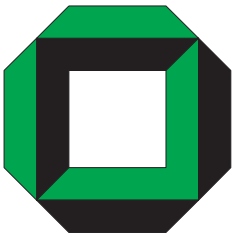
Universität Karlsruhe (TH)



Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Diplomarbeit

Strukturelle Adaption von BPMN/BPEL-Workflows und Integration mit dem WfMS Intalio

Vorgelegt von: **Cand. Inform. Thorsten Haberecht**

Betreuer:
Prof. Dr.-Ing. Klemens Böhm
Dipl.-Inform. Jutta Mülle

Tag und Ort der Einreichung:

Karlsruhe, den 31. März 2009

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 31.03.2009

.....

Ort, Datum (Thorsten Haberecht)

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Diese Arbeit behandelt ein Thema aus dem Bereich der adaptiven Workflow-Management-Systeme. Untersucht wird, wie es unter Berücksichtigung der Prozessdefinitionsstandards *BPMN* und *BPEL* möglich ist, bei laufenden Workflow-Instanzen Änderungen an der Prozessdefinition vorzunehmen. Um die Umsetzbarkeit des hierfür entwickelten Konzeptes aufzuzeigen, implementieren wir dieses in Form einer unabhängigen Adaptionskomponente und bereiten eine Integration des Moduls in die Workflow-Management-Systeme *FloeCom* und *Intalio* vor.

1.1 Motivation

Workflow-Management-Systeme haben sich in den vergangenen Jahren als wichtiger Baustein in der Unternehmenssoftware-Landschaft etabliert. Mit ihrer Hilfe lassen sich Geschäftsprozesse planen, modellieren und implementieren. Im laufenden Betrieb steuern Workflow-Systeme auf Basis der Geschäftsprozessbeschreibung die laufenden Prozessinstanzen. [Obe96]

Während akademische Untersuchungen auf dem Gebiet der Prozessmodellierung in der Vergangenheit häufig auf Grundlage von Petri-Netzen als Modellierungstechnik erfolgten, war die Praxis geprägt von einer Vielzahl an herstellerspezifischen Prozessdefinitionsmethoden. Hierfür gab es mehrere Gründe:
Zum einen sind die akademisch geschätzten Petri-Netze zur Modellierung von umfangreichen Unternehmensprozessen unzureichend geeignet. Erweiterte Petrinetz-Modelle brachten zwar Verbesserungen, was beispielsweise die Einbindung von Daten oder Zeitbezügen angeht. Trotzdem weisen Petrinetze deutliche Mängel gegenüber anderen Prozess-Beschreibungstechniken auf, insbesondere hinsichtlich der Strukturierung von Abläufen, der Definition von Events oder der Integration von Organisationsmodellen. Also wurden Alternativen entwickelt, die oft für bestimmte Einsatzzwecke optimiert waren und so zu spezialisierten Softwarelösungen führten.
Ein Grund für die Entwicklung von anwendungsgebiet-optimierten Workflow-Lösungen inklusive der entsprechenden Prozessdefinitions-Modelle war auch die unklare Abgrenzung dieser Systeme zu anderen Bereichen der Unternehmenssoftware. Dieses Problem spiegelt sich bis heute in einer teilweise sehr unklar definierten Begriffswelt im Bereich des Workflow-Managements wider. Allein die Begriffsvielfalt zur Benennung eines Workflow-Systems an sich reicht hierbei von „Postkorbsystemen“[1] über „Business-Process-Management-Systeme“[2] bis hin zu „E-Business

[1] Anschaffungsziel einer Stuttgarter Versicherungsgesellschaft zur Unterstützung der Geschäftsabläufe aus dem Jahr 2008

[2] http://www.intalio.com

Integration Solutions“[3].

Wie in anderen IT-Bereichen wuchs aber auch hier der Wunsch nach allgemein anerkannten Standards. Insbesondere wurden seit Beginn dieses Jahrzehnts u.a. zwei Prozessdefinitions-Standards erarbeitet: *BPMN*, die *Business Process Modelling Notation*, für die graphische Prozess-Modellierung und *BPEL*, die *Business Process Execution Language*, als maschinenlesbare Prozessdefinition auf Ausführungsebene. Diese Entwicklung wurde zunächst von den großen Software-Firmen wie IBM und Microsoft angeschoben, fand aber bald Unterstützung durch eine Vielzahl anderer Branchenvertreter.

Die Notwendigkeit standardisierter Techniken auf mehreren betroffenen Ebenen zeigte sich spätestens als der Bedarf nach unternehmensübergreifenden, automatisierten Prozessen aufkam. Im Rahmen der Idee einer Serviceorientierten Architektur (SOA) wurden Standards für alle beteiligten Schichten entwickelt, soweit diese noch nicht vorhanden waren. Dies betrifft auch den Aspekt der Prozessmodellierung und der Prozessdefinition.

So ist es heutzutage möglich, Teile von Prozessen auszulagern und sie über die entsprechenden Schnittstellen trotzdem in die eigenen automatisierten Unternehmensprozesse einzubinden. Mit BPEL existiert ein generischer, anwendungsgebiet-unabhängiger Standard zur Prozessdefinition, der unter anderem Unterstützung bietet für die Einbindung von Daten und Timern, für Fehlerbehandlungsroutinen und Zurücksetzungen, für die Integration von externen Services und natürlich für die Bereitstellung des Prozesses selbst als eigener Webservice.

Ein Problem mit weitreichenden Abhängigkeiten ergibt sich aber aus der Notwendigkeit, Prozesse von Zeit zu Zeit zu ändern. Als Gründe hierfür kommen u.a. Prozessoptimierungen oder aber veränderte Rahmenbedingungen in Frage, die eine Anpassung der Geschäftsabläufe erforderlich machen. Auch laufende Prozessinstanzen können hiervon betroffen sein.
Für eine Änderung von laufenden Prozessinstanzen existiert bisher allerdings keine anerkannte Methode. In den größeren verfügbaren Workflow-Management-Systemen ist eine Durchführung von Prozess-Adaptionen bereits gestarteter Instanzen nicht möglich. Dies hat zur Folge, dass Instanzen, die auf einem zu ändernden Prozessschema basieren, entweder abgebrochen werden müssen, oder die Änderungen werden – soweit möglich – händisch durchgeführt, gegebenenfalls für jede Instanz einzeln.
Deshalb wird seit längerem an Workflow-Systemen geforscht, die eine Anpassung des Prozesses zur Laufzeit ermöglichen. Grundsätzlich unterscheiden lassen sich hierbei zwei Ansätze: Unter *dynamischen Workflow-Management-Systemen* versteht man Systeme, bei denen die Workflows nur lose definiert sind. Beispielsweise ist kein fester Prozessablauf definiert, sondern die einzelnen Prozesselemente, die *Tasks*, liegen als einzelne Bausteine vor. Zur Laufzeit kann hieraus implizit (durch einfaches Ausführen eines Tasks) oder explizit (durch Festlegung einer gewissen Reihenfolge im Voraus) ein Aufgaben-Ablauf gebildet werden, wie man ihn von herkömmlichen Prozessdefinitionen kennt.
Die Alternative hierzu ist, die bisher entwickelten Workflow-Management-Systeme dahingehend zu erweitern, dass die Prozess-Instanzen nach wie vor auf einem festen Prozessschema basieren, dieses aber im laufenden Betrieb geändert werden kann. Die Änderung des Schemas muss sich auf die zugehörigen laufenden Instanzen auswirken. Man spricht hier von *adaptiven WfMS*.

Zu beiden Ansätzen existiert eine Vielzahl wissenschaftlicher Veröffentlichungen. Diese bieten zum Teil interessante Problemlösungsansätze, stützen sich aber fast durchgehend auf eigens für diesen Zweck entwickelte Workflow-Modelle. Während ein solcher Ansatz für das noch recht neue Forschungsgebiet der dynamischen Wf-Management-Systeme naheliegend ist, führt er im Bereich der adaptiven Workflow-Management-Systeme zur Vernachlässigung in Forschung und Industrie mittlerweile anerkannter Standards. Dies betrifft sowohl die Modellierungsebene mit BPMN wie auch die Prozessdefinition auf Ausführungsebene mit BPEL.

[3] http://www.oracle.com/technology/products/integration/workflow/workflow_fov.html

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Beitrag zur Entwicklung eines adaptiven Workflow-Management-Systems (WfMS) zu leisten. Wir entwickeln ein allgemeines Konzept, wie Änderungen an der Ablaufbeschreibung vorgenommen werden können. Exemplarisch führen wir dies für die zwei Basis-Änderungsoperationen *Einfügen* und *Löschen einer Aktivität* aus. Das Adaptionskonzept soll auf die Prozessdefinitions-Standards BPMN und BPEL aufbauen.
Im Rahmen dieser Arbeit möchten wir einen Weg für eine durchgängige Prozessadaption aufzeigen. Hierbei gehen wir von benutzerinitiierten Änderungen am Prozessmodell aus. Dies unterscheidet den hier verfolgten Ansatz von dem vorangegangener Arbeiten, bei denen systemseitig ausgelöste Prozessänderungen untersucht wurden [Wei08]. In unserem Konzept sollen auf Prozessmodellierungsebene Änderungen vorgenommen, diese nach BPEL übersetzt und auf Instanzebene übernommen werden können.
Die gefundene Lösung ist prototypisch als unabhängige Komponente zu entwickeln und zu implementieren. Diese soll dann wiederum in das institutseigene FloeCom-System und in das Open Source Business-Process-Management-System der Firma Intalio eingebunden werden.

1.3 Aufbau der Arbeit

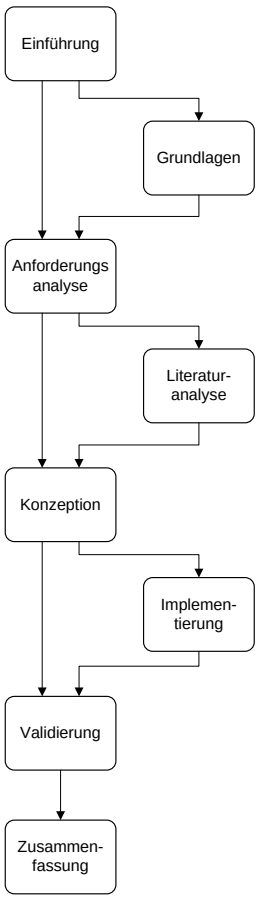


Abbildung 1.1: Kapitelübersicht

Die vorliegende Arbeit ist wie folgt aufgebaut: In Kapitel 2 führen wir kurz in den Bereich des Workflow-Managements ein und stellen die Grundlagen vor, die für unsere Arbeit von Bedeutung sind. Dies betrifft im Einzelnen die wesentlichen Aspekte eines Workflow-Management-Systems, die Prozessdefinitionsstandards sowie die beiden Systeme, in die unsere Adaptionskomponente später integriert werden soll – das am Institut entwickelte FloeCom-System und das WfMS der Firma Intalio.

Im dritten Kapitel gehen wir auf die Anforderungen ein, die an das Adaptionskonzept gestellt werden.
Die daran anschließende Literaturanalyse zeigt ausschnittsweise den für uns relevanten Stand der Forschung auf dem Gebiet der Prozess-Adaption und verdeutlicht einige Ideen anderer Arbeiten, auf die wir im weiteren Verlauf zurückgreifen werden.

Das Konzeptionskapitel stellt das von uns entwickelte Vorgehen zur Prozessadaption vor. Wir stellen hier zunächst das grundsätzliche Vorgehen vor, bevor wir für die Prozessmodellierung wie auch für die Ausführungsebene ein formales Modell definieren. Auf Basis dieser, an den Standards orientierten Modelle, zeigen wir die Durchführung der Änderungsoperationen auf beiden Ebenen auf. Auf die hierfür notwendigen Voraussetzungen wird ebenfalls eingegangen.
Kapitel 6 behandelt die Implementierung der Adaptionskomponente. Ebenso gehen wir auf die Integration in die Workflow-Management-Systeme FloeCom und Intalio|-BPMS ein.

Unser Adaptions-Konzept und die entwickelte Komponente werden in Kapitel 7 anhand zweier Prozesse validiert. Wir greifen hierbei auf Abläufe zurück, die bei der Benutzung des Proceedings-Management-Systems *ProceedingsBuilder* auftreten und im Rahmen der Entwicklung dieses Systems am IPD modelliert wurden.

Im abschließenden Kapitel *Zusammenfassung und Ausblick* zeigen wir noch einmal die wesentlichen Aspekte unseres Konzepts auf und geben einen Ausblick auf das mögliche Vorgehen zur weiteren Ausführung dieses Ansatzes.

Kapitel 2

Grundlagen

In diesem Kapitel führen wir in die Thematik des Workflow-Managements ein. Hierfür erklären wir zunächst die gängigen Fachbegriffe und erläutern Einsatzzweck und Eigenschaften von Workflow-Management-Systemen. Zusätzlich geben wir einen Überblick über die wichtigsten Prozessbeschreibungs-Spezifikationen, BPMN und BPEL. In den beiden letzten Abschnitten des Kapitels stellen wir die zwei Workflow-Systeme vor, in die unsere Adaptionskomponente integriert werden soll.
Wir beschränken uns bei den folgenden Ausführungen hauptsächlich auf die Aspekte, die für die Entwicklung unseres Adaptionskonzeptes von Bedeutung sind. Weiterführende Quellen sind in den jeweiligen Abschnitten angegeben.

2.1 Begrifflichkeiten

Wie bereits angesprochen, ist der Bereich des Workflow-Managements gekennzeichnet durch eine teilweise uneinheitliche Verwendung der spezifischen Fachbegriffe. Dies spiegelt sich am besten wider in der oft undifferenzierten Verwendung der Begriffe *Workflow*, *Prozess* und *Business Process*. Wir versuchen zumindest für diese Begriffe einen kurzen Überblick über die korrekte Art der Verwendung zu geben.

Eine Prozessdefinition hat ihren Ursprung normalerweise auf der Ebene der Betriebsabläufe. Das Ziel ist die Unterstützung eines Geschäftsprozesses durch ein IT-System. Das Verständnis des Begriffs *Geschäftsprozess* ergibt sich recht intuitiv: „Ein Geschäftsprozess ist eine Abfolge von Aktivitäten, die der Erzeugung eines Produktes oder einer Dienstleistung dienen.“ [RHS04] Der in Bezug auf eine IT-Unterstützung automatisierbare Teil eines Geschäftsprozesses wird als üblicherweise *Workflow* bezeichnet.

Auf der System-Ebene wird nun ein *Prozess* definiert, der dazu dient, den Workflow umzusetzen. Gleichbedeutend wird für *Prozess*, engl. *process*, auch auf dieser Ebene oftmals der Begriff *Workflow* verwendet.
Die Prozessdefinition enthält die notwendigen Arbeitsschritte, aber auch die Prozessregeln, die Anbindungen an die benötigten Daten, eine Möglichkeiten zur Einbindung einer Rechteverwaltung und anderes mehr. Der Prozess wird auf einem *Workflow-Management-System* ausgeführt, das sich neben der reinen Prozesssteuerung auch um Aspekte wie die Anbindung von externen Anwendungen und Datenbanken, um die Benutzerverwaltung und die Kommunikation mit anderen Workflow-Management-Systemen kümmert [Wor99]. Dieses Begriffsverständnis ist unter anderem durch das Workflow-Architektur-Modell der *Workflow-Management Coalition* (WfMC) geprägt [Wor95].

In den letzten Jahren setzte sich die Idee einer serviceorientierten Architektur (SOA), in der alles als ein Dienst betrachtet wird, mehr und mehr durch. Dieses Architekturprinzip stellt zum einen eine Weiterentwicklung des zentralistischen Konzepts der WfMC dar, zum anderen wurde das klassische Verständnis eines Workflows als feste Ablaufstruktur von einzelnen Tasks abgelöst durch eine neue Sicht auf einen Prozess: Ein automatisierter Geschäftsprozess (Business Process) kann heute modelliert werden als lose Koppelung von Diensten, die miteinander kommunizieren.
Diese Entwicklung in der Architektur von IT-Systemen führte zu neuen Softwareprodukten, die für eine solche Architektur notwendig sind, wie z.B. dem *Enterprise Service Bus*. Parallel dazu setzte sich – auch im Deutschen – ein neues Begriffsverständnis durch, das unter einem *Business Process* das versteht was früher in anderer Form einmal der Workflow(-Prozess) war und heute ein webservice-basierter Prozess ist. Trotzdem wird der Begriff *Workflow* aber auch oft gleichbedeutend verwendet. Das Schlagwort *Business Process Management (BPM)* umfasst alle Aspekte der Modellierung, Definition und Ausführung von automatisierten Geschäftsprozessen. [Hav05]

Bei dem in dieser Arbeit entwickelten Konzept bewegen wir uns auf Ebene der Modellierung von Prozessen, der Prozessdefinition und der Prozess-Ausführung. Die Spezifikationen, auf die wir zurückgreifen, benutzen hauptsächlich den Begriff des *(Business) Process*. Hieran orientieren wir uns und sprechen analog dazu von *Prozessen*.

2.2 Workflow-Management-Systeme

Die grundlegenden Aufgaben eines Workflow-Management-Systems haben wir bereits angesprochen. Wir möchten nun die Aspekte noch vertiefen, die für diese Arbeit von Bedeutung sind.

2.2.1 Überblick

Ein modernes Workflow-Management-System unterstützt die Modellierung von Prozessen und die Erzeugung einer Prozessdefinition. Es ermöglicht die Einbindung von Daten und die Zuordnung von Benutzern zu bestimmten Prozessschritten wie auch die Einbindung von Schnittstellen zu externen Diensten, Softwarekomponenten und Anwendungsprogrammen. Seine zentrale Aufgabe ist die Ausführung der definierten Prozesse. In den meisten Fällen bietet ein WfMS zusätzliche Administrations-Unterstützung, wie die Überwachung laufender Prozesse anhand verschiedener Kriterien, z.B. der Ressourcenauslastung. Ebenso existiert meist eine Komponente, die die nachträgliche Analyse der Prozessverläufe ermöglicht. Abbildung 2.1 zeigt den grundlegenden Aufbau eines Workflow-Management-Systems.

Auf die Auswirkungen, die die Entwicklung hin zu einer serviceorientierten IT-Architektur (SOA) auf das Aussehen und die Integration von Workflow-Management-Systemen hat, möchten wir an dieser Stelle nicht näher eingehen. Für unser grundlegendes Adaptionskonzept ist die Architektur selbst nur von minimaler Bedeutung, wenn auch dieses Architekturprinzip natürlich in die von uns verwendeten Spezifikationen eingeflossen ist.
Dem interessierten Leser sei als gute Einführung in dieses Thema das Buch *Service-orientierte Architekturen mit Web-Services* von *Ingo Melzer* [Mel07] empfohlen. Es behandelt auf technischer Ebene in gut strukturierter und umfassender Form alle diejenigen Bereiche, die vom Paradigma einer serviceorientierten Architektur betroffen sind.

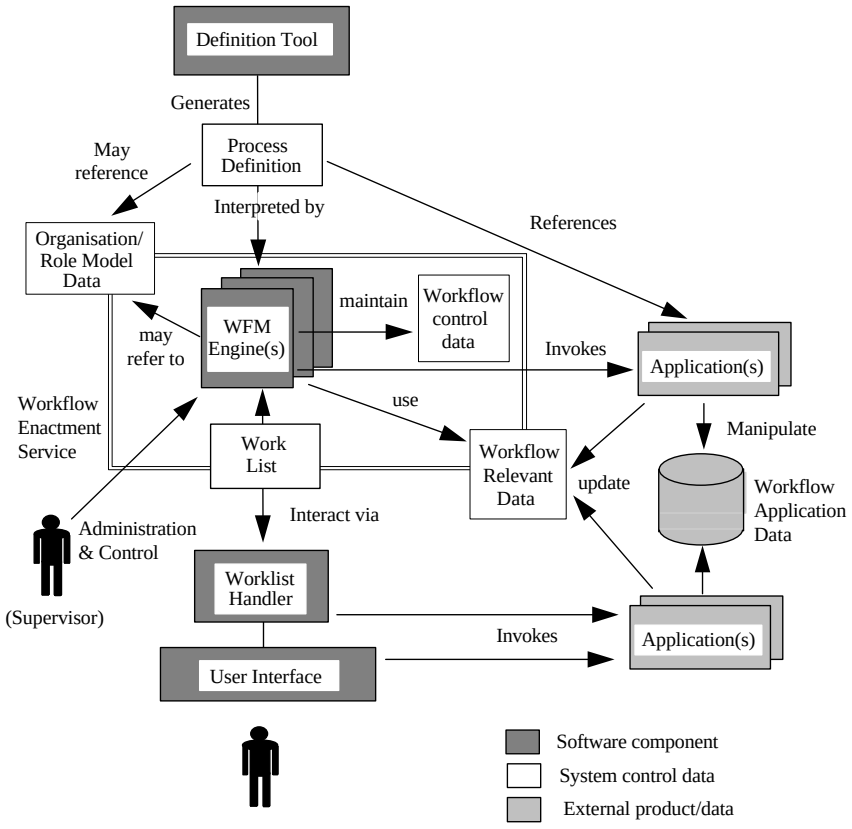


Abbildung 2.1: Aufbau eines Workflow-Management-Systems (aus [Wor95])

2.2.2 Instanzen

Eine wesentliche Rolle in dieser Arbeit spielt der Begriff der Prozess-*Instanzen*. Bisher wurde etwas vereinfacht davon gesprochen, dass ein Prozess auf dem Workflow-System, genauer auf der Workflow-*Engine*, ausgefügt wird. Genauer betrachtet wird zunächst ein Prozessschema definiert und dieses auf die Workflow-Engine geladen. Soll der durch dieses Schema definierte Prozess gestartet werden, wird eine Prozess*instanz* erzeugt. Diese Instanz repräsentiert einen *Fall* (engl.: *case*) in einem Geschäftsprozess, also beispielsweise eine einzelne Kundenanfrage.
Die Instanz ist damit gewissermaßen der Zustand eines Prozesses in Bezug auf einen bestimmten Fall. Sie enthält die zugehörigen Prozessdaten, den Ausführungszustand des Prozesses und systemspezifische Verwaltungsdaten, wie z.B. den Startzeitpunkt der Instanz oder die Durchlaufzeit verschiedener Prozesselemente.

2.2.3 Subprozesse

Subprozess-Elemente dienen als Strukturierungshilfe bei der Definition von Workflows. Sie kommen insbesondere auf der Modellierungsebene zum Einsatz. Ein Teil eines Prozesses wird hierbei als Subprozess definiert und kann in der Folge wie ein einzelnes Prozesselement behandelt werden. Charakteristisch hierfür ist eine gewisse semantische Eigenständigkeit des entsprechenden Prozessabschnittes. So könnte beispielsweise die mehrstufige Zusammenstellung einer Warenlieferung innerhalb eines Bestellvorgangs als Subprozess modelliert werden.
Subprozesse können auf zwei Ebenen verwendet werden. In vielen Modellierungstools sind Sub-

prozesse auf die rein visuelle Ebene beschränkt. Prozesselemente werden hier zu einem einzelnen Subprozess-Element „verkleinert“, um die Übersichtlichkeit des Prozesslayouts zu verbessern. Dieses Prinzip ist aus anderen Bereichen bekannt, unter anderem als Code-Folding bei Entwicklungsumgebungen.
Die mächtigere Art der Anwendung besteht darin, ganze Prozesse oder Prozessteile wiederzuverwenden. Das heißt, ein einmal modellierter Prozess(teil) wird in einer Art Repository abgelegt und kann innerhalb von anderen Prozessen als Subprozess eingebunden werden. Die BPMN-Spezifikation sieht dies vor. Dort kann der referenzierte Prozess(teil) innerhalb des gleichen Diagramms oder in einem anderen Diagramm vorhanden sein.

Bei der Betrachtung der Prozessdefinitionsebene mit der Business Process Definition Language stellt man fest, dass hier kein Subprozess-Konstrukt vorgesehen ist. Bei der Übersetzung von BPMN oder einer anderen Modellierungsart nach BPEL müssen die vorhandenen Subprozesse ausgerollt werden, d.h. alle Subprozesselemente werden zu direkten Elementen des Hauptprozesses. Ist der gleiche Subprozess beispielsweise zweimal vorhanden, wird sein Inhalt in BPEL auch zweimal explizit aufgeführt.
Da diese Redundanz nicht unbedingt wünschenswert ist, gibt es Vorschläge zur Erweiterung von BPEL um Subprozess-Elemente. Hierunter am meisten Beachtung gefunden hat das von IBM und SAP veröffentlichte Whitepaper [KKL+05].
Eine weitere Möglichkeit der Subprozess-Verwendung besteht darin, einen Subprozess als eigenständigen Prozess zu übersetzen und ihn dann auf BPEL-Ebene als Webservice einzubinden. Diese Vorgehensweise findet mittlerweile in der Praxis häufige Anwendung.

2.2.4 Adaptivität

Die Änderung von laufenden Prozessinstanzen ist eine der wesentlichen Herausforderungen bei der Forschung über Workflow-Management-Systeme. Die Praxisrelevanz dieses Themas wird schnell deutlich, wenn man sieht, dass die medizinische Behandlung von Patienten im Krankenhaus das am häufigsten verwendete Adaptivitätsszenario ist. Starre, inflexible Abläufe würden hier nicht den realen täglichen Anforderungen entsprechen, trotzdem ist ein gewisses strukturiertes und IT-unterstütztes Vorgehen auch im Einzelfall unbedingt erwünscht.
Aber auch in Anwendungsbereichen, in denen man es mit lange laufenden Instanzen zu tun hat, können Anpassungen des Workflows im laufenden Betrieb notwendig werden. Beispielsweise kann es erforderlich sein, Änderungen der gesetzlichen Rahmenbedingungen in den Prozess einfließen zu lassen.

Die Forderung nach Adaptivität im Workflow-Management-Bereich betrifft mehrere Ebenen und Aspekte eines Workflows. Vorrangiges Ziel ist es natürlich, Änderungen am Prozessablauf durchzuführen zu können, also beispielsweise einen neuen Task einzufügen oder einen vorhanden zu löschen.
Es kann aber beispielsweise auch vorkommen, dass verwendete Webservice-Schnittstellen geändert wurden und deshalb eine Anpassung der eigenen prozessinternen Zugriffe notwendig ist. Dies kann auch Änderungen am Prozess-Datenmodell nach sich ziehen.
Am Institut für Programmstrukturen und Datenorganisation (IPD) der Universität Karlsruhe wurden im Rahmen der Arbeit an einem Konferenz-Proceedings-Management-System Anforderungsklassen in Bezug auf Adaptivitätsfunktionen erarbeitet [MBRS]. Mehrere darauffolgende Arbeiten am gleichen Institut haben sich mit der Umsetzung der unterschiedlichen Adaptivitäts-Aspekte befasst.

Wir beschränken uns in dieser Arbeit auf die Betrachtung von reinen Prozessablauf-Änderungen. Die vorhandenen wissenschaftlichen Arbeiten zu diesem Aspekt weisen eine breite Spanne unterschiedlicher Herangehensweisen auf. Teilweise wird ein Geschäftsablauf einfach als Menge unzusammenhängender Tasks definiert, die dann zur Laufzeit meist implizit durch eine entsprechende Benutzerauswahl in eine Reihenfolge gebracht werden.

Andere Adaptionsmodelle sehen Änderungen im Prozessablauf bereits bei der Modellierung vor. Das ADEPT-Modell [Rei00] und andere Konzepte kennen beispielsweise Ausnahmekanten als Modellierungselemente, um in Ausnahmefällen einen anderen Prozessverlauf als den Standardfall zu ermöglichen. Ein anderer Ansatz schlägt die Verwendung von *Worklets* vor, subprozess-ähnlichen Elementen, von denen zur Laufzeit an den vorgesehen Stellen die passende Version ausgewählt werden kann. Die Auswahl kann sowohl rein benutzerdefiniert geschehen, wie auch durch einen Regelkatalog unterstützt werden. [AHEA06]
Das Problem dieser beiden Herangehensweisen ist, dass sie ein Prozessmodell voraussetzen, das die erwähnten Adaptivitätsfunktionen unterstützt. In den meisten Fällen verwenden die Autoren der entsprechenden Veröffentlichungen ein individuelles, als Grundlage für den jeweiligen Lösungsvorschlag entwickeltes Prozessmodell. Dieses Vorgehen mag zu grundlegenden Erkenntnissen im Bereich der Adaptivitätsforschung führen. Allerdings lassen sich die entwickelten Konzepte auf mittelfristige Sicht in den wenigsten Fällen in die Praxis übertragen, da hier andere Prozessmodelle zum Einsatz kommen.

Wir haben deshalb einen dritten Weg gewählt. Wir gehen von einem Prozessmodell aus, in dem die Abläufe fest definiert sind und das ursprünglich nicht zur Umsetzung von Adaptivitätsfunktionen vorgesehen war. Auf diesem Modell ermöglichen wir die Durchführung von Änderungen am Prozessablauf. Wir betrachten hierbei sowohl die Modellierungs- als auch die Prozessdefinitions- und Ausführungsebene. Zur Prozessmodellierung verwenden wir die Business Process Modelling Notation (BPMN), bei der Prozessdefinition greifen wir auf die Business Process Execution Language (BPEL) zurück. Beide Spezifikationen sind anerkannte Industrie-Standards.

2.3 Prozessdefinitions-Standards

Noch vor wenigen Jahren äußerte sich Jon Pyke, Vorsitzender der Workflow Management Coalition, kritisch über die zunehmende Zahl an Gremien, die sich mit der Entwicklung eigener Prozessdefinitions-Standards beschäftigen. Im Vorwort des Workflow Handbook 2004 [Fis04] schreibt er: „So far the greatest achievement of this increased activity of standards development is confusion. There is confusion over which standards fit where and which apply to what situation.“

Mittlerweile hat sich die Situation etwas verbessert. Die Spezifikationen wurden weiterentwickelt und es haben sich zwei Standards weitgehend durchgesetzt: Die *Business Process Modellierung Notation (BPMN)* für die Prozessmodellierung und die *Business Process Execution Language (BPEL)* auf Prozessdefinitions- und Ausführungsebene. Die BPMN-Spezifikation liegt in der Version 1.2 vor, die von BPEL in der Fassung 2.0. Fast alle größeren Hersteller von Workflow-Management-Systemen unterstützen diese beiden Standards mittlerweile in irgendeiner Form oder arbeiten zumindest an einer Integration.

Betrachtet man andere Bereiche und Schichten einer SOA-Architektur, so steht eine Vielzahl an weiteren standardisierten Spezifikationen für die unterschiedlichen Anwendungsbereiche zur Verfügung. Eine grafisch aufbereitete Übersicht hierüber findet sich in [STH07]. Wir beschränken uns im Folgenden auf die Vorstellung der für uns relevanten Standards, die den Prozessdefinitionsbereich betreffen – BPMN und BPEL.

2.3.1 BPMN – Business Process Modelling Notation

Als Erstes betrachten wir den Prozess-Modellierungs-Standard BPMN.

Überblick

Die Business Process Modelling Notation (BPMN) ist eine grafische Darstellungsform zur Modellierung von Geschäftsprozessen. Sie wurde mit dem Ziel entwickelt, eine Art der Prozessdarstellung zu schaffen, die sowohl für die am Geschäftsprozess beteiligten Fachkräfte als auch für die Workflow-Spezialisten verständlich ist und somit als gemeinsame Kommunikationsgrundlage dienen kann. Die Spezifikation liegt mittlerweile in der Version 1.2 vor, an der 2.0-Fassung wird gearbeitet. Im Zeitraum der Erstellung dieser Arbeit war die Version 1.1 aktuell – diese dient auch als Grundlage für unser Adaptionskonzept. Koordiniert und betreut wird die Weiterentwicklung der Spezifikation von der *Object Management Group (OMG)*.
Innerhalb eines Prozess-Diagramms erlaubt BPMN die Modellierung von privaten Prozessen, von abstrakten Prozessen und von Prozessen mit mehreren Teilnehmern. Die Spezifikation definiert hierbei sowohl die grafische Darstellung wie auch die semantische Bedeutung. Neben der reinen Notation und ihrer semantischen Bedeutung werden für die einzelnen Diagrammelemente auch Attribute definiert, deren Integration für eine standardkonforme Implementierung erforderlich ist.
Eine explizite Datenflussmodellierung findet in BPMN nicht statt. Es werden zwar Nachrichtenflüsse definiert, eine weitergehende Darstellung des Datenflusses findet aber nicht statt. Ebenso wenig ist die direkte Prozessdaten-Definition Bestandteil der Notation.

Prozessmodellierung

Die Grundlage für eine Geschäftsprozessmodellierung mit BPMN bildet ein Prozess-Diagramm (*Process Diagram*). Darin werden eine oder mehrere *Swimlanes* angelegt, die jeweils Prozessteile enthalten. Die Swimlanes dienen hierbei der Partitionierung des Gesamtprozesses auf mehrere Prozessteilnehmer beziehungsweise der Unterscheidung zwischen dem privatem Prozessteil und externen Prozessabschnitten bzw. Webservices.

Die eigentlichen Prozesselemente lassen sich nach *Aktivitäten*, *Gateways* und *Events* gruppieren. Aktivitäten sind entweder einzelne Tasks oder Subprozess-Elemente. Beide Elementtypen werden weiter unterschieden. Es gibt beispielsweise einfache Tasks, die nicht weiter definiert sind, es existieren *Send*- und *Receive*-Tasks für die Nachrichtenübermittlung, *User*-Tasks für Aufgaben, die von einem menschlichen Benutzer ausgeführt werden müssen und noch weitere Task-Arten. Subprozesse lassen sich als Prozess-Element definieren, das die Subprozess-Elemente direkt enthält. Dies ist die gängigste Art der Subprozess-Verwendung. Es sind aber auch Verweise auf Subprozesse im gleichen oder in anderen Diagrammen erlaubt, was eine Wiederverwendung des einmal definierten Subprozesses möglich macht.
Sogenannte *Gateways* dienen der Modellierung von Verzweigungen, sie stellen die eigentlichen Routing-Konstrukte dar. Es gibt mehrere Gatewaytypen mit mehreren Untertypen, die zusammengenommen die Modellierung von daten- oder eventbasierten AND-, OR- und XOR-Verzweigungen ermöglichen.
Events sind vorgesehene Ereignisse, die im Geschäftsablauf auftreten können. Hierzu zählen beispielsweise das Ablaufen von Deadlines, das Eintreffen von Nachrichten oder des Auftreten von Fehlern. Diese Ereignisse lassen sich abfangen, so dass innerhalb des Prozessablaufes auf sie in angemessener Form reagiert werden kann.

Die Verbindungselemente, die *Connecting Objects*, verknüpfen diese Hauptelemente. Insbesondere definieren die *Sequence Flows* den eigentlichen Ablauf des Prozesses, während *Message Flows* der Modellierung des Nachrichtenaustausches zwischen Tasks verschiedener Prozessteile dienen.

Abbildung 2.2 zeigt beispielhaft einen in BPMN modellierten Prozess. Für eine ausführlichere Darstellung verweisen wir auf die gut lesbare Spezifikation. [OMG08]

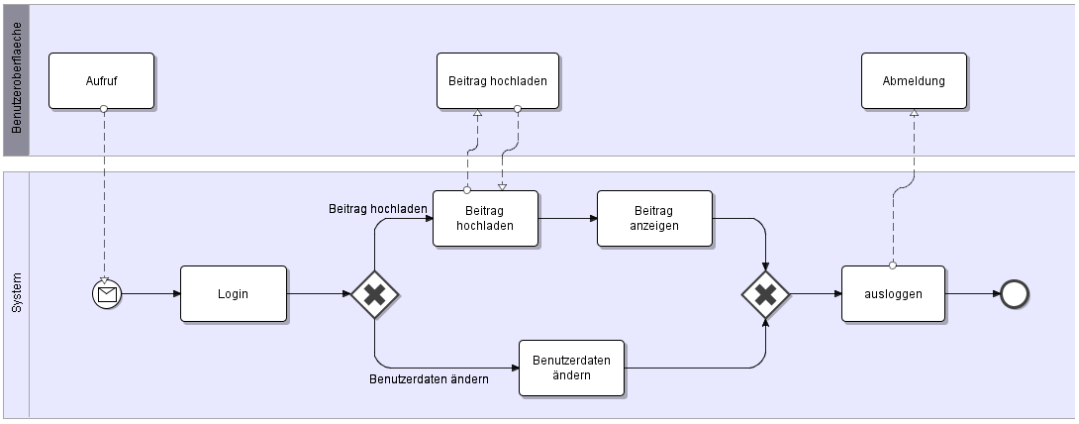


Abbildung 2.2: Prozessmodellierung mit BPMN

2.3.2 BPEL – Business Process Execution Language

Im Folgenden stellen wir die derzeit am weitesten verbreitete nicht-proprietäre Sprache zur Prozessdefinition vor, die Business Process Execution Language (BPEL).

Überblick

BPEL ist eine XML-basierte Sprache zur Definition von Workflows, also der ausführbaren Teile eines Geschäftsprozesses. BPEL beschreibt insbesondere den Prozess*ablauf*. Die eigentlichen Aktivitäten werden als Webservices eingebunden, die WSDL-Schnittstellendefinitionen hierfür sind ebenfalls Bestandteil einer Prozessdefinition. BPEL-Code wird normalerweise nicht manuell geschrieben, sondern von einem grafischen Modellierungswerkzeug automatisch erzeugt. Die erzeugte BPEL-Datei kann dann der Workflow-Engine zur Ausführung übergeben werden.
Die BPEL-Spezifikation [OAS07] entstand ursprünglich aus den Sprachen *XLANG* von Microsoft und *WSFL* von IBM. Sie enthält deshalb sowohl kalkülbasierte wie auch graphbasierte Aspekte. Der Standard wird von der OASIS betreut. Seine aktuelle Versionsnummer ist 2.0, die offizielle Bezeichnung lautet – in Anlehnung an andere Webservice-Standards – *WS-BPEL*.

Prozessdefinition

BPEL enthält eine Vielzahl möglicher Prozesselemente. Für die Definition des eigentlichen Prozessablaufes unterscheidet die Spezifikation zwischen einfachen und strukturierten Aktivitäten (*Basic Activities* bzw. *Structured Activities*). Unter die einfachen Aktivitäten fallen insbesondere die Aktivitäten zur Kommunikation mit den Webservice-Partnern, daneben gibt es aber auch Aktivitäten zur Datenzuweisung, zur Fehlerbehandlung oder zum Beenden des Prozesses. Structured Activities dienen der Strukturierung des Prozessablaufes. BPEL lässt sowohl die parallele wie die sequentielle Ausführung von Aktivitäten zu, es stehen mehrere Schleifentypen zur Verfügung, ebenso wie ein *If*-Element zur konditionalen Verzweigung und ein Element zur eventabhängigen Aktivitätsausführung.

Wir geben eine kurzgefasste Übersicht über die für uns relevanten Aktivitätstypen.

Basic Activities:

<*assign*> – Dient der Zuweisung von Variablenwerten.

<*empty*> – Leere Aktivität, tut nichts.
<*exit*> – Sofortige Beendigung der Prozessinstanz.
<*invoke*> – Ruft einen Webservice auf.
<*receive*> – Empfängt eine Nachricht eines Webservices.
<*reply*> – Sendet eine Antwortnachricht an einen Webservice.
<*throw*> – Signalisiert einen prozessinternen Fehler.
<*validate*> – Validiert den Wert einer Variable anhand der ihr zugeordneten Datendefinition.
<*wait*> – Verzögert die Prozessausführung um eine gewisse Zeit oder bis zu einem bestimmten Zeitpunkt.

Structured Activities:

<*flow*> – Beinhaltete Aktivitäten werden nebenläufig ausgeführt. Zusätzliche Synchronisationsmechanismen sind über die Definition von *Links* möglich.
<*forEach*> – Die Scope-Aktivität innerhalb des Elementes wird mehrfach ausgeführt. Die Ausführung kann wahlweise parallel oder sequentiell erfolgen.
<*if*> – Ermöglicht konditionale Verzweigungen durch die Verknüpfung von Bedingungen an Zielaktivitäten.
<*pick*> – Ordnet mehreren Events jeweils eine Aktivität zu. Die Aktivität, deren Ereignis eingetreten ist, wird ausgeführt.
<*repeatUntil*> – Die umschlossene Aktivität wird solange wiederholt ausgeführt, bis die angegebene Bedingung als wahr evaluiert wird.
<*scope*> – Definiert einen Kontext mit eigenen Variablen, Partner-Links, Event-Handlern usw. Scopes lassen sich hierarchisch schachteln.
<*sequence*> – Die aufgeführten Aktivitäten innerhalb eines Sequence-Elements werden sequentiell ausgeführt.
<*while*> – Die umschlossene Aktivität wird wiederholt ausgeführt, solange die angeführte Bedingung wahr ist.

Während die Ablaufbeschreibung eines BPEL-Prozesses normalerweise durch die Schachtelung von strukturierten Aktivitäten erfolgt, sieht die Spezifikation innerhalb des Flow-Elementes freie Verlinkungen vor. Dieser Punkt macht BPEL zu einem nicht blockstrukturierten Prozessmodell. Für unser Adaptionsmodell müssen wir hier an späterer Stelle Einschränkungen der BPEL-Spezifikation vornehmen, um auf der Prozessdefinitionsebene auf ein vollständig blockstrukturiertes Prozessmodell zurückgreifen zu können.

Als funktionale Schnittstellen sieht BPEL nur die Einbindung von Webservices vor. Es existiert aber ein Standardisierungsvorschlag, die Sprache dahingehend zu erweitern, dass auch menschliche Benutzer direkt in die Prozessdefinition integriert werden können. [AAD+07]

Die eigentlichen Prozesselemente, die einfachen wie auch die strukturierten, werden in BPEL alle als „Aktivitäten“ bezeichnet. In BPMN existieren ebenfalls „Aktivitäten“, diese werden aber nochmals nach „Tasks“ und „Subprozessen“ unterschieden. Auf Modellierungsebene werden wir also im Folgenden vorzugsweise von „Tasks“ sprechen, wenn wir Aufgaben im Geschäftsprozess meinen, auf Prozessdefinitionsebene dementsprechend von „Aktivitäten“.

Das Code-Listing in Abbildung 2.3 zeigt exemplarisch die etwas vereinfachte BPEL-Prozessdefinition des im Kapitel *BPMN – Business Process Modelling Notation* vorgestellten Prozesses.

Wie im Codeausschnitt zu sehen ist, lassen sich in BPEL auch Prozess-Variablen anlegen. Diese sind notwendig für die interne Verarbeitung der ein- und ausgehenden Message-Daten, kommen aber z.B. auch als Entscheidungsgrundlage bei datenbasierten Verzweigungen zum Einsatz. Eine Datenflussdefinition findet aber nur implizit statt, nicht explizit, wie bei manchen anderen Prozessdefinitionsmodellen. BPEL bietet darüber hinaus Möglichkeiten zur Ereignis-,

```
<process xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ... >
  <partnerLinks>
    <partnerLink name="systemAndBenutzeroberflaeche" partnerLinkType= ... />
  </partnerLinks>
  <variables>
    <variable name="thisEventStartMessageRequest" messageType=.../>
    <variable name="beitrag_hochladenRequestMsg" messageType=.../>
    <variable name="beitrag_hochladenResponseMsg" messageType=.../>
    <variable name="abmeldungRequestMsg" messageType=.../>
    <variable name="decision" type="xs:string"/>
  </variables>
  <sequence>
    <receive name="EventStartMessage" partnerLink= ...>
    </receive>
    <empty name="Login"/>
    <if>
      <condition>"upload" = $decision</condition>
      <sequence>
        <invoke name="Beitrag hochladen" partnerLink= ...>
        </invoke>
        <empty name"Beitrag anzeigen"/>
      </sequence>
      <elseif>
        <condition>"userData" = $decision</condition>
        <sequence>
          <empty name="Benutzerdaten ändern"/>
        </sequence>
      </elseif>
    </if>
    <invoke name="ausloggen" partnerLink= ... >
    </invoke>
    <empty name="EventEndEmpty"/>
  </sequence>
</process>
```

Abbildung 2.3: Prozessdefinition in BPEL

Fehler- und Kompensationsbehandlung. Diese Aspekte werden in unserer Arbeit nicht explizit betrachtet, weshalb wir sie an dieser Stelle auch nicht weiter ausführen.

2.3.3 Übersetzung von BPMN nach BPEL

Geschäftsprozesse werden normalerweise mit einem grafischen Modellierungswerkzeug modelliert und dann in ein ausführbares Prozessmodell übersetzt. Idealerweise geschieht dies unter Verwendung der entsprechenden Standards, BPMN und BPEL. Die Übersetzung des grafischen BPMN-Prozessmodells nach BPEL ist bislang allerdings nicht exakt spezifiziert.

Das Hauptproblem liegt hierbei in der unterschiedlichen Ausdrucksmächtigkeit der beiden Prozess-Modelle. Insbesondere können auf BPMN-Ebene viele Details nicht spezifiziert werden, die für eine vollständige Prozessdefinition in BPEL notwendig sind. Die Ursache hierfür liegt im Ansatz, dass BPMN auch den am Geschäftsprozess beteiligten Benutzern und nicht nur den Workflow-Spezialisten als Notationsform dienen soll. Die Beschränkung auf das Wesentliche, auf die reine Ablaufmodellierung, ist hierbei prinzipiell erwünscht. Dies hat zur Folge, dass BPMN

insbesondere im Bereich der Datenbehandlung gegenüber BPEL erheblich eingeschränkt ist.[1]
In manchen Aspekten ist aber das BPMN-Prozessmodell ausdrucksmächtiger als BPEL. Die BPEL-Spezifikation sieht für die Ablaufdefinition eine feste Blockstrukturierung vor. Die einzige Ausnahme hierbei ist das Flow-Element. In BPMN lässt sich der Prozessablauf viel freier modellieren. So müssen weder Schleifen expliziert als solche ausgezeichnet werden, noch müssen Verzweigungen durch Gateways symmetrisch modelliert werden. Je nach Anwendungsfall kann auf das *Split*-Gateway, das *Join*-Gateway oder auch auf beide verzichtet werden.
Jan Recker und Jan Mendling gehen in [RM06] auf den „conceptual mismatch“ zwischen beiden Standards näher ein.

Trotzdem gibt es Bemühungen, die Übersetzung von BPMN nach BPEL zumindest in Bezug auf die wesentlichen Prozessdefinitions-Aspekte möglich zu machen. In der BPMN-Spezifikation findet sich ein Vorschlag zur Übersetzung von BPMN 1.1. nach BPEL 1.1. Sie trifft aber keine Aussage über eine mögliche Übersetzung nach BPEL 2.0, das sich von der Vorgängerversion in vielen Punkten unterscheidet. Neben zahlreichen syntaktischen Modifikationen und Erweiterungen z.B. beim Datenzugriff und bei der Fehlerbehandlung, erfolgten auch Änderungen im Bereich der für uns besonders interessanten Prozessablauf-Definition. So wurde die *Switch*-Aktivität durch eine *If*-Aktivität ersetzt, das *Termination*-Element wich der *Exit*-Aktivität, *forEach* und *repeatUntil* kamen als neue Aktivitäten hinzu.
In der Praxis implementiert jeder Hersteller von Prozessmodellierungswerkzeugen seinen BPEL-Übersetzer nach eigenen Übersetzungsrichtlinien. In den meisten Fällen dürften sich diese aber zumindest an der Übersetzung von BPMN 1.1 nach BPEL 1.1 orientieren.

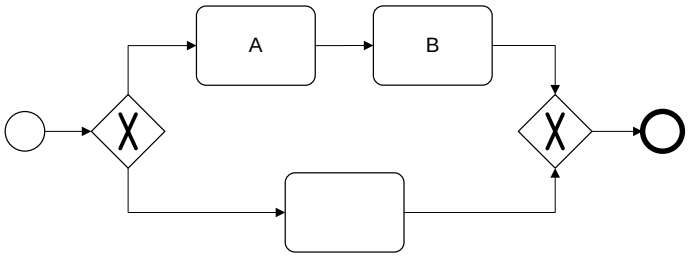
Wie wir später sehen werden, ist eine standardisierte Möglichkeit zur Übersetzung von BPMN 1.1 nach BPEL 2.0 für die Anwendbarkeit unseres Adaptionskonzeptes von elementarer Bedeutung. Wir gehen hier von Prozess-Änderungen auf BPMN-Ebene aus und müssen wissen, wie sich diese Änderungen bei der Übersetzung auf die BPEL-Ebene auswirken.
Allerdings betrachten wir in dieser Arbeit zunächst nur Veränderungen an der Ablaufdefinition, und dabei auch nur einfache Änderungen wie z.B. das Einfügen eines neuen Tasks. Dadurch sind viele Übersetzungsaspekte für uns nicht relevant, sie sind erst bei einer zukünftigen Erweiterung des Adaptionskonzeptes von Bedeutung. Wir können deshalb prinzipiell von einer Übersetzung ausgehen, wie sie in der BPMN-Spezifikation vorgeschlagen wird. Hierbei müssen wir natürlich die geänderten und die neu hinzugekommenen Prozesselemente mit einbeziehen. Für die von uns betrachteten Änderungsoperationen ist aber vielmehr das BPEL 2.0 Prozessmodell relevant, als die Übersetzung selbst. Deshalb werden wir auch keinen eigenen Vorschlag zur BPMN-BPEL-Übersetzung entwickeln, sondern beschränken uns im Folgenden auf die Betrachtung einzelner Transformations-Aspekte, die für unsere Arbeit von Bedeutung sind.

Bei der Übersetzung eines BPMN-Prozesses nach BPEL werden Tasks teilweise 1:1 in entsprechende BPEL-Aktivitäten übersetzt. Es gibt aber auch viele Fälle, bei denen bei der Übersetzung auf BPEL-Ebene entweder neue Aktivitäten dazukommen oder aber Elemente wegfallen, die auf BPMN-Ebene vorhanden sind. Abbildung 2.4 zeigt beide Fälle an einem Beispiel.

Zum einen wird für den oberen Ast der Verzweigung automatisch ein *sequence*-Element generiert, das die Tasks A und B umschließt. Dieses Element ist für die Festlegung einer sequentiellen Ausführung in BPEL notwendig. Auf BPMN-Ebene ist kein vergleichbares Element vorhanden, hier wird die Ausführungsreihenfolge über die Sequence Flow-Pfeile festgelegt.
Andere BPMN-Elemente sind auf BPEL-Ebene aufgrund der XML-Struktur nicht vonnöten und können bei der Übersetzung wegfallen. So werden im obigen Beispiel die beiden BPMN-Gateways zu einem einzigen BPEL-*If*-Element zusammengefasst.

Wir wollen an dieser Stelle etwas vorgreifen und zeigen, wie diese indirekte Übersetzung bei der

[1] In der Praxis behilft man sich in diesen Punkt oftmals durch die Einbindung externer Tools, die zumindest ein Stück weit eine XML-basierte Datendefinition und -behandlung ermöglichen. Diese kann dann zusammen mit dem BPMN-Modell nach BPEL übersetzt werden.



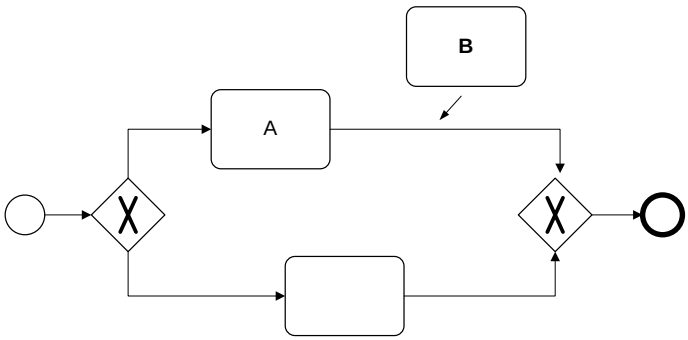
BPMN-Modellierung

```
<process>
  <if>
    <sequence>
      <Task A/>
      <Task B/>
    </sequence>
    <elseif>
      ...
    </elseif>
  </if>
</process>
```

BPEL-Code

Abbildung 2.4: BPMN-BPEL Übersetzung – Beispiel 1

Anwendung von einfachen Änderungsoperationen zu Schwierigkeiten führen kann. Betrachten wir noch einmal ein ähnliches Beispiel in Abb. 2.5. Tasks B war bisher noch nicht vorhanden und soll nun neu eingefügt werden.



BPMN-Modellierung

```
<process>
  <if>
    <sequence>
      <Task A/>
      <Task B/>
    </sequence>
    <elseif>
      ...
    </elseif>
  </if>
</process>
```

BPEL-Code

Abbildung 2.5: BPMN-BPEL Übersetzung – Beispiel 2

Das Einfügen von Task B ist auf BPMN-Ebene eine recht einfache Änderungsoperation. Auf die BPEL-Ebene hat diese allerdings umfangreichere Auswirkungen. Stand bisher Task A alleine innerhalb des oberen Verzweigungspfades, so muss jetzt um Task A und B zusätzlich ein *sequence*-Element gelegt werden, zum Ausdruck der sequentiellen Ausführungsreihenfolge.

Einen ebenso großen Änderungsumfang hätte das spätere Löschen von Task A oder B. Hier sollte das umgebende Sequence-Element bei der Übersetzung wieder entfernt werden, da nur noch ein Task im oberen Pfad vorhanden ist.

Hinsichtlich der Adaption bereitet dieser Umstand Probleme, da wir geänderte Prozesselemente normalerweise auf beiden Ebenen namentlich identifizieren müssen. Gibt es aber für manche BPMN- oder BPEL-Elemente gar keine Entsprechung auf der jeweils anderen Ebene, sind umfangreiche Funktionen zur Sicherstellung der Adaptierbarkeit notwendig. Wir werden im Verlauf dieser Arbeit hierauf noch genauer zu sprechen kommen.

2.4 Das FloeCom-Projekt

Nachdem wir uns bisher mit den allgemeinen Grundlagen des Bereiches der *Workflowsysteme* und mit den hierfür entwickelten Prozessdefinitions-Standards befasst haben, betrachten wir nun die konkreten Systeme, in die unsere Adaptionskomponente integriert werden soll. Wir beginnen mit dem *FloeCom*-System, das am Institut für Programmstrukturen und Datenorganisation (IPD) der Universität Karlsruhe entwickelt wird.

2.4.1 Projektziel und Konzept

Im Rahmen des FloeCom-Projekts wird an einem Workflow-Management-System gearbeitet, das Unterstützung für die Änderung von laufenden Workflow-Instanzen bieten soll. Hierbei werden die unterschiedlichsten Aspekte untersucht, von der Adaptionsunterstützung auf Benutzerebene über datenbasierte Änderungen bis hin zu Veränderungen des Prozessablaufs.

Der grundlegende Ansatz hierbei ist die Nutzung einer relationalen Datenbank zur Ablage der Prozessdefinition und des Prozesszustandes. Auf diese Datenbank greift die Workflow-Ausführungsmaschine zu. Gleichzeitig ist die Datenbank von anderen Komponenten aus zugreifbar. Auf dieser Grundlage lässt sich ein Adaptionskonzept entwickeln, in dem laufende Prozessinstanzen durch Datenmanipulationen innerhalb der zentralen Prozessdatenbank geändert werden können.
Im Rahmen des Projektes werden, soweit dies möglich ist, bereits vorhandene Standards verwendet. Dies betrifft insbesondere den Bereich der Prozessdefinition. Hierfür wird kein spezielles adaptionsgeeignetes Workflow-Modell entwickelt, sondern es greifen alle Arbeiten innerhalb des Forschungsprojektes auf die Industrie-Standards BPMN und BPEL zurück.

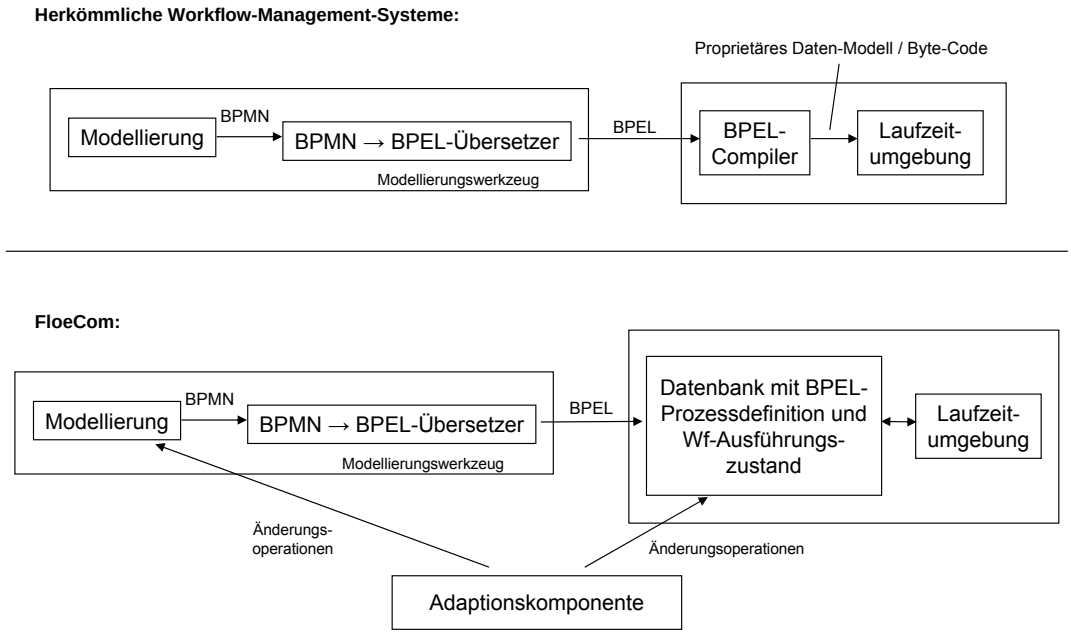


Abbildung 2.6: Datenbank-Ansatz des FloeCom-Systems

Abbildung 2.6 verdeutlicht nochmals den Ansatz, der für das FloeCom-System gewählt wurde: Bei herkömmlichen Workflow-Management-Systemen wird die Prozessdefinition nur in eine

Richtung weitergereicht. Der Prozess wird zunächst grafisch modelliert und dann nach BPEL übersetzt. Die erzeugte BPEL-Definitionsdatei wird dann nochmals in ein herstellerspezifisches Prozessdefinitionsformat umgewandelt, das von der Workflow-Engine ausgeführt werden kann. Die Prozesszustände werden innerhalb der Laufzeitumgebung in einem proprietären Format gespeichert. Hier existiert praktisch kein Ansatzpunkt für eine Änderbarkeit des Prozesses zur Laufzeit.
Im FloeCom-System wird dieser Ablauf in zwei Punkten aufgebrochen: Zum einen erfolgt die Ausführung der Workflows direkt auf Basis der BPEL-Prozessdefinition. Zum anderen werden sowohl die Prozessdefinition als auch die Instanzzustände zur Laufzeit in einer Datenbank vorgehalten, auf die auch andere Komponenten zugreifen können. Hierdurch ist es möglich, die Prozessdefinition wie auch die Instanzzustände von bereits gestarteten Prozessen zu verändern.

2.4.2 Systemarchitektur

Die Architektur des FloeCom-Systems besteht aus mehreren zusammenwirkenden Komponenten. Die zentralen Systembausteine sind hierbei die *FloeCom-Datenbasis* und die *Workflow-Ausführungsmaschine*. Abbildung 2.7 zeigt eine Übersicht über alle Systemkomponenten und die zwischen ihnen stattfindenden Datenflüsse.

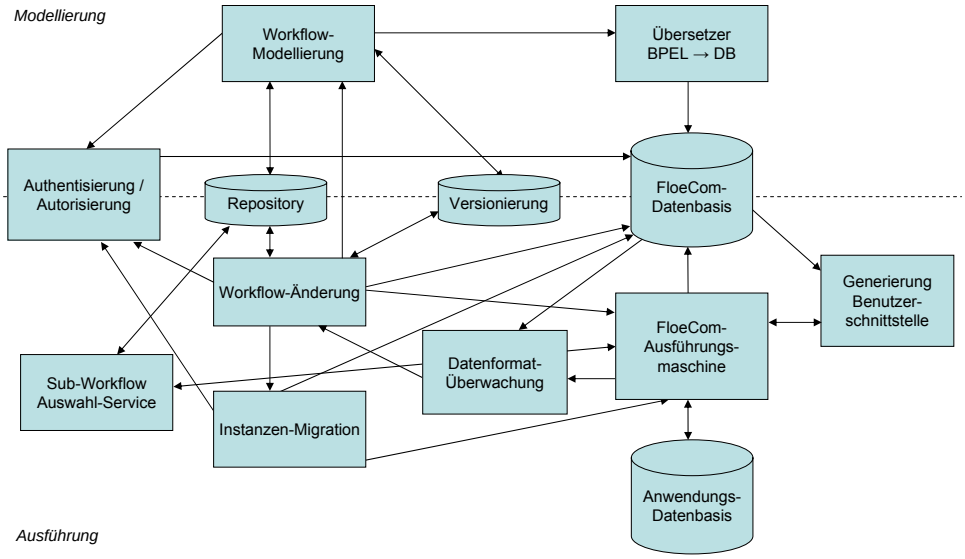


Abbildung 2.7: Architektur des FloeCom-Systems (aus [Wei08])

Der Prozess wird zunächst auf Benutzerebene modelliert. Hierbei kommt ein *Workflow-Modellierungs*-Werkzeug auf BPMN-Basis zum Einsatz. Dieses Tool exportiert den Prozess als BPEL-Datei, deren Inhalt wiederum von einem *Übersetzer* in das relationale Datenbankschema der *FloeCom Datenbasis* übersetzt wird. Auf dieser Datenbasis kann die *Ausführungsmaschine* den darin definierten Prozess steuern. Hierfür kann auch die Verwendung externer Programme notwendig sein, deren Einbindung in der *Anwendungs-Datenbasis* festgehalten wird.
Ein großer Teil der aufgeführten Komponenten dient der Durchführung bereits entwickelter Adaptionsfunktionen. Dominik Weingardt hat sich in seiner Diplomarbeit [Wei08] mit dynamischen Datenänderungen befasst. Er entwickelte ein Konzept, wie mit Datentypänderungen an einem Prozess-Schema bzw. innerhalb von Prozess-Instanzen umgegangen werden kann und integrierte die hierfür notwendigen Komponenten in das FloeCom-System. Die *Datenformat-Überwachung* übernimmt hierbei die Aufgabe, die Änderungen an den verwendeten Datenformaten zu er-

kennen und eine entsprechende Behandlung einzuleiten. Die Anpassung des Prozess-Schemas an die veränderten Datenformate nimmt die *Workflow-Änderungs*-Komponente unter direktem Zugriff auf die FloeCom-Datenbasis vor. Für die Migration der laufenden Instanzen des geänderten Prozessschemas ist die *Instanzen-Migration* zuständig. Um die Beziehungen zwischen den ursprünglichen Prozess-Schemata und den geänderten festhalten zu können, wird eine Versionierung durchgeführt. Die hierfür notwendigen Informationen werden in einer *Versionierungs*-Datenbank gespeichert.
Dominik Weingardt übernahm darüber hinaus den Ansatz zur Nutzung von Subprozessen, wie er im Worklet-Konzept vorgestellt wird. Dieser Ansatz unterstützt geplante Änderungen am Workflow, indem kritische Prozessabschnitte in einen Subworkflow ausgelagert werden, der in mehreren Varianten vorgehalten wird. Im FloeCom-System übernimmt der *Subworkflow-Auswahl-Service* die Auswahl des geeigneten Subprozesses aus allen im *Repository* abgelegten Varianten.
Die *Authentisierungs*-Komponente führt die Überprüfung der Benutzerrechte für den Zugriff auf die einzelnen Prozess-Schemata bzw. deren laufende Instanzen aus.
Zur Unterstützung der Adaptionsfunktionen wie auch der herkömmlichen Administrationsaufgaben ist eine geeignete *Benutzerschnittstelle* notwendig. Bei der Untersuchung dieses Aspektes ist insbesondere die Frage interessant, inwieweit der Benutzer je nach technischen Kenntnissen an dieser Stelle bei der Administration von möglicherweise sehr komplexen Workflows unterstützt werden kann bzw. welche Form der Benutzerunterstützung notwendig ist, damit ein bestimmtes Adaptionskonzept überhaupt sinnvoll anwendbar ist.

