese Vorlagen sollen in XML-Dateien gespeichert werden. In der gerade laufenden Studienarbeit von Achim Eichhorn wird die Form dieser Datei festgelegt und ein Parser entwickelt, der entsprechend einer solchen Datei dann die Aktivitäten initialisiert.

- Einfügen weiterer Aktivitäten. Das läuft prinzipiell wie das Erzeugen einer neuen Aktivität: Zunächst wird über die Factory die gewünschte Aktivität erzeugt und diese anschließend als Kind zur Super-Aktivität hinzugefügt.

  Damit ist die Aktivität aber nur erzeugt. Sie ist noch nicht initialisiert und es sind damit auch noch keine Ressourcen zugeordnet.

- Entfernen von Aktivitäten. Noch laufende Aktivitäten müssen natürlich vorher abgebrochen werden. Wenn die Aktivität hätte starten oder sogar terminieren müssen, soll der Benutzer vorher nochmals gefragt werden, ober er diese Bedingung verletzen will.

- Ändern von Ressourcen. Hierbei müssen Constraints gegen die Ressourcen beachtet werden. Wird die alte Ressource entfernt und einfach die neue hinzugefügt, dann sind die gegen die alte Ressource definierten Constraints nicht mehr vorhanden. Die Constraints müssen neu eingefügt werden wenn sie immer noch relevant sind.

- Ändern von Akteuren. Akteure gibt es immer nur einen pro Aktivität. Zum einen müssen hier die Constraints wie bei den Ressourcen behandelt werden. Zusätzlich müssen aber auch die Worklists des alten und neuen Akteurs angepasst werden.

- Initialisieren von Aktivitäten. Hier werden der Aktivität Ressourcen zugeordnet und gegebenfalls noch weite Sub-Aktivitäten hinzugefügt. Auf die Initialisierung und den im folgenden Punkt erwähnten Start wird in Abschnitt 6.2.3 noch näher eingegangen.

- Starten einer Aktivität. Hiermit wird die mit der Aktivität verbundene Anwendung gestartet.

- Beenden von Aktivitäten. Zuvor muss rekursiv geprüft werden, ob noch eine Sub-Aktivität läuft. Dann sollte man prüfen, ob ein Constraint nicht erfüllt wurde. Unter Umständen hat der Bearbeiter nur etwas vergessen und möchte diesen Constraint doch noch erfüllen.

- Einfügen und Entfernen von Constraints in ein Designflow. Hier kann der Benutzer die Constraints per Name auswählen. Zudem müssen die richtigen Objekte (Aktivität bzw. Ressource) als Parameter übergeben werden.

- Abfragen des Zustandes. Im wesentlichen interessiert den Benutzer der Stand von Aktivitäten und ob ein Constraint schon erfüllt oder bereits unerfüllbar ist.

- Der Administrator einer CASSY-Instanz kann eigene Constraints definieren, die für seine Anwendungen bzw. sein Unternehmen sinnvoll sind.

3.6 CORBA

Die Implementierung CASSY des ASCEND-Modells basiert auf der Middleware CORBA. Vor der Beschreibung von CASSY selbst folgt hier deshalb zunächst eine kurze Einführung in CORBA. Ein etwas ausführlicherer Überblick wird in [Say97] gegeben, ausführliche Bücher zum Thema sind z.B. [OH98] und [Hen99].

Die Common Object Request Broker Architecture (CORBA) ist ein von der Object Management Group (OMG, www.omg.org) definierter Standard zur Kommunikation zwischen Objekten in verteilten Systemen. Die OMG ist einen nicht-kommerzielle Organisation in der über 700 Firmen zusammengeschlossen sind. Ziel ist ein transparenter Zugriff auf verteilte Objekte, die auf unterschiedlichen Systemen laufen und auch in verschiedenen Programmiersprachen implementiert sein können.

Von Objekten werden in CORBA nur die Schnittstellen betrachtet. Diese Schnittstelle muss in der CORBA-IDL (Interface Definition Language) definiert werden, die stark an C++ angelehnt ist. Sie erlaubt die Definition von Datentypen, die als Parameter beim Aufruf von Methoden verwendet werden, Ausnahmen (exceptions) für Fehlermeldungen und Schnittstellen von Objekten.

Ein Beispiel für eine IDL-Datei ist in Abbildung 3.4 zu sehen. Diese Schnittstelle ist der Anfang für einen Rechen-Service. Zu einem "Modul" können beliebige Konstrukte zusammengefasst werden. Mit "struct" wird ein neuer, zusammengesetzter Datentyp definiert. Ein "Interface" beschreibt eine Schnittstelle eines Objektes. Eine Schnittstelle kann per Mehrfachvererbung von mehreren anderen Schnittstellen erben. Innerhalb dieser Schnittstelle wird hier zunächst eine Ausnahme (exception) definiert, die in Fehlersituationen von Methoden ausgelöst werden können. Anschliessend wird hier noch eine Divisions-Methode definiert, die die vorher definierte Ausnahme auslösen kann wenn der Nenner null ist. Desweiteren können in einer Schnittstelle auch Attribute des Objektes definiert werden. Es werden damit aber nicht wirklich Attribute sondern Methoden zum Lesen und Setzen der Attribute definiert.

In dieser Sprache wird aber nur die Schnittstelle beschrieben, es kann mit ihr kein Verhalten implementiert werden. Dazu muss/kann eine andere Programmiersprache verwendet werden. Hier hat die OMG für die gängigen Sprachen Abbildungen (Language-Mappings) definiert. Ein IDL-Compiler übersetzt die IDL-Datei entsprechend dem Language-Mapping in mehrere Dateien der Programmiersprache, so dass der Compiler der Programmiersprache die in der IDL-Datei definierten Konstrukte verwenden kann. Auf diese Weise erreicht man ein strenges und einheitliches Typ-Konzept und kann trotzdem Teile eines Systems in unterschiedlichen Sprachen implementieren.

Das Herzstück der Middleware CORBA ist der Object Request Broker (ORB). Dieser ORB wird oft als Bus dargestellt, mit dem die Objekte verbunden werden. Er verbirgt die Verteilung des Systems vor den einzelnen Objekten. Aus der Sicht eines Aufrufers spielt es keine Rolle, ob ein Objekt an den gleichen ORB angebunden ist, oder ob der Aufruf per Internet Inter-ORB Protocol (IIOP)

```
// Ein Modul/Paket
module MyModule {

  // Ein eigener Datentyp
  struct pair {
    float a;
    float b;
  }

  // Die Schnittstelle für einen Rechner
  interface Calculator {

    // Eine Ausnahme mit zusätzlichen Daten
    exception DivByZero { String information; };

    // eine Methode mit zwei Parametern, die auch
    // eine Ausnahme auslösen kann
    float divide(float divident, float divisor)
      raises (DivByZero);

    readonly attribute float pi;

    // und noch weitere Methoden...
  };
};
```

Abbildung 3.4: Ein Beispiel für eine IDL-Datei

zunächst zu einem anderen ORB weitergeleitet werden muss. Der Aufrufer hat in beiden fällen nur eine Interoperable Object Reference (IOR), die im Vergleich zu Zeigern in C keine Arithmetik erlaubt und auch eine Typ-Sicherheit gewährleistet.

Sowohl das Language-Mapping als auch der ORB spielen beim Aufruf von Methoden zusammen (siehe Abbildung 3.5). Von der Anwendung wird zunächst die Methode eines Stellvertreter-Objektes, dem Stub, aufgerufen. Die Implementierung dieses Objektes wurde vom IDL-Compiler generiert. Dieser Stellvertreter packt dann den Bezeichner des Server-Objektes, den Namen der Methode und die Parameter in eine Nachricht. Diese Nachricht wird dem lokalen ORB übergeben, der sie zu dem ORB schickt, an den das Server-Objekt gebunden ist. Dort wird die Nachricht dann von einem Skeleton, den auch der IDL-Compiler generiert hat, wieder ausgepackt. Schließlich wird die implementierte Methode des Server-Objektes aufgerufen. Hat die Methode ihre Arbeit beendet wird das Ergebnis auf dem umgekehrten Wege zurückgeschickt.

Davon bekommt der Programmierer aber, wenn alles gut geht, überhaupt nichts mit. Die vom IDL-Compiler generierten Stubs und Skeletons verbergen

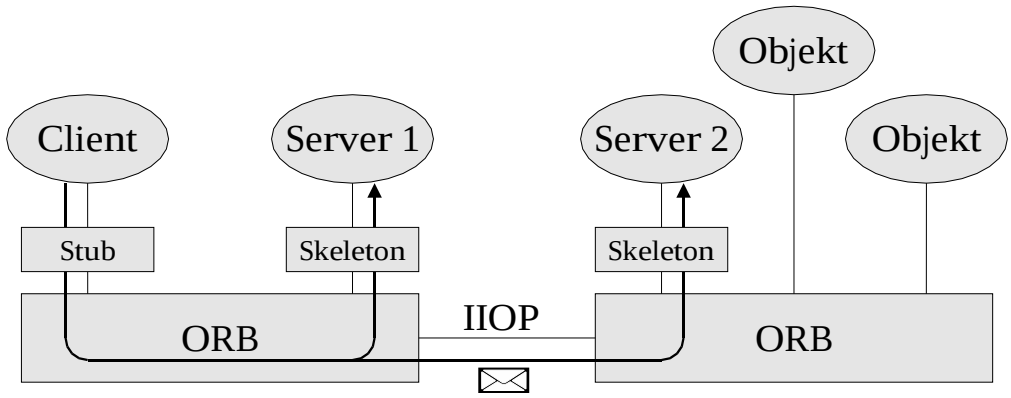

Abbildung 3.5: Kommunikation eines Clients mit Server-Objekten in CORBA

die Kommunikation komplett, so dass für den Programmierer der lokale Stellvertreter wie das eigentlich aufgerufene Objekt aussieht. Es gibt aber doch einige Unterschiede die aus der Tatsache resultieren, dass ein verteiltes System programmiert wird:

- Das Versenden der Nachrichten benötigt Zeit, die je nach Netzwerk auch größeren Schwankungen unterliegen kann.
- Die Verbindung zwischen den beiden Objekten kann zu einem beliebigen Zeitpunkt (vorübergehend) unterbrochen werden.
- Das Server-Objekt kann abstürzen während der Aufrufer noch auf ein Ergebnis wartet.

Neben diesem grundlegenden Mechanismus definiert die OMG auch Standard-Schnittstellen. Diese werden unterteilt in drei Gruppen:

Object Services sind Dienste, die allgemein bei der Entwicklung von verteilten Anwendungen nützlich sind. Beispiele sind Naming Service, Event Service und Transaction Service. In CASSY wird der Naming Service verwendet. Er stellt ein Verzeichnis zur Verfügung, in dem Objekte unter einem Namen abgelegt und von anderen abgefragt werden können.

Common Facilities sind Dienste die von vielen Anwendungen benötigt werden aber nicht so grundlegend sind wir die Common Object Services. Dienste in diesem Bereich behandeln z.B. User Interfaces und System Management.

Domain Interfaces sind Schnittstellen für einzelne Branchen. Hier gibt es z.B. CORBAfinance (Banken), CORBAmed (Medizin) und CORBAtel (Telekommunikation).

In CASSY wird die CORBA-Implementierung ORBacus der Firma Object Oriented Concepts (OOC) verwendet. Da zum einen eine ältere Version eingesetzt wird, die noch auf der CORBA-Version 2.2 basiert, und weil die Firma OOC mittlerweile von IONA aufgekauft wurde, muss CASSY vermutlich irgendwann an einen anderen ORB angepasst werden. Ein Vorteil bei neueren CORBA-Versionen sind die Portable Object Adapter (POA), mit denen die Portabilität eines Objektes zwischen unterschiedlichen CORBA-Implementierungen verbessert wird. Zum anderen ist auch die Persistenz der Zeiger (IORs) in CORBA einheitlich geregelt ist. Diese wird in der Implementierung der CSCW-Facility sehr aufwändig und in der Implementierung CASSY kaum behandelt.

3.7 Die Implementierung CASSY

CASSY (Cooperative Activity Support SYstem) ist die Implementierung des ASCEND-Modells. Die Schnittstellen der einzelnen Komponenten wurden in CORBA-IDL spezifiziert. Die Schnittstelle für die Aktivitäten wurde im wesentlichen von Mariucci entworfen [Mar98]. Die Schnittstellen für die Groupware-Facility wurde von Muchitsch [Muc98] entworfen und auch prototypisch implementiert. Anschließend hat Rösner versucht die beiden Arbeiten zu integrieren [Rös99]. Bei Rösner sind im Anhang auch die IDL-Dateien mit den Schnittstellen aller Objekte von CASSY zu finden.

Abbildung 3.6 zeigt die Architektur des Systems CASSY und den Entwicklungsstand. Dabei steht “Implementierte Komponente” nicht für wirklich verwendbare Komponenten, sondern dafür, dass es schon einen Implementierungsversuch gegeben hat. Mehr zu diesem Punkt folgt in der Bewertung an Ende dieser Arbeit.

Bisher wurde CASSY in C++ implementiert. Ein Vorteil der Implementierung von CASSY in C++ ist die gute Performanz und die direkte Abbildung der IDL-Dateien auf C++-Dateien. Das Constraints-basierte Aktivitäten-Management wurde nun aber in Java implementiert. Java hat den Vorteil einer großen Klassenbibliothek und einer automatischen Speicherbereinigung (garbage collection). Der Nachteil der schlechteren Performanz wird dank Just-In-Time-Compilern und Hot-Spot immer geringer. Ein wirklicher Nachteil bei der Verwendung mit CORBA ist, dass die IDL auf C++ basiert und einige Konstrukte daher etwas umständlich auf Java-Konstrukte abgebildet werden müssen.

Die einzelnen Dienste wurden in CASSY jeweils in einem eigenen Server implementiert. Dadurch können die einzelnen Server auch auf unterschiedlichen Rechnern gestartet werden und vom Absturz eines Servers ist nicht das ganze System betroffen. Nach dem Start melden sich die Server beim NameService, wo sie von anderen Komponenten ermittelt werden können. Jeder Server verwaltet die Objekte einer Klasse, legt neue Objekte dieser Klasse an und macht auch die alten zugreifbar.

Da es in CORBA keinen Mechanismus zum Erzeugen neuer Objekte gibt,

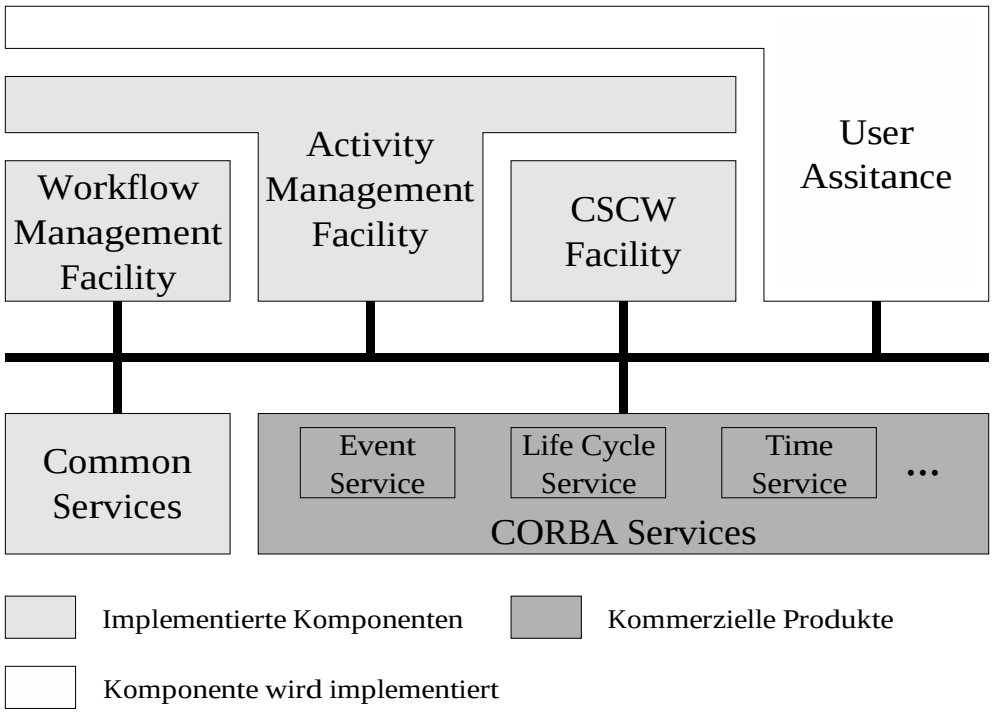

Abbildung 3.6: Architektur und Stand von CASSY

werden sogenannte Fabriken (Factories, siehe [GHJV96]) verwendet. Diese Fabriken bieten eine Methode, die ein neues Objekt lokal erzeugt, beim ORB registriert und anschließend eine Referenz auf das neue Objekt zurückgibt. Hiermit wird auch die Verteilung geregelt, da ein Objekt immer lokal bei der Factory angelegt wird. Auf diesem Rechner werden dann auch die Objekte persistent in Dateien gespeichert.

Kapitel 4

Koordination von Aktivitäten

In diesem Kapitel werden zunächst einige Ansätze aus der Literatur zur Koordination von Aktivitäten vorgestellt. Auf den Ansatz von Singh, der als Grundlage für die weitere Arbeit gewählt wurde, wird dann genauer eingegangen. Dabei wird zunächst die Semantik der verwendeten Abhängigkeiten formal definiert bevor auf zwei mögliche Realisierungen eingegangen wird.

4.1 Ansätze in der Literatur

Die im folgenden betrachteten Ansätze zur Spezifikation des Verhaltens einer Menge von Aktivitäten beruhen alle auf der Definition von Abhängigkeiten zwischen Ereignissen. Der Fokus liegt bei diesen Ansätzen auf der Spezifikation der Koordination dieser Aktivitäten und nicht auf dem Verhalten der einzelnen Aktivitäten. Deshalb beschränkt man sich auf sogenannte "signifikante Ereignisse". Das sind Ereignisse, die auch außerhalb einer Aktivität eine Bedeutung haben können, wie z.B. start, abort und commit. An diese signifikanten Ereignisse werden je nach Ansatz unterschiedliche Restriktionen gestellt.

4.1.1 Advanced Rule Driven Transaction Management

Klein beschreibt in [Kle91] ein Framework zur Formalisierung von Protokollen zwischen verteilten Anwendungen. Er definiert zwei Primitive:

- $e < f$: Wenn beide Ereignisse e und f eintreten, dann muss das Ereignis e vor dem Ereignis f eintreten.
- $e \rightarrow f$: Wenn das Ereignis e eintritt, dann muss auch das Ereignis f eintreten. Damit wird die Reihenfolge der beiden Ereignisse aber nicht eingeschränkt, so dass durchaus auch e vor f eintreten darf. Dann muss aber auf eine andere Weise sichergestellt werden, dass f irgendwann eintreten wird, da sonst diese Implikation nicht erfüllt ist.

Als Beispiel sollen hier die Abhängigkeiten zwischen Eltern- und Kind-Transaktion einer geschachtelten Transaktion [Mos82] dienen, bei denen die

Eltern-Transaktion (p) die Kontrolle über das commit der Kind-Transaktion (c) übernimmt:

$$finish_c < finish_p$$
$$commit_p \rightarrow (finish_c \rightarrow commit_c)$$
$$commit_c \rightarrow commit_p$$

Durch die statischen Regeln sind die Möglichkeiten dieses Modells sehr begrenzt und eine formale Semantik wird nicht angegeben. Günthör beschreibt in [Gün97] die Realisierung einer Abhängigkeitsverwaltung die diese Primitive verwendet.

4.1.2 ACTA

Chrysanthis hat ACTA [Chr91, CR92] als Framework zur Spezifikation von Transaktionsmodellen entwickelt. In ACTA lassen sich die folgenden Bedingungen an die (partielle) Historie stellen:

$e_1 \rightarrow e_2$
: e_1 kommt vor e_2 und beide Ereignisse sind in der Historie enthalten.

$(e \in H) \Rightarrow Bedingung$
: Wenn das Ereignis e in der Historie H ist, dann muss auch die Bedingung erfüllt sein. *Bedingung* ist damit eine Vorbedingung für das Ereignis e.

$Bedingung \Rightarrow (e \in H)$
: Wenn die Bedingung gilt dann muss das Ereignis in der Historie sein.

Damit lässt sich dann z.B. eine commit dependency definieren, bei der das Commit der Transaktion t_i vor dem Commit der Transaktion t_j eintreten muss, wenn beide Transaktionen mit Commit beendet werden (entspricht $Commit_{t_i} < Commit_{t_j}$ bei Klein):

$$(Commit_{t_j} \in H) \Rightarrow ((Commit_{t_i} \in H) \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j}))$$

In den Bedingungen können auch eigene Prädikate verwendet werden, mit denen z.B. die von einer Transaktion verwendeten Objekte oder möglicherweise in Konflikt stehende Transaktionen repräsentiert werden können.

In ACTA und darauf aufbauenden Arbeiten werden als zusätzliche signifikante Ereignisse auch Objekt-Ereignisse definiert, die bei einem lesenden oder schreibenden Zugriff auf ein Objekt eintreten. Damit kann dann mit diesem Ansatz auch die Synchronisation der Daten beschrieben werden.

ACTA ist auf die Modellierung und den Vergleich von Transaktionsmodellen ausgerichtet. Es ist damit mehr als formales Werkzeug denn als Basis für eine Implementierung gedacht. Geppert gibt in [GD93] eine Implementierung an, bei der die Bedingungen in Event-Condition-Action-Regeln (ECA-Regeln)

übersetzt werden. Diese Regeln werden dann in eine Datenbank geladen und bei Änderungen an Tabellen, den Ereignissen, berücksichtigt. Dieser Ansatz ist in CASSY kaum möglich, da hier die Abhängigkeiten zwischen den Aktivitäten nicht a priori feststehen sondern sogar noch während der Ausführung änderbar sein sollen. Ein ständiges Ändern der ECA-Regeln ist aber von Datenbanken nicht vorgesehen.

4.1.3 Singh

Eine wesentlicher Erweiterung zu den vorigen Ansätzen ist die Behandlung von Variablen in den Ereignissen. Dadurch kann zum einen zwischen Ereignis-Instanzen und Ereignis-Typen unterschieden werden, zum anderen können damit mehrfach eintretende Ereignisse und damit auch Schleifen in Aktivitäten behandelt werden. Eine genauere Beschreibung dieses Ansatzes folgt im folgenden Abschnitt.

Die EventEngine und damit der Rest dieses Kapitels basieren auf den beiden Artikeln [Sin97b, Sin97a] von Singh, in denen dieser Ansatz beschrieben wird.

4.2 Formale Spezifikation des Modells

Dieser Abschnitt baut auf dem Ansatz von Singh auf. Dieser Ansatz wurde gewählt, da durch die Verwendung von Variablen ein größerer Bereich behandelbar ist. Zudem schien der Ansatz, auch aufgrund der vielen Definitionen und Beweise in den beiden Artikeln, gut durchdacht zu sein.

Im folgenden werden zunächst die Syntax und die Semantik der Abhängigkeiten definiert, mit denen das Verhalten der EventEngine festgelegt wird.

4.2.1 Ereignisse

Die Menge Σ enthält die Namen der signifikanten Ereignisse. Ein Ereignis setzt sich aus einem solchen Namen und einer Menge von Parametern zusammen. Diese Parameter sind entweder aus der Menge $\mathcal{C}$ der Konstanten oder aus der Menge $\mathcal{V}$ der Variablen, die hier zur Kennzeichnung mit '$' anfangen.

Bevor ich auf die hier verwendete Definition der Ereignisse eingehe, zeige ich zunächst die Definition der Ereignisse nach Singh. Bei Singh hat ein Ereignis eine feste Anzahl an Parametern, die durch ihre Position unterschieden werden. Sei $\delta : \Sigma \rightarrow \mathbb{N}$ die Anzahl der Parameter eines Ereignisses. Dann ist die Menge der Ereignis-Instanzen Γ_S und die Menge der Ereignis-Typen Ξ_S nach Singh definiert als

$$\forall e \in \Sigma, m = \delta(e) :$$
$$p_1, \ldots, p_m \in \mathcal{C} \Rightarrow e[p_1, \ldots, p_m], !e[p_1, \ldots, p_m] \in \Gamma_S$$
$$p_1, \ldots, p_m \in \mathcal{C} \cup \mathcal{V} \Rightarrow e[p_1, \ldots, p_m], !e[p_1, \ldots, p_m] \in \Xi_S$$

In diesen Mengen ist zu jedem Ereignis e auch ein komplementäres Ereignis $!e$ enthalten. Dieses Ereignis tritt dann ein, wenn die Entscheidung gegen das Eintreten von e getroffen wird. $!!e$ soll äquivalent zu e sein.

