Diplomarbeit

Konzeption und Entwicklung einer Schnittstelle zur hierarchischen Abarbeitung räumlich verteilter Workflows

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Technische Universität Chemnitz
Fakultät für Informatik
Professur für Praktische Informatik

Raphael Kunis

Betreuer: Prof. Dr. Gudula Rünger

Chemnitz, den 19. April 2005

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Raphael Kunis
Chemnitz, den 19. April 2005

Danksagung

Danken möchte ich Frau Prof. Dr. Rünger und Herrn D. Beer, die mich im Rahmen dieser Diplomarbeit unterstützt haben.

Den Entwicklern und Nutzern von Enhydra Shark, die mir durch die Mailingliste bei Fragen und Problemen geholfen haben, möchte ich danken.

Dank gilt auch meinen Freunden, im Besonderen Daniela Pecher, Michael Rentzsch und Arne Werner, für die regen Diskussionen und aufmunternden Worte während dieser Arbeit.

Zu guter Letzt möchte ich meinen Eltern und meiner Familie danken. Ohne sie wäre mein Studium der Informatik und auch diese Diplomarbeit nicht möglich gewesen.

Inhaltsverzeichnis

1 Einleitung

Workflow Management Systeme (WfMS) werden heutzutage in Unternehmen vielfach eingesetzt, um die Abläufe des Unternehmens computergestützt und dadurch effizient umzusetzen. Dabei sind die Einsatzbereiche über viele Branchen verteilt. Beispiele hierfür sind Kreditanträge in Banken, Schadensfälle bei Versicherungen oder Softwareentwicklung in größeren IT Unternehmen. Die Gemeinsamkeit aller dieser Fälle ist, dass mehrere Arbeitsaufgaben hintereinander bzw. nebeneinander ablaufen, wobei diese bestimmten Bedingungen und Regeln unterliegen.

Geht man davon aus, dass die Arbeitsabläufe an ein bestimmtes Unternehmen gebunden sind und alle zugehörigen Daten an einer Stelle vorliegen, so wird ein zentrales WfMS eingesetzt. Sobald aber mehrere verteilte Teilbehörden einen Workflow gemeinsam abarbeiten, wie dies z. B. bei der Zusammenarbeit von Behörden üblich ist, müssen Mechanismen zum Einsatz kommen, die eine geteilte Workflowabarbeitung ermöglichen. Dabei kennt jede Teilbehörde nur die eigenen Arbeitsabläufe, verarbeitet aber evtl. Daten von anderen. Betrachtet man jetzt die gemeinsame Arbeit als den Gesamtworkflow, muss die Arbeit der einzelnen Teile koordiniert werden und eine Verteilung der für jeden Bearbeitungsteil relevanten Daten geschehen.

Ziel dieser Arbeit ist es, eine Möglichkeit dieser verteilten Abarbeitung von Geschäftsprozessen zu realisieren. Dabei soll ein bestehendes, jedoch frei wählbares WfMS genutzt werden um dieses Ziel zu erreichen.

In Kapitel 2 wird dazu eine Einführung und Begriffsdefinition der im Zusammenhang mit Workflows und Workflow Management Systemen verwendeten Begriffe gegeben. Weiterhin werden die Grundlagen für eine verteilte Abarbeitung von Geschäftsprozessen beschrieben. Am Ende des Kapitels werden vier ausgewählte WfMS-Implementierungen vorgestellt.

In Kapitel 3 wird das in dieser Arbeit entwickelte Konzept vorgestellt.

Kapitel 4 befasst sich mit der Implementierung des Prototypen und beschreibt detailliert die Umsetzung des Konzepts. Es werden außerdem Vor- und Nachteile der Realisierung sowie Probleme bei der Implementierung des Prototypen behandelt.

In Kapitel 5 werden vier Testfälle vorgestellt, die die Funktionstüchtigkeit des Prototypen zeigen sollen.

Das sechste Kapitel befasst sich mit Möglichkeiten der Erweiterung des Prototypen.

Im letzten Kapitel wird diese Arbeit abschließend zusammengefasst.

Da die meisten Begriffe in Bezug auf Workflow Management Systeme in englischer Sprache definiert sind, wird in dieser Arbeit bei Fachbegriffen ein dem englischen Begriff

entsprechender deutscher Ausdruck verwendet. Dabei werden von der Workflow Management Coalition (WfMC) in [Kre] gegebene Übersetzungen verwendet, die bei erstmaliger Nutzung mit dem englischen Originalbegriff und zusätzlich weiteren deutschen Synonymen in der Form term (*english term*, Synonym1 ,...) angegeben sind. Eine Auflistung der englischen Begriffe mit den zugehörigen deutschen Ausdrücken befindet sich im Glossar am Ende der Arbeit.

Für die Angabe von Quelltext wird die Darstellung `Quelltext` verwendet.

2 Einführung in Workflows und Workflow Management Systeme

Dieses Kapitel soll dem Leser eine grundlegende Einführung in Workflows und Workflow Management Systeme geben. Dazu wird als erstes die Workflow Management Coalition, die sich mit der Standardisierung von WfMS beschäftigt, vorgestellt. Anschließend erfolgt eine Definition der Begriffe Geschäftsprozess (*Business Process*, Geschäftsvorgang, Unternehmensprozess) und Workflow. Danach werden die einzelnen Teile eines WfMS beschrieben, wobei ein Top-Down Ansatz verwendet wird. Dabei werden vom Management System als Ganzes ausgehend die einzelnen Komponenten näher erläutert. Hierbei wird als Quelle das durch die WfMC beschriebene Workflow Referenzmodell (Vgl. [Hol95]) und das WfMC Dokument über Terminologie (Vgl. [Wor99]) verwendet. Einen Überblick über die einzelnen Komponenten gibt Abb. 2.1.

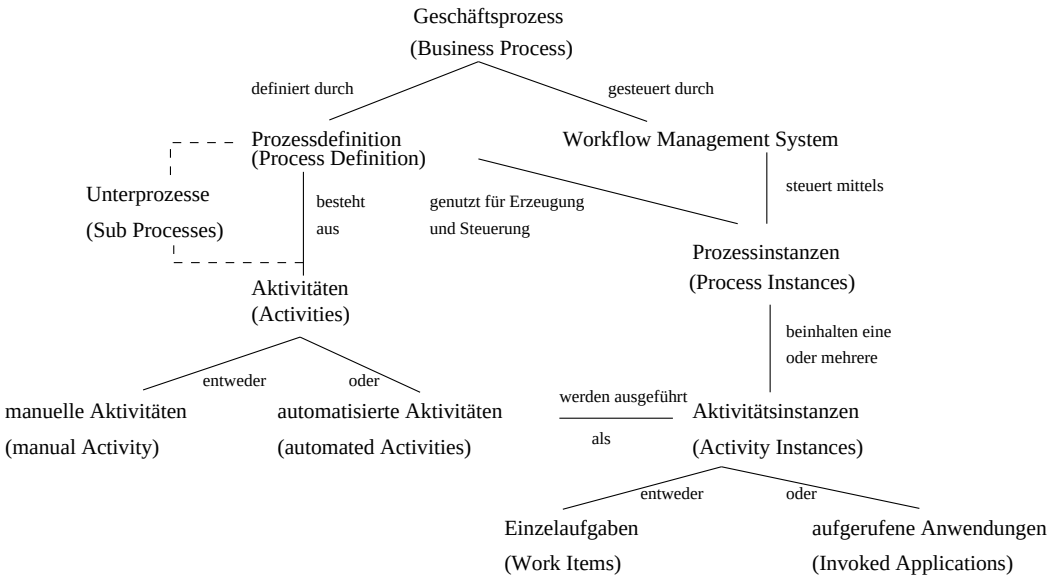

Abbildung 2.1: Überblick und Beziehungen der verwendeten Begriffe

In dieser Abbildung wird die Definitionsebene und die Abarbeitungsebene von Geschäftsprozessen dargestellt. Geschäftsprozesse bestehen aus Prozess- und Aktivitätsbeschreibungen (linker Teilbaum). Diese werden durch ein Workflow Management System instanziiert und abgearbeitet (rechter Teilbaum).

Da für diese Arbeit verteilte Workflows einen besonderen Stellenwert einnehmen, sollen diese anschließend beschrieben werden. Im letzten Unterabschnitt werden vier frei verfügbare Workflow Management Systeme vorgestellt.

2.1 Allgemeines zu WfMS

2.1.1 Workflow Management Coalition

Die Workflow Management Coalition ist eine 1993 als eigennütziges Gremium zur Standardisierung von Workflow Management Systemen gegründete Organisation, die versucht die bestehenden Workflow Management Systeme auf einen einheitlichen Standard zu bringen, um die Interoperabilität zwischen verschiedenen Lösungen zu gewährleisten.

Diese Standardisierung ist notwendig, da nur durch sie eine Zusammenarbeit zwischen unterschiedlichen Systemen möglich ist und Anwender unabhängig von konkreten Implementierungen entwickeln können. Für diese Standards veröffentlicht das Technical Committee (TC) der WfMC so genannte *Technical Committee Documents* (WfMC-TC).

Durch die WfMC wurde 1995 das Workflow Referenzmodell definiert (siehe auch [Hol95]). Dieses wird als Grundlage für die weiteren Betrachtungen von WfMS genutzt und soll im folgenden Abschnitt kurz vorgestellt werden. Zusätzlich hat die WfMC noch ein Glossar herausgegeben, welches die häufig verwendeten Begriffe im Workflow-Umfeld und entsprechende Übersetzungen ins Deutsche angibt[1].

2.1.2 Workflow und Geschäftsprozess

2.1.2.1 Geschäftsprozess

Unter einem Geschäftsprozess versteht man eine Menge von zusammengehörigen Aufgaben (*Activity*, Aktivitäten, Tätigkeit), die in ihrer Gesamtheit ein bestimmtes geschäftliches Ziel erfüllen. Die einzelnen Aktivitäten können hierbei sowohl automatisiert ablaufen als auch durch manuelle Aufgaben gekennzeichnet sein. Dabei ist der Ablauf von automatisierten Aufgaben vollständig durch ein Workflow Management System beschreibbar. Im Gegensatz dazu können manuelle Aufgaben zwar durch das WfMS angegeben werden, die Realisierung der Aufgabe selbst liegt aber außerhalb der Möglichkeiten des Systems. Weiterhin liegt dieser Art von Prozessen eine Organisationsstruktur mit funktionalen Rollen und Beziehungen zugrunde. Die Beschreibung ist informal, also nicht ohne Weiteres durch ein Computersystem abarbeitbar.

Ein Geschäftsprozess ist dabei nicht notwendigerweise nur an eine Firma oder ein Unternehmen gebunden, sondern kann sich auch über mehrere Firmen erstrecken (z. B. eine Kunde-Lieferant Beziehung). Auch innerhalb einer Firma kann sich ein Geschäftsprozess über mehrere unterschiedliche Abteilungen erstrecken, die sowohl an eine Lokalität gebunden oder über mehrere Firmensitze verteilt sein können.

[1] dieses ist derzeit jedoch nur als Vorschlag und nicht als Standard zu sehen und wird auch so bezeichnet

Die einzelnen Geschäftsprozesse sind an bestimmte Startbedingungen (Voraussetzungen) und Nachbedingungen (Ziele) gebunden, die sich bei jeder neuen Abarbeitung verändern können. Als Beispiel soll hier ein Bestellvorgang als Geschäftsprozess vorgestellt werden. Jede Bestellung wird nach einem bestimmten Ablaufplan, der in seiner Gesamtheit vorgegeben ist, abgearbeitet. Dabei sind die einzelnen Artikel der Bestellung und Informationen über den bestellenden Teilnehmer die Vorbedingung, die sich in verschiedenen Fällen unterscheidet. Das Ziel ist in allen Fällen gleich und besteht aus der Erfüllung der Bestellung. Dabei können verschiedene Abteilungen in den Prozess involviert sein. Zum Beispiel die Bestellannahme, das Lager usw.. Die Dauer eines solchen Prozesses kann von einem Tag bis zu einem oder mehreren Monaten variieren.

2.1.2.2 Workflow

Ein Workflow ist eine formale Beschreibung für eine automatisierte Abarbeitung von Geschäftsprozessen. Dabei werden Dokumente, Aufgaben und Informationen zwischen verschiedenen am Workflow beteiligten Teilnehmern (Personen, die für die Bearbeitung einzelner Teile eines Geschäftsprozesses zuständig sind) ausgetauscht. Der Workflow folgt dabei einer bestimmten Menge vordefinierter Regeln und soll ein bestimmtes, für die Firma wichtiges, Ziel erfüllen. Dabei muss er nicht notwendigerweise im IT-Bereich angesiedelt sein. So können die Regeln den Teilnehmern auch in Papierform vorliegen.
Beispiele für Workflows sind die Einhaltung von bestimmten Richtlinien bei der Verfassung und Inkraftsetzung von Gesetzen oder der Ablauf bei der Entwicklung von Software in größeren IT-Unternehmen.
Workflows werden auch häufig mit der Umstrukturierung von Geschäftsprozessen (*Business Process Reengineering*) in Verbindung gebracht, da sie die Beurteilung, Analyse, Modellierung sowie Definition und die daraus folgende betriebliche Durchführung von Geschäftsprozessen abdecken.
Die Modellierung eines Workflows geschieht durch eine Prozessdefinition. Diese legt die verschiedenen auszuführenden Aufgaben, verfahrenstechnische Regeln und eine Menge an Kontrolldaten, die für eine erfolgreiche Ausführung notwendig sind fest.
Es gibt zwei verschiedene Arten von Workflows. Das sind zum einen die „Production Workflows“, bei denen alle zu spezifizierenden Eigenschaften fest vordefiniert sind, und zum anderen die „Ad Hoc Workflows“, die Möglichkeiten bieten, um bestehende Regeln zur Laufzeit zu ändern oder neue Regeln hinzuzufügen.

2.1.3 Workflow Management System

Ein Workflow Management System ist ein Softwaresystem, das die Möglichkeiten für eine Automatisierung von Prozessabläufen in geschäftlichen Umgebungen bereitstellt. Dazu werden sowohl Prozessdefinitionen eingelesen als auch die Ausführung der einzelnen Prozessinstanzen und Prozessschritte (Aktivitäten) verwaltet und die Ausführung der einzelnen Aufgaben überwacht. Dabei ist es möglich Nutzerinteraktion zuzulassen und externe Programme für bestimmte Aufgaben zu nutzen.
Ein solches System lässt sich in drei funktionale Bereiche einteilen:

- **Workflow Entwicklungsfunktionen** umfassen alle Aufgaben, die zur Erstellung einer durch Computer verarbeitbaren Definition von Geschäftsprozessen notwendig sind. Das hierbei erzeugte Modell wird als Prozessdefinition bezeichnet. Diese Definition besteht aus einer oder mehreren Aktivitäten, die zur Ausführungszeit entweder durch Software oder durch menschliches Einwirken bearbeitet werden. Als Sprache zur Beschreibung solcher Definitionen wird durch die WfMC eine XML-basierte Sprache (XML Process Definition Language (XPDL)) vorgeschlagen und eingeführt. Weitere Informationen findet der interessierte Leser in [Nor02]. Neben XPDL gibt es noch eine Menge weiterer Definitionssprachen, die von verschiedenen WfMS-Implementierungen genutzt werden.

- **Laufzeitkontrollfunktionen** sind für die Steuerung des Ablaufs von Workflow Prozessen zuständig und sorgen für die richtige Reihenfolge der Abarbeitung der einzelnen Schritte. Dabei wird ein vorher definierter Prozess durch die Software interpretiert und es werden Instanzen für die Abarbeitung abgeleitet. Die Hauptkomponente, die diese Funktionen bereitstellt, ist die Workflow Management Kontrollsoftware oder auch Vorgangssteuerung (*Workflow Engine*, Vorgangssteuerungssystem).

  Im weiteren Verlauf dieser Arbeit wird der Begriff Vorgangssteuerungssystem verwendet. Dabei muss diese zentrale Komponente nicht notwendigerweise als ein festes System auf einem bestimmten Rechner laufen, sondern kann durchaus verteilt auf mehreren Computern ausgeführt werden.

- **Funktionen für die Laufzeitinteraktion** bezeichnen Funktionen für die Zusammenwirkung zwischen dem Nutzer und Workflow Management Systemen, um die einzelnen Aktivitäten abzuarbeiten. Dazu zählen sowohl Interaktionen zwischen dem Nutzer und dem WfMS als auch zwischen dem Nutzer und einer externen Anwendung. Für die Interaktion zur Laufzeit ist eine Abstimmung mit dem Vorgangssteuerungssystem notwendig, um einzelne Aufgaben zu starten, abzuschließen und Daten zwischen verschiedenen Anwendungen auszutauschen, falls dies notwendig sein sollte.

2.1.3.1 Workflow Referenzmodell

Das Workflow Referenzmodell wurde 1995 durch die WfMC standardisiert und beschreibt die Schnittstellen, die ein WfMS bereitstellen sollte. Dabei soll es die Vielfalt der existierenden WfMS-Implementierungstechniken und -umgebungen auf eine einheitliche Basis vereinigen. Die Hauptbestandteile und die Schnittstellen sind in Abb. 2.2 dargestellt.

Die einzelnen Schnittstellen beschreiben dabei:

- **Schnittstelle 1:** Spezifikation von Prozessdefinitionsdaten und ihr Austausch
- **Schnittstelle 2:** Möglichkeiten der Nutzung durch Klientenapplikationen
- **Schnittstelle 3:** Möglichkeiten der Nutzung externer Applikationen

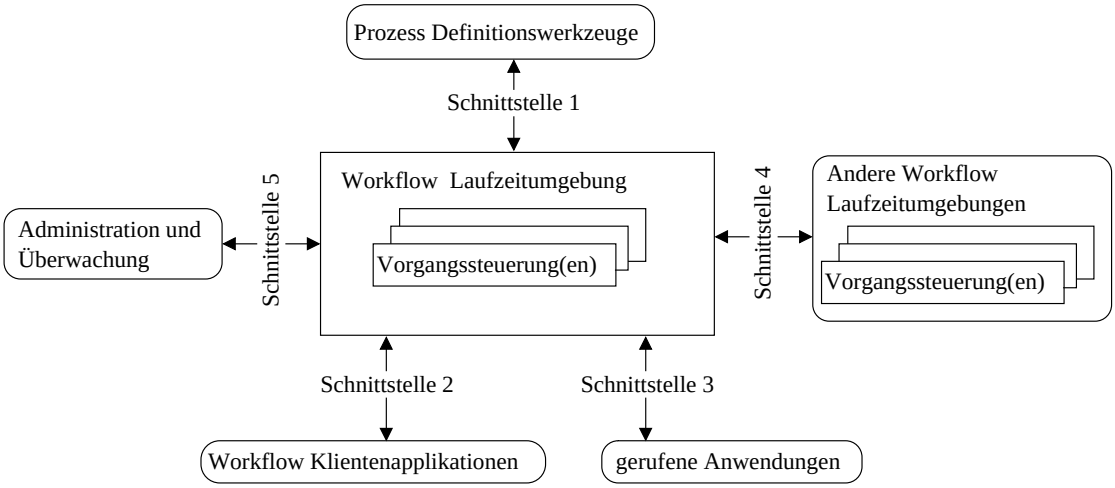

Abbildung 2.2: Workflow Referenzmodell der WfMC

- **Schnittstelle 4:** Kompatibilität für die Zusammenarbeit mehrerer WfMS
- **Schnittstelle 5:** Überwachungs- und Administrationsfunktionen

Für diese Arbeit sind hauptsächlich die Schnittstellen 3, 4 und 5 von Interesse, da sie die Möglichkeiten der Interaktion zwischen WfMS bieten. Dafür ist als erste Möglichkeit die Nutzung der Schnittstelle 4 bei allen beteiligten WfMS zu erwähnen, die genau aus diesem Grund spezifiziert wurde. Allerdings ist diese zum jetzigen Zeitpunkt bei den meisten freien WfMS noch nicht komplett implementiert. Bemühungen in dieser Richtung werden in Abschnitt 2.3 auf Seite 23 vorgestellt. Die zweite Möglichkeit ist die Nutzung von Schnittstelle 5 auf der Seite des gerufenen Systems und Schnittstelle 3 auf Seite des Rufenden.

Schnittstelle 1: Sie wurde entwickelt, um Prozessdefinitionen (*Process Definition*, Vorgangsmodell) zwischen verschiedenen Prozessdefinitionswerkzeugen, WfMS und Prozessdefinitonsspeichern auszutauschen. Dabei wurde eine Metasprache (XPDL) spezifiziert, mit der es möglich ist, Prozesse (in Form von Objekten), deren Attribute und Beziehungen zwischen den verschiedenen Teilelementen (z. B. Aktivitäten, Unterprozesse, prozessbezogene Daten, etc.) zu beschreiben.

Schnittstelle 2: Über diese Schnittstelle können Benutzerapplikationen mit der Laufzeitumgebung kommunizieren, um z. B. aktuelle Aufgaben einzusehen oder erledigte Aufgaben abzuschließen. Sie wurde durch das WfMC standardisiert, um Organisationen, die mit verschiedenen WfMS arbeiten die Möglichkeit zu bieten über *eine* Applikation auf die verschiedenen Systeme einheitlich zuzugreifen. Außerdem ist durch die Nutzung

einer einheitlichen Schnittstelle die Möglichkeit der Austauschbarkeit des Management Systems gegeben.
Der Nutzer greift dabei auf einen Arbeitskorb zu, der auf unterschiedliche Art und Weise zwischen Management System und Klient angeordnet sein kann. Ausführlicher wird diese Thematik in Abschnitt 2.1.3.2 auf Seite 10 behandelt.

Schnittstelle 3: Externe Applikationen werden vom Vorgangssteuerungssystem genutzt, um Anwendungen nutzbar zu machen, die weiterführende Aufgaben bearbeiten können, die vom System selbst nicht realisiert werden. Hier kann man sich eine große Vielfalt von Programmen vorstellen, z. B. Anwendungen zum Versenden von E-Mails, zum Lesen und Bearbeiten von Formularen usw.. Damit diese durch das Workflow Management System ausgeführt werden und dem System Informationen zurückgeben können, wurde diese Schnittstelle spezifiziert.
Dabei unterscheidet man zwischen Anwendungen, die sich für direkte Kommunikation mit einem WfMS eignen (*workflow enabled*) und Applikationsagenten, die ihrerseits wieder Schnittstellen für andere Applikationen bereitstellen.

Schnittstelle 4: Diese Schnittstelle wurde durch die WfMC beschrieben, um verschiedene WfMS über einheitliche Kommunikationsfunktionen miteinander verbinden zu können und somit eine verteilte Abarbeitung von Workflows zu gewährleisten. Dabei soll es möglich sein von einem System aus Aktivitäten bzw. Unterprozesse auf anderen Systemen zu instanziieren und zu starten, den Zustand von Prozessen und Aktivitäten abzufragen, Vorgangsdaten auszutauschen, Prozesse zu synchronisieren und lesend bzw. schreibend auf Prozessdefinitionen zuzugreifen. Dabei wurden im Referenzmodell zwei, für die Kompatibilität von Vorgangssteuerungsdiensten wichtige, Hauptaspekte definiert.
Der erste Aspekt betrifft die Notwendigkeit der Interpretationsmöglichkeit von Prozessdefinitionen durch verschiedene WfMS. Hierbei werden drei Vorschläge für diese gegeben. Der leichteste Fall ist die Nutzung von Systemen, die die gleichen Prozessdefinitionen interpretieren können (weil sie z. B. durch dasselbe Werkzeug erzeugt worden sind). Hier haben beide die gleiche Sicht auf die Daten und der Programmierer kann auf ein einheitliches Namens- und Objektmodell zurückgreifen. Ist dies nicht gegeben, müssen einzelne Teile der Prozessdefinition des einen Systems zur Laufzeit an das andere übergeben (exportiert) werden. Falls kein Austausch der Prozessdefinitionen möglich ist, weder durch geteilte Nutzung noch durch Exportierung, muss eine Gateway Applikation, die die unterschiedlichen Definitionen aufeinander abbildet, eingesetzt werden. Dabei müssen sowohl Objektnamen als auch Attribute durch den Gateway aufeinander abgebildet werden. Er muss also mit den Darstellungen der beteiligten WfMS umgehen und diese ineinander umwandeln können.
Nicht nur unterschiedliche Darstellungen der Prozessdefinitionen können die Kompatibilität von verschiedenen WfMS beeinflussen, sondern auch die Art und Weise, wie Funktionen des jeweils anderen Systems genutzt werden. Geht man davon aus, dass dies, egal welche Technik eingesetzt wird, immer über Funktionsaufrufe des anderen Systems geschieht (das können z. B. der Aufruf von Funktionen eines Webservices oder

Remote Procedure Calls sein), so müssen diese ein und die selbe Schnittstellendefinition implementieren. Sollte das nicht gegeben sein ist wieder ein Gateway notwendig, der die Funktionsrufe aufeinander abbilden kann.

Schnittstelle 5: Über diese wird einem Administrator ermöglicht Nutzer und Rollen zu verwalten, Betriebsmittel zu überwachen, Prozesse und Aktivitäten zu instanziieren und Instanzattribute zu ändern. Weitere Informationen dazu sind in Abschnitt 2.1.3.2 auf Seite 12 zu finden.

2.1.3.2 Allgemeine Workflow Produktstruktur

Im Gegensatz zum Workflow Referenzmodell, das die Schnittstellen beschreibt, die ein WfMS bereitstellen sollte, werden durch die generische Workflow Produktstruktur die einzelnen Komponenten eines Workflow Ausführungsdienstes (*Workflow Enactment Service*, Laufzeitsystem) aufgezeigt. Diese Struktur ist in Abb. 2.3 grafisch dargestellt.

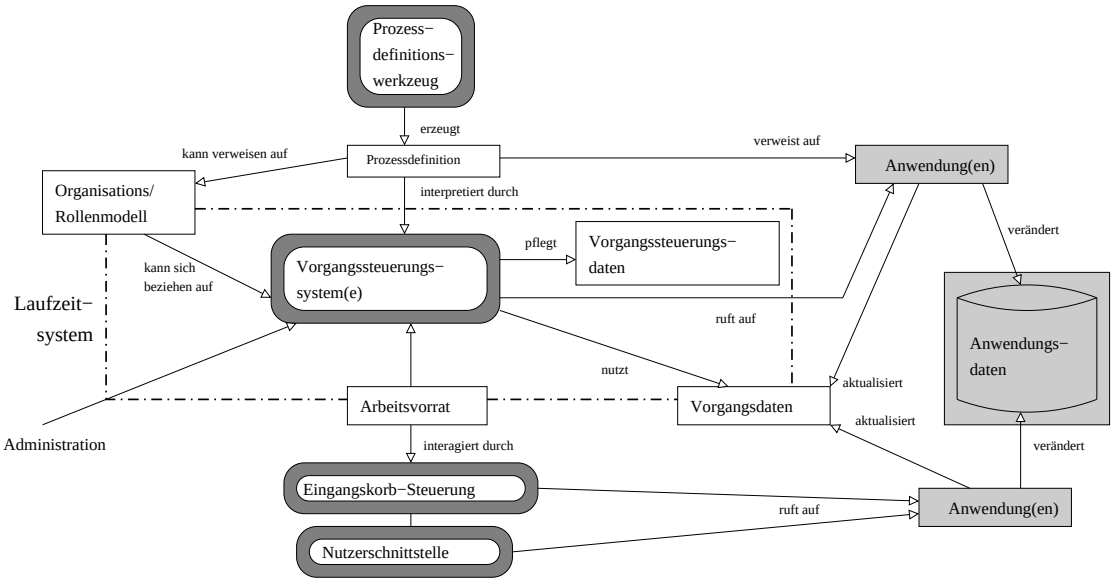

Abbildung 2.3: Allgemeine Workflow Produktstruktur

Die in der Grafik durch unterschiedliche Farben hervorgehobenen Teilkomponenten lassen sich in drei Gruppen einteilen. Das sind Kontrolldaten bzw. systeminterne Daten (weiß), Softwarebestandteile des WfMS (dunkelgrau) und externe Applikationen bzw. externe Daten. Eine detaillierte Beschreibung der einzelnen Komponenten wird in den nachfolgenden Abschnitten gegeben.

Vorgangssteuerungssystem und Laufzeitsystem Das Laufzeitsystem stellt die Softwareumgebung zur Instanziierung und Aktivierung von Prozessen bereit. Es kann genau

ein oder mehrere Vorgangssteuerungssysteme beinhalten. Diese wiederum sind für die Erzeugung von Prozessinstanzen sowie für die Ausführung der in der Prozessdefinition beschriebenen Teile dieser Prozesse zuständig.

Weiterhin werden über das Laufzeitsystem die Interaktionsmöglichkeiten mit externen Ressourcen zur Verfügung gestellt, sofern diese für die Ausführung der einzelnen Aufgaben benötigt werden. Die WfMC hat für diese Interaktionen einen einheitlichen Standard namens Workflow Application Programming Interface (WAPI) definiert. Dieser wird in Abschnitt 2.1.3.3 auf Seite 13 vorgestellt.

Das Laufzeitsystem wird als einzelne, abgeschlossene Einheit betrachtet, wobei es aus mehreren sowohl funktional als auch physisch getrennten Komponenten bestehen kann. Ein Beispiel wäre die Aufteilung von einem für alle Dienste zuständigen Vorgangssteuerungssystem in mehrere Vorgangssteuerungssysteme, wobei jedes für eine bestimmte Prozesskategorie zuständig ist. Diese Kategorien werden dabei vom Administrator willkürlich festgelegt.

Die Vorgangssteuerung ist die zentrale Komponente des Workflow Ausführungsdienstes und stellt die Laufzeitkontroll- bzw. Laufzeitausführungsumgebung bereit. Sollte es mehr als eine Vorgangssteuerung geben, so stellen alle zusammen die Gesamtfunktionalität bereit.

Die Aufgaben der Vorgangssteuerung sind vielfältig und umfassen die Interpretation von Prozessdefinitionen und die Kontrolle der Prozessinstanzen. Das heißt, sie erzeugt und beendet Prozessinstanzen und führt die für die Abarbeitung notwendigen Zustandswechsel durch. Ferner muss die Vorgangssteuerung Einzelaufgaben bestimmten Nutzern zuordnen können und zur Bearbeitung bereitstellen. Dafür sind auch An- und Abmeldemechanismen für die Nutzer durch die Vorgangssteuerung implementiert. Müssen Aufgaben von externer Software abgearbeitet werden bzw. wird in der Prozessdefinition die Nutzung einer externen Software zur Lösung einer Aufgabe definiert, so wird diese durch geeignete Mechanismen genutzt. Als letzte grundlegende Aufgabe soll noch die Bereitstellung von Diensten für Administration und Protokollierung genannt werden.

Wie bereits erwähnt, ist es möglich innerhalb einer Laufzeitumgebung mehrere Vorgangssteuerungssysteme zu nutzen. Dabei kann es - je nach gewählter Workflow Management System-Implementierung - unterschiedliche Möglichkeiten zur Aufteilung geben. Es ist z. B. die vorher angegebene Möglichkeit (Aufteilung anhand von Prozessdefinitionen) denkbar, oder auch eine Aufteilung nach Art des bereitgestellten Dienstes (ein Vorgangssteuerungssystem ist für die Prozessabarbeitung zuständig und eine zweite für Administration und Protokollierung, die Dritte für das Rufen externer Applikationen usw.). Beim ersten Fall ist dabei nicht zwingend ein Austausch der Daten zwischen den einzelnen Vorgangssteuerungssystemen notwendig; im Gegensatz dazu ist es beim zweiten Fall schwer vorstellbar ist, ohne Daten der einzelnen Prozesse, Protokollierungsdaten zu schreiben. Deswegen werden die verschiedenen Datenbestände meist zentral angelegt, damit sie für alle Vorgangssteuerungssysteme zugreifbar sind.

Klientenapplikationen und aufgerufene Applikationen Klientenanwendungen und aufgerufene Applikationen (*Invoked Applications*) werden zusammen als Workflow An-

wendungen bezeichnet. Darunter wird Software verstanden, die ganz oder teilweise die Abarbeitung von Tätigkeiten durch Zusammenarbeit mit dem Laufzeitsystem realisiert.

Als Klientenanwendung oder Klient eines WfMS werden Anwendungen bezeichnet, die Dienste und Hilfsmittel eines WfMS nutzen, um Arbeitslisten abzurufen, Prozesse zu initialisieren und zu kontrollieren (z. B. Aufgaben als erledigt bzw. beendet kennzeichnen), Prozessdefinitionsdaten zu manipulieren (z. B. bei Dokumentenverarbeitung Daten zu einem Dokument hinzufügen oder aus einem Dokument löschen). Zusätzlich kann der Nutzer - je nach Implementierung - auch Administrationsaufgaben über diese Schnittstelle erledigen.

Für die Interaktionen mit einem WfMS wird dabei ein sogenannter Eingangskorb (*Worklist*, Arbeitsliste, Arbeitsvorrat) genutzt. Dabei gibt es verschiedene Möglichkeiten, wie der Zugriff auf diesen erfolgen kann. Es sollen nachfolgend drei dieser Möglichkeiten vorgestellt werden. Dabei wird zwischen der WfMS-Umgebung (charakterisiert durch Hardware und Betriebssystem) und der Klientenumgebung unterschieden.

Beim Hostbasierten Modell (*Host Based model*) befindet sich die Softwarekomponente zum Zugriff auf den Eingangskorb auf der Seite der WfMS-Umgebung. Dieser greift über einen lokalen Funktionsaufruf auf den Arbeitsvorrat zu, der sich innerhalb der Laufzeitumgebung befindet.

Dagegen wird beim Modell des verteilten Datenspeichers (*Shared Filestore Model*) die Arbeitsliste als verteilter, durch Laufzeitumgebung und Klientenapplikation zugreifbarer Datenspeicher genutzt.

Im letzten Fall (Model des (entfernten) Prozedurrufes (*Procedure Call Model*)) befindet sich, wie schon beim Hostbasierten Modell, der Arbeitsvorrat in der Umgebung des Laufzeitsystems und wird über einen entfernten Methodenaufruf durch das Zugriffsmodul des Klienten genutzt.

Aufgerufene Anwendungen dienen dazu, die Funktionalität eines WfMS um die für ein bestimmtes Einsatzgebiet notwendigen Eigenschaften zu erweitern. Dies können z. B. Dokumentenverarbeitungswerkzeuge, E-Mail-Anwendungen oder Datenbankanwendungen sein. Dabei ist es, wie schon bei der Schnittstellenbeschreibung erwähnt, möglich, Anwendungen direkt an das WfMS anzubinden[2] oder einen Anwendungsagenten zwischen die Anwendungen und die Laufzeitumgebung zu schalten.

Genutzte Datenstrukturen/Datenbestände Die für WfMS definierten Datenstrukturen lassen sich in drei Klassen einteilen, die als Steuerungsdaten (*Control Data*, Vorgangssteuerungsdaten), Vorgangsdaten (*Workflow Relevant Data*, sachbezogene Daten, prozessbezogene Daten) und Anwendungsdaten (*Application Data*) bezeichnet werden.

- **Steuerungsdaten** können als interne Daten des WfMS bzw. des Vorgangssteuerungssystems angesehen werden und sind nicht nach außen hin sichtbar bzw. von anderer Software änderbar[3]. Sie repräsentieren den Status der aktuell

[2] sofern diese die APIs des speziellen Systems nutzen

[3] mit Ausnahme von Anwendungen, die über die Administrationsschnittstelle zugreifen

erzeugten und in der Ausführung befindlichen Prozessinstanzen. Dazu gehören z. B. der aktuelle Zustand der Prozessinstanzen und ihrer Aktivitäten, aber auch Wiederherstellungs- (Recovery) und Neustartpunkte für jeden Prozess.

- **Vorgangsdaten** werden vom WfMS genutzt, um die Zustandsübergänge zwischen Aktivitäten einer Prozessinstanz zu bestimmen. Dies kann durch Auswerten von Vor- und Nachbedingungen oder auch direkt durch Nutzung von Übergangsregeln (*Transition Condition*, Prozessübergangsbedingung) geschehen. Außerdem gehören zu diesen auch Daten, die durch das WfMS geändert werden sollen oder keiner externen Applikation zugehörig sind.

  Diese Daten können sowohl vom Management System selbst als auch durch Klientenapplikationen gelesen und verändert werden.

  Da die Daten des Prozesses durch die einzelnen Aktivitäten genutzt werden, müssen sie den Aktivitäten durch einen geeigneten Mechanismus zugänglich gemacht werden. Das ist bei WfMS-eigenen Aufgaben nicht weiter problematisch, sollte aber mit Blick auf Schnittstelle 3 (Aufrufen externer Applikationen) nicht unterschätzt werden.

  Die Vorgangsdaten lassen sich aus Sicht des WfMS in zwei Kategorien einteilen. Zum einen die *Typisierten Daten*, bei denen das System die Struktur kennt und sie auch lesen und verändern kann, und zum anderen die *Untypisierten Daten*, die nur weitergegeben werden können, da das WfMS sie nicht versteht.

- **Anwendungsdaten** unterscheiden sich von den zwei vorher genannten Datenarten in der Weise, dass sie nicht durch das WfMS zugreifbar sind. Sie sind auf eine bestimmte Anwendung zugeschnitten und müssen von dieser selbst verwaltet werden. Dadurch muss die Anwendung selbst dafür sorgen, dass Arbeitsschritten, wenn Sie durch die Applikation selbst ausgeführt werden, alle Daten vorhergehender Schritte dieser Anwendung bereitgestellt werden.

  Diese Daten können aber auch für einen bestimmten Prozess wichtig werden, wenn davon ein Zustandswechsel abhängig ist. Dazu müssen sie dem System zusätzlich als Vorgangsdaten zugänglich sein.

Workflow Administration Die Administration eines Workflow Management Systems übernimmt ein Nutzer oder eine Nutzergruppe mit speziellen Berechtigungen. Der Administrator greift über Schnittstelle 5 oder (bei manchen Implementierungen) über Schnittstelle 2 auf das Laufzeitsystem zu. Genutzt werden kann hierfür eine spezielle Software (falls das Management System nur die Schnittstelle bereitstellt) oder eine mitgelieferte Managementapplikation (Frontend des Laufzeitsystems).

Die Aufgaben des Administrators reichen dabei von der Verwaltung von Prozessen (Instanziierung, Herbeiführen von Zustandswechseln der Prozessinstanzen, Hinzufügen von Attributen, Abbrechen einzelner oder aller aktiven Prozessinstanzen, Versionsverwaltung) über Nutzermanagement bis hin zur Überwachung des Systems.

Beim Nutzermanagement ist nicht nur die Einteilung in Nutzer bzw. Nutzergruppen möglich, sondern es können mittels eines so genannten Organisationsmodells (*Organizational Model*, Kompetenzbeschreibung) Nutzer in bestimmte Anwendergruppen (*Organizational Role*, Rolle) eingeteilt werden. Dadurch kann in der Prozessdefinition festgelegt werden, dass eine bestimmte Aufgabe von einer Person aus einer bestimmten Anwendergruppe bearbeitet werden soll.

Die Überwachung des Systems wird genutzt, um aktuell aktive Prozessinstanzen bzw. Aktivitätsinstanzen und deren bisherigen Verlauf (Zustandswechsel, geänderte Daten, etc.) anzeigen lassen zu können und statistische Auswertungen vorzunehmen (hat sich z. B. die Effektivität der ausgeführten Arbeit im Vergleich zum Vorjahr gesteigert, also konnte die Abarbeitung von Workflows beschleunigt werden oder ist sie langsamer geworden).

2.1.3.3 Workflow API (WAPI)

Die WAPI umfaßt eine Menge von Funktionen und Austauschformaten für die fünf durch das Referenzmodell definierten Schnittstellen. Diese sollten genutzt werden um die Kompatibilität zwischen verschiedenen WfMS und zwischen WfMS und externen Applikationen zu gewährleisten. Dazu ist es möglicherweise notwendig, die spezifischen, vom jeweiligen Anbieter bereitgestellten Funktionen in die durch die WAPI spezifizierten Funktionen zu kapseln.

Die Funktionen decken Bereiche wie Sitzungsverwaltung, Definitions- und Objektoperationen, Prozesskontroll und Prozesszustandsfunktionen, Arbeitslistenhandhabung und weitere ab. Die Implementierung der WAPI ist dabei für die in Abschnitt 2.2 auf Seite 18 beschriebenen verteilten Workflows von zentraler Bedeutung, da ohne standardisierte Funktionen keine oder nur bedingt erreichbare Interoperabilität von Management Systemen zur Verfügung gestellt werden kann.

2.1.4 Prozess

Ein Prozess ist eine computerbasierte Darstellung von Geschäftsprozessen oder deren eigenständiger Teile. Diese Darstellung wird durch eine Prozessdefinition beschrieben, die durch das Vorgangssteuerungssystem eingelesen und interpretiert wird. Jeder Prozess besteht dabei aus einer Menge von eigenständigen Aufgaben, die durch Nutzer oder das System selbst abgearbeitet werden müssen, und Regeln, die die Abarbeitung steuern.

2.1.4.1 Prozessdefinition

Die Prozessdefinition beschreibt ein Netz von Aktivitäten und deren Beziehungen zueinander, Kriterien zum Starten und Beenden von Prozessen sowie Informationen zu den einzelnen Aufgaben (wer soll die Aufgabe bearbeiten, verknüpfte Anwendungen, zugehörige Daten). Diese Definitionen werden vor der Ausführung durch das WfMS erstellt und liegen diesem in einer je nach System unterschiedlichen Beschreibungssprache vor.

In einer Prozessdefinition können neben Aktivitäten auch Teilprozesse (*Sub Processes*, Unterprozesse) als Bausteine verwendet werden. Diese werden dann zu einem Teil des Gesamtprozesses. Üblich ist dies vor allem, wenn einzelne Teile in mehreren Prozessen vorkommen.

Durch das WfMC wurde als standardisierte Beschreibungssprache XPDL definiert.

Die Darstellung von Prozessen erfolgt meist als Aktivitätsnetzwerk oder Petri-Netz. Ein Beispiel eines Aktivitätsnetzwerks zeigt Abb. 2.4.

