ulm university universität
uulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Fortschrittliche Datenflusskonzepte für flexible Prozessmodelle

Diplomarbeit

von
Alexander Forschner
Juni 2009

1. Gutachter:	Prof. Dr. Manfred Reichert
2. Gutachter:	PD Dr. Stefanie Rinderle-Ma
Betreuer:	Dipl.-Inf. (U) Ulrich Kreher

Kurzfassung

Prozess-Management-Systeme haben in den letzten Jahren immer mehr an Bedeutung gewonnen. Das im *AristaFlow*-Projekt entwickelte ADEPT2 übertrifft die meisten der heute am Markt verfügbaren Systeme hinsichtlich Flexibilität und Prozesskorrektheit [DAG+06]. In den mehr als zehn Jahren an Forschung und Entwicklung hat sich gezeigt, dass nahezu alle praxisrelevanten Prozesse mit Hilfe des Metamodells von ADEPT2 abgebildet werden können. Sowohl auf Konsistenz, als auch auf Robustheit muss selbst bei umfangreichen und langlaufenden Geschäftsprozessen nicht verzichtet werden.

In Geschäftsprozessen sind neben der Abfolge der einzelnen Arbeitsschritte, dem *Kontrollfluss*, auch die zwischen den Schritten ausgetauschten Daten von Bedeutung. Dabei kann es sich um einfache Formulardaten, aber auch um komplexe XML-Daten und Dokumente handeln. Bereits heute sind Prozessschritte in ADEPT2 in der Lage, beliebige Daten auszutauschen.

Die Darstellung dieser Daten ist jedoch in vielen Fällen nicht optimal. So müssen die Werte aus Formularfeldern einzeln zwischen den Prozessschritten ausgetauscht werden. Auch auf die Metadaten von Dokumenten (wie Dateigröße oder Dokumententyp) kann nur zugegriffen werden, wenn das Dokument bereits interpretiert wurde.

Damit ADEPT2-Prozesse in Zukunft übersichtlicher erstellt werden können, stellen wir in dieser Arbeit erweiterte Datenflusskonzepte vor. Wir diskutieren, welche Vorteile die einzelnen Konzepte bieten und wie diese unter Berücksichtigung der geforderten Prozesskorrektheit realisiert werden können.

Bei den vorgestellten neuen Datentypen handelt es sich um *strukturierte* Datentypen, um *Listen*, sowie um *Aufzählungen*. Diese drei Datentypen haben sich in der Praxis als nützlich erwiesen, da sich mit diesen und mit deren Kombination die meisten im Alltag auftretenden Probleme lösen lassen. Dabei dienen strukturierte Datentypen der Gruppierung von zusammengehörigen Daten zu einem neuen Datentyp. Listen versetzen den Prozessmodellierer in die Lage, Sachverhalte abzubilden, bei denen erst zur Prozesslaufzeit feststeht, wie viele Werte eines bestimmten Typs zusammengehörend gespeichert werden müssen. Dies ist möglich, da Listen eine variable Länge besitzen. Aufzählungs-Datentypen bieten die Möglichkeit, Einschränkungen für gewisse Szenarien festzulegen, in denen bspw. Verzweigungsentscheidungen anhand einer vorgegebenen, endlichen Menge getroffen werden sollen.

Weiter beschreiben wir die Auswirkungen auf den Datenfluss, die sich ergeben, wenn Daten im Laufe eines Prozesses ungültig werden. Dies geschieht zumeist durch *konsumierende* Lesezugriffe.

Anwendungen können nur dann auf Daten zugreifen, wenn ihnen die Struktur der Daten bekannt ist. Um den Aufwand bei der Anwendungsentwicklung zu verringern, stellen wir das Konzept *benutzerdefinierter Funktionen* vor. Diese dienen als Konverter zwischen Anwendungsbausteinen und benutzerdefinierten Datentypen.

Abschließend vergleichen wir die vorgestellten Konzepte mit derzeit am Markt verfügbaren Systemen, sowie mit anderen Beschreibungssprachen für Geschäftsprozesse.

Inhaltsverzeichnis

1. Einleitung

Die industrielle Revolution zog eine massive Optimierung von Produktions- und Distributionsprozessen nach sich (s. Abb. 1.1). In unserer globalisierten Welt reicht diese Effizienzsteigerung für ein Unternehmen alleine nicht mehr aus, um am Markt bestehen zu können. Die Folge ist eine Industrialisierung von innerbetrieblichen und unternehmensübergreifenden Arbeitsabläufen, die sowohl materielle als auch immaterielle Prozesse umfasst.

Die Informatik hat in den letzten Jahren und Jahrzehnten Technologien entwickelt, die den weltweiten Austausch von Informationen massiv beschleunigt haben. Es liegt nahe, diese Errungenschaften zu nutzen, um die immateriellen Geschäftsprozesse weiter zu optimieren.

1.1. Motivation

Unternehmen nutzen seit einigen Jahren EDV[1], um die Aufgabe zu bewältigen, unternehmensweite Datenbestände benutzerfreundlich aufzubereiten. Für die Verwaltung von Aufträgen, Lieferungen, Lagerhaltung, sowie des Personalwesens (HRM[2]) stehen ERP[3]-Systeme zur Verfügung. Jedoch bieten diese statischen Informationssysteme lediglich verschiedene Sichten auf Unternehmensdaten, ohne die Mitarbeiter aktiv bei der Bewältigung ihrer täglichen Aufgaben zu unterstützen [BDS+93].

Die Arbeitsabläufe sind dabei bestenfalls schriftlich dokumentiert, meist jedoch ausschließlich als Humankapital [Jae04] in den Köpfen der Angestellten oder hart verdrahtet in den Softwaresystemen verankert [DR04], was sowohl die flexible Anpassung an

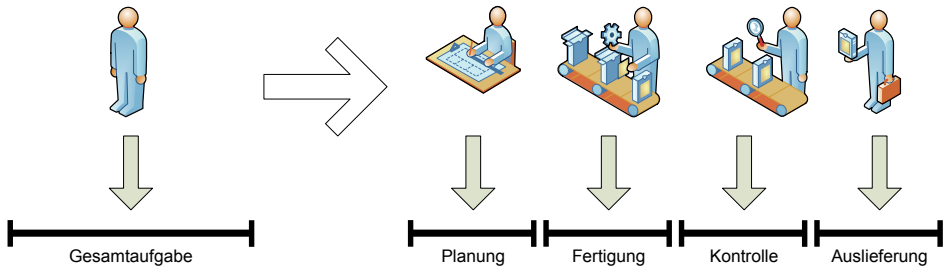


Abbildung 1.1 Arbeitsteilung eines materiellen Prozesses (schematisch)

1 Elektronische Datenverarbeitung
2 Human Resource Management
3 Enterprise Resource Planning

aktuelle Marktsituationen als auch die optimale Nutzung vorhandener Mittel verhindert. Abhilfe schaffen *prozessorientierte* Informationssysteme [DA05], die in der Lage sind, Geschäftsprozesse abzubilden, auszuführen und ihren Ablauf zu dokumentieren.

1.1.1. Management von Geschäftsprozessen

Prozess-Management-Systeme unterstützen Unternehmen bei der Aufnahme bestehender Prozesse, bei der Formulierung gewünschter Prozesseigenschaften und bei der Optimierung der aufgenommenen Prozesse, bis diese allen gestellten Anforderungen gerecht werden. Dabei reicht die Palette von einfachen Werkzeugen zur Prozessvisualisierung, über Simulationssoftware, bis hin zu umfangreichen Systemen zur prozessbezogenen Integration von Anwendungen.

Ausgereifte Prozess-Management-Systeme haben den Zweck, die richtigen Informationen der richtigen Person zur richtigen Zeit am richtigen Ort zur Verfügung zu stellen. Eine strikte Trennung von Prozess- und Anwendungslogik ermöglicht höchste Flexibilität sowohl bei der Gestaltung der Prozesse als auch bei deren Wartung.

Da Geschäftsprozesse keine starren Gebilde darstellen, sondern vielmehr einer ständigen Weiterentwicklung unterliegen, unterstützen moderne PMS[4] die Unternehmen bei der Evolution von Prozessen über Jahre oder Jahrzehnte hinweg. Dies geht so weit, dass neben den Prozessvorlagen auch bereits in Ausführung befindliche *Prozessinstanzen* an sich ändernde Anforderungen angepasst werden können [RD03].

Mit ihrer zunehmenden Verbreitung nehmen PMS eine immer zentralere Rolle in immer mehr Unternehmen ein. Umso erstaunlicher ist es, dass diese Systeme häufig lediglich als Vermittler von Daten zwischen verschiedenen, unabhängigen Informationssystemen angesehen und eingesetzt werden, wobei die Kontrolle über die unterschiedlichen Datenbestände weiterhin dem jeweiligen System obliegt.

1.1.2. Integration von Anwendungen und Daten

Bei einer konsequenten Verfolgung des prozessorientierten Gedankens gestaltet sich schnell ein Bild, in dem PMS nicht nur die Steuerung der angebundenen Informationssysteme vornehmen, sondern diese regelrecht integrieren. Die Ablaufreihenfolge der einzelnen Anwendungen wird dabei durch das Verschalten von Anwendungsbausteinen über den *Kontrollfluss* der Prozesse festgelegt.

Die integrierten Anwendungen können dabei weiterhin über eigene Datenbestände verfügen. Stimmig wird das Gesamtkonzept jedoch erst, wenn der Austausch von Daten der Anwendungen untereinander ebenfalls der Kontrolle des PMS unterliegt. Wie auch der Kontrollfluss ist dieser *Datenfluss* integraler Bestandteil moderner Prozessbeschreibungssprachen. In Abbildung 1.2 sind vier Anwendungsbausteine (A, B, C und D) dargestellt, wobei ein Datenfluss von A zu B und D stattfindet.

Es ist offensichtlich, dass eine derart gravierende Umstellung nicht von heute auf morgen zu bewerkstelligen ist. Dennoch müssen heutige Systeme bereits in der Lage

[4] Prozess-Management-System

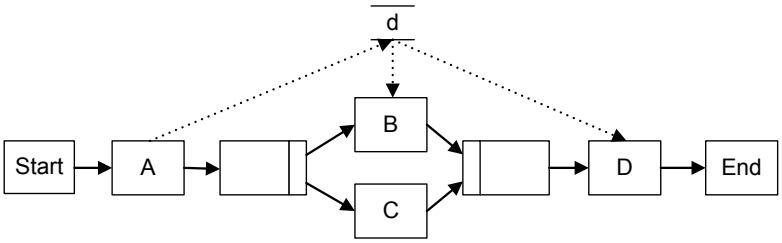


Abbildung 1.2 Kontroll- und Datenfluss eines einfachen Geschäftsprozesses

sein, die auszutauschenden Daten in ihrer gesamten Vielfalt abzubilden und zu verarbeiten. Dabei muss prozessseitig gewährleistet werden, dass die zu verarbeitenden Daten immer dann zur Verfügung stehen, wenn sie benötigt werden, was dazu führt, dass PMS neben dem Zugriff auf Daten, auch Informationen über diese benötigen.

Nur wenn alle zu den Daten gehörigen Informationen systemseitig zur Verfügung stehen, kann ein reibungsloser Prozessablauf gewährleistet werden, was für ein System der Tragweite von PMS unerlässlich ist. Zu diesen Informationen zählt neben der Struktur der Daten auch die Häufigkeit, mit der diese im Verlauf eines Prozesses auftreten, sowie Zusicherungen über die zuverlässige Verfügbarkeit der Daten zu bestimmten Zeitpunkten im Prozessverlauf.

1.2. Aufgabenstellung

Das auf ADEPT [Rei00] basierende ADEPT2-Metamodell [DAG+06, GJA+07] unterstützt derzeit lediglich einfache Datentypen, sowie benutzerdefinierte Datentypen, die systemseitig nicht interpretiert werden können (vgl. BLOB[5] in Datenbanken). Bei der Forschung im Rahmen des *AristaFlow*-Projektes[6] hat sich jedoch gezeigt, dass die Anforderungen an Daten und deren Datentypen in PMS weit über die bisher verfügbaren Konzepte hinausgehen.

Ziel dieser Arbeit ist es, erweiterte Konzepte für den Umgang mit Daten in ADEPT2 zu entwerfen und diese in die bestehende Systemarchitektur einzubinden. Dabei muss insbesondere die durch das Prozess-Metamodell gewährleistete Korrektheit der Prozesse erhalten bleiben, wobei eine erweiterte Flexibilität bei gleichzeitig konstanter Verlässlichkeit erstrebenswert ist.

Eine naheliegende Erweiterung des bestehenden Datenmodells ist die Unterstützung von *strukturierten Datentypen*. Diese ermöglichen es, zusammengehörige Daten auch dem System gegenüber als zusammengehörig zu kennzeichnen, ohne dabei den Zugriff auf einzelne Elemente der Struktur zur verlieren.

Neben Daten mit statischer Struktur sind auch Listen ein Konzept mit großer praktischer Relevanz. Dabei handelt es sich um variable Speicherstrukturen. Variabel bezieht

[5] Binary Large Object
[6] www.aristaflow.de

sich dabei auf die Anzahl der in einer Liste gespeicherten Elemente, wohingegen die Struktur jedes einzelnen Elements statisch festgelegt ist.

Über die reine Speichermöglichkeit für beliebig viele Daten gleichen Typs hinaus unterstützen Listen auch Kontrollflusskonstrukte zur parallelen oder sequentiellen Verarbeitung von Datensätzen, deren Anzahl erst zur Prozesslaufzeit ermittelt werden kann, weswegen der Kontrollfluss ebenfalls zur Laufzeit dynamisch auf die Anzahl der vorhandenen Datensätze reagieren muss. Dabei spielt sowohl der Zugriff auf eine Liste als Ganzes eine Rolle als auch der Zugriff auf einzelne Listenelemente.

Eine deutliche Steigerung der Benutzerfreundlichkeit von bedingten Verzweigungen lässt sich erreichen, indem der Wertebereich des Datums, welches für die Verzweigungsentscheidung verwendet wird, im Voraus eingeschränkt wird. Diese Einschränkung auf eine feste Anzahl von Werten bieten *Aufzählungsdatentypen*. Die Informationen über die zur Laufzeit möglichen Werte eines Datums können für eine anwenderfreundlichere Modellierung bedingter Verzweigungen eingesetzt werden.

Benutzerdefinierte Datentypen (UDT[7]) finden dort Anwendung, wo die Struktur von Daten zu komplex ist, um sie systemseitig vollständig abzubilden (vgl. BLOB). Häufig sollen diese Daten jedoch von Anwendungen verarbeitet werden, denen die Struktur der Daten unbekannt ist. *Benutzerdefinierte Funktionen* (UDF[8]) treten diesem Problem entgegen, indem sie einen UDT (oder Teile davon) beim Zugriff in eine mit der Anwendung kompatible Form konvertieren.

Bei *konsumierenden Lesezugriffen* handelt es sich um ein Konzept, das nicht die Struktur der Daten, sondern vielmehr deren Verfügbarkeit in Prozessen betrifft. Die Auswirkungen ungültiger Daten auf die Korrektheit des Datenflusses erfordern neue Analyseverfahren.

Diese Fülle neuer Datenkonstrukte macht eine systemweite Verwaltung von Datentypen und datennahen Konzepten unumgänglich. Die Diskussion der Besonderheiten jedes vorgestellten Konzeptes bezüglich dieser Verwaltung ist Bestandteil dieser Arbeit.

Eine Übersicht über die Datenflusskonzepte derzeit am Markt verfügbarer Systeme runden die Arbeit ab.

1.3. Aufbau der Arbeit

In Kapitel 2 geben wir zunächst eine Übersicht über die Grundlagen, die zum Verständnis der Arbeit notwendig sind. Danach beginnen wir in Kapitel 3 mit dem Thema des konsumierenden Lesens und dessen Auswirkungen auf bestehende Datenflusskonzepte. Im Anschluss zeigen wir in Kapitel 4, wie strukturierte Datentypen in ADEPT2 Verwendung finden und wie diese den Umgang mit Daten in einem PMS vereinfachen, woraufhin wir uns in Kapitel 5 mit dem Thema der benutzerdefinierten Funktionen befassen. Listen und Anwendung werden in Kapitel 6 beschrieben, wonach mit Aufzählungen in Kapitel 7 der konzeptionelle Teil dieser Arbeit abgeschlossen ist.

---

7 User Defined Type
8 User Defined Function

Dabei verwenden wir in jedem Kapitel dasselbe Schema: Zunächst erörtern wir das jeweilige Thema genauer und klären, welche Probleme es zu lösen gibt. Anschließend beschreiben wir eine oder mehrere Varianten, wie das jeweilige Konzept im Rahmen von ADEPT2 umgesetzt werden kann. Werden verschiedene Möglichkeiten zur Umsetzung diskutiert, enthält der Abschnitt eine begründete Entscheidung für die beste Umsetzungsvariante. Darauf folgt jeweils eine Erläuterung, wie sich die vorgestellte Realisierung auf das bestehende Datenflussmodell und die Datenflussanalyse auswirkt. Für Datentypen und benutzerdefinierte Funktionen geben wir die theoretischen Grundlagen für ein zukünftig zu implementierendes Verwaltungswerkzeug. Falls notwendig ergänzen wir die Themen um weitere Aspekte, die sich nicht in eine einheitliche Struktur fassen lassen, bevor wir das Kapitel mit einer Zusammenfassung der vorgestellten Ergebnisse abschließen.

Zuletzt widmen wir uns in Kapitel 8 verwandten Arbeiten, Systemen und Konzepten, bevor wir die Arbeit mit einer Zusammenfassung, sowie einem Ausblick auf weitere Entwicklungsmöglichkeiten abschließen (Kapitel 9).

2. Grundlagen

Wir beschreiben zunächst die Grundlagen, die im Wesentlichen für das Verständnis dieser Arbeit von Bedeutung sind. Essentiell für dieses Verständnis ist das ADEPT2-Metamodell, eine blockstrukturierte Beschreibungs- und Ausführungssprache für Geschäftsprozesse. Da es sich hierbei um ein *graphenbasiertes* Metamodell handelt, beginnen wir mit einer kurzen Einführung zu diesem Thema.

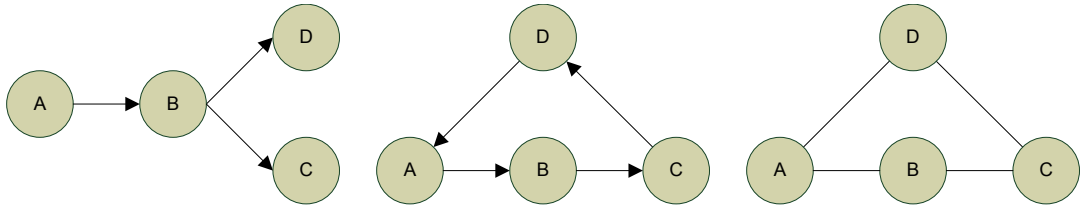
Darüber hinaus werden verschiedene Datentypen und einige Aspekte der Softwaretechnik, insbesondere der objektorientierten Programmierung beschrieben. Dabei sind vor allem die Konzepte der *Vererbung* und der *Schachtelung* für diese Arbeit relevant.

2.1. Graphenbasierte Prozessbeschreibungssprachen

In heutigen PMS werden zumeist *Graphen* verwendet, um Geschäftsprozesse abzubilden. Dabei handelt es sich um eine Menge von *Knoten*, die durch Linien verbunden sind. Diese Linien werden *Kanten* genannt. Sie beginnen bei ihrem *Startknoten* und münden in ihren *Zielknoten*. Im Gegensatz zu ungerichteten Graphen, beschreibt eine Kante bei *gerichteten* Graphen (Abb. 2.1a und 2.1b) eine unidirektionale Beziehung zwischen Start- und Zielknoten. So kann der logischen Ablauf eines Geschäftsprozesses dargestellt werden, was mit ungerichteten Kanten (Abb. 2.1c) nicht möglich ist.

Der Startknoten einer gerichteten Kante wird bezüglich des Zielknotens als *Vorgänger* bezeichnet. Umgekehrt ist der Zielknoten der *Nachfolger* des Startknotens. In Abbildung 2.1a ist A der Vorgänger von B, C und D sind dessen Nachfolger. Die transitive Anwendung dieser Beziehung ergibt die *Nachfolger-* bzw. *Vorgängermenge* eines Knotens. Knoten A in Abbildung 2.1a besitzt bspw. die Nachfolgermenge {B,C,D}. Ist ein Knoten in seiner eigenen Nachfolgermenge enthalten, so sprechen wir von einem *Zyklus* (Abb. 2.1b).

Bei der Darstellung von Geschäftsprozessen mit graphenbasierten Beschreibungssprachen repräsentiert jeder Knoten einen Prozessschritt, die Kanten stellen logische Beziehung zwischen den einzelnen Schritten dar. Die systemseitige Beschreibung eines Prozessschrittes erfolgt in einer sog. *Aktivität*. In einer Aktivität sind dabei alle zum Prozessschritt gehörigen Informationen (Bearbeiterzuordnung, Priorität, auszuführende Anwendung, etc.) hinterlegt. Weiter ist in jeder Aktivität festgelegt, wie der Austausch von Daten zwischen Prozess und auszuführender Anwendung vonstattengeht.



(a) gerichtet **(b)** gerichtet mit Zyklus **(c)** ungerichtet mit Zyklus

Abbildung 2.1 Verschiedene Beispiele für Graphen

2.2. ADEPT2-Metamodell

Bei ADEPT2 handelt es sich um eine auf ADEPT [Rei00, RRD04] basierende Beschreibungssprache für Geschäftsprozesse in der symmetrische Blockstrukturen zum Einsatz kommen. Das Metamodell aus [Rei00] wurde dabei um das Konzept der Schemaevolution [RD03], sowie um weitere Kontrollflusskonzepte [Wol08] erweitert. Die nachfolgenden Ausführungen beziehen sich im Wesentlichen sowohl auf ADEPT als auch auf ADEPT2. Wo Eigenschaften angesprochen werden, die nur auf ADEPT2 zutreffen, wird dies gesondert vermerkt.

Da es sich bei in ADEPT2 modellierten Prozessen um direkt ausführbare Geschäftsprozesse handelt, wird der zur Modellierzeit erzeugte Kontrollflussgraph[1] (CFS[2]) zur Laufzeit mit Markierungen versehen. Diese Markierungen der Knoten und Kanten stellen, im Gegensatz zu auf Petrinetzen [Pet62, Bau96] basierenden Konzepten, jederzeit die Ausführungshistorie einer Prozessinstanz dar[3]. Dadurch sind bereits ausgeführte, noch nicht ausgeführte oder in Ausführung befindliche Knoten sofort anhand ihrer Markierung erkennbar.

Wie bei den meisten heutigen PMS, die in der Lage sind, Prozesse nicht nur darzustellen, sondern auch auszuführen, wird bei ADEPT2 zwischen *Prozessvorlage* (auch *Prozessschema*) und *Prozessinstanz* unterschieden. Eine Prozessvorlage stellt dabei die vom Modellierer zur Entwurfszeit erstellte Beschreibung eines Geschäftsprozesses dar. Zur Ausführung erstellt das PMS zur Laufzeit eine logische[4] Kopie dieser Vorlage. Diese Kopie, welche zur Laufzeit mit Markierungen an Knoten und Kanten versehen wird, wird als *Instanz* bezeichnet, der Vorgang des Kopierens und Ausführens entsprechend als *instantiieren*. In Abbildung 2.2 ist eine Prozessvorlage mit drei ihrer Instanzen dargestellt.

ADEPT2 unterscheidet sich von den Prozess-Metamodellen der meisten anderen Systemen dadurch, dass der Graph laufender Instanzen geändert werden kann, wobei jederzeit die Korrektheit des Prozesses gewährleistet bleibt. Darüber hinaus be-

1 Hierbei handelt es sich um einen gerichteten, azyklischen Graphen, dessen Zyklenfreiheit nur durch kontrollierte Verwendung eines speziellen Schleifenkonstruktes aufgehoben werden kann.
2 Control Flow Schema
3 Abgesehen von der Verwendung des expliziten Schleifenkonstruktes.
4 Eine reale Kopie würde den Ressourcenbedarf unnötig erhöhen.

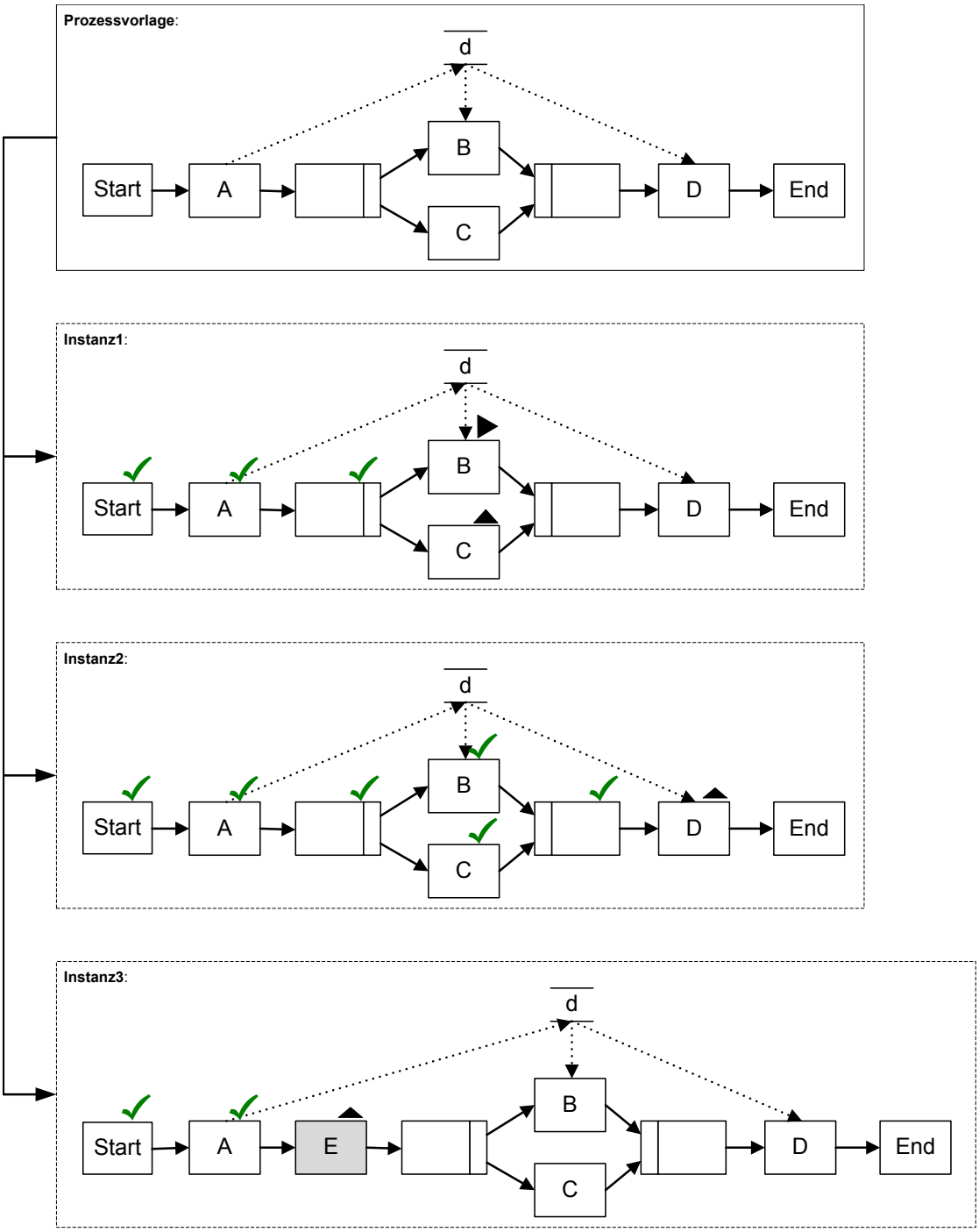


Abbildung 2.2 Mehrfach instantiierte Prozessvorlage

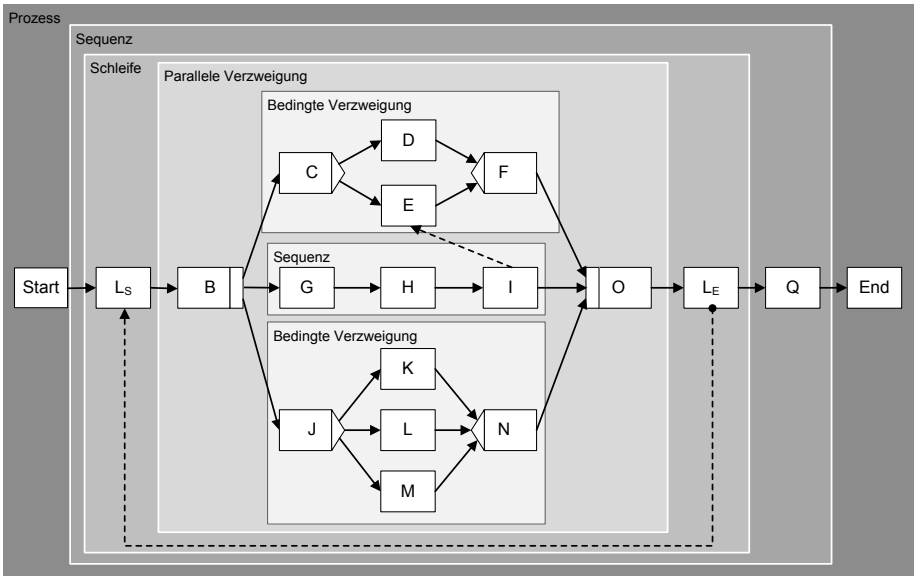


Abbildung 2.3 Schematische Darstellung der Kontrollflusskonstrukte des ADEPT2-Metamodells (nach [Rei00])

steht dank der in [RD03] vorgestellten Konzepte die Möglichkeit, Änderungen an Prozessvorlagen in bereits laufende Instanzen einzubringen, ohne deren Korrektheit zu verletzen. Dabei werden sowohl manuelle als auch durch *Schemaevolution* vorgenommene Änderungen systemseitig abgelehnt, falls sie die Korrektheit einer Instanz i. S. v. [Rei00] verletzen. In Abbildung 2.2 wurde an Instanz 3 zur Laufzeit zusätzlich ein Knoten mit Namen E eingefügt.

2.2.1. Kontrollfluss

Wie bereits eingangs erwähnt, werden Prozesse in ADEPT2 mithilfe einer symmetrischen Blockstruktur dargestellt. Dies bedeutet, dass zur Prozessmodellierung Blöcke mit eindeutigem Start- und Endknoten verwendet werden. Diese Blöcke können sequentiell angeordnet oder ineinander verschachtelt werden. Eine überlappende Anordnung der Blöcke ist nicht erlaubt.

Als Blockkonstrukte stehen *Sequenzen*, *Schleifen*, sowie bedingte und parallele Verzweigungen zur Verfügung. Weiter stellt jeder Prozess selbst einen Block dar, da er ebenfalls einen eindeutigen Start- und Endknoten besitzt.

Knoten werden untereinander mit *Kontrollkanten* verbunden, wobei abgesehen von Start- und Endknoten eines Prozesses jeder Knoten mindestens eine eingehende und mindestens eine ausgehende Kontrollkante besitzt. Mehr als eine ausgehende Kontrollkante können Verzweigungsknoten (Splitknoten) von bedingten oder parallelen Verzweigungen besitzen. Mehr als eine eingehende Kontrollkante sind Vereinigungsknoten (Joinknoten) vorbehalten. Soweit nicht anders beschrieben, beziehen sich Knoten-

namen in diesem Abschnitt auf Abbildung 2.3, die eine Übersicht über die im ADEPT2-Metamodell vorhandenen Kontrollfluss-Konstrukte zeigt.

Sequentielles Ausführen von Knoten

In *Sequenzen* (bspw. Knoten G, H, I) werden alle enthaltenen Knoten zur Laufzeit nacheinander ausgeführt. Dabei ist die Ausführung eines Knotens $n+1$ von der vorherigen Beendigung seines Vorgängerknotens n abhängig.

Paralleles Ausführen von Knoten

Da jedoch eine rein sequentielle Ausführung von Aufgaben für die meisten Geschäftsprozesse nicht ausreichend ist, bietet ADEPT2 die Möglichkeit, *parallele Verzweigungen* abzubilden. Dabei werden alle Nachfolgerknoten des Verzweigungsknotens (Knoten B) parallel ausgeführt, sobald dieser beendet wurde. Entsprechend kann der Vereinigungsknoten (Knoten O) einer parallelen Verzweigung erst dann ausgeführt werden, wenn alle seine Vorgängerknoten beendet wurden.

Bedingte Verzweigungen

Bei *bedingten Verzweigungen* wird zur Laufzeit immer genau ein Nachfolgerknoten des Verzweigungsknotens (Knoten C und J) aktiviert. Die Auswahl erfolgt abhängig von einer am Verzweigungsknoten hinterlegten *Verzweigungsbedingung*. Dabei ist jeder ausgehenden Kontrollkante des Verzweigungsknotens ein Wert oder Wertebereich zugeordnet, bei dessen Übereinstimmung mit der Verzweigungsentscheidung der Zweig gewählt wird. Eine Überlappung der Wertebereiche verschiedener Zweige oder die Zuweisung eines Wertes an mehr als einen Zweig ist bei ADEPT2 nicht zugelassen.

Da nur genau ein Nachfolger des Verzweigungsknotens ausgeführt wird, werden alle weiteren Nachfolger übersprungen. Der Vereinigungsknoten (Knoten F und N) der bedingten Verzweigung wird ausgeführt, sobald einer seiner Vorgänger beendet wurde. Dabei ist durch das Überspringen nicht gewählter Zweige sichergestellt, dass nach dem Ausführen des Vereinigungsknotens kein anderer seiner Vorgänger mehr zur Ausführung kommt. Da Zweige, die nicht mehr zur Ausführung kommen können, als *tot* bezeichnet werden [Bau96], wird dieses Verfahren *dead path elimination* [LA94] genannt.

Schleifen

Aufgaben, die innerhalb eines Prozesses mehrfach ausgeführt werden sollen, können nicht einfach durch eine statische Anzahl von Knoten repräsentiert werden, da häufig beim Erstellen eines Prozesses noch nicht klar ist, wie oft die zu wiederholende Aufgabe ausgeführt werden soll. Um diesem Umstand gerecht zu werden, bietet ADEPT2 die Möglichkeit bestimmte Bereiche eines Prozesses in *Schleifen* einzubetten.

Wie Verzweigungen besitzen auch Schleifen eindeutige Start- (Knoten L_S) und Endknoten (Knoten L_E). Der Bereich zwischen diesen beiden Knoten wird *Schleifenkörper*

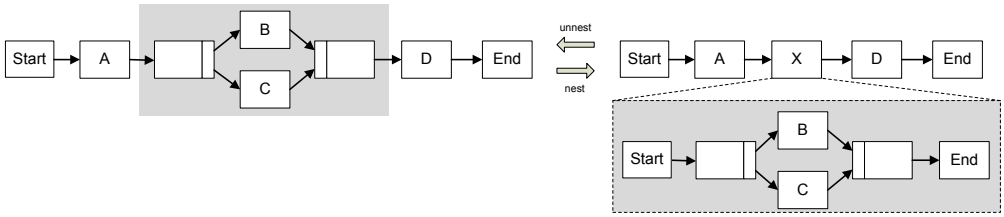


Abbildung 2.4 Bildung und Auflösung von Subprozessen (nach [Rei00])

genannt. Zur Laufzeit wird der Schleifenkörper so oft wiederholt, bis eine am Schleifenendknoten hinterlegte Abbruchbedingung erfüllt ist. Dies hat zur Folge, dass die im Schleifenkörper enthaltenen Knoten und Blöcke mindestens einmal durchlaufen werden.

Synchronisationskanten

Mit den bisher vorgestellten Block-Konstrukten lassen sich bereits viele Geschäftsprozesse beschreiben. Jedoch gibt es Fälle, die mit diesen Konstrukten noch nicht oder nur mit sehr komplexen Prozessgraphen dargestellt werden können [RKD96, RKD97]. Daher gibt es in ADEPT2 *Synchronisationskanten* (Sync-Kanten), die dazu dienen, die Reihenfolge parallel ablaufender Prozessschritte zu synchronisieren (Knoten I und E). Folglich dürfen Synchronisationskanten nur zwischen zwei Knoten aus unterschiedlichen, parallel verlaufenden Zweigen eingesetzt werden. Sie dienen der Festlegung der Reihenfolge zwischen ihrem Start- und ihrem Zielknoten.

Hierarchische Strukturierung

Die Transformation von beliebigen Blöcken eines Prozesses in einen *Subprozess* (*nest*) ist ebenso möglich wie die Transformation eines Subprozesses in einen Block des ihm übergeordneten Prozesses (*unnest*; s. Abb. 2.4).

Die hierarchische Strukturierung dient folglich nur der Übersichtlichkeit von komplexen Prozessen. Die Ausführungsreihenfolge der einzelnen Knoten ist unabhängig davon, ob ein Block direkt in einem Prozess enthalten ist oder ob er als Subprozess an einem Knoten hinterlegt wird. Das Konzept eignet sich jedoch hervorragend, um jedem Prozessbeteiligten die für ihn passende Sicht auf den modellierten Prozess zu bieten.

Funktionen zur Beschreibung von Beziehungen

In ADEPT2 sind einige Funktionen definiert, mit deren Hilfe sich Beziehungen zwischen Elementen eines Prozessgraphen beschreiben lassen. So ist bspw. die Menge aller Vorgänger des Knotens n durch die Funktion $\texttt{pred}^*(\texttt{n})$ gegeben. Eine Übersicht über alle verfügbaren Funktionen ist in Anhang D zu finden. Die Ausführliche Beschreibung dieser Funktionen ist in [Rei00] nachzulesen.

2.2.2. Datenfluss

Um Daten zwischen Prozessschritten auszutauschen, werden in ADEPT2 *Datenelemente* verwendet, die innerhalb eines Prozesses zur Verfügung stehen. Aufgrund dieser prozessweiten Sichtbarkeit besitzen Datenelemente einen Namen, der innerhalb des Prozesses eindeutig ist. Sie werden über Datenkanten (Lese- und Schreibkanten) mit den *Parametern* (Ein- und Ausgabeparametern) von Aktivitäten verbunden.

Um sicherzustellen, dass über die zur Modellierzeit hergestellte Verknüpfung auch zur Laufzeit Daten korrekt ausgetauscht werden können, besitzen sowohl diese Aktivitätenparameter als auch Datenelemente einen Datentyp. Für eine gültige Verknüpfung zwischen einem Datenelement und einem Aktivitätenparameter müssen deren Datentypen identisch sein. Der Name eines Parameters muss bezüglich der Menge der Ein- bzw. Ausgabeparameter einer Aktivität eindeutig sein. Die Namen der Aktivitätenparameter sind für die Verknüpfung mit Datenelementen nicht von Bedeutung.

Für jeden Aktivitätenparameter muss festgelegt werden, ob es sich um einen *optionalen* oder einen *obligaten* Parameter handelt. Dabei zeigen optionale Eingabeparameter an, dass die Aktivität auch dann korrekt ausgeführt wird, wenn der zur Laufzeit aus dem Datenelement übergebene Wert `NULL` ist[5]. Für Ausgabeparameter legt diese Eigenschaft fest, dass der von der Aktivität ausgeführte Schreibzugriff zur Laufzeit möglicherweise nicht stattfindet. Das Nichtausführen des Schreibzugriffes hat zur Folge, dass der Wert des Datenelementes unverändert bleibt, welches mit dem optionalen Ausgabeparameter verknüpft ist.

Demgegenüber darf der zur Laufzeit an einen obligaten Eingabeparameter übergebene Wert nicht `NULL` sein. Dies kann durch obligate Ausgabeparameter sichergestellt werden. Ist der Ausgabeparameter einer Aktivität mit der Eigenschaft *obligat* versehen, so sagt dies aus, dass beim Beenden der Aktivität sicher ein Wert ($\neq$ `NULL`) aus diesem Aktivitätenparameter in das mit ihm verknüpfte Datenelement geschrieben wird.

Eine Verknüpfung von obligaten Aktivitätenparametern wird als *obligater Zugriff* bezeichnet. Entsprechend heißen Verknüpfungen von optionalen Parametern mit Datenelementen *optionale Zugriffe*. Werden Eingabeparameter mit einem Datenelement verknüpft, so sprechen wir von *Lesezugriffen*, bei der Verknüpfung von Ausgabeparametern mit Datenelementen von *Schreibzugriffen*. In Abbildung 2.5 ist ein einfacher Prozess mit jeweils einem Lese- und einem Schreibzugriff auf zwei unterschiedliche Datenelemente dargestellt. Dabei ist zu beachten, dass bei der graphischen Darstellung nicht zwischen obligaten und optionalen Zugriffen unterschieden wird.

2.2.3. Korrektheit

Prozesse, die mit den bisher vorgestellten Elementen modelliert werden, sind in der Lage Sachverhalte aus der Geschäftswelt korrekt abzubilden. Jedoch kann für diese Prozesse noch nicht sichergestellt werden, dass sie auch korrekt vom System ausgeführt werden. Beispielsweise können Verklemmungen durch die Verwendung von Synchronistaionskanten entstehen, oder es besteht die Möglichkeit, dass obligate Lesezugriffe

[5] Dies bedeutet, dass der Wert zum Zeitpuntk des Zugriffs nicht gesetzt ist.

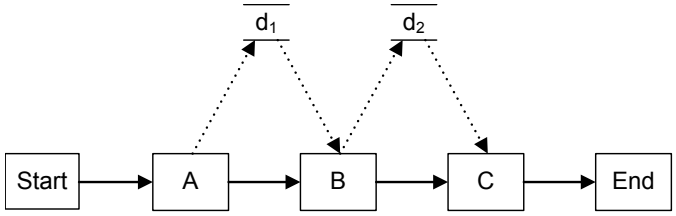


Abbildung 2.5 Einfacher ADEPT2-Prozess mit Datenelementen und Datenflusskanten

nicht mit Daten versorgt werden, da ein vorausgehender obligater Schreibzugriff fehlt. In [Rei00] werden daher Korrektheitskriterien in Form von Regeln für den Kontroll- und Datenfluss formuliert, die im Folgenden informell vorgestellt werden.

Regeln für korrekten Kontrollfluss

Ein ADEPT2-Prozess muss sieben in [Rei00] beschriebene Kriterien (KF-1 – KF-7) erfüllen, damit seine korrekte Ausführung gewährleistet werden kann. Die Einhaltung dieser Korrektheitskriterien kann bereits durch geeignete Implementierung von sog. *Änderungsoperationen* gewährleistet werden. Dadurch entfällt eine aufwendige Analyse und dem Prozessmodellierer werden lästige Korrekturarbeiten erspart. Die Anwendung dieser Änderungsoperationen auf einen korrekten Kontrollflussgraphen führen implizit wieder zu einem korrekten Kontrollflussgraphen.

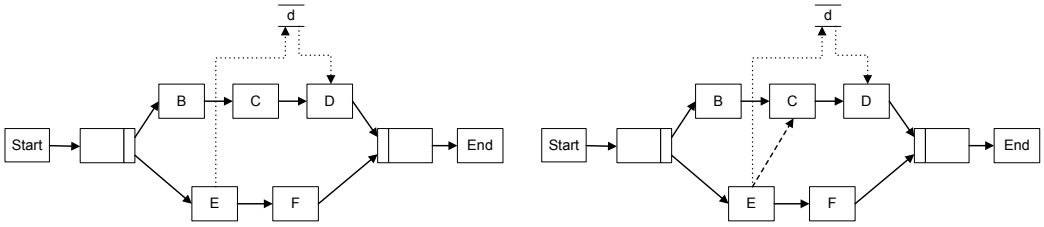
Um die Implementierung zu erleichtern, wurden in [Rei00] Änderungsoperationen beschrieben, die einen gültigen Prozessgraphen in einen neuen, ebenfalls gültigen Prozessgraphen überführen. Eine Implementierung dieser Änderungsoperationen und eine übersichtliche Zusammenfassung ist in [Jur06] zu finden.

Regeln für korrekten Datenfluss

Für die Korrektheit des Datenflusses eines ADEPT2-Prozesses sind drei Regeln festgelegt. Diese lauten wie folgt:

DF-1 (Sichere Versorgung obligater Eingabeparameter) Ist ein Datenelement mit einem obligaten Eingabeparameter einer Aktivität A verbunden, so muss sichergestellt werden, dass dieses vor der Ausführung von A sicher mit Daten versorgt wurde. Das bedeutet, dass auf jedem Pfad, der zur Laufzeit zur Ausführung von A führt, mindestens eine Knoten existieren muss, der das Datenelement obligat schreibt.

DF-2 (Vermeidung paralleler Schreibzugriffe) Auf ein Datenelement darf nicht von mehr als einem Knoten aus unterschiedlichen parallelen Zweigen schreibend zugegriffen werden, es sei denn, die Reihenfolge der Knoten ist durch die Verwendung von Synchronisationskanten eindeutig festgelegt.



(a) Ungültiger Lesezugriff **(b)** Durch Synchronisation gültiger Lesezugriff

Abbildung 2.6 Korrekter Datenfluss durch die Verwendung von Sync-Kanten

DF-3 (Vermeidung direkt aufeinanderfolgender Schreibzugriffe) Zwischen zwei Schreibzugriffen auf ein Datenelement soll dieses mindestens einmal gelesen werden. Diese Regel muss jedoch nicht zwingend erfüllt werden, da ein Verstoß den korrekten Ablauf eines Prozesses nicht gefährdet. Sie dient lediglich der Erkennung unsinniger Modellierung von sog. „lost updates“.

Datenflussanalyse

Im Gegensatz zur impliziten Korrektheit des Kontrollflusses, muss die Datenflussanalyse explizit nach jeder Änderungsoperation erfolgen, die auf einem Prozessgraphen durchgeführt wurde. Dies hat den einfachen Grund, dass für die Behebung eines fehlerhaften Datenflusses mehrere Lösungen existieren, wie bspw. das Einfügen neuer Datenkanten oder das Entfernen bedingter Verzweigungen.

Für die Analyse des Datenflusses werden in [Rei00] zwei Algorithmen vorgestellt. Der `WriterExists`-Algorithmus dient der Sicherstellung von DF-1, wohingegen der Algorithmus `ParallelWriterExists` analysiert, ob parallele Schreibzugriffe aus nicht synchronisierten Zweigen erfolgen (DF-2). Die beiden Algorithmen werden im Folgenden grob beschrieben. Für eine detaillierte Ausführung verweisen wir auf [Rei00].

`WriterExists`-Algorithmus Dieser Algorithmus wird für die Analyse der sicheren Versorgung obligater Lesezugriffe verwendet. Er basiert auf der Idee, dass ein solcher Lesezugriff eines Knotens n dann sicher mit Daten versorgt ist, wenn er für die direkten Vorgänger von n versorgt ist oder diese Vorgänger das zu lesende Datenelement selbst obligat mit Daten versorgen.

Darauf aufbauend markiert der Algorithmus so lange Nachfolgerknoten schon obligat schreibender Knoten oder bereits versorgter Knoten als *versorgt*, bis der zu untersuchende Knoten selbst versorgt ist oder keine Knoten mehr als *versorgt* markiert werden können. Dabei wird die Markierung nicht nur über Kontrollflusskanten hinweg propagiert, sondern auch über Synchronisationskanten, da eine geeignete Synchronisation parallel ausgeführter Knoten ebenfalls für die Versorgung eines Lesezugriffs sorgen kann. So ist in Abbildung 2.6a der lesende Zugriff von Knoten D auf das Datenelement d nicht erlaubt, wohingegen dieser durch das Einfügen einer Synchronisationskante (Abb. 2.6b) möglich wird.

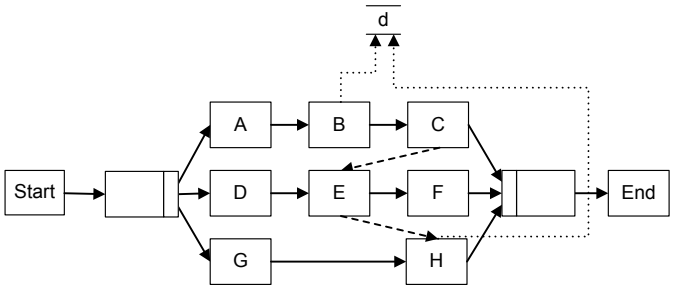


Abbildung 2.7 Durch Sync-Kanten (transitiv) gewährleistete eindeutige Ausführungsreihenfolge paralleler Schreibzugriffe

`ParallelWriterExists`-**Algorithmus** Um nicht-synchronisierte parallele Schreibzugriffe auf ein Datenelement zu erkennen, wird zunächst festgestellt, ob Knoten existieren, die ein Datenelement d schreiben[6] und die in parallel auszuführenden Zweigen liegen. Existieren Knoten mit diesen Eigenschaften, so wird geprüft, ob die Reihenfolge ihrer Ausführung eindeutig festgelegt ist. Eine eindeutig festgelegte Reihenfolge entsteht etwa durch eine Synchronisationskante zwischen den beiden Zweigen. Weiter ist es möglich, dass die Eindeutigkeit der Reihenfolge transitiv über mehrere Zweige hinweg zustande kommt (s. Abb. 2.7).

2.2.4. Dynamisches Verhalten

Um zur Laufzeit den Zustand von Aktivitäten und Kanten einer Prozessinstanz darzustellen, werden in ADEPT2 Knoten- bzw. Kantenmarkierungen verwendet. Die verfügbaren Markierungen und die durch sie angezeigten Zustände sind in Abbildung 2.8 dargestellt.

Beim Start eines Prozesses befinden sich alle Knoten im Zustand `NOT_ACTIVATED`, alle Kontrollkanten haben den Zustand `NOT_SIGNALED`. Ist die Ausführung eines Knotens abgeschlossen, so wechselt er in den Zustand `COMPLETED` und alle seine ausgehenden Kanten werden mit `TRUE_SIGNALED` markiert. Die einzige Ausnahme stellt der Verzweigungsknoten einer bedingten Verzweigung dar. Nachdem dieser beendet wurde, wird lediglich die Kontrollflusskante mit `TRUE_SIGNALED` markiert, die der Verzweigungsentscheidung entspricht. Alle anderen Kanten werden mit `FALSE_SIGNALED` markiert (dead path elimination).

Besitzen alle eingehenden Kanten eines Knotens einen Zustand, welcher sich von `NOT_SIGNALED` unterscheidet, so wechselt er in den Zustand `ACTIVATED`, wenn mindestens eine eingehende Kante den Zustand `TRUE_SIGNALED` besitzt. Andernfalls (alle eingehenden Kanten haben den Zustand `FALSE_SIGNALED`) wird der Knoten als `SKIPPED` markiert, woraufhin seine ausgehenden Kanten ebenfalls mit `FALSE_SIGNALED` markiert werden.

[6] Hier kommen auch optional schreibende Knoten als parallele Schreiber in Frage.

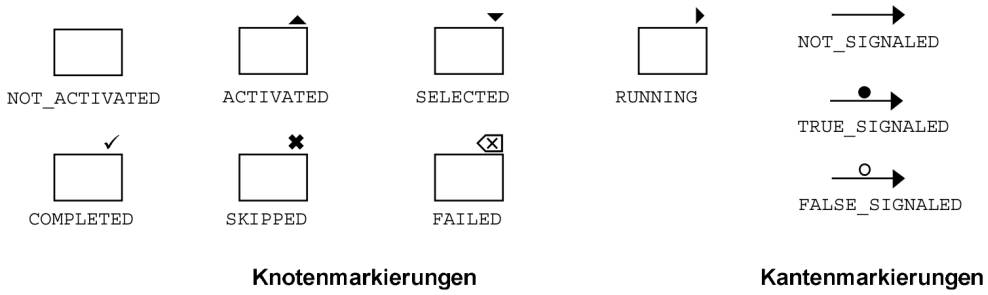


Abbildung 2.8 In ADEPT2 verwendete Knoten- und Kantenmarkierungen (aus [Rei00])

Knoten, die sich im Zustand ACTIVATED befinden, können entweder vom Benutzer reserviert (sie bekommen die Zustandsmarkierung SELECTED) oder direkt ausgeführt werden (sie wechseln in den Zustand RUNNING). Nach erfolgreicher Beendigung eines Knotens wechselt dieser in den Zustand COMPLETED. Schlägt die Ausführung eines Knotens fehl, so wird dieser mit FAILED markiert und die Ausführung der Prozessinstanz abgebrochen. Die Zustände von Knoten in einem ADEPT2-Prozess und die Übergänge zwischen diesen Zuständen sind in Abbildung 2.9 dargestellt.

Datenfluss zur Laufzeit

Beim Start einer Prozessinstanz haben alle Datenelemente den Wert NULL. Dieser wird beim ersten Schreibzugriff durch eine Aktivität verändert. Es ist nicht möglich, den Initialzustand eines Datenelementes wieder herzustellen. Durch das Schreiben von NULL in ein Datenelement bleibt dessen Wert unverändert.

Der Datenfluss in Leserichtung findet zur Laufzeit immer dann statt, wenn eine Aktivität in den Zustand RUNNING wechselt. Dabei werden die Daten aus den Datenelementen in den mit ihnen verknüpften Eingabeparameter der Aktivität zur Verfügung gestellt. Der ausgehende Datenfluss wird ausgeführt, wenn die Aktivität in den Zustand COMPLETED wechselt. Dabei werden alle Daten aus den Ausgabeparametern in die verknüpften Datenelemente übertragen.

2.2.5. Instanzspezifische Änderungen

Im Unterschied zu den meisten heute am Markt verfügbaren PMS erlaubt ADEPT2 die Änderung von bereits in Ausführung befindlichen Prozessinstanzen. Diese instanzspezifischen Änderungen („Ad-hoc-Änderungen“) bieten die Möglichkeit, flexibel auf nicht planbare Abweichungen vom ursprünglichen Prozessablauf zu reagieren. Dabei stehen alle Operationen für Änderungen zur Verfügung, die auch für das Erstellen von Prozessvorlagen verwendet werden.

