ulm university universität
uulm

Effiziente Repräsentation von Vorlagen- und Instanzdaten in Prozess-Management-Systemen

Ulrich Kreher, Manfred Reichert,
Stefanie Rinderle-Ma, Peter Dadam

Ulmer Informatik-Berichte

Nr. 2009-08
Oktober 2009

Effiziente Repräsentation von Vorlagen- und Instanzdaten in Prozess-Management-Systemen*

Ulrich Kreher, Manfred Reichert, Stefanie Rinderle-Ma, Peter Dadam

Institut für Datenbanken und Informationssysteme, Universität Ulm,
{Ulrich.Kreher, Manfred.Reichert, Stefanie.Rinderle, Peter.Dadam}@uni-ulm.de

Zusammenfassung

Ein Prozess-Management-System (PMS) muss umfangreiche Funktionen für die Steuerung, Verwaltung und Änderung von Geschäftsprozessen bieten. Um breit einsetzbar zu sein, ist eine performante Ausführung dieser Funktionen unverzichtbar, insbesondere bei großer Anzahl von Prozessinstanzen. In diesem Beitrag untersuchen wir einen wichtigen Performanzaspekt von PMS, die effiziente Speicherrepräsentation von Vorlagen- und Instanzdaten. Dazu setzen wir ein logisches Prozessmetamodell in programmiersprachenunabhängige technische Datenstrukturen um und zwar sowohl für Primär- als auch Sekundärspeicher. Wichtigstes Ziel ist die Minimierung des Primärspeicherbedarfs. Nur dadurch lässt sich zeitaufwendiges Ein- und Auslagern der zur Laufzeit benötigten Vorlagen- und Instanzdaten minimieren. Wir diskutieren grundlegende Realisierungsalternativen für die Speicherrepräsentation dieser Daten, und gehen darüber hinaus auf fortschrittliche Speicherkonzepte, wie die Clusterung von Prozessinstanzen, ein. Die vorgestellten Konzepte sind aktuell im ADEPT2-PMS umgesetzt.

1 Einleitung

Mit Service-orientierten Architekturen, die eine flexible Komposition von Anwendungsdiensten erlauben sollen, rückt die Idee von Prozess-Management-Systemen (PMS) verstärkt in den Blickpunkt [10]. Grundlegend ist hier die Trennung von Prozess- und Anwendungslogik sowie die explizite Steuerung der Prozesse durch eine Prozess-Engine [28]. In vielen Anwendungsdomänen darf ein PMS die Benutzer jedoch nicht einschränken, sondern muss bei Bedarf auch Änderungen der Prozesse zur Laufzeit zulassen [19, 23]. Heutige PMS beachten diesen Aspekt nicht in ausreichendem Maße [35, 45]. Zwar gibt es mittlerweile einige Prototypen adaptiver PMS (z. B. [31, 46, 40]), diese sind jedoch weder auf Benutzerfreundlichkeit noch auf Performanz ausgelegt. In diesem Beitrag untersuchen wir einen wichtigen Performanz-Aspekt eines jeden PMS: die effiziente Speicherrepräsentation von Vorlagen- und Instanzdaten [38, 34].

Ausgangspunkt jeglicher Performanzbetrachtungen bei PMS bildet ihre Architektur (s. Abb. 1) [17, 30]. Mit einem *Vorlagen-Editor* werden Prozessmodelle (sog. *Prozessvorlagen*) erstellt, die den Ablauf und die Struktur der Prozesse in maschinenlesbarer Form darstellen. Gespeichert werden Prozessvorlagen vom *Prozessmanager* und der dazugehörigen *Prozess-Datenbank*. Davon ausgehend können neue (*Prozess-*)*Instanzen* erzeugt und gestartet werden. Die dann jeweils zur Ausführung anstehenden (*Prozess-*)*Aktivitäten* – sofern es sich nicht um automatisch auszuführende Schritte handelt – werden Benutzern über ihre Arbeitslisten zur Ausführung zugeordnet. Der *Ausführungsmanager* steuert dann diesen Prozess, z. B. das Starten externer Anwendungsprogramme oder das *Weiterschalten* einer Prozessinstanz nach Beenden einer Aktivität.

Für die breite Akzeptanz eines PMS ist eine ausreichende Performanz unerlässlich. Ein ebenfalls wichtiger Aspekt, der in existierenden PMS wie TIBCO Staffware, IBM WebSphere Process

*gefördert durch das Projekt „AristaFlow“ im Rahmen des Forschungsverbundes Unternehmenssoftware Baden-Württemberg

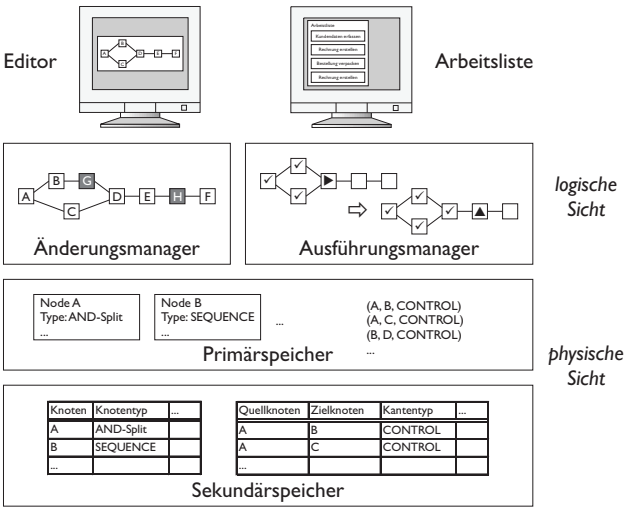

Abbildung 1: Architektur adaptives PMS

Server, jBPM oder Ultimus noch immer stark vernachlässigt wird, betrifft die Flexibilität des Systems. Das von uns entwickelte ADEPT-PMS [27, 10] gestattet es autorisierten Benutzern, zur Laufzeit Änderungen sowohl an einzelnen Instanzen (*instanzspezifische Änderungen*) als auch an Prozessvorlagen vorzunehmen (*typspezifische Schemaevolution*). Letztere können, soweit gewünscht und möglich, auch auf bereits laufende Instanzen propagiert werden [36]. Dies ist sehr performanzkritisch, da viele Instanzen von einer solchen Propagation betroffen sein können und die notwendigen Überprüfungen und Anpassungen für jede Instanz relativ aufwendig sind [36]. Gleichzeitig darf die Durchführung einer Schemaevolution aber nicht dazu führen, dass andere Funktionen des PMS zu sehr beeinträchtigt sind und sich das Antwortzeitverhalten des Systems signifikant verschlechtert.

Bisher gibt es zur effizienten Implementierung solcher fortschrittlichen PMS-Konzepte nur wenig Veröffentlichungen (z. B. [46, 40, 18]). Existierende Ansätze beschreiben zumeist nur ein logisches Modell, auf dem konzeptionelle Fragestellungen (z. B. die Verifikation und korrekte Adaption von Prozessinstanzen betreffend) untersucht werden. Dagegen wird das *Laufzeitmodell*, also die Realisierung entsprechender Datenstrukturen und darauf implementierter Funktionen, nicht näher betrachtet. Prototypische Implementierungen beschränken sich meist auf Modellaspekte und Laufzeit-Simulationen. Damit wird gezeigt, dass die jeweilige Funktionalität prinzipiell realisiert werden kann, jedoch nicht wie dies erfolgen muss, um in der Praxis performant zu sein. Zu kommerziellen Systemen (z. B. WebSphere MQ Workflow, Staffware) gibt es nur wenig detaillierte Informationen. Hinzu kommt, das diese die eingangs motivierte Flexibilität bei weitem nicht bieten.