In den in dieser Arbeit genutzten Aktivitätsnetzwerken werden, obwohl es sich um gerichtete Graphen handelt, Richtungspfeile weitgehend weggelassen, um eine bessere Übersichtlichkeit zu erreichen. Hat eine Kante keinen Richtungspfeil, so führt sie von links nach rechts.

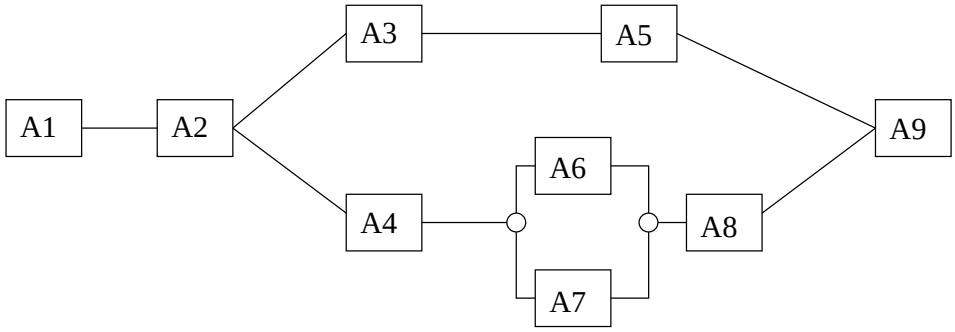

Abbildung 2.4: Beispielprozess aus 8 Aufgaben mit paralleler und verzweigender Abarbeitung

2.1.4.2 Prozessinstanz

Eine Prozessinstanz ist die Repräsentation der Ausführung eines Prozesses. Werden von einem Prozess mehrere Instanzen erzeugt, so sind diese völlig unabhängig voneinander. Jede Prozessinstanz wird vom WfMS erzeugt, verwaltet und beendet. Sie kann durch einen endlichen Automaten beschrieben werden, der bestimmte Zustände besitzt. Diese werden nachfolgend näher erläutert.

Prozessinstanzzustand

- **Initialisiert** die Prozessinstanz wurde erzeugt und alle notwendigen Datenstrukturen angelegt, aber es sind noch nicht alle Vorbedingungen zur Abarbeitung erfüllt
- **Laufend** die Abarbeitung wurde begonnen und die einzelnen Aktivitäten können abgearbeitet werden
- **Unterbrochen** die Prozessinstanz ist untätig und es werden keine neuen Tätigkeiten gestartet, bis sie wieder in den Zustand „Laufend“ versetzt wird
- **Aktiv** eine oder mehrere Aufgaben des Prozesses werden ausgeführt

- **Abgebrochen** die Prozessinstanz wurde angehalten, bevor sie die Nachbedingungen erfüllen konnte
- **Beendet** alle Aktivitäten des Prozesses sind komplett abgearbeitet und die Nachbedingungen sind erfüllt

Prozessablauf Als Prozessablauf (*Routing*) werden die Vorschriften, die die Ausführung von Prozessinstanzen beschreiben und der dadurch definierte Ablauf bei ihrer Verarbeitung verstanden. Dabei wird zwischen parallelem Ablauf (*parallel Routing*) und sequentiellem Ablauf (*sequential Routing*) unterschieden.

Beim sequentiellen Ablauf werden die einzelnen Aktivitäten immer (im gesamten Prozess) nacheinander ausgeführt, wogegen beim parallelen Ablauf mehrere Aktivitäten eines Prozesses gleichzeitig zur Abarbeitung bereit sind und ausgeführt werden können. Dabei werden parallele Verzweigungen (*AND-Split*) und Synchronisierungen (*AND-Join*) zwischen den einzelnen Aktivitäten genutzt.

Verzweigungen in Prozessabläufen In Prozessabläufen werden verschiedene Verzweigungen genutzt, um anstatt eines festen Plans die Möglichkeit zu bieten dynamische Funktionalitäten darzustellen (die in der realen Welt auch gegeben sind). Dabei gibt es UND-Verknüpfungen, ODER-Verknüpfungen und die Möglichkeit der Iteration von Aktivitäten.

Bei der Iteration wird - wie der Name schon andeutet - eine Aktivität bzw. mehrere Aktivitäten[4] wiederholt bis eine bestimmte Bedingung erfüllt ist.

Die ODER-Verknüpfung besteht aus einer Alternative (*OR-Split*, bedingte Verzweigung) und einer asynchronen Zusammenführung (*OR-Join*). Die Alternative führt dazu, dass, nachdem eine Aktivität beendet wird und bevor die nächste gestartet werden kann, eine Entscheidung gefällt wird (in der Regel automatisch durch das WfMS), welche Aufgabe aus einer in der Prozessdefinition beschriebenen Menge von Aufgaben als nächstes ausgeführt werden muss. Dabei ist der Begriff „OR-Split“ etwas irreführend, da es sich um ein exklusives Oder handelt. Es wird also genau eine Aktivität ausgewählt.

Das Gegenstück zur Alternative ist die asynchrone Zusammenführung. Hier ist der Term Zusammenführung allerdings nicht wörtlich zu nehmen, da durch die Alternative nur genau eine von mehreren möglichen Aktivitäten abgearbeitet wird und deshalb bei der Zusammenführung nur genau eine Aktivität beendet wird bevor die nachfolgende gestartet werden kann. Es findet also eigentlich eine sequentielle Abarbeitung eines möglichen Pfades im Aktivitätsnetzwerk statt. Die nachfolgende Abb. 2.5 zeigt die Darstellung einer ODER-Verknüpfung in einem Graph.

Für die parallele Ausführung von Tätigkeiten wird die UND-Verknüpfung genutzt, die aus einer parallelen Verzweigung und einer Synchronisation besteht.

Die synchrone Verzweigung stellt einen Punkt in der Prozessinstanzausführung dar, an dem die Abarbeitung von einem Abarbeitungsstrang auf mehrere übergeht. Es werden ab

[4] hier spricht man dann von einem Aktivitätsblock (*Activity Block*)

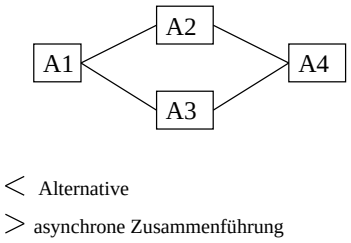

Abbildung 2.5: ODER-Verknüpfung in einem Aktivitätsnetzwerk

diesem Zeitpunkt mehrere Aktivitäten gleichzeitig ausgeführt. Diese Stränge werden unabhängig voneinander abgearbeitet und werden erst durch eine Synchronisierung wieder zu einem einzelnen Aktivitätsstrang zusammengeführt. Dabei gibt es in unterschiedlichen Workflow Management System Implementierungen verschiedene Möglichkeiten die parallelen Aktivitätsstränge wieder zusammenzuführen. Manche verlangen, dass *alle* Stränge mit *einer* Synchronisierung zusammengeführt werden müssen. Andere lassen auch eine Synchronisierung von Teilmengen zu. Zum endgültigen Abschluss eines Prozesses ist es jedoch notwendig, dass alle Stränge wieder zu einem einzelnen zusammengeführt werden. Ein Beispiel für beide Möglichkeiten ist in Abb. 2.6 zu finden.

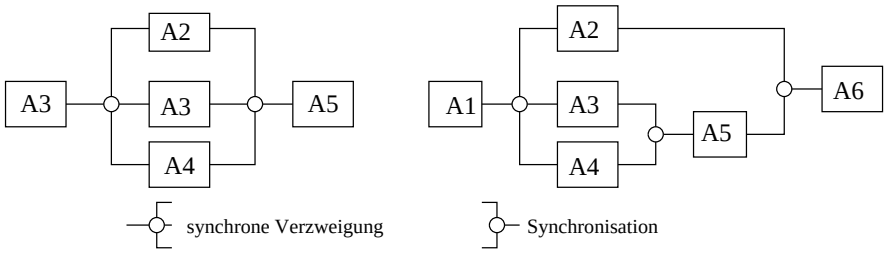

Abbildung 2.6: UND-Verknüpfung in einem Aktivitätsnetzwerk

Vor- und Nachbedingungen Vor- und Nachbedingungen sind jeweils logische Ausdrücke, die durch das Vorgangssteuerungssystem ausgewertet werden um zu entscheiden ob eine Prozess- oder Aktivitätsinstanz gestartet werden kann (Vorbedingung) oder beendet ist (Nachbedingung). Diese Ausdrücke werden in der Prozessdefinition festgelegt und können sich auf Vorgangsdaten beziehen, Systemvariablen (z. B. Datum oder Zeit) prüfen oder auch externe Ereignisse einbeziehen.

Für die Aktivitäts- und Prozessinstanzen können dabei mehrere Bedingungen definiert werden. Diese müssen dann in ihrer Gesamtheit positiv ausgewertet werden, damit die Prozessabarbeitung weitergeführt wird.

Prozessübergänge Prozessübergänge (*Transition*) bezeichnen den Übergang zwischen Aktivitäten A1 und A2, wenn nach Beendigung von A1 die Ausführung von A2 beginnt. Dabei ist eine Verknüpfung mit einer Bedingung möglich aber nicht zwingend notwendig.

Im Gegensatz zu Vor- und Nachbedingungen werden als Prozessübergangsbedingungen nur solche bezeichnet, die den Fluss der Aktivitäten eines Prozesses bestimmen. Dabei ist auch eine Kombination aus Vor- und Nachbedingungen denkbar. Die Prozessübergangsbedingungen werden von einigen WfMS-Implementierungen auch in dieser Weise umgesetzt.

Diese Bedingungen werden in der Prozessdefinition festgelegt.

2.1.5 Aktivität

Unter einer Aktivität wird eine Aufgabe bzw. ein Arbeitsschritt verstanden. Dieser kann von einer oder mehreren Ressourcen (z. B. Mensch, Computer, etc.) abgearbeitet werden und wird als kleinster Teil eines durch die Laufzeitumgebung abgearbeiteten Workflows angesehen. Es können dabei durchaus mehrere Einzelaufgaben eines Nutzers in einer Aktivität zusammengefasst sein. Es wird zwischen manuellen und automatisierten Aktivitäten unterschieden.

2.1.5.1 Manuelle und automatisierte Aktivitäten

Eine manuelle Aktivität (*manual Activity*, manueller Schritt) entspricht einer Aufgabe, die zwar Teil des Gesamtprozesses ist und für die Vollständigkeit des Geschäftsprozesses auch modelliert werden muss, aber nicht durch das WfMS abgearbeitet wird. Ein Beispiel dafür ist eine Aufgabe, in der der Nutzer Ware für eine Bestellung aus dem Lager holen muss.

Eine automatisierte Aktivität (*automated Activity*, automatischer Schritt) ist Teil eines Workflows und kann durch den Computer (respektive das Workflow Management System) abgearbeitet werden.

Beim Start einer Tätigkeit wird aus der in der Prozessbeschreibung festgelegten Definition eine Instanz erzeugt, die

- ein externes Programm ruft,
- Nutzern neue Aufgaben in die Arbeitsliste stellt, die von System selbst verwaltet werden oder
- Nutzern Aufgaben erteilt, die außerhalb des Systems zu erledigen sind, über deren Zustand das System aber Kenntnis haben will.

2.1.5.2 Aktivitätsinstanz

Aktivitätsinstanzzustand Aktivitäten, wie auch die vorher definierten Prozesse, sind mit Objekten in objektorientierten Programmiersprachen vergleichbar, die bestimmte

Attribute und Methoden besitzen. Von einer Aktivität können Aktivitätsinstanzen (*Activity Instance*) abgeleitet werden. Diese Instanzen können unabhängig voneinander durch verschiedene Ressourcen abgearbeitet werden. Eine Aktivitätsinstanz gehört zu genau einem Prozess, wobei dieser aus mehreren Instanzen von Aktivitäten bestehen kann und meist auch wird.

Mit jeder Aktivitätsinstanz ist ein bestimmter Aktivitätszustand verknüpft, der einen der folgenden Werte annehmen kann.

- **Inaktiv** eine neue Aktivitätsinstanz wurde erzeugt aber noch nicht aktiviert (das kann zum Beispiel vorkommen, wenn ihre Voraussetzungen noch nicht vollständig erfüllt sind)
- **Unterbrochen** die Instanz ist untätig und es wird keine neue Einzelaufgabe zugeteilt bis sie wieder in den Zustand „Aktiv“ versetzt wird
- **Aktiv** eine Einzelaufgabe wurde erzeugt und der Aktivitätsinstanz zur Ausführung zugeordnet
- **Abgeschlossen** die Instanz der Aufgabe wurde vollständig abgearbeitet

2.1.6 Einzelaufgabe

Eine Einzelaufgabe (*Work Item*, Elementaraufgabe, Aktion) entspricht der Arbeit, die im Rahmen einer Aktivitätsinstanz abgearbeitet werden muss. Dies geschieht in der Regel durch einen Nutzer des WfMS. Wird die Aktivitätsinstanz gänzlich durch eine gerufene Applikation abgearbeitet, so werden unter Umständen keine Einzelaufgaben erzeugt.

Beispiele für Einzelaufgaben sind die Bearbeitung von Dokumenten oder das Ausfüllen von Formularen.

Der Nutzer kann sich seine aktuell zu erledigenden Aufgaben über den Eingangskorb anzeigen lassen. Dabei nutzt er eine Eingangskorbsteuerung (*Worklist Handler*), die die Applikation des Nutzers mit dem Eingangskorb des WfMS verbindet. Über den Eingangskorb erhält er aber nicht nur die Aufgabe, sondern auch (sofern vorhanden) weitere Details, wie z.B. die Bearbeitungsfrist.

Es gibt verschiedene Möglichkeiten, wie dieser Zugriff auf den Eingangskorb geschehen kann. Die wichtigsten wurden in Abschnitt 2.1.3.2 gegeben.

2.2 Verteilte Workflows

Durch die WfMC wurden in [Mar99] Workflow Management System-Implementierungen in acht verschiedene Level der Interoperabilität eingeteilt. Diese reichen von „keinerlei Möglichkeiten der Zusammenarbeit“ bis zur vollständigen Unterstützung der Hauptfunktionalitäten mit einer einheitlichen Benutzerschnittstelle der Systeme. In Tabelle 2.1 werden diese Level aufgelistet und nachfolgend kurz erläutert.

In den folgenden Beschreibungen der einzelnen Level werden durch MS1 und MS2 zwei unterschiedliche Workflow Management System-Implementierungen bezeichnet.

Level	Bezeichnung
1	keine Interoperabilität
2	Koexistenz
3	Gateways
4	eingeschränkte Gateway Untermenge
5	komplette Workflow API
6	geteilte Definitionsformate
7	Protokollkompatibilität
8	einheitliches Aussehen und Benutzen

Tabelle 2.1: Level der Interoperabilität von WfMS

- **Level 1:** In diesem Level gibt es keine Möglichkeit der Interoperabilität zwischen Workflow Management Systemen.
- **Level 2:** MS1 und MS2 teilen sich die gleiche Ausführungsumgebung (z. B. Hardware, Betriebssystem). Es gibt keine direkte Zusammenarbeit zwischen beiden, aber es können unterschiedliche Teile eines Gesamtworkflows auf die beiden Systeme verteilt werden. In diesem Level ist eine Kommunikation von MS1 und MS2 zwar möglich, geschieht aber nicht über standardisierte Schnittstellen.
- **Level 3:** MS1 und MS2 können über einen Gateway miteinander kommunizieren. Ein Gateway ist hier als Vermittlungsanwendung zu verstehen. Dieser bildet die Funktionen und Datenbestände (sowohl Vorgangsdaten als auch Anwendungsdaten) der beteiligten Systeme aufeinander ab, sollten sie unterschiedlich sein. Nutzen mehr als zwei Systeme den Gateway, muss dieser zusätzlich noch Routingfunktionalität bieten.
- **Level 4:** Hier werden Kommunikations- und Datenaustauschfunktionen durch eine festgelegte Programmierschnittstelle (API) von den beteiligten Systemen bereitgestellt. Diese API-Funktionen können dabei direkt im WfMS implementiert sein oder es wird eine Kapselung genutzt, um die herstellerspezifischen Funktionen nach außen auf API-Funktionen abzubilden. Dabei ist es je nach verwendeter Implementierung notwendig ein neutrales Informationsformat für Daten zu spezifizieren.
- **Level 5:** Dieser Level nutzt wie Level 4 eine gemeinsame Programmierschnittstelle, wobei diese den vollen Umfang an Operationen aller WfMS implementieren muss. Davon ausgenommen sind branchenspezifische Funktionen.
- **Level 6:** WfMS, die diesen Level erreichen wollen, müssen über ein standardisiertes, gemeinsam genutztes Format zur Definition von Prozessen verfügen. Diese Definition beinhaltet Routing-Entscheidungen, Nutzerzugriffsrechte und Ressourcenverwaltung. Dieser Level ermöglicht es einer Organisation somit, *eine* Definition eines Prozesses zu erstellen und diese wird auf *allen* Systemen *gleich* ausgeführt[5].

[5] bzw. verhält sich auf allen Systemen gleich

- **Level 7:** In diesem Level muss ein Standard für alle Klient/Server Aufrufe für die Kommunikation vorliegen. Das schließt die Übermittlung von Definitionen, Workflow Transaktionen und Wiederherstellungsmechanismen mit ein.
- **Level 8:** Der speziellste Level, der alle vorherigen einschließt und zusätzlich für alle Implementierungen gleiches Aussehen und gleiche Bedienung fordert.

Neben der Einteilung in Level werden in [Hol95](Kapitel 3.7) und [Mar99] verschiedene Szenarien der Möglichkeiten, wie Workflows verteilt abgearbeitet werden können, gegeben. Die WfMC hat dabei vier Szenarien genauer definiert, die jetzt vorgestellt werden sollen. In den nachfolgenden Abbildungen stellen Quadrate Aufgaben dar. Unterschiedliche Farben bedeuten dabei, dass die Aktivitäten in Prozessen definiert sind, die innerhalb verschiedener Laufzeitumgebungen abgearbeitet werden. Wie schon in den vorhergehenden Beispielen werden, obwohl es sich um gerichtete Graphen handelt, Pfeile nur wenn unbedingt nötig eingesetzt. Dies dient einzig dem Zweck einer besseren Übersichtlichkeit. Eine Kante ohne Pfeil führt dabei immer von links nach rechts.

Szenario 1: Modell der verbundenen eigenständigen Prozesse Dieses Modell wird auch als Modell der verketteten Prozesse bezeichnet. Es stellt den einfachsten Fall der vier behandelten Szenarien dar. Prozess 1 (P1) ist hierbei durch genau eine Verbindung mit Prozess 2 (P2) verbunden. Obwohl der Verbindungspunkt in Abb. 2.7 am Ende von P1 dargestellt ist, ist dieser an jeder beliebigen Stelle möglich.

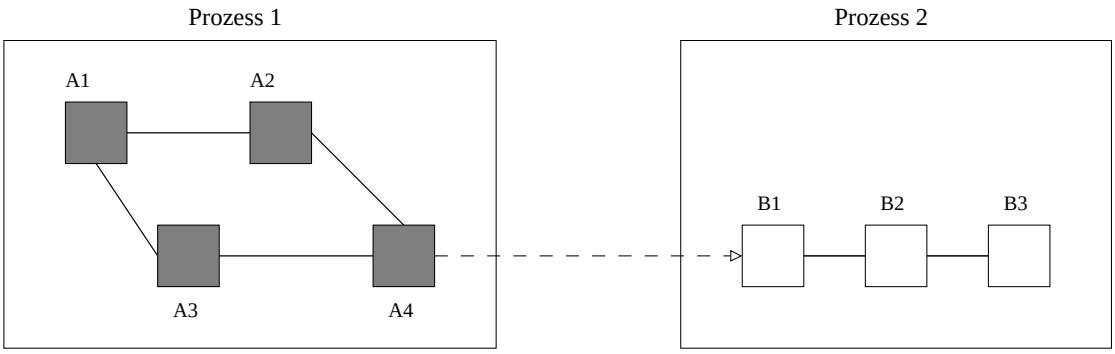

Abbildung 2.7: Modell der verketteten Prozesse

Gleichermaßen kann die Ausführung eines weiteren Prozesses P3 von P1 angestoßen werden. Nachdem P1 die Ausführung von P2 angestoßen hat (diesen also initialisiert und gestartet hat), fährt er ohne weitere Beachtung dieses Prozesses in seiner Ausführung fort.

Zu beachten ist hierbei, dass eventuell Daten von P1 an P2 übergeben werden müssen. Falls dabei verschiedene WfMS-Implementierungen zum Einsatz kommen, kann eine Abbildung der übergebenen Daten und Namen notwendig sein.

Szenario 2: Modell der hierarchischen (Unter-)Prozesse In diesem Szenario, das in Abb. 2.8 dargestellt ist, wird P2 von P1 als eine Tätigkeit angesehen. Das heißt P1 führt die Aktivität A1 lokal aus. Bei Ausführung von A2 wird entweder durch Nutzung einer Funktion eines Gateways oder direkt durch einen API-Aufruf der Laufzeitumgebung von P2 dieser Prozess initialisiert und die Abarbeitung gestartet. Dabei wird die Kontrolle abgegeben (P1 wartet auf Beendigung von A2) und P1 fährt erst mit der nächsten Aufgabe (A4) fort, wenn P2 beendet ist. Wird A3 statt A2 ausgeführt, ist kein externer Prozess beteiligt.

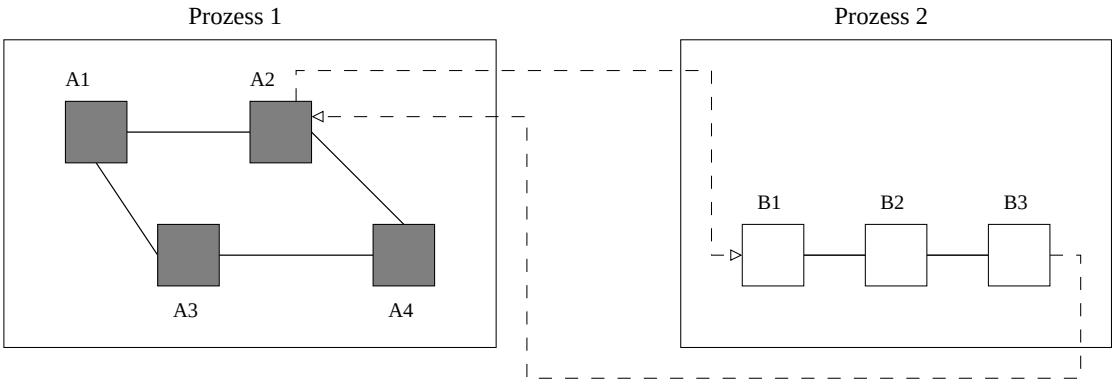

Abbildung 2.8: Modell der verschachtelten (hierarchischen) Unterprozesse

In diesem Szenario gibt es zwei Schnittstellen zwischen den Systemen, an denen Daten ausgetauscht werden müssen. Zum einen, wie schon in Szenario 1, bei Aufruf von P2 und ein zweites Mal nach Beendigung von P2. Eine rekursive Weiterführung (z. B. Aktivität B3 ist ein Unterprozess P3 von P2, der in einer weiteren Laufzeitumgebung abgearbeitet wird) ist dabei möglich und hängt von der jeweiligen Implementierung ab. Dabei könnte Prozess P3 auch wieder in der Laufzeitumgebung von P1 abgearbeitet werden oder ein drittes System nutzen.
Ein wichtiges, nicht zu unterschätzendes, Kriterium sind Prozesseigenschaften von P2, die zu einem Abbruch oder einer vorzeitigen Terminierung dieses Prozesses vor der Abarbeitung von B3 führen. In diesen Fällen müssten, obwohl dies nicht der eigentliche Rückkehrpunkt ist, die Kontrolle und relevante Daten an P1 zurückgegeben werden.

Szenario 3: Modell der gemeinsam genutzten Domäne Unter einer Domäne soll eine Art geschäftliches Abkommen zur Zusammenarbeit verschiedener Vorgangssteuerungssysteme verstanden werden. Das heißt durch die Domäne werden verschiedene Systeme zu einem Gesamtsystem mit einem festgelegten Kontext vereinigt. In diesem Szenario wird die Regelung für Verbindungspunkte weiter aufgeweicht. Ein Prozess 3 besteht dabei, wie in Abb. 2.9 gezeigt, aus Aktivitäten von P1 und P2, wobei diese in beliebiger Reihenfolge ablaufen können.
Wird nach Ai eine Tätigkeit Aj (i<j) ausgeführt, so kann dies in derselben Laufzeitumgebung geschehen oder die Kontrolle muss an eine andere abgegeben werden. Im ersten

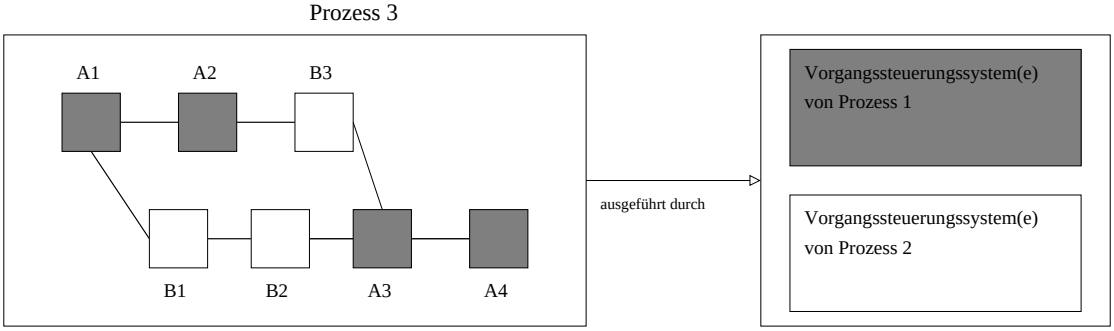

Abbildung 2.9: Modell der gemeinsam genutzten Domäne

Fall ist dies mit der lokalen Abarbeitungsfolge gleichzusetzen. Beim zweiten Fall müssen bei jedem Wechsel Daten und zugehörige Namen übergeben werden. Die Daten können dabei sowohl prozessbezogen sein als auch einer externen Anwendung zugehörig.

Die verschiedenen Laufzeitumgebungen[6] müssen in diesem Szenario eine gemeinsame Prozessdefinition besitzen, die entweder bei Start des Prozesses vorliegt oder zur Laufzeit ausgetauscht wird. Besondere Aufmerksamkeit verlangen in diesem Modell Administrationsaufgaben, Sicherheitsmaßnahmen und Wiederherstellungsmechanismen bei Fehlern.

Szenario 4: Modell der parallelen Synchronisation Behandelten die bisherigen Szenarien jeweils nur die Möglichkeiten der verteilten Abarbeitung von Aufgaben, ohne diese (bis auf eine festgelegte Reihenfolge der Aufgabenabarbeitung) steuern zu können, so wird in diesem Modell auf spezielle Synchronisationsregeln eingegangen.

Durch Synchronisationspunkte zwischen Aktivitäten von Prozessen verschiedener Laufzeitumgebungen lassen sich von Abarbeitungsregeln unabhängige Prozesse miteinander verknüpfen. Dies geschieht durch sogenannte Ereignisse. Sinnvoll ist dies z. B. wenn ein Scheduling von Prozessteilen, die durch eine synchrone Verzweigung entstanden sind, geplant ist, um Kontrollpunkte für die Erzeugung von Wiederherstellungsdaten zu setzen oder Vorgangsdaten zwischen Instanzen eines Prozesses auszutauschen.

Obwohl dieses Szenario durch die WfMC spezifiziert wurde, sind derzeit keine Bemühungen für Standards in dieser Richtung vorgesehen. Ein Beispiel dieser Art der verteilten Workflowabarbeitung ist in Abb. 2.10 gegeben.

Neben diesen Szenarien gibt [Mar99] noch Beispiele, welche Funktionalitäten die einzelnen Systeme für die Szenarien 1 und 2 bereitstellen sollten und wie der grundlegende Ablauf der Kommunikation ablaufen sollte.

Ein solches Beispiel soll abschließend vorgestellt werden. In der Folge wird als WfMS1 dasjenige bezeichnet, auf dem P1 abgearbeitet wird. Analog wird P2 auf WfMS2 abgearbeitet. Die Kommunikation für die Implementierung von Szenario 1 lässt sich in drei Teile

[6] es können mehr als die zwei gegebenen sein

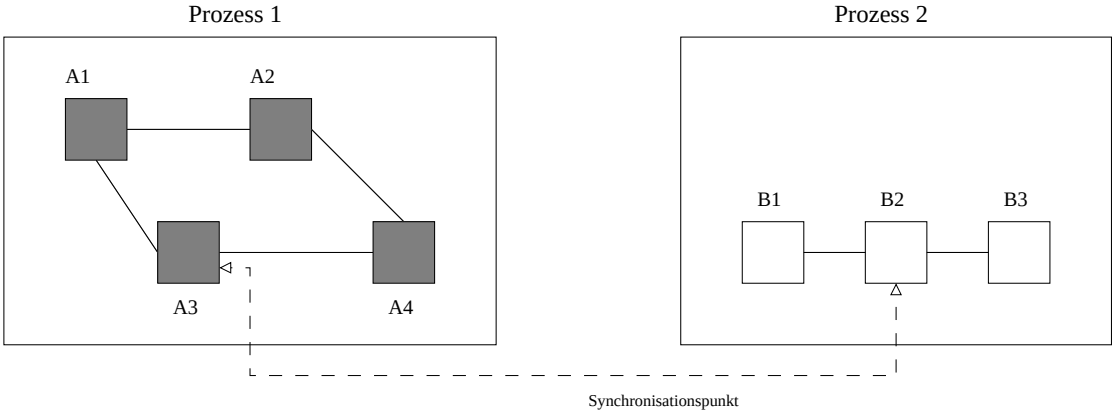

Abbildung 2.10: Modell der parallel synchronisierten Prozesse

aufschlüsseln. Als erstes muss eine Prozessdefinition von WfMS2 ausgewählt werden. Dabei kann diese (a) von WfMS1 an WfMS2 übergeben werden, (b) bei WfMS2 vorliegen oder (c) in einem von WfMS1 und WfMS2 genutzten zentralen Speicher abgelegt sein. Nachdem die Prozessdefinition ausgewählt wurde, müssen dieser alle prozessbezogenen Daten und die Speicherstellen der Anwendungsdaten mitgeteilt werden. Ist dieser Schritt ausgeführt, sollten alle Bedingungen erfüllt sein, um eine Instanz dieser Prozessdefinition zu erzeugen und diese auszuführen. Der Start der Instanz ist somit auch der letzte Schritt.

2.3 Derzeitige Bemühungen verteilte Workflows abzubilden

2.3.1 Von SWAP zu Wf-XML

In Verbindung mit der Implementierung von Schnittstelle 4 des Workflow Referenz Modells wird bei Standards häufig der Begriff „asynchroner Web Service“ genutzt. Ein asynchroner Web Service ist ein Dienst, der Funktionen bereitstellt, die das Ergebnis der zu erbringenden Aufgabe nicht sofort zurückliefern. Dies liegt darin begründet, dass vor allem bei der Abarbeitung von Workflows die Funktion Aufgaben enthält, die in den meisten Fällen nicht sofort abgearbeitet werden können. Dazu stellt im Regelfall der Rufende einen Dienst („Beoachter“) bereit, der auf das Ergebnis wartet und übergibt bei Aufruf der Funktion die Adresse dieses Dienstes. Sobald die Aufgaben der gerufenen Funktion vollständig abgearbeitet sind (was über einen längeren Zeitraum geschehen kann) wird dieser „Beobachter“ darüber informiert.

Die folgenden Beschreibungen von Konzepten und Standards geben einen Überblick über die Entwicklungsgeschichte solcher asynchroner Web Services. Dabei werden in den Beschreibungen keine Implementierungsdetails angegeben, sondern nur Funktionalitäten, die von Implementierungen dieser Standards erfüllt werden sollten.

2.3.1.1 SWAP

Das „Simple Workflow Access Protocol“ (SWAP) ist das älteste der hier vorgestellten Protokolle für asynchrone Web Services. Es wurde erstmals im August 1998 als IETF[7] Draft[8] vorgestellt. Es ist bisher aber nicht über den Konzeptstatus hinausgekommen. Es soll vor allem als Vorreiter der nachfolgenden Standards bzw. Standardisierungsvorschläge betrachtet werden.

Spezifiziert wurden in diesem Protokoll hauptsächlich zwei wichtige Ziele von asynchronen Web Services. Das sind zum einen Kontrolle und zum anderen Überwachung von „Arbeit“[9]. Zur Kontrolle zählen dabei Funktionen zum Erzeugen, Starten und Stoppen der „Arbeit“, sowie zum Setzen der Eingangsdaten und zur Übermittlung von Informationen über Ausnahmen, Beendigung und die Ergebnisse dieser „Arbeit“. Überwachungsfunktionen sind das Abrufen des aktuellen Status der „Arbeit“ und das Abfragen von Informationen über den bisherigen Ablauf.

Des Weiteren wird in diesem Konzept schon die Verwendung von XML[10] zur Datenbeschreibung und dem HTTP-Protokoll[11] als Transportprotokoll vorgeschlagen. Wie bei den folgenden Weiterentwicklungen zu erkennen ist, wird dies auch bis in die aktuellen Standards beibehalten (sofern man das „Simple Object Access Protocol“ (SOAP) auf HTTP-Basis nutzt).

2.3.1.2 ASAP/AWSP

Das „Asynchronous Service Access Protocol“ (ASAP) wurde von der „Organization for the Advancement of Structured Information Standards“ (OASIS)[12] als Standard für die Steuerung von asynchronen Instanzen bestimmter Dienste in [SRK04] definiert. Es soll dabei hauptsächlich für extern gestartete Dienste genutzt werden, die eine lange Ausführungszeit besitzen. Dabei wird als Übertragungsprotokoll SOAP und für die Nachrichtendarstellung XML spezifiziert. Der ASAP Standard wurde inzwischen in „Asynchronous Webservices Protocol“ (AWSP) umbenannt.

Als Dienstinstanzen werden Web Services bezeichnet, die nach den einzelnen nutzbaren Methoden verschiedenen Ressourcentypen zugeordnet werden. Es werden genau drei Ressourcentypen beschrieben. Diese sind:

- **Betriebsobjekt** Ein Betriebsobjekt (*Factory Resource*, Betriebsressource) stellt eine Aufgabenbeschreibung dar, wobei die Art dieser Arbeit nicht genauer spezifiziert wird. Ein Klient kann das Betriebsobjekt anweisen, die in der Aufgabenbeschreibung definierte Tätigkeit auszuführen. Wenn dies geschieht, wird aus der Definition eine spezielle Instanz erzeugt, der, sofern angegeben, Kontextdaten durch

[7] Internet Engineering Task Force
[8] entspricht Konzept
[9] bedeutet in diesem Zusammenhang ein Objekt, das eine bestimmte Aufgabe erledigt
[10] http://www.w3.org/TR/REC-xml
[11] http://www.ietf.org/rfc/rfc2616.txt
[12] http://www.oasis-open.org/

den Klienten übergeben werden. Von einem Betriebsobjekt können beliebig viele Instanzobjekte erzeugt werden. Diese erhalten einen eindeutigen Schlüssel[13] über den der Klient nach der Erzeugung mit der neuen Instanz kommunizieren kann.

- **Instanzobjekt** Ein Instanzobjekt (*Instance Resource*, Instanzressource, Instanz) stellt ein in Arbeit befindliches Betriebsobjekt dar. Jede Anfrage zur Abarbeitung der durch die Betriebsressource definierten Arbeit wird dabei zu einer eigenen Instanz. Das Instanzobjekt bietet Methoden zum Abfragen des aktuellen Zustands, Setzen und Abfragen von Eigenschaften, usw. an.
- **Beobachterobjekt** Über das Beobachterobjekt (*Observer Resource*, Beobachterressource) können Informationen über die abgearbeiteten Instanzobjekte abgefragt werden. Außerdem wird es durch die Instanzobjekte über Ereignisse, die bei der Ausführung der einzelnen Instanzen aufgetreten sind, informiert. Diese sind z. B. Beendigung und Abbruch der Ausführung einer Instanz.

Die einzelnen Ressourcen können in einem Web Service oder als eigenständige Web Services implementiert sein.
Eine grafische Darstellung der Anordnung dieser Ressourcen bzw. Dienste ist in Abb. 2.11 zu finden. Die in der Grafik grün dargestellten Teile werden nicht durch ASAP definiert. Sie werden in der später beschriebenen Erweiterung des ASAP-Standards, Wf-XML, hinzugefügt. Diese sind für die jetzige Betrachtung irrelevant.
Ein Beobachterobjekt[14] kann mittels einer Anfrage an den asynchronen Web Service, der ein Betriebsobjekt bereitstellt, die Erzeugung einer Instanz dieses Objekts anfordern. Dazu sendet er alle für die Erzeugung der Instanz erforderlichen Daten sowie seinen URI in der Anfrage. Als Antwort erhält er die URI des neu erzeugten Instanzobjekts. Nachdem die Instanzressource erzeugt ist und ausgeführt wird, kann diese dem Beobachter Informationen (z. B. über Zustandswechsel und Ausnahmen) übermitteln. Dies ist möglich, da die Instanz dessen URI kennt. Das Beobachterobjekt seinerseits kann Anfragen der Art „Wer bearbeitet dich gerade“, „Wie lange wirst du voraussichtlich noch für die Abarbeitung benötigen“, usw. an die Instanz stellen. Nach der vollständigen Abarbeitung des Instanzobjekts sendet dieses dem Beobachterobjekt die Meldung, dass es korrekt beendet wurde und das Ergebnis. Im Falle eines vorzeitigen Abbruchs wird dieser dem Beobachterobjekt auch mitgeteilt.
Jede der eben genannten Anfrageoperationen besitzt eine zugehörige Antwort. Diese kann für einzelne Funktionen jedoch in einer Anfrage abgeschaltet werden. Die Antwort ist dabei nicht die Bestätigung der Anfrage, die sofort gegeben wird, sondern die Beantwortung der gestellten Aufgabe.
Die genauen Details der einzelnen Operationen[15] werden im Standard beschrieben und sollen an dieser Stelle nicht betrachtet werden. In Abschnitt 3.2 wird noch einmal genauer auf Teile des Standards eingegangen. Weiterhin können Einzelheiten in der angegebenen Literatur gefunden werden.

[13] entspricht einem Uniform Resource Identifier (URI)
[14] entspricht einer Implementierung der Beobachterressource als Web Service
[15] Parameter, Format der SOAP-Nachrichten

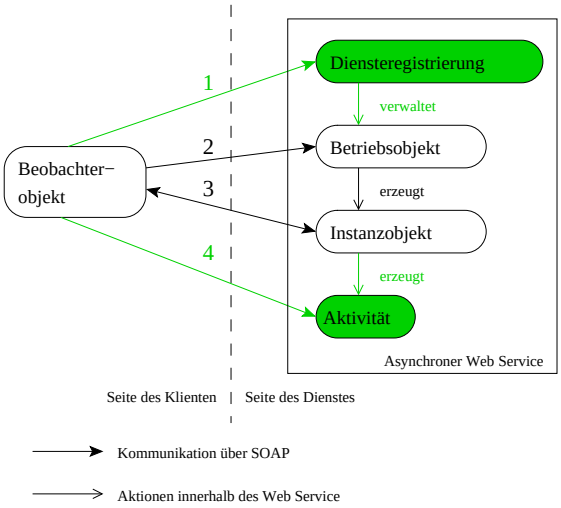

Abbildung 2.11: Modell eines asynchronen Web Service

2.3.1.3 Wf-XML

Die „Workflow-Extensible Markup Language“ (Wf-XML) ist eine Erweiterung von ASAP mit speziellen Anpassungen zur verteilten Workflowabarbeitung. Definiert wird dieses Protokoll in [SGP04].

Die Spezifikation übernimmt die von ASAP definierten Ressourcen (Betriebsobjekt, Instanzobjekt und Beobachterobjekt) und definiert zu diesen noch zwei weitere. Dies sind zum einen die Diensteregistrierung (*Service Registry*) und zum anderen die Aktivitätsressource (*Activity Resource*). Ein Betriebsobjekt wird in diesem Standard mit einer Prozessdefinition gleichgesetzt und eine Instanz mit einer Prozessinstanz.

Die Dienstregistrierung besitzt Methoden, um Prozessdefinitionen abzurufen sowie neue hinzuzufügen und bestehende zu ändern. Diese Prozessdefinitionen können dann durch entsprechende Betriebsobjekte genutzt werden.

Aktivitätsressourcen stellen die Aktivitäten innerhalb der Prozessinstanz dar. Der Beobachter kann deren aktuelle Eigenschaften abfragen („Wer ist dein Bearbeiter“, „Wie lange wird auf die Ausführung gewartet“, usw.) und sie beenden.

Die neuen Ressourcen sind in der schon für die Beschreibung von ASAP genutzten Abb. 2.11 durch grüne Darstellung hervorgehoben.

In diesem erweiterten Szenario fragt das Beobachterobjekt die Diensteregistrierung nach den vorhandenen Diensten und deren aktuellen Versionen. Nachdem es sich für eine der in der Antwort gegebenen Betriebsressourcen entschieden hat, erfolgt die Abarbeitung analog zum Beispiel für ASAP. Einziger Unterschied ist, dass das Beobachterobjekt jetzt noch die Möglichkeit hat, Informationen zu den einzelnen, zum Zeitpunkt der Abfrage gerade bearbeiteten, Aktivitäten einer Prozessinstanz abzufragen.

2.3.2 OMG jFlow

Im Gegensatz zu den eben vorgestellten Standards zur Abarbeitung verteilter Workflows, setzt das von der OMG in [Obj98] standardisierte jFlow bei der Kommunikation auf den eigenen „Common Object Request Broker Architecture“ (CORBA)-Standard. Die definierten Schnittstellen sind dabei sehr generell gehalten um eine große Anzahl von Workflow Management Systemen zu unterstützen. Zusätzlich zu den Schnittstellen werden im Standard ihre Beziehungen und Abhängigkeiten beschrieben. Im Folgenden werden die Hauptschnittstellen vorgestellt und ein Beispiel für eine mögliche Nutzung zur verteilten Workflowabarbeitung gegeben.