Im Gegensatz zu Singh unterscheide ich Parameter nicht nach ihrer Position sondern durch einen Namen. Dadurch werden Ereignisse und vor allem die Spezifikationen der Constraints leichter lesbar.

Definition 4.2.1 (Ereignis-Instanzen und -Typen) *Sei ID die Menge der Namen der Parameter. Dann sind die Menge der Ereignis-Instanzen Γ und die Menge der Ereignis-Typen Ξ definiert als:*

$$\Gamma = (\Sigma \times (ID \nrightarrow \mathcal{C}))$$
$$\Xi = (\Sigma \times (ID \nrightarrow (\mathcal{C} \cup \mathcal{V})))$$

Syntaktisch schreibe ich für ein Ereignis den Namen des Ereignisses gefolgt von den Parametern in eckigen Klammern, also z.B. $start[id = 3]$ für das Ereignis $(start, (id \mapsto 3))$. 'start' ist hier der Name des Ereignisses, 'id' der Name des ersten und einzigen Parameters und '3' der Wert dieses Parameters. In den Beispielen werde ich öfters den Namen der Parameter weg lassen, wenn er keine Rolle spielt.

Um einfacher mit den Parametern von Ereignissen arbeiten zu können, seien für alle Ereignisse die Menge der Parameter $\wp$ und eine Notation zum Zugriff auf einzelne Parameter folgendermaßen definiert (dom gibt den Definitionsbereich einer Funktion an, $dom(params)$ ist also die Menge der Parameternamen des Ereignisses):

$$\begin{aligned}\forall e = &(name, params) \in \Xi : \\ &\wp(e) = dom(params) \\ &\forall id \in ID : e(id) = params(id)\end{aligned}$$

Bei dieser Definition der Ereignisse mit Parametern ist es nun auch möglich, bei der Angabe von Ereignis-Typen nur die gerade signifikanten Attribute anzugeben. Damit können Ereignisse nicht mehr einfach per "="-Operator verglichen werden, sondern es wird ein eigener Vergleichs-Operator benötigt.

Definition 4.2.2 (Vergleichs-Operator $\trianglelefteq$) *Durch den Operator $\trianglelefteq \subseteq (\Xi \times \Xi)$ wird festgelegt, ob das linke Ereignis eine (Teil-)Instantiierung des rechten Ereignisses ist. Er ist definiert als*

$$e \trianglelefteq f \Leftrightarrow \wp(f) \subseteq \wp(e) \wedge \forall p \in \wp(f) : e(f) = e(p)$$

Die folgenden Beispiele sollen die Bedeutung des Operators verdeutlichen:

$$\begin{aligned} start[id = 13, time = 983890248] &\trianglelefteq start[id = 13, time = 983890248] \\ start[id = 13, time = 983890248] &\trianglelefteq start[id = 13] \\ start[id = 13, time = 983890248] &\trianglelefteq start[] \\ start[id = 13] &\not\trianglelefteq terminate[id = 13] \\ start[id = 1] &\not\trianglelefteq start[id = 13] \end{aligned}$$

Aufbauend auf diesem Vergleich ist der folgende Enthalten-Operator definiert, der angibt, ob ein Ereignis $e \in \Xi$ zu irgend einem Ereignis in der Menge $P \subseteq \Xi$ passt:

$$e \lhd P \;\Leftrightarrow\; \exists p \in P : e \trianglelefteq p$$

Abschließend folgen nun noch einige Beispiele für diesen Operator:

$$\begin{aligned}
start[id = 13] &\lhd \{start[id = 13], \dots\} \\
start[id = 13] &\lhd \{start[]\} \\
start[id = 13] &\not\lhd \{terminate[id = 13]\} \\
start[id = 1] &\not\lhd \{start[id = 13]\} \\
start[id = 1] &\not\lhd \emptyset
\end{aligned}$$

Die Beispiele zeigen auch, dass immer mindestens so viele Parameter angeben werden sollten, dass damit die Ereignisse eindeutig unterschieden werden können. Zum Beispiel ist es in der Regel nicht sinnvoll, ein Ereignis $start[]$ in einer Abhängigkeit zu verwenden, da davon jedes folgende *start*-Ereignis betroffen ist, egal für welche Aktivität es eintritt.

4.2.2 Formalisierung der Abhängigkeitsregeln

In diesem Abschnitt werden die Abhängigkeiten zwischen Ereignissen formal spezifiziert. Die Sprache enthält als Atome zum einen die Ereignistypen und zum anderen zwei Konstanten: 0 für falsch und $\top$ für wahr. Desweiteren können Ausdrücke in der Sprache durch die Operatoren $\wedge$ (logisches und), $\vee$ (logisches oder) und $-$ (Sequenz) verknüpft werden.

Definition 4.2.3 (Syntax der Abhängigkeiten) *Formal ist eine Abhängigkeit (dependency) ein Element aus der Menge $\mathcal{E}$ der Abhängigkeiten:*

$$\begin{gathered}
\Xi \subseteq \mathcal{E} \\
0, \top \in \mathcal{E} \\
E_1, E_2 \in \mathcal{E} \Rightarrow E_1 \vee E_2, E_1 \wedge E_2, E_1 - E_2 \in \mathcal{E}
\end{gathered}$$

Die Ausdrücke können durch Klammern gegliedert werden. Ansonsten gilt, dass ! stärker bindet als $-$, $-$ stärker als $\wedge$ und $\wedge$ stärker als $\vee$. Die folgenden Beispiele für Abhängigkeiten sind alle syntaktisch korrekt, teilweise aber nicht erfüllbar:

$$\begin{aligned}
&start[id = \$i] \\
&start[id = \$i] - terminate[id = \$i] \\
&start[] - 0 \\
&start[id = 2] \wedge !start[id = 2] \\
&start[id = 2] - (terminate[id = 2] \;\vee\; cancel[id = 2])
\end{aligned}$$

Die Menge der Ereignisse und ihrer Komplemente, die in einer Abhängigkeit E vorkommen, wird dargestellt als Γ_E . Γ_E ist rekursiv definiert als: $\Gamma_0 = \Gamma_\top = \emptyset$, $\Gamma_{E \wedge F} = \Gamma_{E \vee F} = \Gamma_{E-F} = \Gamma_E \cup \Gamma_F$, $\Gamma_e = \Gamma_{!e} = \{e, !e\}$. Für die Abhängigkeit $E = a[] \ \vee \ b[id = 3]$ ist beispielsweise $\Gamma_E = \{a[], !a[], b[id = 3], !b[id = 3]\}$.

Die Semantik der Abhängigkeiten wird mit Hilfe von Spuren (traces) definiert. Eine Spur ist eine Folge von Ereignissen und wird dargestellt als $\tau = \langle ef \dots \rangle$. Auf die einzelnen Ereignisse in einer Spur kann per τ_i zugegriffen werden. Es gilt z.B. $\langle efg \rangle_1 = e$. $\tau\upsilon$ ist die Spur, die durch Anhängen der Spur τ an die Spur υ entsteht.

Definition 4.2.4 (Konsistente Spuren) *Das Universum der Spuren $U_\Gamma \subseteq \Gamma^*$ enthält nur konsistente (legale) Spuren. In einer konsistenten Spur kommt ein Ereignis und sein Komplement höchstens einmal vor:*

$$e \in \tau \ \Leftrightarrow !e \notin \tau \tag{4.1}$$

$$\forall i, j \in \mathbb{N} : i \neq j \ \Rightarrow \ \tau_i \neq \tau_j \tag{4.2}$$

Definition 4.2.5 (Semantik der Abhängigkeiten) *Die Bedeutung bzw. Denotation eines Ausdrucks wird definiert durch die Menge der konsistenten Spuren, die den gegebenen Ausdruck erfüllen. Die Denotation der einzelnen Ausdrücke wird durch die folgende Gleichungen festgelegt:*

$$[\![e[p_1 = c_1, \dots, p_m = c_m]]\!] = \{\tau \in U_\Gamma | e[p_1 = c_1, \dots, p_m = c_m] \lhd \tau\} \tag{4.3}$$

$$[\![E(\$v)]\!] = \bigcap_{c \in C} [\![E(\$v ::= c)]\!] \tag{4.4}$$

$$[\![0]\!] = \emptyset \tag{4.5}$$

$$[\![\top]\!] = U_\Gamma \tag{4.6}$$

$$[\![E_1 \vee E_2]\!] = [\![E_1]\!] \cup [\![E_2]\!] \tag{4.7}$$

$$[\![E_1 \wedge E_2]\!] = [\![E_1]\!] \cap [\![E_2]\!] \tag{4.8}$$

$$[\![E_1 - E_2]\!] = \{\tau\upsilon \in U_\Gamma | \tau \in [\![E_1]\!] \wedge \upsilon \in [\![E_2]\!]\} \tag{4.9}$$

$E(\$v ::= c)$ steht für den Ausdruck, der durch Ersetzen der Variablen $\$v$ an allen Stellen durch c entsteht. Dabei spielen die Namen der Parameter keine Rolle. Beispielsweise gilt

$$s[id = \$i] - t[a = \$i](\$i = 2) \ \equiv \ s[id = 2] - t[a = 2]$$

Bis auf die Gleichung (4.4) ist die Definition einfach nachvollziehbar. Warum wird in dieser Gleichung aber die Schnittmenge und nicht die Vereinigung verwendet? Zunächst erlaubt die Vereinigung jede Spur, die die Abhängigkeit auch für nur eine Konstante irgendwann erfüllt. Das würde für den Ausdruck $b[\$t] - e[\$t]$ ("zu jedem begin kommt irgendwann später das passende end") auch die ungewollte Spur $\langle b[1]e[1]e[5]e[3]b[5] \rangle$ erlauben. Mit der Bindung $\$t = 1$ wird diese Spur akzeptiert obwohl die Ereignisse mit der Bindung $\$t = 5$

so nicht erlaubt sind. Im Gegensatz dazu wird bei der Verwendung der Schnittmenge eine Spur zwar nie vollständig akzeptiert, da in der Regel nie alle Werte für alle Variablen gesetzt werden, es werden aber zumindest alle ungewollten Spuren verhindert.

Die Semantik des Komplements soll hier noch durch ein Beispiel verdeutlicht werden. Als Beispiel soll das Ereignis b nicht zwischen dem Ereignis a und dem Ereignis c stattfinden. Ein erster Ansatz für eine Abhängigkeitsregel ist

$$a - !b - c$$

Hier darf b aber gar nicht eintreten, auch nicht vor a oder nach c. Nur die Entscheidung gegen b muss zeitlich zwischen dem Eintreten der Ereignisse a und b fallen. Die eigentlich gewünschte Regel erhält man durch Aufzählen aller erlaubten Sequenzen:

$$\begin{aligned} & b - a - c \\ \vee\ & a - c - b \\ \vee\ & !b \end{aligned}$$

Als weiteres Beispiel sollen die Primitive von Klein (siehe Abschnitt 4.1.1) mit diesen Abhängigkeiten definiert werden:

$$e <_{Klein} f ::= !e \vee !f \ \vee\ e - f \tag{4.10}$$

$$e \rightarrow_{Klein} f ::= !e \ \vee\ f \tag{4.11}$$

Für eine einfachere Bearbeitung sollen die Abhängigkeiten in eine Normalform umgeformt werden. Dazu muss zunächst geklärt werden, welche Abhängigkeiten äquivalent sind.

Definition 4.2.6 (Äquivalenz von Abhängigkeiten) *Zwei Abhängigkeiten sind äquivalent, wenn sie die gleichen Spuren erlauben:*

$$\forall E_1, E_2 \in \mathcal{E} : \quad E_1 \equiv_{\mathcal{E}} E_2 \Leftrightarrow [\![E]\!] = [\![F]\!]$$

Da sich die Ausdrücke nur auf die Spuren beziehen, die die erwähnten Ereignisse enthalten, gelten die folgenden Eigenschaften:

$$\forall e \in \Xi : \top \ \not\equiv_{\mathcal{E}} \ (e \vee !e) \tag{4.12}$$

$$\forall e \in \Xi : 0 \ \not\equiv_{\mathcal{E}} \ (e \wedge !e) \tag{4.13}$$

Bei 0 und $\top$ besteht die Möglichkeit, dass die Entscheidung für oder gegen e nie getroffen wird.

Die Abhängigkeiten lassen sich in eine äquivalente Abhängigkeit in einer speziellen konjunktiven Normalform (KNF), der Two-Sequence-Form (TSF), umwandeln.

Definition 4.2.7 (Two-Sequence-Form (TSF)) *Die Two-Sequence-Form ist eine spezielle konjunktive Normalform. In dieser Normalform darf eine Disjunktion keine Konjunktion enthalten und ein Sequenz darf maximal zwei Ereignisse enthalten (und keine Konjunktionen und Disjunktionen).*

Um eine Abhängigkeit in konjunktiver Normalform in eine Abhängigkeit in Two-Sequence-Form umzuwandeln, wird eine Regel benötigt, mit der Sequenzen mit einer Länge größer als zwei in kleinere Sequenzen aufgeteilt werden können. Diese Regel lautet:

$$a - b - c \equiv_{\mathcal{E}} (a - b) \wedge (b - c)$$

Die Umwandlung in konjunktive Normalform (und damit auch in TSF) ist unter Umständen sehr aufwändig. Die Größe einer Abhängigkeit gemessen in Anzahl der Ereignisse und Operanden kann dabei exponentiell anwachsen. Bei ungünstigen Abhängigkeiten verbraucht daher der von mir in Java implementierte Normalisierer den ganzen für Java freigegebenen Speicher von etwa 64MB. Das kam aber erst bei Abhängigkeiten mit drei geschachtelten Operatoren vor, die ganz außen einen Sequenz-Operator hatten, der für die Umformung besonders ungünstig ist. Solche Abhängigkeiten sollte man also vermeiden. Da von dieser Einschränkung nur der Administrator beim Definieren der Constraints betroffen ist, und nicht der Anwender wenn er das System verwendet, ist das hier akzeptabel.

Die konjunktive Normalform ist bei Singh schon dadurch gegeben, dass er einen Workflow als Menge von per Konjunktion verknüpften Abhängigkeiten betrachtet, und eine Abhängigkeit als Menge von alternativen Ausführungsreihenfolgen (Disjunktion von Sequenzen).

4.3 Direktes Abarbeiten der Abhängigkeiten

Singh beschreibt in [Sin97b, Sin97a] zwei Varianten, wie die Folge der Ereignisse entsprechend der im System definierten Abhängigkeiten kontrolliert werden können. Beide behandeln die Abhängigkeiten symbolisch, d.h. nur durch betrachten der Abhängigkeiten und ohne z.B. die erlaubten Möglichkeiten aufzuzählen. Die erste Variante arbeitet zentral die Abhängigkeiten direkt ab, die zweite Variante ist verteilt und bestimmt für die einzelnen Ereignisse Wächter.

4.3.1 Das Verfahren

Bei der ersten Variante gibt es eine zentrale Komponente die alle Abhängigkeiten kennt und alle Ereignisse behandeln muss. Diese Komponente muss nach dem Eintreffen eines Ereignisses die im System definierten Abhängigkeiten an die neue Situation anpassen. Diesen Vorgang nennt Singh “residuation” und definiert dazu einen Residuation-Operator.

Definition 4.3.1 (Residuation-Operator /) *Die Semantik des Residuation-Operators* $/ : \mathcal{E} \times \Gamma \rightarrow \mathcal{E}$ *ist definiert als*

$$\upsilon \in [\![D/e]\!] \Leftrightarrow (\forall \tau \in [\![e]\!] : (\upsilon\tau \in U_\Gamma \Rightarrow \tau\upsilon \in [\![D]\!]))$$

Wie schon erwähnt muss dieser Operator aber nicht wie in der Definition durch aufzählen der gültigen Spuren bestimmt werden, sondern er kann symbolisch berechnet werden. Dieses schrittweise Einarbeiten von Ereignissen in die Abhängigkeiten wird durch die folgenden Gleichungen definiert. In diesen Gleichungen sind $e, f \in \Gamma$ Ereignisse, $E, E_1, E_2 \in \mathcal{E}$ Sequenzen oder $\top$ (für leere Sequenzen) und $D \in \mathcal{E}$ ist eine Sequenz mit mindestens einem Element.

$$0/e = 0 \tag{4.14}$$

$$\top/e = \top \tag{4.15}$$

$$(E_1 \wedge E_2)/e = ((E_1/e) \wedge (E_2/e)) \tag{4.16}$$

$$(E_1 \vee E_2)/e = ((E_1/e) \vee (E_2/e)) \tag{4.17}$$

$$(e - E)/e = E, \text{ if } e \not\triangleleft \Gamma_E \tag{4.18}$$

$$D/e = D, \text{ if } e \not\triangleleft \Gamma_D \tag{4.19}$$

$$(f - E)/e = 0, \text{ if } e \triangleleft \Gamma_E \tag{4.20}$$

$$(!e - E)/e = 0 \tag{4.21}$$

$$E(\vec{v})/e[\vec{c}] = (\forall \vec{d} : \vec{d} \neq \vec{c} \Rightarrow E(\vec{v})) \wedge E(\vec{v} ::= \vec{c})/e[\vec{c}] \tag{4.22}$$

In der letzten Zeile ist $\vec{v}$ ein Vektor mit den Variablen des Ereignisses e und $\vec{c}$ ist ein gleich großer Vektor mit Konstanten. Die letzte Gleichung wird auch noch ausführlicher in Abschnitt 5.4 behandelt. Singh zeigt in [Sin95, Sin97a] die Korrektheit und Vollständigkeit seiner Regeln. Meine Änderungen an der Definition der Ereignisse haben darauf keinen Einfluss und sind daher auch korrekt.

4.3.2 Beispiel

Als Beispiel für die schrittweise Abarbeitung der Abhängigkeiten sollen hier einige Regeln dienen, die aus dem Aktivitätenmanagement stammen könnten. Gegeben seien die Ereignisse i(nit), s(tart), c(ancel) und t(erminate). Um den Ablauf von Aktivitäten zu kontrollieren sei die folgende Abhängigkeit definiert:

$$\begin{aligned}&(i[\$A] - s[\$A] - c[\$A] \ \vee \ i[\$A] - s[\$A] - t[\$A])\\ \wedge &(!c[\$A] \vee !t[\$A])\end{aligned}$$

Der obere Teil modelliert die beiden möglichen Abläufe der Aktivität. Der untere Teil sorgt dafür, dass entweder terminate oder cancel nicht eintreten darf. Nur mit den oberen beiden Sequenzen dürften auch beide eintreten, da die Sequenzen durch ein inklusives Oder verknüpft sind.

		$i[2], i[1]$	$s[1]$	$t[1]$	$s[2]$	$c[2]$	$!t[2]$	$!c[1]$
1 2	$\vee$	$i[\$A] - s[\$A] - c[\$A]$ $i[\$A] - s[\$A] - t[\$A]$						
3 4	$\vee$	$!c[\$A]$ $!t[\$A]$						
5		$t[1] - s[2]$		$s[2]$	$\top$			
6 7	$\vee$	$s[2] - c[2]$ $s[2] - t[2]$			$c[2]$ $t[2]$	$\top$		
8 9	$\vee$	$!c[2]$ $!t[2]$				0	$\top$	
10 11	$\vee$	$s[1] - c[1]$ $s[1] - t[1]$	$c[1]$ $t[1]$	$\top$				
12 13	$\vee$	$!c[1]$ $!t[1]$		0				$\top$

Tabelle 4.1: Abarbeitung der Abhängigkeiten

Im Beispiel seien nun zwei Aktivitäten 1 und 2 vorhanden, d.h. obige Abhängigkeit wird einmal mit der Bindung $\$A = 1$ und einmal mit der Bindung $\$A = 2$ instantiiert. Zwischen den beiden Aktivitäten ist zudem noch die Abhängigkeit $t[1] - s[2]$ definiert. Diese Abhängigkeit beschreibt eine strenge Sequenz, bei der die Aktivität 1 terminieren, d.h. sich erfolgreich beenden muss, und danach die Aktivität 2 starten muss.

Tabelle 4.1 zeigt einen möglichen Ablauf bei diesen Abhängigkeiten. In der obersten Zeile stehen die Ereignisse von links nach rechts in der Reihenfolge, in der sie im Beispiel eintreten sollen. In den Spalten darunter sind jeweils nur die durch das Ereignis veränderten Zeilen dargestellt. Die ganz durchgezogenen horizontalen Linien trennen die Abhängigkeiten. Alle Zeilen, die nicht durch eine Linie getrennt sind, sind Alternativen in einem Oder. Die Zeilen (1) bis (5) enthalten die ursprünglich definierten Abhängigkeiten. Von diesen ändern sich die ersten vier, die den allgemeinen Ablauf von Aktivitäten modellieren, nie. Von ihnen werden nur weitere Instanzen (die Zeilen 6-13) angelegt.

Zunächst kann nur ein i-Ereignis eintreten, da alle anderen Ereignisse an einer späteren Stelle in einer Sequenz vorkommen und somit blockiert sind (nicht aufgeführte Ereignisse wie z.B. $x[]$ oder $a[]$ könnten natürlich eintreten). Im Beispiel treten als erstes die beiden Ereignisse $i[2]$ und $i[1]$ ein. Danach kann nur $s[1]$ eintreten, da nach der Abhängigkeit in Zeile 5 $s[2]$ erst nach $t[1]$ eintreten darf.

Nach den bisherigen Regeln dürfte dann durchaus $c[1]$ eintreten. Dann müsste aber sowohl $t[1]$ (wegen Zeile 5) als auch $!t[1]$ (wegen Zeile 13) eintreten, was nicht erfüllbar ist. Wie dieser Fall verhindert werden kann, wird in Abschnitt 5.3 beschrieben.

Es wird nun mit $t[1]$ fortgefahren. Anschließend darf dann endlich auch die Aktivität 2 mit $s[2]$ starten. Aktivität 2 beendet sich dann in diesem Beispiel durch einen Abbruch $c[2]$, was hier auch erlaubt ist.

4.4 Abarbeitung mit Wächter

Die zweite Variante zur Behandlung der Abhängigkeiten sind Wächter (guards). Ein Wächter ist einem Ereignis zugeordnet und verhindert das Eintreten des Ereignisses wenn dies von Abhängigkeiten verlangt wird. Die Wächter werden in einer temporalen Logik angegeben, die im Vergleich zu den Abhängigkeiten noch drei weitere Operatoren enthält:

$\Box E$ bedeutet, dass die Abhängigkeit E immer gelten wird.

$\Diamond E$ heißt, dass E irgendwann gelten wird. Die Entscheidung ist bereits gefallen, das Ereignis selbst ist aber noch nicht eingetreten.

$\neg E$ heißt, dass E noch nicht gilt. Es wurde noch nicht entschieden, ob die Abhängigkeit jemals gelten wird.

Definition 4.4.1 (Syntax der Wächter)

$$\begin{aligned} \Gamma &\subseteq \mathcal{G} \\ 0, \top &\in \mathcal{G} \\ E_1, E_2 \in \mathcal{G} &\Rightarrow E_1 \vee E_2, E_1 \wedge E_2, E_1 - E_2 \in \mathcal{G} \\ E \in \mathcal{G} &\Rightarrow \Box E, \Diamond E, \neg E \in \mathcal{G} \end{aligned}$$

Die Definition der Semantik unterscheidet sich etwas von der Definition bei den Abhängigkeiten. Eine Spur τ kann eine Abhängigkeit E auch dann erfüllen, wenn nicht alle Ereignisse der Abhängigkeit eingetreten sind. Dagegen wird ein Wächter von einer Spur erst dann erfüllt, wenn alle Ereignisse oder ihre Komplemente eingetreten sind. Solche Spuren werden vollständige Spuren genannt.

