Architektur und Implementierung eines Datenzugriffsdienstes für eine Transaktionsmaschine

Diplomarbeit am Lehrstuhl für Technische Informatik
der Fakultät für Informations- und Kognitionswissenschaften
an der Universität Tübingen

Martin Schlodder <martin@schlodder.de>

betreut durch
Joachim Franz <joachim_franz@de.ibm.com>
und
Prof. Dr. Wilhelm G. Spruth <spruth@informatik.uni-tuebingen.de>

28. Mai 2004

Verwendete Hilfsmittel:

Die vorliegende Ausarbeitung wurde mit Hilfe des AucTeX-Modes [1] von GNU-Emacs [2] geschrieben und mit LaTeX [3, 4, 5, 6, 7] gesetzt. Die Zeichnungen sind, soweit nicht anders angegeben, mit XFig [8] erstellt worden, die Performance-Diagramme mit gnuplot [9]. Für das Lesen der Dokumentation kamen der Adobe Acrobat Reader (*PDF*) [10] und der IBM Softcopy Reader (Bookmanager) [11] zum Einsatz. Die gesamte Diplomarbeit wurde unter Linux [12, 13, 14] erstellt, und soweit wie nötig unter den Unix System Services (*USS*) von z/OS [15, 16, 17] compiliert und getestet. Zum Erstellen der Java-Quellen kam das *eclipse.org*-IDE [18] zum Einsatz, übersetzt wurden die Quellen unter Linux mit dem J2SE 1.4.2 SDK von Sun [19]. Die C-Quellen wurden im GNU-Emacs erstellt und mit IBMs C/C++ für z/OS [20, 21, 22, 23] übersetzt. Die Programmdokumentation wurde mit *javadoc* aus dem JDK von Sun sowie mit *doxygen* [24] automatisch aus den dokumentierten Quellen erstellt.

Erklärung zur Selbständigkeit:

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum Unterschrift

Danksagung

Ich möchte mich hier bei allen bedanken, die mich bei meiner Diplomarbeit unterstützt haben. Insbesondere möchte ich meiner Frau, Esther Hufnagel, danken, die mir während der vergangenen Monate viel Arbeit abgenommen und mich trotz Nachwuchs liebevoll umsorgt hat, so daß ich den Kopf für die Arbeit frei hatte.

Vielen Dank auch an Herrn Franz, der sich immer Zeit für mich genommen hat und alle Aspekte des Designs intensiv mit mir durchgesprochen hat. Auch und gerade in unseren kontroversen Diskussionen hat das Design viel an Substanz gewonnen.

Herrn Professor Spruth möchte ich für die Vermittlung der Arbeit danken und dafür, daß er immer ansprechbar war, wenn Probleme auftraten, und sich insbesondere um eine Lösung bemühte, als es Schwierigkeiten mit dem z/OS-Rechner gab.

Thomas Hornung möchte ich dafür danken, daß er sich die Zeit genommen hat, mir seine Diplomarbeit zu erklären, und auch während der Einarbeitungzeit immer Zeit für die Beantwortung meiner Fragen hatte.

Den Mitarbeitern der Firma TPS DATA aus Zürich möchte ich schließlich dafür danken, daß sie sich um die Einrichtung des mir zur Verfügung stehenden z/OS-Rechner bemüht haben. Insbesondere gilt mein Dank Herrn Rombach, der immer für mich ansprechbar war, wenn es Probleme gab.

Meinem Vater, Dr. Dietrich Schlodder, und meinem Studienkollegen David Jürgens danke ich dafür, daß sie kurzfristig bereit waren, meine Diplomarbeit Korrektur zu lesen.

Schließlich möchte ich noch Herrn Dr. Paul Herrmann von der Uni Leipzig und Herr Andreas Zientek dafür danken, daß sie mir die von ihnen betreuten z/OS-Rechner für erste Versuche zur Verfügung gestellt und auch einige Änderungen für mich vorgenommen haben.

Zusammenfassung und Überblick

Die vorliegende Diplomarbeit setzt die Diplomarbeit von Thomas Hornung [25] an der von Prof. W. Spruth und Joachim Franz u.a. in [26] vorgestellten Transaktionsmaschine fort. Dabei wurde auch auf die Ergebnisse der Diplomarbeit von Marc Beyerle [27] zurückgegriffen.

Das Ziel der Diplomarbeit war es, den von Thomas Hornung entworfenen Subworkflowcontroller um eine Fehlerbehandlung und einen Datenzugriffsdienst zu erweitern, und ihn für den Ablauf unter IMS und CICS und als Batch-Job einzurichten. Schließlich sollte der Subworkflowcontroller anhand eines realen Geschäftsprozesses getestet werden, wobei ein WebSphere-Frontend als provisorischer Masterflowcontroller zum Einsatz kommen sollte.

Der Schwerpunkt der gesamten Diplomarbeit sollte auf der Performance des Datenzugriffsdienstes liegen, die hauptverantwortlich für den Durchsatz des Subworkflowcontrollers ist, von dem wiederum der Gesamtdurchsatz und die Performance der Transaktionsmaschine direkt abhängt.

Diese Aufgabenstellung konnte aufgrund von funktionalen Einschränkungen der für diese Diplomarbeit von der Uni Leipzig vermittelten z/OS-Maschine nicht vollständig umgesetzt werden. Der Schwerpunkt wurde deshalb auf ein gründliches Design und die Implementierung eines Batch-Carriers gelegt, außerdem sollten die in Kapitel 2 ab Seite 9 beschriebenen Compiler zumindest ansatzweise implementiert werden.

Mit diesem Dokument liegt nun ein umfassender Entwurf des Subworkflowcontrollers (inklusive Datenzugriffsdienst und Einbindung in die Carrier CICS, IMS und Batch), eines Batch-Carriers (zur Ausführung des Subworkflowcontrollers in einem Batch-Job) und eines Business Process Designers (zur Umsetzung von Geschäftsprozessen in Programme der Transaktionsmaschine) mit einem Schnittstellen-Compiler und einem Subworkflow-Compiler vor, der auch teilweise implementiert wurde.

Weitgehend fertiggestellt wurden der Schnittstellen-Compiler, der Batch-Carrier und der Subworkflowcontroller. Der Subworkflow-Compiler beherrscht das Parsen aller legalen Workflow-Beschreibungen.

Der Datenzugriffsdienst wurde soweit implementiert, daß er die transaktionale Sicherung von Subworkflows beherrscht. Noch zu implementieren ist der Datenbankzugriff. Die Kommunikation zwischen Subworkflowcontroller und Datenzugriffsdienst über TCP/IP hat sich zudem als ineffizient erwiesen und sollte durch System V Message Queues ersetzt werden.

Die Quelltexte und Makefiles der erstellten Programme finden sich auf der beigelegten CD, auf der außerdem die Test-Programme und die Original-Logs enthalten sind, die den Performance-Diagrammen zugrunde liegen. Weiterhin sind einige der im Literatur-Verzeichnis angegebenen Quellen enthalten, und schließlich findet sich auch die vorliegende Ausarbeitung in verschiedenen Formaten mitsamt den dazugehörigen Quelltexten und Bilddateien auf dieser CD.

Der Entwurf wird in den folgenden Kapiteln der Ausarbeitung vorgestellt, beginnend mit einer Einführung in die Thematik der Transaktionsmaschine in Kapitel 1 ab Seite 1.

Darauf folgt eine Darstellung des Business Process Designers und der für die Entwicklung von Programmen für die Transaktionsmaschine entworfenen Business Process Definition Language in Kapitel 2 ab Seite 9 und das gegenüber der Arbeit von Thomas Hornung stark überarbeitete Design des Subworkflowcontrollers in Kapitel 3 ab Seite 21.

In Kapitel 4 ab Seite 31 wird das Programmiermodell der technischen Aktivitäten und System-Plugins vorgestellt, Kapitel 5 ab Seite 35 enthält eine ausführliche Beschreibung des Datenzugriffsdienstes für den Subworkflowcontroller, und in Kapitel 6 ab Seite 55 wird die Einbindung des Subworkflowcontrollers in IMS und CICS sowie der Batch-Carrier erläutert.

Betrachtungen zur Performance des vorgestellten Designs finden sich schließlich in Kapitel 7 ab Seite 63. In Kapitel 8 ab Seite 75 werden nochmals die Ergebnisse und die Möglichkeiten zur Weiterentwicklung aufgeführt.

Im Anhang finden sich Use Cases zur Beschreibung der Funktionalität des Batch-Carriers, des Subworkflowcontrollers und des Datenzugriffsdienstes (Anhang A ab Seite 77), ein Beispiel für die Umseztung einer BPDL-Beschreibung in Java-Programme durch die Compiler des Business Process Designers (Anhang B ab Seite 93), die Definition der BPDL und Beispiele dazu (Anhang C ab Seite 107) sowie eine Auflistung der Verzeichnisse auf der beigelegten CD (Anhang D ab Seite 157).

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Diplomarbeit basiert auf einem Aufsatz, der von Prof. W. Spruth und Joachim Franz unter dem Titel „Reengineering von Kernanwendungssystemen auf Großrechnern“ [26] veröffentlicht wurde, und der die Konzeption einer Transaktionsmaschine (*Transaction Engine, TE*) und ihre Anwendung im Umfeld des Finanzsektors beschreibt.

Ein erster Teil dieser Transaktionsmaschine, der sogenannte Subworkflowcontroller, wurde von Thomas Hornung im Rahmen einer Diplomarbeit an der Universität Tübingen entwickelt: „Entwurf und Implementierung einer transaktionalen Subworkflowsteuerung“ [25]. Diese Arbeit bildete die Basis für die im folgenden vorgestellten Weiterentwicklungen.

Für die Beurteilung der Möglichkeiten der Persistent Reusable Java Virtual Machine (*PR JVM*), auf der der Subworkflowcontroller ausgeführt wird, sowie für die Planung der Performancemessungen wurde zudem auf die Ergebnisse der Diplomarbeit von Marc Beyerle zurückgegriffen, die dieser an der Universität Tübingen unter dem Titel „Architekturanalyse der Java Virtual Machine unter z/OS und Linux“ [27] abgegeben hat.

Im weiteren Verlauf der Einleitung soll erläutert werden, welche Motivation zum Konzept der Transaktionsmaschine führte, welche Vorteile sie gegenüber den klassischen Lösungen hat, und wie sie aufgebaut ist.

1.1 Sinn und Zweck der Transaktionsmaschine

In den letzten Jahren wurden IT-Systeme weitgehenden Veränderungen unterzogen, die sich bisher jedoch vor allem auf das Frontend, also die Präsentationslogik, konzentrierten. Im Backend-Bereich, also dem Bereich der Geschäftslogik, wurden nur die allernötigsten Veränderungen vorgenommen, wie etwa die Umschiffung der Jahr-2000-Klippe oder die Einführung des Euro.

Während also im Frontendbereich neuere Techniken bereits weit verbreitet sind, etwa der Einsatz grafischer Benutzerführung oder die Verwendung von Java und .NET für komponentenbasierte Systeme, sind die durchaus vorhanden Bemühungen, auch im Backend objektorientierte Techniken wie J2EE einzusetzen, noch nicht sehr weit gediehen.

In größeren Unternehmen sind die Backend-Systeme üblicherweise Großrechner auf z/OS-Basis mit IMS, CICS, DB2 und Tuxedo als Transaktionsumgebung und den Datenbanksystemen VSAM, DB2, Oracle oder Adabas. Die Backend-Anwendungen sind zum größten Teil in COBOL geschrieben, das aufgrund seiner guten Wartbarkeit auch heute noch gerne eingesetzt wird, andererseits aber auch eine Sprache ist, die modernere Programmiertechniken nicht oder nur schlecht unterstützt.

Um diese älteren Backend-Systeme an moderne, Java-basierte Frontends anzubinden, kommt häufig eine Konnektor-Technik wie JCA oder CCF zum Einsatz. So können zwar die vorhandenen Backends auch in modernen Umgebungen noch vernünftig eingebunden werden, eine grundsätzliche Modernisierung der Systeme steht aber noch aus. Um diese vorzubereiten, wurde in letzter Zeit begonnen, die alten Backend-Anwendungen durch sogenanntes *Wrapping* zu isolieren und so in eine moderne Backend-Architektur einzubinden, beispielsweise als EJBs in J2EE.

Derzeit entwickelt sich ein weiterer Trend, der ein komplettes Reengineering der Backend-Systeme nötig macht und damit die Gelegenheit bietet, die komplette Geschäftslogik in ein objektorientiertes, komponentenbasiertes System zu überführen: Im Zuge der Konzentration auf ihre Kernkompetenzen versuchen viele Unternehmen, auch ihre IT zu verschlanken und auf die zentralen Abläufe hin zu optimieren, wobei „Straight-Through-Processing"-Fähigkeit (*STP*) angestrebt wird, was bedeutet, daß ein Auftrag von der Annahme bis zu seiner vollständigen Abwicklung keines manuellen Eingriffs mehr bedarf. Ein typisches Beispiel für einen STP-Prozeß wäre die automatische Kreditvergabe über ein Web-Frontend.

Genau diese Anforderungen soll nun die Transaktionsmaschine (*TE*) erfüllen: Sie bietet eine komponentenbasierte, objektorientierte Infrastruktur, die gut skaliert und eine hohe Performance und Durchsatzrate aufweist und für STP optimiert ist. Außerdem ermöglicht die TE eine relativ direkte Umsetzung des fachlichen Entwurfs eines Geschäftsprozesses auf den Anwendungsentwurf, wobei trotzdem eine strikte Trennung der Fachlogik von der technischen Logik gewährleistet ist.

1.2 Merkmale der Transaktionsmaschine

Im folgenden sind noch einmal die zentralen Anforderungen an die Transaktionsmaschine zusammengestellt:

Performance und Durchsatz müssen für bis zu 50 Mio. Einzeltransaktionen in 5 Stunden und für maximal 20 Mio. Transaktionen pro Stunde ausreichen.

Echtzeitfähigkeit, also die Möglichkeit, Transaktionen direkt zu bearbeiten oder auch an andere Systeme weiterzuleiten. Alternativ auch Stapelverarbeitung für die Abarbeitung großer, nicht zeitkritischer Aufträge.

Unterbrechungsfreier Betrieb, also die Möglichkeit, Änderungen am System online durchführen zu können.

Restart-Fähigkeit, um durch Systemfehler unterbrochene Transaktionen trotzdem fehlerfrei ausführen zu können.

Komponentenbasiertes System, das die Mehrfachverwendbarkeit von Komponenten und die Einbindung vorhandener, auch fremder Komponenten (z.B. aus Standardsoftware) ermöglicht.

Prozeßsteuerung, die neben der Verarbeitungssteuerung ein Transaktions- und SLA-Monitoring ermöglicht und die Trennung von geschäftlicher und technischer Logik sicherstellt.

Multiplattformfähigkeit für den Zugriff auf verschiedene vorhandene Datenbanken, Transaktionsmonitore und Betriebssysteme.

Mandantenfähigkeit, also die Möglichkeit, die Daten mehrerer unabhängiger Unternehmen entsprechend der vereinbarten Service Level Agreements (*SLA*) und den damit festgelegten Quality of Service (*QoS*) Parametern vollkommen getrennt zu halten und zu verarbeiten.

1.3 Definition und Abbildung eines Geschäftsprozesses

Um einen Geschäftsprozeß auf die Transaktionsmaschine (*TE*) umsetzen zu können, bedarf es zunächst eines fachlichen Entwurfs, der in einem Top-Down-Verfahren aus der ursprünglichen Definition des Geschäftsprozesses erzeugt wird (siehe Abbildung 1.1).

Dazu wird der STP-fähige Gesamtprozeß, der eine „long running transaction“ (*LRT*) darstellt, (und damit ACID-Eigenschaften besitzt,) zunächst in Teilprozesse zerlegt, die wiederum in einzelne, ununterbrechbare fachliche Aktivitäten (*Units of Work, UoW*) aufgegliedert werden. Durch eine Workflow-Beschreibung wird schließlich festgelegt, in welcher Reihenfolge die fachlichen Aktivitäten auszuführen sind, ob sie parallel ausgeführt werden können und durch welche Kompensationsmaßnahmen sie im Fehlerfall rückgängig gemacht werden können.

Im letzten Schritt des Entwurfs werden schließlich Basisoperationen der fachlichen Aktivitäten und Kompensationsmaßnahmen identifiziert und spezifiziert. Dabei muß darauf geachtet werden, daß eine ausreichend detaillierte Unterteilung erreicht wird, auf der anderen Seite aber auch keine zu starke Zergliederung erfolgt, da sonst der Verwaltungsaufwand zu hoch wird, und der Bezug zur ursprünglichen Aufgabe verloren gehen kann.

Für die Umsetzung auf die Transaktionsmaschine wird die Workflow-Beschreibung auf einen sogenannten *Masterflow* abgebildet, der den Gesamtprozeß mit seinen Teilprozessen implementiert. Die einzelnen fachlichen Aktivitäten werden durch *Subworkflows* beschrieben, die aus Sicht des Masterflow eine ununterbrechbare Einheit darstellen.

Jeder Subworkflow enthält *technische Aktivitäten* und *System-Plugins*. Die technischen Aktivitäten sind die im fachlichen Entwurf spezifizierten Basisoperationen der fachlichen Aktivitäten. Die System-Plugins sind Komponenten, die die Möglichkeiten der Transaktionsmaschine erweitern oder einfache, nicht aus dem fachlichen Entwurf abgeleitete Aufgaben wie Monitoring (evtl. nicht transaktionsrelevant) oder Accounting (transaktionsrelevant) wahrnehmen.

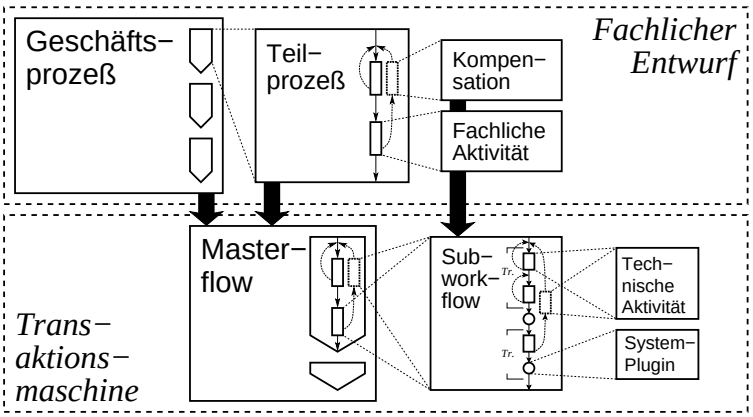


Abbildung 1.1: Abbildung eines Geschäftsprozesses auf die Transaktionsmaschine

Ein Subworkflow kann mehrere technische Transaktionen enthalten, die als „short running transactions“ (*SRT*) ausgeführt werden. Alle technischen Aktivitäten müssen in eine dieser Transaktionen eingebettet sein, während System-Plugins auch außerhalb der Transaktionen aufgerufen werden dürfen.

In einem anschließenden Bottom-Up-Vorgang werden mit Hilfe des *Business Process Desigers* die technischen Aktivitäten entwickelt und zusammen mit System-Plugins in Subworkflows eingebettet, die schließlich zu einem Masterflow zusammengefügt werden.

Das hier beschriebene Vorgehen bietet den Vorteil, daß sich die fachliche Gliederung der Aufgabe direkt in der technischen Implementierung wiederfindet, was einerseits die Übersichtlichkeit fördert und andererseits nachträgliche Änderungen deutlich erleichtert.

1.4 Aufbau der Transaktionsmaschine

Die Transaktionsmaschine (*TE*) ist selbst modular aufgebaut. Sie besteht aus den im folgenden aufgezählten Komponenten, deren Zusammenspiel in Abbildung 1.2 aufgezeigt ist:

Business Process Designer (*BPD*): Der BPD ist ein graphisches Tool, mit dessen Hilfe auf bequeme Weise ein Geschäftsprozeß für die Transaktionsmaschine umgesetzt werden kann. Er ist in fünf Module untergliedert: Im Aktivitäten-Modul werden die Ein- und Ausgabedaten der technischen Aktivitäten und der System-Plugins definiert. Unabhängig davon werden im Datenbank-Modul die Datenbanken und Tabellen festgelegt, die an diesem Geschäftsprozeß beteiligt sind. Auf die Definitionen der Aktivitäten und der Datenbanken aufbauend werden im Subworkflow-Modul einzelne Subworkflows zusammengestellt, aus denen im Masterflow-Modul der Workflow des gesamten Geschäftsprozesses aufgebaut wird. Im Scheduling-Modul schließlich werden alle Parameter für den CC/P angegeben.

Diese Angaben werden in ein modulares System von Beschreibungsdateien umgesetzt, das dem Aufbau der Transaktionsmaschine entspricht. Aus diesen Beschreibungen werden zudem Skeletons für die technischen Aktivitäten und System-Plugins sowie Programm-Module zur Workflow-Steuerung generiert. Die Definitionsdateien und Programmobjekte werden schließlich in einem zentralen Verzeichnis abgelegt, auf das die anderen Komponenten der TE zugreifen.

Eine genauere Beschreibung des BPD und der von ihm verwendeten Beschreibungssprache (*Business Process Definition Language*, *BPDL*) folgt in Anhang C ab Seite 107. Diese Komponente wurde im Rahmen der vorliegenden Diplomarbeit entworfen und spezifiziert.

Kanalkontroller und Profiler (*Channel Controller/Profiler, CC/P*): Aus den verschiedenen Ein-/Ausgabekanälen (Internet, Datenträger, Channel-to-Channel etc.) übernimmt der Kanalkontroller neue Aufträge und überführt sie, um von der Eingabe vollkommen abstrahieren zu können, in ein einheitliches internes Format: die Service Request Work Unit (*SRWU*). Diese SRWU enthält die Kennung des Auftrags und des Auftraggebers, den Zeitpunkt der Auftragsannahme und alle Parameter, die mit dem Auftrag übergeben wurden.

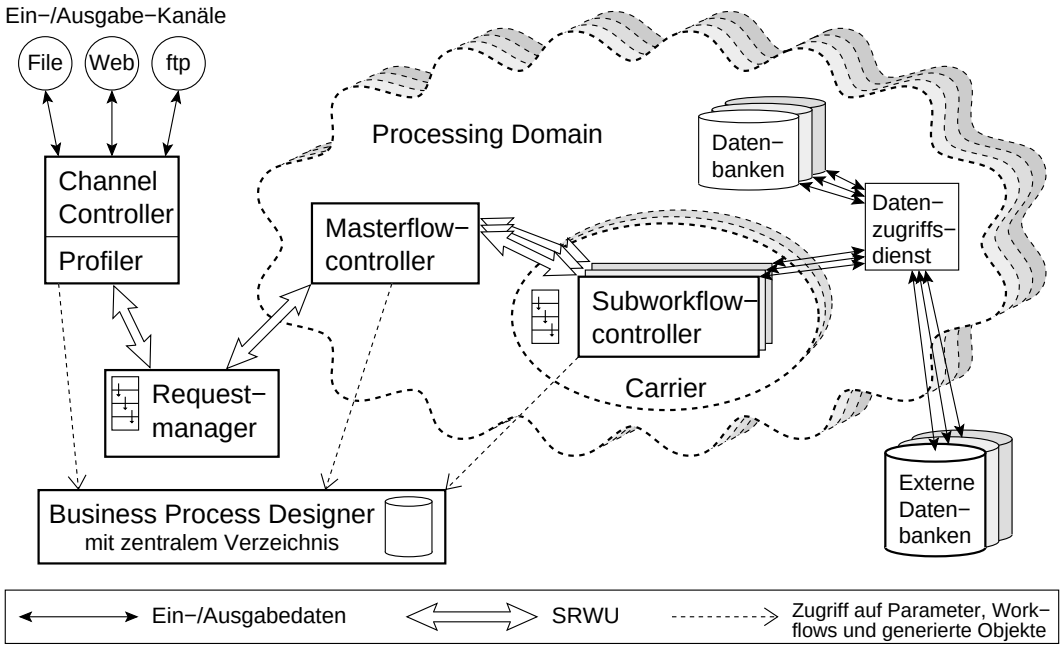


Abbildung 1.2: Die Transaktionsmaschine

Der Profiler entscheidet anschließend auf Basis der im Service Level Agreement (SLA) festgelegten QoS-Parameter und der aktuellen Lastsituation, welcher Verarbeitungsdomäne jeder einzelne Auftrag zugeordnet wird, und übergibt die zugehörige SRWU an den *Masterflowcontroller* dieser Domäne.

Auch die Kommunikation mit externen Partnern wird über den CC/P abgewickelt, indem er vom Masterflowcontroller entsprechende Nachrichten erhält. Die Antworten der externen Partner werden asynchron an den Masterflowcontroller weitergeleitet.

Die Ergebnisse der Auftragsbearbeitung erhält der CC/P in Form einer Service Response Unit (*SRU*). Diese wird in das entsprechende Ausgabeformat gebracht und an den Auftraggeber zurückgeliefert.

Der CC/P kann in jeder Verarbeitungsdomäne ausgeführt werden, um eine hohe Verfügbarkeit zu gewährleisten.

Requestmanager: Diese Komponente liefert die Infrastruktur für die Kommunikation innerhalb der Transaktionsmaschine. Insbesondere werden Warteschlangen und Ereignislisten zur Verfügung gestellt.

Verarbeitungsdomänen (*Processing Domain, PD*): Eine Verarbeitungsdomäne ist ein logischer Zusammenschluß aller Komponenten der IT-Infrastruktur, die an der Verarbeitung eines Auftrags der TE beteiligt sind. Jede PD hat ihre spezifischen Einsatzgebiete, wie etwa Online- oder Batch-Verarbeitung. Außerdem ist jede PD in mehrere Bänder unterteilt, die verschiedene QoS-Parameter aufweisen und so dem CC/P eine sehr genaue Ressourcenzuteilung ermöglichen.

Masterflowcontroller (*MFC*): Innerhalb jeder Verarbeitungsdomäne ist ein Masterflowcontroller dafür zuständig, die Aufträge auszuführen. Er entspricht dem dezentralen Teil eines Grid-Brokers, dessen zentralen Teil der CC/P darstellt, und greift für seine Aufgaben auf die Workload- und Ressourcen-Management-Fähigkeiten seiner Verarbeitungsdomäne zurück.

Aus der BPDL-Beschreibung zur aktuellen Auftragskennung entnimmt der MFC alle Angaben darüber, welche fachlichen Aktivitäten (Subworkflows) mit welchen Daten in welcher Reihenfolge und unter welchen Bedingungen (abhängig vom Ergebnis vorhergehender fachlicher Aktivitäten) auszuführen sind, und an welchen Stellen Nachrichten mit externen Partnern ausgetauscht werden sollen. Auch Angaben zu den benötigten Ressourcen und den möglichen Carriern finden sich hier.

Mit der Abarbeitung der einzelnen Subworkflows werden Subworkflowcontroller beauftragt, indem eine entsprechende Service Request Work Unit für den Subworkflow (*SWF-SRWU*) an den nach QoS-Kriterien ausgewählten Carrier übergeben wird. Diese SWF-SRWU enthält die Kennung des betreffenden Subworkflow (zusammengesetzt aus dem Namen und der Version der dazugehörigen BPDL-Beschreibung), alle Daten aus der SRWU, die dieser Subworkflow benötigt[1], sowie eine in der Verarbeitungsdomäne eindeutige Auftrags-ID, die aus der Kennung des Auftraggebers, der Art des Auftrags und dem Zeitpunkt der Auftragsannahme besteht.

Ergebnisse und Meldungen des Subworkflowcontrollers werden vom Carrier in Form von Ausgabe-, Meldungs- oder Fehler-SWF-SRUs gelesen und ausgewertet. Einige Meldungen werden an den CC/P weitergeleitet, andere verworfen.[2] Auf Fehler kann in vielfältiger Weise reagiert werden, beispielsweise kann der fehlgeschlagene Subworkflow wiederholt werden, oder der komplette Workflow wird mit einer Fehlermeldung abgebrochen, und vorangegangene Arbeitsschritte werden über Kompensations-Subworkflows rückgängig gemacht.

Die Ergebnisse des gesamten Workflow werden schließlich an den CC/P zurückgemeldet, womit dieser Auftrag für den MFC erledigt ist.

Carrier: Der Carrier startet abhängig von der Anzahl der vom MFC übergebenen Aufträge mehrere Subworkflowcontroller, denen er die Auftragsdaten (SWF-SRWUs) weiterreicht. Außerdem stellt er den SWFCs eine transaktional abgesicherte Umgebung zur Verfügung. Da auch die Subworkflowcontroller selbst transaktionsgeschützt arbeiten, kann auf ein Commit-Protokoll zwischen Masterflowcontroller und Subworkflowcontroller verzichtet werden, was den Overhead bei der Ausführung der fachlichen Aktivitäten deutlich verringert.

Von der Transaktionsmaschine werden verschiedenste Carrier unterstützt, die abhängig vom Auftrag ausgewählt werden können. Darunter sind Online-Carrier wie IMS, CICS oder MQ wie auch ein Batch-Carrier für die Abarbeitung großer Aufträge, die im Hintergrund abgearbeitet werden können. Der Batch-Carrier kann besonders effizient eingesetzt werden, wenn er exklusiven Zugriff auf die Betriebsmittel der PD hat.

[1] Welche Daten das sind, wird in der Workflow-Beschreibung des Masterflow und in der Interface-Beschreibung des Subworkflow festgelegt.

[2] Die SWF-SRUs folgen dabei einer at-least-once Semantik: Sie werden dem MFC auf jeden Fall einmal zugestellt, können nach einem Systemfehler aber vereinzelt auch mehrfach eintreffen.

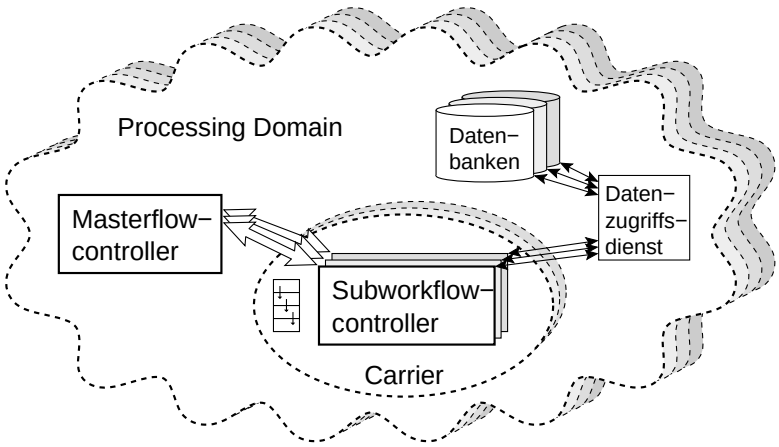


Abbildung 1.3: Die Verarbeitungsdomänen der Transaktionsmaschine

Subworkflowcontroller (*SWFC*): Der Subworkflowcontroller läuft innerhalb eines Carriers (z.B. CICS, IMS oder Batch) ab, von dem er zu Beginn eine SWF-SRWU erhält. Er führt nun den in dieser SWF-SRWU festgelegten Subworkflow mit Hilfe eines endlichen Automaten aus, der den Anweisungen aus der BPDL-Beschreibung des Subworkflow folgt.

Ein Subworkflow kann dabei mehrere technische Transaktionen enthalten, die ihrerseits die Ausführung von mehreren technischen Aktivitäten umfassen. Zusätzlich können an beliebigen Stellen des Subworkflow System-Plugins aufgerufen werden. Die Ein- und Ausgabedaten der technischen Aktivitäten und System-Plugins werden vom SWFC zur Verfügung gestellt, der Zugriff auf Datenbanken erfolgt hierbei über den Datenzugriffsdienst.

Rückmeldungen an den MFC erfolgen an beliebigen Stellen des Subworkflow durch Nachrichten in Form von Meldungs-SWF-SRUs, am Ende des Subworkflow in Form der Ausgabe-SWF-SRU oder im Fehlerfall durch eine Fehler-SWF-SRU, wobei auch die laufende technische Transaktion abgebrochen und der Subworkflowcontroller beendet wird.

Eine detaillierte Beschreibung des Subworkflowcontrollers findet sich in Kapitel 3 ab Seite 21.

Datenzugriffsdienst (*Data Service, DS*): Der Datenzugriffsdienst steht allen Subworkflowcontrollern in derselben Rechnerumgebung[3] gemeinsam zur Verfügung. Er ist als eine weitere Komponente zwischen den SWFC und die Datenbanken geschaltet, und übernimmt auch Aufgaben für die transaktionale Sicherung der Subworkflows.

In Zusammenarbeit mit dem Subworkflowcontroller ermöglicht der Datenzugriffsdienst den technischen Aktivitäten und System-Plugins einen einheitlichen Zugriff auf alle globalen und lokalen Daten des Subworkflow, unabhängig davon, ob diese aus der SRWU oder den Datenbanken stammen. Er wird detailliert in Kapitel 5 ab Seite 35 beschrieben.

[3]Mit „Rechnerumgebung“ ist hier ein IT-System gemeint, das gemeinsamen Zugriff auf ein Filesystem und praktisch verzögerungsfreie Kommunikation über TCP/IP bietet, was zum Beispiel auch für einen Multirechner mit sehr schnellem Switch gilt. Die Rechnerumgebung kann mit der Verarbeitungsdomäne identisch sein oder nur einen Teil davon darstellen.

Fachkomponenten: Die Fachkomponenten fügen sich in Form von technischen Aktivitäten in den Subworkflow der Transaktionsmaschine ein. Die technischen Aktivitäten sollten möglichst atomar und damit vielseitig wiederverwendbar sein, aber nicht zu trivial werden, da sonst der Verwaltungsaufwand ungerechtfertigt hoch sein würde.

Systemkomponenten: Systemkomponenten werden als System-Plugins in den Subworkflow eingebunden und erweitern die Möglichkeiten der Transaktionsmaschine um vielfältige, von der Fachlogik unabhängige Möglichkeiten. Sie werden beispielsweise für die Ablaufverfolgung und das Accounting eingesetzt.

Kapitel 2

Umsetzung eines Geschäftsprozesses

Alle Vorgänge innerhalb der Transaktionsmaschine (*TE*) und alle Parameter, die zur Zeit der Erstellung eines Geschäftsprozesses bekannt sind, werden in verschiedenen Beschreibungsdateien genau definiert. Diese Beschreibungsdateien sind in der Business Process Definition Language (*BPDL*, siehe Abschnitt 2.2 ab Seite 13) verfaßt und werden von einem graphischen Tool erzeugt, dem Business Process Designer (*BPD*).

In Abbildung 2.1 auf Seite 11 sind die durch den Business Process Designer erzeugten BPDL-Beschreibungen zusammengefaßt und mit den dazugehörigen Komponenten der Transaktionsmaschine dargestellt.

2.1 Der Business Process Designer

Die Entwicklung des Business Process Designers (*BPD*) ist nicht Teil dieser Diplomarbeit, weshalb er hier nur in groben Zügen beschrieben werden soll. Er ist eine Entwicklungsumgebung für Geschäftsprozesse, die durch die Transaktionsmaschine ausgeführt werden sollen.

2.1.1 Anwendung des BPD

Vorraussetzung für den Einsatz des Business Process Designers ist ein fachlicher Entwurf des umzusetzenden Geschäftsprozesses, wie er durch das in Abschnitt 1.3 ab Seite 3 beschriebene Top-Down-Verfahren erzeugt werden kann. Der BPD wird dann in einem Bottom-Up-Designprozeß eingesetzt, um den fachlichen Entwurf umzusetzen.

Um den Benutzer nicht zu überfordern, aber auch um die Oberfläche nicht allzu sehr zu überladen, bietet der BPD fünf „Sichten“ oder „Views“ der Transaktionsmaschine, die sich auch in Form eines „Wizards“ nacheinander durcharbeiten lassen könnten.

Datenbanken: In dieser Sicht werden die beteiligten Datenbanksysteme (zum Beispiel DB2, VSAM, IMS/DB oder Oracle) benannt und die benötigten Tabellen definiert, die auf Wunsch auch gleich angelegt werden können. Für jede Datenbank wird eine eigene BPDL-Datenbankbeschreibung erzeugt.

Außerdem muß für jeden Datenbankzugriff eine eigene Schnittstellenbeschreibung angelegt werden, die angibt, auf welche Weise welche Daten aus der Datenbank ausgelesen oder in die Datenbank geschrieben werden sollen, und diesen Daten eindeutige Namen zuordnet.

Auch mengenorientierte Datenbankabfragen sind möglich. Die Ergebnisse werden hierbei einer Feldvariablen zugewiesen.

Aktivitäten und Plugins: In der Aktivitäten-Sicht werden die Schnittstellen der technischen Aktivitäten und der System-Plugins festgelegt, also deren Ein- und Ausgabedaten und im Falle von Java-Aktivitäten und -Plugins auch Fehlerdaten.[1] Diese Informationen werden für jede Aktivität und jedes Plugin einzeln in einer BPDL-Schnittstellenbeschreibung abgelegt.

Aus den Schnittstellenbeschreibungen werden mit Hilfe des Schnittstellen-Compilers (siehe Unterabschnitt 2.1.2 auf Seite 11) Datenobjekte für die Ein- und Ausgabedaten der Aktivitäten / Plugins und außerdem Programmgerüste (Skeletons) für Java- und COBOL-Aktivitäten und -Plugins automatisch erzeugt.

Subworkflows: In der Subworkflow-Sicht werden die technischen Aktivitäten und System-Plugins zu Subworkflows zusammengestellt. Dafür steht ein graphischer Editor zur Verfügung, der Aktivitäten, Plugins, Variablen und transaktionale Steuerungselemente als Symbole darstellt und in einen Ablaufplan integrieren kann. Die Variablen können dabei lokal oder global sein, wobei globale Variablen sowohl auf die Ein- und Ausgabedaten des Subworkflow und seiner technischen Transaktionen als auch auf Datenbankzugriffe gemapped werden können.

Jeder Subworkflow des Geschäftsprozesses wird wieder in einer eigenen BPDL-Workflowbeschreibung abgelegt, und in einer BPDL-Schnittstellenbeschreibung werden die Ein- und Ausgabeparameter (die SWF-SRWU und die SWF-SRUs) definiert. Konfigurationsdaten des Subworkflow, wie etwa Debugging-Ausgaben, Timeouts und Wiederholungsversuche, werden in einer eigenen BPDL-Parameterbeschreibung abgelegt.

Aus der Schnittstellenbeschreibung des Subworkflow werden mit Hilfe des Schnittstellen-Compilers (siehe Unterabschnitt 2.1.2 auf Seite 11) Java-Objekte für die SWF-SRWU, die temporären SRWUs zur Übergabe globaler Variablen zwischen den technischen Transaktionen und die verschiedenen SWF-SRUs generiert (siehe Abschnitt 1.4 ab Seite 4, Unterabschnitt 2.2.3 ab Seite 17 und Abschnitt 3.2 ab Seite 22), und aus der Subworkflowbeschreibung und den Schnittstellenbeschreibungen der beteiligten Aktivitäten und Plugins wird mit Hilfe des Subworkflow-Compilers (siehe Unterabschnitt 2.1.3 auf Seite 12) eine Subworkflow-Runtime (*SWFR*, siehe Unterabschnitt 3.4.3 ab Seite 25) erzeugt, die den Subworkflow ausführt.

Masterflow: Der Masterflow wird aus den einzelnen Subworkflows in einem graphischen Editor zusammengesetzt, der ähnlich dem Subworkflow-Editor aufgebaut ist. Auch hier werden alle Informationen in entsprechenden BPDL-Beschreibungen für den Workflow, die Schnittstelle (also die SRWU und die SRU) und die Konfigurationsdaten abgelegt.

Scheduling: In der Scheduling-Sicht werden alle Parameter festgelegt, die das Scheduling des Geschäftsprozesses durch den CC/P und den MFC beeinflussen, wie etwa Parameter für die Auswahl der PD oder des Carriers. Diese Informationen werden in den Parameterbeschreibungen des Subworkflow und des Masterflow abgelegt.

[1] Wenn man den BPD als Plugin für eine flexibel erweiterbare Entwicklungsumgebung wie Eclipse entwickeln würde, könnte man in der gleichen Umgebung auch die Java-Aktivitäten und -Plugins schreiben.

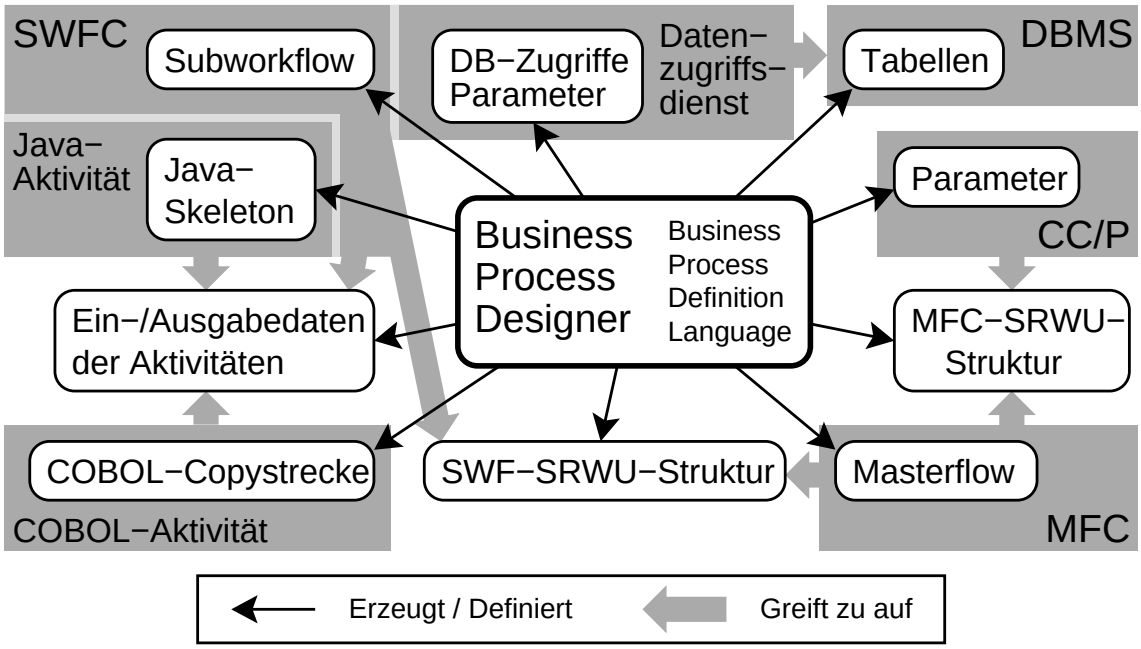


Abbildung 2.1: Durch den Business Process Designer erzeugte Definitionen

Transaktionsmaschine: In dieser Sicht können die vom Geschäftsprozeß unabhängigen Konfigurationsdaten der Transaktionsmaschine festgelegt werden. Dazu gehören die Positionen der Beschreibungen, der Aktivitäten und Plugins wie auch Angaben über die vorhandene Infrastruktur. Diese Daten werden in je einer eigenen Parameterbeschreibung für den CC/P, den MFC, den Batch-Carrier, den SWFC und den Datenzugriffsdienst abgelegt.

Jede der beschriebenen Sichten ist in zwei Bereiche gegliedert: einen Navigator, in dem sich die zur jeweiligen Sicht gehörenden Elemente auswählen lassen, und einen Editor, in dem alle Aspekte des ausgewählten Elements bearbeitet werden können. Diese Editoren enthalten meist Textfelder und Listen, sind zum Teil aber auch grafikorientiert mit drag'n'drop von Symbolen.

Die erzeugten Beschreibungen und generierten Programm-Objekte werden vom BPD in einem zentralen Verzeichnis (LDAP, Datenbank oder NFS) abgelegt, auf das die Komponenten der Transaktionsmaschine direkten Zugriff haben. Auf diese Weise können geänderte oder neue Geschäftsprozesse der TE ohne Umwege sofort zugänglich gemacht werden.

2.1.2 Der Schnittstellen-Compiler

Der Schnittstellen-Compiler erzeugt aus BPDL-Schnittstellenbeschreibungen von technischen Aktivitäten und System-Plugins Java-Klassen oder COBOL-Copystrecken für die darin definierten Datencontainer [2]. Außerdem werden Java- und COBOL-Skeletons aus den dort definierten `portType`*s* erzeugt, um den Programmierern die Arbeit zu erleichtern. Die Umsetzung einer BPDL-Beschreibung durch den Schnittstellen-Compiler wird in Anhang B ab Seite 93 an einem praktischen Beispiel gezeigt.

[2] In der WSDL-Terminologie werden sogenannte Nachrichten (`message`) als universelle Datencontainer für alle Ein- und Ausgabedaten wie auch Fehlerdaten verwendet.

Welche Objekte jeweils generiert werden, entscheidet der Compiler anhand der `exec`-Einträge (siehe Unterabschnitt 2.2.2 ab Seite 15) in der Beschreibung der Schnittstellen. Ist kein solcher Eintrag vorhanden, so handelt es sich weder um eine Aktivität noch um ein Plugin. Die Beschreibung kann dann vom Schnittstellen-Compiler ignoriert werden; sie wird, falls nötig, vom Subworkflow-Compiler eingelesen. Existiert ein `exec`-Eintrag für eine Java-Aktivität oder ein Java-Plugin, so werden Java-Klassen für alle Datencontainer, eine Java-Interface-Klasse[3] und ein Java-Skeleton für die Aktivität oder das Plugin generiert. Steht der Eintrag für ein COBOL-Programm, dann wird eine COBOL-Copystrecke und ein COBOL-Skeleton erzeugt.

Als Parser-Generator für den Schnittstellen-Compiler und den im nächsten Abschnitt beschriebenen Subworkflow-Compiler wird die *Java Architecture for XML Binding* (*JAXB*) [28] eingesetzt, die aus einer XML-Schema-Datei [29] automatisch den passenden Parser erzeugt.

2.1.3 Der Subworkflow-Compiler

Der Subworkflow-Compiler generiert aus einer Subworkflow-Beschreibung, der Parameter-Beschreibung und den Schnittstellen-Beschreibungen des Subworkflow und der beteiligten Aktivitäten ein Java-Programm, das den Subworkflow steuert: Die Subworkflow-Runtime (*SWFR*, siehe Unterabschnitt 3.4.3 ab Seite 25).

Der Compiler protokolliert zunächst die Daten-Abhängigkeiten innerhalb des Subworkflow und erzeugt anschließend den Code für den Subworkflow. Dabei werden die Anweisungen des Subworkflow im Prinzip der Reihe nach umgesetzt. Weitere Angaben dazu, wie die dabei erzeugte SWFR aussieht, finden sich in Unterabschnitt 3.4.3 ab Seite 25.

Um den Subworkflow in Hinblick auf seine Ausführungszeit zu optimieren, wird der Programmcode der technischen Aktivitäten und System-Plugins bereits zu Beginn des Subworkflow angefordert und asynchron geladen, und auch die Datenbankzugriffe werden so früh wie möglich initiiert und asynchron ausgeführt. Hierbei werden die Namen der BPDL-Beschreibungen der Datenbankzugriffe und die Reihenfolge der Definitionen innerhalb dieser Beschreibungen verwendet, um die Zugriffe auf die Datenbanken innerhalb der SWFR so weit als möglich zu ordnen. Dadurch wird eine recht effiziente Deadlock-Vermeidung erzielt. In den wenigen Fällen, in denen es trotzdem zu einer Verklemmung kommen würde, muß der Datenzugriffsdienst (siehe Unterunterabschnitt 5.6.4.2 auf Seite 52) dafür sorgen, daß die kritische Anforderung zurückgewiesen wird. Ein Beispiel für die Erstellung einer einfachen SWFR findet sich in Anhang B ab Seite 93.

2.1.4 Organisation des zentralen Verzeichnisses

Die von den Compilern des BDP erzeugten Objekte werden nach ihrer Übersetzung durch Java- und COBOL-Compiler in verschiedenen Unterverzeichnissen innerhalb des zentralen Verzeichnisses abgelegt.

Subworkflow-Runtimes, Datencontainer und die Exceptions der Java-Aktivitäten werden im Java-Middleware-Verzeichnis abgelegt. In Java verfaßte technische Aktivitäten und System-Plugins werden, abhängig von ihrem `persistent`-Attribut (siehe Unterabschnitt 2.2.2 auf Seite 15), im Sharable-Application- oder Transient-Application-Verzeichnis abgelegt. COBOL-Programme schließlich landen in einem eigenen Unterverzeichnis.

[3] Diese Interface-Klasse wird benötigt, um der SWFR einen schnellen Zugriff auf die Aktivität zu ermöglichen. Ohne ein solches Interface müßte die Aktivität über Reflection nicht nur initialisiert sondern auch aufgerufen werden, was sehr teuer wäre.

2.2 Die Business Process Definition Language

Die Definition von Geschäftsprozessen für die Transaktionsmaschine (*TE*) sollte in einer Beschreibungssprache erfolgen, die relativ weit verbreitet ist, da eine reine Insellösung Anpassungen an Entwicklungen im Bereich des „Business Processing“ erschweren würde. Auf der anderen Seite gibt es aber anscheinend derzeit keine solche Sprache, die alle Anforderungen der TE erfüllen kann.

Insbesondere hat der Autor keine Prozeßbeschreibungssprache finden können, die Transaktionsschutz, eine direkte Einbindung von Programmen über deren Namen und ein brauchbares Modell für den Zugriff auf Datenbanksysteme bietet. Deshalb hat sich der Autor entschieden, eine vorhandene Sprache um die notwendigen Fähigkeiten zu erweitern.

Die Wahl für diese Basissprache fiel auf die Prozeßbeschreibungssprache „Business Process Execution Language for Web Services“ (*BPEL4WS*)[4] [31], die ihrerseits auf den Daten- und Interfacebeschreibungen der Webservice-Beschreibungssprache „Web Service Definition Language“ (*WSDL*) [32] aufsetzt. Beide Sprachen basieren auf XML [33]. BPEL4WS ist eine der mächtigsten Prozeßbeschreibungssprachen und erlaubt sie eine flexible Erweiterung der Sprachdefinition. Sie ist gerade dabei, sich als Standard für Prozeßbeschreibungssprachen zu etablieren.

Die für die Transaktionsmaschine entwickelte *Business Process Definition Language* (*BPDL*)[5] erweitert BPEL4WS um die Definition von Datenbankzugriffen (DB2), die Angabe von auszuführenden Programmen (Java und COBOL), die transaktionale Sicherung von Prozeßabschnitten, Savepointing und Parameterdefinitionen. Die Definition von Datenbankzugriffen ist für den Datenzugriffsdienst nötig, damit dieser alle von den technischen Aktivitäten benötigten Daten aus den am Geschäftsprozeß beteiligten Datenbanksystemen lesen kann. Für die Ausführung von technischen Aktivitäten und System-Plugins benötigt der Subworkflowcontroller die Angaben darüber, welches Programm dazu aufgerufen werden muß. Für die transaktionale Absicherung des Subworkflow sind die transaktionalen Erweiterungen und das Savepointing nötig, und die Parameterdefinitionen werden für die Konfigurationsdaten der Komponenten der TE verwendet.

Die BPDL löst die ebenfalls auf BPEL4WS basierende Subworkflow Definition Language (*SWFDL*) ab, die in der Diplomarbeit von Thomas Hornung [25] beschrieben wurde. Eine formale Beschreibung der BPDL und Beispiele zu den verschiedenen Beschreibungsklassen finden sich in Anhang C ab Seite 107.

2.2.1 Aufbau einer Prozeßbeschreibung

In einer stark modularisierten Struktur werden durch die Business Process Definition Language (*BPDL*) alle Workflows, alle Ein- und Ausgabedaten, alle beteiligten Datenbanken und alle Konfigurationsdaten beschrieben, die die Transaktionsmaschine zur Ausführung eines Geschäftsprozesses benötigt (siehe dazu auch Abbildung 2.1 auf Seite 11).

[4] BPEL4WS wurde von den ursprünglichen Autoren (Bea, IBM, Microsoft, SAP und Siebel) inzwischen an OASIS übergeben, und wird dort unter dem Namen WSBPEL weiterentwickelt [30]. Da die hier beschriebene BPDL jedoch auf der Version 1.1 von BPEL4WS basiert, (deren Beschreibung von OASIS bisher unverändert übernommen wurde,) wird hier weiter der alte Name verwendet.

[5] Der Begriff „Business Process Definition Language“ wird in verschiedenen Veröffentlichungen generisch für Sprachen verwendet, die der Beschreibung von Geschäftsprozessen dienen, darunter BPEL4WS. Hier wurde dieser Begriff eingesetzt, um anzudeuten, daß es sich um eine spezielle Sprachvariante handelt, die jedoch keinen Standard darstellt.

Dafür existieren verschiedene Subsets der BPDL zur Beschreibung von Workflows, von Schnittstellen, von Datenbanken und von Parametern, die in den folgenden Abschnitten erläutert werden. Die Modularisierung der Beschreibung erlaubt es zum Beispiel, einzelne Aktivitäten in mehreren verschiedenen Subworkflows und auch diese wieder in mehreren verschiedenen Masterflows einzusetzen. Dabei werden jeder Komponente der Transaktionsmaschine die für die Ausführung eines Geschäftsprozesses notwendigen Informationen durch jeweils eigene Beschreibungen zur Verfügung gestellt.

Kanalkontroller/Profiler (*CC/P*) und Verarbeitungsdomäne (*PD*)

- Alle Angaben über die Systemkonfiguration, die der CC/P für das Auffinden der BPDL-Beschreibungen und für das Scheduling benötigt, finden sich in der Parameterbeschreibung des CC/P.
- In der Parameterbeschreibung für den Masterflow des auszuführenden Geschäftsprozesses finden sich die für diesen Masterflow spezifischen Anforderungen an die zu verwendende Verarbeitungsdomäne.

Masterflowcontroller (*MFC*) und Carrier

- Alle Angaben über die Systemkonfiguration, die der MFC für den Zugriff auf BPDL-Beschreibungen und für die Lastverteilung innerhalb seiner PD benötigt, finden sich in der Parameterbeschreibung des MFC.
- In der Schnittstellenbeschreibung des Masterflow werden dessen Ein- und Ausgabedaten definiert, also das Format der SRWU und der SRU. Diese Schnittstelle wird für die Kommunikation des CC/P mit dem MFC benötigt.
- Der Ablauf des Masterflow wird in einer Workflowbeschreibung festgelegt. Diese referenziert die Schnittstellenbeschreibungen der beteiligten Subworkflows und die Beschreibung der SRWU und SRU in der Schnittstellenbeschreibung des Masterflow.
- Alle weiteren Angaben zur Ausführung des Masterflow, wie die Festlegungen von Timeouts oder Wiederholungen nach Fehlern, finden sich in der Parameterbeschreibung des Masterflow.
- In der Parameterbeschreibung jedes Subworkflow findet der MFC die Angaben darüber, für welche Arten von Carriern dieser Subworkflow ausgelegt ist.