Nicht alle der erwähnten Systembausteine sind bislang vollständig implementiert. Die relationale Datenbank und der zugehörige Übersetzer wurden von Dominik Weingardt bereits in seiner Studienarbeit fertiggestellt [Wei06]. Elena Tentyukova entwickelte und implementierte in ihrer Abschlussarbeit eine Basisversion der zentralen BPEL-Ausführungsmaschine [Ten08]. Die bisher vorhandenen, oben vorgestellten Adaptionskomponenten wurden ebenfalls fertiggestellt, mit Ausnahme des *Subworkflow-Auswahl-Service*. Die restlichen Komponenten wurden bisher entweder noch gar nicht entwickelt oder aber liegen nur in konzeptioneller Form vor.

2.5 Intalio|BPMS

Das Business-Process-Management-System der Firma Intalio[2], *Intalio|BPMS* genannt, ist eine umfassende Softwarelösung zur Modellierung, Steuerung und Integration von Geschäftsprozessen. Das System bietet ein – für ein Open Source-Produkt in diesem Bereich – sehr hohes Maß an Funktionalität und verfügt über eine komfortable Benutzeroberfläche.
Da am IPD in Rahmen eines anderen Forschungsprojektes bereits eine Zusammenarbeit mit der Firma Intalio besteht, lag es nahe, die Einbindung dieser mächtigen Open Source-Lösung auch in andere Projekte in Erwägung zu ziehen. Im Projekt FloeCom fehlt bisher als elementarer Baustein ein Prozessmodellierungs-Werkzeug. Mittelfristig ist hier die Integration der BPMN-Modellierungskomponente geplant, auf die auch Intalio für ihr BPM-System zurückgreift.

2.5.1 Produkt und Komponenten

Das Intalio-BPMS besteht im Wesentlichen aus drei Hauptkomponenten. Mit dem *Designer* lassen sich BPMN-Prozesse modellieren, die BPEL-Engine (*Server*) übernimmt die Ausführung und Steuerung der Prozesse. Schließlich sorgt die *Workflow*-Komponente für die Integration des menschlichen Benutzers. Drei große Open Source-Pakete bilden die Grundlage dieser Systemkomponenten. Der *BPMN Modeler* als Subprojekt der *Eclipse SOA Tools Platform (STP)* dient

[2] http://www.intalio.com

als Framework für das Modellierungstool, die Entwicklung der BPEL-Engine erfolgt im Rahmen des *Apache Ode*-Projektes. Schließlich dient das Framework *Intalio Tempo* als Basis für die Miteinbeziehung menschlicher Benutzer in den Prozess, gemäß der BPEL4People-Spezifikation.

Das BPM-System als Ganzes ist in mehreren Varianten erhältlich. Zum einen werden die drei Hauptkomponenten *Designer*, *Server* und *Workflow* als separate Open Source-Pakete veröffentlicht.[3]
In integrierter Form sind diese als *Community Edition* kostenlos zu beziehen.[4] Die Community Edition stellt eine integrierte Software-Lösung aus den drei Hauptkomponenten und einigen zusätzlichen Tools dar. Dieses Software-Paket lässt sich mit wenigen Mausklicks installieren und enthält schon die notwendigen Systemkomponenten wie eine SQL-Datenbank und den Anwendungsserver und ebenso die komplette Funktionalität, die für eine Workflow-Modellierung und -Ausführung notwendig sind. Hervorzuheben ist dabei, dass der vollständige Arbeitsfluss von der Modellierung des Prozesses inklusive der Datenmodelle über die grafische Entwicklung der Benutzeroberfläche bis zum Deployment auf die Prozess-Engine in ein Programm integriert wurde. Allerdings liegen viele der eingebundenen Tools, die eine vollständige Prozessentwicklung bzw. ein komfortables Arbeiten erst möglich machen, nur in proprietärer Form vor. Hiervon betroffen sind so elementare Bausteine wie beispielsweise der *Data Mapper* zur Einbindung von Daten oder der BPEL-Übersetzer, der die BPMN-Notation in eine BPEL-Definition transformiert.
Für den wirklichen Unternehmenseinsatz im größeren Umfeld ist die kostenpflichtige *Enterprise Edition* gedacht. Im Vergleich zur Community Edition sind für diese zusätzliche Systemkomponenten für die Integration in die IT-Architektur eines Unternehmens verfügbar. Hier reicht das Spektrum von zusätzlichen SOA- und BPM-Bausteinen wie einem Enterprise Service Bus oder einer Business Rule-Engine über alternative Datenbank-Schnittstellen bis zu Anbindungsmöglichkeiten an Unternehmenssoftware-Pakete von SAP oder IBM.

Abbildung 2.8 gibt noch einmal eine Übersicht über die drei wichtigsten Programm-Versionen und die darin enthaltenen Haupt-Komponenten.

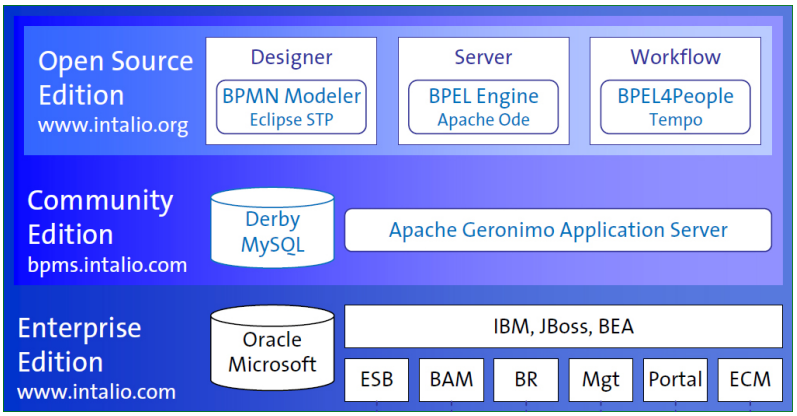


Abbildung 2.8: Intalio|BPMS-Versionen und enthaltene Komponenten (aus [Int07])

Seit kurzen sind zwei weitere Programmpakete des Business-Process-Management-Systems verfügbar: Die *Business-Edition* ist als Software as a Service verfügbar und ist so ohne aufwendige Integration in die Unternehmens-IT-Landschaft einsetzbar. Bei der Lizenzierung der *Developer Edition* erhält der Kunde einen erweiterten Support und beispielsweise auch Zugriff auf aktuelle Beta-Versionen.

[3] http://www.intalio.org
[4] http://bpms.intalio.com

2.5.2 Architektur

Für unsere Arbeit stehen zwei Komponenten im Fokus des Interesses: Der BPMN-Modeler und die BPEL-Engine.
Der BPMN-Modeler ist ein BPMN-Prozess-Modellierungswerkzeug auf Basis des *Eclipse Graphical Modeling Frameworks (GMF)* und des *Eclipse Modeling Frameworks (EMF)*. Mit diesem Tool lassen sich über eine komfortable Benutzeroberfläche BPMN-Prozesse modellieren. Für eine vollständige Workflow-Modellierung, die über die reine Ablaufsicht hinausgeht, wird diese Open Source-Komponente in der Community-Edition um mehrere Plugins erweitert: Über einen Data-Mapper lassen sich Daten einbinden und mit den Prozessmodellen verknüpfen. Ein WSDL-Editor ermöglicht die Definition von Webservice-Schnittstellen auf grafischer Ebene. Hier kommt der *Eclipse WSDL Editor* als Teil der *Eclipse Web Tools Platform (WTP)*[5] zum Einsatz. Mit dem *Forms-Designer* lassen sich XForms erstellen, die für die Prozessausführung notwendige Benutzereingaben entgegennehmen. Nachdem der BPMN-Prozess vollständig modelliert wurde, übernimmt ein *Compiler* die Übersetzung nach BPEL. Der *Deployment-Manager* stellt alle für die Prozessausführung notwendigen Dateien, wie die BPEL-Datei, die WSDL-Definition und die XForms zusammen und übernimmt das Deployment auf die Prozess-Engine (Ode) bzw. auf die Workflow-Komponente (Tempo).
Im Gegensatz zum eigentlichen BPMN-Modeler werden diese Zusatz-Komponenten größtenteils nicht als Open Source veröffentlicht. Unsere Arbeit ist hierdurch insbesondere im Fall des Data-Mappers, des Compilers und Deployment-Managers betroffen.

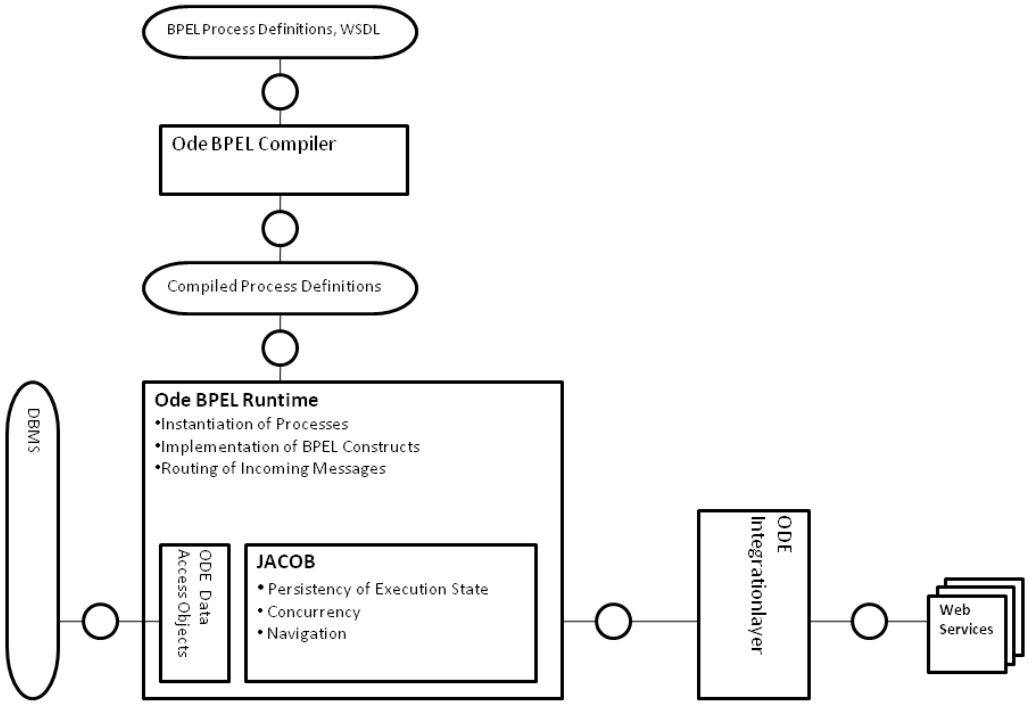


Abbildung 2.9: Ode – Architektur (aus [Apaa])

Die Prozessausführung übernimmt die BPEL-Engine *Ode* (*Orchestration Director Engine*). Sie ist vollständig als Open Source-Paket veröffentlicht und benötigt keine weiteren proprietären Zusatzkomponenten, um sinnvoll eingesetzt werden zu können. Abbildung 2.9 zeigt die System-Architektur in einer Übersicht.

[5] http://www.eclipse.org/webtools

Während das Prozessmodell in der oben vorgestellten Intalio-Engine direkt auf Basis der BPEL-Definition ausgeführt wird, benötigt die Ode-Ausführungsumgebung eine spezielle interne Repräsentation des Prozessmodells. Diese interne Prozessrepräsentation baut allerdings auf dem Objektmodell der BPEL-Spezifikation auf – Voraussetzung für die Integration in ein BPEL-basiertes Adaptionskonzept wie das unsere. Für die Übersetzung der BPEL-Datei in das für Ode ausführbare Datenformat sorgt der *Ode BPEL Compiler*.[6]
Innerhalb der Ode Laufzeitumgebung existiert ein separates Modul, das sich um die eigentliche Ablaufsteuerung der Prozess-Instanzen kümmert, das *Java Concurrent Objects Framework (Jacob)*. Das wesentliche Merkmal von Jacob ist die Ermöglichung von Nebenläufigkeit auf Anwendungsebene, also ohne die Verwendung von Threads. [Apab] bietet eine Einführung in das Framework.
Zur Gewährleistung der Persistenz müssen verschiedene Arten von Daten in einer Datenbank gespeichert werden. Im Wesentlichen sind dies die aktiven Instanzen, Daten zum Message-Routing, die Werte der Prozessvariablen und Partner-Links der einzelnen Instanzen und natürlich der jeweilige Ausführungszustand. Der Zugriff erfolgt über Data Access Objects, die den Zugriff auf die konkrete Datenquelle kapseln.
Die *ODE Integration Layers* ermöglichen die Kommunikation mit anderen Systemen. Hierfür stellen sie Schnittstellen bereit, beispielsweise für die message-basierte Kommunikation mit externen Webservices. [Apaa]

2.5.3 Umsetzung des BPMN- und des BPEL-Standards

Für die spätere Implementierung ist es von Bedeutung, ob die Tools, die wir verwenden wollen, auch den Standards entsprechen, auf denen unser Adaptionskonzept basiert.

Über den BPMN-Modeler trifft Intalio die Aussage, er unterstütze „die gesamte BPMN 2.0 Spezifikation“[7]. Da diese zum Zeitpunkt der Fertigstellung dieser Arbeit allerdings noch gar nicht verabschiedet ist und unser Adaptionskonzept auf der BPMN Version 1.1. beruht, werfen wir einen kurzen Blick auf die Unterstützung dieser früheren Fassung durch das Modellierungstool. Alle Aspekte in Zusammenhang mit den in dieser Arbeit betrachteten Änderungsoperationen sind im BPMN-Modeler ausreichend abgedeckt. Was aber auffällt ist, dass keine Möglichkeit besteht, gezielt auf die in der Spezifikation beschriebenen Objekt-Attribute zuzugreifen. Eine Verwendung dieser Attribute erfolgt allenfalls systemintern.
Deutliche Einschränkungen weist der Modeler hinsichtlich der Verwendung von Subprozessen auf. Die BPMN-Spezifikation kennt neben der direkten Einbettung eines Subprozesses auch die Möglichkeit der Referenzierung eines Subprozesses des gleichen oder eines anderen Diagrammes. Der BPMN-Modeler unterstützt Subprozesse allerdings nur in eingebetteter Form.

Bei der Erzeugung des BPEL-Codes macht der Intalio-Compiler von der Möglichkeit Gebrauch, die BPEL-Spezifikation um eigene Aktivitätsattribute zu erweitern. So werden für einfache Aktivitäten die Attribute *bpmn:label* und *bpmn:id* angelegt. Hierdurch wird eine Zuordnung der BPEL-Elemente zu den entsprechenden BPMN-Tasks ermöglicht.
Diese Erweiterung von Intalio zeigt zwei Dinge auf: Zum einen wurde auch hier die Notwendigkeit nach einer Möglichkeit zur Zuordnung der Elemente auf BPMN- und BPEL-Ebene erkannt. Zum Einsatz kommt diese Zuordnung beispielsweise innerhalb der *Ode Management Console*, wo sich der Benutzer anhand eines BPMN-Diagrammes den aktuellen Ausführungszustand einer Instanz anzeigen lassen kann. Zum anderen werden anscheinend nur die einfachen Aktivitäten mit den genannten Attributen versehen. Bei strukturierten Aktivitäten steht dem, wie bereits angesprochen, die teilweise fehlende bzw. uneindeutige Entsprechung auf BPMN-Ebene entgegen.

[6] Dieser ist nicht zu verwechseln mit dem Compiler des BPMN-Modelers, der die BPMN-BPEL-Übersetzung übernimmt.
[7] http://www.intalio.com/products/designer/#bpmn

Auf die Verwendung des optionalen *name*-Attributes, über das auch eine Referenzierung des entsprechenden BPMN-Elementes möglich wäre, verzichtet Intalio. Wie wir noch sehen werden, führt dies, zusammen mit der Nichtbenennung der strukturierten Aktivitäten, zu Schwierigkeiten bei unserer Implementierung.

Die Prozess-Engine selbst bietet eine umfangreiche Unterstützung der BPEL-Spezifikation. Die Abweichungen hiervon sind unter [Apac] gut dokumentiert.

Kapitel 3

Anforderungen

In diesem Kapitel erarbeiten wir die fachlichen Anforderungen auf höhere Ebene, die an unser Adaptions*konzept* gestellt werden. Auf die detaillierten funktionalen sowie technischen Anforderungen an unsere Adaptions*komponente* gehen wir im Kapitel 6 ein.

3.1 Allgemeine Zielsetzung

Die Zielvorgabe für diese Arbeit ist, ein Konzept für die Umsetzung von Workflow-Änderungen zur Laufzeit zu entwickeln. Unser Konzept soll möglichst allgemein gehalten werden, um als Basis für verschieden Arten von Änderungsoperationen dienen zu können. Exemplarisch führen wir in dieser Arbeit das Vorgehen bei einfachen Änderungen am Prozessablauf aus.
Als Grundlage für unser Vorgehen verwenden wir die Workflow-Standards BPMN (Modellierungsebene) und BPEL (Prozessdefinitionsebene). Unser Konzept soll einen Weg zur Umsetzung einer Adaption aufzeigen, der alle Schichten umfasst, die von Änderungen am Prozess betroffen sind. Dies beinhaltet die Ebene der grafischen Prozessmodellierung, die Prozessdefinitionsebene und die Ausführungsschicht. Wir möchten ein Vorgehen entwickeln, wie Prozessschemata oder einzelne Prozessinstanzen benutzerfreundlich auf grafischer Ebene geändert werden können. Hierbei ist die direkte Einflussnahme auf die laufenden Prozessinstanzen der entscheidende Punkt. In bisherigen Workflow-Management-Systemen wirken sich Änderungen an der Prozessdefinition nur auf neu zu startende Instanzen aus, nicht aber auf bereits in Ausführung befindliche.

3.2 Die Anforderungen im Einzelnen

Im Folgenden zeigen wir die Anforderungen auf, die an das von uns zu entwickelnde Konzept gestellt werden.

A – Berücksichtigung der entsprechenden Industrie-Standards

Unser Adaptionskonzept basiert auf den anerkannten Prozessdefinitions-Spezifikationen BPMN 1.1 und BPEL 2.0. Noch zu untersuchen ist, ob diese eine ausreichende Adaptionsunterstützung bieten können oder ob die Spezifikationen für eine sinnvolle Adaptionslösung erweitert oder eingeschränkt werden müssen.

B – Durchgängiger Adaptionspfad

Das Adaptionskonzept soll einen durchgängigen Adaptionspfad aufzeigen. Das bedeutet, dass der typische Weg einer Prozessentwicklung berücksichtigt und adaptiv unterstützt werden soll. Wir gehen davon aus, dass ein Prozess mit einem BPMN-Modellierungstool modelliert und dann – im Normalfall automatisch – nach BPEL übersetzt wird. Die Prozessdefinition im BPEL-Format wird von der Prozess-Engine eingelesen, die nun auf Basis dieser Prozessbeschreibung Instanzen starten kann. Diese führt die Engine im Folgenden aus und übernimmt unter anderem die Zustandshaltung für die laufenden Instanzen.
Für unsere Adaptionslösung ergibt sich hieraus beispielsweise, dass benutzerseitig initiierte Änderungen am Prozess auf BPMN-Ebene erfolgen sollen, da hier die Prozessmodellierung erfolgt.

C – Benutzerunterstützung

Bestandteil unserer Untersuchung ist auch die Frage, wie eine sinnvolle Benutzerunterstützung für Prozessanpassungen aussehen kann. Die Notwendigkeit einer Adaptionsunterstützung auch auf dieser Ebene lässt sich direkt aus der vorangehend genannten Forderung nach einem durchgängigen Adaptionspfad ableiten.
Die Erarbeitung einer ergonomischen Benutzeroberfläche ist allerdings nicht Bestandteil dieser Arbeit. Vielmehr geht es darum, Vorschläge rein konzeptioneller Natur zu unterbreiten, an welchen Stellen und auf welche Art und Weise der menschliche Benutzer in die Durchführung einer Adaption zu integrieren ist.

D – Korrektheits- und Konsistenzerhaltung

Grundlegende Randbedingung bei der Durchführung von Prozess-Änderungen ist der Erhalt von Korrektheit und Konsistenz sowohl hinsichtlich der Prozessdefinition auf allen Ebenen, als auch in Bezug auf die Zustandshaltung der laufenden Prozesse:

– Die Korrektheit der BPMN- wie auch der BPEL-Prozessmodelle muss erhalten bleiben.

– Die Ausführungszustände der Instanzen- bzw. der einzelnen Aktivitäten müssen sich nach der Adaption wieder in einem konsistenten Zustand befinden. Hierauf werden wir in den Kapiteln 5.4.2 und 5.4.3 näher eingehen.

– Die Datenflusskorrektheit auf Instanzebene sollte durch den Adaptionsvorgang nicht beeinträchtigt werden. Der Erhalt bzw. die Analyse der Datenfluss-Korrektheit sind wesentliche Bestandteile unserer Konzeption und werden in Kapitel 5.4.4 ausführlich behandelt.

E – Erweiterbarkeit des Adaptionskonzeptes

Unser Adaptionskonzept soll grundsätzlich beliebige Änderungen am Prozessmodell unterstützen. Konkret betrachtet und exemplarisch umgesetzt werden in dieser Arbeit zunächst nur einfache strukturelle Änderungen am Prozessablauf. Das Konzept ist aber hinsichtlich aller relevanter Adaptions-Aspekte möglichst offen zu konzipieren, sodass spätere Erweiterungen um andere Änderungsoperationen vorgenommen werden können.

F – Implementierung als unabhängige Komponente

Das entwickelte Konzept soll anschließend als unabhängige Adaptions-Komponente implementiert werden, die prinzipiell an beliebige Process-Management-Systeme angebunden werden kann. Im Fokus steht aber die Integrierbarkeit in das institutseigene Workflow-Management-System FloeCom und das BPM-System Intalio|BPMS.

3.3 Beispiel-Szenario

Zur späteren Validierung unseres Konzeptes betrachten wir zwei Prozesse, an denen wir jeweils eine Änderung vornehmen. Diese beiden Prozesse möchten wir bereits an dieser Stelle vorstellen. Sie vermitteln einen ersten Eindruck eines möglichen Anwendungsszenarios für unser Adaptionskonzept.

3.3.1 Anwendungsfall: ProceedingsBuilder

Das FloeCom-Projekt hat seinen Ursprung in der vorangehenden Entwicklung eines webbasierten Content-Management-Systems zur Unterstützung des Proceedings Chairs einer wissenschaftlichen Konferenz. Dieses System, der sogenannte *ProceedingsBuilder*, hilft dem Vorsitzenden beim Zusammentragen und Publizieren der Konferenzbeiträge. Hierfür erhalten die Konferenzteilnehmer einen Account, über den sie ihre Kontaktdaten verwalten und ihre Beiträge hochladen können. Im Rahmen der vollständigen Funktionalität des Systems ist eine Vielzahl von Abläufen hinterlegt. Kommen die Teilnehmer beispielsweise mit der Abgabe in Verzug, werden sie automatisch an die einzuhaltenden Fristen erinnert. Insofern weist der ProceedingsBuilder zahlreiche Merkmale eines Workflow-Management-Systems auf.
Beim praktischen Einsatz dieses System zeigte sich allerdings an vielen Stellen der Bedarf nach einer flexiblen Anpassbarkeit des Systems zur Laufzeit. Die gewünschten Möglichkeiten reichen hierbei von Anpassungen der internen Prozessabläufe bis hin zu Änderungen an der Datenstruktur. Die diesbezüglichen Anforderungen an ein solches System wurden in einem Paper vorgestellt [MBRS].

3.3.2 Die Beispiel-Prozesse

Die Prozesse, die innerhalb des ProceedingsBuilder-Systems vorhanden sind, wurden vor kurzem in einer anderen Arbeit am IPD in BPMN-Darstellung modelliert. Wir hätten gerne auf diese Modellierungen zurückgegriffen, die wirklichen ProceedingsBuilder-Prozesse stellten sich aber für unsere Zwecke als ungeeignet heraus. Wir haben deshalb das Szenario beibehalten und zeigen im Folgenden zwei Prozesse hieraus, wie sie in ähnlicher Form auch in anderen Anwendungen zu finden sind. Diese Prozesse wurden im Vergleich zu den realen Abläufen teilweise vereinfacht und für unser Validierungsvorhaben angepasst. Dies betrifft insbesondere auch das Datenmodell.

Wir stellen die beiden Prozesse hier aus didaktischen Gründen in BPMN-Darstellung vor und zeigen auf dieser Ebene die kritischen Punkte hinsichtlich der Prozessänderungen auf. Die BPEL-Übersetzung, auf der – wie wir später sehen werden – im Rahmen unseres Adaptionskonzeptes die eigentlich relevanten Adaptions-Analysen stattfinden, ist im Anhang B zu finden.

Hochladen eines Konferenzbeitrages – Einfügeoperation

Als ersten Beispielprozess ziehen wir einen möglichen Ablauf für das Hochladen eines Papers heran. Im ProceedingsBuilder ist dieser Prozess Teil eines größeren Prozesses, in dem noch weitere Operationen zu Auswahl stehen. Der Ablauf ist in Abbildung 3.1 dargestellt. Wir betrachten hiervon nur den Prozessteil, der auf Systemebene abläuft. Im Diagramm explizit als Datenelemente dargestellt sind nur diejenigen Daten, die für uns hinsichtlich der Adaptionsoperation von Interesse sind.

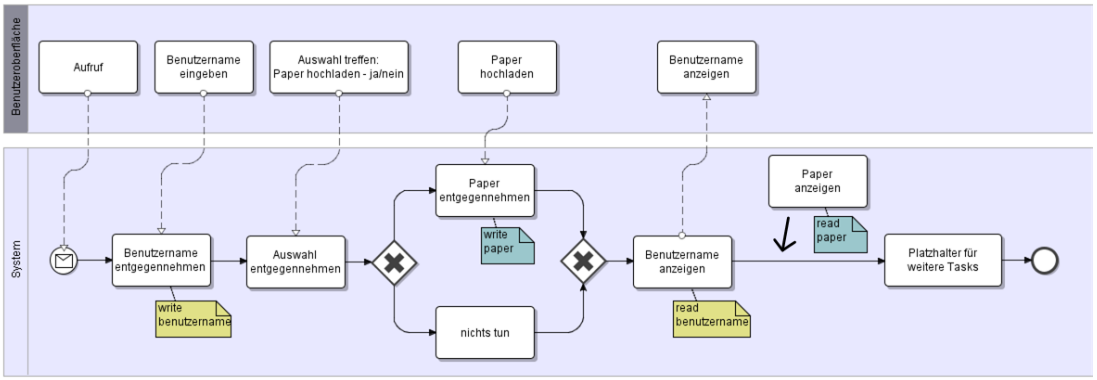


Abbildung 3.1: Einfügeoperation

Prozessbeschreibung

Der Prozessaufruf erfolgt aus der Benutzeroberfläche heraus. Zunächst wird der Benutzer aufgefordert seinen Benutzernamen einzugeben. Dieser wird als Prozessvariable im System gespeichert. Danach hat der Benutzer die Wahl, ob er ein Paper auf den Server hochladen möchte oder nicht. Entscheidet er sich für das Hochladen, wird dieser Task ausgeführt und das Paper ebenfalls in einer Variable gespeichert, ansonsten erfolgt keine Operation. Im weiteren Prozessverlauf (der hier nicht dargestellt ist), soll zukünftig immer der Benutzername innerhalb der Benutzermaske angezeigt werden, weshalb vor dem Senden des Benutzernamens an die Benutzeroberfläche ein Lesezugriff auf die Variable *benutzername* erfolgt.

Änderungsoperation

Das Prozess-Schema soll nun dahingehend geändert werden, dass nach dem Hochladen des Papers und dem Anzeigen des Benutzernamens auch das Paper angezeigt werden soll. Hierzu wird der neue Task *Paper anzeigen* eingefügt, in dem ein Lesezugriff auf die Variable *paper* erfolgt. Es ist direkt ersichtlich, dass das Einfügen dieser Leseaktivität zu einem Datenflussproblem führt. Während die Variable *benutzername* vor ihrem Lesen sicher geschrieben wird, ist dies für die Prozessvariable *paper* nicht garantiert. Erfolgt innerhalb der Verzweigung kein Hochladen eines Papers, ist die Variable *paper* bei Erreichen des Tasks *Paper anzeigen* noch nicht beschrieben worden und damit auch nicht initialisiert.

Registrierung eines Benutzers – Löschoperation

Der zweite Prozess, den wir betrachten, dient der Registrierung eines Benutzers. Der Registrierungsvorgang ist hier vereinfacht dargestellt, Abbildung 3.2 zeigt das BPMN-Modell. Relevant für unsere Betrachtung ist nur der mittlere Bereich, die *System*-Lane. Dieser stellt den eigentlichen ausführbaren Prozess dar.

Prozessbeschreibung

Von der Benutzeroberfläche aus, z.B. aus einem Web-Client heraus, erfolgt der Start des Registrierungsvorganges auf Systemebene. Als Daten werden Nachname und Vorname erfasst. Im dritten Task wird der Benutzer mit seinem Nachnamen angesprochen und gefragt, ob die eingegebenen Daten gespeichert werden sollen oder ob er den Vorgang abbrechen möchte. Je

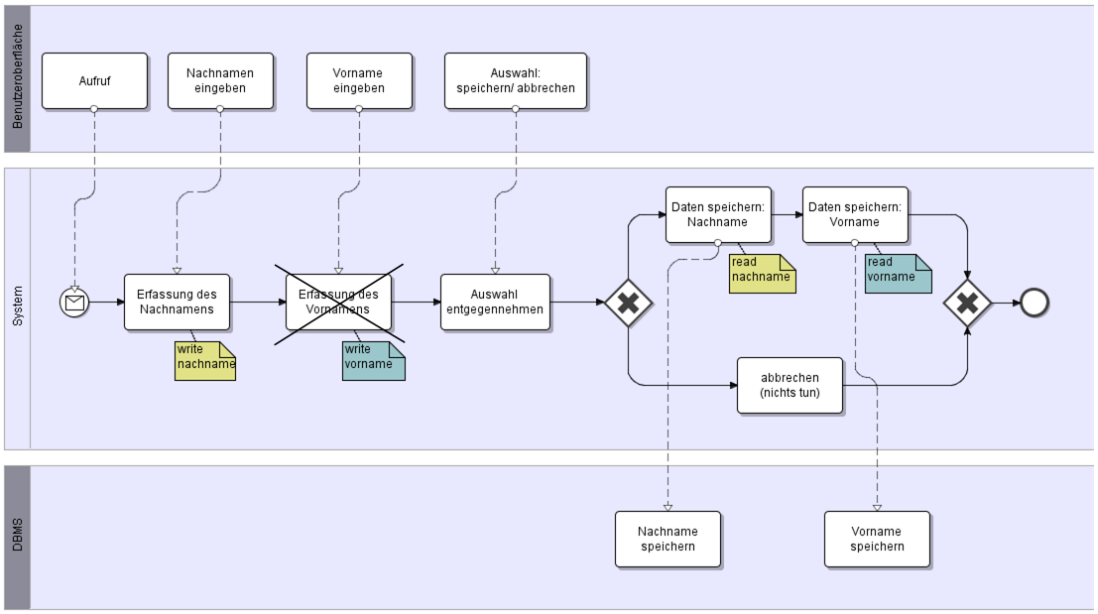


Abbildung 3.2: Löschoperation

nachdem wie der Wunsch des Benutzers aussieht, wird der entsprechende Ast der Verzweigung gewählt und die Daten werden nacheinander gespeichert oder der Vorgang wird ohne Speicherung fortgesetzt und kurz darauf beendet.

Änderungsoperation

Der Prozess soll nun dahingehend geändert werden, dass keine Erfassung des Vornamens mehr erfolgt. Der Task *Erfassung des Vornamens* wird hierfür gelöscht. Weitergehende Änderungen am Prozess erfolgen in unserem Szenario zunächst nicht.
Dies führt zu dem Problem, dass die Variable *vorname* innerhalb der Verzweigung eventuell gelesen werden muss, ohne dass sie zuvor geschrieben wurde. Dieses Problem muss im Rahmen unseres Adaptionskonzeptes in geeigneter Art und Weise behandelt werden.

Kapitel 4

Literaturanalyse

Im vorangehenden Kapitel haben wir die Anforderungen aufgezeigt, die an unser Adaptionskonzept gestellt werden. Wir führen nun eine Literaturanalyse durch um festzustellen, auf welche bereits gewonnenen wissenschaftlichen Erkenntnisse wir für die nachfolgende Konzeption zurückgreifen können. Hierbei betrachten wir das *ADEPT* Adaptions-Modell, das unter den uns bekannten Arbeiten zur Workflow-Adaption am ehesten Ansätze aufzeigt, die sich auch auf ein Adaptionskonzept auf BPMN/BPEL-Basis übertragen lassen. Des Weiteren untersuchen wir verschiedene Arbeiten zur Datenflussanalyse in BPEL, die für unsere Konzeption von Bedeutung sein können.

4.1 ADEPT

An der Universität Ulm wurde ein Workflow-Management-System entwickelt, das einer Vielzahl von Adaptivitäts-Anforderungen gerecht wird. Im Rahmen des ADEPT-Projektes kam es zu mehreren Veröffentlichungen. Wir nehmen die Dissertation von Manfred Reichert [Rei00] als Grundlage unserer Analyse und beziehen uns in unseren Ausführungen hierauf.

4.1.1 Überblick

Das ADEPT-System wurde mit dem Ziel entwickelt, Workflow-Flexibilität auf unterschiedlichen Ebenen zu ermöglichen. So sind sowohl Ad-hoc-Rückwärtssprünge im Prozessablauf möglich, als auch die Vormodellierung geplanter Abweichungen. Hauptbestandteil des Konzepts sind vordefinierte Änderungsoperationen auf Prozessschema-Ebene, die sich auch auf laufende Instanzen auswirken. Dies ist gleichzeitig der für uns interessanteste Aspekt.

Grundlage von ADEPT ist ein eigens hierfür entwickeltes Workflow-Metamodell, das *ADEPT-Basismodell*. Von seinem Ursprung her lässt es sich als graphorientiert bezeichnen, die Workflow-Definition selbst erfolgt aber mengenbasiert. Das Akronym *ADEPT* steht für *Application DEvelopment Based on Pre-Modeled Process Templates* und ist auf den Ansatz zurückzuführen, dass jeder Aktivität eine wiederverwendbare Aktivitätsvorlage zugeordnet wird, welche die durchzuführende Aufgabe spezifiziert. Abbildung 4.1 zeigt, wie die Definition eines Workflows, hier *Kontrollflussgraph* genannt, zu erfolgen hat. Das Metamodell ermöglicht sowohl die Modellierung des Kontrollflusses wie auch die Definition des Datenflusses.

Das ADEPT-Basismodell verwendet einen blockorientierten Ansatz zur Kontrollflussmodellierung. Kontrollstrukturen wie Verzweigungen oder Schleifen bilden hierbei einen Kontrollblock,

Definition 3-1 (Kontrollflußgraphen, KF-Graphen)

Ein ADEPT-Kontrollflußgraph (kurz: KF-Graph) ist ein Tupel

$$\boldsymbol{CFS = (N, E, D, NT, NP, V_{out}, V_{in}, DP, EC, Template)}$$

mit Knotenmenge N, Kantenmenge E und Datenelementen D sowie den Abbildungen:

NT: $N \mapsto$ *NodeTypes* (*Knotentypen*)

NP: $N \mapsto$ *Priority* (*Knotenprioritäten* der Aktivitätenknoten)

V_{out}: $N \mapsto$ *OutBehaviour* (*Ausgangssemantik* der Knoten)

V_{in}: $N \mapsto$ *InBehaviour* (*Eingangssemantik* der Knoten)

DP: $N \mapsto D \cup$ {UNDEFINED} (*Entscheidungsparameter* bei bedingten Verzweigungen)

EC: $E \mapsto$ *EdgeCode* $\cup$ {UNDEFINED} (*Auswahlcode* der Kanten bei bedingten Verzweig.)

Template: $N \mapsto$ *Templates* $\cup$ {NONE} (Zugeordnete *Aktivitätenvorlage* der Knoten)

Abbildung 4.1: ADEPT – Prozessdefinition

der über eindeutige Start- und Endaktivitäten verfügen muss. Diese Kontrollblöcke können geschachtelt sein, dürfen sich aber nicht überlappen. Hierdurch wird eine regelmäßige Schachtelung erreicht, die wesentliche Vorteile für eine Adaptionsunterstützung mit sich bringt.
In seiner strikten Blockstrukturierung unterscheidet sich das ADEPT-Metamodell von vielen anderen Workflow- bzw. Prozess-Metamodellen. Auch BPMN und BPEL sind nicht vollständig blockstrukturiert. In BPMN ist eine recht freie Modellierung des Prozessablaufes erlaubt, nicht explizit ausgezeichnete Schleifen sind ebenso möglich wie Verzweigungen ohne Split- oder Join-Gateway. BPEL kommt einem blockorientierten Ansatz schon sehr nahe, allerdings widerspricht diesem die Möglichkeit zur freien Synchronisationsmodellierung über sogenannte *Links* innerhalb der *flow*-Elemente.

Das Metamodell von ADEPT umfasst nur die Definition eines lokalen Prozesses. Die Kommunikation mit anderen Prozessen oder Prozessteilen, wie wir sie heutzutage auf Basis von Webservices kennen, fand zur Projektlaufzeit von ADEPT noch keine Beachtung.
Dafür umfasst das Basismodell aber nicht nur – wie in den meisten anderen Fällen – die Schemaebene, sondern auch die Instanzebene. Es ist explizit definiert, wie die Instanzzustände festgehalten werden. Dies wurde ebenfalls in einem mengenbasierten Ansatz umgesetzt.

4.1.2 Adaptivitätsmodell

Ziel unserer Arbeit ist es, ein Adaptivitätskonzept auf Basis der weitverbreiteten Industriestandards BPMN und BPEL zu entwickeln. Diese Prozess-Metamodelle bieten im Gegensatz zum ADEPT-Basismodell keine Unterstützung zur Modellierung planbarer Abweichungen im Kontrollfluss. Aus diesem Grund ist diese Funktionalität von ADEPT für uns nicht weiter interessant. Auch Ad-hoc-Rückwärtssprünge betrachten wir an dieser Stelle nicht weiter, da sie allenfalls einen Nebenaspekt innerhalb eines umfassenden Adaptionskonzeptes darstellen.

Das ADEPT-Modell bietet allerdings einen interessanten Ansatz, was die Definition von Änderungsoperationen auf Schema- und Instanzebene betrifft. Im $ADEPT_{flex}$-Kalkül wird ein vollständiger Satz von Änderungsoperationen definiert, der prinzipiell die Durchführung jeder beliebigen Änderung an Workflow-Schemata oder -Instanzen ermöglicht. Im Einzelfall ist der Erhalt der Korrektheit und der Konsistenz die Vorraussetzung für die Durchführung einer Änderung.

Das Adaptionsmodell ADEPT$_{flex}$ baut in einer dreistufigen Struktur auf dem Workflow-

Metamodell auf. Abbildung 4.2 zeigt eine Übersicht. Auf der untersten Ebene des Adaptionsmodells werden *Änderungsprimitive* definiert. Dies sind elementare Operationen, wie das Einfügen eines Knotens, einer Kante oder das Löschen derselben Elemente.

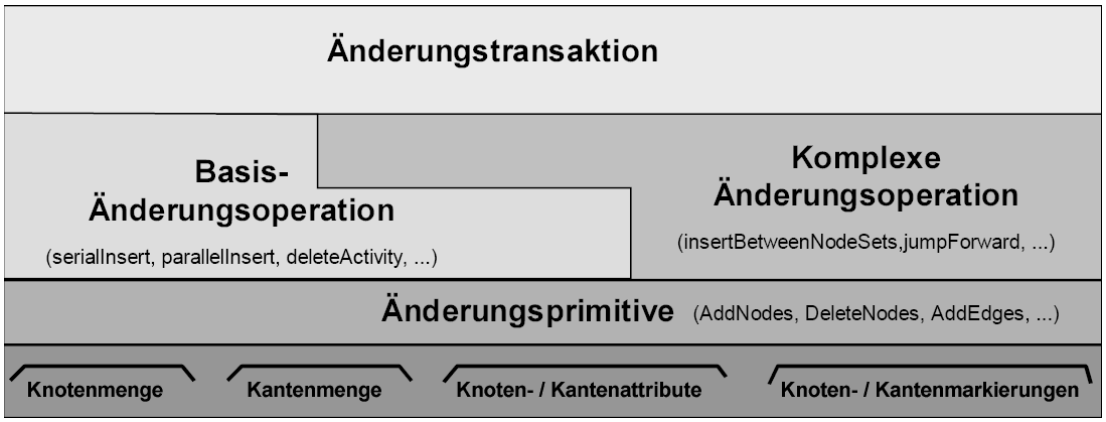


Abbildung 4.2: ADEPT$_{flex}$ – Operationshierarchie

Die mittlere Stufe besteht aus *Änderungsoperationen*. Dies sind Operationen, die für sich genommen schon sinnvoll angewendet werden können. Beispielsweise ist hier das Einfügen einer Aktivität in eine Kette von seriellen Aktivitäten definiert. Für die Umsetzung dieser Operationen wird auf die Änderungsprimitive zurückgegriffen. Die Definition einer Änderungsoperation umfasst aber nicht nur die einzelnen Schritte, die beispielsweise auf ein Prozess-Schema angewandt werden müssen, sondern auch die Kriterien, die zur Erhaltung der Korrektheit und der Konsistenz des Prozessschemas bzw. der zu ändernden Instanzen erfüllt sein müssen. So dürfen beispielsweise keine Operationen an einer Prozessinstanz durchgeführt werden, die Änderungen an dem bereits ausgeführten Prozessabschnitt vornimmt.

In der Praxis sind meistens nicht einzelne Änderungsoperationen notwendig, sondern Prozessänderungen, die aus mehreren Einzelschritten bestehen. Ein Beispiel hierfür wäre die Notwendigkeit, an einer bestimmten Stelle eine Verzweigung einzubauen. Im ADEPT-Modell würde diese Änderung eventuell darin bestehen, erst eine Verzweigung einzufügen, dann die notwendigen Tasks in den neuen Zweig einzusetzen, eventuell den anderen Zweig zu bearbeiten und letztendlich noch den Datenfluss anzupassen.
Solche komplexen Änderungsvorgänge werden in ADEPT als *Änderungstransaktionen* zusammengefasst. Es ist hierbei vorgesehen, dass innerhalb einer Transaktion ungültige Prozessmodelle entstehen und inkonsistente Ausführungszustände erreicht werden können. Entscheidend ist aber, dass nach Abschluss aller Operationen, die die Transaktion umfasst, wieder ein korrektes Prozess-Schema vorliegt und sich alle Instanzen in einem konsistenten Zustand befinden.

4.2 Vergleich von ADEPT, BPMN und BPEL

Nachdem wir im Grundlagenkapitel bereits die BPMN- und die BPEL-Spezifikation vorgestellt haben, möchten wir an dieser Stelle einen Vergleich zum ADEPT-Modell ziehen. Wir beschränken uns hierbei auf einige Aspekte, die für unser Adaptionsmodell von wesentlicher Bedeutung sind.
Die Autoren der ADEPT-Veröffentlichungen sprechen in diesen von ADEPT als einem *Workflow*-System, während BPMN und BPEL *Prozess*-Notationen bzw. -Sprachen spezifizieren. Wir passen uns dieser spezifischen Begriffsverwendung in den folgenden Ausführungen an.

Durchgängigkeit des Adaptionsmodells

Der Ansatz von ADEPT zeigt ein durchgängiges Adaptionskonzept auf, das die Schema-, die Instanz- und ein Stück weit auch die Benutzerebene umfasst. Als Grundlage und Voraussetzung für dieses Konzept dient ein eigens hierfür entwickeltes Workflow-Metamodell.
BPMN und BPEL hingegen sind anerkannte Standards im Bereich der Prozessdefinition. Möchte man auf ihrer Basis ein Adaptionskonzept entwickeln, wird die Durchgängigkeit dieses Ansatzes bereits bei der Übersetzung von BPMN nach BPEL stark beeinträchtigt. BPMN ist eine grafische Prozess-Repräsentation, BPEL eine textuelle Prozessdefinition. Eine Transformation von BPMN nach BPEL ist nicht standardisiert bzw. in Teilaspekten auch gar nicht möglich.
Wenn die Entwicklung eines Adaptionskonzeptes auf BPMN/BPEL-Basis möglich ist, dann stellt dies eine nachträglich aufgesetzte Lösung dar. Im Gegensatz zum ADEPT-Basismodell ist in BPMN und BPEL keinerlei Adaptionsunterstützung vorgesehen.

Validierung

Auch hinsichtlich der Validierung bereitet das homogene ADEPT-Modell weniger Probleme als der BPMN/BPEL-Ansatz. Für das mengenbasierte ADEPT-Modell wurden sowohl auf der Prozessdefinitionsebene als auch auf der Instanzebene Korrektheitskriterien definiert. Die jeweiligen Überprüfungen auf Einhaltung dieser Kriterien sind auch für die einzelnen Änderungsoperationen spezifiziert.
Die Business Process Modelling Notation ist ein rein grafisches Modell, mit Attributen hinterlegt. Korrektheitskriterien sind hierbei rein verbal beschrieben, aber nicht in einem formalen Modell spezifiziert. Für BPEL existiert ein formales Modell in Form eines XML-Schema-Dokumentes. Für die Ausführungsebene ist wiederum kein Metamodell definiert. Innerhalb der BPEL-Spezifikation wird teilweise in informeller Form auf eine sinnvolle Art der Implementierung eingegangen.
Wird eine Adaptionsoperation angewandt, so muss also auf drei Ebenen gegen drei verschiedene, teilweise gar nicht vorhandene Metamodelle validiert werden. Dies betrifft natürlich auch die eigentliche Entwicklung der Adaptionsoperationen. Auch hier ist eine dreistufige Validierung notwendig.

Zustandshaltung

Das ADEPT-Modell definiert auf Instanzebene Ausführungszustände für Knoten und Kanten. Die für uns relevanten Knotenzustände sind *NON-ACTIVATED, ACTIVATED, SELECTED, RUNNING, COMPLETED, SKIPPED* und *FAILED*. Ihre Bedeutung ergibt sich aus der Bezeichnung.
Überraschenderweise spezifiziert BPMN sowohl Prozess- wie auch Aktivitätszustände. Diese sind zur Verwendung innerhalb von *Assignment*-Ausdrücken vorgesehen. *None, Ready, Active, Cancelled, Aborting, Aborted, Completing* und *Completed* stehen zur Auswahl.
Die BPEL-Spezifikation selbst ist als Prozessdefinitionssprache nicht zustandsbehaftet. Sie unterbreitet auch keinen Vorschlag für die Verwendung von Prozess- oder Aktivitätszuständen innerhalb standardkonformer Implementierungen. Dies hat zur Folge, dass in der Praxis jedes Prozess-Management-System über eine proprietäre Zustandshaltung verfügt, was einen Datenaustausch erschwert.

Aktivitätsdefinition

Die eigentlichen durchzuführenden Aufgaben, die ein Geschäftsprozess enthält, werden in ADEPT als *Knoten*, in BPMN als *Tasks* und in BPEL als *Activities* bezeichnet.[1]
Wie zuvor bereits erläutert, wird einem ADEPT-Knoten eine Vorlage zugeordnet, die die genaue Implementierung der Aufgabe enthält. BPMN und BPEL kennen keine solchen Vorlagen. In BPMN wird ein Task über sein *TaskType*-Attribut einer Kategorie zugeordnet. Er kann entweder vom Typ *Service, Receive, Send, User, Script, Manual, Reference* oder *None* sein. Je nach Task-Typ müssen dann weitere Attribute mit Werten belegt werden. So sind bei einem Service-Task zum Beispiel Verweise auf die einkommende und auf die ausgehende Nachricht anzugeben.
Die BPEL-Spezifikation sieht keine direkte Benutzereinbindung vor, wie dies in BPMN über den Tasktyp *User* möglich ist. Zur Ausführung von Aufgaben muss in BPEL ein Webservice aufgerufen werden, der das entsprechende Vorgehen implementiert. Die eigentliche Bearbeitung wird also ausgelagert, es findet nur eine Kommunikation mit der ausführenden Instanz über Webservice-Nachrichten statt.

Subworkflows

Zur besseren Strukturierung von Workflows sieht ADEPT die Definition von Subworkflows als sogenannte *Blockaktivitäten* vor. Als vollständige Kontrollblöcke fügen sich Subworkflows nahtlos in das ADEPT-Konzept der regelmäßigen Blockstrukturierung ein.
BPMN bietet ebenfalls eine Unterstützung des Subprozess-Konzepts an. Hier lassen sich Subprozesse auf drei verschiedene Arten einbinden: als *embedded*, *reference* und als *reusable*. Ein eingebetteter Subprozess wird direkt innerhalb des Subprozess-Elementes als kleinerer Prozess modelliert. Bei *reference* erfolgt ein Verweis auf einen anderen Subprozess im gleichen Diagramm, bei *reusable* auf einen Subprozess, der in einem anderen Diagramm definiert ist.
Die BPEL-Spezifikation hingegen kennt keine Subprozesse. BPMN-Subprozesse müssen bei der Übersetzung entweder vollständig ausgerollt oder als eigenständige Webservices eingebunden werden.

Elementidentifikation

Wie wir später sehen werden, ist es für die Entwicklung unseres Adaptionskonzeptes von grundlegender Bedeutung, dass wir einzelne Prozess-Elemente eindeutig identifizieren können. Bei Änderungen am BPMN-Diagramm müssen die geänderten Elemente auf der Prozessdefinitions- und auf der Ausführungsebene „wiedergefunden“ werden können.
Im ADEPT-Basismodell sind die Prozesselemente als Menge N, als „eine endliche Menge von eindeutigen (Aktivitäten-) Knotenbezeichnern“ definiert [Rei00]. Aufgrund des durchgängigen Konzepts kann ein Element sowohl auf Instanzebene, wie auch bei der Durchführung von Änderungsoperationen angesprochen werden.
Der BPMN-Standard sieht für jedes Element eines Prozessdiagrammes eine eindeutige ID vor. Auf der Prozessdefinitionsebene kennt BPEL analog hierzu das *name*-Attribut einer Aktivität. Dieses ist aber als optional deklariert und muss nicht eindeutig sein. In diesem Punkt müssen wir eine Einschränkung der BPEL-Spezifikation vornehmen. Wir schreiben die zwingende Verwendung eines eindeutigen *name*-Attributes vor. Ohne diese Maßnahme kann unserer Ansicht nach kein sinnvolles Adaptionskonzept auf BPMN/BPEL-Basis entwickelt werden.
Sinnvollerweise sollte der BPEL-Compiler den Inhalt der BPMN-ID als Wert für das *name*-Attribut in BPEL übernehmen. Ist dies nicht der Fall, muss zumindest eine 1:1-Zuordnung von

[1] Im Umkehrschluss ist hieraus aber nicht zu folgern, dass alle ADEPT-Knoten Geschäftsprozess-Aufgaben darstellen! Gleiches gilt für BPMN-Tasks und BPEL-Activities.

BPMN-ID und BPEL-Name bekannt sein. Gleiches gilt für die Zustandshaltung auf der Ausführungsebene. Auch hier muss über den BPMN- bzw. BPEL-Identifier ein eindeutiger Zugriff auf den Zustand einer Aktivität möglich sein.

4.3 Datenfluss-Analyse in BPEL

Im Gegensatz zu manchen anderen Prozessdefinitions-Sprachen wird der Datenfluss in BPEL nicht explizit modelliert. Es werden zwar Prozess-Variablen und Datenzugriffe definiert, der hierdurch implizit definierte Datenfluss lässt sich aber nur mit Hilfe von Datenfluss-Analyseverfahren in expliziter Form extrahieren.

Eine Datenfluss-Repräsentation eines BPEL-Prozesses ist für mehrere Zwecke einsetzbar. In dieser Arbeit benötigen wir den Datenfluss für eine Def-Use-Analyse innerhalb eines lokalen Prozesses, also zur Überprüfung, ob eine zu lesende Variable zuvor initialisiert bzw. beschrieben wurde. Aber nicht nur für die lokale Betrachtung von Prozessen sind Datenfluss-Analyseverfahren für BPEL relevant. Gerade auch bei der Verifizierung und Optimierung verteilter Workflows sind sie essentiell notwendig. Beispielhafte Anwendungen hierfür werden im Rahmen der vorgestellten Analyseverfahren im Folgenden noch erläutert.

Es gibt mehrere Ansätze und eine größere Zahl von Arbeiten zur Kontrollflussanalyse in BPEL. Zur Datenflussanalyse existieren aber nur zwei relevante umfassende Ansätze, die bislang auch noch nicht vollständig ausgearbeitet wurden. Beide Verfahren wurden erst innerhalb der letzten Monate entwickelt, so dass in nächster Zeit noch weitergehende Veröffentlichungen zu diesem Themengebiet zu erwarten sind.

Die Datenfluss-Analyse wird im Compilerbau seit Jahrzehnten benutzt und wurde hier in der Vergangenheit umfassend untersucht. Deshalb liegt der Gedanke nahe, für die Prozess-Analyse Ansätze aus dem Compilerbau zu verwenden. Der im zweiten Abschnitt vorgestellte Ansatz der Datenfluss-Analyse auf Basis einer *Concurrent-Single-Static-Assignment-Form (CSSA-Form)* verwendet direkt eine für den Compilerbau entwickelte Repräsentationsform.

4.3.1 Datenabhängigkeiten zwischen BPEL-Fragmenten

Im Rahmen des *Tools4BPEL*-Projektes wurde am Institut für Architektur von Anwendungssystemen (IAAS) der Universität Stuttgart in Zusammenarbeit mit Rania Khalaf vom IBM TJ Watson Research Center untersucht, wie sich bestehende BPEL-Prozesse aufspalten lassen. Dies ist beispielsweise dann notwendig, wenn ein Teil des Geschäftsprozesses ausgegliedert und dieser Teil des Prozesses damit in Zukunft durch die Workflow-Engine des Partnerunternehmens gesteuert werden soll.

Motivation

Bei der Aufspaltung des Prozesses werden Variablenzugriffe, die bisher lokal erfolgten, jetzt aber über die Grenzen des lokalen Prozesses hinweg vorgenommen werden müssen, durch Message Flows ersetzt. Es ist sinnvoll, die Anzahl der Datenabhängigkeiten möglichst gering zu halten, um den Kommunikationsoverhead zwischen den beteiligten Partnern zu minimieren. Zu diesem Zweck ist eine detaillierte Datenflussanalyse des BPEL-Prozesses notwendig, um Datenabhängigkeiten herauszufinden und bestimmen zu können, welche Prozess-Variablen zwischen den Partnern mit Hilfe von Nachrichten ausgetauscht werden müssen.

Eine wesentliche Rolle hierbei spielt die *Dead Path Elimination (DPE)*, wie sie in der BPEL-Spezifikation vorgesehen ist. Innerhalb von *flow*-Aktivitäten lassen sich in BPEL mit Hilfe

von *Links* Ausführungsabhängigkeiten zwischen Aktivitäten definieren. Optional ist hierbei die Angabe einer *joinCondition*, die zu *true* evaluiert werden muss, bevor die Zielaktivität gestartet werden kann. Wird sie zu *false* ausgewertet, bedeutet dies, dass die Zielaktivität dauerhaft nicht ausgeführt werden kann, sie wird deaktiviert. Unter der DPE versteht man die Vorwärts-Propagierung einer Aktivitäts-Deaktivierung im Graphen. Würde diese Propagierung nicht erfolgen, würden alle nachfolgenden Aktivitäten endlos auf die Ausführung der deaktivierten Aktivität warten, bevor ihre *joinCondition* ausgewertet wird. Die Dead Path Elimination ermöglicht eine Evaluierung der nachfolgenden *joinConditions*, sobald alle anderen eingehenden Pfade ausgeführt wurden.

Ansatz

Der Ansatz der Autoren war nun, die DPE im Rahmen der Datenfluss-Analyse zu berücksichtigen und somit die Anzahl der Datenabhängigkeiten zu minimieren. Einen guten Überblick hierzu bietet der Technische Bericht [KKL07].

Die Betrachtung der Datenfluss-Analyse beschränkte sich in den ersten Veröffentlichungen auf diejenigen BPEL-Konstrukte, die für die Dead Path Elimination relevant sind. Hinsichtlich der strukturierten Aktivitäten ist dies insbesondere die *flow*-Aktivität, innerhalb der die DPE vorgesehen ist. Weiterhin wurden *while*- und *scope*-Elemente als strukturierte Aktivitäten in die Datenfluss-Analyse mit einbezogen.

Als Grundlage der Analyse-Algorithmen wurde zunächst eine Notation vorgestellt, die als Grundlage für die Datenflussanalyse-Algorithmen dient. Auf diese Notation lässt sich der gesamte Prozess abbilden, insbesondere werden auch Elementhierarchien und Links innerhalb der *flow*-Elemente explizit erfasst. Auf Datenebene relevant ist die Erfassung des Zustandes aller Schreibaktivitäten für jede Variable. Dies geschieht jeweils für alle Vorgänger einer Aktivität oder eines Links, jeweils vor $(writes_{\circ})$ und nach $(writes_{\bullet})$ ihrer bzw. seiner Interpretierung.

Für schreibende Aktivitäten wird ein Zustand definiert, der sich aus vier Komponenten zusammensetzt. Für die einzelnen Komponenten wurde jeweils eine Rückgabefunktion definiert:

1. $poss(x,v_e)$: Liefert alle möglicherweise schreibenden Aktivitäten bezüglich der angegebenen Aktivität bzw. des angegebenen Links x und der benannten Variable v_e. „Möglicherweise schreibend“ heißt, dass der Variablenwert, den die zurückgegebene Aktivität schreibt, die Aktivität / den Link x erreichen *kann*. Dies ist der Fall, wenn der Variablenwert nicht zwingend von einer nachfolgenden Aktivität überschrieben wird.

2. $dis(x,v_e)$: Diese Funktion liefert alle schreibenden Aktivitäten zurück, deren Variablenzugriffe (Schreibzugriffe) durch nachfolgende Schreibaktivitäten überschrieben werden.

3. $inv(x,v_e)$: Je nach Ergebnis der Ausführungsbedingungen im Prozessablauf kann es sein, dass eine durch *dis* (wie *disabled*) erfasste Aktivität für nachfolgende Anfrage-Aktivitäten bzw. -Links wieder den Zustand *possibleWriter* (*poss*) annehmen kann. Allen Schreibaktivitäten, deren Schreibzugriffe aber definitiv überschrieben werden, auch aus Sicht aller nachfolgenden Anfrage-Aktivitäten / -Links, wird der Zustand *inv* (wie *invalid*) zugeordnet.

4. $mbd(x,v_e)$: Die boolsche Funktion *mbd* (wie *may be dead*) trifft eine Aussage darüber, ob die Aktivität bzw. der Link x eventuell nicht ausgeführt wird, bedingt durch eine Dead Path Elimination betreffend den Pfad von der direkt vorhergehenden Schreibaktivität auf die Variable v_e bis zum Anfrageelement x.

Wie bereits angedeutet sind die Werte der Zustandskomponenten direkt von der statischen Interpretierung der Ausführungsbedingungen innerhalb der *flow*-Elemente des BPEL-Prozesses abhängig. Zum tiefergehenden Verständnis der Zustandsbestimmung in Bezug auf die Dead Path Elimination sei auf den Technischen Bericht verwiesen. In Kombination mit [KKL08]

ermöglicht die ausführliche Darstellung dort ein besseres Verständnis für die grundlegenden Zusammenhänge, als es durch die stark verkürzte Wiedergabe im Rahmen der vorliegenden Arbeit möglich ist.

Der eigentliche Algorithmus zur Bestimmung des Datenflusses besteht zunächst aus einer Bestimmung der Menge aller Variablen, die im Prozessverlauf durch eine Schreibaktivität geschrieben werden. Für jede Variable wird dann eine Tiefensuche über den Prozess ausgeführt, die jeweils die Zustände $\text{writes}_\circ$ und $\text{writes}_\bullet$ für jede schreibende Aktivität bestimmt.