Definition 4.4.2 (Vollständige Spuren (maximal traces)) *Eine konsistente Spur $\tau \in U_\Gamma$ heißt vollständig bzgl. einer Menge von Ereignissen E, wenn jedes Ereignis $e \in E$ oder sein inverses Ereignis enthalten ist:*

$$\forall e \in E : e \in \tau \vee !e \in \tau$$

$U'_\Gamma \subseteq U_\Gamma$ sei im folgenden die Menge der vollständigen Spuren der Menge von Ereignissen Γ. Die Semantik der Wächter wird nun mit Hilfe von Teil-Sequenzen von vollständigen Spuren definiert. $\tau \vDash_{i,k} E$ bedeutet, dass für die Teil-Sequenz von τ zwischen i und k die Bedingung E erfüllt ist.

Definition 4.4.3 (Semantik der Wächter) *Eine Teil-Sequenz der Spur $\tau \in U_\Gamma$ erfüllt einen Wächter ($E, E_1, E_2 \in \mathcal{G}, i, k \in \mathbb{Z}$) wenn:*

$$
\begin{aligned}
&\tau \models_i E \Leftrightarrow u \models_{0,i} E\\
&\tau \models_i E(v) \Leftrightarrow (\forall c \in C : \tau \models_i E(v ::= c))\\
&\tau \models_{i,k} e \Leftrightarrow (\exists j : i \leq j \leq k \wedge \tau_j = e), e \in \Gamma\\
&\tau \models_{i,k} E_1 \vee E_2 \Leftrightarrow \tau \models_{i,k} E_1 \vee \tau \models_{i,k} E_2\\
&\tau \models_{i,k} E_1 \wedge E_2 \Leftrightarrow \tau \models_{i,k} E_1 \wedge \tau \models_{i,k} E_2\\
&\tau \models_{i,k} E_1 - E_2 \Leftrightarrow (\exists j : i \leq j \leq k \wedge \tau \models_{i,j} E_1 \wedge \tau \models_{j+1,k} E_2)\\
&\tau \models_{i,k} \top\\
&\tau \not\models_{i,k} 0\\
&\tau \models_{i,k} \neg E \Leftrightarrow \tau \not\models_{i,k} E\\
&\tau \models_{i,k} \Box E \Leftrightarrow (\forall j : k \leq j \Rightarrow u \models_{i,j} E)\\
&\tau \models_{i,k} \Diamond E \Leftrightarrow (\exists j : k \leq j \wedge u \models_{i,j} E)
\end{aligned}
$$

Definition 4.4.4 (Äquivalenz von Wächtern) *Zwei Wächter $E_1, E_2 \in \mathcal{G}$ sind äquivalent, wenn sie auf den gleichen Teil-Sequenzen der Spuren $\tau \in U'_\Gamma$ gültig sind:*

$$E_1 \equiv_\mathcal{G} E_2 ::= (\forall i, k : u \models_{i,k} E_1 \Leftrightarrow u \models_{i,k} E_2)$$

Da die Äquivalenz von Wächtern im Gegensatz zur Äquivalenz von Abhängigkeiten auf vollständigen Spuren und nicht nur auf konsistenten Spuren basiert, gilt:

$$
\begin{aligned}
&\forall e \in \Gamma : \top \equiv_\mathcal{G} e \vee !e\\
&\forall e \in \Gamma : 0 \equiv_\mathcal{G} e \wedge !e\\
&\forall e \in \Gamma : \top \equiv_\mathcal{G} \Diamond e \vee \Diamond !e
\end{aligned}
$$

Um die Bedeutung der drei neuen Operatoren zu verdeutlichen folgen nun einige Eigenschaften:

$\Box e \equiv_\mathcal{G} e$**:** Ereignisse sind stabil, d.h. wenn ein Ereignis eingetreten ist, dann ist es auch in der Spur und andersrum.

$\Diamond(e_1 - e_2) \wedge \Box e_2 \equiv_\mathcal{G} \Box(e_1 - e_2)$**:** Wenn eine Sequenz irgendwann gelten wird, und ein hinterer Teil schon gilt, dann muss auch der vordere Teil und damit die gesamte Sequenz schon gelten.

$\Diamond(e_1 - e_2) \wedge \neg e_1 \equiv_\mathcal{G} \Diamond(e_1 - e_2) \wedge \neg e_1 \wedge \neg e_2$**:** Wie die vorige Eigenschaft nur andersherum: wenn der Anfang einer Sequenz noch nicht gilt, dann kann auch der hintere Teil noch nicht gelten.

alter Wächter G	Nachricht M	neuer Wächter $G \div M$
$G_1 \vee G_2$	M	$G_1 \div M \vee G_2 \div M$
$G_1 \wedge G_2$	M	$G_1 \div M \wedge G_2 \div M$
$\Box e$	$\Box e$	$\top$
$\Box !e$	$\Box e, \Diamond e$	0
$\Diamond e$	$\Box e, \Diamond e$	$\top$
$\Diamond !e$	$\Box e, \Diamond e$	0
$\Box(e_1 - e_2)$	$\Box e_1 \wedge \neg e_2$	$\Box e_2$
$\Box(e_1 - e_2)$	$\Box e_2 \wedge \neg e_1$	0
$\Box(e_1 - e_2)$	$\Box !e_i, \Diamond !e_i, i \in \{1,2\}$	0
$\Diamond(e_1 - e_2)$	$\Box e_1 \wedge \neg e_2$	$\Diamond e_2$
$\Diamond(e_1 - e_2)$	$\Box e_2 \wedge \neg e_1$	0
$\Diamond(e_1 - e_2)$	$\Box !e_i, i \in \{1,2\}$	0
$\neg e$	$\Box e$	0
$\neg !e$	$\Box e, \Diamond e$	$\top$
$G(\$v)$	$M(c)$	$G(\$v) \wedge G(\$v ::= c) \div M(c)$
G	M	G, sonst

Tabelle 4.2: Der $\div$-Operator: Fortschreibung von Wächtern

4.4.1 Kontrolle der Abhängigkeiten mit Wächtern

Nach dem Einfügen von Abhängigkeiten werden diese zunächst in Two-Sequence-Form umgeformt und müssen dann in Wächter für die betroffenen Ereignisse übersetzt werden. Dieser Vorgang ist durch den folgenden G-Operator festgelegt.

Definition 4.4.5 *Der Operator* $\mathsf{G} : \mathcal{E} \times \Xi \rightarrow \mathcal{G}$ *bestimmt zu einem Ausdruck (in konjunktiver Normalform) die Wächter:*

$$\mathsf{G}(D_1 \vee D_2, e) ::= \mathsf{G}(D_1, e) \vee \mathsf{G}(D_2, e)$$
$$\mathsf{G}(D_1 \wedge D_2, e) ::= \mathsf{G}(D_1, e) \wedge \mathsf{G}(D_2, e)$$
$$\mathsf{G}(e_1 - \ldots - e_i - \ldots - e_n, e_i) ::= \Box e_1 \wedge \ldots \wedge \Box e_{i-1} \wedge \Diamond(e_{i+1} - \ldots - e_n)$$
$$\mathsf{G}(e_1 - \ldots - e_n, e) ::= \Diamond(e_1 - \ldots - e_n), \textit{ wenn } \{e, !e\} \not\subseteq \{e_1, !e_1, \ldots e_n, !e_n\}$$
$$\mathsf{G}(e_1 - \ldots - e_i - \ldots - e_n, !e_i) ::= 0$$
$$\mathsf{G}(0, e) ::= 0$$
$$\mathsf{G}(\top, e) ::= \top$$

Beim Eintreten eines Ereignisses werden die Wächter mit dem Assimilationsoperator $\div : \mathcal{G} \times M \rightarrow \mathcal{G}$ entsprechend Tabelle 4.2 aktualisiert. Das ganze Verfahren mit den Wächtern soll mit den folgenden Beispielen verdeutlicht werden.

4.4.2 Einfaches Beispiel mit Wächtern

Als erstes Beispiel soll die bevor-Relation von Klein dienen ($e < f$ ::=$!e \vee !f \vee e - f$). Die Wächter für die zwei Ereignisse und ihre Komplemente sind:

$$\begin{aligned} \mathsf{G}(e) &= 0 \vee \Diamond !f \vee \Diamond f = \top \\ \mathsf{G}(!e) &= \top \vee \Diamond !f \vee 0 = \top \\ \mathsf{G}(f) &= \Diamond !e \vee 0 \vee \Box e = \Diamond !e \vee \Box e \\ \mathsf{G}(!f) &= \Diamond !f \vee \top \vee 0 = \top \end{aligned}$$

Die erste Zeile wird zu $\top$, da klar ist, dass irgendwann entweder f oder $!f$ eintreten wird. Im Gegensatz dazu wird der Wächter von f nicht $\top$. f muss warten bis entweder e eingetreten ist oder bis sicher ist, dass $!e$ eintreten wird. $\Diamond e$ statt $\Box e$ reicht hier nicht aus, da e vor f geschehen muss.

4.4.3 Beispiel mit Variablen

Dieses Beispiel mit Variablen ist eine ausführlichere Version eines Beispiels von Singh [Sin97a, Example 18]. Die beiden Abhängigkeiten modellieren einen gegenseitigen Ausschluß von b_1 und b_2:

$$\begin{aligned} D_1(\$x, \$y) &= b_2[\$y] - b_1[\$x] \vee !b_1[\$x] \vee !b_2[\$y] \vee e_1[\$x] - b_2[\$y] \\ D_2(\$x, \$y) &= b_1[\$x] - b_2[\$y] \vee !b_1[\$x] \vee !b_2[\$y] \vee e_2[\$y] - b_1[\$x] \end{aligned}$$

Die Wächter für die vier Ereignisse sind:

$$\begin{aligned} G_1(b_1[\$x]) =& (\Box b_2[\$y] \vee \Diamond !b_2[\$y] \vee \Diamond(e_1[\$x] - b_2[\$y])) \\ & \wedge (\neg b_2[\$y] \vee \Box e_2[\$y]) \\ G_1(e_1[\$x]) =& \top \\ G_1(b_2[\$y]) =& (\Box b_1[\$x] \vee \Diamond !b_1[\$x] \vee \Diamond(e_2[\$y] - b_1[\$x])) \\ & \wedge (\neg b_1[\$x] \vee \Box e_1[\$x]) \\ G_1(e_2[\$y]) =& \top \end{aligned}$$

Da e_1 und e_2 immer möglich sind (es wird davon ausgegangen, dass sich die Tasks selbst korrekt verhalten), werden ihre Wächter im folgenden nicht mehr betrachtet. Jetzt tritt $b_1[1]$ ein. Damit ändern sich die Wächter zu:

$$\begin{aligned} G_2(b_1[\$x]) =& G_1(b_1[\$x]) \wedge (\Box b_2[\$y] \vee \Diamond !b_2[\$y] \vee \Diamond(e_1[x] - b_2[\$y])) \\ G_2(b_2[\$y]) =& G_1(b_2[\$y]) \wedge \underbrace{(\Box b_1[1] \vee \dots)}_{\top} \wedge (\underbrace{\neg b_1[1]}_{0} \vee \Box e_1[1]) \\ =& G_1(b_2[\$y]) \wedge \Box e_1[1] \end{aligned}$$

$b_2[\$y]$ kann nun für kein y eintreten, da dazu vorher $\Box e_1[1]$ gelten muss. $b_1[\$x]$ könnte dagegen nochmals eintreten (mit einem anderen $\$x$, da die Ereignisse eindeutig sein müssen).

Tritt nun $e_1[1]$ ein, dann nimmt der Wächter von $b_2[\$y]$ wieder seine ursprüngliche Form $G_1(b_2[\$y])$ an. Aus der Sicht des Wächters könnte dann auch $b[1]$ wieder eintreten. Das muss aber der Erzeuger des Ereignisses verhindern, da Ereignisse eindeutig sein müssen.

4.4.4 Beispiel mit Aktivitäten

Als weiteres Beispiel sollen hier wieder die Abhängigkeiten des Beispiels aus Abschnitt 4.3.2 dienen:

$$\begin{aligned}&(i[\$A] - s[\$A])\\ \wedge&((s[\$A] - c[\$A]) \vee (s[\$A] - t[\$A]))\\ \wedge&(!c[\$A] \vee !t[\$A])\\ \wedge&(t[1] - s[2])\end{aligned}$$

Damit ergeben sich zunächst die folgenden Wächter für die hier erwähnten Ereignisse:

	Wächter
$i[\$A]$	$\Diamond s[\$A] \wedge (\Diamond(s[\$A] - c[\$A]) \vee \Diamond(s[\$A] - t[\$A]))$ $\wedge(\Diamond !c[\$A] \vee \Diamond !t[\$A]) \wedge \Diamond(t[1] - s[2])$
$!i[\$A]$	0
$s[\$A]$	$\Box i[\$A] \wedge (\Diamond c[\$A] \vee \Diamond t[\$A]) \wedge (\Diamond !c[\$A] \vee \Diamond !t[\$A])$ $\wedge\Diamond(t[1] - s[2])$
$!s[\$A]$	0
$c[\$A]$	$\Diamond(i[\$A] - s[\$A]) \wedge (\Box s[\$A] \vee \Diamond(s[\$A] - t[\$A]))\wedge$ $\Diamond !t[\$A] \wedge \Diamond(t[1] - s[2])$
$!c[\$A]$	$\Diamond(i[\$A] - s[\$A]) \wedge \Diamond(s[\$A] - t[\$A]) \wedge \Diamond(t[1] - s[2])$
$t[\$A]$	$\Diamond(i[\$A] - s[\$A]) \wedge (\Diamond(s[\$A] - c[\$A]) \vee \Box s[\$A])$ $\wedge\Diamond !c[\$A] \wedge \Diamond(t[1] - s[2])$
$!t[\$A]$	$\Diamond(i[\$A] - s[\$A]) \wedge \Diamond(s[\$A] - c[\$A]) \wedge \Diamond(t[\$A] - s[\$B])$

Hier wurden zunächst nur die Regeln für die Berechnung von Wächtern angewendet. Diese Wächter können aber noch vereinfacht werden. Zunächst können konstante Terme $(\top, 0)$ berücksichtigt und logische Umformungen angewendet werden. Eine weitere Möglichkeit bietet ein Theorem von Singh (Theorem 22 in [Sin97a]), wonach Wächter für Abhängigkeiten auf $\top$ gesetzt werden können, wenn das Ereignis (direkt oder als Komplement) nicht in der Abhängigkeit enthalten ist. Mit diesen Vereinfachungen ergeben sich die folgenden Wächter:

	Wächter
$i[\$A]$	$\Diamond s[\$A]$
$!i[\$A]$	0
$s[\$A]$	$\Box i[\$A] \wedge (\Diamond c[\$A] \vee \Diamond t[\$A]) \wedge \Diamond(t[1] - s[2])$
$!s[\$A]$	0
$c[\$A]$	$(\Box s[\$A] \vee \Diamond(s[\$A] - t[\$A])) \wedge \Diamond !t[\$A]$
$!c[\$A]$	$\Diamond(s[\$A] - t[\$A])$
$t[\$A]$	$(\Diamond(s[\$A] - c[\$A]) \vee \Box s[\$A]) \wedge \Diamond !c[\$A]$
$!t[\$A]$	$\Diamond(s[\$A] - c[\$A])$

Jetzt muss der Wächter von $i[\$A]$ als wahr angenommen werden (wenn eine Aktivität initialisiert wurde, dann wird sie in diesem Modell auch irgendwann starten). Damit kann $i[\$A]$ eintreten und das soll auch für die beiden Aktivitäten 1 und 2 geschehen:

	Wächter
$i[1]$	$\Diamond(s[1] - (c[1] \vee t[1]))$
$i[2]$	$\Diamond(s[2] - (c[2] \vee t[2]))$
$s[1]$	$\Box i[1] \wedge \Diamond(c[1] \vee t[1])$
$s[2]$	$\Box i[2] \wedge \Diamond(c[2] \vee t[2]) \wedge \Box t[1]$
$c[1]$	$\Box i[1] \wedge \Box s[1]$
$c[2]$	$\Box i[2] \wedge \Box s[2]$
$t[1]$	$\Box i[1] \wedge \Box s[1]$
$t[2]$	$\Box i[2] \wedge \Box s[2]$

Die $\Diamond$-Ausdrücke verhindern nun, wie schon vorher, dass etwas geschehen kann. Soll nun $i[1]$ eintreten, dann muss sein Wächter zu $\top$ umgewandelt werden. Singh verwendet dazu Zusicherungen oder Versprechen (promises). Der Wächter schickt an die Zuständigen für alle benötigten Ereignisse ($s[1]$, $c[1]$ und $t[1]$) eine Nachricht mit der Bitte, das jeweilige Ereignis zuzusichern. Treffen dann die nötigen Zusicherungen ein, kann der Wächter sein Ereignis ($i[1]$) eintreten lassen. Die zugesicherten Ereignisse müssen dann irgendwann eintreten, was aber nicht mehr Aufgabe des Wächters von $i[1]$ ist.

Bei den folgenden Wächtern sind die beiden Ereignisse $i[1]$ und $i[2]$ eingetreten (die beiden Aktivitäten wurden initialisiert):

	Wächter
$s[1]$	$\Diamond(c[1] \vee t[1])$
$s[2]$	$\Diamond(c[2] \vee t[2]) \wedge \Box t[1]$
$c[1]$	$\Box s[1]$
$c[2]$	$\Box s[2]$
$t[1]$	$\Box s[1]$
$t[2]$	$\Box s[2]$

Nun muss sich der Wächter von $s[1]$ zusichern lassen, dass irgendwann $c[1]$ oder $t[1]$ eintreten wird. Dann kann $s[1]$ starten usw.

4.5 Vergleich der beiden Ansätze

Beim ersten Ansatz werden die Abhängigkeiten von einem zentralen Dienst verwaltet und bei eintreffenden Ereignissen jeweils angepasst. Durch die direkte Bearbeitung der Abhängigkeiten ist dieses Verfahren auch deutlich einfacher nachvollziehbar.

Im Vergleich dazu erlaubt der zweite Ansatz eine sehr verteilte Implementierung. Im Extremfall kann jedes Ereignis von einer eigenen Maschine behandelt werden und selbst lokal anhand seines Wächters entscheiden, ob es eintreten darf. Es entsteht dadurch aber ein großer Kommunikationsbedarf, da die Ereignisse und das Einfügen und Entfernen von Abhängigkeiten jeweils allen betroffenen Wächtern gemeldet werden muss.

Zusätzliche Kommunikation wird im zweiten Ansatz durch die Zusicherungen nötig. Sie werden in einigen Fällen benötigt damit Ereignisse eintreten können, deren Wächter $\Diamond$-Ausdrücke enthält. Bevor z.B. bei einer Abhängigkeit $e - f$ mit dem Ereignis e begonnen werden kann, muss sichergestellt werden, dass auch das Ereignis f irgendwann eintreten wird. Ohne die Zusicherungen würde in diesem Fall sonst keines dieser Ereignisse eintreten, da keiner der Wächter $\top$ ist.

Ein Nachteil der Zusicherungen zeigt sich bei einer Erweiterung des Beispiels aus dem vorigen Absatz. Bei der Abhängigkeit $e - f \ \vee \ e - g$ muss schon vor dem Eintreten des Ereignisses e eines der Ereignisse f und g zusichern, dass es später eintreten wird. Prinzipiell müsste diese Entscheidung aber noch nicht so früh getroffen werden.

Kapitel 5

Implementierung der EventEngine

In diesem Kapitel wird die EventEngine (Ereignismaschine) beschrieben. Die EventEngine erfüllt die folgende Funktionen:

1. Sie verwaltet die eingetretenen Ereignisse und kann andere Komponenten über die daraus resultierenden Änderungen informieren. Dies entspricht etwa der Funktionalität der EventEngine "EvE" von Geppert und Tombros [GT98], die auch in [JBS97] beschrieben wird.

2. Die EventEngine übernimmt zusätzlich auch die Aufgabe der Broker in dem oben genannten Artikel, die die Zulässigkeit von Ereignissen (events) entsprechend der von der Anwendung definierten Abhängigkeiten kontrollieren.

Die Implementierung der EventEngine basiert auf dem ersten Verfahren von Singh, bei dem die Abhängigkeiten direkt und ohne die Wächter berücksichtigt werden. Dieses Verfahren wurde wegen der einfacheren Implementierung und dem geringeren Kommunikationsaufwand gewählt. Der Nachteil der zentralen Implementierung wurde in Kauf genommen.

In diesem Kapitel wird zunächst die Schnittstelle der EventEngine beschrieben. Danach wird auf eine Änderung der Semantik des Komplements eingegangen. Als nächstes wird ein Problem bei der schrittweisen, kurzsichtigen Abarbeitung der Abhängigkeiten eingegangen. Zum Schluss wird die Behandlung von Variablen als Parameter in Abhängigkeiten beschrieben.

5.1 Schnittstelle der EventEngine

In Abbildung 5.1 ist die EventEngine und ihre Umgebung dargestellt. Oben ist die Schnittstelle zum Einfügen und Entfernen von Abhängigkeiten dargestellt. Links ist die Abfrage und das Auslösen von Ereignissen dargestellt. Die Pfeile mit einem Fragezeichen stellen eine Abfrage dar und die Pfeile mit dem Kreuz bzw. dem Haken sind die jeweiligen Antworten. Das obere Ereignis darf nicht eintreten, weil es durch die mittlere Abhängigkeit verhindert wird. Das untere

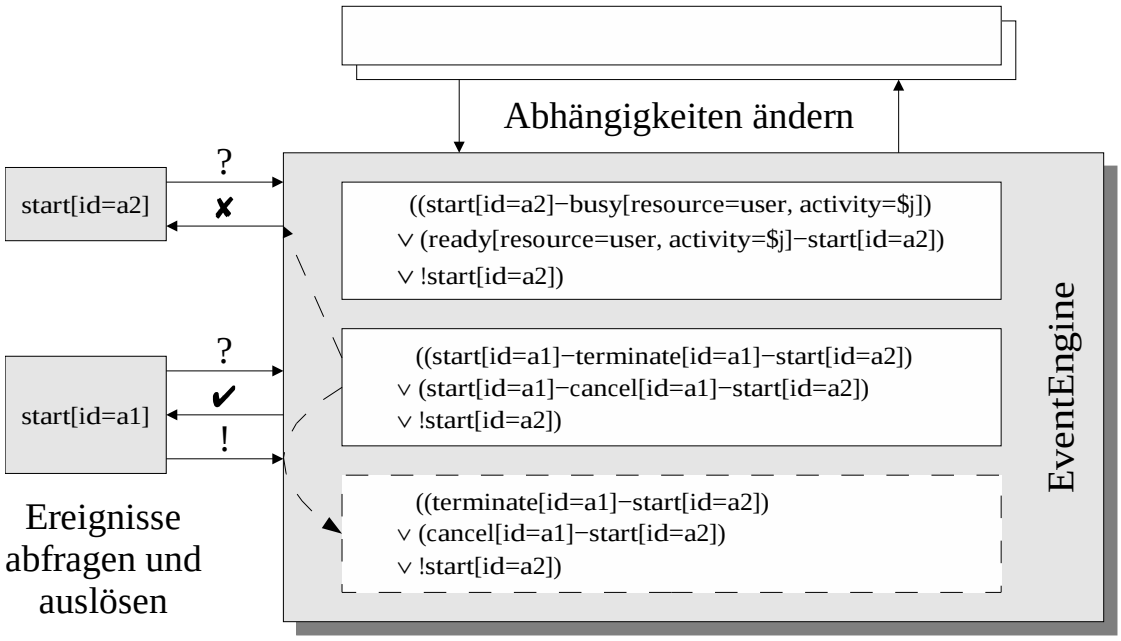

Abbildung 5.1: Die EventEngine

Ereignis darf hingegen eintreten. Durch dieses Ereignis muss die mittlere Abhängigkeit durch die untere Abhängigkeit ersetzt werden, in der das Ereignis eingearbeitet wurde.