Batch-Carrier

- Jeder Batch-Carrier wird durch eine eigene Parameterbeschreibung konfiguriert, in der die Pfade zum Subworkflowcontroller und zu den benötigten Java-Bibliotheken abgelegt sind, ebenso wie Angaben zur Anzahl der zu startenden Subworkflowcontroller.

Subworkflowcontroller (*SWFC*)

- Alle Angaben über die Systemkonfiguration, die der SWFC für den Zugriff auf den Datenzugriffsdienst und die BPDL-Beschreibungen sowie für die Ausführung der technischen Aktivitäten und System-Plugins benötigt, finden sich in der Parameterbeschreibung des SWFC.

- Die Ein- und Ausgabedaten (also die SWF-SRWU und die verschiedenen SWF-SRUs) für jeden Subworkflow sind in eigenen Schnittstellenbeschreibungen definiert. Über diese Schnittstellen findet die Kommunikation zwischen MFC und SWFC statt.
- Der Ablauf eines jeden Subworkflow wird in einer eigenen Workflowbeschreibung definiert, in der die Schnittstellenbeschreibung des Subworkflow, der eingebunden Aktivitäten und Plugins sowie der beteiligten Datenbanken verwendet werden.
- Alle Angaben zur Ausführung des Subworkflow, die nicht in dessen Workflowbeschreibung enthalten sind, finden sich in der Parameterbeschreibung des Subworkflow. Dazu gehören Debugging-Ausgaben, Timeouts oder Wiederholungen nach Fehlern.

Technische Aktivitäten und System-Plugins

- Für jede technische Aktivität und jedes System-Plugin wird in einer Schnittstellenbeschreibung festgelegt, welche Ein- und Ausgabevariablen benötigt werden. Der SWFC stellt sicher, daß Aktivitäten und Plugins nur auf diese Daten zugreifen können.

Datenbanken

- Für jeden Zugriff auf eines der beteiligten Datenbanksysteme wird eine Schnittstellenbeschreibung benötigt, die auch die Definition mehrerer zusammengehörender Zugriffe enthalten kann.
- Außerdem existiert für jede beteiligte Datenbank eine Datenbankbeschreibung, in der die Tabellen und Felder (Columns) der Datenbank definiert werden, auf die der Geschäftsprozeß Zugriff haben soll.

Datenzugriffsdienst

- In der Parameterbeschreibung des Datenzugriffsdienstes sind alle Angaben abgelegt, die dieser für seine Arbeit benötigt. Dazu gehören Angaben über die vorhandenen Datenbanken und ob er exklusiven Zugriff auf diese hat, wie auch Parameter für die Lastverteilung innerhalb des Datenzugriffsdienstes.

2.2.2 Schnittstellenbeschreibungen

Für die Definition der Ein- und Ausgabewerte von technischen Aktivitäten und System-Plugins und von Master- und Subworkflows werden Schnittstellenbeschreibungen angelegt. Auch für die Definition von Datenbankzugriffen wird eine spezielle Variante dieser Beschreibungen verwendet.

Schnittstellenbeschreibungen werden durch einen Subset von BPDL dargestellt, genannt Business Process Definition Language for Interfaces, kurz BPDL/I. Dieser Subset baut direkt auf WSDL auf, wobei jedoch nur die Tags[6] `message` und `portType` übernommen wurden und dafür die im folgenden aufgezählten Tags dazugekommen sind.

[6]In diesem und den folgenden Unterabschnitten wird für Elemente der Auszeichnungssprache XML der Begriff „Tag“ verwendet, da der Begriff des „Elements“ im Deutschen zu vielseitig ist und man deshalb nur von XML-Elementen reden dürfte. Attribute von XML-Elementen werden hier einfach als „Attribute“ bezeichnet.

Zunächst soll jedoch kurz auf die Verwendung der Tags `message` und `portType` in der BPDL/I eingegangen werden. In `message`-Tags werden die Ein- und Ausgabedaten aller in der BPDL/I definierten Schnittstellen festgelegt. Im `portType` wird dann die über die Schnittstelle verfügbare Operation definiert, die im Fall von Masterflows und Subworkflows nur die SRWU sowie die Ausgabe-, Meldungs- und Fehler-SRUs definiert, im Fall von Aktivitäten und Datenbanken aber zusätzliche Angaben in Form der nachfolgend vorgestellten Tags `exec` und `database` enthalten.

Eine formale Spezifikation der Schnittstellenbeschreibungen ist in Unterabschnitt C.1.1 ab Seite 107 zu finden, einige Beispiele dazu in Unterabschnitt C.2.1 ab Seite 139.

`persistent` Wird dieses Attribut im Top-Element der Schnittstellenbeschreibung einer Java-Aktivität gesetzt, so verbleibt die Klassendefinition dieser Aktivität für die gesamte Laufzeit des SWFC im Speicher und kann so schneller ausgeführt werden.

`settings` Dieses Tag ist nur für Masterflow- und Subworkflow-Beschreibungen relevant. Über `settings` werden die BPDL-Parameterbeschreibungen des Masterflow oder Subworkflow eingebunden, damit die aufrufende Komponente der TE (CC/P oder MFC) Zugriff auf die benötigten Scheduling-Parameter hat:

```
<bpdl:settings name="Parameter-Referenz"/>
```

`exec` Dieses Tag ist in ein `operation`-Tag innerhalb eines WSDL `portType` eingebettet und erlaubt in Aktivitäten- und Plugin-Beschreibungen die Angabe von Programmen, die bei Aktivierung einer Schnittstelle (`invoke`) ausgeführt werden. Unterstützt werden derzeit COBOL-Programme und Java-Methoden. Java-Operationen dürfen dabei ein `fault`-Tag enthalten, das eine Java-Exception gleichen Namens repräsentiert. Für COBOL-Interfaces wird das `fault`-Tag nicht unterstützt.

```
<bpdl:exec lang="COBOL" object="Programm"/>
```
bzw.
```
<bpdl:exec lang="Java" method="Methode"/>
```

`database, select, update, insert, destination, source` Diese Tags erlauben, eingebettet in das `operation`-Tag eines `portType` von WSDL, die Spezifikation von Datenbankzugriffen. Derzeit werden ausschließlich SQL-Datenbanken (z.B. DB2) unterstützt. Das `mode`-Attribut im `select`-Tag gibt dabei an, ob die Daten für das Lesen (`read`) oder Schreiben (`update`) gelocked werden müssen, oder ob gar kein Lock erforderlich ist (`fetch`). Außerdem kann durch einen Submodus angegeben werden, ob Datenbank-Locks eine hohe Nebenläufigkeit ermöglichen (`concurrent`) oder eine hohe Abschirmung gegenüber anderen Datenbankzugriffen (`isolated`) aufweisen sollen. Über einen `fault`-Eintrag mit leerer Nachricht wird ein aufgetretener Fehler angezeigt.

```
<bpdl:database name="DB-Name" type="DB2">
  <bpdl:select field="Feld" table="Tabelle" where="Bedingung"
    mode="update,cuncurrent"/>
  <bpdl:destination part="Element"/>
<bpdl:database/>
```
bzw.

```
<bpdl:database name="DB-Name" type="DB2">
  <bpdl:update field="Feld" table="Tabelle" where="Bedingung"/>
  <bpdl:source part="Element"/>
</bpdl:database>
```

bzw.

```
<bpdl:database name="DB-Name" type="DB2">
  <bpdl:insert table="Tabelle"/>
  <bpdl:source part="Element"/>
</bpdl:database>
```

2.2.3 Workflowbeschreibungen

Die Definition von Master- und Subworkflows wird in einem weiteren Subset von BPDL formuliert, der Business Process Definition Language for Workflows, kurz BPDL/F, die eine Erweiterung von BPEL4WS um die explizite Angabe von Timeouts, transaktionalen Steuerungselementen, Anweisungen zum Savepointing, Systemvariablen und um ein Schleifenkonstrukt zur effizienten Handhabung von mengenorientierten Datenbankabfragen darstellt.

Dafür wurden die folgenden Elemente von BPEL4WS nicht übernommen, um die Komplexität in einem vertretbaren Rahmen zu halten: `correlationSets` und `eventHandlers` und die dazugehörigen Elemente sowie `wait`, `terminate`, `pick`, `compensate` und `scope`, außerdem der globale `compensationHandler` und der lokale von `invoke`.

Die Erweiterungen werden im folgenden kurz beschrieben, die formale Spezifikation von BPDL/F findet sich in Unterabschnitt C.1.2 ab Seite 118. Beispiele für einen Subworkflow und einen Masterflow sind in Unterabschnitt C.2.2 ab Seite 145 zu finden.

`timeout` Dieses Attribut kann in `invoke`-Tags eingebettet werden, um einen expliziten Timeout für diesen Aufruf zu setzen. Die Zeitangabe erfolgt dabei standardmäßig in Nanosekunden, kann jedoch über die Angabe der entsprechenden Einheiten (*us, ms, s, m, h, d*) auch in gröberen Schritten gesetzt werden.

```
<invoke partnerLink="Partnerlink" portType="Porttyp" operation="Operation"
  inputVariable="Eingabe" outputVariable="Ausgabe"
  bpdl:timeout="Zeit-Angabe"/>
```

`transaction, rollback` Diese Tags steuern den transaktionalen Kontext des Subworkflow. `transaction` bildet eine Umgebung, die dem BPEL4WS-Tag `scope` nachempfunden ist[7]. Durch die explizite Angabe von Ein- und Ausgangsdaten (SRWU und Ausgabe-SRU bzw. temporäre SRWUs zwischen zwei Transaktionen) durch `inputVariable` und `outputVaria ble` sind die globalen Daten der Transaktion klar gekennzeichnet und können persistiert werden. Die Angabe eines `retryCount` legt fest, wie oft die Transaktion bei nicht in der Workflowbeschreibung abgefangenen Fehlern wiederholt werden soll.

[7] Tatsächlich wird das `scope`-Tag in BPEL4WS ebenfalls als transaktionaler Kontext verstanden, nur ist es dort nicht möglich, den Kontext zurückzusetzen (kein Rollback), und auch die Angabe der globalen Variablen entfällt.

Die Transaktion kann durch `rollback` neu gestartet und durch das BPEL4WS-Tag `throw` abgebrochen werden. Das Commit ist implizit durch das Ende der Transaktion angegeben. Außerdem müssen für alle Transaktionen Fehlerbehandlungs- (`faultHandlers`) und Kompensations-Maßnahmen (`compensationHandler`) angegeben werden.[8]

Die Kompensations-Maßnahmen selbst werden wie technische Transaktionen gesichert, Anfang und Ende sind wie bei `transaction` implizit. Sie arbeiten auf den globalen Daten, die zum Ende der zugehörigen Transaktion gültig waren. Schlägt eine Kompensations-Maßnahme fehl, so kann sie mit `throw`[9] abgebrochen werden, wobei jedoch ein inkonsistenter Zustand entsteht, der manuell behoben werden muß.

```
<bpdl:transaction name="Name" inputVariable="Eingabe"
  outputVariable="Ausgabe" retryCount="Versuche">
    Definition der lokalen Variablen, Fehlerbehandlungs- und Kompensationsmechanismen
    wie in scope, gefolgt von BPEL4WS-Befehlen und zusätzlich
  <bpdl:rollback/>
</bpdl:transaction>
```

`createSavepoint, restoreSavepoint` Diese Tags ermöglichen das Setzen und Zurücklesen von Savepoints. Savepoints sichern den aktuellen Zustand des Workflow und ermöglichen es dem Workflow, jederzeit wieder zu einem zuvor gesicherten Zustand zurückzukehren.

```
<bpdl:createSavepoint name="Savepoint"/>
<bpdl:restoreSavepoint name="Savepoint"/>
```

Systemvariablen können in `assign`-Tags eingesetzt werden, um System-Plugins mit Daten über die Ausführung des Subworkflow zu versorgen. Bisher wurden dafür zwei Klassen von Systemvariablen definiert:

`namespace` enthält die Namen aller umgebenden Tags, beispielsweise also den aktuellen Subworkflow oder die aktuelle Transaktion.

`system` enthält Angaben über den Systemzustand, die augenblickliche Systemzeit oder die bereits verbrauchte Prozessorzeit.

Hier noch ein Beispiel für den Zugriff auf Systemvariablen:

```
<assign>
  <copy>
    <from bpdl:namespace="transaction"/>
    <to variable="Variablenname" part="Element"/>
  </copy>
  <copy>
    <from bpdl:system="systemtime"/>
    <to variable="Variablenname" part="Element"/>
  </copy>
</assign>
```

[8] Fehlerbehandlung und Kompensation sind in der gleichen Weise für das `scope`-Tag von BPEL4WS definiert.

[9] Die an das `throw` innerhalb einer Kompensation übergebene Nachricht darf nur Text- und Zahlenfelder enthalten, die hintereinander in das Log des Subworkflowcontrollers geschrieben werden und eine Fehlermeldung ergeben, die dem Administrator die Arbeit erleichtern soll.

`foreach` Durch dieses Tag können alle Elemente eines großen Feldes[10] nacheinander einzeln adressiert werden:

```
<bpdl:foreach loopVariable="Schleifenvariable" loopPart="Feld"
  fieldVariable="Variable mit Array-Elementen" fieldPart="Array-Element">
    Zugriff auf die Schleifenvariable.
</bpdl:foreach>
```

Um einen Überblick über den Einsatz von BPEL4WS in der BPDL/F zu ermöglichen, sollen hier den Konzepten des Subworkflowcontrollers in aller Kürze die verwendeten Sprach-Konstrukte zugeordnet werden:

Der gesamte Subworkflow wird durch einen `process` dargestellt. Darin werden zunächst über `partnerLinks` die Schnittstellen der verwendeten Aktivitäten, Plugins und Datenbankzugriffe sowie des Subworkflow selbst eingebunden, wobei die letztgenannte über `partners` identifiziert wird. In einer anschließenden `variables`-Definition erfolgt eine Zuweisung aller außerhalb der Transaktionen relevanten `messages` an Variablen.

In der darauf folgenden `sequence` wird der Ablauf des Subworkflow festgelegt. Dieser besteht immer aus dem Einlesen der Eingangs-SRWU durch `receive`, mindestens einer `transaction` und dem Versenden der Ausgangs-SRU über `reply`. Auf dieser Ebene sind ansonsten nur `invokes` von System-Plugins und Meldungen an den MFC erlaubt.

Innerhalb jeder Transaktion werden zunächst die lokalen Variablen in `variables` definiert, dann folgt ein `compensationHandler` und schließlich eine `sequence` mit den Aktionen der Transaktion. Auch der `compensationHandler` enthält eine `sequence` ähnlich der von Transaktionen mit den gleichen Möglichkeiten.

Innerhalb dieser `sequence` kann per `invoke` eine Aktivität aufgerufen, auf eine Datenbank zugegriffen oder eine Meldung an den MFC abgesetzt werden. Außerdem stehen die zuvor beschriebenen Möglichkeiten der Transaktionsverwaltung und des Savepointing zur Verfügung, ebenso wie die prozeduralen Steuerungselemente `switch`, `while` und foreach. Durch `assign` können zudem Variablen neue Werte zugewiesen und dabei auch einfache Berechnungen durchgeführt werden. Schließlich erlaubt `flow` das parallele Ausführen mehrerer Aktionen, synchronisiert über `links`.

Eine Anmerkung sei hier noch zur transaktionalen Semantik von Meldungen an den MFC eingefügt. Unter bestimmten Carriern (insbesondere Batch) sind nur solche Meldungen innerhalb einer Transaktion persistent, die direkt vor dem Ende der Transaktion abgesetzt werden. Vor einem Savepoint versendete Meldungen können nach einem Systemabsturz verloren sein. Es ist hierbei die Aufgabe des Business Process Designers, diesen Sachverhalt darzustellen.

2.2.4 Datenbankbeschreibungen

Für jede beteiligte Datenbank muß eine Datenbankbeschreibung vorhanden sein, die in der Business Process Definition Language for Databases, kurz BPDL/D, definiert wird. Dieser Subset von BDPL wurde direkt auf der Basis von XML und XML-Schema definiert und enthält die im folgenden aufgeführten Elemente. Die formale Spezifikation von BPDL/D findet sich in Unterabschnitt C.1.3 ab Seite 137, Beispiele dazu in Unterabschnitt C.2.3 ab Seite 154.

[10] In der Notation von BPEL4WS enthält jede Variable eine Struktur mit einem oder mehreren Elementen. Eines dieser Elemente muß in diesem Fall ein Feld sein, auf das `foreach` angewendet wird.

`database, table, field` Diese Tags erlauben es, eine Datenbank mit Tabellen und Feldern zu definieren, wie sie SQL-Datenbanken zugrunde liegt:

```
<bpdl:database name="DB-Name">
  <bpdl:table name="Tabellenname">
    <bpdl:field name="Feldname" type="XMLSchema-Typ"/>
  </bpdl:table>
</bpdl:database>
```

2.2.5 Parameterbeschreibungen

Für die Konfiguration des CC/P, der Master- und Subworkflowcontroller, der Batch-Carrier und der Datenzugriffsdienste wie auch für die Konfiguration und das Scheduling von Master- und Subworkflows existieren jeweils eigene Parameterbeschreibungen. Diese sind in der Business Process Definition Language for Settings, kurz BPDL/S verfaßt, deren Elemente im folgenden erläutert werden. Beispiele für diesen Subset von BPDL finden sich in Unterabschnitt C.2.4 ab Seite 154, die formale Spezifikation in Unterabschnitt C.1.4 ab Seite 138. Auch BPDL/S wurde direkt auf Basis von XML und XML-Schema definiert.

`settings, property, option` Mit diesen Tags und Attributen können einfache Parameterbeschreibungen vom Typ *Option=Wert* angelegt werden, in denen durch Properties definierte Variablen verwendet werden dürfen:

```
<bpdl:settings name="Parameter">
  <bpdl:property name="Variable" value="Wert"/>
  <bpdl:option name="Option" value="${Variable}Wert" type="XMLSchema-Typ"/>
</bpdl:settings>
```

2.3 Versionskontrolle

Jede BPDL-Beschreibung enthält in ihrem Top-Element Versions-Informationen, aufgeteilt in eine explizite Versionsnummer im `version`-Attribut, die zudem Teil des Namens der Beschreibung ist, und eine implizite Revisionsnummer im `revision`-Attribut.

Die explizite Versionsnummer muß bei jedem Zugriff auf eine Beschreibungsdatei angegeben werden, und muß somit auch in den verschiedenen SRWUs enthalten sein. Sie wird erhöht, wenn eine Änderung an einem Interface oder eine andere tiefgreifende Änderung vorgenommen wurde.

Durch diese Vorgehensweise ist sichergestellt, daß es nicht aus Versehen zu groben Inkompatibilitäten zwischen den verschiedenen Beschreibungen kommt. Außerdem können mehrere Versionen parallel eingesetzt und je nach Bedarf ausgewählt werden. So müssen zum Beispiel alle Beschreibungen, die auf ein geändertes Interface zugreifen sollen, ebenfalls angepaßt werden.

Die implizite Revisionsnummer wird erhöht, wenn Korrekturen oder Verbesserungen vorgenommen wurden, die das funktionale Verhalten des Geschäftsprozesses nicht verändern. Solchermaßen geänderte Beschreibungen werden beim nächsten Zugriff automatisch übernommen.

Nach einer entsprechenden Erweiterung der Komponenten der TE könnte die Revisionsnummer aber auch dazu verwendet werden, um zu einem festgelegten Zeitpunkt auf neue Versionen bestimmter Komponenten umzustellen, etwa, wenn ab dem 1. Januar ein neuer Steuersatz gültig ist. Diese Funktionalität ist bisher aber noch nicht spezifiziert.

Kapitel 3

Der Subworkflowcontroller

In der Diplomarbeit von Thomas Hornung [25] wurde ein Subworkflowcontroller (*SWFC*) für die Transaktionsmaschine entworfen, der die Basis für die vorliegende Diplomarbeit bildete. Der Aufbau des Subworkflowcontrollers wurde komplett überarbeitet und dabei stark vereinfacht, während seine Funktionalität im Sinne der Aufgabenstellung der vorliegenden Arbeit deutlich erweitert wurde.

In Abbildung 3.1 ist der „alte“ SWFC dem „neuen“ gegenübergestellt. Die hierbei verwendeten Abkürzungen werden im Laufe dieses Kapitels erklärt und finden sich außerdem im Abkürzungsverzeichnis ab Seite 163.

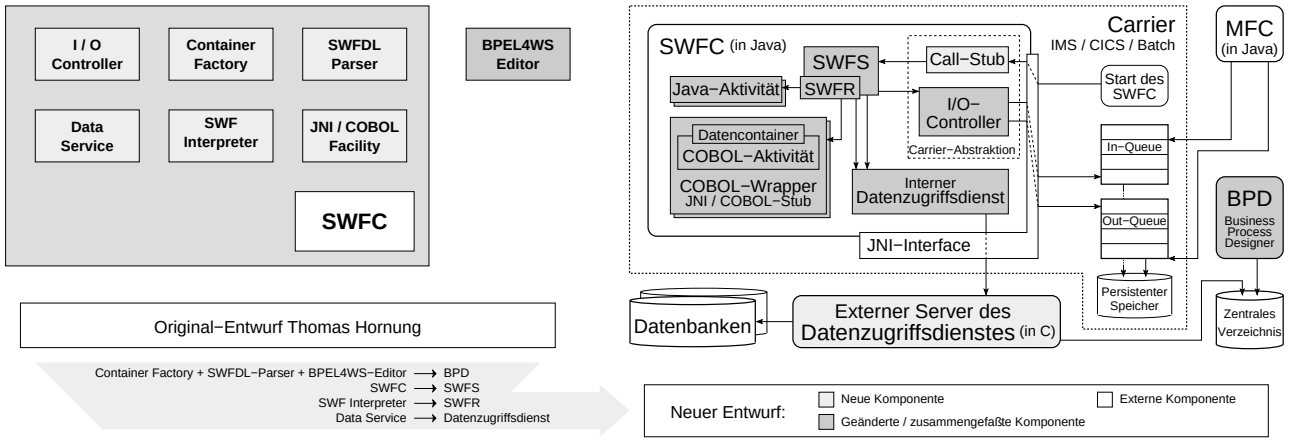


Abbildung 3.1: Änderungen am SWFC

Eine Anmerkung zu den verwendeten Begriffen: Da für den Subworkflowcontroller kein Unterschied in der Handhabung von technischen Aktivitäten und System-Plugins besteht, werden in den folgenden Kapiteln beide als „Aktivitäten“ bezeichnet und gemeinsam betrachtet, soweit nicht explizit zwischen ihnen unterschieden wird.

3.1 Lokale und globale Daten

Vorweg noch eine Anmerkung zur im folgenden verwendeten Terminologie bezüglich der Klassifikation der vom Subworkflowcontroller verwalteten Daten:

Lokale Daten sind alle Daten, deren Sichtbarkeit auf eine technische Transaktion innerhalb des Subworkflow beschränkt sind, also im wesentlichen temporäre Variablen Variablen und Ausgabedaten vorangegangener Aktivitäten.

Globale Daten sind Daten, die auch außerhalb einer technischen Transaktion ihre Gültigkeit behalten. Dazu gehören einerseits die Daten der SWF-SRWU, der verschiedenen SWF-SRUs und der temporären SRWUs, über die die globalen Daten von einer technischen Transaktion an die nächste weitergegeben werden (siehe Unterabschnitt 2.2.3 ab Seite 17), und andererseits die Daten, die aus den verschiedenen Datenbanken gelesen wurden oder dorthin geschrieben werden.

3.2 Konzept des Subworkflowcontrollers

Im Konzept der Transaktionsmaschine werden technische Aktivitäten auf die fachliche Logik und somit auf reine Datenverarbeitung reduziert. Dadurch vereinfacht sich das Programmiermodell der technischen Aktivitäten, und der gesamte Geschäftsprozeß wird weniger fehleranfällig.

Die Aufgabe des Subworkflowcontrollers (*SWFC*) besteht nun darin, technische Aktivitäten transaktionsgeschützt in einer festgelegten Reihenfolge auszuführen, ihnen ihre Eingabedaten zur Verfügung zu stellen und ihre Ausgabedaten zu übernehmen.[1] Außerdem überwacht der SWFC die Ausführung der technischen Aktivitäten und ergreift im Fehlerfall die nötigen Maßnahmen.

Zu Beginn jedes Subworkflow erhält der Subworkflowcontroller vom Carrier (IMS / CICS im Online-Betrieb und Batch im Offline-Betrieb) einen neuen Auftrag in Form einer Service Request Work Unit für den Subworkflow (*SWF-SRWU*), die neben der Kennung des auszuführenden Subworkflow (ID, Version) und einer eindeutigen Auftrags-ID (siehe Abschnitt 1.4 ab Seite 4) die Eingabe-Daten für den Subworkflow enthält.

In einem Subworkflow können nacheinander mehrere technische Transaktionen gestartet und festgeschrieben (committed) oder auch zurückgesetzt (rolled back), abgebrochen (aborted) und rückgängig gemacht (compensated) werden. In deren Kontext sind die technischen Aktivitäten eingebunden. Die von den letzteren benötigten Daten werden vom SWFC aus der SWF-SRWU, den Ergebnissen vorangegangener technischer Aktivitäten und, mit Hilfe des Datenzugriffsdienstes, aus verschiedenen Datenbanken zusammengetragen. Nach der Ausführung der technischen Aktivitäten werden deren Ergebnisse ausgewertet. Die nötigen Datenbankänderungen werden an den Datenzugriffsdienst übergeben, der sie bis zum Ende der Transaktion zwischenspeichert und dann in die Datenbanken schreibt.

Innerhalb jeder technischen Transaktion kann zudem über Savepoints die Arbeit vorangegangener Aktivitäten abgesichert werden, ohne den transaktionalen Kontext zu verlassen. Dies kann zu einer deutlichen Beschleunigung des Subworkflow führen, da das aufwendige Festschreiben von Änderungen in den Datenbanken nicht nach jeder Aktivität stattfinden muß.

[1] Durch die Flexibilität der Beschreibungssprache ist es dem SWFC aber auch möglich, einfache arithmetische Ausdrücke selbst auszuwerten, wodurch in einigen Fällen der Overhead beim Aufrufen einer Aktivität eingespart werden kann.

Außerdem ermöglichen Savepoints ein schnelles Rücksetzen des Subworkflow innerhalb einer Transaktion. Durch das Setzen eines impliziten Savepoint zu Beginn jeder Transaktion kann zudem das Zurücksetzen einer Transaktion effizienter durchgeführt werden. Das Savepointing wird über den Datenzugriffsdienst abgewickelt.

An beliebiger Stelle des Subworkflow (also auch außerhalb der technischen Transaktionen) können System-Plugins aufgerufen werden, deren Eingabedaten aus den globalen und lokalen Daten des Subworkflow und aus Systemvariablen (siehe Unterabschnitt 2.2.3 ab Seite 17) zusammengestellt werden können. Außerhalb einer technischen Transaktion haben System-Plugins ausschließlich Zugriff auf die Daten der SRWU und auf Systemvariablen, da weder Datenbankdaten noch lokale Daten verfügbar sind.

Der gesamte Subworkflow kann jederzeit mit einer Fehler-SWF-SRU abgebrochen werden, wobei eine gerade aktive technische Transaktion zurückgesetzt wird, und bereits abgeschlossene Transaktionen über Kompensations-Workflows rückgängig gemacht werden. Wird der Subworkflow erfolgreich beendet, so gibt er dem Carrier eine Ausgabe-SWF-SRU zurück. Während der Ausführung des Subworkflow können außerdem an beliebigen Stellen Nachrichten in Form von Meldungs-SWF-SRUs durch den Carrier an den Masterflowcontroller übergeben werden.

Mit Hilfe des Business Process Designers (*BPD*) werden alle Angaben zur Ausführung des Subworkflow in einer Workflow-Beschreibung (*BPDL/F*, siehe Unterabschnitt 2.2.3 ab Seite 17) festgelegt, während die SWF-SRWU und die verschiedenen SWF-SRUs in der Interface-Beschreibung des Subworkflow (in BPDL/I, siehe Unterabschnitt 2.2.2 ab Seite 15) festgelegt werden. Aus diesen Beschreibungen wird vom BPD die Subworkflow-Runtime generiert, die den Ablauf des Subworkflow steuert.

3.3 Laufzeitumgebung des Subworkflowcontrollers

Der Subworkflowcontroller läuft als Java-Programm unter der „Persistent Reusable Java Virtual Machine“ (*PRJVM*) [34], die vom verwendeten Carrier (IMS / CICS oder Batch) gestartet wird. Die PRJVM ist eine IBM-eigene Weiterentwicklung einer JVM mit JIT-Compiler speziell für die IBM-Mainframes, die einige Besonderheiten aufweist. Die wichtigste davon ist die Möglichkeit, die JVM „resetten“ zu können, um so eine transaktionale Isolation zu ermöglichen, ohne die JVM für jede neue Transaktion[2] neu starten zu müssen. Das erhöht die Performance von Transaktionen teilweise um einige Größenordnungen, wie Marc Beyerle in [27] nachweisen konnte.

Für die Verwendung der PRJVM werden die Klassen eines Java-Programms in sogenannte „trusted middleware“-Klassen und Applikationsklassen („sharable application classes“ und „transient classes“) aufgeteilt. Die Middleware-Klassen dürfen keine Verweise auf Objekte der Applikationsklassen speichern und müssen sich bei einem Reset der PRJVM selbst reinitialisieren, wobei alle für die letzte Transaktion spezifischen Einstellungen gelöscht werden sollten.[3] Die Applikationsklassen werden unterschiedlich gehandhabt, je nachdem, ob sie „sharable application“ Klassen oder transiente Klassen sind.[4]

[2] Eine Transaktion der PRJVM entsprecht der Abarbeitung eines Auftrags durch den SWFC und darf nicht mit den technischen Transaktionen des Subworkflow verwechselt werden.

[3] Die Objekte der Middleware-Klassen werden erst bei einer *Garbage Collection* freigegeben, die jedoch nicht zu häufig durchgeführt werden sollte, da sie sehr aufwendig ist. Die Objekte der Applikationsklassen hingegen werden bei jedem Reset aus dem Speicher entfernt.

[4] „sharable application“ Klassen verbleiben bis zum Neustart der JVM im Speicher und können so sehr schnell wieder aktiviert werden, transiente Klassen werden beim Neustart gelöscht.

Durch die beschriebene Vorgehensweise stellt die PRJVM sicher, daß eine nachfolgende Transaktion keine Überreste einer vorangegangenen Transaktion mehr vorfindet und so die Isolation der einzelnen Transaktionen gesichert ist, natürlich unter der Voraussetzung, daß die Middleware-Klassen diese Isolation nicht unterwandern.

Der SWFC nutzt die PRJVM in der Weise, daß technische Aktivitäten und System-Plugins als Applikationsklassen geladen werden (abhängig vom `persistent`-Attribut in ihrer Definition als „sharable application"-Klassen oder als transiente Klassen), während die Subworkflow-Runtime (*SWFR*) und alle anderen Klassen als Middleware-Klassen geladen werden. Dadurch bleiben die System-Klassen des SWFC aktiv, so daß der Zugriff durch nachfolgende Subworkflows verzögerungsfrei erfolgen kann.

Nach der Abarbeitung jedes Subworkflow wird zudem ein Reset der PRJVM durchgeführt, bei dem sämtliche technischen Aktivitäten und System-Plugins sowie die SWFR für den SWFC unzugänglich werden, und die anderen Komponenten des SWFC auf definierte Zustände zurückgesetzt werden. Dies garantiert eine weitestgehende Isolierung der einzelnen Subworkflows.

3.4 Komponenten des Subworkflowcontrollers

In Abbildung 3.2 findet sich eine Übersicht über den Aufbau des Subworkflowcontrollers. Die Pfeile in dieser Abbildung entsprechen, soweit nicht anders angegeben, einer „greift zu auf"-Relation. Die einzelnen Komponenten des SWFC werden im folgenden beschrieben, wobei jeweils auch erwähnt wird, was sich gegenüber der Arbeit von Thomas Hornung geändert hat.

3.4.1 Der Call-Stub

Der Call-Stub wird aufgerufen, wenn die PRJVM gestartet wurde. Er überprüft zunächst, ob der Aufruf legitim ist[5], liest dann die Konfigurationsdaten des Subworkflowcontrollers aus der entsprechenden BPDL-Parameter-Beschreibung ein[6], initialisiert den benötigten I/O-Controller und übergibt schließlich die Kontrolle an den Subworkflow-Überwacher (*Subworkflow Supervisor, SWFS*). Wenn der SWFC beendet werden soll, erhält der Stub wieder die Kontrolle, gibt noch belegte Ressourcen frei und kehrt zum aufrufenden Carrier zurück.

Diese Komponente ist neu hinzugekommen. Sie stellt zusammen mit dem I/O-Controller eine Abstraktion gegenüber dem Carrier dar, die die anderen Komponenten des SWFC unabhängig vom verwendeten Carrier macht. Angaben dazu, in welcher Weise der Call-Stub und der I/O-Controller an die verschiedenen Carrier angepaßt werden, finden sich in Kapitel 6 ab Seite 55.

[5] Da der Call-Stub auch von Java-Aktivitäten aufgerufen werden könnte, muß er sich gegen unbefugte Zugriffe schützen. Dies geschieht, indem er bei jedem Aufruf mit Hilfe einer abgefangenen Exception den Aufrufer ermittelt. Alternativ könnte auch beim ersten Aufruf ein Flag gesetzt werden, das bei jedem Aufruf überprüft wird und damit jeden weiteren Aufruf sofort abbrechen läßt.

[6] Die Position der BPDL-Parameter-Beschreibung für den Subworkflowcontroller muß, abhängig vom Carrier, vor der Compilierung des Call-Stub festgelegt werden oder diesem beim Aufruf übergeben werden. Die Datei muß an einer Stelle abgelegt sein, auf die der Call-Stub direkt Zugriff hat, da sich alle Angaben zur vorhandenen Infrastruktur erst in dieser Datei finden. Insbesondere ist dort auch die Position des externen Servers des Datenzugriffsdienstes (siehe Abschnitt 5.4 ab Seite 41 und Abschnitt 5.6 ab Seite 45) angegeben.

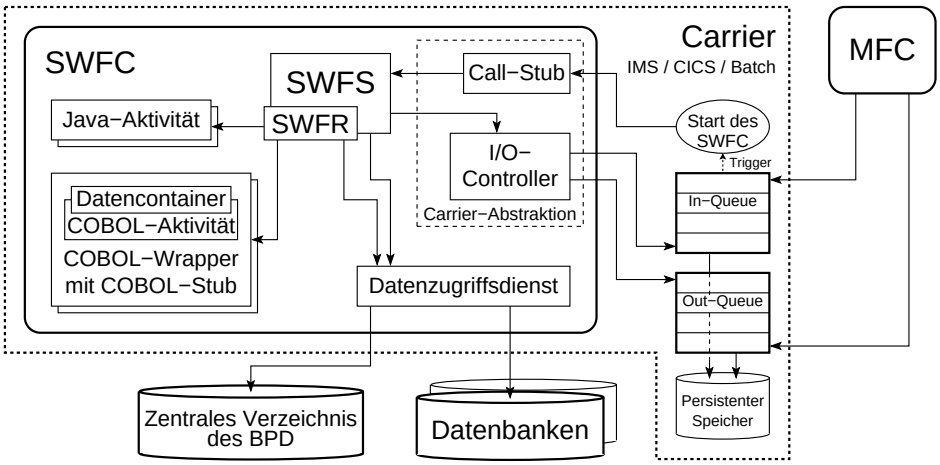


Abbildung 3.2: Aufbau des SWFC

3.4.2 Der Subworkflow-Überwacher

Der Subworkflow-Überwacher (*Subworkflow Supervisor, SWFS*) initialisiert zunächst den Datenzugriffsdienst und geht dann in eine Schleife, in der er über den I/O-Controller neue SWF-SRWUs einliest. Aus diesen ermittelt er die Kennung des Subworkflow (ID, Version) und lädt mit Hilfe des Datenzugriffsdienstes die passende Subworkflow-Runtime (*SWFR*).

Parallel dazu wird der Datenzugriffsdienst beauftragt, nach alten Savepoints dieses Subworkflow zu suchen. Sobald die SWFR geladen und die Suche abgeschlossen ist, wird die Kontrolle an die SWFR übergeben. Als Parameter erhält diese dabei die SWF-SRWU sowie einen eventuell gefundenen alten Savepoint.

Nach der Rückkehr von der SWFR[7] wird der letzte Savepoint des Auftrags über den Datenzugriffsdienst gelöscht, dem Carrier über den I/O-Controller der Abschluß des Subworkflow mitgeteilt und die SWFR dereferenziert, so daß sie bei der nächsten Garbage-Collection entfernt werden kann. Danach wird beim Carrier nach weiteren SWF-SRWUs geschaut. Kann keine neue SWF-SRWU mehr gelesen werden, so übergibt der SWFS die Kontrolle an den Call-Stub zurück, der den gesamten Subworkflowcontroller beendet.

Der SWF-Überwacher entspricht in etwa der „SWFC“ genannten Komponente aus der Definition des Subworkflowcontrollers von Thomas Hornung.

3.4.3 Die Subworkflow-Runtime

Der Ablauf des Subworkflow wird von einem deterministischen endlichen Automaten (*DEA* oder *FSM* von englisch *Finite State Machine*) gesteuert, der Subworkflow-Runtime (*SWFR*), die auch alle lokalen und globalen[8] Daten des Subworkflow verwaltet. Die SWFR ist eine eigenständige Java-Klasse, welche vom BPD aus den BPDL-Beschreibungen des Subworkflow und der Aktivitäten automatisch generiert wird. Sie ist die compilierte Version des „SWFDL-Interpreters“ aus der alten Definition des Subworkflowcontrollers.

[7] Die SWFR hat vor ihrem Ende die Ausgabe-SWF-SRU bereits an den Carrier übergeben.

[8] Auch die aus den Datenbanken gelesenen Daten werden in der Subworkflow-Beschreibung festen Variablen zugeteilt und von der SWFR verwaltet, solange diese Variablen ihre Gültigkeit behalten.

Dabei wird der Subworkflow an den Savepoints aufgetrennt und in ein einziges Switch-Statement eingebettet, so daß jedes Case-Statement an einem Savepoint beginnt. Über Veränderungen der lokalen und globalen Daten des Subworkflow führt die SWFR mit Hilfe eines Flagfeldes Buch, um ein effizientes Savepointing zu ermöglichen. Der Zustand des Subworkflow besteht also aus allen von der SWFR verwalteten Daten, dem Flagfeld und der Nummer des aktuell ausgeführten Case-Statement.

Beim Start der SWFR wird ihr ein alter Savepoint übergeben, falls der Subworkflow durch einen Fehler des Systems unterbrochen wurde. Abhängig davon startet die SWFR den Subworkflow neu, wobei die SWF-SRWU durch den impliziten Savepoint zu Beginn der ersten technischen Transaktion des Subworkflow gesichert wird, oder setzt ihn bei dem übergebenen Savepoint fort. Nach dem (regulären) Ende des Subworkflow wird die Ausgabe-SWF-SRU an den Carrier übergeben und die Kontrolle an den SWFS zurückgegeben.

Während der Abarbeitung des Subworkflow werden von der SWFR die folgenden Aktionen ausgeführt:

Datenbankzugriffe: Werden Daten aus einer Datenbank benötigt, so werden die in der BPDL-Beschreibung dieses Datenbankzugriffs abgelegten Informationen an den Datenzugriffsdienst übergeben, der daraufhin die angeforderten Daten zurückliefert. Dieser Vorgang kann auch asynchron erfolgen, wobei die Daten zunächst nur angefordert und erst später abgeholt werden.

Sollen veränderte Daten in eine Datenbank zurückgeschrieben werden, so werden die Daten mitsamt den entsprechenden Informationen aus der BPDL-Beschreibung an den Datenzugriffsdienst übergeben, der sie am Ende der aktuellen Transaktion in das betroffene Datenbanksystem überträgt und persistiert.

Mengenorientierte Datenbankzugriffe sind ebenso möglich wie direkte Zugriffe auf einzelne Datenbankfelder. Wenn die Datenbankabfrage mehrere Felder zurückliefern soll, so muß dafür in der BPDL-Beschreibung des Datenbankzugriffs eine Feldvariable vorgesehen sein. Diese kann später direkt einer Aktivität übergeben werden, oder in einer `foreach`-Schleife aufgelöst und elementweise verarbeitet werden. Ein Zurückschreiben ganzer Felder von Daten ist in SQL prinzipbedingt ausgeschlossen, so daß hier nur der Weg über die `foreach`-Schleife bleibt.

Aufruf von Aktivitäten: Zu Beginn eines Subworkflow wird der Datenzugriffsdienst beauftragt, die Programmobjekte aller Aktivitäten über ihre Kennung (ID, Version) zu lokalisieren und zu laden. Dies geschieht asynchron im Hintergrund, so daß die SWFR sofort mit der nächsten Tätigkeit fortfahren kann. Vor der Ausführung von Aktivitäten muß dann nur noch beim Datenzugriffsdienst nachgefragt werden, ob die Aktivität bereits geladen wurde.

Java-Aktivitäten (technische Aktivitäten und System-Plugins) werden direkt von der Runtime gestartet, die ihnen auch in einem aus der BPDL-Beschreibung der Aktivität generierten Objekt ihre Eingabedaten übergibt. Ihre Ergebnisse liefern sie in einem weiteren aus ihrer BPDL-Beschreibung generierten Objekt an die SWFR zurück, und im Fehlerfall können sie eine Exception werfen. Die SWFR sichert die Ergebnisdaten, wertet sie aus und ergreift im Fehlerfall die in der BPDL-Beschreibung des Workflow dafür vorgesehenen Maßnahmen.

Für den Aufruf von COBOL-Aktivitäten übergibt die SWFR ein Array mit den Eingabedaten und den Pfad des Programm-Objekts der Aktivität dem COBOL-Wrapper, von dem sie auch wieder die Ausgabedaten der Aktivität erhält. Ansonsten verhalten sich COBOL-Aktivitäten nicht anders als ihre Java-Pendants.

Weitere Angaben zu den Aktivitäten finden sich in Unterabschnitt 3.4.6 und Unterabschnitt 3.4.7 auf Seite 29.

Savepoints: Soll ein Savepoint geschrieben werden, so übergibt die SWFR dem Datenzugriffsdienst alle seit dem letzten Savepoint veränderten Daten und die Nummer des als nächstes auszuführenden Case-Statements. Nachdem der Savepoint erfolgreich geschrieben wurde, wird das Flagfeld zurückgesetzt.

Soll auf einen früheren Savepoint zurückgesetzt werden, so übergibt die SWFR dem Datenzugriffsdienst die Nummer des angeforderten Savepoint und die Angaben aus dem Flag-Feld darüber, welche Daten seit dem letzten Savepoint verändert wurden. Der Datenzugriffsdienst ermittelt daraufhin den zum Zeitpunkt des angeforderten Savepoint gültigen Zustand aller Daten, die sich seit diesem Zeitpunkt verändert haben, löscht alle nachfolgenden Transaktionsdaten und gibt die ermittelten Daten an den SWFR zurück. Dieser speichert die übergebenen Daten, löscht das Flag-Feld und setzt seine Arbeit bei dem durch die Savepoint-Nummer bestimmten Case-Statement fort.

Transaktionen: Wird eine technische Transaktion des Subworkflow gestartet, so wird dies dem Datenzugriffsdienst unter Angabe der Eingangsdaten dieser Transaktion mitgeteilt, der dabei einen (impliziten) Savepoint schreibt. Danach wird das Flagfeld initialisiert und die Transaktion ausgeführt.

Soll eine laufende Transaktion zurückgesetzt werden (*Rollback*), so wird auch dies dem Datenzugriffsdienst mitgeteilt, der daraufhin alle Savepoints der Transaktion (außer dem ersten) und alle gespeicherten Datenbank-Updates löscht und die Daten des ersten Savepoint zurückgibt. Der Subworkflow wird dann am Anfang der zurückgesetzten Transaktion fortgesetzt, nachdem alle Variablen und das Flagfeld gelöscht und die zurückgegebenen Variablen eingetragen wurden.

Am (regulären) Ende einer technischen Transaktion werden dem Datenzugriffsdienst die Ausgabedaten der Transaktion übergeben, woraufhin dieser wiederum einen impliziten Savepoint schreibt,[9] alle zwischengespeicherten Datenbank-Updates ausführt (*Commit*) und alle Savepoints der beendeten Transaktion (außer dem letzten) löscht. Außerdem wird dem I/O-Controller das Ende der Transaktion mitgeteilt. Die SWFR fährt dann mit der nächsten Aktion des Workflow fort.

[9] Dieser Savepoint am Ende der Transaktion ermöglicht es, auf die Ergebnisse der Transaktion zurückzusetzen, falls vor Beginn der nächsten Transaktion ein Fehler auftritt. Insbesondere sind so die Ergebnisse des gesamten Workflow gesichert, falls die Rückgabe der Ergebnisse an den Carrier fehlschlägt. Außerdem greifen Kompensationsmaßnahmen auf die Daten dieses Savepoint zurück, um die Aktionen der betroffenen Transaktion revidieren zu können.

Meldungen: Nachrichten an den Masterflowcontroller werden mit Hilfe des I/O-Controllers in Form von Meldungs-SWF-SRUs an den Carrier übergeben.

Fehler: Der Abbruch des Subworkflow (ausgelöst durch ein `throw`-Statement in der BPDL-Beschreibung) wird dem Datenzugriffsdienst mitgeteilt, der daraufhin alle Savepoints der laufenden Transaktion und alle Datenbank-Updates löscht. Anschließend führt die SWFR die Kompensationsmaßnahmen aller vorangegangenen Transaktionen in umgekehrter Reihenfolge aus. Wird eine Kompensationsmaßnahme abgebrochen, so schlägt der komplette Subworkflow fehl, und dabei aufgetretene Inkonsistenzen in den Datenbanken müssen von Hand aufgelöst werden.

3.4.4 Der I/O-Controller

Der I/O-Controller ist der andere Teil der Abstraktion, die den Subworkflowcontroller vom verwendeten Carrier unabhängig macht. Er holt neue SWF-SRWUs beim Carrier ab, schickt durch Übergabe an den Carrier Meldungs-SWF-SRUs an den MFC und liefert das Ergebnis des Subworkflow in Form einer Ausgabe-SWF-SRU oder Fehler-SWF-SRU über den Carrier zurück.

Außerdem wird er durch den SWFS vom Ende des Subworkflow und durch die SWFR vom Ende oder Abbruch jeder Transaktion benachrichtigt. Am regulären Ende einer Transaktion teilt er dem Carrier mit, daß dieser alle gesammelten Meldungs-SRUs weiterleitet, falls dieser das nicht unverzüglich erledigt. Beim Abbruch der Transaktion, oder wenn die Transaktion zurückgesetzt wird, werden noch nicht weitergeleitete Meldungs-SRUs verworfen.[10] Am Ende des Subworkflow wird mit Hilfe des Carriers ein Reset der PRJVM ausgelöst.

Der I/O-Controller wurde gegenüber der alten Definition verschlankt: Er enthält nun nur noch die umgebungsabhängigen Teile des SWFC, die nicht bereits im Call-Stub integriert sind. Näheres zu der Abstraktion, die für die Einbindung des SWFC in die Carrier IMS und CICS sowie Batch notwendig ist, findet sich in Kapitel 6 ab Seite 55.

3.4.5 Der Datenzugriffsdienst

Der Datenzugriffsdienst (*englisch Data Service, kurz DS*) stellt erstens die Verbindung des SWFC zu externen Datenbanken her, wobei er gelesene Daten in einem Cache verwaltet und Schreibzugriffe bis zum Ende einer Transaktion zwischenspeichert. Zweitens verwaltet er die Daten, die durch das Savepointing und die transaktionale Sicherung des Subworkflow anfallen, und drittens lädt er von anderen Komponenten des SWFC benötigte Programm-Objekte aus dem zentralen Verzeichnis des BPD.

An dieser Komponente hängt ein Großteil der Performance des gesamten Subworkflowcontrollers (und damit auch der Transaktionsmaschine), was in der vorangegangenen Arbeit zum Subworkflowcontroller von Thomas Hornung [25] deutlich sichtbar wurde.

[10] Das Rücksetzen auf einen früheren Savepoint wird vom I/O-Controller nicht registriert, da der Aufwand dafür, auch in diesem Fall einige, aber nur bestimmte Meldungen zu verwerfen, einfach zu groß wäre.

Die Funktionalität des Datenzugriffsdienstes wurde in einen externen Server ausgelagert, um einen maximalen Synergieeffekt durch den Lese-Cache zu erreichen, der von allen in der gleichen Rechnerumgebung laufenden SWFCs geteilt wird. Im Subworkflowcontroller selbst bleibt nur ein Interface zurück, das den Aufruf der Funktionen des DS für die anderen Komponenten des SWFC transparent macht.

Der Datenzugriffsdienst war in der alten Definition des SWFC nur teilweise entworfen worden und wurde auch nur rudimentär implementiert. Er wird ausführlich in Kapitel 5 ab Seite 35 beschrieben.

3.4.6 Java-Aktivitäten

Java-Aktivitäten werden direkt als Applikationsklassen unter der PRJVM ausgeführt, nachdem sie über ihre Kennung mit Hilfe des Datenzugriffsdienstes in den „sharable application classpath“ bzw. den normalen „classpath“ der JVM geladen wurden (siehe dazu Abschnitt 3.3 ab Seite 23). Ihnen wird als Aufrufparameter ein Objekt mit allen Eingabedaten übergeben, das zuvor von der SWFR entsprechend den Anweisungen in der BPDL-Beschreibung des Subworkflow mit Daten gefüllt wurde. Als Rückgabewert liefern die Java-Aktivitäten entweder ein weiteres Objekt mit ihren Ausgabedaten, oder sie werfen eine Exception, deren Format durch den `fault`-Eintrag in ihrer Interface-Beschreibung definiert ist.

Der „Message Container“, der im bisherigen SWFC lokale wie globale Daten verwaltet hat, fällt weg. Seine Funktion ist Teil der Subworkflow-Runtime. Die bisher als „lokale Sicht“ bezeichnete Funktionalität wurde durch die Ein- und Ausgabeobjekte der Aktivitäten ersetzt, wodurch eine klare Trennung in lesbare und schreibbare Daten erreicht wird. Das Programmiermodell von Java-Aktivitäten wird in Abschnitt 4.1 ab Seite 31 detailliert erläutert.

3.4.7 COBOL-Aktivitäten und der COBOL-Wrapper

Für die Ausführung von COBOL-Aktivitäten wird eine spezielle Java-Klasse aufgerufen, der COBOL-Wrapper, dem neben einem Array mit den Eingabedaten in Form von Java-Objekten[11] auch ein Array mit den Typen der Ein- und Ausgabedaten der Aktivität sowie der Pfad übergeben wird, unter dem das Programm-Objekt der COBOL-Aktivität abgelegt wurde.

Der COBOL-Wrapper übergibt alle Parameter über JNI an den COBOL-Stub, eine C-Routine, die die Eingabedaten mit Hilfe der mitgelieferten Typ-Beschreibung in ein COBOL-kompatibles Format umwandelt und schließlich die COBOL-Aktivität mit diesen Daten aufruft. Nach Abschluß der Aktivität konvertiert der COBOL-Stub deren Ausgabedaten mit Hilfe der Typ-Beschreibung wieder in ein Objekt-Array und gibt dieses an den COBOL-Wrapper zurück, der es wiederum an die SWFR weiterleitet.

Der COBOL-Wrapper entspricht der „COBOL / JNI Facility“ des Entwurfs von Thomas Hornung, und die „C-Bridge“ wurde durch den COBOL-Stub ersetzt. Die „COBOL API“ genannte Funktionalität des „Message Containers“ wurde in den BPD integriert, der entsprechenden Code in der SWF-Runtime ablegt. Nähere Angaben zum Programmiermodell von COBOL-Aktivitäten finden sich in Abschnitt 4.2 ab Seite 33.

[11] Zu jedem primitiven Datentyp von Java existiert auch eine Klasse, in der dieser Datentyp gekapselt wird. Diese Möglichkeit wird auch von den Container-Klassen von Java verwendet.

3.5 Fehlerbehandlung im Subworkflowcontroller

Tritt während der Ausführung einer Komponente des Subworkflowcontrollers ein Fehler auf, so wird zunächst überprüft, ob eine (eventuell mehrfache) Wiederholung der fehlgeschlagenen Aktion sinnvoll wäre. Die maximale Anzahl von Wiederholungen ist in der BPDL-Parameter-Beschreibung des Subworkflowcontrollers angegeben.

Konnte (oder sollte) der Fehler nicht durch Wiederholungen behoben werden, so wird eine Java-Exception geworfen, die es der aufrufenden Komponente erlaubt, den Fehler zu berücksichtigen und eventuell zu kompensieren. Tut sie dies nicht, so wird der Fehler automatisch der Aufrufkette entlang zurück bis zum Subworkflow-Überwacher propagiert, der den Subworkflow daraufhin abbricht.

Fehler, die sich auf die in der Subworkflow-Beschreibung abgelegten Anweisungen zurückführen lassen, wie etwa ein fehlgeschlagener Datenbankzugriff, können (und sollten) durch diese Beschreibung auch abgefangen werden. Geschieht dies nicht, führt ein solcher Fehler zum Abbruch des gesamten Subworkflow durch den Subworkflow-Überwacher.

Wenn der Subworkflow aufgrund eines Fehlers abgebrochen werden muß, so macht die Subworkflow-Runtime bereits erfolgreich abgeschlossene Transaktionen über die zugehörigen Kompensationsmaßnahmen wieder rückgängig. Tritt dabei wieder ein Fehler auf, so kann eine Inkonsistenz der Datenbanken entstehen, die von einem Operator aufgelöst werden muß. Ansonsten wird nach Abschluß der Kompensationsmaßnahmen die Fehler-SWF-SRU des Subworkflow an den Carrier übergeben und der Subworkflow beendet.