Damit existiert eine detaillierte Datenfluss-Repräsentation, die als Grundlage für weitere Analysen benutzt werden kann. In diesem Fall wird sie für die Bestimmung von Datenabhängigkeiten zwischen Prozessfragmenten und der damit verbundenen Minimierung von Message Flows zwischen diesen Fragmenten eingesetzt.

Einschränkungen / Besonderheiten

Wie alle bislang entwickelten Datenflussmodelle für BPEL ist auch das gerade vorgestellte gewissen Einschränkungen unterworfen. Wir haben bereits erwähnt, dass in den Arbeiten von Oliver Kopp, Rania Khalaf und Frank Leymann unter den strukturierten Aktivitäten nur *flow*-, *while*- und *scope*-Elemente betrachtet werden. Diese Einschränkung hebt allerdings Sebastian Breier in [Bre08] auf und erweitert den Algorithmus um die fehlenden Aktivitäten. Ebenso werden durch seine Erweiterungen nun Event-, Fault-, Termination- und Compensation-Handler unterstützt, was zuvor ebenfalls nicht der Fall war.

Eine wesentliche Einschränkung, die die Autoren in ihrer Arbeit vornehmen, ist das Verbot des Schreibens und „gleichzeitigen“ Lesens einer Variable in parallelen Pfaden. Diese Annahme liegt zwar nahe, wenn man zu einem exakten Datenflussmodell gelangen möchte, wie man es aus dem Compilerbau kennt. Die BPEL-Spezifikation sieht diese Einschränkung allerdings nicht vor.

4.3.2 CSSA-basierte Datenflussanalyse

In [MMG+07] stellt eine Forschergruppe um Simon Moser von IBM, Böblingen, und Wolfram Amme von der Universität Jena einen CSSA-basierten Datenflussanalyse-Ansatz vor. Ihr Ziel besteht darin, eine Möglichkeit zu finden, wie bestehende BPEL-Verifikationsansätze um eine statische Datenfluss-Überprüfung erweitern werden können. Bei bisherigen Verfahren findet nur eine Kontrollfluss-Überprüfung statt, der Datenfluss bleibt unberücksichtigt, was zu vielen false-positive-Resultaten bei der Überprüfung von WS-BPEL-Prozessen führt.

Ansatz

CSSA steht für *Concurrent Single Static Assignment* und ist eine aus dem Compilerbau bekannte Repräsentationsform zur Datenflussüberprüfung. Sie ist eine um die Erfassung von Nebenläufigkeiten erweiterte *SSA-Form (Single Static Assignment-Form)*. In einer CSSA-Form kann sowohl der Kontroll-, wie auch der Datenfluss eines Programmes – in diesem Fall eines Prozesses – erfasst und analysiert werden.

Die grundlegende Idee bei einer CSSA-Form besteht darin, jede Variable als Konstante anzusehen, der einmalig ein Wert zugewiesen wird. Da Variablen in BPEL wie auch in anderen Sprachen mehrfach beschrieben werden können, wird der Variablenname um einen Index erweitert. Dieser Index erhöht sich jedes Mal, wenn die Variable beschrieben wird (siehe Abb. 4.3). So lässt sich an jeder Stelle des Kontrollflusses die Herkunft eines Variablenwertes direkt zurückverfolgen, da die für die Zuweisungsoperation eventuell verwendeten Variablen eindeutig

benannt sind und sich somit wiederum auf ihre jeweils letzte Schreiboperation zurückführen lassen.

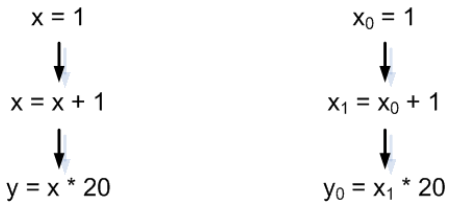


Abbildung 4.3: CSSA – Variablenindizierung

Bei dieser Herangehensweise ergeben sich zwei grundlegende Probleme. Zum einen müssen die indizierten Variablen bei einer Zusammenführung des Kontrollflusses nach einer Verzweigung ebenfalls wieder zusammengeführt werden. Dies geschieht durch die Verwendung einer speziellen ϕ-Funktion. Mit diesem Funktionsaufruf wird eine neue indizierte Variable definiert, die ihren Wert je nach Ablauf des Kontrollflusses von einer Variable aus einer der Verzweigungen erhält.
Im Beispiel in Abbildung 4.4 ist die Verwendung dieser Funktion dargestellt: Je nach Auswertungsergebnis der Bedingung folgt der Kontrollfluss dem linken oder dem rechten Pfad der OR-Verzweigung. Der Wert der Variablen x_0 wird um 10 bzw. 20 erhöht und in der Variable x_1 bzw. x_2 gespeichert. Beim Zusammenführen der Verzweigung ist zur Definitionszeit nicht klar, welcher von beiden Pfaden gewählt wurde und welchen Wert die Variable x nach Durchlauf der Verzweigung annimmt. Aus diesem Grund wird eine neue Variable x_3 definiert. Dieser wird der Funktionswert der ϕ-Funktion zugewiesen. Als Parameter werden die Variablen x_1 und x_2 übergeben. Somit wird bei der Analyse des Datenflusses deutlich, dass sich der Wert von x_3 (und somit von y_0) aus dem Variablenwert von x_1 oder x_2 ergibt.

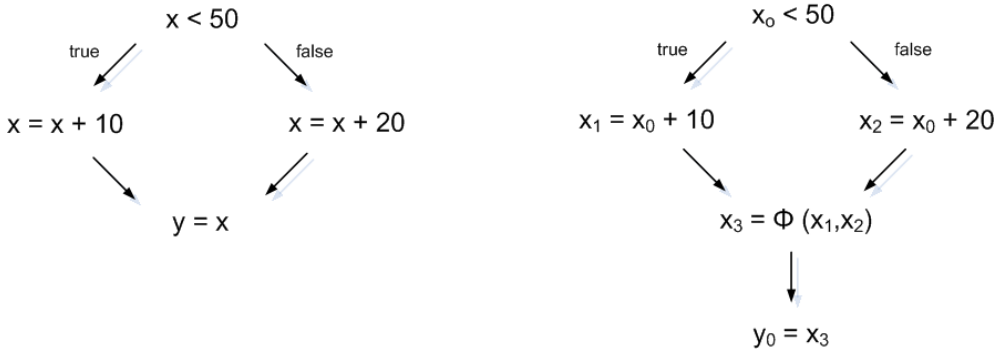


Abbildung 4.4: CSSA – ϕ-Funktion

Der andere Punkt betrifft den Variablenzugriff in parallelen Verzweigungen. Im Gegensatz zum oben vorgestellten Ansatz ist ein paralleler Datenzugriff hier explizit erlaubt. Das heißt, es muss der Fall berücksichtigt werden, dass eine in einem Pfad gelesene Variable bereits zuvor in einem anderen Pfad verändert wurde. Dies wird durch die Verwendung einer π-Funktion modelliert. Abbildung 4.5 zeigt einen solchen Fall. Hier wird die Variable x in dem einem Pfad um 10 erhöht, im anderen mit 20 multipliziert. Die Ausführungsreihenfolge ist entscheidend für das letztendliche Ergebnis. Zur Darstellung der Datenabhängigkeiten wird in beiden parallelen Pfaden vor der eigentlichen Operation eine Zuweisung eingefügt. Diese definiert eine neue Variable und gibt mittels der π-Funktion ihre Abhängigkeit vom Variablenwert vor Ausführung der Verzweigung, aber auch vom Variablenwert nach Ausführung des jeweils anderen Pfades an. Betrachtet man beispielsweise den linken Pfad, so kann x_4 den Wert von x_0 annehmen. Es kann

aber auch sein, dass der rechte Pfad bereits vor dem linken ausgeführt wurde. In diesem Fall wird der Variable x_4 der Wert von x_2 zugewiesen.

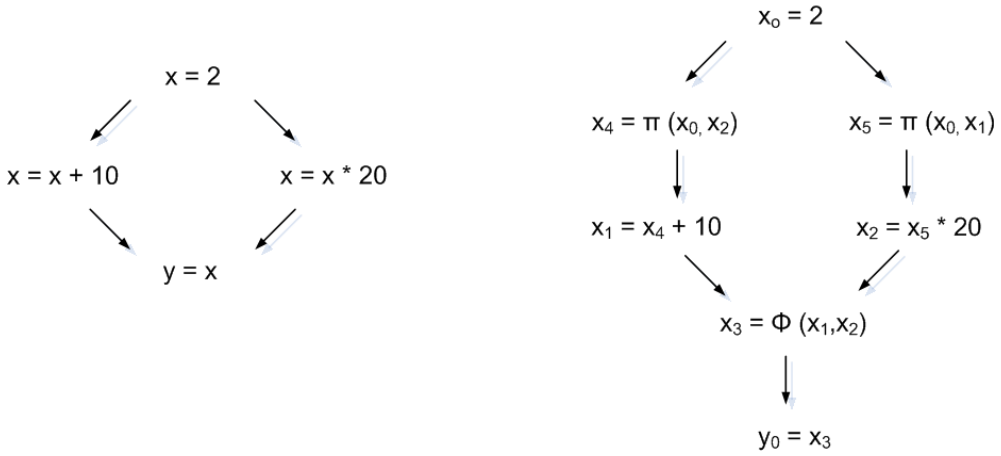


Abbildung 4.5: CSSA – π-Funktion

Zur Überführung des BPEL-Codes in eine CSSA-Form stellen die Autoren ein dreistufiges Verfahren vor. Zunächst wird ein *Concurrent Control Flow Graph* (*CCFG*, s. [Gos04]) erzeugt. Die Abbildung der einzelnen BPEL-Elemente auf die jeweilige CCFG-Repräsentation ist in [MMG+07] übersichtlich dargestellt. [GÖ8] erweitert diese Übersetzung und erläutert sie ausführlich.
Aus dem Concurrent Control Flow Graph wird dann eine SSA-Form erzeugt, die im nächsten Schritt um die nötigen π-Funktionen zur Analyse paralleler Datenzugriffe erweitert wird.

Der Algorithmus, der aus der CSSA-Form die eigentlichen Datenabhängigkeiten gewinnt, ähnelt teilweise dem, der im Rahmen des ADEPT-Projektes entwickelt wurde. Zunächst wird jeder Knoten mit einer leeren Datenabhängigkeitsmenge initialisiert. Im ersten Schritt werden dann zunächst die direkten Datenabhängigkeiten derjenigen Knoten festgehalten, die Nachrichten empfangen können. Im zweiten Schritt wird der Graph iterativ durchschritten, und die Datenanhängigkeiten der einzelnen Knoten werden nach und nach erfasst.
Aus der hieraus gewonnenen Datenabhängigkeitsrepräsentation lässt sich ablesen, von welchen Datenelementen ein Knoten bzw. seine Datenelemente abhängig sind.

Einschränkungen / Besonderheiten

Der vorgeschlagene Ansatz basiert auf einem Datenmodell, dem CSSA-Graphen, das im Compilerbau seit langer Zeit eingesetzt wird und wissenschaftlich intensiv untersucht wurde. Für dieses Datenmodell wurde eine BPEL-Variante entwickelt, die alle BPEL-Kontrollflusselemente unterstützt. Auch die Analyse paralleler Datenzugriffe ist im gezeigten Modell ohne Einschränkungen möglich.
Katharina Görlach erweitert dieses Modell in [GÖ8], insbesondere um eine detaillierte Repräsentation von XPath-Ausdrücken. Hiermit ist es beispielsweise möglich, für beliebige Stellen im Prozess exakte Wertebereiche von Variablen angeben zu können.
Im vorgestellten Verfahren fehlen aber nach wie vor einige relevante Funktionen bzw. wurden manche Aspekte bislang vernachlässigt. So ist es zum Beispiel nicht möglich, Knotenmengen durch die Angabe von Prädikaten in XPath-Lokalisierungspfaden zu filtern.

4.4 Beurteilung der Ergebnisse

Das ADEPT-Konzept zeigt einen durchgängigen Adaptionspfad vom Prozess-Schema in die Prozess-Instanzen auf und dient in wesentlichen Punkten als Leitlinie bei der Entwicklung unseres BPMN/BPEL-basierten Adaptions-Modells. Wir greifen insbesondere auf die vorgeschlagene Operationshierarchie zurück und nutzen die Konsistenzkriterien als Orientierungshilfe bei der Entwicklung eigener modellspezifischer Kriterien. Des Weiteren gehen wir auf BPMN- und BPEL-Ebene von einer Beschränkung auf ein blockstrukturiertes Prozessmodell aus.

Die betrachteten Datenfluss-Analyseverfahren für BPEL lassen sich nicht sinnvoll in unsere Arbeit integrieren. Die vorgestellten Modelle sind mächtig, sehr komplex und teilweise auch noch nicht ausgereift. Wir hingegen benötigen, wie wir später sehen werden, nur recht einfache Methoden zur Datenflussüberprüfung.
Eine Einbindung der Verfahren in unser Modell würde einen erheblichen Overhead bedeuten, sowohl was die Erarbeitung des Datenfluss-Konzeptes und die spätere Implementierung, als auch die schriftliche Darlegung angeht. Zudem hat das von uns entwickelte und in Kapitel 5 vorgestellte Verfahren den Vorteil, dass es direkt auf dem BPEL-Code ausführbar ist und keiner Zwischenrepräsentation bedarf.
Eine Entwicklung von Bibliotheken für BPEL-Datenflussanalysen, basierend auf den vorgestellten Modellen, wäre wünschenswert. Würden solche existieren, wäre auch eine Einbindung in unser Adaptions-Konzept sinnvoll. Gerade in Hinblick auf die zukünftige Ausarbeitung von komplexeren Adaptionsoperationen oder die Betrachtung von Adaptionstransaktionen könnte ein standardisiertes, verifiziertes und modular anwendbares Datenfluss-Analyseverfahren von Vorteil sein, da unser Vorgehen dann eventuell nicht mehr ausreicht bzw. die notwendigen Erweiterungen ebenfalls zu einer hohen Komplexität führen.

Einordnung dieser Arbeit

Für das zu entwickelnde Adaptionskonzept auf BPMN/BPEL-Basis existieren keine uns bekannten alternativen Ansätze. Die Mehrzahl der wissenschaftlichen Arbeiten im Bereich der Workflow-Adaptivität stützt sich auf spezielle Prozessmodelle, die die Durchführung von Adaptionsvorgängen vereinfachen. Andere Arbeiten basieren auf allgemein anerkannten Prozessmodellen, betrachten aber nur einzelne Adaptions-Aspekte. Uns ist keine Arbeit bekannt, die auf der Grundlage von BPMN- und BPEL-Modellen einen durchgängigen Adaptionspfad für Änderungen am Prozessablauf aufzeigt.

Bei der Erarbeitung des Vorgehens lassen sich Ideen aus dem ADEPT-Projekt auf unser BPMN/BPEL-Konzept übertragen. Gerade in Bezug auf die Korrektheits- und Konsistenzerhaltung weist die Instanzebene unseres Modells aber Besonderheiten auf, die bislang in keiner Arbeit vollständig betrachtet wurden.

Kapitel 5

Konzeption

In diesem Kapitel stellen wir das entwickelte Adaptionskonzept vor. Wir erläutern zunächst die grundsätzliche Vorgehensweise. Als Basis für die Formulierung der Adaptionsoperationen erstellen wir ein formales Modell für die BPMN-Ebene wie auch für die Instanzebene. Darauf aufbauend nehmen wir auf beiden Ebenen die detaillierte Ausarbeitung der Adaptionsfunktionalität vor.

5.1 Gesamtkonzept

Es stellt sich zunächst die Frage, wie eine Prozess-Adaption durch den Benutzer zu formulieren ist. Grundsätzlich stehen zwei Möglichkeiten zur Auswahl:
Entweder werden dem Benutzer mehrere feste Adaptions-*Operationen* zur Verfügung gestellt. Hiervon kann er eine auswählen, um seine gewünschte Änderung durchführen zu lassen.
Die andere Möglichkeit wäre, dass der Benutzer den bestehenden Prozess mit einem BPMN-Modellierungswerkzeug vollkommen frei ändern kann, wie er es auch von der normalen Prozessmodellierung gewohnt ist. Anschließend analysiert die Software auf BPMN oder BPEL-Ebene, welche Änderungen vorgenommen wurden, überprüft, ob die hierfür nötigen Voraussetzungen gegeben sind und sorgt – soweit dies möglich ist – für eine korrekte Adaption der laufenden Prozesse.

Wir haben uns in dieser Arbeit für die erstgenannte Möglichkeit entschieden. Dem Benutzer stehen also nur vordefinierte Änderungsoperationen zur Verfügung. Dies stellt zunächst eine erhebliche Einschränkung der Ausdrucksmächtigkeit möglicher Änderungen dar, vereinfacht die Entwicklung eines grundlegenden Adaptionskonzeptes allerdings erheblich.
Unsere Vorgehensweise ist hinsichtlich der Ausdrucksmächtigkeit keine Sackgasse. Es ist denkbar, in einem weiterführenden Schritt ein Konzept dafür zu entwickeln, wie alle Arten von freien Prozess-Änderungen auf die von uns vorgeschlagenen Änderungsoperationen abgebildet werden können. Voraussichtlich müssen die vorgestellten Operationen hierfür ergänzt und eventuell auch angepasst werden. Die Entwicklung eines solchen Abbildungskonzeptes wurde in dieser Arbeit aus Aufwands- und Komplexitätsgründen bewusst ausgeklammert.

5.1.1 Ablauf über alle Komponenten hinweg

Im Folgenden zeigen wir auf, wie wir die Umsetzung einer Adaption durch alle Workflow-Ebenen hindurch vornehmen. Dieser Ansatz dient als Grundlage unseres Adaptionskonzeptes. Die detaillierte Umsetzung der Adaptionsfunktionalität auf den einzelnen System-Ebenen wird

im weiteren Verlauf dieses Kapitels detailliert ausgearbeitet.

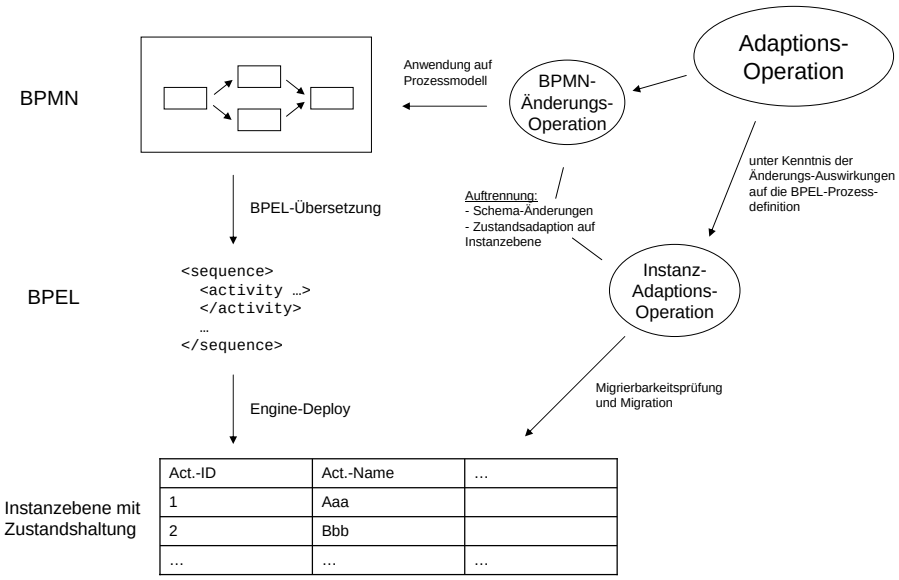


Abbildung 5.1: Adaptionskonzept

Wir haben den Adaptions-Ablauf in Abbildung 5.1 skizziert. Dem Benutzer steht ein Satz von Adaptionsoperationen zur Verfügung. Aus diesem wählt er eine Operation aus und gibt die benötigten Parameter an.
Die Adaptionsoperation wird im zweiten Schritt in eine Änderungsoperation bzgl. des Prozess-Schemas (BPMN-Ebene) und eine Zustands-Adaptionsoperation auf Instanzebene aufgespaltet. Die Schema-Änderungsoperation beschreibt die Anpassungsschritte, die auf BPMN-Ebene vorzunehmen sind, um zu dem gewünschten neuen Prozessmodell zu gelangen. Hierin enthalten sind alle strukturellen Anpassungen wie auch alle Änderungen, die sich auf die Dateneinbindung beziehen.
Die Instanzadaptionsoperation beschreibt die eigentliche Logik der gewünschten Prozessadaption. Sie enthält verschiedene Vorbedingungen, die erfüllt sein müssen, um einzelne Instanzen unter Beibehaltung ihrer Integrität migrieren zu können. Des Weiteren werden hier die genauen Schritte angegeben, wie die Zustände der einzelnen Instanzen auf das neue Prozessmodell zu migrieren sind.
Die BPEL-Repräsentation selbst wird nicht „von Hand“ geändert. Der BPEL-Compiler erzeugt aus dem abgeänderten BPMN-Modell automatisch ein neues BPEL-Modell, auf das die laufenden Instanzen dann migriert werden.

Insgesamt ergibt sich also ein dreistufiges Konzept:

1. Auswahl und detaillierte Spezifizierung der durchzuführenden Adaptionsoperation durch den Benutzer.
2. Eine Adaptionsoperation auf Benutzerebene löst zwei Operationen aus, die die eigentliche Adaptionsfunktionalität enthalten: Eine BPMN-Änderungsoperation und – nach Übersetzung des neuen Schemas – eine Instanz-Adaptionsoperation.
3. Umsetzung der BPMN-Änderungsoperation und der Instanz-Adaptionsoperation auf allgemeingehaltene Änderungs- bzw. Adaptionsprimitive.

Es wäre auch eine alternative Vorgehensweise denkbar, die unserer Ansicht nach aber weniger sinnvoll ist. Bei dieser würde der BPEL-Compiler im Rahmen eines Adaptionsvorganges gar nicht mehr zum Einsatz kommen. Ersatzweise würde die Migrationskomponente strukturelle

Änderungen direkt auf der BPEL-Datei vornehmen.
Dieser Ansatz hat den Nachteil, dass für alle Änderungsoperationen die jeweils zugehörige BPEL-Änderung genau definiert werden muss. Wie wir in Kapitel 2.3.3 bereits gezeigt haben, können einfache Änderungen am BPMN-Modell umfangreichere Änderungen der BPEL-Prozessdefinition nach sich ziehen. Es müssten deshalb sämtliche Spezialfälle berücksichtigt und in das Adaptionskonzept integriert werden.
Der Vorteil hierbei wäre, dass man die konkrete Umsetzung aller Änderungen selbst in der Hand hätte und sich nicht darauf verlassen müsste, dass der BPEL-Compiler die Übersetzung des BPMN-Modells immer wie gewünscht vornimmt.

Um eine korrekte Aufspaltung der allgemeinen Änderungsoperation in eine BPMN- und eine Instanz-Änderungsoperation vornehmen zu können, muss aber auch bei der von uns gewählten Vorgehensweise die genaue Art und Weise der Übersetzung von BPMN nach BPEL bekannt sein. Wir gehen in dieser Arbeit von einer Übersetzung aus, wie sie in der BPMN-Spezifikation vorgeschlagen wird. Die Spezifikation beschreibt allerdings nur die Übersetzung von BPMN 1.1 nach BPEL 1.1. Die Transformation auf eine BPEL 2.0-Übersetzung ergibt sich in den für uns relevanten Punkten größtenteils intuitiv. Wo dies nicht der Fall ist, gehen wir von einer Übersetzung aus, wie sie der Intalio BPEL-Compiler vornimmt.
Für unser Adaptionskonzept müssen wir den offiziellen Vorschlag allerdings etwas abwandeln bzw. Einschränkungen am BPEL-Modell vornehmen. Analog zum ADEPT-Modell gehen wir sowohl auf BPMN- wie auch auf BPEL-Ebene von einem vollständig blockstrukturierten Prozess-Modell aus. Hiervon betroffen ist insbesondere die *flow*-Aktivität in BPEL. Wir erlauben keine *Links* innerhalb von *flow*-Elementen, sondern fordern eine saubere Schachtelung der Aktivitäten. Ausführungsbedingungen sind über BPMN-*Gateways* ins Prozessmodell zu integrieren, die Verwendung von *joinConditions* bzw. *transitionConditions* in BPEL ist nicht zulässig. Damit ist auch die Definition von *targets* und *sources* innerhalb von Aktivitäten nicht notwendig.
Eine exakte Definition der von uns vorausgesetzten Form der Übersetzung wäre sicherlich wünschenswert, würde aber den Rahmen dieser Arbeit sprengen. Für die Änderungsoperationen, die wir im Folgenden vorstellen werden, ist dieser Ansatz als Grundlage aber ausreichend und kann in späteren Arbeiten verfeinert werden.

5.1.2 Betrachtete Änderungsoperationen

Aus zeitlichen Gründen werden in dieser Arbeit nur zwei grundlegende Änderungsoperationen betrachtet. Trotzdem soll insgesamt ein möglichst allgemein gehaltenes Adaptions-Modell auf BPMN/BPEL-Basis entwickelt werden, das später um zusätzliche Änderungsoperationen erweitert werden kann.

Serielles Einfügen einer einfachen Aktivität

Als elementare Änderungsoperation betrachten wir zunächst das serielle Einfügen einer neuen einfachen „Aktivität“, also eines *Tasks* auf BPMN-Ebene. Wir beschränken uns in dieser Arbeit auf das Einfügen zwischen zwei schon bestehende Tasks. Auf die Gründe hierfür gehen wir in den Kapiteln 5.4.2 und 5.4.4 ein.

Aufruf:

Adaptionsoperations-Aufruf auf Benutzerebene:

TaskSerialInsert(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*, *TaskName*, *PredecessorElementIdentifier*, *SuccessorElementIdentifier*, {*CurrentInstanceIdentifier*})

Diese Funktion löst zwei Aufrufe aus:

Auf BPMN-Ebene:

TaskSerialInsert(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*, *TaskName*, *PredecessorElementIdentifier*, *SuccessorElementIdentifier*)

Auf Instanz-Ebene (nach der Übersetzung des neuen BPMN-Schemas nach BPEL):

BasicActivitySerialInsert(*NewProcessIdentifier*, *TaskIdentifier*, {(*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)})

Erläuterung:

Auf Benutzerebene, wo die Änderungsoperation ausgelöst wird, sind folgende Angaben erforderlich:
Mit *CurrentProcessIdentifier* wird das zu ändernde Prozess-Schema spezifiziert. Der *NewProcessIdentifier* referenziert das neue Prozess-Schema nach Durchführung der Änderungsoperation. *TaskIdentifier* ist der eindeutige Identifier des einzufügenden Tasks. Beim Einfügen eines BPMN-Elementes kann die Spezifizierung einer Vielzahl von Elementattributen notwendig sein. Exemplarisch für alle weiteren führen wir aus Gründen der Übersichtlichkeit nur das *TaskName*-Attribut auf. *PredecessorElementIdentifier* und *SuccessorElementIdentifier* geben die Prozess-Elemente an, zwischen denen der neue Task eingefügt werden soll. Die Menge *{CurrentInstanceIdentifier}* enthält die IDs der Prozess-Instanzen, die auf das neue Schema migriert werden sollen.

Die Benutzer-Änderungsoperation *TaskSerialInsert* muss nun auf die BPMN-Ebene übertragen werden. Was dies inhaltlich bedeutet, zeigen wir später auf. Bezüglich der benötigten Attribute fällt auf, dass die Angabe der zu migrierenden Instanzen auf BPMN-Ebene nicht nötig ist. Dies ergibt sich daraus, dass auf dieser Ebene nur das Prozess-Schema selbst geändert wird, die Instanzen sind an dieser Stelle nicht berührt.

Auf Instanzebene spielen hingegen nur die instanzbezogenen Attribute eine Rolle: *NewProcessIdentifier* ist der Identifier des neu erzeugten Prozess-Schemas, welches auf BPMN-Ebene erzeugt wurde. Die Angabe des *TaskIdentifiers*, der später in den entsprechenden BPEL-Aktivitätsidentifier übersetzt wird, ist für verschiedene Prozessstatus- und Datenfluss-Überprüfungen notwendig. Die Menge der (*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)-Tupel gibt für jede laufende Prozess-Instanz diejenige Instanz-ID an, auf die migriert werden soll. Letztere ist eine neu erzeugte Instanz des durch *NewProcessIdentifier* referenzierten Prozess-Schemas, die sich zunächst im Ausgangszustand befindet. Sie wird von der Workflow-Engine bzw. der Migrationskomponente automatisch generiert und muss deshalb auf Benutzerebene nicht benannt werden.

Serielles Löschen einer einfachen Aktivität

Parallel zum Einfügen eines Tasks betrachten wir auch das serielle Entfernen eines Tasks aus einem Prozessablauf. Auch diese Operation können wir leider nicht allgemein halten, sondern müssen uns auf das Löschen eines Tasks aus einer Reihe von weiteren Tasks heraus beschränken. Der zu löschende Task darf hierbei nicht an erster und nicht an letzter Stelle stehen.

Aufruf:

Adaptionsoperations-Aufruf auf Benutzerebene:
TaskSerialRemove(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*, {*CurrentInstanceIdentifier*})

Diese Funktion löst zwei Aufrufe aus:

Auf BPMN-Ebene:
TaskSerialRemove(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*)

Auf Instanz-Ebene:
BasicActivitySerialRemove(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*, {(*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)})

Erläuterung:

Im Gegensatz zur Einfügeaktivität ist beim Aufruf der Löschaktivität auf Instanzebene die Angabe des alten Prozess-Schemas notwendig. Dieses wird für die Prozessstatusüberprüfung benötigt, die bei Einfügeoperationen auf den neuen Instanzen, bei Löschoperationen aber auf den alten Instanzen und damit auf Basis des alten Prozess-Schemas durchgeführt wird.

5.1.3 Einschränkungen/Voraussetzungen

Wir müssen gewisse Einschränkungen treffen, um den Umfang und die Komplexität unseres Adaptionskonzeptes im Rahmen zu halten.

Reine Orchestrierungsbetrachtung

Wir beschränken uns bei der Entwicklung eines Adaptivitätskonzepts auf die Betrachtung eines einzelnen ausführbaren Prozesses. Wir führen also keine Analyse im Bereich der Webservice-Choreographie durch.

Blockstrukturiertheit

Wir gehen von einem blockstrukturierten Prozessmodell aus. Auf BPMN-Ebene hat dies zur Folge, dass das BPMN-Modellierungstool nur streng blockstrukturierte Prozessmodelle als gültig erkennen darf. Das bedeutet beispielsweise, dass einem aufsplittenden Gateway ein zusammenführendes nachfolgen muss. Verzweigungen müssen generell durch die Verwendung von Gateways modelliert werden, implizite Verzweigungen, also Tasks mit mehreren ausgehenden oder eingehenden Kanten, sind nicht zulässig.
Die Auswirkungen, die unsere Anforderung auf das BPEL-Modell hat, wurden in Kapitel 5.1.1 bereits angesprochen. Abweichungen von der Spezifikation ergeben sich insbesondere für die *flow*-Aktivität.

Sonstige Voraussetzungen

Die Benennung der Aktivitäten muss auf allen Ebenen (BPMN, BPEL, Zustandshaltung) konsistent erfolgen, oder es muss zumindest eine Abbildung der einzelnen Namensbereiche aufeinander bekannt sein. Das bedeutet, dass der Aktivitätsidentifikator bei jeder Übersetzung entweder beibehalten werden muss, oder aber die Umbenennung muss transparent erfolgen. Im Folgenden gehen wir aus Vereinfachungsgründen davon aus, dass der eindeutige Name bei den Übersetzungsvorgängen erhalten bleibt (siehe Abb. 5.2).

Diese Forderung impliziert, dass auch BPEL-Aktivitäten eindeutig benannt sein müssen, obwohl die Spezifikation das *name*-Attribut nur als optionales Attribut deklariert.

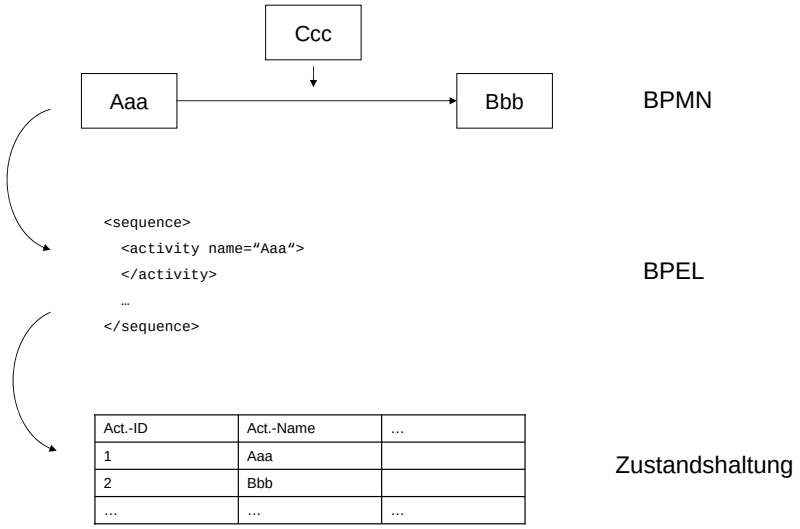


Abbildung 5.2: Durchgängige Aktivitätsbenennung

5.2 Formale Modelle

Um eine Grundlage für die Definition der Änderungsoperationen zu schaffen, müssen wir zunächst ein formales Meta-Modell entwickeln. Diese Notwendigkeit besteht sowohl für die BPMN- wie auch für die Instanzebene.[1] Auf Basis dieser beiden Meta-Modelle können wir dann die Änderungsprimitive definieren, die für die Definition der eigentlichen Änderungsoperationen benötigt werden.

In dieser Arbeit werden für BPMN und für die Instanzebene zwei unterschiedliche Modellarten verwendet. Die Spezifikation von BPMN legt die Entwicklung eines mengenbasierten Metamodells nahe. Für die Beschreibung der XML-basierten Business Process Definition Language ist ein Mengenmodell aber ungeeignet. Deshalb greifen wir auf Instanz-Ebene auf das bestehende XML-Schema-Modell für BPEL zurück. Dieses wird zusätzlich um eine Zustandshaltung der Prozessvariablen und um die Erfassung der Aktivitäts-Ausführungszustände erweitert.

Eigentlich müsste im Rahmen des hier aufgezeigten Adaptionskonzeptes auch eine Überführung einer Instanz des BPMN-Modells in das zustandslose BPEL-Modell definiert werden. Diese Übersetzung kann aus zeitlichen Gründen aber nicht entwickelt werden.

Wir haben die einzelnen Modelle nicht vollständig definiert. Es werden vorrangig die Elemente betrachtet, die für unser Adaptionskonzept relevant sind. Beispielsweise werden nicht für alle Element-Attribute die zugehörigen Abbildungen definiert. Insbesondere werden Attribute, die allein der visuellen Darstellung dienen (z.B. das *BoundaryVisible*-Attribut des BPMN *Pool*-Elements) oder Rollenzuweisungen (*ParticipantRef* des BPMN *Pool*-Elements, *Performers*-Attribut der BPMN-*Activities*) außen vor gelassen. Das gleiche gilt für BPMN-*Associations* und -*Artifacts*, da diese nicht auf BPEL gemapped werden und damit für diese Arbeit nicht von Bedeutung sind. Ad-hoc Prozesse können ebenfalls nicht nach BPEL übersetzt werden, deshalb werden auch sie nicht betrachtet.
Die Beschränkung auf die wesentlichen Bestandteile dient neben der Reduzierung des Ausarbeitungsaufwandes auch der Verbesserung der Übersichtlichkeit. Die Modelle wurden mit Blick

[1] Für BPEL existiert bereits ein solches formales Modell. Wir führen auf dieser Ebene auch keine Änderungsoperationen durch.

auf eine mögliche spätere vollständige Ausarbeitung entwickelt – sie sind zwar nicht vollständig, aber auch nicht beschränkt hinsichtlich einer Erweiterung, die beispielsweise die Spezifikationen vollständig abdeckt.

5.2.1 BPMN-Ebene

Die BPMN-Spezifikation definiert auf oberster Hierarchieebene ein BPMN-*Diagram* als ein Objekt mit den Attributen *Id*, *Name*, *Version*, *Author*, *Language*, *QueryLanguage*, *CreationDate*, *ModificationDate*, *Pools* und *Documentation*. Das Attribut Pools verweist hierbei auf die Prozess-Pools, die das Diagramm enthält. Jeder *Pool* enthält wiederum einen optionalen *Process*. Dieser wird über das Attribut *ProcessRef* referenziert.
Auf dieser Ebene, bei der Betrachtung eines einzelnen Prozesses, setzt unser Adaptionskonzept an. Für einen BPMN-*Process* enthält das Attribut *GraphicalElements* Verweise auf all diejenigen Elemente, die im jeweiligen Prozess enthalten sind. Über die Analyse der Attributwerte dieser Elemente lässt sich die gesamte Prozess-Struktur herleiten. Beispielsweise existieren für Kanten Attribute, die den Start- und den Ziel-Knoten der Verbindung referenzieren.

Wir übernehmen den Ansatz der Spezifikation und definieren allgemein ein BPMN-Diagramm als Tupel einer Menge von BPMN-Elementen und einer Menge zugehöriger Abbildungen, über die die Attributwerte festgelegt werden:

Diagram = (BPMN Elements, Abbildungen)

Diagram ist hierbei ein einzelnes, eindeutig referenzierbares Diagramm.
BPMN Elements ist eine Menge von eindeutig referenzierbaren BPMN-Elementen.
Abbildungen ist die Menge der den BPMN-Elementen zugehörigen Abbildungen.

Wir definieren im Folgenden zunächst die Typhierarchie der BPMN-Elemente. Daran anschließend definieren wir für jeden Elementtyp die zugehörigen Abbildungen, die unter anderem die Aufbauhierarchie bestimmen.

BPMN-Elements

BPMN Elements = {Graphical Elements, Supporting Elements}

Graphical Elements = {Flow Objects, Connecting Objects, Swimlanes, Artifacts}
Supporting Elements = {Assignments, Categories, Conditions, Entities, Event Details, Expressions, Gates, Inputs, Messages, Outputs, Participants, Processes, Properties, Roles, Transactions, Web Services}

Graphical Elements

Flow Objects = {Activities, Gateways, Events}
Connecting Objects = {Sequence Flow, Message Flow, Association}
Swimlanes = {Pools, Lanes}
Artifacts = {Data Objects, Groups, Annotations}

Flow Objects:
Activities = {Tasks, Sub-Processes}
Gateways = {Exclusive Gateways, Inclusive Gateways, Complex Gateways, Parallel Gateways}
Events = {Start Events, End Events, Intermediate Events}

Task ∈ String[2]
Sub-Processes = {Embedded Sub-Processes, Reusable Sub-Processes, Reference Sub-Processes}
Embedded Sub-Processes, Reusable Sub-Processes, Reference Sub-Processes ∈ String
Exclusive Gateways, Inclusive Gateways, Complex Gateways, Parallel Gateways ∈ String
Start Events, End Events, Intermediate Events ∈ String

Connecting Objects:
Sequence Flows ⊆ Flow Objects x Flow Objects
Message Flows ⊆ {Pool, Flow Object} x {Pool, Flow Object} (nur zwischen *verschiedenen* Pools erlaubt)

Supporting Elements

Assignments, Categories, Conditions, Entities ∈ String
Event Details ={Cancel Event Details, Conditional Event Details, Compensate Event Details, Error Event Details, Link Event Details, Message Event Details, Signal Event Details, Terminate Event Details, Timer Event Details}
Gates, Inputs, Messages, Outputs, Participants ∈ String
Process = (Flow Objects, Connecting Objects, Swimlanes, Artifacts) [3]
Properties, Roles, Transactions, Web Services ∈ String

Event Details:
Conditional Event Details, Compensate Event Details, Error Event Details, Expressions, Link Event Details, Message Event Details, Timer Event Details ∈ String

Es muss gelten:
In keiner der oben genannten Mengen sind Duplikate erlaubt. (Eindeutigkeit der Element-Identifier.)

Abbildungen

Umsetzung der Attributzuordnungen. Die Benennung der Bildmengen erfolgt analog zur BPMN-Spezifikation. Das Attribut *Name* des BPMN-Elementes *Task* wird beispielsweise zur Bildmenge *Task_Name*.

Diagram

DiagramID: Diagram → Diagram_Id
(Diagram_Id ∈ String.)

[2] Auf eine mathematisch korrekte Definition der Menge *String* wird an dieser Stelle verzichtet. Gemeint ist die Menge aller Zeichenketten.

[3] Die BPMN-Spezifikation sieht das Prozess-Attribut *GraphicalElements* vor. Als Wertebereich ist hier die Menge der Graphical Elements ohne weitere Einschränkungen vorgesehen. Unserer Meinung nach wäre aber eine Beschränkung dahingehend notwendig, dass die Menge *GraphicalElements* maximal ein *Pool*-Element enthalten darf. Dies ergibt sich direkt aus der Definition eines Prozesses: „Each Process may have its own Sub-Processes and would be contained within a Pool“ (BPMN-Spez., S. 32). Es ist also nicht möglich, dass sich ein Prozess über mehrere Pools erstreckt.
Die Definition eines Prozesses als Tupel von Elementen ist – zusätzlich zu der bestehenden Abbildung eines Prozesses auf die Menge der darin enthaltenen Elemente – nicht unbedingt notwendig. Sie entspricht aber eher dem intuitiven Verständnis eines Prozesses als Menge miteinander verbundener Aktivitäten und erleichtert die Darstellung bei der Definition der Änderungsprimitive. Da das Modell hierdurch weder einschränkt wird noch seine Mächtigkeit zunimmt, steht diese Schreibweise in keinem Widerspruch zur BPMN-Spezifikation.

DiagramName: Diagram → Diagram_Name
(Diagram_Name ∈ String)

DiagramPools: Diagram → Diagram_Pools
(Diagram_Pools ⊆ Pools)

DiagramVersion: Diagram → Diagram_Version
(Diagram_Version ∈ String)

BPMN-Elements

BPMNElementID: BPMNElement → BPMNElement_Id
(BPMNElement_Id ∈ String.)

Bedingungen:
BPMNElementID(BPMN Element) = BPMN Element [4]

Graphical Elements

Flow Objects:

FlowObjectName: Flow Object → FlowObject_Name
(FlowObject_Name ∈ String)

Connecting Objects:

ConnectingObjectName: Connecting Object → ConnectingObject_Name
(ConnectingObject_Name ∈ String.)

ConnectingObjectSourceRef: Connecting Object → ConnectingObject_SourceRef
(ConnectingObject_SourceRef ∈ Graphical Elements)

ConnectingObjectTargetRef: Connecting Object → ConnectingObject_TargetRef
(ConnectingObject_TargetRef ∈ Graphical Elements)

ConnectingObjectType: Connecting Object → ConnectingObject_Type [5]
(ConnectingObject_Type ∈ {"Sequence Flow", "Message Flow", "Association"}

Swimlanes:

SwimlaneName: Swimlane → Swimlane_Name
(Swimlane_Name ∈ String)

Artifacts:

Hier nicht betrachtet, da kein Mapping der Artifacts nach BPEL erfolgt.

[4] Jedes BPMN-Element wird in den von uns definierten Mengen durch seine *BPMNElement_Id* repräsentiert. Der Wert der mengeninternen Repräsentation eines Elementes und der seiner BPMNElementID-Abbildung sind also stets gleich.

[5] Während sich unser Modell ansonsten streng an den Vorgaben der BPMN-Spezifikation orientiert, nehmen wir in diesem Punkt eine Erweiterung vor. Die Spezifikation definiert aus uns unbekannten Gründen kein Attribut, das den Typ eines *Connecting Objects* definiert. Wir benötigen allerdings eine Möglichkeit zur Unterscheidung der Untertypen, sehen diese Erweiterung aber auch generell als sinnvoll an.

Flow Objects

Activities:

ActivityType: Activity → Activitiy_ActivityType
(Activity_ActivityType ∈ {"Task", "Sub-Process"})

ActivityStatus: Activity → Activitiy_Status
(Activity_Status ∈ {"None", "Ready", "Active", "Cancelled", "Aborting", "Aborted", "Completing", "Completed"})

ActivityLoopType: Activity → Activitiy_LoopType
(Activity_LoopType ∈ {"None", "Standard", "MultiInstance"})

Gateways:

GatewayType: Gateway → Gateway_GatewayType
(Gateway_GatewayType ∈ {"Exclusive", "Inclusive", "Compley", "Parallel"})

GatewayGates: Gateway → Gateway_Gates
(Gateway_Gates ⊆ Gates)
Je nach Gateway-Typ Einschränkungen bzgl. der Anzahl. Hier nicht ausgeführt.

Events:

EventType: Event → Event_EventType
(Event_EventType ∈ {"Start", "End", "Intermediate"})

Connecting Objects

Sequence Flows:

SequenceFlowConditionType: Sequence Flow → SequenceFlow_ConditionType
(SequenceFlow_ConditionType ∈ {"None", "Expression", "Default"})
(Es gibt Einschränkungen, je nachdem von welchem Typ die Quellaktivität ist.)

∀ n ∈ {Sequence Flow | SequenceFlowConditionType(n) = "Expression"}:
SequenceFlowConditionExpression: Sequence Flow → SequenceFlow_ConditionExpression

Message Flows:

MessageFlowMessageRef: Message Flow → MessageFlow_MessageRef
(MessageFlow_MessageRef ∈ Messages.)

Associations:

Hier nicht weiter betrachtet, da kein Mapping der Associations nach BPEL erfolgt.

Swimlanes

Pools:

Pool → Pool_MainPool
(Pool_MainPool ∈ {"false", "true"})

Pool → Pool_ProcessRef
(ProcessRef ∈ Processes)

Pool → Pool_Lanes
(Pool_Lanes ⊆ Lanes)

Lanes:

Lane → Lane_Lanes
(Lane_Lanes ⊆ Lanes)

Activities

Task:

TaskType: Task → Task_TaskType
(Task_TaskType ∈ {"Service", "Receive", "Send", "User", "Script", "Manual", "Reference", "None"})

∀ n ∈ {Task | Task_TaskType = "Service"}:
TaskInMessageRef: Task → Task_InMessageRef
(Task_InMessageRef ∈ Messages)
TaskOutMessageRef: Task → Task_OutMessageRef
(Task_OutMessageRef ∈ Messages)
TaskImplementation: Task → Task_Implementation
(Task_Implementation ∈ "Web Service", "Other", "Unspecified")

∀ n ∈ {Task | Task_TaskType = "Receive"}:
TaskMessageRef: Task → Task_MessageRef
(Task_MessageRef ∈ Messages)
TaskInstantiate: Task → Task_Instantiate
Task_Instantiate ∈ {"false", "true"}
TaskImplementation: Task → Task_Implementation
(Task_Implementation ∈ {"Web Service", "Other", "Unspecified"})

∀ n ∈ {Task | Task_TaskType = "Send"}:
TaskMessageRef: Task → Task_MessageRef
(Task_MessageRef ∈ Messages)
TaskImplementation: Task → Task_Implementation
(Task_Implementation ∈ {"Web Service", "Other", "Unspecified"})

∀ n ∈ {Task | Task_TaskType = "User"}:
TaskInMessageRef: Task → Task_InMessageRef
(Task_InMessageRef ∈ Messages)
TaskOutMessageRef: Task → Task_OutMessageRef
(Task_OutMessageRef ∈ Messages)
TaskImplementation: Task → Task_Implementation
(Task_Implementation ∈ {"Web Service", "Other", "Unspecified"})

∀ n ∈ {Task | Task_TaskType = "Script"}:
TaskScript: Task → Task_Script
(Task_Script ∈ Scripts)

∀ n ∈ {Task | Task_TaskType = "Reference"}:
TaskReference: Task → Task_Reference
(Task_Reference ∈ Tasks)

Sub-Processes:

Sub-ProcessType: Sub-Process → Sub-Process_SubProcessType
Sub-Process_SubProcessType ∈ {"Embedded", "Reusable", "Reference"})

∀ n ∈ {Sub-Process | Sub-Process_SubProcessType = "Embedded"}:
Sub-ProcessGraphicalElements: Sub-Process → Sub-Process_GraphicalElements
(Sub-Process_GraphicalElements ⊆ GraphicalElements)

∀ n ∈ {Sub-Process | Sub-Process_SubProcessType = "Reusable"}:
Sub-ProcessDiagramRef: Sub-Process → Sub-Process_DiagramRef
(SubProcess_DiagramRef ∈ Diagrams)
Sub-ProcessProcessRef: Sub-Process → Sub-Process_ProcessRef
(SubProcess_ProcessRef ∈ Processes)

∀ n ∈ {Sub-Process | Sub-Process_SubProcessType = "Reference"}:
Sub-ProcessSubProcessRef: Sub-Process → Sub-Process_SubProcessRef
(Sub-Process_SubProcessRef ∈ Sub-Processes)

Supporting Elements

Assignments, Categories, Entities und weitere mehr werden nicht aufgeführt.

Event Details:

EventDetailType: Event Detail→ EventDetail_EventDetailType
EventDetail_EventDetailType ∈ {"Cancel", "Conditional", "Compensate", "Error", "Link", "Message", "Signal", "Terminate", "Timer"}

Expressions:

ExpressionBody: Expression → Expression_ExpressionBody
(Expression_ExpressionBody ∈ Sting)

ExpressionLanguage: Expression → Expression_ExpressionLanguage
(Expression_ExpressionLanguage ∈ Sting)

Messages:

MessageName: Message → Message_Name
(Message_Name ∈ String)

MessageFromRef: Message → Message_FromRef
(Message_FromRef ∈ Participant)

MessageToRef: Message → Message_ToRef

(Message_ToRef ∈ Participant)

Processes:

ProcessName: Process → Process_Name
(Process_Name ∈ String)

ProcessType: Process → Process_ProcessType
(Process_ProcessType ∈ {"None", "Private", "Abstract", "Collaboration"})

ProcessStatus: Process → Process_Status
(Process_Status ∈ {"None", "Ready", "Active", "Cancelled", "Aborting", "Aborted", "Completing", "Complete"})

ProcessGraphicalElements: Process → Process_GraphicalElements
(Process_GraphicalElements ⊆ Graphical Elements)

Operationen:

Neben den gewöhnlichen Mengenoperationen wie Durchschnitt, Vereinigung oder Differenz definieren wir die Hilfsfunktion gI (für getIdentifier), die zu einer Quell- und Zielobjekt-Kombination den zugehörigen FlowObject-Identifier liefert:

gI(*SourceRef*, *TargetRef*) = {BPMNElement | (ConnectingObjectSourceRef(BPMNElement) = *SourceRef*) ∧ (ConnectingObjectTargetRef(BPMNElement) = *TargetRef*)}

5.2.2 Instanzebene

Um ein formales Modell für die Instanzebene entwickeln zu können, das als Grundlage für die Definition unserer Adaptionsprimitive dient, müssen wir zunächst den Zusammenhang zwischen den verschiedenen Prozessdefinitions-„Instanzen“ analysieren.

Ausgangspunkt ist das für BPEL definierte XML-Schema. Entsprechend dieses Schemas kann in BPEL ein konkreter Prozess definiert werden. Diese Prozessdefinition wird in der Ausführungsmaschine für jeden auftretenden Geschäftsvorfall instanziiert. Die Prozessinstanz basiert auf der Prozessdefinition, stellt selbst aber einen Instanzzustand dar, der den aktuellen Ausführungszustand und die momentane Prozessvariablen-Belegung umfasst. Abbildung 5.3 zeigt die drei zu unterscheidenden Instanziierungsebenen.

Modellvoraussetzungen

Die Definition eines Mengenmodells wie für BPMN ist auf Zustandsebene ungeeignet. In BPMN findet die Prozessmodellierung – von Sub-Prozessen abgesehen – grundsätzlich auf einer Hierarchie-Ebene statt. Deshalb lässt sich ein Prozess gut als Tupel von Prozess-Elementen beschreiben.
BPEL ist durch seine XML-Struktur und die Rekursion des *sequence*-Elementes für eine mengenbasierte Schreibweise nicht geeignet, da diese eine Schachtelung nicht bzw. nur unzureichend unterstützt. Davon abgesehen liegt bereits ein formales BPEL-Modell in Form des BPEL-Schemas vor. Hierauf lassen sich Operationen auf Basis von XPath/XQuery-Operatoren definieren, die wir als Grundlage für unsere Instanz-Adaptionsoperationen verwenden können.

Ein Modell, das nur die Prozessdefinitions-Ebene umfasst, ist aber – wie gerade dargestellt – nicht ausreichend. Das Modell muss zusätzlich die Zustände der Prozess-Instanzen umfassen,

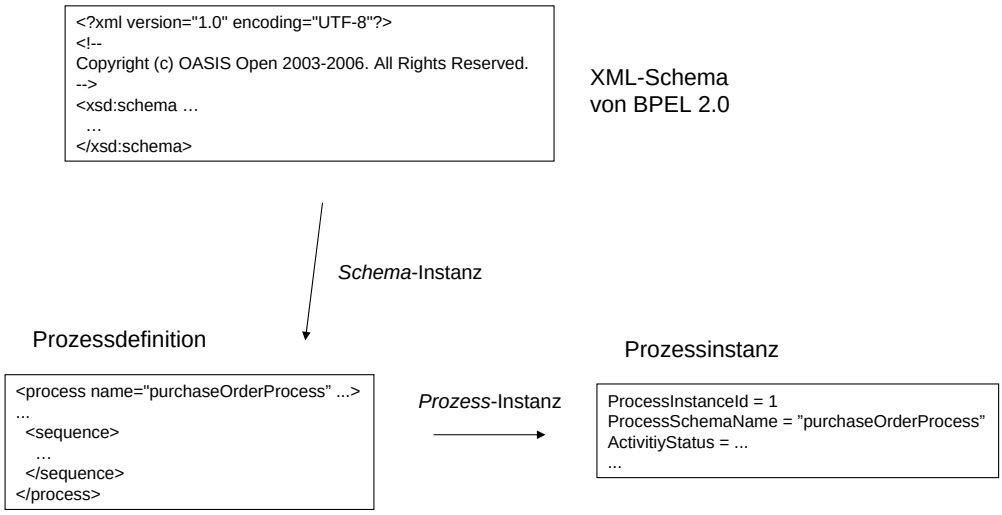


Abbildung 5.3: BPEL-Instanziierung

die sich aus dem Ausführungszustand und den aktuellen Prozessvariablen-Werten ergibt.

Wir definieren deshalb das Instanzmodell als Verknüpfung einer BPEL-Prozessdefinition mit den aktuellen Aktivitätszuständen und Variablenbelegungen.

Als Metamodell für die Prozessdefinition greifen wir auf das in der BPEL-Spezifikation definierte XML-Schema zurück. Wie bereits angesprochen müssen wir allerdings Einschränkungen vornehmen: Wir gehen von einem vollständig blockstrukturierten Modell aus und lassen aus diesem Grund keine *link*-Elemente innerhalb von *flow*-Aktivitäten zu. Das *flow*-Element dient bei uns nur der Modellierung uneingeschränkt parallel ausführbarer Zweige. In Folge dessen ist auch die Verwendung von *source*- und *target*-Elementen als optionale Standard-Elemente jedes Aktivitätstyps nicht erlaubt.

Instanz-Modell

Instance = (InstanceIdentifier, ProcessDefinition, ActivityStatus, VariableValue)

Die Tupelelemente sind hierbei:
InstanceIdentifier := eindeutiger Identifier vom Typ String
ProcessDefinition := BPEL-Prozessdefinition
ActivityStatus := Abbildung einer Aktivität auf ihren aktuellen Ausführungszustand
VariableValue := Abbildung einer Variable auf ihren aktuellen Variablenwert

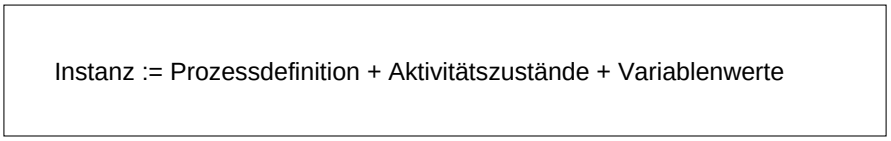


Abbildung 5.4: Definition – Prozess-Instanz

Wir definieren im Folgenden die Abbildungen *ActivityStatus* und *VariableValue*. Hierbei ist *Instances* die Menge aller laufenden Instanzen.

ActivityStatus:

Bildet eine Aktivität auf ihren aktuellen Ausführungszustand ab.

$\forall$ a $\in$ {Instance.ProcessDefinition.Activity}:
ActivityStatus: a/@name $\rightarrow$ Status [6]

Instance.ProcessDefinition.Activity $\in$ {doc("*ProcessDefinition*")//assign $\cup$ ...//compensate $\cup$ ...//compensateScope $\cup$ ...//empty $\cup$ ...//exit $\cup$ ...//extensionActivity $\cup$ ...//flow $\cup$ ...//forEach $\cup$ ...//if $\cup$ ...//invoke $\cup$ ...//pick $\cup$ ...//receive $\cup$ ...//repeatUntil $\cup$ ...//reply $\cup$ ...//rethrow $\cup$ ...//scope $\cup$ ...//sequence $\cup$ ...//throw $\cup$ ...//validate $\cup$ ...//wait $\cup$ ...//while}

Status $\in$ {"None", "Ready", "Active", "Cancelled", "Aborting", "Aborted", "Completing", "Completed"}

VariableValue:

Bildet eine Variable auf ihren aktuellen Wert ab.

VariableValue: Variable $\rightarrow$ Value

- für alle Variablen mit Attribut *messageType*:
Value $\in$ {//variable/@messageType}

- für alle Variablen mit Attribut *type*:
Value $\in$ {//variable/@type}

- für alle Variablen mit Attribut *element*:
Value $\in$ {//variable/@element}

Anmerkungen:

Für eine BPEL-Instanz sind in der BPEL 2.0-Spezifikation keine Ausführungszustände definiert. Mangels Alternativen greifen wir für unser Instanzmodell auf die Prozess-Zustände der BPMN-Spezifikation zurück. Dies sind die oben aufgeführten Zustände *None*, *Ready*, *Active*, *Cancelled*, *Aborting*, *Aborted*, *Completing* und *Completed*. Eine so detaillierte Differenzierung der Aktivitätszustände ist im Rahmen unseres Adaptionskonzeptes bislang zwar nicht notwendig, eventuell müssen zukünftige Erweiterungen aber hierauf zurückgreifen.
Bei der Ausarbeitung unseres Konzeptes gehen wir von den drei Grundzuständen *None*, *Active* und *Completed* aus und überlassen eine differenziertere Betrachtung weiterführenden Arbeiten.

5.3 Umsetzung auf BPMN-Ebene

Nach der Definition der formalen Modelle formulieren wir nun in einem zweiten Schritt sogenannte *Änderungsprimitive* hierauf. Auf diese grundlegenden Änderungsfunktionen werden wir bei der Definition der eigentlichen Änderungsoperationen zurückgreifen.

[6] Eigentlich müsste dem Activity-*Element* selbst ein Zustand zugewiesen werden und nicht dem Aktivitäts-*Namen*. Wir ordnen in unserem Modell aber trotzdem dem Namen der Aktivität einen Ausführungszustand zu. So können wir später bei der Definition der Zugriffsprimitive direkt über Aktivitätsnamen den Zustand der Aktivität ermitteln. Der Zwischenschritt, dass üblicherweise der Aktivitätsnamen als Parameter übergeben wird, dann erst die zum Namen passende Aktivität ausgewählt und schließlich deren Status ermittelt wird, kann so vermieden werden.

5.3.1 Änderungsprimitive auf BPMN-Ebene

Vorab geben wir einige Erläuterungen zur Art der Darstellung:
Um korrekt zu sein, müsste jeweils auch das Diagramm genannt werden, das bearbeitet werden soll. Aus Übersichtlichkeitsgründen verzichten wir hierauf, da wir uns in dieser Betrachtung immer innerhalb eines bestimmten Diagramms bewegen. Im Folgenden sind also immer die dem gerade betrachteten Diagramm zugeordneten Elemente und Abbildungen gemeint.
Ebenso werden die einzelnen Funktionen immer innerhalb des jeweiligen Prozesses definiert. Auf eine vorausgehende, explizit aufgeführte Selektion der betroffenen Elemente wird verzichtet; diese wird vorausgesetzt.
Bei Mengenoperationen wird die neu definierte Menge im Folgenden mit *Bezeichner** angegeben.

AddTasks(*ProcessIdentifier*, {(*TaskIdentifier*, *TaskName*)})

Fügt neue Tasks zur bisherigen Task-Menge des Prozesses *ProcessIdentifier* hinzu. Die Taskattribute werden mit Standardwerten bzw. den als Aufruf-Parameter übergebenen Werten belegt.[7]

Parameter:
ProcessIdentifier ∈ String
TaskIdentifier ∈ String
TaskName ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes

Eindeutigkeit der BPMN Element-Identifier:
TaskIdentifiers := {*TaskIdentifier*} ist duplikatfreie Menge.
BPMNElementID(TaskIdentifiers) ∩ BPMNElementID(BPMNElements) = ∅

Semantik:

Mengenoperationen:
∀ p ∈ {Process | Process_Id(p) = *ProcessIdentifier*}:
p.Tasks* := p.Tasks ∪ p.TaskIdentifiers [8]

Diese Schreibweise wird im Folgenden abgekürzt zu:
Tasks* := Tasks ∪ TaskIdentifiers

Abbildungsdefinitionen:
∀ t ∈ {(*TaskIdentifier*, *TaskName*)}: [9]
ActivityType(t.*TaskIdentifier*) := "Task"
ActivityLoopType(t.*TaskIdentifier*) := "None"
FlowObjectName(t.*TaskIdentifier*) := *TaskName*
ProcessGraphicalElements(*ProcessIdentifier*) := ProcessGraphicalElements(*ProcessIdentifier*) ∪ TaskIdentifiers

[7] Diejenigen Parameter, die für unser Adaptionskonzept nicht von Bedeutung sind, sind nicht aufgeführt.
[8] p.Tasks := {t | t ∈ GraphicalElements(p) ∧ t ∈ Tasks}
[9] Auch diese Schreibweise wird im Folgenden abgekürzt. Die Mengen-, Abbildungs- und Variablenzuweisungsoperationen sind jeweils auf alle Elemente der Parameter-Menge anzuwenden, soweit nicht anders angegeben.