Eine fundamentale Aufgabe der EventEngine ist zu prüfen, ob ein gegebenes Ereignis eintreten darf. Dazu dient die *query*-Methode, die zu einem Ereignis einen der folgenden Rückgabewerte liefert:

unknown: Das Ereignis ist der EventEngine nicht bekannt. Dies ist der Fall, wenn in der EventEngine keine Abhängigkeiten definiert sind, die das Ereignis erwähnen. Es gibt somit auch keine Abhängigkeiten die gegen das Eintreten des Ereignisses sprechen. Das Ereignis darf also eintreten.

possible: Das Ereignis ist bekannt und kann eintreten. Es gibt aber auch noch Alternativen zu diesem Ereignis.

required: Das Ereignis darf eintreten und es gibt auch keine Alternativen mehr dazu. Dies ist der Fall, wenn das Ereignis in allen Alternativen eines oder-Ausdrucks vorkommt.

blocked: Das Ereignis darf noch nicht eintreten, weil es in einer Sequenz einer Abhängigkeit nicht an der ersten Stelle steht. Das Ereignis muss aber irgendwann schließlich eintreten, da *query* ansonsten possible zurückgeben würde.

impossible: Dieses Ereignis darf nicht eintreten, weil es zu einem Komplement keine Alternative mehr gibt.

requested: Das Ereignis wurde schon angefordert und kann somit nicht nochmals angefordert werden. Es sollte bald eintreten.

happened: Das Ereignis ist schon eingetreten und darf deshalb nicht nochmals eintreten.

Ein Ereignis darf eintreten, wenn die *query*-Methode entweder unknown, possible oder required zurückgegeben hat. In den anderen Fällen ist es entweder verboten oder muss noch warten.

Die EventEngine muss aber nicht nur prüfen, ob ein Ereignis erlaubt ist, sondern sie muss eingetretene Ereignisse auch in die Abhängigkeiten aufnehmen. Um die dazu nötigen Funktionen zu definieren, muss zunächst geklärt werden, wie das Eintreten eines Ereignisses abläuft. Davon sind zunächst die folgenden drei Komponenten betroffen:

- Ein Client der das Ereignis auslösen möchte. Dies kann z.B. die Worklist sein, die nach dem Anklicken einer Aktivität durch den Benutzer eine Aktivität starten soll. Ein Ereignis kann aber auch von einer Aktivität oder einer beliebigen anderen Komponente im System ausgelöst werden.
- Die EventEngine, die anhand der definierten Abhängigkeiten die Zulässigkeit des Ereignisses überprüft.
- Ein Server, der die mit dem Ereignis verbundenen Aufgaben erledigt. Dies kann z.B. eine *PrimitiveActivity* sein, die bei einem *start*-Ereignis die zugehörige Anwendung starten muss.

Für den Ablauf beim Eintreten eines Ereignisses gibt es die folgenden vier, auch in Abbildung 5.2 gezeigten Möglichkeiten:

a) In diesem Szenario ruft der Client direkt die Methode des Servers auf. Der Server muss dann vor dem Erledigen der Aufgabe selbst prüfen, ob dieses Ereignis erlaubt ist. Dies ist eigentlich der sauberste Ansatz, da von außen nur die Aktivitäten sichtbar sind und ein Aufrufer nichts von der EventEngine wissen muss. Er wurde aber nicht gewählt, da dadurch Änderungen an allen schon existierenden Aktivitäten nötig gewesen wären.

b) Hier gibt der Client der EventEngine den Auftrag das Ereignis auszulösen. Die EventEngine prüft dann die Abhängigkeiten, gibt die Aufgabe an den Server weiter und passt, wenn alles gut ging, die Abhängigkeiten an. In diesem Fall müsste die EventEngine wissen, wie sie das Ereignis an den jeweiligen Server weiterreicht, damit dieser die Aufgaben erledigen kann.

c) Hier lässt der Client die Aufgabe zunächst vom Server erledigen. Nach deren Beendigung wird die EventEngine informiert. Kann diese das Ereignis nicht akzeptieren, muss der Client die Änderungen wieder zurücknehmen (undo). Dieser Ansatz ist eigentlich nur sinnvoll wenn man beispielsweise globale Transaktionen verwendet oder voraussetzen kann, dass zu jedem *do* auch ein *undo* existiert.

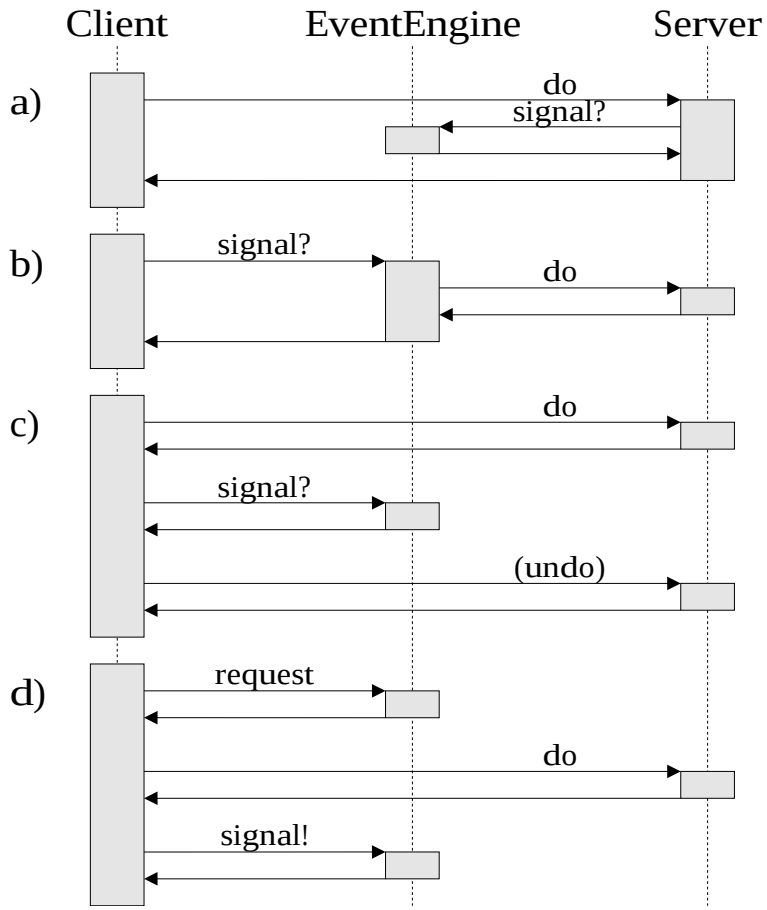

Abbildung 5.2: Alternativen für Anfragen an die EventEngine

d) Dieses Szenario entspricht dem Ablauf, den Günthör [Gün97] verwendet hat. Dabei wird die EventEngine zwei Mal aufgerufen. Zunächst frägt der Client die EventEngine mit der *request*-Methode, ob das Ereignis eintreten darf. Dabei versichert er der EventEngine auch, dass er bei einer positiven Antwort die Aufgabe erledigen wird. Nach dem Erledigen der Aufgabe meldet der Client der EventEngine dann mit der *signal*-Methode, dass die Aufgabe erledigt wurde und damit das Ereignis eintreten soll. Dieser Ansatz hat den Nachteil, dass die EventEngine sich die Menge der angeforderten Ereignisse merken und diese dann auch bei weiteren Anfragen berücksichtigen muss. Solange ein Ereignis requested ist muss sowohl die ursprüngliche Abhängigkeit als auch die schon abgearbeitete (residuierte) Abhängigkeit berücksichtigt werden.

Implementiert wurde eine einfache Variante des Ansatzes d). Dabei merkt sich die EventEngine die angeforderten Ereignisse und berücksichtigt diese auch bei Anfragen, die diese Ereignisse betreffen. Allerdings werden daraus in der Zukunft anstehende Änderungen an den Abhängigkeiten noch nicht be-

rücksichtigt. So können nach dem request eines Ereignisses durchaus Ereignisse eintreten, die das spätere signal des Ereignisses verbieten. In einem praktisch einsetzbaren System sollte das Eintreten eines Ereignisses auch durch eine globale Transaktion zwischen EventEngine, Client und Server gesichert werden.

Die EventEngine bietet auch noch Funktionen, mit denen sich andere Objekte im System auf Zustandsänderungen von Ereignissen registrieren lassen können. Es ist damit z.B. möglich, eine Aktion auszulösen, wenn ein Ereignis possible oder impossible wird. Dies kann beispielsweise bei selbst-startenden Aktivitäten (siehe Abschnitt 6.7) verwendet werden.

5.2 Behandlung des Komplements

Nach der Definition müsste bei der Abhängigkeit $a-!b-c$ zeitlich zwischen dem Ereignis a und dem Ereignis c explizit die Entscheidung gegen das Ereignis b getroffen werden.

Das explizite Auslösen der Komplemente durch den Benutzer hat den Vorteil, dass der Benutzer diese Entscheidung selbst fällt und sie somit auch kontrollieren kann. Dagegen ist ist das natürlich zusätzliche Arbeit, die man unter Umständen einsparen kann. Im folgenden wird eine Möglichkeit gesucht, wie die Entscheidung für ein Komplement nicht explizit gefällt werden muss, sonder die Komplemente nur verhindern, dass das zugehörige Ereignis eintritt. Die folgende Auflistung zeigt einige untersuchte Ansätze:

1. Soll ein Ereignis einer Sequenz eintreten, vor dem noch einige Komplemente stehen, könnten die Komplemente in eine Menge aufgenommen werden, die in der Zukunft berücksichtigt wird. Das Ereignis könnte danach eintreten. Allerdings ist dieser Ansatz nicht korrekt: Das Komplement könnte vorübergehend oder für immer durch Constraints verhindert sein, z.B. weil das eigentliche Ereignis erzwungen ist. Für die Komplemente müsste also zunächst auch die query-Methode aufgerufen werden. Durch diese Aufrufe könnten weitere Komplemente notwendig werden, die auch wieder überprüft werden müssten. Das wird nicht nur sehr aufwändig sondern für die Komplemente muss auch eine sinnvolle Reihenfolge ermittelt werden, da die Reihenfolge Einfluss auf die verbleibenden Alternativen haben kann. Daher kann die Reihenfolge aber kaum ohne Hilfe durch den Benutzer festgelegt werden.

   Mit diesem Ansatz ist es somit möglich, den vermutlich größten Teil der Komplemente automatisch eintreten zu lassen. Nur in Problemfällen muss der Benutzer durch Auswahl der Reihenfolge der Komplemente eingreifen.

2. Eine zweite Möglichkeit, mit der zumindest einige Komplemente selbstständig ausgelöst werden könnten, ist, bei einem *terminate*-Ereignis auch das entsprechende *!cancel* auszulösen und andersrum. Allerdings können

auch hier die Komplemente blockiert oder gar verboten sein, so dass das *!cancel*-Ereignis nicht direkt nach dem *terminate* eintreten kann. Es ergibt sich also das gleiche Problem wie im ersten Fall.

3. Als weitere Möglichkeit können die erlaubten Abhängigkeiten so eingeschränkt werden, dass die Überprüfung nicht so aufwändig ist. Dies erreicht man z.B. wenn Komplemente nicht in Sequenzen verwendet werden dürfen. Damit kann es vor einem Komplement keine weiteren Komplemente und Ereignisse geben, die vorher eintreten müssen, und auch vor einem Ereignis müssen keine Komplemente eintreten.

Realisiert wurde der dritte Ansatz. Er erleichtert die Arbeit des Benutzers, da er weniger Ereignisse eintreten lassen muss. Zudem ist die Beschränkung der erlaubten Abhängigkeiten in der Praxis keine große Einschränkung. Alle Beispiel-Constraints hatten auch schon vor dem Aufstellen der Regel eine entsprechende Form.

5.3 Probleme bei der schrittweisen Abarbeitung

Die schrittweise Abarbeitung der Abhängigkeiten hat das Problem, dass sie nicht vorausschauend ist. Dadurch können Ereignisfolgen erlaubt werden, die das System in eine "Sackgasse" bringen, in der die Erfüllung aller Abhängigkeiten nicht mehr möglich ist. Ein solcher Problemfall sind z.B. die folgenden beiden Abhängigkeiten:

1		$a - b$
2	$\vee$	$b - a$
3		$!c$

Hier darf eigentlich c nicht eintreten, da sonst der Widerspruch aus den Zeilen 1 und 2 nicht mehr umgehbar ist. Dieses Problem könnte vermieden werden, wenn man die folgende Regel berücksichtigt: Sobald die Reihenfolge von Ereignissen feststeht, weil sie in allen Alternativen eines Oders vorkommt, können alle Komplemente der in der Sequenz enthaltenen Ereignisse und alle anderslautenden Sequenzen als nicht-erfüllbar gestrichen werden.

Die problematischen Fälle sind aber nicht immer so einfach zu erkennen:

			a	
1	$\vee$	$a - c$	c	c
2		$b - !c$	$b - !c$	0
3	$\vee$	$a - !c$	$!c$	$!c$
4		$b - c$	$b - c$	0
5	$\vee$	$!a$	0	0
6		$!b$	$!b$	$!b$

Wenn in diesem Beispiel a eintritt, dann darf wegen den Zeilen 5 und 6 b nicht mehr eintreten. Nach der obigen Regel können dann die Zeilen 2 und 4 als unerfüllbar gestrichen werden, was in der letzten Spalte gemacht wurde. Wie man sieht muss nun sowohl c als auch $!c$ eintreten, was nicht erlaubt ist. Ähnlich verhält es sich, wenn mit b begonnen wird.

In diesem Beispiel erkennt man erst nach einigen Umformungen der ersten vier Zeilen das Problem:

$$\begin{aligned}
&(a - c \vee b - !c) \wedge (a - !c \vee b - c) \\
=&(a - c \wedge a - !c) \vee \underbrace{(a - c \wedge b - c)}_{0} \\
&\vee \underbrace{(b - !c \wedge a - !c)}_{0} \vee (b - !c \wedge b - c) \\
=&(a - c \wedge b - c) \vee (b - !c \wedge a - !c) \\
=&(a \wedge b) - c \vee (b \wedge a) - !c \\
=&(a \wedge b) - (c \vee !c)
\end{aligned}$$

Hier werden über die Sequenzen zu den c-Ereignissen in den Zeilen 1 bis 4 die beiden Ereignisse a und b erzwungen, was im Widerspruch zu den Zeilen 5 und 6 steht. Das Problem ist hier also nicht ein Zyklus sondern die Unerfüllbarkeit der Formel.

Die Überprüfung, ob eine Menge von Abhängigkeiten erfüllbar ist, ist aber NP-vollständig, d.h. mit einiger Sicherheit nicht in polynomieller Zeit bestimmbar. Ohne den Sequenz-Operator ist das Problem das klassische Satisfiability-Problem (SAT), wobei mit den Variablen entschieden wird, ob ein Ereignis eintreten darf oder nicht. Der zusätzliche Operator macht das Problem nur noch schwerer, die Erkennung von Zyklen ist aber zumindest nicht NP-vollständig.

In der Literatur wird dieses Problem auf unterschiedliche Weisen behandelt:

- Singh betrachtet dieses Problem nicht. Er entfernt zwar entsprechend der bei ihm definierten Eigenschaften von Ereignissen Alternativen in Abhängigkeiten, die nicht erzwingbar sind, dies wirkt sich aber nur innerhalb einer Abhängigkeit aus. Widersprüche, die sich aus der Menge der Abhängigkeiten im Laufe der Abarbeitung ergeben, werden nicht berücksichtigt.

- In [ASSR93, ASE+96] wird ein Scheduling-Algorithmus für die Abhängigkeiten von Klein angegeben, der nach lebensfähigen Pfaden sucht. In dem Artikel wird für jede Abhängigkeit ein Automat erstellt, der den Zustand der jeweiligen Abhängigkeit repräsentiert. Soll nun ein Ereignis eintreten wird ein lebenfähiger Pfad gesucht, der von allen Automaten akzeptiert wird und auf dem nur bereits geschehene oder erzwingbare Ereignisse enthalten sind. Hier wird also jeweils eine Lösung für das NP-vollständige Problem gesucht bevor ein Ereignis akzeptiert wird. In [ASE+96] wird

vermutet, dass die bei einigen Workflow-Anwendungen auftretenden Problem-Instanzen effizient lösbar sind: "Although the worst case time complexity is still exponential, we have reason to believe that in many interesting cases, e.g., certain workflows in telecommunications applications, the time complexity is polynomial".

- Auch [Gün97] verwendet als Grundlage für seine Abhängigkeitsverwaltung die Primitive von Klein. Günthör sorgt stets dafür, dass das System weiter arbeiten kann ohne in eine "Sackgasse" zu gelangen, in der einige Abhängigkeitsregeln nicht mehr erfüllt werden können. Durch eine Einschränkung der erlaubten Abhängigkeiten kann er die Zulässigkeit eines Ereignisses durch Tests sicherstellen, die nur ein polynomielle Laufzeit haben. Die wesentliche Einschränkung ist dabei, dass auf den rechten Seiten aller $<$-Ausdrücke einer Abhängigkeitsregel das gleiche Ereignis stehen muss. Ist das nicht der Fall wird die Abhängigkeit in mehrere per und-Operator verknüpfte Regeln aufgeteilt. Der Vorteil ist, dass bei der Suche nach Zyklen kein Backtracking gemacht werden muss um alternative Reihenfolgen zu testen. Diese Einschränkung ist mir für CASSY aber zu streng, da damit keine alternativen Reihenfolgen mehr spezifiziert werden können. So lässt sich z.B. keine Abhängigkeit definieren, mit der die nicht-gleichzeitige Ausführung zweier Aktivitäten festgelegt wird.

Im Aktivitäten-Management sollen Historien nicht schon dann ausgeschlossen werden, wenn eine erfolgreiche Beendigung nicht garantiert werden kann. Allerdings sollten Situationen, in denen irgendwann ein Constraint verletzt werden muss, möglichst durch das System erkannt und vermieden werden. Zur Behandlung dieses Problems gibt es die folgenden Möglichkeiten:

- Nichts tun. Die Benutzer müssen selbst aufpassen und wenn Constraints nicht mehr erfüllbar sind, dann sind sie eben nicht mehr erfüllbar. Der Benutzer muss dann durch Entfernen von Constraints eingreifen. Die Event-Engine sorgt aber auf jeden Fall dafür, dass kein Constraint verletzt wird.

- Die Constraints so definieren, dass einfache Fälle verhindert werden. So kann z.B. bei einem verbotenen *terminate*-Ereignisse auch ein dann unter Umständen erst noch folgender Start der Aktivität verhindert werden. Dies soll am Beispiel des *xor*-Constraints gezeigt werden, der in der erweiterten Form in Abbildung 5.3 zu sehen ist.

  Die ersten vier Zeilen sind der eigentliche *xor*-Constraint, wie er auch im Anhang definiert ist. Die restlichen Zeilen sorgen dafür, dass nach der Terminierung einer der beiden Aktivitäten die andere nicht mehr starten kann. Wie man sehen kann werden die Constraints durch diese Erweiterung deutlich komplexer. Um die vordefinierten Constraints aber nicht zu kompliziert zu machen, wurden solche Erweiterungen dort vorerst nicht durchgeführt.

$$
\begin{array}{lll}
 & (& terminate[id = \$0.id] \\
 & \vee & terminate[id = \$1.id] \quad) \\
\wedge & (& !terminate[id = \$0.id] \\
 & \vee & !terminate[id = \$1.id] \quad) \\
\wedge & (& start[id = \$0.id] - terminate[id = \$1.id] \\
 & \vee & !start[id = \$0.id] \\
 & \vee & !terminate[id = \$1.id] \quad) \\
\wedge & (& start[id = \$1.id] - terminate[id = \$0.id] \\
 & \vee & !start[id = \$1.id] \\
 & \vee & !terminate[id = \$0.id] \quad)
\end{array}
$$

Abbildung 5.3: Ein erweiterter *xor*-Constraint

- Regelmäßig, am besten wenn gerade sonst nicht viel zu tun ist, kann ein Algorithmus nach Zyklen und unerfüllbaren Teil-Abhängigkeiten suchen und die Abhängigkeiten anpassen. Dadurch kann natürlich nicht verhindert werden, dass zwischen zwei Läufen des Algorithmus Ereignisse akzeptiert werden, durch die Constraints unerfüllbar werden. Jedoch können einige Fälle abgefangen werden.

  Das Entfernen von Alternativen, die aufgrund der aktuellen Situation nicht mehr möglich sind, kann aber auch Nachteile haben. Wird später eine der Abhängigkeiten aus dem System entfernt, wegen der die Alternative unmöglich war, bleibt die Alternative zunächst weiterhin unmöglich. Beim Entfernen von Abhängigkeiten sollten also die gelöschte Alternativen in anderen Abhängigkeiten wieder eingefügt werden.

- Jedes Mal, wenn durch das Akzeptieren eines Ereignisses eine Alternative in einem Oder wegfällt, muss man prüfen, ob die restlichen Alternativen zusammen mit den anderen definierten Abhängigkeiten noch erfüllbar sind. Dies entspricht dem oben erwähnten Verfahren nach [ASE+96]. Allerdings ist das Lösen eines NP-vollständigen Problems bei jeder Überprüfung eines Ereignisses nicht akzeptabel.

- Nur in einer begrenzten Umgebung des aktuellen Zustandes die Erfüllbarkeit der Abhängigkeiten prüfen. Bei der Suche nach einer von den Abhängigkeiten akzeptierten Historie wird hier nach einer bestimmten Anzahl von Schritten abgebrochen.

  Der Algorithmus liefert nun eines von drei Ergebnissen: erfüllbar, unerfüllbar oder unbekannt. Auf das unbekannt-Ergebnis kann auf zwei Weisen reagiert werden:

  – Das Ereignis wird abgelehnt. Hier ist wie bei der vollständigen Suche die Fortsetzbarkeit garantiert. Es können aber Ereignisse verhindert

werden, die eigentlich eintreten dürften.

– Das Ereignis wird akzeptiert. Die Hoffnung ist dabei, dass Widersprüche möglichst bald herleitbar sind. Der Nachteil ist, dass der unter Umständen auch vom Bearbeiter noch überschaubare Bereich der Abhängigkeiten vom System kontrolliert wird, und im Einzelfall doch nach einer längeren Berechnung eine Unerfüllbarkeit herleitbar ist.

Beide Varianten haben ihre Nachteile, wobei ich für CASSY die zweite Variante bevorzugen würde.

Die Implementierung behandelt dieses Problem noch nicht. Sie kann also durch eine Folge von erlaubten Ereignissen in einen Zustand kommen, in dem nicht mehr alle Abhängigkeiten erfüllbar sind. Das ist kein Fehler des Systems oder der Implementierung, da kein Constraint verletzt wird. Dies ist aber für den Benutzer nicht die beste Unterstützung bei seiner Arbeit. Deshalb wäre die Implementierung des letzten Ansatzes in der optimistischen Variante durchaus sinnvoll. Dabei spielt die Wahl der Grenze für die Suche und die Reihenfolge, in der gesucht wird, eine wichtige Rolle.

5.4 Variablen in Ereignissen

Ereignisse müssen eindeutig sein, dürfen also nur ein Mal eintreten. Um dennoch wiederholende Ereignisse, wie sie z.B. in Schleifen vorkommen, zu erlauben, werden Variablen in den Ereignissen zugelassen. Ein Ereignis mit einer Variablen als Parameter kann dann für jede Bindung der Variablen an eine Konstante ein Mal eintreten.

Die Behandlung der Variablen wird hier extra beschrieben, da sie zum einen eine Besonderheit der Arbeit von Singh ist, und weil hier auch einige Teile falsch und nicht ganz klar waren.

Nach der Regel $E(\vec{v})/e[\vec{c}] = E(\vec{v}) \wedge E(\vec{v} ::= \vec{c})/e[\vec{c}]$ von Singh könnten alle Instanzen einfach per UND verknüpft werden. Allerdings stimmt diese Regel nicht ganz. Als Gegenbeispiel soll die Abhängigkeit $a[\$x] - b[\$x]$ dienen. Tritt hier $a[1]$ ein wird die Abhängigkeit zu $a[\$x] - b[\$x] \ \wedge \ b[1]$. Hier kann $b[1]$ aber noch nicht eintreten, weil es durch die Sequenz noch verhindert wird. Die Folge davon ist, dass $b[1]$ und auch kein anderes $b[\$x]$ jemals eintreten kann.