`WfBaseExecutionObject` Das `WfBaseExecutionObject` ist kein implementiertes Interface sondern die abstrakte Basisschnittstelle, die durch `WfProcess` und `WfActivity` implementiert wird. Bereitgestellte Funktionen sind z. B. das Auslesen des aktuellen Zustands, das Setzen des Zustands und die Manipulation des Kontexts (der prozessbezogenen Daten).

`WfRequester` Der `Wfrequester` ist die Schnittstelle, die einen Anfragenden mit einem oder mehreren Prozessen verknüpft. Durch ihn sollen Möglichkeiten gegeben werden, den Anfragenden über wichtige Zustandsänderungen und die Beendigung des Prozesses zu informieren. Der `WfRequester` ist in den meisten Fällen die Komponente, die den Prozess startet, die prozessbezogenen Daten initialisiert und das Ergebnis empfängt. Der Anfragende kann dabei eine Aktivität `WfActivity` (Schachtelung von Prozessen) oder eine Anwendung bzw. Rolle (also ein Nutzer des Laufzeitsystems) sein.

`WfProcessMgr` Diese Schnittstelle stellt die Vorlage eines bestimmten Prozesses dar. Durch sie können Instanzen des entsprechenden Prozesses erzeugt und Informationen über die erwarteten Eingabedaten und das Ergebnis abgerufen werden. Jeder `WfProcessMgr` besitzt einen in dem entsprechenden Geschäftsbereich eindeutigen Namen und kann z. B. über den OMG Namensdienst gefunden werden.

`WfProcess` Ein `WfProcess` stellt die auszuführende Arbeit dar. Er entspricht also dem durch die WfMC definierten Prozess. Der Prozess wird von `WfExecutionObject` abgeleitet und besitzt, zusätzlich zu den durch diese abstrakte Schnittstelle definierten, Methoden zum Erzeugen einer neuen Instanz des Prozesses und Abfragen des Ergebnisses.

`WfActivity` Durch die `WfActivity` werden Teilschritte eines Prozesses beschrieben. Es ist die Anfrage für zu erledigende Aufgaben innerhalb des Prozesses. Durch Hinzufügen einer Operation zum Abschließen der Aufgabe wird das `WfExecutionObject` erweitert.

`WfResource` Eine `WfResource` ist ein abstraktes Objekt. Es repräsentiert eine Person oder ein „Ding“ (laut Standard), das Anfragen einer `WfActivity` zur Abarbeitung von Aufgaben entgegennimmt. Dabei wird in der angegebenen Literatur nicht näher auf die Implementierung diese Objekts eingegangen.

`WfAssignment` Ein `WfAssignment` verknüpft Anforderungen von `WfResources` durch eine `WfActivity` mit möglichen oder tatsächlich vorhandenen Ressourcen. Dabei ist die Lebenszeit eines `WfAssignments` an die der zugehörigen `WfActivity` geknüpft, d. h. wird die Aktivität beendet, so werden auch alle `WfAssignments` dieser zerstört.

Über diese Schnittstellen können verschiedene Systeme, die sie implementieren, miteinander kommunizieren. Dabei ist es ohne Bedeutung, ob es sich um ein System handelt oder mehrere über ein Netzwerk verteilte Systeme in die Abarbeitung involviert sind. Ein mögliches Beispiel für die Abarbeitung eines Prozesses durch ein entferntes Laufzeitsystem wird in Abbildung 2.12 vorgestellt.

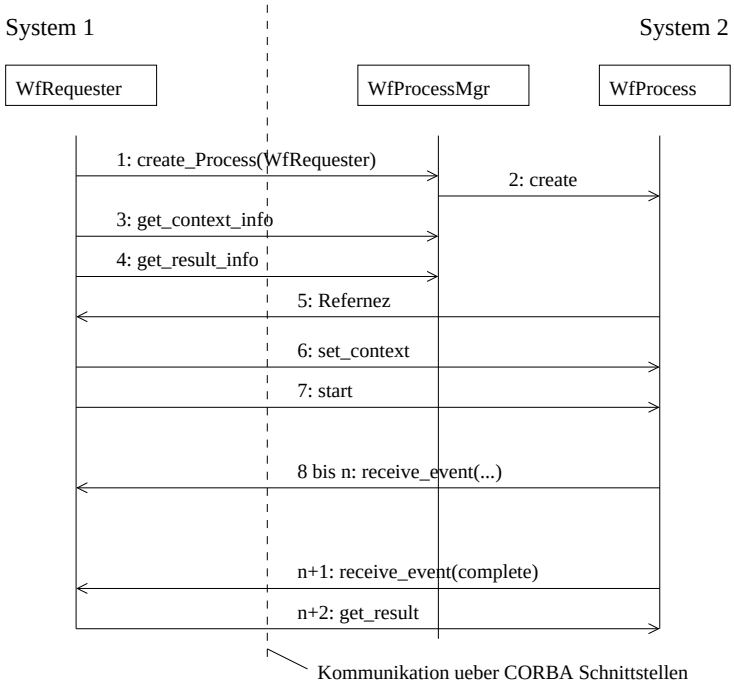

Abbildung 2.12: Nutzungsbeispiel der Schnittstellen

Der `WfRequester` ist in diesem Fall z. B. eine Aktivität innerhalb eines Prozesses in der Laufzeitumgebung auf System 1. Durch die Anfrage an einen `WfProcessMgr` auf System 2 wird die Erzeugung des entfernten Prozesses angestoßen (Schritt 1). Diese Anfrage nutzt die CORBA-Schnittstelle des `WfProcessMgr`. Er erzeugt daraufhin den Prozess in seiner Laufzeitumgebung (Schritt 2).

Danach fragt der `WfRequester` Informationen über den Kontext und das Ergebnis des entfernten Prozesses ab (Schritte 3 und 4). Nachdem der Prozess in der Laufzeitumgebung von System 2 erzeugt wurde, gibt der `WfProcessMgr` die Referenz auf den erzeugten Prozess zurück (Schritt 5). Dieser wird vom `WfRequester` benötigt, um die prozessbezogenen Daten des Prozesses zu initialisieren (Schritt 6) und den Prozess zu starten (Schritt 7).

Nachdem der Prozess gestartet wurde erzeugt er Aktivitäten, verknüpft diese mit `WfResources` und steuert die Abarbeitung der repräsentierten Arbeit. Dabei werden wichtige Zustandsänderungen an den `WfRequester` übermittelt (Schritte 8 bis n).

Nachdem alle Aufgaben des Prozesses abgearbeitet wurden, signalisiert dieser dem Anfragenden seine Beendigung (Schritt n+1). Das anfragende Objekt liest daraufhin das Ergebnis des Prozesses (Schritt n+2). Ist dieser Schritt beendet, kann die als `WfRequester` genutzte Aktivität auf System 1 beendet und die Abarbeitung des rufenden Prozesses fortgesetzt werden.

2.4 Einordnung ausgewählter freier Workflow Management Systeme

2.4.1 Enhydra Shark

Enhydra Shark ist ein durch das Enhydra.org Projekt entwickeltes Workflow Management System. Es wird durch das Objectweb Consortium unterstützt und stellt seine Softwarelösungen frei als Open Source unter der LGPL zur Verfügung. Bei diesem WfMS ist eine strikte Einhaltung der durch das WfMC vorgegebenen Standards gegeben. Als Prozessdefinitionssprache wird XPDL verwendet, das ebenfalls durch die WfMC standardisiert wurde. Zum Erstellen der XPDL Beschreibungen wird zusätzlich das Tool *JaWE* (Java Workflow Editor)[16] von Enhydra.org als grafische Oberfläche zur Generierung der Prozessbeschreibungen angeboten. Die mittels JaWE erstellten Prozessdefinitionen können dabei nicht nur in Enhydra Shark, sondern bspw. auch von Open Business Engine (OBE)[17] oder WfMOpen[18] genutzt werden.

Enhydra Shark ist ein stark komponentenbasiertes Workflow Management System. Die einzelnen funktionalen Komponenten werden separat definiert. Es gibt z. B. Komponenten für den Kern des WfMS (stellt Funktionen für Prozesse, Aktivitäten, den Zugriff auf das System, usw. bereit), für die Datenspeicherung, für die Nutzerverwaltung, für Sicherheitsmechanismen, usw.. Durch die Möglichkeit des Austausches der einzelnen Komponenten des Systems ist es möglich, dieses an spezielle Bedingungen und Nutzungsszenarien anzupassen. So können z. B. die Cache Komponente oder die Komponente zum persistenten Speichern der Daten durch eigene Implementierungen ersetzt werden. Es ist sogar möglich den Kern der Shark-Laufzeitumgebung durch eine eigene Implementierung zu ersetzen.

Enhydra Shark speichert seine Prozesse und Aktivitäten in einer relationalen Datenbank. Es bietet die Möglichkeit über JDBC-Treiber eine Reihe von Datenbanken Management Systemen zu nutzen. Zu nennen wären hier HSQLDb (eine in Java implementierte Speicherdatenbank), mySQL, Oracle, PostgreSQL, usw..

[16] zu finden unter http://jawe.objectweb.org
[17] zu finden unter http://www.openbusinessengine.org/
[18] zu finden unter http://wfmopen.sourceforge.net/

Für die Ausführung von automatisierten Aktivitäten wird die WfMC „Tool Agents API“ (Schnittstelle 3 des Workflow Referenzmodells) bereitgestellt. Es sind vordefinierte Anwendungsagenten z. B. für Java Klassen, JavaScript, E-Mail und Webservice-Aufrufe in Enhydra Shark enthalten.

Enhydra Shark kann sowohl als Bibliothek in bestehende Projekte eingebunden als auch als eigenständiger Dienst ausgeführt werden. Es stehen CORBA-Schnittstellen zur entfernten Nutzung als Server zur Verfügung.

Wie bereits erwähnt, nutzt Shark als Prozessbeschreibungssprache XPDL. In den folgenden Absätzen soll eine kurze Einführung in diese XML-basierte Sprache erfolgen. Die Ausführungen beziehen sich auf [Nor02].

Eine XPDL-Beschreibung besteht aus einem Paket („`Package`“), in dem ein oder mehrere Prozesse definiert sind. Diese Prozesse bestehen aus Aktivitäten und Übergangsregeln. Jedes Element besitzt die grundlegenden Attribute Identifikator („`Id`“), Name („`Name`“), Beschreibung („`Description`“) und erweiterte Attribute („`Extended Attributes`“).

Ein Paket definiert alle für die beinhalteten Prozesse gültigen Teilnehmer („`Participant`“), Anwendungen („`Application`“) und sachbezogenen Daten („`Workflow Relevant Data`“). Dabei wird im Standard vorgeschlagen, alle für einen Workflow benötigten Prozesse in einem Paket zu definieren. Durch die Einbindung externer Pakete ist es möglich, für mehrere Workflows benötigte Anwendungen und Teilnehmer *einmal* zu definieren und in allen diesen Workflows zu nutzen.

Eine Prozessbeschreibung kann alle im Paket definierten Elemente nutzen. In ihr werden Aktivitäten definiert, die durch Übergangsregeln verbunden sind.

Als Aktivität eines Prozesses können die Elemente atomare Aktivität („`Atomic Activity`“), Unterprozess („`SubFlow`“) und Aktivitätsblock („`Block Activity`“) genutzt werden.

2.4.2 jBpm

jBPM ist aus dem Projekt „Java Business Process Management“ hervorgegangen und wird ab Version 2.0 durch die JBoss Company weiterentwickelt. Es wird als flexibles, erweiterbares WfMS bezeichnet und nutzt als Prozessbeschreibungssprache jPdl (jBPM Process definition language). Die folgenden Betrachtungen beziehen sich auf die Dokumente [Koe04] für jBPM und [jPD] für jPdl.

JBoss jPBM ist als eigenständige Software nutzbar, wird aber als fundamentaler Bestandteil der „JBoss Middleware Plattform“ entwickelt. Der Kern von JBPM ist eine sogenannte „State Engine“. Darunter ist eine Softwarekomponente zu verstehen, die den Workflow auf der Basis von den Zuständen der Aktivitäten abarbeitet. Der JBoss jBPM Server verwaltet dazu die Zustände der Prozesse, protokolliert den Verlauf der Abarbeitung und führt automatisierte Aktivitäten aus.

Die Ausführung von Prozessen basiert auf vier Teilaufgaben, die durch drei Komponenten verwaltet werden. Die erste Komponente ist der Definitionsdienst[19]. Mittels dieser

[19] Definition Service

können durch einen oder mehrere Entwickler erstellte Prozessdefinitionen eingelesen werden (Teilaufgabe 1). Die zweite Komponente ist der Ausführungsdienst[20]. Diese ermöglicht es Anwendern oder dem System, Aufgaben zu be- bzw. verarbeiten (Teilaufgabe 2) und externe Anwendungen auszuführen (Teilaufgabe 3). Die letzte Komponente ist der Administrationsdienst[21]. Diese Komponente wird genutzt, um Statistiken und Ausführungsprotokolle zu schreiben und somit dem Administrator Informationen über den Verlauf der Abarbeitung zu liefern.

JBoss jBPM verfolgt zwei Grundprinzipien. Zum einen soll es neuen Nutzern des Systems möglich sein, ohne viel Aufwand Prozessdefinitionen zu entwickeln und auszuführen und zum anderen sollen auch komplizierte Workflows beschreib- und abarbeitbar sein.

Wie am Anfang dieses Abschnitts erwähnt, verwendet jBPM zur Beschreibung von Workflows ein eigenes Format. Dieses soll leicht erlernbar sein und dem Nutzer die Möglichkeit bieten, schnell und einfach Prozessdefinitionen zu entwickeln. jPdl ist eine XML-basierte Sprache mit der sich Zustände, Verknüpfungen von Aufgaben mit Nutzern, Variablen und Variablentypen sowie Verzweigungen von Abarbeitungssträngen und zugehörige Zusammenführungen beschreiben lassen. Um einen Prozess für die Abarbeitung durch jBPM gänzlich zu beschreiben, wird ein sogenanntes Prozessarchiv erstellt. Dieses besteht aus den folgenden drei Teilen:

- einer deklarativen Beschreibung des Geschäftsprozesses,
- der Programmierlogik und
- anderen Ressourcen.

Die deklarative Beschreibung entspricht dabei der jPdl-Beschreibung des Prozesses. Durch die Programmierlogik werden dem Archiv zusätzliche Java Klassen, die zur Abarbeitung notwendig sind, hinzugefügt. Die als „andere Ressourcen“ bezeichneten Hilfsmittel stellen Hilfen für den Nutzer dar, bspw. Beschreibungen von auszufüllenden Formularen u. ä..

2.4.3 OpenWFE

OpenWFE (Open source Workflow Engine) wird nicht nur als Vorgangssteuerungssystem, sondern vielmehr als Geschäftsprozess-Steuerungs-System bezeichnet. Die folgenden Ausführungen beziehen sich auf [Met] und [Ope].

OpenWFE besteht aus fünf Komponenten.

Als erstes ist hier die sogenannte „Engine“ zu erwähnen, die den Hauptbestandteil des Systems darstellt und hinter der sich nichts anderes als die Vorgangssteuerung verbirgt. Sie dient dem Einlesen von Prozessbeschreibungen, interpretiert diese und führt sie aus. Für die Beschreibungen der Prozesse wird dabei ein eigenes proprietäres Format genutzt. Die Vorgangssteuerung besteht aus zwei Komponenten. Dies ist zum einen die

[20] Execution Service
[21] Administration Service

Teilnehmer-Aufgaben-Zuordnung mit Informationen, wie Aufgaben den einzelnen Teilnehmern zugewiesen werden, und zum anderen ein Empfänger („Listener“), der auf gesendete Aufgaben von Teilnehmern wartet.

Die zweite Komponente ist die Arbeitsliste („Worklist“). Diese speichert die Aufgaben der einzelnen, am Workflow beteiligten, Nutzer des Systems. Außerdem bietet sie Möglichkeiten zum Abrufen der Aufgaben. Für jeden Teilnehmer wird ein eigener Aufgabenspeicher bereitgestellt. Dieser kann entweder das Dateisystem oder ein Datenbank Management System (z. B. mySQL oder PostgreSQL) sein.

Als dritte Komponente ist die „Automatic Participants Runtime Environment“ (APRE) zu erwähnen. Sie dient dem automatisierten Abarbeiten von Aktivitäten. Das können sowohl Java Klassen, als auch Python Skripte (unter Nutzung von „ jython“[22]) sein.

Die vierte Komponente ist ein Web Klient, mittels dessen man die Arbeitsliste von OpenWFE durch einen Web Browser nutzen kann. Dabei kann entweder „Remote Method Invocation“ (RMI) oder das sogenannte „REpresentational State Transfer“-Protokoll (REST) zum Zugriff auf die Aufgabenliste genutzt werden. Bei REST handelt es sich um eine zustandslose Klient/Server-Architektur auf der Basis von HTTP-Get und HTTP-POST als entfernte Methoden und URIs als Identifikatoren der Ressourcen.

Die letzte Komponente wird in [Ope] nicht erwähnt, da sie laut [Met] erst neu in die Dokumentation aufgenommen wurde. Sie ist das sogenannte „Draw Flow“ (Droflo) mit dem man in einem Web Browser Prozessbeschreibungen erzeugen kann. Diese Beschreibungen werden als XML Dateien gespeichert und durch ein eigenes Format definiert. Dieses basiert auf Workflow Mustern („Workflow Patterns“).

2.4.4 WfMOpen

WfMOpen ist ein J2EE-basiertes Workflow Management System. Es implementiert, wie Enhydra Shark, die WfMC- und OMG-Spezifikationen. Als Prozessbeschreibungssprache wird XPDL mit speziellen Erweiterungen genutzt. Die Erweiterungen werden dabei in den erweiterten Attributen der einzelnen Beschreibungselemente definiert und stellen somit keine Änderung der XPDL-Spezifikation dar. Sie betreffen Einstellungen zur Entfernung bei Abschluss eines Prozesses, Abarbeitung von Prozessen im Debug-Modus und die Aufschiebung der Auswahl der nächsten Aktivität bei Verzweigungen.

Mit der Entfernung von Prozessen bei Abschluss kann bestimmt werden, ob ein Prozess bei seiner Beendigung sofort gelöscht werden oder bis zu einer manuellen Entfernung durch den Administrator in der Laufzeitumgebung verbleiben soll. Bei der Aufschiebung der Auswahl der nächsten Aktivität werden, wie bei einer synchronen Verzweigung zunächst mehrere Aufgaben in den Arbeitskorb gestellt. Die Auswahl des Pfades der Abarbeitung wird durch den Nutzer bestimmt. Nimmt er eine der möglichen Aktivitäten an, werden die anderen verworfen und aus dem Arbeitskorb entfernt.

Die Kernkomponente von WfMOpen ist die Vorgangssteuerung, bestehend aus Enterprise Java Beans und Java Messaging Service-Warteschlangen. Eine webbasierte Ad-

[22] http://www.jython.org/

ministrationsanwendung liegt dem Installationspaket bei. Diese benötigt einen J2EE-Applikationsserver, z. B. JBoss.

Die Ausführungen zu WfMOpen beziehen sich auf [LSW].

Zusammenfassung

In diesem Abschnitt wurden verschiedene freie WfMS beschrieben. Es wurde dabei sowohl auf die Systeme als auch auf die Möglichkeiten der Prozessbeschreibungen eingegangen. Anhand dieser kurzen Einführung soll nun die Entscheidung für Enhydra Shark als WfMS für die zu entwickelnde Schnittstelle dargelegt werden. Eine abschließende Übersicht der Systeme ist in Tabelle 2.2 zu finden.

Eigenschaft	Enhydra Shark (1.0)	jBPM	OpenWFE	WfMOpen
Prozess-beschreibungs-sprache	XPDL	JPDL	eigenes proprietäres Format	XPDL mit Erweiterungen
WfMC konform	ja	nein	nein	ja
Implementierte WfMC-Schnittstellen	1,2,3,5	-	-	1,2,3,5
Administrations-anwendung	ja	ja	ja	ja
verteilte Prozessab-arbeitung möglich	nein	nein	nein	nein

Tabelle 2.2: Eigenschaften der betrachteten WfMS

Durch die Betrachtungen und die Tabelle wurde dargelegt, dass von den betrachteten Systemen nur Enhydra Shark und WfMOpen die Schnittstellen der WfMC implementiert. Keines der betrachteten Systeme implementiert Möglichkeiten für eine verteilte Workflowabarbeitung. Eine Administrationsanwendung bzw. Klientenapplikation bieten alle betrachteten Systeme. Allerdings wird bei jBPM und WfMOpen ein J2EE-Applikationsserver zur Nutzung dieser Anwendung benötigt.

Enhydra Shark wurde als WfMS für die zu implementierende Schnittstelle wurde aus drei Gründen gewählt. Zum Ersten implementiert es die spezifizierten Schnittstellen der WfMC. Zum Zweiten benötigt die Administrations- bzw. Klientenapplikation keinen J2EE-Anwendungsserver. Zum Dritten ist die Prozessbeschreibungssprache (XPDL) durch die WfMC spezifiziert und somit zu allen Punkten der Beschreibungen in Kapitel 2 kompatibel. Ausserdem können mittels JaWE Prozessdefinitionen leicht erstellt werden.

3 Konzeption der Schnittstelle

3.1 Struktur

Der in dieser Arbeit zu entwickelnde Prototyp soll die Ideen des in Abschnitt 2.3.1.2 auf Seite 24 vorgestellten ASAP/AWSP Protokolls aufgreifen. Die zentralen Komponenten sind das Beobachterobjekt, das Betriebsobjekt und das Instanzobjekt. Die im Standard definierten Objekte wurden dabei angepasst und um für dieses Projekt nicht benötigte Methoden erleichtert.

Eine Übersicht der einzelnen Komponenten und ihrer Beziehungen zueinander ist in Abb. 3.1 dargestellt.

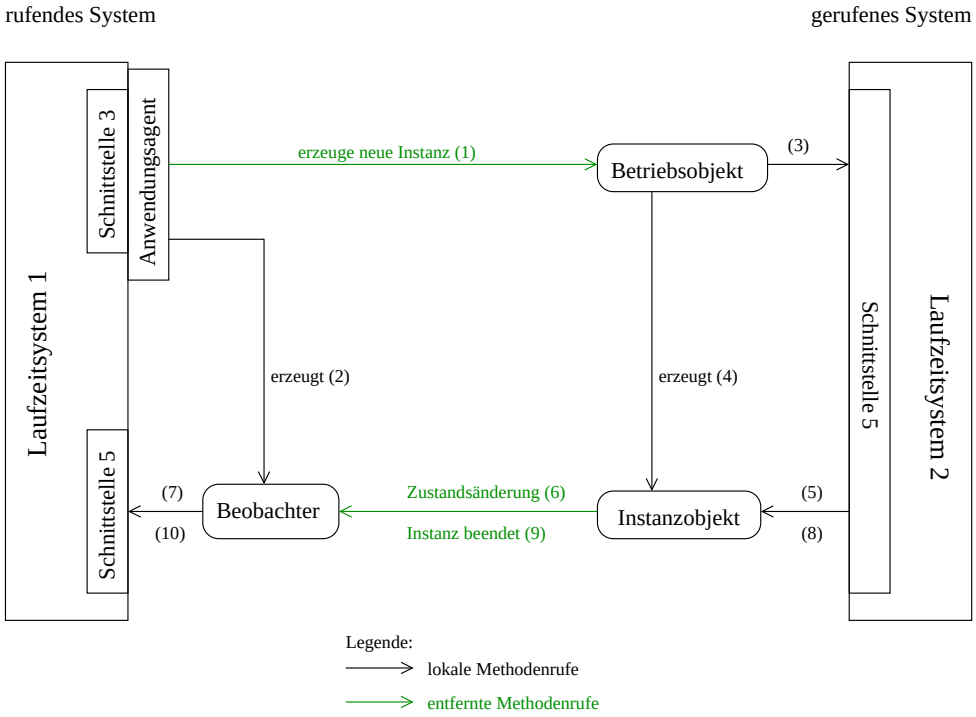

Abbildung 3.1: Konzept der zu entwickelnden Schnittstelle

Bei den weiteren Betrachtungen wird davon ausgegangen, dass die verwendete Laufzeitumgebung die von der WfMC standardisierten Schnittstellen implementiert.

Die Eigenschaften der Laufzeitumgebung 1, die das rufende System darstellt, sollen um die notwendige Funktionalität zum Starten externer Unterprozesse erweitert werden. Dies geschieht durch Nutzung der Schnittstellen 3 und 5 des Workflow Referenzmodells. Dabei muss der Anwendungsagent alle für das Starten des externen Unterprozesses notwendigen Daten von Laufzeitumgebung 1 erhalten. Dies sind:

- die Betriebsobjektadresse (welche gleichzeitig der eindeutige Identifikator ist),
- der Identifikator des Elternprozesses (rufenden Prozesses) und
- der Prozesskontext.

Die Betriebsobjektadresse ist die eindeutige Adresse unter der das Betriebsobjekt, das den externen Unterprozess repräsentiert, erreichbar ist. Die Übergabe des Prozessidentifikators ist essentiell, da der Beobachter bei Nutzung von Schnittstelle 5 des Workflow Referenzmodells nur über diesen auf den internen Prozess der Laufzeitumgebung 1 zugreifen kann. Der Prozesskontext beinhaltet alle prozessbezogenenen Daten des Elternprozesses, die dem externen Prozess übergeben werden müssen.

Nachdem der Anwendungsagent gestartet wurde, ruft er das Betriebsobjekt des zu startenden Unterprozesses unter Nutzung der übergebenen Adresse. An diese sendet er eine Anfrage zum Anlegen und Starten einer neuen Instanz dieses Betriebsobjekts. Dies ist in Abbildung 3.1 als Schritt (1) dargestellt.

Für die Anfrage an das Betriebsobjekt werden die in ASAP/AWSP vorgeschlagenen Parameter übergeben. Dies sind

- der Identifikator des Beobachters,
- der Identifikator des Betriebsobjekts und
- der Prozesskontext.

Zurückgegeben wird der Identifikator des erzeugten Instanzobjekts. Dieser wird für die weitere Kommunikation zum Überprüfen des Anfragenden benötigt.

Nachdem der externe Unterprozess gestartet wurde, legt der Anwendungsagent ein neues, zu diesem Unterprozess gehöriges Beobachterobjekt an (Schritt (2)). Dieses ist, im Gegensatz zu dem durch das ASAP/AWSP Protokoll definierten Objekt mit Klienten- und Serverfunktionalität, ein reines Serverobjekt. Es wird nur durch das erzeugte Instanzobjekt über Änderungen informiert. Die Attribute des Beobachterobjekts sind:

- ein eindeutiger Identifikator,
- der Identifikator des zugehörigen Prozesses in Laufzeitumgebung 1 und
- der Identifikator des erzeugten Instanzobjekts.

Auf dem System von Laufzeitumgebung 2 wird durch die Anfrage des Anwendungsagenten in Schritt 2 ein neuer Prozess gestartet. Dazu werden die im Betriebsobjekt abgelegten Informationen über den zu startenden Prozess genutzt. Diese Aktion wird durch Schritt 3 in der Abbildung verdeutlicht. Nachdem der Prozess gestartet wurde, wird er mit den übergebenen Kontextdaten initialisiert. Dabei muss der übergebene

Prozesskontext (also die Variablen) mit den im zu startenden Prozess definierten Eingangsdaten übereinstimmen. Sollte dies nicht der Fall sein, wird der gestartete Prozess wieder beendet und eine Fehlermeldung an den Anwendungsagenten zurückgegeben.

Wenn der Prozess gestartet und der Prozesskontext gesetzt werden konnte, wird ein neues Instanzobjekt erzeugt. Dieses hat die Aufgabe den Beobachter über Änderungen des Zustandes des gestarteten Prozesses und über Änderungen der Zustände der einzelnen Aktivitäten des Prozesses zu informieren. Die Benachrichtigung des Beobachterobjekts über Zustandsänderungen der Aktivitäten stellt dabei eine Erweiterung des ASAP/AWSP Protokolls dar, die in Wf-XML in der neu definierten Aktivitätsressource spezifiziert wurde. Das Instanzobjekt besitzt die folgenden Attribute:

- einen eindeutiger Identifikator,
- den Identifikator des erzeugenden Betriebsobjekts,
- den Identifikator des zugehörigen Beobachters und
- den Identifikator des gestarteten Prozesses in Laufzeitumgebung 2.

Nachdem die Erzeugung und Initialisierung abgeschlossen ist, wartet der Beobachter auf Benachrichtigungen über Änderungen durch das Instanzobjekt. Bei Auftreten einer Zustandsänderung[1] (vom Prozess selbst oder von einer Aktivität des Prozesses) in Laufzeitumgebung 2, wird diese an die erzeugte Instanzressource weitergegeben (Schritt 5). Die Instanz informiert den Beobachter durch Nutzung der von ihm bereitgestellten Methoden (Schritt 6). Dabei wird eine Unterscheidung zwischen Prozesszustandsänderung und Aktivitätszustandsänderung vollzogen. Die möglichen Zustände des Prozesses und der Aktivitäten wurden in den Abschnitten 2.1.4.2 und 2.1.5.2 auf den Seiten 14 und 18 bereits vorgestellt. Bei einer Zustandsänderung des Prozesses werden die folgenden Daten an den Beobachter übermittelt:

- der Identifikator der Instanz,
- der Identifikator des Betriebsobjekts,
- der vorherige Zustand,
- der aktuelle Zustand und
- der Zeitpunkt der Zustandsänderung.

Die beiden Identifikatoren werden zur Überprüfung der Anfrage benötigt. Dies ist bei allen genutzten Methoden gleich. Der Sender und der Empfänger werden übermittelt, um zu prüfen, ob der Empfänger die angefragte Ressource ist und der Sender, ob er das passende Gegenstück darstellt. Die beiden Zustände und der Zeitpunkt der Zustandsänderung beschreiben das aufgetretene Ereignis.

Dagegen werden bei einer Aktivitätszustandsänderung folgende Daten übermittelt:

- der Identifikator der Instanz,
- der Identifikator des Betriebsobjekts,
- der Identifikator der Aktivität,

[1] ausgeschlossen ist eine Beendigung des Prozesses, da diese gesondert behandelt wird

- der Name der Aktivität,
- der vorherige Zustand,
- der aktuelle Zustand und
- der Zeitpunkt der Zustandsänderung.

Die beiden ersten sowie die letzten drei Attribute sind besitzen die gleiche Bedeutung, wie bei der Zustandsänderung eines Prozesses. Hinzu kommen der Name der Aktivität und ihr eindeutiger Identifikator. Diese besitzen allerdings in Laufzeitumgebung 1 keine Gültigkeit, da es sich um interne Variablen von Laufzeitumgebung 2 handelt. Sie haben deshalb nur informativen Charakter.

Der Beobachter informiert den Elternprozess über die Zustandsänderungen des Unterprozesses (Schritt 7). Es werden hierbei keine prozessbezogenen Daten innerhalb von Laufzeitumgebung 1 geändert.

Der externe Unterprozess in Laufzeitumgebung 2 informiert also den Elternprozess über alle Aktivitäts- und Prozesszustandswechel (Schritte 5 bis 7).

Nachdem der Prozess abgeschlossen ist, müssen die prozessbezogenen Daten, die für den Elternprozess relevant sind[2] zurückgegeben werden. Dies geschieht bei Wechsel des Unterprozesses in den Zustand „Beendet“. Sobald dieser erfolgt, wird das Instanzobjekt von Laufzeitumgebung 2 darüber informiert und erhält alle prozessbezogenen Daten des Unterprozesses (Schritt 8). Diese werden, sofern es sich um Ausgabevariablen des in Laufzeitumgebung 1 definierten externen Unterprozesses handelt, an den Prozess zurückgegeben und sein Kontext wird erneuert (Schritt 9).

Nachdem die Übergabe der prozessbezogenen Daten erfolgreich abgeschlossen ist, werden das Beobachterobjekt (auf dem System von Laufzeitumgebung 1) und das Instanzobjekt (auf dem System von Laufzeitumgebung 2) beendet.

3.2 Unterschiede zu ASAP/AWSP

In diesem Abschnitt sollen die Kernkomponenten des erstellten Konzepts ihren im ASAP/AWSP Protokoll definierten Pendants gegenübergestellt werden. Dabei werden speziell die Unterschiede näher beleuchtet.

Alle Methoden der im ASAP/AWSP Protokoll definierten Objekte bestehen aus einer Anfrage und einer zugehörigen Antwort. Diese sind zeitlich unabhängig voneinander. Wird eine Anfrage gestellt, so wird diese in der direkten Rückgabe nur bestätigt. Die eigentliche Antwort wird dann zum Zeitpunkt der Beendigung des angebotenen Dienstes gegeben. Alle Anfragen beinhalten die folgenden Daten im Kopf der SOAP-Nachricht:

- den Identifikator des Senders,
- den Identifikator des Empfängers,
- eine Ja/Nein Angabe ob eine Antwort notwendig ist (nur bei einer Anfrage) und
- einen Anfrageidentifikator.

[2] das können geänderte Eingabevariablen oder erst im Unterprozess erzeugte Variablen sein

Die Verwendung der beiden Identifikatoren des Senders und Empfängers ist nötig, da es im ASAP/AWSP Standard möglich ist, mehrere Beobachterobjekte an eine Instanz zu binden
Übernommen wurden von diesen Daten nur die ersten beiden Identifikatoren. Eine Antwort wird also immer erwartet. Weiterhin ist der Anfrageidentifikator aufgrund der Tatsache, dass es nur einen Beobachter pro Instanz gibt, nicht notwendig.
Diese Daten werden in den fogenden Tabellen nicht in die Parameterlisten der Funktionen des Standards übernommen, da sie im Kopf der Nachricht und nicht in der eigentlichen Funktion auftreten.

Beobachterobjekt

Das im ASAP/AWSP Protokoll definierte Beobachterobjekt besitzt sowohl Klienten- als auch Serverfunktionalität. Es nutzt Methoden des Betriebsobjekts um neue Instanzen zu erzeugen. Weiterhin werden Methoden der Instanz verwendet um Zustandswechsel anzustoßen, sich bei einer Instanz an- und abzumelden und Attribute der Instanz auszulesen oder zu setzen. Zusätzlich bietet es selbst Methoden zum Erhalten von Nachrichten über Zustandswechsel oder zum Auslesen seiner Attribute für ein Instanzobjekt an.
Im Gegensatz dazu besitzt das im Konzept beschriebene Beobachterobjekt nur Serverfunktionalität. Die Anfrage zur Erzeugung einer Instanz an ein Betriebsobjekt wird vom Anwendungsagenten übernommen. Außerdem werden keine Methoden zur Abfrage eines Instanzobjekts benötigt. Der Grund dafür ist, dass der in diesem Konzept spezifizierte Beobachter nur genau einer Instanz zugeordnet ist und alle notwendigen Informationen über diese bei seiner Erzeugung durch den Anwendungsagenten gesetzt werden. Zusätzlich wurde die Methode zum Abfragen der Attribute des Beobachterobjekts, die im Standard definiert ist nicht in das Konzept aufgenommen, da auch das Instanzobjekt alle notwendigen Informationen über den Beobachter bei seiner Erzeugung kennt. In Tabelle 3.1 sind die Attribute und Methoden des Beobachters im Standard und im hier entwickelten Konzept gegenübergestellt.
Eine Besonderheit des Beobachterobjekts ist die hinzugefügte Methode zur Informierung über die Zustandsänderung einer Aktivität des entfernten Prozesses. Diese wird im ASAP/AWSP Standard nicht definiert. Sie ist aber in diesem Konzept vorgesehen, da der Elternprozess damit zu jedem Zeitpunkt Kenntnis darüber besitzt, welche Aufgabe des externen Prozesses gerade bearbeitet wird.

Betriebsobjekt

Das Betriebsobjekt stellt sowohl in diesem Konzept als auch im Standard die Beschreibung einer abarbeitbaren Aufgabe dar. Eine Anfrage zur Abarbeitung an dieses Objekt erzeugt eine neue Instanz, initialisiert sie mit übergebenen Daten und startet die Ausführung. Im Standard werden keine Aussagen getroffen, um welche Art von Aufgaben es sich handelt. Im Konzept werden dagegen diese Aufgaben auf Workflowprozesse eingeschränkt. Dadurch entstehen je nach verwendetem WfMS unterschiedliche Anforderungen an die Attribute des Betriebsobjekts. Im Fall von Enhydra Shark wird ein Prozess

Beobachterobjekt des Konzepts	Beobachterobjekt des Standards
Attribute:	
- eindeutiger Identifikator (`key`)	- eindeutiger Identifikator (`Key`)
Methoden:	
-	- `GetProperties()`
- `completed(senderKey, receiverKey, result)`	- `Completed(InstanceKey, ResultData)`
- `processStateChanged(senderKey, receiverKey, data)` `data = actState, lastState, stateChangedTime`	- `StateChanged(State, PreviousState)`
- `activityStateChanged(senderKey, receiverKey, data)` `data = activityKey, activityName, actState, lastState, stateChangedTime)`	-

Tabelle 3.1: Gegenüberstellung der Beobachterobjekte im Standard und im Konzept

anhand seines Paketidentifikators, der Paketversion und des Prozessidentifikators eindeutig beschrieben. Da dieses WfMS eingesetzt werden soll, werden diese in das Konzept übernommen. Bei Nutzung eines anderen Management Systems müssen diese Attribute eventuell angepasst werden.

Der Standard definiert weiterhin ein Reihe von Attributen, die hauptsächlich sinnvoll sind, wenn ein Mensch unter Nutzung des Beobachterobjekts mit dem Betriebsobjekt kommuniziert. Bei einer rein automatisiert ablaufenden Interaktion werden Attribute, wie die Beschreibungen des Dienstes (kurz und lang) oder der für Menschen lesbare Name der Ressource, nicht benötigt. Die maximale Zeit der Erreichbarkeit der erzeugten Instanz nach ihrer Beendigung wird ebenfalls aus den Attributen entfernt, da die Instanz nach der Benachrichtigung des Beobachters sofort beendet wird. Dies liegt darin begründet, dass es nur einen Beobachter pro Instanz gibt und dieser auch nach Beendigung des entfernten Prozesses beendet wird.

Die schematischen Beschreibungen der Eingangs- und Rückgabedaten des im Standard beschriebenen Betriebsobjekts dienen dem Definieren von Daten für viele verschiedene Arbeitsaufgaben. Diese können für jedes Betriebsobjekt verschieden sein. Da die Daten im zu implementierenden Prototyp an die prozessbezogenen Daten des eingesetzten WfMS gebunden sind und sie fest durch dieses definiert werden, sind diese Attribute nicht mehr zwingend notwendig. Sie werden deshalb nicht in das Konzept übernommen.

Die Methoden zum Ausgeben der erzeugten Instanzen und der Attribute des Betriebsobjekts werden nicht in das Konzept übernommen, da diese im zugrundeliegenden Einsatzgebiet nicht notwendig sind. Der Beobachter kommuniziert nur mit der ihm zugehörigen Instanz. Er muss also nicht alle weiteren Instanzen kennen. Der Anwendungsagent kennt alle für ihn notwendigen Attribute des Betriebsobjekts zum Erzeugen der Instanz.

Betriebsobjekt des Konzepts	Betriebsobjekt des Standards
Attribute:	
- eindeutiger Identifikator (`key`)	- eindeutiger Identifikator (`Key`)
-	- Name des Betriebsobjekts (`Name`)
-	- Kurzbeschreibung des Dienstes (`Subject`)
-	- ausführlicher Kurzbeschreibung des Dienstes (`Description`)
-	- Schematische Beschreibung der Eingangsdaten (`ContextDataSchema`)
-	- Schematische Beschreibung der Rückgabedaten (`ResultDataSchema`)
-	- die minimale Lebensdauer der erzeugten Instanz nach deren Beendigung (`Expiration`)
- Paketidentifikator (`xpdlPackage`)	-
- Paketversion (`xpdlVersion`)	-
- Prozessidentifikator (`xpdlProcess`)	-
Methoden:	
- `createInstance(senderKey, receiverKey, context)`	- `CreateInstance(ObserverKey, Name, Subject, Description, ContextData, InstanceKey)`
-	- `GetProperties()`
-	- `ListInstances(Filter, FilterType)`

Tabelle 3.2: Gegenüberstellung der Betriebsobjekte im Standard und im Konzept

Dadurch werden die Methoden zum Auslesen der Attribute dieses nicht mehr benötigt. Die detaillierte Übersicht der Attribute und der einzelnen Methoden mit den Parameterlisten ist in Tabelle 3.2 gegenübergestellt.

Instanzobjekt

Im Gegensatz zum im Standard definierten handelt es sich bei dem im Konzept entwickelten Instanzobjekt um ein reines Klientenobjekt. Es besitzt also keine Methoden, die durch ein entferntes Objekt (etwa den Beobachter) aufgerufen werden könnten. Das ist nicht nötig, da sein einziger Zweck darin besteht, den Beobachter über Ereignisse bei der Ausführung des externen Unterprozesses zu informieren.