DeleteTasks(*ProcessIdentifier*, {*TaskIdentifier*})

Löscht die als Parameter übergebenen Tasks im Prozess *ProcessIdentifier*.

Parameter:
ProcessIdentifier ∈ String
TaskIdentifier ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
TaskIdentifier ∈ Tasks

Semantik:

Mengenoperationen:
Tasks* := Tasks − TaskIdentifiers

Abbildungsdefinitionen:
ProcessGraphicalElements(*ProcessIdentifier*)* := GraphicalElements(*ProcessIdentifier*) − TaskIdentifiers

SetTaskTypes(*ProcessIdentifier*, {(*TaskIdentifier*, *TaskType*)})

Weist den genannten Tasks den jeweiligen Task-Typ zu.

Parameter:
ProcessIdentifier ∈ String
TaskIdentifier ∈ String
TaskType ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
TaskIdentifier ∈ Tasks
TaskType ∈ {"Service", "Receive", "Send", "User", "Script", "Manual", "Reference", "None"}

Semantik:

Abbildungsdefinitionen:
TaskType(*TaskIdentifier*) = *TaskType*

AddSequenceFlows(*ProcessIdentifier*, {(*SourceRef*, *TargetRef*)})

Fügt jeweils einen neuen Sequence Flow zwischen dem Flow Object *SourceRef* und dem Flow Object *TargetRef* ein.

Parameter:
ProcessIdentifier ∈ String
SourceRef, *TargetRef* ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
SourceRef, *TargetRef* ∈ {Activities, Gateways, Events}

Semantik:

Mengenoperationen:
Sequence Flows* := Sequence Flows ∪ {gI(*SourceRef*, *TargetRef*)}

Abbildungsdefinitionen:
ConnectingObjectsSourceRef(gI(*SourceRef*, *TargetRef*)) := *SourceRef*
ConnectingObjectsTargetRef(gI(S*ourceRef*, *TargetRef*)) := *TargetRef*

DeleteSequenceFlows(*ProcessIdentifier*, {(*SourceRef*, *TargetRef*)})

Löscht jeweils den Sequence Flow zwischen dem Flow Object *SourceRef* und dem Flow Object *TargetRef*.

Parameter:
ProcessIdentifier ∈ String
SourceRef, *TargetRef* ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
SourceRef, *TargetRef* ∈ {Activities, Gateways, Events}

Semantik:

Mengenoperationen:
Sequence Flows* := Sequence Flows − {gI(*SourceRef*, *TargetRef*)}

***PrecedingElement* GetPrecedingElement(*ProcessIdentifier*, *TaskIdentifier*)**

Bestimmt das Flow Object, das dem Task mit der ID *TaskIdentifier* vorangeht und das mit diesem über einen Sequence Flow verbunden ist.

Parameter:
ProcessIdentifier ∈ String
TaskIdentifier ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
TaskIdentifier ∈ Tasks

Semantik:

PrecSequenceFlow := {ConnectingObject_Id | (ConnectingObjectTargetRef(ConnectingObject_Id) = *TaskIdentifier*) ∧ (ConnectingObjectType((ConnectingObject_Id) = "SequenceFlow")}
PrecElement := {FlowObject_Id | ConnectingObjectSourceRef(PrecSequenceFlow) = FlowObject_Id}
RETURN PrecElement

***SucceedingElement* GetSucceedingElement(*ProcessIdentifier*, *TaskIdentifier*)**

Bestimmt das Flow Object, das dem Task mit der ID *TaskIdentifier* nachfolgt und das mit diesem über einen Sequence Flow verbunden ist.

Parameter:
ProcessIdentifier ∈ String
TaskIdentifier ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes

Semantik:

SuccSequenceFlow := {ConnectingObject_Id | (ConnectingObjectSourceRef(ConnectingObject_Id) = *TaskIdentifier*) ∧ (ConnectingObjectType((ConnectingObject_Id) = "SequenceFlow")}
SuccElement := {FlowObject_Id | ConnectingObjectTargetRef(SuccSequenceFlow) = *FlowObject_Id*}
RETURN SuccElement

{*ConditionType*} GetSequenceFlowConditionTypes(*ProcessIdentifier*, {(*SourceRef*, *TargetRef*}))

Liest jeweils den Ausführungsbedingungs-Typ des Sequence Flows von *SourceRef* nach *TargetRef* und weist ihn der Variable *ConditionType* zu.

Parameter:
ProcessIdentifier ∈ String
SourceRef, *TargetRef* ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
SourceRef, *TargetRef* ∈ {Activities, Gateways, Events}

Semantik:

Variablenzuweisung:
∀ t ∈ {(*SourceRef*, *TargetRef*)}:
ConditionType := SequenceFlowConditionType(gI(t))
RETURN {ConditionType}

SetSequenceFlowConditionTypes(*ProcessIdentifier*, {(*SourceRef*, *TargetRef*, *ConditionType*)})

Weist dem Sequence Flow zwischen *SourceRef* und *TargetRef* den Bedingungstyp *ConditionType* zu.

Parameter:
ProcessIdentifier ∈ String
SourceRef, *TargetRef* ∈ String
ConditionType ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
SourceRef, *TargetRef* ∈ {Activities, Gateways, Events}
ConditionType ∈ {"None", "Expression", "Default"}

Semantik:

Abbildungsdefinitionen:
SequenceFlowConditionType(gI(*SourceRef*, *TargetRef*)) := *ConditionType*

{*ConditionExpressionIdentifier*} GetSequenceFlowConditionExpressions(*ProcessIdentifier*, {*SourceRef*, *TargetRef*})

Liest jeweils den Identifier der Ausführungsbedingung des Sequence Flows von *SourceRef* nach *TargetRef* und weist ihn der Variable *ConditionExpression* zu.

Parameter:
ConditionExpressionIdentifier
ProcessIdentifier ∈ String
SourceRef, *TargetRef* ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
SourceRef, *TargetRef* ∈ {Activities, Gateways, Events}
SequenceFlowConditionType(gI(*SourceRef*, *TargetRef*)) = "Expression"

Semantik:

Variablenzuweisung:
∀ t ∈ {(*SourceRef*, *TargetRef*)}:
ConditionExpressionIdentifier := SequenceFlowConditionExpression(gI(t))
RETURN {ConditionExpressionIdentifier}

SetSequenceFlowConditionExpressions(*ProcessIdentifier*, {*SourceRef*, *TargetRef*, *ConditionExpressionIdentifier*})

Weist dem Sequence Flow zwischen *SourceRef* und *TargetRef* die Bedingung mit dem Identifier *ConditionExpressionIdentifier* zu.

Parameter:
ProcessIdentifier ∈ String
SourceRef, *TargetRef* ∈ String
ConditionExpression ∈ String

Vorbedingungen:
ProcessIdentifier ∈ Processes
SourceRef, *TargetRef* ∈ {Activities, Gateways, Events}
ConditionExpression ∈ Expressions
SequenceFlowConditionType(gI(*SourceRef*, *TargetRef*)) = "Expression"

Semantik:

Abbildungsdefinitionen:
SequenceFlowConditionExpression(gI(*SourceRef*, *TargetRef*)) := *ConditionExpression*

5.3.2 Umsetzung der Änderungsoperationen

BPMN-Änderungsoperationen werden auf der Ebene definiert, auf der sie der Benutzer auch von Hand vornehmen würde. Dieser hat beispielsweise die Möglichkeit einen Task einzufügen – deshalb wird die Änderungsoperation *Serielles Einfügen eines Tasks* definiert. Das generelle Einfügen einer Aktivität oder – noch allgemeiner – eines BPMN-Elements kann selbst nicht definiert werden, sondern ergibt sich aus der Summe der Einzeloperationen.

A. Serielles Einfügen eines Tasks

Wir betrachten exemplarisch nur das Einfügen eines Tasks vom Typ *None*, also eines leeren Tasks. Zur Behandlung anderer Task-Typen wären teilweise umfangreichere Änderungsoperationen auf dem BPMN-Modell notwendig. Eine Umsetzung würde zusätzliche Änderungsprimitive erfordern, beispielsweise zur Einbindung von Message Flows, aber keinen wissenschaftlichen Zugewinn bedeuten. Aus diesem Grund verzichten wir auf eine weitere Ausführung.

Tasks vom TaskType *None*

Erläuterung:

Es wird vorausgesetzt, dass das alte Prozess-Schema vorab kopiert und ihm die ID *NewProcessIdentifier* zugewiesen wird. Alle Änderungen erfolgen auf dem neuen Prozess-Schema. Diese Voraussetzung gilt für alle betrachteten Änderungsoperationen.

Der Algorithmus fügt zunächst ein neues Task-Element in den angegebenen Prozess ein. Ist für die Eingangskante des späteren Nachfolgerknotens eine *FlowCondition* spezifiziert, so wird diese ausgelesen und gespeichert. Die Kante zwischen dem Vorgängerknoten und dem Nachfolgerknoten wird gelöscht und durch zwei Kanten ersetzt, die den neu eingefügten Task in den Prozessablauf integrieren. Die *FlowCondition* der gelöschten Kante wird der neuen Eingangskante des eingefügten Tasks zugewiesen.

Aufruf:

TaskSerialInsert(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*, *TaskName*, *PredecessorElementIdentifier*, *SuccessorElementIdentifier*) [10]

Parameter:

CurrentProcessIdentifier ∈ String
NewProcessIdentifier ∈ String
TaskIdentifier ∈ String
TaskName ∈ String
PredecessorElementIdentifier ∈ String
SuccessorElementIdentifier ∈ String

[10] Wie schon bei der Definition der Änderungsprimitive haben wir konzeptionell nicht entscheidende Task-Attribute aus Übersichtlichkeitsgründen weggelassen. Beispielhaft für alle weiteren Attribute wird hier die Behandlung des *TaskName*-Attributes gezeigt.

Vorbedingungen:

– *CurrentProcessIdentifier* ∈ Processes
– *NewProcessIdentifier* ∈ Processes
– *PredecessorElementIdentifier* ∈ Tasks
– *SuccessorElementIdentifier* ∈ Tasks
– Der Nachfolgerknoten (*SuccessorElementIdentifier*) ist direkter Nachfolger des Vorgängerknotens (*PredecessorElementIdentifier*).

Semantik:

begin

string incomingSeqFlowCondType
string incomingSeqFlowCondExpression

AddTasks(*NewProcessIdentifier*, (*TaskIdentifier*, *TaskName*))
SetTaskTypes(*NewProcessIdentifier*, (*TaskIdentifier*, None))
incomingSeqFlowCondType := GetSequenceFlowConditionType(*NewProcessIdentifier*, (*PredecessorElementIdentifier*, *SuccessorElementIdentifier*))

if incomingSeqFlowCondType = "Expression" **then**
  incomingSeqFlowCondExpression := GetSequenceFlowConditionExpression(*NewProcessIdentifier*, (*PredecessorElementIdentifier*, *SuccessorElementIdentifier*))
else
  incomingSeqFlowCondExpression := NULL
end if

DeleteSequenceFlow(*NewProcessIdentifier*, (*PredecessorElementIdentifier*, *SuccessorElementIdentifier*))
AddSequenceFlows(*NewProcessIdentifier*, {(*PredecessorElementIdentifier*, *TaskName*),(*TaskName*, *SuccessorElementIdentifier*)})
SetSequenceFlowConditionType(*NewProcessIdentifier*, (*PredecessorElementIdentifier*, *TaskIdentifier*, incomingSeqFlowCondType))

if incomingSeqFlowCondType = "Expression" **then**
  SetSequenceFlowConditionExpression(*NewProcessIdentifier*, (*PredecessorElementIdentifier*, *TaskIdentifier*, incomingSeqFlowCondExpression))
end if

end

Anmerkung:

Die alte Kantenbedingung zwischen A und C wird nach Einfügen von B zur Bedingung der A–B Kante. Dies entspricht am ehesten der Semantik der Spezifikation, die angibt, dass der gedachte Token erst die Quellaktivität verlässt, wenn die Bedingung der ausgehenden Kante als wahr ausgewertet wurde. Bei einer Verzweigung wird die Entscheidung über den zu nehmenden Pfad also schon in der Quellaktivität getroffen. [11]

[11] „This means that the condition expression must be evaluated before a Token can be generated and then leave the source object to traverse the Flow. The conditions are usually associated with Decision Gateways, but can also be used with activities.“ (BPMN-Spez., S. 98)

B. Serielles Löschen eines Tasks

Erläuterung:

Der gezeigte Löschalgorithmus ist auf Tasks beliebigen Typs anwendbar.
Er bestimmt zuerst den Vorgänger- und den Nachfolgerknoten des zu löschenden Tasks und liest die FlowCondition der eingehenden Kante aus. Anschließend wird der Task gelöscht, ebenso wie seine ein- und ausgehenden Kanten. Zur Wiederherstellung eines korrekten Kontrollflusses wird eine Kante zwischen Vorgänger- und Nachfolgerknoten angelegt und mit der zuvor ausgelesenen FlowCondition belegt.

Aufruf:

TaskSerialRemove(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *TaskIdentifier*)

Parameter:

CurrentProcessIdentifier ∈ String
NewProcessIdentifier ∈ String
TaskIdentifier ∈ String

Vorbedingungen:

–*CurrentProcessIdentifier* ∈ Processes
– *NewProcessIdentifier* ∈ Processes
– Es darf keine Referenz auf diesen Task existieren. (Kein Task vom Typ *Reference* mit dem Attribut TaskRef = *TaskIdentifier*.)

Semantik:

begin

string predElementIdentifier
string succElementIdentifier
string incomingSeqFlowCondType
string incomingSeqFlowCondExpression

predElementIdentifier := GetPrecElement(*NewProcessIdentifier*, *TaskIdentifier*)
succElementIdentifier := GetSuccElement(*NewProcessIdentifier*, *TaskIdentifier*)

incomingSeqFlowCondType := GetSequenceFlowConditionType(*NewProcessIdentifier*, (predElementIdentifier, *TaskIdentifier*))

if incomingSeqFlowCondType = "Expression" **then**
  incomingSeqFlowCondExpression := GetSequenceFlowConditionExpression(*NewProcessIdentifier*, (predElementIdentifier, *TaskIdentifier*))
else
  incomingSeqFlowCondExpression := NULL
end if

DeleteSequenceFlow(NewProcessIdentifier, (predElementIdentifier, *TaskIdentifier*))
DeleteSequenceFlow(NewProcessIdentifier, (*TaskIdentifier*, succElementIdentifier))
DeleteTasks(*NewProcessIdentifier*, *TaskIdentifier*)

AddSequenceFlows(*NewProcessIdentifier*, (predElementIdentifier, succElementIdentifier))

SetSequenceFlowConditionType(*NewProcessIdentifier*, (predElementIdentifier, succElementIdentifier, incomingSeqFlowCondType))

if incomingSeqFlowCondType = "Expression" **then**
SetSequenceFlowConditionExpression(*NewProcessIdentifier*, predElementIdentifier, succElementIdentifier, incomingSeqFlowCondExpression))
end if

end

Anmerkungen:

Beim Löschen eines Task vom Typ *Service*, *Receive*, *Send* oder *User* ist zu beachten, dass der mit ihm verknüpfte Message Flow in der Folge keinem Task mehr zugeordnet sind. Dieser sollte also entweder einem neuen Task zugewiesen oder aber gelöscht werden.
Man könnte einen existierenden Message Flow auch automatisch zusammen mit dem entsprechenden Task löschen, müsste dann aber wiederum beachten, dass der Partnertask ebenfalls gelöscht, sein Typ geändert oder ein neuer Message Flow angelegt wird. Da wir uns auf die Betrachtung eines Prozesses beschränken, haben wir uns für das separate Löschen von Message Flows entschieden.

5.4 Umsetzung auf BPEL-/Instanzebene

Im vorangehenden Kapitel haben wir die Umsetzung der Änderungsoperationen auf BPMN-Ebene gezeigt. Im Folgenden werden wir darauf eingehen, welche Adaptionsfunktionalität auf BPEL- bzw. Instanzebene vorhanden sein muss. Wir sehen hierbei das BPEL-Prozessschema als Bestandteil einer Instanz an und sprechen deshalb im Allgemeinen auch dann von der *Instanzebene*, wenn es um Analysen auf der BPEL-Prozessdefinition geht.

5.4.1 Herausforderungen

Die Prozess-Schema-Änderungen werden auf BPMN-Ebene vorgenommen. Das neue Prozess-Schema wird dort vor der Übersetzung nach BPEL auf Korrektheit überprüft. Wir gehen deshalb davon aus, dass die BPEL-Repräsentation korrekt ist. Tiefergreifende strukturelle Vorbedingungsüberprüfungen, wie sie z.B. bei ADEPT vorgenommen werden, sind deshalb bei unserem Konzept nicht nötig.

Es treten aber an anderer Stelle Schwierigkeiten auf. Dies betrifft zum einen die Instanz-Migration. In unserem Konzept werden bei einer Einfügeoperation die ausgewählten Instanzen zunächst testweise auf das neue Prozess-Schema migriert. Erst danach werden auf den neu erzeugten Instanzen die Bedingungen überprüft, die Voraussetzung sind für eine korrekte und sinnvolle Migration. Zum einen betrifft dies den Ausführungszustand der jeweiligen Instanz. Auch die Überprüfung der Datenfluss-Korrektheit erfolgt instanzspezifisch. Sind die Migrationsvoraussetzungen nicht erfüllt, wird die neue Instanz verworfen und die alte Instanz, die auf dem bisherigen Prozess-Schema basiert, weitergeführt. (Änderungsoperationen, bei denen ein Prozesselement gelöscht wird, unterscheiden sich von dem dargestellten Vorgehen dahingehend, dass der Ausführungszustand der Instanzen *vor* der testweisen Migration bestimmt wird.)

Die Überprüfung des Ausführungsstatus einer Instanz und die Datenflussüberprüfung sind

gleichzeitig die anderen beiden großen Herausforderungen in dem von uns entwickelten Konzept. Auf Basis des für die Instanzmigration entwickelten Vorgehens ist es uns gelungen, die Bestimmung des Ausführungszustandes sehr einfach zu halten.
Für die Datenflussanalyse auf den laufenden Instanzen ist allerdings ein recht umfangreiches Verfahren notwendig.

Für die drei genannten Punkte werden in den folgenden Abschnitten die hiermit verbundenen Probleme aufgezeigt und die von uns entwickelten Lösungen vorgestellt. Der Schwerpunkt der Betrachtung lag in dieser Arbeit auf der Datenflussanalyse. Dies ist der Grund, weshalb die beiden anderen Aspekte nicht in allen Details analysiert werden konnten.

5.4.2 Instanzen-Migration

Bei der Instanzmigration geht es darum, den Zustand einer laufenden Prozess-Instanz vollständig auf eine neue Instanz, die auf dem geänderten Prozess-Schema basiert, zu übertragen.
Der Zustand einer Instanz ist ihr momentaner Prozess-Ausführungszustand zusammen mit den aktuellen Variablenwerten. Der Ausführungszustand wiederum ergibt sich implizit aus den Ausführungszuständen der Prozesselemente dieser Instanz.

Eine BPEL-Instanz kann problemlos auf eine andere Instanz migriert werden, sofern diese auf dem gleichen Prozess-Schema basiert. Hierfür genügt es, alle Aktivitäts-Zustände und alle Variablenwerte auf die Elemente der neuen Instanz zu kopieren. Komplizierter wird das Vorgehen bei einer Migration auf eine Instanz, die auf einem abgeänderten Schema basiert. Hier werden die alten Aktivitätszustände und Variablenwerte so weit es geht kopiert, es sind aber zusätzliche Anpassungsoperationen notwendig.

Wir betrachten zunächst die Variablenmigration. Diese ist auch bei geänderten Prozess-Schemata recht einfach zu handhaben. Abgesehen von dem Standardfall, dass die alte Variable in der neuen Prozessdefinition auch enthalten ist, können zwei weitere Fälle auftreten:
– In der geänderten Prozessdefinition ist eine Variable definiert, die in der alten nicht vorhanden war. In diesem Fall bleibt die Variable in den neuen Instanzen zunächst in einem nicht initialisierten Zustand, bis sie im weiteren Prozessverlauf mit einem Wert belegt wird.
– Der Wert einer Variable, die im neuen Prozess-Schema nicht mehr enthalten ist, wird einfach gelöscht.
Eventuelle Probleme in Bezug auf Datenflussabhängigkeiten, die durch diese Operationen beeinflusst werden, werden durch die Datenflussanalyse entdeckt und behandelt. (Siehe hierzu Kapitel 5.4.4.)

Die Statusanpassung der Aktivitäten gestaltet sich komplizierter. Wie wir in Kapitel 2.3.3, *Übersetzung von BPMN nach BPEL*, ausgeführt haben, ist es bei der Übersetzung von BPMN nach BPEL möglich, dass auf BPMN-Ebene modellierte Elemente wegfallen (z.B. schließende Gateways). Ebenso können auf BPEL-Ebene neue Elemente hinzukommen (z.B. das *sequence*-Element als strukturierte Aktivität).

Für die korrekte Behandlung des Ausführungszustands bei der Migration sind mehrere Fälle zu unterscheiden:

1. Einfügen einer Aktivität im Prozessablauf *nach* dem aktuellen Ausführungsstatus

Dieser Fall ist unkritisch. Wird eine einzelne einfache oder strukturierte BPEL-Aktivität *nach* dem aktuell aktiven Element eingefügt, wird der Zustand der neuen Aktivität auf den Default-Zustand *None* gesetzt.

2. Einfügen einer einzelnen Aktivität im Prozessablauf *vor* dem aktuellen Ausführungsstatus

Wird eine einzelne einfache Aktivität *vor* der gerade ausgeführten Aktivität eingefügt, ist ihr Zustand zunächst der Default-Zustand. Dieser muss aber an den schon erreichten Instanzzustand angepasst werden, damit nicht mitten im Bereich der schon ausgeführten Aktivitäten eine neue Aktivität im Ausgangszustand existiert. Der Prozesszustand wäre damit nicht mehr konsistent. Wurde die Nachfolgeaktivität der neu eingefügten Aktivität bereits ausgeführt, wird auch die neue Aktivität in den Zustand *Completed* versetzt.
Dies geschieht, obwohl sie nie ausgeführt wurde. Eine solche Instanz wird in der nachfolgenden Zustands-Prüfung als nicht migrierbar erkannt. Hierzu muss die Instanz aber zunächst testweise migriert werden.

Die Schwierigkeit liegt in der Ermittlung der Nachfolgeaktivität, deren Status den der eingefügten Aktivität bestimmt.
Die aktuellen Aktivitätszustände sind normalerweise innerhalb der Workflow-Engine festgehalten und können über Schnittstellen zugegriffen werden. Ein größeres Problem stellt aber die Frage dar, welches überhaupt die Nachfolgeaktivität ist, die betrachtet werden muss.
Auf BPMN-Ebene müssen im Rahmen der Änderungsoperation die Vorgänger- und die Nachfolgeaktivität als Attribute spezifiziert werden, um die gewünschte Stelle im Prozess-Schema zu bestimmen, an der die neue Aktivität eingefügt werden soll. Wie schon angesprochen erfolgt allerdings keine 1:1-Übersetzung der BPMN-Elemente in BPEL-Elemente. Auf BPEL-Ebene können sowohl Elemente hinzukommen, wie auch wegfallen. Es ist also nicht möglich, zum Zweck der Statusanpassung einfach auf die Nachfolgeaktivität der BPMN-Ebene zurückzugreifen.

Im hier betrachteten Fall, wie auch im zuvor genannten, muss direkt auf BPEL-Ebene die nachfolgende Aktivität bestimmt werden, wobei umgebende Elemente wie z.B. *sequence*- oder *if*-Aktivitäten ebenfalls als Nachfolgeaktivitäten gelten.[12] Der Status der Nachfolgeaktivität gibt dann den Status der eingefügten Aktivität vor. Ist die Nachfolgeaktivität eine einfache BPEL-Aktivität und nimmt sie den Ausführungszustand *None* an, wird der Status der neuen Aktivität ebenfalls auf *None* gesetzt. Das gleiche passiert in dem Fall, dass eine strukturierte Aktivität die Nachfolgeaktivität ist und diese noch nicht beendet wurde. Unter dem Gesichtspunkt des Prozessstatus ist die Instanz damit migrierbar.
Hat die Ausführung der einfachen Nachfolgeaktivität schon begonnen, befindet sie sich also z.B. im Zustand *Active* oder *Completed*, muss implizit die vorangehende Aktivität schon ausgeführt worden sein. Diese bekommt also den Status *Completed* zugewiesen. Gleiches gilt für den Fall, dass die umgebende, strukturierte Nachfolgeaktivität schon beendet wurde. Wie wir im nächsten Kapitel erläutern werden, ist eine endgültige Migration dann nicht mehr sinnvoll und wird deshalb abgelehnt.

Auch wenn wir den Ausführungen zur Prozesszustands-Überprüfung etwas vorgreifen, möchten wir an dieser Stelle kurz eine Besonderheit aufzeigen, die unser Vorgehen mit sich bringt. Aufgrund der unterschiedlichen Art und Weise der Prozessdefinition sind die Migrations-Voraussetzungen, was den Ausführungszustand betrifft, auf BPEL-Ebene strikter zu fassen, als es auf BPMN-Ebene nötig erscheint. Wir gehen davon aus, dass eine Aktivität eingefügt werden kann, solange ihre Nachfolgeaktivität noch nicht gestartet wurde. In Abbildung 5.5 nimmt auf BPMN-Ebene das schließende Gateway die Rolle der Nachfolgeaktivität von C ein.

Auf BPMN-Ebene betrachtet könnte Task C solange eingefügt werden, bis das schließende Gatewayelement gestartet wird. Auf BPEL-Ebene ist allerdings das *sequence*-Element die Nachfolgeaktivität von Task C. Dort kann der Task also schon nicht mehr eingefügt werden, sobald

[12] Dieses Vorgehen entspricht bei genauerer Betrachtung einer Überführung der BPMN-Welt mit Vorgänger- und Nachfolgeaktivität in die BPEL-Welt mit öffnenden und schließenden XML-Tags.

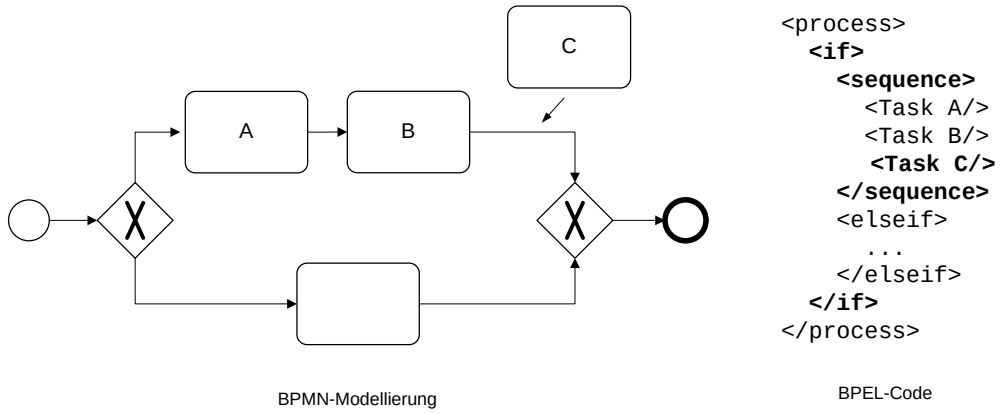


Abbildung 5.5: Striktere Migrations-Bedingung auf BPEL-Ebene

die *sequence*-Aktivität beendet wurde.
Möchte man diese Einschränkung umgehen bliebe nur die Möglichkeit, das *sequence*-Element auf den Status *Active* zurückzusetzen. Ein solches Vorgehen wäre bei manchen Aktivitätstypen ohne größere Auswirkungen an anderer Stelle möglich, wir sehen in dieser Arbeit dennoch davon ab.

3. Einfügen einer Aktivität einschließlich umgebender strukturierter Aktivitäten

Beim Einfügen einer einfachen Aktivität kann es vorkommen, dass bei der BPEL-Übersetzung zusätzliche strukturelle Aktivitäten erzeugt werden müssen, die die einfache Aktivität umschließen. Abbildung 5.6 zeigt ein Beispiel.

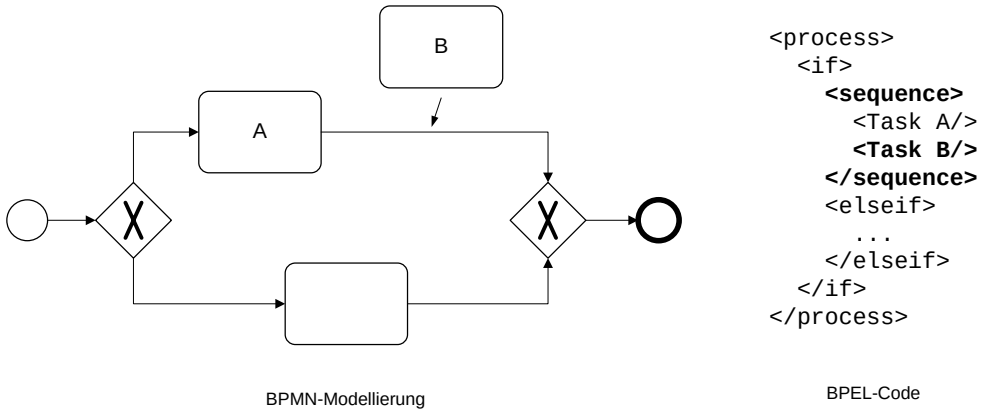


Abbildung 5.6: Einfügen einer Aktivität erzeugt zusätzliche strukturierte Aktivität

Wenn auf BPMN-Ebene Task B eingefügt wird, wird bei der BPEL-Übersetzung automatisch ein *sequence*-Element erzeugt, das die sequentiell auszuführenden Aktivitäten A und B umschließt. Wie der Status der Aktivität B wird auch der Status des *sequence*-Elementes zunächst auf den Default-Zustand gesetzt, muss aber anschließend an den Ausführungszustand der Instanz angepasst werden. Wurde beispielsweise Aktivität A gerade erst beendet, bleibt der Ausgangszustand der Aktivität B erhalten, der Status des beide Aktivitäten umgebenden *sequence*-Elementes muss aber auf *Active* gesetzt werden.

Je nach Prozessstruktur sind in einem solchen Fall nicht nur die unmittelbar umschließende Aktivität betroffen, sondern auch weitere umgebende. Hier bleibt für kommende Arbeiten die Aufgabe, zu untersuchen, wie genau die Statusanpassung dieser strukturieren Aktivitäten zu erfolgen hat und wie weitreichend die diesbezüglichen Folgen bei den jeweiligen Änderungsoperationen sein können.
Wir klammern in dieser Arbeit den Fall 3 aus und beschränken uns auf das Einfügen von einfachen Aktivitäten, ohne dass zusätzliche strukturierte Aktivitäten erzeugt werden müssen. Um dies zu gewährleisten betrachten wir nur Einfügeoperationen zwischen zwei einfache Aktivitäten und Löschoperationen an gleicher Stelle. Die Fälle, die bei der Durchführung von Löschoperationen auftreten können, analysieren wir im Folgenden.

4. Löschen einer noch nicht aktivierten Aktivität

Wird eine noch nicht aktivierte Aktivität aus der Prozessdefinition gelöscht, führt dies zu keinen Schwierigkeiten auf Instanzebene. Der Ausführungszustand der Aktivität wird einfach verworfen, da er keinem Element mehr zugeordnet werden kann. Inkonsistenzen hinsichtlich des Ausführungszustandes der Gesamtinstanz treten nicht auf.

Ein anderer Fall, das Löschen einer schon beendeten oder laufenden Aktivität, spielt bei der Migration keine Rolle, da bei Löschoperationen die testweise Migration erst nach der Zustandsprüfung durchgeführt wird. Die Zustandsprüfung würde in diesem Fall signalisieren, dass eine Migration aufgrund des bereits fortgeschrittenen Ausführungsstatus der Instanz nicht vorgenommen werden kann.

5. Löschen einer Aktivität inklusive der umgebenden strukturierten Aktivität

Wie wir gerade gesehen haben, ist das direkte Löschen von laufenden oder schon beendeten Aktivitäten aus einer Prozessinstanz nicht möglich. Es kann aber der Fall eintreten, dass das Löschen einer noch nicht ausgeführten Aktivität Auswirkungen auf die umgebende strukturierte Aktivität hat und diese ebenfalls gelöscht werden muss. Die strukturierte Aktivität kann hierbei schon gestartet sein, ohne dass die Instanz durch den Löschvorgang in einen inkonsistenten Zustand gerät.

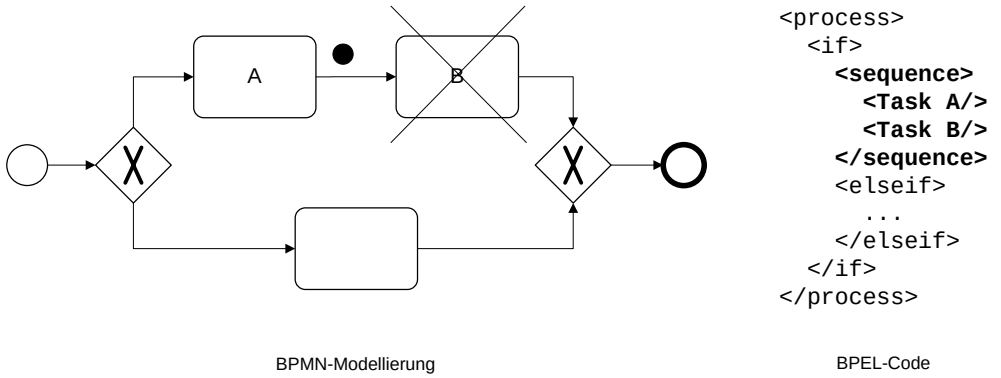


Abbildung 5.7: Löschen einer Aktivität – Auswirkungen auf umgebende Elemente

Abbildung 5.7 verdeutlicht dies an einen Beispiel: Aktivität A und B bilden einen Zweig einer Verzweigung. Auf BPEL-Ebene umgibt ein *sequence*-Element die beiden Tasks.
Task B soll nun gelöscht werden. Der Status der Instanz vor Durchführung der Änderungsoperation ist der, dass Task A ausgeführt, Task B aber noch nicht gestartet wurde. Das Löschen

der Aktivität B ist also möglich.
Durch das Löschen von Task B wird gleichzeitig das *sequence*-Element mit entfernt, da es nicht mehr benötigt wird. Sein Zustand vor der Migration war *Active*, das *sequence*-Element war gestartet, aber noch nicht beendet worden. Nach Durchführung der Änderungsoperation ist der Prozess in dem Status, dass das *if*-Element gestartet und Aktivität A beendet ist. Das *sequence*-Element ist nicht mehr vorhanden und hat damit keinen Status mehr. Dies stellt kein Problem dar, der Zustand des Prozesses ist nach wie vor konsistent. Der nächste Schritt der Workflow-Engine würde nun darin bestehen, zu erkennen, dass das einzige Element des *if*-Zweiges beendet ist und damit auch das *if*-Element selbst den Zustand *Completed* annehmen kann.

Die von uns aufgeführten Fälle müssen in zukünftigen Arbeiten formal auf ihre Vollständigkeit und Korrektheit überprüft werden. Wir haben in dieser Arbeit nur die Migration in Bezug auf Einfüge- und Löschvorgänge einfacher Aktivitäten umgesetzt. Eine Erweiterung auf weitere Änderungsoperationen ist ebenfalls notwendig, wobei die von uns ausgearbeiteten Operationen *Einfügen* und *Löschen* als Basisoperationen dienen können.

Ein weiterer Aspekt hinsichtlich der korrekten Durchführung einer Instanzmigration betrifft die Frage, in welchem Ausführungszustand sich eine Instanz befinden darf, die migriert werden soll. Auf diese Frage gehen wir im nachfolgenden Kapitel ein.

5.4.3 Ausführungszustandsüberprüfung

Vor der endgültigen Migration einer Prozessinstanz muss geprüft werden, ob der aktuelle Prozesszustand eine Migration zulässt. Dies betrifft zum einen systemspezifische Aspekte, wie z.B. die Frage, ob man einen Prozess in jedem Zustand einfach anhalten kann. Eintreffende Nachrichten könnten dann eventuell nicht korrekt behandelt werden, Probleme mit prozessinternen Timern könnten auftreten usw. Diese Aspekte sind nicht Betrachtungsgegenstand dieser Arbeit.

Wir behandeln nur Fragen in direktem Zusammenhang mit der Prozessausführung. Eine Prozessinstanz, in die eine Aktivität B zwischen zwei Aktivitäten A und C eingefügt werden soll, kann nicht mehr auf das neue Prozess-Schema migriert werden, wenn Aktivität C bereits ausgeführt wurde. Beziehungsweise wäre eine Migration zwar möglich, aber die Änderungsoperation würde sich in der vorhandenen Instanz nicht auswirken und es wäre bei einer nachträglichen Analyse des Prozesses ohne zusätzliche Verfahren nicht nachvollziehbar, welche Aktivitäten wirklich ausgeführt wurden und welche erst nachträglich eingefügt worden sind. Aus diesen Gründen haben wir ein solches Vorgehen ausgeschlossen.
Die Notwendigkeit dieses strikten Vorgehens erschließt sich noch eher in Hinblick auf mögliche Aggregate von Änderungsoperationen. Würde man hier Änderungen vor und nach der aktuell ausgeführten Aktivität zulassen, würde dies zu erheblichen Schwierigkeiten bei der Datenflussüberprüfung auf Instanzebene führen. Die Entwicklung neuer Verfahren für diesen Fall wäre notwendig.

Einen Spezialfall stellen Änderungsoperationen innerhalb von Schleifen dar. Hier wäre es möglich, dass die eingefügte Aktivität bei einem erneuten Schleifendurchlauf zur Ausführung kommt, obwohl sie auf einen früheren Schleifendurchlauf bezogen „zu spät“ eingefügt wurde. Wir haben diese Möglichkeit in unserem Konzept ausgeklammert. Für Änderungsoperationen auf Schleifen wäre ein speziell angepasstes Migrationsvorgehen notwendig, dessen Ausarbeitung wir zukünftigen Arbeiten überlassen.
Denkbar wäre hier zum Beispiel ein Zurückstellen der Änderungsoperation, bis der aktuelle Schleifendurchlauf beendet ist [Rei00], oder eben doch das Zulassen der Durchführung von Änderungsoperationen in jedem Prozesszustand. In letzterem Fall müssten die negativen Auswirkungen dieses Vorgehens gegen die Vorteile abgewogen werden, was aber erst dann sinnvoll durchgeführt werden kann, wenn unser Konzept einmal auf Aggregationen von Änderungsope-

rationen erweitert worden ist.

Bisher ebenfalls nicht berücksichtigt ist die mögliche Zurücksetzung von *scope*-Umgebungen durch *Compensation Handlers*. Auch in diesem Zusammenhang kann es vorkommen, dass Änderungsoperationen aufgrund des fortgeschrittenen Prozessstatus abgelehnt wurden, später rückblickend aber doch sinnvoll hätten zum Einsatz kommen können, da der Bereich, in dem die Operation erfolgen sollte, zurückgesetzt wurde.

Aufgrund unseres Vorgehens bei der testweisen Instanzenmigration inklusive der Statusanpassung lässt sich die Zustandsprüfung jetzt recht einfach durchführen. Es müssen Einfüge- und Löschoperationen unterschieden werden.
Einfügeoperationen lassen sich durchführen, solange sich die fragliche Aktivität nach dem testweisen Einfügen und der temporären Migration im Zustand *None* befindet. Nimmt sie den Ausführungszustand *Completed* an, ist eine endgültige Migration nicht möglich.[13]
Bei **Löschoperationen** wird die Zustandsprüfung *vor* Durchführung der probeweisen Migration vorgenommen. Dies ist einfach nachvollziehbar, da die Aktivität im neuen Prozess-Schema und damit in der migrierten Instanz nicht vorhanden ist und somit ihr Zustand dort nicht ermittelt werden kann. Zur Durchführung von Löschoperationen darf die fragliche Aktivität noch nicht gestartet sein, d.h. sie muss sich im Zustand *None* befinden.[14]

5.4.4 Datenfluss-Analyse

Neben der Forderung nach einem migrierbaren Ausführungszustand der Prozessinstanz ist die Korrektheit des Datenflusses die zweite wichtige Bedingung für eine erfolgreiche Migration. Dies liegt aber zunächst nicht auf der Hand.
Weder die BPMN- noch die BPEL-Spezifikation sehen tiefergreifende Korrektheitskriterien bzgl. des Datenflusses vor. In anderen Workflow-Modellen teilweise ein bedeutender Aspekt, werden Datenfluss-Aspekte in expliziter Form hier gar nicht erwähnt. Als einziges Korrektheitskriterium hinsichtlich des Datenflusses sieht die BPEL-Spezifikation für das Lesen einer Variable notwendigerweise eine vorangehende Initialisierung vor. Dieses Kriterium wird erst zur Laufzeit überprüft, also jeweils dann, wenn eine Variable gelesen werden soll. Wurde die Variable zuvor nicht beschrieben, wird der Fehler *bpel:uninitializedVariable* ausgelöst. Die Beachtung der semantischen Korrektheit zur *Definitions*zeit wird dem Benutzer überlassen.

Das letztgenannte Prinzip wird grundsätzlich auch bei unserem Adaptionskonzept beibehalten. Anders verhält es sich zur Laufzeit auf Instanzebene. Hier sollte dem Benutzer zumindest eine Hilfestellung geboten werden, da er keinen direkten Einblick in die unmittelbaren Auswirkungen seiner geplanten Änderungsoperationen auf die einzelnen Prozessinstanzen hat. Es könnte sonst vorkommen, dass bei einer Prozessänderung ein aus Benutzersicht (auf BPMN-Ebene) semantisch vollkommen korrektes Prozessmodell auf Instanzebene ein unerwünschtes Verhalten zeigt. Beispielsweise könnte eine unbeschriebene Variable gesendet werden, da die vorhergehende Schreibaktivität verschoben wurde. Ein Beispiel hierfür zeigt Abbildung 5.8.

Der obere Bereich zeigt das ursprüngliche Prozess-Schema. In der markierten Aktivität wird eine Variable geschrieben und ihr Wert kurz darauf als Inhalt einer Nachricht gesendet. Der Prozess ist vom Datenfluss her gesehen korrekt modelliert. Zum Zeitpunkt der Betrachtung befindet sich eine Instanz in dem Zustand, dass als nächster Schritt die Aktivität *Sonstige Operation* auszuführen ist.
Wir nehmen jetzt eine Änderung am Prozess-Schema vor und verschieben den Task *Variable schreiben* im Prozessablauf nach vorne. Anschließend migrieren wir die Instanz auf das neue

[13] In einem anderen Zustand kann sich die Aktivität nach der testweisen Migration nicht befinden.
[14] Man könnte auch hier die Bedingung etwas weicher fassen, beispielsweise den Zustand *Ready* erlauben und gegebenenfalls Rücksetzungen vornehmen. Dies erfordert weiterreichende Betrachtungen auf anderer Ebene. Wir begnügen uns mit dem strikten Fall.

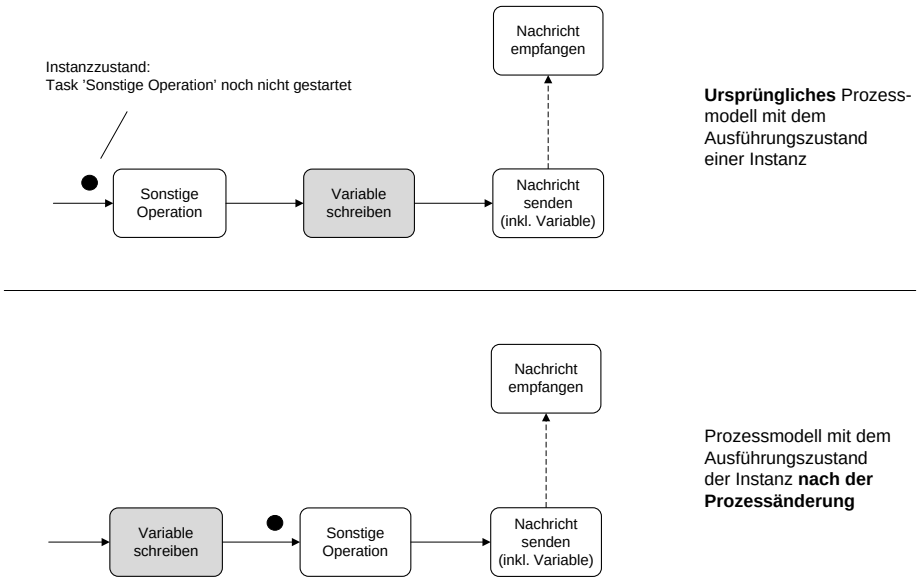


Abbildung 5.8: Beispiel: Senden einer leeren Variablen

Schema. Das Prozess-Schema selbst ist wiederum unter Datenflussaspekten korrekt. Allerdings treten bei der laufenden Instanz Schwierigkeiten auf, da dort die über den aktuellen Ausführungsstatus hinweg verschobene Aktivität nicht mehr ausgeführt werden kann. Dieses Problem ist auf Schemaebene nicht ersichtlich, sondern wird erst bei einer detaillierten Analyse auf Instanzebene erkannt.

Obwohl es unter rein formalen Gesichtspunkten nicht notwendig ist, haben wir uns entschlossen, im Rahmen der Datenflussüberprüfung auf Instanzebene auch eine entsprechende Validierung auf Schemaebene durchzuführen. Dies bringt Vorteile für die Benutzerinteraktion mit sich, die im folgenden Abschnitt erläutert werden.

Wir führen die Datenflussanalyse allerdings nur in Bezug auf diejenigen Aktivitäten durch, die direkt von der jeweiligen Änderungsoperation betroffenen sind. Bestehen bereits vor der Änderung des Prozessmodells Datenflussprobleme an anderer Stelle, werden diese in der Regel nicht erkannt. Die Datenfluss-Korrektheit am Ort der Änderungsoperation ist eine notwendige Bedingung für die Datenfluss-Korrektheit des ganzen Prozesses, aber keine hinreichende.

Theoretische Betrachtung

Hinsichtlich der Datenflussüberprüfung in unserem Konzept ist nur einem Kriterium relevant: Wird eine Variable initialisiert (beschrieben), bevor sie gelesen wird? Dieses Kriterium wird als *Def-Use-Analyse* bezeichnet.

Das Def-Use-Kriterium wird bei uns allerdings nicht strikt gehandhabt, wie es in anderen Anwendungsfällen – so auch z.B. im ADEPT-Modell – teilweise der Fall ist. In unserem Adaptionsmodell wird zwar überprüft, ob eine Variable eventuell nicht rechtzeitig initialisiert wird; dies ist aber kein Ausschlusskriterium für die Migration, sondern dient nur der Benutzerinformation. Würden wir eine eventuelle Nichtinitialisierung als ein Ausschlusskriterium ansehen, würde unser Adaptionsmodell strengere Vorgaben machen, als es die BPEL-Spezifikation tut. Dies ist nicht gewollt. Würden wir das Def-Use-Kriterium strikt halten, könnte die Situation auftre-

ten, dass wir an einem Prozess eine Änderung durchführen möchten, der zu 100% der BPEL-Spezifikation entspricht und seit langem zuverlässig läuft. Obwohl die Änderung den Datenfluss gar nicht berührt, könnte die Migrationskomponente den Prozess trotzdem als fehlerhaft erkennen und somit als nicht migrierbar einstufen würde. Ein Beispiel verdeutlicht diese Möglichkeit.

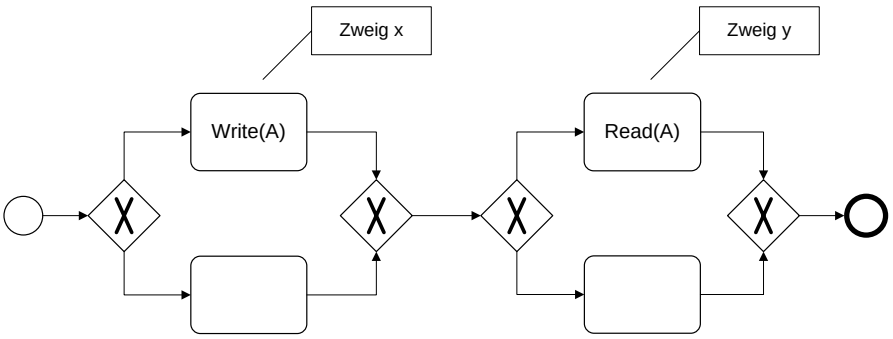


Abbildung 5.9: Beispiel: Nichterfüllung des Def-Use-Kriteriums

Eine Def-Use-Analyse des in Abbildung 5.9 gezeigten Prozesses würde als Ergebnis ein nicht korrektes Datenmodell melden: Variable A wird vor dem Lesen nicht notwendigerweise beschrieben.
Eine solche Prozessdefinition kann aber sinnvoll sein, wenn der Prozessentwickler weiß, dass der Zweig y nur dann gewählt wird, wenn zuvor auch der Zweig x gewählt wurde. Gründe hierfür können bekannte Abhängigkeiten im speziellen Anwendungsgebiet sein, die der Prozessdefinition selbst nicht zu entnehmen sind. Die BPEL-Spezifikation lässt deshalb eine solche Prozessdefinition zu. Sie sieht nur den Fehler *bpel:uninitializedVariable* vor, für den Fall, dass doch einmal ein nicht vorgesehener Pfad gewählt wird und eine nicht initialisierte Variable gelesen werden soll.

Für die Datenflussüberprüfung im Rahmen unseres Konzeptes wurde ein zweistufiges Verfahren entwickelt. Zunächst wird das Prozess-Schema des geänderten Prozesses auf seine Korrektheit hinsichtlich des Datenflusses überprüft. Danach werden alle laufenden Prozess-Instanzen betrachtet und dort die Datenflusskorrektheit analysiert, die auf Instanzebene vom Prozess-Zustand zum Migrationszeitpunkt abhängig ist.

Abbildung 5.10 zeigt den Ablauf der Datenfluss-Überprüfung für alle Änderungsoperationen. Zunächst wird das Prozess-Schema an eventuell kritischen Stellen mit Hilfe der *SLDataFlowValid*-Funktion seine Korrektheit hinsichtlich des Datenflusses geprüft. (*SLDataFlowValid* überprüft auf *S*chema-*L*evel, ob das Datenelement geschrieben wird, bevor es gelesen werden soll.) Bei einem *bezüglich der Änderungsoperation*[15] korrekten Datenfluss auf Schemaebene werden anschließend die zu migrierenden Instanzen mit der *ILDataFlowValid*-Funktion auf ihre Datenflusskorrektheit hin geprüft. Treten auch auf Instanzebene keine Probleme auf, kann die jeweilige Instanz migriert werden.

Wie bereits angesprochen, kann es aber auch vorkommen, dass der Datenfluss auf Schemaebene korrekt modelliert ist, auf Instanzebene aber Probleme bei der Migration auftreten. Beispiele hierfür bei der Durchführung einfacher Änderungsoperation zeigen die Abbildungen 5.11 und 5.12. In den gezeigten Beispielen wird die Funktion *SLDataFlowValid* (Datenflussüberprü-

[15] Es können an anderer Stelle im Prozess-Schema Fehler in der Datenflussmodellierung vorhanden sein, die aber die Änderungsoperation nicht betreffen und somit auch nicht durch die *SLDataFlowValid*-Prüfung entdeckt werden.

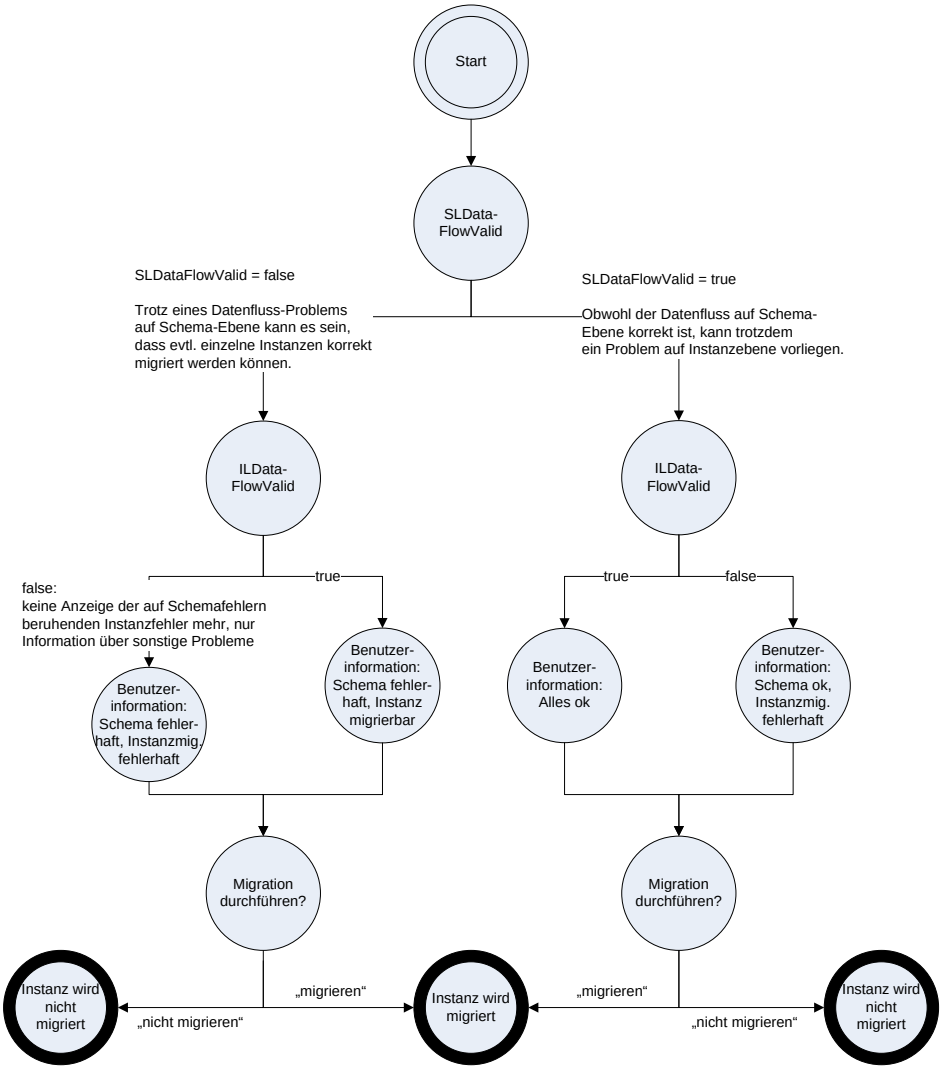


Abbildung 5.10: Zweistufige Datenfluss-Analyse

fung auf Schemaebene) jeweils zu *true* evaluiert, *ILDataFlowValid* (Datenflussüberprüfung auf Instanzebene) aber zu *false*.

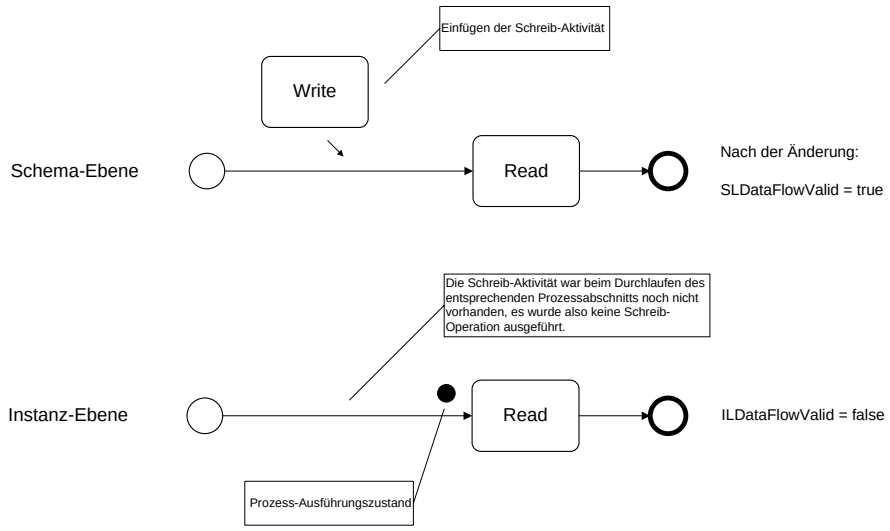


Abbildung 5.11: Unterschiedliche Bewertung der Datenflusskorrektheit auf Schema- und auf Instanzebene – Beispiel 1

Abbildung 5.11 zeigt ein Prozess-Schema, das im ursprünglichen Zustand bezüglich des Datenflusses fehlerhaft ist. In der *Read*-Aktivität wird eine Variable gelesen, aber zuvor nicht geschrieben. Die Änderungsoperation macht hieraus ein korrektes Datenflussmodell, indem vor der *Read*-Aktivität eine schreibende Aktivität eingefügt wird. *SLDataFlowValid* wird danach zu *true* evaluiert.
Auf Instanzebene ist der Prozessablauf aber bereits soweit fortgeschritten, dass die eingefügte *Write*-Aktivität nicht mehr ausgeführt wird. Die *ILDataFlowValid*-Funktion, die prüft, ob die Variable von der laufenden Prozessinstanz geschrieben wurde, bevor sie gelesen wird, liefert deshalb *false* als Rückgabewert.

In Abbildung 5.12 ist der Datenfluss im Ursprungszustand sowohl auf Schema- wie auch auf Instanzebene korrekt. Nachdem wir die Datenzugriffe von Variable A auf Variable B geändert haben, bestätigt *SLDataFlowValid* die bestehende Korrektheit des Datenflusses auf Schemaebene.
Auf Instanzebene wurde die Schreib-Aktivität aber bereits in ihrer alten Form durchlaufen, das heißt Variable A wurde geschrieben. Im nächsten Schritt soll nun die geänderte Aktivität ausgeführt und Variable B gelesen werden – was nicht möglich ist. *ILDataFlowValid* erkennt das instanzbezogene Datenfluss-Problem und gibt *false* als Ergebnis aus.

Ist der Datenfluss auf Schemaebene korrekt modelliert, führt auf Instanzebene aber zu Problemen, wird der Benutzer hierüber informiert und hat die Wahl, ob er die kritischen Instanzen migrieren möchte oder nicht.

Wird *SLDataFlowValid* zu *false* evaluiert, ist also der Datenfluss bereits auf Schemaebene fehlerhaft modelliert, wird im Folgenden trotzdem für jede Instanz überprüft, ob eine hinsichtlich des Datenflusses unkritische Migration möglich ist. Wie die Abbildungen 5.13 und 5.14 zeigen, ist es prinzipiell möglich, dass der Datenfluss einzelner Instanzen bei der Migration korrekt erhalten bleibt, während er auf Schemaebene fehlerhaft ist.

In Abbildung 5.13 ist zunächst ein korrekter Datenfluss auf Schemaebene vorhanden. Die Korrektheit wird aber durch das Löschen der schreibenden Aktivität beeinträchtigt. *SLDataFlowVa-*

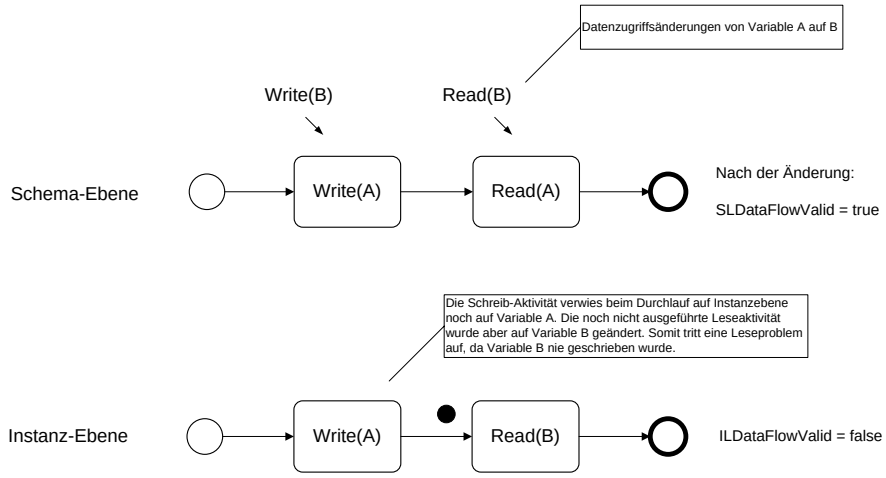


Abbildung 5.12: Unterschiedliche Bewertung der Datenflusskorrektheit auf Schema- und auf Instanzebene – Beispiel 2

lid liefert somit für das geänderte Prozess-Schema das Ergebnis *false*.
Auf Instanzebene ist der Prozess aber bereits soweit fortgeschritten, das die *Write*-Aktivität bereits ausgeführt wurde, bevor die Migration auf das neue Prozessschema erfolgte. Die betroffene Variable wurde also bereits geschrieben und kann somit im Folgenden problemlos gelesen werden. *ILDataFlowValid* liefert *true* als Rückgabewert.

Abbildung 5.14 veranschaulicht einen Fall, wie er – auch unabhängig von Änderungen am Prozessmodell – in Zusammenhang mit Verzweigungen eintreten kann. Das dargestellte Prozess-Schema ist offensichtlich fehlerhaft. Die Leseaktivität wird auf jeden Fall durchlaufen, es kann aber nicht garantiert werden, dass zuvor die Schreibaktivität ausgeführt wird.
Wenn wir aber eine Instanz betrachten, in welcher der obere Pfad der Verzweigung bereits gewählt wurde, dann ist in diesem Fall der Datenfluss auf Instanzebene in Ordnung.

Ist der Datenfluss auf Instanzebene unproblematisch, auf Schemaebene aber nicht korrekt modelliert, wird der Benutzer hierüber informiert und hat die Möglichkeit, selbst über die Durchführung der Migration zu entscheiden.

Bereitet der Datenfluss auf Instanzebene ebenfalls Probleme, werden dem Benutzer die grundsätzlichen Fehler im Prozessschema präsentiert und die zusätzlichen Schwierigkeiten auf Instanzebene, die *nicht* auf den Datenflussfehlern der Schemaebene basieren. Der Benutzer erhält so eine strukturierte Übersicht über die auftretenden Schwierigkeiten, gruppiert nach der Ebene, auf der das eigentliche Problem auftritt.