Kapitel 4

Das Programmiermodell der Aktivitäten und Plugins

Ein zentrales Ziel der Transaktionsmaschine (TE) ist es, durch eine gegenüber anderen Anwendungssystemen reduzierte Komplexität bei der Anwendungsprogrammierung die Fehleranfälligkeit zu verringern. Eines der wichtigsten Konzepte hierbei ist es, die Freiheiten des Anwenders teilweise einzuschränken, und dabei trotzdem eine genügend große Flexibilität für die Implementierung beliebiger Geschäftsprozesse zu erhalten.

Ein wichtiger Schritt auf dem Weg zu einem sicheren System ist die Beschränkung der technischen Aktivitäten auf die reine Fachlogik, und auch System-Plugins sollten ausschließlich der Datenverarbeitung dienen, da diese beiden Komponenten der TE als einzige von Hand geschrieben werden und somit besonders fehleranfällig sind. Der Subworkflowcontroller (*SWFC*) kümmert sich um alles andere: Er ermöglicht den technischen Aktivitäten und System-Plugins (die im folgenden wieder gemeinsam als Aktivitäten bezeichnet werden) einerseits den Zugriff auf die von diesen benötigten Daten (und nur auf diese Daten) und sorgt andererseits für deren transaktionale Sicherung.

In der Interface-Beschreibung der Aktivitäten wird festgelegt, welche Ein- und Ausgabedaten sie benötigen (siehe Unterabschnitt 2.2.2 ab Seite 15). Diese Beschreibung ist für eine Java- oder COBOL-Aktivität identisch, abgesehen von der Angabe des auszuführenden Programms und der Möglichkeit, einen Fehler über eine durch den `fault`-Eintrag definierte Exception anzuzeigen.

Aus der Sicht der Aktivitäten liefert der SWFC die Eingabedaten und übernimmt die Ausgabedaten. Außerdem kann er der Subworkflow-Beschreibung entnehmen, wie er einen eventuell aufgetretenen Fehler der Aktivität erkennt, und was er in diesem Fall unternehmen muß.

4.1 Java-Aktivitäten

Technische Aktivitäten und System-Plugins, die in der Sprache Java entwickelt wurden, könnten theoretisch direkt auf alle Daten des ebenfalls in Java geschriebenen Subworkflowcontrollers zugreifen. Um das zu verhindern, müssen die Java-eigenen Möglichkeiten genutzt werden, um eine geschützte Klassenhierarchie für den SWFC aufzubauen. Die einzigen Klassen, auf die eine Java-Aktivität zugreifen darf, sind ihre Ein- und Ausgabedaten.

Um in einer Java-Applikation einzelnen Klassen den Zugriff auf ein System von Klassen zu verwehren, die untereinander Zugriff auf die jeweiligen Methoden der anderen Klassen benötigen, werden die zu isolierenden Klassen in ein eigenes Paket verlegt, und die Klassen des Systems erhalten keinerlei Sichtbarkeits-Attribute, also weder `public` noch `private` noch `protected`. Dadurch sind sie „Paket-Sichtbar“.

Diese Vorgehensweise wird auch für die Isolierung der Aktivitäten vom SWFC eingesetzt, wobei der SWFC und die Subworkflow-Runtime dem Paket „swfc“ angehören, während die Applikationsklassen dem Paket „app“ zugeordnet werden.[1] Das kann natürlich nicht verhindern, daß Aktivitäten sich über die Klassen der Standard-Java-Bibliotheken Zutritt zur Außenwelt verschaffen. Ein solches Verhalten kann nur unterbunden werden, indem die Aktivitäten vor der Compilierung auf externe Referenzen und Paket-Definitionen untersucht werden.

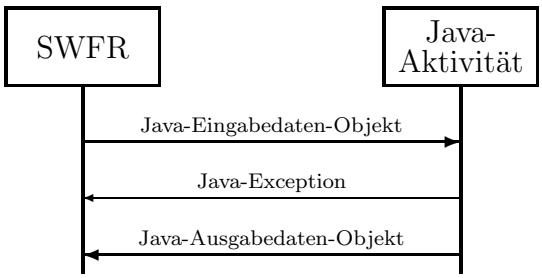


Abbildung 4.1: Aufruf einer Java-Aktivität

In Abbildung 4.1 ist der Aufruf einer Java-Aktivität durch die Subworkflow-Runtime dargestellt. Den Java-Aktivitäten stehen nur die Daten zur Verfügung, die ihnen in einem Eingabedaten-Objekt von der Subworkflow-Runtime (*SWFR*) übergeben wurden. Ihre Ergebnisse übergeben sie der SWFR über ein Ausgabedaten-Objekt. Einen Fehler können Java-Aktivitäten außerdem anzeigen, indem sie eine Exception mit einem Fehler-Objekt werfen. Das Format dieser drei Datenobjekte wird durch die Angaben zu Eingabe-, Ausgabe- und Fehlernachricht der Aktivität in deren BPDL-Interface-Beschreibung [2] (siehe Unterabschnitt C.1.1 ab Seite 107) definiert, der Name der Exception entstammt dem `fault`-Eintrag.

Die Eingabe- und Ausgabe-Objekte und die Exception für die Fehlerbehandlung sind auch in den vom Business Process Designer (*BPD*) generierten Interfaces und Skeletons angegeben. Die Exception wird zur Laufzeit von der Subworkflow-Runtime entsprechend den Angaben in der BPDL-Workflow-Beschreibung, aus der die SWFR generiert wurde, abgefangen und behandelt. Das Werfen anderer Exceptions ist der Aktivität nicht möglich[3], das Werfen von Errors nicht erlaubt[4].

[1] Eine Ausnahme bei der Sichtbarkeit bildet der Call-Stub, der eine als `public` deklarierte Methode zum Einsprung von außen benötigt und deshalb selbst `public` sein muß. Hier wird der Zugriff dadurch gesichert, daß der Stub die aufrufende Instanz überprüft und nötigenfalls eine „illegal access exception“ wirft.

[2] Schnittstellenbeschreibung

[3] Für jede Java-Aktivität wird vom Schnittstellen-Compiler eine Interface-Klasse generiert, in der die Methode deklariert wird, die die Aktivität ausführt. Diese Deklaration enthält auch die Angabe aller gültigen Exceptions.

[4] In Java-Programmen können jederzeit Errors geworfen werden. Die SWFR fängt diese nicht ab. Wirft eine Aktivität einen Error, so führt dies unmittelbar zum Abbruch der Transaktion durch den Subworkflow-Überwacher und damit zu einem möglicherweise inkonsistenten Zustand der Datenbanken.

Alle Angaben darüber, wie Java-Aktivitäten von der SWFR aufgerufen werden, finden sich in Unterabschnitt 3.4.6 ab Seite 29. In Anhang B ab Seite 93 findet sich zudem ein einfaches, aber vollständiges Beispiel für die Umsetzung eines Subworkflow mit Datenbankzugriff und einer Aktivität in Java-Programme, an dem sich das zuvor geschilderte Modell gut nachvollziehen läßt.

4.2 COBOL-Aktivitäten

Auch für COBOL-Aktivitäten wird vom BPD ein Skeleton erzeugt, das neben der Beschreibung der Ein- und Ausgabedaten eine Copystrecke zum Zugriff auf diese Daten enthält. COBOL-Aktivitäten haben nur diese beiden Datenobjekte zur Kommunikation mit dem Subworkflow-controller.

Im Gegensatz zu Java-Aktivitäten haben die COBOL-Aktivitäten prinzipbedingt auch keinerlei Zugriff auf den Subworkflowcontroller, so daß hier keine weiteren Sicherheitsmaßnahmen nötig und möglich sind. Natürlich dürfen auch COBOL-Aktivitäten nicht direkt auf das System zugreifen, was aber nur verhindert werden kann, indem der Quelltext vor der Compilierung auf entsprechende Konstrukte durchsucht wird, entweder manuell oder mit Hilfe eines Programms.

Wie die Daten von der SWF-Runtime zur COBOL-Aktivität und wieder zurück gelangen, spielt für die Programmierung zwar keine Rolle, soll aber zum besseren Verständnis anhand von Abbildung 4.2 kurz erläutert werden.

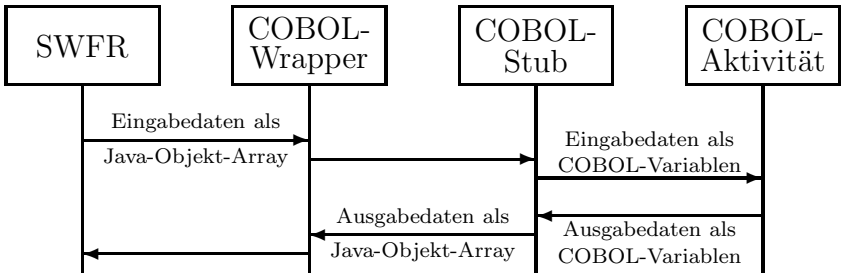


Abbildung 4.2: Aufruf einer COBOL-Aktivität

Zunächst erzeugt die SWFR ein Array mit den Eingabedaten in Objekt-Form und ein weiteres Array, in dem die Typen der Ein- und Ausgabedaten codiert abgelegt sind[5], und übergibt diese Arrays zusammen mit dem Pfad der COBOL-Aktivität dem COBOL-Wrapper. Dieser gibt seine Parameter unverändert per JNI an den COBOL-Stub weiter. Mit Hilfe der Informationen im Typ-Array kann der COBOL-Stub nun die Eingabedaten in COBOL-Datentypen umwandeln und schließlich die COBOL-Aktivität mit diesen Daten aufrufen.

Die Rückgabedaten der COBOL-Aktivität werden vom COBOL-Stub anhand der Informationen aus dem Typ-Array wieder in ein Java-Objekt-Array transformiert und über den COBOL-Wrapper an die SWFR zurückgegeben. Weitere Angaben über die Aufrufsequenz von COBOL-Aktivitäten finden sich in Unterabschnitt 3.4.7 ab Seite 29.

[5]Die Codierung der Datentypen muß zwischen dem Subworkflow-Compiler und dem COBOL-Stub abgesprochen sein.

Diese Prozedur ist natürlich aufwendiger als der Aufruf eines Java-Objekts. Umgehen ließe sich dieser Aufwand durch den Einsatz von IBM Enterprise COBOL [35] und dessen objektorientierten Erweiterungen[6], was jedoch wiederum die Umsetzung vorhandener COBOL-Programme erschweren würde. Eine weitere Lösung wäre die Implementierung aller Aktivitäten als JavaBeans und der Einsatz des PERCobol-Compilers von LegacyJ [36], der COBOL-Programme in JavaBeans verpacken kann.

[6]IBMs Enterprise COBOL kann mit Hilfe seiner objektorientierten Syntaxerweiterungen direkt auf Java zugreifen und unterstützt zu diesem Zweck auch JNI.

Kapitel 5

Der Datenzugriffsdienst

Bei allen Aufgaben des Subworkflowcontrollers (*SWFC*), für die auf Daten zugegriffen wird, die weder lokal durch den SWFR verwaltet noch durch den Carrier in die SWF-SRWU eingebracht wurden, kommt der Datenzugriffsdienst ins Spiel. Er ist für Datenbankzugriffe verantwortlich, liefert die Infrastruktur für die transaktionale Sicherung der Subworkflows und das Savepointing und lädt alle vom Subworkflow benötigten Java- oder COBOL-Objekte aus dem zentralen Verzeichnis, in dem sie der Business Process Designer (*BPD*) abgelegt hat.

Hier noch eine Anmerkung zur ursprünglichen Aufgabenstellung für diese Diplomarbeit: Der Datenzugriffsdienst sollte den Aktivitäten einen einheitlichen Zugriff auf ihre Daten ermöglichen, unabhängig davon, aus welcher Quelle sie stammen. Diese Aufgabe wurde (im Zuge des Redesign des SWFC) der Subworkflow-Runtime zugeschlagen, da bei der Generierung dieser Komponente durch den BPD alle Informationen zur Verfügung stehen, um die Verwaltung der Ein- und Ausgabedaten der Aktivitäten in einer sehr effizienten Weise zu programmieren.

Desweiteren konnte der gesamte Datenbankzugriff nur rudimentär implementiert und nicht getestet werden, da der Zugriff auf DB2 über ODBC auf dem für die vorliegende Arbeit zur Verfügung stehenden z/OS-Rechner nicht rechtzeitig eingerichtet werden konnte.

5.1 Anforderungen an den Datenzugriffsdienst

Die erste und wichtigste Anforderung an den Datenzugriffsdienst ist ein sehr hoher Durchsatz und eine hohe Performance, um die Eignung der Transaktionsmaschine (`TE`) für den Einsatz als Hochtransaktionssystem in kommerziellen Anwendungen zu gewährleisten.

Aufbauend auf dieser zentralen Forderung wurden folgende Anforderungen an den Datenzugriffsdienst formuliert, die dem in diesem Kapitel vorgestellten Design zugrunde liegen:

- Mengenorientierter Datenbankzugriff[1]
- Prefetch von Daten aus Datenbanken zu Beginn des Subworkflow durch asynchrone Datenbankzugriffe[2]

[1] Durch die Möglichkeiten der BPDL-Beschreibung eines Datenbankzugriffs ist ein mengenorientierter Datenbankzugriff problemlos möglich. Der Datenzugriffsdienst muß sich darum nur insoweit kümmern, als er mehrere Daten einlesen und in einer Feldvariablen ablegen muß. (Siehe dazu Unterabschnitt 3.4.3 ab Seite 25.)

[2] Das Prefetching wurde teilweise an den BPD delegiert, der beim Erzeugen der SWF-Runtime aus der BPDL-

- Optimierte Datenbankabfragen durch Caching aller gelesenen Daten
- Optimierte Datenbank-Updates durch Caching der Änderungen bis zum Commit
- Verzögertes, gemeinsames Commit mehrerer technischer Transaktionen bei exklusivem Zugriff auf die Datenbanken im Batch-Betrieb
- Schnelle und zuverlässige Verwaltung von Savepoints
- Effizienter Zugriff auf benötigte BPDL-Beschreibungen und Programmobjekte

5.2 Grundlagen: Effizientes Caching und MMDBs

Da der Schwerpunkt der vorliegenden Arbeit auf der Performance des Datenzugriffsdienstes und insbesondere der Caching-Möglichkeiten für Daten aus Datenbanken, Savepoint-Daten und den aus den BPDL-Beschreibungen erzeugten Programm-Objekten liegt, soll an dieser Stelle auf Mechanismen für ein effizientes Caching eingegangen werden.

Besonders interessant ist in diesem Zusammenhang das Konzept der Main Memory Database (*MMDB*), wie sie in [37] beschrieben wurde. Bei diesem Konzept werden prinzipiell alle Daten einer Datenbank im Hauptspeicher gehalten. Um die Persistenz der Daten zu gewährleisten, wird regelmäßig eine Kopie der Datenbank erstellt und abwechselnd in eine von zwei Dateien (oder in zwei Datensätze einer Disk-gebundenen Datenbank) geschrieben, und zwischen diesen Kopien werden alle Änderungen in einem Logfile gespeichert.[3]

Lese- und Schreibvorgänge auf die Datenbank werden nun ausschließlich im Hauptspeicher ausgeführt, wo sie extrem schnell ablaufen können. Erst beim Festschreiben (Commit) wird ein Eintrag in das Logfile geschrieben. Dadurch kann natürlich das Committen (sogar in stärkerer Weise als bei Disk-gebundenen Datenbanksystemen) zum Flaschenhals für die Performance der Datenbank werden.

Durch Pre-Committing, also das Freigeben von Locks auf Tabelleneinträge, noch bevor die Log-Daten im persistenten Logfile angekommen sind, kann der Durchsatz gesteigert werden, ohne jedoch die Antwortzeit der betroffenen Transaktion zu reduzieren. Group-Commits können die Anzahl der Schreibvorgänge in die Log-Dateien verringern, indem die Log-Daten von mehreren Commits gesammelt und dann gemeinsam geschrieben werden, was auf der anderen Seite natürlich zum Verlust der Daten solcher Commits führen kann.

Für die Implementierung des Caching verschiedener Daten durch den Datenzugriffsdienst wird teilweise auf diese Konzepte zurückgegriffen. Dabei führt das Schreiben eines Savepoint oder Datenbank-Updates des Subworkflow (*SWF*) in den Cache dazu, daß dieser eine dem MMDB-Commit vergleichbare Aktion ausführt, indem er einen Eintrag in das Logfile schreibt.

Beschreibung des Subworkflow frei festlegen kann, wann die Eingabevariablen der Aktivitäten mit den Datenbankdaten gefüllt werden und ob diese asynchron im Hintergrund geladen werden sollen. Dadurch kann der BPD die gesamte Zugriffsstruktur auf die Datenbanken optimieren.

[3] Während die Kopie der Datenbank erstellt wird, werden neue Einträge zwischengespeichert und in einem neuen Logfile vermerkt. Nach Abschluß der Kopie wird das alte Logfile gelöscht, und die inzwischen aufgelaufenen Änderungen werden in die Datenbank eingepflegt.

Bei einem Commit des SWF werden dann die im Cache zwischengespeicherten Daten an die externen Datenbanken übertragen und dort festgeschrieben, die dazugehörigen Savepoints werden verworfen.

Bei der Übertragung von Datenbank-Updates an die externen Datenbanken ist ein Group-Commit möglich, wenn die betroffenen Datenbanken dem Datenzugriffsdienst exklusiv zur Verfügung stehen. Da bis zu diesem Commit alle Locks gehalten werden müssen, würde der Zugriff anderer Systeme auf die Datenbank unverhältnismäßig verzögert, wenn das Group-Commit auch bei gemeinsamem Zugriff auf die Datenbank eingesetzt würde. Deshalb sind Group-Commits nur für den Batch-Betrieb vorgesehen.

Die Persistenz des Cache wird erreicht, indem in regelmäßigen Abständen der komplette Cache, bestehend aus einem einzigen, großen Byte-Array, in eine Datei geschrieben wird. Da diese Aktion verhältnismäßig schnell durchgeführt werden kann, wird während dieser Zeit der Cache für schreibenden Zugriff gesperrt. Nach erfolgreicher Sicherung wird das alte Logfile gelöscht und ein neues eingerichtet.

Im Fehlerfall wird die letzte Sicherung des Cache zurückgeladen und die im Logfile verzeichneten Aktionen nochmals durchgeführt (replay). Dabei werden auch die Datenbank-Locks überprüft und gegebenenfalls erneuert. Auf diese Weise kann der letzte konsistente Zustand des Cache vor dem Fehler sicher wieder hergestellt werden.

5.3 Aufgaben des Datenzugriffsdienstes

In diesem Abschnitt werden die Aufgaben des Datenzugriffsdienstes (*DS*) aus der Sicht des Subworkflowcontrollers geschildert. Details zur Implementierung und damit auch zur inneren Struktur und zu Optimierungen des DS finden sich in Abschnitt 5.4 ab Seite 41. Ein Überblick über die Aufgaben des Datenzugriffsdienstes findet sich in Abbildung 5.1 auf Seite 38.

Der Datenzugriffsdienst ist in seiner jetzigen Ausführung nicht mehr für die Versorgung der Aktivitäten mit ihren Ein- und Ausgabedaten zuständig, im Gegensatz zur ursprünglichen Aufgabenstellung für diese Diplomarbeit und zur Implementierung von Thomas Hornung [25]. Deshalb bleiben als Aufgaben für den DS der Datenbankzugriff, das Savepointing und das Laden von Programm-Objekten aus dem zentralen Verzeichnis des Business Process Designers (*BPD*). Diese drei Aufgabenbereiche sollen im folgenden erläutert werden.

Der Zugriff auf das zentrale Verzeichnis des BPD wird vom Subworkflowcontroller in Anspruch genommen, um die Subworkflow-Runtime (*SWFR*) und den Programmcode der Aktivitäten laden zu können. Der Datenzugriffsdienst übernimmt diese Aufgaben vor allem deshalb, weil auf diese Weise alle Zugriffslogik für Datenbanken, Verzeichnisse (beispielsweise LDAP) und Filesysteme aus dem SWFC herausgehalten wird und einmal geladene Objekte auch für weitere Subworkflows zur Verfügung stehen. Eine detaillierte Beschreibung dieser Funktionalität folgt in Unterabschnitt 5.3.3 ab Seite 40.

Auch das Savepointing profitiert von den Caching-Möglichkeiten des DS. Auf diese Weise ist eine sehr effiziente Handhabung von Savepoints und technischen Transaktionen möglich. Die technischen Transaktionen werden dabei bis auf das abschließende Commit und den Abbruch nach einem Fehler ausschließlich über Savepoints implementiert. Das Savepointing-System wird gemeinsam vom Datenzugriffsdienst und von der Subworkflow-Runtime realisiert.

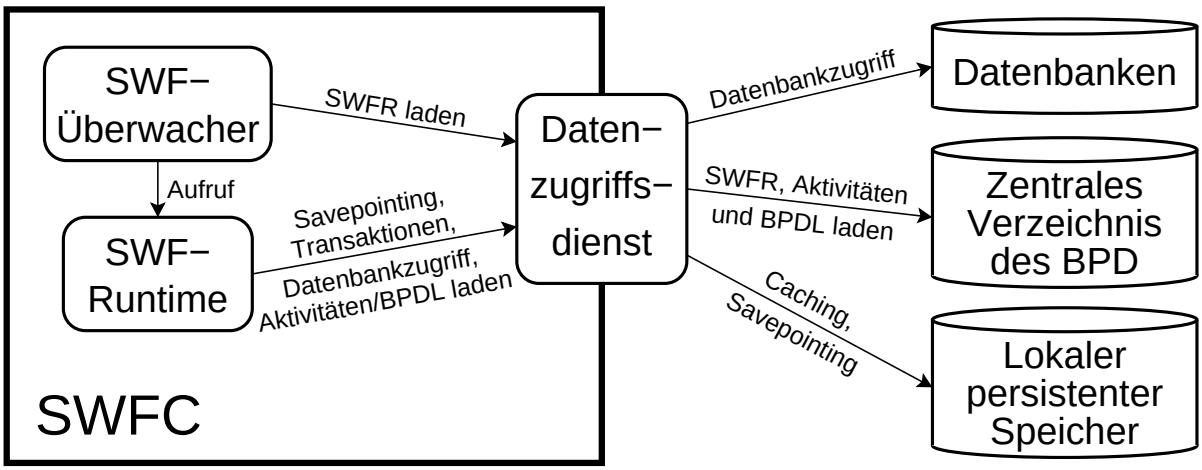


Abbildung 5.1: Aufgaben des Datenzugriffsdienstes

Die SWFR notiert dabei alle Änderungen von lokalen und globalen Variablen, so daß beim Schreiben eines neuen Savepoint eindeutig feststeht, welche Daten gesichert werden müssen. Der DS ermittelt beim Zurücksetzen auf einen alten Savepoint, welche Daten seitdem verändert wurden und mit ihren damals gültigen Werten überschrieben werden müssen. Außerdem kann der Datenzugriffsdienst den letzten geschriebenen Savepoint heraussuchen und zurückgeben, wenn der Subworkflowcontroller abgestürzt ist und neu gestartet werden muß. Die gesamte Thematik des Savepointing wird in Unterabschnitt 5.3.2 ab Seite 39 eingehend beschrieben.

Die wichtigste Aufgabe des Datenzugriffsdienstes ist jedoch die Anbindung des SWFC an externe Datenbanksysteme. Hier kommt sowohl der Datencache des DS zum Tragen, als auch die eng mit dem Savepointing verbundene Fähigkeit, Änderungen der Datenbanken bis zum Festschreiben einer technischen Transaktion zwischenzuspeichern und beim Rücksetzen auf frühere Savepoints gegebenenfalls zu löschen. Dieser Aufgabenbereich wird in Unterabschnitt 5.3.1 ab Seite 38 erläutert.

5.3.1 Die Handhabung von Datenbankzugriffen

Benötigt die Subworkflow-Runtime (*SWFR*) Daten aus externen Datenbanken, so übergibt sie dem Datenzugriffsdienst alle nötigen Informationen[4], um diese Daten aus der Datenbank zu laden, sowie einen Datencontainer mit Typ-Informationen für die zu lesenden Daten. Zurückgegeben werden die gelesenen Daten in einem neuen Datencontainer, der neben den Daten auch deren Typ-Informationen enthält.

Alternativ können die Datenbankdaten zunächst asynchron angefordert und dann in einem synchronen Zugriff abgerufen werden. Alle aus den Datenbanken gelesenen Daten werden außerdem vom Datenzugriffsdienst zwischengespeichert, so daß weitere Subworkflows beschleunigt auf diese Daten zugreifen können.

Müssen Daten während der Ausführungszeit einer technischen Transaktion konsistent mit der Datenbank sein, so wird vom Datenzugriffsdienst ein nicht-exklusiver Lock auf diese Daten

[4]Das sind zum Beispiel zuvor in der BPDL-Interface-Beschreibung des Datenbankzugriffs abgelegte SQL-Kommandos zum Zugriff auf DB2-Datenbanken.

gesetzt. Sollen die Daten im Laufe der Transaktion verändert werden, muß ein exklusiver Lock gesetzt werden. Solche Locks werden beim Festschreiben, Abbruch oder Zurücksetzen der Transaktion und auch beim Zurücksetzen auf einen Savepoint, der vor der Anforderung der Daten liegt, wieder freigegeben.

Sollen Daten in einer Datenbank geändert oder neue Daten eingefügt werden, so übergibt die SWFR dem Datenzugriffsdienst einen Datencontainer mit diesen Daten und allen nötigen Informationen für den Datenbankzugriff. Dieser Container wird vom Datenzugriffsdienst zwischengespeichert, bis seine Daten entweder beim Festschreiben der technischen Transaktion, in deren Kontext der Datenbankzugriff erfolgt ist, in eine Datenbank geschrieben werden, oder bis die Daten beim Zurücksetzen auf einen davor liegenden Savepoint oder beim Abbruch oder Zurücksetzen der Transaktion verworfen werden können.

Da schreibende Datenbankzugriffe immer zwischengespeichert werden, kann der Subworkflow ohne große Wartezeiten fortgesetzt werden. Dafür entsteht beim Festschreiben der technischen Transaktion eine größere Verzögerung, die sich jedoch nicht vermeiden läßt, solange kein exklusiver Zugriff auf die Datenbanken gewährleistet ist. Kann dies garantiert werden[5], so können durch ein verzögertes Group-Commit der Datenbankdaten auch die Commit-Zeiten der Subworkflows deutlich verringert werden.

5.3.2 Die Verwaltung von Transaktionen und Savepoints

Zu Beginn eines neuen Subworkflow wird der Datenzugriffsdienst unter Angabe der eindeutigen Kennung des Subworkflow angewiesen, nachzusehen, ob bereits Transaktionsdaten zu diesem Subworkflow im Cache vorhanden sind. Das ist der Fall, wenn der Subworkflowcontroller den Subworkflow nicht zu Ende führen konnte, weil er selbst oder der DS abgestürzt ist. Der DS speichert die Kennung des Subworkflow, ermittelt den als letztes geschriebenen Savepoint und die zur Zeit von dessen Erstellung aktuellen Daten und gibt diese Daten zusammen mit der Nummer des Savepoint zurück. Waren keine Transaktionsdaten vorhanden, so richtet er einen neuen Cachebereich für den Subworkflow ein.

Technische Transaktionen werden vom Datenzugriffsdienst größtenteils auf Savepoints umgesetzt. Am Anfang einer Transaktion wird dem DS ein Container mit den Eingangsdaten der Transaktion und der Nummer des Rücksetzpunktes übergeben. Der Datenzugriffsdienst schreibt daraufhin einen neuen impliziten Savepoint, bei dem der Subworkflow fortgesetzt wird, wenn die gesamte Transaktion zurückgesetzt werden soll.

Um einen expliziten Savepoint zu schreiben, übergibt die Subworkflow-Runtime dem Datenzugriffsdienst einen Container mit der Nummer des zu schreibenden Savepoint (siehe Unterabschnitt 3.4.3 ab Seite 25), allen seit dem letzten Savepoint geänderten Daten und einem Flagfeld, in dem alle Änderungen verzeichnet sind. Dadurch können Savepoints inkrementell geschrieben werden: Nur geänderte Daten müssen übertragen werden.

Das Zurücklesen eines Savepoint geschieht analog: Die SWFR übergibt dem Datenzugriffsdienst einen Container mit der Nummer des zu lesenden Savepoint und einem Flagfeld, das die

[5]Exklusiver Zugriff auf eine Datenbank kann einerseits im Batch-Betrieb vom Administrator explizit eingeräumt werden. Andererseits ist es auch denkbar, der Transaktionsmaschine bestimmte Datenbanken exklusiv zur Verfügung zu stellen, die nur innerhalb einer PD zugänglich sind und von einem Datenzugriffsdienst verwaltet werden.

Änderungen seit dem letzten Savepoint aufzeigt. Dieser gibt einen Container mit allen Daten zurück, die seit dem angeforderten Savepoint verändert wurden. Außerdem löscht er alle Locks und Schreibzugriffe auf externe Datenbanken und alle Savepoints, die nach dem angeforderten Savepoint übermittelt wurden.

Soll eine technische Transaktion zurückgesetzt werden, so wird einfach der erste, implizite Savepoint der Transaktion zurückgelesen. Beim Abbruch einer technischen Transaktion löscht der DS alle Savepoints, Datenbank-Locks und -Änderungen, die seit Beginn dieser Transaktion übergeben wurden. Die anschließend ausgeführten Kompensationen der vorangegangenen Transaktionen greifen auf die impliziten Savepoints am Ende der Transaktionen und damit auf deren (globale) Ausgangsvariablen zu, um die von ihnen benötigten Angaben der zugehörigen Transaktion zu erhalten. Kompensationsmaßnahmen werden ähnlich wie technische Transaktionen abgearbeitet, bei deren Fehlschlagen allerdings ein inkonsistenter Zustand entsteht, der manuell behoben werden muß.

Wenn eine Transaktion festgeschrieben werden soll, übergibt die SWFR dem Datenzugriffsdienst einen Container mit den Ausgangsdaten der Transaktion und der Nummer des Endpunktes der Transaktion im Subworkflow. Der DS führt zunächst die zwischengespeicherten Schreibzugriffe auf den externen Datenbanken aus und schreibt sie dort fest, wobei auch alle Datenbank-Locks freigegeben werden. Anschließend löscht er alle gespeicherten Savepoints und schreibt einen neuen impliziten Savepoint, der das Ende der Transaktion markiert. Dieser enthält alle Variablen der technischen Transaktion und wird für die Kompensation der technischen Transaktion benötigt.

Nachdem das Ergebnis am Ende des Subworkflow an den Carrier übergeben wurde, wird der Datenzugriffsdienst ein letztes Mal aufgerufen, um die verbliebenen[6] Transaktionsdaten zu löschen.

5.3.3 Das Laden von Programm-Objekten

Der Datenzugriffsdienst ist in der Lage, auf Anforderung des Subworkflow-Überwachers die Subworkflow-Runtime zu laden, und auf deren Anforderung hin die Programm-Objekte von Aktivitäten. Dazu greift er auf das zentrale Verzeichnis des BPD zu, in dem alle BPDL-Beschreibungen und daraus generierten Programme der Transaktionsmaschine abgelegt sind.

Um ein beliebiges Programm-Objekt zu laden, wird dem Datenzugriffsdienst jeweils ein Name und eine Versionsnummer übergeben, mit deren Hilfe er das angeforderte Objekt im zentralen Verzeichnis des BPD lokalisiert, lädt und im lokalen Filesystem[7] ablegt. Nach dem erfolgreichen Laden eines COBOL-Programm-Objekts gibt er dessen Position als Ergebnis zurück, für Java-Klassen ist dies nicht nötig, da die JVM diese automatisch lädt.

Zur Beschleunigung kann der Datenzugriffsdienst auch in einem ersten Schritt angewiesen werden, ein Programm-Objekt asynchron zu laden. Über den zuvor beschriebenen synchronen Mechanismus wird dann im zweiten Schritt der Erfolg überprüft, und nötigenfalls auf den Abschluß der Operation gewartet.

[6] Im Normalfall bleiben nur die Savepoints übrig, die implizit am Ende jeder Transaktion geschrieben wurden. Trat ein Fehler auf, können auch noch andere Transaktionsdaten im Cache verblieben sein.

[7] Die Position im lokalen Filesystem, an der Java-Klassen und COBOL-Programme abgelegt werden, ist in der BPDL-Parameter-Beschreibung des Datenzugriffsdienstes abgelegt. Sie ist für den SWFC unerheblich, da ihm die Position von COBOL-Programmen mitgeteilt wird und die Klassenpfade der JVM vom Launcher festgelegt wurden.

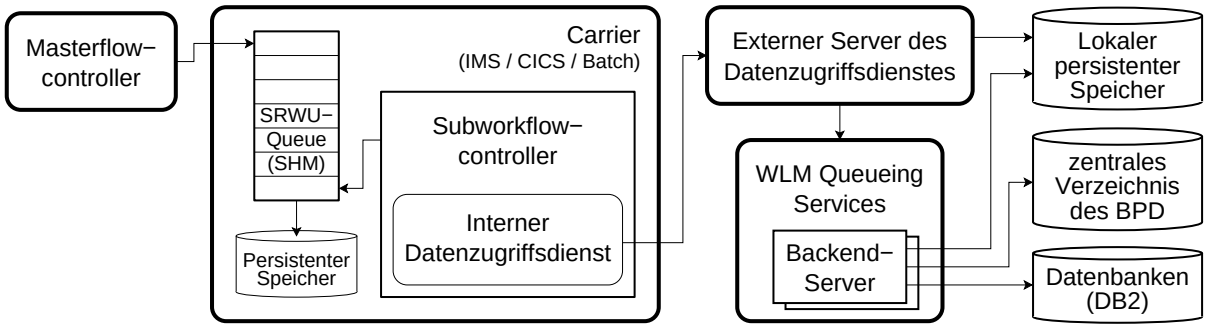


Abbildung 5.2: Aufteilung des Datenzugriffsdienstes und Position im Gesamtsystem

5.4 Aufteilung des Datenzugriffsdienstes

Um gelesene Daten aus externen Datenbanken wie auch geladene Programm-Objekte von Subworkflow-Runtimes und Aktivitäten allen technischen Transaktionen aller auf demselben Rechner ausgeführten Subworkflows zugänglich zu machen, wurde ein Teil der Aufgaben des Datenzugriffsdienstes (*DS*) in einen externen Server ausgelagert, der im folgenden als „externer Server des Datenzugriffsdienstes“ (externer Server des DS) bezeichnet wird.

Dieser externe Server verwaltet alle seine Daten in einem Cache im Hauptspeicher, der auf dem Konzept der MMDBs[8] basiert, was einerseits die Persistenz dieser Daten sichert und andererseits eine hohe Performance ermöglicht. Deshalb lohnt es sich, andere Aufgaben, bei denen Daten persistent zwischengespeichert werden müssen, ebenfalls an den externen Server zu delegieren. Aus diesem Grund wird er auch für das Zwischenspeichern von Schreibzugriffen auf externe Datenbanken und für die Verwaltung der Savepoints eingesetzt.

Und da für das Savepointing und das Zwischenspeichern von Datenbank-Updates nun bereits auf den externen Server zugegriffen wird und so alle Daten für das Festschreiben von technischen Transaktionen bereits dort liegen, wird mit dem Commit auch die letzte Funktion der transaktionalen Sicherung eines Subworkflow an den externen Server delegiert. Der Start, das Rücksetzen und Abbrechen der technischen Transaktionen werden schließlich bereits über das Savepointing realisiert.

Der im SWFC verbleibende Teil des DS, im folgenden als „interner Datenzugriffsdienst (interner DS)“ bezeichnet, stellt zwar immer noch die gesamte Funktionalität des DS den anderen Modulen innerhalb des SWFC zur Verfügung, greift aber nun für die meisten Aufgaben direkt auf den externen Server zurück.

Der Datenzugriffsdienst weist also intern eine klassische Client-Server-Architektur auf, weshalb im folgenden der interne DS gelegentlich auch als Client und der externe Server schlicht als Server bezeichnet wird, wenn es um die Kommunikation innerhalb des DS geht.

Eine Übersicht über die Aufteilung des Datenzugriffsdienstes und seine Einbindung in das Gesamtsystem ist in Abbildung 5.2 zu finden. Die Pfeile haben dabei jeweils die Bedeutung „greift zu auf“.

[8] „main memory databases“, siehe Abschnitt 5.2 ab Seite 36 für die Grundlagen und Unterabschnitt 5.6.2 ab Seite 46 für Details.

5.4.1 Kommunikation innerhalb des Datenzugriffsdienstes

Der interne Datenzugriffsdienst muß über eine schnelle Kommunikations-Schnittstelle mit dem externen Server des Datenzugriffsdienstes verbunden werden, damit der Datenzugriffsdienst nicht zum Flaschenhals für den Subworkflowcontroller wird. Für diese Schnittstelle wurden die UDP Sockets[9] der Unix System Services (*USS*) [15] gewählt, da der SWFC als Java-Programm sowieso auf die USS zurückgreift, und da Sockets von allen Programmiersprachen unterstützt werden.

Das UDP-Protokoll ist vollkommen ungesichert. Um die transaktionale Sicherung des Subworkflow an dieser Stelle nicht zu unterwandern, muß also das Kommunikationsprotokoll zwischen Client (interner DS) und Server (externer Server des DS) so ausgelegt sein, daß die Persistenz der übertragenen Daten gesichert ist. Dafür muß sichergestellt sein, daß jeder Auftrag sein Ziel erreicht und auch nicht versehentlich mehrfach ausgeführt wird. Es wird also eine Exactly-Once-Semantik gefordert.

Um diese zu erreichen, wird zunächst ein Drei-Phasen-Protokoll vereinbart: Auf eine Anfrage des Client hin liefert der Server das Ergebnis, das vom Client wieder bestätigt wird, woraufhin der Server das Ergebnis der Anfrage verwerfen kann. Außerdem ist jede dieser Anfragen mit der Kennung des Subworkflow sowie mit einer innerhalb eines Subworkflow fortlaufenden Nummer versehen, die vom Client bei jeder Anfrage an den Server hochgezählt wird.

Sämtliche Anfragen und sämtliche Antworten darauf werden vom Server protokolliert. Auf diese Weise erkennt dieser sofort, ob eine Anfrage zum wiederholten Mal gestellt wird. Ist die erneut gestellte Anfrage noch in Bearbeitung, so wartet der Server bis zum Abschluß der Bearbeitung der ursprünglichen Anfrage. Ansonsten kann er sofort das Ergebnis zurückliefern.

Der Server bietet keine Persistenz für Kommunikationsdaten, da das persistente Logging sämtlicher Server-Anfragen und -Antworten zu aufwendig wäre. Aus diesem Grund ist nach einem Absturz des Servers immer ein Neustart aller zu diesem Zeitpunkt aktiven Subworkflows beim zuletzt geschriebenen Savepoint erforderlich. Wenn ein Subworkflowcontroller abstürzt, so wird der unterbrochene Subworkflow ohnehin beim zuletzt geschriebenen Savepoint fortgesetzt.

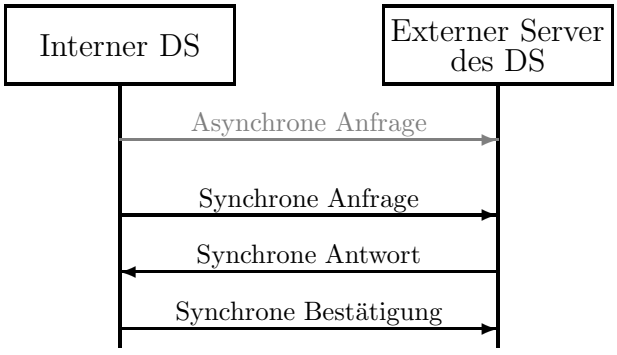


Abbildung 5.3: Kommunikationsprotokoll zwischen internem DS und externem Server

Mit dem Drei-Phasen-Protokoll, wie es in auch Abbildung 5.3 dargestellt ist, der Protokollierung der Kommunikation durch den Server und dem soeben beschriebenen Restart-Verhalten läßt

[9] Tatsächlich implementiert wurde die Kommunikation über TCP Sockets, da UDP Sockets unter z/OS eine deutlich schlechtere Performance aufweisen, wie in Unterabschnitt 7.2.1 ab Seite 69 gezeigt wird. Aber auch die Performance der TCP-Sockets ist nicht ausreichend, daher wird für die Weiterentwicklung die Umstellung auf System C Message Queues empfohlen.

sich die Exactly-Once-Semantik der Kommunikation garantieren, was im folgenden anhand der möglichen Fehlersituationen kurz erläutert werden soll:

1. Der Client erhält innerhalb einer einstellbaren Zeitspanne keine Antwort vom Server. In diesem Fall sendet er seine Anfrage erneut[10]. Die Reaktion des Servers hierauf hängt von den möglichen Ursachen des Fehlers ab:

   (a) Die ursprüngliche Anfrage ist verloren gegangen. Der Server erkennt anhand der eindeutigen Nummer der Anfrage, daß er diese noch nicht erhalten hatte, und bearbeitet sie.

   (b) Die Antwort des Servers ist verloren gegangen. In diesem Fall erkennt der Server, daß er die Anfrage schon bearbeitet hat, und sendet das zwischengespeicherte Ergebnis noch einmal.

   (c) Der Client sendet seine Anfrage erneut, obwohl der Server mit der Bearbeitung der ursprünglichen Anfrage noch nicht fertig ist. Der Server trägt den neuen Antwort-Port des Client in die protokollierte Anfrage ein und überläßt die Beantwortung der erneuten Anfrage dem gerade mit dieser Anfrage beschäftigten Thread.

   (d) Der Server stürzt während der Bearbeitung der Anfrage ab. Die erneute Anfrage des Client nach dem Neustart des Servers wird mit einem Fehlercode beantwortet, der Client muß zum letzten Savepoint zurücksetzen.

2. Der Server erhält den Auftrag, nach dem letzten gespeicherten Savepoint zu suchen. In diesem Fall löscht er alle noch vorhandenen Kommunikationsdaten einer eventuell vorangegangenen Ausführung des Subworkflow.

3. Der Server erhält die Mitteilung, daß der Subworkflow beendet wurde. Alle Kommunikationsdaten werden gelöscht.

4. Erhält der Server von einem noch nicht abgeschlossenen Subworkflow keine Nachrichten mehr, so muß er trotzdem alle Transaktionsdaten dieses Subworkflow speichern. Er könnte zwar über einen Timeout nach einer ausreichend langen Zeit sämtliche Kommunikationsdaten löschen, der dadurch bedingte zeitliche Overhead wäre im Vergleich zum wiedergewonnenen Speicherplatz jedoch nicht zu rechtfertigen.

Einen Sonderfall stellt die asynchrone Kommunikation dar, die für das Prefetching von Datenbankdaten oder Programm-Objekten eingesetzt wird. Auf die Anforderung der Daten antwortet der Server zunächst gar nicht, und der Client wartet auch nicht auf Ergebnisse. Der Server führt aber die Anfrage wie gewohnt durch und speichert das Ergebnis. In einer zweiten, synchronen Anfrage holt der Client dann das zwischengespeicherte Ergebnis beim Server ab.

Während des Subworkflow könnte für synchrone Anfragen ein Zwei-Phasen-Protokoll vereinbart werden. Der Client würde einfach die Bestätigung weglassen, der Server eine ältere Antwort löschen, sobald eine neuere Anfrage kommt. Die Ergebnisse von asynchronen Anfragen müßten natürlich auch hier gespeichert werden, bis sie durch eine synchrone Anfrage abgeholt wurden. Da der Overhead für die abschließende Bestätigungsnachricht des Drei-Phasen-Protokolls jedoch minimal ist, wurde vorerst auf die Implementierung dieses Verfahrens verzichtet.

[10] Die Anzahl der Kommunikationsversuche des Client läßt sich für jeden Subworkflow und jede Anfrage individuell konfigurieren, ebenso der Timeout für die Kommunikation mit dem Server.

5.5 Aufgaben des internen Datenzugriffsdienstes

Der interne Datenzugriffsdienst stellt der Subworkflow-Runtime und den anderen Modulen des Subworkflowcontrollers die volle Funktionalität des Datenzugriffsdienstes zur Verfügung, wie sie in Abschnitt 5.3 ab Seite 37 beschrieben wurde. Für die meisten dieser Aufgaben greift er direkt auf die Fähigkeiten des externen Servers zurück.

Im folgenden wird in relativer Kürze dargestellt, über welche Klassen-Methoden die Aufgaben an den internen Datenzugriffsdienst übermittelt werden, welchen Teil dieser Aufgaben der interne DS jeweils selbst wahrnimmt, und wie und wann er zur Bewältigung seiner Aufgaben auf den externen Server zurückgreift. Dabei werden die Details des Drei-Phasen-Kommunikations-Protokolls zwischen internem DS und externem Server, wie zum Beispiel Bestätigungen der Server-Antworten, nicht mehr erwähnt.

5.5.1 Datenbankzugriffe

Sobald die SWF-Runtime eine synchrone Datenbankabfrage durchführen will, übergibt sie eine entsprechende Anfrage über die Methode `readDB` des internen Datenzugriffsdienstes (siehe Unterabschnitt 5.3.1 ab Seite 38). Dieser leitet sie wiederum direkt an den externen Server weiter und wartet auf dessen Antwort, die wiederum direkt der SWFR übergeben wird. Fehler bei der Datenbankabfrage werden direkt an die SWFR weitergeleitet und müssen von dieser entsprechend der Anweisungen aus der Workflow-Beschreibung behandelt werden.

Bei einer asynchronen Datenbankabfrage (`readDBAsynch`) kehrt der interne DS sofort ohne Ergebnis zurück, nachdem er den Auftrag an den externen Server weitergeleitet hat. Werden die Daten aus dieser Antwort von der SWFR benötigt, so fordert sie diese über die synchrone `readDB`-Methode beim internen DS an. Auch diese synchrone Anfrage wird direkt an den Server weitergeleitet, der die angeforderten Daten sofort zurückgibt, falls sie bereits vorliegen, und ansonsten wartet, bis der Datenbankzugriff ausgeführt ist.

Wenn die SWFR Daten in die Datenbanken schreiben möchte, so ruft sie dafür die `updateDB`- oder `insertDB`-Methode des internen DS auf. Auch diese Anfragen werden sofort an den externen Server weitergeleitet, der sie bis zum nächsten Commit zwischenspeichert. Ein Fehler, der bei der Überprüfung des Updates durch den externen Server auftritt, wird sofort zurückgemeldet.[11] Fehler, die erst beim tatsächlichen Update der Datenbanken während der Commit-Phase auftreten, führen zum Fehlschlagen des gesamten Commit, und damit abhängig von den Parametern der fehlgeschlagenen Transaktion zu deren Abbruch oder Neustart (Rollback).

5.5.2 Subworkflows, Transaktionen und Savepointing

Anfragen, die die Verwaltung von Subworkflows, Transaktionen und Savepoints betreffen, werden unverändert an den externen Server durchgereicht. Als Interface stehen dafür die Methoden `startSubworkflow`, `beginTransaction`, `setSavepoint`, `restoreSavepoint`, `rollbackTransaction`, `abortTransaction`, `commitTransaction`, `beginCompensation`, `rollbackCompensation`, `abortCompensation`, `commitCompensation` und `endSubworkflow` des internen Datenzugriffsdienstes zur Verfügung.

[11] Diese Überprüfung kann nur feststellen, ob der erforderliche Zugriffsmodus für die Daten erteilt wurde.

5.5.3 Laden von Programm-Objekten

Für das Laden der SWF-Runtime durch den SWF-Überwacher oder von Java- und COBOL-Aktivitäten durch die SWF-Runtime stellt der interne Datenzugriffsdienst jeweils synchrone und asynchrone Interfaces zur Verfügung.

Das synchrone Laden der SWFR, von Java- und von COBOL-Aktivitäten erfolgt durch den Aufruf der Methoden `loadSWFR`, `loadJavaObject` und `loadCobolObject` des internen DS, der diese Anfragen sofort an den externen Server weiterleitet. Dieser antwortet erst, wenn das angeforderte Objekt zur Verfügung steht, woraufhin der interne DS den Empfang der Antwort bestätigt. Bei der SWFR und Java-Aktivitäten ist die Antwort des externen Servers und des internen DS eine reine Bestätigung, daß die Java-Klassen an der richtigen Stelle abgelegt wurden, bei COBOL-Aktivitäten dagegen wird der Zugriffspfad auf das Programm-Objekt zurückgegeben.

Sollen Programm-Objekte asynchron geladen werden, so werden die asynchronen Pendants der zuvor aufgeführten Methoden des internen DS aufgerufen: `loadSWFRAsynch`, `loadJavaObjectAsynch` und `loadCobolObjectAsynch`. Auch diese Anfragen leitet der interne DS unverändert an den Server weiter, kehrt aber sofort zurück. Die nachfolgende synchrone Anfrage kann nun vom Server im Idealfall sofort beantwortet werden.

5.6 Aufbau des externen Servers

Der externe Server des Datenzugriffsdienstes übernimmt im Prinzip alle Aufgaben, die dem Datenzugriffsdienst in Abschnitt 5.3 ab Seite 37 zugeteilt wurden, also Datenbankzugriffe mit Caching, Transaktionssicherung, Savepointing und Zugriff auf das zentrale Verzeichnis des Business Process Designers, um Programm-Objekte nachzuladen.

Der interne Datenzugriffsdienst reicht, wie in Abschnitt 5.5 beschrieben, fast alle Aufgaben ohne Änderungen an den externen Server weiter, er übernimmt nur in der asynchronen Kommunikation für das Prefetch von Datenbankdaten und Programm-Objekten eine eigene Rolle. Eine genaue Beschreibung darüber, wie der externe Server die verschiedenen Aufgaben bewältigt, findet sich in den folgenden Unterabschnitten, ebenso wie eine Beschreibung der Cache-Verwaltung und der internen Struktur des Servers.

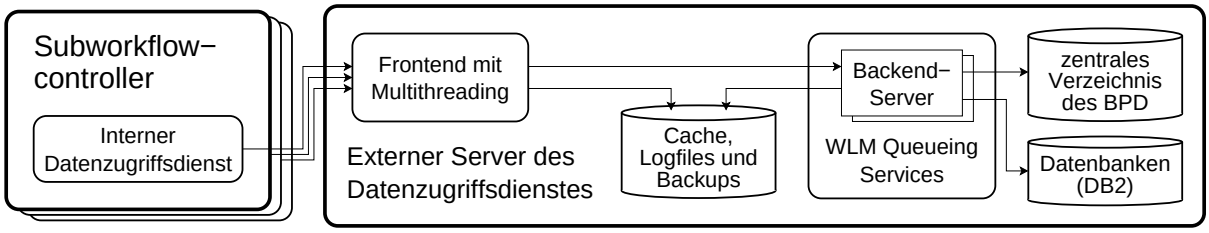


Abbildung 5.4: Aufbau des externen Servers

5.6.1 Interne Struktur des externen Servers

Der externe Server steht allen SWFCs in einer Rechnerumgebung gemeinsam zur Verfügung, und diese sollen auch gleichzeitig bedient werden können. Außerdem sollen asynchrone Datenbankabfragen im Hintergrund ausgeführt werden, während derselbe SWFC weitere Anfragen an den Server stellen kann. Aus diesen Gründen muß der externe Server intern so weit wie möglich parallel arbeiten können.

Um die Skalierbarkeit des externen Servers zu verbessern, wird der Server in ein Frontend und ein Backend unterteilt, die über die Queueing Manager Services der Workload Management Services (*WLM-Services*, siehe [38]) von z/OS kommunizieren. Das Frontend wird zusätzlich als nebenläufiger (multithreaded) Server ausgeführt, um eine möglichst geringe Latenzzeit zu gewährleisten. Dieser Aufbau ist in Abbildung 5.4 dargestellt.

Beide Teile des externen Servers des Datenzugriffsdienstes werden in der Programmiersprache C implementiert, da der Zugriff auf die WLM-Services nur unter Assembler und C möglich ist. Außerdem bietet C eine vergleichsweise hohe Performance, die nur wenig unter der von Assembler liegt, ist auf der anderen Seite aber eine vollwertige, wenn auch einfache prozedurale Hochsprache.

Das Frontend des externen Servers fungiert als Queue Manager. Es nimmt die Aufträge der SWFCs entgegen. Aufträge mit sehr kurzer Laufzeit, wie das einfache Zurückgeben von im Cache bereits vorhandenen Informationen, werden direkt von der Server-Instanz erledigt, was zu einem sehr hohen Durchsatz und einer sehr niedrigen Latenz für solche Anfragen führt.[12]

Für alle anderen Aufträge, insbesondere für alle Datenbankzugriffe, startet der Server jeweils einen eigenen Thread, der nach der Abarbeitung des Auftrags wieder beendet wird.[13] Dieser Thread hält die Verbindung zum anfragenden SWFC, während er den Auftrag an die Queueing Manager Services übergibt, die lastabhängig mehrere Instanzen des Backend verwalten und einer von diesen den Auftrag übermitteln. Das Ergebnis wird über den Thread des Frontends direkt an den anfragenden SWFC weitergeleitet.

5.6.2 Der Server-Cache

Alle vom externen Server verwalteten Daten werden in einem gemeinsamen Cache gespeichert, der von einem Cache-Manager-Thread des Frontend kontrolliert wird. Über ein zweistufiges (exklusiver und gemeinsamer Zugriff) und feingranulares Locking können die Threads des Frontend und alle Instanzen des Backend gleichzeitig auf diesen Cache zugreifen, ohne dessen Konsistenz zu verletzen. Das Locking erfolgt mit Hilfe der Read-Write-Locks, die vom POSIX-Thread-Subsystem zur Verfügung gestellt werden.[14]

Durch Logging und regelmäßige Backups wird die Persistenz aller Daten des Cache gesichert, für die dies erforderlich ist. Eine detaillierte Beschreibung von Organisation und Verwaltung des Cache findet sich in den folgenden Abschnitten, einen Überblick über den Aufbau des Cache gibt Abbildung 5.5.