Neben den bereits vorgestellten Korrektheitskriterien für Kontroll- und Datenfluss müssen instanzspezifische Änderungen aber auch die jeweilige Markierung einer Prozessinstanz berücksichtigen. So ist es beispielsweise weder gestattet, einen Knoten zu

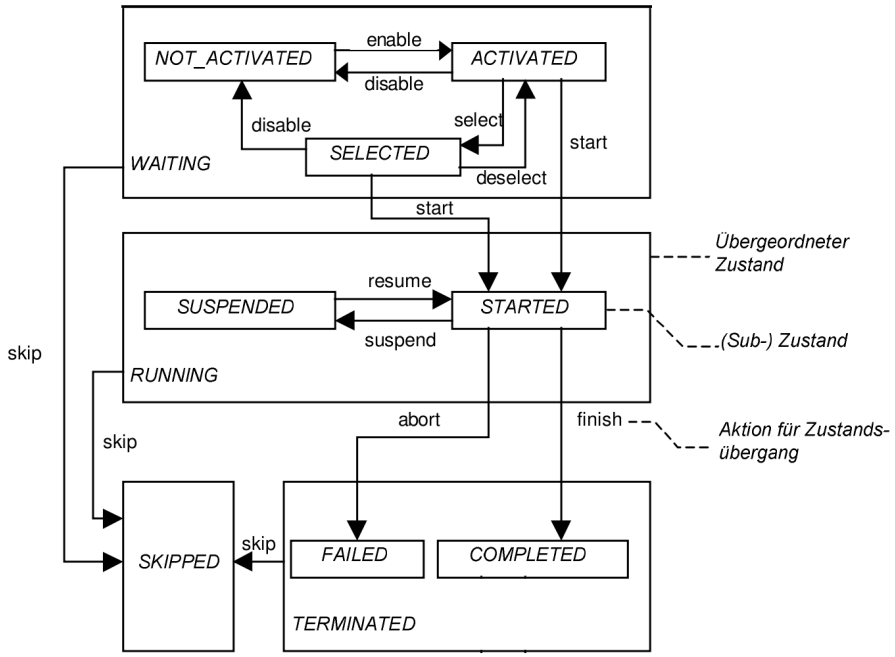


Abbildung 2.9 Zustände und Zustandsübergänge Aktivitäten in ADEPT2 zur Laufzeit (aus [Wol08])

löschen, der bereits ausgeführt wurde, noch ist es, erlaubt Knoten zwischen zwei anderen einzufügen, die übersprungen wurden. Die gesamten Korrektheitsanforderungen für instanzspezifische Änderungen können wie folgt zusammengefasst werden: Eine Prozessinstanz darf zur Laufzeit nur so abgeändert werden, dass ihre Ausführungshistorie auch mit den vorgenommenen Änderungen wieder erzeugt werden kann. Die erlaubten Änderungsoperationen sind ausführlich in [Rei00] beschrieben.

2.2.6. Schemaevolution

Wurde das Schema eines Prozesse geändert, auf dessen Grundlage bereits Instanzen gestartet wurden, so lassen sich diese Änderungen auf die bereits laufenden Instanzen übertragen [RD03, Rin04]. Die sogenannte *Schemaevolution* gibt dabei an, welche Voraussetzungen erfüllt sein müssen, damit laufende Prozessinstanzen auf ein neues Schema migriert werden können.

Äquivalent zu instanzspezifischen Änderungen gilt auch für die Migration *unveränderter* Prozessinstanzen, dass sie nur dann auf ein neues Prozessschema migriert werden können, wenn ihre bisherige Ausführungshistorie mit dem neuen Schema wieder erreicht werden kann. Dies ist insbesondere dann der Fall, wenn für die Migration keine Änderungen an Knoten oder Zweigen vorgenommen werden müssen, die bereits den Zustand `COMPLETED` erreicht haben (s. Abb. 2.10).

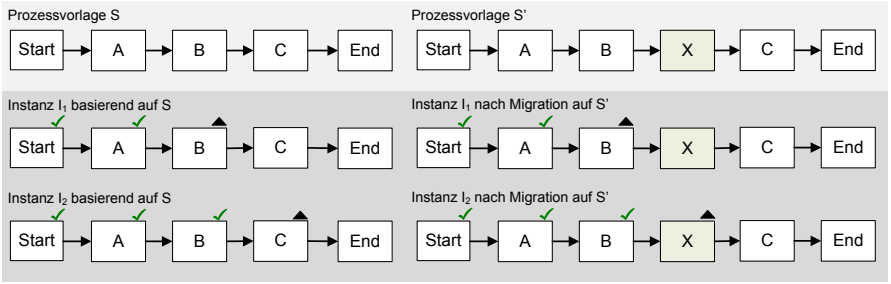


Abbildung 2.10 Schemaevolution: Serielles Einfügen des Knotens X zwischen B und C (nach [Rin04])

In [Rin04] werden acht Korrektheitskriterien (*compliance conditions*) für die Migration unveränderter Prozessinstanzen auf ein neues Prozessschema vorgestellt. Dabei wird differenziert, ob die Unterschiede zwischen Instanz und Prozessschema *additive*, *subtraktive* oder *umordnende* Änderungen an der laufenden Instanz erfordern, oder ob es sich dabei um Änderungen handelt, die den Datenfluss betreffen. Das Korrektheitskriterium für den Datenfluss ist für diese Arbeit von besonderer Bedeutung. Die vier darin enthaltenen Regeln lauten wie folgt:

- Das *Einfügen* eines neuen Datenelementes in eine laufende Prozessinstanz ist jederzeit möglich.
- Ein Datenelement darf nur dann *gelöscht* werden, wenn es mit keiner Aktivität verbunden ist, die sich im Zustand `RUNNING` oder `COMPLETED` befindet.
- Eine lesende Datenkante darf in eine laufende Instanz *eingefügt* oder *gelöscht* werden, wenn sich die zu verknüpfende Aktivität nicht im Zustand `RUNNING` oder `COMPLETED` befindet.
- Eine schreibende Datenkante darf in eine laufende Instanz *eingefügt* oder *gelöscht* werden, wenn sich die zu verknüpfende Aktivität nicht im Zustand `COMPLETED` befindet.

Die Migration von Prozessinstanzen, die bereits durch den Benutzer verändert wurden, ist für diese Arbeit nicht von Bedeutung, da die wesentlichen Konzepte bereits für unveränderte Prozessvorlagen erläutert wurden. Für interessierte Leser verweisen wir auf [Rin04].

2.2.7. Das ADEPT2-Softwaresystem

Die Konzepte aus dem ADEPT2-Metamodell wurden im Rahmen des *AristaFlow*-Projektes zunächst prototypisch als ADEPT-Softwaresystem implementiert. Die dabei gemachten Erfahrungen wurden genutzt, um das ADEPT2-(Software-)System zu entwickeln. Dabei handelt es sich um ein vollständiges PMS, dessen Komponenten im Folgenden kurz beschrieben werden.

Erstellen und Verwalten von Aktivitätenvorlagen

Um Anwendungen in ADEPT2 zu integrieren, müssen diese zunächst in einer *Aktivitätenvorlage* beschrieben werden. Diese Vorlage enthält Informationen darüber, um welche Art von Anwendung (Java-Klasse, ausführbare Binärdatei, Webservice, etc.) es sich handelt, wie diese Anwendung zur Laufzeit auszuführen ist und welche Parameter die Anwendung für die Konfiguration, sowie für die Ein- und Ausgabe von Daten besitzt. Wie eine Aktivitätenvorlage zu erstellen ist, hängt von der zu verwendenden Anwendung ab. Meist wird für die Integration bestehender Softwaresysteme vom *Aktivitätenentwickler* ein spezieller Anwendungsbaustein entwickelt, welcher die Kommunikation zwischen ADEPT2 und der einzubindenden Anwendung durchführt. Dabei kann es sich um eine bereits von einem *Anwendungsentwickler* implementierte Anwendung handeln, oder um eine Anwendung, die vom Aktivitätenentwickler zusammen mit dem Anwendungsbaustein implementiert wird. So kann etwa das bestehende Paket für Büroanwendungen, OpenOffice.org[7], durch einen zusätzlichen Anwendungsbaustein (*Wrapper-Aktivität*) integriert werden, wohingegen ein einfaches Formular für das Einreichen eines Urlaubsantrages vom Aktivitätenentwickler direkt innerhlab der Aktivität implementiert werden kann.

Die Verwaltung der Aktivitätenvorlagen findet im *Aktivitäten-Repository* statt und wird dort vom (Aktivitäten-)*Administrator* durchgeführt. Neben der Speicherung der Aktivitätenvorlagen übernimmt das Aktivitäten-Repository auch die Verwaltung verschiedener Versionen einer Aktivitätenvorlage. Dadurch ist es möglich, verschiedene Versionen einer Aktivitätenvorlage in unterschiedlichen Prozessen zu verwenden.

Entwickeln von Anwendungsbausteinen

Wie bereits erwähnt, muss der Aktivitätenentwickler für jede zu integrierende Anwendung einen Anwendungsbaustein entwickeln. Damit dieser Anwendungsbaustein in ADEPT2 als Aktivität verwendet werden kann, muss er die Schnittstelle `ExecutableComponent` implementieren. Diese Schnittstelle dient der Kommunikation zwischen ADEPT2 und dem Anwendungsbaustein, der entweder selbst die zu integrierende Anwendung darstellt, oder seinerseits mit der zu integrierenden Anwendung kommuniziert.

Für den Zugriff auf Ein- und Ausgabeparameter der Aktivität steht die Schnittstelle `DataContext` zur Verfügung. Sie bietet lesenden Zugriff auf die Eingabeparameter, sowie lesenden und schreibenden Zugriff auf die Ausgabeparameter der Aktivität. Dadurch lassen sich die Ausgabeparameter zur Laufzeit der Aktivität als Zwischenspeicher verwenden. Der Zugriff auf den Wert eines Aktivitätenparameters erfolgt, indem die entsprechende Funktion der Schnittstelle mit dem Namen des Parameters als Argument aufgerufen wird. Weitere Schnittstellen existieren für den Zugriff auf Konfigurationsparameter der Aktivität, welche jedoch für diese Arbeit nicht relevant sind.

[7] www.openoffice.org

Verwaltung des Organisationsmodells

In einem *Organisationsmodell* werden Benutzer eines Softwaresystems und deren Beziehungen abgebildet. Diese orientieren sich zumeist an der Stellung der Personen innerhalb des Unternehmens (*org position*) und an deren Abteilungszugehörigkeit (*org unit*). Das ADEPT2-Organisationsmodell [Ber05] erlaubt es darüber hinaus, einem Benutzer (*agent*) Fähigkeiten (*abilities*) zugewiesen und ihn in verschiedene Projekt- (*project group*) und Organisationsgruppen (*org group*) einzuordnen.

Erstellen und Verwalten von Prozessvorlagen

Prozessvorlagen werden im ADEPT2-System vom *Prozessmodellierer* gemäß der Spezifikationen des ADEPT2-Metamodells erstellt. Die dafür verwendete Softwarekomponente heißt PTE[8]. Beim PTE handelt es sich um ein graphisches Modellierungswerkzeug zur Manipulation von Prozessgraphen gemäß der in ADEPT2 spezifizierten Änderungsoperationen. Dies hat zur Folge, dass sich ausschließlich gültige ADEPT2-Prozessvorlagen mit diesem Werkzeug erstellen lassen.

Um lauffähige Prozessschritte zu erhalten, wird an jedem Knoten eine Aktivitätenvorlage aus dem Aktivitäten-Repository hinterlegt. Die Verknüpfung aus Knoten und Aktivitätenvorlage wird als *Aktivität* bezeichnet. Bereits während der Zuordnung von Knoten und Aktivitätenvorlage können die Ein- und Ausgabeparameter der entstehenden Aktivität mit bestehenden oder neuen Datenelementen verknüpft werden. Das dabei entstehende Datenflussschema wird permanent auf Gültigkeit bzgl. der Korrektheitskriterien geprüft, die des ADEPT2-Metamodells vorschreibt. Weiter wird jede Aktivität mit einer entsprechenden Regel für die Bearbeiterzuordnung (SAR[9]) versehen. Diese legt fest, welcher Benutzer den entsprechenden Prozessschritt später ausführen soll oder ob die Aktivität automatisch ausgeführt wird.

Die fertigen Prozessvorlagen werden im *Prozess-Repository* gespeichert, von wo aus sie mit dem *ADEPT2-Client* instantiiert werden können. Das Prozess-Repository ist zusätzlich für die Verwaltung verschiedener Versionen einer Prozessvorlage zuständig. Dadurch ist es möglich, neue Prozessinstanzen auf der Basis einer geänderten Prozessvorlage zu starten, während bereits laufende Instanzen weiter das alte Prozessschema verwenden. Sind die an der Prozessvorlage vorgenommenen Änderungen auch für bereits laufende Instanzen relevant, so können diese durch Schemaevolution auf das neue Schema migriert werden.

Die Sicht des Prozessmodellierers wird als *prozesslogische* Sicht oder *Prozesslogik* bezeichnet. Sie beinhaltet das Erstellen der Prozessgraphen (Kontroll- und Datenfluss), sowie das Zuordnen von Aktivitätenvorlagen zu Knoten des Kontrollflussgraphen. Das Erstellen dieser Aktivitätenvorlagen und das Entwickeln von Anwendungsbausteinen sind keine Bestandteile der Prozesslogik.

---

8 Process Template Editor
9 Staff Assignment Rule

2.3. Datentypen

Um Daten in der EDV darstellen zu können, sind Informationen über deren Beschaffenheit notwendig. Diese Informationen werden in *Datentypen* abgebildet. Datentypen beschreiben dabei die Struktur und den Wertebereich eines Datums, sowie die Operationen, die auf diesen Daten ausgeführt werden können. Datentypen kommen in Datenbanken, in Tabellenkalkulationen, in Programmiersprachen, aber auch in PMS zum Einsatz. Wir beschreiben im Folgenden den Unterschied zwischen elementaren und zusammengesetzten Datentypen und erläutern anschließend, welche Datentypen derzeit in ADEPT2 verwendet werden.

2.3.1. Elementare Datentypen

Das grundlegende Auszeichnungsmerkmal *elementarer* Datentypen ist die Fähigkeit, immer genau einen Wert aufnehmen zu können. Dabei gibt es unterschiedliche Auffassungen darüber, wie dieser Wert beschaffen sein kann. Im Allgemeinen handelt es sich dabei um nicht weiter teilbare Werte, wie eine ganze Zahl (`Integer`) oder ein einzelnes Zeichen (`Char`). Neben den bereits genannten Datentypen zählen auch Aufzählungen, Gleitkommazahlen und boolesche Werte zu den elementaren Datentypen. Die Verwendung elementarer Datentypen ist in Listing 2.1 beispielhaft dargestellt.

```
char zeichen;
int zahl;
float kommazahl;
```

Listing 2.1 Verwendung elementarer Datentypen in C

2.3.2. Zusammengesetzte Datentypen

Aus diesen elementaren Datentypen lassen sich weitere Datentypen ableiten. Da sie aus elementaren Datentypen zusammengesetzt werden, tragen diese Datentypen die Bezeichnung *zusammengesetzte* Datentypen.

Zu den zusammengesetzten Datentypen gehören Zeichenketten (`String`), die aus aneinandergereihten Zeichen (`Char`) entstehen, Aneinanderreihungen elementarer Datentypen (`Array`), sowie *strukturierte* (auch *zusammengesetzte*) Datentypen, die aus Feldern verschiedenen Typs zusammengesetzt werden. Strukturierte Datentypen haben in unterschiedlichen Programmiersprachen unterschiedliche Namen, sowie leicht unterschiedliche Bedeutungen.

In *prozeduralen* Programmiersprachen wie C oder PASCAL werden strukturierte Datentypen (sie heißen dort `Struct` bzw. `Record`) als reiner Verbund mehrerer Datentypen erstellt (s. Listing 2.2). Für die einzelnen Komponenten dieses Verbundes ist jeder beliebigen Datentyp zulässig. Dieser kann also elementar oder selbst wieder zusammengesetzt sein.

```
struct auto{
  char farbe[20];  // Die Farbe des Autos
  int leistung,    // Leistung des Autos in PS
  geschwindigkeit, // aktuelle Geschwindigkeit des Autos
  v_max;           // Höchstgeschwindigkeit des Autos
}
```

Listing 2.2 Deklaration eines zusammengesetzten Datentyps in C

2.3.3. Datentypen in ADEPT2

Für die bisherigen Datenflusskonzepte ist der Typ von Datenelementen und Aktivitätenparametern nur für deren Kompatibilität von Bedeutung. Bei der Datenflussanalyse wird davon ausgegangen, dass es sich bei den enthaltenen Daten um einzelne Werte handelt, die jedoch im Sinne der vorangehenden Definition nicht elementar sein müssen. So ist beispielsweise auch eine Zeichenkette ein einzelner Wert.

Da in der Praxis jedoch konkrete Datentypen unumgänglich sind, werden in der Implementierung des ADEPT2-Systems die folgenden Datentypen unterstützt: Zeichenketten (`String`), boolesche Werte (`Boolean`), Ganzzahlen (`Integer`), Fließkommazahlen (`Float`), Zeitangaben (`Date`), Ressourcenbezeichner (`URI`) und benutzerdefinierte Datentypen (`UDT`).

UDTs sind dabei keine eigenen Datentypen, die einem Datenelement oder Parameter zugeordnet werden können, sondern bilden die Oberklasse für alle benutzerdefinierten Datentypen. Diese einzelnen benutzerdefinierten Datentypen unterscheiden sich von den restlichen Datentypen dadurch, dass ihre Struktur dem System nicht bekannt ist. So kann beispielsweise ein Datentyp XML[10] implementiert werden, der sich aus Sicht von ADEPT2 wie ein einzelner Wert verhält. Der Wert muss dann atomar gelesen und geschrieben werden und muss von der Anwendung selbst interpretiert werden.

Außer UDTs können alle Datentypen vom System interpretiert werden. Sie können daher als Entscheidungsgrundlage für Verzweigungs- und Schleifenbedingungen verwendet werden.

2.4. Grundlagen objektorientierter Programmierung

In der Softwaretechnik hat sich in den vergangenen Jahren das Paradigma der *objektorientierten Programmierung* [PH09, GR83] etabliert. Objektorientierte Programmiersprachen sind bspw. Java, C# oder Smalltalk. Im Gegensatz zur *prozeduralen* Programmierung, bei welcher die (evtl. in Prozeduren aufgeteilten) Anweisungen eines Programms in linearer Abfolge ausgeführt werden, werden bei der objektorientierten Programmierung Eigenschaften von Objekten durch zu diesen Objekten gehörende Methoden verändert oder abgerufen. Die Anweisungen innerhalb einer Methode können dabei selbst

[10] Extensible Markup Language

```
class Auto{
  String farbe;
  int leistung, geschwindigkeit, v_max;
}
```

Listing 2.3 Deklaration einer einfachen Klasse

wieder Aufrufe von Objektmethoden oder aber auch ein imperativer Programmablauf sein.

Wir interessieren uns im Rahmen dieser Arbeit vor allem für das *Typsystem* der Objektorientierung. Mit einem objektorientierten Typsystem ist es möglich, sowohl neue Objekte zu definieren als auch Beziehungen zwischen Objekten zu beschreiben. Wir erläutern zunächst den Aufbau von Objekten und anschließend sowohl, wie *Vererbung* von Objekten funktioniert als auch, wie Objekte *geschachtelt* werden können.

2.4.1. Definieren von Objekten und Objekteigenschaften

Objekte bilden den Grundbaustein der objektorientierten Programmierung. Wir haben in Abschnitt 2.3 gezeigt, wie Datentypen in der imperativen Programmiersprache C deklariert werden. Die Deklaration eines äquivalenten Datentyps ist auch in objektorientierten Sprachen möglich (Listing 2.3). Die Deklaration selbst unterscheidet sich in diesem einfachen Fall nur unwesentlich von der imperativen Deklaration. Trotzdem ist bereits zu erkennen, dass die Deklaration des Typs `Auto` nicht mehr als strukturierter Datentyp, sondern als *Klasse* erfolgt.

Eine Klasse stellt dabei eine Vorlage für Objekte dar, ähnlich der bereits vorgestellten Prozessvorlage in PMS. Wie auch Prozessvorlagen, müssen Klassen zunächst instantiiert werden, damit aus ihnen ein verwendbares Objekt entsteht. Wir abstrahieren jedoch von dieser Tatsache und verwenden im weiteren die Konvention, dass eine Klasse ein Objekt selbst beschreibt, obwohl sie tatsächlich nur die Vorlage für Objektinstanzen darstellt.

In der Klasse werden dabei nicht nur die Eigenschaften des Objektes in Form von *Objektvariablen* festgelegt, sondern auch die Funktionsweise des Objektes in Form von *Methoden*. In Beispiel 1 sind einige Aspekte der objektorientierten Programmierung anhand eines vereinfachten Autos dargestellt.

Beispiel 1 (Vereinfachtes Auto als Objekt) *Ein Auto besitzt die Eigenschaften* `Farbe`, `Leistung`, `Geschwindigkeit` *und* `v_max` *(Höchstgeschwindigkeit). Weiter kann ein Auto (über Objektmethoden) gestartet, beschleunigt oder abgebremst werden. Dabei ist die Beschleunigung abhängig von der Leistung des Fahrzeugs. So erhöht bei einem Wagen mit 100PS der Aufruf von* `auto.beschleunigen()` *die* `Geschwindigkeit` *um 10km/h, die selbe Anweisung führt jedoch bei einer Leistung von 350PS zur Erhöhung der Geschwindigkeit um 20km/h. Die Erhöhung der Geschwindigkeit ist selbstverständlich nicht beliebig oft möglich, weswegen die Anweisung* `auto.beschleunigen()` *nur so lange zu einer Veränderung derselben führt, bis die Höchstgeschwindigkeit* `v_max` *erreicht ist.*

```
class Auto{
  String farbe;
  int leistung, geschwindigkeit, v_max;

  public void beschleunigen(){}

  public void verzoegern(){}
}
```

Listing 2.4 Um Objektmethoden erweiterte Klasse

Wir erweitern die bereits vorgestellte Klasse beispielhaft um zwei Methoden zum Beschleunigen und zum Abbremsen des Fahrzeuges (Listing 2.4). Die Funktionsweise der Methoden spielt für uns dabei keine Rolle. Die Beschreibung der beiden Methoden reicht aus, um den Unterschied zur imperativen Definition deutlich zu machen: Mit einem Objekt wird nicht nur ein Datentyp festgelegt, sondern es ist auch eine Interaktion mit dem Objekt möglich.

2.4.2. Schachtelung

Wir haben in Listing 2.4 und 2.3 zur Deklaration der Farbe des Autos bereits eine weitere Klasse verwendet. Die Klasse `String` bezeichnet dabei eine Zeichenkette beliebiger Länge. Durch die derartige Deklaration der Farbe des Autos entsteht eine sogenannte „hat-ein“-Beziehung (engl. „has-a“). Das Auto hat also eine Farbe.

Weiter könnten wir bspw. das Getriebe des Autos als eigenständige Klasse festlegen, die dann Funktionen für das Wechseln der Gänge innerhalb des Objektes Auto bereitstellt. Der Zugriff auf derartig verschachtelte Objekte erfolgt über die Aneinanderreihung der einzelnen Objektnamen. So wird auf das `getriebe` über den Aufruf `auto.getriebe` zugegriffen. Nehmen wir an, dass dieses Getriebe in der Lage ist, jeweils einen Gang nach oben oder unten zu schalten. Ein Schaltvorgang erfolgt dann über den Aufruf von `auto.getriebe.gangAuf` bzw. `auto.getriebe.gangAb`.

2.4.3. Vererbung

Durch die Verwendung objektorientierter Programmierung lässt sich jedoch nicht nur die eben beschriebene „hat-ein“-Beziehung darstellen, sondern auch eine sog. „ist-ein“-Beziehung (engl. „is-a“) [Hun97]. Analog zur realen Welt werden durch diese Beziehung Eigenschaften eines übergeordneten Objektes auf ein untergeordnetes Objekt *vererbt*. Umgekehrt *erbt* das untergeordnete Objekt die Eigenschaften des übergeordneten Objektes.

Dabei ist es möglich, dass ein Objekt seine Eigenschaften an mehrere andere Objekte vererbt. Auch ist es umgekehrt möglich, dass ein Objekt die Eigenschaften mehrerer übergeordneter Objekte erbt, sofern es dabei nicht zu Überschneidungen kommt.

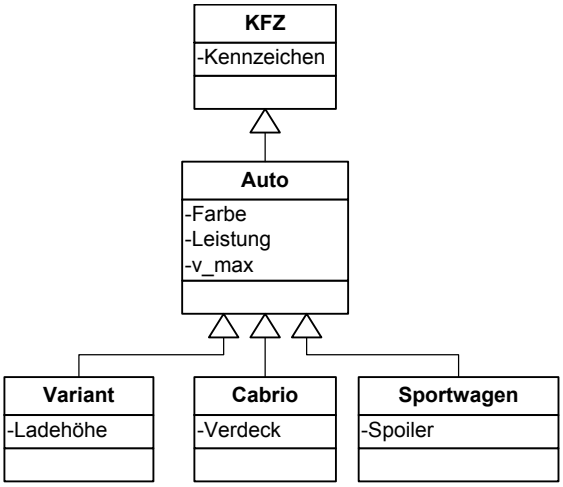


Abbildung 2.11 Darstellung einer Vererbungshierarchie (in UML-Darstellungsweise)

Das letztgenannte Konzept wird *Mehrfachvererbung* genannt. Die Vererbung kann auch transitiv über mehrere Objekte hinweg geschehen (s. Abb. 2.11).

Das dargestellte Cabrio erbt somit alle Eigenschaften des Auto, sowie die Eigenschaften, die das Auto bereits von KFZ geerbt hat. Das Objekt vom Typ `cabrio` hat also neben dem Verdeck auch eine Farbe, eine Leistung, eine Höchstgeschwindigkeit und ein Kennzeichen.

3. Konsumierendes Lesen

In den meisten prozessorientierten Informationssystemen sind Daten, die einmal im Laufe der Ausführung einer Prozessinstanz eingebracht wurden, etwa durch Auslesen von Kundendaten aus einem CRM[1]-System, über die gesamte Lebenszeit der Instanz verfügbar. In einigen Fällen kommt es jedoch auch vor, dass Daten nach einmaligem Lesen ungültig werden. Die Daten werden in diesem Fall nicht nur benutzt, sondern vielmehr *verbraucht*.

Das ADEPT2-Metamodell sieht bisher keine Möglichkeit vor, Datenelemente (bzw. deren Werte) für „ungültig“ zu erklären, nachdem sie einmal im Prozess zur Verfügung gestellt worden sind. Auch gibt es keine Möglichkeit, den Ursprungszustand von Datenelementen wieder herzustellen.

Wir erweitern in diesem Kapitel das in [Rei00] vorgestellte Datenmodell um die Möglichkeit, lesende Zugriffe auf Datenelemente als *konsumierend* zu kennzeichnen. Dazu führen wir zunächst eine zusätzliche Datenzugriffskante ein, um den konsumierenden Lesezugriff für Modellierer und System erkennbar darstellen zu können. Danach klären wir, welchen Einfluss dieses Konzept auf den Datenfluss, sowie auf dessen Analyse hat. Anschließend erläutern wir, wie sich konsumierende Lesezugriffe auf die unterschiedlichen Datentypen aus dem ADEPT2-Prozessmodell auswirken.

3.1. Motivation

Szenarien, in denen Daten durch einen lesenden Zugriff ungültig werden, sind nicht immer offensichtlich. Daher werden im Folgenden, anhand eines Beispieles, Einsatzmöglichkeiten für konsumierende Lesezugriffe illustriert. Wir wenden uns dem Bankwesen zu. Dort betrachten wir den Vorgang einer Onlineüberweisung, welcher den derzeit ca. 150 Mio. Nutzern von Internetbanking in der EU bekannt ist [Eur08].

Beispiel 2 (Konsumierendes Lesen beim Onlinebanking) *Abbildung 3.1 zeigt einen Ausschnitt aus einem Onlinebanking-Prozess. In diesem wird zunächst vom Benutzer eine TAN[2] abgefragt, welche dann zur Autorisierung der nachfolgenden Banktransaktion verwendet wird. Im Fehlerfall, etwa im Falle eines nicht ausreichend gedeckten Kontos, muss die Transaktion wiederholt werden, weswegen sie in einer Schleife eingebettet ist. Wird der Sachverhalt wie abgebildet modelliert, kommt es im Falle einer Fehlerbehandlung und einem damit verbundenen erneuten Durchlaufen des Schleifenkörpers, erneut zu einem Fehler bei der Transaktion, da die in einem vorherigen Schritt vom Benutzer eingegebene Transaktionsnummer nun bereits verbraucht ist. Im Sinne des ADEPT2-Metamodells ist der Prozess*

[1] Customer Relationship Management
[2] Transaktionsnummer

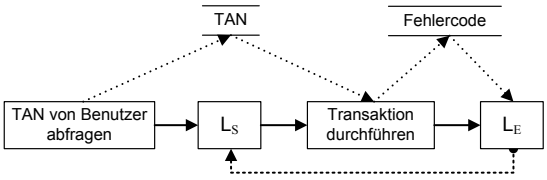


Abbildung 3.1 Mehrfache Verwendung einer Transaktionsnummer

dennoch korrekt modelliert und der Fehler wird nicht erkannt, da er erst zur Laufzeit des Prozesses auftreten wird.

Das Laufzeitverhalten dieses Prozesses im Falle eines Fehlers an Knoten *Transaktion durchführen* ist in Abbildung 3.2 dargestellt. Dabei tritt nach der Eingabe der TAN, die in Zustand (a) bereits erfolgt ist, der Fehler „Konto ungedeckt“ bei der Ausführung von *Transaktion durchführen* auf (b). Dadurch wird der Weg über die Schleifenrücksprungkante gewählt und die Aktivität *Transaktion durchführen* wird erneut aktiviert (c). Da nun die TAN aber aus Sicht der Bank bereits verbraucht ist, kann sie für keine weitere Transaktion verwendet werden. Es kommt erneut zu einem Fehler, der besagt, dass die Transaktionsnummer ungültig ist, weswegen die Schleife erneut durchlaufen wird wird (d). Somit ist der Prozess in eine Endlosschleife geraten und kann nicht mehr korrekt beendet werden. Da die Endlosschleife vom System nicht erkannt wird, muss der Prozess manuell abgebrochen werden.

Dieses Beispiel zeigt einen semantischen Fehler in einem korrekt modellierten Prozess. Mögliche Quellen für derartige Fehler sind die Bearbeitung einer Prozessvorlage durch mehrere Personen oder ein schlechter Informationsaustausch zwischen den Implementierern der Aktivitätenvorlagen und den Prozessmodellierern. Auch ist es in umfangreichen Prozessen, selbst wenn sie nur von einer Person erstellt werden, nahezu unmöglich, die Funktionsweise aller verwendeten Aktivitäten zu kennen. Daher ist es notwendig, die Informationen über das Verhalten von Aktivitäten bezüglich ihres Datenzugriffs im Prozessmodell zu verankern, da nur so die Konsistenz des Datenflusses durch das System sichergestellt werden kann.

3.2. Realisierung

Wir zeigen nun, wie konsumierende Lesezugriffe realisiert werden können, damit sie sowohl für das System als auch für den Modellierer erkennbar sind. Dazu wird zunächst eine neue Datenkante eingeführt. Anschließend wird erläutert, wie diese neue Kante verwendet wird, um die Konsistenz des Datenflusses weiterhin sicherzustellen. Dazu wird der in [Rei00] vorgestellte Algorithmus *WriterExists* derart erweitert, dass er konsumierende Lesezugriffe erkennen und behandeln kann. Dieser Algorithmus überprüft für einen beliebigen Knoten eines Prozesses, ob zur Prozesslaufzeit alle seine Eingabeparameter sicher mit Daten versorgt sind. Das bedeutet nicht nur, dass die Parameter mit Datenelementen verknüpft sind, sondern auch, dass die verknüpften Da-

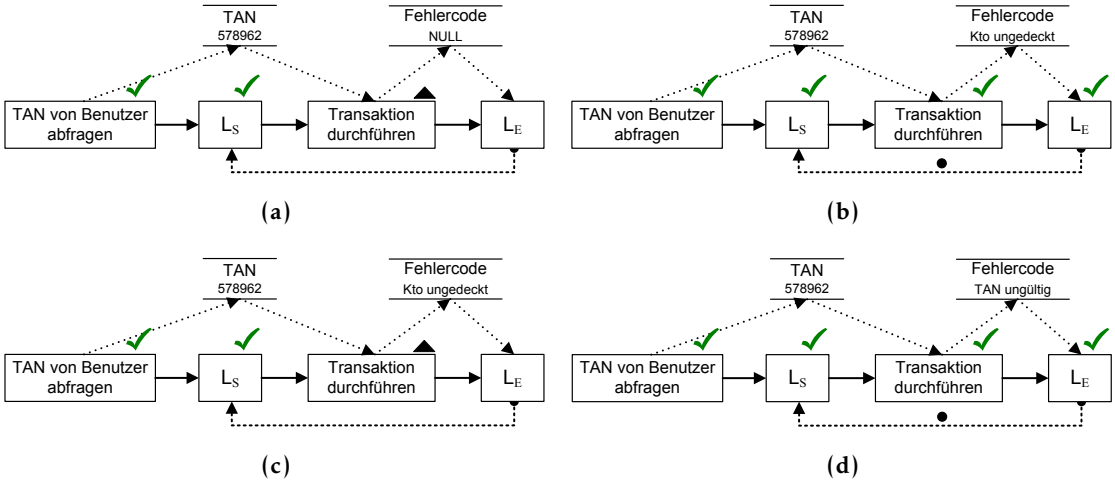


Abbildung 3.2 Laufzeitverhalten eines Prozesses mit mehrfacher Verwendung derselben Transaktionsnummer

tenelemente tatsächlich Daten enthalten. Zuletzt beschreiben wir, welche Auswirkungen das hier vorgestellte Datenflusskonstrukt auf die bisher in ADEPT2 verfügbaren Datentypen hat.

3.2.1. Datenkante für konsumierendes Lesen

Um konsumierende Lesezugriffe für Anwender und Software erkennbar darstellen zu können, ist ein neues Datenflusskonstrukt notwendig. Dabei handelt es sich um eine Datenkante, die das Datenflussmodell aus [Rei00] ergänzt. Diese Datenkante verbindet Datenelemente mit Eingabeparametern von Aktivitäten. Formal:

Definition 1 (Datenkante für konsumierenden Lesezugriff)
Sei durch CFS ein allgemeiner Kontrollflussgraph gegeben.

Eine Datenkante für konsumierenden Lesezugriff beschreibt die Verknüpfung eines Datenelementes d *mit einem Eingabeparameter einer Aktivität. Nachdem die verknüpfte Aktivität ihren Zustand von* `ACTIVATED` *nach* `RUNNING` *geändert hat, ist der Wert von* d *entsprechend* `NULL`. *Wie andere Datenkanten auch, ist die konsumierende Lesekante durch ein 4-Tupel*

$$dfc = (d,n,par,access\_mode)$$

beschrieben.

Dabei gilt: $d \in D$, $n \in N$, $par \in InParams^n$ *und* $access\_mode = read\_consuming$.

Eine konsumierende Lesekante wird durch einen Pfeil mit abwechselnd gestrichelter und gepunkteter Linie dargestellt (vgl. Abb. 3.3)

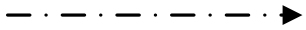


Abbildung 3.3 Datenkante für konsumierenden Lesezugriff

Abgesehen von der konsumierenden Wirkung bezogen auf das gelesene Datenelement verhält sich die konsumierende Lesekante wie eine normale Datenkante für lesende Zugriffe. Insbesondere bleibt die Bedeutung optionaler und obligater Zugriffe erhalten.

Bei der Anwendung dieser Datenkante auf den Prozess aus Beispiel 2 ergibt sich ein konsumierender Lesezugriff der Aktivität „Transaktion durchführen“ (s. Abb. 3.4). Dadurch wird die TAN *verbraucht* und kann nicht mehr für eine weitere Transaktion verwendet werden.

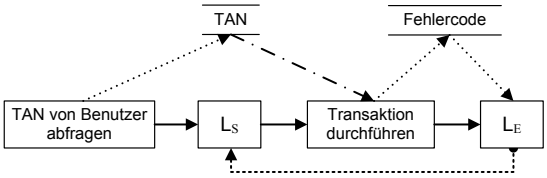


Abbildung 3.4 Korrekte Modellierung mit einmaliger Verwendung der TAN

3.2.2. Anpassung von Aktivitätenparametern

Damit die Information über das konsumierende Verhalten einer Aktivität beim lesenden Zugriff auf ein Datenelement auch im System hinterlegt werden kann, muss die Schnittstelle für den Zugriff auf Aktivitätenparameter in ADEPT2 erweitert werden. Ohne eine solche Erweiterung ist es dem Modellierer zwar möglich, konsumierende Lesezugriffe manuell zu modellieren, jedoch kann eine Aktivität selbst noch nicht anzeigen, dass ihre Lesezugriffe konsumierend sind. Wir ergänzen daher die Parameterbeschreibung wie folgt:

Definition 2 (Eingabeparameter von Aktivitätenvorlagen)
Sei par *ein Parameter einer Aktivitätenvorlage* V.

par $\in$ vParams^{V} *ist durch folgendes Tupel beschrieben:*

$$(pDir^{par}, pName^{par}, pDom^{par}, pDefault^{par}, pDescription^{par}, pDm^{par}, pRm^{par}, pTempl^{par})$$

Dabei ändert sich die Bedeutung von $pDir^{par}$ *wie folgt:*

$pDir^{par} \in \{IN, IN\_CONSUMING, OUT\}$ *gibt an, ob* par *Eingabe-, konsumierender Eingabe- oder Ausgabeparameter von V ist. Die Eingabeparameter einer Aktivitätenvorlage V sind durch die Menge* $InParams^{V} := \{p \in vParams^{V} \mid pDir^{p} \in \{IN,IN\_CONSUMING\}\}$ *zusammengefasst.*

Alle anderen Parametereigenschaften behalten ihre Bedeutung aus [Rei00].

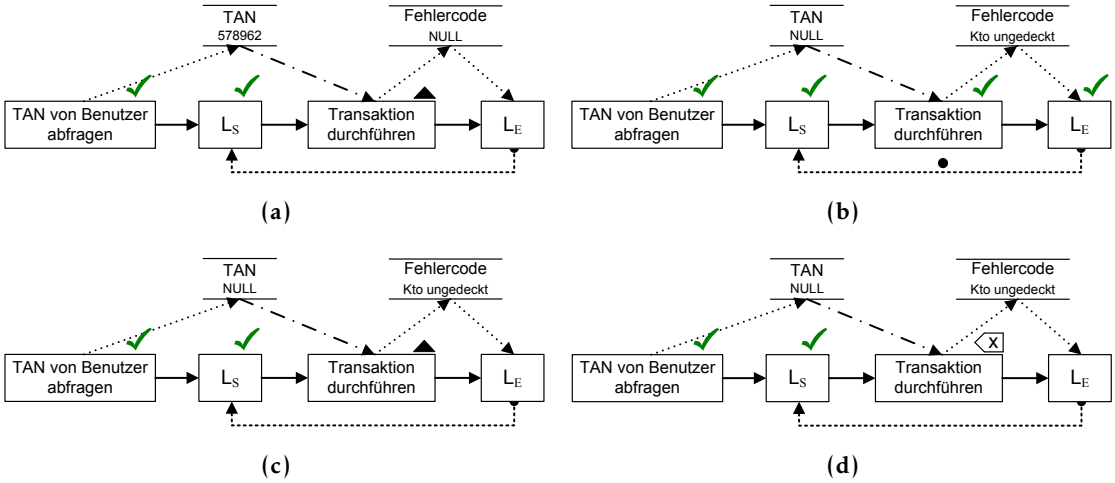


Abbildung 3.5 Laufzeitverhalten eines Prozesses mit *konsumierender* Verwendung einer TAN

Der *access_mode* einer Datenkante, die den lesenden Zugriff einer Aktivität auf ein Datenelement anzeigt, wird automatisch auf *read_consuming* geändert, wenn sie mit einem Parameter *par* verknüpft wird, der mit der Eigenschaft $pDir^{par}$={*IN_CONSUMING*} versehen ist.

Zu beachten ist, dass konsumierende Lesekanten auch mit Eingabeparametern verknüpft werden können, deren Richtungsangabe $pDir^{par}$={*IN*} ist. Dies entspricht einem konsumierenden Lesezugriff, der manuell vom Modellierer eingefügt wurde. Umgekehrt ist jedoch das Aufheben der konsumierenden Eigenschaft eines Eingabeparameters durch den Modellierer nicht möglich. Diese Änderung ist nur an der Aktivitätenvorlage selbst möglich. Andernfalls würde der Modellierer Gefahr laufen, einen Prozess zu erstellen, der alle Konsistenzprüfungen besteht und dennoch zur Laufzeit fehlschlägt, obwohl der Grund für das Fehlschlagen bereits beim Modellieren hätte erkannt werden müssen.

3.2.3. Auswirkungen auf den Datenfluss

Die neue Art des Lesezugriffs hat gravierende Auswirkungen auf den Datenfluss. Es kann nicht länger davon ausgegangen werden, dass einmal obligat geschriebene Daten im gesamten weiteren Prozessverlauf zur Verfügung stehen. Das Laufzeitverhalten des Prozesses aus Beispiel 2 ist in Abbildung 3.5 zu sehen. In Zustand (d) ist erkennbar, dass keine Endlosschleife mehr entsteht. Stattdessen wird die Aktivität „Transaktion durchführen“ systemseitig abgebrochen, da der Eingabeparamter für die TAN nicht mehr versorgt ist.

Ein derartiges Laufzeitverhalten gilt es durch geeignete Prozessmodellierung zu verhindern. Folglich muss die Analyse des Datenflusses erweitert werden, damit konsumierende Lesezugriffe erkannt werden und der Prozessmodellierer auf diese hingewie-

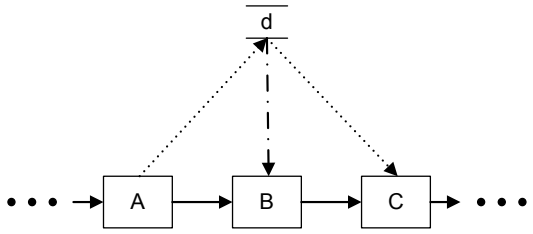


Abbildung 3.6 Ungültiger Lesezugriff nach konsumierendem Lesen

sen werden kann. Da sich diese Erweiterungen unterschiedlich auf zyklische und azyklische Kontrollflussgraphen auswirken, betrachten wir zunächst die Folgen, die sich für azyklische Kontrollflussgraphen ergeben. Anschließend besprechen wir, welche zusätzlichen Auswirkungen konsumierende Lesezugriffe auf zyklische Prozessgraphen haben.

Obligate Lesezugriffe setzen das sichere Vorhandensein von Daten voraus, wohingegen obligate Schreibzugriffe dieses Vorhandensein gewährleisten. Einem obligaten Lesezugriff muss daher immer ein obligater Schreibvorgang vorausgehen. Im Gegensatz dazu sind optionale Lesezugriffe immer gestattet und über optionale Schreibzugriffe kann zur Modellierzeit keine Aussage getroffen werden, ob sie zur Laufzeit tatsächlich erfolgen [Rei00]. Wenn in diesem Kapitel von Lese- und Schreibzugriffen gesprochen wird, so sind immer obligate Operationen gemeint. Die Versorgung für optionale Lesezugriffe muss nicht sichergestellt werden und optionale Schreibzugriffe können nicht als Versorger für obligate Lesezugriffe angesehen werden. Betrachtungen für optionale Operationen können somit entfallen.

Kontrollflussgraph ohne Schleifen

In schleifenlosen Kontrollflussgraphen bedeutet ein konsumierender Lesezugriff, dass das gelesene Datenelement für alle nachfolgenden Knoten, die lesend darauf zugreifen möchten, als nicht versorgt anzusehen ist. Eine Versorgung wird erst wieder durch einen obligaten Schreibzugriff hergestellt, welcher nach dem konsumierenden Lesezugriff erfolgen muss. Dabei ist es unerheblich, ob der konsumierende Lesezugriff optional oder obligat erfolgt. Zwar wird ein Wert bei optionalen Lesezugriffen nur dann gelesen (und damit auch konsumiert), wenn er auch im Datenelement vorhanden ist, jedoch entspricht das Nichtvorhandensein des Wertes vor dem Lesezugriff genau dem Zustand, den das Datenelement annimmt, nachdem der in ihm enthaltenen Wert konsumierend gelesen wurde.

Abbildung 3.6 zeigt einen konsumierenden Lesezugriff durch Knoten B auf das Datenelement *d*. Dieser Zugriff ist durch den Schreibvorgang von Knoten A versorgt. Da die Daten im Datenelement *d* durch den konsumierenden Lesezugriff von Knoten B ungültig geworden sind, ist der Lesezugriff von Knoten C nicht mehr versorgt.

Kontrollflussgraph mit Schleifen

Liegt ein konsumierend lesender Knoten innerhalb einer oder mehrerer Schleifen, so hat dieser Zugriff nicht nur Auswirkungen auf die Nachfolgermenge des Knotens, sondern beeinträchtigt auch seine Vorgänger innerhalb der Schleife. Auch wenn sonst kein weiterer Lesezugriff auf das Datenelement innerhalb der Schleife erfolgt, so unterbricht der konsumierende Lesezugriff wenigstens seine eigene Versorgung für den nächsten Schleifendurchlauf. Folglich muss sichergestellt werden, dass nach dem konsumierenden Lesezugriff wenigstens ein obligater Schreibzugriff erfolgt. Dies kann entweder zwischen dem lesenden Knoten und dem Schleifenendknoten, oder zwischen dem Schleifenanfangsknoten und dem lesenden Knoten geschehen. Ungeachtet dessen muss der lesende Zugriff im ersten Durchlauf der Schleife versorgt sein. Diese Versorgung kann auch von einem Schreibzugriff herrühren, der vor dem Schleifenanfangsknoten erfolgt. Weiterhin wirken sich konsumierende Lesezugriffe innerhalb von Schleifen natürlich auch auf die Nachfolger des konsumierend lesenden Knotens aus, welche außerhalb der Schleife liegen.

In Abbildung 3.7 sind vier unterschiedliche Konstellationen von konsumierenden Lesezugriffen und Schreibzugriffen dargestellt. Hier sei angemerkt, dass lediglich in Prozess (a) eine gültige Konstellation vorliegt. Prozess (b) ist ungültig, weil Knoten B zwar beim ersten Schleifendurchlauf noch Daten aus d lesen kann, die Daten jedoch für jeden weiteren Schleifendurchlauf ungültig sind. Sie werden vom konsumierenden Lesezugriff an Knoten C ungültig. Der Prozess in Abbildung 3.7 (c) weist gleich zwei Fehler auf: Zum einen kann die Schleife nur einmal durchlaufen werden, da sich der konsumierende Lesezugriff an Knoten C selbst die Versorgung über die Schleifenrücksprungkante nimmt. Andererseits ist der Lesezugriff an Knoten E ungültig, unabhängig davon, ob die Schleife nur ein- oder mehrmals durchlaufen wird. In Prozess (d) sind zwei geschachtelte Schleifen abgebildet. Der Lesezugriff an Knoten B ist nach einem Rücksprung über die außenliegende Schleife nicht versorgt.

3.3. Datenflussanalyse

Wir erläutern nun einführend kurz die Funktionsweise des bestehenden *WriterExists*-Algorithmus [Rei00], da dieser die Basis für den nachfolgend erweiterten Algorithmus darstellt. In den beiden darauffolgenden Abschnitten wird dann diese Erweiterung erläutert, zunächst für azyklische Kontrollflussgraphen (Abschnitt 3.3.3) und danach für Prozessgraphen mit Schleifen (Abschnitt 3.3.4). Als letztes klären wir, wie sich konsumierende Lesezugriffe auf parallele Verzweigungen auswirken.

3.3.1. Bestehender WriterExists-Algorithmus

Der `WriterExists`-Algorithmus aus [Rei00] geht davon aus, dass einmal geschriebene Daten für den gesamten weiteren Prozessverlauf zur Verfügung stehen. Um für einen Knoten n zu ermitteln, ob er ein Datenelement d obligat lesen darf, werden zunächst alle Knoten ermittelt, die in der Vorgängermenge von n liegen und d obligat schreiben.

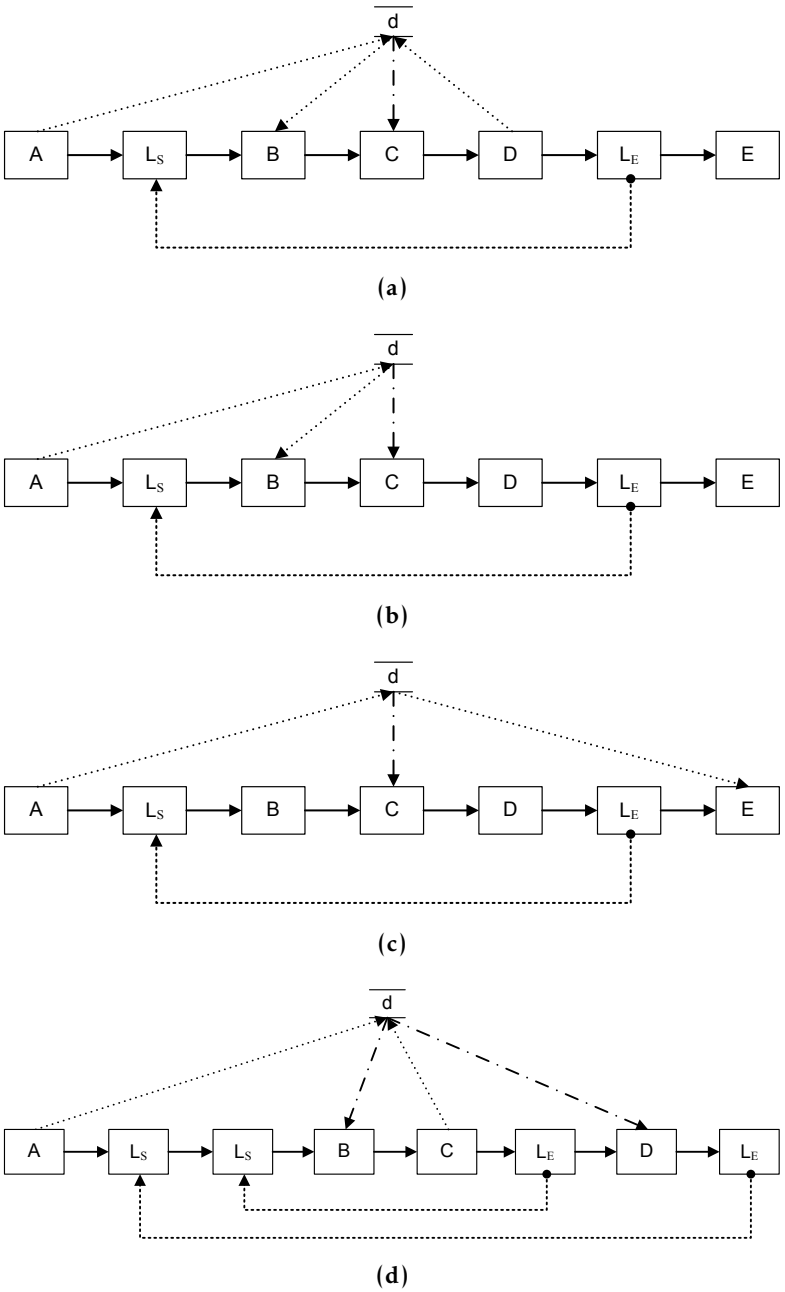
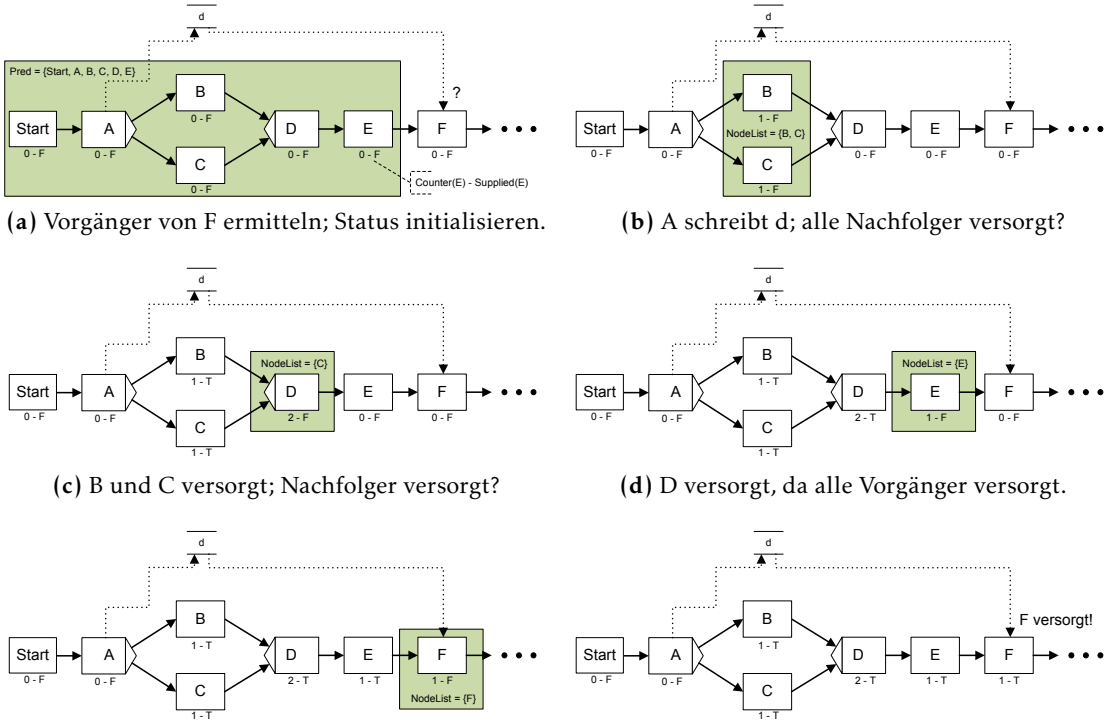


Abbildung 3.7 Konstellationen von konsumierenden Lesezugriffen und Schreibzugriffen



(a) Vorgänger von F ermitteln; Status initialisieren.

(b) A schreibt d; alle Nachfolger versorgt?

(c) B und C versorgt; Nachfolger versorgt?

(d) D versorgt, da alle Vorgänger versorgt.

(e) Weiterreichen der Versorgung.

(f) F darf d obligat lesen!

Abbildung 3.8 Einfaches Beispiel für die Funktionsweise des `WriterExists`-Algorithmus (nach [Rei00])

Im ersten Durchgang wird eine Untersuchung für alle Vorgänger bezüglich normaler Kontrollkanten (beschrieben durch c_pred*(n), s. Anhang D) durchgeführt. Dabei werden alle Nachfolgerknoten der obligat schreibenden Knoten aus o. g. Vorgängermenge als *versorgt* markiert und deren Nachfolger zur weiteren Untersuchung vorgemerkt.

Da bei bedingten Verzweigungen sichergestellt werden muss, dass die Versorgung über jeden Zweig erreicht werden kann, wird für alle Knoten ein Zähler angelegt, der erhöht wird, wenn seine Versorgung über einen noch nicht untersuchten Pfad sichergestellt werden kann. Entspricht der Zähler eines Knotens der Anzahl seiner eingehenden Kontrollkanten, wird er als versorgt markiert. Ist der Zähler niedriger, bedeutet dies, dass der Knoten noch nicht als versorgt markiert wird. Auf normale Aktivitätenknoten hat dieser Zähler keine Auswirkung, da sie lediglich eine einzige eingehende Kontrollkante besitzen. Somit können sie als versorgt markiert werden, wenn ihr direkter Vorgänger bereits als versorgt markiert ist oder *d* obligat schreibt. Auf diese Weise wird die Menge der Knoten, die *d* obligat lesen dürfen, sukzessive erweitert, bis entweder keine Knoten mehr untersucht werden können oder der gewünschte Knoten *n* als versorgt markiert wurde. Die Funktionsweise dieses Teils des `WriterExists`-Algorithmus ist in Abbildung 3.8 dargestellt.