Generell ist es wünschenswert, dass der Benutzer nach Prüfung der Datenflusskorrektheit auf Schema- und Instanzebene detaillierte Informationen darüber erhält, wo im Prozessablauf Probleme auftreten, welche Datenelemente betroffen sind, ob dieses Problem nur bei einem bestimmten Prozesszustand auftritt und weiteres mehr. Die Umsetzung dieser Informationsbereitstellung und ihrer vorausgehenden -Gewinnung ist teilweise recht einfach möglich, teilweise ist hierfür auch eine tiefergreifende Prozessanalyse erforderlich. Wir begnügen uns an dieser Stelle damit, auf die Wünschenswertigkeit einer möglichst aussagekräftigen und umfassenden Benutzerinformation hinzuweisen, ohne diese in dieser Arbeit aber umzusetzen. Die Integration einer solchen Analyse in unser Konzept würde die im Folgenden gezeigten Änderungsprimitive zum Teil erheblich aufblähen und unübersichtlicher machen. Insbesondere würden die Parameterlisten einzelner Algorithmen erheblich umfangreicher werden.

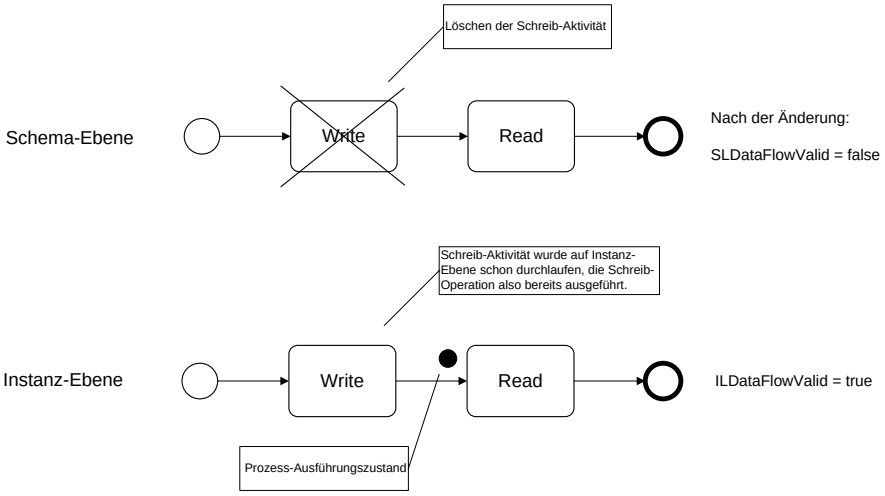


Abbildung 5.13: Unterschiedliche Bewertung der Datenflusskorrektheit auf Schema- und auf Instanzebene – Beispiel 3

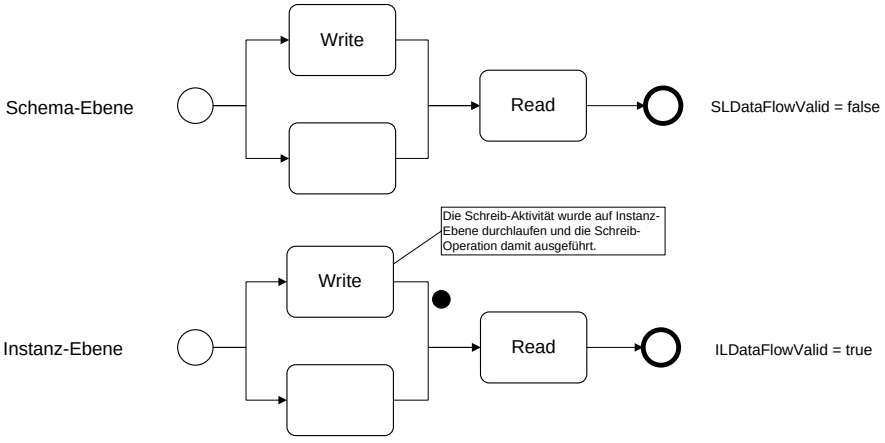


Abbildung 5.14: Unterschiedliche Bewertung der Datenflusskorrektheit auf Schema- und auf Instanzebene – Beispiel 4

Umsetzung der Datenfluss-Überprüfung

In Kapitel 4 haben wir zwei Ansätze zur Datenflussüberprüfung in BPEL vorgestellt. Beide Verfahren sind sehr mächtig, aber auch sehr komplex und über die vorgestellten Arbeiten hinaus bisher wenig untersucht worden. Eine Verifizierung dieser Algorithmen, ihre Anpassung an unsere Bedürfnisse und ihre Implementierung wären mit einem erheblichen Aufwand verbunden, der im Rahmen dieser Arbeit nicht geleistet werden kann.

Wir haben deshalb ein eigenes Verfahren dafür entwickelt, eine Def-Use-Analyse auf einer BPEL-Prozessdefinition auszuführen. Die Datenflussüberprüfung wird direkt auf Basis des in Kapitel 5.2.2 definierten formalen Modells durchgeführt. Das bedeutet insbesondere, dass der Datenfluss auf Schemaebene direkt aus der Prozess-Definition im BPEL-Format bestimmt wird. Somit ist unser Konzept in diesem Punkt mit existierenden BPEL 2.0-Implementierungen kompatibel, sofern diese blockstrukturiert sind. Wir verzichten bewusst auf den Aufbau einer Zwischenrepräsentation, wie sie im Rahmen der in Kapitel 4 untersuchten Verfahren benötigt wird, oder auf eine Erweiterung von BPEL um ein Datenflussmodell, wie es beispielsweise mit *BPEL-D* an der Universität Stuttgart entwickelt wurde ([Kha08] und [VF07]).
Aus Vereinfachungsgründen differenzieren wir nicht nach Zugriffen auf einzelne Variablen*teile* (*WSDL Message Parts*), wie sie in der BPEL-Spezifikation vorgesehen sind. Wir beschränken uns auf die Analyse von Zugriffen auf ganze Variablen.

Unser Ausgangspunkt ist eine BPEL-Aktivität, die eine Variable lesen möchte. Es soll geprüft werden, ob die Variable zuvor von einer anderen Aktivität initialisiert wurde.
Wir verwenden eine intuitive Herangehensweise, vergleichbar mit der in [Hei03] verwendeten. Zunächst identifizieren wir diejenigen BPEL-Aktivitätstypen, die Variablen lesen bzw. schreiben können. Deren Instanzen gilt es in der Prozessdefinition aufzufinden und zu untersuchen.

Leseaktivitäten

Im Folgenden sind alle Aktivitätstypen aufgeführt, die einen Lesezugriff enthalten bzw. optional enthalten können.

Einfache Aktivitäten:

Die *assign*-Aktivität kopiert einen Variablenwert von einer Variable auf eine andere. Dies wird im Rahmen eines <*from*>/<*to*>-Konstruktes spezifiziert. Änderungsoperationen an einem Variablenwert können hierbei ebenfalls durchgeführt werden.

Mittels *invoke*-Aktivitäten lassen sich Web-Services aufrufen. Normalerweise spezifiziert die Aktivität hierzu einen Request-Response-Aufruf, es sind aber auch einfache (One-Way) Aufrufe möglich. In beiden Fällen wird über die *inputVariable* das zu sendende Datenelement angegeben.

Die *reply*-Aktivität dient dazu, eine Nachricht an einen Partner-Webservice zu senden. Die zu sendende Nachricht kann in einer Variablen gespeichert sein, die dann von der *reply*-Aktivität gelesen wird.

Mit der *throw*-Aktivität lassen sich prozessinterne Fehler signalisieren. Daten zur Fehlerbeschreibung können in einer Variable übermittelt werden, die im Attribut *faultVariable* spezifiziert wird.

wait bestimmt ein Zeitintervall oder eine Deadline. Bis das Zeitintervall bzw. die Deadline abgelaufen ist, wird die Prozessausführung an dieser Stelle angehalten. Sowohl das *for*- wie auch das *until*-Element lassen sich als Expression auf eine Variable umsetzen.

Strukturierte Aktivitäten:

Das *forEach*-Element definiert eine For-Schleife. Über die Elemente *startCounterValue* und *finalCounterValue* wird definiert, wie oft die Schleife ausgeführt werden soll. Es ist aber auch möglich, eine Abbruchbedingung zu formulieren, sodass die Schleifenausführung beendet wird, sobald die Bedingung erfüllt ist.
Die Ausführung des im *forEach*-Element enthaltenen Scopes kann entweder sequentiell oder parallel erfolgen. Das heißt, es ist auch möglich, dass der Scope entsprechend der Laufvariable n-fach instanziiert wird und die Instanzen parallel ausgeführt werden.
Sowohl bei der Definition des *counterName*-Attributs, als auch bei der Spezifizierung der Anzahl der Durchläufe und bei der Angabe der Abbruchbedingung kann auf Variablen lesend zugegriffen werden.

Mit Hilfe der *if*-Aktivität lassen sich bedingte Verzweigungen modellieren. Jede enthaltene Bedingung kann als Expression auf eine Variable definiert werden.

Innerhalb einer *pick*-Aktivität sind mehrere Ereignisse mit jeweils einer zugeordneten Aktivität angegeben. *pick* wartet auf das Auftreten einer dieser Events und führt dann die damit assoziierte Aktivität aus.
Für den Fall, dass (lange Zeit) kein Ereignis aufgelöst wird, kann zusätzlich eine Timeout-Aktivität angegeben werden. Das hierbei verwendete *for*- (Zeitspanne) bzw. *until*- (Deadline) Element kann über eine Expression auf eine Variable definiert sein.

Die in einem *repeatUntil*-Element enthaltene Aktivität wird so oft ausgeführt, bis die angegebene Bedingung erfüllt ist. Diese lässt sich über eine Expression auf eine Variable definieren.

Analog zu *repeatUntil* gibt es auch eine *while*-Aktivität. Auch hier kann die enthaltene Aktivität mehrmals ausgeführt werden. Während die *while*-Aktivität aber mindestens einmal durchlaufen wird, ist es bei *repeatUntil* möglich, dass die enthaltene Aktivität gar nicht ausgeführt wird, wenn die angegebene Bedingung gleich zu *false* evaluiert wird. Diese Bedingung kann, wie oben, durch eine Expression auf eine Variable definiert sein.

Schreib-Aktivitäten

Es existieren vier Aktivitätstypen, die Variablen schreiben und damit initialisieren können. Teilweise erfolgt zusätzlich auch ein Lesezugriff innerhalb dieser Aktivitäten.

Eine *assign*-Aktivität enthält ein *from*-Element, das angibt, welche Variable gelesen werden muss, um ihren Wert dann einer anderen Variable zuweisen zu können.

invoke-Aktivitäten sind sowohl als einfache, wie auch als Request-Response-Aufrufe verwendbar. Im Falle einer Request-Response-Operation wird die Variable angegeben, in der die Antwortnachricht gespeichert werden soll (*outputVariable*).

Eine *receive*-Aktivität empfängt Nachrichten von einem Partner-Webservice und speichert diese in einer Variable.

pick ist die einzige strukturierte Aktivität, die eine Variable schreiben kann. Das Auftreten eines vorgesehenen Events wird über eine eintreffende Nachricht signalisiert. Diese Nachricht wird in einer Variable gespeichert.

Vorgehen

Für jeden Typ einer Änderungsoperation ist eine spezielle Form der Datenflussanalyse notwendig. Dies ergibt sich daraus, dass keine allgemeine Datenflusskorrektheit für BPEL vorgesehen

ist. Wir beschränken uns in unserer Analyse folglich auch auf die Auswirkungen unserer Änderungsoperationen. Es wird keine allgemeine Datenflusskorrektheits-Analyse über den ganzen Prozess durchgeführt.

Am Beispiel der in dieser Arbeit vorgestellten Änderungsoperationen, dem Einfügen und dem Löschen einer einfachen Aktivität, stellen wir die zugehörige Datenflussüberprüfung vor. Die vorgeschlagenen Algorithmen können als Grundgerüst für eine Datenflussüberprüfung bei vollständiger Umsetzung aller denkbaren Änderungsoperationen dienen. In weiteren Arbeiten noch umzusetzende Operationen können entweder auf den vorgestellten Algorithmen aufbauen oder müssen um speziell angepasste Datenfluss-Algorithmen ergänzt werden.

Serielles Einfügen einer einfachen Aktivität:

Beim Einfügen einer *Lese*operation ist auf Schemaebene eine Rückwärtsprüfung in der Prozessdefinition notwendig, bei der festgestellt wird, ob die Variable zuvor geschrieben wird.
Ist der Datenfluss auf Schemaebene korrekt, trifft dies auch auf die Instanzebene zu. Ist er auf der Schemaebene nicht korrekt, besteht die Möglichkeit, dass auf Instanzebene im Einzelfall trotzdem kein Datenflussproblem auftritt. Diese Möglichkeit wurde in Abbildung 5.14 aufgezeigt. Auf Instanzebene wird deshalb überprüft, ob die kritischen Variablen nicht eventuell doch bereits beschrieben wurden.[16]

Das Einfügen einer *Schreib*operation ist unkritisch, das dies normalerweise keine neuen relevanten Datenabhängigkeiten mit sich bringt. Dies wäre nur der Fall, wenn die ursprüngliche Prozessdefinition bereits fehlerhaft war, es also eine Leseaktivität gibt, ohne dass eine Schreibaktivität vorhanden ist. Wir setzen aber eine korrekte Prozessdefinition voraus.[17]

Beinhaltet die eingefügte Aktivität sowohl einen Lese- wie auch einen Schreibzugriff (z.B. bei einer *Assign*- oder *Invoke*-Aktivität), ist in diesem Fall die Datenflussüberprüfung für sie als Leseaktivität durchzuführen.

Serielles Löschen einer Aktivität:

Im Falle einer seriellen Löschoperation ist das Entfernen einer *Lese*aktivität aus dem Prozessablauf unkritisch. Zu einer Leseaktivität gibt es keine von ihr abhängigen Aktivitäten, deshalb kann sie ohne Einfluss auf die Datenflusskorrektheit entfernt werden.

Beim Löschen einer *Schreib*aktivität ist zu untersuchen, ob im späteren Verlauf des Prozesses Leseaktivitäten auf die entsprechende Variable existieren. Ist dies der Fall, muss mindestens eine weitere sichere Schreibaktivität existieren, die im Prozessablauf vor der entsprechenden Leseaktivität steht.
Ist der Datenfluss auf Schemaebene inkorrekt, ist auf Instanzebene der Ausführungszustand der jeweiligen Prozessinstanz maßgebend. Wurde die zu löschende Schreibaktivität schon durchlaufen, die Variable also bereits initialisiert, wirkt sich die Änderungsoperation nicht auf die Instanz aus und der Datenfluss ist weiterhin korrekt. Diese Möglichkeit wurde in Abbildung 5.13 aufgezeigt. Wurde die Schreibaktivität, die gelöscht werden soll, noch nicht ausgeführt, besteht sowohl auf Schema-, wie auch auf Instanzebene ein Datenflussproblem.

Beinhaltet die zu löschende Aktivität einen Lese- und einen Schreibzugriff, dann ist die Datenflusskorrektheit hinsichtlich ihrer Rolle als Schreibaktivität zu prüfen.

[16] Den Fall, dass der schreibende Ast einer Verzweigung bereits ausgewählt ist, die schreibende Aktivität selbst aber noch nicht ausgeführt wurde, können wir mit unserem bewusst einfach gehaltenen Datenflussmodell leider nicht abfangen.

[17] Wie zu Beginn von Kapitel 5.4.4 beschrieben gehen wir nicht unbedingt von einer hinsichtlich des allgemeinen Datenflusses korrekten Prozessdefinition aus. Wir setzen aber zumindest voraus, dass der Datenfluss bezogen auf die Anwendungsdomäne und den vorgesehenen Prozessablauf korrekt ist.

Such-Algorithmen für das Auffinden von Lese- und Schreibaktivitäten

Zunächst betrachten wir die Datenfluss-Überprüfung auf der Schemaebene, daran anschließend auf der Instanzebene.

A. Schemaebene

Für die Korrektheitsüberprüfung des Datenflusses auf *Schemaebene* ruft die ausgewählte Änderungsoperation die Funktion *SLDataFlowValid* auf.
Diese Funktion veranlasst – je nach Art der aufrufenden Änderungsoperation und je nach Typ der geänderten Aktivität (Lese- oder Schreibaktivität) – die entsprechende Überprüfung der Datenflusskorrektheit. Sie setzt also das in den beiden vorangehenden Abschnitten beschriebene Konzept der individuellen Datenflussanalyse um und ruft die jeweiligen Such-Algorithmen auf. Zurückgegeben wird die Menge der Variablen, die aufgrund fehlender Schreibzugriffe zu einem inkorrekten Datenfluss auf Schemaebene führen. Ist die zurückgelieferte Variablen-Menge leer, ist der Datenfluss korrekt. Wir sprechen im Folgenden manchmal vereinfachend von einem Rückgabewert *true* bzw. *false* und meinen damit eine leere bzw. eine nichtleere Rückgabemenge. Abbildung 5.15 verdeutlicht nochmals die möglichen Entscheidungswege.

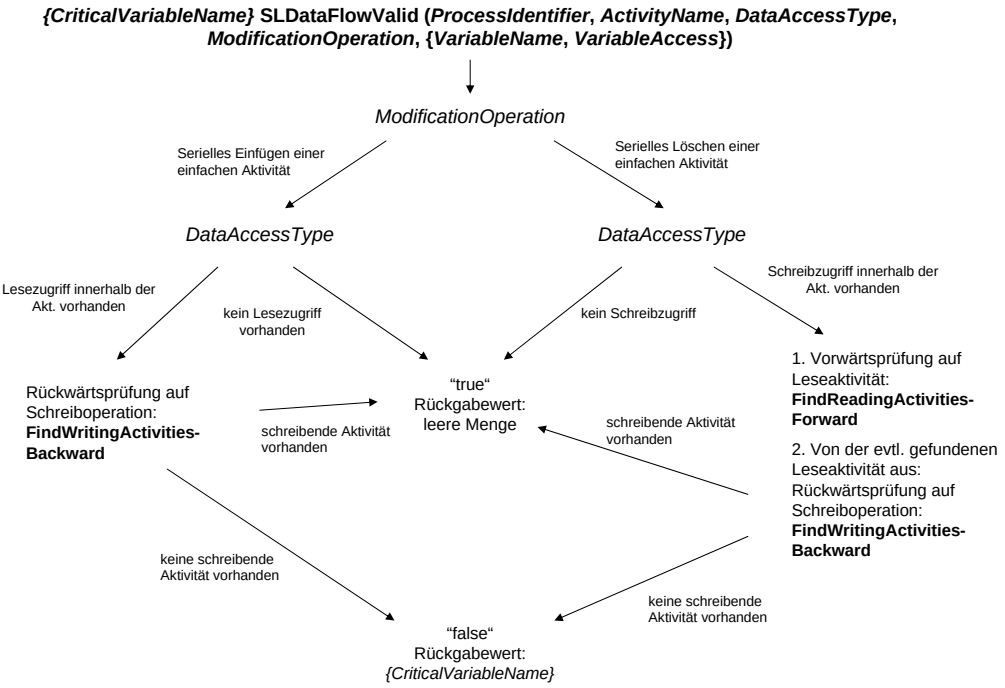


Abbildung 5.15: *SLDataFlowValid* – Aufrufstruktur

Wie der Abbildung zu entnehmen ist, bietet die Funktion bislang nur Unterstützung für die Änderungsoperationen *Serielles Einfügen einer Aktivität* und *Serielles Löschen einer Aktivität*. Für die Unterstützung weiterer Operationen muss die Funktion erweitert werden.

Im Folgenden erläutern wir den Suchalgorithmus *FindWritingActivitiesBackward* inklusive der von ihm aufgerufenen Unterfunktionen. *FindWritingActivitiesBackward* wird dazu benutzt, vorangehende Schreibaktivitäten auf eine Variable zu finden. Daran anschließend wird der Ablauf des *FindReadingActivitiesForward*-Algorithmus erklärt. Mit diesem lassen sich nachfolgende Leseaktivitäten ermitteln.

Rückwärtssuche nach Schreibaktivitäten (FindWritingActivitiesBackward)

Relevant für das Auffinden einer eventuellen Variablen-Initialisierung sind nur diejenigen Aktivitäten, die im Prozessablauf *vor* der betrachteten Leseaktivität ausgeführt werden. Da wir auf keine Zwischenrepräsentation zurückgreifen können, in der wir mittels eines *GetPredecessors*-Befehls bequem die Vorgänger selektieren können, müssen wir diese Aktivitäten erst einmal direkt über die BPEL-Prozessdefinition bestimmen.

FindWritingActivitiesBackward ist eine Funktion, deren Aufgabe es ist, für jede relevante Variable die jeweils erste vorangehende Schreibaktivität zu finden. Hierzu ruft sie für jede übergebene Variable die Funktion *PrecedingWriter* auf, die die eigentliche Such-Funktionalität enthält.

Der in Abbildung 5.17 dargestellte Algorithmus *PrecedingWriter* benötigt als Parameter den *ProcessIdentifier*, den eindeutigen *ActivityName* der Leseaktivität und die entsprechende *Variable*. Mit der Leseaktivität auf der ersten Ebene beginnend bestimmen wir jeweils das parent-Element des betrachteten Knotens. Ist der parent-Knoten vom Aktivitätstyp *sequence*, erfolgt eine Tiefensuche (*WriterDeepSearch*) über alle vorangehenden Aktivitäten auf dieser Ebene, also über alle preceding-sibling-Elemente der Leseaktivität. Liefert die Tiefensuche das Ergebnis, dass die Variable korrekt initialisiert wird, wird der Algorithmus abgebrochen und liefert als Ergebnis den Namen der Schreibaktivität zurück.
Falls der parent-Knoten ein umgebendes Scope ist, wird überprüft, ob die genannte Variable eine lokale Variable innerhalb dieses Scopes ist. Ist dies der Fall, steht fest, dass die Variable vor dem Lesezugriff nicht zwingend initialisiert wird. *PrecedingWriter* wird abgebrochen und gibt als Ergebnis die leere Menge zurück. Es existiert also keine vorangehende Schreibaktivität auf diese Variable, die sicher ausgeführt wird. Ist die Variable keine lokale Variable dieses Scopes, wird in der nächsten Stufe das parent-Element des Scopes betrachtet.
Ist die parent-Aktivität vom Typ *if*, *while*, *repeatUntil*, *pick*, *flow* oder *forEach*, wird ebenfalls in der nächsten Stufe das jeweilige parent-Element betrachtet. Dass wir beispielsweise bei einer *if*-Aktivität die anderen Pfade nicht betrachten müssen, liegt daran, dass bei der Betrachtung einer Leseaktivität innerhalb eines Pfades nur potentielle Schreibaktivitäten in genau diesem Pfad relevant sind. Wird zur Laufzeit ein anderer Pfad gewählt, wird die Leseaktivität nie ausgeführt und es ist aus diesem Grund auch keine Datenflussüberprüfung für die mit ihr assoziierte Variable notwendig.
Abbildung 5.16 skizziert nochmals das Vorgehen.

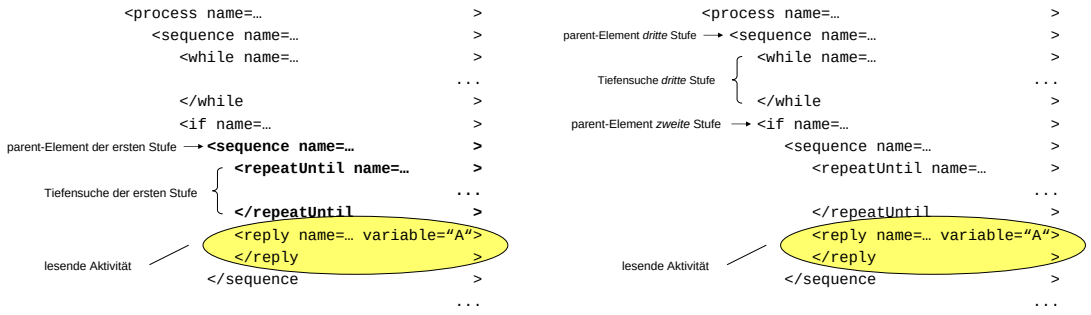


Abbildung 5.16: Datenflussanalyse – Ablaufskizzierung

Der Algorithmus *WriterDeepSearch* führt eine Tiefensuche in der ihm übergebenen Aktivität durch. Das Verfahren ähnelt dem oben dargestellten rekursiven Vorgehen, allerdings in umgekehrter Richtung, in Richtung der Unterelemente. Je nach Elementtyp der betrachteten Aktivität wird der Algorithmus auf allen oder auf einem Teil der Aktivitäten der darunterliegenden Ebene neu aufgerufen.

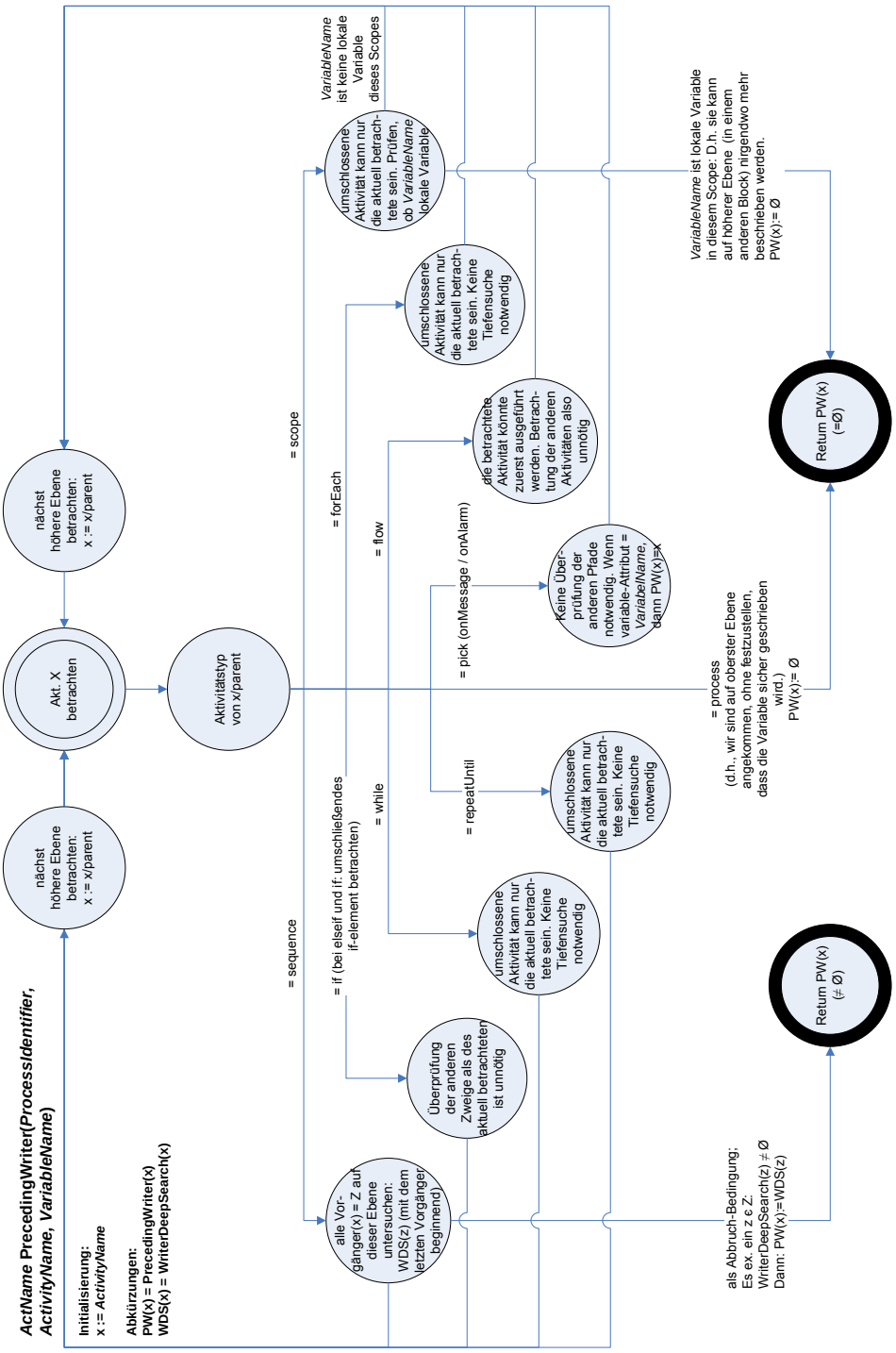


Abbildung 5.17: Ablaufdiagramm des *PrecedingWriter*-Algorithmus

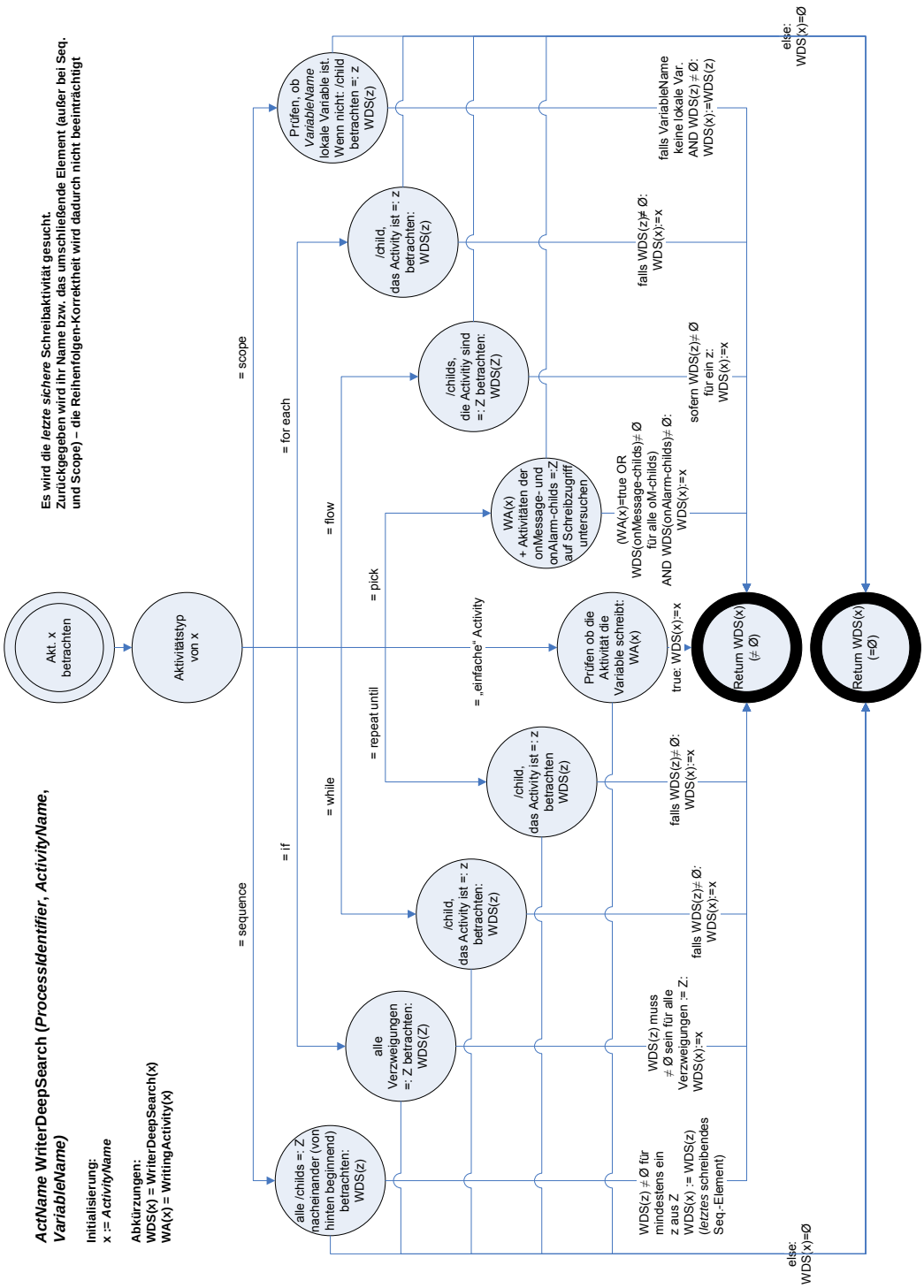


Abbildung 5.18: Ablaufdiagramm des *WriterDeepSearch*-Algorithmus

Damit ein sicheres Schreiben des entsprechenden Datenelementes im untersuchten Block sichergestellt ist, muss im Falle einer *sequence*-Aktivität mindestens eine umschlossene Aktivität die Variable schreiben. Zu dieser Überprüfung wird der Algorithmus auf allen Unteraktivitäten jeweils neu aufgerufen. Die Ergebnisse werden mit einem logischen OR verknüpft.
Handelt es sich bei der betrachteten Aktivität um ein *if*-Element, also eine Verzweigung, muss die Variable in allen Teilzweigen geschrieben werden, damit der Algorithmus ein positives Ergebnis zurückliefern kann.
Bei einem *scope*-Element wird geprüft, ob die gesuchte Variable in diesem Block als lokale Variable definiert wird. Ist dies der Fall, können keine umschlossenen Aktivitäten im übergeordneten (globalen) Gültigkeitsbereich der Variable als Schreibaktivitäten auftreten. Folglich kann die Suche an dieser Stelle mit negativem Ergebnis abgebrochen werden.
Zur Gewährleistung eines sicheren Schreibzugriffs bei einer *pick*-Aktivität muss sichergestellt werden, dass die Variable zum einen für jede eintreffende Nachricht geschrieben wird. Dies ist der Fall, wenn alle *onMessage*-Kindelemente entweder in ihrem *variable*-Attribut auf die fragliche Variable verweisen oder diese innerhalb der enthaltenen Aktivität geschrieben wird. Zusätzlich muss aber auch der Fall abgedeckt sein, dass ein Timeout auftritt. Deshalb müssen auch alle *onAlarm*-Aktivitäten die Variable schreiben.
Alle weiteren Fälle sind weniger komplex. In den meisten Fällen muss die jeweilige Unteraktivität auf ihre Eigenschaft als Schreibaktivität untersucht werden. Die Details sind dem Ablaufdiagramm in Abbildung 5.18 zu entnehmen.

Stößt der Algorithmus auf eine einfache, unstrukturierte Aktivität, wird der Algorithmus *WritingActivity* aufgerufen, der prüft, ob die Variable in dieser Aktivität geschrieben wird (siehe Abb. 5.19).

Die Aufruf-Hierarchie der vorgestellten Algorithmen wird in Abbildung 5.20 noch einmal verdeutlicht: Für alle durch die Änderungsoperation in kritischer Weise beeinflussten, also eingefügten oder gelöschten Aktivitäten wird gegebenenfalls die Funktion *FindWritingActivities-Backward* aufgerufen. Diese liefert für jede betroffene Variable die jeweils letzte vorangehende Schreibaktivität zurück.
Hierfür wird für jede Variable die Suchfunktion *PrecedingWriter* aufgerufen.
Diese führt eine Suche in Richtung des root-Elements, wie auch gegebenenfalls eine Tiefensuche in einzelne Prozesselemente (*WriterDeepSearch*) durch, um vorhandene Schreibaktivitäten aufzufinden.
Zur direkten Überprüfung einer Aktivität auf Schreibzugriffe auf die betrachtete Variable dient die Funktion *WritingActivity*, die einen booleschen Wert zurückliefert.

Der Algorithmus *PrecedingWriter* findet die im Prozessablauf letzte vorangehende Schreibaktivität und liefert deren Bezeichnung bzw. die der umschließenden Aktivität. Wir benötigen nur *eine* sicher ausgeführte Schreibaktivität, um die nötige Variablen-Initialisierung zu garantieren. Dabei wird im Unteralgorithmus *WriterDeepSearch* als Bezeichnung der Schreibaktivität die der umschließenden Aktivität zurückgegeben, soweit dies die Reihenfolge in der Prozessdefinition nicht beeinflusst. Dies trifft für alle strukturierten Aktivitäten zu, mit Ausnahme der *sequence*-Aktivität. Alle anderen strukturierten Aktivitäten umschließen entweder nur einzelne oder parallel bzw. selektiv auszuführende Unteraktivitäten. Analog verfahren wir bei der im folgenden Abschnitt vorgestellten Suche nach nachfolgenden Leseaktivitäten.

Der Sinn dieses Vorgehens wird deutlich, wenn man in Abbildung 5.15 den Zweig ganz rechts betrachtet. Hier muss zunächst untersucht werden, ob – von der geänderten Aktivität aus betrachtet – nachfolgende Leseaktivitäten existieren. Von diesen ausgehend ist dann mit Hilfe einer Rückwärtssuche zu prüfen, ob eine vorangehende Schreibaktivität existiert.
Suchen wir vorab nicht nach mehreren, sondern nach *einer*, und zwar der *ersten* Leseaktivität im Prozessverlauf, vereinfacht dies unsere Suche nach vorangehenden Schreibaktivitäten erheblich.

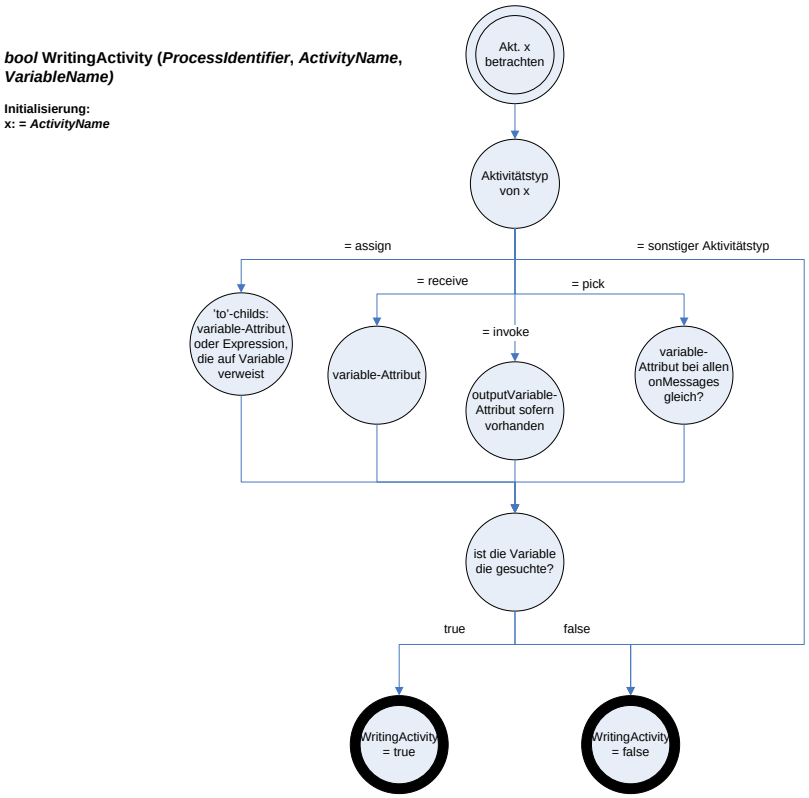


Abbildung 5.19: Ablaufdiagramm des *WritingActivity*-Algorithmus

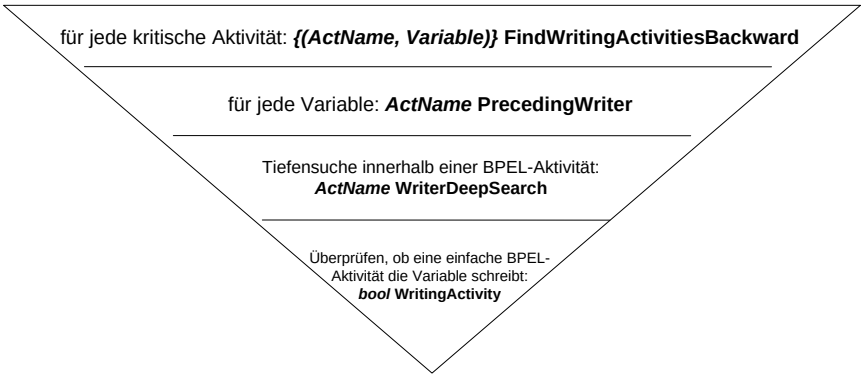


Abbildung 5.20: *FindWritingActivitiesBackward* – Hierarchie der Funktionsaufrufe

Vorwärtssuche nach Leseaktivitäten (FindReadingActivitiesForward)

Die Vorwärtssuche nach Leseaktivitäten verläuft von ihrer Struktur her analog zu der gerade vorgestellten Rückwärtssuche nach Schreibaktivitäten. Im Detail ergeben sich allerdings an manchen Stellen Unterschiede.

Die Funktion *FindReadingActivitiesForward* ruft für jede durch die Änderungsoperation betroffene Variable den Algorithmus *SucceedingReader* auf (siehe Abbildung 5.22). Dieser stellt das Analogon zum *PrecedingWriter*-Algorithmus dar und sucht nach einer nachfolgenden Leseoperation.
Hierzu wird ebenfalls die Prozessstruktur durchschritten, wobei bei der Analyse einer *sequence*-Struktur nicht die letzte vorangehende Schreibaktivität ermittelt wird, sondern die erste nachfolgende. Zur Durchführung der Tiefensuche wird der Algorithmus *ReaderDeepSearch* (Abb. 5.23) aufgerufen. Dieser liefert die *erste mögliche* Leseaktivität zurück. Im Gegensatz zum *WriterDeepSearch*-Algorithmus ist es hier in Bezug auf Verzweigungsstrukturen also nicht notwendig, dass die gesuchte Leseaktivität in *allen* möglichen Zweigen vorkommt. Es genügt, dass sie in *einem* der Zweige vorhanden ist und eventuell zur Ausführung kommt.

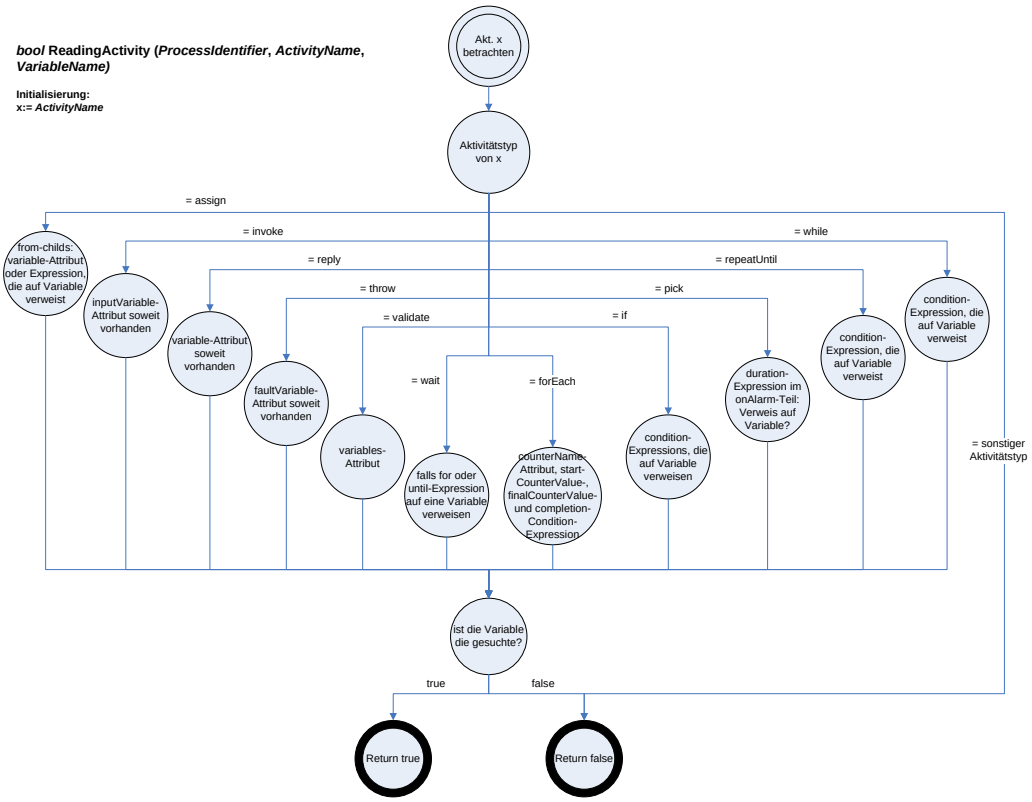


Abbildung 5.21: Ablaufdiagramm des *ReadingActivity*-Algorithmus

Während Schreibzugriffe nur in den oben gezeigten vier Aktivitätstypen möglich sind, können Lesezugriffe in einer Vielzahl von Elementen erfolgen, darunter zahlreiche strukturierte Aktivitäten. Die Funktion *ReadingActivity* (Abb. 5.21) führt die Überprüfung auf Lesezugriffe durch. Sie wird von *ReaderDeepSearch* sowohl dann aufgerufen, wenn eine einfache Aktivität vorliegt, als auch beim Betrachten der meisten strukturierten Aktivitäten. Erfolgt nämlich beispielsweise ein Lesezugriff bereits durch eine Attributangabe im jeweiligen strukturierten Element, ist eine Tiefensuche über die umschlossenen Aktivitäten nicht mehr notwendig.

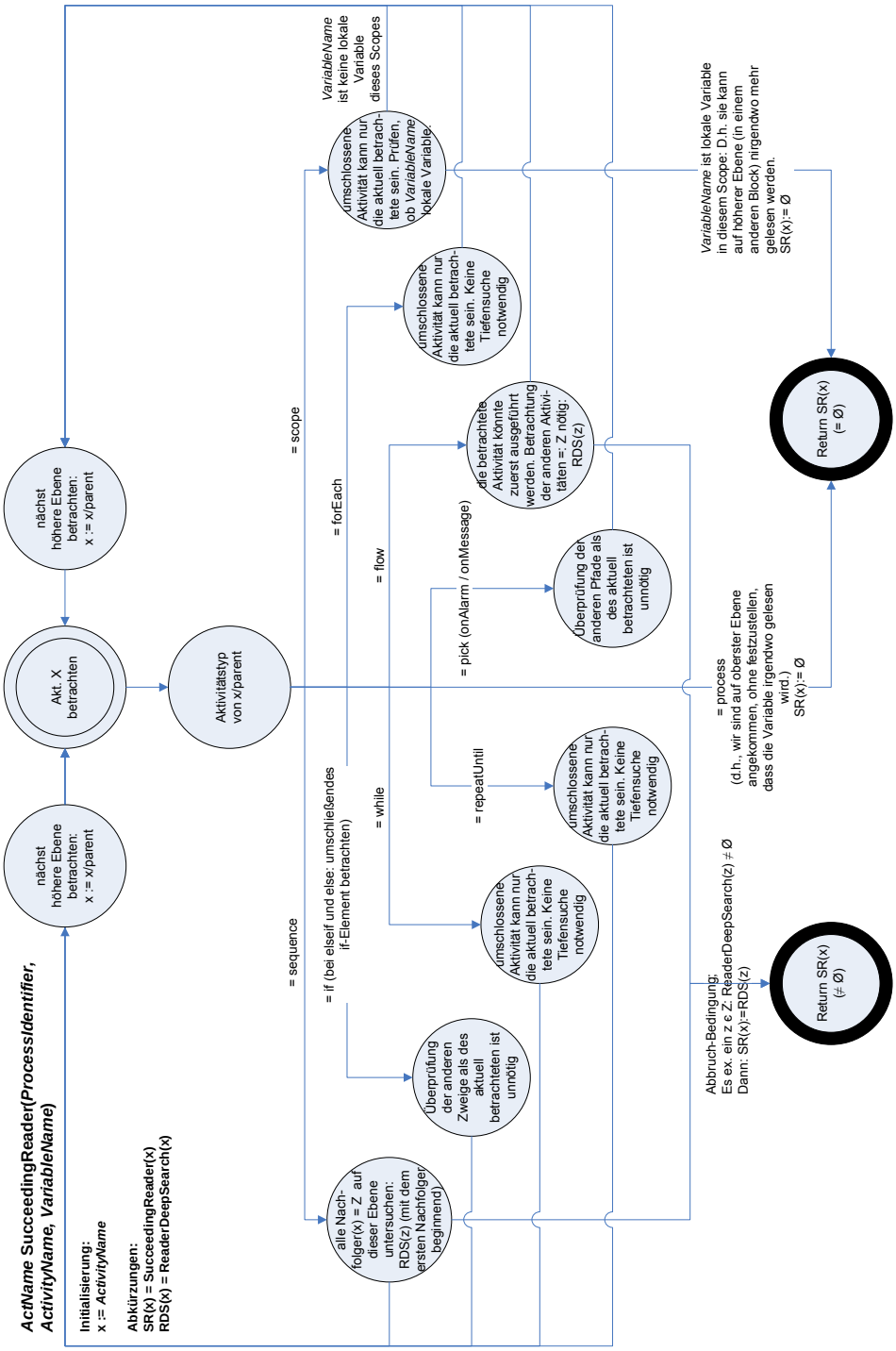


Abbildung 5.22: Ablaufdiagramm des *SucceedingReader*-Algorithmus

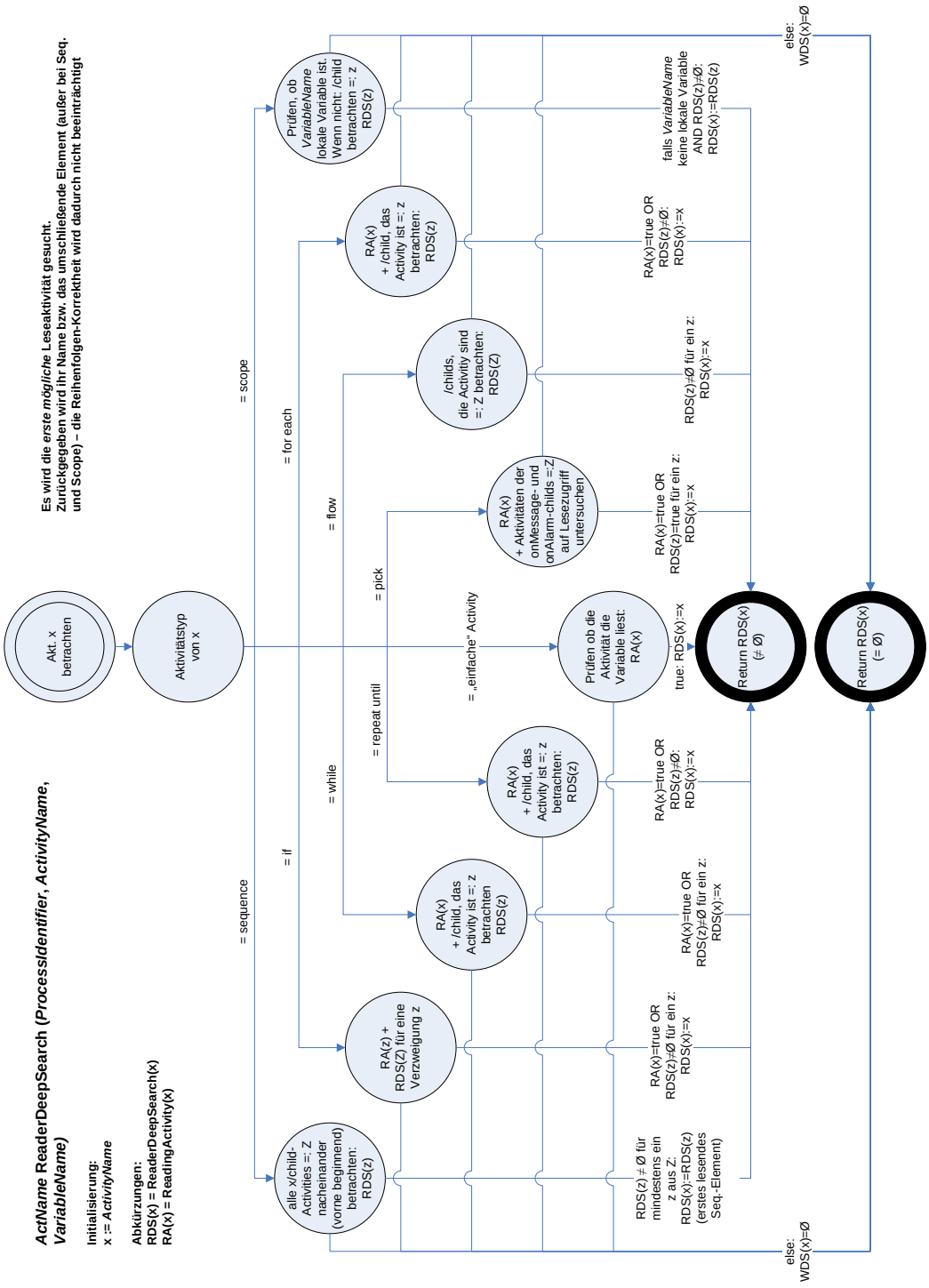


Abbildung 5.23: Ablaufdiagramm des *ReaderDeepSearch*-Algorithmus

Abbildung 5.24 zeigt nochmals die Aufrufhierarchie für die Suche nach einer nachfolgenden Leseaktivität.

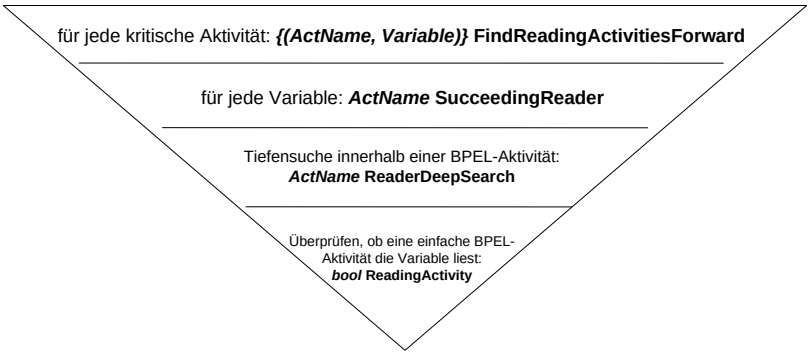


Abbildung 5.24: *FindReadingActivitiesForward* – Hierarchie der Funktionsaufrufe

B. Instanzebene

In Abbildung 5.25 ist der Ablauf der Datenflussüberprüfung auf Instanzebene dargestellt. Wir haben das Vorgehen in groben Zügen bereits in den Abschnitten *Serielles Einfügen einer einfachen Aktivität* und *Serielles Löschen einer einfachen Aktivität* erläutert.

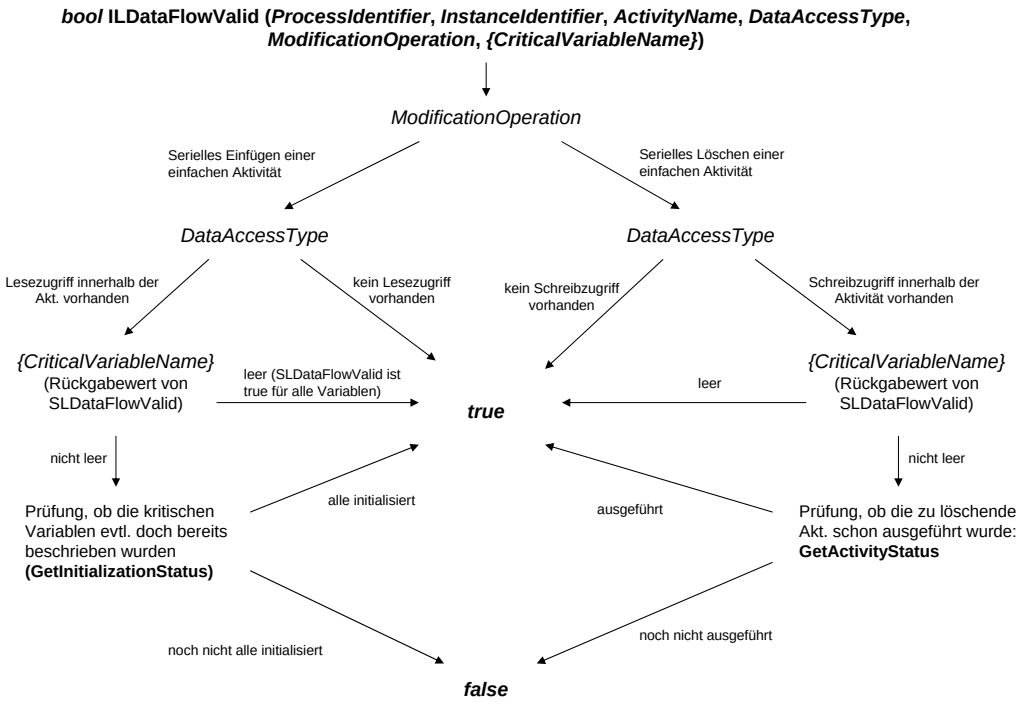


Abbildung 5.25: *ILDataFlowValid* – Aufrufstruktur

Wir betrachten zunächst eine mögliche Einfügeoperation. Ist innerhalb der neu hinzugekommenen Aktivität ein Lesezugriff vorhanden, wird zunächst das Ergebnis der *SLDataFlowValid*-Funktion analysiert. Ist der Datenfluss auf Schemaebene korrekt, wird als Rückgabewert $\{CriticalVariableName\}$ von *SLDataFlowValid* also eine leere Menge zurückgeliefert, ist auch der Datenfluss auf Instanzebene als korrekt zu bewerten.

Existieren auf Schema-Ebene allerdings Probleme hinsichtlich des Datenflusses, wird geprüft, ob die als kritisch eingestuften Variablen auf Instanzebene nicht doch bereits beschrieben wurden. Ist dies nicht der Fall, wird der Datenfluss auf Instanzebene ebenfalls als ungültig bewertet, *ILDataFlowValid* wird somit zu *false* evaluiert.

Der andere interessante Fall ist das Löschen einer schreibenden Aktivität. Auch hier ist das Ergebnis von *SLDataFlowValid* maßgebend für die Datenfluss-Korrektheit auf Instanzebene. Wurden Datenflussprobleme auf Schemaebene entdeckt, besteht noch die Möglichkeit, dass die zu löschende Aktivität im Rahmen der betrachteten Instanz schon durchlaufen und die Schreiboperation somit bereits ausgeführt wurde. Diese Überprüfung nimmt die Funktion *GetActivityStatus* vor. Wurde die Aktivität bereits ausgeführt, liefert *ILDataFlowValid true* als Ergebnis, andernfalls wird *false* zurückgegeben.

Die Überprüfung, ob eine Aktivität bereits ausgeführt wurde, lässt sich leicht auf Basis des Instanzzustandes ermitteln, wie er in Kapitel 5.2.2 definiert wurde.

Im bisherigen Status des Migrationskonzeptes ist die Abfrage des Ausführungszustands der zu löschenden Aktivität eigentlich unnötig. Wie wir später sehen werden, überprüft die Änderungsoperation bereits den Ausführungszustand der Instanz. Bei Instanzen, deren Prozessablauf schon über die Stelle der Änderungsoperation hinaus fortgeschritten ist, wird die Datenflussprüfung auf Instanzebene gar nicht aufgerufen.
Der Vollständigkeit halber wird die Zustandsüberprüfung innerhalb der Funktion *ILDataFlowValid* trotzdem durchgeführt, da sie innerhalb der Datenfluss-Analyse selbst von Bedeutung ist. Des Weiteren ist es möglich, dass sie bei einer Erweiterung des Migrationskonzeptes auf Aggregate von Änderungsoperationen in der gezeigten Form benötigt wird.

Besonderheiten

Es sei an dieser Stelle auf ein Problem hingewiesen, das die Umsetzung des Datenfluss-Konzeptes, wie es oben erklärt wurde, etwas komplizierter macht. Die Suchalgorithmen *FindWritingActivitiesBackward* und *FindReadingActivitiesForward* laufen auf dem geänderten Prozessschema, sie beginnen ihre Suche an der Stelle der geänderten Aktivität. Bei Einfügeoperationen ist dies problemlos möglich. Wird aber eine Aktivität aus dem Schema gelöscht, ergibt sich der Startpunkt des Suchalgorithmus nicht mehr intuitiv.
Wir haben die Problematik, dass Änderungsoperationen auf einfachen Aktivitäten größere Änderungen der umgebenden Elemente nach sich ziehen können, bereits in Kapitel 5.4.2 besprochen. Dieses Problem kommt auch hier, bei der Durchführung von Löschoperationen, zum Tragen. Als Startpunkt der Vorwärtssuche nach Leseaktivitäten muss diejenige einfache oder strukturierte Aktivität ausgewählt werden, die der gelöschten einfachen Aktivität vorangeht und die – nach Durchführung der Änderungsoperation – im neuen Schema noch vorhanden ist. Der zweite Aspekt ist der kritische Punkt. Wir verdeutlichen dies an einem Beispiel: In Abbildung 5.26 soll die Schreibaktivität gelöscht werden Hierfür ist es notwendig festzustellen, ob im weiteren Prozessverlauf eine Leseaktivität existiert. Das Löschen der einfachen Schreib-Aktivität führt gleichzeitig zur Löschung des *sequence*-Elements aus der Prozessdefinition, da dieses nun nicht mehr benötigt wird.

Eine direkte *einfache* Vorgängeraktivität der zu löschenden Aktivität existiert nicht. Die direkte *strukturierte* Vorgängeraktivität, das *sequence*-Element, wird mitgelöscht. Also ist das Vorgängerelement des *sequence*-Elementes zu bestimmen. Da auch hier keine vorangehende einfache Aktivität vorhanden ist, nimmt das *if*-Element die Rolle des Vorgängerelementes ein.
Bei strukturierten Vorgängerelementen ist es jetzt aber nicht möglich, das Nachfolgeelement zu bestimmen und dieses als Startaktivitäts-Parameter an die Suchfunktion zu übergeben. Würde man so vorgehen, würde im Beispiel die Leseaktivität nicht gefunden werden, da sich diese *innerhalb* des *if*-Elementes befindet und nicht diesem nachfolgt.