[12] Die Ausführung von solchen sehr „billigen“ Aufgaben durch Server-Threads würde zwar die Parallelverarbeitung auch dieser Aufträge ermöglichen, der Overhead der Thread-Verwaltung ist jedoch immer noch vergleichsweise hoch, so daß das direkte Abarbeiten dieser Anfragen kaum eine Mehrbelastung der Server-Instanz bedeutet.

[13] Es wird empfohlen, in der weiteren Entwicklung des DS mehrere Threads in einem Pool vorhalten. Der Aufwand zum Starten und Beenden von Threads ist groß genug, um dieses Vorgehen zu rechtfertigen.

[14] Diese Read-Write-Locks der POSIX-Threads können auch prozeßübergreifend eingesetzt werden, wenn sie in einem „shared memory“-Bereich liegen.

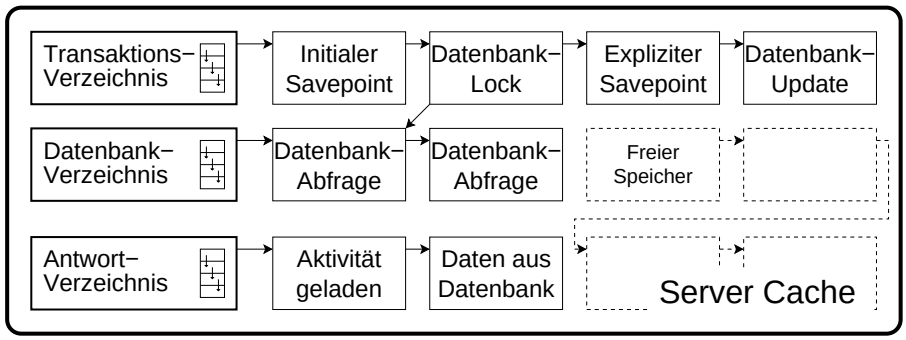


Abbildung 5.5: Aufbau des Server-Cache

5.6.2.1 Organisation des Cache

Für den Zugriff auf den Cache gibt es drei zentrale Verzeichnisse, die innerhalb des persistenten Cache[15] liegen und deren Persistenz durch Logging aller Veränderungen gesichert wird: Ein Verzeichnis für die Transaktionsdaten, ein weiteres Verzeichnis für die Resultate von Datenbank-Abfragen und ein Verzeichnis, in dem aus dem zentralen Verzeichnis des BPD geladene Programm-Objekte notiert werden.

Zwei zusätzliche, nicht persistente Verzeichnisse dienen der Protokollierung sämtlicher Server-Anfragen und der Antworten darauf. Diese beiden letzten Verzeichnisse werden über die Subworkflow-Kennung und die Sequenz-Nummer der Anfragen indiziert.

Sämtliche Verzeichnisse des Server-Cache sind als Hash-Tabelle mit Linked-List-Einträgen organisiert. Das garantiert eine kurze Zugriffszeit und vermeidet dabei den Aufwand, den ein Sortieren der Einträge in den Verzeichnissen mit sich bringen würde.

Das Transaktions-Verzeichnis: Im Transaktions-Verzeichnis werden alle Savepoints, Datenbank-Updates, Datenbank-Inserts und Datenbank-Locks eines jeden Subworkflow chronologisch eingetragen. Die Datenbank-Locks verweisen auf Einträge im Datenbank-Verzeichnis. Indiziert wird das Transaktions-Verzeichnis über die eindeutige Kennung des Subworkflow.

Das Datenbank-Verzeichnis: Ergebnisse von Anfragen an die Datenbanken werden im Datenbankverzeichnis eingetragen, in dem auch der Lock-Status dieser Daten angegeben ist, bestehend aus dem Lock-Modus (S für geteilten Zugriff, U für angekündigten exklusiven Zugriff und X für exklusiven Zugriff, siehe Unterabschnitt 5.6.4), einer Lock-Granularität (CS, RS und RR, siehe abermals Unterabschnitt 5.6.4), der Anzahl der S-Locks und einem Zeiger auf den Eintrag des Subworkflow im Subworkflowverzeichnis, der einen U- oder X-Lock hält. Zusätzlich wird im Transaktions-Verzeichnis ein Lock-Eintrag in Form eines Zeigers auf den Eintrag im Datenbankverzeichnis gesetzt. Die Daten selbst werden im persistenten Teil des Cache abgelegt. Indiziert wird das Datenbankverzeichnis über den Text der Anfrage.

Das Objektverzeichnis: Das Objektverzeichnis wird über den Namen und die Version der referenzierten Objekte indiziert. Die Objekte selbst werden im lokalen Dateisystem abgelegt.

[15]Der persistente Cache ist als Byte-Array implementiert, so daß eine Sicherung sehr schnell durchgeführt werden kann, indem das gesamte Byte-Array in eine Datei geschrieben wird. Auch die persistenten Verzeichnisse sind in diesem Array untergebracht. Ihre Sicherung besteht also darin, sich den Offset innerhalb des Byte-Array zu merken, an dem das Verzeichnis abgelegt ist.

5.6.2.2 Verwaltung des Cache

Für die Verwaltung des persistenten Cache greift der Cache-Manager-Thread auf die drei persistenten Verzeichnisse des Cache zu. Eine weitere Liste verzeichnet den freien Speicher innerhalb des Cache.

Die nicht persistenten Verzeichnisse des Cache und deren Elemente, also die protokollierten Anfragen und Ergebnisse, müssen nicht extra verwaltet werden. Sie werden gelöscht, sobald der Server gesicherte Kenntnis vom Ende des zugehörigen Subworkflow hat.

Die Einträge aller persistenten Verzeichnisse werden durch einen den MMDBs [37] (siehe Abschnitt 5.2 ab Seite 36) vergleichbaren Datenbank-Manager verwaltet, der regelmäßig Backups des gesamten Cache erstellt und nach deren erfolgreicher Durchführung das alte Logfile löscht. Neue Einträge in diesen Verzeichnissen werden zusätzlich in ein Logfile geschrieben, um deren Persistenz garantieren zu können. Einträge im Datenbank-Verzeichnis, auf die kein Lock besteht, und Einträge im Objekt-Verzeichnis gelten als flüchtig und werden von einem Cache-Manager auf LRU[16]-Basis verwaltet.

Die Liste für den freien Speicher wird durch einen Speicher-Manager verwaltet, der die dauerhafte Fragmentierung des Speichers verhindert, indem er alle Listen durchgeht und dabei die Belegung des Cache kompaktiert. Alle drei Manager werden in regelmäßigen Abständen durch den Cache-Manager-Thread des Server-Frontend ausgeführt.

5.6.2.3 Maßnahmen bei Speicherknappheit

Wird der Speicher des Cache knapp, so wird zunächst versucht, einen größeren Cache zu allozieren. Schlägt das fehl, so versucht der Cache-Manager, nicht-persistente Einträge aus dem Cache zu entfernen, indem der Alterungsprozeß der Einträge beschleunigt wird. Wenn es nicht anders möglich ist, werden auch für Datenbanken, auf die exklusiver Zugriff besteht, keine Daten mehr gespeichert, für die kein Locking erforderlich ist.

Man könnte auch das Speichern von Programm-Objekten unterlassen, was jedoch nicht viel Sinn macht, da diese nicht viel Raum einnehmen, und der Performance-Verlust doch spürbar wäre. Das Auslagern von persistenten Einträgen im Cache, also ge'locked'e Datenbank-Einträge und Transaktionsdaten, ist bislang nicht vorgesehen, da diese Maßnahme den Sinn des Datenzugriffs, nämlich die Beschleunigung der Zugriffe auf Transaktionsdaten und Datenbanken, ad absurdum führen würde. Der Administrator des Systems ist hier in der Verantwortung, dem Server-Prozeß genügend Speicher zur Verfügung zu stellen.

5.6.3 Subworkflows, Transaktionen und Savepointing

Im folgenden wird das Verhalten des Servers bei Anfragen beschrieben, die mit der Steuerung des Subworkflow und dessen transaktionaler Absicherung zu tun haben. Diese Aktionen des Servers wurden der besseren Übersicht wegen in Tabelle 5.1 zusammengefaßt.

Wenn der interne Datenzugriffsdienst dem Frontend einen neuen Subworkflow meldet, so wird zunächst nachgesehen, ob für diesen Subworkflow, der durch seine Kennung eindeutig bestimmt ist, bereits ein Eintrag im Cache existiert. Ist das nicht der Fall, wird ein neuer Eintrag erzeugt.

[16] *Least Recently Used* ist ein verbreitetes Verfahren für die Cache-Verwaltung und wird beispielsweise für Prozessor- oder Festplatten-Caches eingesetzt.

Aktion	Cache-Eintrag		Savepoint		Transaktionsdaten		Backend
	anlegen	löschen	anlegen	lesen	ausführen	löschen	aufrufen
Subworkflow starten	(•)[a]			(•)		(•)	
Subworkflow beenden		•				•	
Transaktion starten			•				
Transaktion rücksetzen				•		•	
Transaktion abbrechen			•			•	
Transaktion festschreiben			•		•	•	•
Savepoint schreiben			•				
Savepoint lesen				•		•	•
Kompensation starten				•			
Kompensation rücksetzen				•		•	
Kompensation abbrechen						•	
Kompensation beenden					•	•	•

[a]Falls der Subworkflow erneut gestartet wurde, wird der letzte Savepoint gesucht, und nachfolgende Transaktionsdaten werden gelöscht. Ansonsten wird ein neuer Eintrag für den Subworkflow erzeugt.

Tabelle 5.1: Ausführung von Anfragen zur Subworkflow-Steuerung

Ansonsten wird von einem Thread des Frontend der letzte verfügbare Savepoint gesucht. Das Ergebnis der Suche wird danach an den internen DS gemeldet.

Die Anfrage, die zu Beginn einer neuen Transaktion an den externen Server gesendet wird, enthält einen Container mit den Eingangsdaten der Transaktion, eine Savepoint-Nummer für den Einsprung beim Rücksetzen der Transaktion (siehe Unterabschnitt 3.4.3 ab Seite 25) und die eindeutige Subworkflow-Kennung. Diese Anfrage wird vom Frontend direkt an das Backend weitergereicht, das die mitgelieferten Daten als initialen Savepoint der Transaktion unter dem Eintrag des Subworkflow im Cache ablegt und persistiert, indem es einen Eintrag in das Logfile schreibt.

Um einen expliziten Savepoint zu schreiben, werden dem externen Server neben dessen Nummer und der Kennung des Subworkflow alle seit dem letzten Savepoint veränderten Daten übergeben, sowie ein Flagfeld, in dem verzeichnet ist, welche Daten das sind. Das Frontend gibt alle Daten an das Backend weiter, das den Savepoint im Cache-Eintrag des Subworkflow persistent ablegt.

Soll der initiale Savepoint im Falle des Rücksetzens einer Transaktion zurückgelesen werden, so reicht das Frontend diesen Auftrag an das Backend weiter, das anhand der übergebenen Subworkflow-Kennung die Eingangsdaten der Transaktion lokalisiert. Anschließend werden alle späteren Savepoints und Datenbank-Updates der Transaktion verworfen und alle Datenbank-Locks freigegeben, und schließlich werden die Eingangsdaten der Transaktion über das Frontend an den internen DS zurückgegeben.

Das Zurücklesen der expliziten Savepoints wird ebenfalls an ein Backend delegiert, das mit Hilfe der Subworkflow-Kennung die Savepoints der laufenden Transaktion lokalisiert und dann mit Hilfe der Savepoint-Nummer und der übergebenen Flag-Liste ermittelt, welche Daten seit dem angeforderten Savepoint überschrieben wurden. Diese Daten werden schließlich über das Frontend an den internen DS zurückgegeben, nachdem alle späteren Savepoints, Datenbank-Locks und -Updates gelöscht wurden.

Soll eine Transaktion abgebrochen werden, so übergibt das Frontend diese Anforderung wieder an ein Backend, das alle Savepoints, Datenbank-Updates und Datenbank-Inserts dieser Transaktion löscht und alle Datenbank-Locks freigibt. Beim Festschreiben einer Transaktion führt das Backend zunächst die gespeicherten Datenbank-Updates und -Einfügungen durch und persistiert sie, wobei auch alle Locks freigegeben werden. Danach werden alle Savepoints dieser Transaktion gelöscht und mit den übergebenen Ausgangsdaten der finale Savepoint der Transaktion geschrieben.

Soll eine Kompensations-Maßnahme durchgeführt werden, so wird ein Backend-Server beauftragt, den finalen Savepoint der zugehörigen Transaktion zu lokalisieren und zurückzugeben. Beim Festschreiben der Kompensation werden alle Datenbank-Aktionen ausgeführt, alle Locks freigegeben und alle expliziten Savepoints gelöscht. Anschließend wird der finale Savepoint der kompensierten Transaktion entfernt. Beim Abbruch der Kompensation werden alle Transaktionsdaten gelöscht. Die Datenbanken befinden sich danach in einem möglicherweise inkonsistenten Zustand, der vom Operator aufgelöst werden muß.

Wenn der Subworkflow beendet werden soll, wird das dem externen Server ebenfalls mitgeteilt. Das Frontend löscht dann alle verbliebenen Savepoints und Kommunikations-Daten. Erhält der Server keine Nachricht vom Ende des Subworkflow, so muß er die Transaktionsdaten dieses Subworkflow „für alle Ewigkeit“ speichern. Die übriggebliebenen Daten von fehlgeschlagenen Subworkflows müssen demnach zu einem späteren Zeitpunkt manuell aus dem System entfernt werden.

5.6.4 Datenbankanbindung

Die Datenbankanbindung konnte aufgrund der Situation auf dem für diese Diplomarbeit zur Verfügung stehenden z/OS-Rechner nicht implementiert werden. Deshalb ist die nachfolgende Beschreibung als Konzept anzusehen, dessen Umsetzbarkeit zwar plausibel, aber noch nicht überprüft worden ist.

Derzeit werden vom externen Server ausschließlich Datenbanken unterstützt, die sich per SQL über das ODBC-Interface ansprechen lassen. Dies spiegelt sich auch in der Interface-Beschreibung wieder, die für jeden Zugriff auf eine Datenbank angelegt werden muß (siehe Unterabschnitt 2.2.2 ab Seite 15).

Dort ist neben den an die SQL-Syntax angelehnten Informationen für den Zugriff auf die Datenbank auch angegeben, ob gelesene Daten in der Folge verändert werden sollen (`mode="update"`), ob sie bis zum Ende der Transaktion konsistent sein müssen (`mode="read"`), oder ob nur ihr augenblicklicher Wert benötigt wird (`mode="fetch"`). Diese Information wird genutzt, um ein effizientes und sicheres Locking der zwischengespeicherten Daten aus Datenbank-Anfragen zu gewährleisten.

In Tabelle 5.2 auf Seite 53 ist das im folgenden beschriebene Verhalten des Servers tabellarisch aufgeführt, um eine bessere Übersicht zu ermöglichen.

5.6.4.1 Locking

Die folgenden Überlegungen basieren auf der Dokumentation zum relationalen Datenbanksystem *DB2* von IBM [39, 40, 41]. Da jedoch nur Elemente der SQL-Sprache dieser Datenbank verwendet wurden, können die Ergebnisse leicht an andere SQL-Datenbanksysteme angepaßt werden. Das Zustandsdiagramm in Abbildung 5.6 soll die Vorgänge beim Locking verdeutlichen.

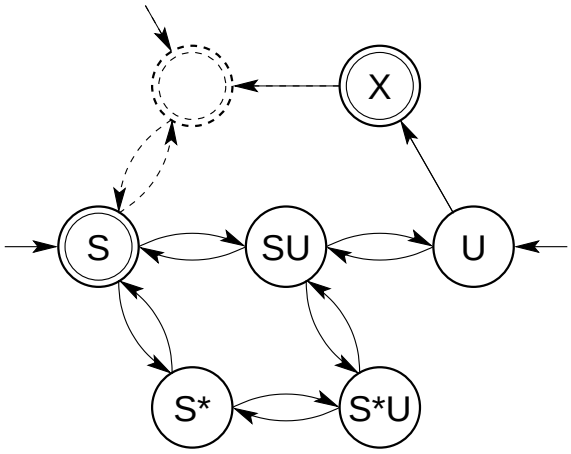


Abbildung 5.6: Zustandsdiagramm für das Datenbank-Locking

Für Daten im Cache, die mit dem Modus `fetch` angefordert wurden, wird kein Lock gesetzt, beim `read`-Modus wird ein geteilter Lock (S von *Shared*) und beim `update`-Modus ein angekündigter exklusiver Lock[17] (U von *Update*) gesetzt. Ein exklusiver Lock (X von e*X*clusive) kann nur gesetzt werden, wenn zuvor bereits ein U-Lock desselben Subworkflow existiert. X-Locks werden im Normalfall[18] nur gesetzt, während die betroffenen Daten in eine Datenbank zurückgeschrieben werden, anschließend werden diese Locks freigegeben und die Daten aus dem Cache gelöscht.

Ein X-Lock auf Cache-Daten bewirkt, daß alle Anfragen, die nicht den Modus `fetch` haben, warten müssen. Auf einen U-Lock müssen nur weitere U-Locks und X-Locks warten, während beliebig viele S-Locks gesetzt werden dürfen. Auf S-Locks schließlich müssen nur X-Locks warten, während ein U-Lock und beliebig viele S-Locks hinzukommen dürfen.

Auf den Datenbanken wird normalerweise für einzeln angeforderte Daten die Lock-Granularität[19] `CS` (*Cursor Stability*) verlangt, also ein Locking auf Datensatz-Ebene, das die größte Nebenläufigkeit von Datenbank-Zugriffen erlaubt. Für arrayweise gelesene Daten wird dagegen die Lock-Granularität `RS` (*Read Stability*) gewählt, also tabellenweises Locking mit der Einschränkung, daß von der Anfrage nicht betroffene Daten geändert und neue hinzugefügt werden können. Normalerweise nicht verwendet wird die Lock-Granularität `RR` (*repeatable read*), die eine komplette Tabelle gegen jede Veränderung sperrt.

Die Lock-Granularität läßt sich durch die Parameter-Beschreibung für den externen Server vorwählen oder durch den Submodus in jeder einzelnen Datenbank-Anfrage ändern. Dieser Submodus ist dabei entweder `concurrent` oder `isolated`, wobei der erste eine kleinere Lock-Granularität wählt (etwa CS statt RS) und der zweite eine größere (RR statt RS). Die Lock-Granularität wird wie auch der Lock-Modus im Cache eingetragen, wobei größere Locks Vorrang vor kleineren haben.

[17]Dieser Lock-Typ erhöht zwar die „Concurrency“, erhöht aber gleichzeitig auch die Gefahr von Deadlocks. Er wurde hier trotzdem eingeführt, da durch die Allokations-Strategie des Subworkflow-Compilers die meisten Deadlocks vermieden werden können.

[18]In dem Fall, daß der externe Server exklusiven Zugriff auf eine Datenbank hat, wird der X-Lock gesetzt, während die betroffenen Daten verändert werden, und danach wieder freigegeben.

[19]In der Dokumentation zur DB2 wird die Lock-Granularität als *isolation level* bezeichnet.

Alle Locks, die im Cache gesetzt werden, müssen auch auf die Datenbanken übertragen werden, um die Konsistenz zu wahren. Um trotzdem nicht bei jedem Cache-Zugriff einen Datenbank-Zugriff durchführen zu müssen, gibt der externe Server nur *U*-Locks (über `SELECT ... FOR UPDATE`) und neue *S*-Locks (per `SELECT ... FOR FETCH ONLY`) oder eine Erhöhung der Lock-Granularität (über `SELECT ... WITH RR/RS/CS`) an die Datenbanken weiter und verwaltet duplizierte *S*-Locks intern. *X*-Locks werden von der Datenbank während eines Update-Vorgangs selbständig angelegt.

Allerdings muß der externe Server sicherstellen, daß bei einem Commit nur solche Locks freigegeben werden, die von keinem Subworkflow mehr benötigt werden. Dazu führt er jedes `SELECT ... FOR FETCH ONLY` in einem eigenen Kontext aus, wohingegen alle `SELECT ... FOR UPDATE` eines Subworkflow in einem gemeinsamen Kontext ausgeführt werden, in dem schließlich beim Commit auch die Updates ausgeführt werden.

5.6.4.2 Lesende Datenbank-Zugriffe

Erhält der externe Server den Auftrag, bestimmte Daten aus einer Datenbank bereitzustellen, so überprüft das Frontend zunächst, ob die angeforderten Daten bereits im Cache liegen.

Ist das der Fall, so wird in der Anfrage nachgesehen, ob die angeforderten Daten gelocked werden sollen. Wenn nicht, werden sie direkt zurückgegeben. Ansonsten werden sie an ein Backend übergeben, das versucht, den entsprechenden Lock zu setzen, und eventuell auf eine Freigabe des Locks wartet. Kommt es dabei zu einem Deadlock, so wird dieser erkannt und durch das Abbrechen einer der beteiligten Anfragen aufgelöst.[20] Konnte der Lock gesetzt werden, dann werden die angeforderten Daten aus dem Cache gelesen und an den internen DS übergeben.

Wenn die Daten noch nicht im Cache vorhanden sind wird die Anfrage ebenfalls an ein Backend weitergereicht. Dieses führt den entsprechenden Datenbankzugriff aus, legt die Daten persistent und mit dem geforderten Lock-Modus im Cache ab und reicht sie über das Frontend an den internen DS weiter.

Bei einer mengenorientierten Datenbank-Abfrage, die ein Feld von Ergebnissen liefert, werden alle Ergebnisdaten gemeinsam aus der Datenbank gelesen (ODBC-Funktion `SQLExtendedFetch`) und in einer Feldvariablen zurückgeliefert.

5.6.4.3 Schreibende Datenbank-Zugriffe

Vom internen DS überstellte Änderungen von Datenbankdaten werden an ein Backend weitergereicht, das sie auf offensichtliche Fehler überprüft, persistent und transaktionsgebunden im Cache ablegt und dann zurückkehrt. Erst bei einer Commit-Anforderung an den externen Server, die vom Frontend ebenfalls an ein Backend durchgereicht wird, werden die gesicherten Änderungen an die Datenbanken übertragen. Ein Fehler hierbei führt zum Fehlschlag des Commit und damit zum Abbruch oder Neustart der Transaktion.

[20] Das Erkennen eines Deadlock geschieht entweder durch Überprüfen der bereits gesetzten Locks oder durch ein Timeout beim Warten. Eine abgebrochene Datenbankabfrage bewirkt im allgemeinen, daß die aktuelle Transaktion des betreffenden Subworkflow zurückgesetzt wird. Dieses Verhalten läßt sich aber über die Parameter des Subworkflow und der Transaktion genau einstellen.

Aktion[a]	Eintrag im Datenbank-Cache				Datenbank-Locks					DB-Zugriff	Backend aufrufen
	anleg.	lesen	ändern	löschen	setzen S/U	setzen X	löschen S	löschen U	löschen X		
Daten lesen	•				•					•	•
		•			•						
Daten updaten				•		•		•	•	•	•
... bei exkl. Zugriff			•			•		•	•		
Daten einfügen										•	•
... bei exkl. Zugriff				(•)[b]						•	•

[a] Beim Update und beim Einfügen neuer Daten sind die Aktionen angegeben, die beim Commit ausgeführt werden.

[b] Ein Eintrag für das Ergebnis einer mengenorientierten Anfrage wird gelöscht, wenn es Daten aus dem gerade geschriebenen neuen Datensatz enthalten könnte.

Tabelle 5.2: Ausführung von Anfragen zum Datenbank-Zugriff

Für Updates bestehender Datensätze muß vor dem Übertragen an die Datenbanken versucht werden, den bereits existierenden U-Lock in einen X-Lock umzuwandeln. Ist das nicht möglich, weil noch S-Locks auf den Datensatz existieren, so muß das Backend warten, bis diese freigegeben wurden. Für das Einfügen neuer Daten ist dieser Schutz nicht nötig.

Alle gespeicherten Savepoints und alle Datenbank-Updates einer Transaktion können nach einem erfolgreichen Commit gelöscht werden, bei dem auch alle Datenbank-Locks freigegeben werden.

Auch wenn die Transaktion abgebrochen oder zurückgesetzt wird, zum Beispiel, weil das Festschreiben fehlgeschlagen ist, werden alle Savepoints und Datenbank-Updates gelöscht und alle Locks freigegeben. Beim Zurücksetzen auf einen explizit gesetzten Savepoint werden alle Savepoints, Datenbank-Updates und Locks verworfen, die nach diesem Savepoint geschrieben wurden.

5.6.4.4 Optimierungen

Zur Optimierung der Commit-Zeiten können die Commits von mehreren parallel von verschiedenen Subworkflowcontrollern ausgeführten technischen Transaktionen gesammelt und dann gemeinsam in die Datenbanken geschrieben werden. Dieses Verfahren entspricht einem verzögerten Group-Commit gegenüber den Datenbanken. Solange die Daten von den gesammelten Transaktionen noch nicht zurückgeschrieben wurden, müssen dann allerdings auch die Locks auf diese Daten gehalten werden, was zu Verzögerungen von anderen Anwendungen führen kann, die ebenfalls auf diese Daten zugreifen wollen. Außerdem erfordert dieses Vorgehen eine zu den Locks auf Datenbank-Einträgen im Cache asynchrone Verwaltung der Datenbank-Locks.

Deshalb werden Commit-Anforderungen nur dann verzögert, wenn die betroffenen Datenbanken dem externen Server des Datenzugriffsdienstes exklusiv zur Verfügung stehen. Für diese Datenbanken wird das Datenbank-Locking ausschließlich im Cache durchgeführt, und Datenbank-Updates werden direkt in die Cache-Einträge geschrieben. Außerdem werden gelesene Daten auch dann im Cache behalten, wenn keine Locks auf sie existieren. Sie gelten dann aber als flüchtig und werden durch den Cache-Manager freigegeben, wenn der Speicher im Cache knapp wird.

Für die Ausführung von Group-Commits ist ein weiteres Verzeichnis im Cache erforderlich, in dem die verzögerten Updates und Neueintragungen abgelegt werden. Updates werden vor dem Eintrag in dieses Verzeichnis im Cache ausgeführt, wobei die Locks im Cache entsprechend angepaßt werden. In regelmäßigen Abständen werden alle Einträge im Group-Commit-Verzeichnis an die externen Datenbanken weitergegeben, um das Wiederherstellen des Cache nach einem Absturz des Servers zu erleichtern, und auch, um den Cache nicht ins Unermeßliche anwachsen zu lassen.[21]

Damit verzögerte Neueintragungen von Datensätzen in mengenorientierten Datenbank-Abfragen aufgenommen werden können, werden diese in einem weiteren neuen Cache-Verzeichnis notiert. Dieses Verzeichnis wird vor der Ausführung eines mengenorientierten Datenbank-Zugriffs überprüft. Wurde dabei ein Datensatz gefunden, der für den mengenorientierten Zugriff relevant ist, so wird vor der Ausführung des lesenden Zugriffs das Group-Commit ausgeführt. Das Ändern mengenorientierter Abfragen im Cache wäre zu teuer und wird deshalb unterlassen.

Durch das in diesem Abschnitt beschriebene Vorgehen läßt sich eine deutlich höhere Performance für Datenbankzugriffe erreichen, wenn die meisten oder alle Datenbanken dem Server exklusiv zur Verfügung stehen. Stehen nur wenige Datenbanken exklusiv zur Verfügung, bringt das Group-Commit nicht mehr sehr viel, die Beschleunigung wiederholter lesender Zugriffe auf bestimmte Daten bleibt aber erhalten.

5.6.5 Programm-Objekte

Bei der Anforderung eines Programm-Objekts (SWF-Runtime, Aktivitäten) durch den internen Datenzugriffsdienst überprüft das Frontend zunächst, ob dieses Objekt bereits geladen und im lokalen Dateisystem abgelegt wurde. In diesem Fall wird dem internen DS sofort mitgeteilt, daß das Objekt vorhanden ist, wobei für COBOL-Programme die Position im Filesystem mit übergeben wird.

Ansonsten wird die Anfrage an ein Backend weitergeleitet, von dem das angeforderte Objekt aus dem zentralen Verzeichnis des Business Process Designer geladen, lokal abgelegt und im Cache verzeichnet wird. Das Verzeichnis, in das das Objekt geschrieben wird, ist in der Konfiguration des Datenzugriffsdienstes festgelegt, und hängt davon ab, ob es sich bei dem Objekt um eine Subworkflow-Runtime, um eine Ein- oder Ausgabedaten-Klasse, um eine Exception, um eine persistente oder nicht persistente Java-Aktivität oder um eine COBOL-Aktivität handelt. Über das Frontend wird anschließend das Ergebnis, bei COBOL-Programmen wieder unter Angabe der Position, dem internen DS mitgeteilt.

[21] Man könnte theoretisch auf automatische Group-Commits verzichten und diese nur durchführen, bevor der Server heruntergefahren wird. Außerdem könnte der Server bestimmte Datenbanken auch vollständig in seinem Cache simulieren. Sinnvoller wäre in diesem Fall aber wohl die Integration des Servers in eine bestehende Datenbank als erweiterte Funktionalität.

Kapitel 6

Die Einbindung des SWFC in verschiedene Carrier

Da der Subworkflowcontroller (*SWFC*) unter verschiedenen Carriern zum Einsatz kommen soll, muß er an die Eigenheiten und Fähigkeiten des jeweiligen Carrier angepaßt werden. Solche Anpassungen sollten generell auf möglichst wenige Komponenten beschränkt sein und so weit wie möglich gekapselt sein, damit einerseits neue Anpassungen leichter möglich sind und andererseits möglichst wenig Logik in den angepaßten Bereichen dupliziert wird.

Für den SWFC wurden die nötigen Anpassungen an verschiedene Carrier auf den I/O-Controller und einen Call-Stub beschränkt, die für jeden Carrier neu implementiert werden müssen. Ihre Position im Gesamtsystem ist in Abbildung 6.1 dargestellt.

Die Funktionalität des Call-Stub beschränkt sich darauf, die Parameter für den SWFC einzulesen, den passenden I/O-Controller zu starten und die Kontrolle an den Subworkflow-Überwacher (*SWFS*) weiterzugeben. Außerdem beendet er den Subworkflowcontroller, wenn der SWFS die Kontrolle zurückgibt. Der Call-Stub wird nie aus dem SWFC heraus angesprochen.

Der I/O-Controller dient ausschließlich der Kommunikation mit dem Carrier. Er erweitert die abstrakte Klasse `IOController`, um transparent von allen Komponenten des Subworkflowcontrollers angesprochen werden zu können. Diese abstrakte Klasse bietet die folgenden (abstrakten) Methoden, die von der jeweiligen Anpassung des I/O-Controllers an die verschiedenen Carrier implementiert werden.

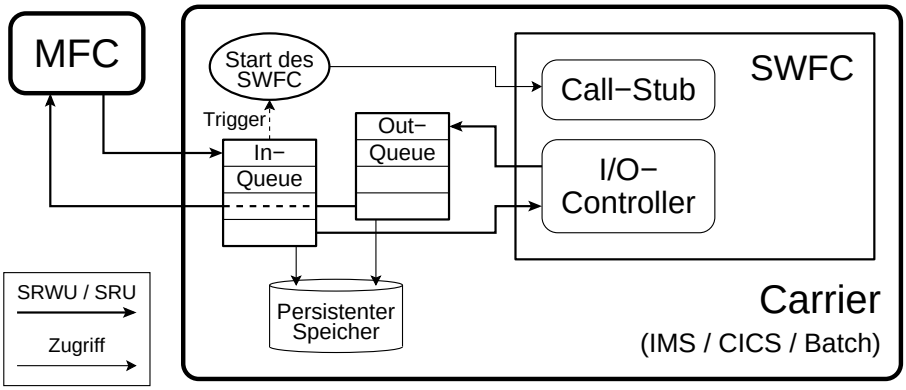


Abbildung 6.1: Einbindung des SWFC in die verschiedenen Carrier

`getSRWU` liefert eine neue SRWU zurück und `putSRU` übergibt eine SRU an den Carrier. `abortTransaktion` wird aufgerufen, wenn eine Transaktion abgebrochen oder zurückgesetzt wird, und sorgt dafür, daß noch nicht versendete Meldungen verworfen werden, während `commitTransaction` dafür sorgt, daß bereits übergebene Meldungen versendet werden. Per `endSubworkflow` wird dem Carrier das Ende des Subworkflow mitgeteilt.

Alle anderen Komponenten des SWFC (inklusive des externen Servers des Datenzugriffsdienstes) enthalten keinen von einem Carrier abhängigen Code, so daß sie unverändert mit jedem Carrier eingesetzt werden können. Geplant ist derzeit der Einsatz des SWFC unter den Carriern IMS, CICS und Batch. Die Einbindung in diese Carrier wird in den folgenden Abschnitten beschrieben.

Tatsächlich implementiert ist nur die Einbindung des SWFC in den Batch-Carrier, da die Java-Interfaces von CICS und IMS auf dem für diese Arbeit zur Verfügung gestellten Rechner nicht eingerichtet werden konnten.

6.1 Einbindung unter IMS

Der Subworkflowcontroller läuft unter IMS in einer Java Message Processing region (*JMP*) (siehe dazu die online verfügbare Literatur zu IMS im allgemeinen [42, 43, 44, 45, 46, 47] und IMS und Java im besonderen [48, 49, 50, 51]). Dadurch kann IMS die Verteilung der SRWUs auf mehrere SWFCs transparent durchführen.[1] Der komplette Vorgang bei der Ausführung des SWFC unter IMS ist in Abbildung 6.2 dargestellt, die dünnen Pfeile stehen hierbei für den Programmfluß.

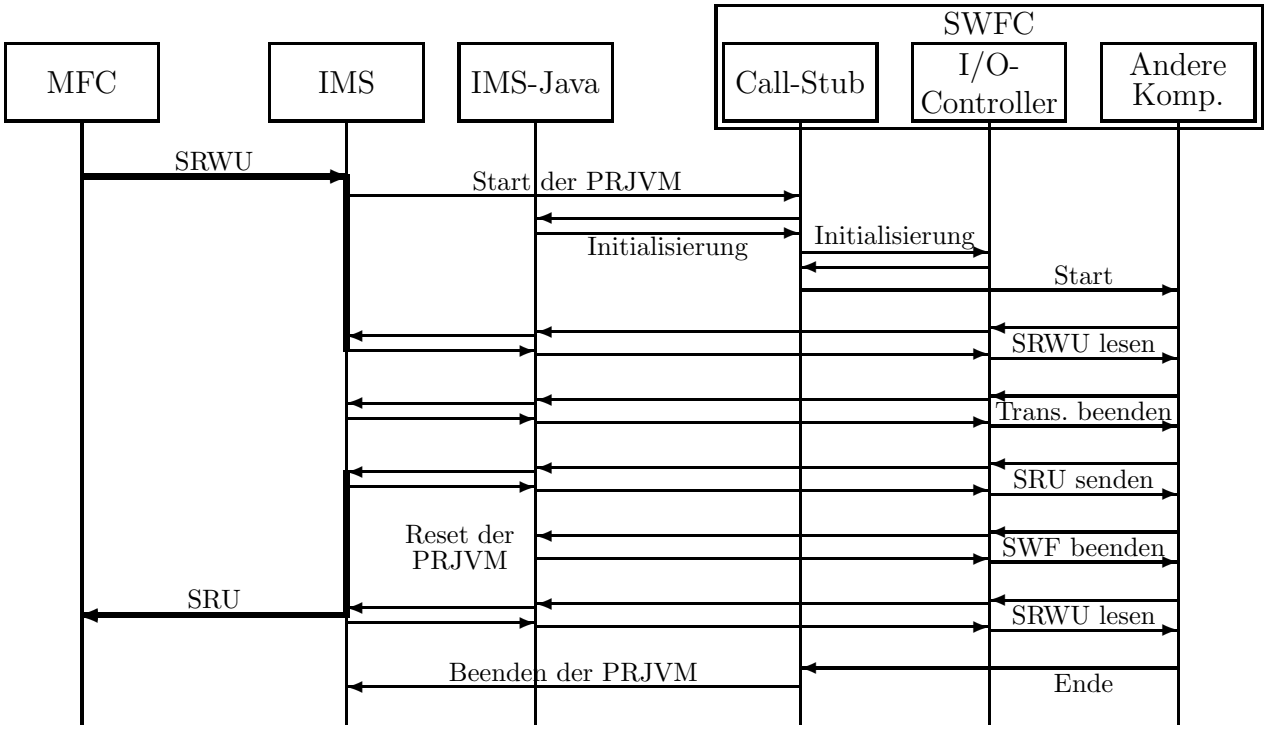


Abbildung 6.2: Ausführung des SWFC unter IMS

[1] IMS greift für die Verteilung der eingehenden Transaktionsanforderungen, wie in [52] ab Seite 171 geschildert, auf die Workload Manager Services von z/OS [38] zurück.

6.1.1 Ausführung von Java-Programmen durch IMS

Wenn IMS einen neuen Auftrag für eine JMP erhält, schaut es nach, ob schon genügend Instanzen dieses Programms laufen. In diesem Fall wird der Auftrag in die In-Queue einer JMP geschrieben. Ansonsten wird eine neue MPR erzeugt, in der die PRJVM gestartet wird. Anschließend wird die `Main()`-Methode einer Klasse aufgerufen, die die abstrakte Klasse `IMSApplication` implementieren muß. Diese initialisiert IMS-Java durch Aufruf der `begin()`-Methode der abstrakten Basisklasse, die schließlich die Kontrolle an die `doBegin()`-Methode und damit wieder an das eigene Programm übergibt.

Im eigentlichen IMS-Java-Programm wird in einer Schleife ein neuer Auftrag mit `IMSMessage Queue.getUniqueMessage()` (dies entspricht dem *DL/I*-Kommando *GU*) aus der IMS-Queue gelesen, bearbeitet, die Ergebnisse über `IMSMessageQueue.insertMessage()` (*ISRT*) wieder in die IMS-Queue geschrieben und per `IMSTransaction.commit()`[2] freigegeben.

6.1.2 Anpassungen des Subworkflowcontrollers an IMS

Für die Integration des Subworkflowcontrollers in IMS implementiert der IMS-Call-Stub die `IMSApplication`-Klasse. In seiner `doBegin()`-Methode lädt er die Parameter des SWFC, startet den I/O-Controller für IMS und ruft dann den SWFS auf, dem er die ge'parsed'en Parameter und den I/O-Controller übergibt.

Um neue SWF-SRWUs zu lesen oder Ergebnisse und Meldungen (in der Form von Ergebnis-, Fehler- und Meldungs-SWF-SRUs) zurückzugeben, greift der I/O-Controller auf die IMS-Queues zu. Am Ende jeder Transaktion führt er ein `JavaToDLI.execute("CHKP", ...)` aus, um Nachrichten an den Masterflowcontroller freizugeben, und am Ende des Subworkflow wird mit `IMSTransaction.commit()` ein Reset der PRJVM durchgeführt. Wird eine Transaktion abgebrochen oder zurückgesetzt, so werden über `JavaToDLI.execute("ROLB", ...)` noch nicht versendete Meldungen verworfen.

6.2 Einbindung unter CICS

Wird CICS als Carrier eingesetzt, so läuft der Subworkflowcontroller als JCICS-Applikation unter der Kontrolle von CICS (siehe dazu die Dokumentation zu CICS [53, 54, 55, 56] und zu CICS und Java [57]). CICS kann mehrere Java-Applikationen parallel ausführen, und abhängig vom aktuellen Workload neue JVMs starten oder nicht mehr gebrauchte beenden.

6.2.1 Ausführung von Java-Programmen durch CICS

Wird für einen neuen Auftrag eine neue Instanz der Applikation benötigt, so wird eine neue Region mit einer eigenen JVM gestartet. Die JVM wird nun vom CICS-eigenen Launcher angewiesen, die `main(CommAreaHolder ca)`-Methode der angegebenen Klasse aufzurufen.

[2] Ein Commit ist im DL/I-Interface nicht vorgesehen, dort wird diese Funktion von *GU* mit übernommen. Die Commit-Funktion von IMS-Java wurde eingeführt, um das Programmiermodell von IMS-Applikationen beibehalten zu können und trotzdem die PRJVM nach jedem Auftrag neu zu starten.

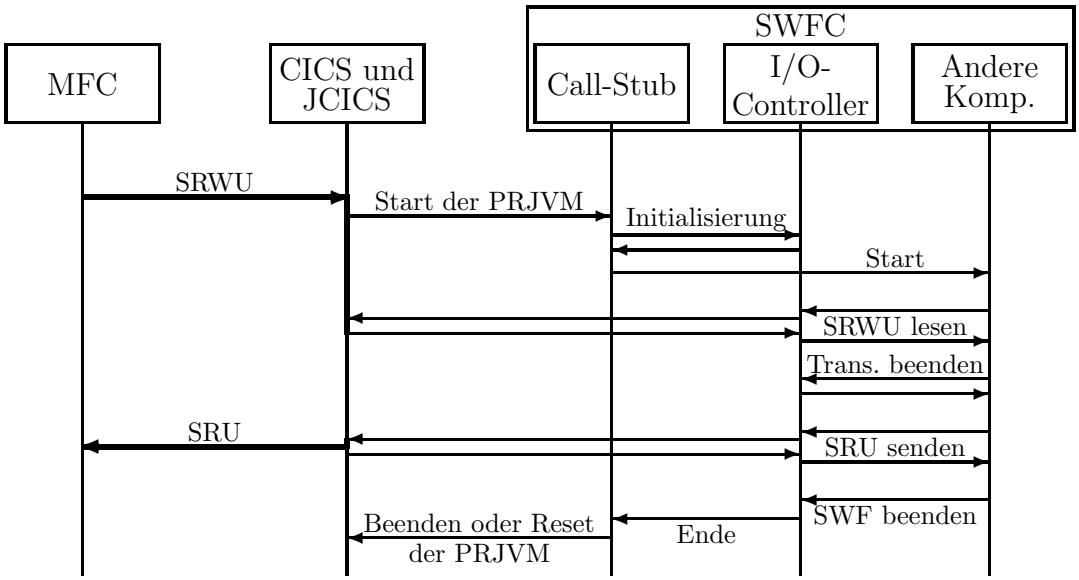


Abbildung 6.3: Ausführung des SWFC unter CICS

Ein- und Ausgabedaten von CICS-Programmen können über zwei verschiedene Kanäle laufen: Entweder über die sogenannte *CommArea* der Applikation oder über ein CICS Terminal. Die erste Methode erlaubt die Übergabe von Parametern zu Beginn eines Programms und von Ergebnissen an dessen Ende, die zweite Methode erlaubt eine vollkommen flexible Kommunikation.

CICS-Applikationen beenden sich nach getaner Arbeit. Dieses Programmiermodell wurde auch für JCICS-Applikationen übernommen. Der Reset der PRJVM findet deshalb direkt nach Beenden der Applikation statt.

6.2.2 Anpassungen des Subworkflowcontrollers an CICS

Der Subworkflowcontroller wird in CICS integriert, indem der Call-Stub die von CICS aufgerufene `main(CommAreaHolder ca)`-Methode enthält und von dieser aus den SWFS startet, wobei diesem die eingelesenen Parameter und der passende I/O-Controller übergeben werden.

Der I/O-Controller verwendet für das Einlesen der SWF-SRWU und für die Rückgabe der Ergebnis- und Fehler-SRUs die CommArea. Meldungen werden über das Standard-CICS-Terminal per `TerminalPrincipalFacility.send()` an den Masterflowcontroller gesendet. Das Ende einer Transaktion wird unter CICS ignoriert, da alle Meldungen sofort an den MFC weitergegeben werden.

Am Ende des Subworkflow muß die Kontrolle über den Call-Stub an CICS zurückgegeben werden. Das wird durch das Werfen eines speziellen Java-Errors erreicht, der vom Call-Stub abgefangen wird. Der SWFC muß unter CICS für jeden Subworkflow neu gestartet werden, seine Klassen bleiben jedoch aufgrund der Eigenschaften der PRJVM (siehe Abschnitt 3.3 ab Seite 23) geladen und initialisiert.

In Abbildung 6.3 sind alle den Carrier betreffenden Vorgänge beim Ausführen des SWFC unter CICS dargestellt. Die dicken Pfeile sind Transportvorgänge der SRWU und der SRU, die dünnen symbolisieren die Weitergabe der Kontrolle an eine andere Komponente.

6.3 Einbindung als Batch-Job

Um ein Java-Programm von der PRJVM (siehe Abschnitt 3.3 ab Seite 23) ausführen zu lassen, wird ein sogenannter *Launcher* benötigt, der die PRJVM initialisiert und die auszuführende Applikation lädt und startet.

Ein solcher Launcher ist in IMS und CICS bereits integriert, für den Batch-Job wurde diese Komponente im Rahmen dieser Diplomarbeit erstellt. Der Batch-Launcher übernimmt dabei auch die Funktion eines Carriers, auf den vom Masterflowcontroller zugegriffen wird und der die Subworkflowcontroller verwaltet.

Im folgenden werden die nötigen Anpassungen des Subworkflowcontrollers, der Aufbau und die Funktionsweise des Launchers sowie das Interface zum Masterflowcontroller beschrieben. In Abbildung 6.4 auf Seite 60 ist das Zusammenspiel von MFC, Batch-Carrier und SWFC dargestellt, wobei die dicken Pfeile den Weg von SRWU und SRU markieren und die dünnen Pfeile den Kontrollfluß verdeutlichen.

6.3.1 Anpassungen des Subworkflowcontrollers

Der Call-Stub für den Batch-Betrieb liest wie üblich die Konfigurations-Parameter ein, startet den passenden I/O-Controller und übergibt dann die Kontrolle an den Subworkflow-Überwacher. Der I/O-Controller kommuniziert mit dem Launcher über Message Queues, die in einer Shared Memory Region verwaltet werden.

Über diese Queues werden neue Aufträge geholt bzw. Meldungen und Ergebnisse übergeben. Am Ende jeder Transaktion werden über einen Commit-Vorgang vom Typ `CHKP` die übertragenen Meldungen für die Weitergabe an den Server freigegeben. Wird die Transaktion zurückgesetzt oder abgebrochen, so werden über einen Commit-Vorgang vom Typ `ROLB` bis dahin übergebene und noch nicht versendete Meldungen verworfen.

Am Ende des Subworkflow führt der I/O-Controller per `ResetJavaVM()` den Reset der PRJVM durch und sorgt über einen Commit-Vorgang vom Typ `COMMIT` dafür, daß der Subworkflow als erfolgreich abgeschlossen notiert wird.

Falls der Subworkflow nicht ordnungsgemäß beendet werden konnte, wird er über einen Commit-Vorgang vom Typ `ABORT` beendet. Dabei werden wieder alle noch nicht versendeten Meldungen verworfen, außerdem wird der Subworkflow dann nicht als ausgeführt vermerkt und kann so bei einer Wiederholung nochmals ausgeführt werden.

Für seine Aufgaben greift der I/O-Controller durchweg auf JNI-Routinen zurück, da weder ein Reset der PRJVM noch die Kommunikation über Shared Memory Regions unter Java möglich sind.

6.3.2 Der Launcher

Da die PRJVM auf den Unix System Services (*USS*) [15] aufbaut, und da auch die vollständigen POSIX-Fähigkeiten unter z/OS nur bei Einbindung der USS zur Verfügung stehen, wurde der Launcher für den Batch-Job als USS-Programm implementiert. Wie alle USS-Programme kann der Launcher entweder über eine Shell aus der USS-Umgebung heraus oder über JES [58, 59] und TSO [60] mit Hilfe des BPXBATCH-Utility [17] gestartet werden.

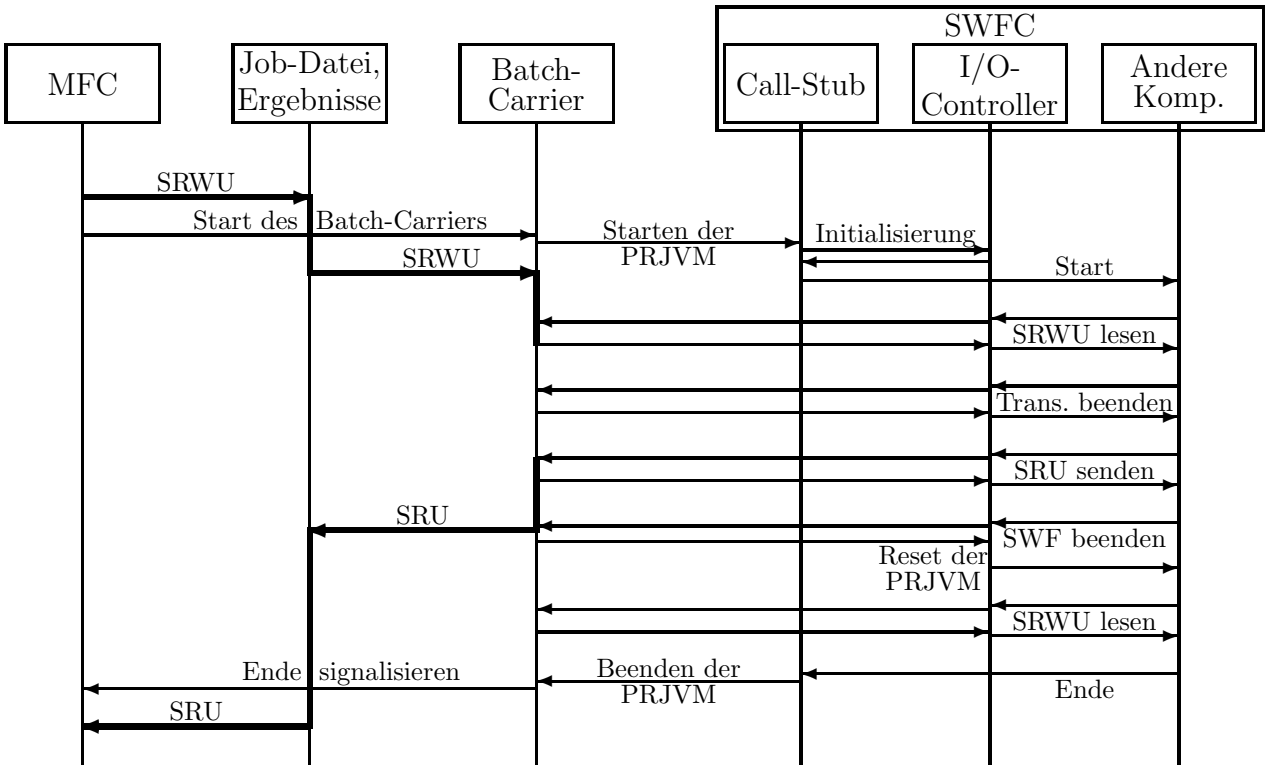


Abbildung 6.4: Ausführung des SWFC durch den Batch-Carrier

Nach seinem Start liest der Launcher zunächst eine Datei mit seinen Konfigurations-Parametern[3] ein und wertet diese aus. Dort sind unter anderem Angaben über die Ein- und Ausgabekanäle zur Kommunikation mit dem Masterflowcontroller (Datei, UNIX-Socket, Pipe oder TCP-Socket), die Klassen-Pfade der PRJVM und die aus Port und Hostname[4] bestehende Adresse des Datenzugriffsdienstes abgelegt.

Nachdem der Launcher seine Parameter ausgewertet hat, öffnet er das Log-File für Meldungen des Launchers und das Savefile, in dem erfolgreich abgeschlossene Subworkflows vermerkt werden, und initialisiert die Shared Memory Region für die Kommunikation mit den Subworkflowcontrollern.

Die Shared Memory Region enthält für diesen Zweck zwei Message Queues: Eine für Aufträge an die Subworkflowcontroller (SRWUs), und eine für Ergebnisse und Meldungen derselben (SRUs). Außerdem enthält die Shared Memory Region Felder für ein synchronisiertes Commit zuvor übergebener Meldungen und Ergebnisse. Alle Queues und der Commit-Bereich werden über Semaphoren geschützt, der Zugriff auf die Queues wird außerdem über Semaphoren nach dem Producer-Consumer-Prinzip synchronisiert.

[3]Die Position der Konfigurations-Datei wird dem Launcher beim Aufruf als Kommandozeilen-Parameter übergeben. Über die Kommandozeile können auch einige der Parameter übergeben werden, die normalerweise in die Konfigurations-Datei eingetragen werden. Kommandozeilen-Parameter haben dabei Vorrang vor den Einträgen der Konfigurations-Datei.

[4]Da sich Sockets, wie in Kapitel 7 ab Seite 63 beschrieben, als wenig performant herausgestellt haben, wird empfohlen, statt dessen System V Message Queues einzusetzen. In diesem Fall ist in der Konfigurationsdatei statt Hostname und Port des Socket die eindeutige ID der Message Queue angegeben.

Im nächsten Schritt initialisiert der Launcher die PRJVM und startet einen oder mehrere Subworkflowcontroller in jeweils eigenen JVMs,[5] indem er in jeder JVM den Call-Stub aufruft. Anschließend öffnet der Launcher die Ein- und Ausgabe-Kanäle, liest neue Aufträge (SRWUs) aus dem Eingabekanal und stellt sie in die SRWU-Queue im Shared Memory Segment, aus der sie die Subworkflowcontroller über ihren I/O-Controller entnehmen können.

Nachrichten und Ergebnisse der Subworkflows (Meldungs-, Fehler- und Ergebnis-SRUs) werden vom Launcher aus der SRU-Queue im Shared Memory Segment entnommen und unter der Subworkflow-Kennung zwischengespeichert. Bei dem am Ende jeder Transaktion und am Ende des Subworkflow vom I/O-Controller initiierten Commit-Vorgang werden alle Ressourcen des Subworkflow freigegeben, und abhängig vom Typ des Commit-Vorgangs die zwischengespeicherten SRUs an den Ausgabe-Kanal übergeben (`CHKP` und `COMMIT`) oder verworfen (`ROLB` und `ABORT`).[6] Bei einem Commit-Vorgang vom Typ `COMMIT` wird zudem der erfolgreiche Abschluß der Transaktion im Savefile vermerkt.

6.3.3 Kommunikation mit dem Masterflowcontroller

Zur Kommunikation des MFC mit dem Launcher ist eine eigene Komponente vorgesehen, der Batch-Konnektor. Dieser kennt zwei prinzipiell verschiedene Betriebsmodi. Wird er online eingesetzt, so kommuniziert er direkt mit einem Launcher, während beim Offline-Einsatz der Informationsaustausch über Dateien oder Datenbanken abläuft.