Da es genau einen Beobachter pro Instanzobjekt gibt, werden die Methoden zum An- und Abmelden von Beobachtern nicht benötigt. Die relevanten Attribute der Instanz sind dem Beobachter bekannt und können durch diesen nicht verändert werden. Die letzte im Standard definierte Methode zum Ändern des Zustands der Instanz ist im hier verwendeten Ansatz nicht von Interesse, da dies nur durch die Laufzeitumgebung 2

Instanzobjekt des Konzepts	Instanzobjekt des Standards
Attribute:	
- eindeutiger Identifikator (`key`)	- eindeutiger Identifikator (`Key`)
-	- aktueller Zustand der Instanz (`State`)
-	- Name des Betriebsobjekts (`Name`)
-	- Kurzbeschreibung der Instanz (`Subject`)
-	- ausführlicher Kurzbeschreibung der Instanz (`Description`)
- der Identifikator des erzeugenden Betriebsobjekts (`factoryKey`)	- der Identifikator des erzeugenden Betriebsobjekts (`FactoryKey`)
- der zugehörige Beobachter (`observerKey`)	- alle lauschenden Beobachter (`Observers`)
-	- die prozessbezogenen Daten (`ContextData`)
-	- die Rückgabedaten (`ResultData`)
-	- Reihenfolge von Ereignissen mit den Zeiten (`History`)
Methoden:	
-	`GetProperties()`
-	`SetProperties(Data)`
-	`Subscribe()`
-	`UnSubscribe()`
-	`ChangeState(State)`

Tabelle 3.3: Gegenüberstellung der Instanzobjekte im Standard und im Konzept

geschehen darf[3].
Im eigentliche Sinne wird durch das Betriebsobjekt kein Objekt erzeugt, sondern es werden nur die Daten (eigener Identifkator, Identifikatoren des zugehörigen Betriebs- und Beobachterobjekts) persistent gespeichert. Diese werden später bei einer Erzeugung der Instanz, die durch Laufzeitumgebung 2 im Falle eines Ereignisses des entsprechenden Prozesses geschieht, zur korrekten Initialisierung der Instanz genutzt. Alle prozessrelevanten Informationen dieses Instanzobjekts liegen in Laufzeitumgebung 2 vor und werden bei einer Zustandsänderung zum Zeitpunkt des Auftretens einer solchen an das temporär erzeugte Instanzobjekt übergeben.

3.3 Ausfallsicherheit

Unter Ausfallsicherheit soll in diesem Kontext verstanden werden, wie die einzelnen Komponenten auf temporäre Unerreichbarkeit der entfernten Objekte reagieren. Dabei spielen nur die beiden Serverobjekte eine Rolle, da die Instanz als reines Klientenobjekt keine entfernten Methoden zur Verfügung stellt. Ein Objekt kann aus mehreren

[3] die Instanz repräsentiert, im Gegensatz zum Standard, nur den Prozess innerhalb von Laufzeitumgebung 2 und ist nicht dieser selbst

Gründen nicht erreichbar sein, z. B. bei Ausfall des Systems auf dem der Dienst läuft, Wartungsarbeiten, Ausfall der Netzverbindung, usw.. Der Ausfallsicherheit muss besondere Aufmerksamkeit geschenkt werden, da die Abarbeitung des externen Prozesses sich über eine sehr lange Zeitspanne erstrecken kann[4].

Vorgehensweise bei Unerreichbarkeit des Betriebsobjekts

Für eine temporäre Unerreichbarkeit des Betriebsobjekts müssen keine speziellen Vorkehrungen getroffen werden. Ist die Erzeugung der externen Instanz nicht erfolgreich, so bleibt die Abarbeitung des Elternprozesses am Ende der Aktivität stehen, die vor dem Prozessübergang zur Aktivität des externen Unterprozess abgearbeitet wurde. Das heißt der Prozess in Laufzeitumgebung 1 muss solange auf die Abarbeitung des Unterprozesses warten, bis das Betriebsobjekt wieder erreichbar ist. Daten können dabei nicht verloren gehen, denn beim späteren, neuen Versuch die externe Ressource zu erzeugen werden alle Daten nochmals übergeben.

Vorgehensweise bei Unerreichbarkeit des Beobachters

Schwieriger ist die Situation wenn die Instanz den Beobachter bei einem Prozessübergang oder gar bei der Beendigung des Prozesses nicht erreichen kann.

Im ersten Fall müssen die Daten (vorheriger Zustand, neuer Zustand, Zeitpunkt des Zustandswechsels) temporär persistent abgelegt und später an den Beobachter übermittelt werden. Hierzu kann z. B. ein Thread das Vorliegen von nicht gesendeten Zustandsänderungen zyklisch überprüfen oder die Prüfung kann manuell erfolgen.

Im Falle der Beendigung des Prozesses müssen die Rückgabedaten zwischengespeichert werden, bis der Beobachter informiert werden konnte.

[4] in Abschnitt 2.1.2.1 wurde dies bereits erwähnt

4 Implementierung

In diesem Kapitel wird die Implementierung des Prototypen vorgestellt. Als Programmiersprache wurde Java[1] in der Version 1.4.2 eingesetzt. Als Middleware-Technologie kam RMI zum Einsatz. Zusätzlich zu Java und Enhydra Shark ist eine mySQL-Datenbank für die Ausführung des Prototypen notwendig. Der Prototyp ist auf der beiliegenden CD in zwei Versionen enthalten: einmal als vorübersetzte Version und einmal als Quellcode. Für die Übersetzung des Quellcodes ist ausserdem Apache Ant[2] erforderlich.

Der implementierte Prototyp kann auf allen Systemen ausgeführt werden, auf denen Java in der erwähnten Version und eine mySQL-Datenbank verfügbar sind. Die Schritte zur Installation auf einem Unix/Linux-System sind auf der beiligenden CD beschrieben. Die während dieser Arbeit veröffentlichte Version 1.5 von Java kann, zumindest mit der vorübersetzten Version, nicht genutzt werden. Der Grund dafür ist, dass Enhydra Shark 1.0.1 nicht zu ihr kompatibel ist.

Um die Schnittstelle für Enhydra Shark zu nutzen sind zwei zusätzlich zu Enhydra Shark zu startende Serverobjekte notwendig. Diese werden später detailliert beschrieben. Für Windows und andere Betriebssysteme ist eine Anpassung der Skripte zum Start der Serverobjekte leicht möglich.

Die Implemtierung wurde auf verschiedenen PCs der x86-Architektur mit unterschiedlichen Linux-Distributionen als Betriebssystem getestet. Für die im nächsten Kapitel folgenden Testfälle wurde ein Pentium 3-System mit Suse Linux 9.2 und ein Athlon-System mit Fedora Core 2 als Betriebssystem genutzt. Die Computer waren über Ethernet miteinander verbunden.

4.1 Vorbetrachtungen

In diesem Abschnitt sollen für die Implementierung wichtige Eigenschaften und Eigenheiten von Enhydra Shark vorgestellt werden. Außerdem werden einige der Ideen beschrieben, die während der Entwicklung des Prototypen verworfen wurden, aber zu dieser Lösung beigetragen haben. Im einzelnen beinhaltet diese Vorbetrachtung die folgenden Aspekte: Konfigurationsdetails von Enhydra Shark für die Nutzung des Prototypen, Gründe für die Verwendung eines Unterprozesses als lokale Repräsentation des externen Prozesses, Begründung der Verwendung eines eigenständigen Servers für die Erzeugung der Beobachter. Die Ausführungen beziehen sich dabei auf Enhydra Shark in der Version 1.0.1. Die genutzte Dokumentation ist in [Sha] zu finden.

[1] http://java.sun.com
[2] http://ant.apache.org

Konfigurationsdetails von Enhydra Shark

In der im anschließenden Abschnitt beschriebenen prototypischen Implementierung des vorgestellten Konzepts wird Enhydra Shark von den implementierten Objekten als Bibliothek genutzt. Dafür ist es notwendig, das standardmäßig verwendete Datenbank Management System „Hypersonic SQL Database“ (HSQLDb) durch ein anderes (etwa mySQL oder PostgreSQL) zu ersetzen. Bei HSQLDb handelt es sich um eine reine Speicherdatenbank die nur für Testzwecke geeignet ist, nicht aber für den produktiven Einsatz. Die Daten werden bei einer Verbindung zur Datenbank aus einer Datei eingelesen und bei Schließen der Verbindung in diese Datei zurückgeschrieben. Leider zeigte diese Datenbank bei der gleichzeitigen Nutzung von Shark als Bibliothek durch mehrere „Java Virtual Machines“ Maschinen (JVM) ein fehlerhaftes Verhalten. So wurden Änderungen, die durch eine JVM erfolgten nicht an die anderen weitergegeben und waren manchmal nach einem Neustart des Programms nicht persistent in der Datenbank gespeichert.

Enhydra Shark bietet für diesen Problemfall die Möglichkeit, bei der Konfiguration aus mehreren Datenbanken auszuwählen. Dazu muss vor dem Ausführen von `configure.[sh|bat]` die Datei `configure.properties` angepasst werden. Hier kann man durch Setzen des Parameters `db_loader_job` das genutzte Datenbank Management System auf mySQL oder PostgreSQL umstellen.

Weiterhin muss das Caching von Daten innerhalb von Enhydra Shark abgeschalten werden, da sonst Änderungen (wie z. B. der Abschluss des externen Prozesses und damit die Erzeugung neuer Aufgaben für einen Nutzer) nicht sofort in einer anderen JVM sichtbar sind. Würde man das Caching nicht abschalten, müsste der Nutzer, im schlimmsten Fall, jedesmal wenn er seinen Aufgabenvorrat abruft die Klientenapplikation neu starten, um auch wirklich sicher zu sein, dass alle derzeitigen Aufgaben darin enthalten sind.

Um das Caching abzuschalten, müssen die folgenden Zeilen in der Datei `conf/Shark.conf` geändert werden:

- `DatabaseManager.defaults.cache.maxCacheSize=0`
- `DatabaseManager.defaults.cache.maxSimpleCacheSize=0`
- `DatabaseManager.defaults.cache.maxComplexCacheSize=0`
- `# CacheManagerClassName=org.enhydra.shark.caching.LRUCacheMgr`

Unterprozess statt Aktivität als lokale Repräsentation

In der ersten Idee für die prototypische Implementierung sollte der externe Prozess durch eine einzelne Aktivität gestartet werden, die entweder auf die Beendigung wartet (externer Unterprozesses) oder nach dem Starten mit der Ausführung der nächsten Aktivität fortfährt. Dies war aus zwei Gründen, die nachfolgend näher erläutert werden, nicht möglich.

Das erste Problem bestand darin, dass ein XPDL-Paket keine einzelnen Aktivitäten enthalten kann. Aktivitäten müssen innerhalb eines Prozesses definiert werden. Dadurch

wäre kein Einbinden der vordefinierten Aktivität möglich und sie müsste in jedem Prozess definiert werden. In der prototypischen Implementierung sollte die Definition und das Einfügen aber mit möglichst wenig Aufwand realisierbar sein. Außerdem wäre eine nachträgliche Änderung, z. B. der Parameterliste, in allen Prozessen erforderlich.
Das zweite Problem betrifft die verwendete Version 1.0.1 von Enhydra Shark. In dieser Version muss ein Anwendungsagent als Aktivität mit automatischem Start *und* automatischen Ende definiert werden. Dadurch kann, im Falle eines externen Unterprozesses, die Ausführung der Aktivität nicht angehalten werden. Zwar wäre es theoretisch möglich, die Aktivität durch Iteration solange auszuführen bis der externe Unterprozess beendet ist, das lässt sich aber nicht umsetzen, da der Aufwand für die ständige Prüfung in Enhydra Shark zu hoher Systemlast führt.

Gründe für ein eigenständiges Objekt zur Bereitstellung der Beobachter

Bei einem ersten Prototypen sollte die entwickelte Lösung ohne ein eigenständiges Objekt für die Bereitstellung der gestarteten Beobachter auskommen. In dieser Idee sollte jeder Beobachter in einem eigenen Thread durch die Shark-Laufzeitumgebung gestartet werden und ablaufen. Er sollte durch den Anwendungsagenten erzeugt werden. Als Anwendungsagent wird in Enhydra Shark kein eigenständiges Programm, sondern ein Objekt mit einer vordefinierten Methode, verstanden. Wird der Anwendungsagent gestartet, wird eine neue Instanz des Objekts erzeugt und diese Methode gerufen.
Diese Idee führte zu mehreren Problemen. Das erste Problem bestand darin, dass Enhydra Shark die Anwendungsagenten im selben Thread startet, wie die gesamte Laufzeitumgebung. Dadurch bleibt die Ausführung der Laufzeitumgebung beim Start des entsprechenden Anwendungsagenten gestoppt, bis die gerufene Methode dieses beendet ist. Die Methode ist aber erst abgeschlossen, wenn der Thread durch einen Thread-Join beendet wird. Das wiederum ist erst nach Beendigung des externen Prozesses möglich. Die Shark-Laufzeitumgebung bleibt also unnutzbar, bis der externe Prozess abgeschlossen ist.
Das zweite Problem war die Beendigung der Laufzeitumgebung und ein späterer Neustart. Will man dabei die Funktionalität zum Bereitstellen der Beobachter ohne eine externe Komponente realisieren, müssen früher erzeugte Beobachter noch nicht abgeschlossener, externer Prozesse bei Start von Enhydra Shark erzeugt werden. Dazu müsste eine neue Komponente für Enhydra Shark entwickelt und beim Start geladen werden.
Das dritte und zugleich größte Problem stellt die Möglichkeit von Enhydra Shark dar, als Bibliothek und nicht als Server genutzt werden zu können. Dadurch wäre mit der Lösung ohne eigenständigen Server die Erreichbarkeit der Beobachter nur zu dem Zeitpunkt, an dem die Bibliotheksfunktionen genutzt werden, gegeben. Das ist nicht akzeptabel, da so Änderungen der externen Prozesse möglicherweise nie an den Beobachter übermittelt werden können. Zwar wird, wie dies später beschrieben ist, versucht nicht übertragene Änderungen in zyklischen Abständen an den Beobachter zu übermitteln. Man kann sich aber leicht vorstellen, dass im schlechtesten Fall diese zyklische Übermittlung genau dann stattfindet, wenn die Beobachter nicht erreichbar sind. So kann es vorkommen, dass externe Prozesse nie lokal abgeschlossen werden können.

4.2 Architekturdesign

In diesem Abschnitt soll die Implementierung des Konzepts beschrieben werden, das in Kapitel 3 vorgestellt wurde. Dabei werden die einzelnen Komponenten in der Reihenfolge ihrer Abarbeitung behandelt und das Gesamtarchitekturdesign Schritt für Schritt entwickelt.
Anfangs werden Hilfsobjekte definiert, die für die Implementierung notwendig sind. Das sind die Datenspeicher für die Beobachter- und Instanzobjekte, der Beobachterserver und der Betriebsobjektserver. Anschließend wird der den externen Prozess repräsentierende Unterprozess detailliert dargestellt. Danach folgt die Definition des Anwendungsagenten, die für die Anfrage zur Erzeugung des externen Unterprozesses notwendigen Datenstrukturen und das Betriebsobjekt. Darauffolgend werden das Instanzobjekt, die für Rückmeldungen über den Fortschritt der Ausführung notwendigen Datenstrukturen und der Beobachter behandelt.
Abschließend werden die einzelnen Komponenten zu einem Gesamtbild zusammengefasst.
Die Schnittstellendefinitionen der Klassen, die entfernte Methoden anbieten, sind im Anhang ab Seite 103 zu finden.

4.2.1 Datenspeicher

Die Datenspeicher werden verwendet, um den Serverobjekten (Beobachterserver, Betriebsobjektserver) bei einem Neustart Informationen über aktive Ressourcen bereitzustellen. Dadurch kann das System auch bei zwischenzeitlichem Ausfall die erfolgreiche Abarbeitung externer Unterprozesse gewährleisten. Die Wahrscheinlichkeit eines Ausfalls wird bei sehr langlebigen Prozessen (mehrere Wochen) deutlich höher sein als bei eher kurzen (Minuten, Stunden), sollte aber in jedem Fall berücksichtigt werden.
Die Beobachter legen weiterhin Informationen über Zustandsänderungen des entfernten Prozesses in ihrem Datenspeicher ab, um bei Bedarf dem Administrator den Abarbeitungsablauf zur Verfügung zu stellen.
Die Instanzen speichern Informationen über fehlgeschlagene Benachrichtigungen des Beobachters, um diese zu einem späteren Zeitpunkt erneut übertragen zu können. Das ist von zentraler Bedeutung, da Datenverlust bei Geschäftsprozessen ein sehr kritisches Problem darstellt.
Als Datenspeicher wird im Prototyp eine mySQL[3] Datenbank eingesetzt.

Beobachter-Datenspeicher

Der Beobachter-Datenspeicher enthält die folgenden Relationen:

- `ObserverProcesses`,
- `ObserverProcessData`,
- `ObserverProcessInfo` und
- `ObserverProcessAcivityInfo`.

[3] www.mysql.com

ObserverProcesses Die `ObserverProcesses`-Relation enthält die grundlegenden Informationen über alle gestarteten Beobachter. Das sind der Schlüssel des Beobachters (`ObserverKey`), der gleichzeitig Primärschlüssel ist, der eindeutige Identifikator des zugehörigen lokalen Unterprozesses (`ProcessId`) und der Schlüssel des gestarteten, externen Instanzobjekts (`InstanceKey`). Weiterhin sind das Startdatum des externen Prozesses (`StartDate`) und das Datum des spätesten Beendigungszeitpunkts (`Deadline`), also der Abarbeitungsfrist, in dieser Relation abgelegt.

Der `ObserverKey` repräsentiert die RMI-URI des Beobachters, bestehend aus dem Protokoll „`rmi://`“, der IP oder dem DNS Namen des Systems, auf dem der Beobachter ausgeführt wird (z. B. „`192.168.0.1`“ oder „`miranda.informatik.tu-chemnitz.de`“), dem Port falls nicht der RMI-Standardport[4] genutzt wird und dem lokal eindeutigen Namen des Unterprozesses. Eine vollständiger `ObserverKey` ist:

„`rmi://192.168.0.2:8000/102_ObserverProcess_ObserverDefault`“.

Der eindeutige Prozessidentifikator wird zum Übermitteln von Zustandsänderungen an die Shark-Laufzeitumgebung und zum Abschließen des lokalen Unterprozesses innerhalb der Laufzeitumgebung benötigt.

Der Schlüssel der mit einem Beobachter verknüpften Instanz wird zum Prüfen von Anfragen an den Prozess benötigt. Damit soll verhindert werden, dass versehentlich falsche Informationen an einen Beobachter gesendet werden können. Im schlimmsten Fall kann dies das Abschließen eines externen Prozesses mit falschen Rückgabedaten sein. Wie dieser Identifikator gebildet wird, wird später bei der Beschreibung des Instanzobjekt-Datenspeichers genauer erläutert.

Die Attribute `StartDate` und `Deadline` haben in der derzeitigen prototypischen Umsetzung rein informative Bedeutung. Dies liegt darin begründet, dass Enhydra Shark in Version 1.0.1 nur sehr eingeschränkte Funktionalität im Umgang mit Abarbeitungsfristen besitzt. Es müsste zyklisch das Überschreiten einer solchen Frist geprüft werden. Dies sollte, falls Enhydra Shark diese Eigenschaft erweitert, noch in den Prototypen implementiert werden.

In Tabelle C.1 auf Seite 111 sind die Eigenschaften dieser Relation zusammengefasst dargestellt.

ObserverProcessData In der Relation `ObserverProcessData` werden Informationen zu den durch den Beobachter übermittelten Daten abgelegt. Daten werden durch einen eindeutigen Datenidentifikator (`DataId`) bestehend aus dem Prozessidentifikator und dem Namen des Datums identifiziert. Dieser ist zusammen mit dem Identifikator des Beobachters der Primärschlüssel in dieser Relation.

Der Name eines Datums innerhalb des lokalen Unterprozesses (`Name`) wird gespeichert, um das Zurückschreiben des Wertes zu ermöglichen.

Jedes Datum lässt sich in eine Kategorie einordnen, die im Attribut `Mode` gespeichert wird. Die Kategorien sind „Ausgangsdaten“ (`OUT`), „Eingangsdaten“ (`IN`) und „Ein-

[4] 1099

/Ausgangsdaten“ (`INOUT`). Die Kategorie bezieht sich dabei auf den externen Unterprozess. Das heißt Eingangsdaten sind Daten, die an den externen Unterprozess übermittelt und bei dessen Beendigung *nicht* zurückgeschrieben werden. Ausgangsdaten werden als leeres Datum übermittelt und bei Beendigung des externen Prozesses gesetzt. Ein-/Ausgangsdaten sind eine Kombination der beiden vorherigen.
Eine Zusammenfassung der Attribute und ihrer Bedeutungen befindet sich in Tabelle C.2 auf Seite 111.

ObserverProcessInfo Die Relation `ObserverProcessInfo` dient dem Speichern von Zustandsänderungen eines exteren Unterprozesses. Dabei wird eine Zustandsänderung eindeutig durch die `ProcessId` und den Zeitpunkt der Änderung (`StateChangedTime`) identifiziert, weswegen diese beiden Attribute den Primärschlüssel bilden. Die `ProcessId` allein ist hier nicht ausreichend, da sich der Zustand des Prozesses mehrmals ändern kann.
Der aktuelle und der vorige Zustand (`ActState` und `LastState`) sind Informationen, die genutzt werden können um den genauen Ablauf der Abarbeitung des externen Prozesses zu rekonstruieren. Dieser kann z. B. genutzt werden, um Optimierungen am externen Prozess vorzunehmen und Engpässe bei der Abarbeitung zu finden.
Die einzelnen Attribute und ihre Wertebereiche werden in Tabelle C.3 auf Seite 112 zusammengefasst.

ObserverProcessActivityInfo Zum Speichern von Aktivitätszustandsänderungen wird die Relation `ObserverProcessActivityInfo` verwendet. Der Primärschlüssel und damit eindeutige Identifikator eines Datensatzes dieser Tabelle wird aus dem Identifikator der Aktivität (`ActivityId`), dem Zeitpunkt der Zustandsänderung (`StateChangedTime`) und dem Identifikator des Beobachters, der den entsprechenden exteren Prozess überwacht, gebildet. Der Aktivitätsidentifikator ist in diesem Fall nicht mehr eindeutig, da es theoretisch mehrere gleichnamige Aktivitäten in verschiedenen Shark-Laufzeitumgebungen geben kann. Der Zeitstempel wird wiederum in den Primärschlüssel aufgenommen, da es mehrere Zustandswechsel einer Aktivität geben kann.
An dieser Stelle soll noch auf eine Besonderheit hingewiesen werden. Der Prozessidentifikator in der Tabelle `ObserverProcessInfo` ist der Identifikator des lokalen Prozesses und deshalb eindeutig. Die Nutzung des lokalen Identifikators ist möglich, da der Prozess des rufenden Systems genau einem Prozess des gerufenen Systems zugeordnet ist. Bei den Aktivitäten ist dies nicht möglich, da im den externen Unterprozess repräsentierenden, lokalen Unterprozess keine Zuordnung von lokalen zu externen Aktivitäten[5] existiert.
Der Name der Aktivität (`Name`) dient der besseren Lesbarkeit in einem Abarbeitungsablauf. Die zwei verbleibenden Attribute (`ActState` und `LastState`) beschreiben wie schon bei den Prozesszustandsänderungen die Art dieser.
Diese Relation wird in Tabelle C.4 auf Seite 112 kurz zusammengefasst.
In Abbildung 4.1 sind die einzelnen Elemente des Datenspeichers mit ihren Attributen und Beziehungen abschließend als ER-Diagramm[6] dargestellt.

[5] Diese sind dem Elternprozess nicht bekannt.
[6] Entity Relationship - Diagramm

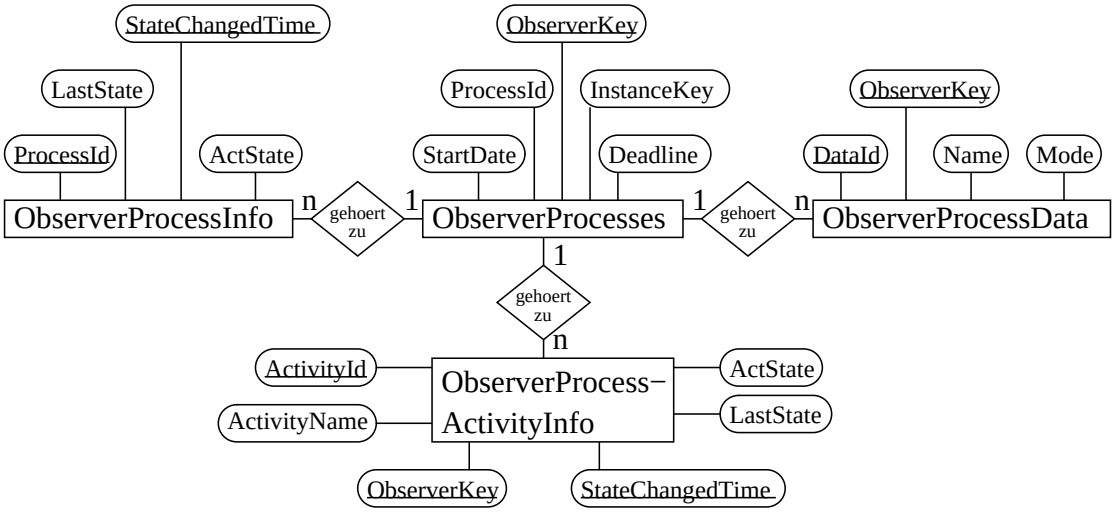

Abbildung 4.1: ER-Diagramm des Beobachter-Datenspeichers

Instanzobjekt-Datenspeicher

Der Instanzobjekt-Datenspeicher enthält die folgenden Relationen:

- `InstanceProcesses`,
- `InstanceProcessData`,
- `InstanceProcessInfo` und
- `InstanceProcessAcivityInfo`.

InstanceProcesses Informationen über jeden durch das Betriebsobjekt gestarteten Prozess werden in dieser Relation abgespeichert. Dabei wird der Identifikator der erzeugten Instanz als Primärschlüssel verwendet. Dieser besteht aus der Adresse des zugehörigen Betriebsobjektes (`host:port` bzw. `host`) gefolgt von „/“ und dem Prozessidentifikator des gestarteten Prozesses in der („Kind“) Shark-Laufzeitumgebung. Theoretisch würde für das lokale System der Identifikator innerhalb von Enhydra Shark als eindeutige Kennung ausreichen. Da der zugehörige Beobachter aber im Regelfall auf einem anderen System arbeitet, ist die Eindeutigkeit des Identifikators für den Beobachter nicht mehr gegeben. Deshalb werden die Informationen des Systems in die Instanzidentifikation übernommen. Ein Beispiel für einen solchen Identifikator ist:

```
192.168.0.9:8000/305_EntfernteProzesse_EntfernterProzess_Default
```

In dieser Tabelle werden außerdem der Schlüssel des Betriebsobjekts, durch das diese Instanz erzeugt wurde (`FactoryKey`), der Schlüssel des zugehörigen Beobachters (`ObserverKey`) und der lokale Prozessidentifikator (`ProcessId`) abgelegt. Der Beobachterschlüssel dient dem Auffinden des Beobachters, da, wie im Konzept beschrieben, die Instanz nur im Falle eines Ereignisses (Zustandswechsel, Beendigung eines Prozesses) erzeugt und nach

Benachrichtigung des Beobachters wieder gelöscht wird. Die Attribute aller Instanzen müssen also persistent gespeichert werden.
Der Prozessschlüssel dient dem Finden der Instanzdaten in der Datenbank, falls die Benachrichtigung des Beobachters durch die angepassten Objekte innerhalb der Shark-Laufzeitumgebung erfolgt. In diesem Fall ist der Instanz nur der Prozessidentifikator zugänglich. Warum Objekte innerhalb von Enhydra Shark angepasst werden mussten und wie dies genau geschehen ist, wird in Abschnitt 4.2.8 beschrieben und soll an dieser Stelle nicht weiter vertieft werden.
Durch das Attribut `StartDate` wird der Zeitpunkt des Erzeugens der Instanz gespeichert. `Deadline` hat im aktuellen Prototyp keine Bedeutung und wird deshalb leer initialisiert. Der Grund hierfür wurde bei der Behandlung der Relation `ObserverProcesses` gegeben.
Die Attribute `Completed` und `ObserverInformed` sind für den Überwachungsthread vorgesehen um anzuzeigen, dass ein Prozess ohne Benachrichtigung des Beobachters beendet wurde[7].
Die Zusammenfassung dieser Relation befindet sich in Tabelle C.5 auf Seite 113.

InstanceProcessData In dieser Relation werden die prozessbezogenen Daten gespeichert, falls der Beobachter bei der Beendigung eines Prozesses nicht informiert werden konnte. Diese werden bei Erzeugung und Start des Prozesses durch das Betriebsobjekt angelegt. Dabei werden alle Daten (d. h. auch reine Eingangsdaten) in die Tabelle aufgenommen.
Der Identifikator (`DataId`) besteht aus dem Schlüssel des gestarteten Prozesses und dem Namen des Datums. Der Schlüssel des zugehörigen Instanzobjekts (`InstanceKey`) wird als Fremdschüssel verwendet und dient dazu eine eindeutige Zuordnung von Daten zu Instanzen zu ermöglichen.
Eine Verbindung zwischen dem hier gespeicherten Datum und dem später übermittelten wird über seinen Namen innerhalb des Prozesses der Laufzeitumgebung des Beobachters hergestellt. Dieser wird im Attribut `Name` gespeichert.
Auch für diese Relation wird eine Zusammenfassung gegeben. Sie ist in Tabelle C.6 auf Seite 113 zu finden.

InstanceProcessInfo In der Relation `InstanceProcessInfo` werden die Zustandsänderungen des lokalen Prozesses abgelegt. Dies geschieht allerdings nur im Falle der Unerreichbarkeit des Beobachters. Wurden die Informationen an diesen übermittelt ist eine Speicherung nicht notwendig.
Die einzelnen Attribute entsprechen denen der `ObserverProcessInfo` mit zwei Unterschieden. Der Erste ist, dass der Identifikator des Prozesses `ProcessId` dem eindeutigen Schlüssel des in dieser Laufzeitumgebung gestarteten Prozesses entspricht. Der zweite Unterschied ist ein Attribut namens `ObserverInformed`, das genutzt wird um dem Überwachungsthread anzuzeigen, dass er diese Information noch an den Beobachter weitergeben muss.
Die einzelnen Attribute und ihre Wertebereiche werden in Tabelle C.7 auf Seite 114 dargestellt.

[7] `Completed`='1' und `ObserverInformed`='0'

InstanceProcessActivityInfo Als letzte Relation soll in diesem Abschnitt die `InstanceProcessActivityInfo` vorgestellt werden. In ihr werden alle nicht übermittelten Zustandsänderungen von Aktivitäten gespeichert um diese zu einem späteren Zeitpunkt erneut übermitteln zu können.

Diese Datenstruktur ist fast identisch zur `ObserverProcessActivityInfo` mit zwei wesentlichen Unterschieden. Zum einen wird das Attribut `ObserverInformed` hinzugefügt, das angezeigt, ob der entsprechende Beobachter informiert werden konnte. Zum anderen wird statt eines zugehörigen Beobachters, der in der Umgebung des entfernten Prozesses nicht vorhanden ist, das zugehörige Instanzobjekt durch seinen Schlüssel aufgenommen.

Die Attribute `ActivityId`, `ActivityName`, `ActState`, `LastState` und `StateChangedTime` besitzen die gleiche Funktion wie ihre entsprechenden Gegenstücke im Beobachter-Datenspeicher.

Diese Relation wird in Tabelle C.8 auf Seite 114 zusammengefasst.

Wie vorher beim Beobachter-Datenspeicher wird abschließend in Abb. 4.2 das zugrundeliegende Datenmodell als ER-Diagramm dargestellt.

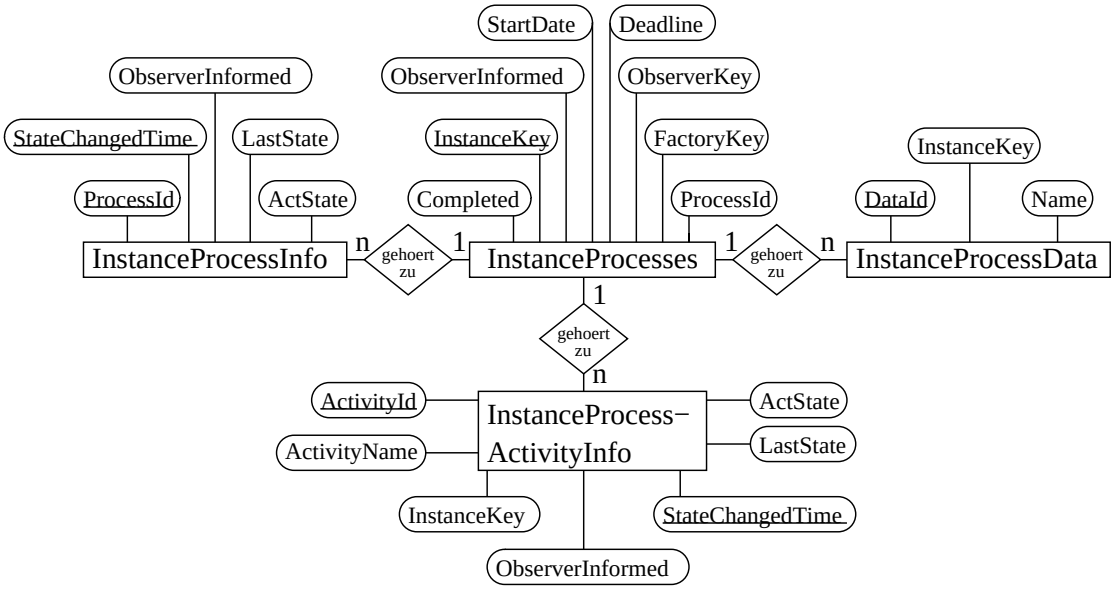

Abbildung 4.2: ER-Diagramm des Instanzobjekt-Datenspeichers

4.2.2 Der Beobachterserver

Der Beobachterserver ist für die Erzeugung und Bereitstellung der Beobachterobjekte zuständig. Dies können sowohl zur Laufzeit gestartete als auch nach einem Ausfall oder Neustart geladene sein. Er ist, wie der nachfolgend behandelte Betriebsobjektserver, kein Teil des in Kapitel 3 aufgestellten Konzepts, da er ein Hilfsmittel für die Implementierung darstellt. In den folgenden Absätzen soll die Funktionsweise dieses Objekts näher beschrieben werden.

Als erstes liest der Beobachterserver seine RMI-URI aus der Konfigurationsdatei `$SHARK_HOME/sharkinterop/conf/sharkinterop.conf` und bindet sich, mit Hilfe des RMI-Namensdienstes, an diese. `$SHARK_HOME` ist dabei eine Umgebungsvariable, in der der Installationsordner der Shark-Laufzeitumgebung gespeichert ist. Diese wird in den folgenden Betrachtungen häufig verwendet. Die entsprechende Eigenschaft für die Serveradresse innerhalb der Konfigurationsdatei heißt `observerserver.uri`.

Der Zugriff auf den Beobachterserver sollte nur vom System, auf dem dieser ausgeführt wird, gewährt werden. Die Berechtigung zum Zugriff kann durch die von RMI verwendeten Sicherheitsrichtlinien (*Policies*) eingeschränkt werden. Die Sicherheitsrichtlinien für den Beobachterserver werden in einer Datei mit dem Namen `observerserver.policy` abgelegt, die sich im Verzeichnis `$SHARK_HOME/sharkinterop/conf/` befindet. Die Grundeinstellung ist, dass der Zugriff nur dem lokalen System (IP-Adresse 127.0.0.1) gewährt wird.

Die Registrierung dieser Komponente als RMI-Serverobjekt und die Nutzung der bereitgestellten Methode zur Erzeugung neuer Beobachter durch den Anwendungsagenten ist zwingend notwendig, da Enhyda Shark in zwei unterschiedlichen Modi eingesetzt werden kann. Dies ist zum einen der Einsatz als CORBA Server, bei dem sich der Nutzer über einen Klienten mit der Laufzeitumgebung verbindet. Zum anderen - das ist der eigentliche Einsatzmodus - wird es als Bibliothek genutzt. Das Besondere beim zweiten Fall ist, dass das Bibliotheksobjekt beliebig erzeugt und wieder zerstört werden kann und mit ihm auch alle in dieser Virtuellen Maschine angelegten Objekte. Die erzeugten Beobachter müssen jedoch immer auf Datenübertragungen von entfernten Prozessen warten.

Nachdem der Beobachterserver sich beim RMI-Namensdienst registriert hat, lädt er alle Daten bereits existierender Beobachter aus dem Beobachter-Datenspeicher (Schritt (1) in Abbildung 4.3), erzeugt für diese die entsprechenden Beobachter und registriert sie beim RMI-Namensdienst (Schritt (2) der Abbildung).

Der letzte Schritt des Beobachterservers ist die Initialisierung der Shark-Laufzeitumgebung für die gestarteten Beobachter. Diese geschieht, da Enhydra Shark zum Abschließen von Prozessen als Bibliothek genutzt wird. Theoretisch könnte diese Initialisierung auch beim ersten Zugriff auf die Laufzeitumgebung durch eines der Beobachterobjekte geschehen. Da die Initialisierung aber zwischen 20 und 40 Sekunden benötigt, findet sie vor der eigentlichen Nutzung statt, um später keine Verzögerungen zu erzeugen.

Damit ist die Initialisierung des Beobachterservers abgeschlossen und er wartet auf Anfragen zur Erzeugung neuer Beobachter durch den Anwendungsagenten.

Beendet werden kann diese Komponente durch Eingabe von `SHUTDOWN`. Geschieht dies, werden alle Beobachter beim RMI-Namensdienst abgemeldet und durch den GC[8] der JVM zerstört. Als letztes meldet sich auch der Beobachterserver beim Namensdienst ab. Beim nächsten Start wird dann derselbe Zustand wieder hergestellt, wie vor dem Herunterfahren des Dienstes.

[8] Garbage Collector

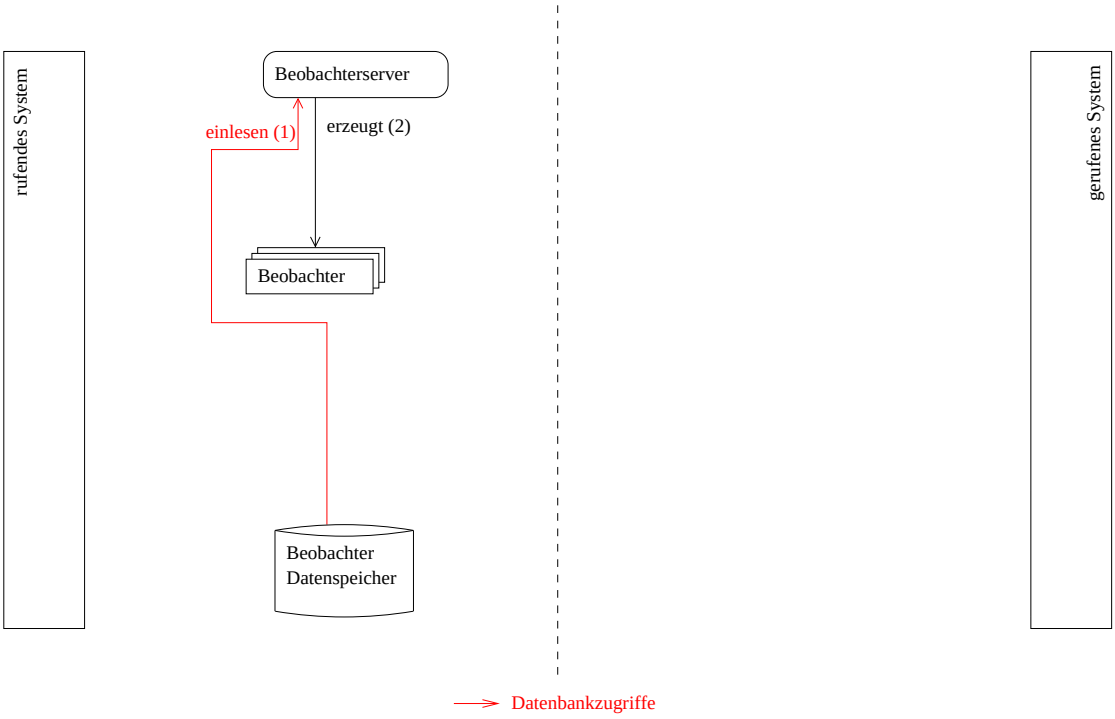

Abbildung 4.3: Implementierung des Beobachterservers

4.2.3 Der Betriebsobjektserver

Der Betriebsobjektserver ist dafür verantwortlich alle definierten Betriebsobjekte zu laden. Er besitzt daher Ähnlichkeit mit der im Wf-XML Standard definierten Dienste-registrierung. Allerdings ist es in diesem Prototyp nicht möglich zur Laufzeit neue Betriebsobjekte hinzuzufügen. Der Name Betriebsobjektserver kann an dieser Stelle verwirrend sein, da es sich, wie eben angedeutet, nicht um ein Serverobjekt handelt. Da er jedoch das Gegenstück zum Beobachterserver darstellt und in einer späteren Version um die Möglichkeit des Ladens zur Laufzeit erweitert werden könnte, soll dieser Name verwendet werden.

Jedes zur Verfügung gestellte Betriebsobjekt muss in der Datei `factories.properties` im Verzeichnis `$SHARK_HOME/sharkinterop/conf` anhand seiner Eigenschaften definiert werden. Die für jedes Betriebsobjekt zu definierenden Eigenschaften und ihre Funktion sind in Tabelle 4.1 zu finden. Dabei müssen diese an einen eindeutigen Namen für das Betriebsobjekt durch '.' getrennt angehängt werden. Für einen Beobachter mit dem Namen `beobachter1` bedeutet das, dass die Eigenschaft `host` als `beobachter1.host` angegeben wird. Die anderen Eigenschaften sind entsprechend zu definieren.