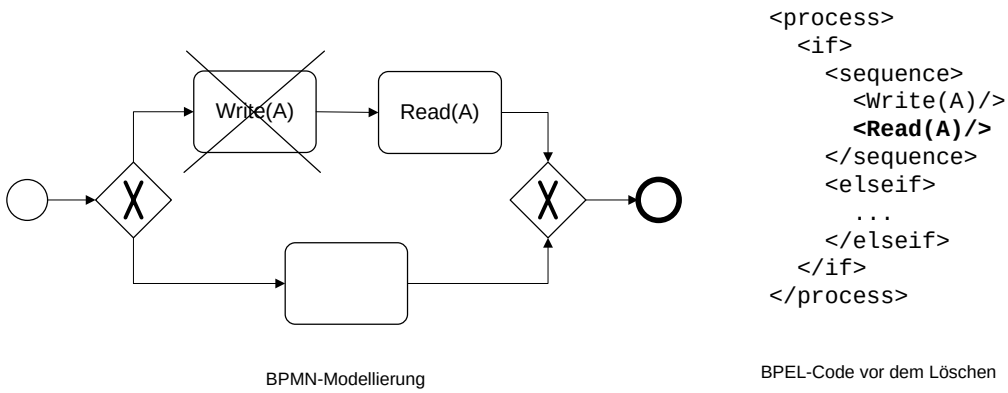


Abbildung 5.26: Finden der Startaktivität für die Datenflussanalyse bei Löschoperationen

Vor Aufruf des *FindReadingActivitiesForward*-Algorithmus müsste zunächst eine vorgeschaltete Suchfunktion prüfen, ob Leseaktivitäten *innerhalb* der gefundenen strukturierten Vorgängeraktivität vorhanden sind, bevor der eigentliche Suchalgorithmus über den gesamten nachfolgenden Prozessabschnitt aufgerufen wird. Dieser würde seine Suche mit der strukturieren Aktivität als Startaktivität beginnen.
Eine allgemein gehaltene Umsetzung dieses Konzeptes würde tiefergehendere Analysen der Fälle erfordern, die bei Löschoperationen auftreten können. Die erläuterte Problematik entspricht weitgehend den in Kapitel 5.4.2 angesprochenen Schwierigkeiten, die dort bei Fall 3 auftreten. Deshalb beschränken wir uns in dieser Arbeit – analog zu den Einfügeoperationen – auch bei den Löschoperationen auf das Löschen einer einfachen Aktivität zwischen zwei anderen einfachen Aktivitäten heraus. Dadurch ist sichergestellt, dass das Vorgängerelement der zu löschenden Aktivität eine einfache Aktivität ist und der Datenfluss-Prüfalgorithmus an dieser Stelle direkt beginnen kann.

Ein weiterer Punkt, auf den wir kurz hinweisen möchten, betrifft die Umsetzung der eigentlichen Adaptionsoperationen, wie sie im weiteren Verlauf dieses Kapitels aufgezeigt werden. Die Datenflussüberprüfung auf Instanzebene wird hier nicht bei allen Änderungsoperationen auf gleiche Weise verwendet.
Bei der von uns betrachteten Einfügeoperation wird die Funktion *ILDataFlowValid* auf die *neue*, die migrierte Instanz angewendet, ebenso wie die Datenflussanalyse auf Schemaebene auf dem neuen Prozessschema läuft. Bei der Löschoperation wird der *ILDataFlowValid*-Algorithmus aber auf die *alte* Instanz angewendet. Dies ergibt sich direkt aus seiner Funktionalität. Für Löschoperationen analysiert er – neben dem Ergebnis des *SLDataFlowValid*-Algorithmus – den Ausführungszustand der zu löschenden Aktivität. Diese ist in der migrierten Instanz logischerweise nicht mehr vorhanden, deshalb muss der Algorithmus hier auf der alten Instanz laufen.[18]
Für später neu hinzukommende Adaptionsoperationen ist die sinnvolle Anwendung auf die jeweils benötigten Instanzarten zu ermitteln.

Einschränkungen

Keinerlei Beachtung fanden in unserem Konzept zur Datenfluss-Überprüfung Compensation-, Event-, Fault- und Termination-Handler, die innerhalb von *scope*-Aktivitäten eingebunden werden können. Auch hier kann es zu Datenflussproblemen innerhalb der Ereignis-, Fehler-,

[18] Die Anwendung der Instanzebenen-Prüffunktion bei Einfügeoperationen auf die neue Instanz ist willkürlich gewählt, da ihr Ergebnis in diesem Fall gar nicht von der Instanz abhängig ist. Wir haben uns für diese Art der Benutzung entscheiden, da sie uns intuitiver erscheint.

Kompensations- oder Terminierungsroutinen kommen, falls dort Datenzugriffe erfolgen. Eine Integration dieser Konstrukte in die bestehenden Algorithmen ist unserer Einschätzung nach möglich, führt aber zu einer deutlich höheren Komplexität.

Ebenso wurden in unserem Konzept keine *extensionActivities* eingebunden. Diese müssten implementierungsabhängig eingepflegt werden.

In BPEL ist die dynamische Zuweisung von PartnerLink-Endpoints mit Hilfe von *assign*-Aktivitäten möglich. Auch hier kann ein Problem auftreten, wenn z.B. die Endpoint-Zuweisungsaktivität gelöscht wird. Wir beschränken uns in unserer Betrachtung aber auf Variablen-Zugriffe.

Die BPEL-Spezifikation sieht im Rahmen mancher Aktivitäten den expliziten Zugriff auf einzelne Variablenteile vor. Dies betrifft Variablen in Form von Nachrichten; hier können einzelne WSDL-Message-Parts direkt zugegriffen werden. Im Rahmen des oben vorgestellten Konzeptes werden aus Vereinfachungsgründen nur Zugriffe auf ganze Variablen betrachtet. Eine Erweiterung auf die Behandlung von Variablenteilen ist aber ohne weiteres möglich.

5.4.5 Adaptionsprimitive auf BPEL-/Instanzebene

Bei den auf der BPEL bzw. Instanzebene-Ebene benötigten Operationen geht es nicht um Änderungen am Prozessmodell, sondern um Konsistenz-Überprüfungen und Zustandsanpassungen. Wir sprechen deshalb auf der Instanzebene von *Adaptions*primitiven, in Abgrenzung zur BPMN-Ebene, wo *Änderungs*funktionen definiert wurden, um die gewünschten Änderungen an der Prozessstruktur vornehmen zu können.
Die genaue Implementierung der Adaptionsprimitive wird im Anhang in ausführlicher Pseudocode-Darstellung aufgezeigt (siehe Anhang C.1). Wir geben dem Leser hier nur eine kurzgefasste Übersicht über die Funktionalität der einzelnen Primitive.

A – Ausführungszustandsüberprüfung

Die folgenden Adaptionsprimitive werden für die Ermittlung des Instanz-Ausführungszustandes benötigt.

GetProcessStatus

Stellt anhand des Zustandes der genannten Aktivität fest, ob sich die Instanz noch in einem migrierbaren Ausführungszustand befindet, oder ob die Ausführung bereits zu weit fortgeschritten ist.

Aufruf: *bool* GetProcessStatus(*ProcessIdentifier*, *ActivityName*, *InstanceIdentifier*)

GetActivityStatus

Liefert den Status der Aktivität mit dem Namen *ActivityName*.

Aufruf: *ActivityStatus* GetActivityStatus(*ActivityName*, *InstanceIdentifier*)

B – Datenfluss-Analyse

Die nachfolgend aufgeführten Funktionen werden für die Überprüfung der Datenfluss-Korrektheit benötigt.

GetVariablesAccessed

Überprüft die angegebene Aktivität auf Datenelement-Zugriffe. Gibt für jeden Zugriff den Variablennamen und den Zugriffsstyp (*read* oder *write*) zurück.

Aufruf: {(*VariableName*, *VariableAccess*)} GetVariablesAccessed(*ProcessIdentifier*, *ActivityName*, *ActivityType*)

GetDataAccessType

Bestimmt in Abhängigkeit von der Art der Variablenzugriffe den Datenzugriffs-Typ der Aktivität, also *readActivity*, *writeActivity*, *readWriteActivity* oder *noVariableAccessActivity*.

Aufruf: *DataAccessType* GetDataAccessType({(*VariableName*, *VariableAccess*)})

SLDataFlowValid

SLDataFlowValid stößt, abhängig von der vorzunehmenden Änderungsoperation und dem Datenzugriffstyp der betroffenen Aktivität, die jeweils nötige Form der Datenflussüberprüfung an. Hierzu ruft er in der jetzigen Ausbaustufe gegebenenfalls die Suchalgorithmen *FindWritingActivitiesBackward* und *FindReadingActivitiesForward* auf. Zurückgegeben wird die Menge der Variablen, die aufgrund fehlender sicherer Initialisierungen zu Datenfluss-Problemen führen. (Siehe hierzu auch Abbildung 5.15 und die zugehörige Erläuterung.)

Aufruf: *{CriticalVariableName}* SLDataFlowValid(*ProcessIdentifier*, *ActivityName*, *DataAccessType*, *ModificationOperation*, {(*VariableName*, *VariableAccess*)})

ILDataFlowValid

ILDataFlowValid greift auf die Ergebnisse der Datenflussüberprüfung auf Schemaebene zurück und ermittelt die Korrektheit des Datenflusses auf Instanzebene. Der aktuelle Ausführungszustand der zu migrierenden Instanz spielt hierbei die entscheidende Rolle (siehe Abb. 5.25).

Aufruf: *bool* ILDataFlowValid(*ProcessIdentifier*, *InstanceIdentifier*, *ActivityName*, *DataAccessType*, *ModificationOperation*, {*CriticalVariableName*})

FindReadingActivitiesForward

Die Funktion *FindReadingActivitiesForward* überprüft auf Schemaebene, ob im Prozessablauf nach der Aktivität *ActivityName* ein Lesezugriff auf die angegebenen Variablen erfolgt. Die Funktion liefert ein Tupel, das für jede Variable die erste gefundene Leseaktivität enthält. Wurde keine Leseaktivität gefunden, nimmt das zugehörige Aktivitätselement des Tupels den Wert EMPTY an.
Die eigentliche Suche nach Leseaktivitäten übernimmt der *SucceedingReader*-Algorithmus. Dieser wird für jede Variable einmal aufgerufen.

Aufruf: {(*ActivityName*, *VariableName*)} FindReadingActivitiesForward(*ProcessIdentifier*, *ActivityName*, {*VariableName*})

FindWritingActivitiesBackward

Überprüft auf Schemaebene, ob die angegebenen kritischen Variablen vor dem Lesezugriff der Aktivität *ActivityName* geschrieben werden. Hierzu wird für jede Variable der Suchalgorithmus *PrecedingWriter* aufgerufen.

Die Funktion gibt ein Tupel zurück, das für jede Variable die zugehörige Schreibaktivität nennt. Existiert keine Schreibaktivität, bleibt das Aktivitätselement im Tupel leer.

Aufruf: {(*ActivityName*, *VariableName*)} FindWritingActivitiesBackward(*ProcessIdentifier*, *ActivityName*, {*VariableName*})

SucceedingReader

Führt eine Suche nach einer Leseaktivität auf die übergebene Variable durch, die der Aktivität *ActivityName* nachfolgt. Der Algorithmus gibt den Namen der gefundenen Leseaktivität zurück, sofern eine solche existiert. (Siehe Abbildung 5.22.)

Aufruf: *ActivityName* SucceedingReader(*ProcessIdentifier*, *ActivityName*, *VariableName*)

PrecedingWriter

PrecedingWriter führt eine Suche nach einer der Aktivität *ActivityName* vorangehenden Schreibaktivität durch (siehe Abb. 5.17). Der Algorithmus liefert den Namen der Schreibaktivität, wenn eine solche vorhanden ist.

Aufruf: *ActivityName* PrecedingWriter(*ProcessIdentifier*, *ActivityName*, *VariableName*)

ReaderDeepSearch

ReaderDeepSearch führt eine Tiefensuche in der übergebenen Aktivität durch, mit dem Ziel, eine auf *VariableName* schreibende Aktivität zu finden. Je nach Typ der Leseaktivität wird ihr Name oder der des umschließenden Elements zurückgegeben. Auf den Algorithmus sind wir in Kapitel 5.4.4 im Abschnitt *Vorwärtssuche nach Leseaktivitäten (FindReadingActivitiesForward)* bereits eingegangen.

Aufruf: *ActivityName* ReaderDeepSearch(*ProcessIdentifier*, *ActivityName*, *VariableName*)

WriterDeepSearch

Der Algorithmus führt eine Tiefensuche in der übergebenen Aktivität durch. Hierbei wird nach einem sicheren Schreibzugriff auf die angegebene Variable gesucht. Zurückgegeben wird – je nach Aktivitätstyp – der Name der Schreibaktivität bzw. der des umschließenden Elements. (Siehe hierzu Abschnitt *Rückwärtssuche nach Schreibaktivitäten* im vorangegangenen Kapitel für eine ausführliche Erläuterung.)

Aufruf: *ActivityName* WriterDeepSearch(*ProcessIdentifier*, *ActivityName*, *VariableName*)

ReadingActivity

Mit der Funktion *ReadingActivity* lässt sich überprüfen, ob die betrachtete Aktivität die angegebene Variable liest.

Aufruf: *bool* ReadingActivity(*ProcessIdentifier*, *ActivityName*, *ActivityType*, *VariableName*)

WritingActivity

WritingActivity stellt fest, ob in der Aktivität ein Schreibzugriff auf *VariableName* stattfindet.

Aufruf: *bool* WritingActivity(*ProcessIdentifier*, *ActivityName*, *ActivityType*, *VariableName*)

C – Instanzen-Migration

Für die Durchführung der Instanzen-Migration greifen wir auf folgende Primitive zurück:

MigrateInstance

Kopiert alle Variablenwerte und Aktivitätszustände von der Instanz mit dem Identifier *SourceInstanceIdentifier* auf die Instanz mit dem Identifier *TargetInstanceIdentifier*.
Die Werte der Variablen bzw. der Zustände, die in der neuen Instanz (*TargetInstanceIdentifier*) nicht vorkommen, werden nicht beachtet. Die Variablen, die in der alten Instanz nicht vorhanden sind, werden in der neuen Instanz im nichtinitialisierten Zustand belassen. Die Ausführungszustände der neu hinzugekommenen Aktivitäten werden an den Ausführungsstatus der Instanz angepasst.

Aufruf: MigrateInstance(*ProcessIdentifier*, *ActivityName*, *SourceInstanceIdentifier*, *TargetInstanceIdentifier*, *ModificationOperation*)

DetermineNewActivityStatus

Diese Funktion bestimmt den Ausführungszustand, der einer neu eingefügten Aktivität zugewiesen werden muss. Hierfür werden zunächst die Zustände der umgebenden Aktivitäten abgefragt. An diese Ausführungszustände wird der Status der neu hinzugekommenen Aktivität angepasst. Das genaue Vorgehen ist von der Adaptionsoperation abhängig.
Bisher unterstützt der Algorithmus nur die Operationen *SerialInsertionOfABasicActivity* und *SerialRemovalOfABasicActivity*, wobei im letzteren Fall keine Zustandsanpassung notwendig ist. Für weitere, zukünftig einzubindende Adaptionsoperationen kann dieser Algorithmus recht komplex werden, wie im Kapitel 5.4.2 aufgezeigt wurde.

Aufruf: *ActivityStatus* DetermineNewActivityStatus(*ProcessIdentifier*, *ActivityName*, *InstanceIdentifier*, *ModificationOperation*)

setActivityStatus

Setzt für die Instanz *InstanceIdentifier* den Ausführungszustand der Aktivität *ActivityName* auf *ActivityStatus*. Dies betrifft die Instanz-Zustandshaltung innerhalb der Workflow-Engine.

Aufruf: setActivityStatus(*ActivityName*, *InstanceIdentifier*, *ActivityStatus*)

D – sonstige Primitive

Die nachfolgend aufgeführten Adaptionsprimitive sind allgemeiner Art und werden an verschiedenen Stellen unseres Adaptionskonzeptes benötigt.

GetActivityType

Liefert den BPEL-Aktivitätstyp der Aktivität zurück, also beispielsweise *assign*, *invoke* usw. Das *process*-Element kann auch zurückgegeben werden, obwohl dies keine Aktivität im eigentlichen Sinne ist.

Aufruf: *ActivityType* GetActivityType(*ProcessIdentifier*, *ActivityName*)

GetInitializationStatus

Stellt fest, ob die Variable *VariableName* bereits initialisiert wurde.

Aufruf: *bool* GetInitializationStatus(*InstanceIdentifier*, *VariableName*)

GetPrecedingElement

Bestimmt das vorangehende BPEL-Element auf XML-Ebene, also entweder eine Aktivität oder das Start-Tag der umgebenden strukturierten Aktivität. Zurückgegeben wird der Typ des Vorgängerelementes und sein Name. (Siehe Abbildung C.5 im Anhang).

Aufruf: (*precElementType*, *precElementName*) GetPrecedingElement(*ProcessIdentifier*, *ActivityName*)

GetSucceedingElement

Bestimmt das nachfolgende BPEL-Element auf XML-Ebene, also entweder eine Aktivität oder das End-Tag der umgebenden strukturierten Aktivität. Zurückgegeben wird der Typ des Nachfolgeelementes und sein Name. (Abbildung C.6).

Aufruf: (*succElementType*, *succElementName*) GetSucceedingElement(*ProcessIdentifier*, *ActivityName*)

UserInformation

Informiert den Benutzer über das Ergebnis der Schema- und der Instanzprüfung bzgl. Datenflusskorrektheit für die jeweilige Instanz.

Aufruf: UserInformation(*InstanceIdentifier*, *SLDataFlowValid*, *ILDataFlowValid*)

UserDecisionPerformMigration

Fragt den Benutzer, ob er die Migration durchführen möchte.

Aufruf: *bool* UserDecisionPerformMigration()

5.4.6 Umsetzung der Adaptionsoperationen

Auf Basis der Adaptionsprimitive definieren wir nun die eigentlichen Adaptionsoperationen. Im Gegensatz zur BPMN-Ebene ist es hier nicht notwendig, beim Einfügen oder Löschen nach Task-Typ, wie z.B. *Service* oder *Receive*, zu unterscheiden. Der BPEL-Aktivitätstyp und die mit der Änderungsoperation verbundenen Auswirkungen werden von der Adaptionsoperation automatisch erkannt.

Die Benutzereinbindung so wie wir sie ansatzweise aufzeigen, mit einer Datenfluss-Mitteilung und einer Migrationsabfrage für jede Instanz einzeln, würde man bei einer Implementierung natürlich aggregieren. Aus Vereinfachungsgründen begnügen wir uns mit der gezeigten Version, die die nötigen Datenfluss-Überprüfungen und Benutzer-Entscheidungen übersichtlich aufzeigt.

A. Serielles Einfügen einer einfachen Aktivität

Der Algorithmus prüft jeweils, ob sich die angegebenen Instanzen in einem migrierbaren Ausführungszustand befinden. Ist dies der Fall, wird die Instanz testweise auf das neue Prozessschema migriert. Auf Schema- wie auch auf Instanzebene wird die Korrektheit des Datenflusses überprüft. Die Datenflusskorrektheit ist erwünscht, aber nicht zwingend notwendig. Der Benutzer erhält die notwendigen Informationen hierüber und entscheidet letztendlich, ob die einzelnen Instanzen endgültig migriert werden sollen oder nicht.

Aufruf:

BasicActivitySerialInsert(*NewProcessIdentifier*, *ActivityName*, {(*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)})

Parameter:

string *NewProcessIdentifier*
string *ActivityName*
string *CurrentInstanceIdentifier*
string *NewInstanceIdentifier*

Vorbedingungen:

– *CurrentInstanceIdentifier*, *NewInstanceIdentifier* ∈ Instances
– Alle *CurrentInstanceIdentifier* sind Instanzen desselben Schemas und alle *NewInstanceIdentifier* sind Instanzen desselben Schemas.
– Das Schema der *NewInstanceIdentifiers* ist das des *NewProcessIdentifiers*.

Semantik:

begin

string currentInstance
string newInstance
string[ ] SLDataFlowValid
bool ILDataFlowValid
string modificationOperation
string actType
bool migratableStatus

modificationOperation := "SerialInsertionOfABasicActivity"
SLDataFlowValid := EMPTY
ILDataFlowValid := false

actType := GetActivityType(*NewProcessIdentifier*, *ActivityName*)
{(VarName, VarAccess)} := GetVariablesAccessed(*NewProcessIdentifier*, *ActivityName*, actType)
DAType := GetDataAccessType({(VarName, VarAccess)})
SLDataFlowValid := SLDataFlowValid(*NewProcessIdentifier*, *ActivityName*, DAType, modificationOperation, {(VarName, VarAccess)})

for each i ∈ {(*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)} **do**

currentInstance := i.CurrentInstanceIdentifier
newInstance := i.NewInstanceIdentifier
MigrateInstance(*NewProcessIdentifier*, *ActivityName*, currentInstance, newInstance, modificationOperation)
migratableStatus := GetProcessStatus(*NewProcessIdentifier*, *ActivityName*, newInstance)

if migratableStatus = true **then**
ILDataFlowValid := ILDataFlowValid(*NewProcessIdentifier*, newInstance, *ActivityName*, DAType, modificationOperation, SLDataFlowValid)
UserInformation(currentInstance, SLDataFlowValid, ILDataFlowValid)
performMigration := UserDecisionPerformMigration()
else
performMigration = false
end if

if performMigration = true **then**
neue Instanz behalten und weiterlaufen lassen, alte Instanz stoppen
else
neue Instanz löschen, alte Instanz weiterlaufen lassen
end if

end for

end

B. Serielles Löschen einer einfachen Aktivität

Der Löschalgorithmus führt zunächst eine Datenflussanalyse auf dem neuen Prozess-Schema durch. Da der gelöschte Task dort nicht mehr existiert, werden zuvor dessen Variablenzugriffe und sein Vorgängerelement auf Basis des alten Prozess-Schemas ausgelesen. Diese Informationen werden dem *SLDataFlowValid*-Algorithmus als Parameter übergeben.
Befindet sich die Instanz in einem migrierbaren Ausführungszustand, wird die Datenflusskorrektheit auf Instanzebene geprüft – in diesem Fall aber die der alten Instanz. Die Gründe hierfür wurden in Kapitel 5.4.4 aufgezeigt.
Die Ergebnisse der beiden Datenfluss-Überprüfungen werden dem Benutzer mitgeteilt. Entscheidet sich dieser für eine Migration, wird die bisherige Instanz auf das neue Prozess-Schema migriert und die alte gelöscht. Entscheidet er sich dagegen, wird die bisherige Instanz beibehalten.

Aufruf:

BasicActivitySerialRemove(*CurrentProcessIdentifier*, *NewProcessIdentifier*, *ActivityName*, {(*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)})

Parameter:

string *CurrentProcessIdentifier*
string *NewProcessIdentifier*
string *ActivityName*
string *CurrentInstanceIdentifier*
string *NewInstanceIdentifier*

Vorbedingungen:

– *CurrentInstanceIdentifier*, *NewInstanceIdentifier* ∈ Instances
– Alle *CurrentInstanceIdentifier* sind Instanzen desselben Schemas und alle *NewInstanceIdentifier* sind Instanzen desselben Schemas.
– Das Schema der *CurrentInstanceIdentifiers* ist das des *CurrentProcessIdentifiers*. Das Schema der *NewInstanceIdentifiers* ist das des *NewProcessIdentifiers*.

Semantik:

begin

string currentInstance
string newInstance
string[ ] SLDataFlowValid
bool ILDataFlowValid
string modificationOperation
string actType
bool migratableStatus
string precElementType
string precElementName

modificationOperation := "SerialRemovalOfABasicActivity"
SLDataFlowValid := EMPTY
ILDataFlowValid := false

actType := GetActivityType(*CurrentProcessIdentifier*, *ActivityName*)
{(VarName, VarAccess)} := GetVariablesAccessed(*CurrentProcessIdentifier*, *ActivityName*, actType)
DAType := GetDataAccessType({(VarName, VarAccess)})
(precElementType, precElementName) := GetPrecedingElement(*CurrentProcessIdentifier*, *ActivityName*)
SLDataFlowValid := SLDataFlowValid(*NewProcessIdentifier*, precElementName, DAType, modificationOperation, {(VarName, VarAccess)})

for each i ∈ {(*CurrentInstanceIdentifier*, *NewInstanceIdentifier*)} **do**

currentInstance := i.CurrentInstanceIdentifier
newInstance := i.NewInstanceIdentifier
migratableStatus := GetProcessStatus(*CurrentProcessIdentifier*, *ActivityName*, currentInstance)

if migratableStatus = true **then**
ILDataFlowValid := ILDataFlowValid(*CurrentProcessIdentifier*, currentInstance, *ActivityName*, DAType, modificationOperation, SLDataFlowValid)
UserInformation(currentInstance, SLDataFlowValid, ILDataFlowValid)
performMigration := UserDecisionPerformMigration()
else
performMigration = false
end if

if performMigration = true **then**
MigrateInstance(*NewProcessIdentifier*, *ActivityName*, currentInstance, newInstance, modificationOperation)

neue Instanz behalten und weiterlaufen lassen, alte Instanz stoppen
else
neue Instanz löschen, alte Instanz weiterlaufen lassen
end if

end for

end

5.5 Zusammenfassung

In diesem Kapitel haben wir ein Konzept aufgezeigt, wie benutzerinitiierte Änderungen an einem BPMN-Prozessmodell auf bereits laufende Prozessinstanzen propagiert werden können. Hierfür war es notwendig, ein Vorgehen zu entwickeln, das zu einem durchgängigen Adaptionspfad führt. Dies konnte durch die Auftrennung der Adaptionsoperation in eine BPMN-Änderungs- und eine Instanzänderungsoperation erreicht werden.
Die Entwicklung der Adaptionsoperationen erfolgte in einem mehrschichtigen Modell. Zunächst wurde sowohl für BPMN wie auch für die Instanzebene ein formales Modell definiert. Auf diesem Modell basieren die möglichst allgemein gehaltenen Änderungs- bzw. Adaptions*primitive*. Bei der Definition der eigentlichen Adaptions*operationen* haben wir wiederum auf diese Primitive zurückgegriffen.

Randbedingung bei der Entwicklung der Operationen war der Erhalt der Korrektheit der Prozessmodelle sowie die Gewährleistung der Konsistenz auf Instanzebene. Wir haben diese Aspekte zunächst auf theoretischer Ebene ausführlich analysiert und sie dann anschließend in den Entwurf der Adaptionsoperationen mit einfließen lassen.

5.6 Nicht betrachtete Fragestellungen

Wir möchten kurz einige Fragen aufzeigen, deren Behandlung wir im Rahmen unseres Adaptionskonzeptes bewusst ausgeklammert haben.

Diese betreffen insbesondere Aspekte auf anderen Systemebenen, wie beispielsweise die Integration eines Rechtemanagements. In der Praxis sollte sicherlich nicht jeder Benutzer die Möglichkeit haben, auf beliebige Prozessmodelle im Repository zuzugreifen und Veränderungen an ihnen wie auch an laufenden Instanzen vorzunehmen. Dieser Punkt bedarf der Entwicklung eines ausgeklügelten Rechtemanagements. Eventuell können Änderungsrechte an einer Instanz beispielsweise nicht nur für Administratoren verfügbar sein, sondern auch – dann aber begrenzt auf bestimmte Prozessabschnitte – einzelnen Prozessteilnehmern zugewiesen werden [Rei00].

Ebenfalls mit der Benutzerebene verknüpft ist die Frage, ob eine laufende Prozessinstanz überhaupt zu jedem beliebigen Zeitpunkt angehalten werden kann, um eine Migration durchzuführen. Kann eine System- oder Benutzer-Interaktion mit dem Prozess jederzeit abgebrochen werden, ohne dass Daten verloren gehen oder zeitkritische Tasks im Geschäftsablauf gestoppt werden müssen? Dieser Aspekt ist für den realen Einsatz eines Business-Process-Management-Systems mit Adaptionsunterstützung von erheblicher Bedeutung.
In unserem Konzept gehen wir vereinfachend davon aus, dass die Ausführungsmaschine vor dem Starten des Migrationsalgorithmus angehalten wird, sich in einem konsistenten Zustand befindet und nach der Migration aller Instanzen wieder fortgesetzt wird.

Auf Prozessebene haben wir Compensation Handler und Schleifen in unserer Betrachtung ausgeklammert. Es müsste geklärt werden, welche Änderungen am Adaptionskonzept notwendig

sind, wenn man beide Elemente mit einbezieht. Das Zulassen von Schleifen führt eventuell dazu, dass es sinnvoll ist, auch an schon ausgeführten Prozessabschnitten Änderungen zuzulassen. Gleiches gilt für die Miteinbeziehung von Compensation Handlern. Hierbei ist es auch notwendig zu erarbeiten, inwieweit Änderungen am Prozessmodell und mögliche Zurücksetzungen von Instanzen miteinander kompatibel sind. Wenn eine Instanz zurückgesetzt wird und zwischenzeitlich eine Änderungsoperation an schon ausgeführten Aktivitäten erfolgt, wird die Prozessinstanz dann beim erneuten Durchlaufen dieses Bereiches auf Basis der alten oder des neuen Prozessmodells fortgeführt?

Diese und einige weitere, in den jeweiligen Kapiteln angesprochene Fragen konnten wir im Rahmen dieser Arbeit nicht behandeln.

Kapitel 6

Implementierung

Im vorangehenden Kapitel haben wir ein umfangreiches Adaptionskonzept erarbeitet. Dieses soll in Form einer eigenständigen Komponente implementiert werden.

6.1 Implementierungskonzeption

Wir müssen nun unser Adaptionskonzept auf eine geeignete Systemarchitektur umsetzen. Hierbei gilt es, die Funktionalität sinnvoll auf die beteiligten System-Bausteine abzubilden. Daran anschließend können die detaillierten funktionalen und technischen Anforderungen an die einzelnen Komponenten herausgearbeitet werden. Diese Beschreibung dient als Basis für die eigentliche Programmierung.

6.1.1 Funktionalitäts-Spezifikation

Die Funktionalitätsanforderungen die an unsere Adaptions-Komponente gestellt werden, haben wir bereits im Konzeptions-Kapitel detailliert herausgearbeitet. Abhängig von der durchzuführenden Adaptionsoperation muss diese im Wesentlichen

- die entsprechende Änderung des Prozess-Schemas auf BPMN-Ebene anstoßen.
- prüfen, ob sich die entsprechenden Instanzen in einem migrierbaren Ausführungszustand befinden.
- prüfen, ob die Korrektheit des Datenflusses durch die Migration beeinträchtig wird.
- die Migration der Instanzen durchführen.

Analyseszenario

Als Grundlage für die Erarbeitung einer Funktionalitätsbeschreibung der beteiligten Systemkomponenten betrachten wir zunächst ein allgemein gehaltenes Integrationsszenario. Dieses veranschaulicht, wie unsere Adaptionskomponente in eine Prozess-Management-Systemarchitektur eingebunden werden kann. Anhand dieses Modells lässt sich intuitiv eine sinnvolle Aufteilung der benötigten Gesamtfunktionalität auf die einzelnen Systemkomponenten erarbeiten. Aus dieser Aufteilung lassen sich wiederum unmittelbar die benötigten Schnittstellen herleiten.

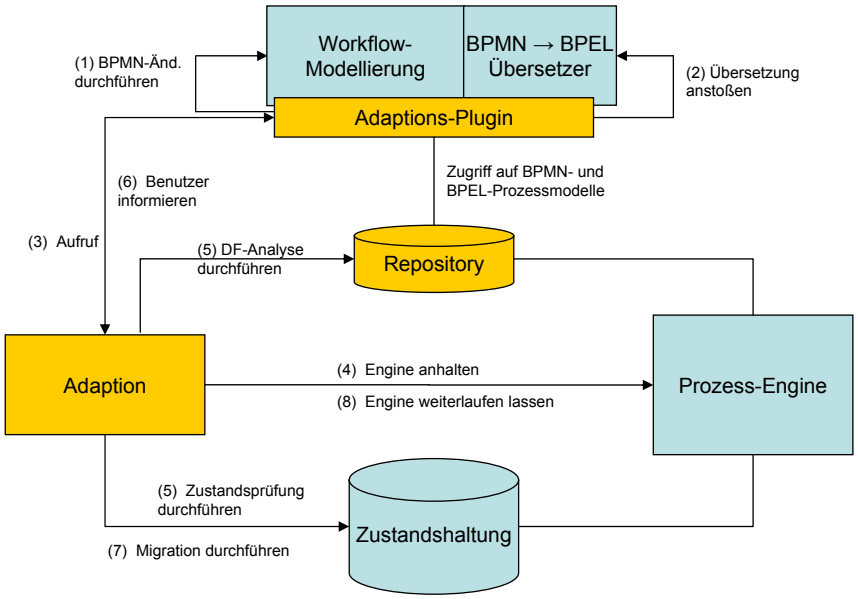


Abbildung 6.1: Integrations-Szenario

Abbildung 6.1 zeigt die wichtigsten Bausteine einer typischen BPM-Systemarchitektur mit der neu hinzugekommenen *Adaptionskomponente*. Unsere Adaptionskomponente ist die zentrale Steuerungseinheit für die Durchführung von Adaptionsoperationen. Sie kommuniziert sowohl mit der Benutzerebene, wo die gewünschten Änderungen am Prozess-Schema vorgenommen werden, als auch mit den Komponenten der Ausführungsebene, auf der die eigentliche Prozessausführung erfolgt.

Unsere Adaptionslösung umfasst neben der Hauptkomponente einen zusätzlichen Baustein. Es wird ein Modul benötigt, das zum einen die Kommunikation mit dem Benutzer ermöglicht und zum anderen die gewünschten Änderungen am Prozess-Schema durchführt. Wie gehen hier von einem *Adaptions-Plugin* im BPMN-Modellierungswerkzeug aus, das diese Aufgaben übernimmt. Es wäre unserer Ansicht nach nicht sinnvoll, diese beiden Funktionalitäten in die zentrale Adaptionskomponente zu integrieren. Der Benutzer ist es gewohnt, Prozessmodelle in seinem Modellierungstool zu bearbeiten. Er wird also auch erwarten, dass Prozess-Änderungen mit diesem Werkzeug vorzunehmen sind. Deshalb bietet es sich schon aus Gründen der Ergonomie an, die nötige Funktionalität hinsichtlich einer Benutzerunterstützung an dieser Stelle einzubinden.
Der andere Vorteil bei einer Kapselung der BPMN- bzw. Benutzeraspekte in einem separaten Plugin ist, dass keine direkte Schnittstelle für die BPMN-Modellierung benötigt wird. Die Änderungen am Prozessmodell erfolgen innerhalb des Modellierungstools über die schon vorhandenen Funktionen, wie z.B. "Task einfügen", die auch für Neumodellierungen bzw. nicht-adaptive Modelländerungen benötigt werden. Es genügt dann, dass der zentralen Adaptionskomponente nur noch die auf BPMN-Ebene bereits erfolgte Änderungsoperation mitgeteilt wird. In die BPMN-Modellierung selbst muss die Adaptionskomponente somit gar nicht integriert werden.

Ein wesentlicher Bestandteil einer Adaptions-Architektur stellt das *Repository* dar, das die BPMN- und BPEL-Prozessmodelle enthält und eine Versionierung dieser Modelle unterstützt. Auf dieses Repository greift der Benutzer über sein Modellierungswerkzeug zu, wenn er beabsichtigt, Änderungen an BPMN-Prozessen vorzunehmen. Nach der Übersetzung eines Prozessmodells nach BPEL wird dieses ebenfalls im Repository abgelegt, wo die Prozess-Engine darauf zugreifen kann. Aber auch die Adaptionskomponente benötigt die BPEL-Repräsentation zur

Durchführung verschiedener Konsistenz-Überprüfungen.
Gerade dieser Aspekt unseres Idealmodells einer Systemarchitektur kann in realen Systemen auch anders aussehen. Teilweise wird die BPEL-Prozessdefinition direkt an die Engine übergeben, die sie intern eventuell nochmals in eine andere Repräsentation übersetzt und dann in einem proprietären Format speichert.

Der Verzicht auf eine Autorisierung ist ein weiterer Punkt, in dem unser Integrationskonzept gegenüber wirklichen BPM-Systemen vereinfacht wurde. In einer realen Einsatzumgebung sollte detailliert definiert sein, welcher Benutzer Änderungen an welchen Prozessen vornehmen kann. Die Berechtigungsstufe kann auch vom Ausführungszustand der betroffenen Instanzen anhängig sein. Wir haben auf die Betrachtung des Autorisierungsaspektes in dieser Arbeit generell verzichtet. Andere parallel laufende Arbeiten im FloeCom-Forschungsprojekt beschäftigen sich mit dieser Frage. Ein hierbei entwickeltes Autorisierungskonzept sollte sich unserer Einschätzung nach ohne größere Schwierigkeiten nachträglich in das vorgestellte Adaptionskonzept integrieren lassen.

Abbildung 6.1 zeigt neben der reinen Architekturübersicht auch die wichtigsten Aufrufe, die bei der Durchführung einer Adaptionsoperation erfolgen:
Der Anstoß zur Durchführung einer Adaptionsoperation kommt vom Benutzer. Dieser lädt zunächst aus einem Repository das anzupassende Prozess-Schema. Daraufhin ruft er das Adaptionsplugin seines Prozess-Modellierungswerkzeuges auf und spezifiziert die auszuführende Änderungsoperation. Der weitere Ablauf erfolgt fast vollständig automatisiert.

1. Das Adaptionsplugin führt die gewünschten Änderungen am BPMN-Prozessmodell aus.
2. Anschließend stößt es die Übersetzung des BPMN-Prozesses nach BPEL an. Traten bei der Übersetzung keine Probleme auf, wird der BPEL-Prozess im Repository abgelegt.
3. Die neue Prozessdefinition ist jetzt sowohl von der Engine wie auch von der Adaptionskomponente aus zugreifbar. Das Adaptionsplugin des Modellierungswerkzeuges kann nun die zentrale Adaptionskomponente aufrufen, die die Steuerung des eigentlichen Adaptionsvorganges übernimmt.
4. Um eine konsistente Adaption von Prozessinstanzen durchführen zu können wird zunächst einmal die Prozess-Engine angehalten.[1]
   Danach ermittelt die Adaptionskomponente, welche Instanzen des betroffenen Schemas gerade ausgeführt werden und in welchem Status sich diese befinden. Die Informationen werden an den Benutzer übermittelt, der auf dieser Grundlage eine Auswahl derjenigen Instanzen trifft, die migriert werden sollen. Wir haben diesen eher unwesentlichen Punkt im Schaubild nicht dargestellt, um die Übersichtlichkeit zu erhöhen.
5. Nun ist es möglich festzustellen, ob sich die betroffenen Instanzen in einem migrierbaren Ausführungszustand befinden. Hierfür benötigt die Adaptionskomponente Zugriff auf die Datenbank der Ausführungsumgebung, in der die Zustände gespeichert sind.
   Des Weiteren muss überprüft werden, ob eine Migration zu Problemen bezüglich des Datenflusses führt. Hierfür werden die entsprechenden Aktivitätszustände benötigt, aber auch die BPEL-Prozessdefinition.
6. Nachdem bekannt ist, welche Instanzen ohne Schwierigkeiten migriert werden können, und welche Probleme bereiten, wird der Benutzer hierüber informiert und kann selbst entscheiden, für welche Instanzen eine endgültige Migration durchzuführen ist. Dies teilt er der Adaptionskomponente mit.

[1] Es würde ausreichen, die Ausführung der betroffenen Instanzen zu stoppen und die neue Erzeugung von Instanzen sowohl des alten wie auch des neuen Prozess-Schemas zu verhindern. So könnte eine Blockierung nicht betroffener Prozessinstanzen vermieden werden. Wir haben unsere Lösung in diesem Punkt stark vereinfacht – wissend, dass ein solches Vorgehen im praktischen Einsatz nicht anwendbar ist.

7. Diese führt daraufhin in ihrem Migrationsbaustein die Instanzenmigration durch.
8. Zuletzt muss der Engine mitgeteilt werden, welche Instanzversion – die neue oder die auf dem alten Prozess-Schema basierende – sie weiter ausführen soll. Nach Abschluss aller Adaptionsfunktionen wird die Prozess-Engine wieder gestartet.

Aus diesem Ablauf lassen sich direkt die grundlegenden Schnittstellen ableiten, über die die Adaptionskomponente verfügen muss. Abbildung 6.2 zeigt sie in einer Übersicht.

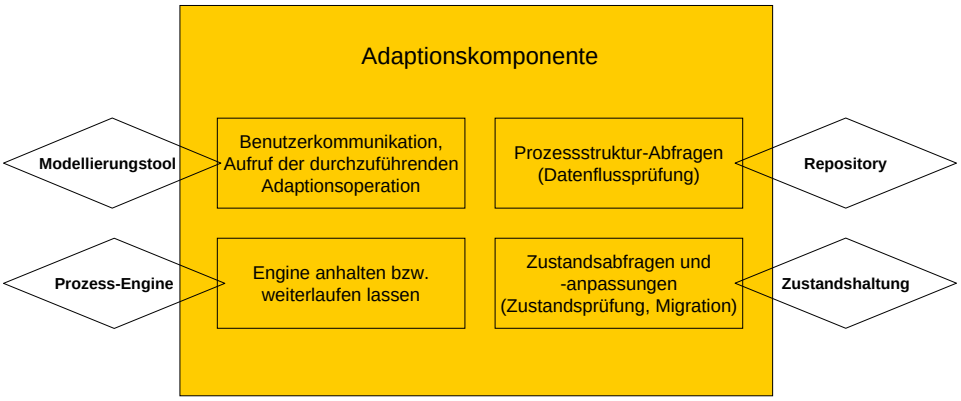


Abbildung 6.2: Adaptionskomponente – Funktionen und Schnittstellen

Im Wesentlichen müssen Schnittstellen zum Adaptions-Plugin des Modellierungswerkzeuges, zur Prozess-Engine, zum Prozess-Repository und zu den Instanz-Zustandsdaten existieren.
Über die Adaptionsplugin-Schnittstelle werden Benutzereingaben bzw. -ausgaben übermittelt. Die Schnittstelle zur Prozess-Ausführungsmaschine ermöglicht das Anhalten der Engine, um Zustandsabfragen und Instanzmigrationen unter Einhaltung eines konsistenten Ausführungszustandes durchführen zu können. Diese Zustandsabfragen werden über die Schnittstelle zur anwendungssystem-spezifischen Zustandsdatenhaltung ausgeführt. Des Weiteren ist eine Zugriffsmöglichkeit auf das Repository notwendig zur Ausführung von Abfragen, die die BPEL-Prozessstruktur betreffen.

Komponentenspezifische Anforderungen

Nachdem die Gesamtfunktionalität des Konzeptes sowie die Eingliederung der benötigten Komponenten in die Systemarchitektur erarbeitet wurden, lassen sich nun die detaillierten Implementierungs-Anforderungen an die einzelnen Komponenten definieren. Diese werden wir im folgenden Abschnitt aufzeigen.

A. Adaptions-Plugin für den BPMN-Modeler

Obwohl wir das Adaptions-Plugin selbst im Rahmen dieser Arbeit nicht implementieren werden, wollen wir auch für dieses die Implementierungsanforderungen darstellen, um eine spätere Umsetzung und Integration in die Systemarchitektur zu vereinfachen. Bezüglich des Funktionalitätsumfangs und des Bedienungskonzeptes des Modellierungstools gehen wir im Folgenden von dem *BPMN Modeler*[2] als Referenz-Komponente aus, da dieses Tool sowohl Bestandteil

[2] http://www.eclipse.org/bpmn/

des Intalio BPMS ist wie auch zukünftig als Modellierungswerkzeug in das FloeCom-System eingebunden werden soll.
Das Adaptions-Plugin ist eine Zusatzkomponente, die das BPMN-Modellierungswerkzeug um die nötige Adaptionsfunktionalität erweitert.

Funktionen:

(1) Bereitstellen einer Benutzeroberfläche zur Adaptions-Unterstützung: Auswahl von Prozessen im Repository, Auswahl von Änderungsoperationen, Anzeige der laufenden Instanzen mit Auswahlmöglichkeit zur Prüfung einer Adaptionsdurchführbarkeit, Anzeige der migrierbaren und nicht migrierbaren Instanzen (mit eventueller Fehlerbeschreibung) zur Auswahl für die Durchführung der Migration.
(2) Änderung der BPMN-Prozesse auf Basis fest definierter Änderungsoperationen.
(3) Starten des BPEL-Übersetzers.
(4) Integration in die Versionsverwaltung der BPMN- und BPEL-Prozesse.

Schnittstellen:

zum BPMN-Modellierungstool:
– BPMN-Prozessdefinition (Input/Output)
– BPMN-Modellierungsfunktionen (Aufruf)
– Sperrung von händischen Prozessänderungen (Aufruf)

zum BPEL-Übersetzer:
– Starten der BPEL-Übersetzung (Aufruf)
– BPEL-Prozessdefinition (Input)

zum Repository:
– BPMN und BPEL-Prozessdefinition (Input/Output)

zur Adaptionskomponente:
– Adaptionsoperation (Aufruf)
– laufende Instanzen (Input)
– testweise zu migrierende Instanzen (Output)
– migrierbare und nicht migrierbare Instanzen inkl. Problembeschreibung (Input)
– endgültig zu migrierende Instanzen (Output)
– Statusbericht nach Durchführung der Migration (Input)

B. Adaptionskomponente

Die Adaptionskomponente ist der Hauptbestandteil der Adaptionslösung. Sie dient als zentraler Baustein zur Steuerung des Adaptionsvorganges und zum Aufrufen der meisten der hierfür notwendigen Funktionen. Ihre Kernfunktionalität, die Durchführung der Zustandsprüfung, der Datenflussanalyse und der Migration, wurde in Kapitel 5 ausführlich erläutert.

Funktionen:

(1) Durchführung der Zustandsprüfung für die zu migrierenden Instanzen.
(2) Durchführung einer Datenflussanalyse auf dem geänderten Prozess-Schema bzw. auf der ursprünglichen Version.
(3) Durchführung der Instanzenmigration.
(4) Integration der schon vorhandenen *Workflow-Änderungs*-Komponente.

Schnittstellen:

zum Adaptionsplugin des Modellierungstools:
– Adaptionsoperation (Methodenbereitstellung)
– laufende Instanzen (Output)
– testweise zu migrierende Instanzen (Input)
– migrierbare und nicht migrierbare Instanzen inkl. Problembeschreibung (Output)
– endgültig zu migrierende Instanzen (Input)
– Statusbericht nach Durchführung der Migration (Output)

zur Prozess-Engine:
– Anhalten und wieder starten (Aufruf)

zum Repository:
– BPEL-Prozessdefinition (Input)

zur Zustandsdatenbasis:
– laufende Instanzen (Input)
– Instanz-Ausführungszustände (Input/Output)
– Variablen-Werte (Input/Output)
– Instanzversion (altes oder neues Schema), die weiter ausgeführt werden soll. (Output)

Daten:

Ebenso wie beim Adaptionsplugin erfolgt auch bei der Adaptionskomponente keine persistente interne Datenhaltung. Allerdings möchten wir an dieser Stelle kurz auf einen wichtigen Aspekt der Datenbehandlung eingehen.
Wie bereits angesprochen, sind in der BPEL-Spezifikation keine Prozess- oder Aktivitätsausführungszustände definiert. Beim Einlesen der Zustände in die Adaptionskomponente muss deshalb eine Übersetzung der herstellerspezifischen Aktivitätszustände, mit denen die jeweilige BPEL-Engine arbeitet, in die entsprechenden Zustände der Adaptionskomponente erfolgen. Diese Übersetzung wird innerhalb des Adapters vorgenommen, der die Anbindung an die Zustandshaltungs-Komponente ermöglicht.

C. Repository

Das Repository dient als Datenablage zur Speicherung unterschiedlicher Versionen von BPMN- und BPEL-Prozessen. Wie alle Komponenten wird auch diese lose an die Adaptionskomponente gekoppelt. Wir unterbreiten auf funktionaler Ebene Vorschläge für eine Referenzimplementierung. Dieses Repository ist damit als dritte Komponente Teil unseres Adaptionskonzeptes. Für den Fall, dass im entsprechenden BPM-System bereits ein solches Repository existiert, kann auch dieses in unsere Adaptionsarchitektur integriert werden.
Wie auch das Adaptionsplugin des Modellierungstools werden wir das Repository selbst nicht implementieren. Die Entwicklung eines vielseitig verwendbaren Repositorys ist Thema einer anderen Arbeit am IPD. Wir unterstützen diese Arbeit dahingehend, dass wir die für unser Konzept notwendigen Anforderungen an eine solche Komponente spezifizieren.

Schnittstellen:

zum Adaptionsplugin des Modellierungstools:
– versioniertes BPMN-Prozessmodell (Input/Output)
– versionierte BPEL-Prozessdefinition (Input)

zur Prozess-Engine:
– versionierte BPEL-Prozessdefinition (Output)

zur Adaptionskomponente:
– versionierte BPEL-Prozessdefinition (Output)

Daten:

Die benötigten Daten sind bei BPMN- und BPEL-Prozessen die gleichen. Ein im Repository abgelegter Prozess-Datensatz besteht aus einem eindeutigen *Identifikator*, einer *Versionsnummer* und einem optionalen *Verweis auf den Eltern-Prozess*, eine Abänderung dessen er darstellt. Des Weiteren erfolgt eine Nennung der *durchgeführten Änderungsoperationen* und eine zusätzliche textuelle *Beschreibung der vorgenommenen Änderungen*. Hauptbestandteil eines Prozess-Datensatzes ist die eigentliche *Prozessdefinition*. Für BPEL-Prozesse wird eine BPEL-Datei erwartet, die zu dem in der Spezifikation definierten BPEL-Schema konform ist. Zusätzlich wird für jedes Paar von BPMN-/BPEL-Prozessdefinitionen ein Eintrag benötigt, der die *Zuordnung des BPMN-Prozesses zum BPEL-Prozess* und umgekehrt festhält.

6.1.2 Technische und formale Anforderungen

Nachdem die funktionalen Anforderungen an die einzelnen Komponenten spezifiziert wurden, gehen wir kurz auf die formalen und technischen Voraussetzungen ein, auf deren Basis die Implementierung erfolgt.

Formale Anforderungen

Die Implementierung stützt sich, wie das gesamte Adaptionskonzept, auf die Prozessdefinitions-Standards BPMN in der Spezifikationsversion 1.1 und BPEL in der Version 2.0.

Technische Anforderungen

Zur Programmierung wird die Entwicklungsumgebung *Eclipse* verwendet. Die Entwicklung erfolgt auf Basis der aktuellen Java-Version 6.0.
Im Rahmen der Integration in das FloeCom-System werden wir mit Hilfe des Datenbank Management Systems *IBM DB2* ein prototypisches Repository anlegen. Die Anfragen an die Datenbank werden in SQL 3 formuliert.

6.2 Entwurf

Wie bereits angesprochen steht die Umsetzung der Adaptionskomponente im Fokus unserer Implementierung. Auf den Entwurf dieses Moduls gehen wir im Folgenden ein. Das Adaptionsplugin des Modellierungswerkzeuges werden wir nicht näher betrachten.

6.2.1 Package-Struktur

Zur sinnvollen Strukturierung des Quellcodes ordnen wir die Java-Klassen folgenden Packages zu:

- adaption_operations

- bpelSchema_algorithms
- repository
- bpelInstance_algorithms
- instance_migration
- bpmn_modeller
- exceptions

Das Paket *adaption_operations* enthält für jede Adaptionsoperation, wie z.B. für das serielle Einfügen einer einfachen Aktivität, eine Klasse, in der der Ablauf des Adaptionsvorganges umgesetzt ist. Die Umsetzung erfolgt gemäß der Beschreibung in Kapitel 5.4.6. Innerhalb der Klasse *AdaptionOperation* finden sich in Form von Methoden die Adaptionsprimitive der höheren Ebene, wie beispielsweise *determineSLDataFlowValid*[3] oder *FindReadingActivitiesForward*.

Die Methoden, die direkt auf das BPEL-Schema zugreifen, sind in der Klasse *BPELSchemaQuery* im Paket *bpelSchema_algorithms* implementiert. Hierzu gehören *queryActivityType*, *querySucceedingElement*, *queryVariablesAccessed* und einige mehr.

Das Package *repository* stellt das Interface und die Adapterklassen zum Zugriff auf das Repository bereit. Dieser Zugriff wird unter anderen von der Klasse *BPELSchemaQuery* für die Ausführung von BPEL-Schema-Abfragen benötigt.

Analog hierzu enthält das Paket *bpelInstance_Algorithms* die Klasse *BPELInstanceQuery*, die alle Methoden für Zugriffe auf Instanzen enthält. Im Wesentlichen betrifft dies das Abfragen oder Setzen von Ausführungszuständen. Diese Aufgabe übernehmen die Methoden *queryActivityStatus* oder *setActivityStatus*.

Im Paket *instance_migration* sind alle Komponenten enthalten, die für die Durchführung der Instanzmigration benötigt werden. In der Hauptsache ist dies die Klasse *InstanceMigration* mit der Methode *migrateInstance*, aber auch die Schnittstellen-Definition zum Anhalten und Starten der Engine sowie die zugehörigen Adapter-Klassen sind hier zu finden.

Die Schnittstellen zum Modellierungswerkzeug-Plugin sind im Paket *bpmn_modeller* spezifiziert. Es enthält auch die Adapterklasse zum Modellierungstool von Intalio, dem BPMN-Modeler.

Innerhalb des *exceptions*-Packages werden die anwendungsspezifischen Exceptions definiert, die wir für unsere Adaptionsimplementierung benötigen. Beispielsweise müssen Fehler abgefangen werden, die entstehen, wenn ein BPEL-Dokument nicht der Spezifikation entspricht.

6.2.2 Klassen-Übersicht

Das Klassendiagramm in Abbildung 6.3 zeigt eine Übersicht über die wichtigsten Klassen, die bislang implementiert wurden.
Die Klasse *Adaption* ist das zentrale Element der Klassenstruktur. Sie sorgt für die Durchführung der im Rahmen einer Adaption anfallenden Aufgaben. Hierfür wird sie instanziiert und über ihre *run*-Methode aufgerufen. Die *run*-Methode fragt alle laufenden Instanzen eines Prozess-Schemas ab und überlässt dem Benutzer die Auswahl, welche von diesen er – zunächst testweise – migrieren möchte. Für die spezifizierten Instanzen wird nun die eigentliche Adaptionsoperation aufgerufen, wie z.B. *BasicActivitySerialInsert*. Die Durchführung der eigentlichen

[3] Die Benennung der Java-Methoden weicht leicht von der Namensgebung in der Konzept-Ausarbeitung ab. Die jeweilige Entsprechung sollte aber problemlos erkennbar sein.

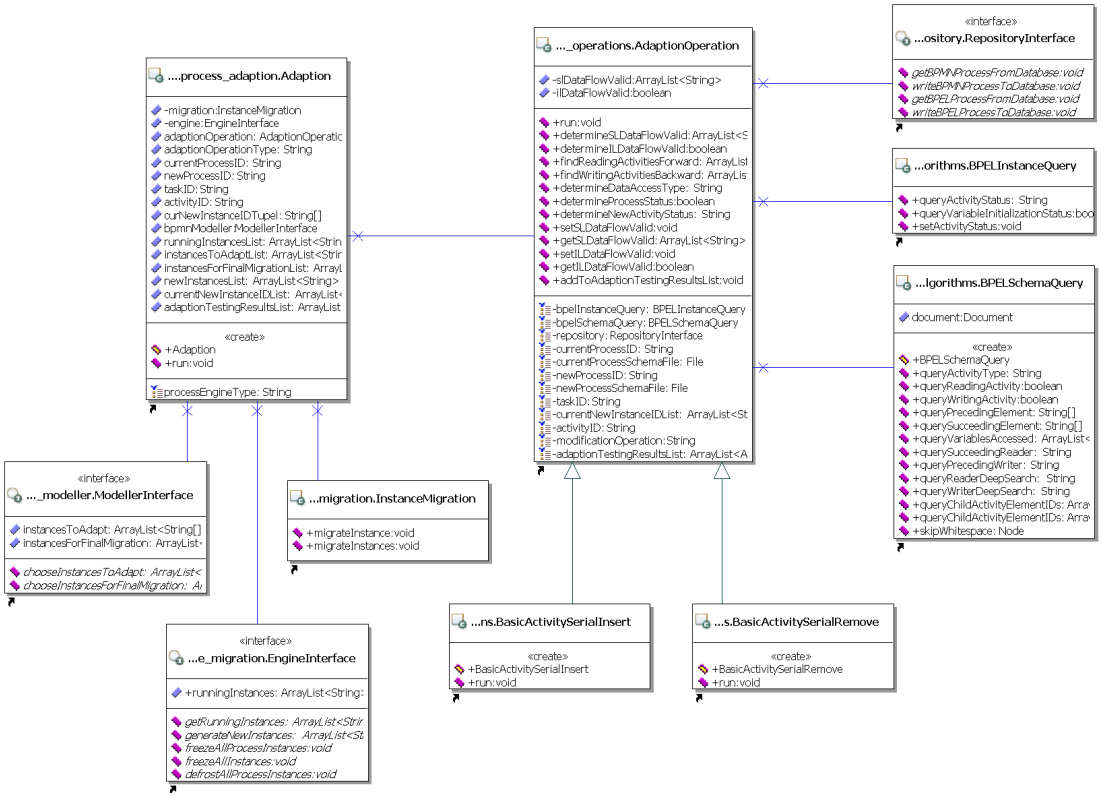


Abbildung 6.3: Adaptionskomponente – Klassendiagramm

Migration übernimmt abschließend die Klasse *InstanceMigration*.
Die von uns in Kapitel 5.4.6 dargestellten Adaptionsoperationen *BasicActivitySerialInsert* und *BasicActivitySerialRemove* sind in eigenen Klassen gleichen Namens definiert. Sie erben die bei Durchführung der Adaptionsoperation benötigten Attribute und Methoden von der Oberklasse *AdaptionOperation*.
Hinsichtlich der Methoden, die im Wesentlichen Implementierungen der im Anhang gezeigten Adaptionsprimitive darstellen, nehmen wir an dieser Stelle nochmals eine Unterscheidung vor: Innerhalb der Klasse *AdaptionOperation* sind alle Methoden „höherer Ebene“ definiert, die keinen direkten Prozess-Schema- oder Datenbank-Zugriff benötigen. Dies ist beispielsweise bei der Methode *determineSLDataFlowValid* der Fall.
Alle „einfacheren“ Funktionen, die direkte Abfragen auf dem Prozess-Schema durchführen, sind in der Klasse *BPELSchemaQuery* enthalten. Analog enthält *BPELInstanceQuery* die Methoden für Instanzabfragen auf der Datenbank. Für die Implementierung der Methoden von *AdaptionOperation* sind Zugriffe auf diese beiden Klassen notwendig.
Als Schnittstellen zu den anderen Systemkomponenten fungieren die Interfaces und die zugehörigen Adapter für das Modellierungstool, das Repository, die Zustandsdatenbank und die Prozess-Engine.

6.2.3 Ablauf-Übersicht

Das Sequenzdiagramm in Abbildung 6.4 verdeutlicht nochmals den Ablauf einer Adaptionsvorgangs auf Implementierungsebene.
Die Klasse *Adaption* wird vom Modellierungswerkzeug-Plugin über das *Modeller*-Interface instanziiert. Notwendige Parameter sind die durchzuführende Adaptionsoperation, Referenzen auf

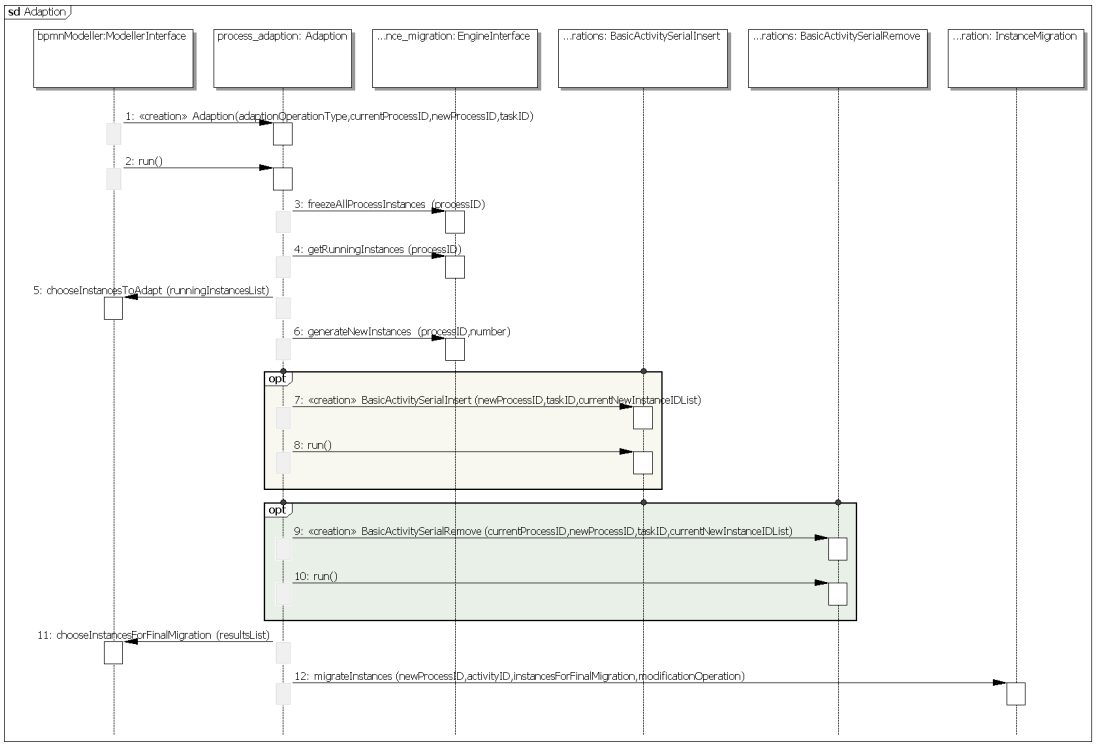


Abbildung 6.4: Adaptionskomponente – Sequenzdiagramm zum allgemeinen Adaptionsablauf

das alte sowie das neue Prozessschema und im Falle des Einfügens oder Löschens eines Tasks die Angabe der Task-ID. Der Adaptionsvorgang wird über den Aufruf der *run*-Methode gestartet.