In einem Beispiel wendet Singh eine bessere Regel an, die auch schon in Abschnitt 4.3.1 angegeben wurde:

$$E(\vec{v})/e[\vec{c}] = (\forall \vec{d} : \vec{d} \neq \vec{c} \Rightarrow E(\vec{v})) \wedge E(\vec{v} ::= \vec{c})/e[\vec{c}]$$

Hier ist nun die nicht instantiierte Regel aus dem Beispiel von oben explizit nicht mehr für alle Ereignisse mit der Bindung $\$x = 1$ zuständig.

Die Regel kann so aber prinzipiell auf verschiedenen Teil-Ausdrücke einer Abhängigkeit angewendet werden. Um Mehrdeutigkeiten zu vermeiden, und

damit auch die Normalform der Abhängigkeit nicht zerstört wird, sollte die Regel immer für die ganze Abhängigkeit und nicht nur auf einen Teilausdruck angewendet werden. Die Semantik der Variablen soll mit den folgenden Beispielen verdeutlicht werden.

5.4.1 Beispiel: Ressource verfügbar

Ein praktisches Beispiel für die Verwendung von Variablen ist die folgende Abwandlung des Beispiels von Singh (gegenseitiger Ausschluss, [Sin97b, Example 13]). Hier wird damit die Verfügbarkeit einer Ressource vor dem Start einer Aktivität sichergestellt. Sie lautet:

$$start[] - busy[\$j] \vee !start[] \vee ready[\$j] - start[]$$

Der Parameter $\$j$ bei ready und busy gibt die Aufgabe (Job-ID) an, an der die Ressource arbeitet. Er ist wichtig um die beiden Ereignisse einander zuordnen zu können. $busy[12]$ tritt ein, wenn die Ressource mit der Aufgabe 12 begonnen hat, $ready[12]$ tritt ein, wenn die Ressource die Aufgabe beendet hat und $start[]$ ist das Start-Ereignis der Aktivität, für die die Ressource benötigt wird (die Aktivitäten-ID wurde hier weggelassen).

Ereignis	*Rest*
$busy[1]$	$!start[] \vee ready[1] - start[]$
$ready[1]$	$start[] - busy[1] \vee !start[] \vee start[]$
$start[]$	$busy[\$j]$

Die Abhängigkeit ist dabei so aufgebaut, dass die Instanzen auch per UND hinzugefügt werden könnten. Der Grund dafür ist, dass die Abhängigkeit mit jedem Ereignis beginnen kann. Es bleiben dann aber unter Umständen Reste übrig, wie z.B. ein $busy[1]$, wenn mit $ready[1]$ begonnen wurde.

Die Abhängigkeit wird zwar nie erfüllt weil der Rest nach dem $start[]$ nie erfüllt. Die Ressource ist aber zumindest verfügbar wenn die Aktivität gestartet wird.

Wenn wie in diesem Beispiel ein Ereignis nur zwischen zwei zusammengehörenden Ereignissen (hier $busy$ und $ready$) eintreten darf, dann müssen die beiden Ereignisse durch die gleiche Bindung aller Variablen einander zugeordnet werden können. Im Beispiel war das die Job-ID, es könnte aber auch ein einfacher Zähler verwendet werden, der bei jedem $busy$ um eins erhöht wird.

Soll mit einem Constraint erreicht werden, dass ein Ereignis nur zwischen 9:00 Uhr und 17:00 Uhr eintreten darf, kann im Beispiel $busy$ und $ready$ durch Zeit-Ereignisse (siehe Abschnitt 6.6) ersetzt werden, bei denen die Variable das Datum ist:

$$\begin{aligned} & start[] - time[t = 17:00, d = \$d] \\ \vee\ & !start[] \\ \vee\ & time[t = 9:00, d = \$d] - start[] \end{aligned}$$

5.4.2 Beispiele mit Variablen bei Singh

Bei Singh gibt es zwei Beispiele in denen mehrere Variablen vorkommen. Im Beispiel zum gegenseitigen Ausschluss wird aber nur eine Variable instantiiert, die andere Variable sorgt dafür, dass das ausgeschlossene Ereignis für alle Parameter nicht eintritt.

Im Beispiel der Iteration werden die Variablen entlang der Iteration gebunden. Es werden dabei stets alle Variablen durch das vorderste Ereignis einer Sequenz gebunden. Damit werden nur von den deklarierten Abhängigkeiten Instanzen erzeugt und es gibt keine Instanzen von Instanzen.

Auch für eigene Abhängigkeiten ist es durchaus sinnvoll, nicht mit beliebigen Instantiierungsfolgen zu arbeiten.

5.4.3 Einige konstruierte Beispiele

Zur Klärung des Verhaltens von Abhängigkeiten mit Variablen folgen nun einige konstruierte Beispiele:

$a - b[v = \$v]$: Bevor ein beliebiges b eintreten darf, muss das a eintreten. Ansonsten wird die gesamte Abhängigkeit unerfüllbar, was auch sinnvoll ist.

$a[u = \$u] - b$: Hier müssen erst alle as (d.h. für alle möglichen Werte von u) eintreten bevor b eintreten darf. Das wird praktisch nie der Fall sein, da die Werte der Parameter beliebige Strings sein können.

$a[u = \$u] - b[v = \$v]$: Alle as müssen vor allen bs kommen. Nach dem Ereignis $b[v = 3]$ ergibt sich:

$$(\forall \$v : \$v \neq 3 \Rightarrow a[\$u] - b[\$v]) \wedge (\underbrace{a[u = \$u] - b[v = 3]/b[v = 3]}_{=0})$$

Es darf also am Anfang kein b eintreten. Wie sieht es aber aus, wenn vorher schon ein a eingetreten ist? Im folgenden Beispiel ist nun $a[u = 1]$ eingetreten:

$$(\forall \$v : \$u \neq 1 \Rightarrow a[\$u] - b[\$v]) \wedge (\underbrace{a[u = \$u] - b[v = \$v]/a[u = 1]}_{=b[v=\$v]})$$

Die ganze Abhängigkeit wird also unerfüllbar, wenn ein b eintritt ohne dass vorher ein a eingetreten ist.

$a[u = \$u] - b[u = \$u]$: Das ist noch einmal das Beispiel vom Anfang dieses Abschnitts, bei dem im Gegensatz zum vorigen Beispiel beide Ereignisse

die gleiche Variable binden. Tritt hier z.B. $a[u = 1]$ ein ändert sich die Abhängigkeit zu

$$\begin{aligned}&(\forall \$u : \$u \neq 1 \Rightarrow a[u = \$u] - b[u = \$u])\\ \wedge&(\underbrace{a[u = 1] - b[u = 1]/a[u = 1]}_{=b[u=1]})\end{aligned}$$

Hier darf nun also $b[u = 1]$ eintreten aber kein anderes b.

5.4.4 Instantiierungsbäume

Zur Implementierung dieser Semantik der Bindung von Variablen wird ein Instantiierungsbaum verwendet. Dabei werden neue Instanzen an die Abhängigkeit angehängt, von der aus sie instantiiert wurden. Die Kante wird mit der Bindung der Variablen beschriftet.

Soll geprüft werden, ob ein Ereignis eintreten darf (*query*), müssen alle betroffenen Abhängigkeiten betrachtet werden. Für eine Abhängigkeit werden dabei zunächst rekursiv alle Instanzen überprüft, die eine verträgliche Bindung der Variablen haben. Verträglich heißt hier, dass es keine Variable gibt, die an einen anderen Wert gebunden ist. Wurde dabei keine Instanz gefunden, die mindestens eine weitere Variable gleich bindet, so muss schließlich auch die Abhängigkeit selbst geprüft werden. Das Ergebnis der Überprüfung muss wie bei einer und-Verknüpfung der einzelnen Instanzen berechnet werden.

Auch wenn ein Ereignis in die Abhängigkeiten eingearbeitet wird (*signal*), müssen zunächst die verträglichen Instanzen rekursiv betrachtet werden. Wurde dabei keine Instanz gefunden, in der alle durch das Ereignis erzwungenen Bindungen enthalten sind, so muss schließlich eine neue Instanz mit diesen Bindungen angelegt werden. Das Ereignis muss in der Abhängigkeit selbst eingearbeitet werden, wenn durch das Ereignis keine zusätzlichen Variablen gebunden wurden.

Abbildung 5.4 soll diesen Ablauf verdeutlichen. Die dicken Pfeile stehen für das Anlegen einer neuen Instanz, bei den dünnen Linien werden die vorhanden Instanzen verändert. Zunächst tritt das *busy*-Ereignis ein, durch das eine neue Instanz der Abhängigkeit mit der Bindung $\$j = other$ angelegt. Als zweites wird der Start der Aktivität act versucht. Die ursprüngliche Abhängigkeit hat auch nichts dagegen (es bleibt ein Rest übrig, aber das stört nicht), aber die neue Instanz würde damit unerfüllbar werden. Deshalb muss dieses Ereignis abgelehnt werden. Als drittes tritt das *ready*-Ereignis zum vorangegangenen *busy*-Ereignis ein. Das vierte Ereignis ist ein *ready*-Ereignis zu dem das zugehörige *busy*-Ereignis eigentlich fehlt. Dies soll hier nur zeigen, dass die Abhängigkeit das akzeptiert, sie kann aber ohne das *busy*-Ereignis natürlich nicht das Starten der Aktivität verhindern. Das letzte Ereignis ist das *start*-Ereignis der Aktivität, das nun akzeptiert wird, da die drei betroffenen Instanzen alle zu $\top$ werden.

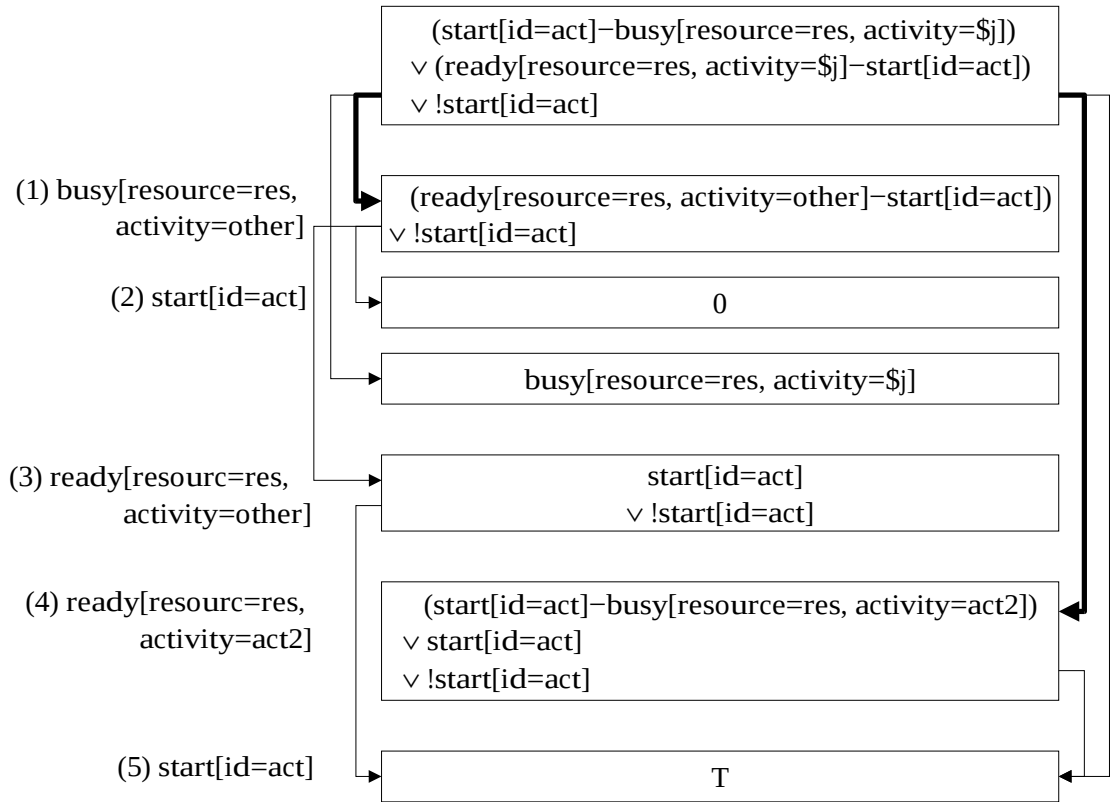

Abbildung 5.4: Die Abarbeitung von Instanzen am Beispiel der Verfügbarkeit einer Ressource

5.4.5 Verhalten beim Scheitern einer Instanz

Die EventEngine soll zwar die Abhängigkeiten beim Akzeptieren von Ereignissen beachten, einzelne Ereignisse können aber auch erzwungen werden. Der Grund dafür ist, dass das darauf aufbauende Aktivitäten-Management eher eine unterstützende bzw. führende und weniger eine die korrekte Ausführung erzwingende Funktion hat. Wurde ein Ereignis erzwungen, obwohl eine Abhängigkeit dies eigentlich nicht erlaubt hat, dann wird diese Abhängigkeit als "unerfüllbar" markiert.

Wird nun eine Instanz einer Abhängigkeit mit Variablen unerfüllbar, so wird nach Definition 4.2.5 die gesamte Abhängigkeit unerfüllbar. Tritt z.B. bei dem Constraint $a[u = \$u] - b[u = \$u]$ das Ereignis $b[u = 2]$ ein, bevor das zugehörige Ereignis $a[u = 2]$ eingetreten ist, so wird der Constraint zu 0 (unerfüllbar) und damit in der Folge nicht mehr betrachtet. Eine unerfüllbare Abhängigkeit kann nicht noch "unerfüllbarer" werden.

In CASSY möchte man aber eigentlich schon verhindern, dass nach einem zu früh eingetretenen b-Ereignis nicht in der Zukunft alle Folgen von as und bs erlaubt werden. Deshalb wird hier die Semantik entsprechend abgeändert. Dazu wird eine Instanz beim Scheitern durch 0 ersetzt und dieses Ergebnis wird

auch weitergemeldet. Bei den folgenden Anfragen, ob ein Ereignis eintreten darf, wird diese Instanz aber nicht mehr berücksichtigt. Die gescheiterten Instanzen sind so aber dennoch abfragbar und die Abhängigkeit selbst gilt auch als gescheitert.

Diese Änderung hat keinen Einfluss auf die normale Arbeit der Event-Engine. Sie wird erst relevant wenn durch ein erzwungenes Ereignis ein Constraint schon verletzt wurde und die Historie damit eigentlich schon nicht mehr korrekt ist.

Kapitel 6

Aktivitäten-Management

In diesem Kapitel wird gezeigt, wie aufbauend auf der EventEngine aus dem vorigen Kapitel ein Aktivitäten-Management realisiert werden kann. Dieses Aktivitäten-Management soll dafür sorgen, dass an die Aktivitäten gestellte Restriktionen (die Constraints) möglichst eingehalten werden.

Zunächst wird die gesamte Architektur vorgestellt bevor auf die Designflow-Aktivitäten und Constraints eingegangen wird. Anschließend werden Ansätze zur Realisierung von mehrfach auftretenden Aktivitäten oder Aktivitätentypen vorgestellt. Schließlich werden noch Constraints beschrieben, die sich auf Ressourcen und Zeiten beziehen. Den Abschluss bildet die Betrachtung von selbststartenden bzw. automatischen Aktivitäten.

6.1 Architektur

In diesem Abschnitt wird der Aufbau des Aktivitäten-Managers und die Umgebung des Aktivitäten-Managements beschrieben. Die für das Aktivitäten-Management relevanten Komponenten, wie in Abbildung 6.1 dargestellt, sind:

- Worklist: Der Benutzer hat auf seiner Worklist die Aktivitäten, die er starten und bearbeiten darf. Eine Aktivität wird dabei in der Regel nur einmal in die Worklist eines Users eingetragen. Danach kann die Aktivität aber durchaus zeitweise (oder sogar für immer) durch Constraints verhindert werden. Diese unausführbaren Aktivitäten werden nicht entfernt, damit der Benutzer sehen kann, welche Aktivitäten noch anstehen, verhindert oder gescheitert sind. Er kann dann z.B. einen Constraint entfernen um die unausführbare Aktivität doch startbar zu machen. Die gerade nicht ausführbaren Aktivitäten können bei der Anzeige der Worklist aber auch ausgeblendet werden.

- Aktivitäten-Manager: Der Aktivitäten-Manager prüft auf Anfrage von Benutzern oder Aktivitäten, ob ein Ereignis (*init*, *start*, *stop*, *resume*, *terminate*, *cancel*, *reset*) eintreten darf. Ist das Ereignis erlaubt ruft der Aktivitäten-Manager dann die entsprechende Methode der Aktivität auf.

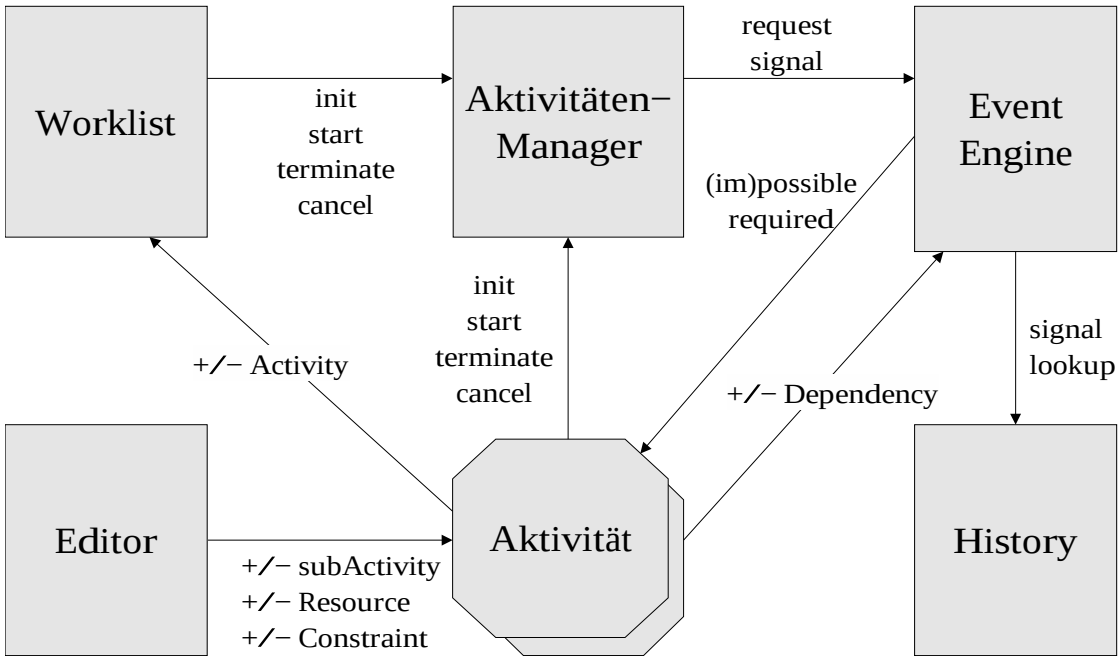

Abbildung 6.1: Architektur des Aktivitäten-Managements

- Aktivitäten: Die hier im wesentlichen betrachtete Designflow-Aktivität verwaltet ihre Sub-Aktivitäten und Constraints. Eine Designflow-Aktivität wird von der EventEngine informiert, wenn triggerbare Ereignisse ausführbar werden oder wenn notwendige Ereignisse unausführbar werden.

- EventEngine: Die EventEngine arbeitet entsprechend der eintreffenden Ereignisse die definierten Abhängigkeiten nach dem in Kapitel 5 beschriebenen Verfahren ab. Zudem können sich andere Objekte auf Zustandsänderungen von Ereignissen registrieren lassen und werden dann gegebenenfalls benachrichtigt.

- Historie: In der Historie werden die eingetretenen Ereignisse gespeichert. Sie kann beim Starten zur Rekonstruktion des Zustandes beim letzten Beenden verwendet werden. Zusätzlich sorgt sie aber auch dafür, dass der Ablauf der Aktivitäten, und damit auch die getroffenen Entscheidungen, nachvollzogen werden können.

6.2 Aufbau und Ablauf von Designflow-Aktivitäten

Die Designflow-Aktivität, die in dieser Arbeit im wesentlichen entwickelt wird, ist vergleichbar mit einer Workflow-Aktivität (siehe [Kuh00, Jab95]). Es sollen hier aber eher unstrukturierte und noch länger laufende Aktivitäten unterstützt werden. Der Schwerpunkt liegt auch nicht so sehr auf dem Vorantreiben eines

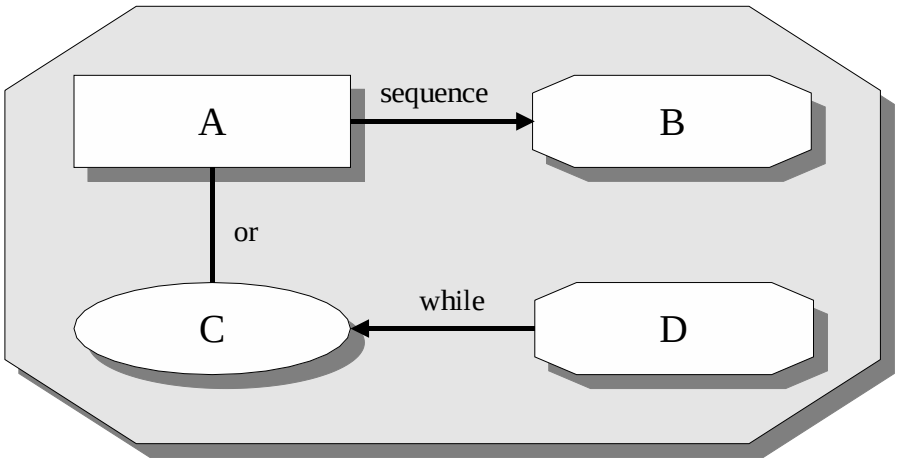

Abbildung 6.2: Ein Designflow mit vier Sub-Aktivitäten

Prozesses, sondern vielmehr in der Unterstützung eines Mitarbeiters. Bei lange laufenden Aktivitäten, wie z.B. der Entwicklung von Software, steht der vollständige Ablauf mit allen Sub-Aktivitäten am Anfang oft noch nicht fest. Die Designflow-Aktivität soll daher noch während der Laufzeit änderbar sein, so dass nachträglich weitere Aktivitäten hinzugefügt und Constraints hinzugefügt oder entfernt werden können.

Sehr ausführlich wurden drei Beispiele für das ganze System CASSY in [Mar98] entwickelt. Das eine Beispiel behandelt dabei die Softwareentwicklung, das andere ein Maschinenbau-Szenario und das dritte Abläufe in einem Krankenhaus. Ein weiteres Beispiel in [Fra00] beschreibt die Bestimmung eines neuen Standortes für eine Filiale.

Eine Designflow-Aktivität kann, als Unterklasse der *BasicActivity*, Sub-Aktivitäten besitzen. Diese Sub-Aktivitäten werden aber nicht nur zu einer Gruppe (z.B. entsprechend einer Teil-Aufgabe) zusammengefasst, sondern es können zwischen den Sub-Aktivitäten Constraints festgelegt werden. Ein Constraint wird als Abhängigkeit bei der EventEngine angemeldet und dort dann bei den eintreffenden Ereignissen berücksichtigt. Constraints können z.B. die sequentielle Abarbeitung zweier Aktivitäten festlegen, oder definieren, dass der Start einer Aktivität vor dem Start einer anderen geschehen muss (siehe Abschnitt 6.3).