Die ersten drei Eigenschaften werden zur Bildung des eindeutigen Identifikators verwendet, der gleichzeitig die RMI-URI des zugehörigen Betriebsobjekts darstellt. Wenn kein `port` angegeben ist, wird der Identifikator nach dem Schema `rmi://host/factoryname` gebil-

Eigenschaft	Funktion
`host`	die IP Adresse oder der DNS Name des Systems des zu startenden Dienstes
`port`	der Port (wenn nicht angegeben wird der RMI-Standardport verwendet)
`factoryname`	der Name des Betriebsobjekts
`engine`	Name der entsprechenden Shark-Laufzeitumgebung (sollte für jedes Betriebsobjekt unterschiedlich sein)
`scope`	der Gültigkeitsbereich (muss derzeit nicht gesetzt werden)
`user`	der Name des Nutzers, unter dem sich das Betriebsobjekt anmelden soll
`pass`	das Passwort des Nutzers
`xpdlpackage`	das Paket des zu startenden Prozesses
`xpdlversion`	die Version dieses Pakets
`xpdlprocess`	der Name des Prozesses innerhalb des Pakets
`chaining`	kennzeichnet den Prozess als verketteten Unterprozess wenn der Wert gleich `yes` ist
`startdelay`	Zeit vor dem erstem Prüfen von nicht übermittelten Zustandsänderungen der Instanzen
`period`	Intervall zum Überprüfen des Datenspeichers ob Zustandsänderungen der Instanzen des Betriebsobjekts nicht an den Beobachter übermittelt werden konnten

Tabelle 4.1: Definierbare Eigenschaften eines Betriebsobjekts

det. In diesem Fall wird der RMI-Standardport genutzt. Wird der `port` angegeben, lautet die Bildungsvorschrift `key = rmi://host:port/factoryname`.

Danach folgen die Eigenschaften für die Verbindung mit der Shark-Laufzeitumgebung. Je nach Einsatz von Enhydra Shark kann entweder bei allen Betriebsobjekten derselbe Nutzer oder verschiedene Nutzer angegeben werden. Bei Angabe eines Nutzers für alle Betriebsobjekte können die einzelnen Aktivitäten der Prozesse innerhalb von Enhydra Shark durch Nutzung sogenannter „Mappings“ an die eigentlichen Bearbeiter weitergegeben werden. Es ist hier also nicht der Fall, dass der angegebene Nutzer den Prozess auch bearbeitet. Aus Sicherheitsgründen sollte ein eigener Nutzer für die Betriebsobjekte angelegt werden, da das Passwort im Klartext in dieser Datei gespeichert wird.

Die Eigenschaft `chaining` wird für eine besondere Art der verteilten Abarbeitung benötigt, die in Kapitel 2.2 auf Seite 20 vorgestellt wurde. Es handelt sich um die Verkettung von Prozessen. Dabei wird der Prozess nur gestartet und initialisiert und es soll keine Benachrichtigung des Rufenden erfolgen. Die möglichen Werte sind `yes` und `no`.

Die beiden letzten Angaben zu einem Betriebsobjekt werden für die Ausfallsicherheit benötigt. Sie geben den Zeitpunkt des Beginns der Arbeit des Überwachungsthreads und die Intervalle, in denen er prüft, ob der Beobachter über Änderungen informiert werden

muss, an. Der Beginn wird dabei relativ zum Erzeugungszeitpunkt des Betriebsobjekts spezifiziert.

In Abbildung 4.4 wird der Beobachter mit seinen Funktionen grafisch dargestellt.

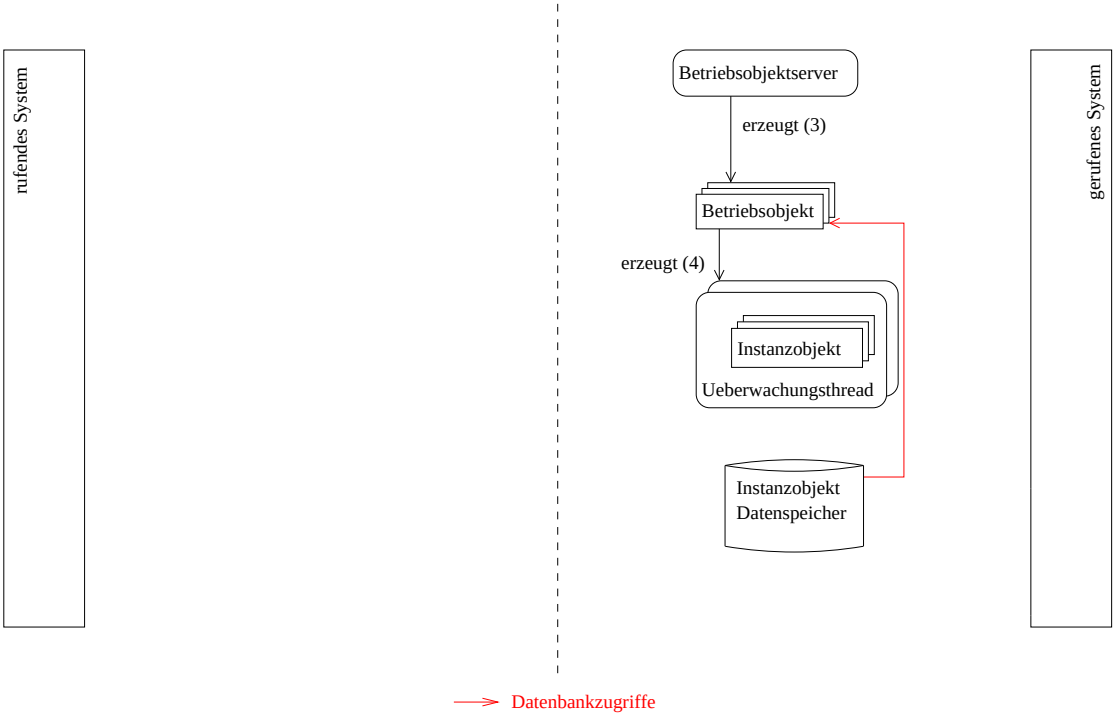

Abbildung 4.4: Implementierung des Betriebsobjektservers

In Schritt (3) der Abbildung erzeugt der Betriebsobjektserver aus den Eigenschaften die einzelnen Betriebsobjekte und bindet diese an den RMI-Namensdienst.

An dieser Stelle soll eine Funktion des Betriebsobjekts vorgegriffen werden, da sie zeitlich gesehen besser hierher passt. Beim Erzeugen der Betriebsobjektressource werden alle Instanzen, die in einem früheren Lauf gestartet und noch nicht beendet worden sind, aus der Datenbank geladen und an einen Überwachungsthread gebunden (Schritt (4) in Abbildung 4.4).

Danach wartet er auf seine Beendigung, bei der er die erzeugten Objekte wieder beim Namensdienst abmeldet. Die Beendigung wird durch Eingabe von `SHUTDOWN` eingeleitet.

4.2.4 Die lokale Repräsentation des externen Prozesses

Für die in dieser Arbeit umgesetzte Lösung einer verteilten Prozessabarbeitung in Enhydra Shark wird ein Ansatz gewählt, bei dem der externe Unterprozess durch einen eigenen lokalen Unterprozess repräsentiert ist. Dieser stellt die Funktionalität zum Starten, Beenden und Anzeigen von Informationen der Abarbeitung des entfernten Prozesses

zur Verfügung. Die Gründe für die Verwendung eines Prozesses statt einer einzelnen Aktivität wurden in Abschnitt 4.1 gegeben.

Da die zu übergebenden Parameter, z. B. die prozessbezogenen Daten, der Identifikator des externen Systems und der Name des externen Prozesses, sich in den verschiedenen Nutzungsszenarien unterscheiden, wird im Paket `ObserverPacket.xpdl` ein Vorgabeprozess definiert, der als Grundlage aller selbst definierten dient. Von ihm können durch Replikation mehrere, für jeden externen Prozess entsprechende, eigene Prozesse abgeleitet werden, die an die speziellen Bedürfnisse angepasst sind. Wie dies genau geschehen muss, wird am Ende dieses Abschnitts erklärt.

Der Vorgabeprozess besteht aus drei Aktivitäten und wird in Abbildung 4.5 dargestellt. Die XPDL Beschreibung befindet sich im Anhang in Kapitel A auf Seite 100.

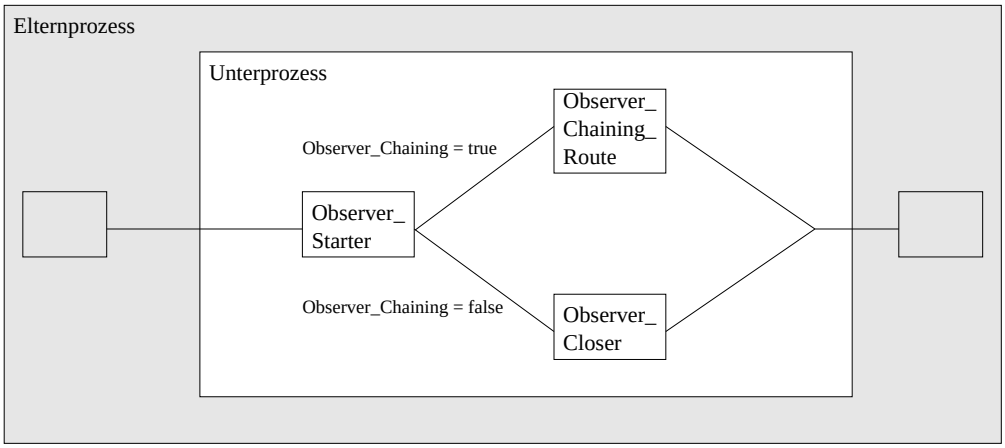

Abbildung 4.5: Der Vorgabeprozess

Er enthält die folgenden prozessbezogenen Daten, die in der Aktivität `Observer_Closer` abgerufen werden können, falls der externe Prozess nicht als verketteter Prozess gestartet wurde:

- `externalProcessState` der aktuelle Zustand des entfernten Prozesses,
- `externalProcessLastState` der vorige Zustand des entfernten Prozesses,
- `externalProcessStateChangedTime` der Zeitpunkt der letzten Prozesszustandsänderung,
- `externalProcessActivityName` der Name der Aktivität des externen Prozesses, die als letztes den Zustand gewechselt hat,
- `externalProcessActivityState` der aktuelle Zustand dieser Aktivität,
- `externalProcessActivityLastState` der vorige Zustand dieser Aktivität und
- `externalProcessActivityStateChangedTime` der Zeitpunkt dieser Zustandsänderung.

Zusätzlich wird das prozessbezogene Datum `Observer_Chaining` definiert, das den weiteren Abarbeitungsweg des Unterprozesses festlegt.

Der `Observer_Starter` ist eine automatisierte Aktivität, d. h. er wird ohne Nutzerinteraktion gestartet, abgearbeitet und beendet. Seine Aufgabe ist es den Anwendungsagenten auszuführen und ihm die notwendigen Daten bereitzustellen. Die Datenübergabe und die Funktionsweise des Anwendungsagenten werden später detailliert beschrieben. Nachdem diese Tätigkeit abgeschlossen ist, wird basierend auf dem prozessbezogenen Datum `Observer_Chaining` entweder die Aktivität `Observer_Closer` oder die Tätigkeit `Observer_Chaining_Route` gestartet.

Der `Observer_Closer` hält den Unterprozess im Zustand „wartend“, bis er von einem speziellen Nutzer abgeschlossen wird. Dies sollte durch einen Nutzer geschehen, auf den der in der Prozessbeschreibung definierte `Observer_User` abgebildet wird. Wird diese Abbildung in Enhydra Shark nicht definiert, wird die Aktivität dem Nutzer in den Eingangskorb gestellt, der die Aktivität beendet hat, die als letzte vor dem Unterprozess definiert ist. Das kann zu Problemen führen, da der Nutzer dann die Aktivität beenden kann und somit keine Informationen des entfernten Systems mehr entgegengenommen werden können. Außerdem kann der Unterprozess dann nicht mehr durch den Beobachter abgeschlossen werden.

Im Gegensatz zum `Observer_Closer` ist die Aktivität `Observer_Chaining_Route` keine Aktivität im eigentlichen Sinne. Sie wird automatisch gestartet und sofort beendet. Dafür bietet XPDL eine spezielle Aufgabenform an, die nicht durch einen Nutzer bearbeitet werden muss. Dadurch kann ein externer Prozess gestartet werden und die Abarbeitung des Elternprozesses sofort weitergeführt werden.

Erstellen neuer Unterprozesse Um einen neuen Unterprozess zum Paket `ObserverPacket` hinzuzufügen und mit einem Betriebsobjekt zu verknüpfen, sind die nachfolgend beschriebenen Schritte notwendig.

Als erstes muss der Vorgabeprozess kopiert und mit einem neuen Prozessidentifikator versehen werden. Dies kann durch ein im Installationspaket beinhaltetes Tool erreicht werden. Es wird durch das Skript `createNewXPDLObserver.sh` im Verzeichnis `$SHARK_HOME/sharkinterop/bin` gestartet. Als Kommandozeilenparameter erwartet es den Pfad zum `ObserverPacket` und den Identifikator des neuen Prozesses.

```
createNewXPDLObserver.sh <Pfad zum ObserverPacket> <Name des neuen Prozesses>
```

In der neuen JaWE-Version 1.4.2 kann ein Prozess auch über die grafische Oberfläche kopiert werden.

Nachdem der neue Prozess erstellt wurde, müssen die Eigenschaften an den speziellen Einsatzzweck angepasst werden. Das heißt die zu übergebenden Daten müssen definiert, die Nutzung als verketteter externer Prozess muss an oder abgeschaltet und die Informationen über das zu nutzende Betriebsobjekt müssen definiert werden. Diese Anpassungen sollen mit Hilfe von Enhydra JaWE nun genauer erläutert werden.

Einlesen in JaWE Als erstes muss das `ObserverPacket` in Enhydra JaWE eingelesen und der neu erzeugte Prozess geöffnet werden.

Formale Parameter des Unterprozesses anpassen Nach dem Einlesen müssen die prozessbezogenen Daten des Prozesses definiert werden. Dies sind Eingangsdaten für den externen Prozess, Ausgangsdaten des externen Prozesses und Ein-/Ausgangsdaten. In JaWE geschieht dies über den Reiter „`Formale Parameter`“. Hier müssen alle vom Elternprozess übergebenen Variablen definiert werden. Eine Besonderheit muss bei Nutzung des externen Prozesses als verketteten Prozess beachtet werden. Hier sollten nur Eingabevariablen spezifiziert werden, da keine Rückgabe von Daten erfolgt.

Bearbeiten des Anwendungsagenten Der Anwendungsagent startet den externen Prozess. Deshalb müssen die eben definierten prozessbezogenen Daten an ihn weitergegeben werden. Dazu muss der im Reiter „`Applikationen`“ definierte Anwendungsagent („`ObserverToolAgent`“) bearbeitet werden. Im geöffneten Fenster mit den Eigenschaften des Agenten werden die formalen Parameter angegeben. Diese sollten mit den Parametern des externen Prozesses übereinstimmen. Wichtig ist, dass sowohl die Namen und der Modus *gleich* sind und die Reihenfolge dieselbe ist. Der Ausgabeparameter `Observer_Chaining` sollte unbearbeitet bleiben. Er wird durch das erweiterte Attribut `Chaining` durch den Anwendungsagenten gesetzt.

Neben den Parametern sind hier noch die erweiterten Attribute `FactoryName` und `Chaining` des Anwendungsagenten anzupassen. Der `FactoryName` ist ein Verweis auf die in der Datei `$SHARK_HOME/sharkinterop/conf/observertoolagent.conf` definierte RMI-URI des Betriebsobjekts. Diese URI wird nicht direkt im Prozess gespeichert, da sonst die Anpassung dieser eine nachträgliche Änderung des Prozesses zur Folge hätte. Die in der Datei unter diesem Namen definierte Eigenschaft muss einen gültigen Betriebsobjektidentifikator referenzieren. Das Attribut `Chaining` kann einen der Werte „`true`“ oder „`false`“ annehmen. Ist der Wert „`true`“, wird der externe Prozess gestartet und der Unterprozess beendet. Im Fall von „`false`“ wartet er in der Aktivität `Observer_Closer`, bis der externe Prozess beendet ist.

Anpassen der Aktivität `Observer_Starter` Als letztes muss noch die Aufgabe zum Starten des Anwendungsagenten abgeändert werden, um die Parameter des Prozesses mit denen des Anwendungsagenten zu verknüpfen. Dazu muss im Reiter „`Aktivitäten`“ der `Observer_Starter` bearbeitet werden. Im sich öffnenden Fenster wird dazu im Reiter „`Werkzeuge`“ der `ObserverToolAgent` ausgewählt. Daraufhin können den formalen Parametern des `Anwendungsagenten` die entsprechenden aktuellen Parameter zugewiesen werden. Auch hier ist die Reihenfolge von zentraler Bedeutung.

Speichern Nachdem alle Änderungen vorgenommen wurden kann das Paket gespeichert werden und der neue Prozess einem Elternprozess als Unterprozess zugewiesen werden.

Einbinden erzeugter Unterprozesse Um einen externen Prozess in einen Elternprozess einzubinden sind die folgenden Schritte notwendig. Als erstes muss des Paket `Observer-Packet` als externes Paket in das Paket des Elternworkflows eingebunden werden. Danach

muss ein neuer Unterprozess an der Stelle erzeugt werden, an der der externe Prozess gestartet werden soll. Start- und Beendigungsmodus müssen auf „automatisch“ gesetzt werden. Der Unterprozess wird auf den gewünschten Prozess aus dem `ObserverPacket` abgebildet. Dies kann innerhalb von JaWE durch Setzen der Eigenschaft „`Workflowprozess`“ im Eigenschaftsfenster des Unterprozesses (Reiter „`Subflow`“) erreicht werden. Als letztes müssen die Parameter des Unterprozesses gesetzt werden. Diese müssen in der gleichen Reihenfolge definiert werden, wie die formalen Parameter des Unterprozesses.

4.2.5 Der Anwendungsagent

Der Anwendungsagent (`ObserverToolAgent`) ist die Komponente, mittels derer die Ausführung des entfernten Teilprozesses angestoßen wird. Er wird als automatisierte Aktivität durch die Elternlaufzeitumgebung gestartet. Dabei werden die in der Prozessdefinition definierten Variablen ausgelesen und an das Betriebsobjekt auf dem System der Laufzeitumgebung des externen Prozesses übergeben. Zusätzlich werden alle relevanten Prozesseigenschaften in den Beobachter-Datenspeicher geschrieben[9], um den entsprechenden Beobachter bei einem Ausfall wiederherstellen zu können. Die einzelnen Schritte des Anwendungsagenten werden nachfolgend detailliert erläutert und sind in Abbildung 4.6 dargestellt.

Als erstes liest der Anwendungsagent die Variablen für die Hostname-Port Kombination der RMI-URI des zu startenden Beobachters.
Diese sind in der Datei `$SHARK_HOME/sharkinterop/conf/observertoolagent.properties` als Eigenschaften `rmiregistry.host` und `rmiregistry.port` abgelegt und bilden die Hostname-Port Kombination der URI. Mittels dieser URI und des Identifikators des Unterprozesses innerhalb der Shark-Laufzeitumgebung wird anschließend der Identifikator des Beobachters gebildet.

Als nächstes wird der Kontext des Prozesses für die Übermittlung an das Betriebsobjekt angepasst. Das Datum `Observer_Chaining` wird entfernt, genauso wie die erweiterten Eigenschaften, die bei Start des Anwendungsagenten in einer XML-Repräsentation im Kontext gespeichert wurden.

Danach wird die RMI-URI des Beobachterservers ausgelesen. Dies geschieht analog der Beschreibung in Abschnitt 4.2.2. Sollte keine Verkettung genutzt werden, kann nun mittels der RMI-URI das Beobachterobjekt durch den Beobachterserver erzeugt und beim RMI-Namensdienst registriert werden (Schritte (5) und (6)). Das stellt einen Widerspruch zum erstellten Konzept dar, da dort der Beobachter erst erzeugt wird, wenn die Anfrage zur Erstellung des externen Prozesses an das Betriebsobjekt beendet ist. In der Implementierung ist dies aber notwendig, weil der Beobachter über den Beobachterserver erzeugt wird. Das kann theoretisch zu einem Fehler führen, wenn dieser nicht erreichbar ist. Da die Zurücksetzung der Erzeugung des externen Prozess schwieriger ist als das Abmelden des Beobachters beim RMI-Namensdienst, wird dieser im Prototyp vorher erzeugt. Wenn der externe Prozess als verketteter Prozess gestartet werden soll, ist kein

[9] sofern es sich nicht um einen verketteten externen Prozess handelt

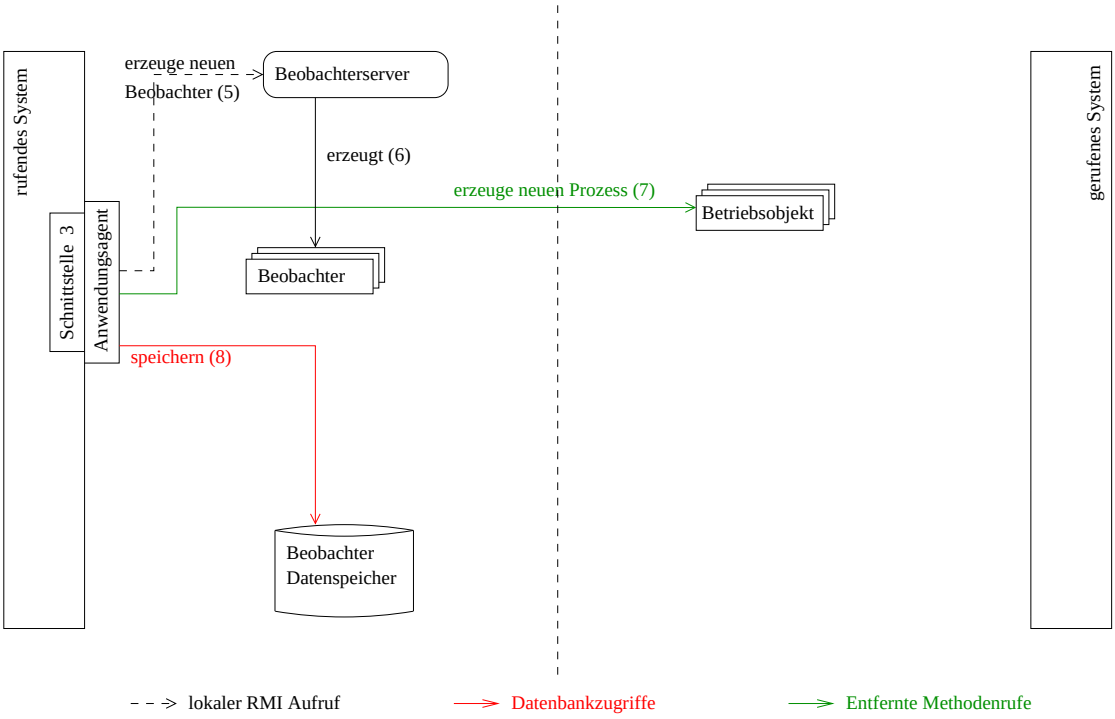

Abbildung 4.6: Implementierung des Anwendungsagenten

Beobachter notwendig, da der Elternprozess in diesem Fall keine weiteren Informationen des externen Prozesses erwartet und in seiner Ausführung fortfährt.

Bevor alle Parameter für die Anfrage zur Erzeugung des externen Prozesses vollständig sind, muss noch der Identifikator des Betriebsobjekts aus der Konfigurationsdatei gelesen werden. Dazu wird das erweiterte Attribut `FactoryName` genutzt, wie dies bei der Beschreibung des Unterprozesses erläutert wurde. Da nun alle Voraussetzungen für das Erzeugen des externen Prozesses erfüllt sind, wird in Schritt (7) die Anfrage zur Erzeugung an das Betriebsobjekt gesendet. Ist die Erzeugung erfolgreich, wird der Identifikator der neu erzeugten Instanz zurückgegeben.

Als letzter Schritt werden die Daten des Beobachters in Schritt (8) der Abbildung in den Beobachter-Datenspeicher geschrieben. Dies geschieht nur, wenn keine Verkettung genutzt wird, da für diesen Fall kein Beobachter erzeugt wurde. Welche Daten das im Speziellen sind, wurde in Abschnitt 4.2.1 näher beschrieben.

Damit ist die Erzeugung des externen Prozesses bzw. Unterprozesses abgeschlossen.

4.2.6 Datenstrukturen der Anfrage an das Betriebsobjekt

Die Anfrage an das Betriebsobjekt beinhaltet den Schlüssel des Beobachters (`ObserverKey`), den Schlüssel des Betriebsobjekts (`FactoryKey`) und die prozessbezogenen Daten.

Diese bestehen aus Eingangs-, Ausgangs- und Ein-/Ausgangsdaten. In Enhydra Shark werden Daten als Java-Datentyp `Map` gespeichert. Dieser Datentyp stellt eine Zuordnung von Name-Wert Paaren dar. Obwohl reine Ausgangsdaten nicht übertragen werden müssten, ist dies in der prototypischen Implementierung notwendig, da das Betriebsobjekt anhand der Variablennamen in der eingehenden `Map` die Variablen im Instanzobjekt-Datenspeicher ablegt.

Die möglichen Datentypen des Kontexts werden jetzt näher vorgestellt. In der XPDL Beschreibung können die folgenden Datentypen beschrieben werden. Die Grundtypen beinhalten die Typen `String`, `Float`, `Integer`, `Reference`, `Boolean`, `DateTime` und `Performer`. Zusätzlich gibt es noch die Typen `ExternalReference`, `Declared Type` und `Schema Type` sowie die als abgelehnt („deprecated“) definierten Typen `Array Type`, `Enumeration Type`, `List Type`, `Record Type` und `Union Type`.

Durch die Nutzung eines Anwendungsagenten zum Erzeugen des externen Prozesses werden durch Enhydra Shark die nutzbaren Datentypen eingeschränkt. Es sind nur die Grunddatentypen und alle nicht „abgelehnten“ Typen nutzbar. Das stellt kein größeres Problem dar, da über die Möglichkeit der Definition eigener Datentypen jeder serialisierbare Java-Datentyp genutzt und an den Agenten übergeben werden kann. Dadurch können die nicht zugelassenen Typen und jeder eigene Typ genutzt werden. Will man eigene Datentypen verwenden so kann z. B. durch die Erweiterung der Schnittstellenklasse `de.raphaelkunis.sharkinterop.datatypes.AbstractExternalData` eine neue Klasse abgeleitet werden.

Dies soll am Beispiel von Kundendaten für den später in den Testfällen genutzten Bestellprozess gezeigt werden. Die abgeleitete Klasse mit den Kundendaten:

```
import de.raphaelkunis.sharkinterop.datatypes.AbstractExternalData;

public class Kunde extends AbstractExternalData {

    private String  kundennummer       = new String("0");
    private String  name               = new String("Mustermann");
    private boolean zahlungsfaehigkeit = false;

    public Kunde() {}

    public String getKundennummer() {
        return kundennummer;
    }
    public void setKundennummer(String kundennummer) {
        this.kundennummer = kundennummer;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean getZahlungsfaehigkeit() {
        return zahlungsfaehigkeit;
    }
```

```
    public void setZahlungsfaehigkeit(boolean zahlungsfaehigkeit) {
        this.zahlungsfaehigkeit = zahlungsfaehigkeit;
    }
}
```

Um dieses Objekt in die prozessbezogenen Daten aufnehmen zu können, muss der Datentyp als deklarierter Typ in die XPDL Beschreibung des Prozesses aufgenommen werden. Dies kann z. B. durch:

```
<TypeDeclaration Id="Kunde" Name="Kundendaten">
  <ExternalReference location="de.raphaelkunis.sharkinterop.testfaelle.Kunde"
                     xref="de.raphaelkunis.sharkinterop.testfaelle.Kunde"/>
</TypeDeclaration>
```

erreicht werden. Danach ist der Java-Datentyp unter dem Identifikator `Kunde` nutzbar und es können beliebig viele Variablen mit diesem Typ erzeugt werden.

```
<DataField Id="Testkunde" IsArray="FALSE" Name="Test Kunde">
  <DataType>
    <DeclaredType Id="Kunde"/>
  </DataType>
</DataField>
```

4.2.7 Das Betriebsobjekt

Ein Betriebsobjekt (`Factory(Impl)`) repräsentiert einen Prozess der Shark-Laufzeitumgebung, der als externer Kindprozess ausgeführt werden kann.

Bei der Erzeugung der einzelnen Betriebsobjekte durch den Betriebsobjektserver werden alle noch nicht beendeten Instanzobjekte aus der Datenbank gelesen und erzeugt. Die genaue Vorgehensweise wurde in Abschnitt 4.2.3 beschrieben. Außerdem werden die Attribute des Betriebsobjekts, wie z. B. Angaben zum repräsentierten Prozess der Shark-Laufzeitumgebung, durch den Betriebsobjektserver gesetzt.

Jedes Betriebsobjekt erzeugt einen Überwachungsthread, der in regelmäßigen Abständen das Vorhandensein nicht übermittelter Nachrichten an den Beobachter prüft und diese gegebenenfalls erneut überträgt.

Nach dem Einlesen der Daten und dem Erzeugen des RMI-Servers können sich Klienten über die RMI-URI bestehend aus *rmi://host:port/factoryname* mit dem entsprechenden Betriebsobjekt verbinden, um den damit verbundenen Prozess der Shark-Laufzeitumgebung zu starten. Die einzelnen Schritte des Betriebsobjekts bei einer Anfrage zur Erzeugung eines externen Prozesses sind in Abbildung 4.7 angegeben und sollen jetzt behandelt werden.

Fordert ein Anwendungsagent die Erzeugung eines Prozesses bei einem Betriebsobjekt an, so prüft dieses zuerst, ob der Empfänger korrekt ist (nämlich das Objekt selbst). Dazu wird der Parameter `receiverKey` der Anfrage ausgewertet. Eine Prüfung des Senders findet

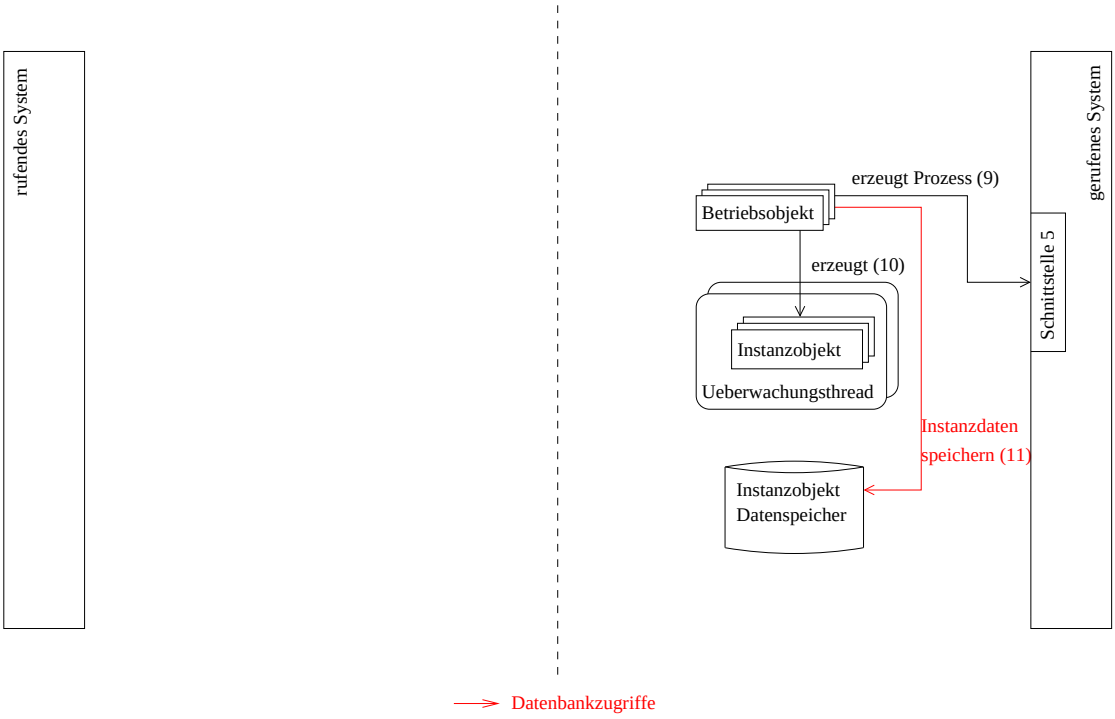

Abbildung 4.7: Implementierung des Betriebsobjekts

nicht statt, da prinzipiell jede Shark-Laufzeitumgebung die Instanziierung des Prozesses des Betriebsobjekts anfragen kann.

Ist der Empfänger gültig, wird der durch die Attribute `xpdlpackage`, `xpdlversion` und `xpdlprocess` beschriebene Prozess in der konfigurierten Shark-Laufzeitumgebung gestartet. Dazu wird das Objekt `de.raphaelkunis.sharkinterop.sharkutils.ProcessStarter` genutzt. Dieses prüft als erstes, ob es in der Shark-Laufzeitumgebung eine entsprechende Prozessdefinition gibt. Ist dies nicht gegeben, wird ein Fehler erzeugt und dem Anwendungsagenten übermittelt. Im Falle der Existenz dieser Prozessdefinition wird als nächstes der Prozess gestartet und, wenn dies gelingt, der in der Anfrage übermittelte Prozesskontext gesetzt. Schlägt die Erzeugung fehl, wird eine Ausnahme an den Anwendungsagenten übermittelt. Sollte der übergebene Kontext nicht zu den prozessbezogenen Daten des Prozesses passen, wird ebenfalls eine Ausnahme übermittelt.

Nachdem der Prozess erfolgreich gestartet wurde (der eben beschriebene Schritt (9) in Abb. 4.7), wird ein neues Instanzobjekt erzeugt und an den Überwachungsthread gebunden (Schritt (10)). Der Identifikator der erzeugten Instanz besteht aus der `host:port` Kombination des Betriebsobjekts und den Identifikator des erzeugten Prozesses durch „/“ getrennt.

Danach werden die Informationen der Instanz in den Instanzobjekt-Datenspeicher geschrieben (Schritt (11)). Als letztes wird der Identifikator der erzeugten Instanz an den

Anwendungsagenten zurückgegeben. Über diesen kann der Beobachter später die Zustandsänderungen des Instanzobjekts gegenprüfen. Im Falle eines verketteten Prozesses entfallen die Schritte 10 und 11.

4.2.8 Das Instanzobjekt

Für jeden durch ein Betriebsobjekt gestarteten Prozess wird ein Instanzobjekt erzeugt. Dieses ist dafür zuständig den Beobachter über Zustandsänderungen zu informieren und ihm bei Abschluss des Prozesses die Rückgabedaten zu liefern. Das Instanzobjekt wird dabei dem Überwachungsthread bekannt gemacht und seine Attribute werden im Instanzobjekt-Datenspeicher abgelegt. Der Zugriff bei Benachrichtigung des Beobachters erfolgt dabei entweder durch die Shark-Laufzeitumgebung (dann werden die im Datenspeicher abgelegten Informationen genutzt) über das Objekt `InstanceShark` oder den Überwachungsthread. In den folgenden Absätzen sollen die beiden Möglichkeiten der Benachrichtigung des Beobachters durch das Instanzobjekt vorgestellt werden. Abbildung 4.8 zeigt die einzelnen Schritte.

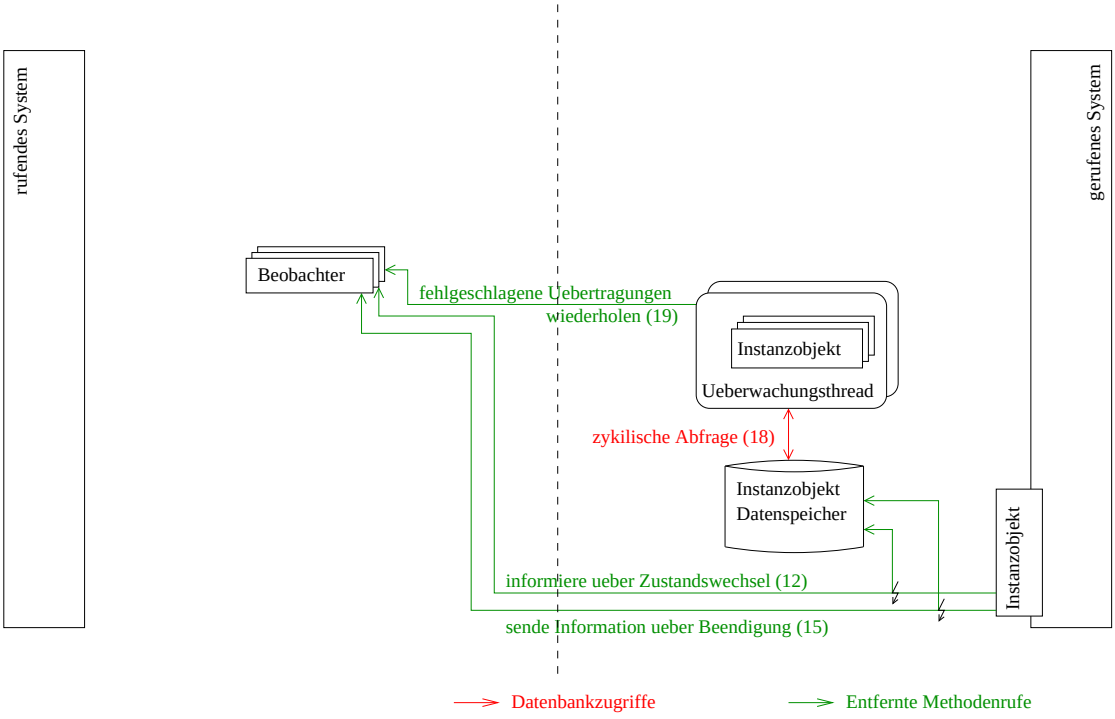

Abbildung 4.8: Implementierung des Instanzobjekts

Bevor die Implementierung des Objekts `InstanceShark` beschrieben werden kann, sollen die Änderungen an zwei internen Objekten von Enhydra Shark beschrieben werden. Diese waren notwendig, da die eigentliche Idee des Konzepts (Benachrichtigung über

Zustandsänderungen durch Nutzung der Schnittstelle 5 des Workflow Referenzmodells) mit Enhydra Shark nicht umsetzbar war.

Um in Enhydra Shark Informationen über Zustandsänderungen von Prozessen zu erhalten, ist es möglich einen sogenannten `Requester` zu nutzen. Über diesen kann ein Prozess erzeugt werden. Die Prozessinstanz kann durch Nutzung der Methode `receive_event` des erzeugenden `Requesters` Zustandsänderungen an diesen weitergeben. In der in dieser Arbeit verwendeten Version 1.0.1 von Enhydra Shark ist es nicht möglich externe `Requester` zu nutzen. Extern bedeutet in diesem Fall, dass anstatt des Standardobjekts ein eigenes implementiert wird. In der aktuellen CVS-Version ist dies dagegen möglich. Allerdings ist auch in der CVS-Version die Weitergabe von Informationen über Zustandsänderungen von Aktivitäten des Prozesses an den `Requester` nicht implementiert. Das heißt selbst bei Nutzung der aktuellsten Version müsste ein Polling-Mechanismus verwendet werden, um zyklisch eventuell stattgefundene Aktivitätszustandsänderungen abzufragen.

In der entwickelten Lösung wurde deshalb die Methode `change_state` der beiden Objekte `WfActivityImpl` und `WfProcessImpl` angepasst. Diese Methode wird bei jeder Zustandsänderung gerufen. Die Änderung umfasst eine zusätzliche Abfrage am Ende dieser Methode, bei der geprüft wird, ob der entsprechende Prozess durch das Betriebsobjekt erzeugt wurde oder nicht. Dazu wird eine Instanz vom Typ `InstanceShark` erzeugt und geprüft, ob der Identifikator des Prozesses im Instanzobjekt-Datenspeicher abgelegt ist. Ist dies der Fall, wird zuerst versucht den Beobachter zu informieren. Schlägt dies fehl, werden die Daten in den Instanzobjekt-Datenspeicher geschrieben, um später durch den Überwachungsthread erneut übertragen zu werden.

Nachdem die Gründe für die Änderungen an internen Klassen von Enhydra Shark und diese Änderungen näher erläutert wurden, soll jetzt die Implementierung des Instanzobjekts `InstaceShark` vorgestellt werden.

Zusatzinformationen über Details der Implementierung dieses Objekts sind in Abschnitt B.2 ab Seite 105 zu finden. Dort wird die Funktionsweise der einzelnen Komponenten anhand ihres Quellcodes verdeutlicht.

InstanceShark Über dieses Objekt werden Informationen über Zustandsänderungen von Prozessen, die durch das Betriebsobjekt gestartet wurden an den zugehörigen Beobachter übermittelt. Dazu werden die Methoden `informObserverActivityStateChanged`, `informObserverStateChanged` und `informObserverComplete` genutzt.

Im Falle der Zustandsänderung einer Aktivität wird das Instanzobjekt dabei durch das Objekt `WfActivityImpl` erzeugt. Nachdem die Prüfung auf einen externen Prozess positiv abgeschlossen wurde, werden die relevanten Daten an das Instanzobjekt übermittelt und es wird versucht den Beobachter über die Zustandsänderung zu informieren (Schritt (12)). Wie schon erwähnt, werden im Falle der Unerreichbarkeit des Beobachters die Daten gespeichert.

Bei einem Zustandswechsel des Prozesses wird geprüft, ob es sich um die Beendigung, den Abbruch oder einen anderen Zustandswechsel handelt. Wird der Prozess nicht beendet oder abgebrochen, ist die Vorgehensweise die gleiche wie bei

einem Zustandswechsel einer Aktivität (nur die übermittelten Daten unterscheiden sich). Wird der Prozess abgebrochen, so wird die Zustandsänderung übermittelt und danach werden alle zu dem entsprechenden Instanzobjekt gehörenden Informationen aus dem Datenspeicher gelöscht (d. h. die Abarbeitung wird als beendet angesehen). Im Falle einer Beendigung des Prozesses wird der Methode `informObserverComplete` der aktuelle Kontext übergeben. Aus diesem werden alle nicht bei der Erzeugung des Prozesses übermittelten Daten entfernt und der Beobachter unter Nutzung der entfernten Methode `completed` über die Beendigung informiert (Schritt (15)). Als letztes werden, wie schon bei Abbruch eines Prozesses, die Instanzdaten aus dem Datenspeicher entfernt.

Neben dem vorgestellten Instanzobjekt für die Benachrichtigung über Zustandsänderungen soll jetzt das Instanzobjekt für die nachträgliche Übermittlung vorgestellt werden. Dieses ist eng mit dem Überwachungsthread verknüpft, weshalb dieser mit beschrieben wird.