Falls die Versorgung von n nach der Untersuchung aller Vorgänger bzgl. normaler Kontrollkanten noch nicht sichergestellt werden kann, wird die Untersuchung auf die gesamte Vorgängermenge von n (d. h. $pred^*(n)$) ausgedehnt. Dabei werden die im vorangegangenen Schritt erzeugten Markierungen weiterverwendet. Es wird nun geprüft, ob sich die Menge der versorgten Knoten durch die im Prozess verwendeten Sync-Kanten weiter ausdehnen lässt. Wieder gilt, dass die Untersuchung beendet ist, wenn entweder keine weiteren Knoten mehr als versorgt markiert werden können oder der zu untersuchende Knoten n als versorgt markiert wurde.

Da sich Schleifen nicht auf die Versorgung auswirken, müssen diese nicht gesondert untersucht werden. Nun kann eine Aussage darüber getroffen werden, ob d zum Zeitpunkt des Lesezugriffs durch n sicher mit Daten versorgt ist.

Ist der Knoten n auch nach der erweiterten Untersuchung, unter Berücksichtigung von Sync-Kanten, nicht als versorgt markiert worden, so darf er das Datenelement d nicht obligat lesen, da nicht sichergestellt werden kann, dass es zur Prozesslaufzeit mit Daten versorgt wurde. Enthält das Datenelement d beim Starten der Aktivität an Knoten n tatsächlich keinen Wert, so befindet sich diese in einem undefinierten Zustand, da sie auf die Eingabedaten aus dem Datenelement angewiesen ist.

Im Folgenden wird detailliert erläutert, welche Anpassungen an diesem Algorithmus notwendig sind, damit konsumierende Lesezugriffe von ihm berücksichtigt werden. Wir betrachten zunächst azyklische Kontrollflussgraphen mit Sync-Kanten (CFS_{sync}) und gehen danach näher auf zyklische Kontrollflussgraphen (CFS_{all}) ein. Diese gesonderte Betrachtung ist notwendig, da der ursprüngliche Algorithmus nicht durch die Verwendung von Schleifen beeinträchtigt wird und sie deswegen nicht beachtet. Schleifen wirken sich jedoch auf die Versorgung aus, wenn Datenelemente innerhalb des Schleifenkörpers konsumierend gelesen werden.

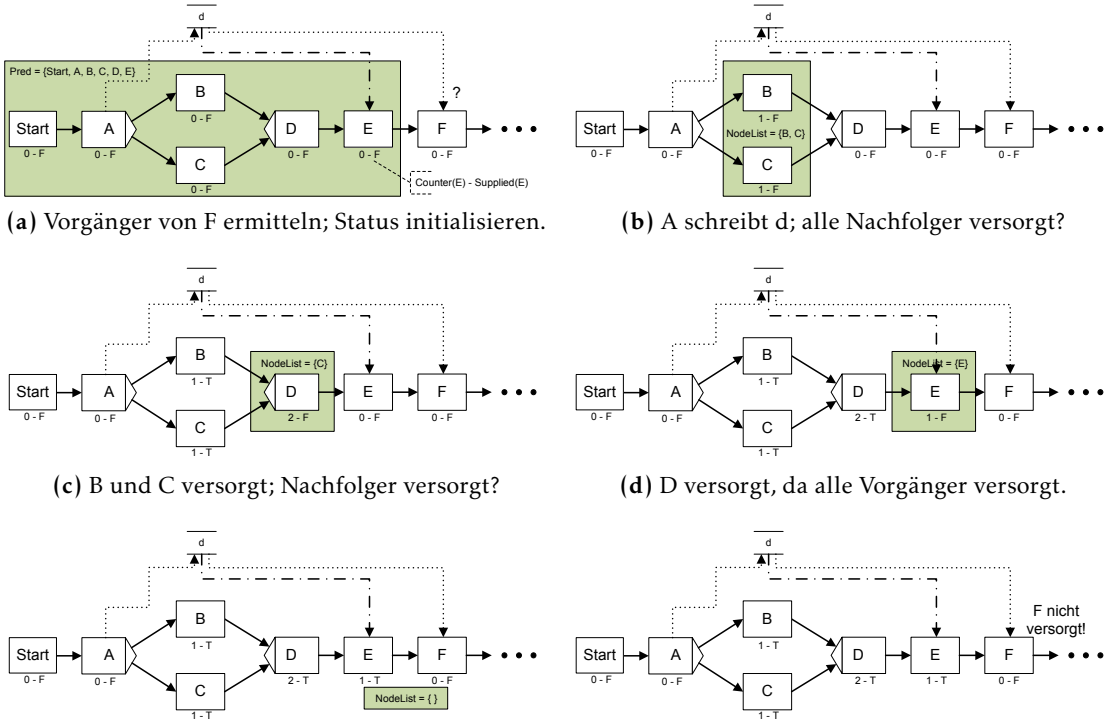
3.3.2. Erkennen konsumierender Lesezugriffe

Zunächst klären wir, wie konsumierende Lesezugriff systemseitig erkannt werden können. Dies ist notwendig, damit die anschließend erläuterte Untersuchung diese Zugriffe erkennen und geeignet darauf reagieren kann.

Ein konsumierender Lesezugriff eines Knotens n auf ein Datenelement d liegt vor, wenn es im Datenflussschema DFS eine konsumierende Datenkante gibt, deren Quelle das Datenelement d und deren Ziel ein Eingabeparameter der an n hinterlegten Aktivität ist. Diese Überprüfung führt die Funktion `ReadsConsuming` durch, welche detailliert in Anhang E (Algorithmus 1) beschrieben ist. Sie liefert `TRUE` zurück, falls eine solche Kante existiert, ansonsten `FALSE`.

3.3.3. Azyklische Kontrollflussgraphen mit Sync-Kanten

Wir verwenden nun die im vorherigen Abschnitt vorgestellte Funktion, um Nachfolgerknoten von konsumierend lesenden Knoten aus der Untersuchung des Algorithmus auszunehmen, da diese durch den konsumierenden Lesezugriff nicht mehr versorgt sind.



(a) Vorgänger von F ermitteln; Status initialisieren.

(b) A schreibt d; alle Nachfolger versorgt?

(c) B und C versorgt; Nachfolger versorgt?

(d) D versorgt, da alle Vorgänger versorgt.

(e) Versorgung durch konsum. Lesen unterbrochen!

(f) F darf d *nicht* obligat lesen!

Abbildung 3.9 Funktionsweise des `WriterExists`-Algorithmus mit Änderungen zum Erkennen konsumierender Lesezugriffe

Ein Knoten darf dann nicht weiter untersucht werden, wenn sein Vorgänger *d* konsumierend liest, es sei denn, dieser schreibt *d* wieder obligat. Der erste Teil dieser Bedingung ist leicht umzusetzen, indem alle Nachfolger von konsumierend lesenden Knoten nicht weiter untersucht werden. Der zweite Teil der Bedingung ist bereits im Algorithmus enthalten, da all diejenigen Knoten sicher untersucht werden, deren Vorgänger *d* obligat schreiben. Es reicht also aus, diejenigen Stellen im Algorithmus zu modifizieren, an denen Knoten für die weitere Untersuchung ausgewählt werden.

Dies geschieht im *WriterExists*-Algorithmus aus [Rei00] in Schritt 3. Hier werden Knoten als versorgt markiert, wenn ihre Eingangssemantik entweder ONE_OF_ONE oder ALL_OF_ALL ist oder ihr Zähler der Anzahl ihrer Eingangskontrollkanten entspricht. Wird ein Knoten n* als versorgt markiert, so werden alle seine Nachfolger zur weiteren Untersuchung vorgemerkt, sofern sie in der Vorgängermenge von n_{read} enthalten sind und nicht bereits als versorgt markiert wurden. Hier fordern wir zusätzlich, dass n* das Datenelement *d* nicht konsumierend liest und schließen so aus, dass die Versorgung über konsumierendes Lesen hinweg propagiert wird (s. Abb. 3.9e).

Dieselbe Unterbrechung fügen wir in den zweiten Teil des ursprünglichen Algorithmus ein, in dem zur Überprüfung der Versorgung neben normalen Kontrollkanten auch Sync-Kanten herangezogen werden. Dort werden in Schritt 8 ebenfalls die Nach-

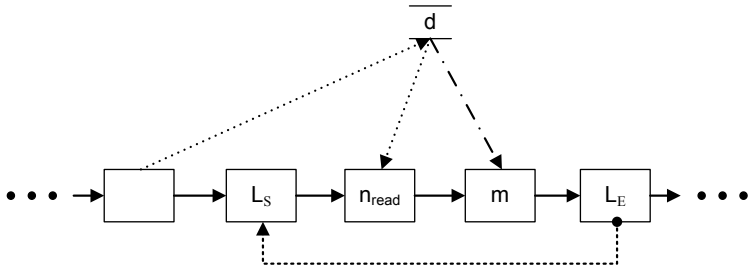


Abbildung 3.10 Konsumierender Lesezugriff in einer Schleife, der von `WriterExists` nicht erkannt wird.

folger von bereits versorgten Knoten zur weiteren Untersuchung ausgewählt. Auch hier verhindern wir die weitere Untersuchung eines Knotens, falls sein Vorgänger d konsumierend liest.

Diese Änderungen reichen aus, um auf konsumierende Lesezugriffe in einem azyklischen Kontrollflussgraphen zu reagieren und ihren Einfluss auf den Datenfluss algorithmisch zu beschreiben. Da aber durch die Einführung von konsumierenden Lesezugriffen nicht länger davon ausgegangen werden kann, dass eine Versorgung auch über eine Schleifenrücksprungkante hinweg besteht, wird im folgenden Abschnitt beschrieben, wie die Untersuchung des Prozessgraphen angepasst werden muss, damit die Untersuchung der Versorgung von in Schleifen enthaltenen Knoten korrekt abläuft. Die Funktionsweise des geänderten `WriterExists`-Algorithmus ist in Abbildung 3.9 anhand eines einfachen Prozesses (ohne Sync-Kanten und Schleifen) dargestellt.

3.3.4. Zyklische Kontrollflussgraphen mit Sync-Kanten

Liegt der zu untersuchende Knoten n_{read} innerhalb einer Schleife mit Startknoten L_S und Endknoten L_E, existiert zwischen n_{read} und L_E ein Knoten m, der das Datenelement d konsumierend liest und existiert zwischen L_S und n_{read} kein Knoten, der d obligat schreibt (s. Abb. 3.10), so reicht die im vorigen Abschnitt beschriebene Untersuchung nicht mehr aus. In diesem Fall ist es notwendig, auch Nachfolgerknoten von n_{read} zu untersuchen, die bezüglich der Schleifenrücksprungkante zu Vorgängern von n_{read} werden. Die Untersuchung wird auf die Menge $l\_pred^*(n_{read})$ ausgedehnt. Formal ist diese erweiterte Untersuchung durchzuführen, wenn Folgendes gilt (implementiert in *LoopKillsSupply*, Anhang E):

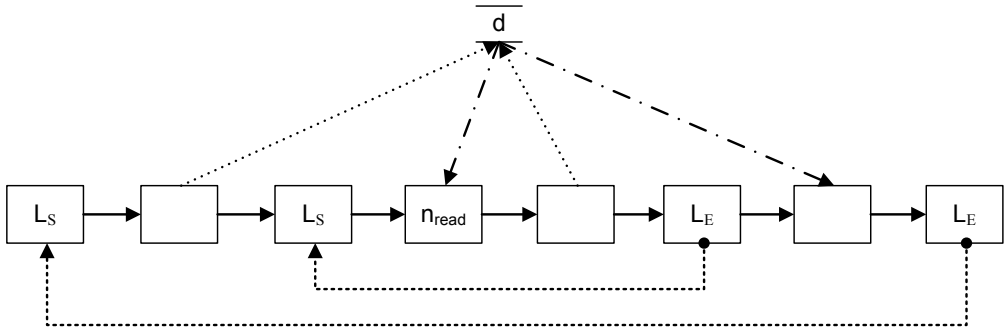


Abbildung 3.11 Konsumierende Lesezugriffe in verschachtelten Schleifen

Definition 3 (Unterbrechung der Versorgung durch Schleifen)
Sei CFS ein allgemeiner Kontrollflussgraph mit zugehörigem Datenflussschema DFS.

Die Versorgung eines Knotens n_{read} mit Daten aus einem Datenelement d *ist genau dann durch eine Schleife mit Anfangsknoten L_S und Endknoten L_E nicht mehr gewährleistet, wenn n_{read} zwischen L_S und L_E liegt und gilt:*

$$\exists x \in \{c\_succ^*(n_{read}) \cap c\_pred^*(L_E)\} \cup \{n_{read}, L_E\} \mid ReadsConsuming(DFS, x, d) \wedge$$
$$\nexists y \in \{c\_pred^*(n_{read}) \cap c\_succ^*(L_S)\} \cup \{L_S\} \mid ActivityWrites^{DFS}(y, d, MANDATORY)$$

Da n_{read} sicher ein Vorgängerknoten des Schleifenendknotens ist, reicht es aus, die Analyse statt für n, für den Schleifenendknoten der n_{read} umgebenden Schleife durchzuführen. Durch die Konstruktion des *WriterExists*-Algorithmus ist sichergestellt, dass alle Knoten auf Versorgung geprüft werden, die zwischen dem zu untersuchenden Knoten und den d schreibenden Knoten liegen.

Mit dieser Änderung alleine ist es möglich, dass der Schleifenendknoten als versorgt markiert ist, bevor n_{read} eine solche Markierung erhalten hat. Dies ist beispielsweise der Fall, wenn der direkte Vorgänger des Schleifenendknotens das Datenelement d obligat schreibt. Da dieses Ergebnis nun nicht aussagt, ob die an Knoten n_{read} hinterlegte Aktivität bei ihrem Start von einer Versorgung des untersuchten Eingabeparameters ausgehen kann, müssen wir den Algorithmus so abändern, dass er erst dann die sichere Versorgung meldet, wenn sowohl n_{read} als auch der Schleifenendknoten als versorgt markiert sind.

Diese Überprüfung führt die Funktion `LoopSupplied` (s. Anhang E, Algorithmus 4) durch. Sie deckt zudem einen Spezialfall ab: ist der Schleifenendknoten als versorgt markiert und liest er selbst das Datenelement d konsumierend, so ist die Versorgung über die Schleifenrücksprungkante nur dann gesichert, wenn er d selbst wieder obligat schreibt.

Geschachtelte Schleifen

Liegt n_{read} in mehreren verschachtelten Schleifen (s. Abb. 3.11), so muss für jede Schleife geprüft werden, ob die Bedingung aus Definition 3 auf sie zutrifft. Ist sie für mehre-

re Schleifen erfüllt, so müssen die Schleifenendknoten aller betreffenden Schleifen auf Versorgung überprüft werden.

Dem ursprünglichen `WriterExists`-Algorithmus wird ein Abschnitt vorangestellt, in dem zunächst die genannte Bedingung aus Def. 3 für alle n_{read} umgebenden Schleifen überprüft wird. Ist die Bedingung für mindestens eine n_{read} umgebende Schleife erfüllt, so wird n_{read} zusammen mit allen zu überprüfenden Schleifenendknoten in einer Menge N_{read} gesammelt. Anschließend wird der `WriterExists`-Algorithmus für den Schleifenendknoten L_E der äußersten, n umgebenden Schleife durchgeführt. Die Versorgung aller innenliegenden Schleifenendknoten, sowie die Versorgung des ursprünglich zu untersuchenden Knotens n_{read} ergeben sich nach Beendigung der Untersuchung automatisch. Die Konjunktion des Versorgungszustandes aller in N_{read} enthaltener Knoten ist das Ergebnis der gesamten Untersuchung. Damit ist n_{read} bezüglich aller umgebenden Schleifen versorgt, wenn alle in der Menge N_{read} enthaltenen Knoten versorgt sind. Die Funktion `LoopSupplied` ist bereits so konstruiert, dass sie beliebig viele Knoten überprüfen kann.

Hier sei angemerkt, dass die Komplexität des angepassten `WriterExists`-Algorithmus auch bei der Untersuchung mehrerer verschachtelter Schleifen nicht steigt. Der Grund dafür ist, dass weiterhin lediglich ein Knoten (der Endknoten der äußersten Schleife) auf Versorgung untersucht werden muss. Sowohl die Überprüfung des Einflusses der Schleifen auf die Versorgung als auch die Überprüfung der Versorgung aller in N_{read} enthaltenen Knoten, besitzen lediglich lineare Komplexität bzgl. der Anzahl der geschachtelten Schleifen.

3.3.5. Ausschluss konsumierender Zugriffe in parallelen Zweigen

In [Rei00] wird gefordert, dass Daten niemals gleichzeitig aus unterschiedlichen parallelen Zweigen geschrieben werden, da konkurrierende Schreibzugriffe in ADEPT2 nicht synchronisiert werden können. Sie können jedoch durch die Verwendung von Sync-Kanten verhindert werden. Nachdem konsumierende Lesezugriffe auch eine Manipulation der Daten darstellen (Schreiben von `NULL` in das Datenelement), fordern wir, dass neben parallelen Schreibzugriffen auch parallele konsumierende Lesezugriffe aus unterschiedlichen parallelen Zweigen ausgeschlossen sind, sofern ihre Reihenfolge nicht durch die Verwendung von Sync-Kanten eindeutig festgelegt ist.

Gleiches gilt für die Kombination von Schreibzugriffen und konsumierenden Lesezugriffen. Aus Sicht der in [Rei00] vorgestellten Strukturierungsregel *DF-2* müssen konsumierende Lesezugriffe also gleich behandelt werden wie Schreibzugriffe. Entsprechend wandeln wir diese Datenflussregel wie folgt ab:

Definition 4 (Erweiterte Strukturierungsregel DF-2a)
CFS = (N,E,D,...) sei ein allgemeiner KF-Graph und DFS das diesem Graphen zugeordnete Datenflusschema. Ferner seien $n_1, n_2 \in N, n_1 \neq n_2$ zwei Aktivitätenknoten und $d \in D$ gegeben mit

$$\begin{array}{llll} ActivityWrites^{DFS}(n_1,d,ANY) & \wedge & ActivityWrites^{DFS}(n_2,d,ANY) & \vee \\ ActivityWrites^{DFS}(n_1,d,ANY) & \wedge & ReadsConsuming^{DFS}(n_2,d,ANY) & \vee \\ ReadsConsuming^{DFS}(n_1,d,ANY) & \wedge & ReadsConsuming^{DFS}(n_2,d,ANY) & \end{array}$$

Das heißt, das Datenelement d *wird sowohl von n_1 als auch n_2 manipuliert.*

Dann dürfen die beiden Aktivitätenknoten n_1 und n_2 nicht in verschiedenen Zweigen einer parallelen Verzweigung mit AND-Join-Knoten liegen, es sei denn, ihre Ausführungsreihenfolge ist durch die Verwendung von Sync-Kanten eindeutig festgelegt. Formal:

Falls $(split, join) := BranchNodes^{CFS}(n_1, n_2) \neq (UNDEFINED, UNDEFINED)$ gilt (d. h. die Knoten n_1 und n_2 liegen in unterschiedlichen Zweigen der durch $(split, join)$ gebildeten Verzweigung), so muss ebenfalls gelten:

$$(V_{in}^{join} \neq ALL\_OF\_ALL) \vee (n_1 \in succ^*(n_2) \vee n_1 \in pred^*(n_2))$$

3.4. Auswirkungen auf unterschiedliche Datentypen

Konsumierende Lesezugriffe wirken sich unterschiedlich auf verschiedene Datentypen aus. Bei der in Abschnitt 3.3 vorgestellte Analyse sind wir davon ausgegangen, dass Datenelemente einen atomaren Datentyp haben. Auf einfache Datentypen trifft diese Atomarität zu. Daher sind für diese keine weiteren Besonderheiten zu beachten, der erweiterte Algorithmus kann direkt auf sie angewendet werden. Bei allen bisher in ADEPT2 vorhandenen Datentypen handelt es sich um einfache Datentypen mit atomaren Werten, weswegen für diese keine Detailbetrachtung des beschriebenen Algorithmus notwendig ist. Die Auswirkungen auf weitere in dieser Arbeit vorgestellte Datentypen werden im jeweiligen Kapitel erläutert.

3.5. Zusammenfassung

Das Konzept des *konsumierenden Lesens* bietet dem Prozessmodellierer umfassende Unterstützung bei der korrekten Umsetzung eines Sachverhaltes, dessen Analyse bisher nicht von ADEPT2 durchgeführt werden konnte. Durch die Einführung der neuen Datenkante für konsumierende Lesezugriffe und der Erweiterung des WriterExists-Algorithmus sind sowohl der Modellierer als auch das System in der Lage, die Auswirkungen von konsumierenden Datenzugriffen korrekt zu erfassen und geeignet darauf zu reagieren.

4. Strukturierte und komplexe Datentypen

Ziel der in diesem Kapitel vorgestellten Konzepte ist es, ADEPT2 die Struktur von Daten bekannt zu machen, damit möglichst viele Daten möglichst übersichtlich und einfach direkt im PMS zur Verfügung stehen. Neben der reinen Strukturierung von Daten, zu der auch die Einschränkung des Wertebereichs von Datentypen gehört, werden wir die Möglichkeit der Schachtelung und der Vererbung von Datentypen diskutieren.

4.1. Motivation

Wir erläutern zunächst, wie die Struktur von Daten derzeit in ADEPT2 abgebildet werden kann. Anschließend erläutern wir verschiedene Varianten der Darstellung von Strukturen in PMS. Dabei dient uns eine *Adresse* als Beispiel. Anhand ihrer Struktur und ihrer Bestandteile lassen sich die verschiedenen Konzepte gut erläutern.

4.1.1. Derzeitige Möglichkeiten in ADEPT2

Mit den in Kapitel 2 vorgestellten Datentypen bietet ADEPT2 dem Benutzer derzeit zwei Möglichkeiten, strukturierte Daten in Prozesse zu integrieren. Die Strukturen der Daten können in ihre atomaren Bestandteile zerlegt werden, wonach jeder Teil der Struktur durch ein Datenelement mit atomarem Datentyp abgebildet wird. Alternativ können die Daten samt ihrer Struktur in einem UDT gespeichert werden. Offensichtliche Nachteile der Variante mit atomaren Datentypen sind einerseits die hohe Komplexität bei stark strukturierten Daten, andererseits die Unübersichtlichkeit und aufwändige Modellierung von Prozessen mit sehr vielen Datenelementen. Gegen die Repräsentation der Daten in UDTs spricht vor allem die Tatsache, dass das PMS keinen Einblick in und damit keine Kontrolle über die Struktur der Daten hat.

4.1.2. Zusammengesetzte Datentypen

Datentypen, deren Werte nicht atomar sind, nennt man *zusammengesetzte Datentypen* (alternativ: *Verbund*, *Struct* oder *Record*). Sie besitzen in der einfachsten Form ein oder mehrere Felder, deren Datentypen atomar sind. In Beispiel 3 ist eine Adresse als zusammengesetzter Datentyp dargestellt.

Beispiel 3 (Adresse als strukturierter Datentyp) *Eine* Adresse *besteht zumeist aus den Bestandteilen* Name, Straße, Hausnummer, PLZ, Ort *und* Land. *Ein Datentyp, der eine Adresse aufnehmen kann, zeigt Abbildung 4.1. Der Einfachheit wegen gehen wir davon aus, das Hausnummern und Postleitzahlen nur aus Ziffern bestehen.*

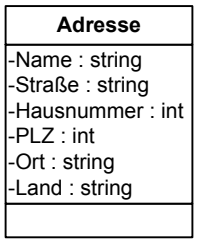


Abbildung 4.1 Einfache Darstellung des strukturierten Datentyps *Adresse*

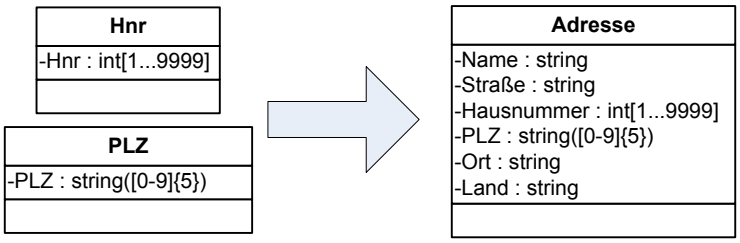


Abbildung 4.2 Adresse mit genauerer Beschreibung von Hausnummer und PLZ

4.1.3. Einschränkung einfacher Datentypen

Neben der Gruppierung von Datentypen zu einem neuen Datentyp ist auch die Einschränkung einfacher Datentypen eine wichtige Maßnahme, um dem System zusätzliche Informationen über die zu verarbeitenden Daten zur Verfügung zu stellen. Dabei wird der Wertebereich durch die Angabe von Regeln eingeschränkt. So sind in manchen Fällen nur Zahlen in einem bestimmten Bereich zulässig oder eine Zeichenkette muss eine bestimmte Form haben. In Beispiel 4 erweitern wir die Adresse um zusätzliche Informationen zu Hausnummer und Postleitzahl.

Beispiel 4 (Adresse mit eingeschränktem Datentyp) *Im Falle von Adressen können eingeschränkte Datentypen dazu verwendet werden, die Zahlen für* Hausnummer *und* PLZ *genauer zu spezifizieren. Für dieses Beispiel beschränken wir uns auf deutsche Postleitzahlen (fünfstellig, nur Ziffern), sowie auf Hausnummern, die aus bis zu vier Ziffern bestehen. Wir integrieren diese neuen Datentypen in die Struktur der Adresse aus Beispiel 3. Dabei verwenden wir für die Postleitzahl eine Zeichenkette statt einer Ganzzahl, da Letztgenannte keine führende Null darstellen kann.*

Eingeschränkte Datentypen können dabei nicht nur als eigenständiger Datentyp verwendet werden, sondern auch als Datentyp für ein Feld eines zusammengesetzten Datentyps (s. Abb. 4.2).

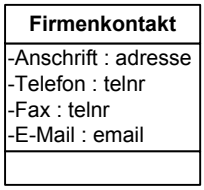


Abbildung 4.3 Datentyp *Firmenkontakt* mit Feld vom Typ *Adresse*

4.1.4. Komplexe Datentypen

Weiter ist es möglich, dass strukturierte Datentypen wieder Felder enthalten, deren Datentyp selbst strukturiert ist. Mit dieser *Schachtelung* lassen sich nahezu beliebig komplexe Datentypen bilden.

Beispiel 5 (Schachtelung strukturierter Datentypen) *Wir verwenden die in Beispiel 4 erweiterte Adresse um einen neuen, komplexen Datentyp namens* Firmenkontakt *zu realisieren. Dieser Firmenkontakt besteht aus einer Anschrift vom Typ* Adresse*, sowie aus den drei weiteren Feldern* Telefon, Fax *und* E-Mail*. Auf die Datentypen der drei letztgenannten Felder gehen wir nicht genauer ein, weisen jedoch darauf hin, dass es in der Praxis sinnvoll ist, diese mit entsprechend eingeschränkten Zeichenketten darzustellen.*

4.1.5. Vererbung von Datentypen

Komplexe Datentypen bieten durch Schachtelung bereits die Möglichkeit, Datentypen hierarchisch anzuordnen. Eine weitere Möglichkeit, eine hierarchische Struktur zu ermöglichen, ist die *Vererbung* (s. Abschnitt 2.4). Dank der „ist-ein“-Beziehung, welche durch Vererbung entsteht, ergibt sich für die abgeleiteten Datentypen eine *Polymorphie* [CW85], die es erlaubt, die Datentypen an Stellen im System zu verwenden, an denen ihr übergeordneter Datentyp verlangt wird.

Beispiel 6 (Vererbung von Datentypen) *Wenden wir das Konzept der Vererbung auf den Firmenkontakt aus Beispiel 5 an, so erhalten wir statt eines Feldes vom Typ* Adresse *einen neuen Datentyp, der zusätzlich zu seinen eigenen Feldern auch diejenigen, des Typs* Adresse *besitzt (s. Abb. 4.4).*

4.1.6. Probleme abstrakter Datentypen

Alle Systeme, die unterschiedliche Anwendungen integrieren, haben das Problem, dass für die ausgetauschten Daten ein einheitliches Modell existieren muss. Ist diese Aufgabe für häufig verwendete Datentypen, wie Zeichenketten oder Ganzzahlen, noch einfach zu lösen, gestaltet es sich für abstrakte Datentypen um ein Vielfaches schwieriger. Abstrakte Datentypen können nicht direkt zur Speicherung von Daten verwendet werden. Stattdessen müssen erst Ausprägungen dieser Datentypen angelegt werden. In den vorangehenden Beispielen haben wir gesehen, wie solche Ausprägungen aussehen

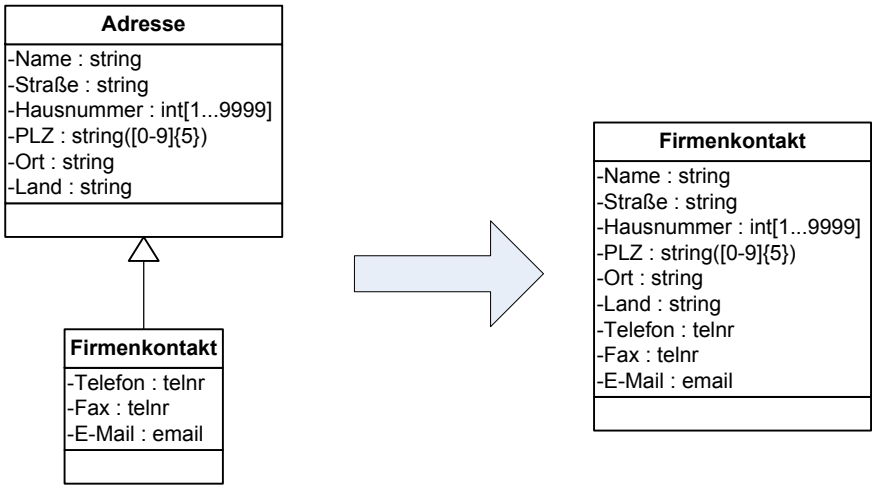


Abbildung 4.4 Datentyp *Firmenkontakt* mit geerbten Feldern von *Adresse*

können. Da die Ausprägungen abstrakter Datentypen vom Anwender angelegt und verändert werden können, ist es schwierig, diese den gekoppelten Anwendungen bekannt zu machen. Für ADEPT2 bedeutet dies, dass zu jedem Parameter einer Aktivität ein passender Datentyp im System vorhanden sein muss.

Dieses Problem tritt auch bei konkreten (also nicht abstrakten) Datentypen auf, ist dort jedoch einfacher zu lösen: Da die Datentypen einheitlich in jeder Installation des Systems vorhanden sind, können sie im Voraus definiert und in andere Systeme integriert werden. So können sich zwei verbundene Systeme auf die vordefinierten Datentypen verlassen und müssen nicht damit rechnen, unbekannte, vom Benutzer definierte Datentypen vorzufinden.

Um jedoch möglichst viele Anwendungssysteme integrieren zu können, muss eine gemeinsame Basis von Datentypen ausgehandelt werden. Diese Datentypen sind jeweils mit einem eindeutigen Bezeichner versehen. So darf es nicht passieren, dass eine `Adresse` bei Anwender *A* eine andere Bedeutung hat, als im System von Anwender *B*. Wäre dies der Fall, so könnte sich eine eingebundene Anwendung nicht auf die ihr bekannten strukturellen Informationen verlassen.

Eine Möglichkeit dieses Problem zu lösen ist, dass Anwendungen ihre eigenen Datentypen mit in das System einbringen. Wenn die Datentypen der gekoppelten Anwendungen dem System bekannt sind, ist es möglich Konverter zu entwickeln, welche die Transformation der Daten vornehmen. Diese Konverter können dann bspw. als Vorschaltdienst einer Aktivität [Rei00] verwendet werden.

Das Thema der automatischen Anwendungsintegration ist bereits Gegenstand verschiedener Forschungsarbeiten. Wir sehen es als gegeben an, dass Datentypen eindeutige Bezeichner besitzen, mit denen es möglich ist, eine gemeinsame Datenbasis für alle integrierten Anwendungen zu verwenden.

4.2. Realisierung

Wir klären nun, welche Möglichkeiten dem Benutzer zur Verfügung gestellt werden müssen, damit er strukturierte Datentypen verwenden kann. Dazu erläutern wir zunächst, wie die Struktur der Daten beschrieben wird und erklären anschließend, wie Einschränkungen auf Datentypen vorgenommen werden können. Zuletzt werden die Besonderheiten beschrieben, welche für die Schachtelung strukturierter Datentypen notwendig sind.

4.2.1. Definieren von strukturierten Datentypen

Ein strukturierter Datentyp wird mit einem Namen versehen, der innerhalb des PMS eindeutig sein muss. Weiter muss für einen strukturierten Datentyp mindestens ein Feld definiert werden. Es ist keine Obergrenze für die Anzahl der Felder eines strukturierten Datentyps festgelegt.

Jedes Feld besitzt einen Namen, welcher im Kontext des strukturierten Datentyps eindeutig ist und einen Datentyp. Dabei kann es sich um jeden bereits im System definierten Datentyp handeln. Insbesondere kann der Datentyp eines Feldes selbst wieder strukturiert sein. Im Gegensatz zur Beschreibung von Datenstrukturen mit XML-Schema ist es nicht möglich, die Häufigkeit anzugeben mit welcher der Wert eines Feldes aufzutreten hat. Jedes Feld kann genau einen (möglicherweise wieder strukturierten) Wert enthalten. Ist es notwendig, dass ein Feld mehrere Werte des selben Typs aufnehmen kann, so muss ein Feld mit einem listenwertigen Datentyp verwendet werden.

4.2.2. Einschränkung von Datentypen

Im Folgenden beschäftigen wir uns mit dem Anlegen eingeschränkter Datentypen. Wie auch strukturierte Datentypen, erhalten eingeschränkte Datentypen einen systemweit eindeutigen Namen. Weiter wird ein *Basisdatentyp* festgelegt, auf welchen sich die Einschränkung bezieht. Dabei unterscheidet sich die Einschränkung von Zeichenketten und die Einschränkung von Zahlen hinsichtlich der Art und Weise, sowie hinsichtlich der Komplexität der Einschränkung. Wir beschreiben zunächst, wie Zeichenketten mit regulären Ausdrücken [Sch00, Sch05] eingeschränkt werden können und anschließend, welche Möglichkeiten bestehen, den Wertebereich von Zahlen einzuschränken.

Einschränkung von Zeichenketten mit regulären Ausdrücken

Zur Einschränkung von Zeichenketten verwenden wir reguläre Ausdrücke. Damit können Zeichenketten beschrieben werden, deren Struktur im Voraus bekannt ist. Eine deutsche Postleitzahl ließe sich etwa mit „`(D-)?(0[1-9]|[1-9][0-9])[0-9]{3}`“ beschreiben. Dabei wird berücksichtigt, dass im Ausland der Postleitzahl ein führendes „D-“ vorangestellt werden kann. Weiter sind Postleitzahlen, die mit „00“ beginnen in Deutschland unzulässig.

Neben der manuellen Eingabe von regulären Ausdrücken ist es wünschenswert, den Anwender mit geeigneten Werkzeugen zur Generierung von regulären Ausdrücken zu unterstützen. So können etwa häufig verwendete reguläre Teilausdrücke über eine graphische Benutzeroberfläche zu einem gesamten regulären Ausdruck verbunden werden. Darüber hinaus ist es sinnvoll, eine Möglichkeit zu bieten, gegebene Zeichenketten auf ihre Gültigkeit bezüglich des eingegebenen regulären Ausdrucks hin zu überprüfen.

Zur Laufzeit wird überprüft, ob der Wert, welcher an ein Datenelement von eingeschränktem Typ übergeben wird, den für ihn festgelegten Einschränkungen entspricht. Im Falle einer Inkompatibilität muss der Benutzer mit einem Laufzeitfehler darauf hingewiesen werden, dass die übergebenen Daten nicht den vorgegebenen Einschränkungen entsprechen. In einigen Fällen ist es sinnvoll, dass die Überprüfung nicht erst bei der Übergabe der Daten in das Datenelement, sondern bereits bei der Eingabe der Daten stattfindet, etwa beim Ausfüllen eines Formulars durch den Benutzer. Diese frühe Überprüfung der Werte verhindert Laufzeitfehler, da beim Beenden einer Aktivität nur Werte an ein Datenelement weitergegeben werden, die zuvor vom Benutzer korrekt eingegeben wurden.

Einschränkung von Zahlen

Neben Zeichenketten ist es möglich, Ganz- und Fließkommazahlen in ihrem Wertebereich einzuschränken. Hierzu kann der Benutzer eine untere und eine obere Grenze festlegen. Für Fließkommazahlen ist darüber hinaus eine Beschränkung der Nachkommastellen sinnvoll.

Wie auch bei eingeschränkten Zeichenketten führt das Nichtbeachten der Beschränkung beim Schreiben von Datenelementen zu einem Laufzeitfehler. Auch hier ist die Auswertung der Beschränkung vor dem Schreiben der Daten möglich, um Laufzeitfehler zu vermeiden.

Eine automatische Anpassung der geschriebenen Werte an die Einschränkungen ist nicht vorgesehen, da eine implizite Konvertierung unbemerkt zu Fehlern führen kann. Somit obliegt die Implementierung der Anpassung dem Aktivitätenentwickler. Dieser hat so die Möglichkeit, beispielsweise das Rundungsverhalten für Fließkommazahlen anzupassen, für welche die Anzahl der Nachkommastellen beschränkt ist.

4.2.3. Schachtelung von strukturierten Datentypen

Wie eingangs erwähnt können strukturierte Datentypen verschachtelt werden, wodurch komplexe Datentypen entstehen. Auch durch Vererbung von Datentypen [WS07, HM05, Sch05, Mey96] ist die Gestaltung komplexer Datentypen möglich. Das Anbieten des Konzeptes der Vererbung steigert jedoch die Komplexität der Datentypverwaltung für den Benutzer in einem Maße, welches ihrem Nutzen nicht gerecht wird. Die Erweiterung von Datentypen um zusätzliche Felder ist gleichwertig auch durch Verschachtelung erreichbar. Die wenigen Fälle, in denen tatsächlich eine „Ist“-Beziehung zwischen Datentypen besteht, können durch das Kopieren eines bestehenden Datentyps in einen

neuen realisiert werden. Somit ist auch das Entfernen von Feldern oder das Überschreiben von Feldern mit einem anderen Datentyp möglich. Wir verzichten daher darauf, Vererbung von Datentypen anzubieten und beschränken uns darauf, dem Benutzer die Möglichkeit zu geben, komplexe Datentypen durch Schachtelung zu erstellen.

Vermeidung von Zyklen

Eine Zyklenbildung ist bei der Definition komplexer Datentypen untersagt. Dies erklärt sich dadurch, dass sich ein Datentyp A mit einem Feld vom Typ B, der wieder ein Feld vom Typ A enthält, zu einer unendlich langen Liste entwickeln würde. Als Gegenargument kann angeführt werden, dass nicht alle Felder einer zyklischen Verkettung ausgefüllt werden müssen. Dieses ist spätestens dann hinfällig, wenn alle Felder auf Versorgung mit Daten überprüft werden sollen (s. Abschnitt 4.3). Eine unendliche Liste würde zwangsläufig zu einer Analyse mit unendlicher Laufzeit führen. Wir fordern daher für komplexe Datentypen, dass sie sich nicht gegenseitig enthalten dürfen. Formal:

Definition 5
Seien A und B zwei strukturierte Datentypen. Bezeichne $A \subseteq B$, dass B ein Feld vom Typ A besitzt (auch transitiv). Für die Beziehung von A und B gilt:

$$A \subseteq B \Rightarrow B \not\subseteq A \text{ und } B \subseteq A \Rightarrow A \not\subseteq B$$

4.2.4. Umsetzung auf bestehende Datenkonstrukte

Komplexe Datentypen wirken sich auf zwei Benutzergruppen des PMS aus. Wir erläutern zunächst, wie der Prozessmodellierer komplexe Datentypen dargestellt bekommt und gehen anschließend auf die technische Umsetzung ein, welche für die Implementierer von Aktivitäten von Bedeutung ist.

Sicht des Modellierers auf komplexe Datentypen

Für den Prozessmodellierer erscheint ein Datenelement von komplexem Typ wie eines von atomarem Typ, mit dem Unterschied, dass es nicht nur als Ganzes, sondern auch in Teilen mit Parametern einer Aktivität verknüpft werden kann. Da neben dem Zugriff auf das gesamte Datenelement auch der Zugriff auf einzelne Felder möglich ist, können diese ebenfalls mit Parametern verknüpft werden. Es ist damit zu rechnen, dass durch die Verfügbarkeit komplexer Datentypen die Anzahl der Datenelemente/Datenfelder in einem Prozess sprunghaft ansteigt. Daher ist auf darauf zu achten, dass eine geeignete Präsentation (Baumansicht, hierarchisches Auswahlmenü, Auswahlliste mit Filterfunktion, etc.) für den Prozessmodellierer gewählt wird.

Da mehrere Felder eines Datenelementes mit derselben Aktivität verknüpft werden können, muss dieser Umstand visuell erkennbar gemacht werden. Dafür führen wir eine neue Visualisierung für Datenzugriffskanten ein. Diese bedarf keiner formalen

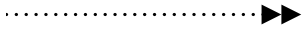


Abbildung 4.5 Datenkanten, deren Quelle und Ziel identisch sind, werden zu einer Datenkante mit Doppelpfeil am Ende zusammengefasst.

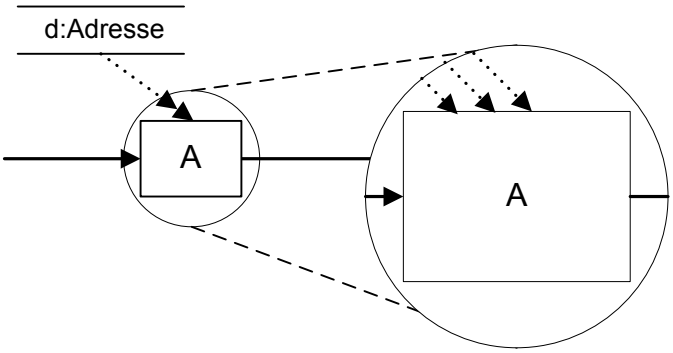


Abbildung 4.6 Systeminterne Sicht auf ein mit mehreren Datenkanten verknüpftes Datenelement

Definition, da sie keinen Modellaspekt darstellt, sondern lediglich mehrere, sich überlagernde Datenzugriffskanten zusammenfasst. Das Vorhandensein mehrerer, sich überlagernder Datenkanten wird durch eine einzelne Datenkante mit einem Doppelpfeil am Ende dargestellt (s. Abb. 4.5). Systemseitig wird weiterhin für jedes verbundene Feld eine Datenkante verwendet. Die Verwendung einer solchen *aggregierten* Datenkante und die Sicht des Systems sind in Abbildung 4.6 dargestellt.

Sicht des Implementierers auf komplexe Datentypen

Die zweite Benutzergruppe, die mit komplexen Datentypen in Berührung kommt, ist die der Aktivitätenentwickler. Die bestehende Schnittstelle für den Zugriff auf Parameter aus dem Datenkontext von ADEPT2 muss so erweitert werden, dass der Zugriff auf die einzelnen Datenfelder eines Aktivitätenparameters von komplexem Typ möglich ist.

Da der Punkt (.) als Verkettungszeichen für den Aufruf von Methoden und den Zugriff auf Objektattribute gebräuchlich ist [Sch05], verwenden auch wir diesen als Trennzeichen in Parametern von strukturiertem Datentyp. Der Zugriff auf einzelne Felder eines Aktivitätenparameters erfolgt über eine Zeichenkette, bestehend aus dem Parameternamen gefolgt von einem Punkt, auf den der Name des Feldes folgt. Da komplexe Datentypen hierarchisch geschachtelt sein können, darf der Zugriffspfad mehrere Punkte enthalten, welche jeweils den Zugriff auf eine Schachtelungsebene darstellen. Auf das Feld *Postleitzahl* des Firmenkontakts aus Beispiel 5 kann folglich über die Zeichenkette `Firmenkontakt.Anschrift.PLZ` zugegriffen werden.

Da der Punkt als Trennzeichen für Felder verwendet wird, dürfen Feldnamen keine Punkte enthalten. Dadurch werden Mehrdeutigkeiten eines Parameternamens vermie-

den. Um weitere Komplikationen zu vermeiden, schließen wir sämtliche Sonderzeichen aus dem Zeichensatz für Feldnamen aus.

Aus Sicht des Aktivitätenentwicklers stellen Parameter mit strukturiertem Datentyp eine Zusammenfassung von Parametern mit atomarem Datentyp dar. Daher besitzt ein Aktivitätenparameter mit strukturiertem Datentyp selbst keinen Wert, sondern fasst die Werte seiner Felder logisch zusammen. Ein Zugriff auf einen Aktivitätenparameter mit strukturiertem Datentyp oder der Zugriff auf ein Feld, dessen Datentyp strukturiert ist, ist daher nicht möglich. Der Zugriff erfolgt immer auf diejenigen Felder, die einen atomaren Datentyp besitzen. Stellt man sich den strukturierten Datentyp als Baum vor, so erfolgt der Zugriff immer nur auf die Blätter, aber niemals auf die Wurzel oder auf Knoten, die Kinder besitzen.

Diese Restriktion begründet sich dadurch, dass nicht eindeutig festgelegt werden kann, wie strukturierte Daten an den Aktivitätenentwickler übergeben werden. Im Falle objektorientierter Programmierung der Aktivität bietet es sich an, die Daten als eigenes Objekt zu übergeben. Dafür ist jedoch eine entsprechende Klasse für jeden strukturierten Datentyp notwendig. Alternativ können die Daten in einem generischen Objekt repräsentiert werden. Dieses gleicht dann aber der Schnittstelle für den Zugriff auf Aktivitätenparameter, weswegen sich die Frage nach dem Sinn dieser Repräsentation stellt.

Eine weitere Möglichkeit stellt die Umwandlung der strukturierten Daten in ein XML-Dokument dar. Dabei werden alle Felder des strukturierten Datentyps in einen entsprechenden (komplexen) XML-Datentyp umgewandelt. Daraus resultiert, dass für jeden strukturierten Datentyp ein entsprechendes XML-Schema erzeugt werden muss. Die so übermittelten Daten werden anschließend von der Aktivität selbst interpretiert. Ist dies gewünscht, können die Daten auch in einem UDT gespeichert werden, wodurch der Aufwand für die Transformation entfällt. Daher wird auch diese Art der Repräsentation strukturierter Datentypen nicht unterstützz.

Wie jeder Parameter von atomarem Typ auch, bieten Parameter von strukturiertem Typ die Möglichkeit zu überprüfen, ob der gesamte Parameter `NULL` ist. Weiter ist diese Überprüfung auch für jedes Feld eines Parameters von strukturiertem Typ möglich.

4.3. Datenflussanalyse

Im Sinne von [Rei00] ist der Typ eines Datenelementes unerheblich für die Analyse der Versorgung des Datenelementes mit Daten. Diese Annahme gilt auch für strukturierte Datentypen, solange lediglich die Analyse der vollständigen Versorgung gewünscht ist. Wir beschreiben im Folgenden die Analysemöglichkeiten, die sich durch die Strukturierung der Daten ergeben.

4.3.1. Versorgung einzelner Felder

Um die Vorteile der systemseitig bekannten Struktur der Daten zu nutzen, muss die Analyse auf die atomaren Bestandteile der strukturierten Datentypen angewendet wer-

den. Da jede Komponente eines strukturierten Datentyps wie ein eigenständiges Datenelement angesehen werden kann, kann auch der *WriterExists*-Algorithmus[1] auf einzelne Felder angewendet werden. Somit kann bei einem obligaten Lesezugriff auf ein einzelnes Feld eines strukturierten Datentyps untersucht werden, ob dieses Feld mit Daten versorgt ist. Die Versorgung von einzelnen Feldern hängt dann nicht mehr zwingend von der Versorgung des gesamten Datenelementes ab.

Teilweise obligater Zugriff auf strukturierte Datentypen

Greift eine Aktivität lesend oder schreibend auf ein gesamtes Datenelement mit strukturiertem Datentyp zu, so kann es vorkommen, dass nur eine Teilmenge der Felder obligat, der Rest aber optional gelesen bzw. geschrieben wird. Dieser Sachverhalt kann dargestellt werden, indem für jedes obligat gelesene (geschriebene) Feld ein eigener Parameter zur Verfügung gestellt wird. Um die Handhabung zu vereinfachen, bieten wir für Parameter mit strukturiertem Datentyp die Möglichkeit anzugeben, auf welche Felder obligat und auf welche optional zugegriffen wird. Die Versorgung der einzelnen Felder ist dann äquivalent zur Umsetzungsvariante mit einem Parameter pro Feld zu analysieren.

4.3.2. Versorgung gesamter Datentypen

Für den Zusammenhang der Versorgung des gesamten Datenelementes und der einzelnen Felder können wir Folgendes festhalten:

Definition 6 (Versorgung von Datenelementen mit strukturiertem Datentyp)
Ist ein Datenelement d *mit strukturiertem Typ für den obligaten Lesezugriff eines Knotens* n_{read} *versorgt, so gilt diese Versorgung auch für alle seine Felder* f. *Die Umkehrung gilt ebenfalls. Formal:*

$$WriterExists(CFS, DFS, n_{read}, d) \Leftrightarrow \forall f \in d : WriterExists(CFS, DFS, n_{read}, k)$$

Für die Analyse des Datenflusses bedeutet dies, dass für die Versorgung eines Datenelementes nicht nur Schreiber des gesamten Datenelementes in Frage kommen, sondern dass auch die teilweise Versorgung eines Datenelementes in der Summe zu einer Versorgung des gesamten Datenelementes führt.

Daher muss der WriterExists-Algorithmus, im Falle dass er für das gesamte Datenelement fehl schlägt, nochmals für die einzelnen Komponenten des Datenelementes angewendet werden [Bla96]. Handelt es sich um ein Datenelement mit komplexem Typ, so muss die Analyse eventuell rekursiv für alle strukturierten Komponenten durchgeführt werden.

Wird die Versorgung nicht für ein Datenelement als Ganzes sichergestellt, sondern entsteht sie durch die Versorgung der einzelnen Felder durch verschiedene Aktivitäten, so können die enthaltenen Werte inkonsistent sein. So ist es etwa möglich, dass

[1] Dies gilt sowohl für die Variante aus [Rei00] als auch für den abgewandelten Algorithmus, der in Kapitel 3 vorgestellt wurde.

Firmenname und Anschrift eines Adressdatensatzes von unterschiedlichen Aktivitäten geschrieben werden. Damit ist zwar der gesamte Adressdatensatz versorgt, jedoch kann die semantische Zusammengehörigkeit von Firmenname und Anschrift nicht vom System gewährleistet werden. Der Prozessmodellierer ist daher auf die möglicherweise inkonsistenten Daten hinzuweisen.

4.3.3. Parallele Schreibzugriffe

In [Rei00] wird festgelegt, dass keine Schreibzugriffe aus parallelen Zweigen auf ein Datenelement erfolgen dürfen, sofern ihre Reihenfolge nicht durch Sync-Kanten eindeutig festgelegt ist. Selbiges gilt auch für Schreibzugriffe auf ein Datenelement mit strukturiertem Datentyp. Da jedoch jedes Feld eines solchen Datentyps als eigenes Datenelement angesehen werden kann, ist ein Schreibzugriff aus parallelen Zweigen auf unterschiedliche Felder eines Datenelementes mit strukturiertem Datentyp gestattet. Ausgeschlossen sind dagegen parallele Schreibzugriffe auf ein einzelnes Feld oder der Schreibzugriff auf das gesamte Datenelement parallel zu einem Schreibzugriff auf ein einzelnes Feld. Dieser Sachverhalt ist in Definition 7 dargestellt.

Definition 7 (Erweiterte Strukturierungsregel DF-2b)
CFS = (N,E,D,...) sei ein allgemeiner KF-Graph und DFS das diesem Graphen zugeordnete Datenflussschema. Ferner seien $n_1, n_2 \in N, n_1 \neq n_2$ zwei Aktivitätenknoten und $d \in D$ von strukturiertem Typ mit der Menge aller Felder F, sowie $f \in F$ gegeben, mit

$$\begin{array}{lcllcl} ActivityWrites^{DFS}(n_1,d,ANY) & \wedge & ActivityWrites^{DFS}(n_2,d,ANY) & \vee \\ ActivityWrites^{DFS}(n_1,d,ANY) & \wedge & ActivityWrites^{DFS}(n_2,f,ANY) & \vee \\ ActivityWrites^{DFS}(n_1,f,ANY) & \wedge & ActivityWrites^{DFS}(n_2,f,ANY) & \end{array}$$

Dann dürfen die beiden Aktivitätenknoten n_1 und n_2 nicht in verschiedenen Zweigen einer parallelen Verzweigung mit AND-Join-Knoten liegen, es sei denn, ihre Ausführungsreihenfolge ist durch die Verwendung von Sync-Kanten eindeutig festgelegt. Formal:

Falls $(split, join) := BranchNodes^{CFS}(n_1, n_2) \neq (UNDEFINED, UNDEFINED)$ gilt (d. h. die Knoten (n_1 und (n_2 liegen in unterschiedlichen Zweigen der durch $(split, join)$ gebildeten Verzweigung), so muss ebenfalls gelten:

$$(V_{in}^{join} \neq ALL\_OF\_ALL) \vee (n_1 \in succ^*(n_2) \vee n_1 \in pred^*(n_2))$$

4.3.4. Konsumierendes Lesen

Da bei strukturierten Datentypen der Zugriff sowohl auf das gesamte Datenelement als auch auf einzelne Komponenten erfolgen kann, wirkt sich ein konsumierender Lesezugriff teilweise nicht nur auf den direkt gelesenen Wert, sondern auch auf den gesamten Datentyp aus, in dem der konsumierend gelesene Wert enthalten ist.

Zugriff auf gesamtes Datenelement

Wird ein Datenelement mit strukturiertem Datentyp als Ganzes konsumierend gelesen, so wird auch der Inhalt der einzelnen Felder des Datenelementes konsumiert. Nachfolgende obligate Lesezugriffe sind also weder auf das gesamte Datenelement, noch auf einzelne Felder gestattet, bevor nicht ein erneuter Schreibzugriff erfolgt.

Ein nachfolgender Schreibzugriff kann dann entweder einzelne Felder oder das gesamte Datenelement (wieder) versorgen. Die erneute Versorgung des gesamten Datenelementes hat auch die Versorgung der einzelnen Felder zur Folge, die Umkehrung gilt nur, wenn *alle* Felder wieder obligat geschrieben werden.