In diesem Beitrag setzen wir ein logisches Prozess-Metamodell [27] in programmiersprachenunabhängige, implementierbare Datenstrukturen, sowohl für Primär- als auch Sekundärspeicher, um. Wichtigstes Ziel ist die Minimierung des Primärspeicherbedarfs. Nur dadurch lässt sich zeitaufwendiges Ein-/Auslagern der zur Laufzeit benötigten Vorlagen- und Instanzdaten weitgehend vermeiden. Dies ist deshalb kritisch, da bei naiver Implementierung der logischen Konzepte und bei großer Anzahl gleichzeitig laufender Prozesse (100.000) mehrere Gigabyte an Primärspeicher allein für die Instanzrepräsentation nötig wären. Mit dem nachfolgend beschriebenen Ansatz wird eine Speicherplatzeinsparung um den Faktor 7 möglich. Gleichzeitig hat der verfolgte Ansatz nur minimale Auswirkungen auf das Laufzeitverhalten des PMS.

Im Gegensatz zur Repräsentation von Prozessdaten im Primärspeicher steht beim Sekundärspeicher weniger der Speicherplatzbedarf im Vordergrund, als vielmehr eine bestmögliche Unterstützung bestimmter PMS-Funktionen direkt im Sekundärspeicher. Unterstützt die Datenrepräsentation im Sekundärspeicher beispielsweise Anfragen an Kollektionen von Instanzen direkt, lässt

sich teures Ein-/Auslagern von Instanzdaten zumindest teilweise vermeiden.

Die bei der Umsetzung logischer Konzepte und Strukturen in eine Speicherrepräsentation getroffenen Entwurfsentscheidungen können sich gravierend auf die Funktionalität und Performanz des PMS auswirken. So kann eine „schlechte“ Repräsentation dazu führen, dass bestimmte Funktionen nicht mehr oder nur unter großem Aufwand bewerkstelligbar sind. In dieser Arbeit zeigen wir dies beispielhaft anhand des *Weiterschaltens im Kontrollfluss* und der *Schemaevolution* mit anschließender *Änderungspropagation* auf laufende Instanzen. Beide Funktionen sind fundamental für jedes PMS und stellen unterschiedliche Anforderungen an die Repräsentation der Daten. Das Weiterschalten [27] ist eine einfache, häufig benötigte Funktion und damit sehr performanzkritisch. Dies erfordert zwingend eine optimierte Datenrepräsentation. Dagegen ist die Propagation von Anderungen an Prozessvorlagen auf die zugehörigen laufenden Instanzen eine sehr komplexe Funktion, die eher selten benötigt wird [36]. Sie stellt allerdings grundlegende Anforderungen an die Datenrepräsentation.

Für nachfolgende Ausführungen legen wir das von uns entwickelte ADEPT-Prozess-Metamodell [27] zugrunde. Aufgrund der Trennung zwischen logischem Modell und internem Ausführungsmodell lassen sich die angestellten Betrachtungen auch auf andere Prozess-Metamodelle anwenden. In bisherigen Veröffentlichungen zu ADEPT haben wir uns hauptsächlich mit konzeptionellen Fragestellungen adaptiver PMS befasst. Dazu gehören, wie erwähnt, Ad-hoc Änderungen einzelner Instanzen [27] und deren kontrollierte Wiederverwendung [43], ebenso wie Änderungen an Prozessvorlagen und deren Propagation auf laufende Instanzen [36]. Diese sind in einem mächtigen Prototyp implementiert [31, 33], der nicht nur Demonstrationszwecken dient, sondern ein vollwertiges PMS darstellt. Damit ist nicht nur die Modellierung und Ausführung von Prozessen möglich, es werden auch leistungsfähige Programmier-Schnittstellen, benutzerfreundliche Werkzeuge und beispielhafte Anwendungen angeboten. Ein erfolgreicher Einsatz durch Anwender aus Forschung [5, 6, 24, 42] und Industrie zeigen den hohen Nutzen der entwickelten Software.

Der vorliegende Beitrag bildet den Einstieg in weitere Untersuchungen von Performanzaspekten in PMS. Die entwickelten Konzepte wurden im Nachfolgerprototyp des ADEPT-Systems [29] realisiert, in dem neben einer umfassenden Funktionalität großes Gewicht auf einer effizienten und performanten Implementierung liegt. Nach der Vorstellung von Grundlagen eines PMS (Kapitel 2) untersuchen wir in Kapitel 3 konzeptionell die Speicherrepräsentation der Vorlagen- und Instanzdaten. Nach der Untersuchung der spezielle Repräsentation mittels Instanzcluster in Kapitel 4 vergleichen wir die verschiedenen Repräsentationen qualitativ. Anschließend erläutern wir in Kapitel 5 physische Speicherstrukturen. Abschließend folgen in Kapitel 6 eine Zusammenfassung und ein Ausblick.

2 Grundlagen

Für die interne Datenrepräsentation eines PMS ist es einerseits wichtig, welche Informationen bereitzustellen sind, andererseits, wofür diese benötigt bzw. welche Operationen darauf ausgeführt werden. Hieraus ergeben sich funktionale Anforderungen an die Repräsentation, die in diesem Kapitel eingehend untersucht werden. Abschließend werden verschiedene Ausführungsmodelle vorgestellt, also die logische Repräsentation und Interpretation von Prozessdaten.

2.1 Vorlagen- und Instanzdaten

Ausgangspunkt der Diskussion zur effizienten Speicherrepräsentation ist die Betrachtung der Daten selbst (Abb. 2). *Prozessvorlagen* werden vom Benutzer erstellt und bilden ein Modell eines realen Arbeitsablaufs. Die für die Vorlagendefinition verfügbaren Konstrukte sind durch das Prozess-Metamodell festgelegt. Viele Metamodelle basieren auf Graphen (z. B. [21, 20, 27, 25, 2]). Dabei werden Aktivitäten eines Prozesses als Knoten dargestellt. Diese besitzen verschiedene Attribute, die z. B. Bearbeiterzuordnungen oder Anwendungskomponenten angeben. Die Kanten zwischen Knoten geben Relationen (z. B. Reihenfolgebeziehungen) an. Neben *Kontrollflusskanten* gibt es

Kanten für die Abbildung des Datenflusses zwischen Aktivitäten. Die zugehörigen Datenelemente (*Prozessvariablen*) werden durch spezielle Knoten abgebildet.