InstanceFactory und Überwachungsthread Jedes Betriebsobjekt, das einen nicht verketteten externen Prozess repräsentiert, startet bei seiner Erzeugung einen Überwachungsthread (`FactoryTimerTask`). Dieser prüft zyklisch das Vorhandensein nicht übertragener Benachrichtigungen der Instanzobjekte (Schritt (18)). Das geschieht anhand der durch das Betriebsobjekt initial übergebenen und bei Start eines neuen Prozesses hinzugefügten Instanzobjekte vom Typ `InstanceFactory`. Das Hinzufügen geschieht dabei nicht durch Nutzung einer Methode, sondern durch die Übergabe der Referenz auf eine die Instanzen beinhaltenden Liste bei Start des Überwachungsthreads. Erzeugt das Betriebsobjekt einen neuen Prozess, so fügt es das neue Instanzobjekt zu dieser Liste hinzu.

Der Überwachungsthread prüft bei jeder zyklischen Ausführung zuerst auf beendete Instanzen, danach auf Zustandsänderungen der Aktivitäten, dann auf Zustandsänderungen der Prozesse und als letztes auf Beendigung von Prozessen. Dazu werden die Instanzobjekte der Liste und der Instanzobjekt-Datenspeicher zum Feststellen von nichtübermittelten Änderungen verwendet. Die Übermittlung von Nachrichten an den Beobachter (Schritt (19)) erfolgt dabei analog der Vorgehensweise bei Übermittlung durch das Objekt `InstanceShark`. Allerdings müssen jetzt bei Unerreichbarkeit des Beobachters keine Daten mehr gespeichert werden.

4.2.9 Datenstrukturen der Benachrichtigung des Beobachters

In diesem Abschnitt sollen die Datenstrukturen für die Benachrichtigung des Beobachters vorgestellt werden. Diese sind das `ActivityStateChangedObject` für die Übermittlung von Zustandsänderungen von Aktivitäten und das `ProcessStateChangedObject` für die Übermittlung von Prozesszustandsänderungen. Bei der Beendigung des externen Prozesses wird, wie schon bei der Erzeugung, der Kontext in einer „`Map`“ übertragen. Diese beinhaltet in der aktuellen Version alle bei der Erzeugung übermittelten Daten.

ActivityStateChangedObject Die Datenstruktur zur Übermittlung von Aktivitätszustandsänderungen besteht aus dem Identifikator der Aktivität (`activityId`), ihrem Namen (`activityName`), dem neuen Zustand (`actState`), dem alten Zustand (`lastState`) und dem Zeitpunkt der Zustandsänderung (`stateChangedTime`). Sie werden alle als Java-Datentyp „`String`“ in diesem Objekt gespeichert. Außerdem werden für jedes Attribut Methoden zum Setzen und Auslesen bereitgestellt.

ProcessStateChangedObject Das Objekt zur Übermittlung von Prozesszustandsänderungen ist dem für Zustandsänderungen von Aktivitäten sehr ähnlich. Es besitzt die Attribute `actState`, `lastState` und `stateChangedTime`. Auch hier werden wieder Methoden zum Setzen und Auslesen aller Attribute definiert.

4.2.10 Das Beobachterobjekt

Das Beobachterobjekt ist das Bindeglied zwischen dem externen Prozess und seiner lokalen Repräsentation. Er wird im Falle eines nicht verketteten externen Prozesses durch den Anwendungsagenten gestartet. Die Funktionen des Beobachters sind in Abb. 4.9 dargestellt.

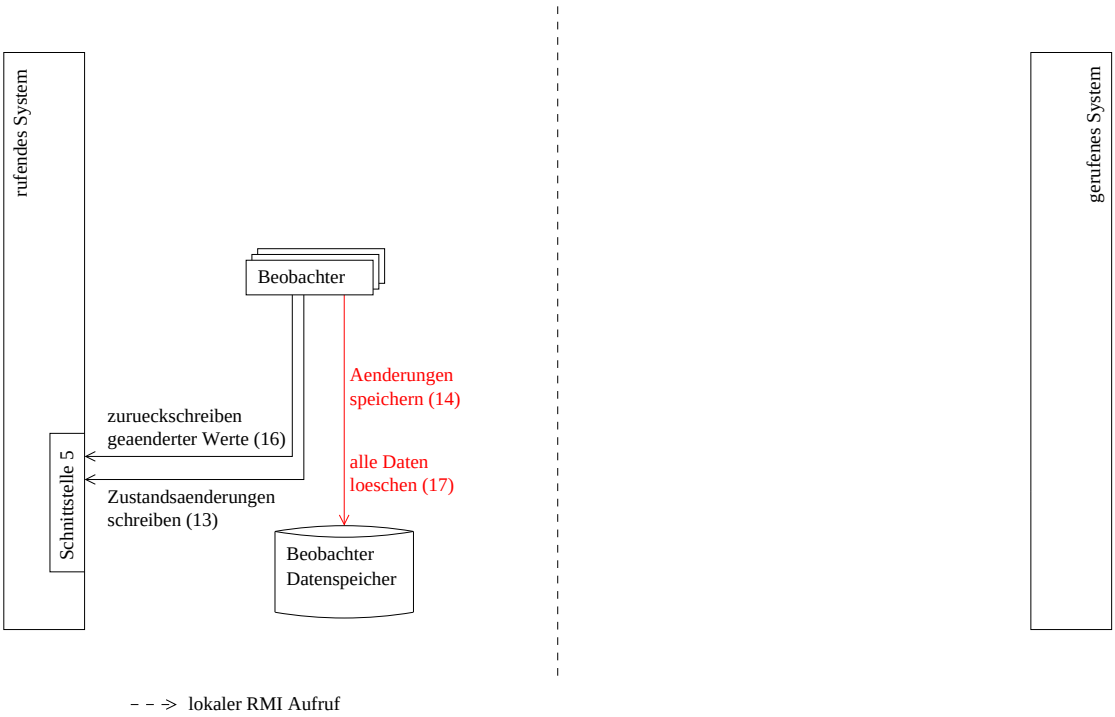

Abbildung 4.9: Implementierung des Beobachterobjekts

Wird eine Zustandsänderung durch das Instanzobjekt an den Beobachter übermittelt, gibt er diese an die lokale Repräsentation weiter (Schritt (13) in der Abbildung) und legt

sie im Beobachter-Datenspeicher ab (Schritt (14)). Die Weitergabe an die lokale Repräsentation bedeutet in diesem Zusammenhang das Setzen der folgenden prozessbezogenen Daten:

- `externalProcessState`,
- `externalProcessLastState` und
- `externalProcessStateChangedTime` bei Änderung des Prozesszustands.
- `externalProcessActivityName`,
- `externalProcessActivityState`,
- `externalProcessActivityLastState` und
- `externalProcessActivityStateChangedTime` bei Änderung des Zustands einer Aktivität des externen Prozesses.

Für die Übermittlung von Zustandsänderungen bietet der Beobachter zwei entfernte Methoden an. Das sind `activityStateChanged` für Zustandsänderungen der Aktivitäten und `processStateChanged` für Zustandsänderungen des Prozesses.

Bei beiden Methoden werden zuerst der Sender (`senderKey`) und Empfänger (`receiverKey`) überprüft. Danach wird über das Objekt `ProcessUpdater` die Information an die lokale Repräsentation übergeben. Bei einer Prozesszustandsänderung wird zusätzlich geprüft, ob es sich um einen Abbruch des externen Prozesses handelt. Dazu wird geprüft, ob der aktuelle Zustand „`terminated`“ oder „`aborted`“ ist. Ist das der Fall, wird nicht die Information über eine Zustandsänderung in die prozessbezogenen Daten geschrieben, sondern der Zustand des lokalen Unterprozesses auf diesen Zustand gesetzt. Das hat zur Folge, dass der Unterprozess in der Shark-Laufzeitumgebung des rufenden System auch abgebrochen wird. Zusätzlich werden wie bei einer Beendigung des externen Prozesses alle Daten aus dem Beobachter-Datenspeicher gelöscht und der Beobachter beim RMI-Namensdienst abgemeldet.

In diesem Zusammenhang soll noch auf einen Fehler in der Shark Implementierung hingewiesen werden. Wird der Unterprozess in den Zustand „Abgebrochen“ (`terminated` oder `aborted`) versetzt, bleibt der Elternprozess im Zustand „Laufend“. Er kann aber nicht fortgesetzt werden, da der Unterprozess nicht beendet wurde. Theoretisch müsste in diesem Fall entweder der Elternprozess mit dem Unterprozess abgebrochen werden oder die Ausführung des Prozesses an den Anfang des Unterprozesses zurückspringen. Laut Mailingliste wird dieses Problem in einer kommenden Version von Enhydra Shark behoben.

Wird ein externer Prozess beendet, kann dies dem Beobachter über die entfernte Methode `completed` übermittelt werden. Diese prüft als erstes wieder Sender und Empfänger. Anschließend werden die übermittelten prozessbezogenen Daten mit den im Datenspeicher bei Erzeugung des externen Prozesses gespeicherten Modi abgeglichen. Das heißt es werden alle Daten, die Eingangsdaten des externen Prozesses waren, aus dem übermittelten Kontext entfernt. Danach wird durch das Objekt `ProcessCloser` der Kontext des lokalen Unterprozesses aktualisiert und dieser beendet (Schritt (16)). Als letzter Schritt werden alle zu diesem Prozess im Beobachter-Datenspeicher abgelegten Daten entfernt (Schritt (17)) und das Beobachterobjekt beim RMI-Namensdienst abgemeldet.

Bei Abschließen des Prozesses ist es wichtig, dass der in XPDL Beschreibung des Beobachterprozesses definierte Nutzer `Observer_User` auf den in der Konfigurationsdatei `sharkinterop.conf` definierten Nutzer `observer.user` abgebildet wird. Ansonsten ist es dem Beobachter nicht möglich den lokalen Unterprozess abzuschließen.

4.2.11 Gesamtbild der Implementierung

Eine Übersicht der einzelnen beschriebenen Komponenten und ihrer Beziehungen zueinander ist in Abb. 4.10 dargestellt.

Auf eine Beschreibung desr Abbildung wird an dieser Stelle verzichtet, da die einzelnen Teile in diesem Abschnitt bereits detailliert beschrieben wurden.

4.3 Bemerkungen zur Implementierung

In diesem Abschnitt werden die Implementierung und damit verbunden Teile des Konzepts zusammenfassend betrachtet. Dabei werden die Möglichkeiten des Systems vorgestellt und Vor- und Nachteile der Lösung aufgezeigt.

Implementierte Szenarien der Interoperabilität Mit dem implementierten Prototypen werden die Szenarien 1 (Modell der verbundenen eigenständigen Prozesse) und 2 (Modell der hierarchischen Unterprozesse) der Interoperabilität von Workflow Management Systemen für Enhydra Shark realisiert. Die in den Szenarien 3 (Modell der gemeinsam genutzten Domäne) und 4 (Modell der parallelen Synchronisation) beschriebenen verteilten Abarbeitungsmechanismen sind in der derzeitigen Realisierung der Schnittstelle nicht ohne Zusatzaufwand möglich. In Abschnitt 6.3 werden Möglichkeiten vorgestellt, wie Szenario 3 sowohl ohne Änderungen am Prototypen als auch mit Änderungen realisiert werden könnte.

Ausfallsicherheit Bei der Implementierung wurde großer Wert auf Ausfallsicherheit gelegt. Es ist möglich auf Ausfälle der Serverobjekte (Beobachter und Betriebsobjekt) zu reagieren. Die Wahrscheinlichkeit von Datenverlust wurde dadurch minimiert. Im Falle der Unerreichbarkeit des Beobachterobjekts und eines Ausfalls der Datenbank für den Instanzobjekt-Datenspeicher ist Datenverlust trotzdem möglich. Da aber der Ausfall des Datenspeichers sehr unwahrscheinlich ist (dieser wird lokal genutzt, also ist keine Kommunikation über ein Netzwerk notwendig) sollte dies akzeptabel sein.

Durch die implementierten Mechanismen zur Ausfallsicherheit ist es weiterhin möglich, die Serverobjekte jederzeit abzuschalten und zu einem späteren Zeitpunkt neu zu erzeugen. Dadurch können die Serverrechner z. B. nachts abgeschalten werden, ohne dass die Funktionalität des Gesamtsystems beeinträchtigt wird.

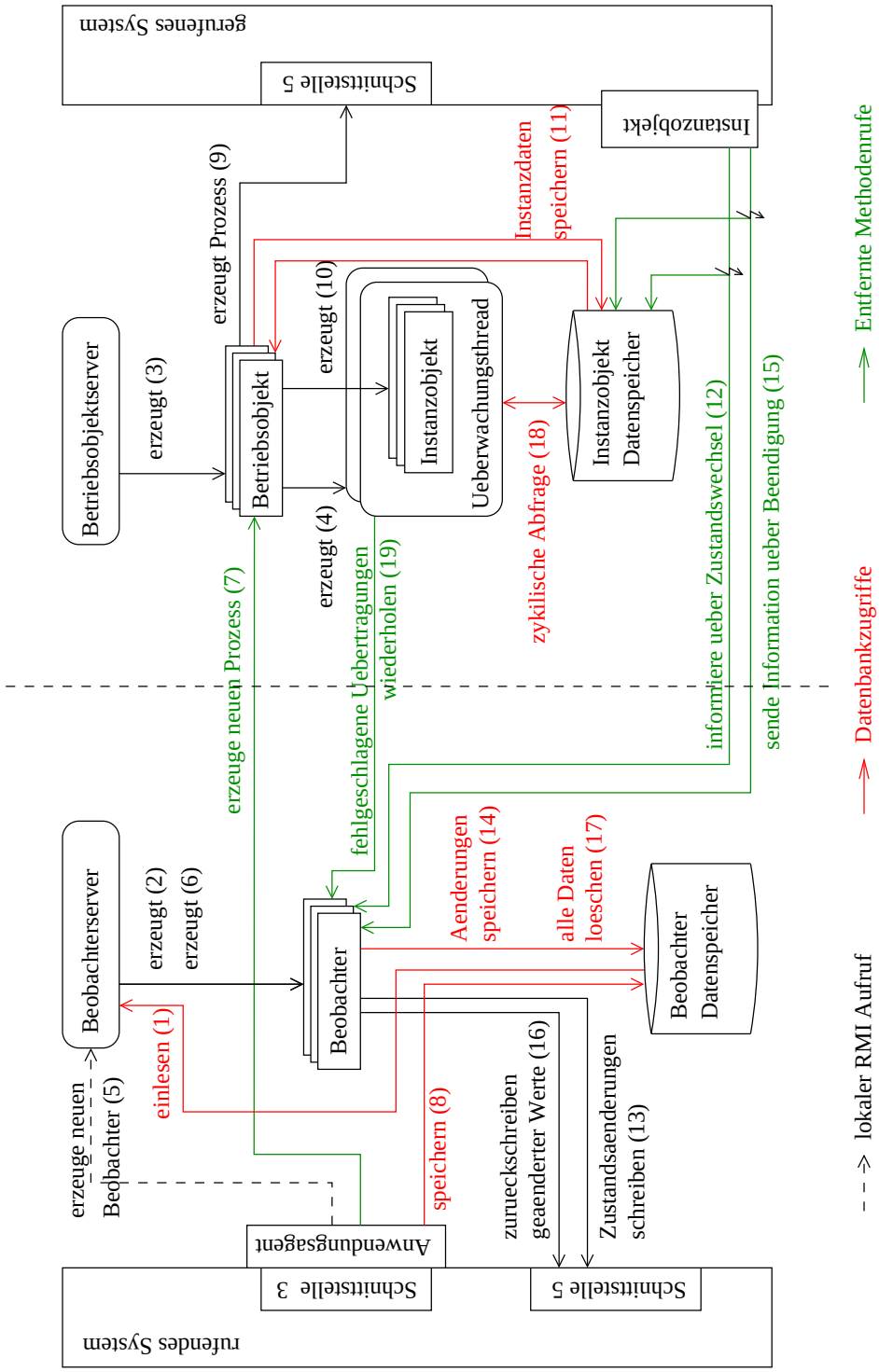

Abbildung 4.10: Gesamtbild der Implementierung

Kommunikation Durch die Nutzung von RMI statt SOAP[10] als Middleware-Technologie wurde die Implementierung der Schnittstelle für Enhydra Shark erleichtert. Es musste keine Zwischenschicht zur Datenabbildung eingefügt werden, da mit den Java-Typen gearbeitet werden konnte.

Nachteil der Nutzung von RMI ist, dass die Programmiersprachenunabhängigkeit verloren gegangen ist. Die Schnittstelle kann nur dann zwischen zwei Systemen genutzt werden, wenn beide in Java implementiert sind.

Einhergehend mit der Nutzung von RMI ist es möglich, über Sicherheitsrichtlinien (Policies) die Systeme, die die Objekte nutzen dürfen, einzuschränken. Weiterhin ist es, durch Austausch der von RMI genutzten Socket-Objekte, möglich, die Daten über eine verschlüsselte Verbindung zu übertragen.

Auslagerung des Betriebsobjektschlüssels aus der Prozessbeschreibung Die Auslagerung des Betriebsobjektidentifikators aus der Prozessbeschreibung der lokalen Repräsentation ermöglicht es, das Betriebsobjekt während der Laufzeit des rufenden Systems auszutauschen. Außerdem ist dadurch, z. B. bei Wechsel des Rechners auf dem das Betriebsobjekt ausgeführt wird, keine Änderung an der Prozessbeschreibung des Elternprozesses notwendig.

Durch die Änderung des Identifikators zur Laufzeit ist ein Szenario denkbar, in dem, anhand der Abarbeitungsdaten der externen Prozesse im Beobachter-Datenspeicher, eine Lastverteilung möglich ist. So könnte der externe Prozess bei jedem neuen Lauf auf einem anderen System und somit z. B. von einer anderen Firmenabteilung abgearbeitet werden. Dazu könnte, anhand der Daten, durch ein zusätzliches Tool mittels eines festzulegenden Kriteriums der Betriebsobjektschlüssel geändert werden.

Änderung interner Klassen von Enhydra Shark Normalerweise sollten Anwendungen, die Zusatzfunktionalitäten zu einem bestehenden System hinzufügen, das System nicht ändern. Allerdings war dies für die Schnittstelle bei Enhydra Shark nicht realisierbar. Die Gründe dafür wurden in Abschnitt 4.2.8 schon näher erläutert. Die

Datenübertragung Bei der Datenübertragung werden derzeit, sowohl bei Start des externen Prozesses als auch bei dessen Beendigung, alle spezifizierten Daten (Eingabe-, Ausgabe- und Ein-/Ausgabedaten) übertragen. Das heißt bei Start des Prozesses werden die Ausgabedaten mit übertragen und bei seiner Beendigung die Eingabedaten. Diese müssten aber normalerweise nicht übertragen werden. Der Grund für die Übertragung der nicht notwendigen Daten ist, dass der Kontextparameter die Modi nicht enthält. Die Modi der Eingabe- und Rückgabedaten werden im Standard im Betriebsobjekt abgelegt (`ContextDataSchema`, `ResultDataSchema`) und können über die Funktion `getProperties()` abgerufen werden. Da das Betriebsobjekt im Prototypen keine Kenntnisse über die Daten und ihre Modi besitzt, werden diese Informationen, durch den Anwendungsagenten, im Beobachter-Datenspeicher abegelegt.

[10] im ASAP-Protokoll als Übertragungsmechanismus definiert

Da das Betriebsobjekt anhand der eingehenden Daten die Felder der Eingabe-, Ausgabe-, Ein-/Ausgabedaten des Prozesses im Instanzobjekt-Datenspeicher anlegt[11], müssen die Ausgabedaten mit übertragen werden. Auch das Instanzobjekt hat keine Informationen über den Modus eines Datums (dieser ist nur im Beobachter-Datenspeicher abgelegt). Deshalb muss es alle eingegangenen Daten zurückübertragen. Aus diesen Daten werden vor dem Zurückschreiben in den Prozesskontext des Elternprozesses vom Beobachter alle Eingangsdaten des externen Prozesses entfernt.

Anwendungsdaten Die entwickelte Schnittstelle bezieht in die Datenübertragung derzeit nur prozessbezogene Daten ein. Der Grund dafür ist, dass Anwendungsdaten nicht durch das WfMS zugreifbar bzw. ihm nicht bekannt sind. Sollte ein externer Prozess Anwendungsdaten nutzen, die auf dem System des Elternprozesses vorliegen, so müssen diese durch einen geeigneten Mechanismus übermittelt werden. Das Problem ist, dass es sich bei Anwendungsdaten im Regelfall um Daten in einer Datenbank oder Dokumente handelt. Für die Übertragung größerer Dateien ist RMI nicht geeignet. Auch für Daten einer Datenbank ist eine Übertragung schwierig. Für eine gemeinsame Nutzung von Anwendungsdaten wäre daher z. B. eine Möglichkeit diese auf einem zentralen Serverrechner abzulegen, auf den alle WfMS Zugriff haben.

Will man die Daten durch Nutzung der Schnittstelle übertragen, so müssten diese Enhydra Shark über die prozessbezogenen Daten bekannt gemacht werden. Das könnte z. B. durch einen selbstdefinierten Java-Datentyp geschehen, der aus einer Angabe über den „Ort“ der Daten (z. B. Dateisystempfad) und einem Byte-Array besteht. Wird der externe Prozess gerufen, müsste ein Tool die Daten vor dem Senden in den Byte-Array schreiben. In der Umgebung des externen Prozesses müsste dann eine Beschreibung vorliegen, an welchen „Ort“ die Daten auf diesem System abgelegt werden sollen. Ein Tool auf diesem System müsste sie dann an diesen schreiben.

Konfigurationsaufwand In der Implementierung ist die Konfiguration umständlich und hat den Nachteil, dass einige Einstellungen mehrfach vorgenommen werden müssen. Die Angabe über die Verkettung muss z. B. sowohl in der Prozessbeschreibung des Elternprozesses als auch in der Konfiguration des Betriebsobjekts angegeben werden. Das kann bei falschen Angaben zu Fehlern führen. Wird die Einstellung zur Verkettung bei der Konfiguration des Betriebsobjekts falsch gesetzt, kann es passieren, dass versucht wird einen nicht existierenden Beobachter zu informieren oder ein existierender Beobachter wird nie informiert und der Unterprozess des Elternprozesses somit nie abgeschlossen.

Hier könnte entweder ein Hilfsprogramm zur Erstellung der Konfigurationsdatei implementiert werden oder die entfernten Methoden müssten abgeändert werden und Konfigurationsinformationen berücksichtigen.

Abbruch des externen Prozesses Die verwendete Shark Version, enthält einen Fehler bei Abbruch des externen Prozess. Wird dieser in den Zustand „Abgebrochen“ versetzt,

[11] ohne Modusinformation

so wird zwar der Unterprozess abgebrochen, der Elternprozess bleibt aber im Zustand „Laufend“. Das führt zu einem nicht fortsetzbaren Prozess, der nur durch den Administrator abgebrochen werden kann.

Änderungen des Konzepts in der Implementierung Obwohl im Konzept keine Implementierungsdetails angegeben und nur die Funktionalität und die zentralen Komponenten der Schnittstelle definiert wurden, war es notwendig zwei Änderungen des Konzepts in der Implementierung zu vollziehen. Die Änderungen wurden bei der Behandlung der einzelnen Komponenten bereits erwähnt und an dieser Stelle sollen die Gründe erläutert werden, warum sie nicht nachträglich in das Konzept aufgenommen wurden.

Die erste Änderung betrifft den Zeitpunkt des Startens des Beobachters. Im Konzept wird der Beobachter erst erzeugt, wenn der externe Prozess gestartet ist. In der Implementierung erfolgt die Erzeugung jedoch vorher. Der Grund dafür ist, dass es einfacher ist die Erzeugung des Beobachters zurückzusetzen als die des extern gestarteten Prozesses, wie in Abschnitt 4.2.5 auf Seite 59 erwähnt wird. Diese Änderung wurde nicht in das Konzept übernommen, da sie vollzogen wurde, weil die Beobachter durch den Aufruf einer entfernten Methode des Beobachterservers erzeugt werden müssen. In der Implementierung ist dieser entfernte Methodenaufruf notwendig, da Enhydra Shark als Bibliothek genutzt werden kann. Eine ausführliche Begründung wurde in Abschnitt 4.1 auf Seite 45 gegeben. Dies muss bei einem anderen WfMS jedoch nicht der Fall sein.

Die zweite Änderung betrifft die Art und Weise, wie Zustandsänderungen an das Instanzobjekt übermittelt werden. In Enhydra Shark war die Nutzung von Schnittstelle 5 des Workflow Referenzmodells nicht möglich. Die Gründe dafür sind in Abschnitt 4.2.8 auf Seite 65 angegeben. Da in das Konzept keine Aussagen über das verwendete Workflow Management System und Details für die Implementierung aufgenommen wurden[12] und das Konzept allgemein für eine Schnittstelle zwischen WfMS nutzbar sein soll, wurde dieses Detail nicht nachträglich in das Konzept übernommen. Das Instanzobjekt (`InstanceShark`) wurde aber so implementiert, dass eine nachträgliche Änderung der Implementierung ohne Änderungen an diesem Objekt vollzogen werden kann. Die einzige Änderung wäre, dass statt Shark-interne Klassen anzupassen ein Objekt (z. B. der `Requester`) das Instanzobjekt temporär erzeugt. Das Instanzobjekt sendet die Informationen dann in der gleichen Weise an den Beobachter wie in der derzeitigen Implementierung.

[12] Ausnahme: Attribute für eine eindeutige Identifikation des repräsentierten Prozesses im Betriebsobjekt

5 Testfälle

In diesem Kapitel sollen verschiedene Testfälle vorgestellt werden, um die Funktionen der Schnittstelle zu demonstrieren. Es werden insgesamt vier Testfälle erstellt und dokumentiert.

Im ersten Test wird ein verketteter Prozess auf einem entfernten System erzeugt. Der zweite Testfall zeigt die Funktionsweise der Schnittstelle an einem externen Unterprozess, der Daten empfängt und diese verändert zurückliefert. Beim dritten Testfall wird von einem System S1 ein Unterprozess auf System S2 gestartet. Dieser startet während seiner Abarbeitung einen dritten Prozess auf einem Laufzeitsystem S3. Der vierte und letzte Testfall demonstriert die Möglichkeiten bei temporärem Ausfall des Betriebsobjekts und des Beobachters.

Bei den Testfällen 1, 2 und 4 soll durch S1 das System des Elternprozesses und S2 das des gestarteten externen Prozesses repräsentiert werden. Die Konfigurationsdateien für die Betriebsobjekte und Anwendungsagenten der einzelnen Testfälle befinden sich im Anhang in Kapitel D.

Um die Testfälle zu erläutern, werden die Ausgaben der einzelnen Komponenten in ihren Logdateien in verkürzter Form verwendet. Diese haben den folgenden Aufbau:

`<DATUM> <ART DES EINTRAGS> [<KOMPONENTE>] - <OBJEKT> -> Nachricht`

Da die genauen Ausgaben zu lang wären, sind sie nur auf der beigelegten CD zu finden. Die für die Arbeit aufbereiteten Ausgaben bestehen deshalb aus:

`<Nummer>: Nachricht`

In der `Nachricht` wurden ausserdem die Identifikatoren der einzelnen Objekte (Intanz-, Beobachter- und Betriebsobjekt) ersetzt. Ein Beobachteridentifikator wird als `OKEY`, ein Instanzidentifikator als `IKEY` und ein Betriebsobjektidentifikator als `BKEY` dargestellt. Sind mehrere Objekte vom gleichen Typ in einem Testfall vorhanden, ist eine Unterscheidung durch eine angefügte Zahl möglich.

5.1 Testfall 1: Verkettung von Prozessen

Zur Verkettung von Prozessen wird ein Beispielworkflow genutzt, bei dem eine Bestellung aufgenommen und der Versand der bestellten Ware als externer Prozess gestartet wird. Der Workflow wird in Abbildung 5.1 als Aktivitätsnetzwerk dargestellt.

Als erste Aufgabe (`Kundendaten aendern`) müssen die Kundendaten eingegeben werden. Damit soll gleichzeitig die Möglichkeit zur Nutzung serialisierbarer Java Klassen in den prozessbezogenen Daten aufgezeigt werden. Die Kundendaten basieren auf der in Abschnitt 4.2.6 gegebenen Klasse `Kunde`. Diese wurde um den `vorname` und die `anschrift`

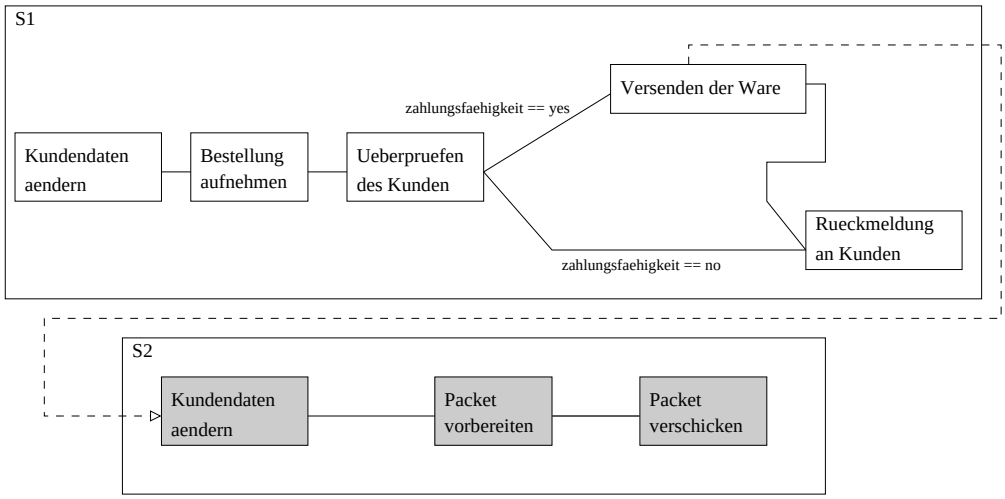

Abbildung 5.1: Testfall 1: Aktivitätsnetzwerk

erweitert. Die Eingabe der Kundendaten sollte in einem real eingesetzten Prozess durch Abfrage einer Datenbank zum Auslesen der Kundendaten ersetzt werden. Nachdem die Kundendaten eingelesen wurden, kann der Nutzer in der Tätigkeit `Bestellung aufnehmen` zwei zu bestellende Artikel mit Namen und Menge angeben. Die Überprüfung der Kundendaten wird durch die Aufgabe `Ueberpruefen des Kunden` durch einen Mitarbeiter der Firma erledigt und anhand des Attributs `Kunde.zahlungsfaehigkeit` die nachfolgende Abarbeitung bestimmt. Ist der Kunde zahlungsfähig, wird die Bestellung in einem entfernten Lager verpackt und versendet (`Versenden der Ware`) und der Bearbeiter erhält die Aufgabe den Kunden über den Versand zu informieren (`Rückmeldung an Kunden`). Im Falle der Zahlungsunfähigkeit erhält der Mitarbeiter nur die Aufgabe, den Kunden über die Nichtzustellung, aufgrund von Zahlungsunfähigkeit, zu benachrichtigen.

Als Nachweis der Funktionstüchtigkeit des erstellten Prototypen werden nachfolgend die Ausgaben der einzelnen Objekte und Screenshots der Applikation auf den Systemen S1 und S2 dargestellt. Die Ausgaben sind in Tabelle 5.1 dargestellt.

Ablauf des Testfalls: Als erstes erfolgte die Eingabe der Kundendaten. Diese ist nachfolgend angegeben.

```
Kunde (0):
  Name: Mustermann  Vorname: Max  Anschrift: Musterstrasse 12
  Zahlungsfaehigkeit ist NICHT gegeben
Wollen Sie die Daten aendern?
Y
Kundennummer:
1
Kundenname:
Kunis
Kundenvorname:
Raphael
```

System S1	System S2	
Beobachter	Betriebsobjekt	Instanzobjekt
0: Registriere den ObserverServer ...	0: Keine gespeicherte Instanzen geladen, da es sich um einen verketteten Prozess handelt	
0: Erzeuge und registriere gespeicherte Observer	0: Starte keinen Timer, da es sich um einen verketteten Prozess handelt	
	0: Factory ... gebunden - Key = **FKEY**	
1: Beobachterschluessel erfolgreich erzeugt		
2: Kein Observer gestartet, da es sich um einen verketteten Unterprozess handelt		
3: Kontaktiere Betriebsobjekt (**FKEY**)		
	4: Observer (**OKEY**) hat Instanzerzeugung angefragt	
	5: neuer Prozess hat die Id: **PID**	
	6: Keine Instanz erzeugt und keine Datenbankeintraege vorgenommen, da es sich um einen verketteten Prozess handelt	keine erzeugt
7: Keine Daten gespeichert, da es sich um einen verketteten Unterprozess handelt		
8: Externe Prozessinstanz (**IKEY**) erfolgreich gestartet		

Tabelle 5.1: Ausgaben der Objekte bei Testfall 1

```
Kundenanschrift:
Teststrasse 61
Zahlungsfaehig? (Y) oder (N):
Y

Kunde (1):
  Name: Kunis  Vorname: Raphael  Anschrift: Teststrasse 61
  Zahlungsfaehigkeit ist gegeben
```

Nachdem der Kunde (`Kunde(1)`) angelegt wurde, ist die Bestellung durch Eingabe der Namen und Mengen von zwei Artikeln erfolgt. Diese Daten sind in Abbildung 5.2 dargestellt.

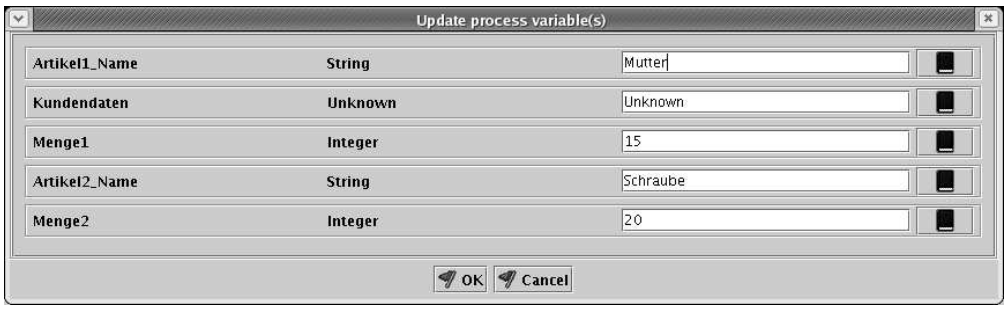

Abbildung 5.2: Testfall 1: Eingegebene Daten S1

Es wurden 2 Artikel (`Mutter` und `Schraube`) in den Mengen (`15` und `20`) bestellt. In der Abbildung wird der Typ und Wert der Kundendaten als „`Unknown`“ durch Enhydra Shark angegeben. Das liegt daran, dass der Administrationsapplikation nur die in XPDL beschreibbaren Datentypen für die Ausgabe zur Verfügung stehen. Im Falle einer Definition anderer Typen muss die Schnittstellenapplikation zwischen Nutzer und Shark-Laufzeitumgebung die definierten Typen deshalb kennen und behandeln können.

Nachdem die Bestellung aufgenommen wurde, erfolgte die Überprüfung des Kunden. Diese wurde hier durch das Attribut `zahlungsfaehigkeit` des Objekts `Kundendaten` vorgenommen. Die Zahlungsfähigkeit war gegeben, wie dies in der Ausgabe der Kundendatenänderung für `Kunde(1)` zu sehen ist. Danach erfolgte die Erzeugung des externen Prozesses.

In Ausgabe 0 in Tabelle 5.1 wurden vorher die beiden Server gestartet. Dabei hat der Betriebsobjektserver weder den Überwachungsthread gestartet, noch versucht gespeicherte Instanzen zu laden. Dies ist nicht notwendig, da für den externen Unterprozess keine Instanzobjekte angelegt werden und auch keine Informationen an den Beobachter übermittelt werden müssen. Die Ausgaben 1 bis 3 zeigen die Vorgänge des Anwendungsagenten, die nach positiver Überprüfung der Zahlungsfähigkeit und automatischem Start des Unterprozesses erfolgten.

Ausgabe 4 zeigt, dass der Anwendungsagent die Erzeugung angefragt hat. Die Ausgaben 5 und 6 zeigen die darauffolgenden Aktionen des Betriebsobjekts. Dieses startete

den repräsentierten Prozess (Ausgabe 5) und erzeugte weder ein Instanzobjekt, noch speicherte es Daten in den Instanzobjekt-Datenspeicher (Ausgabe 6).

Die Ausgaben 7 und 8 zeigen, dass der Anwendungsagent die Rückmeldung über die positive Erzeugung des externen Prozesses erhalten hat und seinerseits keine Daten in den Beobachter-Datenspeicher schreibt. Damit wurde die Erzeugung des externen Prozesses beendet.

Auf System S2 wurde durch den Start des neuen Prozesses die automatische Aktivität „`Kundendaten aendern`“ gestartet, die die übermittelten Kundendaten ausgibt. Dabei wurden die folgenden Daten ausgegeben:

```
Kunde (1):
  Name: Kunis  Vorname: Raphael  Anschrift: Teststrasse 61
  Zahlungsfaehigkeit ist gegeben
Wollen Sie die Daten aendern?
N
```

Wie zu sehen ist, sind diese Daten mit denen auf System S1 identisch. Dieser selbstdefinierten Datentyp wurde also korrekt übermittelt.

Um nachzuweisen, dass auch die anderen Daten korrekt übermittelt wurden, ist in Abbildung 5.3 ein Screenshot der Administrationsapplikation auf System S2 dargestellt.

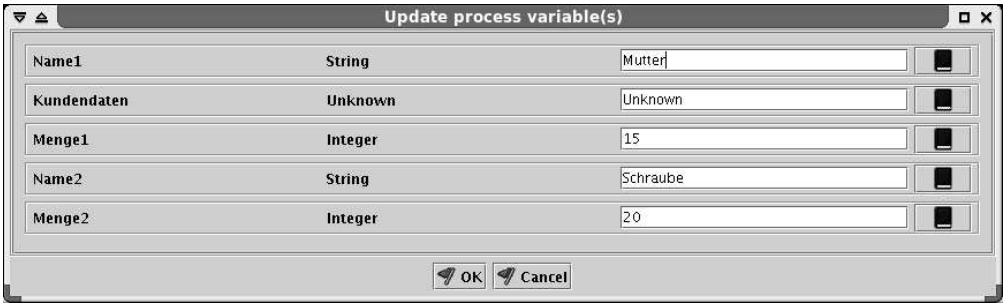

Abbildung 5.3: Testfall 1: Übergebene Daten von S1 an S2

Resultat: Mit diesem Testfall wurde gezeigt, dass es mit dem entwickelten Prototypen möglich ist verkettete externe Prozesse zu starten. Weiterhin wurde aufgezeigt, dass eigene Datentypen an den externen Prozess übermittelt werden können.

5.2 Testfall 2: Externer Unterprozess mit Rückgabedaten

Für den Test des Prototypen bei nicht verketteten externen Unterprozessen mit Rückgabedaten wird das Beispiel aus Testfall 1 etwas verändert. In Abbildung 5.4 ist das Aktivitätsnetzwerk dieses Prozesses dargestellt.

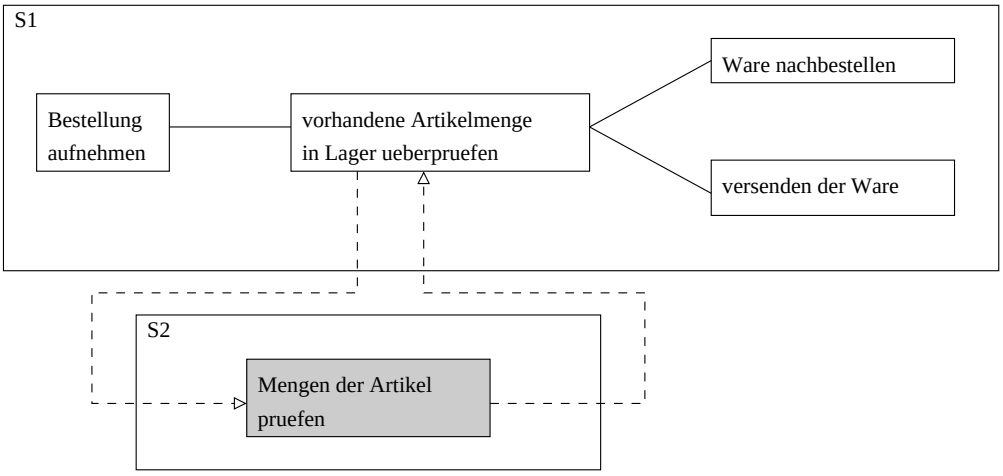

Abbildung 5.4: Testfall 2: Aktivitätsnetzwerk

Es wird weiterhin eine Bestellung aufgenommen. Allerdings wird in diesem Testszenario auf dem externen System die Artikelmenge überprüft. Dazu erhält der Unterprozess die Namen und Mengen der zwei bestellten Artikel als Eingabedaten. Zusätzlich werden die Differenzen zwischen bestellter und tatsächlich vorhandener Menge der Artikel als Ein-/Ausgabedaten definiert (`Differenz1` und `Differenz2`). Der Eingabewert der beiden Variablen ist dabei der Wert 0. Reine Ausgabedaten werden durch zwei Wahrheitswerte zurückgegeben (`Mengeausreichend1` und `Mengeausreichend2`). Je nach Wert der beiden Wahrheitswerte, wird entweder Ware nachbestellt oder die Ware versendet.

Die Ausgaben der Objekte der beiden Systeme sind in den Tabelle 5.2 dargestellt. Dabei soll an dieser Stelle noch darauf hingewiesen werden, dass für jede Aktivität zwei Zustandsänderungen erfolgen. Die erste Zustandsänderung beschreibt den Übergang von „Initialisiert“ zu „Laufend“. Die Zweite den Übergang von „Laufend“ zu „Beendet“.