Zugriff auf einzelne Felder

Der konsumierende Lesezugriff auf einzelne Felder eines Datenelementes von strukturiertem Typ hat nicht nur zur Folge, dass dieses eine Feld nicht mehr versorgt ist, sondern wirkt sich direkt auf die Versorgung des gesamten Datenelementes aus. Nachdem das Feld konsumierend gelesen wurde, darf weder dieses eine Feld, noch das gesamte Datenelement obligat gelesen werden, bevor nicht wieder ein obligater Schreibzugriff erfolgt ist.

Die Funktion *ReadsConsuming(DFS,n,d)* aus Anhang E muss daher so erweitert werden, dass sie nicht nur den konsumierenden Zugriff auf das gesamte Datenelement d erkennt, sondern auch konsumierende Zugriffe auf einzelne Felder.

4.3.5. Inkonsistenzen durch partielle Schreibzugriffe

Jede Aktivität, die ein Datenelement mit strukturiertem Datentyp schreibt, hat die Möglichkeit, nur Teile der Struktur mit Werten zu versorgen. Dadurch können die in diesem Datenelement enthaltenen Werte inkonsistent werden. Diese Inkonsistenzen treten auf, wenn eine Aktivität *B* weniger Felder mit Daten versorgt, als von einer vorausgehenden Aktivität *A* bereits mit Daten versorgt worden sind. Die von *B* nicht oder nur optional geschriebenen Felder behalten dann möglicherweise den Wert, der ihnen durch den Schreibzugriff von Aktivität *A* zugewiesen wurde. So kann eine Aktivität bspw. das Feld `Ort` einer bereits geschriebenen Adresse überschreiben, nicht aber die dazugehörige PLZ.

Dieses Verhalten kann gewünscht sein und widerspricht nicht den Korrektheitskriterien für den Datenfluss, da durch den Zugriff von Aktivität *B* nicht weniger Felder mit Daten versorgt sind, als vor ihrem Zugriff. Um sicherzustellen, dass die Daten konsistent bleiben, müsste Aktivität *B* zunächst das gesamte Datenelement konsumierend lesen (vgl. Kapitel 3) und danach diejenigen Felder wieder obligat schreiben, für die sie eine Versorgung sicherstellen kann.

Um den Prozessmodellierer auf eventuell inkonsistente Daten hinzuweisen, kann der *WriterExists*-Algorithmus aus Kapitel 3 so abgewandelt werden, dass er nicht mehr konsumierende Zugriffe als Unterbrechung des Datenflusses ansieht, sondern solche, die weniger Felder eines strukturierten Datentyps schreiben, als von der lesenden Aktivität obligat gefordert werden. Da der Algorithmus selbst bereits ausführlich beschrie-

ben wurde, beschränken wir uns darauf zu beschreiben, wie eine Funktion zum Erkennen störender Schreibzugriffe umgesetzt werden kann.

Funktion zur Ermittlung inkonsistenter Schreibzugriffe

Die folgende Definition beschreibt, wann die von einer Aktivität gelesene Feldmenge eines Datenelementes d inkonsistente Daten enthalten kann.

Definition 8 (Eigenschaften potentiell Inkonsistenzen erzeugender Aktivitäten) *Sei n eine Aktivität die lesend auf die Feldmenge F_n des Datenelementes d mit strukturiertem Datentyp* TYPE *zugreift. Sei F_n durch mindestens einen Schreibzugriff einer einzelnen Aktivität v versorgt (d. h. $WriterExistsConsuming(CFS, DFS, n, F_n)$). Die in F_n enthaltenen Daten können Inkonsistenzen aufweisen, wenn gilt:*

$$\exists m \in \{pred^*(n) \cap succ^*(v)\} \text{ mit } ActivityWrites^{DFS}(m, F_m \subset F_n, ANY)$$

Diese Bedingung ist in der Funktion `WritesInconsistent` (Alg. 3, Anhang E) realisiert. Der abgewandelte *WriterExists*-Algorithmus aus Kapitel 3 kann damit zur Erkennung von potentiell inkonsistenten Daten verwendet werden, indem alle Aufrufe der Funktion *ReadsConsuming(DFS,n,d)* durch *WritesInconsistent(DFS,m,F_n)* ersetzt werden.

4.3.6. Auswirkung von Schachtelung

Schachtelung wirkt sich nur insofern auf die Analyse des Datenflusses aus, als dass bei der Überprüfung alle geschachtelten Felder auf Versorgung überprüft werden müssen. Es ist irrelevant, ob es sich bei den zu überprüfenden Feldern um Felder des Datentyps selbst handelt oder um Felder eines eingebetteten, wiederum strukturierten Datentyps.

4.3.7. Kompatibilität eingeschränkter Datentypen

Die Einschränkung des Wertebereichs von Datentypen hat auf die Versorgung von Datenelementen keinen Einfluss. Es ist jedoch notwendig, die Kompatibilität der Datentypen von Parametern und Datenelementen zu gewährleisten.

Jeder eingeschränkte Datentyp ist zwingend zu seinem Basisdatentyp kompatibel. Darüber hinaus kann untersucht werden, ob ein eingeschränkter Datentyp im Wertebereich eines anderen enthalten ist oder ob die Einschränkungen den selben Wertebereich bezeichnen, obwohl es sich ihrem Namen nach um unterschiedliche Datentypen handelt.

Für reguläre Ausdrücke ist es nicht effizient möglich, ihre Gleichheit zu überprüfen[2], die Überprüfung der *Inklusion* ist jedoch mit einem Aufwand von $O(n^2)$ realisierbar [CGS09]. Die Inklusion ist eine Teilmengenbeziehung und besagt hier, dass ein regulärer Ausdruck eine Teilmenge der Sprache eines anderen regulären Ausdrucks beschreibt.

[2] Das Gleichheitsproblem für reguläre Ausdrücke ist NP-hart [Sch00].

Datentypen, die einen eingeschränkten numerischen Wertebereich besitzen, sind genau dann zueinander kompatibel, wenn der Wertebereich des Quelldatentyps eine Teilmenge des Zieldatentyps darstellt. Für die einfache Überprüfung des Wertebereichs genügt es, die obere und die untere Grenze der Einschränkung zu vergleichen.

Wir sehen also, dass ein Wert, der die Einschränkungen eines Datentyps A erfüllt, auch im Sinne der Einschränkungen eines Datentyps B gültig sein kann. Dies erlaubt es bspw. Datenelemente mit eingeschränktem Datentyp mit einem Aktivitätenparameter zu verbinden, dessen Datentyp weniger restriktiv ist. Eine derartige Verknüpfung würde jedoch zu einer impliziten Typkonvertierung führen, wobei Informationen über die ursprüngliche Bedeutung eines Wertes verloren gehen. So kann zwar aus technischer Sicht ein Ausgabeparameter mit dem Datentyp `PLZ` auch mit einem Datenelement des Typs `Integer` verknüpft werden, aus semantischer Sicht ist dies allerdings nicht sinnvoll.

Wir untersagen daher die Verknüpfung von Datenelementen und Aktivitätenparametern, deren Typ nicht identisch ist. Dadurch wird sichergestellt, dass keine ungewollte Typkonvertierung durchgeführt wird. Ist eine derartige Konvertierung dennoch gewünscht, so kann diese explizit durchgeführt werden. Dazu wird eine entsprechende *Konverteraktivität* entwickelt, welche die Konvertierung durchführt.

4.3.8. Kompatibilität verschiedener strukturierter Datentypen

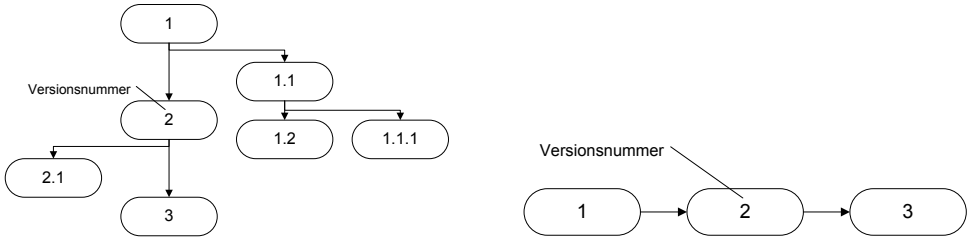
Die automatische Analyse der Kompatibilität strukturierter und geschachtelter Datentypen mit unterschiedlichem Bezeichner ist nicht möglich. Der einfache Grund dafür ist die möglicherweise unterschiedliche Semantik von Feldern im Kontext des sie umgebenden Datentyps. So kann ein Feld `Name` Menschen, Tiere und Gegenstände benennen. Es ist zwar unwahrscheinlich, jedoch nicht gänzlich auszuschließen, dass zwei unterschiedliche Datentypen exakt die selben Felder besitzen. Diese dann als *kompatibel* anzusehen ist riskant. Daher verweigern wir die Kompatibilität verschiedener strukturierter Datentypen und verweisen auch hier auf die Möglichkeit, Konverter zu implementieren.

4.4. Verwaltung

Wie auch das Organisationsmodell und die Aktivitätenvorlagen müssen Datentypen verwaltet werden. Da Datentypen sehr nahe mit Aktivitätenvorlagen und deren Parametern zusammenhängen, ist die Verwaltung der Datentypen im AR[3] sinnvoll. So hat der Administrator des AR jederzeit einen Überblick darüber, welche Datentypen von welchen Aktivitätenvorlagen benötigt werden. Ein Zugriff über andere Verwaltungswerkzeuge, etwa über den PTE ist dadurch nicht ausgeschlossen.

Die Verwaltungskomponente bietet eine Übersicht über alle im System verfügbaren Datentypen, sowie über deren Verwendung in Prozessen. Diese Information ist wichtig, da nur Datentypen gelöscht werden dürfen, die in keinem Prozess verwendet werden.

[3] Aktivitäten-Repository



(a) Versionierung mit Baumstruktur **(b)** lineare Versionierung

Abbildung 4.7 Zwei verschiedene Versionierungs-Schemata

Wurden bereits Instanzen eines Prozesses generiert, die einen Datentyp enthalten, so kann der Datentyp zwar gelöscht werden, wenn die dazugehörige Prozessvorlage nicht mehr instantiiert werden kann, jedoch wird der Datentyp dann nur aus der Anzeige entfernt, da sonst die bereits abgeschlossenen Instanzen ungültig werden würden. Das Löschen von Datentypen ist ebenfalls untersagt, wenn sie in einem anderen Datentyp als Feld enthalten sind. Neben der Verwendung von Datentypen in Prozessen ist daher auch noch die Verwendung von Datentypen als Felder in anderen Datentypen anzuzeigen.

Neben der Möglichkeit, Datentypen anzulegen, abzuändern und zu löschen, ist es sinnvoll eine Möglichkeit für den Import von Datentypen zu bieten. Diese können als eigenständige Sammlung von Datentypen geliefert werden oder zusammen mit Aktivitätenvorlagen ins System importiert werden. Darüber hinaus besitzen Datentypen ein boolesches Attribut `veraltet`. Wird dieses Attribut für einen Datentyp gesetzt, so zeigt es an, dass der Datentyp nicht mehr verwendet werden bzw. durch einen anderen Datentyp ersetzt werden soll. Zusätzlich zu diesem Attribut ist ein Kommentarfeld vorhanden, in dem angegeben werden kann, welcher Datentyp als Ersatz verwendet werden soll oder welchen Grund der Administrator hat, den Datentyp als *veraltet* zu kennzeichnen.

4.4.1. Versionierung

Datentypen können nicht nur manuell als veraltet markiert werden, sondern auch automatisch. Beispielsweise geschieht dies automatisch, wenn von einem Datentyp eine neue Version erzeugt wird. Notwendig werden neue Versionen, wenn ein Datentyp geändert werden soll, der bereits in Prozessvorlagen verwendet wurde.

Zur Realisierung von Versionierung stehen unterschiedliche Konzepte, etwa eine *Baumstruktur* (s. Abb. 4.7a) oder eine *lineare Versionierung* (s. Abb. 4.7b) zur Verfügung. Welche Art der Versionierung am besten für Datentypen geeignet ist, muss in der Praxis erprobt werden. Daher sprechen wir keine Empfehlung für ein bestimmtes Versionierungskonzept aus. O. B. d. A gehen wir im weiteren Verlauf der Arbeit von einer linearen Versionierung aus. Diese bietet ein für uns ausreichendes Konzept für eine auf den Versionsnummern definierte Ordnungsrelation. Dadurch lässt sich für zwei

gegebene Versionsnummern bestimmen, welche der beiden neuer und welche älter ist oder ob beide identisch sind.

Diese Versionsnummer fließt, unabhängig von der Art der Versionierung, immer mit in den Bezeichner eines Datentyps mit ein. Dadurch wird sichergestellt, dass Datentypen mit unterschiedlicher Versionsnummer zueinander inkompatibel sind. Es ist folglich nicht möglich, Parameter und Datenelemente miteinander zu verbinden, deren Datentypen unterschiedlich sind, selbst wenn dieser Unterschied nur in der Versionsnummer besteht. Diese Inkompatibilität begründet sich dadurch, dass derselbe Datentyp in unterschiedlichen Versionen eine unterschiedliche Struktur aufweisen kann.

4.4.2. Evolution

Werden Datentypen verändert, so müssen auch die Prozessmodelle angepasst werden, in welchen diese Datentypen verwendet werden. Weiter sind die Änderungen der Prozessvorlagen auf laufende Prozessinstanzen zu übertragen, die auf den geänderten Prozessvorlagen basieren.

Aktualisierung von Prozessvorlagen

Wurde eine neue Version eines Datentyps erzeugt, so ist der Prozessmodellierer an geeigneter Stelle in den betroffenen Prozessvorlagen darauf hinzuweisen. Er kann dann auf die Hinweise reagieren und die Prozessvorlagen aktualisieren, wo dies notwendig ist. Zwar ist es auch möglich, *veraltete* Versionen von Datentypen zu verwenden, was auf Dauer aber nur in wenigen Fällen sinnvoll ist.

Wird der Typ eines Datenelementes in einer Prozessvorlage geändert, so müssen gleichzeitig auch alle Aktivitäten geändert werden, die auf dieses Datenelement zugreifen. Der Datentyp von Parametern und Datenelementen muss weiterhin identisch sein, wenn diese miteinander verknüpft sind. Diese Bedingung ist bei unterschiedlichen Versionsnummern der Datentypen nicht erfüllt, da die Versionsnummer im Identifizierungsmerkmal des Datentyps enthalten ist.

Der Datentyp eines Parameters muss dann *nicht* angepasst werden, wenn dieser nur mit einem einzelnen Feld eines geänderten Datenelementes verbunden ist und dieses Feld von den Änderungen des Datentyps nicht betroffen ist. Dies bedeutet insbesondere, dass auch Aktivitäten nicht geändert werden müssen, welche über Parameter eine Verbindung zu jedem Feld des alten Datentyps besitzen, sofern die Änderung des Datentyps nur aus dem Hinzufügen von Feldern besteht. Es dürfen dann weder bestehende Felder in ihrem Typ geändert werden, noch dürfen bestehende Felder gelöscht werden, damit die Kompatibilität bestehen bleibt. Eine Folge der Datentypänderung ist dann jedoch, dass das Datenelement, im Falle obligater Schreibzugriffe, nicht mehr als Ganzes mit Daten versorgt ist (Definition 6).

Aktualisieren laufender Prozessinstanzen

Der Datentyp eines Datenelementes kann auch in laufenden Prozessinstanzen geändert werden. Dies kann manuell oder durch Schemaevolution geschehen, wobei in beiden Fällen die im Folgenden beschriebenen Korrektheitskriterien zu erfüllen sind.

Existieren in der zu ändernden Prozessinstanz Aktivitäten, die das gesamte zu ändernde Datenelement lesen oder schreiben, so darf sich keine dieser Aktivitäten im Zustand `RUNNING` oder `COMPLETED` befinden. Dafür gibt es zwei Gründe: Zum einen müssen alle mit dem Datenelement verknüpften Aktivitäten ebenfalls ausgetauscht werden, was nur möglich ist, wenn sie sich in keinem der beiden genannten Zustände befinden. Zum anderen ist das Ändern des Datentyps eines Datenelementes mit dessen Löschung zu vergleichen. Das Datenelement mit dem alten Datentyp wird samt aller seiner Datenkanten gelöscht. Anschließend werden ein Datenelement mit dem neuen Datentyp, sowie alle vor dem Löschen vorhandenen Datenkanten wieder eingefügt. Für das Löschen eines Datenelementes darf sich ebenfalls keine der mit ihm verbundenen Aktivitäten in einem der beiden genannten Zustände befinden.

Einen Sonderfall stellen Aktivitäten dar, die nur einzelne Felder des zu ändernden Datenelementes lesen oder schreiben. Sind die gelesenen bzw. geschriebenen Felder nicht von der Änderung des Datentyps betroffen, so darf dieser auch dann geändert werden, wenn sich die Aktivität bereits im Zustand `RUNNING` oder `COMPLETED` befindet. Das Ändern des Datentyps kann hier nicht mit dem Löschen des gesamten Datenelementes gleichgesetzt werden. Vielmehr wird das Korrektheitskriterium aus [Rin04] auf jedes einzelne Feld des strukturierten Datentyps angewendet. Es dürfen also nur solche Felder gelöscht werden bzw. verändert werden, die nicht mit einer Aktivität verknüpft sind, welche sich im Zustand `RUNNING` oder `COMPLETED` befindet.

4.5. Zusammenfassung

Strukturierte, eingeschränkte und komplexe Datentypen bieten Prozessmodellierern die Möglichkeit, große Datenmengen übersichtlich in Prozesse zu integrieren. Die vorgestellte Umsetzung auf bestehende Konzepte macht die neuen Datentypen für Prozessmodellierer und Aktivitätenentwickler verständlich, da diese ihr vorhandenes Wissen weiter nutzen können. Durch die vorgestellte Visualisierung erkennt der Prozessmodellierer leicht, wenn mehrere Felder eines Datentyps mit derselben Aktivität verbunden sind. Wir haben gezeigt, wie die Datenflussanalyse auf Datenelemente von strukturiertem Datentyp anzuwenden ist. Die Nähe der neuen Datenkonstrukte zu bereits vorhandenen Konzepten ermöglicht die Wiederverwendung vorhandener Analysealgorithmen.

Mit strukturierten und komplexen Datentypen ist der Anwender in der Lage, Daten innerhalb eines Prozesses als zusammengehörig zu kennzeichnen. Partielle Schreibzugriffe auf zusammengehörende Daten können zu Inkonsistenzen führen. Wir haben eine Möglichkeit vorgestellt, wie derartige inkonsistente Schreibzugriffe erkannt werden können, indem wir den bestehenden `WriterExists`-Algorithmus entsprechend angepasst haben.

Weiter haben wir Rahmenbedingungen für eine zukünftige Verwaltung von Datentypen aufgestellt. Diese beinhalten ein einfaches Versionierungskonzept. Dieses haben wir genutzt, Korrektheitskriterien für die Evolution geänderter strukturierter und komplexer Datentypen in Prozessvorlagen und -instanzen aufzustellen.

5. Benutzerdefinierte Funktionen

Wie in Kapitel 2 beschrieben, bietet ADEPT2 mit UDTs die Möglichkeit, Daten in Prozesse zu integrieren, deren Struktur dem PMS verborgen bleibt. Der Umgang mit diesen Daten ist Aktivitäten vorbehalten, denen die Struktur dieser Daten bekannt ist. Derzeit gibt es für Aktivitäten, die keine Informationen über die Struktur dieser benutzerdefinierten Datentypen haben, keine Möglichkeit, auf deren Inhalt oder Teile davon zuzugreifen.

Dies ändern die nachfolgend vorgestellten *benutzerdefinierten Funktionen*. Wir erläutern zunächst deren Einsatzgebiet, sowie die Abgrenzung von dem bereits vorhandenen Konzept für *Aktivitäten*. Anschließend klären wir, wie sie im Rahmen des bestehenden Systems umgesetzt werden können und welche Auswirkungen sie auf den Datenfluss haben. Zuletzt beschreiben wir, wie benutzerdefinierte Funktionen verwaltet werden können und wie sich Änderungen auf bereits laufende Prozessinstanzen auswirken.

5.1. Motivation

In diesem Abschnitt wird das Ziel beschrieben, welches mit benutzerdefinierten Funktionen erreicht werden soll und wie sich benutzerdefinierte Funktionen von Aktivitäten unterscheiden. Weiter wird erläutert, welche Möglichkeiten durch die *Parametrisierung* von Funktionen entstehen und inwieweit die Unterstützung verschiedener Programmiersprachen für die Implementierung benutzerdefinierter Funktionen sinnvoll sein kann.

5.1.1. Zielsetzung

Möchte eine Aktivität auf den Inhalt eines benutzerdefinierten Datentyps zugreifen, die nicht für den Umgang mit diesem UDT entwickelt wurde, so ist eine Konvertierung der Daten in ein Format notwendig, welches der Aktivität bekannt ist. Diese Konvertierung kann derzeit von eigens für diesen Zweck entwickelten Konverteraktivitäten durchgeführt werden (s. Abb. 5.1).

Wird häufig von Aktivitäten auf einen ihnen unbekannten UDT zugegriffen, erhöht sich dadurch die Anzahl von Aktivitäten in einem Prozess deutlich, da für jeden Zugriff eine zusätzliche Konverteraktivität eingefügt werden muss. Greift die Aktivität lesend und schreibend auf das Datenelement zu, muss die Konvertierung pro Aktivität zweimal erfolgen. Dies führt im schlimmsten Fall zu einer Verdreifachung der Knotenanzahl, wodurch der Prozess unübersichtlich und ineffizient wird.

Zwar können die konvertierenden Aktivitäten in Vor- bzw. Nachschaltdiensten untergebracht werden [Rei00], jedoch muss die Konvertierung dann entweder vom Pro-

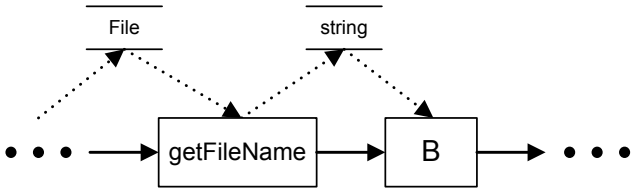


Abbildung 5.1 Extrahieren eines Dateinamens aus einem UDT durch Konverteraktivität

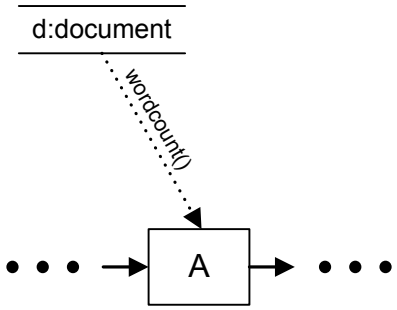


Abbildung 5.2 Zugriff auf die Wortanzahl eines Dokumentes vermittels UDF

zessmodellierer explizit für jede Aktivität durchgeführt werden, indem er die Vor- und Nachschaltdienste mit Konverteraktivitäten versieht, oder aber die Aktivität muss selbst diese Konverterdienste mitbringen, wozu sie aber wiederum Wissen über den zu konvertierenden Datentyp benötigt.

Benutzerdefinierte Funktionen treten sowohl dem Problem der unübersichtlichen und ineffizienten Prozessmodelle als auch dem erheblichen Aufwand entgegen, welcher beim Modellieren dieser Prozesse entsteht, indem sie eine implizite Konvertierung der Daten vornehmen, sobald eine mit ihnen verknüpfte Aktivität ausgeführt wird. Abbildung 5.2 zeigt den direkten Zugriff auf die Anzahl der Wörter eines Dokumentes. Die Aktivität selbst muss dabei das Format des Dokumentes nicht kennen, sofern für diesen UDT eine entsprechende UDF angeboten wird. Diese UDF interpretiert zur Laufzeit den Wert des Datenelementes, dem sie zugeordnet ist. Da ihr die Struktur des Dokumentes bekannt ist, kann sie die Anzahl der enthaltenen Wörter berechnen und als ganze Zahl an die mit ihr verknüpfte Aktivität weitergeben.

5.1.2. Parametrisierte Funktionen

Es liegt nahe, einer benutzerdefinierten Funktion beim Aufruf über Parameter zusätzliche Informationen für den Ablauf der Konvertierung zu übermitteln. Beispielsweise könnte für den Zugriff auf XML-Daten ein XPath[1]-Ausdruck übergeben werden, welcher von der Funktion auf das XML-Dokument angewendet wird. Das Ergebnis des

[1] XML Path Language

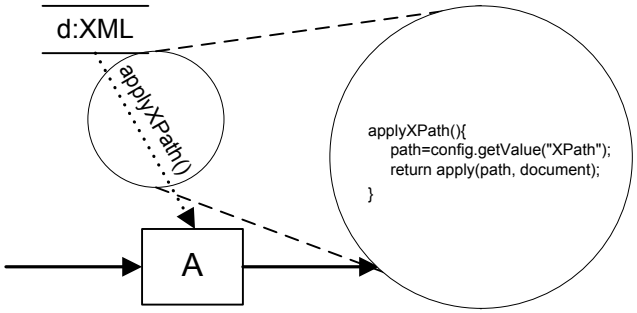


Abbildung 5.3 Funktionsaufruf zur Anwendung eines XPath-Ausdrucks auf ein XML-Dokument

Ausdrucks ist dann der Rückgabewert der Funktion, der an die aufrufende Aktivität übergeben wird.

Es ist nichts dagegen einzuwenden, Funktionsaufrufe zur Modellierzeit mit Parametern zu versehen. Ein Problem entsteht jedoch, wenn der zu verwendende Parameter erst zur Prozesslaufzeit ermittelt werden soll. Da der Funktionsaufruf erst beim Start der aufrufenden Aktivität durchgeführt wird, diese aber den Parameter für die Parametrisierung der Funktion ebenfalls als Eingabeparameter besitzt, wird der Funktionsaufruf gleichwertig mit dem Zugriff auf den Parameter druchgeführt. Der Parameter kann daher nicht für den Funktionsaufruf verwendet werden, da sein Wert beim Aufrufen der Funktion noch nicht bekannt ist.

Darüber hinaus ähneln Funktionen mit mehreren Parametern sehr stark dem Konzept der Aktivitäten. Um die beiden Konzepte deutlich voneinander abzugrenzen, werden wir im Folgenden nur Funktionen zulassen, die für Lesezugriffe keine Parameter entgegennehmen und für Schreibzugriffe nur den Parameter besitzen, der die zu schreibenden Daten enthält.

Dies schließt nicht aus, dass eine Funktion durch unterschiedliche Konfigurationen angepasst wird. Die Konfigurationswerte werden beim Modellieren festgelegt und stehen so bei jedem Funktionsaufruf zur Verfügung. Auf diesem Wege lassen sich dann auch die gerade angesprochenen XPath-Ausdrücke realisieren. Für eine Funktion `applyXPath()` kann zur Modellierzeit beispielsweise bei jeder Verwendung ein eigener XPath-Ausdruck angegeben werden (s. Abb. 5.3).

5.1.3. Unterstützung verschiedener Programmiersprachen

Für Aktivitäten besitzt ADEPT2 eine Schnittstelle, die unabhängig von der Sprache verwendet werden kann, in welcher die Aktivität implementiert ist. Einzige Voraussetzung ist, dass für die jeweilige Sprache eine passende Ausführungsumgebung bereitgestellt wird.

So können beispielsweise Aktivitäten, die in Java entwickelt wurden, direkt vom System ausgeführt werden, da dieses eine Java-Ausführungsumgebung bereitstellt. Weite-

re Ausführungsumgebungen gibt es unter Anderem für EXE-Dateien unter Windows, für den Aufruf von Webservices, sowie für diverse Skriptsprachen.

Diese Flexibilität soll auch für benutzerdefinierte Funktionen beibehalten werden. Damit kann für jeden UDT und die dafür entwickelten UDFs die jeweils am besten geeignete Programmiersprache verwendet werden.

5.1.4. Abgrenzung von Aktivitäten

Um entscheiden zu können, ob die Lösung einer Aufgabe als Aktivität oder als UDF umzusetzen ist, müssen dem jeweiligen Entwickler die Unterschiede der beiden Konzepte bekannt sein. Aus diesen Unterschieden resultieren jedoch keine eindeutigen Regeln, die eine klare Trennung der beiden Konzepte ermöglichen. Vielmehr dienen sie als Grundlage für die von Fall zu Fall neu zu treffende Entscheidung.

Benutzerdefinierte Funktionen übernehmen Aufgaben, welche aus technischer Sicht auch von Aktivitäten durchgeführt werden können. Das Lösen dieser Aufgaben durch Aktivitäten führt jedoch zu unnötig großen und unübersichtlichen Prozessmodellen. Weiter ist der Funktionsumfang von Aktivitäten deutlich größer, als es für Aufgaben notwendig ist, welche von UDFs gelöst werden können. Unnötig für das Ausführen einer UDF sind etwa die Bearbeiterzuordnung oder das Unterbrechen und Wiederaufsetzen der UDF während der Ausführung, wie es bei Aktivitäten möglich ist. Aus prozesslogischer Sicht unterscheiden sich UDFs von Aktivitäten dadurch, dass eine Aktivität explizit als Prozessschritt ausgeführt wird, wohingegen der Aufruf einer UDF implizit beim Zugriff durch eine Aktivität erfolgt.

Ziel benutzerdefinierter Funktionen ist es, Datenkonvertierungen für Aktivitäten durchzuführen, die selbst nicht in der Lage sind, auf die zu konvertierenden Daten zuzugreifen. Ist für eine solche Konvertierung keine Benutzerinteraktion notwendig und kann sie ohne Bearbeiterzuordnung erfolgen, so ist sie als UDF zu realisieren. Für Aufgaben, bei denen eine Unterbrechung zur Laufzeit möglich sein muss, sind Aktivitäten zu verwenden.

5.2. Realisierung

Nachdem wir beschrieben haben, welche Aufgaben benutzerdefinierte Funktionen erfüllen sollen, wenden wir uns deren Realisierung zu. Diese betrachten wir aus drei Blickwinkeln: aus Sicht des Entwicklers von benutzerdefinierten Funktionen, des Prozessmodellierers und des Aktivitätenentwicklers.

5.2.1. Schnittstellen für Prozessmodellierer

Für Prozessmodellierer stellen sich benutzerdefinierte Funktionen als Anknüpfpunkte für Datenkanten an benutzerdefinierten Datentypen dar. Neben der Verknüpfung benutzerdefinierter Funktionen mit Aktivitätenparametern kann weiterhin das ganze Datenelement mit einem Parameter von entsprechendem Typ verknüpft werden,

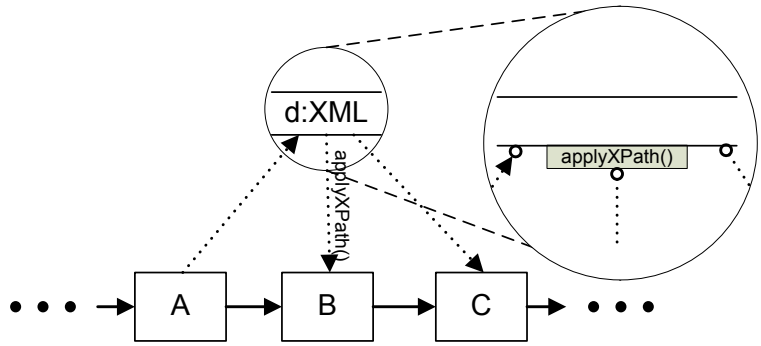


Abbildung 5.4 Gleichzeitige Verknüpfung von Datenelement und UDF

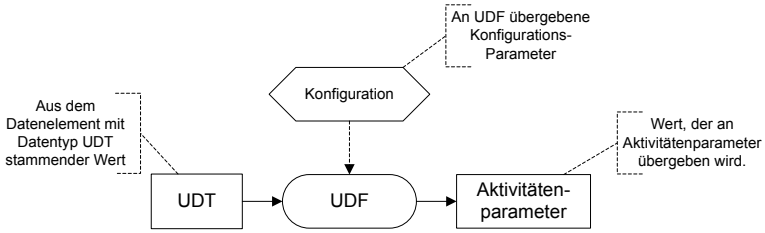


Abbildung 5.5 Schematische Darstellung einer lesenden UDF

die Verwendung benutzerdefinierter Funktionen ist davon unabhängig. Dieser Sachverhalt ist in Abbildung 5.4 dargestellt. Dort schreibt Aktivität A das Datenelement d als Ganzes, Aktivität B greift lesend auf die UDF `applyXPath` zu und Aktivität C liest anschließend den gesamten Wert des Datenelementes.

Unterscheidung zwischen Ein- und Ausgabefunktionen

Wir unterscheiden zwischen *lesenden* und *schreibenden* benutzerdefinierten Funktionen. Lesende Funktionen (s. Abb. 5.5) dienen dem lesenden Zugriff auf Teilbereiche des benutzerdefinierten Datentyps, wohingegen schreibende Funktionen (s. Abb. 5.6) dafür sorgen, dass an sie übergebene Daten mit dem alten Wert des UDT kombiniert und als neuer Wert des Datenelementes gespeichert werden.

Unabhängig von ihrer Richtung besitzt jede Funktion einen Datentyp, der ihre Kompatibilität zu Aktivitätenparametern beschreibt. Korrekterweise handelt es sich dabei für lesende Funktionen um einen Rückgabetyp und bei schreibenden Funktionen um den Datentyp des ersten Funktionsparameters. Mit diesen Details soll der Prozessmodellierer jedoch nicht in Kontakt kommen.

Verknüpfung benutzerdefinierter Funktionen mit Parametern

Konsequenterweise dürfen lesende Funktionen nur mit Eingabeparametern von Aktivitäten verknüpft werden, schreibende Funktionen nur mit Ausgabeparametern. Dabei

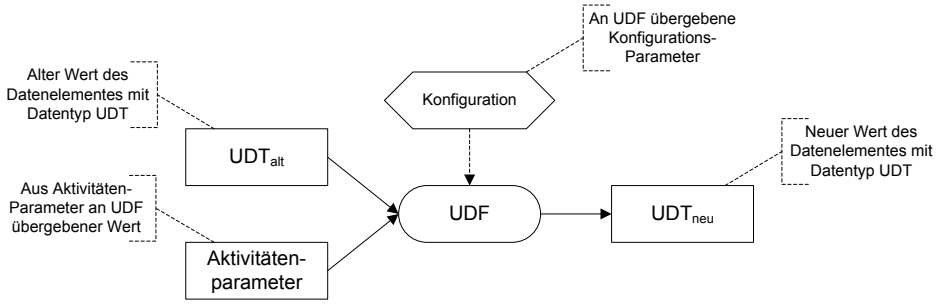


Abbildung 5.6 Schematische Darstellung einer schreibenden UDF

muss der Datentyp der Funktion mit dem jeweiligen Datentyp des Parameters übereinstimmen, mit dem die Funktion verknüpft wird. Das Verknüpfen von lesenden Funktionen mit Ausgabeparametern oder von schreibenden Funktionen mit Eingabeparametern ist unzulässig.

Konfiguration benutzerdefinierter Funktionen

Benutzerdefinierte Funktionen können vom Prozessmodellierer auf zwei unterschiedliche Arten konfiguriert werden: Zum einen kann eine Konfiguration für ein Datenelement festgelegt werden, dessen Datentyp eine UDF bereitstellt, zum anderen besteht die Möglichkeit für jede Verknüpfung mit einer Aktivität eine individuelle Konfiguration anzugeben.

Prozessspezifische Konfiguration Wie in Abschnitt 5.1 beschrieben, können benutzerdefinierte Funktionen prozessspezifisch konfiguriert werden. Dabei werden für eine UDF eines Datenelementes mit benutzerdefiniertem Datentyp Konfigurationswerte angegeben, die dann für jeden Zugriff gültig sind, der innerhalb eines Prozesses auf diese Funktion erfolgt.

Es ist nicht vorgesehen, die Konfigurationen verschiedener Datenelemente gleichen Typs zu synchronisieren. Es besteht jedoch die Möglichkeit, Werte aus der Prozesskonfiguration als Konfigurationsparameter einer UDF zu verwenden. Somit kann ein Wert gleichzeitig für die prozessspezifische Konfiguration verschiedener benutzerdefinierter Funktionen dienen.

Beispiel 7 (Prozessspezifische Konfiguration einer UDF) *In Abbildung 5.7 werden aus drei parallelen Zweigen heraus (Aktivität A, B und C) XML-Dokumente erzeugt. Anschließend wird mit Hilfe einer UDF ein XPath-Ausdruck auf jedes dieser Dokumente angewendet, dessen Ergebnis von Aktivität D gelesen wird. Der anzuwendende XPath-Ausdruck wird dabei als Konfigurationswert der UDF* `applyXPath()` *angegeben. Da er für alle drei Funktionsaufrufe identisch sein soll, ist der XPath-Ausdruck in der Prozesskonfiguration hinterlegt.*

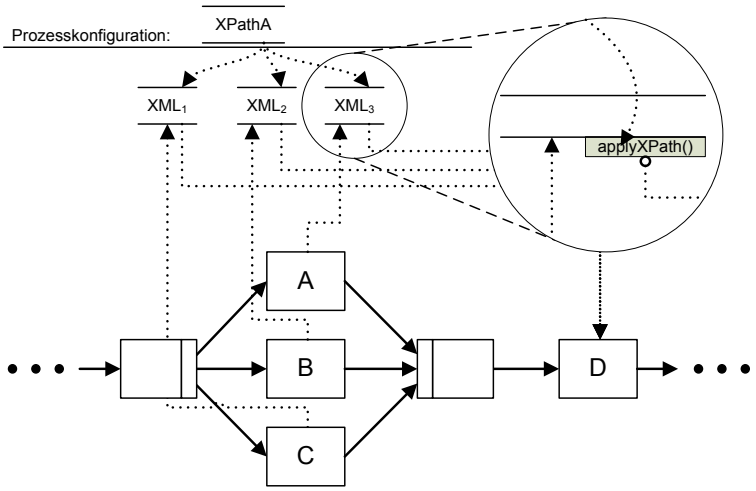


Abbildung 5.7 Prozessspezifische konfiguration mehrerer UDFs

Verknüpfungsspezifische Konfiguration Neben der einheitlichen Konfiguration benutzerdefinierter Funktionen für alle Zugriffe durch Aktivitäten, kann auch eine Konfiguration *individuell für jeden Zugriff* festgelegt werden. Da Datenkanten nicht konfiguriert werden können, wird die Konfiguration bei der UDF gespeichert, die das eine Ende der Verknüpfung darstellt.

Eine solche *verknüpfungsspezifische* Konfiguration ist nicht direkt mit den Konfigurationen anderer UDF-Verknüpfungen synchronisierbar. Jedoch besteht auch hier die Möglichkeit, auf Werte aus der Prozesskonfiguration zuzugreifen, was die Verwendung eines Konfigurationswertes für mehrere verknüpfungsspezifische Konfigurationen ermöglicht.

5.2.2. Schnittstellen für UDF-Entwickler

Aktivitätenentwickler können auf die Schnittstelle `DataContext` zugreifen, um Daten mit dem PMS auszutauschen. Dabei kann der Wert eines jeden Aktivitätenparameters gelesen und die Werte von Ausgabeparametern geschrieben werden. Der Zugriff erfolgt über den Namen des jeweiligen Parameters. Um Entwicklern von benutzerdefinierten Funktionen die gleichen Rahmenbedingungen zur Verfügung zu stellen wie Aktivitätenentwicklern, verwenden wir eine Variante der `DataContext`-Schnittstelle.

Für den Zugriff auf Konfigurationsparameter steht einer UDF eine Schnittstelle zur Verfügung, welche analog zu derjenigen funktioniert, die Aktivitäten für den Zugriff auf Konfigurationsparameter verwenden. Da diese Schnittstelle für benutzerdefinierte Funktionen nicht abgewandelt werden muss, wird sie in dieser Arbeit nur ansatzweise beschrieben. Alle Schnittstellen sind in Anhang C.3 zusammengefasst.

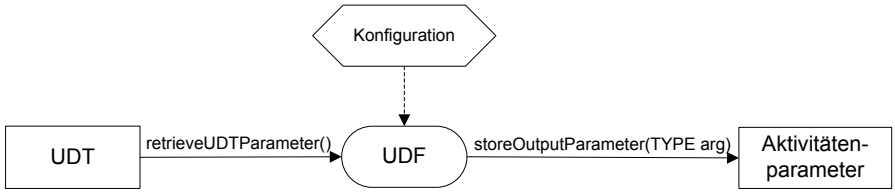


Abbildung 5.8 Schematische Darstellung der Funktionsaufrufe einer lesenden UDF

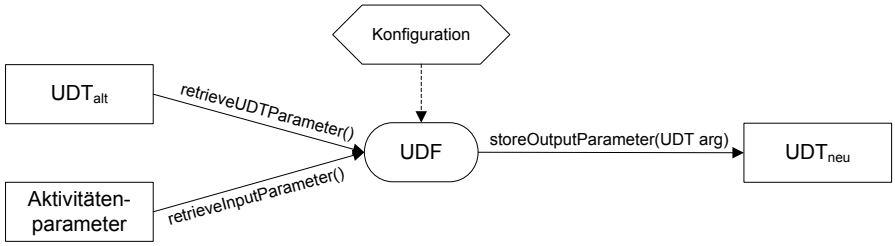


Abbildung 5.9 Schematische Darstellung der Funktionsaufrufe einer schreibenden UDF

Zugriff auf Ein- und Ausgabeparameter

Die Schnittstelle für den Zugriff einer UDF auf ihren Datenkontext besteht aus drei Methoden. Diese bieten einer *lesenden* UDF lesenden Zugriff auf den UDT, zu dem sie gehört und schreibenden Zugriff auf einen Ausgabeparameter ihres Ausgabedatentyps (s. Abb. 5.8). Der Zugriff erfolgt dabei im Gegensatz zu Aktivitäten nicht über Prameternamen, sondern implizit.

Eine *schreibende* UDF besitzen zwei Eingabeparameter (den UDT, mit dem sie verknüpft ist, sowie den in diesen UDT zu integrierenden Wert aus dem Aktivitätenparameter) und den zu manipulierenden UDT als einzigen Ausgabeparameter (s. Abb. 5.9).

Funktion `retrieveUDTParameter()` Diese Methode bietet sowohl einer lesenden als auch einer schreibenden UDF Lesezugriff auf ihren zugehörigen UDT. Der schreibende Zugriff für eine schreibende UDF erfolgt gesondert über deren Ausgabeparameter.

Funktion `retrieveInputParameter()` Eine schreibende UDF benötigt neben dem lesenden Zugriff auf ihren UDT auch noch Zugang zu den an sie übermittelten Daten. Letzterer erfolgt nur lesend. Da eine lesende UDF außer ihrem umgebenden UDT keine Eingabedaten besitzt, hat `retrieveInputParameter()` bei solchen UDFs die selbe Funktionalität wie `retrieveUDTParameter()`.

Funktion `storeOutputParameter(TYPE arg)` Sowohl lesende als auch schreibende benutzerdefinierte Funktionen erzeugen Ausgabedaten, die in einem *Ausgabecontainer* gespeichert werden. Um einen Wert `arg` in diesem Ausgabecontainer abzulegen, wird

die Funktion `storeOutputParameter(TYPE arg)` verwendet. Die in diesem Ausgabecontainer gespeicherten Werte entsprechen dem Rückgabewert der UDF nach deren Beendigung. Bei einer *lesenden* UDF bezieht sich der Datentyp `TYPE` des Argumentes dieser Funktion auf den Ausgabedatentyp der UDF, wohingegen dieser bei einer *schreibenden* UDF den Datentyp des die Funktion umgebenden UDT widerspiegelt.

Zugriff auf Funktions- und Instanzkonfiguration

Da für jede Instanz nur eine Konfiguration gültig sein kann, werden Funktionskonfigurationen von dedizierten Instanzkonfigurationen überschrieben. So entsteht ein dreischichtiges Modell: Ist für einen einzelnen Zugriff auf eine UDF eine Konfiguration angegeben, so ist diese gültig. Fehlende Werte werden aus Konfigurationen mit niedrigerer Priorität aufgefüllt. Für alle Zugriffe, für die selbst keine Konfiguration angegeben ist, sind die Werte aus der prozessspezifischen Konfiguration der UDF gültig. Diese werden wiederum um fehlende Werte aus der Konfiguration ergänzt, die beim Anlegen der UDF in der Datentypverwaltung festgelegt wurde. Bei allen UDFs, für die keine prozessspezifische Konfiguration festgelegt ist, und deren Zugriffe auch nicht einzeln konfiguriert sind, besitzt die globale Konfiguration aus der Datentypverwaltung alleinige Gültigkeit.

Die Vereinigung der verschiedenen Konfigurationen geschieht für den Entwickler benutzerdefinierter Funktionen transparent. Der Zugriff auf die tatsächlich gültige Konfiguration erfolgt immer über dieselbe Schnittstelle. Sie bietet lesenden Zugriff auf alle Konfigurationswerte. Der Zugriff erfolgt dabei über den eindeutigen Namen des Parameters, äquivalent zum Zugriff einer Aktivität auf ihre Eingabeparameter.

5.2.3. Schnittstellen für Aktivitätenentwickler

Für Aktivitätenentwickler stellen benutzerdefinierten Funktionen ein transparentes Konzept dar. Der Rückgabewert einer lesenden UDF wird dem Aktivitätenentwickler über einen Eingabeparameter bereitgestellt. Umgekehrt wird eine schreibende UDF mit einem Eingabewert versorgt, indem dieser im mit der UDF verknüpften Ausgabeparameter bereitgestellt wird.

Wir gehen davon aus, dass Aktivitäten, die auf einen UDT als Ganzes zugreifen, diesen selbst interpretieren können und dies auch tun. Deswegen bietet ein Aktivitätenparameter mit benutzerdefiniertem Datentyp keine Schnittstelle für den Zugriff auf die benutzerdefinierten Funktionen dieses UDT. Werden zusätzlich zu den Rohdaten auch noch Werte benötigt, die über eine UDF bereitgestellt werden, so können diese über einen separaten Parameter abgefragt werden. Dies erspart dem Aktivitätenimplementierer die Verwendung eines zusätzlichen Konzeptes, dessen Nutzen gleichwertig mit bestehenden Konzepten erreicht werden kann.

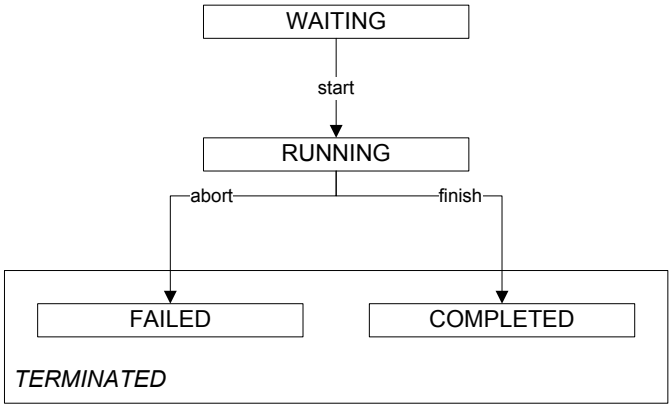


Abbildung 5.10 Zustände und Zustandsübergänge benutzerdefinierter Funktionen (vereinfacht; analog zu Aktivitätenzuständen aus [Rei00])

5.3. Dynamisches Verhalten

Im ADEPT2-Metamodell, welches in Abschnitt 2.2 beschrieben wurde, werden die Daten aus verknüpften Datenelementen beim Start einer Aktivität in deren Eingabeparametern zur Verfügung gestellt. Entsprechend werden die Daten der Ausgabeparameter beim Beenden der Aktivität in die mit ihnen verknüpften Datenelemente zurückgeschrieben.

Analog zu diesem Konzept definieren wir im Folgenden die Ausführungszeitpunkte benutzerdefinierter Funktionen. Dazu beschreiben wir zunächst die Zustände, die eine UDF in einer laufenden Prozessinstanz annehmen kann. Da die Ausführung von benutzerdefinierten Funktionen, im Gegensatz zum Zugriff auf normale Datenelemente, fehlschlagen kann, beschreiben wir anschließend, wie sich das Fehlschlagen benutzerdefinierter Funktionen auf das dynamische Verhalten von Aktivitäten auswirkt.

5.3.1. Zustände und Zustandsübergänge

Initial befindet sich eine benutzerdefinierte Funktion im Zustand `WAITING`. In diesem Zustand wartet sie auf ihre Instantiierung durch eine verknüpfte Aktivität.

Wird eine Aktivität gestartet, so wird eine neue Instanz aller benutzerdefinierten Funktionen erzeugt, die mit den Eingabeparametern der Aktivität verknüpft sind. Eine Instanz einer UDF wird sofort gestartet und befindet sich somit initial im Zustand `RUNNING`. In diesem Zustand verbleibt die Instanz der UDF, bis sie abgeschlossen ist. Anschließend wechselt sie in den Zustand `COMPLETED`, sofern ihre Ausführung erfolgreich war. Andernfalls wechselt sie von `RUNNING` nach `FAILED`. Diese Zustände und Zustandsübergänge einer UDF-Instanz sind in Abbildung 5.10 dargestellt.

5.3.2. Ausführungszeitpunkt

Funktionen für den lesenden Zugriff auf benutzerdefinierte Datentypen werden mit dem Starten einer Aktivität ausgeführt, also sobald diese in den Zustand `RUNNING` wechselt. Nach Beendigung der Funktion wird ihr Rückgabewert im mit ihr verknüpften Eingabeparameter der Aktivität bereitgestellt. Folglich kann die Aktivität erst dann tatsächlich gestartet werden, wenn alle lesenden Funktionen erfolgreich beendet wurden, was der Fall ist, wenn sie sich im Zustand `COMPLETED` befinden.

Entsprechend werden schreibende Funktionen ausgeführt, wenn die Aktivität beendet ist. Da jedoch erst alle Funktionen ausgeführt sein müssen, bevor nachfolgende Aktivitäten über die Beendigung informiert werden können, müssen benutzerdefinierte Funktionen als zur Aktivität gehörend angesehen werden. Um diese Zugehörigkeit zu gewährleisten, werden benutzerdefinierte Funktionen als Vor- bzw. Nachschaltdienste im Sinne von [Rei00] angesehen.

5.3.3. Korrekte Ausführung

Da lesende Funktionen ein Datenelement nicht manipulieren und pro Datenelement nur eine schreibende Funktion mit einer Aktivität verknüpft werden darf, können sowohl alle lesenden als auch alle schreibenden Funktionen, die mit einer Aktivität verknüpft sind, jeweils parallel ausgeführt werden.

Eine Aktivität kann dann nicht gestartet werden, wenn mindestens eine der mit ihren Eingabeparametern verknüpfte Funktion fehlschlägt. Entsprechend kann eine Aktivität nur dann erfolgreich beendet werden, wenn alle mit ihren Ausgabeparametern verknüpften Funktionen erfolgreich ausgeführt wurden.

Transaktionale Ausführung benutzerdefinierter Funktionen

Schlägt eine schreibende Funktion fehl, so muss sichergestellt werden, dass bereits von anderen Funktionen manipulierte Datenelemente wieder auf ihren vorherigen Wert zurückgesetzt werden. Vergleichbar mit Datenbanken wird die Weitergabe der Ausgabedaten einer Aktivität an Datenelemente in einer Transaktion gekapselt. Dabei werden insbesondere *Atomarität*, *Konsistenz* und *Dauerhaftigkeit* der ACID[2]-Eigenschaften [EN09] sichergestellt [3].

Das Transaktionskonzept wird realisiert, indem zunächst alle schreibenden Funktionen gestartet werden. Nach Beendigung einer Funktion stehen ihre Ausgabedaten dem System als ihr jeweiliger Ausgabeparameter zur Verfügung. Da es Aufgabe des Systems ist, die Daten aus dem Datenkontext der UDF in das jeweilige Datenelement zu übernehmen, kann dieser Vorgang so lange verzögert werden, bis alle schreibenden Funktionen erfolgreich beendet wurden. Nach Beendigung aller schreibenden Funk-

[2] atomicity, consistency, isolation and durability; erwünschte Eigenschaft von Transaktionen in Datenbanksystemen

[3] Die Isolation ergibt sich aus dem ADEPT2-Metamodell, welche Schreiboperationen ausschließt, die sich gegenseitig beeinflussen.

tionen werden dann die Werte aller Ausgabeparameter in das jeweilige Datenelement geschrieben.

Wird mindestens eine Funktion nicht erfolgreich beendet, so unterbleibt die Übernahme der Daten für alle Datenelemente.

Zusammenhang zwischen Aktivitätenzuständen und Funktionszuständen

Da benutzerdefinierte Funktionen zum Ausführungskontext einer Aktivität gehören, schlägt diese sowohl fehl, wenn eine lesende Funktion fehlschlägt als auch dann, wenn eine schreibende Funktion nicht korrekt ausgeführt werden kann. Um jedoch kenntlich zu machen, was der genaue Grund für das Fehlschlagen einer Aktivität ist, muss gekennzeichnet werden, ob der Fehler der Aktivität selbst zuzuschreiben ist oder ob dessen Ursache in vor- bzw. nachgeschalteten Funktionen zu suchen ist.

Darüber hinaus muss sichergestellt werden, dass keine Daten zwischen Aktivität, UDF und Datenelement verloren gehen. Dazu werden die Werte aller Ausgabeparameter nach dem Beenden der Aktivität und vor dem Ausführen der nachgeschalteten Funktionen zwischengespeichert. Ob die Aktivität nochmals komplett ausgeführt wird oder ob lediglich die nachgeschalteten Funktionen erneut ausgeführt werden, muss von Fall zu Fall vom Benutzer oder dem Systemadministrator entschieden werden.

5.4. Datenflussanalyse

Nachdem wir nun gesehen haben, wie benutzerdefinierte Funktionen realisiert werden und wie ihr Ausführungsverhalten im Zusammenhang mit den Aktivitäten steht, die mit ihnen verknüpft sind, besprechen wir nachfolgend, wie sich benutzerdefinierte Funktionen aus Sicht der Datenflussanalyse darstellen. Wir erläutern sowohl, wie sich benutzerdefinierte Funktionen auf die Versorgung eines Datenelementes auswirken, zu dem sie gehören als auch verschiedene Varianten, wie benutzerdefinierte Funktionen konsumierend ausgeführt werden können.

5.4.1. Versorgung gesamter Datenelemente

Da das System keine Informationen über die Struktur des benutzerdefinierten Datentyps hat, ist es nicht möglich, Funktionen zu erstellen, die nur Teilbereiche des UDT versorgen. Ist die Struktur (teilweise) bekannt, so können benutzerdefinierte Datentypen, und damit auch benutzerdefinierte Funktionen, mit dem in Kapitel 4 vorgestellten Konzept der strukturierten Datentypen kombiniert werden.

Aufgrund der Implementierung benutzerdefinierter Funktionen wirken sich optionale und obligate Zugriffe auf schreibende Funktionen auf den gesamten UDT aus, mit dem die Funktion bereitgestellt wird. Wird eine schreibende Funktion mit einem obligaten Ausgabeparameter verbunden, so kann dieser Funktionsaufruf als obligater Schreibzugriff auf den gesamten Wert des Datenelementes angesehen werden. Dieses Verhalten ist für die Analyse des Datenflusses von Bedeutung, da die Funktion

$ActivityWrites^{DFS}(n,d,MANDATORY)$ nicht nur TRUE zurückliefert, wenn n das gesamte Datenelement d obligat schreibt, sondern auch dann, wenn ein obligater Ausgabeparameter von n mit einer schreibenden Funktion von d verknüpft ist.