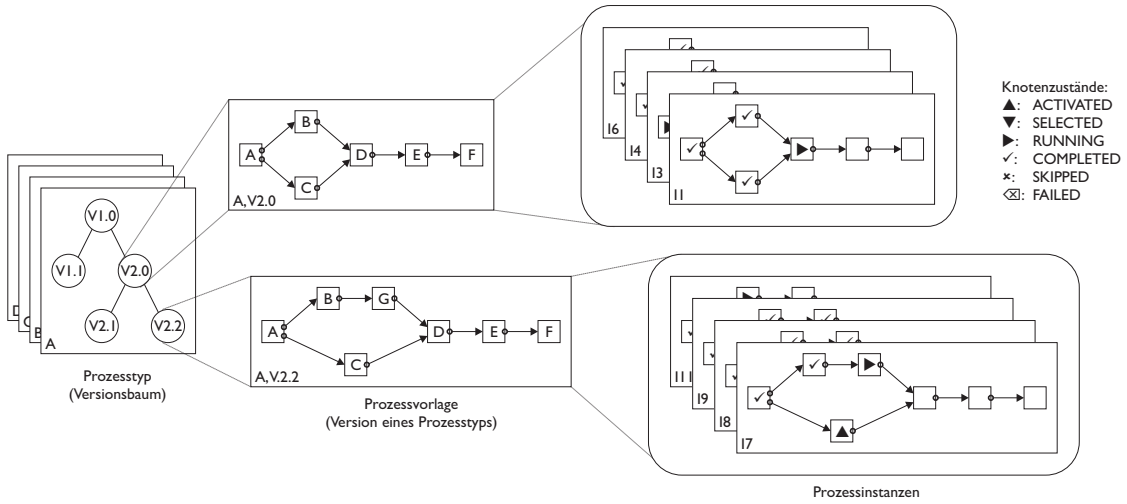

Abbildung 2: Verwaltung von Prozessgraphen (logische Sicht)

Jede Vorlage besitzt einen *Prozesstyp*, der für eine bestimmte Klasse von Prozessvorlagen (z. B. Angebotsbearbeitung, Mahnverfahren) steht. Eine konkrete Vorlage entspricht dann einer bestimmten *Version* des Typs (vgl. Abb. 2). Instanzen einer Vorlage wiederum repräsentieren die Ausführung eines konkreten Prozesses und verwalten den aktuellen Ausführungsstatus einzelner Aktivitäten sowie aktuelle Werte von Prozessvariablen. Die einzelnen Aktivitäten- bzw. Knotenzustände (und damit auch der Gesamtzustand einer Instanz) stellen nur eine Momentaufnahme dar. Um auch vergangene Zustände rekonstruieren zu können (z. B. für Recoveryzwecke), wird zu jeder Instanz eine *Ausführungshistorie* angelegt, in der Informationen zu (wichtigen) Zustandsänderungen protokolliert werden [3, 17]. Hierzu gehört auch eine Datenhistorie für jede Prozessvariable, in der u.a. die Variablenwerte sowie der Zeitpunkt des (schreibenden) Zugriffs festgehalten werden [41]. Einige Prozess-Metamodelle (inkl. dem ADEPT-Metamodell) bieten schließlich eine konsolidierte Sicht auf die Ausführungshistorie, indem die Zustände von Instanzknoten in Form von Markierungen verwaltet werden (Abb. 2).

2.2 Operationen auf Vorlagen- und Instanzdaten

In diesem Abschnitt untersuchen wir ausgewählte Operationen auf Vorlagen- und Instanzdaten. Aus Platzgründen beschränken wir uns auf das Weiterschalten im Kontrollfluss, die Propagation von Vorlagenänderungen auf laufende Instanzen und bestimmte Anfragen an Kollektionen von Instanzen (z. B. Feststellung aktuell ausführbarer Aktivitäten).

Beispiel 1 (Weiterschalten) Das Weiterschalten im Kontrollfluss ist eine wichtige und verhältnismäßig einfache Funktion eines PMS. Sie wird bei bestimmten Zustandsänderungen erforderlich. Ein Beispiel ist das Beenden einer Aktivität, wodurch sich die Zustände der direkten Nachfolgeraktivitäten ändern können. Deshalb müssen diese Knoten neu bewertet werden. Dabei ist zu beachten, dass eine Aktivität nur dann in einen ausführbaren Zustand versetzt werden darf, wenn alle direkten Vorgängeraktivitäten entweder abgeschlossen oder als nicht mehr auszuführen markiert sind. Wir unterstellen hier eine Ausführungssemantik mit Deadpath-Eliminierung. Deshalb müssen bei jeder Neubewertung einer Aktivität die Menge der direkten Vorgänger bestimmt und deren Zustände überprüft werden. Sind nicht alle Vorgänger abgeschlossen, müssen weitere Zustandsänderungen abgewartet werden, bevor die entsprechende Aktivität aktiviert werden kann.

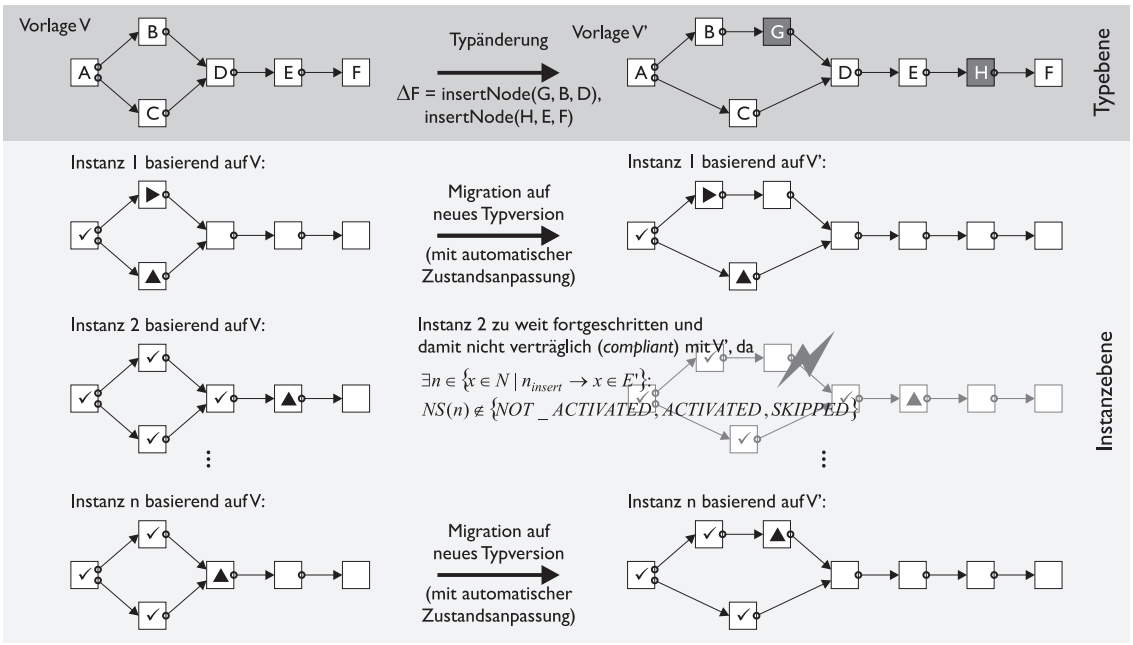

Abbildung 3: Verträglichkeitsprüfung bei Schemaevolution (logische Sicht)

Beispiel 2 (Schemaevolution) Der Benutzer ändert eine existierende Prozessvorlage mittels einer (vollständigen) Menge an vordefinierten Änderungsoperationen ab (siehe [44] für typische Änderungsoperationen), was zu einer neuen Version des Prozesstyps führt (Abb. 2). Insbesondere bei langlaufenden Prozessen ist es wichtig, dass – sofern gewünscht und möglich – die durchgeführten Änderungen auch auf bereits laufende Instanzen der Vorlage propagiert werden können [36, 46]. Jedoch darf dadurch nicht die korrekte Ausführbarkeit der Instanzen verletzt werden. *Verträglichkeit* mit der geänderten Vorlage ist für eine Instanz gegeben, wenn ihre Ausführungshistorie auch auf der geänderten Vorlage erzeugbar ist [8]. Dies ist z. B. nicht der Fall, wenn die Instanz den von der Änderung betroffenen Bereich (teilweise) schon überschritten hat (s. Abb. 3). Um aufwendige Zugriffe und Analysen der kompletten Ausführungshistorien betroffener Instanzen zu vermeiden, gibt es Vorschläge für optimierte Verträglichkeitsprüfungen [35, 36]. Diese beruhen auf einem Vergleich der durchgeführten Vorlagenänderungen mit den aktuellen Zuständen der betroffenen Knoten. Trotz dieser logischen Optimierungen bleiben Verträglichkeitsprüfungen sehr performanzkritisch, da sie potentiell für sehr viele Instanzen durchgeführt werden müssen.