Ablauf des Testfalls: In diesem Testfall wurde wieder eine Bestellung aufgenommen. Dazu wurden `150` Stück von Artikel 1 (`Mutter`) und `100` Stück von Artikel 2 (`Schraube`) bestellt. Die Differenz zwischen den bestellten Mengen und den tatsächlich vorrätigen beträgt vor der Überprüfung für beide Artikel 0. Die Ausgabedaten des externen Unterprozesses (`Mengeausreichendx`) sind beide wahr und werden erst durch den Unterprozess gesetzt. Eine Änderung der Werte dieser Variablen hat keinen Einfluss auf den weiteren Verlauf des Prozesses. Die Eingabedaten sind in Abb. 5.5 dargestellt.

Nachdem die Aktivität „`Bestellung aufnehmen`“ abgeschlossen war, wurde der externe Unterprozess gestartet. Die Ausgaben der Objekte in Tabelle 5.2 beginnen allerdings mit dem Start der beiden Serverobjekte.

In Ausgabe 0 des Beobachters ist zu sehen, dass der Beobachterserver gestartet wurde und er versucht hat, bereits erzeugte Beobachterobjekte zu laden. Das ist im Gegensatz zu Testfall 1 bei der Nutzung nicht verketteter externer Prozesse notwendig. Auch der

System S1	System S2	
Beobachter	Betriebsobjekt	Instanzobjekt
0: Registriere den ObserverServer ...	0: 0 gespeicherte Instanzen geladen	
0: Erzeuge und registriere gespeicherte Observer	0: Starte Timer	
	0: Factory ... gebunden - Key = **FKEY**	
1: Beobachterschluessel erfolgreich erzeugt		
2: Observer (**BKEY**) erfolgreich gebunden		
3: Kontaktiere Betriebsobjekt (**FKEY**)		
	4: Observer (**OKEY**) hat Instanzerzeugung angefragt	
	5: neuer Prozess hat die Id: **PID**	
	6: Neue Instanz **IKEY** wurde erzeugt	
	7: Datenbankeintraege fuer die neu erzeugte Instanz erfolgreich angelegt	
8: Datenbankeintraege fuer den Observer (**OKEY**) erfolgreich angelegt		
9: Externe Prozessinstanz (**IKEY**) erfolgreich gestartet		
		10: Remote Activity Zustandswechsel - ProzessId = **PID**
11: Zustandswechsel einer Aktivitaet des externen Prozesses		
		12: Observer erfolgreich informiert
		13: Remote Activity Zustandswechsel - ProzessId = **PID**

Tabelle wird auf der nächsten Seite fortgesetzt

System S1	System S2	
Beobachter	Betriebsobjekt	Instanzobjekt
14: Zustandswechsel einer Aktivitaet des externen Prozesses		
		15: Observer erfolgreich informiert
		16: Remote Prozess Zustandswechsel - ProzessId = **PID**
17: Externer Unterprozess beendet		
18: Setze den neuen Kontext - Anzahl Variablen:4		
19: lokalen Prozess erfolgreich abgeschlossen		
20: Datenbankeintraege erfolgreich geloescht		
21: Observer (**OKEY**) erfolgreich bei RMI-Namensdient deregistriert		
		22: Observer erfolgreich informiert

Tabelle 5.2: Ausgaben der Objekte bei Testfall 2

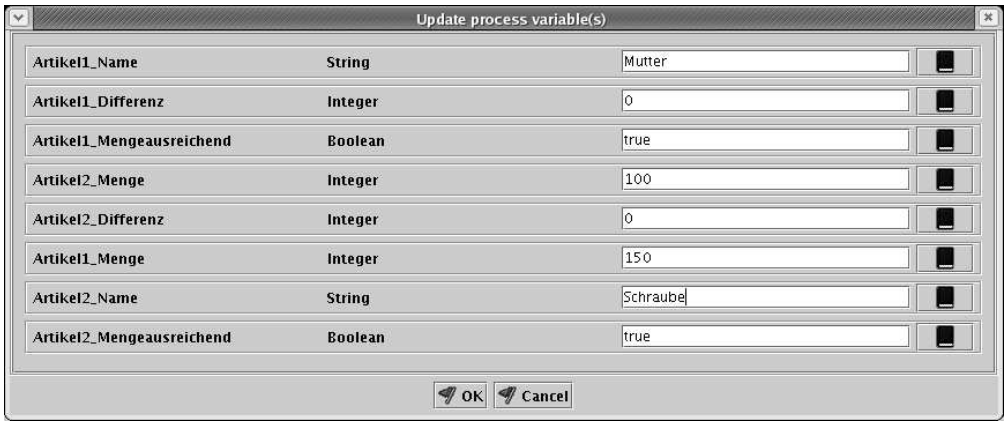

Abbildung 5.5: Testfall 2: Eingegebene Daten S1 vor dem externen Unterprozessaufruf

Betriebsobjektserver verhält sich im Falle nicht verketteter Prozesse anders. Er versuchte, bereits früher erzeugte Instanzobjekte zu laden und startete den Überwachungsthread bevor er das Betriebsobjekt beim RMI-Namensdienst registrierte.

Ausgabe 1 zeigt die Erzeugung des Beobachterobjekts für diesen Testfall. In Ausgabe 2 wurde der Beobachter beim RMI-Namensdienst registriert. Ausgabe 3 symbolisiert die Anfrage an das Betriebsobjekt.

Diese wurde, wie durch Ausgabe 4 gezeigt wird, durch das Betriebsobjekt angenommen. Die Ausgaben 5 bis 7 zeigen, dass ein neuer Prozess erzeugt und ein zugehöriges Instanzobjekt erstellt wurde, sowie die Speicherung der Daten in den Instanzsobjekt-Datenspeicher.

Die Ausgaben 8 und 9 zeigen, dass der Beobachter (Anwendungsagent) darüber informiert wurde und die Daten (insbesondere auch den zurückgelieferten Instanzidentifikator) in seinen Datenspeicher geschrieben hat.

Nachdem der externe Prozess erzeugt wurde, wurde seine erste Aktivität durch einen Nutzer auf System S2 angenommen (Ausgaben 10, 11 und 12) und der Beobachter darüber informiert. Die Eingabedaten entsprachen dabei den vorher eingestellten Werten und sind in Abb. 5.6 dargestellt.

Hier ist zu sehen, dass die Wahrheitwerte für die Verfügbarkeit der bestellten Artikelmengen nicht, wie im rufenden Prozess auf System S1, wahr sind, sondern falsch. Das liegt daran, dass es sich hier um Ausgabedaten handelt. Sie werden also nicht beim Start des Prozesses gesetzt, sondern ihnen wird ein Initialwert zugewiesen. Nachdem diese verändert wurden (siehe Abb. 5.7), wurde die Aktivität und somit der gesamte externe Prozess abgeschlossen.

Dies wird durch die Ausgaben 13 bis 22 dargestellt. Hier ist zu sehen, dass der Beobachter bei Beendigung des externen Prozesses die Rückgabedaten an den lokalen Unterprozess weitergegeben (Ausgabe 18), diesen abgeschlossen (Ausgabe 19), alle Daten aus dem

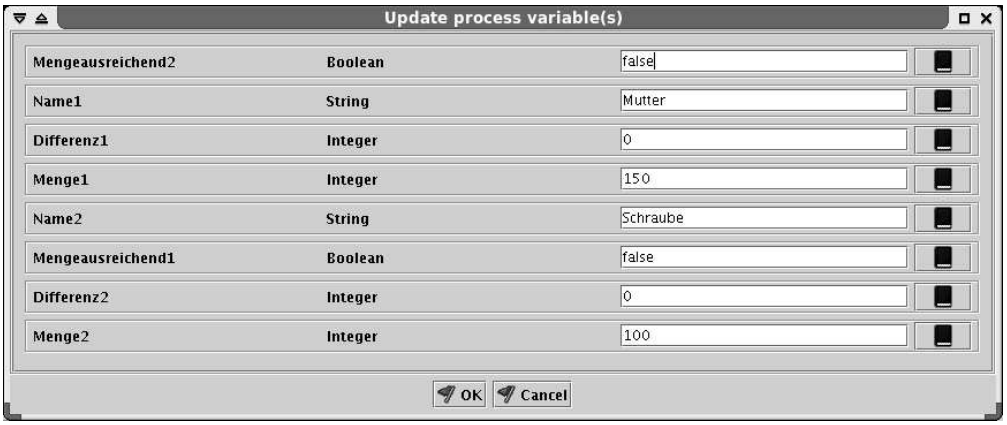

Abbildung 5.6: Testfall 2: Eingabedaten externer Unterprozess

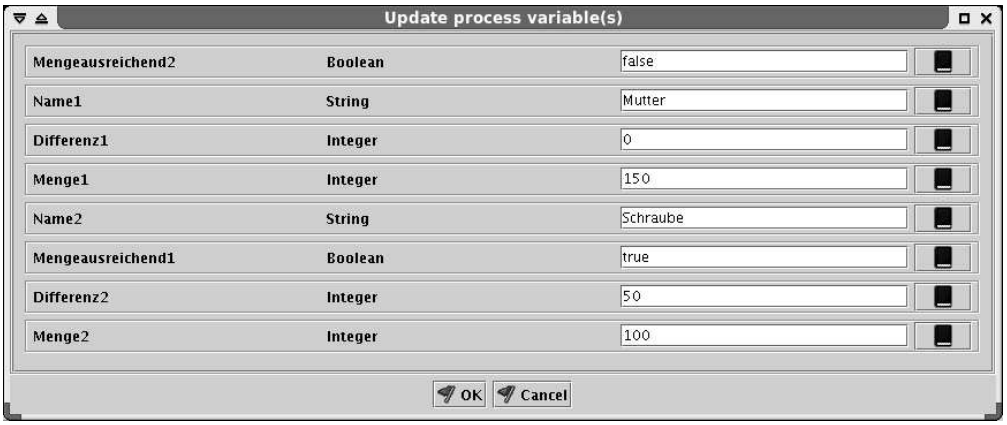

Abbildung 5.7: Testfall 2: Rückgabedaten externer Unterprozess

Beobachter-Datenspeicher gelöscht (Ausgabe 20) und sich beim RMI-Namensdienst abgemeldet (Ausgabe 21) hat.

Die Daten des Prozesses auf System S1 nach dem Abschluss des externen Unterprozesses sind in Abb. 5.8 zu sehen. Die geänderten Werte wurden korrekt zurückgegeben.

Resultat: Dieser Testfall zeigt, dass es möglich ist externe Unterprozesse zu starten und über deren Abarbeitung Informationen zu erhalten. Desweiteren wird die Über- und Rückgabe von Variablen demonstriert. Es ist also möglich, sowohl Eingabe- als auch Ein-/Ausgabe- und Ausgabedaten zu nutzen.

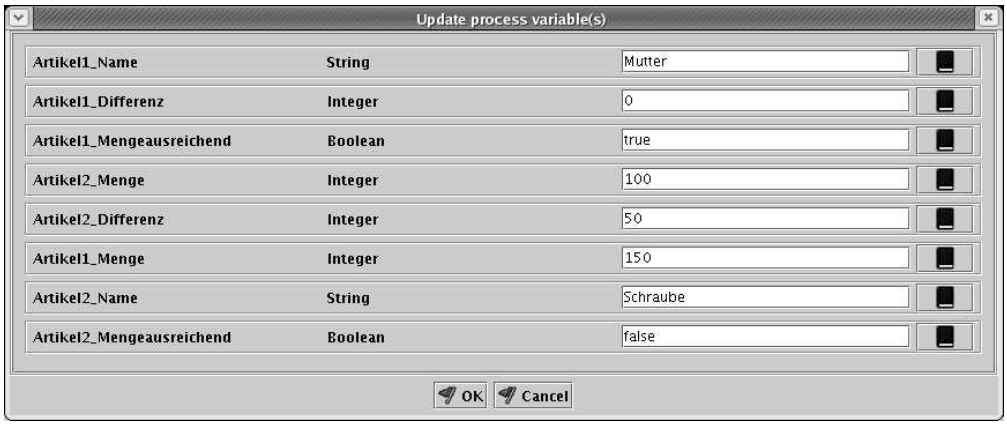

Abbildung 5.8: Testfall 2: Daten S1 nach dem externen Unterprozessaufruf

5.3 Testfall 3: Schachtelung von Unterprozessen

Für die Schachtelung von Unterprozessen wird eine Abwandlung des Workflows von Testfall 1 genutzt. Auch in diesem Testfall soll eine Warenbestellung zur Veranschaulichung dienen. Dabei wird auf System S1 die Bestellung entgegengenommen. Danach wird auf System S2 geprüft, ob ausreichend Kapazitäten der bestellten Ware verfügbar sind. Ist dies der Fall wird der Bestand um die bestellte Ware reduziert und die positive Rückmeldung führt zur Versendung der Ware durch System S1. Ist die Menge nicht ausreichend, wird durch einen dritten Prozess auf System S3 der benötigte Artikel bestellt. Nach Erhalt der Ware wird der Bestand auf System S2 aktualisiert und die Ware durch System S1 versendet. Das Aktivitätsnetzwerk für diesen Testfall ist in Abb. 5.9 dargestellt.

Die Ausgaben der Objekte sind in Tabelle 5.3 für die Kommunikation zwischen den Systemen S1 und S2 und in Tabelle 5.4 für die Kommunikation zwischen den Systemen S2 und S3 abgebildet.

Ablauf des Testfalls: Nachdem die Bestellung aufgenommen wurde, wurde der externe Unterprozess zum Prüfen der vorhandenen Artikelmengen gestartet. In diesem Testfall wurde das vorherige Starten der Serverobjekte nicht in die Ausgabe übernommen. In Ausgabe 1 ist zu sehen, dass der Beobachter auf System 1 erfolgreich gebunden wurde. Die Ausgaben 2 und 3 zeigen die Erzeugung des Instanzobjekts durch das zugehörige Betriebsobjekt. Ausgabe 4 bestätigt, dass der Anwendungsagent die Erzeugung erfolgreich beendet hat und der zugehörige Instanzschlüssel zurückgegeben wurde.

Die Ausgaben 5 bis 7 zeigen den Zustandswechsel der ersten Aktivität des externen Prozesses, als diese durch einen Nutzer angenommen wurde. Dazu soll an dieser Stelle die Rückmeldung an den Beobachter gezeigt werden. Hierfür ist die Ausgabe der in Abschnitt 4.2.4 beschriebenen Variablen in Abb. 5.10 abgebildet.

Obwohl auch die anderen sachbezogenen Daten in dem Fenster der Administrationsapplikation gezeigt werden, wurden diese für eine bessere Übersichtlichkeit entfernt. Wie zu

System S1	System S2	
Beobachter	Betriebsobjekt	Instanzobjekt
1: Observer (**OKEY**) erfolgreich gebunden		
	2: Observer (**OKEY1**) hat Instanzerzeugung angefragt	
	3: Neue Instanz **IKEY1** wurde erzeugt	
4: Externe Prozessinstanz (**IKEY1**) erfolgreich gestartet		
		5: Remote Activity Zustandswechsel ...
6: Zustandswechsel einer Aktivitaet des externen Prozesses		
		7: Observer erfolgreich informiert
		8: Remote Activity Zustandswechsel ...
9: Zustandswechsel einer Aktivitaet ...		
		10: Observer erfolgreich informiert
		11: Remote Activity Zustandswechsel ...
12: Zustandswechsel einer Aktivitaet ...		
		13: Observer erfolgreich informiert
Kommunikation zwischen S2 und S3		
42: Zustandswechsel einer Aktivitaet ...		41: Remote Activity Zustandswechsel ...
		43: Observer erfolgreich informiert
...	...	...
		50: Remote Prozess Zustandswechsel ...
51: Externer Unterprozess beendet		
52: lokalen Prozess erfolgreich abgeschlossen		
		53: Observer erfolgreich informiert

Tabelle 5.3: Ausgaben der Objekte bei Testfall 3 - Kommunikation zwischen S1 und S2

System S2	System S3	
Beobachter	Betriebsobjekt	Instanzobjekt
14: Observer (**OKEY2**) erfolgreich gebunden		
	15: Observer (**OKEY2**) hat Instanzerzeugung angefragt	
	16: neuer Prozess hat die Id: **PID2**	
	17: Neue Instanz **IKEY2** wurde erzeugt	
18: Externe Prozessinstanz (**IKEY2**) erfolgreich gestartet		
		19: Remote Activity Zustandswechsel ...
20: Zustandswechsel einer Aktivitaet des externen Prozesses		
		21: Observer erfolgreich informiert
		22: Remote Activity Zustandswechsel ...
23: Zustandswechsel einer Aktivitaet ...		
		24: Observer erfolgreich informiert
...	...	...
		34: Remote Activity Zustandswechsel ...
35: Zustandswechsel einer Aktivitaet ...		
		36: Observer erfolgreich informiert
		37: Remote Prozess Zustandswechsel ...
38: Externer Unterprozess beendet		
39: lokalen Prozess erfolgreich abgeschlossen		
		40: Observer erfolgreich informiert

Tabelle 5.4: Ausgaben der Objekte bei Testfall 3 - Kommunikation zwischen S2 und S3

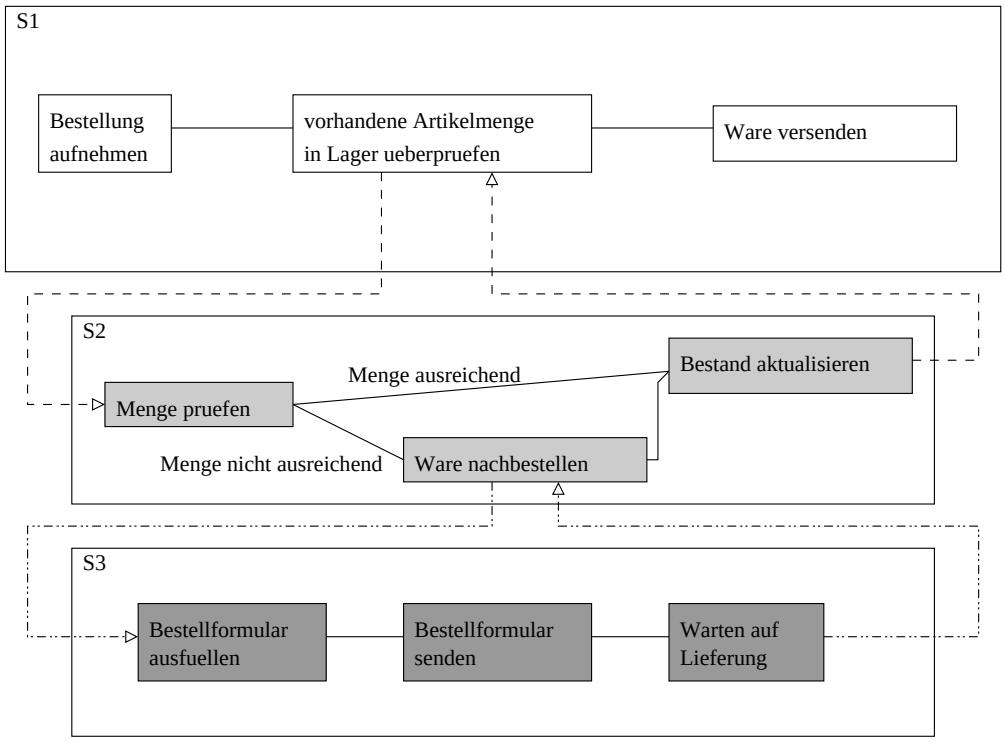

Abbildung 5.9: Testfall 3: Aktivitätsnetzwerk

sehen ist, befanden sich zu diesem Zeitpunkt sowohl der externe Prozess als auch seine erste Aktivität (`Menge pruefen`) im Zustand „Laufend“. Die Ausgabe des Zeitpunkts der Zustandsänderungen wird dabei durch die Administrationsapplikation nur als Datum ausgegeben. Es sind aber die genauen Zeiten bestehend aus Datum und Uhrzeit gespeichert. Durch Nutzung eines selbst implementierten Klienten könnten diese genauer ausgegeben werden.

Nach diesem Zustandswechsel erfolgte der Zustandswechsel zum Beenden dieser Aktivität (Ausgaben 8 bis 10). Daraufhin wurde automatisch der Unterprozess zur Abarbeitung des externen Prozesses gestartet (Zustandswechsel der Ausgaben 11 bis 13).

Die Ausführung des externen Unterprozesses auf System S3 begann mit dem Binden des Beobachters auf System S2 (Ausgabe 14). In den Ausgaben 15 bis 17 wird der angefragte Prozess erzeugt, initialisiert und gestartet, sowie das Instanzobjekt erzeugt. Ausgabe 18 zeigt die erfolgreiche Rückgabe des Instanzschlüssels. Danach folgen die Zustandsänderungen bei der Annahme und dem Abschluss der einzelnen Aktivitäten (Ausgaben 19 bis 36). Dabei wurde die Ausgabe gekürzt. Ausgabe 37 zeigt die Beendigung des externen Prozesses über die der Beobachter in Ausgabe 38 informiert wurde. Dieser schloss danach den Unterprozess ab.

Durch den Abschluss des externen Unterprozesses auf System S3 und dem damit verbundenen Abschluss des Unterprozesses auf System S2 wurde die Information der Zu-

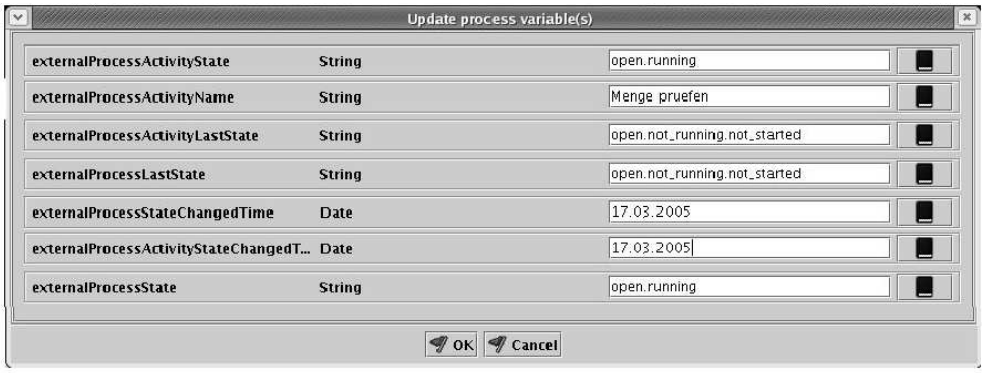

Abbildung 5.10: Testfall 3: Informationen des Beobachters über den externen Prozess

standsänderung an den Beobachter auf System S1 gesendet (Ausgaben 41 bis 43). Dieser wurde danach über die weiteren Zustandsänderungen von Aktivitäten informiert. Bei der Beendigung des durch S1 erzeugten externen Prozesses auf System S2 wurde auch der Unterprozess auf System S1 beendet (Ausgaben 50 bis 53).

Resultat: Mit diesem Testfall wurde gezeigt, dass es möglich ist verschachtelte externe Unterprozesse mit der entwickelten Schnittstelle abzuarbeiten. Dabei ist eine beliebig tiefe Schachtelung möglich, da aus der Sicht des rufenden Systems nur ein externer Prozess abgearbeitet wird. Es bleibt ihm also verborgen, dass dieser wiederum einen weiteren externen Prozess während seiner Abarbeitung nutzt.

5.4 Testfall 4: Ausfall des Betriebsobjekts und Beobachters

Für Testfall 4 wird der Workflow aus Testfall 2 verwendet. Dabei wird zunächst versucht den externen Prozess zu erzeugen, obwohl der Betriebsobjektserver nicht gestartet ist. Danach wird der externe Prozess gestartet und der Beobachterserver gestoppt. Die Ausgaben dieses Testfalls sind in Tabelle 5.5 abgebildet und sollen jetzt näher erläutert werden.

Ablauf des Testfalls: Die in den vorhergehenden Auflistungen dargestellten Ausgaben 0, die zum Starten des Beobachterservers gehören, wurden in diesem Testfall weggelassen. Der Betriebsobjektserver wurde anfänglich nicht gestartet.

In Ausgabe 1 wurde der Beobachter für den externen Unterprozess angelegt und an die entsprechend erzeugte RMI-URI gebunden. Danach wurde versucht das Betriebsobjekt zu kontaktieren, um den externen Unterprozess zu erzeugen. Da jedoch der Betriebsobjektserver nicht gestartet war und somit keine Betriebsobjekte geladen wurden, schlug dies fehl (Ausgabe 2). Daraufhin wurde der Beobachter wieder beim RMI-Namensdienst

System S1	System S2	
Beobachter	Betriebsobjekt	Instanzobjekt
1: Observer (**OKEY1**) erfolgreich gebunden		
2: Fehler bei Anfrage an das Betriebsobjekt- Connection refused to host: 192.168.0.2;		
	3: 0 gespeicherte Instanzen geladen	
	4: Starte Timer	
	5: Factory ... gebunden - Key = **FKEY**	
6: Observer (**OKEY2**) erfolgreich gebunden		
	7: Observer (**OKEY2**) hat Instanzerzeugung angefragt	
	...	
8: Externe Prozessinstanz (**IKEY**) erfolgreich gestartet		
9: Observer (**OKEY2**) erfolgreich deregistriert		
		10: Remote Activity Zustandswechsel ...
		11: Observer NICHT informiert, Daten in DB gespeichert
		12: Informiere Beobachter über Aktivitätszustandsänderung
		13: Konnte den Beobachter (**OKEY2**) nicht informieren
14: Observer (**OKEY2**) erfolgreich gebunden		

Tabelle wird auf der nächsten Seite fortgesetzt

System S1	System S2	
Beobachter	Betriebsobjekt	Instanzobjekt
		15: Informiere Beobachter über Aktivitätszustandsänderung
16: Zustandswechsel einer Aktivitaet des externen Prozesses		
		17: Beobachter (**OKEY2**) erfolgreich ueber Zustandsaenderung von Aktivitaet ...informiert
18: Observer (**OKEY2**) erfolgreich deregistriert		
		19: Remote Activity Zustandswechsel ...
		20: Observer NICHT informiert, Daten in DB gespeichert
		21: Remote Prozess Zustandswechsel ...
		22: Observer NICHT informiert, Daten in DB gespeichert
23: Observer (**OKEY2**) erfolgreich gebunden		
		24: Informiere Beobachter über Aktivitätszustandsänderung
25: Zustandswechsel einer Aktivitaet des externen Prozesses		
		26: Beobachter (**OKEY2**) erfolgreich ueber Zustandsaenderung von Aktivitaet ...informiert
		27: Informiere den Beobachter (**OKEY2**) ueber Beendigung des Prozesses ...
28: Externer Unterprozess beendet		
29: lokalen Prozess erfolgreich abgeschlossen		
		30: Beobachter (**OKEY2**) erfolgreich ueber Beendigung von Prozess ...informiert

Tabelle 5.5: Ausgaben der Objekte bei Testfall 4

abgemeldet und das entsprechende Objekt zerstört. Die Ausführung des Prozesses innerhalb von Enhydra Shark ist dabei an das Ende der letzten Aktivität vor dem Unterprozess zurückgesetzt worden.

Anschließend wurde der Betriebsobjektserver gestartet und die Betriebsobjekte wurden geladen (Ausgaben 4 und 5). In den Ausgaben 6, 7 und 8 ist erneut die Anfrage zur Erzeugung eines externen Unterprozesses übermittelt worden, was diesmal gelang. Zu beachten ist hierbei, dass der Identifikator des Beobachters nicht derselbe ist, wie beim vorigen Versuch. Das ist damit zu begründen, dass im Identifikator der Prozessidentifikator des Unterprozesses mit eingebunden wird. Da es sich um einen Neustart, und somit um die Erzeugung einer neuen Instanz des Unterprozesses handelt, ist dieser Prozessidentifikator nicht derselbe.

Nach dem Start des externen Unterprozesses wurde der Beobachterserver gestoppt. Damit ist auch der eben erzeugte Beobachter beim RMI-Namensdienst abgemeldet worden (Ausgabe 9). Dadurch konnte der externe Prozess die Mitteilung über einen Aktivitätszustandswechsel nicht an den zugehörigen Beobachter übertragen und speicherte die entsprechenden Daten in den Instanzobjekt-Datenspeicher (Ausgaben 10 und 11). Der Überwachungsthread versuchte nach dem definierten Intervall den Beobachter über diesen Zustandswechsel zu informieren, was aber nicht gelang, da der Beobachter zu diesem Zeitpunkt noch nicht wieder gestartet wurde (Schritte 12 und 13).

In Ausgabe 14 ist zu sehen, dass der Beobachter wieder gestartet wurde. Bei einem neuen Lauf des Überwachungsthreads konnte dieser ihn deshalb über die Zustandsänderung informieren (Ausgaben 15, 16 und 17).

Um auch die Möglichkeiten der Sicherung bei der Beendigung eines exteren Prozesses zu verdeutlichen, wurde der Beobachterserver noch einmal beendet und der externe Prozess abgeschlossen. Dies ist in den Ausgaben 18 bis 22 dargestellt. Nach dem anschließenden Neustart des Beobachterservers wurden auch diese Zustandsänderungen an das entsprechende Beobachterobjekt übermittelt und der lokale Unterprozess erfolgreich beendet (Ausgaben 23 bis 30).

Resultat: Durch diesen Testfall wurde gezeigt, dass ein zwischenzeitlicher Ausfall des Betriebs- und/oder Beobachterobjekts zu keinem Datenverlust führt. Es wurden alle Änderungen bei Neustart der beiden Objekte übertragen.

6 Erweiterungsmöglichkeiten

Dieses Kapitel soll Möglichkeiten der Erweiterung des Konzepts und der Implementierung diskutieren. Als erstes soll dazu ein nicht unbedingt als Erweiterung anzusehender Nutzungsfall diskutiert werden: die parallele Abarbeitung eines externen Prozesses. In einer zweiten Diskussion sollen die Möglichkeiten der Nutzung eines anderen WfMS sowohl als rufendes als auch als gerufenes System dargestellt werden. Im dritten Teil dieses Kapitels sollen Möglichkeiten vorgestellt werden, wie das in Abschnitt 2.2 beschriebene Szenario 3 implementiert werden könnte.

6.1 Parallele Abarbeitung des externen Prozesses

In [Mar99], Abschnitt 4.2.4.4 wird eine Möglichkeit der Nutzung eines externen Unterprozesses vorgestellt, bei der der externe Prozess parallel zum lokalen Prozess abgearbeitet wird. Mit der prototypischen Implementierung ist dies auf den ersten Blick nicht möglich, da entweder ein verketteter externer Prozess oder ein Unterprozess als „eine Aktivität“ abgearbeitet werden kann. Durch die Möglichkeit der Nutzung von parallelen Verzweigungen und anschließender Synchronisation ist dies aber dennoch, wenn auch eingeschränkt, möglich. Ein Beispiel für einen parallelen Unterprozess ist in Abb. 6.1 und die Abwandlung für die Nutzung des Prototypen in Abb. 6.2 dargestellt.

Die beiden Varianten sind dabei von der Abarbeitungsreihenfolge identisch. Durch die Synchronisation vor A4 bei der Abwandlung für den Prototypen wird die Abarbeitung der beiden Stränge gestoppt bis beide an dieser Stelle angelangen.

Eine Schwierigkeit dieser Lösung besteht jedoch bei der Rückgabe der prozessbezogenen Daten. In dem im Standard definierten Modell wird das Rückschreiben der Daten

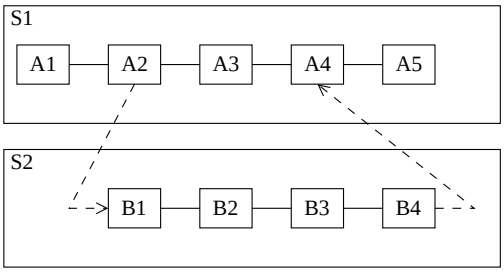

Abbildung 6.1: Parallele Abarbeitung eines externen Prozesses - Standard

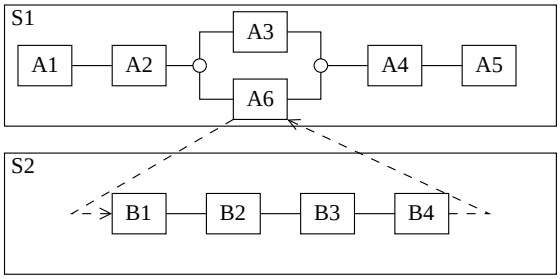

Abbildung 6.2: Parallele Abarbeitung eines externen Prozesses - Prototyp

des externen Prozesses erst vollzogen, wenn der lokale Prozess die Aktivität A4, die den Synchronisationspunkt darstellt, erreicht. Ist der externe Prozess vorher beendet, werden die Daten vorerst in eine Warteschlange gestellt. Diese Vorgehensweise ist mit dem Prototypen nicht realisierbar, da sofort nach Beendigung des externen Prozesses die Daten zurückgeschrieben werden.

Das Problem hierbei ist, dass die prozessbezogenen Daten für alle Aktivitäten des Prozesses gültig sind. Übergibt man dem externen Prozess also Daten, die von diesem verändert werden dürfen (OUT- bzw. INOUT-Parameter) und arbeitet auch der lokale Prozess (Aktivität A3 des Beispiels) auf diesen Daten, so kann ein Zurückschreiben zu inkonsistenten Datenbeständen bei der Abarbeitungen führen. Allerdings sollte man es generell vermeiden in parallelen Abarbeitungssträngen auf den gleichen Daten zu arbeiten.

Eine Lösung wäre, dass im Falle einer synchronen Abarbeitung der externe Prozess nur Eingabeparameter besitzen darf. Das schränkt allerdings die Nutzungsmöglichkeiten beträchtlich ein. Eine andere Lösung ist das Anlegen von Kopien der Daten, die durch den externen Prozess geändert werden dürfen. Dem externen Prozess werden dann diese Kopien übergeben. Auch das ist nicht optimal, da die Anzahl an prozessbezogenen Daten dadurch in manchen Fällen stark erhöht werden würde, in jedem Falle aber zunimmt. Außerdem müsste ein geeigneter Mechanismus zum Erstellen der Kopien und Zurückschreiben der geänderten Werte durch eine automatisierte Aktivität implementiert werden.

Zusammenfassung Wie gezeigt wurde, ist es möglich den externen Unterprozess parallel zum lokalen Prozess auszuführen. Allerdings müssten dabei in Hinblick auf die Datenkonsistenz zusätzliche Mechanismen entworfen und implementiert werden.

6.2 Nutzung eines anderen WfMS als entferntes System

Dieser Abschnitt beschäftigt sich mit Möglichkeiten, ein zweites WfMS als System des Elternprozesses oder des Kindprozesses einzusetzen. Das heißt, wie kann ein bestimmtes Level der Interoperabilität erreicht werden. Diese Level wurden in Abschnitt 2.2 vorgestellt. Die folgenden Ausführungen beziehen sich dabei auf Level 2 (Koexistenz). Unter Level 2 soll hier allerdings nicht die Ausführung auf Basis derselben Ausführungsumgebung, sondern die Kommunikation über nicht standardisierte Schnittstellen verstanden werden. Zwar wird in der Lösung der ASAP-Standard zur Kommunikation genutzt, aber die Datenrepräsentation und das zugrundeliegende Protokoll wurden im Prototypen geändert.

Um ein anderes WfMS mit der entwickelten Lösung zu nutzen, ist es für dieses System notwendig, die drei an der Kommunikation beteiligten Objekte zu implementieren. Außerdem muss der Zugriff auf die jeweils andere Seite über die definierten Schnittstellen stattfinden. Das bedeutet: Soll das andere System den externen Prozess rufen, so muss es das Beobachterobjekt als RMI-Server anhand der Schnittstellenbeschreibung `de.raphaelkunis.sharkinterop.rmi.Observer` implementieren. Soll es jedoch als das System

eingesetzt werden, das den externen Prozess abarbeitet, so muss es das Betriebsobjekt anhand der Schnittstellenbeschreibung `de.raphaelkunis.sharkinterop.rmi.Factory` implementieren. Außerdem muss es in diesem Fall ein Instanzobjekt, das den Beobachter über Änderungen informiert, bereitstellen. Die Kommunikation erfolgt dann analog dem Fall, dass zwei Shark-Laufzeitumgebungen an der Kommunikation beteiligt sind.

Falls das andere WfMS nicht in Java implementiert ist, muss ein Gateway eingesetzt werden, der die entfernten Methoden aufeinander abbildet.

Ein wichtiger, noch zu erwähnender Aspekt ist die Datenkodierung. Die implementierte Lösung nutzt dazu den Java-Datentyp `Map` mit einer Name-Wert-Zuordnung für alle zu übertragenden Daten. Die möglichen Datentypen sind dabei die in XPDL beschreibbaren und eigene Java-Klassen, die serialisierbar sein müssen. Die Daten des anderen Systems müssten also vor dem Übertragen bzw. nach dem Empfangen angepasst werden. Im Falle von XPDL als Prozessbeschreibungssprache des anderen Systems sollte das problemlos möglich sein, da die gleichen Datentypen genutzt werden können. Bei einer anderen Beschreibungssprache könnte es allerdings vorkommen, dass Typen nicht oder nur sehr schwierig aufeinander abgebildet werden können.

Zusammenfassung In diesem Abschnitt wurden Möglichkeiten zur Nutzung eines anderen WfMS kurz dargelegt und mögliche Schwierigkeiten (vor allem in Hinblick auf die Datenabbildung) erläutert.

6.3 Möglichkeiten der Implementierung des Szenario 3 der Interoperabilität

In diesem Abschnitt sollen Ansätze aufgezeigt werden, wie die Implementierung genutzt werden kann bzw. geändert werden müsste, um das in Abschnitt 2.2 beschriebene Szenario 3 (Modell der gemeinsam genutzten Domäne) umzusetzen.

Szenario 3 beschreibt eine Abarbeitung des Workflows, bei der die einzelnen Aktivitäten eines Prozesses P durch verschiedene WfMS ausgeführt werden. Da der Standard keine genaueren Angaben über die Realisierung beschreibt, sollen nachfolgend drei mögliche Realisierungsvarianten mit ihren Vor- und Nachteilen diskutiert werden. Als Beispiel wird dabei der in Abb 6.3 dargestellte Prozess P genutzt.

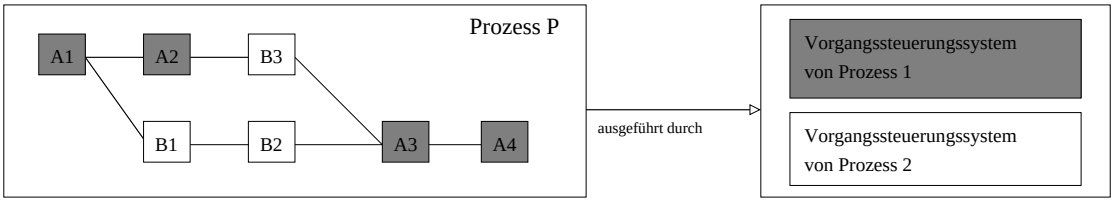

Abbildung 6.3: Erweiterungsmöglichkeiten - Beispielprozess Szenario 3

In diesem Beispiel soll der Prozess P durch zwei Workflow Management Systeme S1 und S2 abgearbeitet werden. Die vorliegende prototypische Implementierung besitzt zwei Schwachstellen für dieses Szenario. Zum ersten ist diese nicht für die externe Abarbeitung von einzelnen Aktivitäten ausgelegt. Die externe Abarbeitung wird im Prototypen *nur* auf Prozessebene realisiert. Zum zweiten ist keine Übermittlung von Zwischenergebnissen möglich. Ein externer Prozess übermittelt die zurückzugebenden prozessbezogenen Daten nur bei seiner Beendigung.

Lösungsvariante 1 In der ersten Lösungsvariante wird der Prozess P auf einem System S3 abgearbeitet. Durch das WfMS auf System S3 werden dabei keine Aktivitäten abgearbeitet, sondern anhand einer vorgegebenen Prozessbeschreibung die Aufgaben verteilt. Es handelt sich also um einen Prozess, der nur aus Repräsentationen der externen Aufgaben besteht. Er startet diese und wartet auf die Beendigung ihrer Ausführung. In dieser Variante müssen alle Aktivitäten bzw. Aktivitätsfolgen (mehrere hintereinander auszuführende Aktivitäten *ohne* parallele und synchrone Verzweigungen zu Aktivitäten eines anderen Systems) als eigenständige Prozessdefinitionen auf dem System, das sie abarbeitet, vorliegen. Das stellt einen sehr großen Zusatzaufwand bei der Prozessdefinition dar und führt zu vielen kleinen Prozessen. Außerdem stellt die Definition des Steuerprozesses zusätzlichen Aufwand dar. In Abbildung 6.4 ist diese Lösungsvariante für den Beispielprozess skizziert.

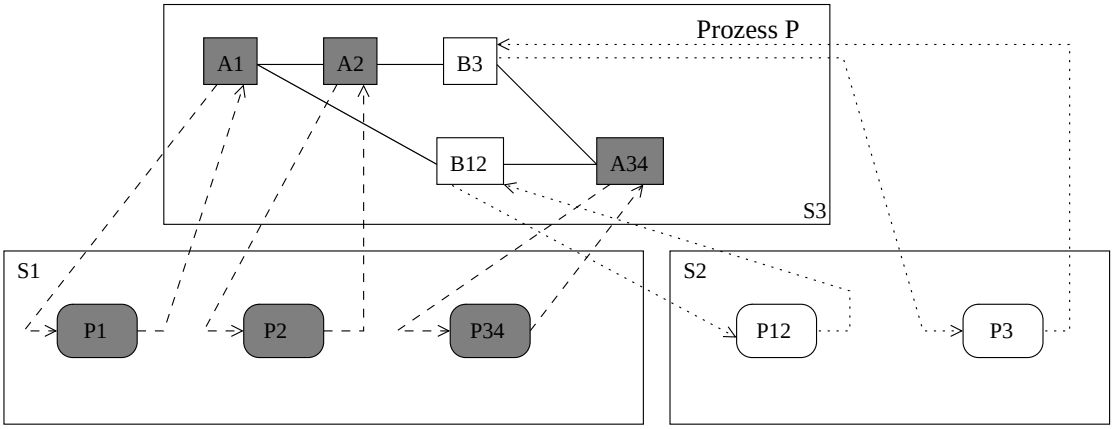

Abbildung 6.4: Erweiterungsmöglichkeiten - Szenario 3 Lösungsvariante 1

Hier mussten die Aktivitäten A1 und A2 als eigenständige Prozesse auf System S1 definiert werden. A3 und A4 konnten zu einem Prozess zusammengefasst werden, da keine Verzweigungen vorhanden sind und sie aufeinanderfolgen. Bei S2 konnten B1 und B2 zusammengefasst werden und B3 musste als eigenständiger Prozess definiert werden.