5.4.2. Versorgung von lesenden Funktionsaufrufen

Mit der Versorgung eines gesamten Datenelementes geht auch die Versorgung einzelner lesender Funktionen einher. Ein Knoten n darf also genau dann obligat auf eine UDF eines Datenelementes d zugreifen, wenn der *WriterExists*-Algorithmus für das gesamte Datenelement TRUE zurückliefert. Dass die Versorgung eines Datenelementes mit benutzerdefiniertem Datentyp sowohl durch obligates Schreiben des gesamten Datenelementes als auch durch das obligate Ausführen einer mit diesem Datentyp verknüpften Funktion gewährleistet werden kann, haben wir bereits in Abschnitt 5.4.1 festgelegt.

5.4.3. Aktivitäteninitiiertes konsumierendes Lesen

Auch auf Datenelemente mit benutzerdefiniertem Datentyp kann konsumierend zugegriffen werden. Das konsumierende Lesen eines gesamten Datenelementes entspricht dem konsumierenden Lesezugriff auf ein Datenelement mit atomarem Typ, da das System keine Informationen über die Struktur des UDT hat.

Dies hat zur Folge, dass ein konsumierender Zugriff auf eine lesende benutzerdefinierte Funktion (s. Abb. 5.11a) zum Konsumieren des gesamten Datenelementes mit benutzerdefiniertem Datentyp führt. Da ein konsumierender Lesezugriff wie ein Lesezugriff mit anschließendem Schreiben von NULL in das Datenelement angesehen werden kann, dürfen keine zwei konsumierenden Lesezugriffe von einer Aktivität auf dasselbe Datenelement ausgeführt werden.

Weiter ist es untersagt, parallel zu einem konsumierenden Lesezugriff auf weitere lesende Funktionen dieses Datenelementes zuzugreifen, da alle Lesezugriffe parallel erfolgen. Somit kann nicht entschieden werden, ob der Inhalt des Datenelementes bereits konsumiert wurde und ein weiterer Lesevorgang mit NULL versorgt würde oder ob dieser noch Daten erhält, weil der konsumierende Zugriff noch nicht abgeschlossen ist.

Wie sich konsumierende Lesezugriffe auf die Datenflussanalyse auswirken, wurde bereits in Kapitel 3 beschrieben. Ob der konsumierende Zugriff auf eine Funktion oder ein gesamtes Datenelement stattfindet, ändert nichts an der Erkennung und Behandlung dieser Zugriffe.

5.4.4. Konsumierend lesende Funktionen

Neben der Angabe *konsumierend*, die für einen Aktivitätenparameter festgelegt werden kann, ist es auch sinnvoll, diese Eigenschaft einer UDF zuzuordnen (s. Abb. 5.11b). So können Einwegfunktionen implementiert werden, nach deren Ausführung das gelesene Datenelement den Wert NULL erhält, unabhängig davon, ob die Aktivität selbst konsumierend auf die Funktion zugreift.

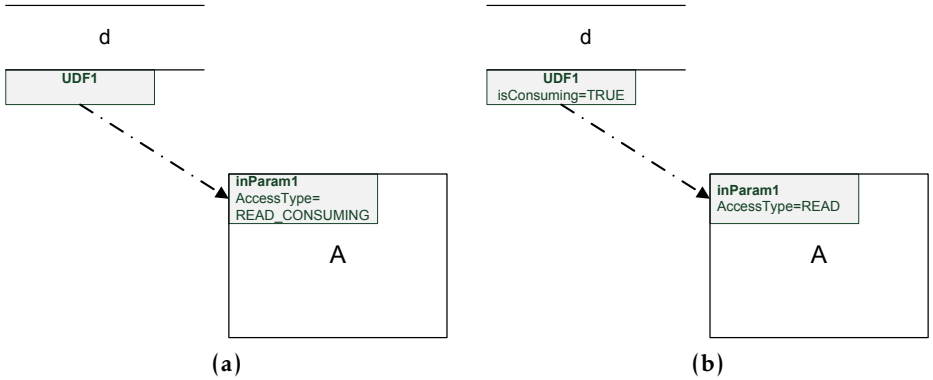


Abbildung 5.11 Konsumierender Lesezugriff (a) und lesender Zugriff auf eine konsumierend lesende (b) UDF

Dies kann genutzt werden, um etwa Funktionen zum Ver- und Entschlüsseln von Daten zu implementieren, deren Aufruf die Daten des anbietenden UDT ungültig werden lässt. Wir gehen in diesem Beispiel davon aus, dass einmal verschlüsselte Daten im Prozess nicht weiter unverschlüsselt verwendet werden sollen, es sei denn, sie werden explizit wieder entschlüsselt.

Da sich bei dieser Variante nur der Initiator des konsumierenden Lesezugriffs ändert, wirkt sie sich nicht anders auf die Datenflussanalyse aus, als der im vorigen Abschnitt beschriebene, aktivitäteninitiierte konsumierende Lesezugriff. Die Datenflussanalyse erfolgt weiterhin auf der Basis von Datenzugriffskanten, deren Zugriffsmodus auch bei konsumierend lesenden Funktionen *read_consuming* lautet.

5.5. Verwaltung

Aufgrund ihrer Konstruktion gehört jede benutzerdefinierte Funktion zwangsweise zu einem benutzerdefinierten Datentyp. Dies erschließt sich daraus, dass jede UDF genau einen UDT als Ein- bzw. Ausgabeparameter besitzt und der Typ dieses Parameters nicht variabel ist. UDFs können daher als Schnittstellen eines UDT angesehen werden. Daher liegt es nahe, benutzerdefinierte Funktionen zusammen mit benutzerdefinierten Datentypen zu verwalten. Wir beschreiben in diesem Abschnitt die Anforderungen an eine Verwaltungskomponente für UDFs, sowie die Auswirkungen der Änderung einer UDF auf vorhandene Prozessvorlagen und -instanzen.

5.5.1. Verwaltungskomponente

Die Ähnlichkeit benutzerdefinierter Funktionen zu Aktivitäten bietet uns auch bei der Verwaltung die Möglichkeit, auf die bereits vorhandenen Konzepte zurückzugreifen. Analog zum Aktivitäten-Repository kann eine Anwendungskomponente für die Verwaltung von benutzerdefinierten Funktionen entwickelt werden. Diese Verwaltungs-

komponente übernimmt neben dem Import benutzerdefinierter Funktionen in das System auch die Aufgaben der Konfiguration benutzerdefinierter Funktionen, sowie deren Versionierung. In der Verwaltungskomponente für benutzerdefinierte Datentypen kann dann auf die UDF-Verwaltung zurückgegriffen werden, um eine UDF mit einem UDT zu verknüpfen. Diese Verknüpfung erfolgt analog zum Einfügen von Aktivitäten in Prozessvorlagen.

Damit ist auch die Wiederverwendung von benutzerdefinierten Funktionen für verschiedene benutzerdefinierte Datentypen möglich, sofern es die Funktion erlaubt, den Typ des UDTs zu verändern, welchen sie als Ein- bzw. Ausgabeparameter erwartet. Ob eine derart generische Implementierung benutzerdefinierter Funktionen sinnvoll ist, muss sich aber erst in der Praxis zeigen.

5.5.2. Evolution

Änderungen an benutzerdefinierten Funktionen wirken sich auch auf Prozessvorlagen und -instanzen aus, in denen diese bereits verwendet werden. Wir beschreiben daher nachfolgend zunächst, wie sich Änderungen an UDFs auf Prozessvorlagen auswirken. Anschließend erläutern wir, unter welchen Voraussetzungen laufende Prozessinstanzen auf eine neues Prozessschema migriert werden dürfen, in welchem eine UDF geändert wurde.

Einbringen von Änderungen an benutzerdefinierten Funktionen in Prozessvorlagen

Das Hinzufügen einer UDF zu einem UDT wirkt sich nicht auf dessen Verwendung in Prozessvorlagen aus. Da es keine Pflicht gibt, jede Funktion mindestens einmal aufzurufen, kann eine UDF auch unverknüpft bleiben. Dies ist vergleichbar mit einem strukturierten Datentyp, bei dem Teile seiner Felder innerhalb eines Prozesses nie geschrieben oder gelesen werden.

Das Ändern der Implementierung einer benutzerdefinierten Funktion ist für die Verwendung in Prozessvorlagen ebenfalls nicht von Bedeutung, solange sich ihr prozessseitig sichtbarer Datentyp nicht ändert. Dabei hat der UDF-Entwickler sicherzustellen, dass die Semantik der UDF trotz der geänderten Implementierung bestehen bleibt. Seitens des Systems wird keine Überprüfung der Semantik vorgenommen.

Eine benutzerdefinierte Funktion darf nur dann von einem sich in Verwendung befindlichen UDT entfernt werden, wenn sie nicht mit einer Aktivität verbunden ist. Eine automatische Anpassung durch das Entfernen aller mit der gelöschten Funktion verbundenen Datenkanten ist möglich.

Migration laufender Instanzen auf Prozessvorlagen mit geänderten UDFs

Wurde eine Prozessvorlage nach einer Änderung an einer UDF aktualisiert, so müssen die vorgenommenen Änderungen ggf. auf bereits laufende Prozessinstanzen übertragen werden. Wir gehen im Folgenden o. B. d. A davon aus, das der Kontrollflussgraph der Prozessvorlage nicht geändert wurde.

Das Ändern einer UDF in einer laufenden Prozessinstanz ist genau dann erlaubt, wenn sich keine Instanz der UDF im Zustand `RUNNING` befindet. Bereits abgeschlossenen Instanzen der geänderten UDF sind von der Änderung nicht betroffen, noch auszuführende Instanzen instantiieren bereits die geänderte UDF. Dies ergänzt die in Kapitel 2 erläuterten Korrektheitskriterien der Schemaevolution aus [Rin04].

5.6. Weitere Aspekte

5.6.1. Dauerhaft versorgte benutzerdefinierte Funktionen

Für benutzerdefinierte Funktionen gelten im Allgemeinen die selben Korrektheitskriterien wie für den direkten Zugriff auf Datenelemente über normale Datenzugriffskanten. Diese Kriterien können gelockert werden, wenn für eine Funktion garantiert werden kann, dass sie auch dann ein korrektes Ergebnis liefert, wenn der Datentyp mit dem sie verknüpft ist, selbst nicht mit Daten versorgt ist (Beispiel 8).

Beispiel 8 (Anwendung benutzerdefinierter Funktionen) *Für einen benutzerdefinierten Datentyp wird eine Funktion implementiert, die überprüft, ob es sich bei dem derzeit in diesem Datenelement hinterlegten Wert um gültige Daten in einem bestimmten Format handelt. Dabei stellt der Wert* `NULL` keine *gültigen Daten im Sinne dieses Formats dar. Die UDF liefert also* `TRUE` *zurück, wenn es sich um gültige Daten handelt und* `FALSE` *in allen anderen Fällen.*

In einem Spezialfall, wie er in Beispiel 8 dargestellt ist, darf auch obligat lesend auf eine UDF zugegriffen werden, wenn das dazugehörige Datenelement vorher nicht geschrieben wurde. Da diese Erlaubnis unabhängig vom Ergebnis der Korrektheitsanalyse des Datenflusses erteilt werden kann, ist von der Analyse selbst abzusehen. Der untersuchende Algorithmus ist so anzupassen, dass er einer dauerhaft versorgten Funktion immer die Versorgtheit bescheinigt, ohne die Analyse durchzuführen. Diese Eigenschaft einer UDF ist anhand eines entsprechenden Konfigurationswertes zu erkennen. Da die Abfrage dieser Eigenschaft trivial ist, verzichten wir auf eine ausführliche Erläuterung der Anpassung des Algorithmus.

5.6.2. Berechnete Datenelemente

Als *berechnete Datenelemente* bezeichnet man solche Datenelemente, deren Wert erst beim Zugriff ermittelt wird. Da ihr Wert unabhängig von einer vorausgehenden Versorgung ermittelt wird, kann dieses Konzept durch die im vorangehenden Abschnitt beschriebenen dauerhaft versorgten Funktionen realisiert werden. Dabei ist sicherzustellen, dass berechnete Datenelemente nicht über normale Datenzugriffskanten geschrieben und gelesen werden dürfen. Ein Schreibzugriff auf ein berechnetes Datenelement ist untersagt, da der geschriebene Wert nicht mehr gelesen werden kann. Der lesende Zugriff ist dagegen nur auf die UDF gestattet, da diese die Berechnung durchführt. Das Datenelement selbst ist nie mit Daten versorgt, da es nicht geschrieben werden darf.

Über berechnete Datenelemente ist auch ein transparenter Zugriff auf extern gehaltene Daten, etwa auf ein DMS[4] möglich. Dabei kann die sichere Versorgung des berechneten Datenelementes, respektive der berechneten UDF, nicht mehr systemseitig gewährleistet werden. Das System muss sich in diesem Fall darauf verlassen, dass beim Zugriff auf eine berechnete UDF sicher Daten geliefert werden, sofern sie mit einem obligaten Eingabeparameter verbunden ist.

Dies ist für Funktionen unproblematisch, die selbst für die Berechnung eines Wertes sorgen. Beispiele dafür sind ein Zufallszahlengenerator, sowie ein selbst inkrementierender Zähler. Werden die Daten durch den Zugriff auf die berechnete UDF aus einem externen System gelesen, so muss im Falle eines Fehlers dafür gesorgt werden, dass obligate Eingabeparameter trotzdem mit einem gültigen Wert versorgt werden. Um eine solch umfangreiche Fehlerbehandlung aus dem Datenfluss fernzuhalten, empfehlen wir den Zugriff auf externe Systeme über geeignete Aktivitäten zu implementieren.

5.7. Zusammenfassung

Durch das Konzept benutzerdefinierter Funktionen werden Aktivitäten in die Lage versetzt, mit benutzerdefinierten Datentypen zu interagieren, auch wenn sie selbst nicht für den Umgang mit diesem UDT entwickelt wurden und deswegen keine Informationen über die Struktur des UDT besitzen. Prozessmodellierer können UDF mit bestehenden Datenkanten im Prozess verwenden und für Aktivitätenentwickler stellen sich benutzerdefinierte Funktionen vollständig transparent dar.

Die Ähnlichkeit von Aktivitäten und benutzerdefinierten Funktionen hinsichtlich der Implementierung erspart Entwicklern eine lange Einarbeitung, wohingegen die klare Aufgabentrennung der beiden Konzepte dafür sorgt, dass jedes für seine individuelle Aufgabe optimiert werden kann. Aufgrund dieser Ähnlichkeit kann die Verwaltung beider auf nahezu die selbe Art und Weise geschehen.

Aufgrund der Eingliederung benutzerdefinierter Funktionen in die derzeitigen Datenflusskonzepte können bestehende Analyseverfahren weiterhin verwendet werden. Mit dauerhaft versorgten benutzerdefinierten Funktionen und berechnende Datenelemente haben wir anhand zweier Beispiele gezeigt, wie das vorgestellte Konzept erweitert werden kann, ohne dass dabei Änderungen an den konzeptionellen Grundlagen vorgenommen werden müssen.

[4] Dokumenten-Management-System

6. Listen

Allen in ADEPT2 verfügbaren Datentypen ist gemein, dass sie zu einem Zeitpunkt immer nur einen Wert enthalten können. Auch das in Kapitel 4 vorgestellte Konzept bietet lediglich die Möglichkeit, den Wert eines Datenelementes zu strukturieren. Trotzdem ist die Struktur der Daten statisch und der enthaltene Wert kann als *Datensatz* bezeichnet werden. Abhilfe schafft das in diesem Kapitel vorgestellte Konzept der *Listen*. Listen bieten die Möglichkeit, zur Laufzeit beliebig viele Werte eines vorgegebenen Typs aufzunehmen.

Wir beschreiben zunächst, wo Listen eingesetzt werden können und diskutieren, welche Probleme sich dabei ergeben (Abschnitt 6.1). Anschließend (Abschnitt 6.2) entwickeln wir ein Konzept zur Eingliederung von Listen in das derzeitige Datenmodell von ADEPT2 und lösen die in Abschnitt 6.1 vorgestellten Probleme. Danach widmen wir uns der Analyse der Datenflusskorrektheit im Bezug auf Listen (Abschnitt 6.3), bevor wir abschließend klären, welcher Aufwand systemseitig für die Verwaltung von Listen betrieben werden muss, um diese dem Prozessmodellierer so komfortabel wie möglich zur Verfügung zu stellen (Abschnitt 6.4).

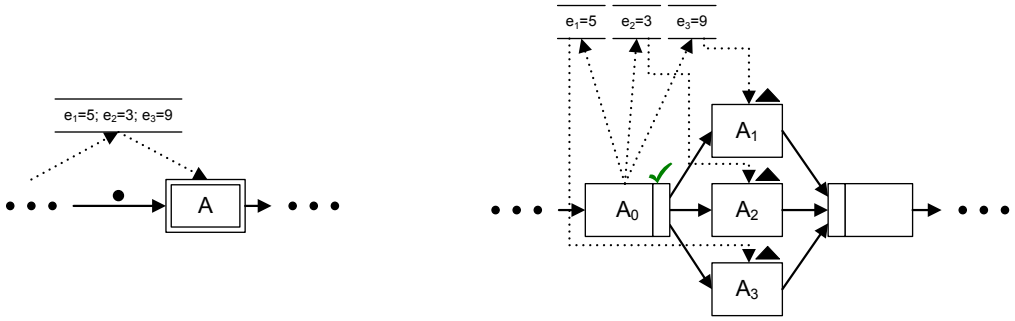
6.1. Motivation

Neben den durch variablen Parallelität gestellten Anforderungen [Wol08], gibt es für Listen vielfältige Anwendungsgebiete, etwa in Formularen für die Benutzereingabe. Wir beschreiben diese nachfolgend kurz und erläutern den Unterschied zwischen Listen und Mengen, sowie verschiedene Eigenschaften, die Listen in PMS haben können.

6.1.1. Variable Parallelität

Das in [Wol08] vorgestellte Konzept der variablen Parallelität bietet die Möglichkeit, Knoten zu modellieren, die zur Laufzeit mehrfach instantiiert werden. Die Anzahl der Instanzen steht dabei erst zur Laufzeit fest, sie entsteht also *dynamisch*. Um dies zu ermöglichen, muss zur Prozesslaufzeit jede Instanz des mehrfach instanziierten Knotens mit eigenen Daten versorgt werden.

Dies ist nur mit Datentypen möglich, die mehr als einen Wert aufnehmen können und deren Anzahl an Elementen zur Laufzeit ebenfalls dynamisch ist. Damit richtet sich die Anzahl der Instanzen nach der Anzahl der im Datenelement enthaltenen Werte. Je ein Wert wird dann an eine der Instanzen übergeben. Das Laufzeitverhalten eines



(a) Multiinstanzknoten vor Aktivierung **(b)** aktivierter Multiinstanzknoten

Abbildung 6.1 Laufzeitverhalten eines Multiinstanzknotens (variable Parallelität)

solchen *Multiinstanzknotens* ist in Abbildung 6.1 dargestellt. In [Wol08] wurden bereits die folgenden Anforderungen an einen listenwertigen Datentyp formuliert:

- Listen besitzen eine dynamische Länge.
- Auf einzelne Listenelemente kann mit einem nullbasierten Index zugegriffen werden.
- Listen haben genau einen inneren Typ, d. h. es können nur Elemente des selben Typs in der Liste gespeichert werden.
- Der Zugriff auf die gesamte Liste ist über normale Datenzugriffskanten möglich.

Weiter werden die in Tabelle 6.1 dargestellten Funktionen auf Listen gefordert, deren Eigenschaften und Probleme im Folgenden kurz erläutert werden.

Funktion `get(int index)`

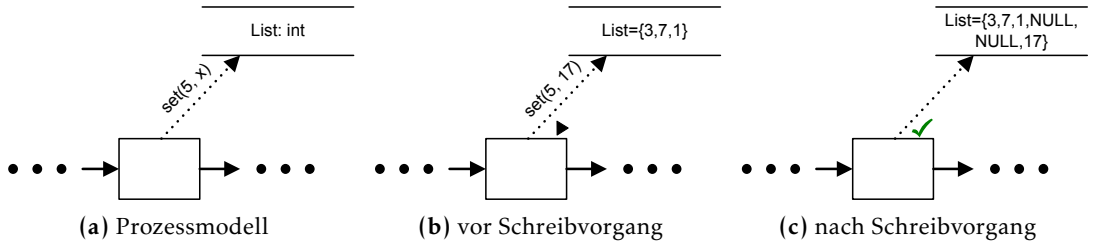
Der obligate Zugriff auf einzelne Listenelemente kann nur gestattet werden, wenn bereits zur Modellierzeit sichergestellt werden kann, dass sich an der betreffenden Position der Liste tatsächlich ein Element befindet. Dies lässt sich über die Summe der bisher ausgeführten `add(TYPE element)`-Operationen berechnen. Diese Überprüfung ist für Indizes unmöglich, welche erst zur Laufzeit ermittelt werden.

Funktion `getOptional(int index)`

Im Gegensatz zu obligaten Zugriffen auf einzelne Listenelemente ist ein optionaler Zugriff immer möglich, da für diesen zur Modellierzeit nicht analysiert werden muss, ob das Listenelement tatsächlich mit Daten versorgt ist. Dennoch gilt auch für optionale indizierte Zugriffe, deren Index zur Modellierzeit feststeht, dass diese äquivalent mit einem anderen Datenflusskonstrukt abgebildet werden können.

Funktion	Rückgabetyp	Beschreibung
`get(int index)`	TYPE	Liefert das Element an Position `index`. Ein ungültiger Index führt zu einem Laufzeitfehler
`getOptional(int index)`	TYPE	Liefert das Element an Position `index` oder `NULL`, falls der Index ungültig ist.
`set(int index, TYPE element)`	-	Setzt das Listenelement an Position `index` auf das übergebene Element.
`add(TYPE element)`	-	Fügt das übergebene Element am Ende der Liste an.
`clear()`	-	Entfernt alle Einträge aus der Liste.
`getLength()`	int	Liefert die Anzahlt der Elemente in der Liste zurück

Tabelle 6.1 Schnittstelle für den Zugriff auf Daten vom Typ `List` (nach [Wol08])



(a) Prozessmodell **(b)** vor Schreibvorgang **(c)** nach Schreibvorgang

Abbildung 6.2 Schreiben eines Wertes an eine Listenposition mit zur Modellierzeit festgelegtem Index

Funktion `set(int index, TYPE element)`

Das Setzen eines Listenelementes mit einem zur Modellierzeit vorgegebenen Index kann zur Folge haben, dass zwischen dem bis dato letzten Element der Liste und dem nun hinzugefügten Element Lücken entstehen. Dies ist der Fall, wenn der zur Modellierzeit vorgegebene Index größer ist, als die Anzahl der Listenelemente, die zum Zeitptukt des Schreibzugriffes in der Liste enthalten sind (s. Abb. 6.2). Sowohl die Analyse als auch die Bedenken eines solchen Vorhabens entsprechen denen obligater indizierter Lesezugriffe. Für das Hinzufügen eines Listenelementes mit einem Listenindex, der zur Modellierzeit noch nicht bekannt ist, gilt, dass nicht überprüft werden kann, ob dadurch Lücken entstehen.

Funktion `add(TYPE element)`

Die Funktion `add(TYPE element)` dient dem Hinzufügen eines einzelnen Listenelementes am Ende der Liste. Für diese Operation kann jederzeit garantiert werden, dass keine Lücken entstehen, sofern sie obligat ausgeführt wird. Bei einer optionalen Ausführung wird der Liste ein Element mit dem Wert `NULL` hinzugefügt, falls beim Beenden der ausführenden Aktivität kein Wert übergeben wird. Die Funktion ist so zu implementieren, dass das übergebene Element automatisch an der Position nach dem bisher letzten Listenelement eingefügt wird.

Funktion `clear()`

Das Löschen einer gesamten Liste entspricht im Wesentlichen dem Löschen des Inhaltes eines Datenelementes. Da jeweils davon ausgegangen werden kann, dass die ausführende Aktivität zunächst mit den Daten arbeitet, bevor der Löschvorgang durchgeführt wird, kann die Funktion mit einem konsumierenden Lesezugriff auf das gesamte Datenelement mit listenwertigem Datentyp gleichgesetzt werden.

Funktion `getLength()`

Da eine Liste in jedem Zustand eine Länge besitzt (initial ist diese 0), kann die Funktion an jeder Stelle im Prozess obligat verwendet werden (vgl. Abschnitt 5.6.1). Ein optionaler Zugriff ist nicht sinnvoll, da bereits sichergestellt ist, dass der Lesezugriff mit Daten versorgt sein wird.

6.1.2. Listen in Benutzeroberflächen

Neben der internen Verwendung für variable Parallelität finden Listen auch Anwendung bei der Interaktion mit Benutzern. Ein intuitives Beispiel stellt der Warenkorb eines Internet-Versandhandels dar. Dieser kann mit beliebig vielen Artikeln befüllt werden, die dann nach der Bestellung einzeln verpackt werden müssen.

Da die Umsetzung von Listen für unterschiedliche Oberflächen dem Aktivitäten- oder Anwendungsentwickler obliegt, gehen wir hier nicht weiter darauf ein. Wir achten jedoch darauf, dass Listen als Datentyp alle Informationen bereitstellen, die für die Generierung eines entsprechenden Anzeigeelementes benötigt werden.

6.1.3. Unterschied zwischen Listen und Mengen

Sowohl Listen als auch Mengen erfüllen zunächst das Kriterium, eine variable Anzahl von Elementen aufnehmen zu können. Dennoch unterscheiden sich beide in einigen Merkmalen, die für das weitere Vorgehen in diesem Kapitel relevant sind.

Eigenschaften von Listen

Eine Liste ist eine geordnete Ansammlung von Elementen, wobei dasselbe Element mehrfach in einer Liste auftreten kann. Die Reihenfolge der Elemente ist durch ih-

re Position in der Liste festgelegt. Neue Elemente können in eine Liste an beliebiger Position eingefügt werden. Dadurch werden einerseits bereits vorhandene Elemente überschrieben, wenn ein Element an einer Stelle eingefügt wird, an der sich vor dem Einfügen bereits ein Element befunden hat. Andererseits entstehen Lücken, wenn die Position des neu eingefügten Elements als mehr um 1 größer ist, als die Position des derzeit letzten Elementes der Liste. Lücken entstehen auch dann, wenn Elemente aus der Liste entfernt werden.

Alternativ können Listen auch so definiert werden, dass sie keine Lücken enthalten. Das Hinzufügen neuer Elemente ist dann nur am Ende der Liste möglich oder an einer Position, die bereits mit einem Element belegt ist. Das Löschen eines Elementes aus der Liste hat zur Folge, dass diese *kollabiert*. Dadurch verringert sich die Länge der Liste, sowie die Position aller nachfolgenden Elemente um 1. Für unsere Anwendung ist von Bedeutung, dass die Länge der Liste dynamisch entsteht und als Summe aus Anzahl der Elemente und Anzahl der Lücken berechnet werden kann.

Eigenschaften von Mengen

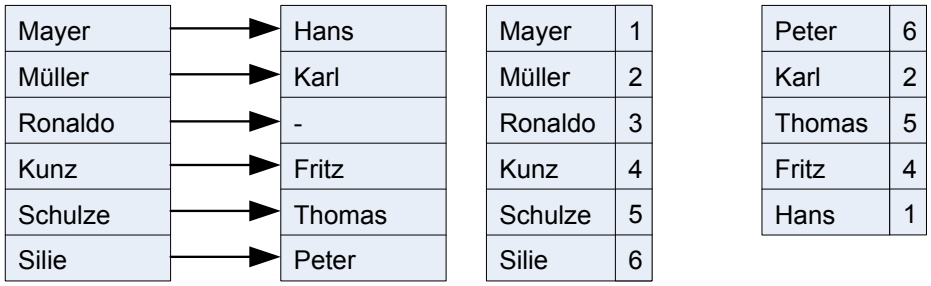
Wie Listen können auch Mengen beliebig viele Elemente aufnehmen, wobei ein Element in einer Menge nur einmal vorkommen kann. Die Elemente einer Menge haben keine Reihenfolge, weswegen auch das Hinzufügen und Entfernen von Elementen nicht über deren Speicherposition erfolgen kann. Aufgrund der fehlenden Reihenfolge können in Mengen keine Lücken entstehen. Ist eine Menge so definiert, dass einzelne Elemente mehrmals vorkommen können, so spricht man von einer *Multimenge*.

6.1.4. Lücken in Listen

In unserem Anwendungsfall erfordern Listen besondere Aufmerksamkeit, die Lücken enthalten können. Wird beispielsweise die Instanzanzahl eines Knotens mit variabler Parallelität durch eine Liste festgelegt, die Lücken enthält, so kann nicht jeder Instanz ein Element der Liste zugeordnet werden. Dieses Verhalten ist zulässig, wenn die am Knoten hinterlegte Aktivität die Listenelemente als optionale Eingabeparameter erwartet [Wol08]. Wird auf diese obligat zugegriffen, ist ein `NULL`-Element als Eingabeparameter unzulässig.

Werden Lücken in Listen zugelassen, so ergeben sich für die Definition von obligaten und optionalen Zugriffen erhebliche Schwierigkeiten. Lücken können jedoch hilfreich sein, wenn Elemente verschiedener Listen einander zugeordnet werden sollen (s. Abb. 6.3a). Dies birgt jedoch die Gefahr, dass durch das Ändern einzelner Listenelemente die Semantik der Zuordnung verändert wird. Sicherer kann diese über Fremdschlüsselbeziehungen, äquivalent zu relationalen Datenbanken [EN09], gelöst werden (s. Abb. 6.3b).

Daher schließen wir Lücken in listenwertigen Datentypen an dieser Stelle aus. Wo das Fehlen von Lücken die Handhabung des Datenflusses vereinfacht, weisen wir gesondert darauf hin.



(a) über Listenposition **(b)** über Fremdschlüssel

Abbildung 6.3 Zuordnung von Listenelementen verschiedener Listen

6.1.5. Indizierter Zugriff auf einzelne Listenelemente

Der indizierte Zugriff auf einzelne Listenelemente kann auf mehreren Ebenen geschehen. Im Prozessmodell kann auf einzelne Listenelemente zugegriffen werden, indem entweder zur Modellierzeit ein Wert als Index direkt mit dem Listenzugriff verknüpft wird, oder indem ein indizierter Zugriff modelliert wird, dessen Index zur Laufzeit über einen Parameter gegeben ist[1]. Der parametrisierte Zugriff auf prozesslogischer Ebene kann dafür genutzt werden, paralleles Schreiben auf Listen zu erlauben, sofern der Zugriff auf unterschiedliche Listenelemente erfolgt.

Weiter ist auch auf der Ebene von Aktivitäten ein indizierter Zugriff auf einzelne Listenelemente möglich, nachdem die Aktivität bereits das gesamte listenwertige Datenelement gelesen hat. Dieses Konzept bietet Aktivitäten die Möglichkeit, auf einzelne Elemente zuzugreifen, ohne die Liste selbst interpretieren zu müssen.

Wir werden in den nun folgenden Abschnitten detailliert klären, wie indizierte Zugriffe auf Listenelemente zustande kommen und welche Probleme dabei entstehen. In Abschnitt 6.2.2 zeigen wir dann, wie diese Probleme gelöst werden können.

Indizierter Zugriff zur Modellierzeit mit festem Indexparameter

Der indizierte Zugriff zur Modellierzeit erfolgt, wenn der Parameter einer Aktivität direkt mit einem festen Listenelement verbunden wird (s. Abb. 6.4). Für diesen Zugriff wird vom Prozessmodellierer ein Index angegeben. Der Parameter wird dann zur Laufzeit mit den Daten des Listenelements versorgt, das sich an der angegebenen Position befindet.

Offensichtlich entsteht ein Problem, wenn ein obligater Zugriff auf eine Listenposition erfolgt, an der zur Laufzeit kein Element abgelegt ist. Weiter könnte es auch geschehen, dass der vom Modellierer angegebene Index größer ist, als die Länge der Liste. Auch in diesem Fall ist ein obligater Zugriff nicht möglich.

[1] Dies schließt bereits aus, Listenzugriffe als UDF umzusetzen, da diese keine parametrisierten Aufrufe gestatten.

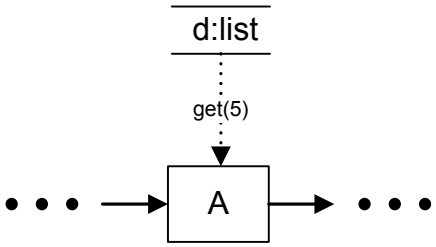


Abbildung 6.4 Indizierter Zugriff auf ein Listenelement mit zur Modellierzeit feststehendem Index.

Indizierter Zugriff zur Modellierzeit mit variablem Indexparameter

Meist steht jedoch beim Modellieren eines Prozesses noch nicht fest, an welcher Position in der Liste sich das Element befindet, auf welches eine Aktivität zugreifen soll. Daher besteht auch die Möglichkeit, die Verknüpfung eines Aktivitätenparameters mit einem Datenelement zu parametrisieren. Der Parameter der Verknüpfung ist dann der Index des zu lesenden Listeneintrags. Dieser Parameter muss dann im Prozessverlauf vor dem Lesezugriff der Aktivität mit Daten versorgt werden, damit es nicht mangels Indexangabe schon vor dem eigentlichen Zugriff zu einem Fehler kommt. Die Probleme, die beim parametrisierten Zugriff auf ein Listenelement entstehen können, entsprechen denen beim indizierten Zugriff zur Modellierzeit.

Indizierter Zugriff auf Aktivitätenebene

Neben dem indizierten Zugriff auf einzelne Listenelemente eines Datenelementes besteht auch die Möglichkeit, dass eine Liste von einer Aktivität als Ganzes gelesen wird. Dann benötigt die Aktivität eine Schnittstelle, um auf die einizelnen Elemente der gelesenen Liste zugreifen zu können.

Wie in den beiden vorherigen Szenarien besteht auch hier die Möglichkeit, dass eine Liste Lücken enthält. Eine entsprechende Definition, welche Anforderungen von einer Aktivität an eine gelesene Liste gestellt werden können, wird in Abschnitt 6.2.2 gegeben. Aus prozesslogischer Sicht treten zunächst keine weiteren Probleme auf, da eine als Ganzes gelesene Liste wie ein Datenelement mit atomarem Typ behandelt werden kann.

Paralleles Schreiben von listenwertigen Datenelementen

Wie bereits erwähnt, sind parallele Schreibzugriffe nur unter bestimmten Voraussetzungen gestattet [Rei00]. Dies ändert sich jedoch bei Datentypen, deren Werte nicht atomar sind (vgl. 4.3.3). Zu diesen Datentypen gehören auch Listen. Der parallele Schreibzugriff auf unterschiedliche Teilbereiche ist, wie auch bei strukturierten Datentypen, möglich. Die Analyse paralleler Schreibzugriffe gestaltet sich jedoch schwieriger als bei strukturierten Datentypen, da zunächst festgestellt werden muss, ob es sich bei den geschriebenen Listenelementen tatsächlich um Elemente mit unterschiedlichem

Index und damit um unterschiedliche Elemente handelt. Ob und unter welchen Bedingungen eine solche Analyse erfolgen kann und wie sichergestellt werden kann, dass dasselbe Listenelement nicht in parallelen Zweigen, ohne Synchronisation der Schreibzugriffe geschrieben wird, diskutieren wir in Abschnitt 6.3.3.

6.2. Realisierung

Nachdem wir nun gesehen haben, wo listenwertige Datentypen zur Anwendung kommen, welche Eigenschaften diese aufweisen und wie der Zugriff auf Datenelemente mit listenwertigem Datentyp aussehen kann, beschäftigen wir uns im Folgenden mit der Umsetzung dieses Datentyps und seiner Eingliederung in das derzeitige Datenmodell. Dabei beschreiben wir, wie sich Listen im Verhältnis zu bestehenden Datentypen darstellen lassen, wie die geforderten Methoden zur Verfügung gestellt werden können und welches Konzept für indizierte Zugriffe realisierbar ist.

6.2.1. Einordnung in das derzeitige Datenmodell

Listen stellen in sofern eine Besonderheit dar, als dass sie immer genau zwei Datentypen zur selben Zeit besitzen, einen äußeren und einen inneren Datentyp.

Äußerer Datentyp

Alle Listen besitzen gemeinsam einen *äußeren* Datentyp, der sie als Listen mit den dazugehörigen Merkmalen identifiziert. Dieser Datentyp bescheinigt einem Datenelement die Fähigkeit, zur Laufzeit beliebig viele Daten aufzunehmen und einem Parameter, dass er über entsprechende Methoden verfügt, auf diese Listenelemente zuzugreifen. Der äußere Datentyp einer Liste entspricht dem `AdeptDataType`. Dieser wird mit `List` benannt. Seine Schnittstelle wird in 6.2.3 beschrieben.

Innerer Datentyp

Der *innere* Datentyp einer Liste zeigt an, von welchem Typ die einzelnen Elemente sind. Eine Liste besitzt genau einen inneren Datentyp, weswegen auch immer nur Elemente genau eines Typs in einer Liste gespeichert werden können.

Identifizierung von listenwertigen Datentypen

Zur Identifizierung eines listenwertigen Datentyps sind sowohl der äußere als auch der innere Datentyp notwendig. Dies stellt sicher, dass sowohl der Funktionsumfang der Liste als auch der Datentyp der einzelnen Elemente bekannt sind. Daher wird eine Liste immer als *List of TYPE* identifiziert, wobei *TYPE* den inneren Datentyp der Liste angibt.

Bedeutung der Reihenfolge von Listenelementen

Wir haben bereits festgelegt, dass Listen mit Lücken in unserem Anwendungsfall ausgeschlossen sind. Für Eingabedaten eines Multiinstanzknotens ist die Reihenfolge der Listenelemente unerheblich. Auch für die Anzeige der Listenelemente an der Benutzeroberfläche ist die Reihenfolge der Listenelemente nicht von semantischer Bedeutung. Wir legen daher allgemein fest, dass die Reihenfolge der Listenelemente keine semantische Bedeutung besitzt. Ist eine solche Reihenfolge erforderlich, bspw. um die einzelnen Listenelemente verschiedener Listen einander zuzuordnen, so muss diese Semantik als Werte innerhalb der einzelnen Liste hergestellt werden (vgl. Fremdschlüsselbeziehungen in Datenbanken [EN09]).

6.2.2. Indizierter Zugriff

Die Probleme indizierter Zugriffe wurden bereits in Abschnitt 6.1 ausführlich besprochen. Wir klären nun, welche der Konzepte realisierbar sind und mit welchen Einschränkungen diese Realisierung verbunden ist. Dabei unterscheiden wir wieder zwischen indizierten Zugriffen, für die der Index bereits beim Modellieren bekannt ist und solchen, deren Index erst zur Prozesslaufzeit ermittelt wird. Das Konzept des indizierten Zugriffs auf Aktivitätenebene, bei dem der Index nicht im Prozessmodell hinterlegt ist, wird anschließend diskutiert.

Indizierter Zugriff mit zur Modellierzeit bekannten Indexparametern

Wie in Abschnitt 6.1 bereits angedeutet, ist der Nutzen dieses Konzeptes fraglich. Ist dem Modellierer bekannt, an welcher Stelle in der Liste sich ein von ihm zu verwendender Wert zur Laufzeit befinden wird, so kann dieser Wert gleichwertig in einem separaten, nicht listenwertigen Datenelement gespeichert werden, dessen Datentyp dem inneren Typ der Liste entspricht. Dies erspart eine aufwendige Analyse der Listenoperationen, die für die sichere Versorgung eines bestimmten Listeneintrages notwendig ist.

Beispiel 9 (Semantische Bedeutung von Listenpositionen) *In einem strukturierten Datentyp* `Firmenkontakt` *soll eine variable Anzahl von Telefonnummern gespeichert werden. Bekannt ist, dass jedem Firmenkontakt mindestens eine Telefonnummer zugeordnet ist (die Durchwahl zur Zentrale). Weiter soll die Möglichkeit gegeben sein, weitere Telefonnummern zu einem Firmenkontakt zu speichern, etwa für weitere Ansprechpartner.*

Die erste Idee führt dazu, dem Firmenkontakt eine Liste von Telefonnummern hinzuzufügen. Dabei wird dem ersten Listenelement die Bedeutung „Durchwahl Zentrale“ zugeordnet. Ein lesender Zugriff auf das erste Listenelement liefert somit immer die Telefonnummer der Zentrale der Firma.

Bei der Optimierung dieser Modellierung entfällt die semantische Bedeutung der ersten Listenposition: Die Telefonnummer der Zentrale wird in einem eigenen Feld `tel_zentrale` *gespeichert. Alle weiteren Telefonnummern werden in einer Liste abgelegt. Die Reihenfolge der zusätzlichen Telefonnummern kann dann beliebig sein.*

Gibt es in einer Liste beispielsweise Einträge, deren Position zur Modellierzeit bekannt ist und welche, deren Position erst dynamisch zur Laufzeit feststeht, so kann dies mit einem strukturierten Datentyp realisiert werden. Für alle bereits zur Modellierzeit bekannten Daten wird ein eigenes Feld in diesem Datentyp angelegt. Für die Daten, deren Position zur Modellierzeit unbekannt ist, kann ein weiteres Feld mit einem listenwertigen Datentyp angelegt werden. Wir gewähren daher keinen indizierten Zugriff auf Listenelemente mit einem zur Laufzeit bekannten Index, sondern verweisen für diesen Zweck auf bereits verfügbare Konstrukte (s. Kapitel 4).

Indizierter Zugriff mit zur Modellierzeit unbekanntem Indexparameter

Ist die Analyse der Datenflusskorrektheit für indizierte Zugriffe mit zur Modellierzeit bekanntem Parameter schon schwierig, ist sie für einen zur Modellierzeit unbekannten Parameter unmöglich, sofern er zur Modellierzeit frei von jeglichen Beschränkungen ist. Die einzige Möglichkeit, Aussagen über den Parameter für einen indizierten Zugriff treffen zu können, besteht darin, Bedingungen für seinen Wert zu formulieren. Wird der indizierte Zugriff dann nur abhängig von diesen Bedingungen ausgeführt, so kann analysiert werden, ob der Zugriff sicher mit Daten versorgt sein wird. Zwar ist eine solche Analyse möglich, jedoch kann der indizierte Zugriff dann nur unter sehr komplexen Bedingungen erfolgen, durch die der Index für den Zugriff soweit eingeschränkt wird, wodurch dieses Konzept nahezu unbrauchbar wird.

Beispielsweise könnte ein obligater Lesezugriff auf einen Index zwischen 0 und 4 nur dann erlaubt werden, wenn der Liste vorher mindestens 5 Listenelemente hinzugefügt wurden. Weiter darf die Liste zwischen dem ersten Hinzufügen eines Elementes und dem lesenden Zugriff nicht als Ganzes geschrieben werden.

Indizierter Zugriff auf Aktivitätenebene

Für die sinnvolle Verwendung von Listen ist ein indizierter Zugriff der Aktivitäten auf einzelne Listenelemente unumgänglich. Dieser ist im Vergleich zu indizierten Zugriffen auf Prozessebene unkritisch, da lediglich sichergestellt werden muss, dass die gesamte Liste mit Daten versorgt ist und die Versorgung nicht für ein einzelnes Listenelement überprüft werden muss. Dabei kann die Aktivität durch obligaten oder optionalen Zugriff auf die Liste festlegen, ob die Liste mindestens ein Element enthalten muss und, im Falle eines strukturierten inneren Typs, welche Felder in jedem Listeneintrag sicher mit Daten versorgt sein müssen.

6.2.3. Zugriff auf Listen aus Sicht des Aktivitätenentwicklers

Im Folgenden beschreiben wir die Schnittstelle, die einem Aktivitätenentwickler für den Zugriff auf Parameter mit listenwertigem Datentyp zur Verfügung steht. Dabei orientieren wir uns an den Vorschlägen aus [Wol08], setzen diese jedoch nur um, soweit dies sinnvoll ist. Eine Zusammenfassung dieser Methoden findet sich im Anhang in Tabelle C.2.2.

Funktion `getLength()` Um die Gültigkeit von Indizes für indizierte Zugriffe auf Listenelemente zu überprüfen, benötigt der Entwickler Zugriff auf die Listenlänge. Diese liefert die Funktion `getLength()`. Sie liefert auch dann einen Wert, wenn die Liste noch nicht initialisiert wurde oder keine Elemente mehr enthält. In beiden Fällen ist die Länge 0.

Funktion `get(int index)` Für den indizierten Zugriff auf Listenelemente steht dem Implementierer von Aktivitäten die Funktion `get(int index)` zur Verfügung. Sie liefert das Element der Liste an Position `index` zurück. Der Index ist nullbasiert, weswegen auf das erste Element der Liste mit `get(0)` zugegriffen wird. Der Index muss daher immer kleiner als die Länge der Liste sein. Das zurückgegebene Listenelement ist vom Typ `TYPE`, dem inneren Typ der Liste. Der Zugriff mit einem ungültigen Index führt zu einem Laufzeitfehler.

Funktion `add(TYPE element)` Fügt das angegebene Element am Ende der Liste hinzu. Da eine Liste keine leeren Elemente enthalten kann, darf der übergebene Wert nicht `NULL` sein.

Funktion set(int index, TYPE element) Fügt das angegebene Element an der angegebenen Position in die Liste ein. Das vorher an dieser Position gespeicherte Element wird überschrieben. Wie auch beim Hinzufügen von Elementen am Ende der Liste darf das einzufügende Element nicht `NULL` sein, da das Vorhandensein von nullwertigen Listenelementen beim Zurückschreiben in das Datenelement zu einem Laufzeitfehler führt. Ein Laufzeitfehler entsteht auch dann, wenn der angegebene Index nicht kleiner als die Listenlänge ist, da sonst leere Listenelemente zwischen dem bisher letzten Element der Liste und dem neu eingefügten Element entstehen.

Funktion `delete(int index)` Soll ein Element aus der Liste gelöscht werden, muss die Funktion `delete(int index)` verwendet werden. Die Funktion hinterlässt keine Lücke an der Stelle des gelöschten Listenelementes. Somit verringert sich die Länge der Liste durch den Aufruf dieser Funktion um 1. Das Entfernen eines Listenelementes mit einem Index größer oder gleich der Listenlänge führt zu einem Laufzeitfehler.

Funktion `clear()` Zum schnellen Löschen einer gesamten Liste kann die Funktion `clear()` verwendet werden. Sie ist äquivalent zum wiederholten Löschen des Elementes an Position 0, bis die Länge der Liste 0 ist. Eine leere Liste darf nicht als obligater Ausgabeparameter verwendet werden, weswegen vor Beendigung der Aktivität mindestens wieder ein Element in die Liste eingefügt werden muss. Ist eine Liste in einem optionalen Ausgabeparameter beim Beenden der Aktivität `NULL`, so hat dies zur Folge, dass der optionale Schreibzugriff nicht durchgeführt wird. Der Wert des Datenelementes, welches mit dem optionalen Ausgabeparameter verbunden ist, wird folglich nicht verändert.

6.2.4. Zugriff auf Listen aus Sicht des Prozessmodellierers

Indizierten Zugriff auf Listen auf prozesslogischer Ebene haben wir bereits in Abschnitt 6.2.2 ausgeschlossen. Aus Sicht des Prozessmodellierers können Listen daher wie Datenelemente mit atomarem Datentyp behandelt werden. Jedoch sind nicht alle Aktivitäten in der Lage, mit listenwertigen Datentypen umzugehen. Nachdem wir aber diesen Aktivitäten nicht gänzlich den Umgang mit Listen verwehren wollen, bieten wir für Listen eine Funktion an, mit deren Hilfe Elemente am Ende einer Liste angefügt werden können.

Dies ist möglich, da das Hinzufügen von Listenelementen keinen Einfluss auf die bereits in der Liste enthaltenen Elemente hat. Außerdem benötigt eine Aktivität für das Aufrufen der Funktion keinerlei Informationen über die Länge oder den Inhalt der Liste. Lediglich der innere Typ der Liste muss der Aktivität bekannt sein. Darüber hinaus haben wir in Abschnitt 6.2.1 festgelegt, dass die Reihenfolge der Listenelemente keine semantische Bedeutung besitzt.

Die listeninterne Funktion `add(TYPE Element)` kann aus prozesslogischer Sicht verwendet werden, wie eine benutzerdefinierte Funktion aus Kapitel 5. Sie wird über eine Datenkante mit dem Ausgabeparameter der Aktivität verknüpft, der vom Typ TYPE sein muss. Nach Beendigung der Aktivität wird der in diesem Parameter bereitgestellte Wert an die Funktion übergeben und von dieser an das Ende der Liste angefügt.

Es liegt zunächst nahe, diese Funktion so zu implementieren wie eine Aktivität, die einen Eingabeparameter vom Typ TYPE und einen Ausgabeparameter vom Typ `List of` TYPE besitzt. Das schließt jedoch das Hinzufügen einzelner Listenelemente aus parallelen Zweigen aus, was nicht praktikabel ist. Daher ist die Funktion `add(TYPE Element)` als interne Funktion der Liste zu implementieren, wodurch parallele, nicht synchronisierte Schreibzugriffe unterstützt werden können.

6.3. Datenflussanalyse

In diesem Abschnitt wird die Bedeutung obligater und optionaler Parameter im Kontext von Listen erläutert. Weiter wird erklärt, welche Voraussetzungen erfüllt sein müssen, damit ein obligater Eingabeparameter mit listenwertigem Datentyp als *versorgt* angesehen werden kann.

6.3.1. Obligate und optionale Zugriffe

Nachdem der innere Datentyp einer Liste nicht nur atomar, sondern auch strukturiert sein kann, muss zwischen den *äußeren* und den *inneren* Versorgungseigenschaften unterscheiden werden. Zusicherungen für den äußeren Datentyp beziehen sich dabei auf das Vorhandensein von Elementen in der Liste, wohingegen Zusicherungen für den inneren Datentyp angeben, welche Annahmen für einzelne Felder eines strukturierten inneren Datentyps gemacht werden können. Die nachfolgenden Zugriffsarten beziehen sich immer auf Listen als Ganzes.

	atomarer innerer Typ	strukturierter innerer Typ
obligat	• Länge der Liste > 0 • jedes Element ≠ NULL	• Länge der Liste > 0 • jedes Element ≠ NULL • alle obligaten Felder ≠ NULL • ohne Regel: alle Felder obligat
optional	• Länge der Liste ≥ 0 • jedes Element ≠ NULL	• Länge der Liste ≥ 0 • jedes Element ≠ NULL • alle obligaten Felder ≠ NULL • ohne Regel: alle Felder obligat

Tabelle 6.2 Anforderungen von listenwertigen Eingabeparameter an die zur Laufzeit an sie übergebenen Daten

Listenwertige Eingabeparameter

Listenwertige Eingabeparameter stellen gewissen Anforderungen an die zur Laufzeit an sie übergebenen Daten. Diese Anforderungen werden im Folgenden erklärt. Dabei wird zwischen obligaten und optionalen Eingabeparametern unterschieden. Die Anforderungen sind in Tabelle 6.2 zusammengefasst.

Obligate Eingabeparameter Für einen listenwertigen Eingabeparameter legt die Eigenschaft *obligat* fest, dass eine zur Laufzeit an sie übergebene Liste mindestens ein Element enthalten muss. Bei listenwertigen Eingabeparametern mit strukturiertem innerem Typ können einzelne Felder als obligat bzw. optional gekennzeichnet werden. Für die Versorgung muss also nicht nur ein vorausgehender obligater Schreibzugriff sicherstellen, dass die Liste mindestens ein Element enthält, sondern es müssen alle vorausgehenden optionalen und obligaten Schreibzugriffe mindestens die vom Eingabeparameter als obligat geforderten Felder obligat schreiben.

Optionale Eingabeparameter Ein optionaler Eingabeparameter mit listenwertigem Datentyp unterscheidet sich von einem obligaten Eingabeparameter lediglich dadurch, dass die an ihn zur Laufzeit übergebene Liste auch die Länge 0 haben darf. Die weiteren Anforderungen, insbesondere bei strukturiertem innerem Datentyp, entsprechen denen von obligaten Eingabeparametern. Daher muss auch für optionale listenwertige Eingabeparameter eine Datenflussanalyse durchgeführt werden, wenn Felder des inneren Datentyps als obligat gekennzeichnet sind.

Listenwertige Ausgabeparameter

Ausgabeparameter mit listenwertigem Datentyp gewährleisten bestimmte Eigenschaften der Daten, die durch sie in ein verknüpftes Datenelement geschrieben werden. Wie auch bei listenwertigen Eingabeparametern wird zwischen obligaten und optionalen Ausgabeparametern unterschieden. Die nachfolgend beschriebenen Zusicherungen sind in Tabelle 6.3 zusammengefasst.

Obligate Ausgabeparameter Wird ein Ausgabeparameter mit listenwertigem Datentyp als *obligat* gekennzeichnet, so sichert dies zu, dass die durch diesen Parameter geschriebene Liste mindestens ein Element enthält. Für atomare innere Datentypen bedeutet dies weiter, dass jedes Listenelement Daten enthält. Folglich gibt es kein Element mit dem Wert `NULL`.

Wird für obligate listenwertige Ausgabeparameter mit strukturiertem innerem Typ nichts weiter angegeben, so bedeutet dies, dass jedes Feld in jedem Listenelement obligat geschrieben wird. Analog zu dem in Abschnitt 4.3.1 vorgestellten Konzept können einzelne Felder eines listenwertigen Ausgabeparameters mit strukturiertem innerem Typ als obligat bzw. optional gekennzeichnet werden. Wird ein Feld als obligat (optional) gekennzeichnet, so gilt diese Festlegung für alle Listenelemente gleichermaßen.

Optionale Ausgabeparameter Ein optionaler Ausgabeparameter mit einem listenwertigen Datentyp zeigt an, dass die zugehörige Aktivität eventuell keine Werte in die Liste schreibt. Wird der Schreibzugriff ausgeführt, so enthält die Liste anschließend mindestens ein Element. Unterbleibt dieser, so bleibt der Wert des verknüpften Datenelementes unverändert.

Auch für optionale Schreibzugriffe kann festgelegt werden, welche Felder bei strukturiertem innerem Datentyp obligat geschrieben werden. Dies eröffnet Aktivitäten die Möglichkeit anzuzeigen, dass sie zwar eventuell keine Daten erzeugen, aber dass die Daten, falls diese erzeugt werden, die angegebenen Bedingungen erfüllen.

6.3.2. Versorgung von Lesezugriffen

Wir beschreiben nun die Datenflussanalyse für Parameter und Datenelemente mit listenwertigem Datentyp. Dabei beschränken wir uns auf Listen, deren innerer Typ strukturiert ist. Die Analyse für Listen mit atomarem innerem Datentyp entspricht derer von Parametern mit atomarem Datentyp.