Die Online-Kommunikation kann dabei entfernt per TCP-Socket oder lokal per UNIX-Socket bzw. Named Pipe erfolgen. In beiden Fällen kann der Launcher auch durch den Batch-Konnektor gestartet werden, wenn das erforderlich ist. Bei der Online-Kommunikation müssen der Launcher und der Batch-Konnektor über ein Drei-Phasen-Protokoll sicherstellen, daß alle Aufträge auch in Fehlersituationen sicher ausgeführt werden.

Die Offline-Kommunikation erfolgt entweder über Dateien, wobei der Batch-Konnektor vom Launcher über eine System V Semaphore informiert wird, sobald alle Aufträge aus der Datei bearbeitet sind. Oder die Aufträge und Ergebnisse werden in einer Datenbank abgelegt, deren Locking-Mechanismus für die Synchronisation mit dem Launcher eingesetzt wird. In jedem Fall unterhält der Launcher hierbei ein Savefile, aus dem er im Fehlerfall entnehmen kann, an welcher Stelle er die Bearbeitung fortsetzen muß.

Der Batch-Konnektor wurde im Rahmen dieser Arbeit nicht implementiert, da der Masterflowcontroller noch nicht spezifiziert wurde. Im Launcher wurde außerdem nur die Offline-Kommunikation über Dateien und die Synchronisation mit dem Batch-Konnektor über Semaphoren implementiert.

[5] Tatsächlich wird vom Launcher ein JVM-Set verwaltet, indem zunächst ein JVM-Master gestartet wird, dem dann ein oder mehrere JVM-Worker zugeordnet werden.

[6] Dieses Vorgehen bewirkt, daß Meldungs-SRUs verloren gehen können, wenn nach ihrer Ausgabe durch die SWFR ein Savepoint geschrieben wurde und der Subworkflow danach durch einen Systemfehler abgebrochen wird und neu gestartet werden muß. Da nach dem Neustart die SWFR beim zuletzt geschriebenen Savepoint fortfährt, wird die durch den Systemfehler ebenfalls verlorengegangene SRU nicht mehr erzeugt. Wie am Ende von Unterabschnitt 2.2.3 ab Seite 17 erläutert, ist dieses Verhalten durch die transaktionale Semantik von Meldungs-SRUs gedeckt.

Kapitel 7

Performance-Analyse

An dieser Stelle soll das in den vorangegangenen Kapiteln vorgestellte Design des Subworkflowcontrollers, des Datenzugriffsdienstes und des Batch-Carriers im Hinblick auf seine Performance untersucht werden. Dabei werden auch alle performance-relevanten Design-Entscheidungen untersucht und begründet. Daß es dabei teilweise zu einer inhaltlichen Wiederholung von bereits in den vorangegangenen Kapiteln vorgestellten Sachverhalten kommen kann, ist unvermeidlich. Soweit wie möglich (und sinnvoll) wird dabei auf die entsprechenden Textstellen verwiesen.

Eine Anmerkung zu den vorgestellten eigenen Performance-Messungen: Sämtliche Messungen wurden auf einem einzelnen Prozessor durchgeführt. Die in den Abbildungen aufgeführten Meßwerte wurden, soweit nicht anders angegeben, durch eine eigens erstellte Statistik-Klasse[1] ermittelt. Dafür wurde die zu messende Aktion jeweils für 60 Sekunden wiederholt durchgeführt. Da Einzelmessungen aufgrund der mangelnden Auflösung ($1ms$) der Zeit-Funktionen unter C und Java nicht möglich sind, wurde ein kleinstes Zeitintervall von $200ms$ festgelegt, innerhalb dessen die ausgeführten Aktionen gezählt und anschließend der Mittelwert gebildet wurde. Das ergab im Idealfall 300 Meßwerte, über die der Mittelwert errechnet wurde, nachdem Ausreißer mit der Methode von Nalimov (implementiert nach den Angaben in [61]) entfernt wurden.

Die Grafiken der Zeitmessungen enthalten generell nur auf einen bestimmten Meßwert normierte, also relative, Zeitangaben.[2] Teilweise wurden statt der Standardabweichung auch die Minimal- und Maximalwerte (vor dem Entfernen der Ausreißer) angegeben, um die reale Streubreite zu verdeutlichen. Die Skalen der Grafiken sind zumeist logarithmisch, da die Einflußgrößen ebenfalls logarithmisch verändert wurden.

7.1 Performance des Subworkflowcontrollers

Den größten Einfluß auf die Performance des Subworkflowcontrollers (*SWFC*) haben sicherlich der Carrier und dessen Kommunikation mit dem MFC, die Java Virtual Machine (*JVM*), durch die der SWFC ausgeführt wird, sowie der Datenzugriffsdienst (*DS*). Aber auch der Aufruf von technischen Aktivitäten und System-Plugins (die im folgenden wieder beide als „Aktivitäten“ bezeichnet werden sollen) oder die Ausführung eines Subworkflow durch die Subworkflow-Runtime (*SWFR*) tragen ihren Teil zur Ausführungs- und Latenzzeit des Systems bei.

[1] Für Messungen unter C wurde die Java-Klasse in eine C-Bibliothek umgeschrieben.

[2] Da die relativen Zeitangaben in den Grafiken von Transaktionszeiten abgeleitet sind, werden sie dort als „relative Transaktionszeit“ bezeichnet.

Der Datenzugriffsdienst wird getrennt in Abschnitt 7.2 ab Seite 68 behandelt, die Ausführung des SWFC durch die JVM, der Aufruf von Aktivitäten und die Handhabung eines Subworkflow durch die SWFR werden in den folgenden Abschnitten untersucht. Der I/O-Controller hängt sehr stark vom verwendeten Carrier ab und wird deshalb in die Beschreibung des Batch-Carriers in Abschnitt 7.3 ab Seite 73 integriert.

Die übrigen Komponenten des SWFC, namentlich der Subworkflow-Überwacher und der Call-Stub, haben nur einen geringen Einfluß auf die Gesamtperformance des Subworkflowcontrollers. Sie werden in diesem Abschnitt dennoch kurz erläutert.

7.1.1 Die Kommunikation mit dem MFC

Die Kommunikation mit dem MFC findet generell über einen Carrier statt. Über die Performance von IMS und CICS können hier keine Aussagen gemacht werden, der Batch-Carrier wird in Abschnitt 7.3 ab Seite 73 behandelt. Was bleibt, ist das Protokoll und das Datenformat für die Kommunikation mit dem MFC. Auf diese Aspekte soll nun näher eingegangen werden.

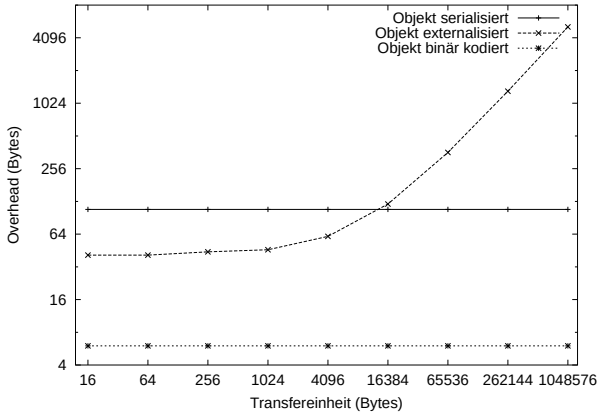


Abbildung 7.1: Overhead bei der Übertragung von Java-Objekten

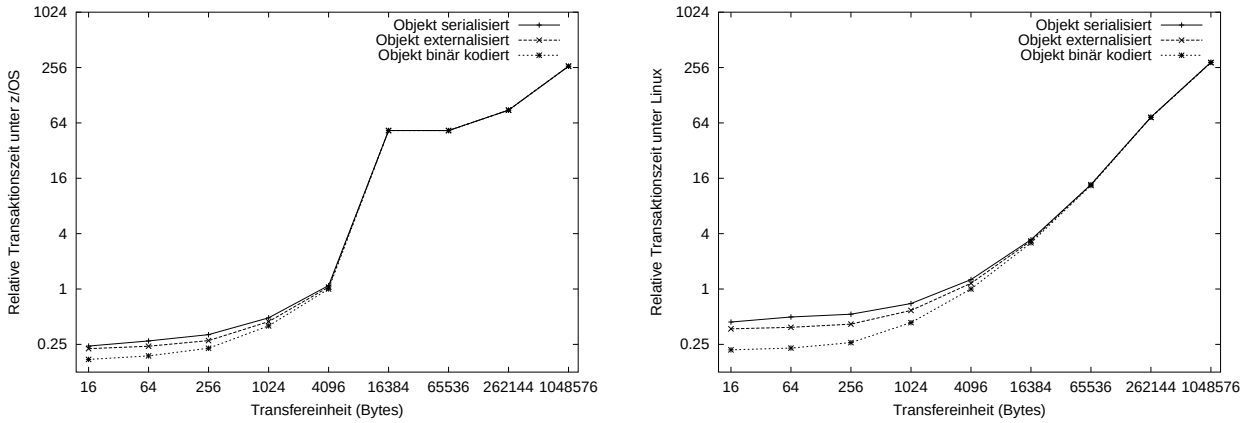


Abbildung 7.2: Transaktionszeiten der Übertragung von Java-Objekten

Das Kommunikations-Protokoll zwischen Masterflow- und Subworkflowcontroller profitiert vor allem davon, daß sämtliche an dieser Kommunikation beteiligten Komponenten transaktionsgeschützt arbeiten. Ein Auftrag des MFC kann also nicht mehr verloren gehen, sobald er dem Carrier übergeben wurde, und das Ergebnis wird garantiert an den MFC zurückgeliefert. Dadurch kann auf ein spezielles Commit-Protokoll zwischen MFC und SWFC verzichtet werden, was den Kommunikations-Aufwand deutlich reduziert.

Das Datenformat der SRWUs und SRUs sollte möglichst schnell zu kodieren sein und möglichst wenig Overhead haben. Da sowohl der MFC als auch der SWFC in Java implementiert werden sollten, bieten sich drei mögliche Übertragungsformate an:

- Am wenigsten Aufwand in der Programmierung bedeutet die Serialisierung. Alle Datentypen und viele System-Klassen von Java enthalten bereits die entsprechende Logik. Eigene Klassen müssen nur das Interface `Serializable` implementieren, um automatisch gespeichert, übertragen und wiederhergestellt werden zu können.[3]

- Die Externalisierung erfordert einen höheren Aufwand, da jede Klasse ihren eigenen Code für die Erzeugung der Übertragungsdaten enthalten muß.[4] Dabei kann nur auf die Fähigkeiten des ObjectStream aufgebaut werden, der die Basis-Datentypen von Java übertragen kann. Dafür erhält der Programmierer die Möglichkeit, selbst Einfluß auf die übertragenen Daten zu nehmen.

- Die dritte Möglichkeit ist die Kodierung der kompletten Übertragungsdaten von Hand. Die dabei gewonnene größtmögliche Flexibilität wird allerdings mit einem größeren Aufwand erkauft. Jedes zu übertragende Objekt muß zerlegt und in ein Byte-Array gepackt und daraus wieder dekodiert werden.

Der Overhead für die Kodierung der Daten durch die drei beschriebenen Methoden ist in Abbildung 7.1 dargestellt. Abbildung 7.2 stellt die Transaktionszeiten unter z/OS und Linux dar, wobei jeweils ein einfaches Java-Objekt per TCP von einem Java-Programm zu einem anderen übertragen und wieder zurückgesendet wird. Die Angaben zu den Laufzeiten der Nachrichten sind auf die durchschnittliche Laufzeit eines Objekts von etwa 4096 Bytes bei binärer Übertragung normiert, die Standardabweichung ist in der Grafik nicht zu erkennen, da sie sehr klein ist.[5]

7.1.2 Die Java Virtual Machine

Da der SWFC als Java-Programm implementiert ist, rückt natürlich bei allen Performance-Betrachtungen das Laufzeitverhalten der verwendeten JVM in den Vordergrund, durch die der SWFC ausgeführt wird. Auf der anderen Seite ist die JVM aus der Sicht des Java-Programmierers aber eine Größe, auf die er nur wenig Einfluß hat.

[3] Nur wenn einzelne Felder einer besonderen Behandlung bedürfen oder vom Typ einer nicht serialisierbaren Klasse sind, muß die eigene Klasse über die Methoden `writeObject()` und `readObject()` selbst für die Speicherung und Wiederherstellung sorgen.

[4] Klassen, die externalisiert werden sollen, müssen das Interface `Externalizable` implementieren und über die Methoden `writeExternal()` und `readExternal()` ihren Status sichern und wiederherstellen können.

[5] Der deutliche Knick in der Übertragungsrate aller Pakete unter z/OS dürfte an einer eigenwilligen „Optimierung“ der TCP-Übertragung unter z/OS liegen, die vor allem Pakete von 16kB und 64kB betrifft.

Für die Ausführung des SWFC auf z/OS-Rechnern wurde die Persistent Reusable Java Virtual Machine (*PRJVM*) von IBM gewählt. Diese bietet den großen Vorteil, daß sie viele Java-Programme hintereinander und vollkommen isoliert voneinander ausführen kann, ohne jedesmal neu gestartet zu werden. Statt dessen führt sie nach der Ausführung eines Programms einen Reset durch, und ermöglicht durch die Trennung in Anwendungs- und System-Logik eine Festlegung, welche Programmteile einen Reset überleben sollen, und welche nicht. Der von der IBM-Implementierung von `javac` erzeugte Code ist zudem etwas kompakter als der von SUNs `javac`, die Ausführungsgeschwindigkeit scheint allerdings vergleichbar zu sein.

Die Performance-Vorteile der PRJVM wurden von Marc Beyerle in seiner Diplomarbeit [27] genau untersucht. Das wichtigste Ergebnis: Die Performance der PRJVM liegt um den Faktor 300 über der Performance einer herkömmlichen JVM, wenn man eine weitestgehende Isolation aufeinanderfolgender Transaktionen erreichen will.[6] Die für den SWFC relevanten Aspekte der PRJVM wurden außerdem in Abschnitt 3.3 ab Seite 23 beschrieben.

Der SWFC nutzt die Möglichkeiten der PRJVM, indem sämtliche Komponenten des SWFC als System-Logik („trusted middleware“) implementiert sind und so nur einmal geladen werden müssen. Sie werden allerdings bei jedem Reset re-initialisiert, um sich auf den nächsten Subworkflow einstellen zu können. Nur die Java-Aktivitäten (also technische Aktivitäten und System-Plugins, die in der Sprache Java implementiert wurden) sind als Anwendungs-Logik („sharable application“ bzw. „transient application“) implementiert, wobei durch die Angabe des `persistent`-Attributs in der Interface-Beschreibung festgelegt werden kann, ob die Klassen-Definitionen beim Reset gelöscht werden sollen oder im Speicher verbleiben können.

7.1.3 Die Aktivitäten

Der Aufruf von Java- und COBOL-Aktivitäten ist sehr unterschiedlich, weshalb im folgenden einzeln auf die jeweiligen Performance-Eigenschaften eingegangen werden soll. Eine genaue Beschreibung der Eigenschaften von Aktivitäten findet sich in Kapitel 4 ab Seite 31, Unterabschnitt 3.4.6 und Unterabschnitt 3.4.7 auf Seite 29 sowie Unterabschnitt 3.4.3 ab Seite 25.

7.1.3.1 Der Aufruf von Java-Aktivitäten

Java-Aktivitäten müssen zunächst über Java-Reflection geladen und instanziiert werden. Der Zugriff auf Reflection ist relativ zeitintensiv[7], aber aufgrund der Struktur der PRJVM nicht zu umgehen.[8]

Um Reflection wenigstens aus dem nachfolgenden Aufruf der Aktivität herauszuhalten, wird diese nach der Instanziierung über ein Interface angesprochen, das zur Middleware zählt. Die dabei erzielte Performance liegt nur wenig unter der des direkten Zugriffs auf Methoden anderer Klassen.

[6] Für die Isolation aufeinanderfolgender Transaktionen muß eine herkömmliche JVM nach jeder Transaktion neu gestartet werden, bei der PRJVM reicht jedoch ein Reset aus.

[7] Aus den Messungen von Thomas Hornung geht hervor, daß die Ausführung einer Aktivität über Reflection in etwa ein Zwölftel der Gesamtausführungszeit des Subworkflowcontrollers ausmacht.

[8] Das Laden von Applikationsklassen aus Middlewareklassen heraus ist nur über Reflection möglich.

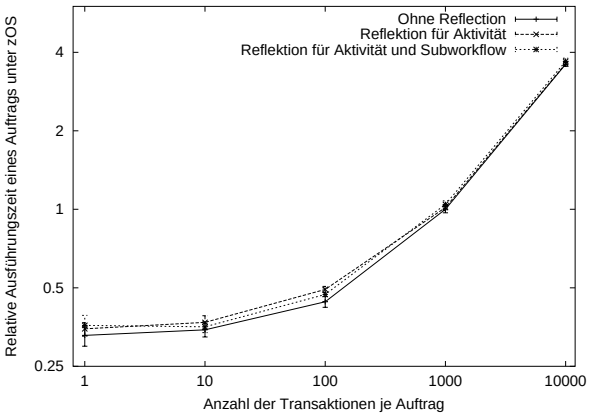


Abbildung 7.3: Ausführungszeiten eines Subworkflow mit und ohne Reflection

Eigene Messungen (siehe Abbildung 7.3) haben ergeben, daß der Einfluß von Reflection auf die Gesamtausführungszeit eines Batch-Job nur einen geringen Einfluß hat, daher kann die gewählte Methode als ausreichend gelten.

Für die Messungen, die Abbildung 7.3 zugrunde liegen, wurden zehn Batch-Jobs mit der angegebenen Anzahl von Transaktionen von jeweils einem einzelnen Subworkflowcontroller ausgeführt. Die Ergebnisse wurden auf die Ausführungszeit eines Auftrags mit 1000 Transaktionen unter Umgehung von Reflection normiert. Die Standardabweichung ist in der Grafik eingetragen.

7.1.3.2 Der Aufruf von COBOL-Aktivitäten

Der Aufruf von COBOL-Aktivitäten wurde nicht implementiert und konnte deshalb auch nicht überprüft werden. Das Konzept ist jedoch so angelegt, daß die einzigen performance-relevanten Komponenten im Aufrufpfad von COBOL-Aktivitäten (neben der Aktivität selber, die aber außerhalb der Kontrolle der Transaktionsmaschine liegt) die SWFR und der in C implementierte COBOL-Stub sind.

Die SWFR muß die Eingangsdaten für die COBOL-Aktivität umgewandelt in ihre Objekt-Form in ein Array packen und ein Array mit Typ-Informationen für die Ein- und Ausgabevariablen erzeugen. Der COBOL-Stub muß die Java-Objekte aus dem Array mit Hilfe von JNI einzeln auslesen und die darin gekapselten Daten in COBOL-Datentypen transformieren. Die Ausgabedaten der COBOL-Aktivität werden vom COBOL-Stub wieder über JNI in Java-Objekten verpackt und in einem Array abgelegt, und schließlich muß die SWFR die Daten aus dem Objekt-Array entnehmen.

Der Zugriff auf die Objekt-Form der einfachen Datentypen unter Java ist nicht sehr aufwendig, und JNI hat sich bei der Entwicklung des Batch-Carriers als sehr schnell erwiesen, daher dürfte der Aufwand für den Aufruf von COBOL-Aktivitäten sogar noch unter dem Aufwand liegen, den das Instanziieren und nachfolgende Aufrufen einer Java-Aktivität bedeutet.

Insgesamt kann die Performance von COBOL-Aktivitäten als mindestens ebenbürtig zu der Performance von Java-Aktivitäten angenommen werden. Dieses Ergebnis hatte auch Thomas Hornung in [25] erhalten, der keinen relevanten Unterschied in der Ausführungszeit von Java- und (simulierten) COBOL-Aktivitäten messen konnte.

7.1.4 Die Subworkflow-Runtime

Die Ausführung eines Subworkflow erfolgt im hier (insbesondere in Unterabschnitt 3.4.3 ab Seite 25) vorgestellten Design durch eine Subworkflow-Runtime, die spezifisch für diesen einen Subworkflow aus dessen BPDL-Beschreibung durch einen Compiler erstellt wurde.

Gegenüber dem Interpreter-Ansatz in der Arbeit von Thomas Hornung bietet die generierte Subworkflow-Runtime einen enormen Performance-Vorteil: Ein einfacher Interpreter benötigt im Vergleich zu einer compilierten Lösung in jedem Fall ein Vielfaches der Zeit, nicht zuletzt dadurch, daß vom Compiler leicht überflüssiger Code entfernt und der verbliebene Code optimiert werden kann.[9]

Der Anteil an der Gesamt-Performance läßt sich mangels empirischer Daten nur sehr schwer einschätzen. Es ist jedoch wahrscheinlich, daß die Verweilzeit innerhalb der SWFR selbst nur einen sehr kleinen Teil der Gesamtlaufzeit einer Transaktion ausmacht, während das Parsen der SWFDL im Subworkflowcontroller von Thomas Hornung einen nicht unerheblichen Anteil an der Gesamtperformance gehabt haben dürfte.

7.1.5 Der Call-Stub

Der Call-Stub wird normalerweise nur einmal bei der Initialisierung des Subworkflowcontrollers aufgerufen (vergleiche dazu Unterabschnitt 3.4.1 ab Seite 24 und Kapitel 6 ab Seite 55). Die einzig performance-relevante Aktion des Call-Stub ist zudem die Überprüfung des Aufrufers über eine Exception. Die dafür aufgewendete Zeit kann im Vergleich zu der in den anderen Komponenten des SWFC verbrauchten Zeit getrost vernachlässigt werden.

Interessant könnte die Performance des Call-Stub lediglich unter CICS werden, da dort der Call-Stub für jede Transaktion durchlaufen werden muß. Hier würde sich als Optimierung anbieten, den Wiedereintritt über eine Statusvariable zu prüfen, die beim Reset zurückgesetzt wird.

7.1.6 Der Subworkflow-Überwacher

Der Subworkflow-Überwacher hat selbst nur eine sehr eingeschränkte Funktionalität (siehe dazu Unterabschnitt 3.4.2 ab Seite 25), die sich auch nicht mehr optimieren läßt. Er trägt zur Gesamtlaufzeit vor allem durch das Laden der Subworkflow-Runtime über den Datenzugriffsdienst und die nachfolgende Instanziierung der SWFR über Reflection bei.

7.2 Performance des Datenzugriffsdienstes

Da der Datenzugriffsdienst im vorliegenden Design (siehe Kapitel 5 ab Seite 35) sowohl für die Transaktionssteuerung als auch für den Datenbankzugriff von zentraler Bedeutung ist, hat er einen großen Einfluß auf die Performance des Subworkflowcontrollers.

[9] Eine Optimierung ist zwar auch auf Interpreter-Basis möglich und wird beispielsweise in Java durch die JIT-Compilierung auch eingesetzt, ist aber aufwendig in der Implementierung und wäre daher für den Subworkflowcontroller kaum die geeignete Lösung.

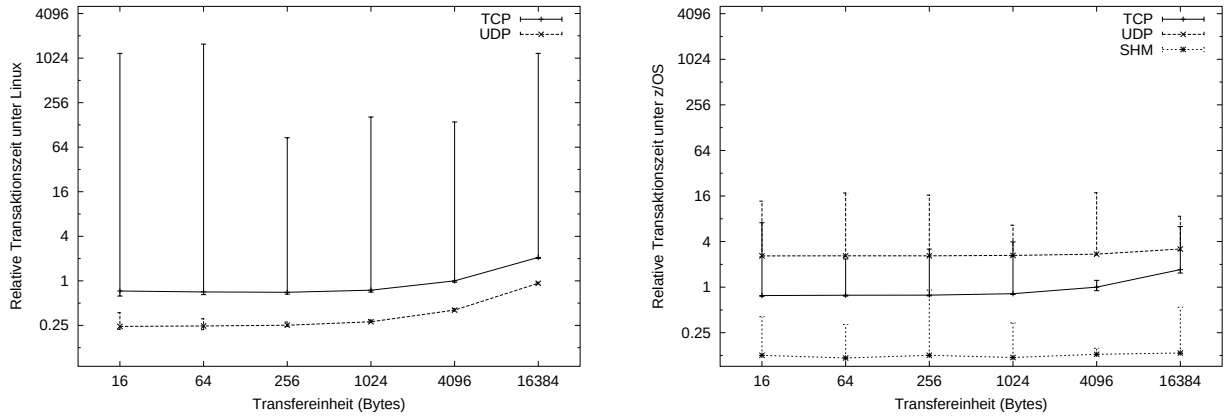


Abbildung 7.4: Transaktionszeiten bei TCP- und UDP-Kommunikation

Kritisch für die Performance des Datenzugriffsdienstes sind dabei in erster Linie drei Bereiche: Die Kommunikation mit dem Subworkflowcontroller, die Implementierung des Cache und der Datenbankzugriff. In den ersten beiden Bereichen ist eine möglichst weitgehende Parallelisierung erstrebenswert, um mehrere Subworkflowcontroller quasi gleichzeitig ohne Performanceverlust bedienen zu können.

7.2.1 Die Kommunikation mit dem SWFC

Für die Kommunikation zwischen dem internen Datenzugriffsdienst innerhalb des SWFC und dem externen Server des Datenzugriffsdienstes wurden UDP-Sockets ausgewählt, wie in Unterabschnitt 5.4.1 ab Seite 42 beschrieben. Der Grund hierfür ist einfach der, daß diese Schnittstelle es erlaubt, den SWFC und den externen Server des DS auf zwei unterschiedlichen Rechnern laufen zu lassen, und daß diese Schnittstelle über Port und Hostname einen definierten Endpunkt für die Kommunikation liefert. Außerdem hat das UDP-Protokoll nur einen sehr geringen Overhead gegenüber IP, so daß es eine gute Performance aufweisen sollte.

Die Persistenz der Kommunikation wird über zwei Listen mit den gerade aktiven und den bereits bearbeiteten Aufträgen gesichert. Diese Listen sind über Hash-Tabellen implementiert und über die Subworkflow-Kennung und eine Sequenz-Nummer indiziert (vergleiche Unterabschnitt 5.4.1 ab Seite 42). Die Zugriffszeit auf diese Listen ist also sehr kurz, so daß, zusammen mit dem schnellen Locking durch die Synchronisationsmethoden der POSIX-Threads, eine Verzögerung für andere Aufträge erst bei einer sehr hohen Last auftreten wird.

Ursprünglich boten sich aus Sicht des Designs die Protokolle TCP und UDP für die Server-Kommunikation an. Um eine Entscheidung zwischen diesen beiden fällen zu können, wurden ein paar Messungen vorgenommen, die in Abbildung 7.4 aufgeführt sind und im folgenden erläutert werden. In der Abbildung sind die Meßwerte auf die Dauer einer TCP-Transaktion mit 4096 Bytes genormt, außerdem ist die Streubreite der Werte eingetragen.

Die in der linken Hälfte von Abbildung 7.4 aufgeführten Tests unter Linux ergaben einen Geschwindigkeitsvorsprung um etwa den Faktor 2 mit dem zusätzlichen Vorteil, daß die in der Java-Implementierung der TCP-Sockets auftauchenden teilweise sekundenlangen Verzögerungen unter UDP überhaupt nicht auftraten.

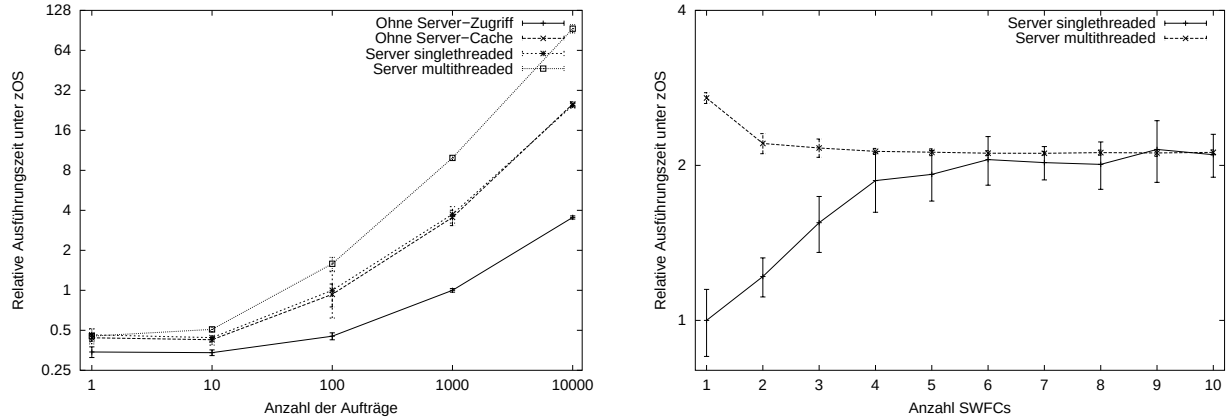


Abbildung 7.5: Ausführungszeiten von Batch-Jobs

Leider scheint die Implementierung der UDP-Sockets unter z/OS weniger performant zu sein. Wie aus dem rechten Diagramm in Abbildung 7.4 hervorgeht, sind diese um etwa den Faktor 3 langsamer als TCP-Sockets, die wiederum um einen Faktor von 8 langsamer als eine Kommunikation über die Shared-Memory-Queues des Batch-Carriers sind, die zum Vergleich in die Grafik mit aufgenommen wurden.

Insgesamt sind Sockets also deutlich langsamer als die System V IPC Mechanismen, weshalb es sinnvoll wäre, die Server-Kommunikation über System V IPC abzuwickeln, auch wenn dadurch die Option wegfällt, SWFC und Server auf verschiedenen Rechnern[10] laufen zu lassen. Der SWFC muß dann zwar auf weitere JNI-Routinen zurückgreifen, da System V IPC von Java nicht direkt unterstützt wird, dies hat jedoch keinen Einfluß auf die Performance, da JNI keinen meßbaren Overhead im Vergleich zum Aufruf von Java-Methoden hat.

In der vorliegenden Implementierung wird die komplette Server-Kommunikation über TCP abgewickelt, da dies in der Design-Phase als ein sinnvoller und gangbarer Weg erschien. Vorläufige Messungen (in der linken Hälfte von Abbildung 7.5 zu sehen) haben aber ergeben, daß sich die Transaktionsrate des SWFC durch den Zugriff auf den externen Server etwa um den Faktor 5 verschlechtert, während andere Messungen[11] ergeben haben, daß für die Auftragsbearbeitung im Server selbst nur etwa ein Zehntel der für eine Server-Anfrage benötigten Zeit verbraucht wird.

Für die Weiterentwicklung wird daher dringend empfohlen, die Kommunikation auf System V Message Queues umzustellen. Diese sind vom Konzept her gut für die vorliegende Aufgabe geeignet, vergleichsweise schnell, und bieten sogar die Möglichkeit, auf einem Rechner eine eindeutige Queue-ID zu vergeben, die in den Konfigurationsdateien abgelegt werden kann.

Um eine möglichst weitgehende Parallelausführung der Aufträge verschiedener Subworkflowcontroller zu erreichen, wird zudem jeder Auftrag durch einen eigenen Thread ausgeführt. Derzeit wird dieser Thread für jeden neuen Auftrag neu erzeugt. Der dadurch verursachte Overhead verlangsamt allerdings die Ausführung nach den in Abbildung 7.5 dargestellten Messungen etwa um den Faktor 4. Auf der anderen Seite führt die Ausführung aller Anfragen von mehreren Subworkflowcontrollern durch einen einzigen Server-Thread zu einer spürbar schlechteren Parallelisierbarkeit, was sich auch deutlich im rechten Diagramm in Abbildung 7.5 zeigt.

[10] System V IPC benötigt einen gemeinsamen physikalischen Hauptspeicher.

[11] Im Debugging-Modus geben SWFC und Server die jeweiligen Ausführungszeiten der Serverzugriffe an.

In einer späteren Implementierung sollte daher der Server einen Pool von Threads vorhalten, dessen Größe maximal der doppelten Anzahl der zur Verfügung stehenden Prozessoren entspricht. Zudem sollten Aufträge, die einen größeren Rechenaufwand haben oder längere Zeit beanspruchen, an ein Backend delegiert werden, das über die Workload Management Services von z/OS angebunden ist. Die Grenze, ab der sich die Ausführung einzelner Aufträge durch ein Backend lohnt, muß in weiteren Experimenten herausgefunden werden, und hängt von vielen Faktoren ab, die möglicherweise auch eine Anpassung dieser Grenze von System zu System notwendig machen.

7.2.2 Der Cache

Der Zugriff auf den Cache, dessen Funktionsweise in Unterabschnitt 5.6.2 ab Seite 46 bereits eingehend beschrieben wurde, erfolgt durchgängig über Hash-Tabellen mit Linked-List-Einträgen. Alle Daten des Cache, inclusive der Listen, sind in einem großen Feld abgelegt, das über eine eigens dafür entwickelte Linked-List-Implementierung indiziert und durch eine Speicherverwaltung auf Basis einer Free-List verwaltet wird.

Dieses Cache-Design ist sehr schnell, da durch die Hash-Tabellen der Zugriff auf die einzelnen Elemente der Verwaltungslisten in nur wenigen Schritten erfolgen kann, vorausgesetzt, die Hash-Funktion verteilt die Einträge gleichmäßig, und die Hash-Tabellen sind ausreichend dimensioniert. In den durchgeführten Performance-Tests (dargestellt im linken Diagramm von Abbildung 7.5) hatte die Cache-Implementierung keinen meßbaren Einfluß auf die Performance des Subworkflowcontrollers, das vorgelegte Design hat sich also bewährt.

7.2.2.1 Cache-Organisation

Die Daten jedes Subworkflow werden über jeweils eine einzelne Liste verwaltet. Dies ist die sinnvollste Lösung, da die Einträge in dieser Liste chronologisch geordnet sein müssen, und da andererseits der Aufwand für Methoden mit einer kürzeren Zugriffszeit wie etwa Red-Black-Trees im Verhältnis zu den anfallenden Daten viel zu groß wäre: Die einzelnen Transaktionen eines Subworkflow sind ja konzeptionell relativ kurz, und ein Subworkflow stellt auch selbst eine „short running transaction“ des gesamten Workflow dar, so daß innerhalb eines Subworkflow nur Transaktionsdaten in der Größenordnung von 10 bis 20 Einträgen anfallen.

Daten aus Datenbanken werden direkt über eine Hash-Tabelle verwaltet, so daß hier ein sehr schneller Zugriff erfolgen kann. Die dadurch erreichte Entkoppelung von externen Datenbanken dürfte den größten Einfluß auf die Gesamtperformance des Subworkflowcontrollers haben.

Die Speicherverwaltung ist derzeit über eine einzige Liste implementiert, und könnte durch die Verwendung mehrerer Listen für Speicherbereiche verschiedener Größen noch optimiert werden. Da die gesamte Cache-Implementierung aber bereits sehr schnell ist, ist eine Optimierung der Speicherverwaltung wohl kaum nötig.

Die Konsistenz der Listen des Cache wird über ein getrenntes Locking jeder Liste[12] mit Hilfe der vergleichsweise schnellen Synchronisationsmechanismen der POSIX-Threads erreicht. Eine Verzögerung durch das Locking ist daher erst bei einer sehr hohen Zugriffslast zu erwarten.

[12] Die Listen mit Transaktionsdaten der Subworkflows müssen gar nicht gelocked werden, da es immer nur einen Subworkflow mit der passenden Kennung geben kann.

7.2.3 Die Datenbankanbindung

Die Datenbankanbindung konnte, wie in der Zusammenfassung auf Seite vii geschildert, nicht implementiert werden. Daher sind die folgenden Betrachtungen zwangsläufig rein theoretischer Natur. Eine eingehende Beschreibung der Funktionalität der Datenbankanbindung findet sich in Unterabschnitt 5.6.4 ab Seite 50.

7.2.3.1 Datenbankanbindung über ODBC

Alle Datenbankzugriffe des Datenzugriffsdienstes erfolgen über die ODBC-Schnittstelle. Den Ausschlag zu dieser Entscheidung gab die große Flexibilität, die mit dieser Lösung erreicht wird. Über ODBC lassen sich alle Datenbanken ansprechen, für die ein ODBC-Treiber existiert, also insbesondere auch Datenbanken, die selber nicht SQL sprechen.

Von der Performance her ist ein statischer SQL-Zugriff zwar schneller, allerdings müßte der Server dann für jeden Datenbankzugriff auf ein aus der Beschreibung des Zugriffs erstelltes Modul zurückgreifen, und Server wie auch Subworkflowcontroller müßten eine eigene Infrastruktur für jeden Datenbanktyp enthalten, was den Verwaltungsaufwand erheblich vergrößern würde. Zudem ist die Performance des Datenzugriffsdienstes durch den Cache des Servers bei mehrfachem Zugriff auf die gleichen Daten weitgehend von der Zugriffsgeschwindigkeit auf die Datenbanken entkoppelt, so daß die geringere Performance von ODBC verglichen mit statischem SQL nicht mehr so stark auf die Gesamtperformance durchschlagen dürfte.

Der Zugriff über ODBC erfolgt dynamisch. Um die Anbindung an SQL-Datenbanken zu beschleunigen, wäre es eventuell sinnvoll, wenn für jeden Datenbankzugriff eine „stored procedure“ erstellt wird, die statische SQL-Kommandos verwendet, und auf die über die ODBC-Schnittstelle zugegriffen wird. Die Performance dieses Verfahrens läßt sich jedoch ohne vorangegangene Tests kaum einschätzen, möglicherweise würde es bei den einfachen Anfragen, die der SWFC erlaubt, auch einen negativen Einfluß auf die Performance haben.

7.2.3.2 Optimierung der Datenbankanbindung

Durch ein verzögertes Group-Commit mehrerer technischer Transaktionen (siehe Unterunterabschnitt 5.6.4.4 ab Seite 53) kann die Commitzeit der Transaktionen deutlich verkürzt werden, da der Subworkflowcontroller nicht mehr auf den Abschluß der Datenbankoperationen warten muß. Von Nachteil hierbei ist allerdings, daß die entsprechenden Datenbankfelder auch länger gesperrt bleiben müssen, was parallel laufende Anwendungen ausbremsen kann. Wenn die betroffenen Datenbanken dem Server exklusiv zur Verfügung stehen, stellt das längere Halten der Locks kein Problem dar, es kann sogar komplett auf ein Locking auf den Datenbanken verzichtet werden.

Das Commit läßt sich nicht mehr verzögern, wenn ein mengenbasierter Datenbankzugriff teilweise Felder aus geänderten, aber noch nicht zurückgeschriebenen Datensätzen enthalten könnte. Ein lokale Aktualisierung solcher mengenbasierter Datenbankzugriffe würde eine datenbankähnliche Implementierung der Cache-Verwaltung erfordern, was einen erheblichen Mehraufwand bedeuten würde. Sollte man dies in Betracht ziehen, käme wohl eher die Integration der Funktionalität des Datenzugriffsdienstes in eine existierende Datenbank in Frage.

7.3 Performance des Batch-Carriers

Die Performance des Batch-Carriers hängt im wesentlichen von der Kommunikation mit dem Subworkflowcontroller und von der Latenzzeit beim Start der JVMs ab, auf denen der SWFC ausgeführt wird. Das Interface zum Masterflowcontroller spielt dagegen kaum eine Rolle, soll aber trotzdem kurz erläutert werden.

7.3.1 Das Interface zum Masterflowcontroller

Die Performance des MFC-Interfaces ist relativ unkritisch: Die Ausführung eines Batch-Jobs erfolgt üblicherweise unabhängig von seiner Initiierung, so daß das Erzeugen der Eingangsdaten kein Problem darstellt. Das Auslesen der Aufträge aus einer Datei hat nach den Beobachtungen, die während der Implementierung des Batch-Carriers gemacht wurden, keinen meßbaren Einfluß auf die Performance. Die Rückmeldung am Ende des Batch-Jobs über Semaphoren ist konzeptionell wenig aufwendig und dürfte deshalb ebenfalls ohne Belang für die Ausführungszeit des gesamten Jobs sein. Die gewählten Methoden zur Kommunikation mit dem Masterflowcontroller sind also vollkommen ausreichend.

7.3.2 Die Initialisierung der Java-Virtual-Machines

Das Starten und Beenden der JVMs am Anfang und am Ende des Batch-Jobs und bei jedem fehlgeschlagenen Reset einer PRJVM nimmt eine verhältnismäßig lange Zeit in Anspruch. Diese Zeit wird immerhin dadurch etwas verkürzt, daß der Batch-Carrier alle JVMs innerhalb eines JVM-Set als Workers startet.

Durch die parallele Ausführung mehrerer Subworkflowcontroller kann dieser Overhead aber problemlos ausgeglichen werden, wie sich an den im rechten Diagramm von Abbildung 7.5 auf Seite 70 dargestellten Performance-Messungen zeigt, die einen deutlichen Performance-Gewinn durch die Parallelausführung mehrerer SWFCs dokumentieren (verständlicherweise nur für den Server mit Multithreading).

7.3.3 Die Kommunikation mit dem Subworkflowcontroller

Um das am besten geeignete Verfahren für die Kommunikation zwischen Batch-Carrier und Subworkflowcontroller zu finden, wurden zunächst die in Abbildung 7.6 auf Seite 74 aufgeführten Vergleichsmessungen (normiert auf die Shared-Memory-Kommunikation bei 4096 Bytes Paketgröße) zwischen einer Kommunikation über die Standard-Ein-/Ausgabekanäle (stdio), über System V Message Queues und über eine eigene Implementierung von Message Queues direkt auf Basis von System V Semaphoren und Shared Memory Regions durchgeführt.

Da die Kommunikation über Shared Memory Regions eine gleichbleibend gute Performance über einen großen Bereich lieferte, wurde diese für die endgültige Implementierung gewählt. Dabei werden eingehende und ausgehende Nachrichten in einem gemeinsamen Ringpuffer abgelegt und über zwei getrennte Warteschlangen verwaltet. Der Ringpuffer ist durch eine Semaphore geschützt, die Warteschlangen werden über vier weitere Semaphoren nach dem Producer-Consumer-Modell synchronisiert. Ein eigener, ebenfalls durch Semaphoren geschützter Kanal dient schließlich dem Festschreiben oder Verwerfen der ausgehenden Nachrichten.

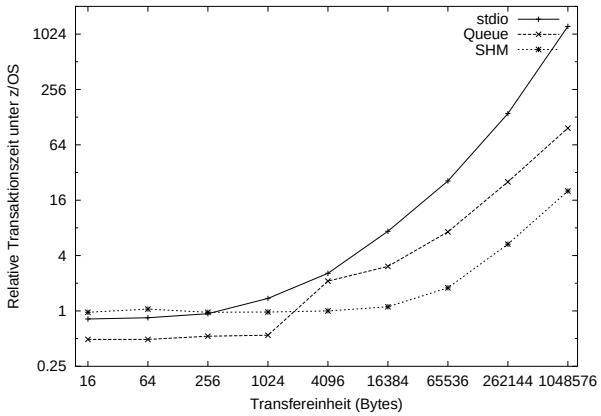


Abbildung 7.6: Transaktionszeiten für stdio-, Queue- und Shared-Memory-Kommunikation

Diese Implementierung weist eine hohe Parallelisierbarkeit auf, da drei verschiedene Aktionen nahezu zeitgleich ausgeführt werden können: Das Schreiben oder Einlesen eines neuen Auftrags, die Rückgabe oder Übernahme einer Antwort und das Festschreiben oder Verwerfen von Antworten am Ende einer Transaktion oder eines Subworkflow. Die ersten beiden Aktionen laufen vollkommen asynchron ab, nur für die letzte ist eine Synchronisation mit dem Batch-Carrier nötig.

Der Zugriff auf die Queues ist sehr schnell beendet und dürfte nur selten einen anderen Zugriffsversuch merklich verzögern. Die Kommunikation zwischen Batch-Carrier und Subworkflowcontroller erwies sich denn auch bei der fortschreitenden Implementierung des SWFC als unkritisch für die Gesamtperformance.

Ein Problem taucht bei der beschriebenen Implementierung der Kommunikation allerdings gelegentlich auf: Wenn der Ringpuffer voll gelaufen ist, weil etwa ein länger dauernder Subworkflow die Wiederverwendung von nicht mehr benötigten Speicherplätzen verhindert (der Ringpuffer kennt nur einen freien Speicherbereich), muß gelegentlich ein Subworkflow abgebrochen und nach einem Neustart fortgesetzt werden (mit Hilfe der Persistenz-Sicherung des Datenzugriffsdienstes).

Um dieses Problem zu vermeiden, wäre es sinnvoll, die Kommunikation zwischen Batch-Carrier und Batch-I/O-Controller von der augenblicklichen Implementierung durch einen Ringpuffer auf die für den Cache des externen Servers entwickelten Listen-Mechanismen umzustellen. Durch die darin enthaltene Speicherverwaltung ist eine effizientere Nutzung des Cache möglich. Dabei könnte eventuell durch ein feineres Locking mit Hilfe der Mechanismen der POSIX-Threads auch die Parallelisierbarkeit der Kommunikation noch weiter gesteigert werden.

Kapitel 8

Ergebnisse und Ausblick

In diesem Kapitel sollen die erreichten Ergebnisse dargestellt und ein Ausblick auf mögliche Weiterentwicklungen gegeben werden.

8.1 Zusammenfassung der Ergebnisse

Auch wenn aufgrund der Rechnersituation[1] nur Teile der Implementierung abgeschlossen werden konnten, so wurde doch konzeptionell eine neue Grundlage für den Subworkflowcontroller und teilweise auch für die Transaktionsmaschine geschaffen. Dies bezieht sich insbesondere auf die Einführung eines Business Process Designers und die damit verbundene Ausführung von Subworkflows durch eine automatisch generierte Subworkflow-Runtime, die ein Interpretieren der Subworkflow-Beschreibung durch den SWFC überflüssig macht. Aber auch der komplett überarbeitete Aufbau des Subworkflowcontrollers selbst und der Entwurf eines externen Servers für den Datenzugriffsdienst stellen eine solide Grundlage für Weiterentwicklungen dar.

Der implementierte Batch-Carrier erwies sich zudem als funktional, stabil und ausreichend schnell, um die Gesamtperformance nicht meßbar zu beeinträchtigen. Aufgrund von nicht verfügbaren Libraries konnte allerdings das Einlesen der Argumente aus einer BPDL/S-Beschreibung noch nicht realisiert werden.

Die Cache-Implementierung des externen Servers zeigte in einfachen Performance-Tests eine hohe Geschwindigkeit und gute Stabilität. Lediglich für die Kommunikation mit dem SWFC sollte auf ein anderes Verfahren umgestellt werden, da sich die Performance von Sockets unter z/OS als nicht ausreichend herausgestellt hat. Mangels ODBC-Anbindung der DB2 nicht implementiert werden konnte die Datenbank-Anbindung des Servers, und auch das Parsen der BPDL/S-Parameter-Datei war aufgrund der fehlenden Libraries nicht möglich.

Die Implementierung des Subworkflowcontrollers scheint ausreichend performant zu sein, was jedoch erst durch einen Vergleich der Performance mit dem Subworkflowcontroller von Thomas Hornung überprüft werden kann. Dieser Vergleich setzt natürlich die vollständige Implementierung der Datenbank-Anbindung des Datenzugriffsdienstes voraus.

[1] Auf dem für diese Diplomarbeit von der Universität Leipzig vermittelten Großrechner konnten einige für die ursprüngliche Aufgabenstellung essenzielle Komponenten nicht eingerichtet werden. Das betraf einerseit die Java-Integration von IMS und CICS und andererseits die ODBC-Anbindung der DB2.

Der Schnittstellen-Compiler konnte fertiggestellt werden, der Subworkflow-Compiler wurde dagegen nur rudimentär implementiert. Er beherrscht bereits das Parsen der beteiligten BPDL-Beschreibungen und die anschließende Einordnung in ein Mapping für schnellen Zugriff auf die einzelnen Elemente der BPDL-Beschreibungen über den `targetNamespace`. Die vollständige Implementierung des Subworkflow-Compilers und seine Optimierung stehen noch aus.

8.2 Weiterarbeit an der Transaktionsmaschine

In jedem Fall sollten die in Kapitel 7 ab Seite 63 angegebenen Verbesserungsvorschläge umgesetzt werden. Insbesondere die Server-Kommunikation sollte auf die deutlich schnelleren System V Message Queues umgestellt werden.

Weitere Aufgaben, die noch offen sind, betreffen die Einbindung des SWFC in die Carrier IMS und CICS, und die Implementierung des Datenbank-Zugriffs. Die Grundlagen dafür sind allerdings bereits geschaffen: Der Subworkflowcontroller ist auf die Einbindung in verschiedene Carrier und die Nutzung des Datenbankzugriffs vollständig vorbereitet, und die Cache-Verwaltung im externen Server des DS bietet volle Unterstützung für den Umgang mit Datenbankzugriffen. Die Semantik der Datenbankzugriffe ist zudem in der vorliegenden Arbeit bereits weitgehend spezifiziert worden, ebenso wie die Funktionsweise der Carrier-abhängigen Komponenten (Call-Stub und I/O-Controller) unter IMS und CICS.

Schließlich steht noch die Implementierung des Masterflowcontrollers mit den entsprechenden Konnektoren (u.a. für den Batch-Carrier) aus, und auch der Business Process Designer und der dazugehörige Subworkflow-Compiler (und möglicherweise auch ein Masterflow-Compiler) müssen noch implementiert bzw. fertiggestellt werden.

Für eine eigenständige Diplomarbeit dürften allerdings weder die Implementierung der Einbindung in IMS und CICS noch die Fertigstellung des externen Servers und die Verbesserungen an der existierenden Implementierung ausreichen. Man könnte aber daraus gut zwei Studienarbeiten machen.

Die Implementierung eines Business Process Designers auf Basis der vorhandenen Sprach-Definitionen und die vollständige Implementierung sowie Optimierung des Subworkflow-Compilers dürfte allerdings ausreichend Tiefe und Volumen für eine eigenständige Diplomarbeit haben.

Und auch die Implementierung des Masterflowcontrollers mit den dazugehörigen Konnektoren und dem Entwurf einer Lastverteilung innerhalb der Verarbeitungsdomänen ist ein gutes Thema für eine weitere Diplomarbeit.

Anhang A

Use Cases

Use-Case-Szenarien dienen dazu, das Verhalten einer Komponente in allen möglichen Situationen genau zu beschreiben. Dadurch sind sie einerseits ein sehr gutes Mittel, um die Korrektheit der Implementierung einer Komponente zu überprüfen, andererseits können sie auch das Verständnis komplexer Abläufe erleichtern.

Im folgenden werden solche Use-Case-Szenarien für die Ausführung des Subworkflowcontrollers (*SWFC*) durch den Batch-Carrier, für die Abarbeitung eines Auftrags durch den SWFC und für die Fähigkeiten des externen Servers des Datenzugriffsdienstes entworfen.

Hierbei wurde eine Darstellung der Use Cases gewählt, die teilweise an das bei IBM intern verwendete Use Case Modell angelehnt ist, und die insofern deutlich von der in [62] verwendeten Notation abweicht, als stärker auch auf interne Prozesse eingegangen wird.

Abbildung A.1: Minimaler Subworkflow

Alle in Abschnitt A.1 ab Seite 78 und Abschnitt A.2 ab Seite 81 beschriebenen Use Cases orientieren sich an dem in Abbildung A.1 dargestellten minimalen Subworkflow. Die Balken am Anfang und Ende des Subworkflow symbolisieren die Übernahme der SRWU und die Übergabe der Ergebnis-SRU, die eckige Klammer dazwischen die Transaktionsgrenzen. Zum Vergleich findet sich in Abbildung A.2 ein erweiterter Subworkflow, der alle grundlegenden Fähigkeiten des Subworkflowcontrollers nutzt.

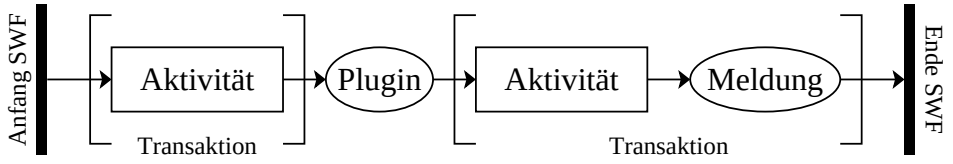


Abbildung A.2: Erweiterter Subworkflow

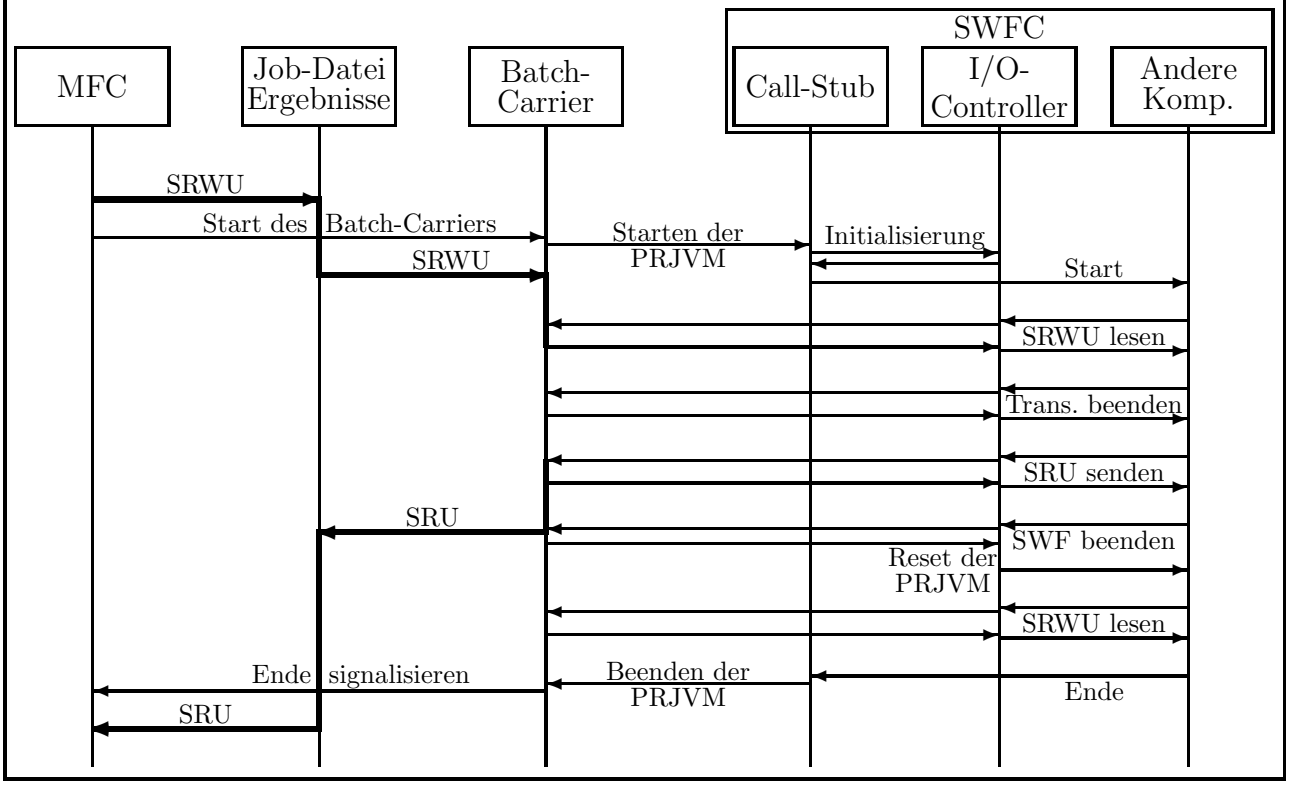
A.1 Der Batch-Carrier

Der folgende Use Case beschreibt die Ausführung des Subworkflowcontrollers mit Hilfe des Batch-Carriers.