Beispiel 3 (Anfragen an Kollektionen von Instanzen) Eine häufige Operation auf Prozessdaten sind Anfragen an Kollektionen von Instanzen, etwa um festzustellen, bei welchen Instanzen aktuell ein bestimmter Schritt ausgeführt wird. Solche Anfragen sind insofern kritisch, als Information von potentiell sehr vielen Instanzen benötigt wird. Bei naiver Realisierung führt dies dazu, dass die betroffenen Instanzen alle in den Primärspeicher eingelagert werden müssen. Dies kostet sowohl Rechenzeit als auch Speicherplatz, was Auswirkungen auf andere Laufzeitfunktionen (z. B. Weiterschalten) hat. Beispiele sind Anfragen nach den aktuell ausführbaren Aktivitäten aller Prozesse, nach Aktivitäten, die ein bestimmter Benutzer ausgeführt hat oder ausführen darf, sowie gezielte Abfragen auf Audits (d. h. Logdaten).

2.3 Existierende Ansätze zur internen Speicherrepräsentation

Vorangehend haben wir die logische Sicht auf Prozessdaten sowie ausgewählte funktionale Anforderungen an die Speicherrepräsentation dieser Daten skizziert. In diesem Abschnitt stellen wir existierende Konzepte zur Repräsentation und Interpretation dieser Daten vor, also Ausführungs-

modelle. Das einfachste Ausführungsmodell ist die direkte Interpretation von Graphstrukturen durch das PMS [14, 27]. Dabei stimmt das Ausführungsmodell weitgehend mit dem logischen Modell überein. Hierdurch ist die vom Benutzer modellierte Semantik in derselben Form auch zur Laufzeit vorhanden. Erst dadurch können fortschrittliche Funktionen effizient realisiert werden. Beispielsweise erfordert die Propagation von Änderungen nach Schemaevolution einen Vergleich zwischen den auf dem logischen Modell durchgeführten Vorlagenänderungen mit den im Ausführungsmodell laufenden Instanzen [32, 37]. Je geringer der Unterschied zwischen Ausführungsmodell und logischem Modell ist, um so einfacher und effizienter ist dieser Vergleich durchzuführen.

Eine andere Möglichkeit besteht darin, einzelne Aktivitäten im Ausführungsmodell als endliche Automaten darzustellen [40, 25]. Der übergeordnete Prozess, der die Aktivitäten miteinander verknüpft, wird durch Kommunikation zwischen den autonomen Automaten realisiert. Diese besitzen Vor- und Nachbedingungen, durch die das korrekte Weiterschalten ermöglicht wird. Ein Vorteil dieses Ausführungsmodells ist die dezentrale und verteilte Ausführung der einzelnen Aktivitäten. Zugriff auf globale Statusinformation ist aber nur eingeschränkt möglich und der Prozessablauf von außen schlecht steuerbar. Dadurch sind auch nur solche Änderungen an Prozessen realisierbar, die einzelne Aktivitäten betreffen (z. B. Anpassungen von Bearbeiterzuordnungen oder von Aktivierungs- und Terminierungsregeln).

Eine effiziente Ausführung ist oftmals erzielbar, wenn das logische Modell (z. B. Prozessgraphen) in eine Menge ausführbarer Regeln übersetzt wird. Dies können sowohl interpretierte als auch kompilierte Regeln (z. B. Datenbank-Trigger und Stored Procedures) sein. Erstere haben den Vorteil, dass spätere Änderungen am Prozess leicht durch zusätzliche Regeln ausdrückbar sind. Ein entscheidender Nachteil ist jedoch, dass eine große Regelmenge für Benutzer schwer verständlich und zudem aufwendig durch das PMS auszuwerten ist.

Einige Ansätze (z. B. [16]) bieten eine datenzentrierte Sicht. Dabei wird ein Datenobjekt durch verschiedene Aktivitäten geschleust, wobei die durchzuführenden Operationen (vergleichbar mit OO-Konzepten) direkt vom Datenobjekt bereitgestellt werden. Dadurch sind Aktivitäten und Datenobjekte weitgehend autonom. Problematisch ist die Realisierung paralleler Verzweigungen, wozu unabhängige Kopien der Datenobjekte erstellt und separat manipuliert werden. Das Zusammenfassen der darauf durchgeführten Änderungen kann anschließend oftmals nur mit Benutzereingriff erfolgen.

Für die effiziente Unterstützung von Prozessänderungen ist die Interpretation von Graphstrukturen am besten geeignet. Durch eine geeignete Repräsentation der Datenstrukturen kann der Aufwand bei der Interpretation zur Laufzeit entscheidend verbessert werden. So können einerseits durch eine möglichst kompakte Repräsentation der Speicherbedarf und damit der Aufwand für das Ein- und Auslagern der Datenstrukturen optimiert werden. Andererseits kann durch zusätzliche Metainformationen der Interpretationsaufwand minimiert werden. Wie diese beiden teilweise komplementären Ziele erreicht werden können, wird in den folgenden Kapiteln ausführlich dargestellt.

3 Speicherrepräsentation

Wir behandeln nun die Speicherrepräsentation und damit die Datenstrukturen auf logischer Ebene, bevor diese später auf physische Datenstrukturen abgebildet werden. Startpunkt unserer Betrachtungen bildet die Paritionierung von Prozessgraphen. Weitgehend unabhängig von der späteren Repräsentation auf physischer Ebene wird durch die sinnvolle Zerlegung der Vorlagen- und Instanzdaten die Grundlage für die Minimierung des Aufwands für das Ein-/Auslagern gelegt. Anschließend betrachten wir weitere Aspekte zur Optimierung von Zugriffen durch Metadaten. Die angestellten Überlegungen zur Repräsentation der Speicherstrukturen sind sehr wichtig für die Performanz des PMS, vergleichbar zu entsprechenden Bestrebungen in Datenbanksystemen. Dort hat beispielsweise die Wahl einer Implementierungsvariante von Indexen große Auswirkungen auf den Speicherbedarf und den effizienten Datenzugriff [39]. Die gewählte Realisierung hängt jedoch auch von funktionalen Anforderungen ab, etwa davon, welche Art von Zugriffen unterstützt werden soll.