Sowohl für obligate als auch für optionale Eingabeparameter muss analysiert werden, ob ihre Anforderungen an die Versorgung des inneren Datentyps durch vorausgehende Schreiboperationen erfüllt werden. Weiter muss für obligate Eingabeparameter untersucht werden, ob durch vorausgehende Schreiboperationen sichergestellt wird, dass in dem verknüpften listenwertigen Datenelement beim Zugriff mindestens ein Element enthalten ist.

	atomarer innerer Typ	strukturierter innerer Typ
obligat	• Länge der Liste > 0 • jedes Element ≠ NULL	• Länge der Liste > 0 • jedes Element ≠ NULL • alle obligaten Felder ≠ NULL • ohne Regel: alle Felder obligat
optional	• Länge der Liste > 0 oder Wert des Datenelementes bleibt unverändert • jedes Element ≠ NULL	• Länge der Liste > 0 oder Wert des Datenelementes bleibt unverändert • jedes Element ≠ NULL • alle obligaten Felder ≠ NULL • ohne Regel: alle Felder obligat

Tabelle 6.3 Zusicherungen listenwertiger Ausgabeparameter über die Beschaffenheit der geschriebenen Daten

Voraussetzungen für die Versorgung optionaler Eingabeparameter

Für die Versorgung von optionalen Eingabeparametern kommen Aktivitäten in Frage, welche die folgenden Bedingungen erfüllen:

1. Die Aktivität ist in der Vorgängermenge der lesenden Aktivität enthalten.
2. Die Aktivität schreibt das mit dem Eingabeparameter verknüpfte Datenelement.
3. Die von der Aktivität obligat geschriebenen Felder des inneren Typs der Liste sind eine Obermenge der obligaten Felder des zu untersuchenden Eingabeparameters.

Voraussetzungen für die Versorgung obligater Eingabeparameter

Für die Versorgung obligater Eingabeparameter kommen keine Aktivitäten in Frage, welche das verknüpfte Datenelement nur optional schreiben. Zwar garantiert ein optionaler Schreibzugriff auch das Vorhandensein mindestens eines Listenelementes, sofern dieser zur Laufzeit tatsächlich erfolgt, jedoch ist nicht sichergestellt, dass Schreibzugriff auch tatsächlich ausgeführt wird. Abgesehen von der Einschränkung der Versorgermenge entspricht die Analyse der Datenflusskorrektheit für obligate listenwertige Eingabeparameter derer für optionale Eingabeparameter.

6.3.3. Parallele Schreibzugriffe

Parallele Schreibzugriffe auf Datenelemente mit listenwertigem Datentyp sind ausschließlich dann zulässig, wenn sichergestellt werden kann, dass niemals dasselbe Lis-

tenelement aus unterschiedlichen Zweigen geschrieben wird. Dies ist nur möglich, wenn die für den indizierten Zugriff verwendeten Parameter zur Prozesslaufzeit bekannt sind oder wenigstens analysiert werden können. Da wir einen parametrisierten Zugriff auf prozesslogischer Ebene ausgeschlossen haben, kann auch paralleles Schreiben verschiedener Listenelemente nicht direkt realisiert werden.

Wenn jedoch zur Modellierzeit feststeht, dass sich die Schreibzugriffe zweier parallel ausgeführter Aktivitäten auf unterschiedliche Bereiche einer Liste beziehen, so können diese Schreiboperationen äquivalent in zwei getrennte Listen geschrieben werden, die nach Abschluss der parallelen Ausführung wieder zu einer Liste zusammengefügt werden können.

Darüber hinaus ist das parallele Hinzufügen von Listenelementen über die Funktion `add(TYPE Element)` erlaubt, da es sich hierbei um eine kommutative Operation[2] handelt. Echtes paralleles Schreiben listenwertiger Datentypen kann somit auch weiterhin nicht direkt umgesetzt werden. Ein entsprechendes Konzept kann mit der Entwicklung von Synchronisationsverfahren für atomare Datentypen einhergehen.

6.3.4. Konsumierendes Lesen listenwertiger Datentypen

Das konsumierende Lesen von listenwertigen Datentypen entspricht dem konsumierenden Lesen von atomaren Datentypen. Da kein indizierter Zugriff auf Listenelemente erlaubt ist, können diese nicht einzeln konsumiert werden. Auch gibt es keine Möglichkeit anzuzeigen, dass nur auf einzelne Felder aller Listenelemente konsumierend zugegriffen wird, da nur die Liste als Ganzes gelesen werden kann.

Sollen nur Teile der Listenelemente konsumiert werden, so müssen die nicht konsumierten Felder nach Beendigung der Aktivität wieder obligat in die Liste zurückgeschrieben werden. Das bedeutet, dass sowohl ein lesender, als auch ein schreibender Zugriff von der Aktivität auf die gesamte Liste erfolgt.

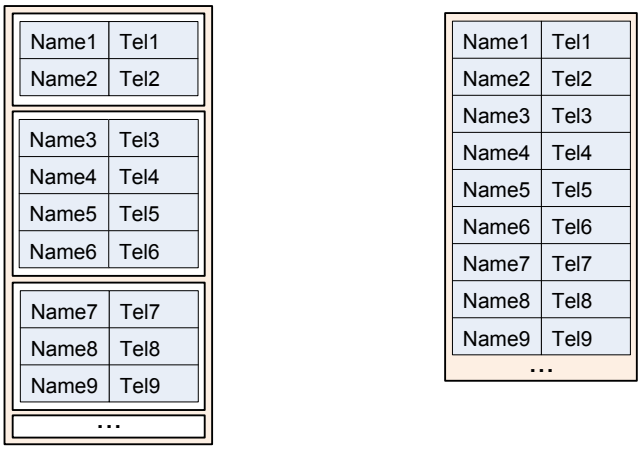
6.3.5. Inkonsistenzen durch partielle Schreibzugriffe

Wie schon bei den in Kapitel 4 vorgestellten strukturierten Datentypen können auch bei Listen mit strukturiertem innerem Typ Inkonsistenzen auftreten. Das Zustandekommen dieser Inkonsistenzen entspricht dem bei strukturierten Datentypen, da die Anzahl der Elemente in einer Liste für die Datenflussanalyse unerheblich ist. Auch für Listen kann die Untersuchung auf mögliche inkonsistente Daten mit demselben *WriterExists*-Algorithmus erfolgen, wie bei strukturierten Datentypen.

6.3.6. Typkompatibilität

Datenelemente und Parameter mit listenwertigem Datentyp sind genau dann zueinander kompatibel, wenn der innere Datentyp TYPE der beiden übereinstimmt. Da die

[2] add(A),add(B) = add(B),add(A), da die Position eines Elementes in der Liste keine semantische Bedeutung besitzt.



(a) Liste von Listen **(b)** aggregierte Listen

Abbildung 6.5 Aggregation von geschachtelten Listen

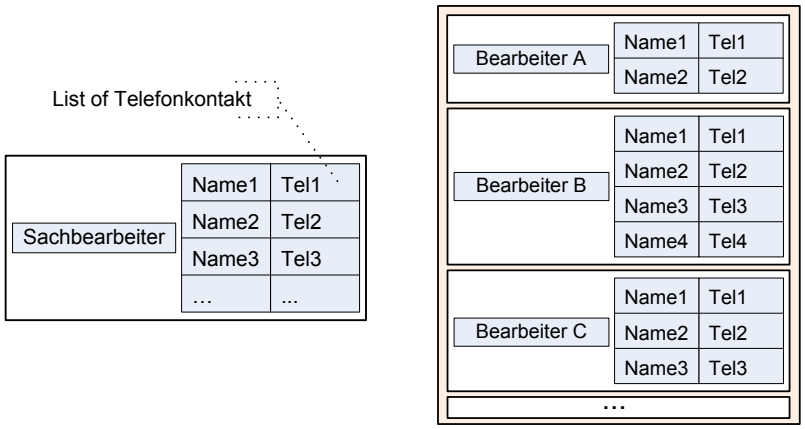
Funktionsweise aller Listen identisch ist, muss nur sichergestellt werden, dass die zugreifende Aktivität mit dem Datentyp der einzelnen Listenelemente kompatibel ist.

6.4. Verwaltung

Listen bieten, unabhängig von ihrem inneren Datentyp, immer die selben Zugriffsmethoden. Daher müssen Listen, im Gegensatz zu strukturierten Datentypen und UDTs, nicht verwaltet werden. Es reicht aus, sie über ihren inneren Typ zu identifizieren. Sowohl für listenwertige Parameter als auch für listenwertige Datenelemente, reicht es aus, dass beim Modellieren der innere Typ der Liste angegeben wird. Bei Aktivitätenparametern kann dieser auch vom Aktivitätenentwickler festgelegt werden. Listen eines bestimmten Typs können dann über eine Kombination der Identifizierungsmerkmale für Listen und den jeweiligen inneren Datentyp auf Kompatibilität geprüft werden.

Um jedoch die in Kapitel 4 vorgestellten Metadaten, etwa zum Markieren eines Datentyps als *veraltet*, auch für Listen nutzen zu können, müssen in der Datentypverwaltung auch Datentypen angelegt werden können, die vom Typ `List of TYPE` sind. Ein derart angelegter Datentyp bekommt ein neues Identifikationsmerkmal und ist daher nicht kompatibel zu anderen Listen des selben inneren Typs.

Es wird dabei nicht gestattet, Listen anzulegen, deren innerer Datentyp selbst wieder eine Liste ist. Dies hat den Grund, dass eine derartige `List of List of TYPE` äquivalent ist zu einer `List of TYPE`. Würde das Schachteln von Listen erlaubt, so müsste der innere Typ aller inneren Listen identisch sein (s. Abb. 6.5a). Damit kann aber auch der innere Typ der inneren Listen als innerer Typ einer nicht geschachtelten Liste verwendet werden, um das gleiche Ergebnis zu erreichen (s. Abb. 6.5b). Ist eine Unterteilung einer Liste in mehrere Teillisten erforderlich, so muss für jede Teilliste ein eigenes



(a) Bearbeitungsliste **(b)** List of Bearbeitungsliste

Abbildung 6.6 Schachtelung von Listen mit Zusatzinformationen zu inneren Listen

Datenelement verwendet werden. Dies stellt auch sicher, dass die unterschiedlichen Teillisten aus parallelen Zweigen heraus geschrieben werden können.

Gestattet ist jedoch die Verwendung von Listen als Felddatentyp eines strukturierten Datentyps. Damit können auch Listen von Listen mit Zusatzinformationen angefertigt werden, was sie für Knoten mit variabler Parallelität verwendbar macht. So kann etwa eine Liste von zu kontaktierenden Kunden in mehrere Listen unterteilt werden, um jede Teilliste einem anderen Sachbearbeiter zuzuordnen (s. Abb. 6.6).

6.5. Zusammenfassung

Die Untersuchungen in diesem Kapitel haben gezeigt, dass der Einsatz von Listen in ADEPT2 möglich ist, wobei das Konzept der Listen dafür eingeschränkt werden muss. Ein indizierter Zugriff auf prozesslogischer Ebene ist nur dann analysierbar, wenn er in einem Prozessschema verwendet wird, das starken Einschränkungen unterliegt. Aufgrund der hohen Komplexität und des dadurch geringen Nutzens können Listen aus prozesslogischer Sicht als ein atomarer Wert angesehen werden. Das Fehlen indizierter Zugriffsmöglichkeiten führt auch dazu, dass nur lückenlose Listen unterstützt werden, was zu einer weiteren erheblichen Vereinfachung der Datenflussanalyse führt.

Ungeachtet dessen bieten Listen die Möglichkeit, Daten aus parallelen Zweigen in einem Datenelement zu vereinen, sowie die Instanzen Verzweigungen mit variabler Parallelität mit Daten zu versorgen. Durch die einfache Handhabung von Listen auf Aktivitätenebene stellen diese auch für Aktivitätenentwickler ein überschaubares Konzept dar.

7. Aufzählungen

In Kapitel 4 haben wir strukturierte Datentypen vorgestellt, die zunächst nur abstrakt im System vorhanden sind. Erst durch das Anlegen eines neuen strukturierten Datentyps durch den Administrator entsteht eine konkreter Datentyp, der für Datenelemente und Aktivitätenparameter verwendet werden kann. Auch bei *Aufzählungen* handelt es sich um einen solchen abstrakten Datentyp, für dessen Verwendung zunächst konkrete Ausprägungen vom Administrator definiert werden müssen. Aufzählungen sind endliche Mengen, deren Objekte eindeutige Namen tragen. Für Datenelemente mit Aufzählungsdatentyp steht fest, welche Werte sie zur Prozesslaufzeit annehmen können. Dabei kann der Wert eines Datenelementes zu jedem Zeitpunkt genau einem Element der Menge entsprechen.

7.1. Motivation

Derzeit gibt es in ADEPT2 keine Möglichkeit, endliche Mengen darzustellen, deren Struktur dem System bekannt ist. Zwar gibt es die Möglichkeit, das Konzept mit vorhandenen Konstrukten umzusetzen, jedoch leidet darunter wie schon bei strukturierten Datentypen die Anwenderfreundlichkeit der Prozessmodellierung. Aufzählungen erweisen sich jedoch für die folgenden beiden Anwendungsfälle als nützlich.

7.1.1. Mengenabhängige Verzweigungen

Die Verzweigungsentscheidung bei XOR- und ODER-Verzweigungen kann derzeit auf der Grundlage jedes atomaren Datentyps erfolgen. Allerdings muss dabei vom Prozessmodellierer für jeden Zweig ein Prädikat angegeben werden, welches zur Laufzeit ausgewertet wird. Diese Verzweigungsprädikate decken den Wertebereich des zur Verzweigung verwendeten Datentyps vollständig ab. Es gibt bisher keine Möglichkeit, diese Prädikate automatisch anhand von Datenstrukturen zu erzeugen. Dem System ist nicht bekannt, in welcher Weise der Wertebereich des für die Verzweigungsentscheidung verwendeten Datentyps aufgeteilt werden soll.

Daher ist es sinnvoll, dem System einen Datentyp bekannt zu machen, dessen Elemente *eindeutig* sind und dessen *Wertebereich* beschränkt ist. Damit ist es möglich, Verzweigungen automatisch zu erzeugen. Wird der XOR-Verzweigungsknoten mit einem Datenelement verknüpft, dessen Datentyp eine Aufzählung ist, so kann das System automatisch einen Zweig für jeden in diesem Datenelement möglichen Wert erzeugen.

7.1.2. Automatisch generierte Auswahllisten in Formularen

Auswahllisten, wie sie unter anderem in Formularen verwendet werden, bieten dem Anwender eine endliche Menge von Werten zur Auswahl an. Häufig werden diese Werte vom Prozessmodellierer für jedes Formular manuell eingegeben oder sie werden in einer Datenbank hinterlegt, damit sie in unterschiedlichen Formularen verwendet werden können.

ADEPT2 bietet die Möglichkeit, Formulare zur Benutzereingabe anhand von Datenstrukturen automatisch zu erzeugen. Dafür wird eine spezielle Aktivität verwendet, für die eine variable Anzahl von Ein- und Ausgabeparametern festgelegt werden kann. Wird eine solche Aktivität gestartet, so erzeugt sie für jeden Parameter ein graphisches Darstellungselement, das zu dessen Datentyp passt. So wird für einen Parameter vom Typ `String` ein Textfeld erzeugt, für einen Parameter vom Typ `boolean` ein Kontrollkästchen (*checkbox*). Die Darstellungselemente für Eingabeparameter sind schreibgeschützt und zeigen die Werte der mit den Eingabeparametern verbundenen Datenelemente an. Die Darstellungselemente für Ausgabeparameter können vom Benutzer ausgefüllt werden. Ihre Werte werden nach Beendigung der Aktivität in die verknüpften Datenelemente übertragen.

Das automatische Generieren einer Auswahlliste setzt voraus, dass ein Datentyp existiert, dessen Wertebereich endlich ist und dessen mögliche Werte bereits beim Generieren des Formulars bekannt sind. Ein solcher Datentyp ist bisher in ADEPT2 nicht verfügbar.

7.1.3. Probleme abstrakter Datentypen

Da es sich bei Aufzählungen, wie auch bei komplexen Datentypen, um abstrakte Datentypen handelt, treten hier zunächst die selben Probleme auf. Für eine ausführliche Diskussion verweisen wir auf Abschnitt 4.1.6. Um diese Probleme möglichst einfach handhabbar zu machen, ist es sinnvoll, alle möglichen Werte einer Aufzählung beim Zugriff einer Aktivität verfügbar zu machen. Damit kann auch eine Aktivität, welche die möglichen Werte der Aufzählung nicht kennt, mit einer solchen Aufzählung umgehen und die Werte korrekt verarbeiten.

7.2. Realisierung

Wir beschäftigen uns nun damit, wie *Aufzählungen* auf vorhandene Datenstrukturen abgebildet werden können und wie deren Verwendung im System aussehen wird.

7.2.1. Umsetzung auf vorhandene Datenstrukturen

Durch die in den vorherigen Kapiteln vorgestellten Konzepte gibt es mehrere Möglichkeiten, Aufzählungen darzustellen. Wir diskutieren im Folgenden kurz, wie diese Möglichkeiten aussehen und beschreiben danach detailliert, wie die von uns gewählte

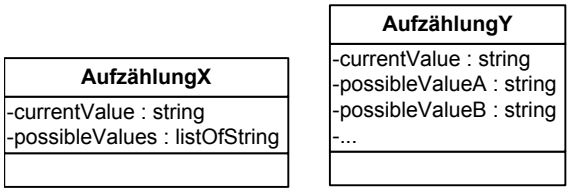


Abbildung 7.1 Umsetzung von Aufzählungen auf komplexe Datentypen

Möglichkeit in das bestehende Datenmodell integriert wird. Unabhängig von der gewählten Realisierung wird die Funktion der Aufzählung in einem eigenen Datentyp namens `ENUM` dargestellt, sodass die zugrundeliegende Umsetzung auf andere Datentypen vor dem Anwender verborgen bleibt.

Realisierung als komplexe Datenstruktur

Wird eine Aufzählung als komplexer Datentyp realisiert, so benötigen wir ein Feld für den gerade aktuellen Wert, sowie eine Liste, in der die möglichen Werte der Aufzählung bereitgestellt werden (s. Abb. 7.1a). Eine Realisierung als reiner strukturierter Datentyp ist ebenfalls möglich, wenn für jeden möglichen Wert ein Feld bereitgestellt wird (s. Abb. 7.1b). Dies bietet die Möglichkeit, den Feldnamen als Werte der Aufzählung zu verwenden, die Werte der Felder können dann beispielsweise für eine textuelle Beschreibung der Aufzählungswerte dienen. Es sind weitere Realisierungsformen, beispielsweise mit der Übersetzung der Beschreibung in unterschiedliche Sprachen denkbar. Diese werden jedoch aufgrund des begrenzten Umfangs dieser Arbeit nicht besprochen.

Die Reihenfolge der einzelnen Werte entspricht ihrem Vorkommen in der Liste bzw. der Reihenfolge der Datenfelder im strukturierten Datentyp. Lesender Zugriff auf das Datenelement bedeutet ein Auslesen des aktuell gesetzten Wertes. Ein lesender Zugriff auf die möglichen Werte der Aufzählung ist ebenfalls möglich. Ein schreibender Zugriff ist nur auf den aktuellen Wert möglich, da die Liste der möglichen Werte zur Laufzeit nicht verändert werden darf.

Realisierung als eingeschränkte Ganzzahl

Mit dem Konzept der eingeschränkten Datentypen aus Kapitel 4 ist es möglich, einen eingeschränkten, ganzzahligen Datentyp zu definieren. Für eine Aufzählung mit fünf Elementen wäre also eine Einschränkung auf den Wertebereich $[0 \ldots 4]$ notwendig. Da neben der Information über den gerade gespeicherten Wert auch das Wissen über die Bedeutung der einzelnen Werte verfügbar sein muss, müssen diese anderweitig im System hinterlegt werden. Eine Möglichkeit, diese direkt im Datentyp zu speichern, gibt es nicht.

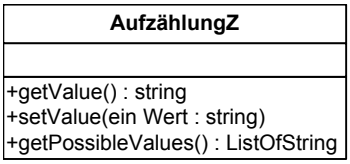


Abbildung 7.2 Umsetzung von Aufzählungen als UDT

Realisierung als Liste

Ähnlich wie die Kombination aus strukturiertem Datentyp mit enthaltener Liste, lässt sich auch eine Liste von strukturierten Datentypen erstellen. Der Datentyp der Listeneinträge stellt dann mindestens zwei Felder bereit. Das erste Feld gibt den Namen des Aufzählungselementes an, das zweite ist vom Typ `Boolean` und gibt an, ob der Eintrag gerade aktiv ist. Die Liste besteht somit aus Tupeln der Form (`wert`, `istGesetzt`).

Wird ein Wert in ein Datenelement mit einem auf diese Weise realisierten Aufzählungstyp geschrieben, so wird das zu diesem `wert` gehörende `istGesetzt`-Feld auf `TRUE` gesetzt, die `istGesetzt`-Felder aller anderen Einträge werden mit `FALSE` überschrieben.

Wie auch bei der Realisierung als komplexe Datentstruktur muss auch bei der hier vorgestellten Variante sichergestellt werden, dass weder die Felder für die Werte überschrieben werden, noch dass mehr als ein `istGesetzt`-Feld auf `TRUE` gesetzt wird. Daher muss der schreibende Zugriff so gekapselt werden, dass das Schreiben eines Wertes zu oben genanntem Verhalten führt.

Mit dieser Variante ist es technisch auch möglich, Aufzählungen mit Mehrfachauswahl zu realisieren. Da dies jedoch nicht den eingangs genannten Anforderungen entspricht, verweisen wir auf die Möglichkeit, ein derartiges Konstrukt mit den bestehenden Datentypen selbst zu realisieren.

Realisierung als benutzerdefinierter Datentyp

In Kapitel 5 wurde gezeigt, wie UDTs mit Funktionen versehen werden können, welche den Zugriff auf einzelne Bestandteile des benutzerdefinierten Datentyps erlauben. Diese können verwendet werden, um eine Aufzählung zu kapseln.

Der abstrakte Datentyp `Aufzählung` wird realisiert, indem ein UDT definiert wird, welcher nach außen hin die Funktionen `getValue()` und `setValue(Wert)`, sowie eine Funktion `getPossibleValues` für den Zugriff auf alle möglichen Werte des Datentyps bietet. Wird nun der rein lesende Zugriff auf ein Datenelement von einem derart konstruierten Aufzählungstyp auf die Funktion *getValue()* umgeleitet und geschieht selbiges für den schreibenden Zugriff und die Funktion *setValue(Wert)*, so ist sichergestellt, dass weder mehrere Werte gleichzeitig ausgewählt sind, noch dass die Liste der zur Verfügung stehenden Werte überschrieben wird.

Realisierung als eigenständiger Datentyp

Neben den bereits vorgestellten Varianten, die allesamt das Konzept der Aufzählung auf bereits bestehende Datenkonstrukte umsetzen, besteht noch die Möglichkeit, Aufzählungen als gänzlich eigenständigen Datentyp zu realisieren. Diese Realisierungsform ist der mit UDTs sehr ähnlich. Der neue Datentyp liefert beim lesenden Zugriff den aktuellen Wert des Datenelementes zurück und prüft beim Schreiben, ob der geschriebene Wert gültig ist. Der Zugriff auf alle möglichen Werte für einen Aufzählungstyp kann nur erfolgen, wenn für diesen Datentyp ein Äquivalent zu benutzerdefinierten Funktionen aus Kapitel 5 zur Verfügung steht.

7.2.2. Entscheidung für eine Umsetzungsvariante

Die Realisierungsvariante als eingeschränkte Ganzzahl ist aufgrund der Tatsache ungeeignet, dass es nicht möglich ist, die Namen der einzelnen Werte direkt mit dem Datentyp zur Verfügung zu stellen. Wird die Aufzählung als Liste dargestellt, so wird der aktuelle Wert durch die Gesamtheit der `isSet`-Felder dargestellt. Die Modifikation aller dieser Felder zur Laufzeit erfordert wesentlich mehr Aufwand, als das Schreiben eines einzelnen Wertes.

Die drei noch verbleibenden Konzepte sind sich sehr ähnlich, da sie jeweils die Möglichkeit bieten, den aktuell gesetzten Wert zu lesen und zu schreiben, sowie lesend auf die Liste der möglichen Werte zuzugreifen. Sowohl für die Realisierung als UDT als auch für die Umsetzung als komplexen Datentyp, muss das jeweilige Konzept so erweitert werden, dass der Zugriff auf Teile der Daten eingeschränkt werden kann. Bei beiden Varianten müssen die Bereiche der Daten, in denen die möglichen Werte der Aufzählung gespeichert sind, vor Schreibzugriffen geschützt werden. Darüber hinaus müssen bei beiden Datentypen Funktionen entfernt werden, damit beispielsweise bei als UDT realisierten Aufzählungen keine benutzerdefinierten Funktionen, bei Aufzählungen, die als komplexer Datentyp realisiert sind, keine zusätzlichen Felder hinzugefügt werden können. Daher entscheiden wir uns für die Implementierung eines neuen Datentyps, dessen Funktionsweise an die Umsetzung als UDT angelehnt ist. Soweit sinnvoll werden auch Konzepte aus anderen Realisierungsvarianten verwendet.

7.2.3. Detaillierte Beschreibung der Realisierung

Beim Datentyp `Aufzählung` handelt es sich, wie eingangs bereits erwähnt, um einen abstrakten Datentyp. Im Falle der Aufzählung bedeutet dies, dass zwar die Schnittstellen und die Art des Zugriffs auf die Daten für alle Aufzählungen identisch sind, dass jedoch der Wertebereich von Aufzählung zu Aufzählung verschieden ist. Dies unterscheidet Aufzählungen von strukturierten Datentypen, deren Struktur von Typ zu Typ unterschiedlich ist.

Folglich besitzt jeder Aufzählungsdatentyp je ein Feld für den aktuellen Wert und eines für die Liste der möglichen Werte. Die Besonderheit des Feldes für den aktuellen Wert ist, dass es nicht einzeln angesprochen werden muss: Sowohl der lesende als auch der schreibende Zugriff auf ein gesamtes Datenelement, dessen Datentyp von

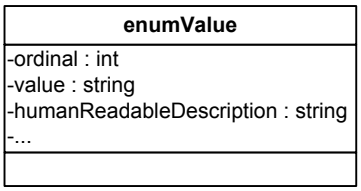


Abbildung 7.3 Datentyp `enumValue`, der den Wert eines Aufzählungsfeldes, sowie dessen Beschreibung aufnimmt. Weitere Felder sind möglich.

`Aufzählung` abgeleitet ist, entsprechen dem Zugriff auf das Feld, das den aktuellen Wert enthält.

Auch die Liste der möglichen Werte muss nicht explizit mit einer Aktivität verbunden werden. Greift eine Aktivität lesend oder schreibend auf ein Datenelement zu, dessen Typ eine Aufzählung ist, so steht dort automatisch die Liste der möglichen Werte zur Verfügung. Zwar ist für viele Anwendungen zur Laufzeit nur der aktuelle Wert der Aufzählung notwendig, jedoch schränkt die Verfügbarkeit der Liste aller möglichen Werte diese Anwendungen nicht ein, sofern die Liste effizient implementiert wird.

Der Datentyp `enumValue`

Bei der Verwendung von Aufzählungsdatentypen muss zwischen zwei Anwendungen unterschieden werden. In vielen Fällen kommen die Werte eines Datenelementes mit einem spezialisierten Aufzählungstyp zum Einsatz, wo Menschen mit dem System interagieren. Andererseits gibt es auch Fälle, in denen der Wert eines Datenelementes lediglich vom System ausgewertet wird, beispielsweise bei Verzweigungsentscheidungen.

Um beiden Umständen gerecht zu werden, bietet ein Datenelement lesenden Aktivitäten neben dem tatsächlich gesetzten Wert auch noch Zusatzinformationen, die dazu dienen, den gesetzten Wert für Menschen verständlich zu machen. Der aktuelle Wert einer Aufzählung ist daher als strukturierter Datentyp realisiert, der mindestens die drei Felder `ordinal`, `value` und `humanReadableDescription` enthält.

Darüber hinaus sind noch weitere Felder, beispielsweise für die Übersetzung der Beschreibung in andere Sprachen denkbar. Da diese jedoch nichts am grundlegenden Konzept der Aufzählung ändern, werden sie hier nicht weiter besprochen. Die Liste aller möglichen Werte einer Aufzählung ist folglich eine Liste vom Typ `enumValue` (s. Abb. 7.3).

Generischer Typ `ENUM` für Parameter

Da Aktivitäten Zugriff auf alle möglichen Werte einer Aufzählung haben, ist es neben der Verwendung eines speziellen Aufzählungstyps als Datentyp für Parameter auch denkbar, Parameter vom generischen Typ `ENUM` zu erzeugen (s. Abb. 7.4). Da jedoch spätestens beim Verbinden eines Parameters mit einem Datenelement feststeht, welchen

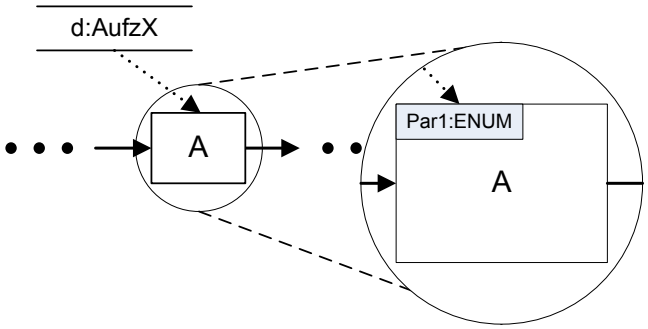


Abbildung 7.4 Verknüpfung von Datenelementen von speziellem Auzählungstyp, gekoppelt mit Parameter von generischem ENUM-Typ

konkreten Datentyp der Parameter annehmen muss, um zum Datenelement kompatibel zu sein, bieten wir die Möglichkeit des abstrakten Datentyps nicht an. Ein Datentyp, der auch zur Prozesslaufzeit noch generisch ist, bietet keine Vorteile. Es ist jedoch nicht ausgeschlossen, dass Aktivitäten implementiert werden, für deren Parameter der konkrete Datenyp erst zur Modellierzeit festgelegt wird.

Aufzählungen aus Sicht des Prozessmodellierers

Bei der Erstellung von Prozessen können Aufzählungen nahezu äquivalent zu atomaren Datentypen verwendet werden. Datenelemente mit dem Typ einer Aufzählung können direkt mit Parametern vom selben Typ verbunden werden. Dies gilt sowohl für lesende als auch für schreibende Zugriffe.

Eine Besonderheit stellt die Verwendung abstrakter Aufzählungsdatentypen für Parameter dar. Es ist dem Modellierer gestattet, Parameter vom abstrakten Typ `ENUM` mit jedem Datenelement zu verbinden, welches einen Datentyp besitzt, der vom Typ `ENUM` abgeleitet ist. Dies gilt sowohl für Eingabe- als auch für Ausgabeparameter.

Ein direkter Zugriff auf einzelne Felder des aktuellen Wertes, also beispielsweise auf das Feld `humanReadableDescription`, ist nicht sinnvoll und daher nicht gestattet. Es wird immer das gesamte strukturierte Datum aus dem Datenelement an die lesende Aktivität übergeben.

Aufzählungen aus Sicht des Aktivitätenentwicklers

Die Schnittstelle, über die der Zugriff auf Aufzählungen erfolgt, bietet einerseits Zugriff auf den aktuell gesetzten Wert, andererseits kann auf die Liste aller möglichen Werte der Aufzählung zugegriffen werden. Der schreibende Zugriff ist nur auf die eindeutige Bezeichnung des Aufzählungswertes, sowie auf dessen Ordnungszahl möglich. Die nachfolgend beschriebenen Details sind in Anhang C zusammengefasst.

Lesender Zugriff Der gerade gesetzte Wert ist als strukturiertes Datum über `parameter.getValue()` verfügbar. Die Ordnungszahl des gerade gesetzten Wertes steht

unter `parameter.getValue().ordinal` zur Verfügung. Auf die eindeutige Bezeichnung des aktuellen Wertes kann über `parameter.getValue().value` zugegriffen werden, die für Menschen lesbare Beschreibung dieses Wertes erhält man über `parameter.getValue().humanReadableDescription`. Wird die Liste aller möglichen Werte der Aufzählung benötigt, ist diese über `parameter.getPossibleValues()` erreichbar. Die zurückgegebene Liste kann wie in Kapitel 6 beschrieben verwendet werden.

Schreibender Zugriff Das Setzen des Wertes einer Aufzählung erfolgt über `parameter.setValue(String)` respektive über `parameter.setValue(int)`, wenn die Ordnungszahl verwendet werden soll. Ordnungszahl und eindeutiger Bezeichner sind korreliert, weswegen je nach Bedarf eine der beiden Funktionen zum Setzen des aktuellen Wertes verwendet werden kann. Ein schreibender Zugriff auf die für Menschen lesbare Beschreibung oder auf die Liste der möglichen Werte ist aus einer Aktivität heraus nicht erlaubt. Daher stehen für diese Operation keine Methoden zur Verfügung.

7.3. Datenflussanalyse

Da Datenelemente, deren Typ eine Spezialisierung des Typs `ENUM` darstellt, zur Laufzeit keine weiteren Funktionen bieten, als dass sie geschrieben und gelesen werden dürfen, wirkt sich dieses neue Konzept nicht auf die Korrektheit des Datenflusses aus. Lediglich die Typkompatibilität muss, wie bei jedem anderen Datentyp auch, sowohl zur Modellier- als auch zur Laufzeit sichergestellt werden.

7.3.1. Sichere Versorgung

Wie bereits eingangs erwähnt, gelten für Aufzählungsdatentypen die gleichen Korrektheitsbedingungen wie für atomare Datentypen. Da sich sowohl der lesende als auch der schreibende Zugriff implizit auf den aktuellen Wert des Datenelementes mit Aufzählungsdatentyp beziehen, kann auch der Algorithmus zur Überprüfung der Versorgung (*WriterExists*) auf diesen Wert angewendet werden.

7.3.2. Typkompatibilität

Die Typkompatibilität beim Verknüpfen von Parametern mit Datenelementen ist auch bei Aufzählungen nur dann gegeben, wenn beide vom selben Datentyp sind. Auch für Aufzählungen ist die Implementierung von Konverteraktivitäten möglich (vgl. Kapitel 5, Abb. 5.1).

Ein abstrakter Datentyp `ENUM` für Parameter ist nicht erforderlich, da zur Modellierzeit feststeht, welchen Datentyp das mit diesem Parameter verknüpfte Datenelement besitzt. So kann auch der Datentyp des Parameters zur Modellierzeit an den des Datenelementes angepasst werden. Für einen Aktivitätenparameter können jedoch bestimmte Rahmenbedingungen festgelegt werden, die der zur Modellierzeit festgelegte Datentyp erfüllen muss. Diese können auch enthalten, dass es sich beim Datentyp des Parameters um einen beliebigen Aufzählungstyp handeln muss.

7.4. Verwaltung

Wie komplexe Datentypen und benutzerdefinierte Funktionen, müssen auch Aufzählungsdatentypen verwaltet werden. Da wir viele allgemeine Details bereits in den vorangehenden Kapiteln besprochen haben, beschränken wir uns im Folgenden auf die Besonderheiten, die für Aufzählungen zu beachten sind.

7.4.1. Anlegen neuer Aufzählungstypen

Jeder neue Aufzählungsdatentyp wird vom abstrakten Datentyp `ENUM` abgeleitet. Es besteht folglich eine „Ist-ein“-Beziehung zwischen jedem Aufzählungstyp und dem Datentyp `ENUM`. Das Identifizierungsmerkmal eines Aufzählungsdatentyps setzt sich, wie bei strukturierten Datentypen auch (vgl. Kapitel 4), aus dem eindeutigen Namen und der aktuellen Versionsnummer des Datentyps zusammen.

Nachdem der neue Datentyp im System registriert ist, können seine möglichen Werte angegeben werden. Diese Werte müssen bezüglich des Datentyps eindeutig sein. Der gleiche Wert kann aber in unterschiedlichen Datentypen auftauchen. Da er jedoch in unterschiedlichen Typen eine andere Bedeutung haben kann, sind die Werte verschiedener Datentypen nicht zueinander kompatibel, auch wenn sie die gleiche Bezeichnung tragen. Die Werte werden im Feld `value` des inneren Datentyps `enumValue` der Liste aller möglichen Werte gespeichert.

Weiter kann jedem Wert eine Beschreibung zugewiesen werden, welche dann in Benutzeroberflächen angezeigt wird. Diese Beschreibung muss nicht zwingend eindeutig sein, da das System die Werte anhand des Feldes `value` unterscheiden kann. Dennoch ist es nicht sinnvoll, eine Beschreibung mehrfach zu verwenden, da der Benutzer dann nicht mehr zwischen verschiedenen Werten unterscheiden kann. Besitzt ein Wert keine Beschreibung, so wird dort der Wert selbst angezeigt, wo eine Interaktion mit dem Benutzer notwendig ist.

7.4.2. Evolution von Aufzählungstypen

Wie bei allen bisher beschriebenen Datentypen, wirkt sich auch das Ändern eines Aufzählungsdatentyps auf die Prozessmodelle aus, in welchen Datenelemente oder Aktivitäten mit Parametern diesen Typs verwendet werden. Die Evolution laufender Prozessinstanzen auf neue Prozessvorlagen wird von geänderten Aufzählungsdatentypen in der Prozessvorlage ebenfalls beeinflusst.

Ändern von Aufzählungstypen

Wir verwenden die in Abschnitt 4.4.1 eingeführte lineare Versionierung auch für Aufzählungsdatentypen. Zum Ändern wird auch von Aufzählungsdatentypen eine neue Version erzeugt. Vorgängerversionen können anschließend als *veraltet* markiert werden, um Prozessmodellierer auf die neue Version des Datentyps hinzuweisen.

Löschen von Aufzählungstypen

Das Löschen von Aufzählungsdatentypen ist nur gestattet, sofern diese weder in Prozessvorlagen, noch in strukturierten Datentypen verwendet werden. Um Modellierer darauf hinzuweisen, dass ein Aufzählungsdatentyp nicht mehr verwendet werden soll, kann dieser als *veraltet* markiert werden.

Wird ein Datentyp weder in Prozessvorlagen, noch in anderen Datentypen verwendet, gibt es aber Prozessinstanzen, in denen dieser Datentyp verwendet wird, so kann er zwar als *gelöscht* markiert werden, er wird jedoch nicht gänzlich aus dem System entfernt. Das endgültige Löschen von Datentypen ist nur gestattet, wenn diese keine Anwendung im System finden, was der Fall ist, wenn diese „aus Versehen“ oder zum Testen angelegt wurden.

Ändern von Prozessvorlagen

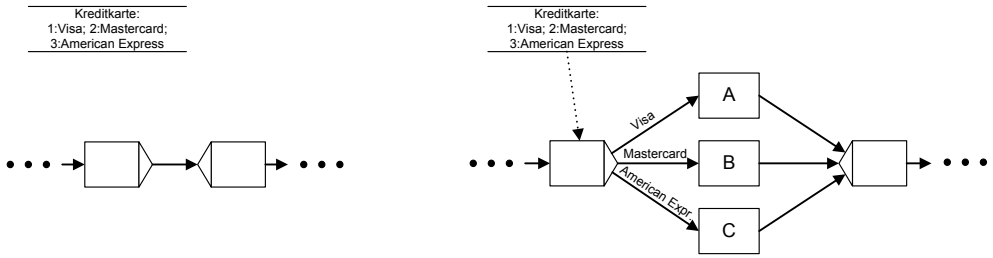
Prozessvorlagen müssen an geänderte Datentypen angepasst werden. Es müssen sowohl die Datentypen von Datenelementen als auch die Parameter von Aktivitäten aktualisiert werden. Entsprechend müssen Aktivitäten aktualisiert werden, damit diese mit den neuen Parameter-Datentypen zurecht kommen.

Auch beim Anpassen von Prozessvorlagen an neue Datentypen bzw. neue Versionen müssen die Datentypen von Datenelementen und von mit ihnen verknüpften Parametern konsistent gehalten werden. Es ist möglich, verschiedene Versionen eines Aufzählungsdatentyps im selben Prozess zu verwenden, sofern die Konsistenz von Datenelementen und Parametern weiterhin gewahrt ist.

Evolution von Prozessinstanzen

Haben sich Datentypen in einer Prozessvorlage geändert, auf deren Basis noch laufende Instanzen vorhanden sind, so muss für diese geprüft werden, ob sie auf das neue Schema migriert werden können. Da jede neue Version eines Aufzählungsdatentyps zu einem neuen Identifizierungsmerkmal führt, dürfen auch Instanzen mit Aufzählungen nur dann auf das neue Prozessschema migriert werden, wenn noch keine mit dem Datenelement verknüpfte Aktivität einen der Zustände `RUNNING` oder `COMPLETED` erreicht hat. Nur dann können noch alle mit dem Datenelement verknüpften Aktivitäten migriert werden, wodurch die Kompatibilität mit der neuen Datentypversion gewährleistet ist.

Evolution von XOR-Verzweigungsaktivitäten Eine Besonderheit stellen dabei die Aktivitäten dar, welche für die Evaluierung von Verzweigungsentscheidungen zuständig sind. Kann sichergestellt werden, dass bei der Wiederausführung der Ausführungshistorie auf dem neuen Prozessschema keine Änderung für die Wahl des Zweiges ergibt, so kann die Migration auf das neue Prozesschema stattfinden, obwohl eine Aktivität verändert wird, deren Zustand `COMPLETED` ist.



(a) Vor dem Verknüpfen **(b)** Nach der Generierung der Zweige

Abbildung 7.5 Vollständig automatische Generierung von Verzweigungen anhand der Struktur einer Aufzählung

7.5. Automatische Generierung von Verzweigungen

Eine Anwendung von Aufzählungen zur Modellierzeit ist das automatische Generieren von Verzweigungen. Dabei ist zu beachten, dass Verzweigungen nicht mehr als einen leeren Pfad enthalten dürfen. Es reicht also nicht aus, für jeden möglichen Wert einer Aufzählung einen leeren Zweig zu erzeugen.

Wir unterscheiden im Folgenden zwischen der vollständig automatisierten und der benutzergeführten Erstellung von Verzweigungen.

7.5.1. Vollständig automatische Generierung

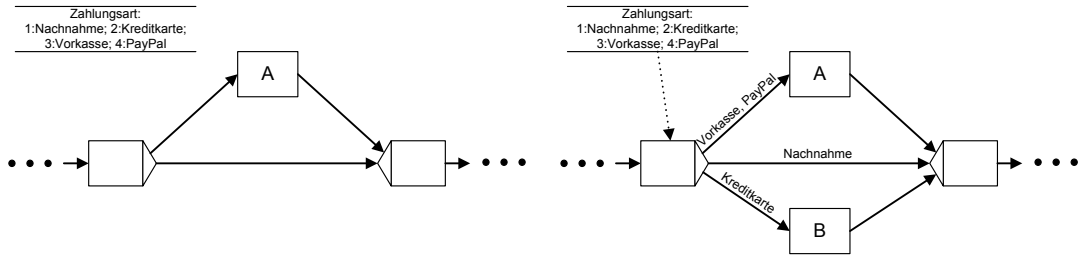
Vollständig automatisch kann eine Verzweigung anhand einer Aufzählung nur dann erzeugt werden, wenn diese bisher nur einen leeren Zweig enthält. Andernfalls ist nicht bekannt, wie die bestehenden Zweige auf die einzelnen Werte der Aufzählung abgebildet werden sollen.

Verbindet der Modellierer den Verzweigungsknoten einer leeren bedingten Verzweigung (s. Abb. 7.5a) mit einem Datenelement, dessen Typ eine Ausprägung des Typs *Aufzählung* darstellt, so kann er für jeden möglichen Wert der Aufzählung einen Zweig generieren lassen (s. Abb. 7.5b). Da eine Verzweigung höchstens einen leeren Zweig enthalten darf und nicht klar ist, für welchen Wert der Aufzählung dieser generiert werden soll, erhalten alle generierten Zweige einen leeren Knoten.

Anschließend kann der Modellierer die leeren Knoten mit Aktivitäten belegen und die Zweige mit zusätzlichen Knoten anreichern. Ist ein leerer Zweig gewünscht, kann der dort befindliche leere Knoten gelöscht werden. Da die Generierung der Zweige vollständig automatisch erfolgen soll, wird dem Benutzer keine Frage nach einem möglichen leeren Zweig gestellt.

7.5.2. Benutzergeführte Generierung

Neben der vollständig automatischen Erzeugung von Zweigen ist auch eine Variante sinnvoll, bei welcher dem Prozessmodellierer die Möglichkeit eingeräumt wird, die



(a) Bestehende Verzweigung vor dem Verknüpfen **(b)** Nach der Generierung der Zweige

Abbildung 7.6 Benutzergeführte Generierung von Verzweigungen mit Hilfe der Struktur einer Aufzählung

erzeugten Zweige zu beeinflussen. Diese benutzergeführte Generierung von Zweigen kann auch auf Verzweigungen angewendet werden, die bereits Zweige enthalten.

Der Prozessmodellierer hat dabei die Möglichkeit festzulegen, welchen Werten der Aufzählung bereits bestehende Zweige zugeordnet werden. Weiter kann er einen Wert der Aufzählung einem leeren Pfad zuordnen und mehrere Werte auf einen Zweig zusammenfassen (s. Abb. 7.6).

Wie schon bei der im vorherigen Abschnitt beschriebenen, vollständig automatischen Generierung von Verzweigungen, müssen auch bei der benutzergeführten Erzeugung alle Zweige (außer dem explizit als leer gekennzeichnetem) einen Knoten enthalten. Jeder Zweig wird daher mit einem leeren Knoten angelegt, dessen Funktion der Modellierer nach Beendigung des Assistenten festlegen kann.

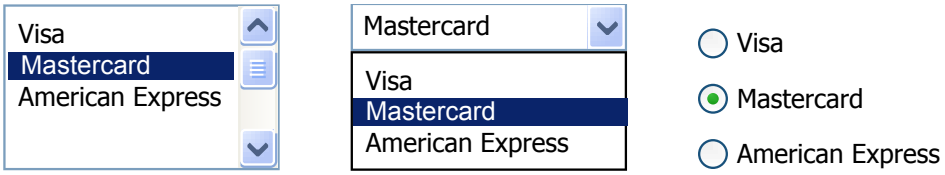
7.6. Graphische Darstellung

Neben dem automatischen Erzeugen von Verzweigungen zur Modellierzeit können Aufzählungen auch zur automatischen Generierung von Elementen der graphischen Benutzeroberfläche verwendet werden. Die Darstellungsform wird dabei nicht von der Aufzählung selbst, sondern von dem Teil der Aktivität bestimmt, welcher für das erzeugen der Oberfläche zuständig ist. Dadurch wird sichergestellt, dass die Darstellung für verschiedene Benutzeroberflächen (Weboberfläche, GUI[1] für verschiedene Betriebssysteme, etc.) unabhängig vom zugrundeliegenden Datenmodell möglich ist.

7.6.1. Mögliche Darstellungsformen

Aufzählungen können in Benutzeroberflächen auf drei verschiedene Arten dargestellt werden: als Auswahlliste (s. Abb. 7.7a), als Auswahlliste zum Aufklappen (*Combobox*) (s. Abb. 7.7b) und als *Optionsfeld* (s. Abb. 7.7c). Dabei wird jedem möglichen Wert der Aufzählung ein Eintrag in der Auswahlliste (respektive des Optionsfeldes) zugeordnet.

[1] Graphical User Interface, graphische Benutzeroberfläche



(a) Auswahlliste (b) Auswahlliste zum Aufklappen (c) Optionsfeld

Abbildung 7.7 Verschiedene Darstellungsformen für Aufzählungen in Benutzeroberflächen

Die für den Benutzer sichtbaren Einträge entstehen aus den Feldern `humanReadableDescreption` aller Einträge der Liste der möglichen Werte.

7.6.2. Interaktion mit der Benutzeroberfläche

Ist einem Datenelement mit einem von *Aufzählung* abgeleiteten Datentyp zum Zeitpunkt der Erzeugung der Oberfläche bereits ein Wert zugeordnet, so wird dieser Eintrag in der vom Formulargenerator erzeugten Darstellungsform als ausgewählt markiert. Andernfalls (das Datenelement hat den Wert `NULL`) wird kein Eintrag ausgewählt. Die Auswahl des Benutzers wird nach Beendigung der Aktivität mit graphischer Oberfläche über einen Ausgabeparameter an ein Datenelement übergeben. Dabei ist darauf zu achten, dass der Ausgabeparameter einen zum Datenelement passenden Aufzählungswert schreibt.

Ist das Anzeigeelement einem optionalen Ausgabeparameter zugeordnet, so ist dem Benutzer die Möglichkeit anzubieten, kein Element oder ein spezielles `NULL`-Element auszuwählen. Andernfalls kann ein versehentlich ausgewählter Eintrag nicht wieder abgewählt werden. Bei obligaten Ausgabeparametern ist ein derartiges `NULL`-Element nicht zulässig.

7.6.3. Weitere Aspekte

Ein alphabetisches Sortieren der Einträge ist möglich, muss allerdings an der Oberfläche durchgeführt werden, damit sie abhängig von der gewählten Darstellungsform geschieht. Eine Anordnung der Elemente anhand ihrer Ordnungszahl ist ebenfalls möglich.

Der abstrakte Datentyp *Aufzählung* bietet Aktivitäten bereits die Möglichkeit, auf alle Werte einer Aufzählung zuzugreifen. Daher kann die Generierung der Anzeigeelemente unabhängig von der Ausprägung eines Aufzählungsdatentyps geschehen. Eine Aktivität mit automatischer Oberflächengenerierung benötigt für eine Aufzählung folglich einen Parameter vom Typ *Aufzählung*. Da dieser generische Datentyp zur Laufzeit nicht verfügbar ist, muss beim Modellieren ein konkreter Aufzählungsdatentyp

gewählt werden. Intern kann die Generierung jedoch weiterhin unabhängig von diesem konkreten Typ erfolgen.

Alle Anzeigevarianten haben gemeinsam, dass der Benutzer pro Anzeigeelement nur genau einen Wert auswählen kann. Dies hat den Grund, dass ein Ausgabeparameter einer Aktivität immer nur genau einen Wert enthalten kann. Für Anzeigeelemente mit Mehrfachauswahl sind die in Kapitel 6 vorgestellten Listen zu verwenden.

7.7. Zusammenfassung

Aufzählungsdatentypen bieten sowohl Prozessmodellierern als auch Anwendern vielfältige Möglichkeiten, mit Daten umzugehen, die eine endliche Menge darstellen. Das vorgestellte Realisierungskonzept vereint die Möglichkeiten aller diskutierten Umsetzungsvarianten. Dabei kommt der Prozessmodellierer nur mit so vielen Details in Kontakt, wie für seine Aufgabe notwendig sind. Auf die Korrektheit des Datenflusses wirkt sich das Konzept der Aufzählung nicht aus. Zur Datenflussanalyse kann der bestehende Algorithmus für atomare Datentypen verwendet werden; die Typkompatibilität von Datenelementen und Aktivitätenparametern wird bei Aufzählungen äquivalent zu anderen Datentypen gewährleistet.

Die Verwaltung von Aufzählungsdatentypen erfolgt analog zur Verwaltung strukturierter Datentypen, wohingegen sich die Evolution geänderter Prozessvorlagen unkomplizierter gestaltet. Sowohl für bedingte Verzweigungen als auch für Formularelemente wurde gezeigt, wie diese anhand der Struktur von Aufzählungen automatisch generiert werden können. Für das Erzeugen bedingter Verzweigungen wurde neben dem vollständig automatischen Erstellen von Zweigen eine Variante mit Benutzerinteraktion vorgestellt. Dadurch erhält der Prozessmodellierer die Möglichkeit, mehrere Werte der Aufzählung zu einem Zweig zusammenzufassen. Bei der graphischen Darstellung von Aufzählungen wurden verschiedene Anzeigeelemente vorgestellt.

8. Verwandte Arbeiten

Derzeit am Markt verfügbare Softwaresysteme und Prozessbechreibungssprachen unterscheiden sich teilweise deutlich hinsichtlich ihrer Datenflusskonzepte. Dies betrifft einerseits die Modellierung des Datenflusses, andererseits die Überprüfung des Datenflusses auf Korrektheit. Auch sind, soweit dokumentiert, Unterschiede bei der konzeptionellen Umsetzung von Datenstrukturen erkennbar.

Um das Vergleichen der Datenflusskonzepte verschiedener PMS zu vereinfachen, wurden sog. *Datenflussmuster* entwickelt [RHEA04a]. Wir beschreiben zunächst, welche verschiedenen Muster es gibt und wie sich diese für die Untersuchung von Datenstrukturen in unterschiedlichen PMS eignen. Anschließend untersuchen wir die Datenflusskonzepte der Prozessbeschreibungssprache BPMN[1], sowie die der freien Systeme YAWL[2] und newYAWL. Darauf folgt die Untersuchung der beiden kommerziellen Systeme *Inubit BPM-Suite* und *IBM WebSphere MQWorkflow*. Die Untersuchung des quelloffenen PMS *Kepler* schließt dieses Kapitel ab.

8.1. Datenflussmuster

In [RHEA04a] werden 39 Datenflussmuster (*data patterns*) vorgestellt, die dazu dienen sollen, verschiedene Prozessbeschreibungssprachen und PMS miteinander zu vergleichen. Die Arbeit erweitert eine Serie von Vergleichsmustern (*workflow patterns*), die neben Mustern für den Datenfluss auch solche für Kontrollfluss [AHKB03, RHAM06], sowie für Ressourcen [RHEA04b] und Ausnahmebehandlungen [RAH06] enthalten. Sie sind system- und sprachunabhängig und ermöglichen so eine einfache Vergleichbarkeit verschiedener Systeme und Sprachen.

Die Datenflussmuster sind in die vier Kategorien *Datensichtbarkeit*, *Dateninteraktion*, *Datentransfermechanismen* und *datenbasiertes Routing* aufgeteilt. Es existieren weder Muster für die Repräsentation von Daten in Prozessen, noch für die Korrektheit des Datenflusses hinsichtlich der sicheren Versorgung von Arbeitsschritten mit Daten.

Durch die Ergebnisse der vorangegangenen Kapitel und die Ergebniss aus [Wol08] unterstützt ADEPT2 derzeit 11 Muster vollständig und 15 Muster teilweise. Durch die beiden Arbeiten ist die Unterstützung der drei Muster für Multiinstanz-Daten hinzugekommen. Zuvor wurden lediglich 8 Muster vollständig unterstützt. Der Grund dafür, dass überdurchschnittlich viele Pattern nur teilweise unterstützt werden, ist das aktivitätenzentrierte Modell von ADEPT2. Da Programmcode in ADEPT2 nur als Aktivität

[1] Business Process Modeling Notation
[2] Yet Another Workflow Language

ausgeführt wird, ist bspw. keine Interaktion einer Prozessinstanz oder einer Prozessvorlage mit anderen Systemen möglich. Eine detaillierte Auflistung der von ADEPT2 derzeit unterstützten Datenflussmuster findet sich in Anhang B. Aufgrund des Fehlens von Vergleichsmustern für Datenflusskorrektheit und Datenrepräsentation sind die Datenflussmuster nicht Gegenstand der folgenden Untersuchungen. Eine ausführliche Untersuchung verschiedener Systeme hinsichtlich der Unterstützung der Datenflussmuster ist in [RHEA04a] zu finden.