Use Case BATCH.1	**Batch-Job ausführen**	
Thema	Einbindung des SWFC in den Batch-Carrier	
Ereignis	Der MFC hat einen neuen Auftrag als Batch-Job zu bearbeiten.	
Beteiligte Komponenten	MFC, Batch-Carrier, SWFC	
Kurzbeschreibung	Der MFC hat einen neuen Auftrag, der als Batch-Job ausgeführt werden soll. Die Kommunikation erfolgt über Dateien.	
Vorbedingungen	• Die Konfiguration des Batch-Carriers ist vollständig und fehlerfrei. • Ein neuer Auftrag ist eingegangen.	
Ergebnis		**Bedingung**
1. Der Batch-Job wurde erfolgreich ausgeführt.		Während der Ausführung sind keine Fehler aufgetreten.
2. Der Batch-Job wurde abgebrochen.		• Der Start der Master-JVM schlug fehl. • Der Batch-Carrier ist abgestürzt oder wurde unterbrochen.
Ablauf	Mit Hilfe des Batch-Connectors startet der Masterflowcontroller den Batch-Carrier und übergibt ihm als Parameter den Namen einer Job-Datei, in der die SRWUs abgelegt sind, einer Ergebnis-Datei, in die die SRUs geschrieben werden, und die Nummer eines Semaphore Set, über die der Batch-Carrier seinen Erfolg zurückmeldet. Der Batch-Carrier überprüft nun zunächst, ob bereits eine Sicherungs-Datei vorliegt. Ist das der Fall, so liest er die Kennungen aller bereits abgearbeiteten Subworkflows aus dieser Datei und legt sie in einem Verzeichnis ab, dessen Einträge bei der nächsten Ausführung übersprungen werden. Parallel dazu werden, entsprechend der Angaben in der Konfigurations-Datei, ein oder mehrere Subworkflowcontroller gestartet, indem zunächst ein Master-JVM und dann entsprechend viele Worker-JVMs in einem JVM-Set gestartet werden, und die Worker angewiesen werden, den Batch-Stub auszuführen und damit den SWFC zu initialisieren.	

Use Case BATCH.1	Batch-Job ausführen (fortgesetzt)
	Nun liest der Batch-Carrier einen Auftrag nach dem anderen aus der Job-Datei und stellt sie, falls sie nicht im Verzeichnis der bereits bearbeiteten Aufträge eingetragen sind, in die Eingabe-Queue der Shared Memory Region. Aus dieser Queue liest ein SWFC mit Hilfe seines I/O-Controllers einen Auftrag und führt den entsprechenden Subworkflow aus. Meldungs-SRUs werden dabei durch den I/O-Controller in die Ausgabe-Queue der Shared Memory Region geschrieben, und vom Batch-Carrier zwischengespeichert. Erst am Ende einer Transaktion werden diese in die Ergebnis-Datei geschrieben. Dafür wird vom I/O-Controller ein Commit-Vorgang vom Typ *CHKP* initiiert. Wird eine Transaktion zurückgesetzt oder abgebrochen, so initiiert der I/O-Controller einen Commit-Vorgang vom Type *ROLB*, durch den alle für diesen Auftrag gesammelten Meldungs-SRUs verworfen werden. Am Ende des Subworkflow schreibt der SWFC über seinen I/O-Controller eine Ergebnis- oder Fehler-SRU in die Ausgabe-Queue und initiiert anschließend einen Commit-Vorgang vom Typ *COMMIT*. Durch diesen wird der Batch-Carrier veranlaßt, alle ausstehenden SRUs in die Ergebnis-Datei zu schreiben, den Subworkflow in die Sicherungs-Datei einzutragen und seinen Erfolg in der Log-Datei zu vermerken. Muß der Subworkflow aufgrund von Fehlern, die vom Subworkflow nicht abgefangen wurden, abgebrochen werden, so wird ein Commit-Vorgang vom Typ *ABORT* durchgeführt. Alle noch ausstehenden SRUs werden dabei verworfen, und eine entsprechende Meldung in die Log-Datei geschrieben. Nachdem alle SRWUs der Job-Datei ausgeführt wurden, werden die Subworkflowcontroller beendet, indem ihnen eine besondere Nachricht übergeben wird. Falls einige Subworkflows fehlgeschlagen (durch *ABORT* beendet) waren und dem Batch-Carrier über die Kommandozeile oder die Konfigurations-Datei eine Anzahl von Wiederholungen vorgegeben wurde, arbeitet dieser die gleiche Job-Datei noch einmal ab, wobei durch die Einträge in der Sicherungs-Datei verhindert wird, daß bereits abgeschlossene Subworkflows erneut ausgeführt werden.

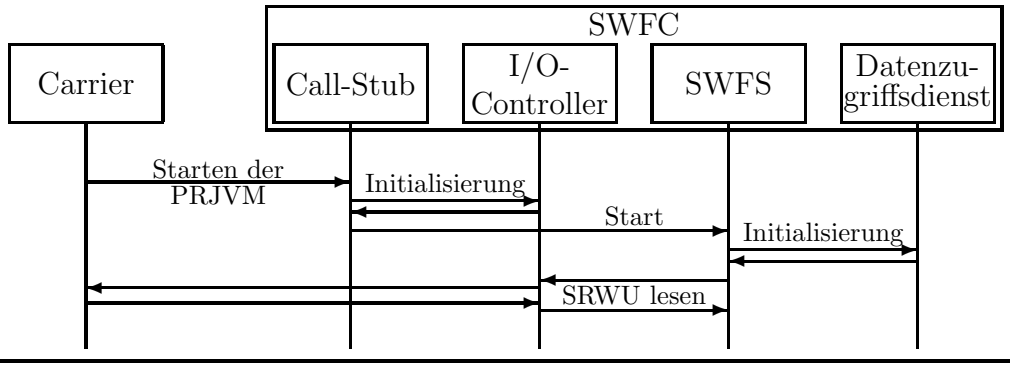
Use Case BATCH.1	Batch-Job ausführen (fortgesetzt)
	Ansonsten wird der Masterflowcontroller über die Semaphoren vom Abschluß des Batch-Job informiert. Den Status erfährt der Master aus der Ergebnis-Datei und aus einer Log-Datei, in der ausgeführte wie auch abgebrochene Aufträge verzeichnet wurden.



A.2 Der Subworkflowcontroller

Die folgenden Use Cases beschreiben den Start des Subworkflowcontrollers, die Ausführung eines Subworkflow und das Beenden des SWFC.

Use Case SWFC.1	**Start des SWFC**	
Thema	Fähigkeiten des Subworkflowcontrollers	
Ereignis	Es liegen neue Aufträge vor, der Carrier benötigt einen (weiteren) SWFC zur Ausführung derselben.	
Beteiligte Komponenten	Carrier, SWFC	
Kurzbeschreibung	Der Carrier startet einen neuen SWFC, der sich initialisiert und auf Aufträge wartet.	
Vorbedingungen	• Der Carrier ist initialisiert und braucht einen (weiteren) SWFC zur Ausführung neuer Aufträge. • Die Konfiguration des Subworkflowcontrollers ist vollständig und korrekt.	
Ergebnis		**Bedingung**
1. Der SWFC wartet auf Aufträge.		Der Server des Datenzugriffsdienstes ist erreichbar, es ist kein Fehler bei der Initialisierung aufgetreten.
2. Der Start des SWFC schlug fehl.		• Der Server ist nicht erreichbar. • Der Hauptspeicher reicht nicht aus.
Ablauf	Der Carrier startet die PRJVM und ruft den passenden Call-Stub auf. Dieser initialisiert zunächst den dazugehörigen I/O-Controller, und übergibt dann die Kontrolle an den Subworkflow-Supervisor, der sogleich den Datenzugriffsdienst initialisiert. Der Datenzugriffsdienst überprüft bei seiner Initialisierung, ob der Server erreichbar ist. Wenn nicht, gibt er eine Fehlermeldung zurück. Schließlich fordert der SWFS den ersten Auftrag in Form einer SRWU an. Die Initialisierung ist erfolgreich abgeschlossen.	

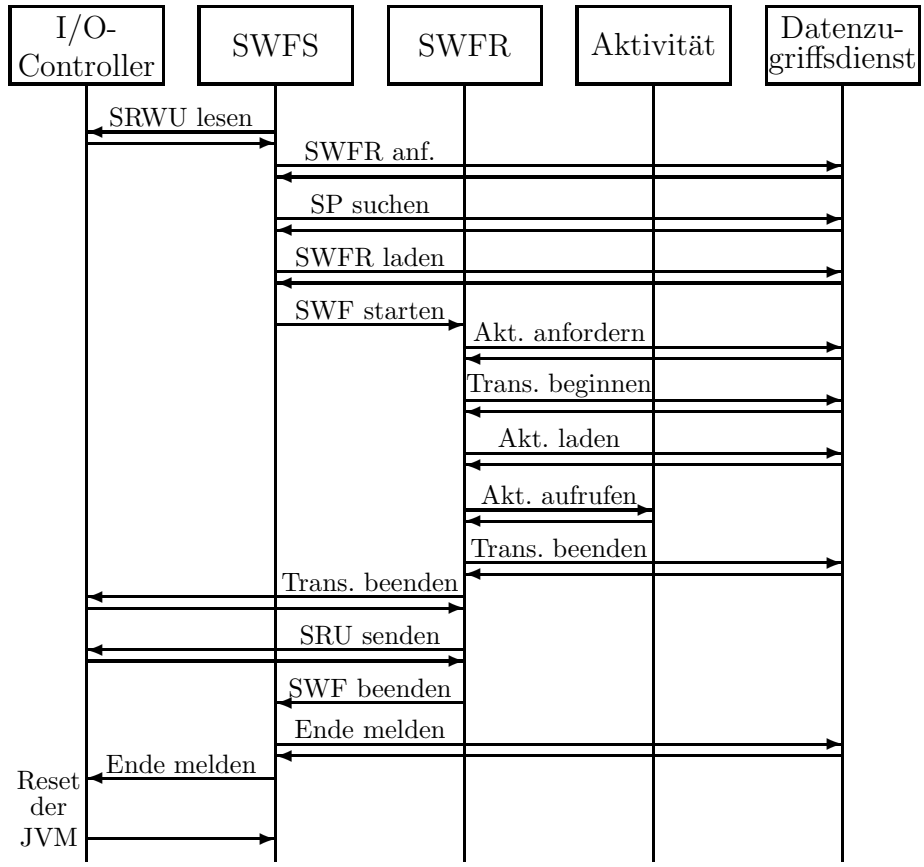


Use Case SWFC.2	**Ausführen eines Subworkflow**
Thema	Fähigkeiten des Subworkflowcontrollers
Ereignis	Ein Subworkflow liegt zur Bearbeitung vor.
Beteiligte Komponenten	SWFC
Kurzbeschreibung	Der Subworkflowcontroller liest eine SRWU ein, initialisiert den Subworkflow, führt ihn aus und zeigt das Ende der Bearbeitung an.
Vorbedingungen	Der Subworkflowcontroller ist ordnungsgemäß hochgefahren und wartet auf einen Auftrag.

Ergebnis	**Bedingung**
1. Der Subworkflowcontroller wartet auf den nächsten Auftrag.	Bei der Ausführung sind keine schwerwiegenden Fehler aufgetreten.
2. Der Subworkflowcontroller beendet sich. Dieses Ereignis wird in Use Case SWFC.3 auf Seite 84 beschrieben.	• Der Server ist nicht mehr erreichbar. • Der Hauptspeicher reicht nicht aus.

Ablauf	Eine Vorbemerkung zur Terminologie: In dieser Beschreibung wird, wie in Abschnitt 5.3 ab Seite 37, die Tatsache übergangen, daß der interne Datenzugriffsdienst für nahezu alle seine Aufgaben auf den externen Server zurückgreift. Die Use Cases in Abschnitt A.3 ab Seite 85 widmen sich der hier ausgelassenen Thematik. Aus der eingelesenen SRWU wird die Kennung und die Bezeichnung des Subworkflow ausgelesen. Anhand der Bezeichnung wird die passende Subworkflow-Runtime über den Datenzugriffsdienst angefordert und anschließend nachgesehen, ob dieser Subworkflow bereits zuvor gestartet und abgebrochen ist. Dazu wird dem Datenzugriffsdienst die Kennung übergeben, und dieser sucht nach einem unter dieser Kennung abgelegten Savepoint. Nun wird die SWFR geladen und aufgerufen. Diese führt alle Anweisungen des Subworkflow aus, und greift dafür auf den Datenzugriffsdienst (Transaktionen, Savepointing, Laden von Aktivitäten und Datenbankzugriffe) und den I/O-Controller (SRUs schreiben, Ende der Transaktionen mitteilen) zu. Im Beispiel-SWF von Abbildung A.1 auf Seite 77 wird zunächst über den DS die Aktivität angefordert, dann eine Transaktion begonnen, wobei dem DS in einem Savepoint die Daten der SRWU zur Sicherung übergeben werden. Anschließend wird das Vorhandensein der Aktivität geprüft und diese ausgeführt, und schließlich wird die Transaktion über den DS beendet und die Kontrolle an den SWFS zurückgegeben.

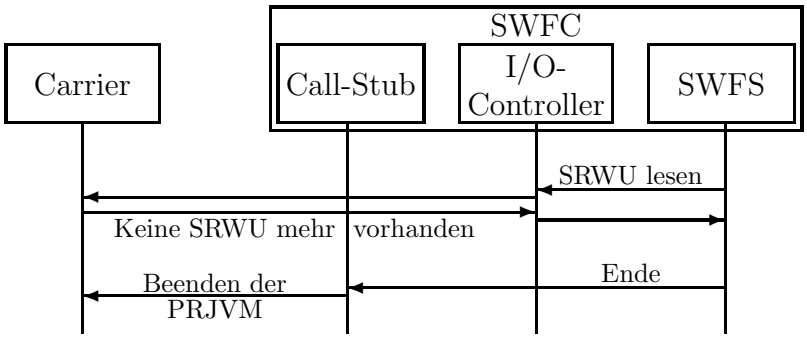
Use Case SWFC.2	**Ausführen eines Subworkflow (fortgesetzt)**
	Der SWFS teilt nun dem I/O-Controller das Ende des Subworkflow mit, der daraufhin einen Reset der PRJVM veranlaßt. Danach kann es von vorne weitergehen. Trat allerdings während der Verarbeitung ein Fehler auf, der nicht vom Subworkflow eingeplant war, so schlägt der Subworkflow fehl, und der Subworkflowcontroller samt PRJVM wird beendet, indem die Kontrolle an den Carrier zurückgegeben wird.



Use Case SWFC.3	**Ende des SWFC**
Thema	Fähigkeiten des Subworkflowcontrollers
Ereignis	Es liegen keine neuen Aufträge mehr vor.
Beteiligte Komponenten	Carrier, SWFC
Kurzbeschreibung	Der SWFC beendet sich, indem er die Kontrolle an den Carrier zurückgibt.
Vorbedingungen	• Der SWFC ist korrekt initialisiert. • Es liegen keine SRWUs mehr vor.

Ergebnis	**Bedingung**
1. Der Subworkflowcontroller ist beendet.	*Keine*

Ablauf	Bei einer Anfrage an den I/O-Controller kann kein weiterer Auftrag mehr gelesen werden. Der SWFS gibt die Kontrolle an den Call-Stub zurück, der den Subworkflowcontroller beendet, indem er seinerseits zum Carrier zurückkehrt.



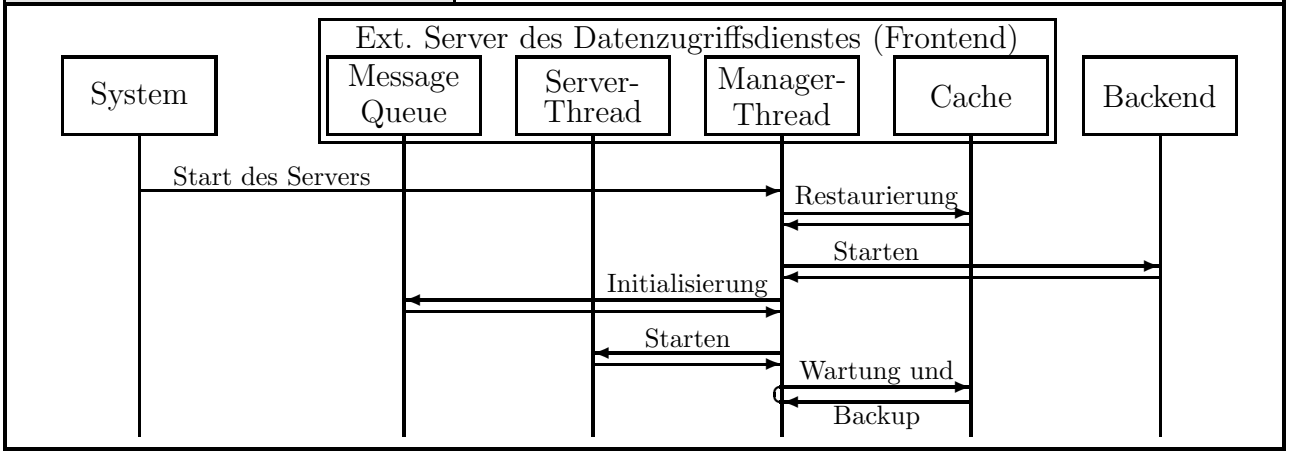
A.3 Der Datenzugriffsdienst

Die folgenden Use Cases beschreiben die Funktionalität des Datenzugriffsdienstes. Dabei wird von der noch nicht implementierten Kommunikation über System V Message Queues und der Verwendung eines über die WLM-Services angebundenen Backend ausgegangen.

Use Case DS.1	**Start des externen Servers**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Auf einem Rechner soll ein Subworkflowcontroller ausgeführt werden.
Beteiligte Komponenten	System, Server, Backend
Kurzbeschreibung	Der Server liest seine Konfiguration, restauriert seinen Cache und wartet auf Anfragen.
Vorbedingungen	• Die Konfiguration des DS ist vollständig und korrekt. • Die voreingestellte Queue-ID ist nicht belegt.

Ergebnis	**Bedingung**
1. Der Server wartet auf Anfragen.	Bei der Initialisierung traten keine Fehler auf.
2. Der Server konnte nicht gestartet werden.	Der Speicher reicht nicht aus.

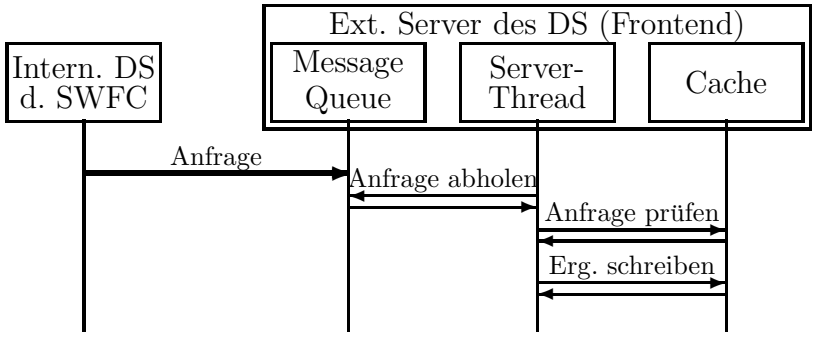
Ablauf	Beim Start des externen Servers (durch den Batch-Carrier, durch einen Systemdienst oder von der Konsole) liest der Manager-Thread zunächst die Konfigurationsdatei, restauriert dann den Cache aus einem eventuell noch vorhandenen Backup, und startet entsprechend den Angaben in der Konfigurationsdatei eine Anzahl von Backends, die sich initialisieren und dann auf Aufträge warten. Anschließend werden die Message-Queues initialisiert und entsprechend der Konfiguration eine Anzahl von Server-Threads gestartet, die auf Anfragen der SWFCs über die Message Queues warten. Der Manager-Thread geht dann in eine Schleife, in der er den Cache wartet und regelmäßig Backups zieht.



Use Case DS.2	**Asynchrone Anfrage ohne Backend**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Der interne Datenzugriffsdienst erhält eine asynchrone Anfrage.
Beteiligte Komponenten	Interner DS, Server
Kurzbeschreibung	Der interne DS übergibt die Anfrage über die Message Queue an einen Server-Thread, der sie ausführt und das Ergebnis speichert.
Vorbedingungen	• Der Server ist aktiv. • Ein Server-Thread ist frei. • Die Bearbeitung der Anfrage ist wenig aufwendig.

Ergebnis	**Bedingung**
1. Der Server-Thread wartet auf weitere Anfragen.	*Keine*

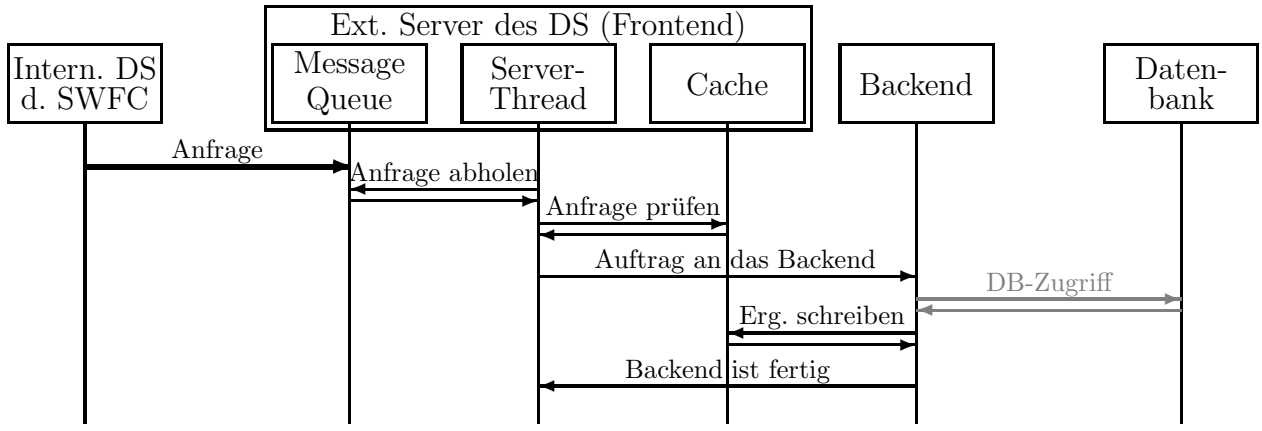
Ablauf	Der interne DS übergibt die Anfrage an die Message Queue des Servers. Aus dieser wird sie anschließend von einem freien Server-Thread ausgelesen und im Cache vermerkt. Nach Abschluß der Bearbeitung durch den Server-Thread schreibt dieser sein Ergebnis in den Cache und ist damit frei für die nächste Anfrage.



Use Case DS.3	**Asynchrone Anfrage mit Backend**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Der interne Datenzugriffsdienst erhält eine asynchrone Anfrage.
Beteiligte Komponenten	Interner DS, Server, Backend
Kurzbeschreibung	Der interne DS übergibt die Anfrage über die Message Queue an einen Server-Thread, der sie durch ein Backend ausführen läßt und das Ergebnis speichert.
Vorbedingungen	• Der Server ist aktiv. • Ein Server-Thread ist frei. • Die Bearbeitung der Anfrage ist so aufwendig, daß es sich lohnt, damit ein Backend zu beauftragen (z.B. Datenbankzugriff).

Ergebnis	**Bedingung**
1. Der Server-Thread wartet auf weitere Anfragen.	*Keine*

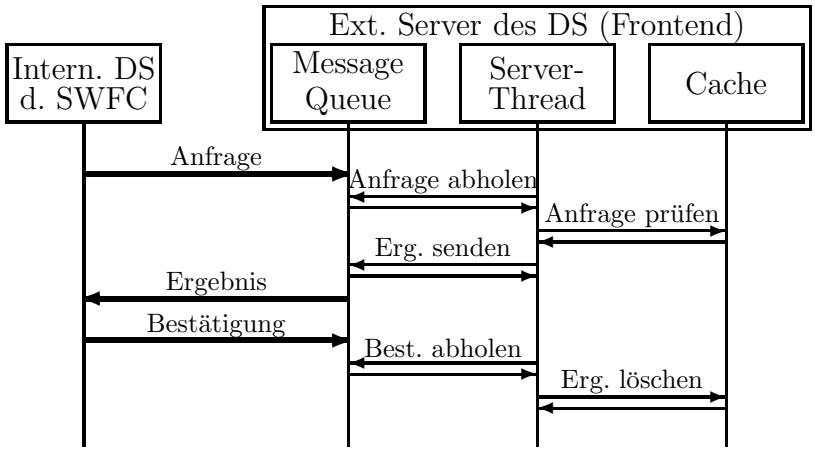
Ablauf	Der interne DS übergibt die Anfrage an die Message Queue des Servers. Aus dieser wird sie anschließend von einem freien Server-Thread ausgelesen und im Cache vermerkt. Die weitere Bearbeitung wird an ein Backend delegiert, das eventuell auch Datenbankzugriffe durchführt. Nach Abschluß seiner Arbeit schreibt das Backend sein Ergebnis in den Cache und meldet dem Server-Thread, daß es fertig ist.



Use Case DS.4	**Wiederholte synchrone Anfrage**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Der interne Datenzugriffsdienst erhält eine synchrone Anfrage.
Beteiligte Komponenten	Interner DS, Server
Kurzbeschreibung	Die Anfrage ist Wiederholung einer fehlgeschlagenen oder zuvor asynchron durchgeführten Anfrage. Der interne DS übergibt die Anfrage über die Message Queue an einen Server-Thread, der das Ergebnis aus dem Cache liest und zurückliefert. Vom internen DS erfolgt daraufhin die Bestätigung des Empfangs.
Vorbedingungen	• Der Server ist aktiv. • Ein Server-Thread ist frei. • Das Ergebnis der Bearbeitung liegt bereits im Cache vor.

Ergebnis	**Bedingung**
1. Der Server-Thread wartet auf weitere Anfragen.	*Keine*

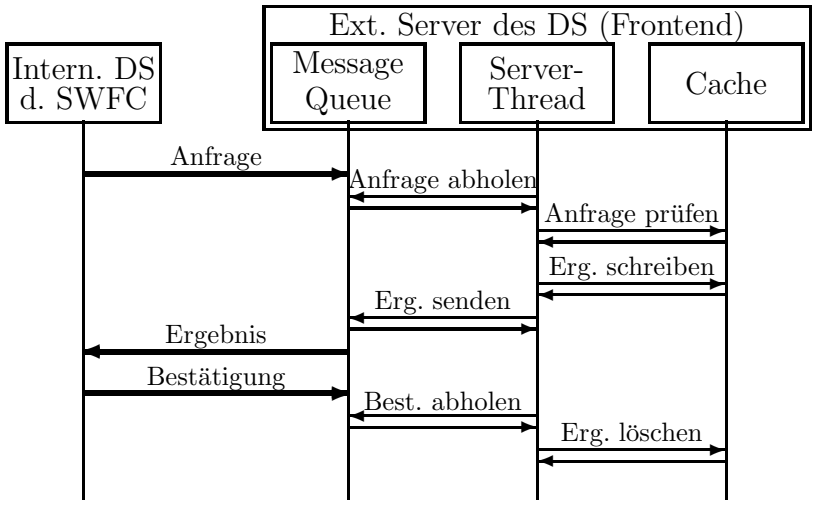
Ablauf	Der interne DS übergibt die Anfrage an die Message Queue des Servers. Aus dieser wird sie anschließend von einem freien Server-Thread ausgelesen und im Cache nachgesehen, ob sie bereits ausgeführt wurde. Dies ist der Fall, also übergibt der Server-Thread das Ergebnis durch die Message Queue dem internen DS. Danach kann sich dieser Server-Thread einer neuen Aufgabe widmen. Die Bestätigung des internen DS wird von einem anderen Server-Thread gelesen, der daraufhin das Ergebnis der bestätigten Anfrage verwirft.



Use Case DS.5	**Synchrone Anfrage ohne Backend**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Der interne Datenzugriffsdienst erhält eine synchrone Anfrage.
Beteiligte Komponenten	Interner DS, Server
Kurzbeschreibung	Der interne DS übergibt die Anfrage über die Message Queue an einen Server-Thread, der sie ausführt und das Ergebnis zurückliefert, das schließlich vom internen DS bestätigt wird.
Vorbedingungen	• Der Server ist aktiv. • Ein Server-Thread ist frei. • Die Bearbeitung der Anfrage ist wenig aufwendig.

Ergebnis	**Bedingung**
1. Der Server-Thread wartet auf weitere Anfragen.	*Keine*

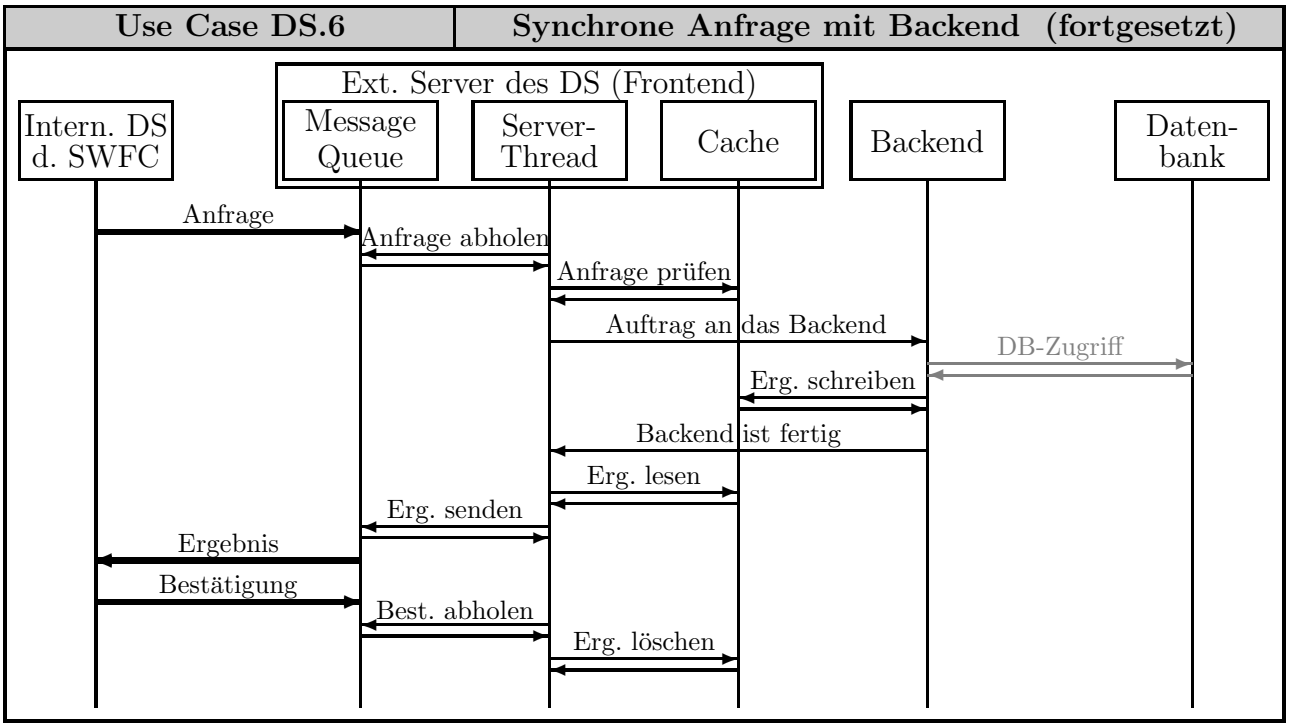
Ablauf	Der interne DS übergibt die Anfrage an die Message Queue des Servers. Aus dieser wird sie anschließend von einem freien Server-Thread ausgelesen und im Cache nachgesehen, ob sie bereits ausgeführt wurde. Dies ist nicht der Fall, deshalb bearbeitet der Server-Thread die Anfrage, schreibt sein Ergebnis in den Cache und übergibt es durch die Message Queue dem internen DS. Danach kann sich dieser Server-Thread einer neuen Aufgabe widmen. Die Bestätigung des internen DS wird von einem anderen Server-Thread gelesen, der daraufhin das Ergebnis der bestätigten Anfrage verwirft.



Use Case DS.6	**Synchrone Anfrage mit Backend**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Der interne Datenzugriffsdienst erhält eine synchrone Anfrage.
Beteiligte Komponenten	Interner DS, Server, Backend
Kurzbeschreibung	Der interne DS übergibt die Anfrage über die Message Queue an einen Server-Thread, der sie durch ein Backend ausführen läßt und das Ergebnis zurückliefert, das schließlich vom internen DS bestätigt wird.
Vorbedingungen	• Der Server ist aktiv. • Ein Server-Thread ist frei. • Die Bearbeitung der Anfrage ist so aufwendig, daß es sich lohnt, damit ein Backend zu beauftragen (z.B. Datenbankzugriff).

Ergebnis	**Bedingung**
1. Der Server-Thread wartet auf weitere Anfragen.	*Keine*

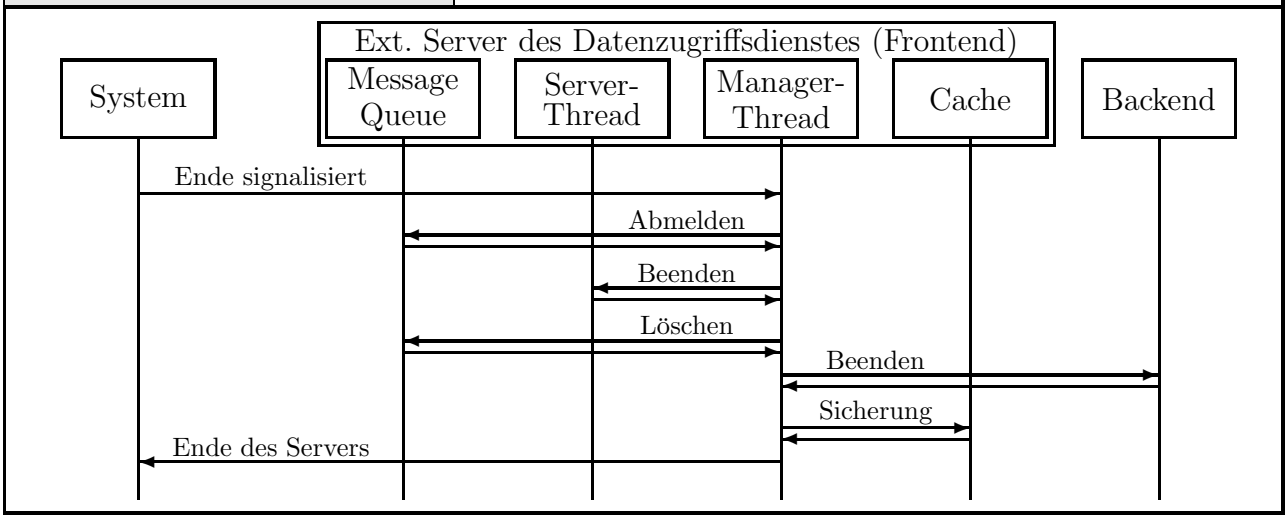
Ablauf	Der interne DS übergibt die Anfrage an die Message Queue des Servers. Aus dieser wird sie anschließend von einem freien Server-Thread ausgelesen und im Cache nachgesehen, ob sie bereits ausgeführt wurde. Dies ist nicht der Fall, deshalb wird die weitere Bearbeitung an ein Backend delegiert, das eventuell auch Datenbankzugriffe durchführt. Nach Abschluß seiner Arbeit schreibt das Backend sein Ergebnis in den Cache und meldet dem Server-Thread, daß es fertig ist. Dieses liest das Ergebnis aus dem Cache und übergibt es durch die Message Queue dem internen DS. Danach kann sich dieser Server-Thread einer neuen Aufgabe widmen. Die Bestätigung des internen DS wird von einem anderen Server-Thread gelesen, der daraufhin das Ergebnis der bestätigten Anfrage verwirft.


Use Case DS.6
Synchrone Anfrage mit Backend (fortgesetzt)
Ext. Server des DS (Frontend)
Intern. DS d. SWFC
Message Queue
Server-Thread
Cache
Backend
Daten-bank
Anfrage
Anfrage abholen
Anfrage prüfen
Auftrag an das Backend
DB-Zugriff
Erg. schreiben
Backend ist fertig
Erg. lesen
Erg. senden
Ergebnis
Bestätigung
Best. abholen
Erg. löschen

Use Case DS.7	**Beenden des externen Servers**
Thema	Funktionalität des Datenzugriffsdienstes
Ereignis	Der Server erhält ein Signal zum Beenden (`SIGBRK`, `SIGINT`, `SIGKILL`)
Beteiligte Komponenten	System, Server, Backend
Kurzbeschreibung	Der Server deaktiviert die Message Queue, beendet alle Threads und Backends und sichert den Cache.
Vorbedingungen	Der Server ist aktiv.

Ergebnis	**Bedingung**
1. Der Server wurde beendet.	*Keine*

Ablauf	Wenn der Server (genauer der Manager-Thread) ein Signal zum Beenden erhält, also ein `SIGBRK`, `SIGINT` oder `SIGKILL`, dann deaktiviert er zunächst die Message Queue und signalisiert dann den Server-Threads, daß sie sich beenden sollen. Wenn alle Threads ihre Arbeit beendet haben, wird die Message Queue gelöscht und die Backends terminiert. Schließlich wird noch ein Backup des Cache geschrieben und der Server beendet.



Anhang B

Umsetzung einer BPDL-Beschreibung in Java-Programme

In diesem Anhang wird ein einfaches, aber vollständiges Beispiel für die Umsetzung der BPDL-Beschreibungen eines Subworkflow mit Datenbankzugriff und einer Java-Aktivität durch die Schnittstellen- und Subworkflow-Compiler des Business Process Designers gezeigt. Ausführliche Beispiele für BPDL-Beschreibungen aller Komponenten der Transaktionsmaschine finden sich in Abschnitt C.2 ab Seite 139.

B.1 Die Beschreibungsdateien

Im folgenden sind BPDL-Schnittstellenbeschreibungen für einen Subworkflow (Quelltext 1), eine Aktivität (Quelltext 2 ab Seite 94) und einen Zugriff auf eine Datenbank (Quelltext 3 ab Seite 95) mit der passenden Datenbankbeschreibung (Quelltext 4 auf Seite 96) sowie die BPDL-Workflowbeschreibung eines Subworkflow (Quelltext 5 ab Seite 96) abgebildet. Die von den Compilern des Business Process Designers aus den dargestellten BPDL-Beschreibungen erzeugten Programme finden sich in Abschnitt B.2 ab Seite 100.

Quelltext 1: Subworkflow_V1.ibpdl

```
<definitions name="Subworkflow" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/Subworkflow_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- subworkflow settings -->
  <bpdl:settings name="http://example.com/bpdl/parameters/Subworkflow_V1"/>

  <!-- input and output messages -->
  <message name="subworkflowInputSRWU">
    <part name="creditCardNumber" type="xsd:string" />
```

```
    <part name="destinationAccount" type="xsd:string" />
    <part name="amount" type="xsd:decimal" />
  </message>

  <message name="subworkflowOutputSRU">
    <part name="amount" type="xsd:decimal" />
    <part name="balance" type="xsd:decimal" />
  </message>

  <!-- interface -->
  <portType name="subworkflowPT">
    <operation name="subworkflow">
      <input message="subworkflowInputSRWU" />
      <output message="subworkflowOutputSRU" />
    </operation>
  </portType>

  <plnk:partnerLinkType name="subworkflowLT">
    <plnk:role name="subworkflow">
      <plnk:portType name="subworkflowPT" />
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 2: JavaActivity_V1.ibpdl

```
<definitions name="JavaActivity" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/JavaActivity_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- input, output and error messages -->
  <message name="inMsg">
    <part name="balance" type="xsd:decimal"/>
    <part name="amount" type="xsd:decimal"/>
  </message>

  <message name="outMsg">
    <part name="result" type="xsd:decimal"/>
  </message>

  <message name="faultMsg">
    <part name="reason" type="xsd:int"/>
  </message>
```

```
  <!-- interface -->
  <portType name="javaActivityPT">
    <operation name="javaActivity">
      <input message="inMsg"/>
      <output message="outMsg"/>

      <!-- exception to be thrown in case of an error -->
      <fault name="fault" message="faultMsg"/>
      <bpdl:exec lang="Java" method="run"/>
    </operation>
  </portType>

  <plnk:partnerLinkType name="javaActivityLT">
    <plnk:role name="javaActivity">
      <plnk:portType name="javaActivityPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 3: Database_V1.ibpdl

```
<definitions name="Database" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/Database_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:db="http://example.com/bpdl/database/Database_V1"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- input, output and fault messages for read operation -->
  <message name="dbReadRequestMsg">
    <part name="creditCardNumber" type="xsd:string"/>
  </message>

  <message name="dbReadResponseMsg">
    <part name="balance" type="xsd:decimal"/>
  </message>

  <message name="dbReadFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- input, output and fault messages for write operation -->
  <message name="dbWriteRequestMsg">
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="balance" type="xsd:decimal"/>
  </message>
```

```
  <message name="dbWriteFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- interface -->
  <portType name="dbAccessPT">
    <operation name="read">
      <input message="dbReadRequestMsg"/>
      <output message="dbReadResponseMsg"/>
      <fault name="dbReadError" message="dbReadFaultMsg"/>
      <bpdl:database name="db:DB" type="DB2">
        <bpdl:select field="BALANCE" table="CREDITCARDS"
          where="CCNUM='${creditCardNumber}'" mode="update"/>
        <bpdl:destination part="balance"/>
      </bpdl:database>
    </operation>
    <operation name="write">
      <input message="dbWriteRequestMsg"/>
      <fault name="dbWriteError" message="dbWriteFaultMsg"/>
      <bpdl:database name="db:DB" type="DB2">
        <bpdl:update field="BALANCE" table="CREDITCARDS"
          where="CCNUM='${creditCardNumber}'"/>
        <bpdl:source part="balance"/>
      </bpdl:database>
    </operation>
  </portType>

  <plnk:partnerLinkType name="dbLT">
    <plnk:role name="db">
      <plnk:portType name="dbPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 4: Database_V1.dbpdl

```
<database name="Database" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/database/Database_V1"
  xmlns="http://www.schlodder.de/schemas/bpdl/d/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <table name="CREDITCARDS">
    <field name="CCNUM" type="xsd:string"/>
    <field name="OWNER" type="xsd:string"/>
    <field name="BALANCE" type="xsd:decimal"/>
  </table>
</database>
```

Quelltext 5: Subworkflow_V1.fbpdl

```
<process name="Subworkflow" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/workflow/Subworkflow_V1"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/f/"
  xmlns:jai="http://example.com/bpdl/interface/JavaActivity_V1"
  xmlns:dbi="http://example.com/bpdl/interface/Database_V1"
  xmlns:swfi="http://example.com/bpdl/interface/Subworkflow_V1">

  <partnerLinks>

    <!-- PartnerLink for the subworkflow -->
    <partnerLink name="SWF"
      partnerLinkType="swfi:subworkflowLT"
      myRole="subworkflow"
      partnerRole="subworkflowMessage"/>

    <!-- PartnerLink for database access -->
    <partnerLink name="db"
      partnerLinkType="dbi:dbLT"
      partnerRole="db"/>

    <!-- PartnerLink for JavaActivity -->
    <partnerLink name="javaActivity"
      partnerLinkType="jai:javaActivityLT"
      partnerRole="javaActivity"/>
  </partnerLinks>

  <!-- partners selects the external interface of the subworkflow -->
  <partners>
    <partner name="Subworkflow">
      <partnerLink name="SWF"/>
    </partner>
  </partners>

  <!-- global variables -->
  <variables>
    <variable name="inSRWU" messageType="swfi:subworkflowInputSRWU"/>
    <variable name="outSRU" messageType="swfi:subworkflowOutputSRWU"/>
  </variables>

  <!-- start of subworkflow code -->
  <sequence>

    <!-- receive input SRWU -->
    <receive partnerLink="SWF" portType="swfi:subworkflowPT"
      operation="subworkflow" variable="inSRWU"/>
```

```
    <!-- begin transaktion -->
    <bpdl:transaction inputVariable="inSRWU" outputVariable="outSRU">
      <variables>

        <!-- variables for JavaAktivity -->
        <variable name="msgIn" messageType="jai:inMsg"/>
        <variable name="msgOut" messageType="jai:outMsg"/>
        <variable name="msgErr" messageType="jai:faultMsg"/>

        <!-- variables for database access -->
        <variable name="dbReadParam" messageType="dbi:dbReadRequestMsg"/>
        <variable name="dbReadRes" messageType="dbi:dbReadResponseMsg"/>
        <variable name="dbReadErr" messageType="dbi:dbReadFaultMsg"/>
        <variable name="dbWriteParam" messageType="dbi:dbWriteRequestMsg"/>
        <variable name="dbWriteErr" messageType="dbi:dbWriteFaultMsg"/>
      </variables>

      <!-- start of transaction code -->
      <sequence>

        <!-- read database entry -->
        <assign>
          <copy>
            <from variable="inSRWU" part="creditCardNumber"/>
            <to variable="dbReadParam" part="creditCardNumber"/>
          </copy>
        </assign>

        <invoke partnerLink="db" portType="dbi:dbPT"
          operation="read" inputVariable="dbReadParam"
          outputVariable="dbReadRes" bpdl:timeout="2s">
          <catch faultName="dbi:dbReadError" faultVariable="dbReadErr">

            <!-- rollback transaktion -->
            <bpdl:rollback/>
          </catch>
        </invoke>

        <!-- set savepoint -->
        <bpdl:createSavepoint name="savepoint1"/>

        <!-- call JavaAktivity -->
        <assign>
          <copy>
            <from variable="inSRWU" part="amount"/>
            <to variable="msgIn" part="amount"/>
```

```
      </copy>
      <copy>
        <from variable="dbReadRes" part="balance"/>
        <to variable="msgIn" part="balance"/>
      </copy>
    </assign>

    <invoke partnerLink="debit" portType="jai:javaActivityPT"
      operation="javaActivity" inputVariable="msgIn"
      outputVariable="msgOut">
      <catch faultName="jai:fault" faultVariable="msgErr">

        <!-- restore savepoint -->
        <bpdl:restoreSavepoint name="savepoint1"/>
      </catch>
    </invoke>

    <!-- set savepoint -->
    <bpdl:createSavepoint name="savepoint2"/>

    <!-- update database -->
    <assign>
      <copy>
        <from variable="inSRWU" part="creditCardNumber"/>
        <to variable="dbWriteParam" part="creditCardNumber"/>
      </copy>
      <copy>
        <from variable="msgOut" part="balance"/>
        <to variable="dbWriteParam" part="balance"/>
      </copy>
    </assign>

    <invoke partnerLink="db" portType="dbi:dbPT"
      operation="write" inputVariable="dbWriteParam" bpdl:timeout="2s">
      <catch faultName="dbi:dbWriteError" faultVariable="dbWriteErr">

        <!-- restore savepoint -->
        <bpdl:restoreSavepoint name="savepoint2"/>
      </catch>
    </invoke>

    <!-- end transaktion -->
    <assign>
      <copy>
        <from variable="inSRWU" part="amount"/>
        <to variable="outSRU" part="amount"/>
      </copy>
```

```
        <copy>
          <from variable=" msgOut" part="balance"/>
          <to variable="outSRU" part="balance"/>
        </copy>
      </assign>
    </sequence>
  </bpdl:transaction>

  <!-- reply output SRU -->
  <reply partnerLink="SWF" portType="swfi:subworkflowPT"
    operation="subworkflow" variable="outSRU"/>
 </sequence>
</process>
```

B.2 Die erzeugten Java-Programme

Durch den Schnittstellen-Compiler wurden eine Java-Aktivität (Quelltext 6, hier wurde etwas Beispiel-Code eingefügt) und die zugehörige Interface-Klasse (Quelltext 7 auf Seite 101) generiert, außerdem Eingabedaten-, Ausgabedaten- und Fehlerdaten-Klassen (Quelltext 8 auf Seite 101 und Quelltext 9 und Quelltext 10 auf Seite 102) sowie die Exception, die im Fehlerfall geworfen wird (Quelltext 11 auf Seite 102). Der Subworkflow-Compiler sollte, wenn er fertiggestellt und optimiert ist, ein Java-Programm generieren, das dem in Quelltext 12 ab Seite 102 entspricht.

Quelltext 6: JavaActivity_V1.java

```
/*
 * Demonstration of a Java activity.
 */

package app;

import swfc.*;

/*
 * Java activity.
 */
public class JavaActivity_V1 extends JavaActivity implements iJavaActivity_V1 {

  /*
   * Version constructor.
   */
  public JavaActivity_V1() {
    super("JavaActivity", 1, 0);
  }

```

```
  /*
   * Entry point.
   */
  public JavaActivity_V1_outMsg run(JavaActivity_V1_inMsg input)
    throws JavaActivity_V1_faultException {

    /* Start of user code. */
    JavaActivity_V1_OutMsg output = new JavaActivity_V1_OutMsg();

    try {
      output.result = input.creditCardBalance - input.amount;
    }
    catch (ArithmeticException e) {
      JavaActivity_V1_FaultException error =
        new JavaActivity_V1_FaultException(new JavaActivity_V1_faultMsg());
      error.reason = 0;
      throw error;
    }
    return output;
    /* End of user code. */
  }
}
```

Quelltext 7: iJavaActivity_V1.java

```
package swfc;

/* Interface class for JavaActivity_V1. */
public interface iJavaActivity_V1 {

  /* Entry point of the activity. */
  public JavaActivity_V1_outMsg run(JavaActivity_V1_inMsg input)
    throws JavaActivity_V1_faultException;
}
```

Quelltext 8: JavaActivity_V1_inMsg.java

```
package swfc;

public class JavaActivity_V1_inMsg {

  /* Variables of the input message. */
  public java.math.BigDecimal balance;
  public java.math.BigDecimal amount;
}
```

Quelltext 9: JavaActivity_V1_outMsg.java

```
package swfc;

public class JavaActivity_V1_outMsg {

  /* Variables of the output message. */
  public java.math.BigDecimal result;
}
```

Quelltext 10: JavaActivity_V1_faultMsg.java

```
package swfc;

public class JavaActivity_V1_faultMsg {

  /* Variables of the fault message. */
  public int reason;
}
```

Quelltext 11: JavaActivity_V1_faultException.java

```
package swfc;

public class JavaActivity_V1_faultException extends Exception {

  /* Content of the exception. */
  public JavaActivity_V1_faultMsg faultMsg;

  /* Empty constructor. */
  public JavaActivity_V1_faultException(JavaActivity_V1_faultMsg msg) {
    super();
    faultMsg = msg;
  }
}
```

Quelltext 12: Subworkflow_V1.java

```
/*
 * Demonstration of a subworkflow runtime.
 */

package swfc;

import swfc.*;
import java.math.BigDecimal;

/*
 * Subworkflow runtime.
```

```
 */
public final class Subworkflow_V1 extends SWFR {

  /*
   * Version constructor.
   */
  public Subworkflow_V1() {
    super("Subworkflow", 1, 0);
  }

  /* global variables */
  private int state = 0;
  private String creditCardNumber;
  private String destinationAccount;
  private BigDecimal amount;
  private BigDecimal balance;

  /*
   * Entry point of subworkflow runtime. Specified in Java interface SWFR.
   */
  void run(IOController ioc, DataService ds, SavePoint savepoint, SRWU srwu,
          int verb) throws Exception {

    /* preload activities */
    ds.loadJavaObjectAsynch(srwu.getId(), "app.JavaActivity_V1");

    /* create database containers */
    DBContainer db1 = new DBContainer();

    /* check savepoint */
    if (savepoint != null) {
      state = savepoint.getSavepoint();
      setVars(savepoint, true);
      savepoint.clear();
    }
    else {
      /* initialize variables */
      savepoint = new SavePoint(3);

      /* parse srwu */
      creditCardNumber = new String(srwu.content, 0, 19, "8859_1");
      savepoint.setVar(0, creditCardNumber.getBytes("IBM-1047"));
      destinationAccount = new String(srwu.content, 19, 17, "8859_1");
      savepoint.setVar(1, destinationAccount.getBytes("IBM-1047"));
      amount = new BigDecimal(new String(srwu.content, 19 + 17 + 1,
                                         srwu.content[19 + 17], "8859_1"));
      savepoint.setVar(2, amount.toString().getBytes("IBM-1047"));
```

```
60      /* preload database data */
        db1.type = new DB.Type[] {DB.Type.decimal};
        ds.readDBAsynch(srwu.getId(), db1, DB.Kind.DB2, "DB", "CREDITCARDS",
                        "BALANCE", "CCNUM =" + creditCardNumber, DB.Mode.update);
      }
65
      /* run subworkflow */
      do {
        switch (state) {
          case 0:
70        state += 1;
          savepoint.setSavepoint(state);
          ds.beginTransaction(srwu.getId(), savepoint);
          savepoint.clear();