Zunächst wird die Ausführung aller aktiven Prozess-Instanzen gestoppt. Eine Liste aller Instanzen des jeweiligen Prozesses wird abgefragt und dem Benutzer übergeben. Dieser wählt aus den aufgeführten Instanzen diejenigen aus, für die er die Durchführung der Adaption beabsichtigt. Damit für die spätere Migration „leere“ Instanzen zur Verfügung stehen, auf die migriert werden kann, werden auf Basis des geänderten Prozess-Schemas so viele neue Instanzen erzeugt, wie für die (testweise) Migration benötigt werden.
Das Einfrieren und das Auslesen der laufenden Instanzen, sowie die Generierung der neuen Instanzen erfolgt durch Aufrufe über das *Engine-Interface*.

Nachdem die notwendigen Vorarbeiten erledigt sind, wird nun die eigentliche Adaptionsoperation gestartet. Zwei mögliche Operationen, *BasicActivitySerialInsert* und *BasicActivitySerialRemove*, hatten wir in Kapitel 5.4.6 bereits in Pseudocode-Darstellung aufgezeigt. Der detaillierte Ablauf dieser beiden Operationen ist im Anhang in den Abbildungen A.1 und A.2 nochmals in Form eines Sequenzdiagrammes dargestellt.
Innerhalb dieser Operationen erfolgen im Wesentlichen die Datenflussüberprüfungen auf Schema- und Instanzebene und die Instanz-Ausführungszustandsprüfung. Je nach Adaptionsoperation werden die Instanzen hierfür auch testweise auf das neue Prozess-Schema migriert.

Die Resultate der Prüfalgorithmen werden dem Benutzer instanzspezifisch mitgeteilt. Auf Grundlage dieser Informationen entscheidet er, welche Instanzen wirklich migriert werden sollen.
Die zu migrierenden Instanzen und die zusätzlich benötigten Angaben werden der Methode *migrateInstances* der Klasse *InstanceMigration* als Parameter übergeben. Diese führt die eigentliche Migration durch.

6.3 Integration in FloeCom und Intalio

Den Entwurf des Adaptionsmoduls als unabhängige Systemkomponente haben wir eben dargestellt. Im Folgenden gehen wir auf die Integration des Moduls in die FloeCom-Umgebung und in das Intalio Business-Process-Management-System ein.

6.3.1 FloeCom

Im den vorangehenden Abschnitten wurde bereits aufgezeigt, welche Schnittstellen für eine Integration in die FloeCom-Architektur, aber auch in andere Systeme, vorhanden sein müssen. Abbildung 6.5 zeigt die wichtigsten FloeCom-Systembausteine inklusive der integrierten Adaptionskomponenten.

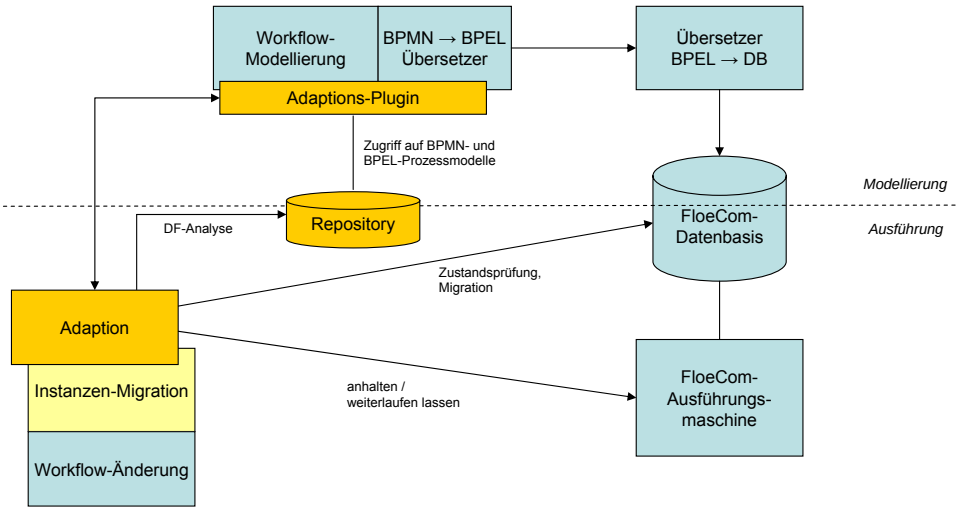


Abbildung 6.5: Integration in die FloeCom-Architektur

Zur Modellierung der BPMN-Prozessmodelle ist die Einbindung des BPMN-Modelers von Intalio geplant. Diese wurde bislang aber noch nicht durchgeführt. Dem Modellierungswerkzeug muss das *Adaptionsplugin* angegliedert werden, das die Änderungen am Prozessmodell vornimmt und die Kommunikation mit dem Repository und der Haupt-Adaptionskomponente übernimmt.
Das ebenfalls noch zu implementierende *Repository* muss auf Grundlage der Anforderungen entwickelt werden, die in dieser und in mehreren vorangehenden Arbeiten an ein solches gestellt wurden.

Wie bereits zu Beginn des Kapitels im Integrations-Szenario skizziert wurde, ist die Adaptions-Komponente der zentrale Baustein für die gesamte Adaptionsfunktionalität. Unter Zugriff auf die FloeCom-Datenbasis, die alle Instanzzustände enthält, führt sie die Zustandsprüfung und die Datenflussanalyse durch, ebenso die Instanzmigration. Damit das System nicht in einen inkonsistenten Zustand gerät, wird die Workflow-Engine vor Durchführung der Adaptionsschritte angehalten. Diese Aufgabe übernimmt ebenfalls die zentrale Adaptionskomponente.
Im Rahmen der Entwicklung eines Adaptionskonzeptes für Daten-Workflow-Interaktionen (siehe [Wei08]) wurde bereits die Komponente *Workflow-Änderung* implementiert. Diese führt die dabei anfallenden Adaptionsoperationen aus und greift hierbei auf die ebenfalls benötigte *Instanzen-Migrationskomponente* zurück. Beide Module müssen in die von uns entwickelte

Adaptionskomponente integriert werden, sodass am Ende eine Komponente existiert, die eine Vielzahl unterschiedlicher Adaptionsaspekte unterstützt.

Schnittstellen zu anderen Komponenten

A - Modellierungstool

Bislang wurde noch kein Modellierungswerkzeug in das FloeCom-System integriert. Wir stellen für die zukünftige Einbindung dieser Komponente und des ebenfalls noch zu entwickelnden Adaptionsplugins die entsprechenden Schnittstellen bereit, wie sie weiter oben in allgemeiner Form dargestellt wurden.

B - Repository

Die Entwicklung eines vielseitig einsetzbaren Repositorys ist Gegenstand einer anderen Arbeit am IPD. Für unsere Zwecke erstellen wir prototypisch ein einfaches Repository nach den oben definierten Anforderungen und binden dieses über eine Adapterschnittstelle ein.
Die notwendigen Daten teilen wir auf insgesamt vier Relationen auf. In der verwendeten DB2-Datenbank legen wir hierzu die Tabellen *ADAPTION*, *ADAPTIONOPERATION*, *BPMNPROCESS* und *BPELPROCESS* an. In die Tabelle *ADAPTION* wird jeder einzelne Adaptionsvorgang eingetragen. Ein solcher Datensatz enthält Verweise auf die durchgeführte Adaptionsoperation, wie z.B. *BasicActivitySerialInsert*, und auf den geänderten BPMN-Prozess, sowie eine zusätzliche textuelle Beschreibung der durchgeführten Prozess-Änderung. Ein Eintrag der Relation *BPMNPROCESS* setzt sich aus der Prozessschema-Version, einer Referenz auf den Elternprozess und natürlich dem Prozess-Schema selbst zusammen. Die *BPELPROCESS*-Tabelle weist die gleichen Attribute auf, allerdings kommt hier noch ein Verweis auf das jeweils zugehörige BPMN-Prozessschema hinzu.
Die Klasse *RepositoryInterface* im Paket *repository* definiert die allgemeine Repository-Schnittstelle, die Klasse *FloecomRepositoryAdapter* enthält den Adapter zur Einbindung des FloeCom-Repositorys.

C - FloeCom-Datenbasis

Das Schema der FloeCom-Datenbasis wurde als direkte Abbildung der BPEL-Struktur entwickelt, ergänzt um Elemente wie die Zustandsdatenhaltung der Prozess-Instanzen. Für uns relevant sind die Daten zum Ausführungszustand einer Instanz und die zugehörigen Variablenbelegungen.
In FloeCom werden Aktivitäten keine expliziten Zustände zugewiesen. Es werden nur die laufenden Aktivitäten erfasst; für diese erfolgt ein Eintrag in der Tabelle *CURRENTACTIVITY*. Es wird auch nicht vermerkt, ob eine Aktivität einer Prozessinstanz bereits ausgeführt wurde. Im Rahmen unserer Zustandsüberprüfung benötigen wir aber zumindest diese Information. In diesem Punkt muss das FloeCom-System an unsere Anforderungen angepasst werden, sonst ist eine Integration unserer Adaptionskomponente nicht möglich. Alternativ wäre es auch möglich, eine Rückwärtsanalyse durchzuführen. Diese müsste bestimmen, ob eine Aktivität beispielsweise im Prozessablauf *vor* der aktuell ausgeführten steht und damit implizit schon ausgeführt worden sein muss. Zu prüfen wäre, inwieweit dieses Vorgehen auch unter Einbeziehung von Schleifen und Compensations korrekte Ergebnisse liefern kann.
Die Variableninstanzen werden von FloeCom in der Relation *VARIABLEINSTANCE* gespeichert. Diese enthält alle von uns benötigten Daten, insbesondere den aktuellen Variablenwert.

Die FloeCom-Ausführungsmaschine und das zugehörige Datenbankschema basieren momentan noch auf der BPEL-Version 1.1, sind also mit unserem Adaptionskonzept, das auf BPEL 2.0 aufbaut, nicht kompatibel. Eine Migration des Workflow-Systems nach BPEL 2.0 ist allerdings

in Arbeit. Diese Programm-Version setzen wir bei der Implementierung des Adapters voraus.

D - Ausführungsmaschine

Wir müssen die Ausführung der laufenden Instanzen vor der Durchführung der wesentlichen Adaptionsfunktionen anhalten und später wieder weiterlaufen lassen. Diese Funktionalität implementieren die Methoden *freeze* und *defrost* der *Main*-Klasse des existierenden FloeCom-Systems. Auch sie werden über einen Adapter in die Funktionalität der Adaptionskomponente eingebunden.

E - Instanzen-Migration und Workflow-Änderung

Um das Ziel einer unabhängigen Adaptionskomponente, die alle Arten von Adaptionsfunktionalität enthält, zu erreichen, sollten die separat von unserem Konzept entwickelten Komponenten *Instanzen-Migration* und *Worklow-Änderung* mit unserer Adaptionskomponente zusammengeführt werden. Diese Aufgabe kann von uns im vorgegebenen Zeitrahmen aber weder auf der Konzeptions- noch auf der Implementierungsebene umgesetzt werden.
So muss zum Beispiel die Migrationskomponente dahingehend erweitert werden, dass auch die gezielte Migration einzelner und nicht nur aller Instanzen eines Prozess-Schemas möglich ist. Andererseits wurde die in der bestehenden Implementierung enthaltene Berücksichtigung der Zustände von Schleifen und Event-Handlern in unserem Konzept beispielsweise gar nicht näher untersucht. Eine größere Herausforderung liegt des Weiteren in der korrekten Anpassung der Instanz-Ausführungszustände nach Änderungen am Prozessmodell. Die hierfür notwendigen Verfahren können teilweise sehr komplex sein und müssen erst noch intensiv untersucht werden. Wir sind auf diesen Aspekt in Kapitel 5.4.2 ausführlich eingegangen.

Zusammenfassung

Eine Integration unserer Adaptionskomponente in das FloeCom-System ist möglich. Hierzu müssen innerhalb des Workflow-Systems kleine Anpassungen vorgenommen werden, insbesondere muss festgehalten werden, ob eine Aktivität bereits ausgeführt wurde oder nicht. Die bereits laufende Anpassung des Systems an die BPEL 2.0-Spezifikation ist zu vollenden.
Größere Herausforderungen liegen in der Einbindung des Intalio BPMN-Modellierungstools und eines Compilers für die Übersetzung von BPMN nach BPEL. Des Weiteren sollte die bereits im FloeCom-System vorhandene Adaptionsfunktionalität in unserer Komponente integriert werden.

6.3.2 Intalio

Die Integration in das Business-Process-Management-System Intalio|BPMS ist aufgrund der auf Entwicklungsebene teilweise eingeschränkten Dokumentation schwieriger vorzunehmen. Soweit dies möglich war, haben wir ein sinnvolles Integrationskonzept erarbeitet und beschreiben dieses im Folgenden. In Abbildung 6.6 ist die Komponentenansicht des integrierten Systems dargestellt.

In der Open Source-Version ist das Modellierungswerkzeug *BPMN-Modeler* nicht in die Ausführungsumgebung *Ode* integriert. Diese Einbindung ist händisch vorzunehmen, ebenso muss ein Übersetzer eingebunden werden, der aus dem BPMN-Modell BPEL-Code erzeugt. Das Repository und die Adaptionskomponente verhalten sich ebenso wie bei der Integration in das FloeCom-System. Die Besonderheit des Intalio-Systems liegt darin, dass die BPEL-Prozessdefinition vor der Ausführung nochmals in eine interne Prozessrepräsentation übersetzt wird.

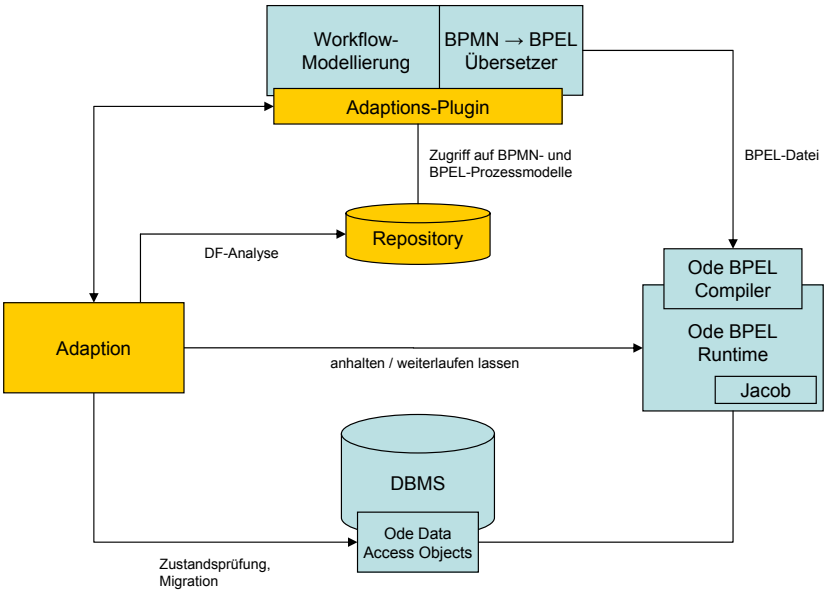


Abbildung 6.6: Integration in die Intalio-Architektur

Diese wird von der *Jacob*-Engine innerhalb der *Ode BPEL Runtime*-Komponente ausgeführt. Die persistente Speicherung der Instanzzustände erfolgt in einem beliebigen Datenbank-Management-System, das über die *Ode Data Objects* angebunden ist.

Unsere Adaptionskomponente führt die Datenflussanalyse auf der im Repository abgelegten Prozessdefinition durch. Das Anhalten oder Starten bestimmter Prozessinstanzen erfolgt unter Zugriff auf die Ode BPEL Runtime. Die in der Datenbank abgelegten Instanz-Zustandsdaten werden für die Durchführung verschiedener Zustandsprüfungen und für die Migration benötigt. Die Ode Data Objects bilden die Schnittstelle hierzu.

Schnittstellen zu anderen Komponenten

A - Modellierungstool

Der *BPMN-Modeler* muss um ein Adaptionsplugin erweitert werden. Dessen Schnittstellen hin zur Adaptionskomponente haben wir bereits beschrieben. Die konkrete Implementierung der Adapterklasse kann erst erfolgen, nachdem das Plugin entwickelt wurde.

B - Repository

Beim Anlegen des Repositorys für das Intalio-System gehen wir ebenso vor, wie wir es bereits im Rahmen der FloeCom-Integration erläutert haben. Hierzu erstellen wir innerhalb des gleichen Datenbank-Management-Systems, das auch der Zustandsdatenhaltung dient, die bereits beschriebenen Relationen *ADAPTION*, *ADAPTIONOPERATION*, *BPMNPROCESS* und *BPELPROCESS*.
Die Klasse *IntalioRepositoryAdapter* im Paket *repository* dient als Adapterklasse zu den *Ode Data Access Objects*.

C - DBMS

Die Intalio-Datenbank enthält Informationen zu allen importierten Prozess-Schemata und allen ausgeführten Instanzen. Letztere werden in der Tabelle *BPEL_INSTANCE* aufgeführt. Für jede Instanz wird der aktuelle Instanzzustand *STATE* festgehalten. Dieser Zustand stellt aber nur den allgemeinen Instanzzustand, wie *gestartet* oder *beendet*, dar. Die einzelnen Aktivitätszustände können nicht ausgelesen werden. Der detaillierte Ausführungszustand einer Instanz wird als *JACOB_STATE* gespeichert. Dieser Eintrag gibt den internen Ausführungszustand der Jacob-Engine an. Ein mögliches Verfahren für ein Mapping auf einzelne BPEL-Aktivitätszustände ist uns nicht bekannt. Damit sind die Informationen zum Ausführungszustand des Intalio-Systems für uns nicht verwertbar.
Auslesbar sind aber die von uns ebenfalls benötigten Variablenwerte bzw. ihr Initialisierungszustand. Die Tabelle *BPEL_XML_DATA* listet alle XML-basierten Variablen auf, die Relation *LARGE_DATA* enthält die zugehörigen Variablenwerte in Form von XML-Objekten.

D - Ausführungsmaschine

Die *Ode BPEL-Management API* sieht eine Abfrage der laufenden Instanzen ebenso vor wie die Möglichkeit, die Ausführung von Prozess-Instanzen zu stoppen und wieder fortzusetzen. Allerdings ist die Schnittstelle als Webservice-Interface definiert und nicht als direkte Java-API.

Intalio BPMN-BPEL-Compiler

Im Rahmen einer vollständigen Workflow- bzw. Adaptionslösung wird – unabhängig vom eigentlichen Adaptionsvorgang – ein Compiler benötigt, der die BPMN-BPEL-Übersetzung vornimmt. Wie in Kapitel 2 bereits angesprochen, wird die BPEL-Übersetzung des Intalio-Compilers unseren Anforderungen nicht gerecht: Es erfolgt insbesondere keine eindeutige Benennung aller Aktivitäten. Einfache BPEL-Aktivitäten erhalten ein *bpmn:id*-Attribut, das den zugehörigen Task auf BPMN-Ebene referenziert. Strukturierte Aktivitäten werden gar nicht benannt, auf eine Verwendung des in der Spezifikation vorgeschlagenen *name*-Attributes wird ganz verzichtet.
Zusätzlich werden alle BPEL-Aktivitäten mit einem *bpel*-Namespace-Präfix versehen, das in der Spezifikation nicht vorgesehen ist.

Das Entfernen der Namespace-Präfixe lässt sich über einen vorgeschalteten Parser vornehmen. Im Rahmen der zwei Adaptionsoperationen, die bislang von uns entwickelt wurden, ist auch eine Umwandlung des *bpmn:id*-Attributes in ein *name*-Attribut problemlos möglich. Die fehlenden Identifier der strukturierten Aktivitäten könnten mit willkürlichem Inhalt ergänzt werden, da eine BPMN-Zuordnung dieser Aktivitäten bei den bislang betrachteten Adaptionsoperationen nicht notwendig ist.
Letzteres ist nicht der Fall bei zukünftig zu entwickelnden Änderungsoperationen, die deutlich komplexer ausfallen können. Eine Anpassung des BPEL-Compilers an unsere Bedürfnisse ist also unumgänglich.

Zusammenfassung

Die Integration der Adaptionskomponente in das Intalio BPMS stellt eine wesentlich größere Herausforderung dar, als die Integration in die FloeCom-Architektur. Die hauptsächliche Schwierigkeit liegt hierbei in der gezielten Abfrage von Aktivitätszuständen, die momentan nicht möglich ist. Intalio müsste hier Abhilfe schaffen, indem die Ausführungszustände aller Aktivitäten in der Datenbank festgehalten werden. Alternativ könnte auch eine Methode

zur Abbildung der internen Zustände der Jacob-Komponente auf die Ausführungszustände der BPEL-Aktivitäten entwickelt werden.
Auch der vom Compiler erzeugte BPEL-Code entspricht in einigen Punkten nicht unseren Anforderungen. Diese Komponente muss also ebenfalls an unser Adaptionskonzept angepasst werden.

Kapitel 7

Validierung

In den beiden vorangehenden Kapiteln haben wir ein detailliertes Adaptionskonzept erarbeitet und aufgezeigt, wie eine Implementierung aussehen kann. Im Folgenden analysieren wir nun, ob unser Konzept den in Kapitel 3 aufgestellten Forderungen gerecht wird. Hierzu gehen wir zunächst auf die allgemeinen Anforderungen ein, die an unser Konzept gestellt wurden. Danach führen wir im Speziellen anhand zweier Beispielprozesse eine Validierung der Korrektheits- und Konsistenzerhaltung durch.

7.1 Validierung der Anforderungen

Im Kapitel *Anforderungen* hatten wir die Anforderungen erläutert, die an das von uns zu entwickelnde Adaptionskonzept bzw. an die Implementierung als unabhängige Java-Komponente gestellt wurden:

- Berücksichtigung der entsprechenden Industrie-Standards
- Durchgängiger Adaptionspfad
- Benutzerunterstützung
- Korrektheits- und Konsistenzerhaltung
- Erweiterbarkeit des Adaptionskonzeptes
- Implementierung als unabhängige Komponente

Wir analysieren nun, inwieweit unser Adaptions-Konzept diese Anforderungen erfüllt.

A – Berücksichtigung der entsprechenden Industrie-Standards

Die formalen Modelle, die von uns für die BPMN- und die BPEL-/Instanzebene aufgestellt wurden, halten sich streng an die Vorgaben der Spezifikationen BPMN 1.1 und BPEL 2.0. In einigen wenigen Punkten mussten wir Einschränkungen vornehmen. Diese wurden im Kapitel 5.1.3 aufgeführt.

B – Durchgängiger Adaptionspfad

Es ist uns gelungen, ein Konzept zu entwickeln, wie Änderungen am Prozessmodell über die BPEL-Übersetzung an die Instanzebene weitergegeben werden können. In Kapitel 5.1.1 haben wir unsere Vorgehensweise erläutert.

C – Benutzerunterstützung

Die Einbindung des Benutzers bei der Durchführung der Adaptionsoperationen wurde von uns nur am Rande betrachtet. Wir haben allerdings dargelegt, welche Informationen dem Benutzer in Bezug auf die zu migrierenden Instanzen bereitgestellt werden sollten. Bei der Entwicklung einer geeigneten Methode zur Datenflussüberprüfung haben wir hierauf Rücksicht genommen und sichergestellt, dass die notwendigen Informationen durch unsere Datenfluss-Algorithmen extrahiert werden.

D – Korrektheits- und Konsistenzerhaltung

Bei Änderungen am Prozessmodell und an den Instanzdaten ist darauf zu achten, dass weder die Korrektheit der Prozessdefinition noch die Konsistenz des Instanzzustandes beeinträchtigt wird. Wir können die Erfüllung dieser Forderung aufgrund vielfältiger Prüf-Mechanismen garantieren.
Bei dem von uns eingesetzten Intalio-Modellierungstool wird das BPMN-Prozessmodell vor der BPEL-Übersetzung auf seine Korrektheit hin geprüft. An der BPEL-Prozessdefinition, die der BPEL-Compiler hieraus generiert, nehmen wir keine Veränderungen vor. Diese kann also ebenfalls als korrekt angesehen werden.
Zur Vermeidung von Fehlern sollte auch die Datenflusskorrektheit auf Instanzebene gewährleistet sein. Wir haben umfangreiche Funktionen entwickelt, die sowohl auf Schema-Ebene wie auch auf Instanzebene eine Datenfluss-Überprüfung vornehmen. Diese wurden in Kapitel 5.4.4 vorgestellt.
Der Ausführungszustand einer Instanz ist dafür maßgeblich, ob diese auf ein geändertes Prozess-Schema migriert werden kann oder nicht. Die Grundlagen, auf denen unsere Zustandsprüfung basiert, haben wir in Kapitel 5.4.3 ausgeführt. Vorab hatten wir bereits die Schwierigkeiten, die bei einer Migration auf ein verändertes Schema auftreten, dargelegt und unser Vorgehen zur Konsistenzerhaltung auch in diesem Punkt erläutert (Kapitel 5.4.2).

E – Erweiterbarkeit des Adaptionskonzeptes

Bei der Entwicklung unseres Adaptionsmodells haben wir darauf geachtet, es möglichst in jeglicher Hinsicht offen zu konzipieren.
In unserem Konzept gehen wir zwar von fest definierten Änderungsoperationen aus, es wäre in einem weiteren Schritt aber möglich, freie Änderungen am Prozessmodell auf diese abzubilden. Späteren Erweiterungen können auf die von uns entwickelten Meta-Modelle, die Änderungs- und Adaptionsprimitive sowie die Verfahren und Funktionen zur Ausführungsstatusüberprüfung, zur Datenflussanalyse und zur Migrationsdurchführung zurückgreifen.

F – Implementierung als unabhängige Komponente

Wir haben unser Adaptionskonzept in Form einer eigenständigen Adaptionskomponente implementiert. Die notwendigen Schnittstellen zu anderen Systembausteinen wurden definiert. So ist es möglich, die Adaptionskomponente in jedes System zu integrieren, für das entsprechende Adapter-Implementierungen möglich sind.

7.2 Validierungs-Szenario

Die Einhaltung der Korrektheits- und Konsistenzbedingungen durch unsere Adaptionskomponente lässt beispielhaft überprüfen. Wir wollen eine solche Validierung anhand zweier Adaptions-Szenarien durchführen.

7.2.1 Voraussetzungen

Am Ende von Kapitel 6 wurde zusammenfassend aufgezeigt, dass bei einer Integration unserer Adaptionskomponente sowohl in Bezug auf das FloeCom-System, wie auch hinsichtlich des Intalio-BPMS Probleme bestehen, was die Abfrage und Anpassung der aktuellen Instanz-Ausführungszustände betrifft. Eine vollständige Validierung ist somit nicht möglich.

Wir führen deshalb eine etwas vereinfachte Validierung durch und beschränken uns auf die Gültigkeitsprüfung der Datenfluss-Analyse sowie der Instanzzustandsüberprüfung. Hierzu haben wir die in Kapitel 3 gezeigten Beispiel-Prozesse mit dem Intalio-Designer modelliert und ins BPEL-Format übersetzt.

Um die Ausgabe als BPEL-Prozessdefinition für unsere Validierung verwenden zu können, müssen wir allerdings einige Anpassungen vornehmen.

- Das *bpel*-Namespace-Präfix, mit dem der Intalio Compiler alle Elemente auszeichnet, wird entfernt.
- Strukturierte Aktivitäten werden vom Intalio BPEL-Compiler weder mit einem *bpmn:id*- noch mit einem *bpmn:label*-Attribut versehen. Letzteres wird nachgeholt, die Attribut-Werte werden hierbei willkürlich, aber paarweise disjunkt gewählt.
- Anschließend werden alle *bpmn:label*-Attribute in *name*-Attribute gleichen Inhaltes umgewandelt.
- Der Intalio-Compiler fügt für alle Variablen automatisch eine vorangehende Initialisierung in die Prozessdefinition ein und verhindert so das Auftreten von *bpel:uninitializedVariable*-Fehlern. Diese Methode wird nicht von der Spezifikation vorgeschlagen und funktioniert auch nur bei statischen Prozessmodellen – Datenflussprobleme, die im Rahmen von Adaptionsoperationen auftreten, können auf diese Weise nicht verhindert werden. Wir entfernen deshalb die Initialisierungsaktivitäten aus der Prozessdefinition.

Wir besprechen die Validierung im Folgenden auf Basis der BPMN-Prozessmodelle, da auf dieser Ebene eine bessere Übersichtlichkeit gegeben ist. Die eigentliche Adaptionsfunktionalität läuft aber auf BPEL-Ebene ab. Die BPEL-Repräsentationen der Validierungsprozesse sind in Anhang B zu finden.

Die beiden Prozesse haben wir bereits in Kapitel 3, im Abschnitt *Beispiel-Szenario* vorgestellt. Ebenso wurden die Änderungsoperationen erläutert, die vorgenommen werden sollen. Wir definieren jetzt zusätzlich noch die Ausführungszustände, in denen sich die Prozess-Instanzen befinden, wenn wir die Adaptionsoperation auf sie anwenden. Anhand dieser Instanzen prüfen wir, – soweit dies in unserer eingeschränkten Validierungsumgebung möglich ist – ob die Adaptionskomponente die einzelnen Schritte wie erwartet durchführt. Die hierbei notwendigen Zustandsabfragen, die eigentlich auf der Instanz-Datenbank ausgeführt werden sollten, werden hart codiert.

7.2.2 Hochladen eines Konferenzbeitrages – Einfügeoperation

Wir definieren zunächst zwei Prozess-Ausführungszustände in denen die Adaptionsoperation *Serielles Einfügen einer einfachen Aktivität* jeweils durchgeführt werden soll. Sie sind im untenstehenden Prozessmodell (Abb. 7.1) grafisch dargestellt.

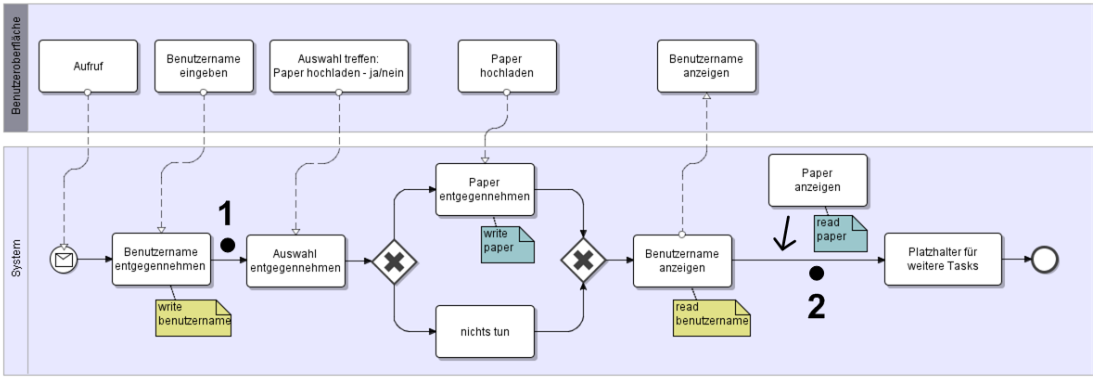


Abbildung 7.1: Einfügeoperation

Zustand 1 wird erreicht, wenn innerhalb einer Prozess-Instanz der Task *Benutzername entgegennehmen* abgeschlossen wurde. Die Variable *benutzername* wurde beschrieben. Der nachfolgende Task *Auswahl entgegennehmen* befindet sich noch in einem nicht aktivierten Zustand, also im Status *None*.
Der zweite Zustand definiert einen fortgeschritteneren Prozess-Ausführungszustand. Der Task *Benutzername anzeigen* wurde beendet, der nachfolgende Task steht zur Ausführung an. Wir definieren darüber hinaus, dass zuvor der obere Pfad der Verzweigung durchlaufen worden ist. Das heißt, die Variable *paper* wurde beschrieben.

Erwartetes Verhalten

Das Einfügen des neuen Tasks ist in beiden Prozesszuständen in Hinblick auf den Ausführungsstatus problemlos möglich. Im Ausführungszustand 1 befindet sich die Prozessausführung noch weit vor der Stelle im Prozess, an der die Änderung erfolgen soll.
Im Zustand 2 ist der Task *Platzhalter für weitere Tasks* noch nicht gestartet. Der Task *Paper anzeigen* kann eingefügt und im Anschluss aktiviert, also zur Ausführung vorgesehen werden.

Die Überprüfung des Datenflusses offenbart allerdings Schwierigkeiten. Nach der Durchführung der Einfügeoperation existiert im Prozess-Schema ein Lesezugriff auf die Variable *paper*, ohne dass garantiert wird, dass diese Variable zuvor beschrieben wird. Das Prozess-Schema wird also hinsichtlich des Datenflusses als fehlerhaft erkannt.
Für den Zustand 1 gilt dies auch für den Datenfluss auf Instanzebene, da nicht klar ist, welcher Ast der Verzweigung ausgeführt werden wird. Im Ausführungszustand 2 ist dies allerdings bekannt: Laut unserer Definition wurde der obere Ast gewählt, das heißt das Paper wurde hochgeladen und die Variable *paper* folglich beschrieben. Obwohl der Datenfluss des abgeänderten Schemas fehlerhaft ist und sich die Änderungsoperation in vollem Maße auf die entsprechende Instanz auswirkt, tritt auf Instanzebene kein Datenfluss-Problem auf.
Die Instanz, die sich im Zustand 2 befindet, kann also unter Einhaltung aller Korrektheits- und Konsistenzbedingungen auf das neue Schema migriert werden. Für Instanzen im Zustand 1 ist dies allerdings nicht möglich (bzw. nicht sinnvoll). Ebenso sollten auch keine neuen Instanzen auf dem abgeänderten Prozess-Schema gestartet werden.

Erfolgtes Verhalten

Die Adaptionsoperation *BasicActivitySerialInsert* wird mit einer Referenz auf das geänderte Prozessschema und mit *Paper anzeigen* als *ActivityName*-Attribut aufgerufen. Der Algorithmus erkennt, dass die Aktivität in lesender Form auf die Variable *paper* zugreift. Diese Informationen werden *SLDataFlowValid* als Attribute übergeben. Die Datenfluss-Überprüfung auf Schemaebene erkennt, dass kein sicherer vorangehender Schreibzugriff existiert und liefert *paper* als kritische Variable zurück.
Auf Instanzebene werden zunächst beide Instanzen als migrierbar eingestuft, da ihre Ausführung noch nicht über die Stelle hinaus fortgeschritten ist, an der die Prozess-Änderung erfolgt. *ILDataFlowValid* gibt für den Ausführungszustand 1 wie erwartet *false* zurück. Im Zustand 2 wird die Variable *paper* als bereits beschrieben erkannt, in diesem Fall tritt somit kein Datenflussproblem auf.
Der Benutzer wird auf die Datenflussprobleme hingewiesen und hat die Wahl, die Instanzen trotzdem zu migrieren oder dies nicht zu tun.

7.2.3 Registrierung eines Benutzers – Löschoperation

Auch für die Überprüfung der Adaptionsoperation *Serielles Löschen einer einfachen Aktivität* definieren wir zwei Ausführungszustände: Instanz-Zustand 1 ist erreicht, nachdem die Aktivität *Erfassung des Nachnamens* beendet wurde. Die Variable *nachname* wurde beschrieben und damit initialisiert. Auf beide Variablen, *nachname* und *vorname*, wird im weiteren Verlauf des Prozesses noch lesend zugegriffen.
Zustand 2 ist der Prozess-Zustand, den die Instanz hat, nachdem die Auswahl *speichern oder abbrechen* entgegengenommen und der Task beendet wurde. Unter anderem wurde der Task *Erfassung des Vornamens* bereits ausgeführt und die Variable *vorname* beschrieben. Es existieren weiterhin nachfolgende Lesezugriffe auf beide Variablen.
Abbildung 7.2 verdeutlicht die beiden Ausführungszustände noch einmal bildlich.

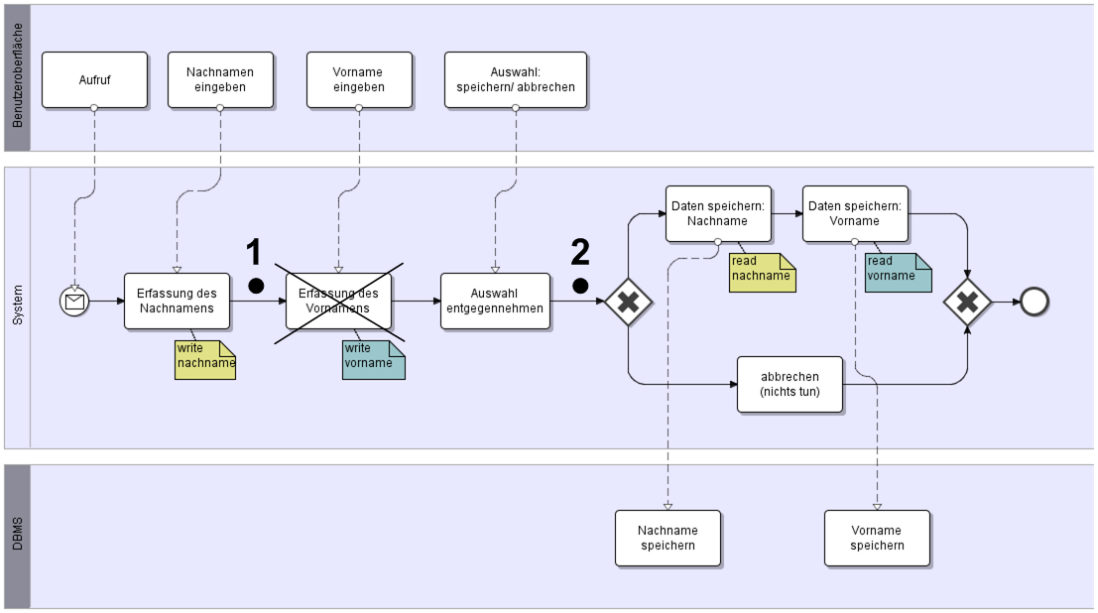


Abbildung 7.2: Löschoperation

Erwartetes Verhalten

Im Zustand 1 befindet sich der aktuelle Stand der Prozessausführung noch *vor* dem zu löschenden Task. Eine Anwendung der Änderungsoperation mit anschließender Migration der Instanz ist von daher also prinzipiell möglich.
Allerdings führt das Löschen des Tasks, der auf die Variable *vorname* schreibend zugreift, zu Schwierigkeiten hinsichtlich des Datenflusses. Die Variable *vorname* soll innerhalb der Verzweigung eventuell gelesen werden. Nach dem Löschen des Tasks *Erfassung des Vornamens* wird diese im Vorfeld aber nie beschrieben. Unsere Adaptionskomponente sollte erkennen, dass ein Datenfluss-Problem auf Schemaebene vorliegt. Da der Task *Erfassung des Vornamens* auch nicht bereits ausgeführt wurde, bevor er gelöscht werden soll, liegt auch auf Instanzebene ein Datenflussproblem vor.
Eine Adaption dieser Prozessinstanz kann damit nicht erfolgreich durchgeführt werden bzw. wird von einer Migration abgeraten.

Für die Prozessinstanz, die sich im Zustand 2 befindet, stellt sich die Situation anders dar. Bezüglich des Datenflusses treten bei ihr keine Schwierigkeiten auf. Zwar ist auch hier das neue Prozess-Schema selbst fehlerhaft, d.h. *SLDataFlowValid* wird zu *false* evaluiert. Da aber der zu löschende Task bereits ausgeführt wurde, wurde die Variable *vorname* bereits beschrieben und kann somit im späteren Verlauf problemlos gelesen werden.
Der Haken an diesem Fall liegt darin, dass eine Änderung der (Prozess-)Vergangenheit durchgeführt werden soll. Ein Task, der bereits ausgeführt wurde, soll gelöscht werden. Wir haben solche Änderungen an bereits durchlaufenden Prozessteilen ausgeschlossen, da externe Effekte teilweise nicht mehr rückgängig zu machen wären. Aus diesem Grund kann auch diese Instanz nicht erfolgreich auf das neue Prozess-Schema migriert werden.

Erfolgtes Verhalten

Zur Validierung der Löschoperation rufen wir die Adaptionsoperation *BasicActivitySerialRemove* auf und übergeben den Namen der zu löschenden Aktivität (*Erfassung des Vornamens*) wie auch Referenzen auf das alte und das neue Prozess-Schema. Der Adaptionsalgorithmus stellt fest, dass eine Aktivität gelöscht werden soll, die die Variable *vorname* beschreibt. *SLDataFlowValid* findet zwar mögliche nachfolgende Lesezugriffe auf diese Variable, aber keine weitere Schreibaktivität. Damit ist der Datenfluss auf Schema-Ebene fehlerhaft, die Funktion gibt *vorname* als kritische Variable zurück.
Auf Instanzebene wird Instanz 2 aufgrund ihres fortgeschrittenen Ausführungszustandes als nicht migrierbar erkannt. Bei der Instanz, die sich in Zustand 1 befindet, wirkt sich das Datenfluss-Problem der Schemaebene direkt auf die Instanzebene aus. *ILDataFlowValid* liefert hier *false* als Ergebnis, da die zu löschende Aktivität noch nicht ausgeführt wurde.
Für Instanz 1 besteht für den Benutzer die Möglichkeit, trotz fehlerhaftem Datenfluss auf Schema- wie auch auf Instanzebene eine Migration durchzuführen.

7.3 Ergebnis der Validierung

Wir haben unser Konzept und unsere Implementierung in Bezug auf die Erhaltung der Korrektheit und der Konsistenz anhand von vier möglichst aussagekräftigen Fällen validiert. Hierbei mussten wir aufgrund der nur teilweise vorhandenen Systemumgebung Einschränkungen vornehmen. Die wesentlichen Aspekte unseres Adaptionskonzeptes konnten allerdings abgedeckt werden.
Das Verhalten der Adaptionskomponente bei der Durchführung der Validierung entsprach unseren Erwartungen. Unser Adaptionskonzept hat sich damit in einer praxisnahen Testumgebung

bewährt.

In Kapitel 3 hatten wir neben dem Erhalt der Korrektheit und der Konsistenz weitere Anforderungen an unser Adaptionskonzept gestellt. Wie wir vorangehend aufgezeigt haben, konnten diese ebenfalls erfüllt werden.

Kapitel 8

Zusammenfassung und Ausblick

In diesem abschließenden Kapitel fassen wir die Ergebnisse unserer Arbeit zusammen und geben einen Ausblick auf mögliche weiterführende Vertiefungen.

8.1 Zusammenfassung

Das Ziel dieser Arbeit bestand darin, ein Adaptions-Konzept für Änderungen am Prozessablauf zu entwickeln. Die Vorgabe war, auf dem Prozessmodellierungs-Standard BPMN und dem Prozessdefinitions-Standard BPEL aufzusetzen. Das Konzept sollte einen Weg aufzeigen, wie Änderungen am Prozessmodell von der BPMN-Ebene, über die BPEL-Ebene, bis hin zur Instanzausführungsebene propagiert werden können. Dieses Ziel konnte erreicht werden.

Bei der Entwicklung des Konzeptes zeigte es sich, dass ein sinnvolles Adaptionsmodell mit unseren Mitteln nur über eine Vorgabe fester Änderungsoperationen zu erreichen ist. Diese können bei einer späteren Erweiterung des Konzeptes allerdings als Basis für die Umsetzung freier Prozessänderungen dienen. Eine vom Benutzer aufgerufene Änderungsoperation setzen wir in eine BPMN-Modell-Änderungsfunktion und eine entsprechende Adaptionsoperation auf Instanzebene um.

Als Voraussetzung für die Spezifizierung dieser Operationen mussten wir zunächst einmal ein formales Modell sowohl für die BPMN- wie auch für die Instanzebene entwickeln. Diese beiden Modelle stehen nun für die Definition beliebiger Adaptionsoperationen zur Verfügung.

Die BPMN-Ebene selbst stellte sich hinsichtlich des Adaptionsvorgangs als weniger bedeutend heraus. Der interessante Teil der Adaption findet auf der BPEL- bzw. auf Instanzebene statt. Als Herausforderungen identifizierten wir dort die Ausführungszustandsüberprüfung, die notwendige Datenflussanalyse und die eigentliche Migrierung der Prozess-Instanzen.

Insbesondere die Schwierigkeiten in Zusammenhang mit der Migration sind bedingt durch die Art der Übersetzung von BPMN nach BPEL. Zum einen erfolgt keine 1:1-Abbildung von BPMN-Elementen auf BPEL-Aktivitäten. Zum anderen existiert auch keine standardisierte Übersetzungsspezifikation für die Transformation eines BPMN 1.x-Modells auf eine BPEL 2.0-Prozessdefinition. Die Analyse des Datenflusses ist deshalb notwendig, da auf Instanzebene Datenflussprobleme auftreten können, die auf der Ebene der Prozessmodellierung für den Benutzer nicht ersichtlich sind. Weist das BPEL-Prozessmodell nach Durchführung einer Änderungsoperation einen korrekten Datenfluss auf, kann es trotzdem sein, dass der Datenfluss der adaptierten Instanz fehlerhaft ist.

Es ist uns gelungen, für alle genannten Herausforderungen Lösungen zu entwickeln. Hinsichtlich der möglichen Ausführungszustände, in denen eine Adaption durchgeführt werden kann, haben wir uns für ein striktes Vorgehen entschieden. So wirken sich Änderungen am Prozessmodell nur dann auf Instanzen aus, wenn deren Ausführung die Stelle im Prozessablauf, an der die Änderung erfolgen soll, noch nicht erreicht hat. Es erfolgt also keine Änderung der „Prozess-Vergangenheit“. Dieser Ansatz ist einerseits naheliegend, andererseits aber auch auf unseren Verzicht auf die Betrachtung von Schleifen und Compensations zurückzuführen. Bei der Erweiterung des Adaptionskonzeptes um diese Aspekte kann es sinnvoll sein, Änderungen an schon durchlaufenen Prozessabschnitten zuzulassen. Unser Adaptionskonzept lässt sich hierauf anpassen. Anpassungen zur Unterstützung dieser Entwurfsentscheidung wären insbesondere im Bereich der Instanz-Zustandsüberprüfung und der Datenflussanalyse notwendig.

Für die Datenflussüberprüfung haben wir ein umfangreiches, aber intuitiv nachvollziehbares Verfahren entwickelt. Es basiert auf einer Definition-Use-Analyse, die direkt auf der BPEL-Prozessdefinition abläuft und feststellt, ob Variablen initialisiert werden, bevor schreibend auf sie zugegriffen wird. Aus den gewonnenen Informationen können wir die Korrektheit des Datenflusses auf Schemaebene ablesen. Durch eine Verknüpfung mit dem Ausführungszustand der jeweiligen Instanz lässt sich hiermit aber auch die Datenfluss-Korrektheit auf Instanzebene überprüfen.

Im Rahmen der Erarbeitung eines Migrationskonzeptes haben wir detailliert die einzelnen Fälle beschrieben, die hierbei zu beachten sind. Auf dieser Basis wurde ein Vorgehen entwickelt, wie eine Instanz auf ein abgeändertes Prozess-Schema migriert werden kann.

Bei der Ausarbeitung unseres Adaptionskonzeptes mussten wir gewisse Einschränkungen vornehmen: Sowohl auf BPMN- wie auch auf BPEL-Ebene setzen wir ein vollständig blockstrukturiertes Prozessmodell voraus. Diese Voraussetzung bedeutet keine grundsätzliche Einschränkung der Ausdrucksmächtigkeit der betroffenen Prozessmodelle, vielmehr zwingt sie den Prozessentwickler zu einer sauberen Prozessmodellierung. Unsere Annahme wirkt sich insbesondere auf die Verwendung von Gateways auf BPMN-Ebene und den Einsatz von *flow*-Aktivitäten in BPEL aus. Ebenfalls auf BPEL-Ebene fanden Compensation-, Event-, Fault- und Termination-Handler im Rahmen unseres Konzeptes keine Beachtung. Auch haben wir nicht untersucht, wie sich Adaptionsvorgänge auf die Behandlung von Schleifen auswirken.

Unser Adaptionskonzept wurde als unabhängige Komponente implementiert. Wir haben überprüft, ob eine Integration in die Workflow-Management-Systeme FloeCom und Intalio möglich ist und sind zu dem Ergebnis gekommen, dass hierfür bei beiden Systemen Anpassungen vorzunehmen sind. Dies betrifft insbesondere die Erzeugung des BPEL-Prozessmodells durch den BPEL-Compiler und die Zustandshaltung der Prozessaktivitäten. Bei der Übersetzung des BPMN-Modells nach BPEL müssen – zusätzlich zu den einfachen Aktivitäten – auch die strukturierten Aktivitäten durch den BPEL-Compiler eindeutig benannt werden. Bislang ist dies nicht der Fall. Hinsichtlich der Zustandshaltung benötigen wir Zugriff auf die aktuellen Ausführungszustände aller BPEL-Aktivitäten einer Instanz. Momentan werden diese Informationen aber weder im FloeCom-System noch im Workflow-Management-System von Intalio in unmittelbar auslesbarer Form vorgehalten.

8.2 Ausblick

Diese Arbeit zeigt ein durchgehendes Adaptions-Konzept für Prozessänderungen auf der Basis von BPMN- bzw. BPEL-Prozessmodellen auf. Zur Vertiefung dieses Konzeptes sind weiterführende, insbesondere auch spezialisiertere Arbeiten notwendig.

So ist beispielsweise zu untersuchen, wie im Rahmen des Adaptionsvorganges eine sinnvolle Benutzerführung aussehen kann.

Auf Prozessebene ist das Konzept um die von uns nicht betrachteten Aspekte, wie Schleifen, Zurücksetzungen und Fehlerbehandlungen, zu erweitern.

Unabhängig von unserem Adaptionskonzept ist die weitere wissenschaftliche Untersuchung der Datenflussanalyse in BPEL und die Entwicklung hierfür geeigneter Verfahren notwendig. Das Ziel sollte die Bereitstellung einer Bibliothek von Analysefunktionen sein, auf die mit geringem Integrations-Aufwand zurückgegriffen werden kann.

Hinsichtlich der Weiterentwicklung der Spezifikationen wäre die Definition von Ausführungszuständen für BPEL-Aktivitäten hilfreich. Verwendet der Hersteller des Workflow-Management-Systems für die Zustandshaltung der Instanzen eine proprietäre Zustandsbeschreibung, müssen wir zumindest die Möglichkeit haben, die herstellerspezifischen Zustände auf die von uns verwendeten abzubilden. Würde die BPEL-Spezifikation Ausführungszustände definieren, könnte der Hersteller diese Aufgabe übernehmen und so die Anbindung beliebiger (Adaptions-)Komponenten ermöglichen, die Zugriff auf diese Informationen benötigen.

Unbedingt notwendig für die Erweiterung unseres Konzeptes um komplexere Adaptionsoperationen ist eine Standardisierung der Transformation von BPMN-Modellen nach BPEL, jedenfalls soweit eine solche Übersetzung möglich ist.

– Anhang –

A Anhang: Implementierung

Im Folgenden ist der detaillierte Ablauf der Adaptionsoperationen *BasicActivitySerialInsert* und *BasicActivitySerialRemove* aufgezeigt, wie er implementiert wurde.

A.1 BasicActivitySerialInsert

Die Klasse *BasicActivitySerialInsert* im Paket *adaption_operations* enthält die Ablauflogik zur Durchführung der gleichnamigen Adaptionsoperation.

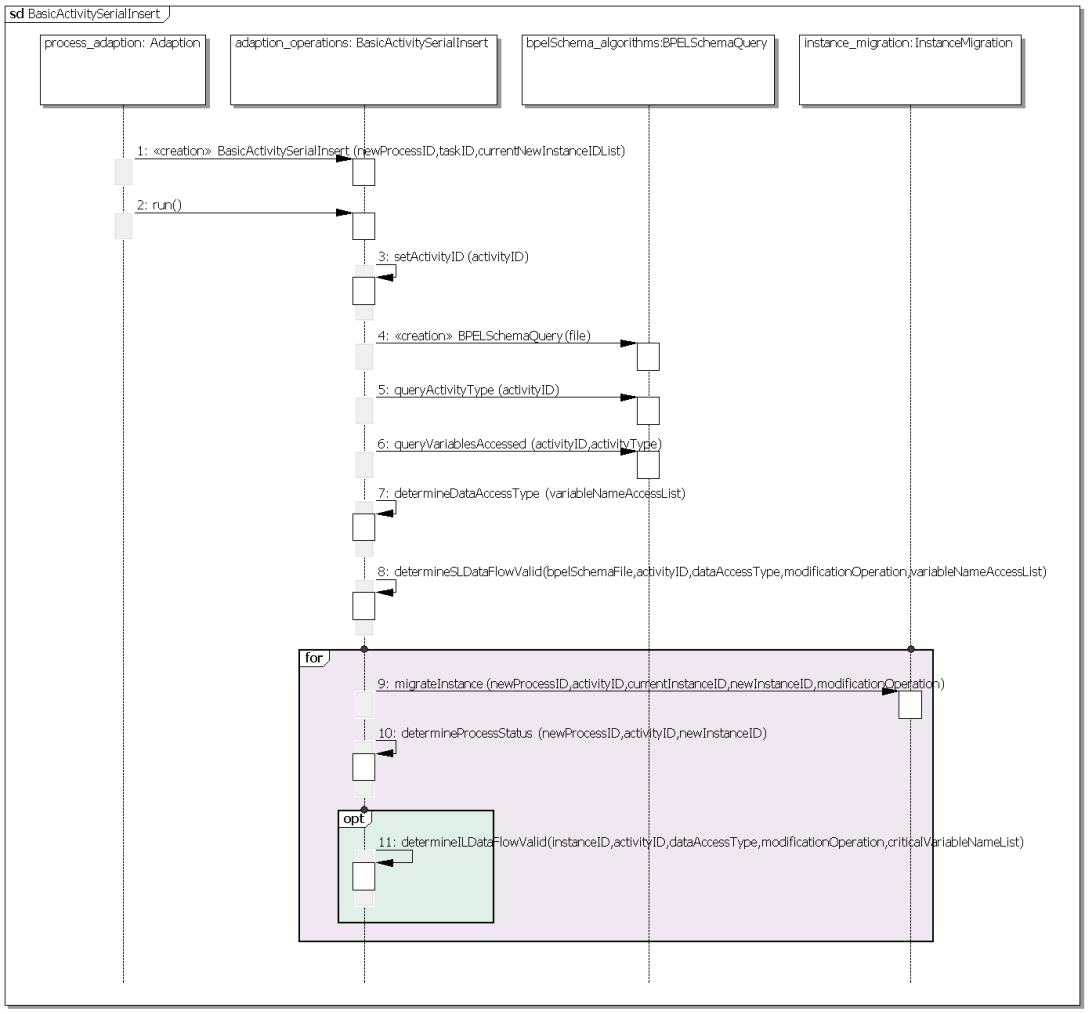


Abbildung A.1: *BasicActivitySerialInsert*-Implementierung – Sequenzdiagramm

A.2 BasicActivitySerialRemove

Zur Durchführung der Adaptionsoperation *BasicActivitySerialRemove* wird die gleichnamige Klasse im Paket *adaption_operations* instanziiert und ihre *run*-Methode aufgerufen.

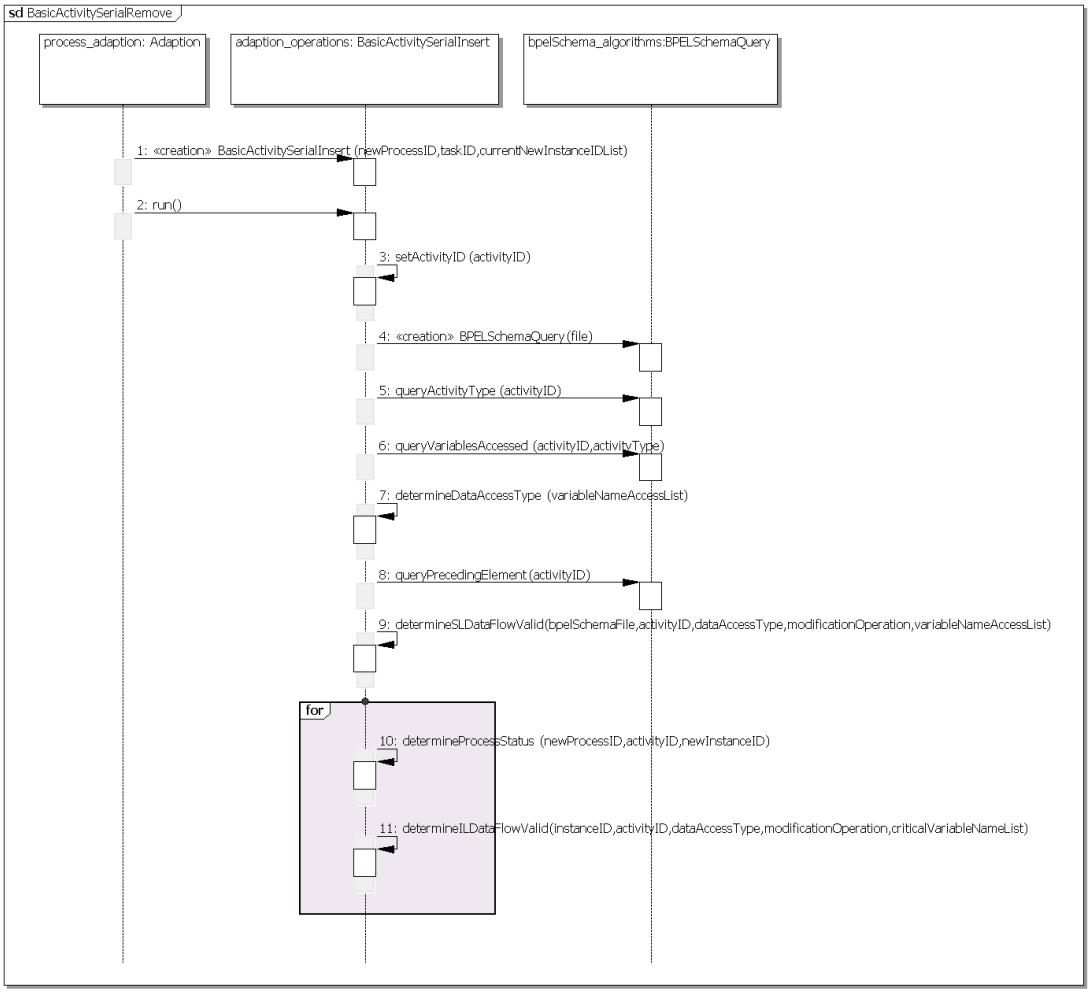


Abbildung A.2: *BasicActivitySerialRemove*-Implementierung – Sequenzdiagramm

B Anhang: Validierungs-Szenario

Auf den folgenden Seiten stellen wir nochmals die Prozesse dar, auf die wir für die Validierung in Kapitel 7 zurückgegriffen haben. Gezeigt sind jeweils die BPMN-Darstellung inklusive der skizzierten Änderungsoperation und die BPEL-Prozessdefinition.

B.1 Einfügeoperation

Einfügen eines neuen Tasks im Szenario *Hochladen eines Konferenzbeitrages*.

BPMN-Darstellung

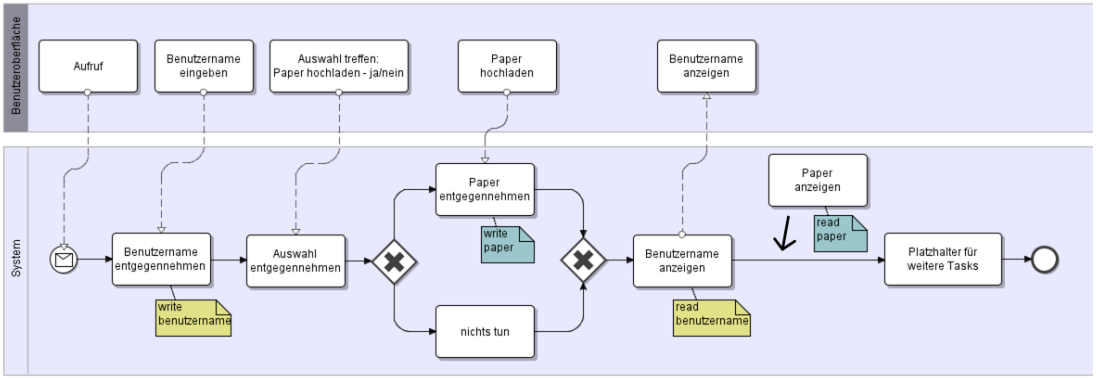


Abbildung B.3: Einfügeoperation

BPEL-Prozessdefinition

Die untenstehende Prozessdefinition stellt die BPEL-Repräsentation des Prozesses *nach* Durchführung der Änderungsoperation, also nach dem Einfügen des neuen Tasks, dar. Teilweise wurden für uns nicht relevante Elemente entfernt, um die Übersichtlichkeit zu erhöhen.