3.1 Partitionierung von Prozessgraphen

Neben einer kompakten Speicherrepräsentation der Daten schränkt auch die Partitionierung von Prozessgraphen den (Primär-) Speicherbedarf ein. Es wird jedoch noch nicht der absolute Speicherbedarf minimiert sondern lediglich die Datenmenge, die zu einem bestimmten Zeitpunkt und für eine bestimmte Funktion (z. B. Weiterschalten) benötigt wird. Dies hat signifikante Auswirkungen: Zum einen wird das Ein-/Auslagern der Daten beschleunigt, zum anderen können zu einem Zeitpunkt die Daten einer größeren Anzahl von Prozessen im Primärspeicher sein, was wiederum die Häufigkeit des Ein-/Auslagern von Daten verringert.

Wir unterscheiden zwischen *horizontaler* und *vertikaler Partitionierung* (Abb. 4). Die Unterteilung der Knotenmenge (*horizontale Partitionierung*) ermöglicht es, nur die Knoten einzulagern, die für die Ausführung einer Funktion aktuell benötigt werden. Dies zahlt sich bei großen Prozessgraphen aus. *Vertikale Partitionierung* dagegen unterstützt das Einlagern ausgewählter Attribute (z. B. Knotenzustände). So reicht es im Kontext von Schemaevolution aus, wenn das Zustandsattribut der Knoten im Primärspeicher bereitgestellt wird; Informationen zum Bearbeiter werden dagegen nicht benötigt – sofern die Bearbeiterzuordnung nicht geändert worden ist. Neben dem Primärspeicherbedarf verbessert eine angemessene Partitionierung auch das Laufzeitverhalten des PMS, da weniger Daten aus dem Sekundärspeicher ausgelesen und ggf. transformiert werden müssen (Abb. 15). Allerdings wird die Implementierung komplizierter, da bei jedem Sekundärspeicherzugriff jeweils anzugeben ist, welche Daten benötigt werden. Nur so kann für eine Anfrage die geeignte Partitionierung ausgewählt und die Datenmenge minimiert werden. Zusätzlich sollten Daten nachgefordert werden können, die beim erstmaligen Einlagern noch nicht angefordert worden sind, damit die bereits eingelagerten Daten nicht nochmals eingelagert werden.

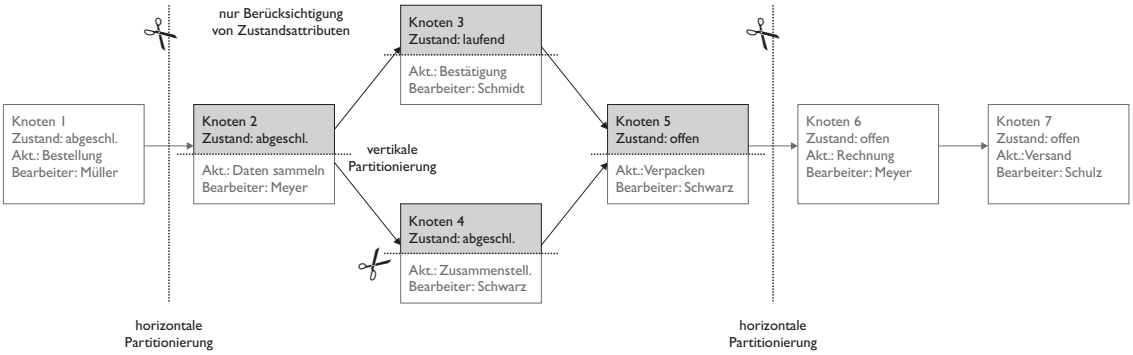

Abbildung 4: Partitionierung von Prozessgraphen

3.2 Prozessvorlagen

Prozessvorlagen stehen mit anderen Entitäten in Beziehung (z. B. Prozesstyp, Super-/Subprozessvorlagen, Instanzen). Da diese Beziehungen relativ selten benötigt werden, reicht es aus, sie ausschließlich im Sekundärspeicher zu hinterlegen. Kritisch dagegen ist die Speicherrepräsentation der Knoten und Kanten einer Vorlage. Aufgrund der zahlreichen Attribute eines Knotens (z. B. auszuführendes Programm, Bearbeiterzuordnung, Ein-/Ausgabedaten) ist es empfehlenswert, sie als eigenständige Entitäten (Objekte) zu repräsentieren. Dies können sowohl Objekte in OO-Programmiersprachen als auch Datenstrukturen strukturierter Programmiersprachen (`STRUCT`/`RECORD`) sein. Die vertikale Partitionierung kann durch leichtgewichtige Proxyobjekte [12] realisiert werden oder indem entsprechende Attributwerte durch Platzhalter (z. B. `NULL`-Werte) ersetzt werden.

Reihenfolgebeziehungen, zeitliche Abhängigkeiten sowie weitere Beziehungen werden über typisierte Kanten (Kontrollkanten, Schleifenkanten, Zeitkanten, usw.) realisiert. Abhängig vom Typ kann eine Kante unterschiedliche Attribute besitzen. Dies können neben der Typinformation der

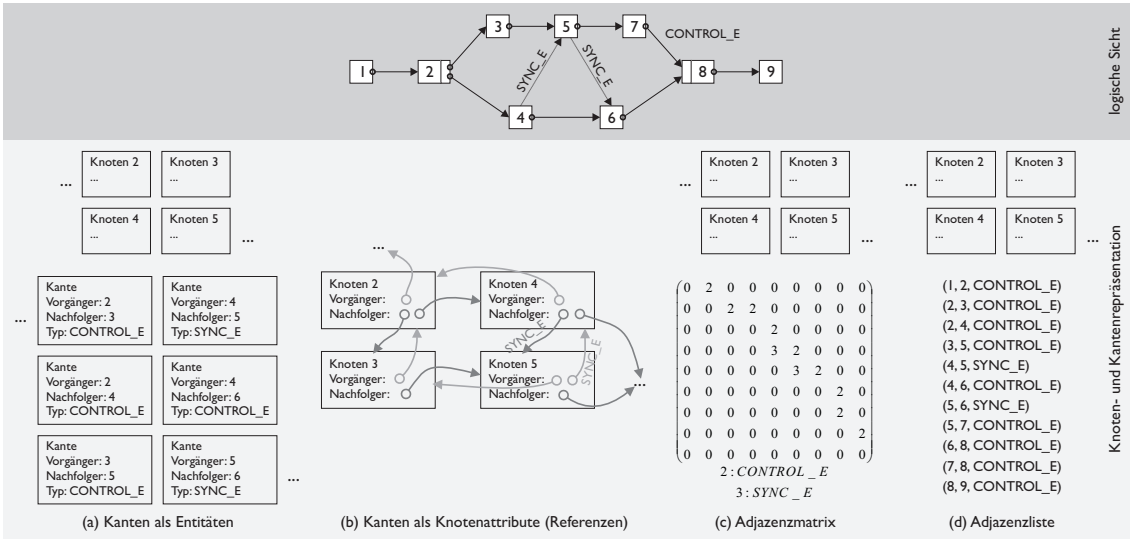

Abbildung 5: Prozessvorlage und entsprechende Kantenrepräsentationen

Quell- und Zielknoten, auch Auswahlprädikate bei alternativen Verzweigungen (sog. Transitionsbedingungen) oder Minimal-/Maximalabstände bei Zeitkanten sein. Insgesamt besitzen Kanten nur wenige (ca. 5) Attribute, weshalb sich mehrere konzeptionelle Realisierungsmöglichkeiten anbieten (Abb. 5):

- Explizite Kanten (Kanten als vollwertige Entitäten)
- Implizite Kanten (Knoten besitzen Attribute, deren Werte Referenzen auf Nachfolger-/ Vorgängerknoten darstellen)
- Repräsentiert als Adjazenzmatrix
- Repräsentiert als Adjazenzliste.