75        case 1:
          state += 1;
          try {
            db1 = ds.readDB(srwu.getId(), db1, DB.Kind.DB2, "DB", "CREDITCARDS",
                            "BALANCE", "CCNUM =" + creditCardNumber, DB.Mode.update);
80        }
          catch (ServerException e) {
            savepoint = ds.rollbackTransaction(srwu.getId());
            ioc.abortTransaction();
            setVars(savepoint, true);
85          state = savepoint.getSavepoint();
            savepoint.clear();
            continue;
          }
          balance = new BigDecimal(new String(db1.content[0], "IBM-1047"));
90        savepoint.setVar(3, balance.getBytes("IBM-1047"));
          savepoint.setSavepoint(state);
          ds.setSavepoint(srwu.getId(), savepoint);
          savepoint.clear();

95        case 2:
          state += 1;
          savepoint.setSavepoint(state);
          ds.loadJavaObject(srwu.getId(), "app.JavaActivity_V1");
          iJavaActivity_V1 act = (iJavaActivity_V1)Thread.currentThread().getContextClas
100 sLoader().loadClass("app.TestActivity_V1").newInstance();
          JavaActivity_V1_inMsg msgIn = {balance, amount};
          JavaActivity_V1_outMsg msgOut;
          try {
            msgOut = act.run(msgIn);
105       }
```

```
        catch (JavaActivity_V1_faultException e) {
          savepoint.setSavepoint(2);
          savepoint = ds.restoreSavepoint(srwu.getId(), savepoint);
          setVars(savepoint, false);
          state = savepoint.getSavepoint();
          savepoint.clear();
          continue;
        }
        balance = msgOut.result;
        savepoint.setVar(3, balance.getBytes("IBM-1047"));
        savepoint.setSavepoint(state);
        ds.setSavepoint(srwu.getId(), savepoint);
        savepoint.clear();

        case 3:
        state += 1;
        db1.content = new byte[][] {balance.toString.getBytes("IBM-1047")}
        try {
          db1 = ds.updateDB(srwu.getId(), db1, DB.Kind.DB2, "DB", "CREDITCARDS",
                            "BALANCE", "CCNUM =" + creditCardNumber);
        }
        catch (ServerException e) {
          savepoint.setSavepoint(3);
          savepoint = ds.restoreSavepoint(srwu.getId(), savepoint);
          setVars(savepoint, false);
          state = savepoint.getSavepoint();
          savepoint.clear();
          continue;
        }
        savepoint.setSavepoint(state);
        ds.setSavepoint(srwu.getId(), savepoint);
        savepoint.clear();

        case 4:
        state += 1;

        String amountS = amount.toString();
        int amountL = amountS.length();
        String balanceS = balance.toString();
        int balanceL = balanceS.length();
        byte[] msg = new byte[1 + amountL + 1 + balanceL];
        msg[0] = (byte)amountL;
        System.arraycopy(amountS.getBytes("IBM-1047"), 0, msg, 1, amountL);
        msg[1 + amountL] = (byte)balanceL;
        System.arraycopy(balanceS.getBytes("IBM-1047"), 0, msg,
                         1 + amountL + 1, balanceL);
        ioc.putSRU(new SRU(srwu.getId(), msg));
```

```
        savepoint.clear();
        savepoint.setSavepoint(state);
        ds.commitTransaction(srwu.getId(), savepoint);
        ioc.commitTransaction();

        case 5:
        state += 1;
      }
    } while(state < 6);
  }

  /*
   * Set global variables from save point.
   */
  private void setVars(SavePoint savepoint) {
    if (savepoint.checkVar(0)) {
      creditCardNumber = new String(savepoint.getVar(0), "8859_1");
    }
    else if (resetAll) {
      creditCardNumber = null;
    }
    if (savepoint.checkVar(1)) {
      destinationAccount = new String(savepoint.getVar(1), "8859_1");
    }
    else if (resetAll) {
      destinationAccount = null;
    }
    if (savepoint.checkVar(2)) {
      amount = new BigDecimal(new String(savepoint.getVar(2), "8859_1"));
    }
    else if (resetAll) {
      amount = null;
    }
    if (savepoint.checkVar(3)) {
      balance = new BigDecimal(new String(savepoint.getVar(3), "IBM-1047"));
    }
    else if (resetAll) {
      balance = null;
    }
  }
}
```

Anhang C

Definition der Business Process Definition Language mit Beispielen

In diesem Anhang werden die formalen Definitionen der vier Subsets der Business Process Definition Language (*BPDL*) sowie einige Beispiele aufgeführt, die zusammen einen Geschäftsprozeß samt Konfiguration der Transaktionsmaschine beschreiben. Dieser Geschäftsprozeß ist zwar nicht unbedingt realistisch, zeigt dafür aber alle in der BPDL neu eingeführten Konstrukte auf.

C.1 Formale Definition

Für die formale Definition der vier Subsets der BPDL wird der Einfachheit halber das jeweilige XML-Schema angegeben. XML-Schemata sind universell genug, um eine BNF-Definition von XML-basierten Sprachen zu ersetzen[1], und gehen teilweise sogar über die rein syntaktische Definition hinaus, da durch die Datentypen auch semantische Informationen[2] enthalten sein können.

C.1.1 Schnittstellenbeschreibungen

Da JAXB mit dem originalen XML-Schema von WSDL nicht zurecht kam, wurde für die Schnittstellenbeschreibungen ein neues XML-Schema erstellt. Dieses optimiert einerseits die Umsetzung durch JAXB, und erlaubt andererseits nur die Verwendung der gewünschten Elemente von WSDL. Und da JAXB außerdem mit verschachtelten Schemata derzeit nur schlecht zurechtkommt, wurden auch die Erweiterungen der BPDL/I integriert.

Die Umsetzung der externen Notation unter Angabe von WSDL als Basissprache und Einbindung der Erweiterungen von BPDL/I durch markierte Tags (wie in den Beispielen angegeben) in die interne Notation, die ausschließlich durch BPDL/I definiert ist, wird von den Compilern des Business Process Designers on-the-fly durchgeführt.

[1] BNF ist eine universelle Notation für kontextfreie Grammatiken. Die Syntax einer computerlesbaren Sprache kann generell durch eine kontextfreie Grammatik beschrieben werden. Daher läßt sich auch die Syntax von allen auf XML basierenden Sprachen durch eine kontextfreie Grammatik definieren.

Da XML-Schemata nun die Definition der Syntax beliebiger XML-basierter Sprachen erlauben, bieten sie eine universelle Notation für eben diese Syntax der auf XML basierenden Sprachen. Auch BNF kann nicht mehr leisten, als die Syntax einer Sprache zu definieren. Daher kann ein XML-Schema die BNF-Definition einer XML-basierten Sprache ersetzen.

[2] Datentypen zählen zur statischen Semantik von Programmiersprachen.

Im folgenden ist zunächst die auf WSDL 1.1 basierende, standardkonforme Erweiterung durch BPDL/I (Quelltext 13) aufgeführt, zusammen mit den relevanten Definitionen der WSDL (Quelltext 14 ab Seite 109) und der Partner-Links (Quelltext 15 auf Seite 115) und danach die intern verwendete, vollständige Definition der BPDL/I (Quelltext 16 ab Seite 116), die auch die verwendeten WSDL-Definitionen und die Partner-Link-Definitionen enthält.

Quelltext 13: BPDL/I

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:i="http://www.schlodder.de/schemas/bpdl/i/"
  targetNamespace="http://www.schlodder.de/schemas/bpdl/i/"
  elementFormDefault="qualified">

  <attribute name="version" type="unsignedShort"/>
  <attribute name="revision" type="unsignedShort"/>
  <attribute name="persistent" type="boolean"/>

  <element name="settings" type="i:tSettings"/>

  <complexType name="tSettings">
    <attribute name="name" type="anyURI" use="required"/>
  </complexType>

  <element name="exec" type="i:tExec"/>

  <complexType name="tExec">
    <attribute name="lang" type="NCName" use="required"/>
    <attribute name="object" type="NCName" use="optional"/>
    <attribute name="method" type="NCName" use="optional"/>
  </complexType>

  <element name="database" type="i:tDatabase"/>

  <complexType name="tDatabase">
    <sequence>
      <element name="select" type="i:tSelect" minOccurs="0"/>
      <element name="update" type="i:tUpdate" minOccurs="0"/>
      <element name="insert" type="i:tInsert" minOccurs="0"/>
      <element name="source" type="i:tSource" minOccurs="0"/>
      <element name="destination" type="i:tDestination" minOccurs="0"/>
    </sequence>
    <attribute name="name" type="QName" use="required"/>
    <attribute name="type" type="string" use="required"/>
  </complexType>

  <complexType name="tSelect">
    <attribute name="field" type="string" use="required"/>
    <attribute name="table" type="string" use="required"/>
    <attribute name="where" type="string" use="required"/>
    <attribute name="mode" type="string" use="optional"/>
  </complexType>

```

```
  <complexType name="tUpdate">
    <attribute name="field" type="string" use="required"/>
    <attribute name="table" type="string" use="required"/>
    <attribute name="where" type="string" use="required"/>
  </complexType>

  <complexType name="tInsert">
    <attribute name="table" type="string" use="required"/>
  </complexType>

  <complexType name="tSource">
    <attribute name="part" type="NCName" use="required"/>
  </complexType>

  <complexType name="tDestination">
    <attribute name="part" type="NCName" use="required"/>
  </complexType>
</schema>
```

Quelltext 14: WSDL 1.1

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--

Copyright 2001-2003 International Business Machines Corporation, Microsoft Corporation.
All rights reserved.

The presentation, distribution or other dissemination of the
information contained herein by Microsoft is not a license,
either expressly or impliedly, to any intellectual property owned or
controlled by Microsoft.

This document and the information contained herein is provided on an
"AS IS" basis and to the maximum extent permitted by applicable law,
Microsoft provides the document AS IS AND WITH ALL FAULTS, and hereby
disclaims all other warranties and conditions, either express, implied
or statutory, including, but not limited to, any (if any) implied
warranties, duties or conditions of merchantability, of fitness for a
particular purpose, of accuracy or completeness of responses, of
results, of workmanlike effort, of lack of viruses, and of lack of
negligence, all with regard to the document. ALSO, THERE IS NO
WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION,
CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THE
DOCUMENT.

IN NO EVENT WILL MICROSOFT BE LIABLE TO ANY OTHER PARTY FOR THE COST
OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE,
LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR
SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE,
ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THIS
DOCUMENT, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE
POSSIBILITY OF SUCH DAMAGES.
```

```
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://schemas.xmlsoap.org/wsdl/"
  elementFormDefault="qualified" >

  <xs:complexType mixed="true" name="tDocumentation" >
    <xs:sequence>
      <xs:any minOccurs="0" maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tDocumented" >
    <xs:annotation>
      <xs:documentation>
        This type is extended by component types to allow them to be documented
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="documentation" type="wsdl:tDocumentation" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tExtensibleAttributesDocumented" abstract="true" >
    <xs:complexContent>
      <xs:extension base="wsdl:tDocumented" >
        <xs:annotation>
          <xs:documentation>
            This type is extended by component types to allow attributes from other namespaces
            to be added.
          </xs:documentation>
        </xs:annotation>
        <xs:anyAttribute namespace="##other" processContents="lax" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tExtensibleDocumented" abstract="true" >
    <xs:complexContent>
      <xs:extension base="wsdl:tDocumented" >
        <xs:annotation>
          <xs:documentation>
            This type is extended by component types to allow elements from other namespaces
            to be added.
          </xs:documentation>
        </xs:annotation>
        <xs:sequence>
          <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
            processContents="lax" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:element name="definitions" type="wsdl:tDefinitions" >
  <xs:key name="message" >
    <xs:selector xpath="wsdl:message" />
    <xs:field xpath="@name" />
  </xs:key>
  <xs:key name="portType" >
    <xs:selector xpath="wsdl:portType" />
    <xs:field xpath="@name" />
  </xs:key>
  <xs:key name="binding" >
    <xs:selector xpath="wsdl:binding" />
    <xs:field xpath="@name" />
  </xs:key>
  <xs:key name="service" >
    <xs:selector xpath="wsdl:service" />
    <xs:field xpath="@name" />
  </xs:key>
  <xs:key name="import" >
    <xs:selector xpath="wsdl:import" />
    <xs:field xpath="@namespace" />
  </xs:key>
</xs:element>

<xs:group name="anyTopLevelOptionalElement" >
  <xs:annotation>
    <xs:documentation>
      Any top level optional element allowed to appear more then once - any child of
      definitions element except wsdl:types. Any extensibility element is allowed in
      any place.
    </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="import" type="wsdl:tImport" />
    <xs:element name="types" type="wsdl:tTypes" />
    <xs:element name="message" type="wsdl:tMessage" >
      <xs:unique name="part" >
        <xs:selector xpath="wsdl:part" />
        <xs:field xpath="@name" />
      </xs:unique>
    </xs:element>
    <xs:element name="portType" type="wsdl:tPortType" />
    <xs:element name="binding" type="wsdl:tBinding" />
    <xs:element name="service" type="wsdl:tService" >
      <xs:unique name="port" >
        <xs:selector xpath="wsdl:port" />
        <xs:field xpath="@name" />
      </xs:unique>
    </xs:element>
  </xs:choice>
</xs:group>

<xs:complexType name="tDefinitions" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleDocumented" >
```

```
      <xs:sequence>
        <xs:group ref="wsdl:anyTopLevelOptionalElement" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="targetNamespace" type="xs:anyURI" use="optional" />
      <xs:attribute name="name" type="xs:NCName" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tImport" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleAttributesDocumented" >
      <xs:attribute name="namespace" type="xs:anyURI" use="required" />
      <xs:attribute name="location" type="xs:anyURI" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tTypes" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleDocumented" />
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tMessage" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleDocumented" >
      <xs:sequence>
        <xs:element name="part" type="wsdl:tPart" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tPart" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleAttributesDocumented" >
      <xs:attribute name="name" type="xs:NCName" use="required" />
      <xs:attribute name="element" type="xs:QName" use="optional" />
      <xs:attribute name="type" type="xs:QName" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tPortType" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleAttributesDocumented" >
      <xs:sequence>
        <xs:element name="operation" type="wsdl:tOperation" minOccurs="0"
          maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required" />
```

```
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tOperation" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleDocumented" >
      <xs:sequence>
        <xs:choice>
          <xs:group ref="wsdl:request-response-or-one-way-operation" />
          <xs:group ref="wsdl:solicit-response-or-notification-operation" />
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required" />
      <xs:attribute name="parameterOrder" type="xs:NMTOKENS" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:group name="request-response-or-one-way-operation" >
  <xs:sequence>
    <xs:element name="input" type="wsdl:tParam" />
    <xs:sequence minOccurs='0' >
      <xs:element name="output" type="wsdl:tParam" />
      <xs:element name="fault" type="wsdl:tFault" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:sequence>
</xs:group>

<xs:group name="solicit-response-or-notification-operation" >
  <xs:sequence>
    <xs:element name="output" type="wsdl:tParam" />
    <xs:sequence minOccurs='0' >
      <xs:element name="input" type="wsdl:tParam" />
      <xs:element name="fault" type="wsdl:tFault" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:sequence>
</xs:group>

<xs:complexType name="tParam" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleAttributesDocumented" >
      <xs:attribute name="name" type="xs:NCName" use="optional" />
      <xs:attribute name="message" type="xs:QName" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="tFault" >
  <xs:complexContent>
    <xs:extension base="wsdl:tExtensibleAttributesDocumented" >
      <xs:attribute name="name" type="xs:NCName" use="required" />
      <xs:attribute name="message" type="xs:QName" use="required" />
    </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tBinding" >
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibleDocumented" >
        <xs:sequence>
          <xs:element name="operation" type="wsdl:tBindingOperation" minOccurs="0"
            maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="required" />
        <xs:attribute name="type" type="xs:QName" use="required" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tBindingOperationMessage" >
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibleDocumented" >
        <xs:attribute name="name" type="xs:NCName" use="optional" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tBindingOperationFault" >
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibleDocumented" >
        <xs:attribute name="name" type="xs:NCName" use="required" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tBindingOperation" >
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibleDocumented" >
        <xs:sequence>
          <xs:element name="input" type="wsdl:tBindingOperationMessage" minOccurs="0" />
          <xs:element name="output" type="wsdl:tBindingOperationMessage" minOccurs="0" />
          <xs:element name="fault" type="wsdl:tBindingOperationFault" minOccurs="0"
            maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="required" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tService" >
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibleDocumented" >
        <xs:sequence>
          <xs:element name="port" type="wsdl:tPort" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="required" />
      </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tPort" >
    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibleDocumented" >
        <xs:attribute name="name" type="xs:NCName" use="required" />
        <xs:attribute name="binding" type="xs:QName" use="required" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:attribute name="arrayType" type="xs:string" />
  <xs:attribute name="required" type="xs:boolean" />
  <xs:complexType name="tExtensibilityElement" abstract="true" >
    <xs:attribute ref="wsdl:required" use="optional" />
  </xs:complexType>

</xs:schema>
```

Quelltext 15: Partner-Links (BPEL4WS 1.1)

```
<?xml version='1.0' encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  targetNamespace="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  elementFormDefault="qualified">

  <element name="partnerLinkType" type="plnk:tPartnerLinkType"/>

  <complexType name="tPartnerLinkType">
    <sequence>
      <element name="role" type="plnk:tRole" minOccurs="1" maxOccurs="2"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>

  <complexType name="tRole">
    <sequence>
      <element name="portType" minOccurs="1" maxOccurs="1">
        <complexType>
          <attribute name="name" type="QName" use="required"/>
        </complexType>
      </element>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>
</schema>
```

Quelltext 16: BPDL/I (intern)

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:i="http://www.schlodder.de/schemas/ibpdl/i/"
  targetNamespace="http://www.schlodder.de/schemas/ibpdl/i/"
  elementFormDefault="qualified">

  <element name="definitions" type="i:tDefinitions"/>

  <complexType name="tDefinitions">
    <sequence>
      <choice>
        <element name="settings" type="i:tSettings" minOccurs="0"/>
        <element name="message" type="i:tMessage" minOccurs="0" maxOccurs="unbounded"/>
        <element name="portType" type="i:tPortType" minOccurs="0" maxOccurs="unbounded"/>
        <element name="partnerLinkType" type="i:tPartnerLinkType" minOccurs="0"
          maxOccurs="unbounded"/>
      </choice>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="version" type="unsignedShort" use="required"/>
    <attribute name="revision" type="unsignedShort" use="required"/>
    <attribute name="targetNamespace" type="anyURI" use="required"/>
    <attribute name="persistent" type="boolean" use="optional"/>
  </complexType>

  <complexType name="tSettings">
    <attribute name="name" type="anyURI" use="required"/>
  </complexType>

  <complexType name="tMessage">
    <sequence>
      <element name="part" type="i:tPart" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>

  <complexType name="tPart">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="type" type="QName" use="required"/>
  </complexType>

  <complexType name="tPortType">
    <sequence>
      <element name="operation" type="i:tOperation" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>

  <complexType name="tOperation">
    <sequence>
      <element name="input" type="i:tInput" minOccurs="0"/>
      <element name="output" type="i:tOutput" minOccurs="0"/>
```

```
    <element name="fault" type="i:tFault" minOccurs="0"/>
    <element name="exec" type="i:tExec" minOccurs="0"/>
    <element name="database" type="i:tDatabase" minOccurs="0"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
</complexType>

<complexType name="tInput">
  <attribute name="message" type="QName" use="required"/>
</complexType>

<complexType name="tOutput">
  <attribute name="message" type="QName" use="required"/>
</complexType>

<complexType name="tFault">
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="message" type="QName" use="required"/>
</complexType>

<complexType name="tExec">
  <attribute name="lang" type="NCName" use="required"/>
  <attribute name="object" type="NCName" use="optional"/>
  <attribute name="method" type="NCName" use="optional"/>
</complexType>

<complexType name="tDatabase">
  <sequence>
    <element name="select" type="i:tSelect" minOccurs="0"/>
    <element name="update" type="i:tUpdate" minOccurs="0"/>
    <element name="insert" type="i:tInsert" minOccurs="0"/>
    <element name="source" type="i:tSource" minOccurs="0"/>
    <element name="destination" type="i:tDestination" minOccurs="0"/>
  </sequence>
  <attribute name="name" type="QName" use="required"/>
  <attribute name="type" type="string" use="required"/>
</complexType>

<complexType name="tSelect">
  <attribute name="field" type="string" use="required"/>
  <attribute name="table" type="string" use="required"/>
  <attribute name="where" type="string" use="required"/>
  <attribute name="mode" type="string" use="optional"/>
</complexType>

<complexType name="tUpdate">
  <attribute name="field" type="string" use="required"/>
  <attribute name="table" type="string" use="required"/>
  <attribute name="where" type="string" use="required"/>
</complexType>

<complexType name="tInsert">
  <attribute name="table" type="string" use="required"/>
</complexType>
```

```
  <complexType name="tSource">
    <attribute name="part" type="NCName" use="required"/>
  </complexType>

  <complexType name="tDestination">
    <attribute name="part" type="NCName" use="required"/>
  </complexType>

  <complexType name="tPartnerLinkType">
    <sequence>
      <element name="role" type="i:tRole" maxOccurs="2"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>

  <complexType name="tRole">
    <sequence>
      <element name="portType" type="i:tPortTypePLT"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>

  <complexType name="tPortTypePLT">
    <attribute name="name" type="QName" use="required"/>
  </complexType>
</schema>
```

C.1.2 Workflowbeschreibungen

Aus den gleichen Gründen wie bei BDPL/I wurde auch für BPDL/F ein eigenes XML-Schema entworfen, das sowohl die unterstützten Elemente von BPEL4WS als auch die Erweiterungen der BPDL/F enthält.

Im folgenden findet sich zunächst die standardkonforme auf BPEL4WS 1.1 basierende Erweiterung durch BPDL/I (Quelltext 17), dann die Definition von BPEL4WS (Quelltext 18 ab Seite 120) und schließlich die vollständige, intern verwendete Version der BPDL/I-Definition (Quelltext 19 ab Seite 130).

Quelltext 17: BPDL/F

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:f="http://www.schlodder.de/schemas/bpdl/f/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  targetNamespace="http://www.schlodder.de/schemas/bpdl/f/"
  elementFormDefault="qualified">

  <import namespace="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    schemaLocation="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
```

```
10
    <attribute name="version" type="unsignedShort"/>
    <attribute name="revision" type="unsignedShort"/>

    <attribute name="timeout" type="duration"/>
15
    <attribute name="namespace" type="string"/>
    <attribute name="system" type="string"/>

    <element name="foreach" type="f:tForeach"/>
20
    <complexType name="tForeach">
      <complexContent>
        <extension base="bpws:tScope">
          <attribute name="loopVariable" type="NCName" use="required"/>
25        <attribute name="loopPart" type="NCName" use="required"/>
          <attribute name="fieldVariable" type="NCName" use="required"/>
          <attribute name="fieldPart" type="NCName" use="required"/>
        </extension>
      </complexContent>
30  </complexType>

    <element name="transaction" type="f:tTransaction"/>

    <complexType name="tTransaction">
35    <complexContent>
        <extension base="bpws:tScope">
          <attribute name="inputVariable" type="NCName" use="required"/>
          <attribute name="outputVariable" type="NCName" use="required"/>
          <attribute name="retryCount" type="int" use="optional"/>
40      </extension>
      </complexContent>
    </complexType>

    <element name="rollback" type="f:tRollback"/>
45
    <complexType name="tRollback">
      <complexContent>
        <extension base="bpws:Activity">
          <attribute name="name" type="NCName" use="prohibited"/>
50      </extension>
      </complexContent>
    </complexType>

    <element name="createSavepoint" type="f:tCreateSavepoint"/>
55
    <complexType name="tCreateSavepoint">
      <complexContent>
        <extension base="bpws:tActivity"/>
      </complexContent>
60  </complexType>

    <element name="restoreSavepoint" type="f:tRestoreSavepoint"/>
```

```
  <complexType name="tRestoreSavepoint">
    <complexContent>
      <extension base="bpws:tActivity"/>
    </complexContent>
  </complexType>
</schema>
```

Quelltext 18: BPEL4WS 1.1

```
<?xml version='1.0' encoding="UTF-8"?>
<!--

Legal Disclaimer

The presentation, distribution or other dissemination of the information
contained in this specification is not a license, either expressly or impliedly,
to any intellectual property owned or controlled by BEA or IBM or Microsoft and\or any other
third party. BEA and IBM and Microsoft and\or any other third party may have patents, patent
applications, trademarks, copyrights, or other intellectual property rights
covering subject matter in this document. The furnishing of this document
does not give you any license to BEA's and IBM's and Microsoft's or any other third party's
patents, trademarks, copyrights, or other intellectual property.

This specification and the information contained herein is provided on an "AS IS"
basis and to the maximum extent permitted by applicable law, BEA and IBM and Microsoft
provides the document AS IS AND WITH ALL FAULTS, and hereby disclaims all other warranties
and conditions, either express, implied or statutory, including, but not limited
to, any (if any) implied warranties, duties or conditions of merchantability, of
fitness for a particular purpose, of accuracy or completeness of responses, of
results, of workmanlike effort, of lack of viruses, and of lack of negligence,
all with regard to the document. ALSO, THERE IS NO WARRANTY OR CONDITION OF
TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR
NON-INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO THE DOCUMENT.

IN NO EVENT WILL BEA or IBM or MICROSOFT BE LIABLE TO ANY OTHER PARTY FOR THE COST OF
PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY
INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES WHETHER UNDER
CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY
OTHER AGREEMENT RELATING TO THIS DOCUMENT, WHETHER OR NOT SUCH PARTY HAD ADVANCE
NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

Copyright Notice

Copyright 2001, 2002 BEA Systems and IBM Corporation and Microsoft Corporation.
All rights reserved

-->

<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  targetNamespace="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  elementFormDefault="qualified">
```

```
45
    <import namespace="http://schemas.xmlsoap.org/wsdl/"
      schemaLocation="http://schemas.xmlsoap.org/wsdl/"/>

    <element name="property">
50    <complexType>
        <attribute name="name" type="NCName"/>
        <attribute name="type" type="QName"/>
      </complexType>
    </element>
55
    <element name="propertyAlias">
      <complexType>
        <attribute name="propertyName" type="QName" />
        <attribute name="messageType" type="QName"/>
60      <attribute name="part" type="NCName"/>
        <attribute name="select" type="string"/>
      </complexType>
    </element>

65  <complexType name="tExtensibleElements">
      <annotation>
        <documentation>
          This type is extended by other component types to allow elements and attributres from
          other namespaces to be added.
70      </documentation>
      </annotation>
      <sequence>
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded" processContents="lax"/>
      </sequence>
75    <anyAttribute namespace="##other" processContents="lax"/>
    </complexType>

    <element name="process" type="bpws:tProcess"/>
    <complexType name="tProcess">
80    <complexContent>
        <extension base="bpws:tExtensibleElements">
          <sequence>
            <element name="partners" type="bpws:tPartners" minOccurs="0"/>
            <element name="containers" type="bpws:tContainers" minOccurs="0"/>
85          <element name="correlationSets" type="bpws:tCorrelationSets" minOccurs="0"/>
            <element name="faultHandlers" type="bpws:tFaultHandlers" minOccurs="0"/>
            <element name="compensationHandler" type="bpws:tCompensationHandler" minOccurs="0"/>
            <group ref="bpws:activity"/>
          </sequence>
90        <attribute name="name" type="NCName"/>
          <attribute name="targetNamespace" type="anyURI"/>
          <attribute name="suppressJoinFailure" type="bpws:tBoolean" default="no"/>
          <attribute name="containerAccessSerializable" type="bpws:tBoolean" default="no"/>
          <attribute name="enableInstanceCompensation" type="bpws:tBoolean" default="no"/>
95        <attribute name="abstractProcess" type="bpws:tBoolean" default="no"/>
        </extension>
      </complexContent>
    </complexType>
```

```
<group name="activity">
  <choice>
    <element name="empty" type="bpws:tEmpty"/>
    <element name="invoke" type="bpws:tInvoke"/>
    <element name="receive" type="bpws:tReceive"/>
    <element name="reply" type="bpws:tReply"/>
    <element name="assign" type="bpws:tAssign"/>
    <element name="wait" type="bpws:tWait"/>
    <element name="throw" type="bpws:tThrow"/>
    <element name="terminate" type="bpws:tTerminate"/>
    <element name="flow" type="bpws:tFlow"/>
    <element name="switch" type="bpws:tSwitch"/>
    <element name="while" type="bpws:tWhile"/>
    <element name="sequence" type="bpws:tSequence"/>
    <element name="pick" type="bpws:tPick"/>
    <element name="scope" type="bpws:tScope"/>
  </choice>
</group>

<complexType name="tPartners">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element name="partner" type="bpws:tPartner" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tPartner">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="serviceLinkType" type="QName" use="required"/>
      <attribute name="myRole" type="NCName"/>
      <attribute name="partnerRole" type="NCName"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tFaultHandlers">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element name="catch" type="bpws:tCatch" minOccurs="0" maxOccurs="unbounded"/>
        <element name="catchAll" type="bpws:tActivityOrCompensateContainer" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCatch">
  <complexContent>
```

```
    <extension base="bpws:tActivityOrCompensateContainer">
      <attribute name="faultName" type="QName" use="optional"/>
      <attribute name="faultContainer" type="NCName" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tActivityContainer">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <group ref="bpws:activity"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="tActivityOrCompensateContainer">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <choice>
        <group ref="bpws:activity"/>
        <element name="compensate" type="bpws:tCompensate"/>
      </choice>
    </extension>
  </complexContent>
</complexType>

<complexType name="tOnMessage">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element name="correlations" type="bpws:tCorrelations" minOccurs="0"/>
        <group ref="bpws:activity"/>
      </sequence>
      <attribute name="partner" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="container" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tOnAlarm">
  <complexContent>
    <extension base="bpws:tActivityContainer">
      <attribute name="for" type="bpws:tDuration-expr" use="optional"/>
      <attribute name="until" type="bpws:tDeadline-expr" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCompensationHandler">
  <complexContent>
    <extension base="bpws:tActivityOrCompensateContainer"/>
```

```
    </complexContent>
  </complexType>

  <complexType name="tContainers">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <sequence>
          <element name="container" type="bpws:tContainer" maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tContainer">
    <!-- container does not allow extensibility elements because otherwise its content model
    would be non-deterministic -->
    <sequence>
      <element name="message" type="wsdl:tMessage"
        minOccurs="0">
        <unique name="part">
          <selector xpath="wsdl:part"/>
          <field xpath="@name"/>
        </unique>
      </element>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="messageType" type="QName" use="optional"/>
    <anyAttribute namespace="##other" processContents="lax"/>
  </complexType>

  <complexType name="tCorrelationSets">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <sequence>
          <element name="correlationSet" type="bpws:tCorrelationSet" maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tCorrelationSet">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <attribute name="properties" use="required">
          <simpleType>
            <list itemType="QName"/>
          </simpleType>
        </attribute>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tActivity">
```

```
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <sequence>
          <element name="target" type="bpws:tTarget" minOccurs="0" maxOccurs="unbounded"/>
          <element name="source" type="bpws:tSource" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="optional"/>
        <attribute name="joinCondition" type="bpws:tBoolean-expr" use="optional"/>
        <attribute name="suppressJoinFailure" type="bpws:tBoolean" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tSource">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <attribute name="linkName" type="NCName" use="required"/>
        <attribute name="transitionCondition" type="bpws:tBoolean-expr" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tTarget">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <attribute name="linkName" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tEmpty">
    <complexContent>
      <extension base="bpws:tActivity"/>
    </complexContent>
  </complexType>

  <complexType name="tCorrelations">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <sequence>
          <element name="correlation" type="bpws:tCorrelation" minOccurs="1"
            maxOccurs="unbounded" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tCorrelation">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <attribute name="set" type="NCName" use="required"/>
        <attribute name="initiation" type="bpws:tBoolean" use="optional" default="no"/>
        <attribute name="pattern" use="optional">
          <simpleType>
```

```
          <restriction base="string">
            <enumeration value="in" />
            <enumeration value="out" />
            <enumeration value="out-in" />
          </restriction>
        </simpleType>
      </attribute>
    </extension>
  </complexContent>
</complexType>

<complexType name="tInvoke">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="correlations" type="bpws:tCorrelations" minOccurs="0" maxOccurs="1"/>
        <element name="catch" type="bpws:tCatch" minOccurs="0" maxOccurs="unbounded"/>
        <element name="catchAll" type="bpws:tActivityOrCompensateContainer" minOccurs="0"/>
        <element name="compensationHandler" type="bpws:tCompensationHandler" minOccurs="0"/>
      </sequence>
      <attribute name="partner" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="inputContainer" type="NCName" use="required"/>
      <attribute name="outputContainer" type="NCName" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tReceive">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="correlations" type="bpws:tCorrelations" minOccurs="0"/>
      </sequence>
      <attribute name="partner" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="container" type="NCName" use="required"/>
      <attribute name="createInstance" type="bpws:tBoolean" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tReply">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="correlations" type="bpws:tCorrelations" minOccurs="0"/>
      </sequence>
      <attribute name="partner" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="container" type="NCName" use="required"/>
```

```
      <attribute name="faultName" type="QName" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tAssign">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="copy" type="bpws:tCopy" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCopy">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element ref="bpws:from"/>
        <element ref="bpws:to"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<element name="from" type="bpws:tFrom"/>
<complexType name="tFrom">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <attribute name="container" type="NCName"/>
      <attribute name="part" type="NCName"/>
      <attribute name="select" type="string"/>
      <attribute name="property" type="QName"/>
      <attribute name="partner" type="NCName"/>
      <attribute name="expression" type="string"/>
      <attribute name="opaque" type="bpws:tBoolean"/>
    </extension>
  </complexContent>
</complexType>

<element name="to">
  <complexType>
    <complexContent>
      <restriction base="bpws:tFrom">
        <attribute name="expression" type="string" use="prohibited"/>
        <attribute name="opaque" type="bpws:tBoolean" use="prohibited"/>
      </restriction>
    </complexContent>
  </complexType>
</element>

<complexType name="tWait">
  <complexContent>
```

```
    <extension base="bpws:tActivity">
      <attribute name="for" type="bpws:tDuration-expr" use="optional"/>
      <attribute name="until" type="bpws:tDeadline-expr" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tThrow">
  <complexContent>
    <extension base="bpws:tActivity">
      <attribute name="faultName" type="QName" use="required"/>
      <attribute name="faultContainer" type="NCName"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCompensate">
  <complexContent>
    <extension base="bpws:tActivity">
      <attribute name="scope" type="NCName" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tTerminate">
  <complexContent>
    <extension base="bpws:tActivity"/>
  </complexContent>
</complexType>

<complexType name="tFlow">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="links" type="bpws:tLinks" minOccurs="0"/>
        <group ref="bpws:activity" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tLinks">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element name="link" type="bpws:tLink" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tLink">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
```

```
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tSwitch">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="case" maxOccurs="unbounded">
          <complexType>
            <complexContent>
              <extension base="bpws:tActivityContainer">
                <attribute name="condition" type="bpws:tBoolean-expr" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
        <element name="otherwise" type="bpws:tActivityContainer" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tWhile">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <group ref="bpws:activity"/>
      </sequence>
      <attribute name="condition" type="bpws:tBoolean-expr" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tSequence">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <group ref="bpws:activity" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tPick">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="onMessage" type="bpws:tOnMessage" maxOccurs="unbounded"></element>
        <element name="onAlarm" type="bpws:tOnAlarm" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="createInstance" type="bpws:tBoolean" use="optional"/>
    </extension>
```

```
    </complexContent>
  </complexType>

  <complexType name="tScope">
    <complexContent>
      <extension base="bpws:tActivity">
        <sequence>
          <element name="faultHandlers" type="bpws:tFaultHandlers" minOccurs="0"/>
          <element name="compensationHandler" type="bpws:tCompensationHandler" minOccurs="0"/>
          <group ref="bpws:activity"/>
        </sequence>
        <attribute name="containerAccessSerializable" type="bpws:tBoolean" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <simpleType name="tListOfNCNames">
    <list itemType="NCName"/>
  </simpleType>

  <simpleType name="tBoolean-expr">
    <restriction base="string"/>
  </simpleType>

  <simpleType name="tDuration-expr">
    <restriction base="string"/>
  </simpleType>

  <simpleType name="tDeadline-expr">
    <restriction base="string"/>
  </simpleType>

  <simpleType name="tBoolean">
    <restriction base="string">
      <enumeration value="yes"/>
      <enumeration value="no"/>
    </restriction>
  </simpleType>
</schema>
```

Quelltext 19: BPDL/F (intern)

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:f="http://www.schlodder.de/schemas/ibpdl/f/"
  targetNamespace="http://www.schlodder.de/schemas/ibpdl/f/"
  elementFormDefault="qualified">

  <element name="process" type="f:tProcess"/>

  <complexType name="tProcess">
    <sequence>
      <element name="partnerLinks" type="f:tPartnerLinks"/>
```

```
    <element name="partners" type="f:tPartners"/>
    <element name="variables" type="f:tVariables"/>
    <element name="faultHandlers" type="f:tFaultHandlers" minOccurs="0"/>
    <element name="sequence" type="f:tSequence"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="version" type="unsignedShort" use="required"/>
  <attribute name="revision" type="unsignedShort" use="required"/>
  <attribute name="targetNamespace" type="anyURI" use="required"/>
</complexType>

<complexType name="tPartnerLinks">
  <sequence>
    <element name="partnerLink" type="f:tPartnerLink" minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="tPartnerLink">
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="partnerLinkType" type="QName" use="required"/>
  <attribute name="myRole" type="NCName" use="optional"/>
  <attribute name="partnerRole" type="NCName" use="optional"/>
</complexType>

<complexType name="tPartners">
  <sequence>
    <element name="partner" type="f:tPartner" minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="tPartner">
  <sequence>
    <element name="partnerLink" type="f:tPartnerLinkP" minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
</complexType>

<complexType name="tPartnerLinkP">
  <attribute name="name" type="NCName" use="required"/>
</complexType>

<complexType name="tVariables">
  <sequence>
    <element name="variable" type="f:tVariable" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="tVariable">
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="messageType" type="QName" use="optional"/>
  <attribute name="type" type="QName" use="optional"/>
</complexType>

<complexType name="tFaultHandlers">
```

```
    <sequence>
      <element name="catch" type="f:tCatch" minOccurs="0" maxOccurs="unbounded"/>
      <element name="catchAll" type="f:tCatchAll" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="tCatch">
    <sequence>
      <group ref="f:activity" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="faultName" type="QName"/>
    <attribute name="faultVariable" type="NCName"/>
  </complexType>

  <complexType name="tCatchAll">
    <sequence>
      <group ref="f:activity" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <complexType name="tCompensationHandler">
    <sequence>
      <group ref="f:activity" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <group name="activity">
    <choice>
      <element name="empty" type="f:tEmpty"/>
      <element name="sequence" type="f:tSequence"/>
      <element name="receive" type="f:tReceive"/>
      <element name="reply" type="f:tReply"/>
      <element name="invoke" type="f:tInvoke"/>
      <element name="assign" type="f:tAssign"/>
      <element name="throw" type="f:tThrow"/>
      <element name="flow" type="f:tFlow"/>
      <element name="switch" type="f:tSwitch"/>
      <element name="while" type="f:tWhile"/>
      <element name="foreach" type="f:tForeach"/>
      <element name="transaction" type="f:tTransaction"/>
      <element name="rollback" type="f:tRollback"/>
      <element name="createSavepoint" type="f:tCreateSavepoint"/>
      <element name="restoreSavepoint" type="f:tRestoreSavepoint"/>
    </choice>
  </group>

  <complexType name="tActivity">
    <sequence>
      <element name="target" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <attribute name="linkName" type="NCName" use="required"/>
        </complexType>
      </element>
      <element name="source" minOccurs="0" maxOccurs="unbounded">
```

```
      <complexType>
        <attribute name="linkName" type="NCName" use="required"/>
      </complexType>
    </element>
  </sequence>
  <attribute name="name" type="NCName" use="optional"/>
</complexType>

<complexType name="tEmpty">
  <complexContent>
    <extension base="f:tActivity">
      <attribute name="name" type="NCName" use="prohibited"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tSequence">
  <complexContent>
    <extension base="f:tActivity">
      <sequence>
        <group ref="f:activity" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="prohibited"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tReceive">
  <complexContent>
    <extension base="f:tActivity">
      <attribute name="name" type="NCName" use="prohibited"/>
      <attribute name="partnerLink" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="variable" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tReply">
  <complexContent>
    <extension base="f:tActivity">
      <attribute name="name" type="NCName" use="prohibited"/>
      <attribute name="partnerLink" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="variable" type="NCName" use="optional"/>
      <attribute name="faultName" type="QName"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tInvoke">
  <complexContent>
```

```
    <extension base="f:tActivity">
      <sequence>
        <element name="catch" type="f:tCatch" minOccurs="0" maxOccurs="unbounded"/>
        <element name="catchAll" type="f:tCatchAll" minOccurs="0"/>
      </sequence>
      <attribute name="name" type="NCName" use="prohibited"/>
      <attribute name="partnerLink" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="inputVariable" type="NCName" use="optional"/>
      <attribute name="outputVariable" type="NCName" use="optional"/>
      <attribute name="timeout" type="duration" use="optional" />
    </extension>
  </complexContent>
</complexType>

<complexType name="tAssign">
  <complexContent>
    <extension base="f:tActivity">
      <sequence>
        <element name="copy" type="f:tCopy" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="prohibited"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCopy">
  <sequence>
    <element name="from" type="f:tFrom"/>
    <element name="to" type="f:tTo"/>
  </sequence>
</complexType>

<complexType name="tFrom">
  <simpleContent>
    <extension base="string">
      <attribute name="variable" type="NCName" use="optional"/>
      <attribute name="part" type="NCName" use="optional"/>
      <attribute name="expression" type="string" use="optional"/>
      <attribute name="namespace" type="string" use="optional"/>
      <attribute name="system" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>

<complexType name="tTo">
  <attribute name="variable" type="NCName" use="optional"/>
  <attribute name="part" type="NCName" use="optional"/>
</complexType>

<complexType name="tThrow">
  <complexContent>
    <extension base="f:tActivity">
```

```
      <attribute name="name" type="NCName" use="prohibited"/>
      <attribute name="faultName" type="QName" use="required"/>
      <attribute name="faultVariable" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tFlow">
  <complexContent>
    <extension base="f:tActivity">
      <sequence>
        <element name="links" type="f:tLinks" minOccurs="0"/>
        <group ref="f:activity" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tLinks">
  <sequence>
    <element name="link" type="f:tLink" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="tLink">
  <attribute name="name" type="NCName" use="required"/>
</complexType>

<complexType name="tSwitch">
  <complexContent>
    <extension base="f:tActivity">
      <sequence>
        <element name="case" type="f:tCase" minOccurs="0" maxOccurs="unbounded"/>
        <element name="otherwise" type="f:tOtherwise" minOccurs="0"/>
      </sequence>
      <attribute name="name" type="NCName" use="prohibited"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCase">
  <sequence>
    <group ref="f:activity" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="condition" type="string" use="required"/>
</complexType>

<complexType name="tOtherwise">
  <sequence>
    <group ref="f:activity" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="tWhile">
```

```
    <complexContent>
      <extension base="f:tActivity">
        <sequence>
          <group ref="f:activity" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="prohibited"/>
        <attribute name="condition" type="string" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tForeach">
    <complexContent>
      <extension base="f:tActivity">
        <sequence>
          <group ref="f:activity" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="loopVariable" type="NCName" use="required"/>
        <attribute name="loopPart" type="NCName" use="required"/>
        <attribute name="fieldVariable" type="NCName" use="required"/>
        <attribute name="fieldPart" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tTransaction">
    <complexContent>
      <extension base="f:tActivity">
        <sequence>
          <element name="variables" type="f:tVariables" minOccurs="0"/>
          <element name="faultHandlers" type="f:tFaultHandlers" minOccurs="0"/>
          <element name="compensationHandler" type="f:tCompensationHandler" minOccurs="0"/>
          <group ref="f:activity" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="inputVariable" type="NCName" use="required"/>
        <attribute name="outputVariable" type="NCName" use="required"/>
        <attribute name="retryCount" type="int" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRollback">
    <complexContent>
      <extension base="f:tActivity">
        <attribute name="name" type="NCName" use="prohibited"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tCreateSavepoint">
    <complexContent>
      <extension base="f:tActivity"/>
    </complexContent>
  </complexType>
```

```
  <complexType name="tRestoreSavepoint">
    <complexContent>
      <extension base="f:tActivity"/>
    </complexContent>
  </complexType>
</schema>
```

C.1.3 Datenbankbeschreibungen

Datenbankbeschreibungen basieren auf einem eigens entworfenen XML-Schema, das eine einfache, klare Definition von Tabellen für relationale Datenbanken erlaubt.

Quelltext 20: BPDL/D

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:d="http://www.schlodder.de/schemas/bpdl/d/"
  targetNamespace="http://www.schlodder.de/schemas/bpdl/d/"
  elementFormDefault="qualified">

  <element name="database" type="d:tDatabase"/>

  <complexType name="tDatabase">
    <sequence>
      <element name="table" type="d:tTable" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="version" type="unsignedShort" use="required"/>
    <attribute name="revision" type="unsignedShort" use="required"/>
    <attribute name="targetNamespace" type="anyURI" use="required" />
  </complexType>

  <complexType name="tTable">
    <sequence>
      <element name="field" type="d:tField" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
  </complexType>

  <complexType name="tField">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="type" type="QName" use="required"/>
  </complexType>
</schema>
```

C.1.4 Parameterbeschreibungen

Die ebenfalls auf einem eigenen XML-Schema beruhenden Parameterbeschreibungen erlauben neben der einfachen Angabe von Name, Wert und Typ eines Parameters auch die Verwendung von Variablen, die hier `property` genannt werden.

Quelltext 21: BPDL/S

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:s="http://www.schlodder.de/schemas/bpdl/s/"
  targetNamespace="http://www.schlodder.de/schemas/bpdl/s/"
  elementFormDefault="qualified">

  <element name="settings" type="s:tSettings"/>

  <complexType name="tSettings">
    <sequence>
      <choice>
        <element name="property" type="s:tProperty" minOccurs="0" maxOccurs="unbounded"/>
        <element name="option" type="s:tOption" minOccurs="0" maxOccurs="unbounded"/>
      </choice>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="version" type="unsignedShort" use="required"/>
    <attribute name="revision" type="unsignedShort" use="required"/>
    <attribute name="targetNamespace" type="anyURI" use="required" />
  </complexType>

  <complexType name="tProperty">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="value" type="string" use="required"/>
  </complexType>

  <complexType name="tOption">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="value" type="string" use="optional"/>
    <attribute name="type" type="QName" use="optional"/>
  </complexType>
</schema>
```

C.2 Beispieldateien

In diesem Abschnitt finden sich Beispieldateien für jeden Subset von BPDL, die zusammengenommen einen einfachen Geschäftsprozeß für die Ausführung auf der Transaktionsmaschine definieren. Dazu gehört auch die Konfiguration der Komponenten der TE, die durch die Beispiele in Unterabschnitt C.2.4 ab Seite 154 angegeben ist.

C.2.1 Schnittstellenbeschreibungen

In diesem Unterabschnitt finden sich Beispiele für Schnittstellenbeschreibungen einer technischen Aktivität (Quelltext 22), eines System-Plugins (Quelltext 23 auf Seite 140), einer Datenbank (Quelltext 24 ab Seite 140), eines Subworkflow (Quelltext 25 ab Seite 142) und eines Masterflow (Quelltext 26 auf Seite 144).