Der Vorteil dieser Lösung ist, dass keine Änderungen am Prototypen vorgenommen werden müssten um diese Variante zu realisieren.

Lösungsvariante 2 In einer zweiten Idee könnte, anstelle eines separaten Steuerprozesses, der Prozess, der die erste Aktivität ausführt die Aufgaben der Steuerung übernehmen. Dadurch entfällt die Aufteilung der Aktivitäten dieses Systems in eigenständige Prozesse. Die einzige Änderung im Vergleich zu vorherigen Lösungsvariante wäre, dass der Steuerprozess (im Beispiel) auf System S1 ausgeführt würde und alle Aktivitäten, die extern auf System S1 ausgeführt werden müssten, könnten durch ihre Realisierungen ersetzt werden. Das würde zumindest den Aufwand bei der Prozessbeschreibung dieses Systems auf die Beschreibung eines Prozesses reduzieren. Der Aufwand auf den anderen Systemen bleibt aber derselbe. Die angepasste Abbildung für diese Realisierung ist in Abb. 6.5 dargestellt.

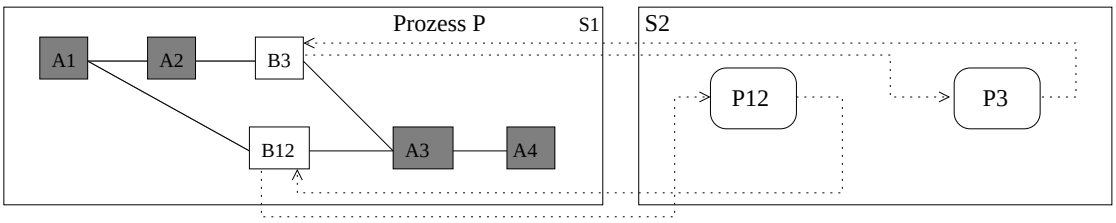

Abbildung 6.5: Erweiterungsmöglichkeiten - Szenario 3 Lösungsvariante 2

Auch in diesem Lösungsansatz sind noch keine Änderungen am vorgestellten Konzept bzw. dem Prototypen notwendig.

Lösungsvariante 3 Die dritte Lösungsvariante versucht die Notwendigkeit der Beschreibung aller Aktivitäten bzw. Aktivitätsblöcke in eigenen Prozessen zu vermeiden. So soll eine Möglichkeit angedeutet werden, bei der jedes beteiligte WfMS nur eine Prozessdefinition für dieses Szenario benötigt.

Für diesen Fall sind erstmals Änderungen an den Objekten notwendig. Das heißt, das aufgestellte Konzept müsste erweitert werden. Die Änderung betreffen dabei das Beobachterobjekt, den Anwendungsagenten und das Instanzobjekt. Um für jedes System nur eine Prozessbeschreibung definieren zu müssen, wäre es notwendig von der Prozessebene auf die Aktivitätsebene bei der Ausführung externer Prozesse zu wechseln.

Am Betriebsobjekt müsste eine Funktion zum Empfang von Daten nach Beendigung einer Aktivität des externen Prozesses hinzugefügt werden. Außerdem müsste der Anwendungsagent, statt nur Prozesse durch das Betriebsobjekt zu starten, mit dem Instanzobjekt kommunizieren und dieses zum Starten von Aktivitäten nutzen können. Dazu muss das Instanzobjekt zu einem Serverobjekt geändert werden, das Methoden zum Starten der nächsten Aktivität bereitstellt.

Außerdem müsste jedes System eine eigene Prozessbeschreibung besitzen, in der nur die von dem entsprechenden System abzuarbeitenden Aktivitäten beinhaltet sind. Zusätzlich müssten Warteaktivitäten eingefügt werden, die die Abarbeitung anhalten, bis der Beobachter die Ausführung der nächsten Aktivität anfordert. Problematisch sind in diesem Zusammenhang Verzweigungen. Die Abarbeitung auf den einzelnen Systemen

müsste diese berücksichtigen, da bei synchronen Verzweigungen eventuell mehrere Aktivitäten abgearbeitet werden müssen und bei bedingten Verzweigungen eine von mehreren möglichen Aktivitäten abgearbeitet werden könnte.

Für den Beispielprozess soll nun eine mögliche Beschreibung für die beiden Systeme skizziert werden. In dieser werden eine „Stop“-Aktivität, eine „Dummy“-Aktivität und eine „Inform“-Aktivität eingeführt. Die „Stop“-Aktivität hält die Ausführung des Prozesses an, bis der Beobachter die Abarbeitung der nächsten Aktivität anfordert und die „Inform“-Aktivität gibt Daten an den Beobachter zurück. Die „Dummy“-Aktivität dient z. B. dazu, bei Alternativen in der Prozessbeschreibung des Gesamtprozesses, bei denen eventuell eine Aktivität des entsprechenden Systems ausgeführt wird, die Abarbeitung ohne Ausführung einer Aktivität anzuhalten. Außerdem müsste sie eingesetzt werden, wenn der Prozess auf einem System mit einer Alternative beginnt. Der Beispielprozess ist für diese Variante in Abb. 6.6 dargestellt. Dabei wurde die Aktivität A4 welches auf System S1 ausgeführt werden müsste durch eine Aktivität B4 von System S2 ersetzt, um die Realisierungsvariante etwas ausführlicher zu demonstrieren.

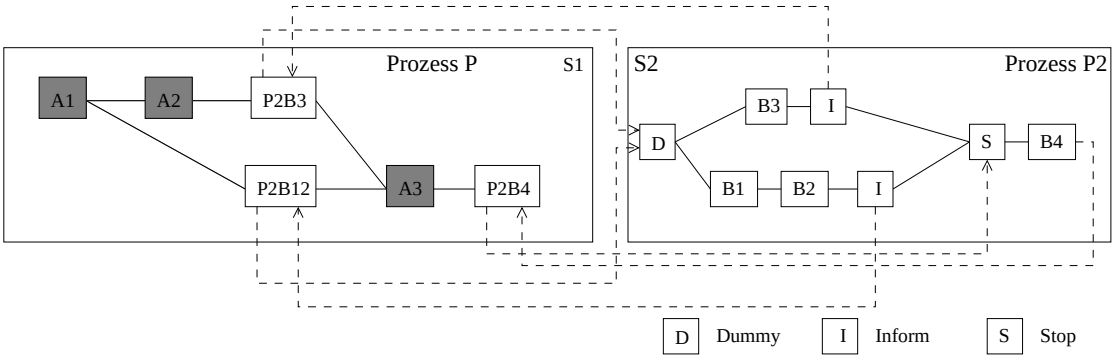

Abbildung 6.6: Erweiterungsmöglichkeiten - Szenario 3 Lösungsvariante 3

Die Abarbeitung wird dabei wie in Lösungsvariante 2 durch das System S1 gesteuert. Wie in der Abbildung zu sehen ist, wird der externe Prozess entweder durch P2B3 oder P2B12 gestartet. Der einzige Unterschied hierbei ist, dass mindestens das prozessbezogene Datum für die Bedingung der Verzweigung am Anfang von P2 unterschiedlich ist. Die Alternativen müssen dabei identisch zum Prozess P auf System S1 definiert werden. Nach der Beendigung von Aktivität B3 bzw. dem Aktivitätsblock mit B1 und B2 werden durch die „Inform“-Aktivität Daten zurückgegeben. Dies müsste über die neue Funktion des Beobachters geschehen. Danach wartet der Prozess in der „Stop“-Aktivität bis der Prozess P die weitere Abarbeitung anfordert. Geschieht dies, wird B4 abgearbeitet. Die „Inform“-Aktivität danach entfällt, da der Prozess komplett abgearbeitet ist und somit die Rücksendung der Daten durch die im Prototypen enthaltene Methode „Complete“ geschehen würde.

Obwohl dieser Lösungsvorschlag für zwei beteiligte Systeme noch relativ einfach zu überschauen ist, steigt der Aufwand für die Prozessbeschreibungen auf den einzelnen Syste-

men bei vielen Systemen stark an (vor allem in Hinblick auf Verzweigungen). Außerdem müsste mit den gegebenen Hilfsaktivitäten noch nachgewiesen werden, dass die Abarbeitung identisch zu einer Abarbeitung auf einem System abläuft.

Zusammenfassung Wie in diesem Abschnitt gezeigt werden sollte, ist es möglich das Szenario 3 ohne Änderungen am Prototypen umzusetzen. Der Nachteil ist, dass die Menge der Prozessbeschreibungen und der Aufwand für die Prozessbeschreibungen mit den Lösungsvorschlägen 1 und 2 stark vergrößert wird. Selbst durch Änderungen am Prototypen kann zwar die Menge an Prozessbeschreibungen vermindert werden, der Aufwand um diese zu erstellen bleibt aber bestehen.

Was in den Ausführungen dabei nicht weiter betrachtet wurde ist die Datenkonsistenz. Wie schon in den Betrachtungen zur synchronen Abarbeitung externer Unterprozesse in diesem Kapitel kann die Verteilung und Bearbeitung der Daten schwierig sein. Die Abhängigkeiten der Daten müssen deshalb schon bei der Prozessbeschreibung berücksichtigt werden. Entweder dürfen die, in den einzelnen parallelen Aktivitäten bearbeiteten Daten sich nicht überschneiden oder die Aktivitäten müssen auf Kopien arbeiten.

7 Zusammenfassung

Ziel dieser Diplomarbeit war es, eine Schnittstelle zur hierarchischen Abarbeitung räumlich verteilter Workflows zu entwickeln. Dabei sollte ein freies Workflow Management System eingesetzt werden.

Es wurde anhand des ASAP-Standards ein Konzept für eine solche Schnittstelle erstellt. Das Konzept greift die Ideen der einzelnen, in diesem Standard beschriebenen Objekte (Beobachter-, Instanz- und Betriebsobjekt) auf. Die Methoden und Attribute wurden für den Einsatz als Protokoll für die verteilte Workflowabarbeitung angepasst. Die durch das Konzept möglichen Einsatzfälle sind das Starten eines verketteten Prozesses und das Starten eines externen Unterprozesses.

Die Umsetzung des Konzepts wurde unter Nutzung von RMI als zugrundeliegende Middleware-Technologie vorgenommen. Als Workflow Management System wurde Enhydra Shark ausgewählt, da es die von der WfMC gegebenen Standards implementiert. Besonderes Augenmerk galt der Ausfallsicherheit. Dies war notwendig, da bei Systemen, die über Rechnernetze kommunizieren, Ausfälle nicht vernachlässigt werden sollten.

In ausführlichen Testfällen wurde anhand ausgewählter Beispiele die Funktionstüchtigkeit der implementierten Schnittstelle nachgewiesen.

Zusätzlich wurden mögliche Erweiterungen des Prototypen vorgestellt. Dazu gehören die Nutzung der Schnittstelle für eine parallele Abarbeitung des externen Prozesses und die Implementierung des vorgestellten Szenario 3 der Interoperabilität von Workflow Management Systemen. Diese könnten in weiterführenden Arbeiten umgesetzt werden.

A XPDL-Beschreibung des Observer-Pakets

```
<?xml version="1.0" encoding="UTF-8"?>
<Package Id="ObserverPacket" Name="ObserverPacket" xmlns="http://www.wfmc.org/2002/XPDL1.0"
         xmlns:observer="http://www.raphaelkunis.de/observer"
         xmlns:xpdl="http://www.wfmc.org/2002/XPDL1.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
         http://wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd">
  <Script Type="text/java"/>

  <Participants>
    <Participant Id="Raphael_Kunis" Name="Raphael_Kunis">
      <ParticipantType Type="HUMAN"/>
    </Participant>
    <Participant Id="Observer_System" Name="Observer_System">
      <ParticipantType Type="SYSTEM"/>
      <Description>Ausführer des ObserverToolagent</Description>
    </Participant>
    <Participant Id="Observer_User" Name="Observer_User">
      <ParticipantType Type="HUMAN"/>
    </Participant>
  </Participants>

  <WorkflowProcesses>
    <WorkflowProcess AccessLevel="PRIVATE" Id="Observer_Default" Name="Observer_Default">
      <ProcessHeader/>

      <DataFields>
        <DataField Id="externalProcessState" IsArray="FALSE" Name="externalProcessState">
           <DataType> <BasicType Type="STRING"/> </DataType>
           <InitialValue>open.running</InitialValue>
        </DataField>
        <DataField Id="externalProcessLastState" IsArray="FALSE" Name="externalProcessLastState">
          <DataType> <BasicType Type="STRING"/> </DataType>
          <InitialValue>open.not_running.not_started</InitialValue>
        </DataField>
        <DataField Id="externalProcessStateChangedTime" IsArray="FALSE"
                   Name="externalProcessStateChangedTime">
          <DataType> <BasicType Type="DATETIME"/> </DataType>
        </DataField>
        <DataField Id="externalProcessActivityName" IsArray="FALSE"
                   Name="externalProcessActivityName">
          <DataType> <BasicType Type="STRING"/> </DataType>
        </DataField>
        <DataField Id="externalProcessActivityState" IsArray="FALSE"
                   Name="externalProcessActivityState">
          <DataType> <BasicType Type="STRING"/> </DataType>
        </DataField>
        <DataField Id="externalProcessActivityLastState" IsArray="FALSE"
                   Name="externalProcessActivityLastState">
          <DataType> <BasicType Type="STRING"/> </DataType>
        </DataField>
        <DataField Id="externalProcessActivityStateChangedTime" IsArray="FALSE"
                   Name="externalProcessActivityStateChangedTime">
```

```
    <DataType> <BasicType Type="DATETIME"/> </DataType>
  </DataField>
  <DataField Id="Observer_Chaining" IsArray="FALSE" Name="Observer_Chaining">
    <DataType> <BasicType Type="BOOLEAN"/> </DataType>
    <InitialValue>false</InitialValue>
  </DataField>
</DataFields>

<Applications>
  <Application Id="ObserverToolAgent" Name="ObserverToolAgent">
    <FormalParameters>
      <FormalParameter Id="Observer_Chaining" Index="Observer_Chaining" Mode="OUT">
        <DataType> <BasicType Type="BOOLEAN"/> </DataType>
      </FormalParameter>
    </FormalParameters>

    <ExtendedAttributes>
      <ExtendedAttribute Name="ToolAgentClass"
                         Value="de.raphaelkunis.sharkinterop.sharkutils.ObserverToolAgent"/>
      <ExtendedAttribute Name="AppName" Value="ObserverToolAgent"/>
      <ExtendedAttribute Name="FactoryName" Value="FactoryDefault"/>
      <ExtendedAttribute Name="Chaining" Value="false"/>
    </ExtendedAttributes>
  </Application>
</Applications>

<Activities>
  <Activity Id="Observer_Closer" Name="Observer_Closer">
    <Implementation> <No/> </Implementation>
    <Performer>Observer_User</Performer>
    <StartMode> <Automatic/> </StartMode>
    <FinishMode> <Manual/> </FinishMode>
    <ExtendedAttributes>
      <ExtendedAttribute Name="ParticipantID" Value="Observer_User"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessState"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessLastState"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessStateChangedTime"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessActivityName"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessActivityState"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessActivityLastState"/>
      <ExtendedAttribute Name="VariableToProcess_VIEW"
                         Value="externalProcessActivityStateChangedTime"/>
    </ExtendedAttributes>
  </Activity>
  <Activity Id="Observer_Starter" Name="Observer_Starter">
    <Implementation>
      <Tool Id="ObserverToolAgent" Type="APPLICATION">
        <ActualParameters>
          <ActualParameter>Observer_Chaining</ActualParameter>
        </ActualParameters>
      </Tool>
    </Implementation>
    <Performer>Observer_System</Performer>
    <StartMode> <Automatic/> </StartMode>
    <FinishMode> <Automatic/> </FinishMode>
    <TransitionRestrictions>
```

```
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="Observer_No_Chaining_Trans"/>
                  <TransitionRef Id="Observer_Chaining_Trans"/>
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="Observer_Chaining_Route" Name="Observer_Chaining_Route">
          <Route/>
          <StartMode> <Automatic/> </StartMode>
          <FinishMode> <Automatic/> </FinishMode>
        </Activity>
      </Activities>

      <Transitions>
        <Transition From="Observer_Starter" Id="Observer_No_Chaining_Trans"
                    Name="Observer_No_Chaining_Trans" To="Observer_Closer">
          <Condition Type="CONDITION">Observer_Chaining == false</Condition>
        </Transition>
        <Transition From="Observer_Starter" Id="Observer_Chaining_Trans"
                    Name="Observer_Chaining_Trans" To="Observer_Chaining_Route">
          <Condition Type="CONDITION">Observer_Chaining == true</Condition>
        </Transition>
      </Transitions>

    </WorkflowProcess>
  </WorkflowProcesses>
</Package>
```

B Ausgewählte Teile des Quellcodes

B.1 Schnittstellen des Prototypen

B.1.1 Beoboachter

```
public interface Observer extends Remote {

  /**
   * Informiert den Observer ueber eine Aenderung des Zustandes einer
   * zum externen Prozess gehörigen Aktivität.
   */
  public void activityStateChanged(String senderKey,
                                   String receiverKey,
                                   ActivityStateChangedObject data)
              throws InvalidKey, OperationFailed,
                     RemoteException, DBManagerException;

  /**
   * Informiert den Observer ueber eine Aenderung des Prozesszustandes
   */
  public void processStateChanged(String senderKey,
                                  String receiverKey,
                                  ProcessStateChangedObject data)
              throws InvalidKey, OperationFailed,
                     RemoteException, DBManagerException;

  /**
   * Informiert den Observer ueber die Beendigung des gestarteten Remote-Prozesses
   */
  public void completed(String senderKey,
                        String receiverKey,
                        Map result)
              throws InvalidKey, OperationFailed, RemoteException,
                     SharkConnectFailed, DBManagerException;
}
```

Die Typen `String` und `Map` sind Java-eigene Typen. Zusätzlich wurden die beiden folgenden Klassen für die Übermittlung von Informkationen über Zustandsänderungen definiert. Für jedes Attribut ist eine `getXXX()`- und `setXXX()`-Methode definiert. Diese werden nicht angegeben.

```
public class ActivityStateChangedObject implements Serializable {

  private String activityId;
  private String activityName;
  private String lastState;
  private String actState;
  private String stateChangedTime;

  public ActivityStateChangedObject() {}
  public ActivityStateChangedObject(String actId, String actName,
                                    String actState, String lastState,
                                    String changedTime) {
    ...
  }
}

public class ProcessStateChangedObject implements Serializable {

  private String lastState;
  private String actState;
  private String stateChangedTime;

  public ProcessStateChangedObject() {}
  public ProcessStateChangedObject(String actState, String lastState,
                                   String changedTime) {
    ...
  }
}
```

B.1.2 Beobachterserver

```
public interface ObserverServer extends Remote {

  /**
   * Erzeugt ein neues ObserverObjekt und bindet dieses an den RMI-Namensdienst
   */
  public void startnewObserver(String key)
              throws RemoteException, DBManagerException,
                     MalformedURLException;
}
```

B.1.3 Betriebsobjekt

```
public interface Factory extends Remote {

  /**
   * Erzeugt den in diesem Betriebsobjekt definierten Prozess in Enhydra Shark
```

```
   * @return der Schluessel des erzeugten Instanzobjekts
   */
  public String createInstance(String senderKey,
                               String receiverKey,
                               Map contextData)
                throws InvalidKey, RemoteException,
                       DBManagerException;
}
```

B.2 Pseudocode der Komponenten des Instanzobjekts

In diesem Abschnitt werden die Komponenten des implementierten Instanzobjekts in einer Mischung aus Java-Quellcode und Pseudocode dargestellt. Das sind: das `InstanceShark`-Objekt, der Überwachungsthread und das `InstanceFactory`-Objekt, sowie die Änderungen der internen Klassen von Enhydra Shark.

Shark-interne Klasse - WfActvityImpl

In dieser Klasse muss am Ende der Funktion `state_change` die Überprüfung auf einen externen Prozess erfolgen. Wenn der Zustandswechsel einer Aktivität eines externen Prozesses erfolgt, wird der Beobachter informiert oder die Daten werden in den Instanzobjekt-Datenspeicher geschrieben. Dasselbe gilt für die im nächsten Abschnitt erwähnten Änderungen der Klasse `WfProcessImpl` für Prozesszustandsänderungen.

```
InstanceShark instS = null;
boolean extProc = false;

try {
  instS = new InstanceShark(this.process_id(t));
  extProc = instS.checkExternalProcess();
} catch (Exception e) { log ErorMessage }

/* nur wenn es sich um einen externen Prozess handelt, wird fortgefahren */
if (extProc) {
  result = instS.informObserverActivityStateChanged(this.key(t), this.name(t),
                                        this.state(t), oldState,
                                        this.last_state_time(t).getTimestamp());
}
```

Shark-interne Klasse - WfProcessImpl

```
InstanceShark instS = null;
boolean extProc = false;

try {
```

```
  instS = new InstanceShark(this.key(t));
  extProc = instS.checkExternalProcess();
} catch (Exception e) { log ErorMessage }

/* nur wenn es sich um einen externen Prozess handelt, wird fortgefahren */
if (extProc) {

  /* wenn der Prozess nicht beendet wurde, wird nur der Zustandswechsel angezeigt */
  if ( !(state.equals(SharkConstants.STATE_CLOSED_COMPLETED)) ) {
      result = instS.informObserverProcessStateChanged(this.state(t), oldState,
                                          this.last_state_time(t).getTimestamp());
  }
  /* sonst werden zusaetzlich die Kontextdaten zurückgegeben und der Prozess
      wird als beendet markiert */
  else {
    String result = instS.informObserverCompleted(getContext(t));
  }
}
```

Instanzobjekt - InstanceShark

Das Objekt `InstanceShark` wird durch die angepassten internen Klassen von Enhydra Shark temporär erzeugt und für die Prüfung auf einen externen Prozess genutzt. Ist der Prozess durch eine andere Laufzeitumgebung angestossen worden, wird zusätzlich die Benachrichtigung des Beobachters über dieses Objekt vollzogen.

```
public class InstanceShark {

  /* der Schluessel dieser Instanz */
  private String key;
  private String procId;

  public boolean checkExternalProcess() throws SQLException {
    /* durchsuche die Datenbank nach Auftreten des in procId gesp. Prozess-
       identifikators */
  }

  public String informObserverActivityStateChanged(String activityId,
                                      String activityName, String actState,
                                      String lastState, Timestamp time) {

    ActivityStateChangedObject actSCO;
    actSCO = new ActivityStateChangedObject(activityId, activityName, actState,
                                            lastState, time.toString());

    boolean observerInformed = true;
    boolean dbWriteOk = true;
```

```
    read instanceKey and observerKey from DB;

    /* Informiere Beobachter */
    try {
      Observer obs = (Observer) Naming.lookup( observerKey );
      obs.activityStateChanged(instanceKey, observerKey, actSCO);
    } catch (Exception e) {
      observerInformed = false;
    }

    if (observerInformed) {
      return "Observer erfolgreich informiert";
    }

    // sonst: versuche Daten zu speichern
    try {
      save data to database;
    } catch (SQLException e) {
      dbWriteOk = false;
    }

    if (dbWriteOk) {
      return "Observer NICHT informiert, Daten in DB gespeichert";
    }
    else {
      // sonst:
      return "Observer NICHT informiert, Daten NICHT gespeichert";
    }
  }

  public String informObserverProcessStateChanged(String actState, String lastState,
                                                   Timestamp time) {
    ProcessStateChangedObject procSCO;
    procSCO = new ProcessStateChangedObject(actState, lastState, time.toString());

    boolean observerInformed = true;
    boolean dbWriteOk = true;

    read instanceKey and observerKey from DB;

    /* Informiere Beobachter */
    try {
    Observer obs = (Observer) Naming.lookup( observerKey );
    obs.processStateChanged(instanceKey, observerKey, procSCO);

    /* Bei Zustandswechsel zu aborted oder terminated werden alle Daten aus
     * der DB geloescht. Das wird nur getan, wenn der Observer informiert werden
     * konnte. Ansonsten erledigt dies der TimerTask
     */
```

```
    if ( actState.equals(SharkConstants.STATE_CLOSED_TERMINATED)
       || actState.equals(SharkConstants.STATE_CLOSED_ABORTED) ){

          deletAllInstanceData(instanceKey);
      }
    } catch (Exception e) {
        observerInformed = false;
    }

    if (observerInformed) {
        return "Observer erfolgreich informiert";
    }

    // sonst: versuche Daten zu speichern
    try {
      save data to database;
    } catch (SQLException e) {
      dbWriteOk = false;
    }

    if (dbWriteOk) {
      return "Observer NICHT informiert, Daten in DB gespeichert";
    }
    else {
      // sonst:
      return "Observer NICHT informiert, Daten NICHT gespeichert";
    }
  }

  public String informObserverCompleted(Map context) {
    boolean observerInformed = true;
    boolean dbWriteOk = true;

    read instanceKey and observerKey from DB;
    remove unsend data from context; /* nicht bei Prozessstart uebertragene Daten */

    try {
      Observer obs = (Observer) Naming.lookup( observerKey );
      obs.completed(instanceKey, observerKey, context);
    } catch (Exception e) {
      observerInformed = false;
    }
    if (observerInformed) {
      delete instance data from database;
      return "Observer erfolgreich informiert";
    }

    try {
      save context to database;
```

```
      set InstanceProcess completed in database;
    } catch ( SQLException e) {
      dbWriteOk = false;
    }

    if (dbWriteOk) {
      return "Observer NICHT informiert, Daten in DB gespeichert";
    }
    else {
      // sonst:
      return "Observer NICHT informiert, Daten NICHT gespeichert";
    }
  }
}
```

Instanzobjekt - InstanceFactory

Dieses Objekt besteht aus einer Menge von Attributen und get- und set-Methoden zum Auslesen und Setzen der Attribute. Im Gegensatz zum eigentlichen Instanzobjekt (`InstanceShark`) ist die Funktionalität zum Informieren des Beobachters im Überwachungsthread implementiert, der eine Referenz auf die Liste dieser vom verknüpften Betriebsobjekt erzeugten Instanzen besitzt.

```
public class InstanceFactory {
  /* der Schluessel dieser Instanz */
  private String key;

  private String factoryKey;
  private String observerKey;
  private String procId;

  public InstanceFactory(String key) {
    this.key = key;
  }

  ... get- und set-Methoden der Attribute ...
}
```

Überwachungsthread - FactoryTimerTask

Die Funktionen zum Benachrichtigen des Beobachters sind ähnlich denen im Objekt `InstanceShark` und werden an dieser Stelle nicht noch einmal dargestellt. Der Unterschied besteht darin, dass die Daten aus dem Datenspeicher gelesen werden und nicht mehr gespeichert werden müssen, wenn der Beobachter nicht erreichbar ist.

```
public class FactoryTimerTask extends TimerTask {
```

```
  private ArrayList instanceList;

  public FactoryTimerTask(String factoryKey, ArrayList instanceList) {
    this.factoryKey = factoryKey;
    this.instanceList = instanceList;
  }

  /* die zyklisch ausgefuehrte Methode */
  public void run() {
    ArrayList instanceKeys = exctractKeysofInstanceList(instanceList);

    /* Abgeschlossene Prozessinstanzen entfernen */
    checkCompletedInstances(instanceKeys);

    /* Aenderungen an Aktivitaetszustanden feststellen (im Instanzobjekt-
       Datenspeicher) und ggf. übertragen */
    checkActivityStateChanges(instanceKeys);

    /* Abfragen von nicht uebermittelten Prozesszustandsaenderungen
       im Instanzobjekt-Datenspeicher. Wenn vorhanden werden diese an
       den Beobachter übermittelt */
    checkProcessStateChanges(instanceKeys);

    /* Abfragen von beendeten Prozessen, wenn gefunden wird der Beobachter informiert
       und die Zeile geloescht */
    checkProcessCompleted(instanceKeys);
  }
}
```

C Datenbankschemata

Wenn ein Name durch (P) gekennzeichnet ist, so ist dieses Attribut ein Primärschlüssel. Die angegebenen Datentypen beziehen sich auf mySQL.

C.1 Datenbanktabellen des Beobachter-Datenspeichers

ObserverProcesses

Name	Typ	Bedeutung
ObserverKey (P)	varchar(192)	der eindeutige Identifikator des Beobachters
ProcessId	varchar(192)	der zugehörige (startende) Prozess
InstanceKey	varchar(192)	der eindeutige Identifikator der erzeugten Instanz
StartDate	datetime	Startzeitpunkt
Deadline	datetime	spätester Beendigungszeitpunkt

Tabelle C.1: Datenbanktabelle sharkinterop.ObserverProcesses

ObserverProcessData

Name	Typ	Bedeutung
DataId (P)	varchar(192)	der eindeutige Identifikator des Datums
ObserverKey (P)	varchar(192)	der eindeutige Identifikator des Beobachters
Name	varchar(128)	der Name des Datums innerhalb des Sharkprozesses
Mode	varchar(8)	der Modus (IN | OUT | INOUT)

Tabelle C.2: Datenbanktabelle sharkinterop.ObserverProcessData

ObserverProcessInfo

Name	Typ	Bedeutung
ProcessId (P)	varchar(192)	der eindeutige Identifikator des lokalen Prozesses
ActState	varchar(32)	der aktuelle Zustand des entfernten Unterprozesses
LastState	varchar(32)	der vorige Zustand des entfernten Unterprozesses
StateChangedTime (P)	datetime	der Zeitpunkt des Zustandswechsels

Tabelle C.3: Datenbanktabelle sharkinterop.ObserverProcessInfo

ObserverProcessActivityInfo

Name	Typ	Bedeutung
ActivityId (P)	varchar(192)	der Identifikator der Aktivität in der entfernten Shark-Laufzeitumgebung
ActivityName	varchar(128)	der Name der Aktivität im entfernten Unterprozess
ObserverKey(P)	varchar(192)	der eindeutige Identifikator des Beobachters
ActState	varchar(32)	der aktuelle Zustand der Aktivität des entfernten Unterprozesses
LastState	varchar(32)	der vorige Zustand der Aktivität des des entfernten Unterprozesses
StateChangeTime (P)	datetime	der Zeitpunkt des Zustandswechsels

Tabelle C.4: Datenbanktabelle sharkinterop.ObserverProcessActivityInfo

C.2 Datenbanktabellen des Instanzobjekt-Datenspeichers

InstanceProcesses

Name	Typ	Bedeutung
InstanceKey (P)	varchar(192)	der eindeutige Identifikator der Instanz
FactoryKey	varchar(192)	der Identifikator des Betriebsobjekts, welches diese Instanz erzeugte
ProcessId	varchar(192)	der eindeutige Identifikator des gestarteten Shark-Prozesses
ObserverKey	varchar(192)	der eindeutige Identifikator des zugehörigen Beobachters
StartDate	datetime	Startzeitpunkt
Deadline	datetime	spätester Beendigungszeitpunkt
Completed	tinyint(1)	zeigt an, ob der zugehörige SharkProzess beendet wurde
ObserverInformed	tinyint(1)	zeigt an, ob der Beobachter über die Beendigung informiert wurde

Tabelle C.5: Datenbanktabelle sharkinterop.InstanceProcesses

InstanceProcessData

Name	Typ	Bedeutung
DataId (P)	varchar(192)	der eindeutige Identifikator des Datums
ObserverKey	varchar(192)	der eindeutige Identifikator des Instanzobjekts
Name	varchar(128)	der Name des Datums innerhalb des Sharkprozesses
Value	mediumblob	die Rückgabedaten in serialisierter Form

Tabelle C.6: Datenbanktabelle sharkinterop.InstanceProcessData

InstanceProcessInfo

Name	Typ	Bedeutung
ProcessId (P)	varchar(192)	der eindeutige Identifikator des gestarteten SharkProzesses
InstanceKey	varchar(192)	der eindeutige Identifikator der zugehörigen Instanz
ActState	varchar(32)	der aktuelle Zustand des SharkProzesses
LastState	varchar(32)	der vorige Zustand des SharkProzesses
StateChangedTime (P)	datetime	der Zeitpunkt des Zustandswechsels
ObserverInformed	tinyint(1)	zeigt an, ob der Beobachter über den Zustandswechsel informiert wurde

Tabelle C.7: Datenbanktabelle sharkinterop.InstanceProcessInfo

InstanceProcessActivityInfo

Name	Typ	Bedeutung
ActivityId (P)	varchar(192)	der eindeutige Identifikator der Aktivität
ActivityName	varchar(128)	der Name der Aktivität im gestarteten Shark-Prozess
InstanceKey	varchar(192)	der eindeutige Identifikator der zugehörigen Instanz
ActState	varchar(32)	der aktuelle Zustand der Aktivität des Shark-Prozesses
LastState	varchar(32)	der vorige Zustand der Aktivität des Shark-Prozesses
StateChangeTime (P)	datetime	der Zeitpunkt des Zustandswechsels
ObserverInformed	tinyint(1)	zeigt an, ob der Beobachter über den Zustandswechsel informiert wurde

Tabelle C.8: Datenbanktabelle sharkinterop.InstanceProcessActivityInfo

D Konfiguration für die Testfälle

Testfall 1:

Auszug der Konfigurationsdatei des Betriebsobjekts `factories.properties` auf System S2:

```
Factory_Testfall1.host=192.168.0.2
Factory_Testfall1.port=1099
Factory_Testfall1.factoryname=Factory_Testfall1
Factory_Testfall1.engine=SharkFactory
Factory_Testfall1.scope=
Factory_Testfall1.user=Factory
Factory_Testfall1.pass=Factory
Factory_Testfall1.xpdlpackage=Testfaelle
Factory_Testfall1.xpdlversion=1
Factory_Testfall1.xpdlprocess=Testfall1_ExternerProzess
Factory_Testfall1.chaining=true
Factory_Testfall1.startdelay=0s
Factory_Testfall1.period=0s
```

Auszug der Konfigurationsdatei des Anwendungsagenten (`observertoolagent.properties`) auf System S1:

```
rmiregistry.host=192.168.0.1
rmiregistry.port=1099

Factory_Testfall1 = rmi://192.168.0.2:1099/Factory_Testfall1
```

Testfall 2 und Testfall 4:

Auszug der Konfigurationsdatei des Betriebsobjekts `factories.properties` auf System S2:

```
Testfall2.host=192.168.0.2
Testfall2.port=1099
Testfall2.factoryname=Factory_Testfall2
Testfall2.engine=SharkFactory
Testfall2.scope=
Testfall2.user=Factory
Testfall2.pass=Factory
Testfall2.xpdlpackage=Testfaelle
Testfall2.xpdlversion=1
Testfall2.xpdlprocess=Testfaelle_Testfall2_ExternerProzess
Testfall2.chaining=false
Testfall2.startdelay=0s
Testfall2.period=20s
```

Auszug der Konfigurationsdatei des Anwendungsagenten (`observertoolagent.properties`) auf System S1:

```
rmiregistry.host=192.168.0.1
rmiregistry.port=1099

Factory_Testfall2 = rmi://192.168.0.2:1099/Factory_Testfall2
```

Testfall 3:

Auszug der Konfigurationsdatei des Betriebsobjekts `factories.properties` auf System S2:

```
Testfall3_1.host=192.168.0.2
Testfall3_1.port=1099
Testfall3_1.factoryname=Factory_Testfall3_1
Testfall3_1.engine=SharkFactory
Testfall3_1.scope=
Testfall3_1.user=Factory
Testfall3_1.pass=Factory
Testfall3_1.xpdlpackage=Testfaelle
Testfall3_1.xpdlversion=1
Testfall3_1.xpdlprocess=Testfaelle_Testfall3_ExternerProzess1
Testfall3_1.chaining=false
Testfall3_1.startdelay=5s
Testfall3_1.period=25s
```

Auszug der Konfigurationsdatei des Betriebsobjekts `factories.properties` auf System S3:

```
Testfall3_2.host=192.168.0.1
Testfall3_2.port=1099
Testfall3_2.factoryname=Factory_Testfall3_2
Testfall3_2.engine=SharkFactory
Testfall3_2.scope=
Testfall3_2.user=Factory
Testfall3_2.pass=Factory
Testfall3_2.xpdlpackage=Testfaelle
Testfall3_2.xpdlversion=1
Testfall3_2.xpdlprocess=Testfaelle_Testfall3_ExternerProzess2
Testfall3_2.chaining=false
Testfall3_2.startdelay=10s
Testfall3_2.period=30s
```

Auszug der Konfigurationsdatei `observertoolagent.properties` auf System S1:

```
rmiregistry.host=192.168.0.1
rmiregistry.port=1099

Factory_Testfall3_1 = rmi://192.168.0.2:1099/Factory_Testfall3_1
```

Auszug der Konfigurationsdatei des Anwendungsagenten (`observertoolagent.properties`) auf System S2:

```
rmiregistry.host=192.168.0.2
rmiregistry.port=1099

Factory_Testfall3_2 = rmi://192.168.0.1:1099/Factory_Testfall3_2
```

E Verzeichnisstruktur der beiliegenden CD

Installation/	
installation-html/	Installationsanleitung HTML
...	
installation.pdf	Installationsanleitung PDF
sharkinterop-shark.zip	vorübersetztes Installationspaket
sharkinterop-src.zip	Quellcode-Installationspaket
shark-1.0.tar.gz	Enhydra Shark 1.0.1
shark-1.0-src.tar.gz	die Quellen von Enhydra Shark 1.0.1
JavaDoc/	Dokumentation der erstellten Java-Klassen
...	
Testfallausgaben/	alle Ausgaben der Testfälle
Testfall1/	
...	
Testfall2/	
...	
Testfall3/	
...	
Testfall4/	
...	
diplomarbeit.pdf	die erstellte Diplomarbeit

Abkürzungsverzeichnis und Glossar

Abkürzungen

APRE ..... Automatic Participants Runtime Environment 32
ASAP ..... OASIS Asynchronous Service Access Protocol 24
AWSP ..... Asynchronous Web Services Protocol 24
CORBA ..... Common Object Request Broker Architecture 27
HSQLDb ..... Hypersonic SQL Database 44
jPdl ..... jBPM Process definition language 30
JVM ..... Java Virtual Machine 44
OASIS ..... Organization for the Advancement of Structured Information Standards 24
REST ..... REpresentational State Transfer 32
RMI ..... Remote Method Invocation 32
SOAP ..... Simple Object Access Protocol 24
SWAP ..... Simple Workflow Access Protocol 24
WAPI ..... Workflow Application Programming Interface 10
Wf-XML ..... Workflow - Extensible Markup Language .. 26
WfMC ..... Workflow Management Coalition 2
WfMC-TC ..... Workflow Management Coalition - Technical Comittee Document 4
WfMS ..... Workflow Management System 1
XPDL ..... XML Process Definition Language 6

Glossar

Activity ..... Aktivität, Aufgabe, Tätigkeit 4
Activity Block ..... Aktivitätsblock 15
Activity Instance ..... Aktivitätsinstanz 18
Activity Resource ..... Aktivitätsressource 26

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Hol95] Hollingsworth, David: *The Workflow Reference Model.* : Workflow Management Coalition, Jan 1995. – Document Number TC00-1003, Document Status: Issue

[jPD] JBoss.com (Hrsg.): *jPdl Reference Manual.* `http://www.jboss.com/products/jbpm/jpdl`. – Online–Ressource, Abruf: 21. 03. 2005

[Koe04] Koenig, John: *JBoss jBPM WHITE PAPER.* Version: 2004. `http://www.jboss.com/pdf/jbpm_whitepaper.pdf`. – Online–Ressource, Abruf: 21. 03. 2005

[Kre] Kreplin, Klaus-Dieter: *Konkordanz englischer und deutscher Begriffe des Workflow Management.* – Diskussionspapier - keine durch die WfMC autho-risierte Version

[LSW] Lipp, Michael ; Schlüter, Holger ; Weidauer, Christian: *The Danet Workflow Component - User Manual.* `http://wfmopen.sourceforge.net/userapi/de/danet/an/workflow/api/doc-files/user-manual.html`. – Online–Ressource, Abruf: 10. 04. 2005

[Mar99] Marin, Mike: *Workflow Standard - Interoperability Abstract Specification.* : Workflow Management Coalition, Nov 1999. – Document Number TC-1012, Document Status: Draft

[Met] Mettraux, John: *OpenWFE - Open source WorkFlow Engine.* `http://www.openwfe.org/docbook/build/index.html`. – Online–Ressource, Abruf: 21. 03. 2005

[Nor02] Norin, Roberta: *Workflow Process Definition Interface – XML Process Definition Language.* : Workflow Management Coalition, Oct 2002. – Document Number: WFMC-TC-1025, Document Status: Final Draft

[Obj98] Object Management Group (OMG): *Workflow Management Facility.* OMG BODTF RFP Submission, July 1998

[Ope] *OpenWFE Technical Presentation.* `http://www.openwfe.org/openwfe.swf`. – Online–Ressource, Abruf: 21. 03. 2005

[SGP04] Swenson, Keith D. ; Gilger, Mike D. ; Predhan, Sameer: *Wf-XML 2.0 - XML Based Protocol for Run-Time Integration of Process Engines.* : Workflow Management Coalition, Nov 2004

[Sha] Enhydra.org (Hrsg.): *Shark 1.0 Documentation.* `http://shark.objectweb.org/doc/1.0/index.html`. – Online–Ressource, Abruf: 18. 03. 2005

[SRK04] Swenson, Keith ; Ricker, Jeffrey ; Krishnan, Mayilraj: *Asynchronous Service Access Protocol (ASAP).* : OASIS, Jun 2004. – Working Draft G

[Wor99] Workflow Management Coalition: *Terminology and Glossary.* Feb 1999. – Document Number: WFMC-TC-1011, Document Status: Issue