8.2. Untersuchte Systeme

Um einen guten Überblick zu bekommen, wird nachfolgend eine Auswahl aus wissenschaftlichen, theoretischen, freien und kommerziellen Produkten und Sprachen untersucht. Den Anfang der Untersuchungen bildet die von der OMG[3] standardisierte *BPMN*. Danach wenden wir uns dem an der Universität Eindhoven entwickelten System *YAWL*, sowie dessen Nachfolger *newYAWL* zu. Anschließend werden die beiden kommerziellen Produkte *Inubit BPM*[4]*-Suite* und *IBM WebSphere MQ* untersucht, bevor zuletzt das PMS *Kepler* besprochen wird, welches zur Modellierung wissenschaftlicher Datenverarbeitungsprozesse verwendet wird.

8.2.1. BPMN

Nachdem die Spezifikation von BPMN 2.0 [Obj08a] noch in den Kinderschuhen steckt, beschäftigen wir uns im Folgenden mit der derzeit aktuellen Version 1.2 [Obj09], die sich aber aus unserem Betrachtungswinkel nicht von BPMN 1.1 [Obj08b] und BPMN 1.0 [Obj06] unterscheidet. Die Neuerungen in Version 1.1 gegenüber Version 1.0 beziehen sich ausschließlich auf den Kontrollfluss und Version 1.2 enthält lediglich einige Fehlerberichtigungen und Formatierungsänderungen im Spezifikationsdokument.

Einführung

Die BPMN ist eine graphische Beschreibungssprache für Geschäftsprozesse, die 2006 von der OMG spezifiziert wurde [Obj06]. Sie dient dazu, technischen, sowie fachlichen Experten jeweils eine möglichst passende Sicht auf einen Geschäftsprozess zu bieten. Da es möglich ist, Prozesse hierarchisch zu strukturieren, können diese so detailliert gestaltet werden, dass sie präzise genug sind, um technisch umgesetzt zu werden, ohne dabei Mitarbeiter aus Fachabteilungen zu überfordern, die nicht mit technischen Details vertraut sind. Letztere bekommen lediglich eine einfache Sicht auf den Prozess, die ihrem fachlichen Wissen angepasst ist.

Der Spezifikation zufolge ist es nicht vorgesehen, dass in BPMN modellierte Prozesse direkt ausgeführt werden. Es gibt einerseits die Möglichkeit, BPMN in die Ausfüh-

[3] Object Management Group
[4] Business Process Management

rungssprachen BPEL[5] zu transformieren [ODHA07] oder in XPDL[6] zu beschreiben, andererseits gibt es Ansätze, BPMN sowohl im Kontroll- [Gro07] als auch im Datenfluss [Sch08] derart zu erweitern, dass die Prozesse direkt ausgeführt werden können. Diese Weiterentwicklungen sind jedoch noch nicht in den Standard der OMG eingeflossen. Die Weiterentwicklung von BPMN [Obj08a] soll die Möglichkeit bieten, Prozesse direkt auszuführen.

Ein Format, in welchem in BPMN modellierte Prozesse gespeichert oder ausgetauscht werden, ist zum gegenwärtigen Zeitpunkt nicht spezifiziert. Folglich ist es derzeit Ziel von BPMN, Prozessmodellierern ein mächtiges Visualisierungswerkzeug an die Hand zu geben.

Datenfluss

Dem Modellierer stehen in BPMN zwei Konstrukte zur Verfügung, mit denen er Daten bzw. deren Austausch kennzeichnen kann: *Message Flows* und *Data Objects*.

Message Flows werden verwendet, um einen Nachrichtenaustausch zwischen zwei Prozesselementen zu modellieren. Dabei können nur Elemente untereinander, Elemente mit anderen Pools[7] oder Pools untereinander verbunden werden. Werden Elemente mit Elementen verbunden, so müssen sich diese in unterschiedlichen Pools befinden. Die ausgetauschten Nachrichten selbst werden dabei nicht genauer spezifiziert. Es ist jedoch möglich, die Nachrichten mit *Data Objects* näher zu beschreiben.

Data Objects symbolisieren (immaterielle) digitale Daten oder materielle Dokumente in einem BPMN-Prozess. Diese Daten oder Objekte können in einem Arbeitsschritt gelesen, geschrieben oder verändert (s. Abb. 8.1), sowie zwischen Arbeitsschritten ausgetauscht werden (s. Abb. 8.2). Dazu werden die *Data Objects* mit (gerichteten oder ungerichteten) *Associations* an einen *Message-* oder *Sequence Flow* angeheftet.

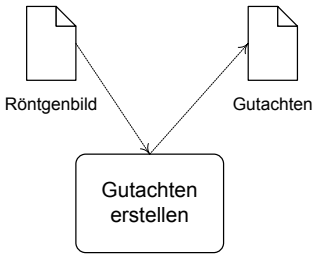


Abbildung 8.1 Beim Start einer Aktivität wird lesend auf das Röntgenbild zugegriffen, beim Beenden wird das radiologische Gutachten geschrieben

5 Business Process Execution Language
6 XML Process Definition Language
7 Ein *Pool* repräsentiert in BPMN einen Teilnehmer eines Prozesses, wie z. B. einen Kunden oder einen Lieferanten.

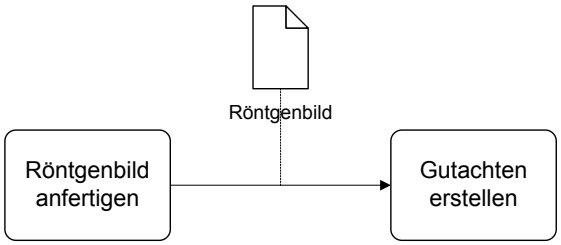


Abbildung 8.2 Ein Röntgenbild wird zwischen zwei Aktivitäten ausgetauscht

Data Objects besitzen einen Namen und können mit einem Status versehen werden, der angibt, in welchem Zustand sich das Objekt an der betreffenden Stelle im Prozess befindet. Darüber hinaus gibt es die Möglichkeit festzulegen, ob ein *Data Object* für den Start einer Aktivität zwingend notwendig (*ArtifactInput*) ist oder ob es bei der Beendigung einer Aktivität sicher erzeugt bzw. geschrieben (*ArtifactOutput*) wurde. Eine Überprüfung dieser Zusicherungen ist in BPMN nicht vorgesehen. Inwieweit hierfür systemspezifische Lösungen existieren wurde nicht untersucht.

Zusätzlich zu den oben genannten Attributen können für ein *Data Object* noch beliebig viele *Properties* angegeben werden. Jede *Property* besitzt einen Namen, einen Datentyp, einen Wert und die Information, ob die Eigenschaft korreliert verwendet wird. Die Datentypen sind von BPMN nicht näher spezifiziert. Sie werden lediglich als Zeichenkette angegeben und nicht interpretiert. Datentypen können hierarchisch strukturiert werden, jedoch liefert BPMN selbst keine näheren Informationen, wie dies zu geschehen hat.

Softwaresysteme

Die Umsetzung der Datentypen ist den Herstellern der BPMN-Systeme überlassen. Daher werden im Folgenden lediglich zwei Softwaresysteme beispielhaft angesprochen und wir beziehen uns weiterhin auf die Spezifikation, wenn wir von BPMN sprechen.

In Oryx[8] ist es möglich, jeder *Property* einen Datentyp zuzuordnen. Eine hierarchische Strukturierung ist nicht möglich. Intalio[9] bietet in seinem Intalio|Designer keine Möglichkeit, *Properties* von *Data Objects* zu bearbeiten, weswegen auch keine Datentypen bearbeitet werden können. Beide Systeme bieten keine Möglichkeit, die modellierten Prozesse auf Plausibilität oder Korrektheit zu überprüfen. Auch eine Überprüfung des Datenflusses ist nicht möglich.

8.2.2. YAWL

YAWL ist eine auf Petrinetzen basierende Prozessbeschreibungssprache. Sie wurde an der Universität Eindhoven entwickelt, um den Defiziten existierender BPM-Systeme entgegenzutreten, die bei der Entwicklung der Workflow-Patterns [AHKB03] entdeckt

8 http://www.oryx-editor.org/
9 http://www.intalio.com/

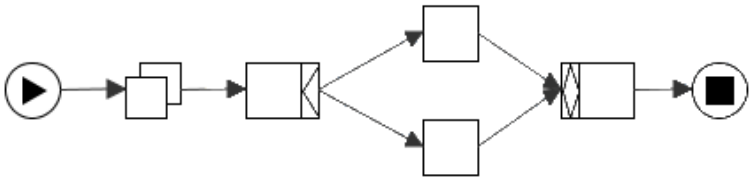


Abbildung 8.3 Einfacher YAWL-Prozess mit Multiinstanz-Aktivität und paralleler Verzweigung und ODER-Zusammenführung

wurden. Die dabei entstandenen theoretischen Konzepte [AH03] wurden anschließend in einem gleichnamigen Softwaresystem umgesetzt [AADH04]. Sowohl in der Theorie als auch in der Umsetzung unterstützt YAWL 19 der 20 identifizierten Kontrollflusspatterns.

Da ursprünglich nur Workflowpatterns für den Kontrollfluss erarbeitet wurden, wurde die Datenflussperspektive bei der Konzeption von YAWL [AADH04] erarbeitet. Die Datenflussperspektive wird in den Handbüchern des YAWL-Systems [KBP07, Ouy05] näher beschrieben. YAWL grenzt sich durch die konsequente Verwendung von XML-basierten Standards (XPath, XQuery[10], etc.) von vielen anderen am Markt verfügbaren Systemen ab, welche größtenteils auf proprietäre Sprachen und Formate zur Beschreibung von Prozessen und Daten setzen [AADH04].

Datentypen

Laut Dokumentation [Ouy05] bringt YAWL von Haus aus bereits die Datentypen *boolean*, *string*, *double*, *long*, *date* (yyyy-mm-dd), *time* (hh:mm:ss) und *duration* mit, wohingegen in der aktuellen Version (2.0 RC1) deutlich mehr Datentypen verfügbar sind. Diese sind in Tabelle 8.1 dargestellt.

Weitere Datentypen können vom Benutzer in einem XML-Schema, direkt im Editor definiert und danach im System verwendet werden. Diese benutzerdefinierten Datentypen werden zusammen mit dem Prozess gespeichert und sind nicht systemweit verfügbar.

Datenfluss

Das Datenflusskonzept in YAWL ermöglicht es dem Benutzer, Variablen auf zwei verschiedenen Ebenen zu definieren. Auf der oberen Ebene heißen diese *Netzvariablen*. Sie sind für alle Arbeitsschritte (*Tasks*) eines Prozesses sichtbar. Die Netzvariablen werden in einem XML-Dokument repräsentiert, auf welches von den einzelnen Tasks aus sowohl lesend als auch schreibend zugegriffen werden kann.

Auf der unteren Ebene sind die sog. *Taskvariablen* angesiedelt. Diese sind nur für den jeweiligen Task sichtbar, in dessen Kontext sie definiert wurden, sowie für des-

[10] XML Query Language

anyType	float	language	QName
anyURI	gDay	long	short
base64Binary	gMonth	Name	string
boolean	gMonthDay	NCName	time
byte	gYear	negativeInteger	token
date	gYearMonth	NMTOKEN	unsignedByte
dateTime	hexBinary	NMTOKENS	unsignedInt
deximal	ID	nonNegativeInteger	unsignedLong
double	IDREF	nonPositiveInteger	unsignedShort
duration	IDREFS	normalizedString	YTimerType
ENTITIES	int	notation	
ENTITY	integer	positiveInteger	

Tabelle 8.1 Derzeit in YAWL verfügbare Datentypen

sen Subprozesse, falls es sich um einen zusammengesetzten Task (*composite task*) handelt. Im letztgenannten Fall werden die *Taskvariablen* des zusammengesetzten *Tasks* als *Netzvariablen* des Subprozesses bereitgestellt. Umgekehrt sind alle *Netzvariablen* des Subprozesses als *Taskvariablen* des übergeordneten *Tasks* verfügbar, dem der Subprozess zugeordnet ist. Werden *Netzvariablen* als *lokal* deklariert, so können sie nicht mehr in übergeordneten Netzen referenziert werden. Die Weitergabe der Daten aus lokalen Variablen an untergeordnete Netze ist jedoch möglich. Bei hierarchisch angeordneten Netzen müssen Eingangsvariablen des Netzes auf der obersten Ebene, sowie lokale Neztvariablen auf allen Ebenen mit einem Startwert vorbelegt werden. Geschieht dies nicht, werden sie zur Laufzeit vom Benutzer abgefragt.

Die Zuordnung von *Netzvariablen* zu *Taskvariablen* werden über Parameter realisiert, die beim jeweiligen *Task* definiert werden (s. Abb. 8.4). Dabei dienen Eingabe-Parameter der Umsetzung von *Netzvariablen* auf *Taskvariablen*, Ausgabeparameter bedienen den umgekehrten Weg. Ein Parameter ist dabei ein Tupel der Gestalt (`XQuery`, `Task Variable`), wobei `XQuery` einen XQuery-Ausdruck darstellt, dessen Ergebnis der *Taskvariablen* `TaskVariable` zugeordnet wird. Dieser XQuery-Ausdruck wird auf das XML-Dokument angewendet, in welchem alle *Netzvariablen* enthalten sind. Somit können einer *Taskvariablen* beliebige Teile von *Netzvariablen*, deren Name, deren Inhalt oder andere Elemente ihrer Struktur zugeordnet werden. Jeder *Taskvariablen* kann immer nur ein Parameter zugeordnet werden, jedoch kann der zugehörige XQuery-Ausdruck mehrere *Netzvariablen* aggregieren und das Ergebnis der *Taskvariable* zuweisen.

Der Austausch von Daten zwischen einzelnen *Tasks* geschieht niemals direkt, sondern wird immer über *Netzvariablen* durchgeführt. Dies begründet sich durch die rein lokale Verfügbarkeit von *Taskvariablen* innerhalb eines *Tasks*.

Korrektheitskriterien Der YAWL-Editor ist in der Lage, den modellierten Datenfluss auf Korrektheit zu prüfen. Dieser Prüfung werden die im Folgenden beschriebenen Kriterien zugrunde gelegt.

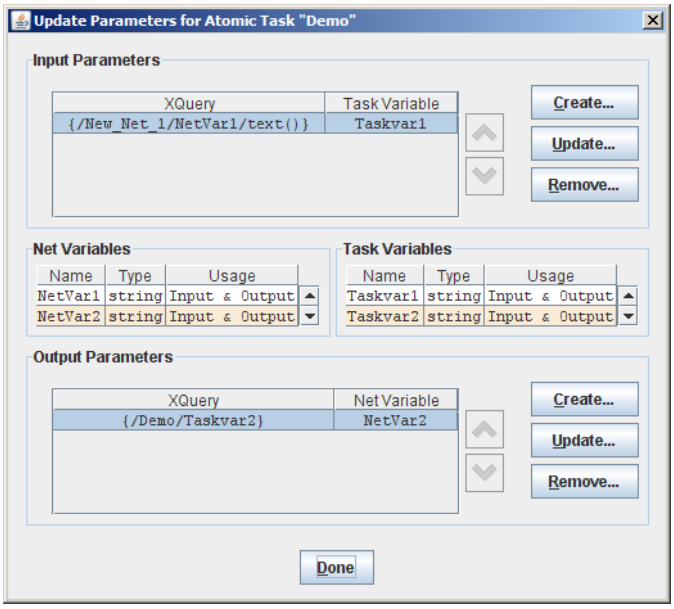


Abbildung 8.4 Dialog für die Zuordnung von Task- zu Netzvariablen über Parameter (YAWL 2.0)

Jeder Eingangsvariablen eines Tasks muss genau ein Parameter zugeordnet sein. Das Ergebnis des verwendeten XQuery-Ausdrucks wird nicht analysiert. Die Analyse des XQuery-Ausdrucks wird bei dessen Erstellung durchgeführt, wobei dieser lediglich auf syntaktische Fehler überprüft wird.

Ein YAWL-Prozess wird bei der Validierung auf seine Ausführbarkeit überprüft. Um sicherzustellen, dass ein modellierter Prozess nicht nur ausführbar, sondern auch inhaltlich korrekt ist, stehen zwei umfangreiche Analyseverfahren zur Auswahl: Die Analyse auf der Basis von *Reset Nets* [WAHE06] und die WofYAWL-Analyse [VAH07].

Verzweigungen

YAWL bietet sowohl exklusive (XOR) als auch parallele (OR) ODER-Verzweigungen. Als Verzweigungsentscheidung wird jedem Pfad ein boolescher XPath-Ausdruck zugeordnet, welcher zu TRUE oder FALSE evaluiert. Die einzelnen Pfade werden in einer virtuellen Liste angeordnet. Bei XOR-Verzweigungen wird der erste Verzweigungspfad in der Liste ausgeführt, dessen XPath-Ausdruck TRUE ist, bei OR-Verzweigungen werden alle Pfade ausgeführt, deren XPath-Ausdrücke zu TRUE evaluieren. Sollten alle Ausdrücke FALSE zurückliefern, wird bei beiden Verzweigungsarten der letzte in der Liste aufgeführte Pfad ausgeführt, unabhängig davon, wie das Ergebnis seiner Evaluierung ausgefallen ist.

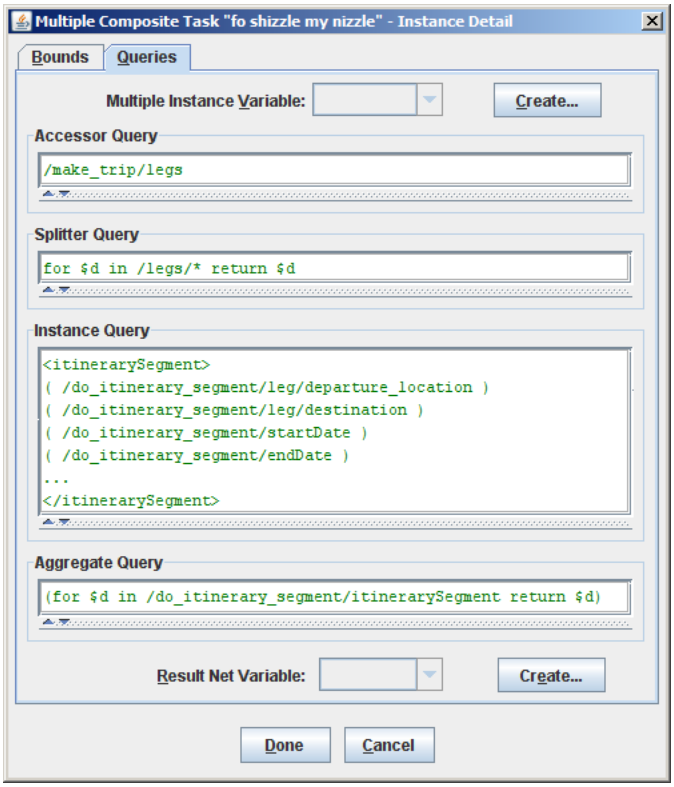


Abbildung 8.5 XQuery-Ausdrücke zur Versorgung von Multiinstanz-Knoten mit Daten (nach [Ouy05])

Multiinstanz-Knoten

Multiinstanz-Knoten entsprechen den Knoten mit variabler Parallelität aus [Wol08]. Sie dienen auch in YAWL dazu, die hinterlegte Aktivität zur Prozesslaufzeit mehrfach zu instantiieren. Dafür müssen die Eingabedaten dieses Multiinstanz-Knotens beim Starten auf die einzelnen Instanzen aufgeteilt und nach deren Beendigung wieder zusammengesetzt werden. Die Verarbeitung, die Aufteilung und das Zusammenfügen erfolgt jeweils durch eine XQuery.

Die Eingabedaten, welche an die einzelnen Instanzen weitergegeben werden, stammen aus einer einzigen, dem Knoten zugeordneten Variable. Der Inhalt dieser Variable kann vor der Verteilung an die einzelnen Instanzen durch die *Accessor Query* aufbereitet werden. Diese Aufbereitung entspricht der Zuweisung des Inhaltes einer *Netzvariable* zu einer *Taskvariable*. Die Verteilung selbst wird von der *Splitter Query* durchgeführt. Nach Beendigung aller Instanzen werden die einzelnen Ergebnisse durch die *Instance Query* so transformiert, dass sie für das anschließende Zusammensetzen vorbereitet sind. Die *Aggregate Query* setzt diese wieder zusammen. Anschließend werden sie in der dafür ausgewählten *Netzvariable* gespeichert. Abbildung 8.5 zeigt die Konfiguration des Datenflusses eines Multi-Instanz-Knotens.

8.2.3. newYAWL

Eine Weiterentwicklung von YAWL stellt *newYAWL* [RHAE07, RHA07] dar. Sie basiert auf farbigen Petrinetzen [Jen81, Jen86] und wurde entwickelt, um eine umfangreichere Unterstützung der Workflow-Patterns zu erreichen, die seit der Entwicklung von YAWL identifiziert wurden. In *newYAWL* werden 42 von 43 Kontrollfluss-Patterns aus [RHAM06], 39 von 40 Datenfluss-Patterns aus [RHEA04a] und 38 von 43 Resource-Patterns aus [RHEA04b] unterstützt [RHAE07].

Die theoretische Konzeption eines auf der *newYAWL*-Sprache basierenden Systems ist in [RAH08] beschrieben, über die praktische Umsetzung dieser Konzepte in Software ist bisher nichts bekannt. Da sich jedoch die nun in die Entwicklung mit einbezogenen Datenflusspatterns, wie in Kapitel 8.1 beschrieben, weder auf Datentypen noch auf Datenflusskorrektheit beziehen, werden von der Weiterentwicklung gegenüber YAWL keine neuen Erkenntnisse erwartet, die für diese Arbeit von Bedeutung sind.

8.2.4. Inubit BPM-Suite

Die von der Berliner Firma Inubit[11] entwickelte *Inubit BPM-Suite* bietet Prozessmodellierung sowohl aus technischer Sicht, als auch auf betriebswirtschaftlicher Ebene an. Im Nachfolgenden werden die beiden unterschiedlichen Arten von Prozessmodellen kurz erläutert. Danach gehen wir detaillierter auf das Prozessmodell ein, welches für die weitere Untersuchung relevant ist. Anschließend wird der Datenfluss in der *Inubit BPM-Suite* beschrieben, um danach dessen Korrektheit und deren Gewährleistung genauer zu beleuchten.

Unterschiedliche Diagrammarten

Neben der Möglichkeit Firmenstrukturen abzubilden, beispielsweise in Organisations-, System- oder Ressourcendiagrammen, können Prozesse zum einen fachlich, zum anderen technisch modelliert werden. Da die verschiedenen Diagramme unterschiedliche Anwendungszwecke haben, kann sich ein Prozess auch über mehrere oder sogar alle Diagrammarten erstrecken. Fachliche und technische Ebene können zudem miteinander verknüpft werden, was wir im Anschluss an die beiden Prozesssichten beschreiben. Eine Verwendung der Organisations-, System- und Ressourcendiagramme zur Prozessmodellierung ist nicht vorgesehen. Sie dienen lediglich der Dokumentation des Unternehmensaufbaus.

Prozessdiagramme Zur Modellierung der fachlichen Prozesse wird in der *Inubit BPM-Suite* die in Abschnitt 8.2.1 beschriebene BPMN verwendet. Diese fachlichen Prozesse werden *Prozessdiagramme* genannt.

Technical Workflows Für die technische Modellierung von Prozessen, sogenannter *technical workflows*, verwendet *Inubit* eine eigene, nicht näher benannte Modellierungs-

[11] http://www.inubit.de/

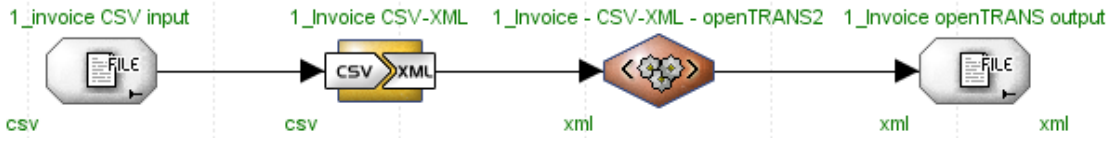


Abbildung 8.6 Technical Workflow in der Inubit BPM-Suite

sprache (s. Abb. 8.6). In *technical workflows* ist es möglich, andere Systeme durch die Nutzung verschiedener Kommunikationsprotokolle anzubinden. Zu diesen gehören, unter anderen, SOAP[12], XML-RPC[13] und TCP/IP[14].

Die einzelnen Arbeitsschritte werden mit sog. *Modulen* erstellt, die für die jeweilige Kommunikation konfiguriert werden können. Um Formulare für die Benutzereingabe zu erstellen, bietet *Inubit* ein Modul namens *Taskgenerator*. Weitere Konnektoren, Adapter und Konverter finden sich in den Modulhandbüchern [Inu08a, Inu08b].

Verknüpfen von Diagrammen Die *Inubit BPM-Suite* bietet die Möglichkeit, die unterschiedlichen Diagrammtypen miteinander zu verknüpfen. Dies vereinfacht die Navigation zwischen *Prozessdiagrammen* und *technical workflows*. Weiter ist es möglich, Diagramme derselben Art mitenander zu verknüpfen. So können hierarchisch strukturierte Diagramme erstellt werden. Um *technical workflows* hierarchisch zu strukturieren wird ein sog. *workflow connector* verwendet. Verknüpfungen haben keine Auswirkung auf den Ablauf von *technical workflows*, sie dienen lediglich dazu, den Benutzer schneller von einer Modellierungsebene zur anderen zu bringen.

Datenfluss

In *technical workflows* ist es möglich, Daten zwischen einzelnen Arbeitsschritten auszutauschen. Die ausgetauschten Daten können dabei entweder als XML- oder CSV[15]-Dokument vorliegen, wobei sich XML-Daten besser für stark strukturierte Daten eignen, wohingegen CSV besser für listenwertige Daten geeignet ist, da hier die Nutzdaten von deutlich weniger beschreibendem Text umgeben sind.

Es gibt spezielle vorgefertigte Arbeitsschritte, die XML in CSV konvertieren und umgekehrt. Auch ist es möglich Daten von von einem XML-Schema in ein anderes zu transformieren. Dabei kann die Zuordnung der einzelnen Felder und Strukturen der XML-Dokumente im *Inubit*-Editor vorgenommen. Die Transformation wird zur Laufzeit mit der XSL[16]-Subsprache XSLT[17] durchgeführt.

12 SOAP (Name eines Protokolls); vormals: Simple Object Access Protocol
13 XML Remote Procedure Call
14 Transmission Control Protocol/Internet Protocol
15 Comma-Separated Values
16 Extensible Stylesheet Language
17 XSL-Transformation

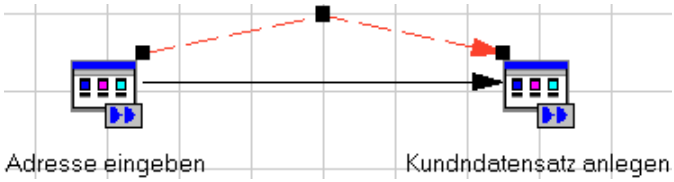


Abbildung 8.7 Ausschnitt aus einem MQWorkflow-Prozess mit Kontrollfluss (schwarz) und Datenfluss (rot)

Korrektheitsprüfung

Eine Korrektheitsprüfung der *Prozessdiagramme* ist, wohl aufgrund der Unzulänglichkeiten von BPMN, nicht implementiert. Auf der Ebene der *technical workflows* wird überprüft, ob die Eingabedaten eines Prozessschrittes mit Ausgabedaten eines vorangehenden Schrittes verknüpft sind. Ob diese Daten zur Laufzeit sicher verfügbar sind, wird nicht geprüft. Auch wird der Kontrollfluss der Prozesse nicht auf Korrektheit überprüft.

8.2.5. IBM WebSphere MQWorkflow

Das zweite untersuchte kommerzielle System ist WebSphere MQWorkflow des Herstellers IBM. Die Ergebnisse dieser Untersuchung beziehen sich auf Version 3.4. Wie die Inubit BPM-Suite bietet auch WebSphere MQWorkflow neben der Möglichkeit Prozesse zu modellieren (s. Abb. 8.7) auch Unterstützung für das Abbilden von Organisations- und IT-System-Strukturen. Im Gegensatz zur Inubit BPM-Suite ist es in WebSphere MQWorkflow jedoch auch möglich, diese Zusatzinfomrationen direkt im Prozess zu nutzen. So kann festgelegt werden, welche Mitarbeiter eine Aktivität zugeteilt bekommen und auf welchem System diese ausgeführt wird.

Datentypen

WebSphere MQWorkflow bietet sowohl atomare als auch zusammengesetzte Datentypen an. Die atomaren Datentypen sind *String*, *Binary*, *Long* und *Float*. Diese Basisdatentypen können verwendet werden, um zusammengesetzte Datentypen, *Arrays* und *Record (nested)*, zu definieren. Eine Schachtelung von *Arrays* und *Record* ist ebenfalls möglich. Dabei ist zu beachten, dass *Arrays* eine feste Länge haben und jedes Element eines Arrays als eigenständiges Datenelement gewertet wird. Dies ist wichtig, da eine Datenstruktur inklusive aller geschachtelter Strukturen maximal 512 Datenelemente enthalten darf. Einen booleschen Datentyp gibt es nicht.

Auch um einfache Datentypen verwenden zu können, muss eine Datenstruktur angelegt werden. Diese muss mit einem Namen versehen werden und bekommt dann lediglich ein Feld eines einfachen Datentyps zugewiesen.

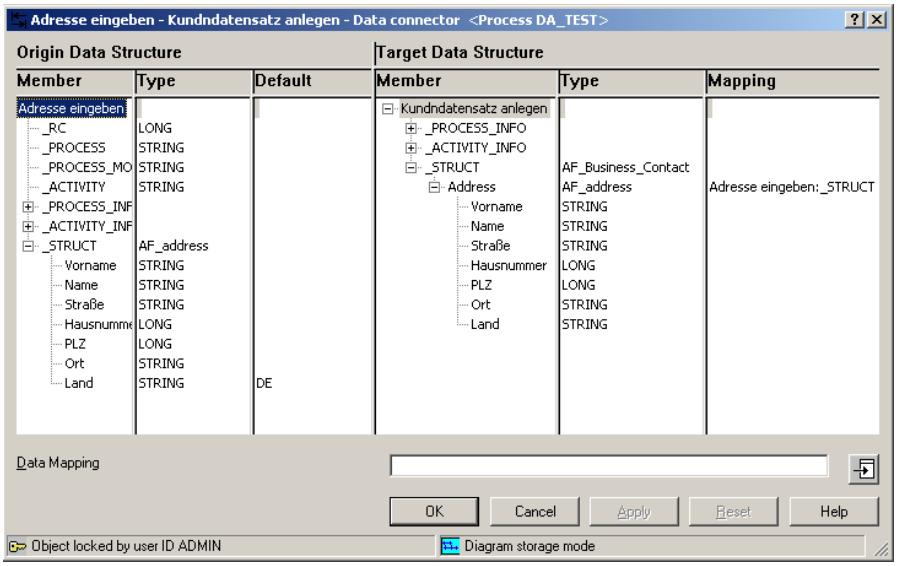


Abbildung 8.8 Parametermapping in IBM WebSphere MQWorkflow

Datenfluss

Jede Aktivität besitzt einen Ein- und einen Ausgabecontainer. Jedem Container wird eine Datenstruktur zugewiesen. Sollen Daten zwischen zwei Prozessschritten *A* und *B* ausgetauscht werden, so müssen diese mit einem *Datenconnector* verbunden werden. Für jeden Datenconnector muss festgelegt werden, wie die Felder des Ausgabecontaiers von Schritt *A* auf die Feldern des Eingabecontainers von Schritt *B* abzubilden sind.

In Abbildung 8.8 ist die Verbindung des Ausgabecontainers der Aktivität *Adresse eingeben* mit dem Eingabecontainer der Aktivität *Kundendatensatz anlegen* dargestellt. Dabei wird die gesamte Datenstruktur des Ausgabecontainers von *Adresse eingeben* auf einen Teil der Datenstruktur des Eingabecontainers von *Kundendatensatz anlegen* abgebildet (Spalte *Mapping)*.

Datenflussanalyse Prozesse in WebSphere MQWorkflow können auf ihre Funktionstüchtigkeit hin überprüft werden. Dabei wird auch überprüft, ob die Ein- und Ausgabedatenstrukturen eines Prozessschrittes mit denen der an diesem Schritt hinterlegten Anwendung übereinstimmen. Weiter wird geprüft, ob jeder Verbindung zweier Prozessschritte vermittels Datenconnectoren eine Abbildung der Datenstrukturen zugeordnet ist. Ob die Ausgabedatenstruktur der Quelle eines Datenconnectors mit der Eingabedatenstruktur dessen Ziel übereinstimmt wird ebenfalls analysiert.

8.2.6. Kepler

Im Gegensatz zu den in den vorangehenden Abschnitten beschriebenen Systemen ist *Kepler* [ABJ+04] ein Softwaresystem, welches nicht zur Abbildung und Ausführung geschäftlicher Prozesse, sondern zur Verarbeitung wissenschaftlicher Daten gedacht ist.

Boolean	General	Scalar	xml token
Complex	Int	short	arrayType(int)
Double	Long	String	arrayType(int,5)
Fixed point	Matrix	Unknown	[Double]
Float	Object	Unsigned byte	{x=double, y=double}

Tabelle 8.2 Datentypen in Kepler

Kepler basiert auf der Programmbibliothek *Ptolemy II* [IGH+02], von welchem es die Oberfläche, sowie die Funktionalität für das Modellieren und Ausführen von Prozessen übernimmt.

Datentypen

Aufgrund der Ptolemy II-Basis sind die Datentypen in Kepler sehr ähnlich zu denen in der zugrunde liegenden Programmbibliothek. Die in Kepler verfügbaren Datentypen sind in Tabelle 8.2 abgebildet. Diese Liste ist nicht vollständig, da der Benutzer die Datentypen modifizieren kann. Beispielsweise kann das Array *arrayType(int)* in *arrayType(Float)* abgeändert werden.

Zwar ist eine Implementierung eigener Datentypen denkbar, da es sich sowohl bei Ptolemy II als auch bei Kepler um Open-Source-Software handelt, jedoch ist eine einfache Einbindung innerhalb des Systems durch den Benutzer nicht möglich.

Datenfluss, Kontrollfluss

Der Datenfluss entspricht in Kepler dem Kontrollfluss, da das System zur Verarbeitung wissenschaftlicher Daten konzipiert ist. Daten werden daher direkt von einem Arbeitsschritt (*Actor)* zum nächsten weitergegeben. Die Verbindungspunkte der Arbeitsschritte heißen *Ports*.

Die Datentypen der Ports zwei verknüpfter Arbeitsschritte müssen wenigstens soweit zueinander kompatibel sein, dass die übergebenen Daten verlustfrei vom einen in den anderen Datentyp umgewandelt werden können. Dies ist dann möglich, wenn es sich um ähnliche Daten unterschiedlichen Formats handelt, wie z. B. Int und Float. Eine Beschreibung der kompatiblen Datentypen ist in [IGH+02] zu finden. Eine manuelle Konvertierung ist ebenfalls möglich, falls der Datentyp, welcher bei der automatischen Konvertierung erzeugt wird, nicht gewünscht ist. Werden inkompatible Datenverknüpfungen verwendet, wird dies beim Starten des Prozesses gemeldet. Eine Untersuchung der Kompatibilität während der Modellierung findet nicht statt. Ob alle *Ports* mit Daten versorgt werden, wird ebenfalls nicht überprüft.

Kepler bietet die Möglichkeit, Prozesse hierarchisch zu schachteln. Für den Austausch von Daten werden den Subprozessen Ein- und Ausgabeports hinzugefügt. Der Subprozess ist dann im übergeordneten Prozess als *Actor* verfügbar. Die Ein- und Ausgabeports des Subprozess-*Actors* können wie diejenigen eines normalen *Actors* verwendet werden.

Datensichtbarkeit

Aufgrund der eben beschriebenen Datenflusskonzepte sind Daten in Kepler lediglich lokal für jeden *Actor* sichtbar. In Subprozessen wird derjenige *Actor* mit Daten aus dem übergeordneten Prozess versorgt, der mit den Eingabeports des Subprozesses verknüpft wird. Auch dort sind Daten nicht prozessweit verfügbar.

Verarbeitung von Daten

Viele der mit Kepler gelieferten Komponenten sind zur Analyse, Manipulation, Transvormation oder Visualisierung von Daten gedacht. Daher ist es schwer zu sagen, welche Funktionalität direkt dem Datenflussmodell von Kepler/Ptolemy-II zugeordnet werden kann. Da auch eigene Komponenten in Kepler eingebunden werden können, sind dem Umgang mit Daten keine Grenzen gesetzt.

8.3. Zusammenfassung

Bei der Untersuchung der Übereinstimmung von ADEPT2 mit den in [RHEA04a] vorgestellten Datenflussmustern wurde deutlich, dass häufig keine eindeutige Aussage darüber getroffen werden kann, ob ein Muster durch ADEPT2 unterstützt wird. Dies begründet sich durch das flexible Metamodell von ADEPT2, welches durch die Unterstütung von Anwendungsintegration für einige Muster mehrere unterschiedliche Umsetzungsvarianten bietet. Trotz der Mehrdeutigkeit einger Datenflussmuster kann sich ADEPT2 im Vergleich mit anderen Systemen gut behaupten (s. Tabelle B.2).

Die BPMN bietet in der aktuellen Fassung ein Modell, welches ausschließlich zur Visualisierung von Geschäftsprozessen geeignet ist. YAWL und die Inubit BPM-Suite unterstützen die Modellierung des Datenflusses mit Hilfe von XML. Dies bietet ein hohes Maß an Flexibilität, erschwert jedoch die Analysierbarkeit des Datenflusses und macht dessen Modellierung unübersichtlich und aufwändig. Die einfache Analyse der Prozesskorrektheit in IBM WebSphere MQWorkflow hilft, Fehler bei der Prozessmodellierung zu vermeiden. Das Softwaresystem Kepler trennt nicht zwischen Daten- und Kontrollfluss, da es sich bei diesem System um eine Software zur Verarbeitung wissenschaftlicher Daten handelt. Eine Verwendung von Kepler zur Modellierung von Geschäftsprozessen ist nicht sinnvoll.

Die untersuchten Prozessmodelle und PMS bieten allesamt weit weniger Möglichkeiten für die Analyse der Datenflusskorrektheit als ADEPT2. Das fehlen von Analyseverfahren für die Datenflusskorrektheit erhöht das Risiko von Laufzeitfehlern. Insbesondere bei langlaufenden, medizinischen oder unternehmenskritischen Prozessen ist jedoch eine hohe Verlässlichkeit der abgebildeten und ausgeführten Prozesse von großer Bedeutung.

9. Zusammenfassung und Ausblick

Aufgrund der Zusammengehörigkeit von Prozesslogik und den ausgetauschten Daten ist die vollständige Integration von Daten in Geschäftsprozesse der nächste logische Schritt für eine konsistente Einführung von Prozess-Management-Systemen in Unternehmen. Dafür ist eine umfassende Unterstützung von Daten- und Datenflusskonstrukten unerlässlich.

Mit den in dieser Arbeit vorgestellten Datentypen und Datenflusskonzepten bietet ADEPT2 die Möglichkeit, mehr Daten als bisher abzubilden, und versetzen den Anwender in die Lage, mehr Sachverhalte bzgl. des Datenflusses einfacher zu modellieren. In diesem Kapitel geben wir eine Zusammenfassung der vorgestellten Konzepte und stellen einige Ideen für weiterführende Untersuchungen vor.

9.1. Zusammenfassung

In der bisherigen Entwicklung wurde bei ADEPT2 vor allem Wert auf Flexibilität und Korrektheit der Prozesse gelegt. Dabei wurde die Korrektheit des Datenflusses zunächst für Datenelemente ohne Typ gezeigt. Da sich die Korrektheit und deren Analyse nicht ändert, wenn Daten einen nicht weiter teilbaren Datentyp besitzen, wurden zunächst einfache Datentypen in die Entwicklung aufgenommen.

Weiter wurde bei der Datenflussanalyse vorausgesetzt, dass sich die Verfügbarkeit von Daten innerhalb eines Prozesses nur im positiven Sinne ändern kann: Daten, deren Verfügbarkeit einmal im Prozess zugesichert wurde, können nicht aus diesem entfernt werden, ohne dass es dabei zu Laufzeitfehlern kommt.

Um den Datenfluss zu vervollständigen, haben wir das Datenflussmodell um das Konzept des *konsumierenden Lesens* erweitert. Damit existiert ein Konstrukt, mit dem der Prozessmodellierer einen Datenzugriff kennzeichnen kann, der die Versorgung der Daten für den weiteren Prozessverlauf unterbricht. Mit der Erweiterung des bestehenden `WriterExists`-Algorithmus wurde die Datenflussanalyse so angepasst, dass sie derartige Datenzugriffe erkennt und die Korrektheit des Datenflusses bereits während der Modellierung sicherstellt.

Die vorgestellten *strukturierten* Datentypen bieten die Möglichkeit, zusammengehörige Daten als solche zu kennzeichnen. Dabei können die systemseitig vorhandenen Strukturinformationen dazu genutzt werden, Teilbereiche von Datentypen mit Aktivitätenparametern zu verknüpfen, die den strukturierten Datentyp selbst nicht kennen.

Auch die *benutzerdefinierten Funktionen* sorgen für eine bessere Kompatibilität zwischen Aktivitäten und Datentypen, indem sie dem System unbekannte Datentypen so aufbereiten, dass diese mit Aktivitätenparametern mit systemseitig bekannten Datentypen verknüpft werden können. Durch den Ausschluss paralleler Ausführungen von

schreibenden benutzerdefinierten Funktionen wird die Korrektheit des Datenflusses auch weiterhin gewährleistet.

Listen sorgen für mehr Flexibilität bzgl. paralleler Schreibzugriffe und variabler Parallelität. Wir haben die dafür notwendigen Schnittstellen vorgestellt und gezeigt, welche Einschränkungen notwendig sind, damit auch für Listenzugriffe die Datenflusskorrektheit bestehen bleibt. Das Verbot von leeren Listenelementen und von indizierten Zugriffen auf prozesslogischer Ebene sind hinnehmbar, da diese durch intelligente Modellierung äquivalent dargestellt werden können.

Für eine einfachere und übersichtlichere Benutzerführung beim Erstellen von bedingten Verzweigungen sorgen die in Kapitel 7 vorgestellten *Aufzählungen*. Diese bieten auch eine einfachere Darstellung von endlichen Mengen in graphischen Benutzeroberflächen. Aufgrund der deutlichen Unterschiede zu anderen Datentypen wurde für Aufzählungen ein neuer, eigener Datentyp vorgestellt, der keine Auswirkungen auf die Datenflusskorrektheit hat, da Aufzählungen zu jedem Zeitpunkt jeweils nur ein Datum enthalten können.

Die Untersuchung verschiedener Systeme und Konzepte hat gezeigt, dass ADEPT2 mit den vorgestellten Erweiterungen seine Führungsposition bzgl. Prozesskorrektheit und Flexibilität weiter ausgebaut hat. Die untersuchten Systeme bieten meist nur rudimentäre Korrektheitsanalysen und verlassen sich für eine umfangreiche Datenflussmodellierung zumeist auf umfangreiche, aber schwer analysierbare Datenmodelle wie z. B. XML. Dadurch steigt das Risiko von Laufzeitfehlern und der Aufwand bei der Prozessmodellierung deutlich.

9.2. Ausblick

Im Verlauf der Arbeit wurden weitere den Datenfluss betreffende Aspekte identifiziert, die jedoch aufgrund ihrer weitgehend orthogonalen Fragestellungen nicht detailliert untersucht wurden. Dennoch sollen zwei dieser Themen nicht unerwähnt bleiben: Die Synchronisation paralleler Schreibzugriffe stellt eine sinnvolle Erweiterung des Datenflussmodells dar. Die Weitergabe von Daten vor der Beendigung einer Aktivität kann bei Aktivitäten mit langer Laufzeit hilfreich sein, um das Fortschreiten des Prozesses nicht zu behindern. Wir stellen diese beiden Konzepte im folgenden kurz vor.

9.2.1. Paralleles Schreiben

Das Scheiben von Datenelementen aus parallelen Zweigen heraus ist auch mit den in dieser Arbeit vorgestellten Konzepten nur eingeschränkt möglich. Strukturierte Datentypen suggerieren, dass man diese aus unterschiedlichen Zweigen zur gleichen Zeit schreiben darf. Jedoch ist dies nur dann möglich, wenn es sich bei den zu schreibenden Feldern um disjunkte Mengen handelt. Das parallele Schreiben eines Feldes ist weiterhin nicht gestattet. Auch bei Listen ist es nur möglich, Daten aus parallelen Zweigen hinzuzufügen, da es sich hierbei um eine kommutative Operation handelt. Die Modifikation einer gesamten Liste ist, wie bei anderen Datentypen auch, nur gestattet, wenn

die Reihenfolge der Schreibzugriffe eindeutig ist. Da die parallele Bearbeitung von Daten mit der vermehrten Einführung von Prozess-Management-Systemen immer mehr an Bedeutung gewinnt, stellt sie ein interessantes und wichtiges Thema für weitere Untersuchungen dar.

9.2.2. Vorzeitige Datenweitergabe

Bei der *vorzeitigen* Weitergabe von Daten [Bla96] handelt es sich um Schreibzugriffe auf Datenelemente, die noch vor Beendigung einer Aktivität ausgeführt werden. Dabei ist der vorzeitig geschriebene Wert sofort für parallel ausgeführte Aktivitäten sichtbar. Das Konzept der vorzeitigen Datenweitergabe kann teilweise bereits mit leichten Modifikationen der heute in ADEPT2 verfügbaren Konstrukten umgesetzt werden: Aktivitäten können zur Laufzeit Daten in ihren Ausgabeparametern zwischenspeichern. Die dort gespeicherten Daten werden derzeit erst beim Beenden der Aktivität in das verbundene Datenelement übertragen. Dieses Verhalten muss derart modifiziert werden, dass die Daten aus den Ausgabeparametern bereits in das Datenelement übertragen werden, sobald diese von der Aktitität in ihre Parameter geschrieben wurden. Um die dabei erzeugten Daten anderen, parallel laufenden Aktivitäten zur Verfügung zu stellen, sind Synchronisationsverfahren erforderlich, die denen zu parallel geschriebenen Daten ähnlich sind. Eine deterministische Aussage darüber, welche Daten dann in einem parallelen Zweig zur Verfügung stehen, ist sowohl bei der vorzeitigen Datenweitergabe als auch bei parallelen Schreibzugriffen nicht möglich.

Anhang

A. Glossar

Aktivität Aktivitäten sind ausführbare Anwendungskomponenten, welche an einem Knoten eines ADEPT2-Prozesses hinterlegt werden. Die Ausführung eines Knotens entspricht somit der Ausführung der hinterlegten Aktivität. Bei einer Aktivität kann es sich um ein Anwendungsprogramm, aber auch um Subprozesse oder spezielle Systemaktivitäten (Verzweigungsentscheidung, Schleifenbedingung, etc.) handeln.

Aktivitätenvorlage Eine Aktivitätenvorlage dient der Konfiguration einer Anwendungskomponente, bevor diese in einem Prozess als Aktivität verwendet werden kann. Damit ist eine prozessübergreifende Konfiguration möglich. Für Konfigurationswerte, die nicht gesperrt sind, kann die Konfiguration für jeden Prozess angepasst werden. Dies geschieht, nachdem eine Aktivitätenvorlage durch Verknüpfung mit einem Knoten zur Aktivität geworden ist.

Aktivitätenparameter Aktivitäten tauschen über Ein- und Ausgabeparameter Daten mit Datenelementen eines Prozesses aus. Beim Start einer Aktivität werden die Werte aller Datenelemente, die mit den Eingabeparametern dieser Aktivität verknüpft sind, in selbige übertragen. Die Aktivität kann dann über die Schnittstelle `InputDataContext` lesend auf diese Daten zugreifen.

Sollen Daten an Datenelemente übergeben werden, so werden sie zunächst von der Aktivität in ihre Ausgabeparameter geschrieben. Dies geschieht über die Schnittstelle `DataContext`, die bidirektionalen Zugriff auf alle Ausgabeparameter der Aktivität erlaubt. Nach Beendigung der Aktivität werden die Daten vom System von den Ausgabeparametern in die mit ihnen verknüpften Datenelemente übertragen.

Wir unterscheiden zwischen *optionalen* und *obligaten* Parametern. Obligate Eingabeparameter müssen zur Laufzeit zwingend mit Daten versorgt werden, obligate Ausgabeparamter gewährleisten eine sichere Versorgung des verknüpften Datenelementes.

Datenelement Datenelemente werden in ADEPT2-Prozessen für den Austausch von Daten zwischen Aktivitäten verwendet. Aktivitäten lesen Daten aus Datenelementen und schreiben sie in diese zurück. Dazu werden die Datenelemente mit Aktivitätenparametern verknüpft. Ein direkter Austausch der Daten von Aktivitäten untereinander findet nicht statt.

Datenfluss Als Datenfluss wird der Austausch von Daten zwischen Aktivitäten und Datenelementen bezeichnet. Die für die Verknüpfung verwendeten Konstrukte heißten

Datenflusskanten. Wir unterscheiden zwischen Lese- und Schreibkanten, wobei Lesekanten Datenelemente mit Eingabeparametern, Schreibkanten Ausgabeparameter mit Datenelementen verknüpfen. Die Menge aller Datenflusskanten eines Prozessmodells wird als *Datenflussschema* oder *Datenflussmodell* bezeichnet.

Datentyp Im Kontext von ADEPT2 beschreibt ein *Datentyp* die Beschaffenheit und die Struktur des Inhaltes von Datenelementen und Aktivitätenparametern. Allgemein werden mit Datentypen Beschaffenheit und Struktur von Daten beschrieben. Ein Datentyp besitzt in ADEPT2 eine eindeutige Bezeichnung, über welche die Kompatibilität von Datenelementen und Aktivitätenparametern geregelt wird. Diese ist nur gewährleistet, wenn die Datentypen beider übereinstimmen.

Wir unterscheiden zwischen *einfachen* Datentypen, die keine systemseitig erkennbare Struktur aufweisen und solchen, die in weitere Teile zerlegt werden können. Die Werte der einfachen Datentypen werden als *atomar* bezeichnet. Die Struktur *strukturierter* Datentypen ist dem System bekannt und kann von diesem interpretiert werden. Eine Ausnahme von dieser Klassifizierung stellen Listen dar, da sie zwar eine systemseitig bekannte Struktur besitzen, diese jedoch nur eingeschränkt beim Modellieren von Prozessen zur Verfügung steht. Die in dieser Arbeit vorgestellten *Aufzählungen*, sowie *benutzerdefinierte* Datentypen zählen zu den atomaren Datentypen, da sie zu jedem Zeitpunkt nur genau einen Wert abbilden können. Die möglicherweise innerhalb eines benutzerdefinierten Datentyps vorhandene Struktur ist dabei unerheblich, da sie dem System selbst nicht bekannt ist.

Kontrollfluss Der Ablauf eines Prozesses, bestehend aus der sequentiellen, parallelen, bedingten oder wiederholten Ausführung von Aktivitäten, wird als *Kontrollfluss* bezeichnet. Dieser wird in einem Prozessmodell durch den *Kontrollflussgraphen* dargestellt. Der Kontrollfluss stellt die logische Abfolge der Prozessschritte dar.

Laufzeit Mit dem Starten eines Prozesses wird aufgrund seiner Prozessvorlage eine neue Prozessinstanz erzeugt. Der Zeitraum zwischen dem Start und der Beendigung der Prozessinstanz wird als *Laufzeit* oder *Prozesslaufzeit* bezeichnet. Zur Laufzeit werden alle Aktivitäten eines Prozesses ausgeführt und die Daten zwischen ihnen ausgetauscht. Weiter werden Prozessgraphen zur Laufzeit mit Markierungen versehen, die den aktuellen Ausführungszustand der Prozessinstanz abbilden.

Modellierzeit Bevor eine Prozessvorlage instanziiert werden kann, muss diese modelliert werden. Die Zeitspanne des Modellierens wird *Modellierzeit* oder *Entwurfszeit* genannt. Zur Modellierzeit wird in ADEPT2 sowohl die Korrektheit des Kontroll-, als auch des Datenflusses gewährleistet. Prozesse, die nicht korrekt sind, dürfen nicht ausgeführt werden, da schon vor der Ausführung feststeht, dass sie nicht fehlerfrei beendet werden können.

Prozess Im Kontext von PMS wird ein Geschäftsprozess oft nur abgekürzt als *Prozess* bezeichnet. Dabei kann es sich um einen realen Geschäftsprozess aus einem Unternehmen, aber auch um einen bereits modellierten Geschäftsprozess innerhalb eines PMS handeln. Um Verwechslungen auszuschließen, wird im PMS zwischen *Prozessvorlage* und *Prozessinstanz* unterschieden.

Prozessinstanz Kommt ein Geschäftsprozess innerhalb eines PMS zur Ausführung, so wird auf der Grundlage seiner *Prozessvorlage* eine neue *Prozessinstanz* erzeugt. In der Prozessinstanz werden Markierungen für den Fortschritt des Prozesses eingefügt, so dass für Benutzer und System erkennbar ist, welche Aktivitäten bereits ausgeführt wurden, welche sich gerade in Bearbeitung befinden und welche Aktivitäten noch auszuführen sind.

Prozess-Management-System Um Geschäftsprozesse informationstechnisch darzustellen und auszuführen werden *Prozess-Management-Systeme* (kurz PMS) verwendet. Dabei ist nicht genau festgelegt, ob ein *PMS* in der Lage ist, Prozesse lediglich zu visualisieren, sie zu simulieren oder sie in vollem Umfang auszuführen. Aus heutiger Sicht ist die Umsetzung aller genannten Funktionen in einem System die effektivste Variante, Unternehmen bei der durchführung von Geschäftsprozesse durch die EDV zu unterstützen.

Prozessvorlage, -schema Die *Prozessvorlage* (auch das *Prozessschema*) ist die Umsetzung eines realen Geschäftsprozesses vermittels eines PMS. Die Prozessvorlage beschreibt dabei, wie der Prozess vonstatten geht. Sie wird selbst nicht direkt ausgeführt.

B. Unterstützung von Datenflussmustern in ADEPT2

Im Rahmen dieser Arbeit wurde Unterstützung der Datenflussmuster aus [RHEA04a] in ADEPT2 untersucht. Die Ergebnisse dieser Untersuchung sind im folgenden ausgeführt. Eine Übersicht über die von ADEPT2 unterstützten Muster bietet Tabelle B.1. Ein Vergleich mit anderen Systemen ist in Tabelle B.2 dargestellt.