Variante KR-1 (Kanten als Entitäten) Trotz weniger Attribute können Kanten als eigene Objekte oder Strukturen realisiert werden (vgl. Abb. 5 a). Dies ermöglicht eine flexible Handhabung, etwa indem alle Kantenobjekte in einer Menge – vergleichbar einer Adjazenzliste (Variante KR-4) – verwaltet werden. Sie können auch direkt als Objektreferenzen bei den jeweiligen Knoten (vgl. Variante KR-2) abgelegt sein. Insgesamt erfordert diese Variante hohen Speicherbedarf für die physische Repräsentation im Primärspeicher. So beträgt der initiale Speicherbedarf pro Objekt (der Objektheader) in Java 8 Byte. Damit lohnt sich eine Repräsentation als eigenständige Entität nur, wenn die Kante viele Attribute oder Attribute „großer“ Datentypen besitzt. Beispielsweise belegt der Objektheader bei 32 Attributen eines kleinen Datentyps oder bei 4 Attributen des größten Datentyps immer noch 20% des gesamten Speicherbedarfs des Objekts.

Variante KR-2 (Kanten als Referenzen) In Programmiersprachen können Kanten direkt als Referenzen realisiert werden, die dann z. B. bei den Quellknoten hinterlegt werden (vgl. Abb. 5 b). Besser sind doppelte Referenzen, da Kanten sowohl vom Quell- zum Zielknoten als auch umgekehrt verfolgt werden können. Problematisch dabei ist, dass Kantentypen schwer zu realisieren sind. Eine Möglichkeit ist, die von einem Knoten ausgehenden Kanten entsprechend ihrer Richtung und ihres Typs jeweils in einer eigenen Menge zu hinterlegen. Dies erfordert aber viel Speicher, da dann bei jedem Knoten für jeden Kantentyp und für jede Richtung (Vorgänger und Nachfolger) eine Kantenmenge benötigt wird. Ein noch größeres Problem ist die Realisierung von Kantenattributen, da jeder Kantentyp verschiedene Attribute besitzen kann.

Bei zahlreichen Funktionen ist es nötig, die Vorgänger/Nachfolger eines Knotens (transitiv) zu bestimmen. Hierzu wird so lange über die Kanten iteriert, bis alle Kanten vom entsprechenden Knoten bis zum Start-/Endknoten der Vorlage festgestellt worden sind. Sind Kanten als Knotenattribute realisiert, müssen dazu die entsprechenden Knotenobjekte eingelagert werden. Liegen Knoten und Kanten getrennt vor, kann hierauf verzichtet werden. Durch Kopplung von Knoten und Kanten vereinfacht sich die horizontale Partitionierung, die direkt durch Aufteilung der Knoten gegeben ist.

Variante KR-3 (Adjazenzmatrix) Hier existiert für jeden Knoten einer Vorlage jeweils eine Zeile und eine Spalte, d. h. jeder Matrixeintrag steht für eine Relation zwischen zwei Knoten [9]. Über eine geeignete Codierung dieser Einträge können verschiedene Relations- und damit verschiedene Kantentypen hinterlegt werden (vgl. Abb. 5 c). Ein Vorteil ist, dass alle Kanten ohne Aufwand sowohl vorwärts (ausgehend von Zeilen) als auch rückwärts (ausgehend von Spalten) verfolgt werden können, Quell- und Zielknoten einer Kante sind implizit durch die jeweilige Zeile und Spalte gegeben. Im Gegensatz zu Variante KR-2 kann dadurch ohne Knotenzugriffe über Kanten iteriert werden. Jedoch stellt sich wieder das Problem der Realisierung von Kantenattributen. Nachteilig ist zudem der Speicherbedarf, der quadratisch von der Anzahl der Knoten in der Prozessvorlage abhängt (Abb. 6).

Variante KR-4 (Adjazenzliste) Jeder Eintrag einer Adjazenzliste steht für eine Kante zwischen zwei Knoten. Im Gegensatz zur Adjazenzmatrix sind Quell-/Zielknoten explizit (vgl. Abb. 5 d), und es werden nur tatsächlich vorhandene Kanten berücksichtigt [9]. Dies hat insbesondere positive Auswirkungen auf den benötigten Speicherplatz (siehe Abb. 6). Zudem können beliebige Attribute hinterlegt und sogar nachträglich ergänzt werden. Auch bei Iterationen sind keine Zugriffe auf Knoten nötig. Allerdings erfordert der Zugriff bei mehreren ausgehenden Kanten eines Knotens und insbesondere bei gerichteten Kanten in der Rückrichtung ein ineffizientes Durchsuchen der Adjazenzliste. Aufgrund des fehlenden wahlfreien Direktzugriffs ist die horizontale Partitionierung problematisch.

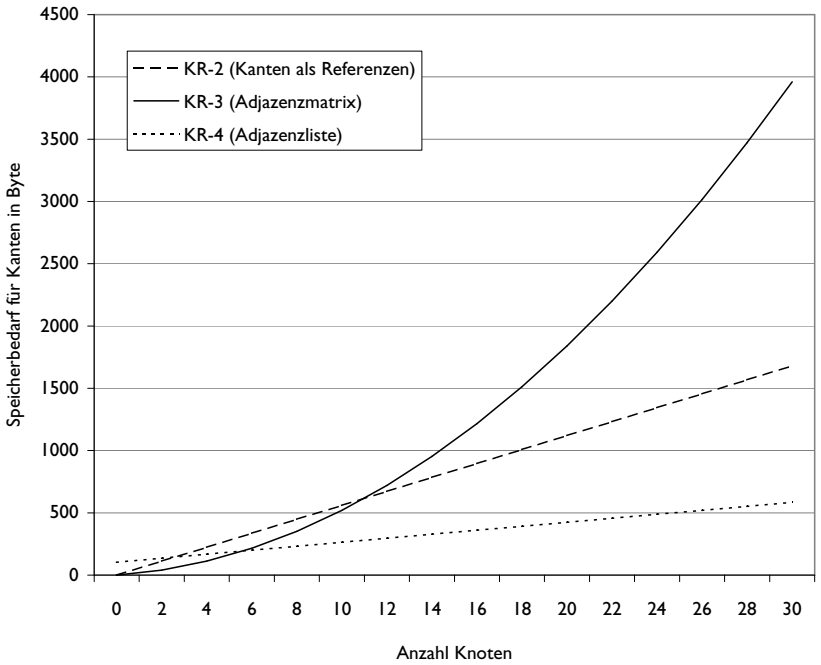

Abbildung 6: Primärspeicherbedarf für verschiedene Kantenrealisierungen

Wichtigstes Kriterium für den Vergleich dieser Varianten ist der Speicherbedarf. Die Anzahl der

Kanten in einer Vorlage entspricht ungefähr der Anzahl der Knoten. Realisiert man Kanten als eigene Entitäten (KR-1), ergibt sich dadurch für jede Vorlage nahezu eine Verdopplung des Speicherbedarfs. Den Speicherbedarf für die anderen Varianten (KR-2, KR-3 und KR-4), abhängig von der Knotenanzahl, zeigt Abb. 6. Auffällig ist der quadratische Speicherbedarf für die Realisierung mittels Adjazenzmatrix. Ab 10 Knoten übersteigt dieser die anderen beiden Varianten. Diese besitzen einen linearen Speicherbedarf, wobei die Adjazenzliste initial mehr Speicher benötigt als Variante KR-2. Dies liegt an der verwendeten Datenstruktur für die Realisierung der Liste. Dieser lohnt sich bereits ab 3 Knoten. Bei einer üblichen Vorlagengröße von 15-30 Knoten ist die Adjazenzliste hinsichtlich Speicherbedarf am besten.