Quelltext 22: Activity.ibpdl

```
<definitions name="Debit" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/Debit_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- Ein- und Ausgabedaten -->

  <message name="debitInMsg">
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="amount" type="xsd:decimal"/>
    <part name="balance" type="xsd:decimal"/>
  </message>

  <message name="debitOutMsg">
    <part name="balance" type="xsd:decimal"/>
  </message>

  <message name="debitFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- Interface -->

  <portType name="debitPT">
    <operation name="debit">
      <input message="debitInMsg"/>
      <output message="debitOutMsg"/>

      <!-- Dies ist die Exception, die im Fehlerfall geworfen wird -->
      <fault name="debitError" message="debitFaultMsg"/>
      <bpdl:exec lang="Java" class="CreditCardHandling" method="debit"/>
    </operation>
  </portType>
```

```
  <plnk:partnerLinkType name="debitLT">
    <plnk:role name="debit">
      <plnk:portType name="debitPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 23: Plugin.ibpdl

```
<definitions name="Monitor" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/Monitor_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- Ein- und Ausgabedaten -->

  <message name="monitorInMsg">
    <part name="message" type="xsd:string"/>
    <part name="transaction" type="xsd:string"/>
    <part name="time" type="xsd:string"/>
  </message>

  <!-- Interface -->

  <portType name="monitorPT">
    <operation name="monitor">
      <input message="monitorInMsg"/>
      <bpdl:exec lang="COBOL" object="Monitor"/>
    </operation>
  </portType>

  <plnk:partnerLinkType name="monitorLT">
    <plnk:role name="monitor">
      <plnk:portType name="monitorPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 24: Database.ibpdl

```
<definitions name="Database" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/Database_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:db="http://example.com/bpdl/database/DB_V1"
  xmlns:et="http://www.schlodder.de/schemas/bpdl/extended_types/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- Ein- und Ausgabedaten fuer SELECT BALANCE FROM CREDITCARDS -->

  <message name="dbReadCCBalanceRequestMsg">
    <part name="creditCardNumber" type="xsd:string"/>
```

```
  </message>

  <message name="dbReadCCBalanceResponseMsg">
    <part name="balance" type="xsd:decimal"/>
  </message>

  <message name="dbReadCCBalanceFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- Ein- und Ausgabedaten fuer UPDATE BALANCE FROM CREDITCARDS -->

  <message name="dbWriteCCBalanceRequestMsg">
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="balance" type="xsd:decimal"/>
  </message>

  <message name="dbWriteCCBalanceFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- Ein- und Ausgabedaten fuer INSERT INTO CREDITCARDS -->

  <message name="dbWriteCCSetRequestMsg">
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="ownerName" type="xsd:string"/>
    <part name="balance" type="xsd:decimal"/>
  </message>

  <message name="dbWriteCCSetFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- Ein- und Ausgabedaten fuer SELECT ALL BALANCES FROM CREDITCARDS -->

  <message name="dbFindCardsRequestMsg">
    <part name="ownerName" type="xsd:string"/>
  </message>

  <message name="dbFindCardsResponseMsg">
    <part name="balances" type="et:listOfDecimal"/>
  </message>

  <message name="dbFindCardsFaultMsg">
    <part name="result" type="xsd:string"/>
  </message>

  <!-- Interface -->

  <portType name="dbAccessPT">
    <operation name="readCCBalance">
      <input message="dbReadCCBalanceRequestMsg"/>
      <output message="dbReadCCBalanceResponseMsg"/>
      <fault name="dbCCReadBalanceError" message="dbReadCCBalanceFaultMsg"/>
```

```
      <bpdl:database name="db:DB" type="DB2">
        <bpdl:select field="BALANCE" table="CREDITCARDS"
          where="CCNUM='${creditCardNumber}'" mode="update"/>
        <bpdl:destination part="balance"/>
      </bpdl:database>
    </operation>
    <operation name="writeCCBalance">
      <input message="dbWriteCCBalanceRequestMsg"/>
      <fault name="dbWriteCCBalanceError" message="dbWriteCCBalanceFaultMsg"/>
      <bpdl:database name="db:DB" type="DB2">
        <bpdl:update field="BALANCE" table="CREDITCARDS"
          where="CCNUM='${creditCardNumber}'"/>
        <bpdl:source part="balance"/>
      </bpdl:database>
    </operation>
    <operation name="writeCCSet">
      <input message="dbWriteCCSetRequestMsg"/>
      <fault name="dbWriteCCBalanceError" message="dbWriteCCSetFaultMsg"/>
      <bpdl:database name="db:DB" type="DB2">
        <bpdl:insert table="CREDITCARDS"/>
      </bpdl:database>
    </operation>
    <operation name="findCards">
      <input message="dbFindCardsRequestMsg"/>
      <output message="dbFindCardsResponseMsg"/>
      <fault name="dbFindCardsError" message="dbFindCardsFaultMsg"/>
      <bpdl:database name="db:DB" type="DB2">
        <bpdl:select field="BALANCE" table="CREDITCARDS"
          where="OWNER='${ownerName}'" mode="fetch"/>
        <bpdl:destination part="balances"/>
      </bpdl:database>
    </operation>
  </portType>

  <plnk:partnerLinkType name="dbLT">
    <plnk:role name="dbAccess">
      <plnk:portType name="dbAccessPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 25: Subworkflow.ibpdl

```
<definitions name="CreditCardTransaction" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/CreditCardTransaction_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- Subworkflow Settings -->

  <bpdl:settings name="http://example.com/bpdl/parameters/creditCardTransaction_V1"/>
```

```
  <!-- Ein- und Ausgabedaten -->

  <message name="creditCardTransactionInputSRWU">
    <part name="creditCardNumber" type="xsd:string" />
    <part name="amount" type="xsd:decimal" />
    <part name="destinationAccount" type="xsd:string" />
  </message>

  <message name="creditCardTransactionTransientSRWU">
    <part name="balance" type="xsd:decimal" />
    <part name="amount" type="xsd:decimal" />
    <part name="destinationAccount" type="xsd:string" />
  </message>

  <message name="creditCardTransactionOutputSRU">
    <part name="balance" type="xsd:decimal" />
    <part name="result" type="xsd:string" />
  </message>

  <message name="creditCardTransactionErrorSRU">
    <part name="result" type="xsd:string" />
  </message>

  <message name="creditCardTransactionMessageSRU">
    <part name="type" type="xsd:int" />
    <part name="message" type="xsd:string" />
  </message>

  <!-- Interface -->

  <portType name="creditCardTransactionPT">
    <operation name="creditCardTransaction">
      <input message="creditCardTransactionInputSRWU" />
      <output message="creditCardTransactionOutputSRU" />
      <fault name="ccTransError" message="creditCardTransactionErrorSRU" />
    </operation>
  </portType>

  <portType name="creditCardTransactionMessagePT">
    <operation name="sendMessage">
      <input message="creditCardTransactionMessageSRU" />
    </operation>
  </portType>

  <plnk:partnerLinkType name="creditCardTransactionLT">
    <plnk:role name="creditCardTransaction">
      <plnk:portType name="creditCardTransactionPT" />
    </plnk:role>
    <plnk:role name="creditCardTransactionMessage">
      <plnk:portType name="creditCardTransactionMessagePT" />
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Quelltext 26: Masterflow.ibpdl

```
<definitions name="InternetPayment" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/interface/InternetPayment_V1"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/i/">

  <!-- Masterflow Settings -->

  <bpdl:settings name="http://example.com/bpdl/parameters/InternetPayment_V1"/>

  <!-- Ein- und Ausgabedaten -->

  <message name="internetPaymentInputSRWU">
    <part name="article" type="xsd:string"/>
    <part name="payment" type="xsd:string"/>
    <part name="amount" type="xsd:decimal"/>
    <part name="shippingaddress" type="xsd:string"/>
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="destinationAccount" type="xsd:string"/>
  </message>

  <message name="internetPaymentOutputSRU">
    <part name="result" type="xsd:string"/>
  </message>

  <message name="subworkflowMessageSRWU">
    <part name="message" type="xsd:string"/>
  </message>

  <!-- Interface -->

  <portType name="internetPaymentPT">
    <operation name="internetPayment">
      <input message="internetPaymentInputSRWU"/>
      <output message="internetPaymentOutputSRU"/>
    </operation>
  </portType>

  <plnk:partnerLinkType name="internetPaymentLT">
    <plnk:role name="internetPayment">
      <plnk:portType name="internetPaymentPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

C.2.2 Workflowbeschreibungen

Hier ist je ein Beispiel für einen Subworkflow (Quelltext 27) und einen Masterflow (Quelltext 28 ab Seite 151) zu finden.

Quelltext 27: Subworkflow.fbpdl

```
<process name="creditCardTransaction" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/workflow/CreditCardTransaction_V1"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/f/"

  <!-- Interface der Debit Aktivit"at -->
  xmlns:di="http://example.com/bpdl/interface/Debit_V1"

  <!-- Interface der CheckBalance Aktivit"at -->
  xmlns:ci="http://example.com/bpdl/interface/CheckBalance_V1"

  <!-- Interface des Monitor Plugin -->
  xmlns:mi="http://example.com/bpdl/interface/Monitor_V1"

  <!-- Interface der Databank -->
  xmlns:dbi="http://example.com/bpdl/interface/Database_V1"

  <!-- Interface des Subworkflow -->
  xmlns:ccti="http://example.com/bpdl/interface/CreditCardTransaction_V1">

  <partnerLinks>

    <!-- PartnerLink f"ur den Subworkflow -->

    <partnerLink name="SWF"
      partnerLinkType="ccti:creditCardTransactionLT"
      myRole="creditCardTransaction"
      partnerRole="creditCardTransactionMessage"/>

    <!-- PartnerLink f"ur Datenbankzugriff -->

    <partnerLink name="db"
      partnerLinkType="dbi:dbLT"
      partnerRole="dbAccess"/>

    <!-- PartnerLink f"ur eine technische Aktivit"at -->

    <partnerLink name="debit"
      partnerLinkType="di:debitLT"
      partnerRole="debit"/>

    <!-- PartnerLink f"ur eine weitere technische Aktivit"at -->

    <partnerLink name="checkBalance"
      partnerLinkType="ci:checkBalanceLT"
      partnerRole="checkBalance"/>
```

```
    <!-- PartnerLink f"ur ein Plugin -->

    <partnerLink name="monitor"
      partnerLinkType="mi:monitorLT"
      partnerRole="monitor"/>
  </partnerLinks>

  <!-- In partners wird das externe Interface des SWF festgelegt -->

  <partners>
    <partner name="CreditCardTransaction">
      <partnerLink name="SWF"/>
    </partner>
  </partners>

  <variables>

    <!-- Variablen f"ur die Transaktionen des Subworkflow -->

    <variable name="inSRWU" messageType="ccti:creditCardTransactionInputSRWU"/>
    <variable name="transSRWU" messageType="ccti:creditCardTransactionTransientSRWU"/>
    <variable name="outSRWU" messageType="ccti:creditCardTransactionOutputSRWU"/>
    <variable name="errSRWU" messageType="ccti:creditCardTransactionErrorSRWU"/>
    <variable name="msgSRWU" messageType="ccti:creditCardTransactionMessageSRWU"/>
  </variables>

  <sequence>

    <!-- Eingabe-SRWU empfangen -->

    <receive partnerLink="SWF" portType="ccti:creditCardTransactionPT"
      operation="creditCardTransaction" variable="inSRWU"/>

    <!-- Erste Transaktion -->

    <bpdl:transaction inputVariable="inSRWU" outputVariable="transSRWU">
      <variables>

        <!-- Variablen f"ur die technische Aktivit"at -->

        <variable name="debitIn" messageType="di:debitInMsg"/>
        <variable name="debitOut" messageType="di:debitOutMsg"/>
        <variable name="debitErr" messageType="di:debitFaultMsg"/>

        <!-- Variablen f"ur das Plugin -->

        <variable name="monitorIn" messageType="mi:monitorInMsg"/>

        <!-- Variablen f"ur die Datenbankzugriffe -->

        <variable name="dbReadCCParam" messageType="dbi:dbReadCCBalanceRequestMsg"/>
        <variable name="dbReadCCRes" messageType="dbi:dbReadCCBalanceResponseMsg"/>
        <variable name="dbReadCCErr" messageType="dbi:dbReadCCBalanceFaultMsg"/>
        <variable name="dbWriteCCParam" messageType="dbi:dbWriteCCBalanceRequestMsg"/>
```

```
      <variable name="dbWriteCCErr" messageType="dbi:dbWriteCCBalanceFaultMsg"/>
    </variables>

    <!-- Kompensations-Workflow -->

    <compensationHandler>
      <sequence>

        <!-- Hier werden alle "Anderungen an Datenbanken r"uckg"angig gemacht, die
             w"ahrend der Transaktion durchgef"uhrt wurden -->

        <empty/>
      </sequence>
    </compensationHandler>

    <sequence>

      <!-- Feld aus Datenbank lesen -->

      <assign>
        <copy>
          <from variable="inSRWU" part="creditCardNumber"/>
          <to variable="dbcReadParam" part="creditCardNumber"/>
        </copy>
      </assign>

      <invoke partnerLink="db" portType="dbi:dbAccessPT"
        operation="readCCBalance" inputVariable="dbReadCCParam"
        outputVariable="dbReadCCRes" bpdl:timeout="2s">
        <catch faultName="dbi:dbReadCCBalanceError" faultVariable="dbReadCCErr">

          <!-- Transaktion zuruecksetzen -->

          <bpdl:rollback/>
        </catch>
      </invoke>

      <!-- Savepoint setzen -->

      <bpdl:createSavepoint name="savepoint1"/>

      <!-- technische Aktivit"at aufrufen -->

      <assign>
        <copy>
          <from variable="inSRWU" part="creditCardNumber"/>
          <to variable="debitIn" part="creditCardNumber"/>
        </copy>
        <copy>
          <from variable="inSRWU" part="amount"/>
          <to variable="debitIn" part="amount"/>
        </copy>
        <copy>
          <from variable="dbReadCCRes" part="balance"/>
```

```
          <to variable="debitIn" part="balance"/>
        </copy>
      </assign>

      <invoke partnerLink="debit" portType="di:debitPT"
        operation="debit" inputVariable="debitIn"
        outputVariable="debitOut">
        <catch faultName="di:debitError" faultVariable="debitErr">

          <!-- Savepoint laden -->

          <bpdl:restoreSavepoint name="savepoint1"/>
        </catch>
      </invoke>

      <!-- Bonus geben -->

      <switch>
        <case condition="getVariableData('debitOut', 'balance') &gt; '10000' &amp;&amp;
          getVariableData('inSRWU', 'amount') &gt; '1000'">
          <assign>
            <copy>
              <from expression="getVariableData('debitOut', 'balance') + '5'"/>
              <to variable="debitOut" part="balance"/>
            </copy>
          </assign>
        </case>
      </switch>

      <!-- Savepoint setzen -->

      <bpdl:createSavepoint name="savepoint2"/>

      <!-- ge"andertes Feld in die Datenbank zur"uckschreiben -->

      <assign>
        <copy>
          <from variable="inSRWU" part="creditCardNumber"/>
          <to variable="dbWriteCCParam" part="creditCardNumber"/>
        </copy>
        <copy>
          <from variable="debitOut" part="balance"/>
          <to variable="dbWriteCCParam" part="balance"/>
        </copy>
      </assign>

      <invoke partnerLink="balance" portType="dbi:dbAccessPT"
        operation="writeCCBalance" inputVariable="dbWriteCCParam">
        <catch faultName="dbi:dbWriteCCBalanceError" faultVariable="dbWriteCCErr">

          <!-- Savepoint laden -->

          <bpdl:restoreSavepoint name="savepoint2"/>
        </catch>
```

```
      </invoke>

      <!-- Plugin aufrufen -->

      <assign>
        <copy>
          <from>"debit completed"</from>
          <to variable="monitorIn" part="message"/>
        </copy>
        <copy>
          <from bpdl:namespace="transaction"/>
          <to variable="monitorIn" part="transaction"/>
        </copy>
        <copy>
          <from bpdl:system="time"/>
          <to variable="monitorIn" part="time"/>
        </copy>
      </assign>

      <invoke partnerLink="monitor" portType="mi:monitorPT"
        operation="monitor" inputVariable="monitorIn"/>

      <!-- Nachricht an MFC senden -->

      <assign>
        <copy>
          <from>"debit completed"</from>
          <to variable="msgSRWU" part="message"/>
        </copy>
      </assign>

      <invoke partnerLink="SWF" portType="ccti:creditCardTransactionMessagePT"
        operation="sendMessage" inputVariable="msgSRWU"/>

      <!-- Transaktion beenden -->

      <assign>
        <copy>
          <from variable="inSRWU" part="creditCardNumber"/>
          <to variable="transSRWU" part="creditCardNumber"/>
        </copy>
        <copy>
          <from variable="inSRWU" part="amount"/>
          <to variable="transSRWU" part="amount"/>
        </copy>
        <copy>
          <from variable="dbReadARes" part="balance"/>
          <to variable="transSRWU" part="balance"/>
        </copy>
      </assign>
    </sequence>
  </bpdl:transaction>

  <!-- Zweite Transaktion -->
```

```
<bpdl:transaction inputVariable="transSRWU" outputVariable="outSRWU">
  <variables>

    <!-- Variablen f"ur die technische Aktivit"at -->

    <variable name="checkBalanceIn" messageType="ci:checkBalanceInMsg"/>
    <variable name="checkBalanceOut" messageType="ci:checkBalanceOutMsg"/>
    <variable name="checkBalanceErr" messageType="ci:checkBalanceFaultMsg"/>

    <!-- Variablen f"ur die Datenbank -->

    <variable name="dbCardsParam" messageType="dbi:dbFindCardsRequestMsg"/>
    <variable name="dbCardsRes" messageType="dbi:dbFindCardsResponseMsg"/>
    <variable name="dbCardsErr" messageType="dbi:dbFindCardsFaultMsg"/>
  </variables>

  <sequence>

    <!-- Passende Felder aus Datenbank lesen -->

    <assign>
      <copy>
        <from>"Manuel Pfeiffer"</from>
        <to variable="dbCardsParam" part="ownerName"/>
      </copy>
    </assign>

    <invoke partnerLink="db" portType="dbi:dbAccessPT"
      operation="findCards" inputVariable="dbCardsParam"
      outputVariable="dbCardsRes">
      <catch faultName="dbi:dbFindCardsError" faultVariable="dbCardsErr">

        <!-- Fehlernachricht senden und Transaktion beenden -->

        <assign>
          <copy>
            <from>"Database access error"</from>
            <to variable="errSRWU" part="result"/>
          </copy>
        </assign>

        <throw faultName="ccTransError" faultVariable="errSRWU"/>
      </catch>
    </invoke>

    <!-- Savepoint setzen -->

    <bpdl:createSavepoint name="savepoint1"/>

    <!-- technische Aktivit"at f"ur jede Kreditkarte aufrufen -->

    <bpdl:foreach loopVariable="checkBalanceIn" loopPart="balance"
      fieldVariable="dbCardsRes" fieldPart="balances">
```

```
        <invoke partnerLink="checkBalance" portType="ci:checkBalancePT"
          operation="checkBalance" inputVariable="checkBalanceIn"
          outputVariable="checkBalanceOut">
          <catch faultName="ci:checkBalanceError" faultVariable="checkBalanceErr">

            <!-- Savepoint laden -->

            <bpdl:restoreSavepoint name="savepoint1"/>
          </catch>
        </invoke>
      </bpdl:foreach>

      <!-- Transaktion beenden -->

      <assign>
        <copy>
          <from variable="transSRWU" part="balance"/>
          <to variable="outSRWU" part="balance"/>
        </copy>
        <copy>
          <from>"OK"</from>
          <to variable="outSRWU" part="result"/>
        </copy>
      </assign>
    </sequence>
  </bpdl:transaction>

  <!-- Ausgabe-SRWU senden -->

  <reply partnerLink="SWF" portType="ccti:creditCardTransactionPT"
    operation="creditCardTransaction" variable="outSRWU"/>
</sequence>
</process>
```

Quelltext 28: Masterflow.fbpdl

```
<process name="InternetPayment" bpdl:version="1" bpdl:revision="0"
  targetNamespace="http://example.com/bpdl/workflow/InternetPayment_V1"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process"
  xmlns:bpdl="http://www.schlodder.de/schemas/bpdl/f/"

  <!-- Subworkflow Interface -->
  xmlns:ccti="http://example.com/bpdl/interface/CreditCardTransaction_V1"

  <!-- Masterflow Interface -->
  xmlns:ipi="http://example.com/bpdl/interface/InternetPayment_V1">

  <partnerLinks>

    <!-- PartnerLink f"ur den Masterflow -->

    <partnerLink name="MF"
```

```
    partnerLinkType="ipi:internetPaymentLT"
    myRole="internetPayment"/>

  <!-- PartnerLink f"ur den Subworkflow -->

  <partnerLink name="ccTrans"
    partnerLinkType="ccti:creditCardTransactionLT"
    partnerRole="creditCardTransaction"
    myRole="creditCardTransactionMessage"/>
</partnerLinks>

<!-- In partners wird das externe Interface des MF festgelegt -->

<partners>
  <partner name="internetPayment">
    <partnerLink name="MF"/>
  </partner>
</partners>

<variables>

  <!-- Variablen f"ur den Masterflow -->

  <variable name="InSRWU" messageType="ipi:internetPaymentInputSRWU"/>
  <variable name="OutSRWU" messageType="ipi:internetPaymentOutputSRWU"/>

  <!-- Variablen f"ur die fachliche Aktivit"at -->

  <variable name="ccIn" messageType="ccti:creditCardTransactionInputSRWU"/>
  <variable name="ccOut" messageType="ccti:creditCardTransactionOutputSRWU"/>
  <variable name="ccErr" messageType="ccti:creditCardTransactionErrorSRWU"/>
  <variable name="ccMsg" messageType="ccti:creditCardTransactionMessageSRWU"/>
</variables>

<sequence>

  <!-- Eingabe-SRWU empfangen -->

  <receive partnerLink="MF" portType="ipi:internetPaymentPT"
    operation="internetPayment" variable="InSRWU"/>

  <!-- Subworkflow aufrufen -->

  <assign>
    <copy>
      <from variable="InSRWU" part="creditCardNumber"/>
      <to variable="ccIn" part="creditCardNumber"/>
    </copy>
    <copy>
      <from variable="InSRWU" part="amount"/>
      <to variable="ccIn" part="amount"/>
    </copy>
    <copy>
      <from variable="InSRWU" part="destinationAccount"/>
```

```
      <to variable="ccIn" part="destinationAccount"/>
    </copy>
  </assign>

  <flow>
    <sequence>
      <invoke partnerLink="ccTrans" portType="ccti:creditCardTransactionPT"
        operation="creditCardTransaction" inputVariable="ccIn"
        outputVariable="ccOut">
        <catch faultName="ccti:ccTransError" faultVariable="ccErr">

          <!-- Fehler bearbeiten -->

          <empty/>
        </catch>
      </invoke>
    </sequence>

    <!-- parallel Nachrichten empfangen (#Hier stimmts noch nicht...) -->

    <sequence>
      <receive partnerLink="ccTrans" portType="ccti:creditCardTransactionPT"
        operation="sendMessage" variable="ccMsg"/>
    </sequence>
  </flow>

  <!-- Ergebnis auswerten -->

  <switch>
    <case condition="bpws:getVariableData(ccOut, result) != 'Error'">
      <assign>
        <copy>
          <from>Credit card operation completed</from>
          <to variable="OutSRWU" part="result"/>
        </copy>
      </assign>
    </case>
    <otherwise>
      <assign>
        <copy>
          <from>"Credit card operation failed"</from>
          <to variable="OutSRWU" part="result"/>
        </copy>
      </assign>
    </otherwise>
  </switch>

  <!-- Ausgabe-SRWU senden -->

  <reply partnerLink="SWF" portType="ipi:creditCardTransactionPT"
    operation="creditCardTransaction" variable="OutSRWU"/>
 </sequence>
</process>
```

C.2.3 Datenbankbeschreibungen

Hier findet sich ein Beispiel für eine Datenbankbeschreibung.

Quelltext 29: Database.dbpdl

```
<database name="DB" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/database/DB_V1"
  xmlns="http://www.schlodder.de/schemas/bpdl/d/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <table name="CREDITCARDS">
    <field name="CCNUM" type="xsd:string"/>
    <field name="OWNER" type="xsd:string"/>
    <field name="BALANCE" type="xsd:decimal"/>
  </table>
  <table name="ACCOUNTS">
    <field name="ACCOUNT" type="xsd:string"/>
    <field name="OWNER" type="xsd:string"/>
    <field name="BALANCE" type="xsd:decimal"/>
  </table>
</database>
```

C.2.4 Parameterbeschreibungen

Im folgenden sind Parameterbeschreibungen für einen Masterflow (Quelltext 30), einen Subworkflow (Quelltext 31), den CC/P (Quelltext 32), den MFC (Quelltext 33), den Batch Carrier (Quelltext 34), den SWFC (Quelltext 35) und den DS (Quelltext 36) abgebildet.

Quelltext 30: Masterflow.sbpdl

```
<settings name="InternetPayment" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/InternetPayment_V1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/">

  <option name="verbose" value="true" type="xsd:boolean"/>
  <option name="maxRetries" value="3" type="xsd:integer"/>
  <option name="PD" value="example.com:te_highperf" type="xsd:string"/>
</settings>
```

Quelltext 31: Subworkflow.sbpdl

```
<settings name="CreditCardTransaction" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/CreditCardTransaction_V1"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <option name="verbose" value="true" type="xsd:boolean" />
  <option name="cobolcalls" value="true" type="xsd:boolean" />
  <option name="maxRetries" value="3" type="xsd:integer" />
  <option name="carrier" value="IMS" type="xsd:string" />
</settings>
```

Quelltext 32: CCP.sbpdl

```
<settings name="CCP" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/CCP_V1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/">

  <option name="bpdlURL" value="file:///etc/te/bpdl" type="xsd:string"/>
</settings>
```

Quelltext 33: MFC.sbpdl

```
<settings name="MFC" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/MFC_V1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/">

  <option name="bpdlURL" value="file:///etc/te/bpdl" type="xsd:string"/>
</settings>
```

Quelltext 34: BC.sbpdl

```
<settings name="BC" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/BC_V1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/">

  <property name="swfc" value="${base}/swfc"/>
  <property name="bc" value="${base}/bc"/>
  <property name="base" value="file:///usr/lib"/>

  <option name="savefile" value="${bc}/savefile" type="xsd:string"/>
  <option name="classpath" value="${swfc}/app/actnonres" type="xsd:string"/>
  <option name="saclasspath" value="${swfc}/app/activities" type="xsd:string"/>
  <option name="tmclasspath" value="${swfc}/app/runtimes:${swfc}/swfc.jar"
    type="xsd:string"/>
  <option name="worker" value="5" type="xsd:int"/>
  <option name="retry" value="3" type="xsd:int"/>
</settings>
```

Quelltext 35: SWFC.sbpdl

```
<settings name="SWFC" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/SWFC_V1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/">

  <property name="swfc" value="file:///usr/lib/swfc"/>

  <option name="bpdlURL" value="${swfc}/bpdl" type="xsd:string"/>
  <option name="activityURL" value="${swfc}/activities" type="xsd:string"/>
  <option name="pluginURL" value="${swfc}/plugins" type="xsd:string"/>
</settings>
```

Quelltext 36: DS.sbpdl

```
<settings name="DS" version="1" revision="0"
  targetNamespace="http://example.com/bpdl/parameters/DS_V1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.schlodder.de/schemas/bpdl/s/">

  <option name="dbms" value="database1:db2" type="xsd:string"/>
</settings>
```

Anhang D

Inhalt der CD

Im folgenden wird der Inhalt der CD erläutert, die dieser Ausarbeitung beiliegt. Verzeichnisse sind dabei durch einen Schrägstrich (/) nach dem Namen gekennzeichnet. Sofern ein `Makefile`[1] vorliegt, können die verfügbaren *Targets*[2] durch Aufruf von `make␣h` im entsprechenden Verzeichnis angezeigt werden.

index.html Dies ist der zentrale Index der CD. Mit seiner Hilfe kann ein Großteil der CD bequem mit einem Web-Browser durchsucht und betrachtet werden.

Dokumentation/ enthält diese Ausarbeitung in verschiedenen Formaten sowie die zugehörigen Quellen.

- **Ausarbeitung.pdf** Das Adobe PDF-Format empfiehlt sich für die Online-Lektüre. Der Adobe Reader ermöglicht die Suche nach Stichworten und zeigt ein Verzeichnis für die schnelle Auswahl der Kapitel und Abschnitte. Außerdem lassen sich Fußnotentexte und andere Verweise durch einfaches Anklicken aufrufen, und auch die Links im Literaturverzeichnis lassen sich einfach aufrufen.
- **Ausarbeitung.ps** Das Postscript-Format ist für den Ausdruck am besten geeignet, da es von den meisten Druckern direkt unterstützt wird.
- **Ausarbeitung.dvi** DVI ist das native Ausgabe-Format von TEX. Diese Version der Ausarbeitung ist nur der Vollständigkeit halber enthalten (und für solche Leute, die auf DVI schwören).
- **html/** In diesem Verzeichnis ist eine HTML-Version der Ausarbeitung zu finden. Allerdings ist diese nur für seltene Fälle sinnvoll, da die Bilder teilweise nicht korrekt konvertiert wurden und die in diesem Dokument enthaltenen Listings fehlen.
- **API/** Hier ist die aus den kommentierten Quelltexten mit Hilfe von **javadoc** und **doxygen** [24] automatisch generierte Online-Dokumentationen des Batch-Carriers, des Subworkflowcontrollers, des Datenzugriffsdienstes und der BPDL-Compiler zu finden. Der Zugriff erfolgt mit einem Browser über die in **API/** abgelegte `index.html`.

[1] Die in dieser Arbeit verwendeten Makefiles sind zum größten Teil für **gnu-make** optimiert, dessen z/OS-Version in [63] beschrieben ist.

[2] Mit *Targets* bezeichnet man die Zieldateien, die von `make` anhand der im Makefile abgelegten Regeln erzeugt werden können. Die Ziele können auch virtuell sein und z.B. eine Gruppe von Dateien erzeugen.

Dokumentation/ (fortgesetzt)

Quellen/ Hier sind alle Quellen der Ausarbeitung zusammengefaßt. Ein Makefile in **Ausarbeitung/** erlaubt die automatische Erstellung der Ausarbeitung in den verschiedenen Formaten.

Ausarbeitung/ Hier liegen die LaTeX-Quellen der Ausarbeitung und das Makefile für die Erstellung der Ausarbeitung unter Linux. Der Aufruf von `make` ohne Argumente erzeugt das DVI-Format, durch `make␣ps`, `make␣pdf` und `make␣html` lassen sich die anderen Formate erstellen.

BPDL/ In diesem Verzeichnis liegen die in der Ausarbeitung verwendeten Beispiele für BPDL-Beschreibungen.

FIG/ Hier befinden sich die durch XFig erzeugten Grafiken im „Quelltext“: Das FIG-Format wird von XFig standardmäßig zum Speichern der Grafiken verwendet.

PICS/ Dieses Verzeichnis enthält die durch **gnuplot** erzeugten Diagramme im encapsulated Postscript (`.eps`) Format.

Performance/ In den folgenden Unterverzeichnissen finden sich die **gnuplot**-Scripte, mit deren Hilfe die Performance-Diagramme generiert wurden, sortiert nach Diagrammen. Die zugrundeliegenden Meßdaten enthalten absolute Meßwerte des verwendeten z/OS-Rechners und können deshalb nicht zur Verfügung gestellt werden.

PipeQueSHM/ Vergleichsmessung von IPC-Mechanismen, erstellt von **ipctest**.

Reflection/ Einfluß von Java Reflection auf die Gesamtperformance, erstellt mit dem Batch-Launcher und dem Subworkflowcontroller ohne Serverzugriff.

SerExtBin/ Vergleich zwischen serialisierter, externalisierter und binärer Übertragung, erstellt durch **extest**.

Server/ Serverzugriff, erstellt mit dem Batch-Launcher und dem Subworkflowcontroller mit Zugriff auf den externen Server.

TCP-UDP/ Geschwindigkeit von TCP und UDP, erstellt durch **iptest**.

SWFC/ In diesem Verzeichnis befinden sich die Beispieldateien für eine Subworkflow-Runtime und eine Java-Aktivität, die in Anhang B ab Seite 93 abgedruckt sind.

Literatur/ Hier findet sich ein Großteil der im Literaturverzeichnis ab Seite 171 aufgeführten Dokumente und einige zusätzliche, die in der Implementierungsphase geholt und nur teilweise benötigt wurden. Die meisten dieser Dokumente sind im Internet frei verfügbar.

Architekturanalyse der Java Virtual Machine unter zOS und Linux.pdf ist die Diplomarbeit von Marc Beyerle über die PRJVM.

Entwurf und Implementierung einer transaktionalen Subworkflowsteuerung .pdf ist die Diplomarbeit von Thomas Hornung.

IBM/ enthält verschiedene, online verfügbare Dokumente und Spezifikationen von IBM. Zu jedem der in diesem Verzeichnis vorhandenen Dokumente ist in Links.html der Link angegeben, über den dieses Dokument heruntergeladen wurde.

LaTeX/ Verschiedene Anleitungen mit hilfreichen Tips zu LaTeX.

Programme/ In diesem Verzeichnis finden sich die Quellen der im Rahmen dieser Diplomarbeit erstellten Programme.

Compiler Hier sind die Quellen zum Schnittstellen- und zum Subworkflow-Compiler untergebracht. Die fertigen Java-Klassen lassen sich per `ant`[3] erzeugen, die Klassen-Dokumentation kann durch Aufruf von `ant␣doc` generiert werden.

SWFC Dieses Verzeichnis enthält die Quellen zum Batch-Carrier, zum Subworkflowcontroller und zum Datenzugriffsdienst sowie für eine Aktivität und eine Subworkflow-Runtime. Außerdem findet sich hier ein Programm zum Generieren von Eingabedateien mit einer festgelegten Anzahl von SRWUs eines bestimmten Typs für den Batch-Carrier.

app/ Die für die Performance-Messungen verwendeten Beispiele einer Subworkflow-Runtime und einer Aktivität sind hier zu finden. Das `Makefile` erzeugt (bei Aufruf von `make` ohne weitere Argumente) die fertigen Java-Klassen. Duch `make␣doc` können die JavaDocs für die vorhandenen Quellen erzeugt werden.

swfc/ enthält die Quellen des Batch-Carriers (`launcher`), des Subworkflowcontrollers (`swfc.jar`), des externen Servers (`server`) und des Generators für Eingabedateien (`testGenerator.jar`). Das `Makefile` erzeugt unter z/OS den Batch-Carrier und den externen Server. Unter Linux (oder anderen Systemen) wird das Java-Archiv des Subworkflowcontrollers (beim Aufruf von `make` ohne Argumente) bzw. die Online-Dokumentation aller in diesem Verzeichnis enthaltenen Programme (`make␣doc`) erzeugt.

Der Aufruf des Batch-Carriers erfolgt durch `launcher`, der externe Server wird durch `server` gestartet[4]. Der Subworkflowcontroller wird nicht direkt gestartet, sondern durch den Batch-Carrier geladen. Der Test-Generator kann schließlich durch `java␣-jar␣testGenerator.jar` ausgeführt werden.

Test-Programme/ In diesem Verzeichnis sind Test-Programme für verschiedene Problembereiche zu finden, die auch teilweise die Meßdaten für die Performance-Diagramme in Kapitel 7 ab Seite 63 lieferten. Außerdem befindet sich hier eine angepaßte Version des Subworkflow-Controllers von Thomas Hornung.

API/ Hier sind die (automatisch generierten) Online-Dokumentationen von einigen der in diesem Verzeichnis enthaltenen Test-Programme enthalten.

SWFC.Hornung.modified/ Dies ist der (an eine geänderte Verzeichnis-Struktur angepaßte) Subworkflowcontroller, den Thomas Hornung in seiner Arbeit erstellt hat, zusammen mit dem Launcher von Marc Beyerle.

comtest/ enthält (teilweise unfertige) Testprogramme zur Kommunikation von Java-Programmen über Sockets unter IMS und CICS sowie nativ unter z/OS UNIX (USS).

extest/ enthält ein Test-Programm zum Vergleich der Übertragungs-Geschwindigkeit von serialisierten, externalisierten und von Hand kodierten Paketen über TCP-Sockets.

[3] `ant` ist ein XML-basierter und in Java geschriebener Make-Ersatz für die Handhabung von Java-Builds. Dieses Programm ist in Eclipse [18] enthalten.

[4] Die erforderlichen Parameter lassen sich jeweils über den Kommandozeilenparameter `-h` in Erfahrung bringen.

Test-Programme/ (fortgesetzt)

ipctest/ ist ein Programm zum Vergleich der Performance verschiedener IPC-Methoden.

iptest/ enthält ein Programm zum Vergleich der Übertragungsraten von TCP und UDP.

jnitest/ Hier findet sich ein Beispiel-Programm für die Verwendung von JNI.

mutextest/ enthält ein Beispiel-Programm für die Verwendung von POSIX-Mutex-Locks.

parsetest/ enthält ein Beispiel für einen BPDL-Parser unter Verwendung von JAXB.

pipetest/ enthält ein Beispiel-Programm für die Prozeß-Kommunikation über Pipes.

queuetest/ Beispiel-Programm für die Kommunikation über System V Message Queues.

shmtest/ enthält ein Beispiel-Programm für die Kommunikation über System V Shared Memory Regions.

Arbeitsumgebung/ Dieses Verzeichnis enthält ein Abbild des unter z/OS verwendeten Benutzer-Verzeichnisses im HFS. Da die Programme unter den Unix System Services (USS) von z/OS entwickelt wurden, sind alle vorgenommenen Einstellungen den Shell-Konfigurationsdateien in diesem Verzeichnis zu entnehmen.

.profile Startup-Skript der voreingestellten Bourne Shell. Hier werden die Pfade und Umgebungsvariablen gesetzt und die komfortablere `csh` gestartet.

.cshrc Alias-Definitionen für den bequemeren Umgang mit den Shell-Commandos der `csh`.

app/ Hier wurden die SWFRs und Java-Aktivitäten bzw. -System-Plugins untergebracht.

bin/ Hier wurden ein paar Hilfsprogramme bzw. Shell-Scripte untergebracht. Außerdem wurden hier die folgenden Programme aus [63] installiert: **less**, **gnu-make**, **diff-utils**, **rcs** und **gzip**. Mit `start.jcl` ist hier außerdem ein JCL-Script für den Aufruf des Batch-Carriers unter TSO enthalten[5].

bpdl/ Hier sollten die BPDL-Quellen abgelegt werden.

comp/ Dieses Verzeichnis enthielt die BPDL-Compiler.

jprogs/ In diesem Verzeichnis wurden **Java 1.4**, **ant** sowie das **Java Web Services Developer Pack (JWSDP)** [64] von Sun untergebracht, das die von den BPDL-Compilern benötigten Laufzeit-Bibliotheken von JAXB und JAXP enthält.

lib/ In diesem Verzeichnis sollten Java- und C-Libraries untergebracht werden.

log/ Hier lagen die von den verschiedenen Komponenten der TE erzeugten Log-Dateien.

old/ Dieses Verzeichnis enthielt den Subworkflowcontroller von Thomas Hornung.

swfc/ Hier waren Batch-Carrier, SWFC und Server untergebracht, ebenso wie der Test-Generator.

SWFC.Hornung/ In diesem Verzeichnis findet sich der komplette Inhalt der CD, die Thomas Hornung seiner Diplomarbeit beigelegt hat. Eine angepaßte Version der Quellen zum Subworkflowcontroller findet sich unter **Test-Programme/**.

[5] Das JCL-Script wurde in einer frühen Implementierungsphase geschrieben. Deshalb müssen vor dem Einsatz des Scriptes die darin enthaltenen Pfade angepaßt werden.

Glossar

ACID faßt die zentralen Bedingungen für die Ausführung einer Transaktion zusammen: Die Atomizität (Atomicity) garantiert, daß die Transaktion entweder ganz oder gar nicht ausgeführt wird. Durch die Konsistenzerhaltung (Consistency) ist sichergestellt, daß das System keinen inkonsistenten Zustand einnimmt. Die Isolation garantiert, daß die Ausführung einer Transaktion keine Auswirkung auf das Ergebnis einer anderen Transaktion haben kann. Die Dauerhaftigkeit (Durability) stellt schließlich sicher, daß kein Ergebnis einer Transaktion verloren gehen kann.

Adabas ist ein Datenbanksystem der Software AG.

CICS ist ein Transaktionssystem der IBM.

COBOL ist eine Sprache, die hauptsächlich für die Programmentwicklung im Business-Umfeld entworfen wurde.

Das Common Connector Framework (CCF) wurde von IBM entwickelt. Es dient der Anbindung vorhandener Backend-Systeme an WebSphere und andere Java-basierte Frontends.

DB2 ist ein Datenbanksystem der IBM.

Enterprice Java Beans (EJB) sind die Komponenten der Java 2 Enterprise Edition.

IMS ist eine Datenbank- und Transaktionsumgebung der IBM.

Die Jahr-2000-Klippe ist ein schönes Beispiel dafür, wie häufig die Lebenszeit einer Software unterschätzt wird. Sehr viele Programme arbeiteten vor dem Jahr 2000 noch mit zweistelligen Jahreszahlen, und nur die wenigsten hatten dabei ein Fenster implementiert, das eine korrekte Handhabung des Jahreswechsels ermöglicht hätte. So kam es zu dem berühmten Millenium-Bug, der fast die gesamte Software-Industrie betraf, und dem erstaunlicherweise erst im Jahr 1999 tatsächlich zu Leibe gerückt wurde.

Java ist eine Sprache, die von Sun für den Einsatz auf sog. „embedded devices“ entwickelt wurde. Als Sun die Universalität des zugrundeliegenden Prinzips erkannte, entschied die Firma, daraus eine Programmiersprache für die neue, vernetzte Welt zu machen, die eine Ausführung von Programmen auf jeder beliebigen Plattform erlauben sollte. Zunächst schien Java ein voller Erfolg zu werden, wurde dann über das berüchtigte embrace-and-extend-Verfahren von einer Firma, die ihr Monopol gefährdet sah, so weit verändert, daß die verschiedenen Dialekte inkompatibel zueinander wurden.

Die Java 2 Enterprise Edition (J2EE) liefert ein komponentenbasiertes Framework für die Entwicklung verteilter Systeme.

Die Java Architecture for XML Binding (JAXB) soll die Handhabung vom XML-kodierten Daten in Web-Anwendungen vereinfachen.

Die J2EE Connector Architecture (JCA) liefert ein Interface für die Anbindung von vorhandenen, meistens nicht Java-basierten Systemen an Java.

JIT-Compiler (just in time comiler) sind eine weit verbreitete Technologie, um interpretierte Sprachen schneller zu machen. Dabei werden einzelne Programmsequenzen (Fuktionen, Klassen-Methoden etc.) bei ihrer ersten Ausführung in Maschinensprache übersetzt, und können dadurch bei einem weiteren Aufruf schneller abgearbeitet werden.

.NET ist die Überall-Architektur von Microsoft, mit der diese Firma vor allem Suns Java Konkurrenz machen will. Nebenbei löst das .NET-Framework Stück für Stück (auch Microsoft-intern) die älteren Architekturen COM und DCOM ab. .NET bietet wie Java die Fähigkeit, einmal erstellte Programme auf verschiedener Hardware ablaufen zu lassen. Dank einiger Initiativen wie Mono könnte sich .NET sogar zu einer plattformübergreifenden Architektur entwickeln.

Oracle ist ein Datenbanksystem der gleichnamigen Firma.

Die Persistent Reusable Java Virtual Machine ist eine IBM-eigene Erweiterung der Technik der von Sun definierten Java Virtual Machine um die Fähigkeit, mehrere Transaktionen hintereinander ablaufen zu lassen. Ohne die Notwendigkeit, die JVM jedes Mal neu starten zu müssen, bleibt die transaktionale Isolierung doch voll erhalten.

Quality of Service (QoS) ist ein Begriff, der eine Priorisierung anhand bestimmter Kriterien umschreibt, die die Gesamtleistung eines Systems unter gegebenen Gesichtspunkten verbessert.

Service Level Agreements (SLAs) dienen dazu, zwischen zwei Partnern eine bestimmte Leistung (zu einem bestimmten Preis) zu vereinbaren.

Tuxedo ist ein Transaktionssystem der Firma BEA.

Units of Work (UoWs) sind klar voneinander getrennte Teile einer Gesamtaufgabe, häufig auch einzelne Aufträge innerhalb eines Systems (z.B. Transaktionssystem). Der Begriff ist relativ generischer Natur und wird entsprechend vielseitig verwendet.

z/OS ist ein Betriebssystem von IBM für die z/Series-Rechnerarchitektur. Es setzt üblicherweise auf z/VM auf, läuft also neben anderen Betriebssystemen in einer virtuellen Rechnerumgebung auf derselben Hardware. z/OS ist der Nachfolger von OS/390, das wiederum MVS beerbt hat. Die drei Begriffe (z/OS, OS/390 und MVS) werden in der Literatur von IBM quasi synonym verwendet und treten infolgedessen häufig vermischt auf.

Abkürzungsverzeichnis

ACID	**A**tomicity, **C**onsistency, **I**solation and **D**urability
APPC	**A**dvanced **P**rogram-to-**P**rogram **C**ommunication (VTAM)
BPD	**B**usiness **P**rocess **D**esigner (TE)
BPDL	**B**usiness **P**rocess **D**efinition **L**anguage (TE)
BPEL4WS	**B**usiness **P**rocess **E**xecution **L**anguage for **W**eb **S**ervices
BMP	**B**atch **M**essage **P**rocessing Program/Region (IMS)
CCF	**C**ommon **C**onnector **F**ramework
CC/P	**C**hannel **C**ontroller / **P**rofiler (TE)
CICS	**C**ustomer **I**nformation **C**ontrol **S**ystem (IBM)
COBOL	**Co**mmon **B**usiness **O**riented **L**anguage
DASD	**D**irect **A**ccess **S**torage **D**evice (MVS)
DEDB	**D**ata **E**ntry **D**atab**a**se (IMS)
EJB	**E**nterprise **J**ava**B**eans (Sun)
HFS	**H**ierarchical **F**ile **S**ystem (MVS)
IFP	**I**MS **F**ast **P**ath Program/Region (IMS)
IMS	**I**nformation **M**anagement **S**ystem (IBM)
IVP	**I**nstallation **V**erification **P**rocedure (IBM)
JAXB	**J**ava **A**rchitecture for **X**ML **B**inding (Sun)
J2EE	**J**ava **2** **E**nterprise **E**dition (Sun)
JCA	**J**ava **C**onnector **A**rchitecture (Sun)
JCL	**J**ob **C**ontrol **L**anguage (MVS)
JECL	Job **E**ntry **C**ontrol **L**anguage (MVS)
JES	**J**ob **E**ntry **S**ubsystem (MVS)
JMP	**J**ava **M**essage **P**rocessing Program/Region (IMS)
JMS	**J**ava **M**essage **S**ervice (J2EE)
MFC	**M**aster**f**low**c**ontroller (TE)
MPP	**M**essage **P**rocessing **P**rogram/Region (IMS)
MPR	**M**essage **P**rocessing **R**egion (IMS)
MVS	**M**ultiple **V**irtual **S**torage (IBM)

ODBC	**O**pen **D**atab**a**se **C**onnectivity
PDSE	**P**artitioned **D**ata **S**et **E**xtended (MVS)
PDS	**P**artitioned **D**ata **S**et (MVS)
PRJVM	**P**ersistent **R**eusable **J**ava **V**irtual **M**achine (IBM)
QoS	**Q**uality **o**f **S**ervice
RACF	**R**esource **A**ccess **C**ontrol **F**acility (MVS)
RMF	**R**esource **M**easurement **F**acility (MVS)
SAC	**S**hareable **A**pplication **C**lass loader (PRJVM)
SLA	**S**ervice **L**evel **A**greement
SPA	**S**cratch **P**ad **A**rea (CICS)
SQL	**S**tructured **Q**uery **L**anguage
SRB	**S**ervice **R**equest **B**lock (MVS)
SRU	**S**ervice **R**esponse **U**nit (TE)
SRWU	**S**ervice **R**equest **W**ork **U**nit (TE)
STP	**S**traight **T**hrough **P**rocessing
SWFC	**S**ub**w**ork**f**low**c**ontroller (TE)
SWFR	**S**ub**w**ork**f**low-**R**untime (TE)
SWFS	**S**ub**w**ork**f**low-**S**upervisor (TE)
TCB	**T**ask **C**ontrol **B**lock (MVS)
TMC	**T**rusted **M**iddleware **C**lass Loader (PRJVM)
TSO	**T**ime **S**haring **O**ption
UML	**U**nified **M**odelling **L**anguage
UoW	**U**nit **o**f **W**ork
USS	**U**NIX **S**ystem **S**ervices (MVS)
VSAM	**V**irtual **S**torage **A**ccess **M**ethod (MVS)
VTAM	**V**irtual **T**elecommunications **A**ccess **M**ethod (MVS)
WSDL	**W**eb **S**ervices **D**escription **L**anguage
WSEL	**W**eb **S**ervices **E**ndpoint **L**anguage
WSFL	**W**eb **S**ervices **F**low **L**anguage

Das vorliegende Abkürzungsverzeichnis enthält einige Akronyme, die ansonsten nicht in diesem Dokument vorkommen. Sie können aber eine große Hilfe sein, wenn man sich erstmals in die Literatur zu z/OS, IMS oder CICS einliest.

Die meisten Abkürzungen sind mit einer Kategorie gekennzeichnet, die die Zuordnung des Begriffs erleichtert. Hierbei steht TE für die Transaktionsmaschine, die anderen Kategorien sind entweder Produktkürzel oder Firmennamen bei Produkten dieser Firmen. Nicht gekennzeichnet sind allgemeine Begriffe und standardisierte Sprachen.

Auf eine Verlinkung der Abkürzungen wurde verzichtet, die passenden Kapitel bzw. Artikel lassen sich leicht über das Inhaltsverzeichnis, den Index oder das Literaturverzeichnis finden. Für Abkürzungen, die mit IMS und MVS bzw. z/OS zu tun haben, sei auf [65, 66] verwiesen.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] Free Software Foundation: *AUCTEX – An integrated TEX/LATEX environment for Emacs.* URL http://www.gnu.org/software/auctex/.

[2] Free Software Foundation: *GNU Emacs.* URL http://www.gnu.org/software/emacs/emacs.html.

[3] The LATEX Project: *LATEX – A document preparation system.* URL http://www.latex-project.org/.

[4] Deutschsprachige Anwendervereinigung TeX e.V. (DANTE): *WWW-Server von DANTE e.V.* URL http://www.dante.de/.

[5] The TeX Users Group (TUG): *TeX Users Group Web Site.* URL http://www.tug.org/.

[6] Lamport, Leslie: *Das LATEX Handbuch.* Addison-Wesley (Deutschland) GmbH, Bonn, 1995. ISBN 3-89319-862-1.

[7] Kopka, Helmut: *LATEX Einführung Band 1.* Addison-Wesley (Deutschland) GmbH, Bonn, 1994. ISBN 3-89319-664-1.

[8] xfig.org: *XFIG Drawing Program for the X Window System.* URL http://www.xfig.org/.

[9] gnuplot: *Gnuplot Homepage.* URL http://www.gnuplot.info/.

[10] Adobe Systems Incorporated: *Adobe Reader.* URL http://www.adobe.de/products/acrobat/readermain.html.

[11] International Business Machines Corporation: *IBM Bookmanager.* URL http://www-306.ibm.com/software/applications/office/bkmgr/.

[12] Kernel.Org: *The Linux Kernel Archives.* URL http://www.kernel.org/.

[13] The Open Source Development Labs (OSDL): *OSDL Web Site.* URL http://www.osdl.org/.

[14] Free Software Foundation: *GNU Operating System.* URL http://www.gnu.org/.

[15] International Business Machines Corporation: *z/OS UNIX System Services User's Guide*, September 2002. URL http://publib.boulder.ibm.com/epubs/pdf/bpxza430.pdf.

[16] International Business Machines Corporation: *z/OS UNIX System Services Programming Tools*, October 2001. URL http://publib.boulder.ibm.com/epubs/pdf/bpxza610.pdf.

[17] International Business Machines Corporation: *z/OS UNIX System Services Command Reference*, September 2002. URL http://publib.boulder.ibm.com/epubs/pdf/bpxza530.pdf.

[18] Eclipse Foundation: *The open platform for tool integration.* URL http://www.eclipse.org/.

[19] Sun Microsystems, Inc.: *Sun J2SE 1.4.2 SDK.* URL http://java.sun.com/.

[20] International Business Machines Corporation: *z/OS C/C++ Run-Time Library Reference*, December 2002. URL http://publib.boulder.ibm.com/epubs/pdf/edclb131.pdf.

[21] International Business Machines Corporation: *z/OS C/C++ Programming Guide IBM*, September 2002. URL http://publib.boulder.ibm.com/epubs/pdf/cbcpg130.pdf.

[22] International Business Machines Corporation: *z/OS C/C++ User's Guide*, October 2001. URL http://publib.boulder.ibm.com/epubs/pdf/cbcug110.pdf.

[23] International Business Machines Corporation: *C/C++ Language Reference*, October 2001. URL http://publib.boulder.ibm.com/epubs/pdf/cbclr110.pdf.

[24] Heesch, Dimitri van: *doxygen.* URL http://www.doxygen.org/.

[25] Hornung, Thomas: *Entwurf und Implementierung einer transaktionalen Subworkflowsteuerung.* Diplomarbeit, Fakultät für Informations- und Kognitionswissenschaften, Eberhard Karls Universität Tübingen, September 2003. URL http://www-ti.informatik.uni-tuebingen.de/~csp/diplom/thornung/.

[26] Spruth, Wilhelm G. und Joachim Franz: *Reengineering von Kernanwendungssystemen auf Großrechnern.* Informatik Spektrum, Heft 2:83 – 93, Mai 2003.

[27] Beyerle, Marc: *Architekturanalyse der Java Virtual Machine unter z/OS und Linux.* Diplomarbeit, Fakultät für Informations- und Kognitionswissenschaften, Eberhard Karls Universität Tübingen, September 2003.

[28] Sun Microsystems, Inc.: *Java Architecture for XML Binding (JAXB).* URL http://java.sun.com/xml/jaxb/.

[29] World Wide Web Consortium (W3C): *XML Schema.* URL http://www.w3.org/XML/Schema.

[30] Organization for the Advancement of Structured Information Standards (OASIS): *Web Services Business Process Execution Language*. URL `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

[31] Andrews, Tony and others: *Business Process Execution Language for Web Services*, May 2003. URL `ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf`.

[32] Christensen, Erik, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana: *Web Services Description Language (WSDL) 1.1*, March 2001. URL `http://www.w3.org/TR/wsdl`.

[33] World Wide Web Consortium (W3C): *Extensible Markup Language (XML)*. URL `http://www.w3.org/TR/2004/REC-xml-20040204/`.

[34] International Business Machines Corporation: *IBM Developer Kit for z/OS™, Java™ 2 Technology Edition – Persistent Reusable Java Virtual Machine User's Guide – Version 1 Release 4*, September 2003. URL `http://www-1.ibm.com/servers/eserver/zseries/software/java/pdf/prjvm14.pdf`.

[35] International Business Machines Corporation: *IBM Enterprise COBOL for z/OS*. URL `http://www-306.ibm.com/software/awdtools/cobol/zos/`.

[36] LegacyJ Corporation: *PERCobol*. URL `http://www.legacyj.com/lgcyj_perc1.html`.

[37] Garcia-Molina, Hector and Kenneth Salem: *Main memory database systems: An overview*. IEEE Transactions on Knowledge and Data Engineering, Vol. 4(No. 6):509 – 516, December 1992.

[38] International Business Machines Corporation: *MVS Programming: Workload Management Services*, June 2003. URL `http://publib.boulder.ibm.com/epubs/pdf/iea2w231.pdf`.

[39] International Business Machines Corporation: *DB2 Universal Database for OS/390 and z/OS – Application Programming and SQL Guide – Version 7SQL Reference*, June 2003. URL `http://publib.boulder.ibm.com/epubs/pdf/dsnaph13.pdf`.

[40] International Business Machines Corporation: *DB2 Universal Database for OS/390 and z/OS – SQL Reference – Version 7*, June 2003. URL `http://publib.boulder.ibm.com/epubs/pdf/dsnsqh13.pdf`.

[41] International Business Machines Corporation: *DB2 Universal Database for OS/390 and z/OS – ODBC Guide and Reference – Version 7*, June 2003. URL `http://publib.boulder.ibm.com/epubs/pdf/dsnodh13.pdf`.

[42] Long, Rick, Mark Harrington, Robert Hain, and Geoff Nicholls: *Ims primer*. URL `http://www.redbooks.ibm.com/redbooks/pdfs/sg245352.pdf`, January 2000.

[43] International Business Machines Corporation: *IMS Version 8 Command Reference*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfscr1f4.pdf.

[44] International Business Machines Corporation: *IMS Version 8 Application Programming: Design Guide*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsapgf4.pdf.

[45] International Business Machines Corporation: *IMS Version 8 Application Programming: Transaction Manager*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsaptf4.pdf.

[46] International Business Machines Corporation: *IMS Version 8 Application Programming: EXEC DLI Commands for CICS and IMS*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsapcf4.pdf.

[47] International Business Machines Corporation: *IMS Version 8 Utilities Reference: System*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsursf4.pdf.

[48] Arkins, Bill, Virgil Aguilar, Daniel Bourque, Giri Giritharan, and Nancy Stein: *Ims version 7 and java application programming*. URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246123.pdf, February 2001.

[49] Levin, Rose, Ken Blackman, and Bob Love: *Ims version 7 java update*. URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246536.pdf, April 2002.

[50] International Business Machines Corporation: *IMS Version 8 Java User's Guide*, March 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsjugf4.pdf.

[51] International Business Machines Corporation: *IMS Version 8 Java Guide and Reference*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsjavf5.pdf.

[52] International Business Machines Corporation: *IMS Version 8 Administration Guide: System*, August 2003. URL http://publib.boulder.ibm.com/epubs/pdf/dfsas1f4.pdf.

[53] International Business Machines Corporation: *CICS: Application Programming Primer*, June 1990. URL http://publib.boulder.ibm.com/epubs/book/dfhzp104.boo.

[54] International Business Machines Corporation: *CICS® Transaction Server for z/OS™ – CICS Supplied Transactions – Version 2 Release 2*, April 2002. URL http://publib.boulder.ibm.com/epubs/book/dfha7p05.boo.

[55] International Business Machines Corporation: *CICS® Transaction Server for z/OS™ – CICS Application Programming Guide – Version 2 Release 2*, January 2002. URL http://publib.boulder.ibm.com/epubs/pdf/dfhp3p00.pdf.

[56] International Business Machines Corporation: *CICS® Transaction Server for z/OS™ – CICS Application Programming Reference – Version 2 Release 2*, January 2002. URL `http://publib.boulder.ibm.com/epubs/pdf/dfhp4p00.pdf`.

[57] International Business Machines Corporation: *CICS® Transaction Server for z/OS™ – Java™ applications in CICS – Version 2 Release 2*, January 2002. URL `http://publib.boulder.ibm.com/epubs/pdf/dfhpjp00.pdf`.

[58] International Business Machines Corporation: *z/OS MVS JCL Reference IBM*, June 2003. URL `http://publib.boulder.ibm.com/epubs/pdf/iea2b631.pdf`.

[59] International Business Machines Corporation: *z/OS MVS JCL User's Guide IBM*, October 2001. URL `http://publib.boulder.ibm.com/epubs/pdf/iea2b510.pdf`.

[60] International Business Machines Corporation: *z/OS TSO/E Command Reference*, December 2002. URL `http://publib.boulder.ibm.com/epubs/pdf/ikj4c531.pdf`.

[61] Institut für Physikalische und Theoretische Chemie an der Universität Tübingen: *Datenauswertung in der Physikalischen Chemie*. URL `http://barolo.ipc.uni-tuebingen.de/pharma/index.html` (Grundlagen der Statistik → Fehler → Ausreißer → Nalimov).

[62] Rumbaugh, James, Ivar Jacobsen, and Grady Booch: *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Inc., Reading, MA, 1999. ISBN 0-201-30998-X.

[63] MacIsaac, Michael and others: *Open Source Software for z/OS and OS/390 UNIX*. International Business Machines Corporation, March 2002. URL `http://www-1.ibm.com/servers/eserver/zseries/zos/unix/redbook/sg245944.pdf`. Die portierten Programme sind unter `http://www-1.ibm.com/servers/eserver/zseries/zos/unix/redbook/index.html` zu finden.

[64] Sun Microsystems, Inc.: *Java Web Services Developer Pack (Java WSDP)*. URL `http://java.sun.com/webservices/jwsdp/index.jsp`.

[65] International Business Machines Corporation: *IMS Glossary*. URL `http://www-306.ibm.com/software/data/ims/shelf/glossary.htm`.

[66] International Business Machines Corporation: *TCP/IP Glossary*. URL `http://www.vm.ibm.com/related/tcpip/glos.html`.

Index