B.1. Untersuchung der einzelnen Pattern

B.1.1. Datensichtbarkeit

Die Muster in diesem Abschnitt beziehen sich auf Zugriffe auf Daten, die unter der Kontrolle des PMS stattfinden. Bei ADEPT2 wird zwischen der Sichtbarkeit der Daten und dem Zugriff auf Daten unterschieden. Innerhalb eines Prozesses sind alle Datenelemente für jede Aktivität sichtbar, sie dürfen aber unter bestimmten Umständen nicht gelesen oder geschrieben werden (vgl. [Rei00]).

Pattern 1 (Task Data)

Daten, die nur innerhalb einer Aktivität verwendet werden, können in ADEPT2 selbst nicht definiert werden. Es steht dem Entwickler der Aktivität frei, derartige Daten, bspw. für Zwischenergebnisse zu verwenden. Da diese dann aber nicht unter der Kontrolle des PMS stehen, ist dies keine adäquate Lösung für dieses Pattern.

Eine Alternative stellen *Blockaktivitäten* dar (siehe Pattern 2). Es ist denkbar, einen Subprozess zu erstellen, der lediglich eine Aktivität enthält. Die in diesem Subprozess definierten Datenelemente sind nur für diese eine Aktivität sichtbar und werden vom PMS kontrolliert. Um diese Datenelemente für Zwischenergebnisse zu verwenden, müsste die Programmaktivität allerdings weiter unterteilt werden, da der Zugriff auf Daten derzeit nur beim Starten und Beenden einer Aktivität möglich ist. Ob eine derartige Realisierung sinnvoll ist, muss sich in der Praxis zeigen. Fakt ist jedoch, dass Pattern 1 dadurch implementiert werden kann.

Pattern 2 (Block Data)

ADEPT2 bietet mit *Subprozessen* die Möglichkeit, Prozesse hierarchisch zu strukturieren. Datenelemente, die in Subprozessen genutzt werden, sind nicht für Aktivitäten in übergeordneten Prozessgraphen sichtbar. Die Übergabe von Daten an einen Subprozess

ist ebenso möglich, wie der Empfang von Rückgabedaten bei Beendigung der Aktivität, welche den Subprozess im übergeordneten Prozess repräsentiert.

Pattern 3 (Scope Data)

In ADEPT2 kann ein Prozess in sog. *Kontrollblöcke* unterteilt werden. Jeder Kontrollblock kann einen Subprozess umgewandelt werden. So können für Kontrollblöcke eigene Datenelemente innerhalb des Subprozesses bereitgestellt werden. Es ist jedoch nicht möglich ist, diese Kontrollblöcke beliebig zu definieren. So ist es bspw. nicht möglich, dass der Startknoten einer Schleife in einem Kontrollblock enthalten ist, ohne dass auch ihr Endknoten enthalten ist. Daher wird dieses Pattern als nicht unterstützt angesehen.

Pattern 4 (Multiple Instance Data)

Multiinstanz-Aktivitäten (MIA) werden in [Wol08] als *Knoten mit variabler Parallelität* für ADEPT2 vorgestellt. Mit den in Kapitel 6 vorgestellten Konzepten für Listen ist ADEPT2 in der Lage, Daten für Multiinstanzaktivitäten bereitzustellen.

Pattern 5 (Case Data)

Instanzspezifische Datenelemente können in ADEPT2 durch dynamische Workflow-Änderungen ($\text{ADEPT}_{\text{flex}}$) zur Laufzeit einer Prozessinstanz hinzugefügt werden. Eine Deklaration zur Modellierzeit ist nicht möglich. Demgegenüber sind alle Datenelemente einer Prozessvorlage in den darauf basierenden Instanzen verfügbar. Die Werte der Datenelemente zur Laufzeit ist instanzspezifisch.

Pattern 6 (Workflow Data)

ADEPT2 bietet die Möglichkeit, Konfigurationsparameter für Prozessvorlagen anzugeben. Der Zugriff auf diese Konfigurationsparameter ist nur lesend möglich und erfolgt analog zum Zugriff auf Datenelemente: Damit eine Aktivität lesend auf einen Parameter der Prozesskonfiguration zugreifen kann, wird einer ihrer Eingabeparameter mit dem Prozesskonfigurationsparameter verbunden.

Parameter der Prozesskonfiguration können folglich als Datenelemente angesehen werden, deren Inhalt prozessübergreifend gelesen werden kann. Schreibender Zugriff, und eine damit verbundene Synchronisation der Daten zwischen unterschiedlichen Instanzen zur Laufzeit findet nicht statt.

Pattern 7 (Environment Data)

In der Beschreibung dieses Patterns ist explizit die Möglichkeit gegeben, seine Erfüllung durch das Ausführen von Anwendungen oder das Aufrufen von Diensten zu gewährleisten. Die in ADEPT2 verwendeten Aktivitäten bieten die Möglichkeit der Ausführung integrierter Anwendungen. Mit einer entsprechend implementierten Aktivität ist der Zugriff auf externe, nicht direkt im PMS verfügbare Daten möglich. Der Zugriff

ist auch über eine UDF möglich, sofern sich die zurückgegebenen Daten in einem geeigneten Datentyp darstellen lassen.

B.1.2. Dateninteraktion

Pattern 8 (Dateninteraktion – Aktivität an Aktivität)

In ADEPT2 tauschen Aktivitäten Daten untereinander durch die Nutzung globaler Datenelemente aus („No data passing“). Aufgrund der geforderten Strukturierungsregeln für den Datenfluss können keine Inkonsistenzen durch konkurrierende Zugriffe entstehen, da Schreibzugriffe aus parallelen Zweigen nicht gestattet sind.

Pattern 9 (Dateninteraktion – Blockaktivität an Subprozess-Dekomposition))

Die Datenübergabe erfolgt in ADEPT2 durch die Verknüpfung von Parametern eines Subprozesses mit Datenelementen des übergeordneten Prozesses. Eingabedaten des Subprozesses werden über die Ausgabeparameter seines Startknotens bereitgestellt. Sie müssen im Subprozess explizit mit Datenelementen verknüpft werden, um weiter genutzt werden zu können.

Pattern 10 (Dateninteraktion – Subprozess-Dekomposition an Blockaktivität)

Auch die Übergabe von Daten von einem Subprozesses an den übergeordneten Prozess ist mit Parametern realisiert: Ausgabedaten des Subprozesses werden über die Eingabeparameter seines Endknotens an den übergeordneten Prozess weitergereicht. Die Ausgabeparameter der Aktivität, welche den Subprozess repräsentiert, müssen im übergeordneten Prozess mit Datenelementen verknüpft werden, um dort genutzt werden zu können.

Pattern 11 (Dateninteraktion – an Multiinstanz-Aktivität)

Die in [Wol08] beschriebene Unterstützung von Multiinstanzaktivitäten klärt die Datenübergabe individueller Werte aus Listen an eine Multiinstanzaktivität. Die dafür benötigten Listen wurden in Kapitel 6 vorgestellt. Die Übergabe von Daten, die alle parallelen Instanzen gemeinsam nutzen, ist durch eine entsprechende Konfiguration der Multiinstanzaktivität möglich.

Pattern 12 (Dateninteraktion – von Multiinstanz-Aktivität)

Die Rückgabedaten der einzelnen Instanzen werden von der Multiinstanzaktivität aggregiert und in einer gemeinsamen Liste ausgegeben.

Pattern 13 (Dateninteraktion – Instanz an Instanz)

ADEPT2 unterstützt keine direkte Datenübergabe zwischen Prozessinstanzen untereinander. Durch geeignete Implementierung von Aktivitäten ist die Möglichkeit gege-

ben, Daten über Instanzgrenzen hinweg auszutauschen, etwa durch Zwischenschalten einer Datenbank. Damit wird die Kontrolle konkurrierender Schreibzugriffe in externe Anwendungen verlagert. Diese Anwendungen müssen dafür Sorge tragen, dass die Daten zu jeder Zeit konsistent bleiben, auch wenn aus mehreren Instanzen heraus der gleiche Datensatz geschrieben wird.

Pattern 14 (Dateninteraktion – Aktivität an Umgebung – Push-orientiert)

Durch das Modell der Aktivitäten ist es in ADEPT2 möglich, Daten an externe Anwendungen zu übergeben. Je nach Art der Implementierung (Java, Binärcode, SQL) ist ein entsprechender Laufzeitumgebung notwendig, welche die Aufrufe von ADEPT2 in die entsprechende Sprache übersetzt bzw. an das aufgerufene Programm übergibt.

Pattern 15 (Dateninteraktion – Umgebung an Aktivität – Pull-orientiert)

Ebenso wie die Übergabe von Daten an externe Anwendungen ist es möglich, Daten von externen Anwendungen an das System zu übergeben. Hier muss wieder eine geeignete Aktivität entwickelt werden, welche die Kommunikation mit der gewünschten externen Anwendung, etwa einem Webservice, herstellt.

Pattern 16 (Dateninteraktion – Umgebung an Aktivität – Push-orientiert)

Das Pattern kann mit Aktivitäten realisiert werden, welche auf den Eingang externer Daten warten. Eine explizite Unterstützung durch ADEPT2 ist nicht gegeben.

Werden die eingehenden Daten nicht zwingend benötigt, kann sich die wartende Aktivität in einem Zweig einer parallelen Verzweigung mit loser Synchronisation befinden [Wol08]. Diese Modellierung verhindert die Blockade eines Prozesses, der evtl. bis zum Eintreffen der Daten keine weiteren Aktivitäten mehr aktivieren kann oder, falls die Daten nie eintreffen, in einer Endlosschleife auf seine externe Terminierung wartet.

Pattern 17 (Dateninteraktion – Aktivität an Umgebung – Pull-orientiert)

Dieses Pattern kann durch zwei der drei in [RHEA04a] vorgeschlagenen Implementierungen in ADEPT2 realisiert werden. Es ist einerseits möglich, Aktivitäten zu entwickeln, die bestimmte Daten in einem externen Repository zum Abruf bereitstellen. Andererseits sind Aktivitäten denkbar, die während ihrer Laufzeit Dienste (z. B. einen Webservice) anbieten, über deren Schnittstellen die gewünschten Daten abgerufen werden können.

Pattern 18 (Dateninteraktion – Instanz an Umgebung – Push-orientiert)

Die Interaktion von einzelnen Prozessinstanzen mit dem PMS ist in ADEPT2 nicht vorgesehen. Da für dieses Pattern eine explizite Unterstützung durch die Workflowengine gefordert ist, gilt es als nicht unterstützt.

Es ist allerdings denkbar, Prozesse geeignet zu modellieren, um den selben Effekt zu erzielen, den eine direkte Unterstützung durch die Engine hätte. In diesem Fall müssten spezielle Report-Aktivitäten die zur Prozesslaufzeit anfallenden Daten an die Umgebung übergeben bzw. die Daten zur Abholung bereitstellen.

Pattern 19 (Dateninteraktion – Umgebung an Instanz – Pull-orientiert)

Siehe Pattern 18.

Pattern 20 (Dateninteraktion – Umgebung an Instanz – Push-orientiert)

Siehe Pattern 18.

Pattern 21 (Dateninteraktion – Instanz an Umgebung – Pull-orientiert)

Siehe Pattern 18.

Pattern 22 (Dateninteraktion – Prozess an Umgebung – Push-orientiert)

Da in ADEPT2 keine Daten auf Prozessebene (Pattern 6) definiert werden können, wird dieses Pattern nicht unterstützt.

Pattern 23 (Dateninteraktion – Umgebung an Prozess – Pull-orientiert)

Siehe Pattern 22.

Pattern 24 (Dateninteraktion – Umgebung an Prozess – Push-orientiert)

Siehe Pattern 22.

Pattern 25 (Dateninteraktion – Prozess an Umgebung – Pull-orientiert)

Siehe Pattern 22.

B.1.3. Datentransfermechanismen

Pattern 26 (Datenübergabe – by Value – eingehend)

Daten werden in ADEPT2 grundsätzlich *by Value* übergeben.

Pattern 27 (Datenübergabe – by Value – ausgehend)

Daten werden in ADEPT2 grundsätzlich *by Value* übergeben.

Pattern 28 (Datenübergabe – Copy in/Copy out)

Der Aktivitätenentwickler hat die Möglichkeit, eine Aktivität derart zu implementieren, dass diese die an sie übergebenen Daten in den eigenen Adressraum kopiert. Es ist jedoch nicht möglich, Daten in einen bestimmten Adressbereich innerhalb des PMS zu kopieren, der lediglich einer Aktivität zur Verfügung steht.

Pattern 29 (Datenübergabe – by reference – unlocked)

ADEPT2 bietet keine Möglichkeit, Daten *by reference* zu übergeben. Es spricht jedoch nichts dagegen, Daten als Referenz zu interpretieren, die *by value* übergeben wurden.

Pattern 30 (Datenübergabe – by reference – locked)

Siehe Pattern 29. Es sind keine Sperrmechanismen für Datenelemente vorgesehen.

Pattern 31 (Datentransformation – Eingabe)

Daten, die an Aktivitäten übergeben werden, sind per Definition typkompatibel. Es ist denkbar, Daten in vorgeschalteten Aktivitäten oder in sog. Hilfsdiensten zu transformieren. Es stehen allerdings keine systeminternen Funktionen zur Transformation von Datentypen mit systemseitig bekannter Struktur zur Verfügung. Für die Transformation benutzerdefinierter Datentypen sind UDFs vorgesehen (vgl. Kapitel 5).

Pattern 32 (Datentransformation – Ausgabe)

Die Transformation von geschriebenen Daten einer Aktivität kann mit nachgeschalteten Hilfsdiensten erfolgen. Weiter können einer Aktivität nachgeschaltete Aktivitäten zur Transformation von Daten verwendet werden. Zum Schreiben eines Wertes in ein Datenelement mit benutzerdefiniertem Datentyp stehen schreibende UDFs (vgl. Kapitel 5) zur Verfügung, sofern die Aktivität den Datentyp selbst nicht kennt.

B.1.4. Datenbasiertes Routing

Eine direkte Unterstützung von Vor-/Nachbedingungen durch die Ausführungsumgebung ist nicht vorgesehen.

Pattern 33 (Vorbedingung – Existenz von Daten)

Die Versorgung von Eingabeparametern ist in ADEPT2 per Konstruktion sichergestellt. Das Warten auf die Versorgung von Parametern ist nicht möglich. Sollte ein Parameter beim Start einer Aktivität noch nicht versorgt sein, so wird sich dies auch im weiteren Verlauf der Prozessinstanz nicht mehr ändern, da nur die von Vorgängern der wartenden Aktivität geschriebenen Daten berücksichtigt werden. Ist eine Aktivität aktiviert, so sind alle ihre Vorgänger bereits beendet, weswegen es während der Wartezeit der Aktivität zu keinem erneuten Schreibzugriff auf das Datenelement kommen kann, auf

welches die Aktivität wartet. Soll die Existenz eines Wertes als Vorbedingung für eine Aktivität gelten, so muss dies explizit mit dem Kontrollfluskonstrukt der bedingten Verzweigung modelliert werden.

Pattern 34 (Vorbedingung – Wert eines Datenelementes)

Da ADEPT2 Schleifen und bedingte Verzweigungen unterstützt ist es möglich, Aktivitäten zu überspringen, falls bestimmte Datenelemente nicht die richtigen Werte aufweisen. Ebenfalls denkbar ist das Warten auf den richtigen Wert, sofern die wartende Aktivität in einer Schleife eingebettet ist, innerhalb deren Schleifenkörpers das zu prüfende Datenelement geschrieben wird.

Es ist möglich eine bedingte Verzweigung zu modellieren, bei der ein Zweig direkt zum Ende des Prozesses führt. Ihr Vereinigungsknoten ist dann der direkte Vorgänger des Endknotens des Prozesses. Somit wäre der Abbruch einer Instanz beim Fehlen eines gültigen Wertes realisierbar. Die Instanz terminiert dann aber korrekt und es ist nur durch evtl. geschriebene Ausgabedaten des Endknotens ersichtlich, dass es sich um eine vorzeitige Terminierung gehandelt hat. Mehr Möglichkeiten bezüglich der vorzeitigen Terminierung von Instanzen bieten die *flexiblen Endknoten* aus [Wol08].

Pattern 35 (Nachbedingung – Existenz von Daten)

Dieses Pattern lässt sich mit Hilfe einer Schleife und einer entsprechenden Schleifenbedingung konstruieren. Es ist denkbar, dem Benutzer an der Oberfläche ein spezielles Konstrukt zu bieten, welches dann in eine einfache Schleife des ADEPT2-Metamodells umgesetzt wird.

Pattern 36 (Nachbedingung – Wert eines Datenelementes)

Dieses Pattern kann implementiert werden wie Pattern 35, indem das Prädikat derart abgeändert wird, dass nicht mehr auf die Existenz eines Datums, sondern auf dessen Wert geprüft wird.

Pattern 37 (Ereignisbasierter Aktivitäten-Trigger)

Die in ADEPT2 verwendeten Arbeitslisten können nicht nur Personen zugeordnet werden, sondern auch automatischen Klienten oder anderen Diensten. Somit ist das Starten von Prozessinstanzen über einen API-Aufruf möglich. Ebenfalls möglich ist das Aufwecken eines pausierten Prozesses.

Isolierte Knoten sind in ADEPT2-Prozessen nicht zulässig.

Pattern 38 (Datenbasierter Aktivitäten-Trigger)

Die Überprüfung von Datenelementen ist nur aus einer Prozessinstanz heraus möglich. Somit kann sich eine pausierte Instanz nicht selbst aufwecken. Da sich Datenelemente einer Instanz auch nicht ändern können, wenn die Instanz suspendiert ist, ist

diese Art der Umsetzung in ADEPT2 nicht sinnvoll. Es ist denkbar, externe Dienste zu implementieren, die externe Datenquellen überprüfen und bei Erfüllung gegebener Bedingungen eine Instanz starten oder wieder aufsetzen.

Pattern 39 (Datenbasiertes Routing)

Dieses Pattern wird durch die Kombination von *bedingten Verzweigungen* [Rei00] (*exclusive choice*) und *ODER-Verzweigungen* [Wol08] unterstützt.

B.2. Zusammenfassung

Die Ergebnisse der dieser Untersuchung sind in Tabelle B.1 zusammengefasst. Der Auswertung wurden die in [RHEA04a], Anhang G beschriebenen Kriterien zugrunde gelegt. Ähnliche Auswertungen sind für andere Systeme in der selben Arbeit in Anhang A-F zu finden. Eine Gegenüberstellung der Ergebnisse mit diesen anderen Systemen ist in Tabelle B.2 zu finden.

Nr	Pattern	ADEPT2
1	Task Data	±
2	Block Data	+
3	Scope Data	–
4	Multiple Instance Data	+
5	Case Data	+
6	Workflow Data	±
7	Environment Data	+
8	Dateninteraktion – Aktivität an Aktivität	+
9	Dateninteraktion – Blockaktivität an Subprozess-Dekomposition	+
10	Dateninteraktion – Subprozess-Dekomposition an Blockaktivität	+
11	Dateninteraktion – an Multiinstanz-Aktivität	+
12	Dateninteraktion – von Multiinstanz-Aktivität	+
13	Dateninteraktion – Instanz an Instanz	±
14	Dateninteraktion – Aktivität an Umgebung – Push-orientiert	±
15	Dateninteraktion – Umgebung an Aktivität – Pull-orientiert	±
16	Dateninteraktion – Umgebung an Aktivität – Push-orientiert	±
17	Dateninteraktion – Aktivität an Umgebung – Pull-orientiert	±
18	Dateninteraktion – Instanz an Umgebung – Push-orientiert	–
19	Dateninteraktion – Umgebung an Instanz – Pull-orientiert	–
20	Dateninteraktion – Umgebung an Instanz – Push-orientiert	–
21	Dateninteraktion – Instanz an Umgebung – Pull-orientiert	–
22	Dateninteraktion – Prozess an Umgebung – Push-orientiert	–
23	Dateninteraktion – Umgebung an Prozess – Pull-orientiert	–
24	Dateninteraktion – Umgebung an Prozess – Push-orientiert	–
25	Dateninteraktion – Prozess an Umgebung – Pull-orientiert	–
26	Datenübergabe – by Value – eingehend	+
27	Datenübergabe – by Value – ausgehend	+
28	Datenübergabe – Copy in/Copy out	±
29	Datenübergabe – by reference – unlocked	–
30	Datenübergabe – by reference – locked	–
31	Datentransformation – Eingabe	±
32	Datentransformation – Ausgabe	±
33	Vorbedingung – Existenz von Daten	±
34	Vorbedingung – Wert eines Datenelementes	±
35	Nachbedingung – Existenz von Daten	±
36	Nachbedingung – Wert eines Datenelementes	±
37	Ereignisbasierter Aktivitäten-Trigger	±
38	Datenbasierter Aktivitäten-Trigger	–
39	Datenbasiertes Routing	+

Tabelle B.1 Nicht (-) teilweise (±) und voll unterstützte (+) DF-Patterns in ADEPT2

System (Version)	+	±	−
Staffware (9)	13	12	14
MQSeries Workflow (3.3.2)	13	11	15
FLOWer (2.05)	20	12	7
COSA (4.2)	17	5	17
XPDL (1.0)	11	5	23
BPEL4WS (1.1)	12	7	20
ADEPT (2)	12	15	12
Durchschnitt	14	9,6	15,4

Tabelle B.2 Vergleich der Unterstützung von Datenflussmustern in verschiedenen PMS (Daten aller Systeme, mit Ausnahme von ADEPT2, aus [RHEA04a])

C. Schnittstellenbeschreibungen

C.1. Strukturierte Datentypen

C.2. Zugriffsmethoden für Listen

C.2.1. Sicht des Prozessmodellierers

Methode	Rückgabetyp	Beschreibung
`add(TYPE element)`	-	Fügt das Element `element` am Ende der Liste an. Die Listenlänge erhöht sich um 1.

C.2.2. Sicht des Aktivitätenimplementierers

Methode	Rückgabetyp	Beschreibung
`getLength()`	int	Gibt die Länge der Liste zurück. Enthält die Liste keine Elemente, ist die Länge 0.
`get(int index)`	TYPE	Gibt das Listenelement an Position `index` zurück.
`add(TYPE element)`	-	Fügt das *Element* am Ende der Liste an. Die Listenlänge erhöht sich um 1.
`set(int index, TYPE element)`	-	Setzt das Element an Position `index` auf den Wert `element`. Dabei muss gelten: $0 \leq$ `index` $\leq$ Listenlänge; `element` $\neq$ `NULL`
`delete(int index)`	-	Löscht das Listenelement an Porition `index`. Es entsteht keine Lücke, die Listenlänge verringert sich um 1.
`clear()`	-	Löscht alle Listenelemente. Die Länge der Liste ist anschließend 0, was einer leeren Liste entspricht.

C.3. Zugriffsmethoden für benutzerdefinierte Funktionen

C.3.1. Zugriff einer UDF auf die ihr zur Verfügung stehenden Daten

Methode	Rückgabetyp	Beschreibung
`retrieveUDTParameter()`	UDT	Liefert den Inhalt des UDTs, mit dem die Funktion verknüpft ist
`retrieveInputParameter()`	TYPE	Liefert für eine *schreibende* UDF den von der Aktivität an die Funktion übergebenen Wert. Für *lesende* UDFs ist der Rückgabewert der zugehörige UDT, äquivalent zu `retrieveUDTParameter()`.
`storeOutputParameter(TYPE arg)`	-	Wird von einer *schreibenden* UDF verwendet, um ihren Ausgabewert `arg` an das zu ihrer Instanz gehörende Datenelement zu übergeben. *Lesende* UDFs verwenden diese Funktion zur Datenübergabe an die zugreifende Aktivität. Der übergebene Wert wird während der Laufzeit der UDF in einem Container zwischengespeichert. Die Übergabe der Daten in das Datenelement bzw. den Aktivitätenparameter findet nach Beendigung der UDF statt.

C.4. Zugriffsmethoden für Aufzählungen

C.4.1. Feldbeschreibung des Datentyps *enumValue*

Feld	Datentyp	Beschreibung
`ordinal`	int	Die Ordnungszahl des Wertes innerhalb der umgebenden Aufzählung.
`value`	String	Der Bezeichner eines Aufzählungswertes. Innerhalb einer Aufzählung eindeutig.
`description`	String	Die für Menschen lesbare Beschreibung eines Wertes. Darf `NULL` sein.

C.4.2. Sicht des Aktivitätenimplementierers

Methode	Rückgabetyp	Beschreibung
`getValue()`	enumValue	Liefert den aktuellen Wert einer Aufzählung.
`getPossibleValues()`	ListOf(enumValue)	Liefert eine Liste mit allen möglichen Werten.
`setValue(String Value)`	-	Setzt den Wert eines Ausgabeparameters anhand seines eindeutigen Bezeichners. Korreliert mit setValue(int).
`setValue(int Value)`	-	Setzt den Wert eines Ausgabeparameters anhand seiner Ordnungszahl. Korreliert mit setValue(String).

D. Wichtige Funktionen in ADEPT2

c_succ(n) (c_pred(n))	Menge aller *direkten* Vorgängerknoten (Nachfolgerknoten) von n bzgl. normaler Kontrollkanten (Kantentyp: `CONTROL_E`).
c_succ*(n) (c_pred*(n))	Menge aller *direkten* und *indirekten* (transitiven) Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten (Kantentyp: `CONTROL_E`).
succ(n) (pred(n))	Menge aller *direkten* Vorgängerknoten (Nachfolgerknoten) von n bzgl. normaler Kontroll- und Sync-Kanten.
succ*(n) (pred*(n))	Menge aller *direkten* und *indirekten* (transitiven) Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontroll- und Sync-Kanten.
$join^{CFS}(s)$ ($split^{CFS}(j)$)	Im Kontrollflussgraphen CFS zugehöriger Join-Knoten (Split-Knoten) zum Split-Knoten s (Join-Knoten j).
$endloop^{CFS}(L_S)$ ($startloop^{CFS}(L_E)$)	Im KF-Graphen CFS zugehöriger Schleifenendknoten (Schleifenanfangsknoten) des Schleifenanfangsnotens L_S (Schleifenendknotens L_E).
$Id_{source}{}^{e}$ ($Id_{dest}{}^{e}$)	Quellknoten (Zielknoten) der Kante e.
$BranchNodes^{CFS}(n_1,n_2)$	Bestimmt den Split- und Join-Knoten (s,j) des kleinsten, die beiden Knoten n_1 und n_2 umschließenden Verzweigungsblocks.
$MinBlock^{CFS}(N^*)$	Minimaler Kontrollblock im KF-Graphen CFS, der alle Knoten der Menge N^* umschließt.
$WriterExists^{CFS\ DFS}(n,d)$	Bestimmt, ob ein obligater Lesezugriff des Knotens n auf das Datenelement d bei dem gegebenen Schema (CFS, DFS) sicher versorgt ist.
$ParallelWriterExists^{CFS\ DFS}(n,d)$	Gibt es einen zu n parallelen Knoten von dem eine Schreibkante auf das Datenelement d ausgeht?
ReadDataElement(...,n,d)	Kontextabhängiger Wert für den Lesezugriff des Knotens n auf d.

Übersicht über die wichtigsten Funktionen in ADEPT2 (aus [Rei00])

E. Algorithmen

Algorithmus 1 ReadsConsuming(DFS,n,d)

```
//Liefert TRUE zurück, falls n in DFS konsumierend-lesend auf d (oder eines der
Felder von d) zugreift, sonst FALSE.
function ReadsConsuming(DFS,n,d)
  //Bei nicht-strukturierten Datentypen bezieht sich f auf das gesamte Datum
  for all f ∈ d do
    if ∃ par ∈ InParams(n)|(f, n, par, read_consuming) ∈ DFS then
      return TRUE;
    end if
  end for
  //Es existiert keine Datenkante für den konsum. Lesezugriff auf n in DFS
  return FALSE;
end function
```

Algorithmus 2 LoopKillsSupply(CFS,DFS,n,d,L_E)

//Liefert TRUE zurück, falls in der zu L_E gehörenden Schleife zwischen n und L_E konsumierend gelesen, aber zwischen L_S und n nicht obligat geschrieben wird. Der WriterExistsConsuming-Algorithmus verwendet diese Funktion um herauszufinden, ob die Versorgung des zu untersuchenden Knotens in dessen Nachfolgermenge beeinträchtigt werden könnte. Dies ist der Fall, wenn die beiden nachfolgend formulierten Bedingungen erfüllt sind.

```
function LoopKillsSupply(CFS,DFS,n,d,L_E)
  L_S := startloop(L_E);
  if ∃x ∈ {c_succ*(n_read) ∩ c_pred*(L_E)} ∪ {n_read, L_E} | ReadsConsuming(DFS, x, d)
then
    if ∄y ∈ {c_pred*(n_read) ∩ c_succ*(L_S)} ∪ {L_S} | Act.Writes^DFS(y, d, MAND.) then
      return TRUE;
    end if
  end if
  return FALSE;
end function
```

Algorithmus 3 WritesInconsistent(DFS,m,F_n)

```
//Liefert TRUE zurück, falls m eine echte Teilmenge von F_n schreibt.
function WritesInconsistent(DFS,m,F_n)
  if ActivityWrites^DFS(m, F_m ⊂ F_n, ANY) then
    return TRUE;
  end if
  return FALSE; //Schreiben von m beeinflusst nicht das Lesen von d durch n
end function
```

Algorithmus 4 LoopSupplied(DFS,N,d)

//Überprüft alle Knoten der übergebene Knotenmenge N auf Versorgung bzgl. des Lesezugriffs auf das Datenelement d. Alle in der Menge enthaltenen Schleifenendknoten können als versorgt angesehen werden, wenn sie als versorgt markiert sind und diese Versorgung auch über ihre Schleifenrücksprungkante weitergeben. Dies ist nicht der Fall, wenn sie d selbst konsumierend lesen, aber nicht obligat schreiben. Die Knotenmenge gillt als nicht *LoopSupplied*, wenn einer der Schleifenendknoten die Versorgung über die Schleifenrücksprungkante nicht weitergibt. Außerdem gilt die Menge als nicht versorgt, wenn ein enthaltener Knoten noch nicht als versorgt markiert ist. Neben den Schleifenendknoten aller n_{read} umgebenden Schleifen ist auch n_{read} selbst in N enthalten

```
function LoopSupplied(DFS,N,d)
  for all n ∈ N do
    if (NodeType(n) = ENDLOOP ∧ ReadsConsuming(CFS,n,d) ∧
¬ActivityWrites^DFS(n, d, MANDATORY)) ∨ ¬Supplied(n) then
      return FALSE;
    end if
  end for
  return TRUE; //Alle Schleifenendknoten sind versorgt und lesen nicht selbst
                 konsumierend, ohne auch wieder obligat zu schreiben.
end function
```

Algorithmus 5 WriterExistsConsuming()

```
//Änderungen ggü. dem urspr. WriterExists-Algorithmus sind hervorgehoben
function WriterExistsConsuming(CFS, DFS, n_read, d)
  N_read = {n_read}; //Menge der zu untersuchenden Knoten. Initial ist nur n_read
                     enthalten.
  //Zunächst wird geprüft, ob sich der lesende Knoten in einer oder mehreren
    Schleife befindet und ob zwischen n_read und L_E konsumierend gelesen wird.
    Vorausgesetzt wird, dass die Menge der Nachfolger geordnet ist und somit
    die Schleifenendknoten verschachtelter Schleifen von „innen nach außen“
    abgearbeitet werden.
  for all L_E ∈ c_succ*(n_read) | startloop^CFS(L_E) ∈ c_pred*(n_read) do
    if LoopKillsSupply(CFS,DFS,n_read,d,L_E) then
      N_read = N_read ∪ {L_E};
    end if
  end for
  n_read := LastElementOf(N_read); //Es wird der Endknoten der äußersten,
                                     n_read umgebenden Schleife untersucht.
  //Liegt n_read in einer oder mehreren Schleifen, in denen zwischen n_read und
    dem Schleifenendknoten L_E konsumierend gelesen wird, wird im Folgenden
    der Endknoten der äußersten, n umgebenden Schleife mit innenliegendem
    konsumierendem Lesezugriff auf Versorgung überprüft.
  Pred := c_pred*(n_read); //Menge aller Vorgänger von n_read
  //Initialisierung der Zähler und des Attributs „versorgt“ für alle Knoten
  for all n ∈ N do
    Supplied(n) := FALSE;
    Counter(n) := 0;
  end for
  NodeList := ∅ //Die zu untersuchende Knotenmenge ist zunächst leer
  //Zunächst werden alle Nachfolger von obligat schreibenden Knoten in die zu
    untersuchende Knotenmenge aufgenommen
  for all n ∈ Pred do
    if ActivityWrites^DFS(n,d,MANDATORY) then
      for all n* ∈ c_succ(n) do
        if n* ∈ Pred ∪ {n_read} then
          Counter(n*)++; NodeList := NodeList ∪ {n*};
        end if
      end for
    end if
  end for
```

```
    //Die Menge der zu untersuchenden Knoten wird dann sukzessive erweitert.
      Dabei werden alle Nachfolger von Knoten, die bereits versorgt markiert sind,
      selbst als versorgt markiert, sofern ihr Vorgänger nicht konsumierend lesend
      auf d zugreift.
30: while NodeList ≠ ∅ ∧¬LoopSupplied(DFS, N_read, d) do
      NewNodeList := ∅;
      for all n ∈ NodeList do
        if V_in^n ∈ {ONE_OF_ONE, ALL_OF_ALL} ∨ (Counter(n) = EKK(n)) then
  //EKK = Anzahl Eingangskontrollkanten von n
          Supplied(n):=TRUE;
35:       for all n* ∈ c_succ(n) do
            if (n* ∈ Pred ∪ {n_read} ∧ ¬Supplied(n*) ∧¬ReadsConsuming(DFS, n, d)
  then
              Counter(n*)++;
              NewNodeList := NewNodeList ∪ {n*};
            end if
40:       end for
        end if
      end for
      NodeList := NewNodeList;
    end while

45: //Sind nun schon alle zu untersuchenden Knoten (n_read und alle zu berück-
      sichtigenden Schleifenendknoten) als versorgt markiert, dann kann die Un-
      tersuchung abgebrochen werden.

    if LoopSupplied(DFS, N_read, d) then
      return TRUE
    end if

    //Sind noch nicht alle zu untersuchenden Knoten versorgt, wird nun die Ver-
      sorgung über Sync-Kanten geprüft (Schritt 6 des orig. WriterExists).

50: SyncPred:=pred*(n_read); //Vorgänger bzgl. Kontroll- und Sync-Kanten
    ParallelNodes:=SyncPred-Pred; //Vorgänger aus parallelen Zweigen
    Writers := {n ∈ ParallelNodes | ActivityWrites^DFS(n, d, MANDATORY)};
    if Writers = ∅ then
      return FALSE;
55: end if
    for all n ∈ SyncPred ∪ {n_read} do
      SyncSupplied(n):=FALSE;
    end for
```

```
    for all e ∈ E do
60:   if Id^e_dest ∈ SyncPred ∪ {n_read} ∧ ¬Supplied(Id^e_dest) then
        Consider(e):=TRUE;
      else
        Consider(e):=FALSE;
      end if
65:   end for
    for all n ∈ Writers do
      for all e ∈ E with (Id^e_source = n∧ Consider(e)) do
        if ET^e = CONTROL_E then
          Counter(Id^e_dest)++;
70:       NodeList := NodeListe ∪ {Id^e_dest};
        else if ET^e ∈ {SOFT_SYNC_E,STRICT_SYNC_E} then
          if SyncDestSupplied(e) then
            SyncSupplied(Id^e_dest):=TRUE;
            NodeList := NodeList ∪ {Id^e_dest};
75:       end if
        end if
      end for
    end for
    if NodeList = ∅ then
80:   for all e ∈ E|ET^e ∈ {SOFT_SYNC_E,STRICT_SYNC_E} ∧ Id^e_source ∉ Writers do
        if Consider(e) ∧ SyncDestSupplied(e) then
          SyncSupplied(Id^e_dest):=TRUE;
          NodeList := NodeList ∪ {Id^e_dest};
        end if
85:   end for
    end if
    while NodeList ≠ ∅ ∧¬LoopSupplied(DFS,N_read,d) do //Alle L_E und n_read
                                                       müssen versorgt sein
      NewNodeList := ∅ //Menge der in der nächsten Iteration zu
                         betrachtenden Knoten
      for all n ∈ NodeList do
90:     //Darf der Knoten n das Datenelement d obligat lesen?
        if (V^n_in ∈ {ONE_OF_ALL,ALL_OF_ALL} ∧ Counter(n) ≥ 1) ∨
   (V^n_in =ONE_OF_ALL∧Counter(n) = EKK(n)) ∨ (SyncSupplied(n)) then
          Supplied:= TRUE;
          for all e ∈ E with Id^e_dest = n do
            Consider(e):=FALSE;
95:       end for
        end if
```

```
        if Supplied(n) ∧¬ReadsConsuming(DFS, n, d) then //Nachfolger nur weiter untersuchen, wenn n nicht konsumierend liest.
          for all e ∈ E with (Id^e_source = n) ∧ Consider(e) do
            if ET^e = CONTROL_E then
100:          Counter(Id^e_dest)++;
              NewNodeList := NewNodeList ∪ {Id^e_dest};
            else if ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E then;
              if SyncDestSupplied(e) then
                SyncSupplied(Id^e_dest) = TRUE
105:            NewNodeList := NewNodeList ∪ {Id^e_dest};
              end if
            end if
          end for
        end if
110:  end for
      if NewNodeList = ∅ then
        for all e ∈ E with ET^e ∈ {SOFT_SYNC_E, STRICT_SYNC_E ∧ Consider(e) do
          if SyncDestSupplied(e) ∧¬ReadsConsuming(DFS, Id^e_source, d) then
            SyncSupplied(Id^e_dest) := TRUE;
115:        Consider(e) := FALSE;
            NewNodeList := NewNodeList ∪ {Id^e_dest};
          end if
        end for
      end if
120:  NodeList := NewNodeList;
    end while
    //Konnten alle zu untersuchenden Knoten aus der Menge N_read mit „versorgt“
      markiert werden, kann auch für n_read „versorgt“ gemeldet werden, ansonsten
      gibt der Algorithmus „FALSE“ zurück.
    return LoopSupplied(DFS, N_read, d);
  end function
```

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACID atomicity, consistency, isolation and durability; erwünschte Eigenschaft von Transaktionen in Datenbanksystemen

AR Aktivitäten-Repository

BLOB Binary Large Object

BPEL Business Process Execution Language

BPMN Business Process Modeling Notation

BPM Business Process Management

CFS Control Flow Schema

CRM Customer Relationship Management

CSV Comma-Separated Values

DMS Dokumenten-Management-System

EDV Elektronische Datenverarbeitung

ERP Enterprise Resource Planning

GUI Graphical User Interface, graphische Benutzeroberfläche

HRM Human Resource Management

OMG Object Management Group

PMS Prozess-Management-System

PTE Process Template Editor

SAR Staff Assignment Rule

SOAP SOAP (Name eines Protokolls); vormals: Simple Object Access Protocol

TAN Transaktionsnummer

TCP/IP Transmission Control Protocol/Internet Protocol

UDF User Defined Function

UDT User Defined Type

XML-RPC XML Remote Procedure Call

XML Extensible Markup Language

XPath XML Path Language

XPDL XML Process Definition Language

XQuery XML Query Language

XSL Extensible Stylesheet Language

XSLT XSL-Transformation

YAWL Yet Another Workflow Language

Literaturverzeichnis

[AADH04] Aalst, Wil M. P. v. d. ; Aldred, Lachlan ; Dumas, Marlon ; Hofstede, Arthur H. M.: Design and Implementation of the YAWL System. In: Persson, Anne (Hrsg.) ; Stirna, Janis (Hrsg.): *CAiSE* Bd. 3084, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–22151–4, S. 142–159

[ABJ+04] Altintas, Ilkay ; Berkley, Chad ; Jaeger, Efrat ; Jones, Matthew ; Ludäscher, Bertram ; Mock, Steve: Kepler: An Extensible System for Design and Execution of Scientific Workflows. In: *Scientific and Statistical Database Management, International Conference on* 0 (2004), S. 423. – ISSN 1099–3371

[AH03] Aalst, W.M.P. van der ; Hofstede, A.H.M. ter: *YAWL: Yet Another Workflow Language (Revised Version)*. Queensland University of Technology, Brisbane : QUT Technical report, FIT-TR-2003-04, 2003

[AHKB03] Aalst, Wil M. P. v. d. ; Hofstede, Arthur H. M. ; Kiepuszewski, Bartek ; Barros, Alistair P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Nr. 1, S. 5–51

[Bau96] Baumgarten, Bernd: *Petri-Netze. Grundlagen und Anwendungen*. 2. Auflage. Spektrum Akademischer Verlag, 1996

[BDS+93] Breitbart, Y. ; Deacon, A. ; Schek, H.-J. ; Sheth, A. ; Weikum, G.: Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. In: *SIGMOD Rec.* 22 (1993), Nr. 3, S. 23–30. – ISSN 0163–5808

[Ber05] Berroth, Marco: *Konzeption und Entwurf einer Komponente für Organisationsmodelle*, Universität Ulm, Diplomarbeit, 2005

[Bla96] Blaser, Rainer: *Konfiguration verteilter Anwendungen aus vorgefertigten Programmbausteinen*, Universität Ulm, Diplomarbeit, 1996

[CGS09] Colazzo, Dario ; Ghelli, Giorgio ; Sartiani, Carlo: Efficient asymmetric inclusion between regular expression types. In: *ICDT '09: Proceedings of the 12th International Conference on Database Theory*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–423–2, S. 174–182

[CW85] Cardelli, Luca ; Wegner, Peter: On Understanding Types, Data Abstraction, and Polymorphism. In: *ACM Computing Surveys* 17 (1985), S. 471–522

[DA05] Dumas, Marlon ; Aalst, Wil M. P. v. d.: *Process Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley-Interscience, 2005

[DAG+06] Dadam, Peter ; Acker, Hilmar ; Göser, Kevin ; Jurisch, Martin ; Kreher, Ulrich ; Lauer, Markus ; Rinderle, Stefanie ; Reichert, Manfred: ADEPT2 - Ein adaptives Prozess-Management-System der nächsten Generation. In: *Science meets Business - Aktuelle Trends aus der Softwaretechnik-Forschung, Tagungsband Stuttgarter Softwaretechnik Forum*, 2006

[DR04] Dadam, Peter ; Reichert, Manfred: ADEPT - Prozess-Management-Technologie der nächsten Generation. In: *Aktuelle Trends in der Softwareforschung, Tagungsband zum doIT Software-Forschungstag 2003*, 2004

[EN09] Elmasri, Ramez ; Navathe, Shamkant: *Grundlagen von Datenbanksystemen*. 3. aktualisierte Auflage. Pearson Studium, 2009

[Eur08] Eurostat: *Internetzugang und Internetnutzung in der EU27 im Jahr 2008*. Dezember 2008

[GJA+07] Göser, Kevin ; Jurisch, Martin ; Acker, Hilmar ; Kreher, Ulrich ; Lauer, Markus ; Rinderle, Stefanie ; Reichert, Manfred ; Dadam, Peter: *Next-generation Process Management with ADEPT2 (Demo Paper)*. Aachen : http://eprints.eemcs.utwente.nl/10835/, September 2007 (CEUR Workshop Proceedings)

[GR83] Goldberg, Adele ; Robson, David ; Harrison, Michael A. (Hrsg.): *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983

[Gro07] Grosskopf, Alexander: *xBPMN Formal Control Flow Specification of a BPMN based Process Execution Language*, Hasso-Plattner-Institut, Universität Potsdam, Diplomarbeit, Juli 2007

[HM05] Harold, Elliotte R. ; Means, W. S.: *XML in a nutshell*. O Reilly, 2005

[Hun97] Hunt, John: *Smalltalk and Object Orientation*. Springer-Verlag Telos, 1997

[IGH+02] Ii, John D. ; Goel, Mudit ; Hylands, Christopher ; Kienhuis, Bart ; Lee, Edward A. ; Liu, Jie ; Liu, Xiaojun ; Muliadi, Lukito ; Neuendorffer, Steve ; Reekie, John ; Smyth, Neil ; Tsay, Jeff ; Xiong, Yuhong: *PTOLEMY II- Heterogeneous Concurrent Modeling and Design in Java*. http://ptolemy.eecs.berkeley.edu/publications/papers/02/ptIIdesign/design.pdf. Version: 2002. – Abgefragt Juni 2009

[Inu08a] Inubit AG (Hrsg.): *Modulhandbuch, Teil 1: Data Converter, Format Adapter, Utilities, Workflow und WS Controls*. Schöneberger Ufer 89-91, 10785 Berlin Germany: Inubit AG, 2008

[Inu08b] Inubit AG (Hrsg.): *Modulhandbuch, Teil 2: System Connectoren*. Schöneberger Ufer 89-91, 10785 Berlin Germany: Inubit AG, 2008

[Jae04] Jaeger, Burkhard: *Humankapital und Unternehmenskultur*. Gabler, 2004

[Jen81] Jensen, K.: Coloured Petri Nets and the invariant-method. In: *Theoretical Computer Science* 14 (1981), S. 317–336

[Jen86] Jensen, K.: Colored petri nets. In: *Lecture Notes Comp. Sci.: Advances in petri nets* 254 (1986), S. 248–299

[Jur06] Jurisch, Martin: *Konzeption eines Rahmenwerkes zur Erstellung und Modifikation von Prozessvorlagen und -instanzen*, Universität Ulm, Diplomarbeit, 2006

[KBP07] Kneipp, Sean ; Bradford, Lindsay ; Prestedge, Jessica: *YAWL Editor 1.5 User Manual - Release 1.5*, August 2007. http://yawlfoundation.org/yawldocs/YAWLEditor1.5UserManual.pdf. – Abgefragt Juni 2009

[LA94] Leymann, Frank ; Altenhuber, Wolfgang: Managing Business Processes in an Information Resource. In: *IBM Systems Journal* 33 (1994), Nr. 2, S. 326–348

[Mey96] Meyer, Bertrand: The many faces of inheritance: a taxonomy of taxonomy. In: *IEEE Computer* 29 (1996), Nr. 5, S. 105–108

[Obj06] Object Management Group (OMG) (Hrsg.): *Business Process Modeling Notation*. Object Management Group (OMG), Februar 2006. http://bpmn.org/Documents/OMG%20Final%20Adopted%20BPMN%201-0%20Spec%2006-02-01.pdf. – Abgefragt Juni 2009

[Obj08a] Object Management Group (OMG) (Hrsg.): *Business Process Modeling Notation 2.0 (BPMN2) - Revised Submission*. Object Management Group (OMG), September 2008. http://www.omg.org/cgi-bin/doc?bmi/2008-09-07. – Abgefragt Juni 2009

[Obj08b] Object Management Group (OMG) (Hrsg.): *Business Process Modeling Notation, V1.1*. Object Management Group (OMG), Januar 2008. http://www.bpmn.org/Documents/BPMN%201-1%20Specification.pdf. – Abgefragt Juni 2009

[Obj09] Object Management Group (OMG) (Hrsg.): *Business Process Modeling Notation, V1.2*. Object Management Group (OMG), Januar 2009. http://www.omg.org/docs/formal/09-01-03.pdf. – Abgefragt Juni 2009

[ODHA07] Ouyang, Chun ; Dumas, Marlon ; Hofstede, Arthur H. ; Aalst, Wil M. P. v. d.: Pattern-based translation of BPMN process models to BPEL web services. In: *International Journal of Web Services Research (JWSR)* 5(1) (2007), March, S. 42–62

[Ouy05] Ouyang, Chun: *How to Manipulate Data in YAWL*, November 2005. http://yawlfoundation.org/yawldocs/YAWLDataManual-beta7.pdf. – Abgefragt Juni 2009

[Pet62] Petri, Carl A.: *Kommunikation mit Automaten*, Technischen Universität Darmstadt, Diss., 1962

[PH09] Poetzsch-Heffter, Arnd: *Konzepte objektorientierter Programmierung*. Springer, 2009

[RAH06] Russell, Nick ; Aalst, Wil M. P. v. d. ; Hofstede, Arthur H. M.: Workflow Exception Patterns. In: Dubois, Eric (Hrsg.) ; Pohl, Klaus (Hrsg.): *CAiSE* Bd. 4001, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–34652–X, S. 288–302

[RAH08] Russell, Nick ; Aalst, Wil M. P. v. d. ; Hofstede, Arthur H.: newYAWL: Designing a Workflow System using Coloured Petri Nets. In: N. Sidorova, H. R. D. Moldt M. D. Moldt (Hrsg.) ; Xidian University (Veranst.): *Proceedings of the International Workshop on Petri Nets and Distributed Systems (PNDS'08)* Xidian University, 2008, S. 67–84

[RD03] Rinderle, Stefanie ; Dadam, Peter: Schemaevolution in Workflow-Management-Systemen. In: *Informatik Spektrum* 26 (2003), Nr. 1, S. 17–19

[Rei00] Reichert, Manfred: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*, Universität Ulm, Diss., 2000

[RHA07] Russell, Nicholas ; Hofstede, Arthur H. ; Aalst, Wil M. P. v. d.: newYAWL: Specifying a Workflow Reference Language using Coloured Petri Nets. In: *Proceedings Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools - CPN'07*, 2007, S. 107–126

[RHAE07] Russell, Nick ; Hofstede, Arthur H. M. ; Aalst, Wil M. P. v. d. ; Edmond, David: newYAWL: Achieving Comprehensive Patterns Support in Workflow for the Control-Flow, Data and Resource Perspectives / BPM Center. 2007 (07-05). – Forschungsbericht

[RHAM06] Russell, Nick ; Hofstede, Arthur H. ; Aalst, Wil M. P. v. d. ; Mulyar, Nataliya: Workflow Control-Flow Patterns, A Revised View / BPMcenter.org. 2006 (BPM-06-22). – Forschungsbericht

[RHEA04a] Russell, Nick ; Hofstede, Arthur H. ; Edmond, David ; Aalst, Wil M. P. v. d.: Workflow Data Patterns / Queensland University of Technology, Brisbane, Australia. 2004 (FIT-TR-2004-01). – Forschungsbericht

[RHEA04b] Russell, Nick ; Hofstede, Arthur H. ; Edmond, David ; Aalst, Wil M. P. v. d.: Workflow Resource Patterns / Queensland University of Technology, Brisbane, Australia. 2004. – Forschungsbericht

[Rin04] Rinderle, Stefanie: *Schema Evolution in Process Management Systems*, Universität Ulm, Diss., 2004

[RKD96] Reichert, Manfred ; Kuhn, Klaus ; Dadam, Peter: Prozeßreengineering und -automatisierung in klinischen Anwendungsumgebungen. In: Baur, M. P. (Hrsg.) ; Fimmers, R. (Hrsg.) ; Blettner, M. (Hrsg.): *GMDS*, MMV Medizin Verlag, 1996, S. 219–223

[RKD97] Reichert, Manfred ; Kuhn, Klaus ; Dadam, Peter: Optimierung und Unterstützung von Leistungsprozessen im Krankenhaus - Perspektiven, Erfahrungen und Grenzen. In: *Proc. 20. Deutscher Krankenhaustag und INTRHOSPITAL/INTERFAB 97*, 1997

[RRD04] Rinderle, Stefanie ; Reichert, Manfred ; Dadam, Peter: Correctness criteria for dynamic changes in workflow systems–a survey. In: *Data & Knowledge Engineering* Bd. 50, 2004. – ISSN 0169–023X, S. 9 – 34. – Advances in business process management

[Sch00] Schöning, Uwe: *Theoretische Informatik - kurzgefasst*. Spektrum Akademischer Verlag, 2000

[Sch05] Schiedermeier, Reinhard: *Programmieren mit Java*. Pearson Studium, 2005

[Sch08] Schreiter, Torben: *xBPMN++ Towards Executability of BPMN: Data Perspective and Process Instantiation*, Hasso-Plattner-Institut, Universität Potsdam, Diplomarbeit, März 2008

[VAH07] Verbeek, H. M. W. ; Aalst, Wil M. P. v. d. ; Hofstede, Arthur H. M.: Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. In: *Comput. J.* 50 (2007), Nr. 3, S. 294–314

[WAHE06] Wynn, Moe T. ; Aalst, Wil M. P. v. d. ; Hofstede, Arthur H. M. ; Edmond, David: Verifying Workflows with Cancellation Regions and OR-Joins: An Approach Based on Reset Nets and Reachability Analysis. In: Dustdar, Schahram (Hrsg.) ; Fiadeiro, José Luiz (Hrsg.) ; Sheth, Amit P. (Hrsg.): *Business Process Management* Bd. 4102, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–38901–6, S. 389–394

[Wol08] Wolz, Jonas: *Erweiterte Kontroll- und Datenflusskonzepte in ADEPT*, Universität Ulm, Diplomarbeit, 2008

[WS07] Wilhelm, Reinhard ; Seidl, Helmut: *Übersetzerbau Virtuelle Maschinen*. Springer, 2007

Danksagung

Allen voran möchte ich mich bei meiner Familie bedanken, die mich in schwierigen Situationen stets unterstützt hat. Durch Eure Liebe und Euer Verständnis konnte ich die Ruhe und die Konzentration aufbringen, die für das Gelingen dieser Arbeit notwendig waren.

Besonderer Dank gilt meinem Betreuer Ulrich Kreher. Durch seine konstruktive Kritik und seine Motivation ist es mir gelungen, mich an dieser Arbeit nicht nur fachlich, sondern auch persönlich immens weiterzuentwickeln. Sein Anspruch und sein Einsatz für seine Diplomanden machen ihn zum besten Betreuer, den man sich für eine Diplomarbeit wünschen kann.

Weiter möchte ich mich beim Institut für *Datenbanken und Informationssysteme* (DBIS) der Universität Ulm bedanken. Namentlich genannt seien Prof. Dr. Manfred Reichert, PD Dr. Stefanie Rinderle-Ma, Andreas Lanz, Jens Kolb und Rüdiger Pryss. Bei Prof. Reichert bedanke ich mich für seine uneingeschränkte Unterstützung, die auch dann noch Bestand hatte, als andere bereits an mir zweifelten. Sein Wissen und seine Erfahrung haben maßgeblich zum Gelingen dieser Arbeit beigetragen. Andreas Lanz danke ich für aufschlussreiche Diskussionen, Jens Kolb und Rüdiger Pryss für die Ausstattung mit Software und wissenschaftlichen Unterlagen.

Bei den Mitarbeitern der *AristaFlow GmbH* möchte ich mich herzlich für meine Unterkunft während der gesamten Arbeit bedanken. Durch die Nähe zu Euch konnte ich stets auf umfangreiches Wissen und eine freundschaftliche Motivation zurückgreifen. Für Euren weiteren Weg wünsche ich Euch alles Gute!

Vierzehn Augen sehen mehr als zwei. Für das Korrekturlesen bedanke ich mich bei Lisa Haberstroh, Susanne Forschner, Thilo Schmitt, Oliver Gableske, Marian Lindenlaub und Ulrich Kreher.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 29. Juni 2009 . . . . . .

Alexander Forschner