Für eine endgültige Bewertung reicht der Vergleich des Primärspeicherbedarfs alleine aber nicht aus. Ebenfalls wichtig sind der wahlfreie Zugriff auf Kanten und die Möglichkeit der horizontalen Partitionierung. Die vertikale Partitionierung ist aufgrund der geringen Attributzahl bei Kanten unbedeutend. Dagegen ist die Realisierung der Attribute wichtig. So sollten sich Attribute später leicht ändern und ergänzen lassen, z. B. zwecks Einführung neuer Kantentypen bei Erweiterungen des Metamodells. Ein weiteres Kriterium ist die Unterstützung für Iterationen direkt über Kanten, ohne dass dabei Knotenobjekte gelesen werden müssen. Dies ist beispielsweise für die Bestimmung von Vor- und Nachfolgerbeziehungen zwischen Knoten nützlich, was sehr häufig benötigt wird. Tabelle 1 bewertet die Varianten KR-2, KR-3 und KR-4 hinsichtlich dieser Aspekte.

Tabelle 1: Vergleich verschiedener Kantenrepräsentationen

	KR-2 Kanten als Referenzen	KR-3 Adjazenzmatrix	KR-4 Adjazenzliste
Speicherbedarf	+	--	++
Realisierung von Attributen	--	-	++
wahlfreier Zugriff	+	++	-
Partitionierung	++	+	-

Aufgrund der sehr guten Realisierung von Attributen sind Adjazenzlisten (KR-4) die bevorzugte Realisierungsvariante – zumindest für Prozessmodelle mit expliziten Schleifenkanten. In diesem Fall ist eine topologische Sortierung der Knoten und damit auch der Kanten möglich. Diese erlaubt es, die Nachteile bezüglich des wahlfreien Zugriffs und der Partitionierung abzumildern. So kann bei jedem Knoten der Index der ersten ausgehende Kante des Knotens in der Adjazenzliste hinterlegt werden. Weitere von diesem Knoten ausgehende Kanten befinden sich direkt hinter dieser Indexposition in der Liste. Für einen effizienten Zugriff für die Vorgängerbeziehung, d. h. die Rückrichtung, werden allerdings eine zusätzliche Adjazenzliste sowie die entsprechenden Indizes bei den Knoten benötigt. Neben dem effizienten Zugriff auf Kanten ermöglichen diese Indizes darüberhinaus die horizontale Partitionierung einer Adjazenzliste. Hierbei wird ausgenutzt, dass die Indizes direkt bei den Knoten abgelegt sind. So geben die Indizes des ersten und des letzten Knotens der Partition den Teil der Adjazenzliste an, der Bestandteil der Partition ist. Die Adjazenzlistenteile vor und nach diesen Indizes werden für diese Partition nicht benötigt.

3.3 Prozessinstanzen

Bisher wurden nur statische Prozessdaten betrachtet. Ein größeres Datenvolumen fällt infolge der großen Anzahl von Prozessinstanzen an (bis zu mehrere Zehntausend). Deshalb ist deren Speicherrepräsentation performanzkritischer als diejenige von Prozessvorlagen. Eine Instanz besteht (logisch) aus der Struktur der zugrundeliegenden Vorlage (sog. *Instanzvorlage*), angereichert um Zustandsinformation (Abb. 2). Beides sind voneinander unabhängige Aspekte.

3.3.1 Instanzvorlagen

Die Repräsentation von Prozessvorlagen wurde in Abschnitt 3.2 untersucht. Diese Überlegungen sind auf Instanzvorlagen übertragbar. Zusätzlich stellt sich die Frage, ob für jede Instanz ein

kompletter Graph gespeichert wird (*Vorlagenkopie*) oder nicht (*Vorlagenreferenz*).

Variante VG-1 (Vorlagenkopie) Bei Vorlagenkopien wird bei der Erzeugung einer Prozessinstanz eine komplette Kopie der entsprechenden Prozessvorlage erstellt und um Attribute zur Aufnahme dynamischer Information ergänzt (z. B. Knotenzustände, Abb. 7 a). Damit entspricht die Speicherrepräsentation einer Instanz unmittelbar der konzeptionellen Sicht. Beim Weiterschalten können die Nachfolger eines beendeten Knotens direkt im Instanzobjekt bestimmt werden. Vorlagenkopien sind problematisch bei Schemaevolution, da jede Instanzvorlage zunächst einmal unabhängig von der ursprünglichen Vorlage ist. Wird letztere geändert, müssen zusätzlich zu den ohnehin aufwendigen Verträglichkeitsprüfungen und Zustandsanpassungen für jede Instanz auch strukturelle Änderungen an der Instanzvorlage vorgenommen werden. Diese sind bei jeder von der Änderung betroffenen (unveränderten) Instanz gleich.

Variante VG-2 (Vorlagenreferenz) Die Instanzierung mit Vorlagenreferenzen legt lediglich die Datenstrukturen für dynamische Daten (z. B. eine Liste aller Knotenzustände) sowie eine Referenz auf die instanzierte Vorlage (Abb. 7 b) an.