In Abbildung 6.2 ist ein Designflow mit vier Sub-Aktivtäten zu sehen. Die unterschiedlichen Formen der Sub-Aktivitäten repräsentieren jeweils einen Aktivitätentyp, der aus der Sicht der Koordinierung der Aktivitäten aber keine Rolle spielt. Zwischen den Aktivitäten A und B ist ein *sequence*-Constraint definiert, der besagt, dass erst nach einem *terminate* oder *cancel* der Aktivität A die Aktivität B gestartet werden darf (die Definition der unterschiedlichen Constraints ist im Anhang zu finden). Die Aktivität A muss aber aufgrund der Definition von *sequence* auf jeden Fall starten. Beendet sich die Aktivität A mit einem *cancel*-Ereignis, so muss wegen des *or*-Constraints zwischen A und C die Akti-

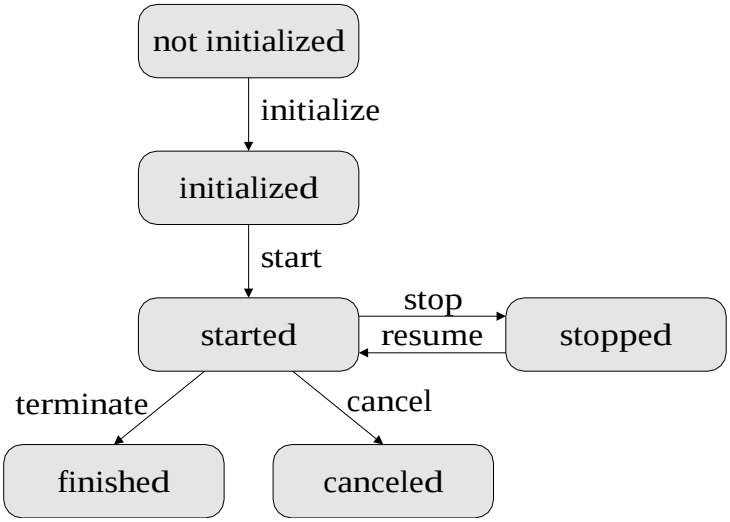

Abbildung 6.3: Zustände und Übergänge einer Aktivität

vität C terminieren. Es dürfen aber auch beide Aktivitäten terminieren (kein exklusives oder). Die Aktivität D darf aufgrund des *while*-Constraints nur ausgeführt werden, solange die Aktivität C läuft. Sie muss aber nicht ausgeführt werden. Soll die Aktivität D terminieren wenn auch die Aktivität C terminiert, dann muss zwischen den beiden Aktivitäten noch ein *implies*-Constraint eingefügt werden.

6.2.1 Zustände einer Designflow-Aktivität

Als Unterklasse der *BasicActivity* bieten die Designflow-Aktivitäten die gleiche Schnittstelle wie eine *BasicActivity*. Auch das grundsätzliche Verhalten unterscheidet sich nicht, so dass sich der Zustandsautomat in Abbildung 6.3 nicht von dem der Aktivität unterscheidet. Die einzelnen Zustände haben die folgende Bedeutung:

not initialized Die Aktivität wurde bisher nur erzeugt. Es können ihr auch schon Ressourcen und Sub-Aktivitäten hinzugefügt worden sein, die Initialisierung ist aber noch nicht abgeschlossen.

initialized Die Aktivität wurde initialisiert. Sie steht nun auf der Worklist der zugeordneten Akteure (Mitarbeiter) und kann von diesen prinzipiell gestartet werden. Durch Constraints kann dieser Start aber noch verhindert sein.

started Die Aktivität wurde gestartet und läuft gerade. Was nun darüberhinaus geschieht, hängt von der jeweiligen Aktivität ab.

stopped Die Aktivität wurde vorübergehend unterbrochen.

finished Die Aktivität wurde erfolgreich beendet. Bei einer Designflow-Aktivität müssen dazu alle Constraints erfüllt sein.

canceled Die Aktivität wurde abgebrochen. Um in diesen Zustand zu gelangen müssen die Constraints nicht erfüllt werden.

Zu Beginn meiner Arbeit habe ich einen größeren Zustandsautomaten betrachtet. Dort gab es zu den meisten Übergängen noch "erlaubt-Zustände", die angenommen werden, sobald der nachfolgende Übergang wirklich ausgeführt werden kann. So war z.B. zum *started*-Zustand noch ein *startable*-Zustand vorgesehen. Diese Zustände werden nun nicht mehr explizit modelliert, sondern das jeweilige Ereignis wird durch die EventEngine verhindert. Eine initialisierte Aktivität ist nun also grundsätzlich startbar, das eigentliche *start*-Ereignis kann aber noch von der EventEngine verhindert werden. Damit ist nur noch die EventEngine von diesen Zustandswechseln betroffen und sie muss diese nicht auch noch jedes Mal den Aktivitäten weitermelden.

6.2.2 Ereignisse

Der Wechsel einer Aktivität von einem Zustand in einen anderen wird durch ein Ereignis (event) signalisiert. Dieses Ereignis tritt ein, wenn alle für den Übergang nötigen Aufgaben erledigt sind. Jedes dieser Ereignisse hat mindestens zwei Parameter: die Aktivitäten-ID und ein Zähler der angibt, wie oft die Aktivität schon gestartet wurde (der Grund hierfür folgt beim *reset*-Ereignis). Die auch in Abbildung 6.3 gezeigten Ereignisse einer Aktivität werden in der folgenden Liste beschrieben:

init: Die Ressourcen und Akteure sind der Aktivität zugeordnet, sie müssen unter Umständen aber noch verfügbar werden.

start: Ein zugeordneter Aktor (z.B. ein Benutzer) startet die Aktivität. Wie bei den anderen Ereignissen müssen damit alle Constraints einverstanden sein. Die eigentliche Bearbeitung der Aufgabe, die mit dieser Aktivität verbunden ist, geschieht nun asynchron.

stop: Die Ausführung der Aktivität wird vorübergehend unterbrochen.

resume: Nach einem Stop weiter machen.

terminate: Damit wird die Designflow-Aktivität erfolgreich beendet. Dieses Ereignis kann nicht mehr zurückgenommen werden, da das aufgrund der fehlenden transaktionalen Behandlung der Informationsräume sowieso nichts ändern würde. Vorher müssen alle Sub-Aktivitäten beendet sein.

cancel: Die Aktivität wird abgebrochen.

start[id=$0.id]−start[id=$1.id]
∨ !start[id=$0.id]
∨ !start[id=$1.id]

start[id=$0.id, starts=$s]−start[id=$1.id] ∧ !reset[id=$0.id, starts=$s]
∨ start[id=$0.id, starts=$s]−reset[id=$0.id, starts=$s]−start[id=$1.id]
∨ !start[id=$0.id, starts=$s]
∨ !start[id=$1.id]

Abbildung 6.4: *starts_before*- Constraint ohne und mit Berücksichtigung eines Resets

reset: Mit diesem Ereignis soll die Aktivität in ihren Ausgangszustand zurückversetzt werden. Problematisch ist dabei, dass die Daten in CASSY in einem gemeinsamen Arbeitsbereich liegen und Änderungen somit kaum rückgängig gemacht werden können. Deshalb werden in CASSY die schon geänderten Daten in Kauf genommen.

Nach einem Reset einer Aktivität können unter Umständen bereits weitere Ereignisse ausgelöst worden sein, die nach einem Reset nicht hätten eintreten dürfen. Dadurch können also Inkonsistenzen entstehen. Ein Beispiel dafür ist ein *starts_before*-Constraint, bei dem nach dem Start der ersten Aktivität die zweite starten darf. Ist nun auch die zweite gestartet und die erste wird wieder per reset zurückgesetzt, dann hätte die zweite eigentlich noch nicht starten dürfen. Eine Möglichkeit dieses Problem zu behandeln, ist eine entsprechende Definition des Constraints, bei der nach dem Start der zweiten Aktivität die erste nicht mehr zurückgesetzt werden darf:

$$start[1] - reset[1] - start[2] \ \vee \ start[1] - start[2] \wedge !reset[1] \ \vee \ \ldots$$

Alternativ könnte der Constraint auch so definiert werden, dass nach dem Zurücksetzen der ersten Aktivität auch die zweite zurückgesetzt werden muss.

In beiden Fällen werden hier aber die oben erwähnten start-Zähler als Parameter der Ereignisse benötigt. Ansonsten würde die oben angedeutete Regel nach dem Eintreten eines *reset*-Ereignisses nach dem Start der Aktivität 1 und noch bevor Aktivität 2 gestartet wurde nur noch auf das *start[2]*-Ereignis warten und weitere Starts der ersten Aktivität gar nicht mehr berücksichtigen. Durch eine Variable wird bei der EventEngine nun für jeden Versuch eine neue Instanz der Regeln angelegt und es können

damit mehrere Versuche für die Aktivität unternommen werden. Abbildung 6.4 zeigt die einfache Definition von *starts_before* und die Variante, die nach dem Start der zweiten Aktivität das Zurücksetzen der ersten verbietet. Die erste Alternative des unteren Constraints ist der normale Ablauf ohne einen Reset. Die zweite Alternative wird verwendet, wenn die erste Aktivität noch rechtzeitig zurückgesetzt wird. Durch das hinten in der Sequenz stehende *start*-Ereignis der zweiten Aktivität kann diese noch nicht gestartet sein. Die restlichen beiden Alternativen treten ein, wenn eine der Aktivitäten nicht gestartet wurde.

6.2.3 Ablauf einer Aktivität

In diesem Abschnitt wird der gesamte Ablauf einer Aktivität, vom Erzeugen durch eine Factory über das Eintreten der verschiedenen Ereignisse bis zu ihrer Beendigung beschrieben.

Nachdem eine Aktivität von einer Factory erzeugt wurde befindet sie sich zunächst im Zustand *not initialized*. Schon in diesem Zustand muss ihr ein Akteur zugeordnet sein, da sonst die Server einiger Aktivitäten abstürzen. Dieser Akteur ist in der Regel nicht derjenige, der später die Aktivität ausführen wird, sondern der Auftraggeber, der am Ergebnis der Aktivität interessiert ist. Dieser Akteur muss die Aktivität dann initialisieren, d.h. ihr die Ressourcen zuordnen, und kann ihr auch weitere Sub-Aktivitäten anlegen. Wechselt die Aktivität dann in den Zustand *initialized* erscheint sie auf den Worklists der Bearbeiter.

Im Zustand *initialized* ist die Aktivität mit allen nötigen Ressourcen versorgt und befindet sich auf den Worklists der Bearbeiter. Die Aktivität muss dort aber nicht ständig startbar sein, da das zugehörige *start*-Ereignis durch Constraints blockiert sein kann. Die Aktivität wird dann nicht von der Worklist heruntergenommen. Damit hat der Bearbeiter einen besseren Überblick über die noch anstehenden Aktivitäten. Zudem können Constraints dann auch entfernt werden, falls sie für den Prozess nicht wichtig sind und der Bearbeiter unbedingt weiter machen will. Dies kann z.B. der Fall sein wenn eine Aktivität eigentlich auf die Daten einer vorangehenden Aktivität warten müsste (*feeds*-Constraint), der Bearbeiter aber im Informationsraum sieht, dass die Qualität der Daten schon für den nächsten Schritt ausreicht.

Wenn die Aktivität gestartet wurde und ihr *start*-Ereignis auch eingetreten ist, dann wird der Akteur der Aktivität auf den Bearbeiter gesetzt und die der Aktivität zugeordnete Anwendung wird schließlich ausgeführt.

Das Ende einer Aktivität kann auf zwei Weisen eintreten:

- Das *terminate*-Ereignis, mit dem das erfolgreiche Ende einer Aktivität signalisiert wird, soll wie jedes andere Ereignis unter der Kontrolle des Aktivitäten-Managers eintreten. Die Aktivität ist hier noch nicht beendet und sie wird auch nur dann beendet, wenn die Constraints dem Ereignis zustimmen.

- Die Aktivität wurde außerhalb von CASSY beendet. Dies kann z.B. bei Aktivitäten der Fall sein, die durch externe Programme bearbeitet werden. Wird das externe Programm beendet kann es in einigen Fällen sinnvoll sein, dass auch die dazugehörende Aktivität sofort beendet wird. Hier kann das Aktivitäten-Management nicht mehr kontrollierend eingreifen und das Ereignis muss so akzeptiert werden. Dadurch können dann aber Constraints unerfüllbar werden. Die Designflow-Aktivitäten, in denen diese Constraints definiert wurden, können dann dank der in Abschnitt 6.3.1 beschriebenen Modifikationen nicht mehr erfolgreich beendet werden ohne dass vorher diese Constraints entfernt werden oder das *terminate*-Ereignis erzwungen wird.

Obwohl die erste Variante die bevorzugte ist, werden vom Aktivitäten-Management und von der EventEngine beide unterstützt. Vor allem um auch Situationen zu unterstützen, in denen ein Ereignis nicht mehr ablehnbar ist.

Die wichtigsten Voraussetzungen für das Beenden einer Designflow-Aktivität sind:

1. Keine Sub-Aktivität ist aktiv. Zunächst sind die zu einer Designflow-Aktivität hinzugefügten Aktivitäten nur eine Menge von Aktivitäten, die der Designflow-Aktivität bekannt sind. Die geforderte Eigenschaft einer Sub-Aktivität kann aber durch den im Anhang definierten *while*-Constraint realisiert werden. Er stellt sicher, dass *start* und *stop* der Sub-Aktivität zwischen *start* und *stop* der Eltern-Aktivität eintreten.

2. Alle vitalen Aktivitäten sind beendet. Eine vitale Aktivität ist eine Aktivität, die für die Bearbeitung unbedingt notwendig ist. Dies kann durch eine Kombination der Constraints *terminates* und *while* erreicht werden. *terminates* stellt sicher, dass eine Aktivität erfolgreich beendet wird. Dazu muss diese auch starten und darf nicht durch ein *cancel* beendet werden. Ähnlich lassen sich mit dem schwächeren *starts*-Constraint Fälle behandeln, in denen eine Aktivität starten muss, diese Aktivität gegebenenfalls aber auch mit *cancel* abgebrochen werden kann.

3. Die Ziele der Aktivität sind erreicht. Hier gibt es zwei Möglichkeiten, wie das Erreichen des Ziels mit Constraints behandelt werden kann: Entweder wird ein Constraint eingefügt, der verhindert, dass die Aktivität terminiert. Ist das Ziel erreicht wird dieser Constraint einfach entfernt und die Aktivität kann damit terminieren. Oder das Erreichen des Ziels wird durch ein eigenes Ereignis angezeigt, das vor dem *terminate*-Ereignis eintreten muss.

   Die zweite Variante hat den Vorteil, dass das *terminate* solange als blockiert (blocked) und nicht als unmöglich (impossible) gilt.

6.2.4 Behandlung der Kooperationsbeziehungen

In diesem Abschnitt wird die Behandlung der Kooperationsbeziehungen Delegation, Negotiation und Usage durch die Aktivitäten betrachtet. Wie schon in Abschnitt 3.3 beschrieben sind die Protokolle, mit denen diese Beziehungen realisiert werden, nicht fest vorgegeben. Je nach Situation können auch unterschiedliche Protokolle innerhalb eines Systems für die Kooperationsbeziehungen verwendet werden. Die Protokolle können somit also nicht in den allgemeinen Klassen implementiert werden sondern müssen von Fall zu Fall entsprechend behandelt werden.

Ein weiterer Punkt speziell bzgl. der Delegation ist, wie die beiden Aktivitäten zueinander stehen. Eine Möglichkeit ist, dass die delegierte Aktivität ein Kind der delegierenden Aktivität wird. Die andere Möglichkeit ist, dass beide Aktivitäten, die delegierende und die delegierte, ein Kind der gleichen Aktivität sind. Im Prinzip ist es aber jeder Aktivität und jedem anderen Prozess im System überlassen, wo sie neu erzeugte Aktivitäten ablegen. Dabei sollte nur immer bedacht werden, zwischen welchen Aktivitäten Constraints sinnvoll sein können. Definiert man nur Constraints zwischen den Sub-Aktivitäten einer gemeinsamen Aktivität bleibt das System übersichtlicher.

6.3 Constraints

Mit Constraints kann der Benutzer Beziehungen zwischen unterschiedlichen Aktivitäten und zwischen Aktivitäten und Ressourcen definieren. Diese Constraints kontrollieren dann das Eintreten von signifikanten Ereignissen.

Dazu könnten direkt die in Kapitel 4 definierten Abhängigkeiten verwendet werden. Diese sind für den Anwender aber zu kompliziert und aufwändig. Daher wird hier mit den Constraints eine Schicht über die Abhängigkeiten gelegt, die auch in Abbildung 6.5 dargestellt ist. Der Benutzer kann damit einfach z.B. ein *after*-Constraint zwischen zwei Aktivitäten definieren und der Constraint trägt dann eine passende Abhängigkeit bei der EventEngine ein.

In Tabelle 6.1 werden die im Aktivitäten-Management vordefinierten Constraints aufgelistet und kurz beschrieben. Die genaue Definition dieser Constraints, die auch von der Implementierung verwendet wird, ist im Anhang zu finden.

Im Gegensatz zu anderen Arbeiten wie z.B. [Rit97] gibt es kein *directly-after*-Constraint sondern nur ein *after*. Ein *direkt* nach gibt es nicht, da es dafür im ASCEND-Modell keine vernünftige Semantik gibt. In Workflow-Systemen bezieht sich das direkt-nach auf einen Kontroll- oder Datenfluss, den es in ASCEND nicht gibt. Die Aktivitäten laufen einfach parallel ab und die Daten werden bei Bedarf aus dem Informationsraum geholt. Ein Fluss kann aber dennoch durch eine Folge von *after*-Constraints gebildet werden.

Eine andere Semantik für "direkt nach" ist, dass nach dem Ende der ersten Aktivität sofort der Start der zweiten Aktivität eintreten muss, ohne dass da-

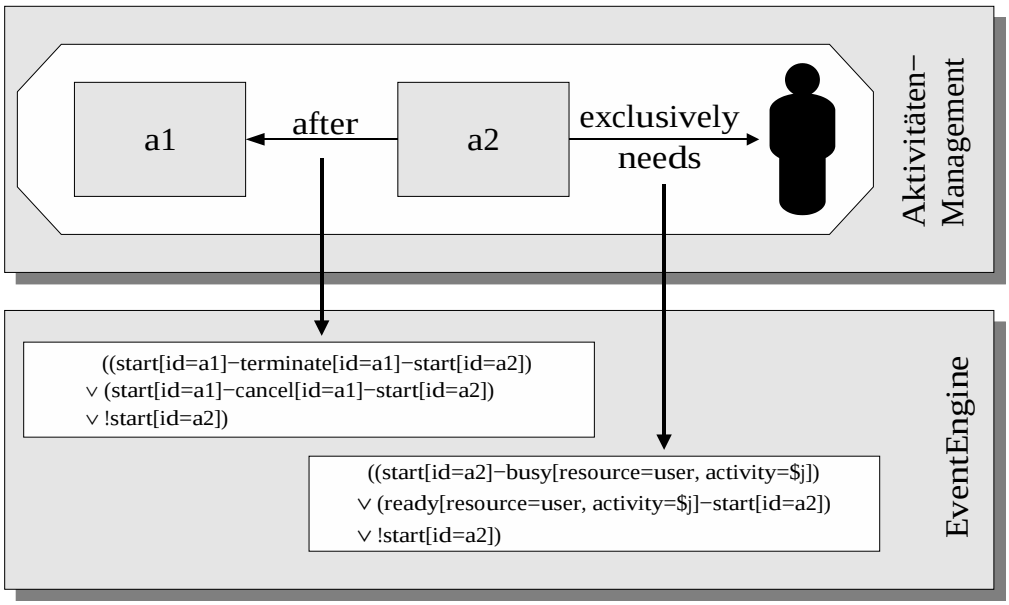

Abbildung 6.5: Abbildung der Constraints auf Abhängigkeiten

zwischen ein anderes Ereignis eintreten darf. Diese Definition ist aber für die meisten Fälle zu restriktiv. Sinnvoll erscheint sie mir nur bei Kompensations-Aktionen, die möglichst schnell nach dem Scheitern der eigentlichen Aktivität gestartet werden sollen. Solche Situationen können aber besser mit einer Workflow-Aktivität behandelt werden, so dass sie hier nicht weiter betrachtet werden.

Die Erzeugung der Abhängigkeit zu einem Constraint ist in Abbildung 6.6 dargestellt. Die Variablen in der Vorlage-Abhängigkeit müssen an die Parameter gebunden werden. Eine Variable setzt sich hier aus zwei Teilen zusammen: Der erste Teil vor dem Punkt gibt die Position des Parameters an (beginnend bei 0) und der zweite Teil ist das geforderte Attribut des Parameters. In der Abbildung wird z.B. die Variable "$1.id" des ersten *start*-Ereignisses an den eindeutigen Bezeichner (Identifier) der als zweiten Parameter von *after* angegebenen Aktivität gebunden.

Derzeit werden als Variablen die IOR von beliebigen CORBA-Objekten und der Bezeichner von Aktivitäten und Akteuren unterstützt. Die Menge der Variablen kann aber einfach erweitert werden da davon nur eine Methode betroffen ist. Z.B. ist für die Zeit-Ereignisse aus Abschnitt 6.6 eine Zeit-Angabe als Parameter sinnvoller als ein Objekt-Bezeichner.

6.3.1 Definition der Constraints

Die Constraints werden vom Administrator in einer XML-Datei ([W3C00, Kob99]) definiert die beim Starten des Aktivitäten-Managements geladen wird. Danach können die Constraints vorerst nicht geändert werden. Eine Versionie-

after	Die erste Aktivität darf erst starten, wenn die zweite fertig ist (terminate) oder abgebrochen (cancel) wurde.
disables	Sobald die erste Aktivität erfolgreich beendet wurde, darf die zweite Aktivität nicht mehr gestartet werden.
exclusively_needs	Dieser Constraint wird zwischen einer Activity und einer Ressource eingefügt und stellt sicher, dass die Ressource exklusiv für die Aktivität zur Verfügung steht.
feeds	Die erste Aktivität erzeugt Daten, die die zweite Aktivität benötigt. Daher darf die zweite Aktivität nur starten, wenn die erste erfolgreich beendet wurde (terminate)
forbids	Wenn die erste Aktivität erfolgreich beendet wurde, dann darf die zweite nicht auch terminieren.
implies	Wenn die erste Aktivität terminiert, dann muss auch die zweite Aktivität terminieren.
or	Eine der beiden Aktivitäten muss terminieren.
sequence	Wenn die erste Aktivität terminiert oder gecanceled wurde, darf und muss die zweite Aktivität starten.
sequence_terminate	Die erste Aktivität muss terminieren und danach muss die zweite Aktivität starten.
sequence_cancel	Wenn die erste Aktivität abgebrochen wurde muss die zweite (Kompensations-)Aktivität starten. Zwischen den Ereignissen kann aber eine lange Zeit liegen.
start_enables	Sobald die erste Aktivität gestartet wurde kann auch die zweite starten.
starts	Die Aktivität muss starten.
starts_before	Wenn beide Aktivitäten starten, dann muss die erste Aktivität vor der zweiten starten.
terminates	Die Aktivität muss erfolgreich beendet werden.
while	Die erste Aktivität darf nur während der zweiten laufen.
xor	Genau eine der beiden Aktivitäten muss terminieren.

Tabelle 6.1: Die vordefinierten Constraints und ihre Bedeutung

rung der Constraint-Definitionen ist aber schon vorgesehen. Sie wird dann notwendig, da schon eingefügte Constraints nicht durch eine nachträgliche Änderung der Definition geändert werden sollen.

Die Document Type Definition (DTD), in der die Struktur der XML-Datei spezifiziert wird, und eine ausführliche Beispiel-Definitions-Datei sind im Anhang enthalten. Abbildung 6.7 zeigt als Beispiel den Abschnitt aus der XML-Datei, in dem der *after*-Constraint definiert wird und die semantisch äquivalent Definition in der hier verwendeten, deutlich kompakteren Form. Der *after*-Constraint wurde hier so definiert, dass die erste Aktivität erst nach dem

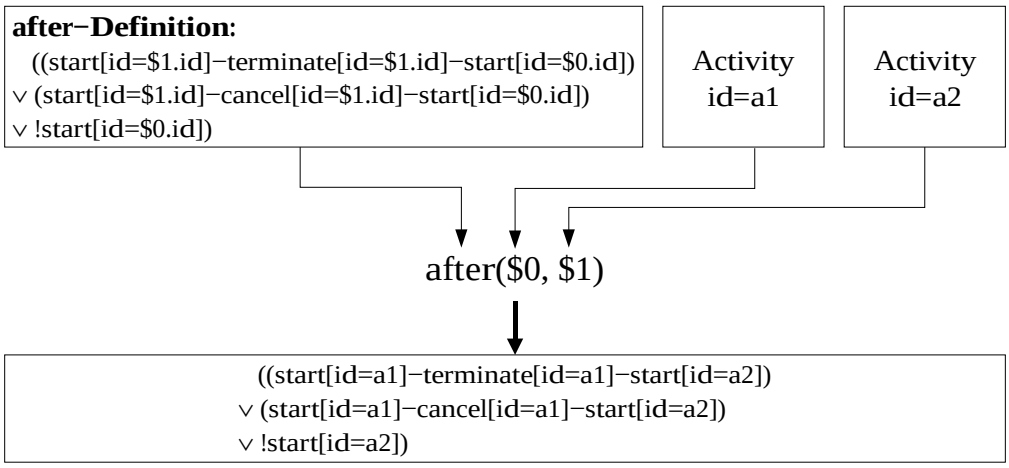

Abbildung 6.6: Setzen der Variablen der Abhängigkeit entsprechend der Parameter des Constraints

terminate- oder *cancel*-Ereignis der zweiten Aktivität startet darf. Sie muss aber nicht starten.