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:this="http://example.com/Einfuegeoperation/System"
  xmlns:diag="http://example.com/Einfuegeoperation"
  ...
  queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
  expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
  name="System" targetNamespace="http://example.com/Einfuegeoperation/System">
  <import namespace="http://example.com/Einfuegeoperation"
    location="Einfuegeoperation.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
  ...
  <partnerLinks>
    <partnerLink name="systemAndBenutzeroberflaechePlkVar"
      partnerLinkType="diag:SystemAndBenutzeroberflaeche" initializePartnerRole="true"
      myRole="System_for_Benutzeroberflaeche"
      partnerRole="Benutzeroberflaeche_for_System"/>
  </partnerLinks>
```

```
  <variables>
    <variable name="eventStart" messageType="this:eventStartMsg"/>
    <variable name="benutzername" messageType="this:benutzernameMsg"/>
    <variable name="paper" messageType="this:paperMsg"/>
    <variable name="auswahl" messageType="this:auswahlMsg"/>
  </variables>
  <sequence name="sequence1">
    <receive partnerLink="systemAndBenutzeroberflaechePlkVar"
      portType="this:ForBenutzeroberflaeche" operation="EventStartMessage"
      variable="eventStart" createInstance="yes" name="EventStartMessage">
    </receive>
    <receive partnerLink="systemAndBenutzeroberflaechePlkVar"
      portType="this:ForBenutzeroberflaeche"
      operation="Benutzername_entgegennehmen" variable="benutzername"
      name="Benutzername entgegennehmen">
    </receive>
    <receive partnerLink="systemAndBenutzeroberflaechePlkVar"
      portType="this:ForBenutzeroberflaeche" operation="Auswahl_entgegennehmen"
      variable="auswahl" name="Auswahl entgegennehmen">
    </receive>
    <if name="if1">
      <condition>$auswahl.body/text() = "paper"</condition>
      <sequence name="sequence2">
        <receive partnerLink="systemAndBenutzeroberflaechePlkVar"
          portType="this:ForBenutzeroberflaeche" operation="Paper_entgegennehmen"
          variable="paper" name="Paper entgegennehmen">
        </receive>
      </sequence>
      <elseif>
        <condition>$auswahl.body/text() = "doNothing"</condition>
        <sequence name="sequence3">
          <empty name="nichts tun"/>
        </sequence>
      </elseif>
    </if>
    <invoke partnerLink="systemAndBenutzeroberflaechePlkVar"
      portType="Benutzeroberflaeche:ForSystem" operation="Benutzername_anzeigen"
      inputVariable="benutzername" name="Benutzername anzeigen">
    </invoke>
    <invoke partnerLink="systemAndBenutzeroberflaechePlkVar"
      portType="Benutzeroberflaeche:ForSystem" operation="Paper_anzeigen"
      inputVariable="paper" name="Paper anzeigen">
    </invoke>
    <empty name="Platzhalter fuer weitere Tasks"/>
    <empty name="EventEndEmpty"/>
  </sequence>
</process>
```

B.2 Löschoperation

Löschen eines Tasks in Szenario *Benutzerregistrierung*.

BPMN-Darstellung

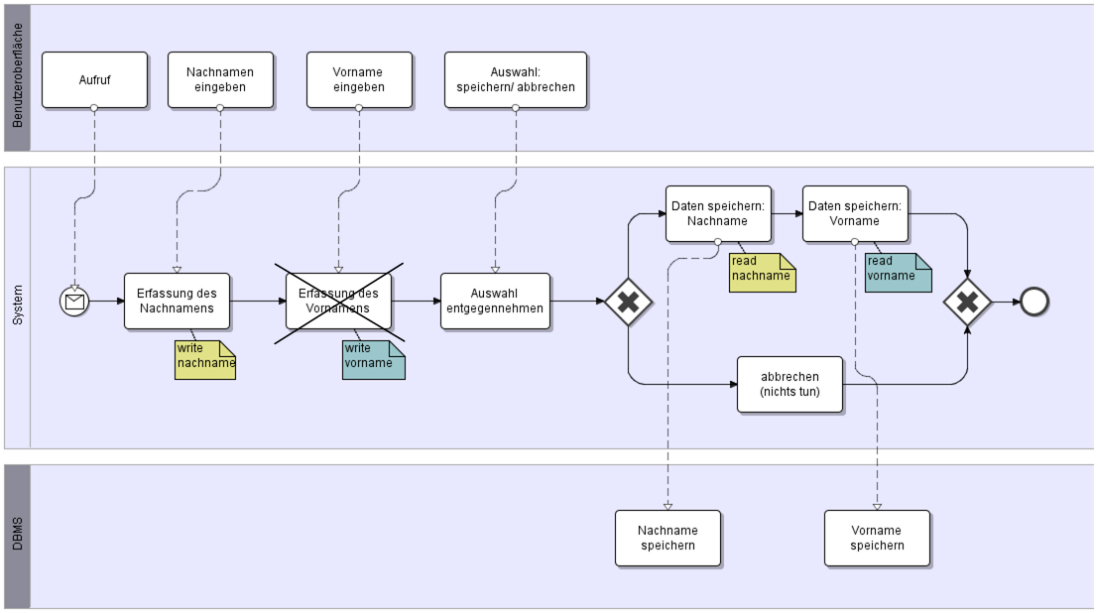


Abbildung B.4: Löschoperation

BPEL-Prozessdefinition

Die untenstehende BPEL-Prozessdefinition ist das Ergebnis der Übersetzung *bevor* das Prozessmodell geändert wurde. Die zu löschende Aktivität ist also noch enthalten.

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:this="http://example.com/Löschoperation/System"
  xmlns:diag="http://example.com/Löschoperation"
  ...
  queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
  expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
  name="System" targetNamespace="http://example.com/Löschoperation/System">
  <import namespace="http://example.com/Löschoperation"
    location="Löschoperation.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
  ...
  <partnerLinks>
    <partnerLink name="systemAndBenutzeroberflächePlkVar"
      partnerLinkType="diag:SystemAndBenutzeroberfläche"
      myRole="System_for_Benutzeroberfläche"/>
    <partnerLink name="dBMsAndSystemPlkVar"
      partnerLinkType="diag:DBMsAndSystem" initializePartnerRole="true"
      partnerRole="DBMS_for_System"/>
```

```
  </partnerLinks>
  <variables>
    <variable name="eventStart" messageType="this:eventStartMsg"/>
    <variable name="nachname" messageType="this:nachnameMsg"/>
    <variable name="vorname" messageType="this:vornameMsg"/>
    <variable name="auswahl" messageType="this:auswahlMsg"/>
  </variables>
  <sequence name="sequence1">
    <receive partnerLink="systemAndBenutzeroberflächePlkVar"
      portType="this:ForBenutzeroberfläche" operation="EventStartMessage"
      variable="eventStart" createInstance="yes" name="EventStartMessage">
    </receive>
    <receive partnerLink="systemAndBenutzeroberflächePlkVar"
      portType="this:ForBenutzeroberfläche" operation="Erfassung_des__Nachnamens"
      variable="nachname" name="Erfassung des Nachnamens">
    </receive>
    <receive partnerLink="systemAndBenutzeroberflächePlkVar"
      portType="this:ForBenutzeroberfläche" operation="Erfassung_des_Vornamens"
      variable="vorname" name="Erfassung des Vornamens">
    </receive>
    <receive partnerLink="systemAndBenutzeroberflächePlkVar"
      portType="this:ForBenutzeroberfläche" operation="Auswahl_entgegennehmen"
      variable="auswahl" name="Auswahl entgegennehmen">
    </receive>
    <if name="if1">
      <condition>$auswahl.body/text() = "speichern"</condition>
      <sequence name="sequence2">
        <invoke partnerLink="dBMsAndSystemPlkVar"
          portType="DBMS:ForSystem" operation="Nachname_speichern"
          inputVariable="nachname" name="Daten speichern: Nachname">
        </invoke>
        <invoke partnerLink="dBMsAndSystemPlkVar"
          portType="DBMS:ForSystem" operation="Vorname_speichern"
          inputVariable="vorname" name="Daten speichern: Vorname">
        </invoke>
      </sequence>
      <elseif>
        <condition>$auswahl.body/text() = "abbrechen"</condition>
        <sequence name="sequence3">
          <empty name="abbrechen (nichts tun)"/>
        </sequence>
      </elseif>
    </if>
    <empty name="EventEndEmpty"/>
  </sequence>
</process>
```

C Anhang: Algorithmen

C.1 Adaptionsprimitive auf BPEL- bzw. Instanzebene

Die folgenden Funktionen werden auf den nächsten Seiten in Pseudocode-Darstellung vorgestellt:

- DetermineNewActivityStatus
- FindReadingActivitiesForward
- FindWritingActivitiesBackward
- GetActivityStatus
- GetActivityType
- GetDataAccessType
- GetInitializationStatus
- GetPrecedingElement
- GetProcessStatus
- GetSucceedingElement
- GetVariablesAccessed
- ILDataFlowValid
- MigrateInstance
- PrecedingWriter
- ReaderDeepSearch
- ReadingActivity
- setActivityStatus
- SLDataFlowValid
- SucceedingReader
- UserDecisionPerformMigration
- UserInformation
- WriterDeepSearch
- WritingActivity

DetermineNewActivityStatus

Diese Funktion bestimmt den Ausführungszustand, der einer neu eingefügten Aktivität zugewiesen werden muss. Hierfür werden zunächst die Zustände der umgebenden Aktivitäten abgefragt. An diese Ausführungszustände wird der Status der neu hinzugekommenen Aktivität angepasst. Das genaue Vorgehen ist von der Adaptionsoperation abhängig.
Bisher unterstützt der Algorithmus nur die Operationen *SerialInsertionOfABasicActivity* und *SerialRemovalOfABasicActivity*. (Im letzteren Fall ist keine Zustandsanpassung notwendig.) Für weitere, zukünftig einzubindende Adaptionsoperationen kann dieser Algorithmus recht komplex werden, wie in Kapitel 5.4.2 aufgezeigt wurde.

Aufruf:

ActivityStatus DetermineNewActivityStatus(*ProcessIdentifier*, *ActivityName*, *InstanceIdentifier*, *ModificationOperation*)

Rückgabewert / Parameter:

string *ActivityStatus*
string *ProcessIdentifier*
string *ActivityName*
string *InstanceIdentifier*
string *ModificationOperation*

Vorbedingungen:

ProcessIdentifier ∈ Instance.ProcessDefinition[1]
ActivityName ∈ Instance.ProcessDefinition.Activities
InstanceIdentifier ∈ Instances
ModificationOperation ∈ {"SerialInsertionOfABasicActivity", "SerialRemovalOfABasicActivity"}

Semantik:

begin

string succElementType
string succElementName
string succElementStatus
string actStatus

SWITCH ModificationOperation

case "SerialInsertionOfABasicActivity" **do**

(succElementType, succElementName) := GetSucceedingElement(*ProcessIdentifier*, *ActivityName*)
succElementStatus := GetActivityStatus(succElementName, *InstanceIdentifier*)

if succElementType = "succeedingActivity" **then**
if succElementStatus = "None" **then**

[1] Verkürzte Schreibweise. Gemeint ist die Instanz, die über den *InstanceIdentifier* referenziert wird.

```
        actStatus := "None"
      else
        actStatus := "Completed"
      end if
    end if

    if succElementType = "structActEndTag" then
      if succElementStatus ≠ "Completed" then
        actStatus := "None"
      else
        actStatus := "Completed"
      end if
    end if

    break
  end case

  return actStatus

  end
```

FindReadingActivitiesForward

Die Funktion *FindReadingActivitiesForward* überprüft auf Schemaebene, ob im Prozessablauf nach der Aktivität *ActivityName* ein Lesezugriff auf die angegebenen Variablen erfolgt. Die Funktion liefert ein Tupel, das für jede Variable die erste gefundene Leseaktivität enthält. Wurde keine Leseaktivität gefunden, nimmt das zugehörige Aktivitätselement des Tupels den Wert EMPTY an.
Die eigentliche Suche nach Leseaktivitäten übernimmt der *SucceedingReader*-Algorithmus. Dieser wird für jede Variable einmal aufgerufen.

Aufruf:

{(*ActivityName*, *VariableName*)} FindReadingActivitiesForward(*ProcessIdentifier*, *ActivityName*, {*VariableName*})

Rückgabewert / Parameter:

string *ActivityName*
string *VariableName*
string *ProcessIdentifier*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

string VarName
string ActName
string RActName
string RVarName

{(RActName, RVarName)} = ∅

for each VarName ∈ {*VariableName*} **do**
  ActName := SucceedingReader(*ProcessIdentifier*, *ActivityName*, VarName)
  {(RActName, RVarName)} := {(RActName, RVarName)} ∪ {(ActName, VarName)}
end for

return {(RActName, RVarName)}

end

FindWritingActivitiesBackward

Überprüft auf Schemaebene, ob die angegebenen kritischen Variablen vor dem Lesezugriff der Aktivität *ActivityName* geschrieben werden. Hierzu wird für jede Variable der Suchalgorithmus *PrecedingWriter* aufgerufen.
Die Funktion gibt ein Tupel zurück, das für jede Variable die zugehörige Schreibaktivität nennt. Existiert keine Schreibaktivität, bleibt das Aktivitätselement im Tupel leer.

Aufruf:

{(*ActivityName*, *VariableName*)} FindWritingActivitiesBackward(*ProcessIdentifier*, *ActivityName*, {*VariableName*})

Rückgabewert / Parameter:

string *ActivityName*
string *VariableName*
string *ProcessIdentifier*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

string VarName
string ActName
string WActName
string WVarName

{(WActName, WVarName)} = ∅

for each VarName ∈ {*VariableName*} **do**
  ActName := PrecedingWriter(*ProcessIdentifier*, *ActivityName*, VarName)
  {(WActName, WVarName)} := {(WActName, WVarName)} ∪ {(ActName, VarName)}
end for

return {(WActName, WVarName)}

end

GetActivityStatus

Liefert den Status der Aktivität mit dem Namen *ActivityName*.

Aufruf:

ActivityStatus GetActivityStatus(*ActivityName*, *InstanceIdentifier*)

Rückgabewert / Parameter:

string *ActivityStatus*
string *ActivityName*
string *InstanceIdentifier*

Vorbedingungen:

ActivityName ∈ Instance.ProcessDefinition.Activities
InstanceIdentifier ∈ Instances

Semantik:

begin

string actStatus

$\forall i \in \{\text{Instances} \mid \text{i.InstanceIdentifier} = \textit{InstanceIdentifier}\}$:
actStatus := i.ActivityStatus(*ActivityName*)

return actStatus

end

GetActivityType

Liefert den BPEL Aktivitätstyp der Aktivität zurück, also beispielweise *assign*, *invoke* usw. Ist die übergebene „Aktivität“ das Prozess-Element selbst, ist auch die Rückgabe von *process* erlaubt.

Aufruf:

ActivityType GetActivityType(*ProcessIdentifier*, *ActivityName*)

Rückgabewert / Parameter:

string *ActivityType*
string *ProcessIdentifier*
string *ActivityName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities ∪ {"process"}

Semantik:

begin

sting actType

actType := BPEL-Element, dessen *name*-Attribut *ActivityName* ist. Hiervon die Elementbezeichnung.

return actType

end

GetDataAccessType

Bestimmt in Abhängigkeit von der Art der Variablenzugriffe den Datenzugriffs-Typ der Aktivität, also *readActivity*, *writeActivity*, *readWriteActivity* oder *noVariableAccessActivity*.

Aufruf:

DataAccessType GetDataAccessType({(*VariableName*, *VariableAccess*)})

Rückgabewert / Parameter:

string *DataAccessType*
string *VariableName*
string *VariableAccess*

Vorbedingungen:

VariableAccess ∈ {"read", "write"}

Semantik:

```
begin

bool r
bool w
string DAType

r := false
w := false

for each t ∈ {(VariableName, VariableAccess)} do
  if t.VariableAccess = "read" then
    r := true
  end if
  if t.VariableAccess = "write" then
    w := true
  end if
end for

if r = true AND w = true then
  DAType := "readWriteActivity"
end if
if r = true AND w = false then
  DAType := "readActivity"
end if
if r = false AND w = true then
  DAType := "writeActivity"
end if
if r = false AND w = false then
  DAType := "noVariableAccessActivity"
end if

return DAType

end
```

GetInitializationStatus

Stellt fest, ob die Variable *VariableName* bereits initialisiert wurde.

Aufruf:

bool GetInitializationStatus(*InstanceIdentifier*, *VariableName*)

Parameter:

string *InstanceIdentifier*
string *VariableName*

Vorbedingungen:

InstanceIdentifier ∈ Instances
VariableName ∈ Instance.ProcessDefinition.Variables

Semantik:

```
begin

bool variableInitialized

for each {i ∈ Instances | i.InstanceIdentifier = InstanceIdentifier} do
  if i.VariableName = null then
    variableInitialized := false
  else
    variableInitialized := true
  end if
end for

return  variableInitialized

end
```

GetPrecedingElement

Bestimmt das vorangehende BPEL-Element auf XML-Ebene, also entweder eine Aktivität oder das Start-Tag der umgebenden strukturierten Aktivität. Zurückgegeben wird der Typ des Vorgängerelementes und sein Name.
Der Algorithmus ist ein vereinfachter *PrecedingWriter*-Algorithmus. Zur Verdeutlichung des Ablaufs siehe Abbildung C.5. Der Spezialfall eines Prozesses, der nur aus einer Aktivität besteht, ist nicht abgedeckt.

Aufruf:

(*precElementType*, *precElementName*) GetPrecedingElement(*ProcessIdentifier*, *ActivityName*)

Rückgabewert / Parameter:

string *precElementType*
string *precElementName*
string *ProcessIdentifier*
string *ActivityName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities

Semantik:

begin

string parentActName
string parentActType
string precedingSibling
string precElementName
string precElementType

parentActName := /parent von *ActivityName*, davon das name-Attribut. (Falls /parent keine Activity und nicht *process* ist ist, sondern z.B. ein *if* oder *onAlarm*, dann ist dessen /parent zu nehmen.)
parentActType := GetActivityType(*ProcessIdentifier*, parentActName)

if parentActType = "flow" OR parentActType = "forEach" OR parentActType = "if" OR parentActType = "pick" OR parentActType = "repeatUntil" OR parentActType = "scope" OR parentActType = "while" **then**
  precElementName := parentActName
  precElementType := "structActStartTag"
end if

if parentActType = "sequence" **then**
  precedingSibling := direkte Vorgängeraktivität von *ActivityName* (Name hiervon)
  if precedingSibling ≠ EMPTY **then**
    precElementName := precedingSibling
    precElementType := "precedingActivity"
  else

```
      precElementName := parentActName
      precElementType := "structActStartTag"
    end if
  end if

  return (precElementType, precElementName)

  end
```

***(precElementType, precElementName)* GetPrecedingElement (*ProcessIdentifier, ActivityName*)**

Initialisierung:
x := *ActivityName*

* leicht vereinfachte Darstellung:
Ist x/parent ein *elseif* oder *else*-Element innerhalb einer *If*-Aktivität, so wird das *If*-Element zurückgegeben.
Analog gilt dies für den Fall, dass die Aktivität x innerhalb der Elemente *onMessage* oder *onAlarm* der *pick*-Aktivität liegt.

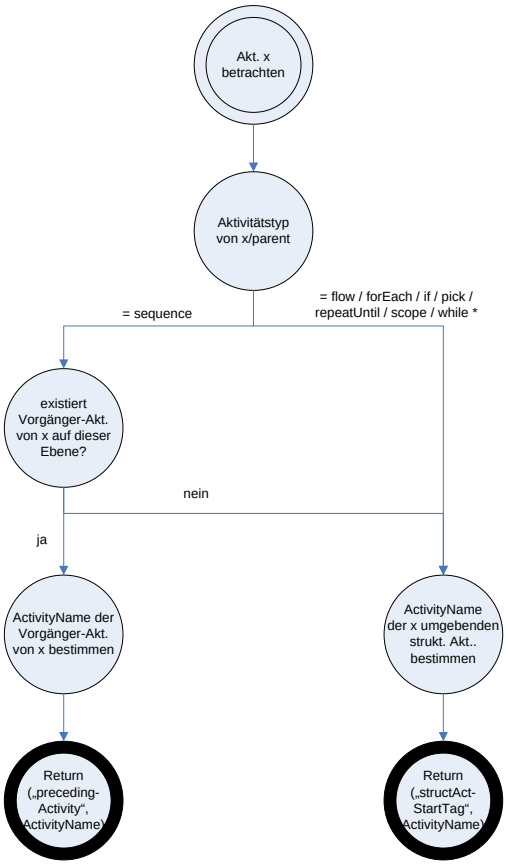


Abbildung C.5: *GetPrecedingElement*-Algorithmus

GetProcessStatus

Stellt anhand des Zustandes der genannten Aktivität fest, ob sich die Instanz noch in einem migrierbaren Ausführungszustand befindet, oder ob die Ausführung bereits zu weit fortgeschritten ist.

Aufruf:

bool GetProcessStatus(*ProcessIdentifier*, *ActivityName*, *InstanceIdentifier*)

Parameter:

string *ProcessIdentifier*
string *ActivityName*
string *InstanceIdentifier*

Vorbedingungen:

ProcessIdentifier ∈ Instance.ProcessDefinitions
ActivityName ∈ Instance.ProcessDefinition.Activities
InstanceIdentifier ∈ Instances

Semantik:

```
begin

string actStatus

actStatus := GetActivityStatus(ActivityName, InstanceIdentifier)
if actStatus = "None" then
   migratableStatus := true
else
   migratableStatus := false
end if

return  migratableStatus

end
```

GetSucceedingElement

Bestimmt das nachfolgende BPEL-Element auf XML-Ebene, also entweder eine Aktivität oder das End-Tag der umgebenden strukturierten Aktivität. Zurückgegeben wird der Typ des Nachfolgeelementes und sein Name.
Der Algorithmus ist ein vereinfachter SucceedingReader-Algorithmus. Zur Verdeutlichung des Ablaufs siehe Abbildung C.6. Der Spezialfall eines Prozesses, der nur aus einer Aktivität besteht, ist nicht abgedeckt.

Aufruf:

(*succElementType*, *succElementName*) GetSucceedingElement(*ProcessIdentifier*, *ActivityName*)

Rückgabewert / Parameter:

string *succElementType*
string *succElementName*
string *ProcessIdentifier*
string *ActivityName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities

Semantik:

begin

string parentActName
string parentActType
string followingSibling
string succElementName
string succElementType

parentActName := /parent von *ActivityName*, davon das name-Attribut. (Falls /parent keine Activity und nicht *process* ist ist, sondern z.B. ein *if* oder *onAlarm*, dann ist dessen /parent zu nehmen.)
parentActType := GetActivityType(*ProcessIdentifier*, parentActName)

if parentActType = "flow" OR parentActType = "forEach" OR parentActType = "if" OR parentActType = "pick" OR parentActType = "repeatUntil" OR parentActType = "scope" OR parentActType = "while" **then**
  succElementName := parentActName
  succElementType := "structActEndTag"
end if

if parentActType = "sequence" **then**
  followingSibling := direkte Nachfolgeaktivität von *ActivityName* (Name hiervon)
  if followingSibling ≠ EMPTY **then**
    succElementName := followingSibling
    succElementType := "succeedingActivity"

```
    else
      succElementName := parentActName
      succElementType := "structActEndTag"
    end if
  end if

  return (succElementType, succElementName)

  end
```

***(succElementType, succElementName)* GetSucceedingElement (*ProcessIdentifier, ActivityName*)**

Initialisierung:
x := *ActivityName*

* leicht vereinfachte Darstellung:
Ist x/parent ein *elseif* oder *else*-Element innerhalb einer *If*-Aktivität, so wird das *If*-Element zurückgegeben.
Analog gilt dies für den Fall, dass die Aktivität x innerhalb der Elemente *onMessage* oder *onAlarm* der *pick*-Aktivität liegt.

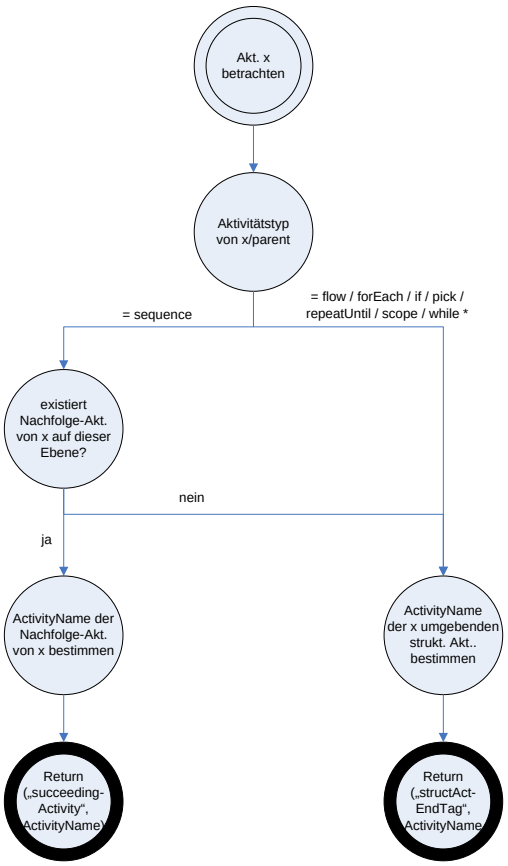


Abbildung C.6: *GetSucceedingElement*-Algorithmus

GetVariablesAccessed

Überprüft die angegeben Aktivität auf Datenelement-Zugriffe. Gibt für jeden Zugriff den Variablennamen und den Zugriffsstyp (*read* oder *write*) zurück.

Bisher ist der Algorithmus nur für einfache Aktivitäten entwickelt, da Änderungsoperationen mit komplexen Aktivitäten in dieser Arbeit nicht betrachtet werden. Eine Erweiterung des Algorithmus auf alle, auch die komplexen BPEL-Aktivitäten ist analog zu den hier aufgeführten vorzunehmen.

Aufruf:

{(*VariableName*, *VariableAccess*)} GetVariablesAccessed(*ProcessIdentifier*, *ActivityName*, *ActivityType*)

Rückgabewert / Parameter:

string *VariableName*
string *VariableAccess*
string *ProcessIdentifier*
string *ActivityName*
string *ActivityType*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
ActivityType ∈ {"assign", "empty", "exit", "flow", "invoke", "receive", "reply", "rethrow", "sequence", "throw", "wait"}

Semantik:

begin

string[ ][ ] variablesAccessed
int counter := 0

SWITCH *ActivityType* {

case "assign" **do**
  if EXISTS *ActivityName*/from-child/variable-Attribut (AttrWert:=x) **then**
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := read
    counter++
  end if
  if EXISTS *ActivityName*/to-child/variable-Attribut (AttrWert:=x) **then**
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := write
    counter++
  end if
  break
end case

```
case "invoke" do
  if EXISTS ActivityName/inputVariable-Attribut (AttrWert:=x) then
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := read
    counter++
  end if
  if EXISTS ActivityName/outputVariable-Attribut (AttrWert:=x) then
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := write
    counter++
  end if
  break
end case

case "receive" do
  if EXISTS ActivityName/variable-Attribut (AttrWert:=x) then
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := write
    counter++
  end if
  break
end case

case "reply" do
  if EXISTS ActivityName/variable-Attribut (AttrWert:=x) then
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := read
    counter++
  end if
  break
end case

case "throw" do
  if EXISTS ActivityName/faultVariable-Attribut (AttrWert:=x) then
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := read
    counter++
  end if
  break
end case

case "wait" do
  if (EXISTS AND enthältVariable ActivityName/forExpression-child (AttrWert:=x)) OR
  (EXISTS AND enthältVariable ActivityName/untilExpression-child (AttrWert:=x)) then
    variablesAccessed[counter][0] := x
    variablesAccessed[counter][1] := read
    counter++
  end if
  break
end case

default
  variablesAccessed = EMPTY
end default
```

return variablesAccessed

end

ILDataFlowValid

ILDataFlowValid greift auf die Ergebnisse der Datenflussüberprüfung auf Schemaebene zurück und ermittelt die Korrektheit des Datenflusses auf Instanzebene. Der aktuelle Ausführungszustand der zu migrierenden Instanz spielt hierbei eine entscheidende Rolle. (Siehe Abbildung 5.25.)

Aufruf:

bool ILDataFlowValid(*ProcessIdentifier*, *InstanceIdentifier*, *ActivityName*, *DataAccessType*, *ModificationOperation*, *{CriticalVariableName}*)

Parameter:

string *ProcessIdentifier*
string *InstanceIdentifier*
string *ActivityName*
string *DataAccessType*
string *ModificationOperation*
string *CriticalVariableName*

Vorbedingungen:

ProcessIdentifier ∈ Instance.ProcessDefinition
InstanceIdentifier ∈ Instances
ActivityName ∈ Instance.ProcessDefinition.Activities
DataAccessType ∈ {"readActivity", "writeActivity", "readWriteActivity", "noVariableAccessActivity"}
ModificationOperation ∈ {"SerialInsertionOfABasicActivity", "SerialRemovalOfABasicActivity"}
CriticalVariableName ∈ Instance.ProcessDefinition.Variables

Semantik:

begin

bool ILDataFlowValid
bool variableInitialized
string actStatus

SWITCH *ModificationOperation*

case "SerialInsertionOfABasicActivity" **do**

ILDataFlowValid := true
variableInitialized := true

if (DataAccessType = "readActivity" OR DataAccessType = "readWriteActivity") **then**
if *{CriticalVariableName}* ≠ ∅ **then**
for each CritVarName ∈ *{CriticalVariableName}* **do**
variableInitialized := GetInitializationStatus(*InstanceIdentifier*, CritVarName)
if variableInitialized = false **then**
ILDataFlowValid := false

```
            end if
          end for
        end if
      end if

      break

    end case

    case "SerialRemovalOfABasicActivity" do

      ILDataFlowValid := true

      if (DataAccessType = "writeActivity" OR DataAccessType = "readWriteActivity") then
        if {CriticalVariableName} ≠ ∅ then
          actStatus := GetActivityStatus(ActivityName, InstanceIdentifier)
          if actStatus = "None" then
            ILDataFlowValid := false
          end if
        end if
      end if

      break

    end case

    return  ILDataFlowValid

    end
```

MigrateInstance

Kopiert alle Variablenwerte und Aktivitätszustände von der Instanz mit dem Identifier *SourceInstanceIdentifier* auf die Instanz mit dem Identifier *TargetInstanceIdentifier*.
Die Werte der Variablen bzw. der Zustände, die in der neuen Instanz (*TargetInstanceIdentifier*) nicht vorkommen, werden nicht beachtet. Die Variablen, die in der alten Instanz nicht vorhanden sind, werden in der neuen Instanz im nichtinitialisierten Zustand belassen. Die Ausführungszustände der neu hinzugekommenen Aktivitäten werden an den Ausführungsstatus der Instanz angepasst.

Aufruf:

MigrateInstance(*NewProcessIdentifier*, *ActivityName*, *SourceInstanceIdentifier*, *TargetInstanceIdentifier*, *ModificationOperation*)

Parameter:

string *NewProcessIdentifier*
string *ActivityName*
string *SourceInstanceIdentifier*
string *TargetInstanceIdentifier*
string *ModificationOperation*

Vorbedingungen:

NewProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
SourceInstanceIdentifier ∈ Instances
TargetInstanceIdentifier ∈ Instances
ModificationOperation ∈ {"SerialInsertionOfABasicActivity", "SerialRemovalOfABasicActivity"}

Semantik:

begin

string newActStatus

∀ i ∈ {Instances | i.InstanceIdentifier = *SourceInstanceIdentifier*}
∀ j ∈ {Instances | j.InstanceIdentifier = *TargetInstanceIdentifier*}
∀ aj ∈ {j.ProcessDefinition.Activities}
∀ ai ∈ {i.ProcessDefinition.Activities | ai.getAttribute(Name) = aj.getAttribute(Name)}: [2]
ActivityStatus(aj.getAttribute(Name)) := ActivityStatus(ai.getAttribute(Name))
VariableValue(aj.getAttribute(Name)) := VariableValue(ai.getAttribute(Name))

if ModificationOperation = "SerialInsertionOfABasicActivity" **then**
  newActStatus := DetermineNewActivityStatus(*NewProcessIdentifier*, *ActivityName*, *TargetInstanceIdentifier*, *ModificationOperation*)
  setActivityStatus(*ActivityName*, *TargetInstanceIdentifier*, newActStatus)
end if

end

[2] Selektiert alle ai-Aktivitäten, deren Name auch in aj, also in der neuen Instanz, vorkommt.

PrecedingWriter

PrecedingWriter führt eine Suche nach einer der Aktivität *ActivityName* vorangehenden Schreibaktivität durch. Auf die Vorgehensweise sind wir in Kapitel 5.4.4, Abschnitt *Rückwärtssuche nach Schreibaktivitäten (FindWritingActivitiesBackward)*, ausführlich eingegangen.
Der Algorithmus liefert den Namen der Schreibaktivität, wenn eine solche vorhanden ist.

Aufruf:

ActivityName PrecedingWriter(*ProcessIdentifier*, *ActivityName*, *VariableName*)

Rückgabewert / Parameter:

string *ActivityName*
string *ProcessIdentifier*
string *VariableName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

int counter
bool localVariable
string PW
string WDS
string actType
string parentActivity
string precWriterActivity
string[ ] precedingSiblings

parentActivity := /parent von *ActivityName*, davon das name-Attribut. (Falls /parent keine Activity und nicht process ist ist, sondern z.B. ein *elseif* oder *onAlarm*, dann ist dessen /parent zu nehmen.)
actType := GetActivityType(*ProcessIdentifier*, parentActivity)

if actType = "flow" OR actType = "forEach" OR actType = "if" OR actType = "repeatUntil" OR actType = "while" **then**
  PW := PrecedingWriter(*ProcessIdentifier*, parentActivity, *VariableName*)
  if PW ≠ EMPTY **then**
    precWriterActivity := PW
  else
    precWriterActivity := EMPTY
  end if
end if

if actType = "pick" **then**
  if parentElement(*ActivityName*) = "onMessage" **then**

```
      if parentElement/variable-Attribut = variableName then
        WE := tue
      end if
    end if
    if WE = true then
      precWriterActivity := parentActivity
    else
      PW := PrecedingWriter(ProcessIdentifier, parentActivity, VariableName)
      if PW ≠ EMPTY then
        precWriterActivity := PW
      else
        precWriterActivity := EMPTY
      end if
    end if
  end if

  if actType = "sequence" then
    precedingSiblings := alle /precedingSiblings von ActivityName, mit dem letzten beginnend
    counter := 0
    while WDS = EMPTY AND counter < precedingSiblings.size do
      WDS := WriterDeepSearch(precedingSiblings[counter])
      counter++
    end while
    if WDS ≠ EMPTY then
      precWriterActivity := WDS
    else
      precWriterActivity := EMPTY
    end if
  end if

  if actType = "scope" then
    localVariable := prüfen, ob VariableName eine lokale Variable des Scopes ist
    if localVariable = false then
      PW := PrecedingWriter(ProcessIdentifier, parentActivity, VariableName)
      precWriterActivity := PW
    else
      precWriterActivity := EMPTY
    end if
  end if

  if actType = "process" then
    precWriterActivity := EMPTY
  end if

  return precWriterActivity

  end
```

ReaderDeepSearch

ReaderDeepSearch führt eine Tiefensuche in der übergebenen Aktivität durch, mit dem Ziel, eine auf *VariableName* schreibende Aktivität zu finden. Je nach Typ der Leseaktivität wird ihr Name oder der des umschließenden Elements zurückgegeben. Der Algorithmus wurde in Kapitel 5.4.4 im Abschnitt *Vorwärtssuche nach Leseaktivitäten (FindReadingActivitiesForward)* bereits näher erläutert.

Aufruf:

ActivityName ReaderDeepSearch(*ProcessIdentifier*, *ActivityName*, *VariableName*)

Rückgabewert / Parameter:

string *ActivityName*
string *ProcessIdentifier*
string *VariableName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

int counter
bool RA
bool localVariable
string actType
string RDS
string readingActName
string childElement
string[ ] childElements

actType := GetActivityType(*ProcessIdentifier*, *ActivityName*)

SWITCH actType {

case "flow" **do**
  childElements := alle /child-Elemente, die Activity sind
  counter := 0
  while (RDS = EMPTY AND counter < childElements.size) **do**
    RDS := ReaderDeepSearch(*ProcessIdentifier*, childElements[counter], *VariableName*)
    counter ++
  end while
  if RDS ≠ EMPTY **then**
    readingActName := *ActivityName*
  else
    readingActName := EMPTY
  end if

```
    break
  end case

  case "forEach" do
    childElement := /child-Element, das Activity ist
    RA := ReadingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
    RDS := ReaderDeepSearch(ProcessIdentifier, childElement, VariableName)
    if (RA = true OR RDS ≠ EMPTY) then
      readingActName := ActivityName
    else
      readingActName := EMPTY
    end if
    break
  end case

  case "if" do
    childElements := alle /child-Elemente in den Verzweigungen, die Activity sind
    RA := ReadingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
    counter := 0
    while (RDS = EMPTY AND counter < childElements.size) do
      RDS := ReaderDeepSearch(ProcessIdentifier, childElements[counter], VariableName)
      counter ++
    end while
    if (RA = true OR RDS ≠ EMPTY) then
      readingActName := ActivityName
    else
      readingActName := EMPTY
    end if
    break
  end case

  case "pick" do
    childElements := alle /child-Elemente in den onMessage bzw. onAlarm-Elementen, die
    Activity sind
    RA := ReadingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
    counter := 0
    while (RDS = EMPTY AND counter < childElements.size) do
      RDS := ReaderDeepSearch(ProcessIdentifier, childElements[counter], VariableName)
      counter ++
    end while
    if (RA = true OR RDS ≠ EMPTY) then
      readingActName := ActivityName
    else
      readingActName := EMPTY
    end if
    break
  end case

  case "repeatUntil" do
    childElement := /child-Element, das Activity ist
    RA := ReadingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
    RDS := ReaderDeepSearch(ProcessIdentifier, childElement, VariableName)
    if (RA = true OR RDS ≠ EMPTY) then
      readingActName := ActivityName
```

```
    else
      readingActName := EMPTY
    end if
    break
  end case

  case "sequence" do
    childElements := alle /child-Elemente, mit dem ersten beginnend
    counter := 0
    while (RDS = EMPTY AND counter < childElements.size) do
      RDS := ReaderDeepSearch(ProcessIdentifier, childElements[counter], VariableName)
      counter ++
    end while
    if RDS ≠ EMPTY then
      readingActName := childElements[counter]
    else
      readingActName := EMPTY
    end if
    break
  end case

  case "scope" do
    childElement := /child-Element, das Activity ist
    localVariable := prüfen, ob VariableName eine lokale Variable des Scopes ist
    RDS := ReaderDeepSearch(ProcessIdentifier, childElement, VariableName)
    if (localVariable = false AND RDS ≠ EMPTY) then
      readingActName := ActivityName
    else
      readingActName := EMPTY
    end if
    break
  end case

  case "while" do
    childElement := /child-Element, das Activity ist
    RA := ReadingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
    RDS := ReaderDeepSearch(ProcessIdentifier, childElement, VariableName)
    if (RA = true OR RDS ≠ EMPTY) then
      readingActName := ActivityName
    else
      readingActName := EMPTY
    end if
    break
  end case

  default
    RA := ReadingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
    if RA = true then
      readingActName := ActivityName
    else
      readingActName := EMPTY
    end if
  end default
```

return readingActName

end

ReadingActivity

Mit der Funktion *ReadingActivity* lässt sich überprüfen, ob die betrachtete Aktivität die angegebene Variable liest.

Aufruf:

bool ReadingActivity(*ProcessIdentifier*, *ActivityName*, *ActivityType*, *VariableName*)

Parameter:

string *ProcessIdentifier*
string *ActivityName*
string *ActivityType*
string *VariableName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
ActivityType ∈ BPEL-ActivityTypes[3]
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

bool RA

RA := false

SWITCH *ActivityType* {

case "assign" **do**
  if *ActivityName*/copy/from/variable-Attributwert = *VariableName* **then**
    RA := true
  end if
  break
end case

case "forEach" **do**
  if *ActivityName*/startCounterValue - Expression enthält *VariableName*
  OR *ActivityName*/finalCounterValue - Expression enthält *VariableName*
  OR *ActivityName*/completionCondition/branches - Expression enthält *VariableName*
  then
    RA := true
  end if
  break
end case

case "if" **do**

[3]Vereinfachte Schreibweise. Gemeint ist, dass alle BPEL-Aktivitätstypen zulässig sind.

```
    if ActivityName/condition - Expression enthält VariableName
    OR ActivityName/elseif/condition - Expression enthält VariableName
     then
      RA := true
    end if
    break
  end case

  case "invoke" do
    if ActivityName/inputVariable-Attributwert = VariableName then
      RA := true
    end if
    break
  end case

  case "pick" do
    if ActivityName/onAlarm/for - Expression enthält VariableName
    OR ActivityName/onAlarm/until - Expression enthält VariableName
     then
      RA := true
    end if
    break
  end case

  case "repeatUntil" do
    if ActivityName/condition - Expression enthält VariableName then
      RA := true
    end if
    break
  end case

  case "reply" do
    if ActivityName/variable-Attributwert = VariableName) then
      RA := true
    end if
    break
  end case

  case "throw" do
    if ActivityName/faultVariable-Attributwert = VariableName then
      RA := true
    end if
    break
  end case

  case "wait" do
    if ActivityName/for - Expression enthält VariableName
    OR ActivityName/until - Expression enthält VariableName
     then
      RA := true
    end if
    break
  end case

  case "validate" do
```

```
    if ActivityName/variables-Attributwert enthält VariableName then
      RA := true
    end if
    break
  end case

  case "while" do
    if ActivityName/condition - Expression enthält VariableName then
      RA := true
    end if
    break
  end case

  default
    RA := false
  end default

  return RA

  end
```

setActivityStatus

Setzt für die Instanz *InstanceIdentifier* den Ausführungszustand der Aktivität *ActivityName* auf *ActivityStatus*.

Aufruf:

setActivityStatus(*ActivityName*, *InstanceIdentifier*, *ActivityStatus*)

Parameter:

string *ActivityName*
string *InstanceIdentifier*
string *ActivityStatus*

Vorbedingungen:

ActivityName ∈ Instance.ProcessDefinition.Activities
InstanceIdentifier ∈ Instances
ActivityStatus ∈ Status

Semantik:

begin

Auf Zustandshaltungsebene:
*State(*ActivityName *in der Instanz* InstanceIdentfier*) :=* ActivityStatus

end

SLDataFlowValid

SLDataFlowValid stößt, abhängig von der vorzunehmenden Änderungsoperation und dem Datenzugriffstyp der betroffenen Aktivität, die jeweils nötige Form der Datenflussüberprüfung an. Hierzu ruft er in der jetzigen Ausbaustufe gegebenenfalls die Suchalgorithmen *FindWritingActivitiesBackward* und *FindReadingActivitiesForward* auf. Zurückgegeben wird die Menge der Variablen, die aufgrund fehlender sicherer Initialisierungen zu Datenfluss-Problemen führen. (Siehe hierzu auch Abbildung 5.15 und die zugehörige Erläuterung.)

Aufruf:

{CriticalVariableName} SLDataFlowValid(*ProcessIdentifier*, *ActivityName*, *DataAccessType*, *ModificationOperation*, {(*VariableName*, *VariableAccess*)})

Rückgabewert / Parameter:

string *CriticalVariableName*
string *ProcessIdentifier*
string *ActivityName*
string *DataAccessType*
string *ModificationOperation*
string *VariableName*
string *VariableAccess*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
DataAccessType ∈ {"readActivity", "writeActivity", "readWriteActivity", "noVariableAccessActivity"}
ModificationOperation ∈ {"SerialInsertionOfABasicActivity", "SerialRemovalOfABasicActivity"}
VariableName ∈ ProcessDefinition.Variables
VariableAccess ∈ {"read", "write"}

Semantik:

begin

string varName
string readingActName
string readingVarName
string writingActName
string writingVarName
string criticalVariableName

{criticalVariableName} := ∅

SWITCH *ModificationOperation*

case "SerialInsertionOfABasicActivity" **do**

if (DataAccessType = "readActivity" OR DataAccessType = "readWriteActivity") **then**

```
      for each t ∈ {(VariableName, VariableAccess)} do
        if t.VariableAccess = read then
          {varName} := {varName} ∪ {t.VariableName}
        end if
      end for
      {(writingActName, writingVarName)} := FindWritingActivitiesBackward(ProcessIdentifier,
      ActivityName, {varName})
      for each w ∈ {(writingActName, writingVarName)} do
        if w.writingActName = EMPTY then
          {criticalVariableName} := {criticalVariableName} ∪ w.writingVarName
        end if
      end for
    end if

    break
  end case

  case "SerialRemovalOfABasicActivity" do

    if (DataAccessType = "writeActivity" OR DataAccessType = "readWriteActivity") then
      for each t ∈ {(VariableName, VariableAccess)} do
        if t.VariableAccess = write then
          {varName} := {varName} ∪ {t.VariableName}
        end if
      end for

      {(readingActName, readingVarName)} := FindReadingActivitiesForward(ProcessIdentifier,
      ActivityName, {varName})
      {varName} := EMPTY
      for each r ∈ {(readingActName, readingVarName)} do
        if r.readingActName ≠ EMPTY then
          {varName} := {varName} ∪ {t.readingActName}
        end if
      end for

      if {varName} ≠ EMPTY then
        {(writingActName, writingVarName)} := FindWritingActivitiesBackward(Process-
        Identifier, ActivityName, {varName})
        for each w ∈ {(writingActName, writingVarName)} do
          if w.writingActName = EMPTY then
            {criticalVariableName} := {criticalVariableName} ∪ w.writingVarName
          end if
        end for
      end if

    end if
    break
  end case

  return {criticalVariableName}

  end
```

SucceedingReader

Führt eine Suche nach einer der Aktivität *ActivityName* nachfolgenden Leseaktivität auf die übergebene Variable durch. Das Verfahren wurde bereits in Kapitel 5.4.4, Abschnitt *Vorwärtssuche nach Leseaktivitäten (FindReadingActivitiesForward)* erläutert.
Der Algorithmus gibt den Namen der gefundenen Leseaktivität zurück, sofern eine solche existiert.

Aufruf:

ActivityName SucceedingReader(*ProcessIdentifier*, *ActivityName*, *VariableName*)

Rückgabewert / Parameter:

string *ActivityName*
string *ProcessIdentifier*
string *VariableName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

int counter
bool localVariable
string SR
string RDS
string actType
string parentActivity
string succReaderActivity
string[ ] siblings
string[ ] followingSiblings

parentActivity := /parent von *ActivityName*, davon das name-Attribut. (Falls /parent keine Activity und nicht *process* ist ist, sondern z.B. ein *elseif* oder *onAlarm*, dann ist dessen /parent zu nehmen.)
actType := GetActivityType(*ProcessIdentifier*, parentActivity)

if actType = "forEach" OR actType = "if" OR actType = "pick" OR actType = "repeatUntil" OR actType = "while" **then**
  SR := SucceedingReader(*ProcessIdentifier*, parentActivity, *VariableName*)
  if SR ≠ EMPTY **then**
    succReaderActivity := SR
  else
    succReaderActivity := EMPTY
  end if
end if

```
if actType = "flow" then
  siblings := alle /precedingSiblings und /followingSiblings von ActivityName
  counter := 0
  while RDS = EMPTY AND counter < siblings.size do
    RDS := ReaderDeepSearch(siblings[counter])
    counter++
  end while
  if RDS ≠ EMPTY then
    succReaderActivity := RDS
  else
    SR := SucceedingReader(ProcessIdentifier, parentActivity, VariableName)
    if SR ≠ EMPTY then
      succReaderActivity := SR
    else
      succReaderActivity := EMPTY
    end if
  end if
end if

if actType = "sequence" then
  followingSiblings := alle /followingSiblings von ActivityName, mit dem ersten beginnend
  counter := 0
  while RDS = EMPTY AND counter < followingSiblings.size do
    RDS := ReaderDeepSearch(followingSiblings[counter])
    counter++
  end while
  if RDS ≠ EMPTY then
    succReaderActivity := RDS
  else
    SR := SucceedingReader(ProcessIdentifier, parentActivity, VariableName)
    if SR ≠ EMPTY then
      succReaderActivity := SR
    else
      succReaderActivity := EMPTY
    end if
  end if
end if

if actType = "scope" then
  localVariable := prüfen, ob VariableName eine lokale Variable des Scopes ist
  if localVariable = false then
    SR := SucceedingReader(ProcessIdentifier, parentActivity, VariableName)
    if SR ≠ EMPTY then
      succReaderActivity := SR
    else
      succReaderActivity := EMPTY
    end if
  else
    succReaderActivity := EMPTY
  end if
end if

if actType = "process" then
  succReaderActivity := EMPTY
```

end if

return succReaderActivity

end

UserDecisionPerformMigration

Fragt den Benutzer, ob er die Migration durchführen möchte.

Aufruf:

bool UserDecisionPerformMigration()

Semantik:

begin

bool performMigration

performMigration := "Migration durchführen? (Ja/Nein)"

return performMigration

end

UserInformation

Informiert den Benutzer für die angegebene Instanz über das jeweilige Ergebnis der Schema- und der Instanzprüfung bzgl. Datenflusskorrektheit.

Aufruf:

UserInformation(*InstanceIdentifier*, *SLDataFlowValid*, *ILDataFlowValid*)

Parameter:

string *InstanceIdentifier*
string[ ]: *SLDataFlowValid*
bool *ILDataFlowValid*

Vorbedingungen:

InstanceIdentifier ∈ Instances

Semantik:

begin

if (SLDataFlowValid = EMPTY && ILDataFlowValid = true) **then**
„Der Prozess ist hinsichtlich des Datenflusses auf Schema- und auf Instanzebene korrekt.“
end if

if (SLDataFlowValid = EMPTY && ILDataFlowValidd = false) **then**
„Der Prozess ist auf Schemaebene korrekt modelliert, auf Instanzebene kommt es aber zu Datenflussproblemen bei der Instanz *InstanceIdentifier*.“
end if

if (SLDataFlowValid ≠ EMPTY && ILDataFlowValid = true) **then**
„Es liegt ein generelles Datenflussmodellierungsproblem auf Prozessschema-Ebene vor. Die Instanz *InstanceIdentifier* kann aber trotzdem korrekt migriert werden.“
end if

if (SLDataFlowValid ≠ EMPTY && ILDataFlowValid = false) **then**
„Es liegt ein generelles Datenflussmodellierungsproblem auf Prozessschema-Ebene vor. Dieses Datenflussproblem betrifft auch die Instanz *InstanceIdentifier*.“
end if

end

WriterDeepSearch

Der Algorithmus führt eine Tiefensuche in der übergebenen Aktivität durch. Hierbei wird nach einem sicheren Schreibzugriff auf die angegebene Variable gesucht. Zurückgegeben wird, je nach Aktivitätstyp, der Name der Schreibaktivität bzw. der des umschließenden Elements. (Siehe hierzu Kapitel 5.4.4, Abschnitt *Rückwärtssuche nach Schreibaktivitäten (FindWritingActivitiesBackward)*, für eine ausführliche Erläuterung.)

Aufruf:

ActivityName WriterDeepSearch(*ProcessIdentifier*, *ActivityName*, *VariableName*)

Rückgabewert / Parameter:

string *ActivityName*
string *ProcessIdentifier*
string *VariableName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
VariableName ∈ ProcessDefinition.Variables

Semantik:

begin

int counter
bool WA
bool localVariable
string actType
string writingActName
string WDS
string WDSOM
string WDSOA
string childElement
string[ ] childElements
string[ ] childElementsOM
string[ ] childElementsOA

actType := GetActivityType(*ProcessIdentifier*, *ActivityName*)

SWITCH actType {

case "flow" **do**
  childElements := alle /childs, die Activity sind
  counter := 0
  while (WDS = EMPTY AND counter < childElements.size) **do**
    WDS := WriterDeepSearch(*ProcessIdentifier*, childElements[counter], *VariableName*)
    counter ++
  end while
  if WDS ≠ EMPTY **then**

```
    writingActName := ActivityName
  else
    writingActName := EMPTY
  end if
  break
end case

case "forEach" do
  childElement := /child-Element, das Activity ist
  WDS := WriterDeepSearch(ProcessIdentifier, childElement, VariableName)
  if WDS ≠ EMPTY then
    writingActName := ActivityName
  else
    writingActName := EMPTY
  end if
  break
end case

case "if" do
  childElements := alle /childs der Bedingungen
  counter := 0
  repeat
    WDS := WriterDeepSearch(ProcessIdentifier, childElements[counter], VariableName)
    counter ++
  until (WDS = EMPTY OR counter = childElements.size)
  if WDS ≠ EMPTY then
    writingActName := ActivityName
  else
    writingActName := EMPTY
  end if
  break
end case

case "pick" do
  childElementsOM := alle /childs, die onMessage sind. Deren /childs, die Activity sind.
  childElementsOA := alle /childs, die onAlarm sind. Deren /childs, die Activity sind.
  WA := WritingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
  counter := 0
  repeat
    WDSOM := WriterDeepSearch(ProcessIdentifier, childElementsOM[counter], Variable-
    Name)
    counter ++
  until (WDSOM = EMPTY OR counter = childElementsOM.size)
  counter := 0
  repeat
    WDSOA := WriterDeepSearch(ProcessIdentifier, childElementsOA[counter], VariableNa-
    me)
    counter ++
  until (WDSOA = EMPTY OR counter = childElementsOA.size)
  if (WA = true OR WDSOM ≠ EMPTY) AND WDSOA ≠ EMPTY then
    writingActName := ActivityName
  else
    writingActName := EMPTY
  end if
```

```
    break
  end case

  case "repeatUntil" do
    childElement := /child-Element, das Activity ist
    WDS := WriterDeepSearch(ProcessIdentifier, childElement, VariableName)
    if WDS ≠ EMPTY then
      writingActName := ActivityName
    else
      writingActName := EMPTY
    end if
    break
  end case

  case "sequence" do
    childElements := alle /childs von ActivityName, mit dem letzten beginnend
    counter := 0
    while (WDS = EMPTY AND counter < childElements.size) do
      WDS := WriterDeepSearch(ProcessIdentifier, childElements[counter], VariableName)
      counter ++
    end while
    if WDS ≠ EMPTY then
      writingActName := childElements[counter]
    else
      writingActName := EMPTY
    end if
    break
  end case

  case "scope" do
    childElement := /child-Element, das Activity ist
    localVariable := prüfen, ob VariableName eine lokale Variable des Scopes ist
    WDS := WriterDeepSearch(ProcessIdentifier, childElement, VariableName)
    if (localVariable = false AND WDS ≠ EMPTY) then
      writingActName := ActivityName
    else
      writingActName := EMPTY
    end if
    break
  end case

  case "while" do
    childElement := /child-Element, das Activity ist
    WDS := WriterDeepSearch(ProcessIdentifier, childElement, VariableName)
    if WDS ≠ EMPTY then
      writingActName := ActivityName
    else
      writingActName := EMPTY
    end if
    break
  end case

  default
    WA := WritingActivity(ProcessIdentifier, ActivityName, actType, VariableName)
```

```
    if WA = true then
      writingActName := ActivityName
    else
      writingActName := EMPTY
    end if
  end default

  return writingActName

  end
```

WritingActivity

WritingActivity stellt fest, ob in der Aktivität ein Schreibzugriff auf *VariableName* stattfindet.

Aufruf:

bool WritingActivity(*ProcessIdentifier*, *ActivityName*, *ActivityType*, *VariableName*)

Parameter:

string *ProcessIdentifier*
string *ActivityName*
string *ActivityType*
string *VariableName*

Vorbedingungen:

ProcessIdentifier ∈ ProcessDefinitions
ActivityName ∈ ProcessDefinition.Activities
ActivityType ∈ BPEL-ActivityTypes
VariableName ∈ ProcessDefinition.Variables

Semantik:

```
begin

bool WA
string[ ] VN

WA := false

SWITCH ActivityType {

case "assign" do
  for each ActivityName/copy/to-Element do
    if variable-Attributwert = VariableName then
      WA := true
      break
    end if
  end for
  break
end case

case "invoke" do
  if ActivityName/outputVariable-Attributwert = VariableName then
    WA := true
  end if
  break
end case

case "pick" do
  for each ActivityName/onMessage-Element do
    if variable-Attributwert = VariableName then
```

```
        WA := true
      else
        WA := false
        break
      end if
    end for
    break
end case

case "receive" do
    if ActivityName/variable-Attributwert = VariableName then
      WA := true
    end if
    break
end case

default
    WA := false
end default

return  WA

end
```

Literaturverzeichnis

[AAD+07] Ashish Agrawal, Mike Amend, Manoj Das, Mark Ford, Chris Keller, Matthias Kloppmann, Dieter König, Frank Leymann, Ralf Müller, Gerhard Pfau, Karsten Plösser, Ravi Rangaswamy, Alan Rickayzen, Michael Rowley, Patrick Schmidt, Ivana Trickovic, Alex Yiu und Matthias Zeller: *WS-BPEL Extension for People (BPEL4People), Version 1.0*, Juni 2007. `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf` (31. März 2009).

[AHEA06] M. Adams, A.H.M. ter Hofstede, D. Edmond und W.M. P. van der Aalst (Herausgeber): *Implementing Dynamic Flexibility in Workflows using Worklets*, Band BPM-06-06, 2006.

[Apaa] Apache Software Foundation: *Apache Ode - Developer Guide: Architectural Overview.* `http://ode.apache.org/architectural-overview.html` (31. März 2009).

[Apab] Apache Software Foundation: *Apache Ode - Developer Guide: Jacob.* `http://ode.apache.org/jacob.html` (31. März 2009).

[Apac] Apache Software Foundation: *Apache Ode - WS-BPEL 2.0 Specification Compliance.* `http://ode.apache.org/ws-bpel-20-specification-compliance.html` (31. März 2009).

[Bre08] Sebastian Breier: *Extended Data-flow Analysis on BPEL Processes.* Diplomarbeit, Universität Stuttgart, 2008. `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2726&engl=0` (31. März 2009).

[Fis04] Layna Fischer: *The Workflow Handbook 2004: Published in association with the Workflow Management Coalition (WfMC).* Future Strategies Inc, Lighthouse Point FL, 2004.

[GÖ8] Katharina Görlach: *Ein Verfahren zur abstrakten Interpretation von XPath-Ausdrücken in WS-BPEL-Prozessen.* Diplomarbeit, Humboldt-Universität zu Berlin, 2008. `http://www.informatik.hu-berlin.de/top/download/publications/Goerlach2008_da.pdf` (31. März 2009).

[Gos04] D. Goswami: *A parallel algorithm for static slicing of concurrent programs.* Concurrency and computation, (Vol. 16):751–770, 2004. `http://www3.interscience.wiley.com/cgi-bin/fulltext/107631053/PDFSTART` (31. März 2009).

[Hav05] Michael Havey: *Essential business process modeling.* O'Reilly, Sebastopol Calif. u.a., 1. Auflage, 2005.

[Hei03] Thomas Heidinger: *Statische Analyse von BPEL4WS-Prozessmodellen.* Studienarbeit, Humboldt-Universität zu Berlin, 2003. `http://www.informatik.hu-berlin.de/top/download/publications/Heidinger2003_sa.pdf` (31. März 2009).

[Int07] Intalio: *Intalio Company and Platform Overview*, 2007. http://bpms.intalio.com/images/stories/start/intalio_overview.pdf (31. März 2009).

[Kha08] Rania Khalaf: *Supporting business process fragmentation while maintaining operational semantics: A BPEL perspective.* Dissertation, Universität Stuttgart, 2008. http://elib.uni-stuttgart.de/opus/volltexte/2008/3514/pdf/Khalaf_Rania_Diss.pdf (31. März 2009).

[KKL+05] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus Riegen, Patrick Schmidt und Ivana Trickovic: *WS-BPEL Extension for Sub-processes – BPEL-SPE*, 2005. http://www.ibm.com/developerworks/library/specification/ws-bpelsubproc (31. März 2009).

[KKL07] Oliver Kopp, Rania Khalaf und Frank Leymann: *Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes.* Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2007. http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2007-04&engl=0 (31. März 2009).

[KKL08] Oliver Kopp, Rania Khalaf und Frank Leymann: *Deriving Explicit Data Links in WS-BPEL Processes.* IEEE International Conference on Services Computing 2008, Seiten 367–376, 2008.

[MBRS] Jutta Mülle, Klemens Böhm, Nicolas Röper und Tobias Sünder: *Building Conference Proceedings Requires Adaptable Workflow and Content Management.* In: *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB2006).* http://dbis.ipd.uni-karlsruhe.de/download/MuelleBoehm-VLDB2006.pdf (31. März 2009).

[Mel07] Ingo Melzer: *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis.* Spektrum Akademischer Verlag, Heidelberg, München, 2. Auflage, 2007.

[MMG+07] Simon Moser, Axel Martens, Katharina Gorlach, Wolfram Amme und Artur Godlinski: *Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis.* IEEE International Conference on Services Computing (SCC 2007), Seiten 98–105, 2007.

[OAS07] OASIS: *Web Services Business Process Execution Language Version 2.0*, 11. April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf (31. März 2009).

[Obe96] Andreas Oberweis: *Modellierung und Ausführung von Workflows mit Petri-Netzen.* Teubner-Reihe Wirtschaftsinformatik. Teubner, Stuttgart u.a., 1. Auflage, 1996.

[OMG08] Object Management Group OMG: *Business Process Modelling Notation, V1.1*, Januar 2008. http://www.bpmn.org/Documents/BPMN%201-1%20Specification.pdf (31. März 2009).

[Rei00] Manfred Reichert: *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* Dissertation, Universität Ulm, 2000.

[RHS04] Cornelia von Richter-Hagen und Wolffried Stucky: *Business-Process- und Workflow-Management: Prozessverbesserung durch Prozess-Management.* Teubner, Stuttgart u.a., 1. Auflage, 2004.

[RM06] Jan Recker und Jan Mendling: *On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages.* In: T. Latour und M. Petit (Herausgeber): *CAiSE 2006 Workshop Proceedings - Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2006)*, Seiten 521–532, Luxembourg, 2006. `http://wi.wu-wien.ac.at/home/mendling/publications/06-EMMSAD.pdf` (31. März 2009).

[STH07] Gernot Starke und Stefan Tilkov (Hrsg.): *SOA-Expertenwissen: Methoden Konzepte und Praxis serviceorientierter Architekturen.* dpunkt-Verlag, Heidelberg, 1. Auflage, 2007.

[Ten08] Elena Tentyukova: *Entwicklung einer Basisausführungsmaschine für änderbare BPEL-Prozesse.* Abschlussarbeit, Universität Karlsruhe, 2008.

[VF07] Javier Vázquez Fernandez: *BPEL with explicit data flow: Model editor and partitioning tool.* Diplomarbeit, Universität Stuttgart, 2007. `http://elib.uni-stuttgart.de/opus/volltexte/2007/3228/pdf/DIP_2616.pdf` (31. März 2009).

[Wei06] Dominik Weingardt: *Verwaltung von WS-BPEL-Spezifikationen in einer relationalen Datenbasis.* Studienarbeit, Universität Karlsruhe, 2006.

[Wei08] Dominik Weingardt: *Erweiterung eines Workflowmanagementsystems um Änderbarkeit in Folge von Daten-Workflow-Interaktionen.* Diplomarbeit, Universität Karlsruhe, 31. März 2008.

[Wor95] Workflow Management Coalition (WfMC): *The Workflow Reference Model, TC00-1003, Issue 1.1.* 1995. `http://wfmc.org/index.php?option=com_docman&task=doc_download&gid=92&Itemid=72` (31. März 2009).

[Wor99] Workflow Management Coalition (WfMC): *Terminology and Glossary English, TC-1011, Issue 3.* 1999. `http://wfmc.org/index.php?option=com_docman&task=doc_download&gid=93&Itemid=74` (31. März 2009).