In der XML-Form werden die Operatoren nicht zwischen die einzelnen Operanden geschrieben, sondern es gibt jeweils einen Anfangs-Operator (z.B. "<OR>") und einen Ende-Operator ("</OR>"). Diese Operatoren werden in XML "Tags" genannt. Die einzelnen Operanden werden ohne besondere Trennung hintereinander geschrieben. Der OR-Ausdruck im Beispiel hat drei Operanden: Zwei Sequenzen und ein Komplement. Die Anfangs-Tags können wie bei den EVENT- und PARAMETER-Ausdrücken noch zusätzliche Parameter besitzen. Eigentlich muss es zu jedem Anfangs-Tag auch wieder ein schliessendes Tag geben. Bei den PARAMETER-Ausdrücken wird eine Kurzform verwendet (<PARAMETER ... />), bei der keine weiteren Teil-Ausdrücke angegeben werden können.

6.3.2 Einfügen von Constraints

Beim Einfügen eines neuen Constraints wird nun eine Abhängigkeit mit den entsprechend der Parameter gebundenen Variablen bei der EventEngine eingetragen. Geschieht das, bevor ein Ereignis eingetreten ist, ist das korrekt. Sind aber schon Ereignisse des Constraints eingetreten, gilt das nicht mehr. Wird z.B. ein *after*-Constraint eingefügt, wenn das *start*-Ereignis der ersten Aktivität schon eingetreten ist, dann wird in der Folge auch ungewollt das *terminate*-Ereignis der ersten Aktivität verhindert, da der Constraint noch auf das eigentlich schon eingetretene *start*-Ereignis wartet.

Zur Behandlung dieses Problems gibt es mehrere Ansätze:

- Die nicht korrekten Constraints zulassen. Wird ein Ereignis wegen eines nachträglich eingefügten Constraints blockiert, so muss der Constraint

```
<DEPENDENCY NAME="after" VERSION="0.1">
 <OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="terminate"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
 </OR>
</DEPENDENCY>
```

((start[id=$1.id]−terminate[id=$1.id]−start[id=$0])
∨ (start[id=$1.id]−cancel[id=$1.id]−start[id=$0.id])
∨ !start[id=$0.id])

Abbildung 6.7: Der *after*-Constraint in XML-Form (oben) und in der kompakten Darstellung (unten)

einfach wieder entfernt werden. Aufgrund von anderen Constraints verbotene Historien sind aber auch mit den fehlerhaften Constraints nicht möglich, so dass zumindest keine ungewollten Historien erlaubt werden.

- Nur Constraints einfügen wenn noch kein Ereignis eingetreten ist, wenn die Aktivität also noch im *not initialized*-Zustand ist. Dieser Ansatz ist nicht akzeptabel, da die lang laufenden Aktivitäten auf jeden Fall noch nachträglich modifizierbar sein müssen.

- Aus der Historie die eingetretenen Ereignisse nachspielen. Dieser Ansatz ist eigentlich der korrekte Ansatz. Hier wird jedoch eine vollständige Historie aller eingetretenen Ereignisse benötigt, in der auch schnell nach einzelnen Ereignissen gesucht werden kann. Günthör erwähnt in seiner Dissertation [Gün97] zwei Verfahren, mit denen eine Historie der noch relevanten Ereignisse implementiert werden könnte. Diese Historie kann dann schneller durchsucht werden als eine vollständige Historie.

Realisiert wurde vorerst der erste Ansatz, der auch nachträgliche Modifikationen erlaubt. Er kann zwar Ereignisse blockieren obwohl das aufgrund der Constraints und der Historie nicht sein müsste, er ist dafür aber auch weniger aufwändig und für einen Prototypen ausreichend.

6.3.3 Erfüllen von Constraints

Die Constraints sind bisher so definiert, dass ungewollte Historien von Ereignissen nicht entstehen können. Es ist aber nicht gesagt, dass ein Constraint auch irgendwann erfüllt sein muss. Soll ein Constraint erfüllt sein, bevor die Designflow-Aktivität in der sie enthalten ist beendet wird, gibt es zwei Möglichkeiten:

- Vor dem Beenden der Designflow-Aktivität wird der Zustand der Constraints abgefragt und es wird dem Bearbeiter gemeldet, wenn einige noch nicht erfüllt sind. Dieser kann dann die Beendigung der Aktivität trotzdem fortsetzen oder aber abbrechen, um die Constraints doch noch zu erfüllen.

- Die zu einem Constraint gehörende Abhängigkeit c wird beim Einfügen erweitert zu $c' = c - terminate[da]$. Damit kann die Designflow-Aktivität auch nicht beendet werden, bevor der Constraint c erfüllt ist. Durch diesen Ansatz wird kein weiterer Mechanismus zur Kontrolle der Constraints benötigt, sondern das kann mit dem schon vorhandenen Kontrollmechanismus erledigt werden.

Für die Implementierung wurde der zweite Ansatz gewählt. Dabei erweitern die Constraints selbst die Abhängigkeit aus der Definition, so dass dies nicht in der XML-Datei für jeden Constraint einzeln durchgeführt werden muss.

Diese Modifikation widerspricht allerdings der in Abschnitt 5.2 angebenen Einschränkung für die Abhängigkeiten. Ist in einer durch einen Constraint definierten Abhängigkeit zunächst nur ein Komplement $!e[]$ enthalten, das sich korrekter Weise nicht in einer Sequenz befindet, dann befindet es sich nach der Erweiterung und Umformung in TSF in einer Sequenz $!e[] - terminate[da]$. Es gibt aber Gründe, weshalb diese Sequenzen in diesem Fall erlaubt werden können:

- Die Komplemente sind auf jeden Fall nicht blockiert, da auch mit der Erweiterung keine Ereignisse vor einem Komplement stehen können. Wenn Komplemente nicht eintreten dürfen muss also das dazugehörende Ereignis erzwungen sein.

- Ein Komplement $!e$ könnte durch ein erzwungenes e in einem anderen Teil der Abhängigkeit erzwungen (required) sein. Aufgrund der Konstruktion der Erweiterung muss dann aber auch für das Ereignis e oder aber für ein späteres Ereignis (z.B. f bei der Sequenz *e-f*) eine Sequenz mit dem *terminate[da]*-Ereignis am Ende vorhanden sein, so dass die Designflow-Aktivität deshalb noch nicht terminieren darf.

- Ist das Ereignis zu einem Komplement aufgrund eines anderen Constraints erzwungen, dann wird durch das Terminieren der Designflow-Aktivität wie sonst auch der Constraint entfernt und das Ereignis bleibt erzwungen.

Durch die Erweiterung der durch die Constraints erzeugten Abhängigkeiten können also Abhängigkeiten entstehen, die eigentlich nicht erlaubt sind. Die obige Auflistung hat aber gezeigt, dass diese Abhängigkeiten in diesem Fall doch akzeptiert werden können. Auf diese Weise kann also einfach die Erfüllung der Constraints vor dem Terminieren der Designflow-Aktivitäten sichergestellt werden.

6.4 Behandlung von mehrfachen Aktivitäten

Bisher wurden hier nur Instanzen von Aktivitäten und Constraints zwischen diesen betrachtet. Für eine Aktivität muss dabei zunächst ein Objekt angelegt werden, bevor sie vom Aktivitäten-Management berücksichtigt werden kann. Dieses Anlegen der Aktivitäten und Constraints geschieht dabei ein Mal am Anfang der Laufzeit einer Designflow-Aktivität entsprechend einer Vorlage, die derzeit noch von Achim Eichhorn entworfen und implementiert wird.

Ein Nachteil dabei ist, dass unter Umständen beim Anlegen noch gar nicht alle Sub-Aktivitäten feststehen oder angelegt werden können. Ein Beispiel dafür sind Wiederholungen. Für Wiederholungen kann es unterschiedliche Gründe geben:

- Eine Folge von Aktivitäten muss so lange wiederholt werden, bis eine Bedingung erfüllt ist.
- Eine Aktivität wurde abgebrochen und soll noch einmal mit den gleichen Randbedingungen wiederholt werden.
- Eine Aktivität erzeugt mehrere Sub-Aktivitäten. Beispielsweise könnte eine Projekt-Planungs-Aktivität ein Projekt in mehrere gleichartige Teil-Projekte aufteilen und für diese Teil-Projekte dann jeweils eine Sub-Aktivität anlegen.

In allen drei Fällen steht beim Anlegen der umgebenden Aktivitäten noch nicht fest, wie viele Instanzen der Sub-Aktivitäten gebraucht werden, so dass diese nicht alle im voraus angelegt werden können. Da die Basic- und Designflow-Aktivitäten nachträglich modifiziert werden können, kann eine Aktivität aber jederzeit weitere Aktivitäten erzeugen und zu einer solchen Aktivität hinzufügen. Bei der *BasicActivity* ist das kein Problem, da sie die Sub-Aktivitäten nur als eine Menge von Aktivitäten betrachetet. Bei einer *DesignFlowActivity* dagegen können Constraints zwischen den Sub-Aktivitäten festgelegt werden. Diese Constraints sind bisher nur zwischen Instanzen von Aktivitäten definiert. Im folgenden soll gezeigt werden, wie auch Constraints zwischen noch nicht instantiierten Aktivitäten realisiert werden können.

Zur Behandlung dieses Problems können die Aktivitäten gruppiert werden. Diese Gruppe bekommt den Namen unter dem die Aktivitäten in der Vorlage referenziert werden. Gibt man nun beim Hinzufügen einer Aktivität zu einer

anderen Aktivität diesen Namen als Parameter an, kann die übergeordnete Aktivität die Constraints der Gruppe einfügen.

Wird nun eine neue Instanz einer Aktivität angelegt, die zu einer solchen Gruppe gehört, müssen die in der Vorlage für diese Gruppe definierten Constraints auch eingefügt werden. Die benötigten Informationen sind aber in der jetzigen Implementierung in einer aktiven Designflow-Aktivität nicht mehr vorhanden. Anstatt hier eine eigene Datenstruktur aufzubauen ist es sinnvoller, die von Eichhorn entwickelte Vorlage in ihrer XML-Form in der *DesignFlowActivity* zu speichern oder zumindest zur Laufzeit von dort aus zugreifbar zu machen. Beim Erzeugen einer Designflow-Aktivität muss dann nicht mehr die Designflow-Aktivität mit allen Sub-Aktivitäten und Constraints auf ein Mal angelegt werden, sondern es wird zunächst nur die Designflow-Aktivität angelegt. Die Sub-Aktivitäten können dann bei Bedarf zusammen mit ihren Constraints angelegt werden.

Der Implementierung dieses Ansatzes stehen nach der Fertigstellung des Tools von Eichhorn aber keine größeren Probleme im Wege. Der wesentliche Punkt ist dabei die Integration der beiden Sichten (Vorlage und erzeugte Aktivitäten) mit einer möglichst intuitiven Schnittstelle zum Benutzer.

6.5 Behandlung von Ressourcen

In diesem Abschnitt soll betrachtet werden, wie Ressourcen (oder in der CASSY-Notation die allgemeineren Akteure) vom Aktivitäten-Management behandelt werden können. Wesentlich ist dabei die Sicherstellung, dass eine benötigte Ressource auch verfügbar ist.

Wie sich die Aktivitäten über gemeinsam benötigte Ressourcen einigen, ist in ASCEND nicht fest vorgegeben. Dadurch kann eine möglichst gute Anpassung an die jeweilige Situation erreicht werden. Im einen Fall reicht ein einfaches Sperr-Verfahren, im anderen Fall können komplexe Interaktionsprotokolle zwischen den betroffenen Aktivitäten und Akteuren eingeleitet werden. Erol Bozak hat in seiner Studienarbeit [Boz99] schon die Spezifikation solcher Protokolle betrachtet und arbeitet in seiner Diplomarbeit daran weiter.

Die Protokolle können nicht in dieser Allgemeinheit vom Aktivitäten-Management berücksichtigt werden. Soll ein Protokoll bei den Ereignissen einer Aktivität berücksichtigt werden, so muss je nach Protokoll ein passender Constraint definiert werden. Dies soll im folgenden am Beispiel der Sperr-Verfahrens gezeigt werden, das auch schon in Abschnitt 5.4.1 als Beispiel zur Verwendung von Variablen verwendet wurde.

Mit dem Sperr-Verfahren soll hier sichergestellt werden, dass während der Ausführung einer Aktivität, d.h. zwischen *start*- und *terminate*-Ereignis, ein Ressource exklusiv für diese Aktivität zur Verfügung steht.

Damit das Verfahren von der EventEngine berücksichtigt werden kann, müssen die Ressourcen zwei Ereignisse erzeugen: $busy(resource = \$r, activity = \$j)$ und $ready(resource = \$r, activity = \$j)$. Die beiden Ereignisse treten ein

```
<DEPENDENCY NAME="exclusively_needs" VERSION="0.1">
 <OR>
   <SEQUENCE>
     <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
     <EVENT NAME="busy">
       <PARAMETER NAME="resource" VALUE="$1.id"/>
       <PARAMETER NAME="activity" VALUE="$j"/>
     </EVENT>
   </SEQUENCE>
   <SEQUENCE>
     <EVENT NAME="ready">
       <PARAMETER NAME="resource" VALUE="$1.id"/>
       <PARAMETER NAME="activity" VALUE="$j"/>
     </EVENT>
     <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
   </SEQUENCE>
   <COMPLEMENT>
     <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
   </COMPLEMENT>
 </OR>
</DEPENDENCY>
```

```
((start[id=$0.id]−busy[resource=$1.id, activity=$j])
∨ (ready[resource=$1.id, activity=$j]−start[id=$0.id])
∨ !start[id=$0.id])
```

Abbildung 6.8: Der exclusively_needs-Constraint

wenn die Ressource $\$r$ einer Aktivität $\$j$ zugeordnet wurde bzw. wenn sie für die Aktivität nicht mehr benötigt wird.

Damit kann dann ein *exclusively_needs*-Constraint wie in Abbildung 6.8 definiert werden. Die letzte Alternative in der Definition ist die einfachste. Sie erlaubt die Möglichkeit, dass die Aktivität gar nicht startet. In den oberen beiden Alternativen kommt eine Variable $\$j$ vor, die bei der Erzeugung des Constraints nicht gebunden wird. Tritt hier nun ein *busy*-Ereignis für eine andere Aktivität ein, so wird eine neue Instanz der Regel angelegt, in der die Variable $\$j$ durch den eindeutigen Bezeichner der Aktivität ersetzt wurde. In dieser ist die erste Alternative nicht mehr möglich, da das *start*-Ereignis vorher noch nicht eingetreten ist. Ein Start ist nun nicht möglich, da für die zweite Alternative nun zwingend zunächst das passende *ready*-Ereignis eintreten muss. Ist dieses eingetreten, kann die Aktivität starten, solange nicht ein weiteres *busy*-Ereignis eintritt.

6.6 Realisierung von Zeit-Constraints

Neben Beziehungen zwischen den signifikanten Ereignissen von Aktivitäten sind diese Ereignisse oft auch durch zeitliche Restriktionen eingeschränkt. Diese treten in unterschiedlichen Formen auf:

1. Ein Ereignis muss vor bzw. nach einem bestimmten Zeitpunkt eingetreten sein. Z.B. sollten alle Y2K-Umstellungen vor dem 31.12.2000, 24:00 Uhr beendet worden sein.

2. Zwischen zwei Ereignissen darf nur eine beschränkte Zeit vergehen. Z.B. darf ein Arbeiter nur maximal 10 Stunden am Tag arbeiten.

3. Ein Ereignis kann nur in einem sich täglich wiederholenden Zeitbereich eintreten. Z.B. kann die Bibliothek der Fakultät nur zwischen 10:00 und 18:00 Uhr besucht werden.

Im folgenden wird gezeigt, wie auch Zeit-Restriktionen mit Constraints behandelt werden können. Dazu werden zunächst Zeit-Ereignisse benötigt. Ein solches Ereignis hat die Form $time[datum = \$d, time = \$t]$. Diese Ereignisse werden von einem Zeitgeber (timer) erzeugt wenn sie von einem Constraint gefordert werden. Sie können wie jedes andere Ereignis auch in Constraints verwendet werden.

Die Behandlung der Restriktionen mit festen Zeitpunkten ist relativ einfach. Hier muss die EventEngine beim Einfügen eines Constraints mit einem Zeit-Ereignis einen Timer initialisieren, der dann zur gegebenen Zeit das Ereignis erzeugt. Eine Abhängigkeit

$$terminate[id = tuwas] - time[date ='' 31.12.2000'', time ='' 24:00'']$$

verlangt z.B., dass die Aktivität *tuwas* vor dem problematischen Zeitpunkt für die YSK-Probleme beendet wurde.

Die Restriktionen zu sich täglich wiederholenden Zeitbereichen können wie der *exclusively_needs*-Constraint im vorigen Abschnitt behandelt werden, wobei hier die Variable nicht der Aktivitäten-Bezeichner sondern das Datum ist.

Problematisch sind Restriktionen, die die vergangene Zeit zwischen zwei Ereignissen betreffen. Hier kann die Zeitschranke für das spätere Ereignis erst dann bestimmt werden, wenn das erste Ereignis eingetreten ist. Hier könnte z.B. zunächst eine Restriktion mit einem festen Zeitpunkt einfügt werden, der so spät liegt, dass das spätere Ereignis so nie eintreten dürfte. Tritt nun das erste Ereignis ein, kann das Zeit-Ereignis an die wirkliche Schranke angepasst werden. Hier zeigt sich aber, dass die die Beschränkung der Constraints rein auf Beziehungen zwischen Ereignissen nicht immer einfach zu realisieren ist. In der Implementierung sind die Zeit-Constraints daher auch noch nicht realisiert.

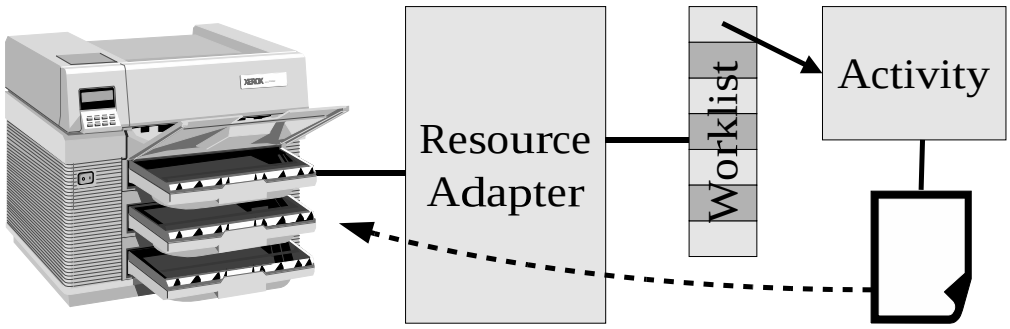

Abbildung 6.9: Ein Drucker-Adapter beim Drucken eines Dokuments

6.7 Selbst-startende Aktivitäten

Als vorletzter Punkt in diesem Kapitel soll nun betrachtet werden, wie selbst- oder automatisch-startende Aktivitäten realisiert werden können. Solche Aktivitäten werden in CASSY nicht durch eine spezielle Kennzeichnung markiert, sondern sie haben als Akteur einen *BasicResourceAdapter* gesetzt. Dahinter verbirgt sich eine Maschine bzw. ein Programm, das die Aktivität selbstständig erledigen kann.

Als Beispiel für eine solchen Ablauf ist in Abbildung 6.9 ein Resource-Adapter dargestellt, der einen Drucker in CASSY repräsentiert. Er nimmt eine (Druck-)Aktivität von seiner Worklist und startet sie. Ist die Aktivität gestartet frägt der Adapter diese nach dem zu druckenden Dokument und schickt dieses dann an den Drucker.

Dieser Adapter kann regelmäßig versuchen die Aktivitäten auf seiner Worklist zu initialisieren und zu starten. Wird eines dieser Ereignisse in einem Durchlauf verhindert, dann wird es einfach im nächsten Durchlauf erneut probiert. Dieses ständige Probieren könnte dadurch vermieden werden, dass sich der Akteur bei der EventEngine für das “Ereignis” *init* bzw. *start* wird möglich (possible) registriert. Die EventEngine informiert dann über eine Callback-Funktion den Akteur. Bisher werden in der Implementierung nur die “Ereignisse” Ereignis wurde angefordert (request) und Ereignis ist eingetreten (signal) erzeugt.

6.8 Die Benutzungsschnittstelle

Neben dem Constraints-basierten Aktivitäten-Manager und der EventEngine wurde in dieser Arbeit auch eine Benutzungsoberfläche implementiert, mit dem Aktivitäten und Constraints erzeugt und Ereignisse ausgelöst werden können. In Abbildung 6.10 ist das Hauptfenster dieser grafischen Oberfläche zu sehen. Im linken Teil werden die Aktivitäten mit ihren Sub-Aktivitäten hierarchisch dargestellt. Zur ausgewählten Aktivität wird im rechten Teil des Fensters der eindeutige Bezeichner (ID), der Name, die IOR und der zugeordnete Aktor an-

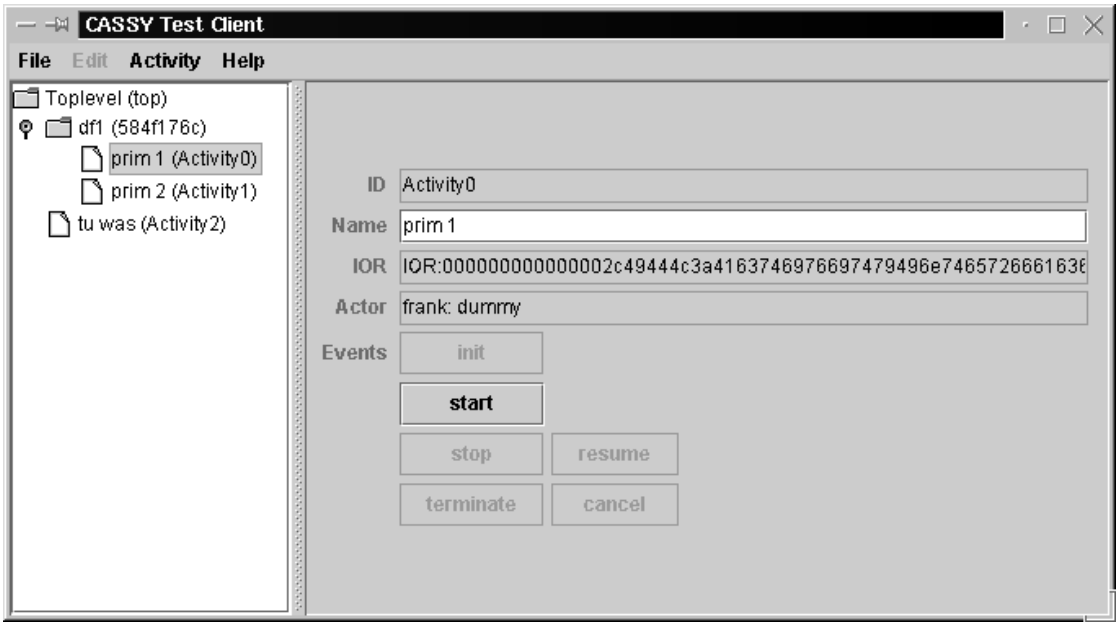

Abbildung 6.10: Das Hauptfenster der Benutzungsschnittstelle

gezeigt. Darunter sind die signifikanten Ereignisse der Aktivitäten als Schaltflächen dargestellt. Die fett dargestellten Schaltflächen können aufgrund des Zustandes der Aktivität geschaltet werden. Allerdings kann das dadurch ausgelöste Ereignis durch Constraints verhindert sein, so dass sich der Zustand nicht ändert.

Unter dem Menüpunkt "Activity" können neue Sub-Aktivitäten angelegt werden. Dazu muss zunächst im linken Teil des Fensters eine Aktivität ausgewählt werden, zu der Sub-Aktivitäten hinzugefügt werden können. Dies sind alle Instanzen der *BasicActivity* oder einer Sub-Klasse dieser. Wählt man den Menüpunkt aus öffnet sich das in Abbildung 6.11 dargestellte Fenster. Hier muss der Name der Factory zum Erzeugen der Aktivität und der Name der

Abbildung 6.11: Dialog zum Erzeugen neuer Aktivitäten

Aktivität selbst eingegeben werden.

Ein weiterer Menüpunkt im Menü "Activity" öffnet einen Dialog in dem neue Constraints eingefügt und schon vorhandene wieder entfernt werden können.

In der Benutzungsschnittstelle sind bisher nur die vom Constraints-basierten Aktivitäten-Manager benötigten und unterstützten Funktionen implementiert. Im wesentlichen fehlt noch ein Zugriff auf die Informationsräume, die auf die Constraints keinen Einfluss haben, und die Behandlung der Ressourcen einer Aktivität.

Kapitel 7

Zusammenfassung

In dieser Arbeit wurde ein Aktivitäten-Manager entworfen und implementiert, mit dem eine Constraints-basierte Koordinierung von Aktivitäten möglich ist. Als Basis für die Implementierung wurde ein formales Modell zur Spezifikation von erlaubten Ereignisfolgen verwendet, das in einer leicht abgeänderten und korrigierten Version implementiert wurde.

Wesentlicher Vorteil dieser Realisierung ist eine einheitliche und relativ einfach erweiterbare Spezifikation der Constraints. Durch den Verzicht auf ein anderes Mittel um Bedingungen für ein Ereignis zu spezifizieren müssen aber auch wirklich alle Bedingungen durch Ereignisse angezeigt werden. Dies ist bei den betrachteten Fällen zwar möglich, wird aber wie bei der Behandlung der Verfügbarkeit von Ressourcen- und von Zeit-Restriktionen gezeigt, schon aufwändiger. Bei solchen Restriktionen wäre es einfacher vor dem Eintreten eines Ereignis direkt den Zustand einer Ressource oder die Zeit abzufragen und mit dem Soll zu vergleichen.

Eine sinnvolle Erweiterung ist die Integration der noch nicht fertiggestellten Implementierung von Vorlagen in das System um damit auch eine schrittweise Instantiierung von Aktivität bei Bedarf zu ermöglichen. Dazu sollte dann aber auch eine Historie implementiert werden, anhand derer beim Einfügen von Constraints die schon geschehenen Ereignisse berücksichtigt werden können.

Um das System auch etwas praktischer testen zu können müssten die meisten Klassen die Aktivitäten oder Akteure implementieren überarbeitet werden. Während meiner Arbeit musste ich leider nach und nach durch Testen und Code-Betrachtung feststellen, dass im bisherigen System kaum eine Funktionalität ist, die auf ein Aktivitäten-Management hindeutet. Über die Speicherung von Attributen (z.B. Name und Bezeichner) und eine nicht besonders robuste Verwaltung der Beziehungen zwischen den Objekten geht die Implementierung kaum hinaus.

Anhang A

IDL-Dateien

A.1 EventEngine.idl

```
// Diese IDL−Datei legt die CORBA−Schnittstelle
// der Event−Engine fest.

module EventEngine {
    interface Engine;
    interface Listener;


    struct NameValue {
        string name;
        string value;
    };

    typedef sequence<NameValue> NameValueSequence;


    // Event
    // =====

    enum EventState { UNKNOWN, POSSIBLE, BLOCKED, IMPOSSIBLE,
                      REQUIRED, REQUESTED, HAPPENED };
    typedef sequence<EventState> EventStateList;

    /**
     * Steht dieses Zeichen am Anfang einer value eines
     * Parameters eines Events, dann wird diese value als
     * eine Variable interpretiert .
     **/
    const char VARIABLE_PREFIX = '$';


    /**
     * Ein Ereignis hat einen Namen und eine Folge
     * von Parametern
     **/
```

```
struct Event {
    string name;
    NameValueSequence parameters;
};

typedef sequence<Event> EventList;


// Dependencies
// ============

enum DependencyOperator { AND_DEP, OR_DEP, SEQ_DEP,
                          EVENT, COMPLEMENT,
                          CONSTANT
};

/**
 * Eine DependencyDefintion ist ein Ausdruck der die
 * gültigen Folgen von Events einschränkt.
 **/
union DependencyDefinition switch (DependencyOperator) {
case AND_DEP:
case OR_DEP:
case SEQ_DEP: sequence<DependencyDefinition> operands;
case EVENT:
case COMPLEMENT: Event event;
case CONSTANT: boolean value;
};



/**
 * Eine Dependency ist eine bei der EventEngine
 * registrierte DependencyDefinition. Sie wird bei
 * eintreffenden Events (query, request, signal)
 * berücksichtigt.
 **/
typedef long long Dependency;
typedef sequence<Dependency> DependencyList;

// Engine
// ======

exception DependencyViolation {
    DependencyList dependencies;
};


/**
 * Die Schnittstelle der EventEngine. Sie legt fest,
 * wie Ereignissen ausgelöst und Abhängigkeiten
 * eingefügt und wieder entfernt werden.
 **/
interface Engine {
```

```
/**
 * Erfrägt den Zustand (EventState) des Ereignisses.
 **/
EventState queryEvent(in Event e);
EventState queryEvent2
  (in Event e,
   out Dependency blockingDependency);

/**
 * Fordert ein Ereignis an und blockiert es
 * damit für andere.
 **/
boolean requestEvent(in Event e);
boolean requestEvent2
  (in Event e,
   out Dependency blockingDependency);
boolean cancelRequestEvent(in Event e);

/**
 * Das Ereignis gilt mit dem Aufruf dieser Methode
 * als eingetreten, auch wenn es nicht hätte
 * eintreten sollen. Die Exception wird erst nach dem
 *  Einarbeiten des Events geschmissen.
 **/
void signalEvent(in Event e)
  raises (DependencyViolation);

/**
 * Hier können sich Listener registrieren lassen,
 * die dann nach dem Eintreten des Ereignisses jeweils
 * informiert werden (notification). Die Listener
 * werden nicht persistent gespeichert.
**/
void addEventListener
  (in Event pattern,
   in EventStateList states,
   in Listener listener);
void removeEventListener(in Event e, in Listener l);


/**
 * Die Methode nimmt die DependencyDefinition in
 * die Menge der zu beachtenden Regeln auf.
 **/
Dependency createDependency
  (in DependencyDefinition definition);
boolean removeDependency(in Dependency dependency);

/**
 * Gibt eine Liste aller aktuell registrierten
 * Dependencies zurück.
 **/
DependencyList dependencies();
```

```
        /**
         * Hiemit kann die Definition einer registrierten
         * Dependency abgefragt werden.
         **/
        DependencyDefinition definition
          (in Dependency dependency);

        /**
         * Die Methode liefert den Rest einer registrierten
         * Dependency zurück. Dieser Rest ist aus der
         * Definition der Dependency im Laufe der eingetroffenen
         * Events entstanden und gibt den noch zu betrachtenden
         * Teil an.
         **/
        DependencyDefinition residual(in Dependency dependency);

        /**
         * Gibt true zurück wenn die Dependency erfüllt ist.
         **/
        boolean isSatisfied (in Dependency dependency);

        /**
         * Gibt true zurück wenn die Dependency nicht
         * mehr erfüllbar ist .
         **/
        boolean isUnsatisfiable(in Dependency dependency );
    };



    /**
     * Ist ein Listener bei der EventEngine regisitriert ,
     * wird er über das Eintreten von Ereignissen
     * informiert (callback).
     **/
    interface Listener {
        void event(in Event event, in EventState state );
    };
};
```

A.2 ActivityManagement.idl

```
#include "ActorInterface.idl"
#include "ActivityInterface.idl"
#include "EventEngine.idl"

module ActivityManagement {
    interface DesignFlowActivity;
    interface DesignItem;
    interface Constraint;

    typedef sequence<string> StringSeq;
```

```
/**
 * Der Manager ist die zentrale Komponente, die zum Ändern
 * des Zustandes verwendet werden muss. Er prüft das
 * jeweilige Ereignis zunächst bei der EventEngine
 * (requestEvent), ruft dann die entsprechende Mehtode der
 * Aktivität auf und meldet der EventEngine am Ende, dass
 * das Ereignis eingetreten ist.
 **/
interface Manager {
    boolean init(in ActivityInterface::Activity activity);
    boolean start(in ActivityInterface::Activity activity);
    boolean stop(in ActivityInterface::Activity activity);
    boolean terminate(in ActivityInterface::Activity activity);
    boolean reset(in ActivityInterface::Activity activity);
    boolean cancel(in ActivityInterface::Activity activity);
    boolean resume(in ActivityInterface::Activity activity);
};


typedef sequence<Object> ObjectSeq;
typedef sequence<Constraint> ConstraintList;

/**
 * Eine DesignFlowActivity ist eine BasicActivity, d.h. eine
 * Activity die weitere Sub-Aktivitäten haben kann. Zwischen
 * diesen Sub-Aktiitäten können Constraints angegeben werden.
 **/
interface DesignFlowActivity
    : ActivityInterface::BasicActivity
{
    ConstraintList constraints();
    Constraint createConstraint(in string name,
                                in ObjectSeq designItems);
    void removeConstraint(in Constraint c);
};


/**
 * Ein Constraint ist ein Beziehung zwischen Objekten
 * (Sub-Aktivitäten, Ressourcen, Zeiten, ...) mit der die
 * Folge der Ereignisse eingeschränkt wird. Die
 * DesignFlowActivity ist die Factory für Constraints.
 **/
interface Constraint {
    readonly attribute string name;
    readonly attribute ObjectSeq parameters;
};


/**
 * Mit dieser Factory können neue Designflow-Aktivitäten
 * erstellt werden.
 **/
```

```
    interface DesignFlowActivityObjectFactory
        : ActivityInterface :: ActivityObjectFactory
    {
        DesignFlowActivity createDesignFlowActivity
            (in string name,
             in string id,
             in ActorInterface :: Actor actor,
             in ActivityInterface :: BasicActivity parent,
             in ActorInterface :: BasicResourceAdapterList
                resources);

        DesignFlowActivity createEmptyDesignFlowActivity
          (in string id);

        StringSeq getConstraintNames();
    };
};
```

Anhang B

DTD-Dateien

B.1 EventEngine.dtd

```
<?xml encoding="ISO−8859−1"?>

<!ELEMENT PARAMETER EMPTY>
<!ATTLIST PARAMETER
  NAME NMTOKEN #REQUIRED
  VALUE CDATA #REQUIRED
>

<!ELEMENT EVENT (PARAMETER)∗>
<!ATTLIST EVENT
  NAME NMTOKEN #REQUIRED
>

<!ENTITY % DEPENDENCYDEFINITION
   "(EVENT | COMPLEMENT | CONSTANT | SEQUENCE | AND | OR)">

<!ELEMENT DEPENDENCY ((%DEPENDENCYDEFINITION;)∗)>
<!ATTLIST DEPENDENCY
  NAME NMTOKEN #IMPLIED
  VERSION CDATA #IMPLIED
>


<!ELEMENT COMPLEMENT (EVENT)>
<!ELEMENT CONSTANT EMPTY>
<!ATTLIST CONSTANT VALUE CDATA #REQUIRED>
<!ELEMENT SEQUENCE (%DEPENDENCYDEFINITION;)∗>
<!ELEMENT AND (%DEPENDENCYDEFINITION;)∗ >
<!ELEMENT OR (%DEPENDENCYDEFINITION;)∗ >


<!ELEMENT DEPENDENCY_STATE
  (DEPENDENCY, DEPENDENCY, INSTANCE∗)>

<!ELEMENT INSTANCE (PARAMETER∗, DEPENDENCYSTATE)>
```

B.2 ActivityManagement.dtd

```
<?xml encoding="ISO−8859−1"?>

<!−− Die dtd der EventEngine einbinden −−>
<!ENTITY % include SYSTEM "EventEngine.dtd">
%include;


<!−− Die Datei mit den Definitionen der Constraints enthält das
     folgende Element, das wiederum beliebig viele Dependencies
     enthalten kann. −−>

<!ELEMENT CONSTRAINT_DEFINITIONS (DEPENDENCY)*>
```

Anhang C

Beispiel-Constraints

C.1 constraints.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE CONSTRAINT_DEFINITIONS SYSTEM "ActivityManagement.dtd">

<CONSTRAINT_DEFINITIONS>

<!-- Die erste Aktivität darf erst starten, wenn die zweite
     fertig (terminate) oder abgebrochen (cancel) wurde. -->
<DEPENDENCY NAME="after" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Sobald die erste Aktivität erfolgreich beendet wurde, darf die
     zweite Aktivität nicht mehr gestartet werden. -->
<DEPENDENCY NAME="disables" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
```

```
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Dieser Constraint wird zwischen einer Activity und einem Actor
      eingefügt und stellt sicher, dass die Ressource exklusiv für die
      Aktivität zur Verfügung steht -->
<DEPENDENCY NAME="exclusively_needs" VERSION="0.1">
  <OR>
    <SEQUENCE>
        <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
        <EVENT NAME="busy">
            <PARAMETER NAME="resource" VALUE="$1.id"/>
            <PARAMETER NAME="activity" VALUE="$j"/>
        </EVENT>
    </SEQUENCE>
    <SEQUENCE>
        <EVENT NAME="ready">
            <PARAMETER NAME="resource" VALUE="$1.id"/>
            <PARAMETER NAME="activity" VALUE="$j"/>
        </EVENT>
        <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    </SEQUENCE>
    <COMPLEMENT>
        <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    </COMPLEMENT>
  </OR>
</DEPENDENCY>

<!-- Die erste Aktivität erzeugt Daten, die die zweite Aktivität
      benötigt. Daher darf die zweite Aktivität nur starten, wenn die
      erste erfolgreich beendet wurde (terminate) -->
<DEPENDENCY NAME="feeds" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </COMPLEMENT>
```

```
</OR>
</DEPENDENCY>

<!-- Wenn die erste Aktivität erfolgreich beendet wurde, dann darf
      die zweite nicht auch terminieren. -->
<DEPENDENCY NAME="forbids" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Wenn die erste Aktivität terminiert, dann muss auch die
      zweite Aktivität terminieren. -->
<DEPENDENCY NAME="implies" VERSION="0.1">
<OR>
  <EVENT NAME="terminate"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  <COMPLEMENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Eine der beiden Aktivitäten mussterminieren. -->
<DEPENDENCY NAME="or" VERSION="0.1">
<OR>
  <EVENT NAME="terminate"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  <EVENT NAME="terminate"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
</OR>
</DEPENDENCY>

<!-- Wenn die erste Aktivität terminiert oder gecanceled wurde darf
      und muss die zweite Aktivität starten. -->
<DEPENDENCY NAME="sequence" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
```

```
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
</OR>
</DEPENDENCY>

<!-- Die erste Aktivität muss terminieren und danach muss die
      zweite Aktivität starten. -->
<DEPENDENCY NAME="sequence_terminate" VERSION="0.1">
<SEQUENCE>
  <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  <EVENT NAME="terminate"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
</SEQUENCE>
</DEPENDENCY>

<!-- Wenn die erste Aktivität abgebrochen wurde muss die zweite
      (Kompensations-)Aktivität starten. Zwischen den Ereignissen kann
      aber eine lange Zeit liegen. -->
<DEPENDENCY NAME="sequence_cancel" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="cancel"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>


<!-- Sobald die erste Aktivität gestartet wurde kann auch die
      zweite starten -->
<DEPENDENCY NAME="start_enables" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>


<!-- Die Aktivität muss starten. -->
<DEPENDENCY NAME="starts" VERSION="0.1">
<EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
```

```
</DEPENDENCY>

<!-- Wenn beide Aktivitäten starten, dann muss die erste Aktivität
     vor der zweiten starten. -->
<DEPENDENCY NAME="starts_before" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Die Aktivität muss erfolgreich beendet werden. -->
<DEPENDENCY NAME="terminates" VERSION="0.1">
<EVENT NAME="terminate"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
</DEPENDENCY>

<!-- Die erste Aktivität darf nur während der zweiten laufen. -->
<DEPENDENCY NAME="while" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Genau eine der beiden Aktivitäten muss terminieren. -->
<DEPENDENCY NAME="xor" VERSION="0.1">
<AND>
  <OR>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </OR>
  <OR>
    <COMPLEMENT>
      <EVENT NAME="terminate">
          <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
```

```
    </COMPLEMENT>
    <COMPLEMENT>
      <EVENT NAME="terminate">
          <PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    </COMPLEMENT>
  </OR>
</AND>
</DEPENDENCY>
</CONSTRAINT_DEFINITIONS>
```

Literaturverzeichnis

[ASE+96] ATTIE, PAUL C., MUNINDAR P. SINGH, E. ALLEN EMERSON, AMIT P. SHETH und MAREK RUSINKIEWICZ: *Scheduling Workflows by Enforcing Intertask Dependencies*. Distributed Systems Engineering Journal, 3(4):222–238, Dezember 1996.

[ASSR93] ATTIE, PAUL C., MUNINDAR P. SINGH, AMIT SHETH und MAREK RUSINKIEWICZ: *Specifying and enforcing intertask dependencies*. In: *Proc. 19th VLDB Conference*, Seiten 134–145, 1993.

[Bai98] BAITINGER, UTZ: *Entwurfsautomatisierung*, 1998. Unterlagen zur Vorlesung im WS 1998/99 an der Fakultät Informatik der Universität Stuttgart.

[Boz99] BOZAK, EROL: *Integration von autonomen Agenten in das Designflow-System CASSY*. Studienarbeit Nr. 1769, Fakultät Informatik, Universität Stuttgart, Dezember 1999.

[Bur97] BURGER, CORA: *Groupware – Kooperationsunterstützung für Verteilte Anwendungen*. dpunkt Verlag, 1997.

[Chr91] CHRYSANTHIS, PANAYIOTIS KYPROS: *ACTA, A Framework for Modelling and Reasoning about Extended Transactions*. Doktorarbeit, Department of Computer and Information Science, University of Massachusetts, 1991.

[CR92] CHRYSANTHIS, PANOS K. und KRITHI RAMAMRITHAM: *ACTA: The SAGA Continues*. In: ELMAGARMID, A. K. (Herausgeber): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, 1992.

[Fra00] FRANK, AIKO, 2000. 22 unveröffentlichte Seiten zu ASCEND. Stand: 31.10.2000.

[GD93] GEPPERT, ANDREAS und KLAUS R. DITTRICH: *Rule-Based Implementation of Transaction Model Specifications*. In: PATTON, N.W. und H.W. WILLIAMS (Herausgeber): *Proc. 1st Intl. Workshop on Rules in Database Systems, Edinburgh, UK*, August 1993.

[GHJV96] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Bonn, 1996.

[Gün97] GÜNTHÖR, ROGER: *Ein Basisdienst für die zuverlässige Abwicklung langdauernder Aktivitäten*. Dissertation, Fakultät Informatik der Universität Stuttgart, 1997.

[GT98] GEPPERT, ANDREAS und DIMITRIOS TOMBROS: *Event-based Distributed Workflow Execution with EVE*. In: DAVIS, N., K. RAYMOND und J. SEITZ (Herausgeber): *Middleware '98, The Lake District, England*, Seiten 427–442, September 1998.

[Hei98] HEISS, GÜNTER: *Evaluierung der Vorschläge für eine CORBA Workflow Facility und die Implementierung eines solchen Dienstes*. Diplomarbeit, Technische Universität München, Institut für Informatik, 1998.

[Hen99] HENNING, MICHI: *Advanced CORBA Programming with C++*. Addison Wesley Longman, Inc., 1999.

[Jab95] JABLONSKI, STEFAN: *Workflow-Management-Systeme: Motivation, Modellierung, Architektur*. Informatik Spektrum, 18:13–24, 1995.

[JBS97] JABLONSKI, STEFAN, MARKUS BÖHM und WOLFGANG SCHULZE (Herausgeber): *Workflow-Management: Entwicklung von Anwendungen und Systemen; Facetten einer neuen Technologie*. dpunkt-Verlag, Heidelberg, Germany, 1997.

[Kle91] KLEIN, JOHANNES: *Advanced Rule Driven Transaction Management*. In: *36th IEEE Computer Society Int'l Conference (CompCon) Spring 1991*, Seiten 562–567, 1991.

[Kob99] KOBERT, THOMAS: *XML – Das bhv Taschenbuch*. bhv, 1999.

[Kuh00] KUHN, ANDREAS: *Integration der OMG Workflow Management Facility in das CORBA-basierte Objektmodell von ASCEND*. Diplomarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Abteilung Anwendersoftware, 2000.

[Mar98] MARIUCCI, MARCELLO: *Untersuchung kooperativer Abläufe anhand von Beispielen und Entwicklung eines kooperativen Entwurfsvorgangmodells für die Integration von Workflow- und CSCW-Aspekten*. Diplomarbeit, Technische Universität München, Institut für Informatik, 1998.

[Mos82] MOSS, E.J.B.: *Nested transactions and reliable distributed computing*. In: *Proc. Reliability in Distributed Software and Database Systems*. IEEE, 1982.

[Muc98] MUCHITSCH, MICHAEL: *Entwurf und prototypische Implementierung einer CORBA CSCW Facility*. Diplomarbeit, Technische Universitäat München, 1998.

[OH98] ORFALI, ROBERT und DAN HARKEY: *Client/server programming with Java and CORBA*. John Wiley & Sons, Inc., 2. Auflage, 1998.

[OMG97] OMG: *jFlow – Joint RFP Submission*, 1997. bom/97-08-05, Object Management Group.

[OMG00] OMG: *WorkflowManagementFacility Specification, V1.2*, 2000.

[Rit97] RITTER, NORBERT: *DB-gestützte Kooperationsdienste für technische Entwurfsanwendungen*. Doktorarbeit, Fachbereich Informatik der Universität Kaiserslautern, Januar 1997. Vorabversion.

[RM97] RITTER, NORBERT und BERNHARD MITSCHANG: *Die Assistenzfunktion kooperativer Designflows verdeutlicht am Beispiel von CONCORD*. Informatik Forschung und Entwicklung, 12:91–100, 1997.

[Rös99] RÖSNER, MARKUS: *Implementierung von gemeinsamen Workflow- und CSCW-Aktivitäten für CASSY*. Studienarbeit Nr. 1748, Universität Stuttgart, Fakultät für Informatik, August 1999.

[Say97] SAYEGH, MICHAEL: *Corba: Standard, Specification, Entwicklung*. Essentials. O'Reilly, 1997.

[Sin95] SINGH, MUNINDAR P.: *Semantical Considerations on Workflows: An Algebra for Intertask Dependencies*. In: *Proceedings of the International Workshop on Database Programming Languages, Gubbio, Umbria, Italy*, September 1995.

[Sin97a] SINGH, MUNINDAR P.: *Formal Aspects of Workflow Management – Part: Distributed Scheduling*. Technischer BerichtTR-97-05, Department of Computer Science, North Carolina State University, Juni 1997.

[Sin97b] SINGH, MUNINDAR P.: *Formal Aspects of Workflow Management – Part: Semantics*. Technischer BerichtTR-97-04, Department of Computer Science, North Carolina State University, Juni 1997.

[W3C00] W3C: *Extensible Markup Language (XML) 1.0*. Recommendation REC-xml-20001006, World Wide Web Consortium, 2000.

[WfM95] WFMC: *The Workflow Reference Model*. Specification TC00-1003, Workflow Management Coalition, 1995.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

_____

(Frank Wagner)