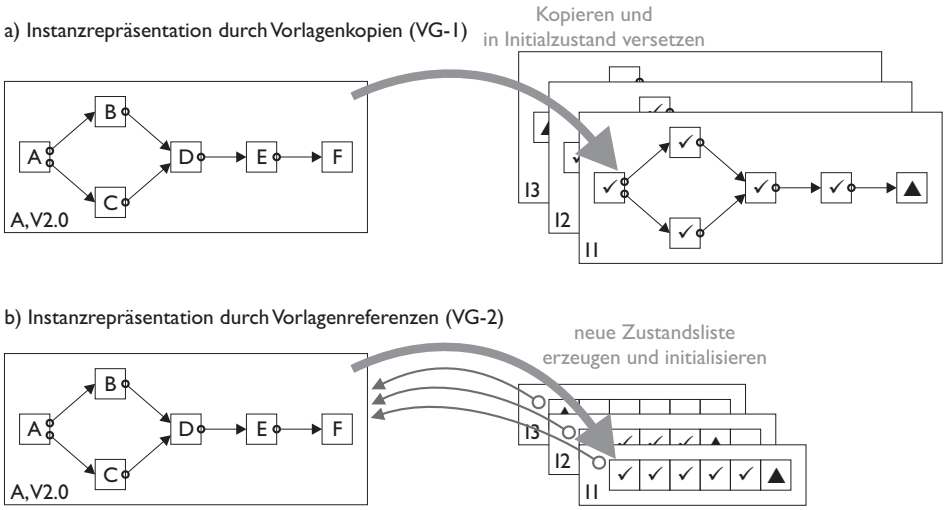

Abbildung 7: Erzeugung neuer Instanzobjekte

Bei Variante VG-2 existiert keine Kopie der Vorlagendaten. Die Bestimmung von Nachfolger-/Vorgängerknoten (z. B. beim Weiterschalten im Prozess) kann in diesem Fall nicht direkt im Instanzobjekt erfolgen sondern nur indirekt über die zugrundeliegende Vorlage, was einige wenige zusätzliche Speicherzugriffe erfordert. Mit Vorlagenreferenzen ist die Propagation von Änderungen bei Schemaevolution wesentlich effizienter durchführbar. So reicht es aus, die strukturellen Änderungen lediglich einmal auf der Prozessvorlage durchzuführen und anschließend die entsprechenden Instanzen auf Verträglichkeit mit der neuen Prozessvorlage zu überprüfen [36, 35]. Bei verträglichen Instanzen müssen dann nur die Vorlagenreferenz geändert und die Knotenzustände angepasst werden. Im Gegensatz zu Variante VG-1 sind keine strukturellen Änderungen für jede Instanz nötig.

Vergleicht man die beiden Varianten (Tabelle 2), so ist VG-2 vorzuziehen. Vorlagenkopien (VG-1) sind aufwendiger zu erzeugen, und die Strukturinformation ist redundant bei jeder Instanz abgelegt, was zu großem Speicherbedarf führt. Variante VG-2 dagegen benötigt weniger Speicher bei minimaler Verschlechterung ($O(1)$) des Zugriffsverhaltens. Daneben lassen sich Änderungen bei Schemaevolution sehr viel effizienter durchführen als bei Variante VG-1. Auch der Aufwand für das Ein-/Auslagern der Instanzen ist deutlich geringer, da Instanzobjekte wesentlich kleiner sind. Vorlagenkopien werden z. B. in den PMS Chameleon [22] und ProMInanD [16, 15]verwendet,

Tabelle 2: Vergleich verschiedener Vorlagengraphen

	VG-1 Vorlagenkopien	VG-2 Vorlagen-referenz
Speicherbedarf	-	++
Schemaevolution	--	++
Zugriff	+	O
Partitionierung	+	++

Vorlagenreferenzen bei Staffware. In der Regel sind nur wenige Prozessvorlagen im System gleichzeitig instanziert (Tabelle 6). Somit können diese dauerhaft im Primärspeicher gehalten werden und müssen nicht ein-/ausgelagert werden, was sich bei Variante VG-2 vorteilhaft auswirkt.

3.3.2 Instanzzustände

Neben struktureller Information besitzt eine Instanz einen definierten Zustand. Abb. 8 zeigt verschiedene Realisierungsmöglichkeiten. Sie beruhen auf der bisherigen Ausführung der Instanz, die in einer (Ausführungs-) Historie protokolliert wird. Diese kann auch zur Rekonstruktion des aktuellen Zustands herangezogen werden.

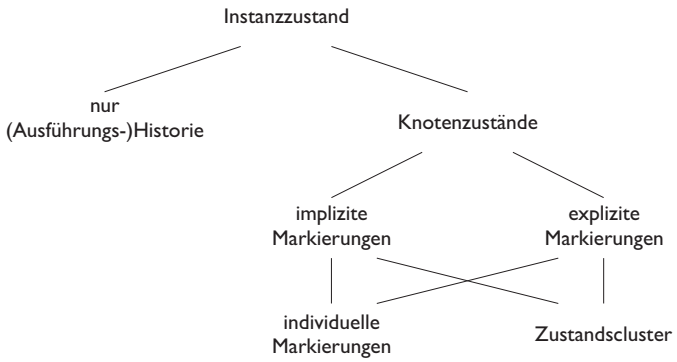

Abbildung 8: Klassifikation der Realisierungsmöglichkeiten von Zuständen

Der aktuelle Zustand eines Prozesses spiegelt sich in den Zuständen der einzelnen Aktivitäten wider. Ihre Repräsentation kann mittels expliziter Markierungen erfolgen, wobei jeder Knoten eine eigene Markierung zugewiesen bekommt [36]. Bei impliziten Markierungen dagegen werden nur die aktiven Knoten verwaltet, also zur Bearbeitung anstehende oder in Bearbeitung befindliche Aktivitäten. Letzteres entspricht der Tokensemantik von Petrinetzen [7, 4]. Die Zustände nichtaktiver Knoten können hieraus jedoch nur teilweise rekonstruiert werden.

Für die Repräsentation der Markierungen bei Knoten gibt es zwei Möglichkeiten: Sie können entweder für jede Instanz individuell verwaltet werden, oder es werden Zustandscluster gebildet. Bei Zustandsclustern werden Instanzen im selben Zustand zusammengefasst und der entsprechende Zustand nur einmal für alle Instanzen realisiert. Im folgenden werden zunächst drei alternative Repräsentationsmöglichkeiten für individuelle Instanzmarkierungen untersucht und verglichen. Auf Zustandscluster wird in Kapitel 4 eingegangen.

Variante ZR-1 (Repräsentation von Zuständen durch Historien, vgl. Abb. 9 a) Um die Ausführung eines Prozesses jederzeit nachvollziehen zu können, können sämtliche Zustandsübergänge inklusive weiterer Informationen wie die Zeit des Zustandsübergangs in einer (Ausführungs-) Historie protokolliert werden. Aus dieser lassen sich aktuelle sowie frühere Instanzzustände (und damit auch die Knotenzustände) rekonstruieren. Dieser Ansatz ist für das Weiterschalten ausreichend. Beim Beenden einer Aktivität müssen, außer dem ohnehin notwendigen Eintrag in der

Historie, keine weiteren Daten manipuliert werden. Die Markierung nachfolgender Aktivitäten ist jedoch aufwendig. Hierzu werden für alle Nachfolgerknoten die Zustände aller Vorgänger ermittelt, indem der letzte Eintrag des entsprechenden Knotens in der Historie bestimmt wird. Dieser gibt den aktuellen Knotenzustand an. Die Zustandsrekonstruktion für Verträglichkeitsprüfungen bei Schemaevolution erfolgt analog. Sie ist für jeden zu überprüfenden Knoten jeder betroffenen Instanz erforderlich. Sind anschließend Zustandsanpassungen nötig, müssen diese aus Konsistenzgründen auch in der Historie vermerkt werden. Insgesamt ist Schemaevolution mit Variante ZR-1 damit sehr aufwendig.

(Ausführungs-) Historien benötigen potentiell sehr viel Speicherplatz. Einerseits ist jeder Historieneintrag relativ groß, da er neben dem Zustandsübergang und dem betroffenen Knoten normalerweise u. a. noch einen Zeitstempel sowie den für den Zustandsübergang verantwortlichen Bearbeiter enthält. Andererseits gibt es für jeden Knoten mehrere Einträge in der Historie (schlimmstenfalls für jeden Zustandsübergang einen), wobei die Anzahl von der im Prozessmetamodell vorhandenen Zustände/Zustandsübergänge abhängt. Beim Auftreten von Schleifen im Prozess kommen noch weitere Einträge für die Knoten innerhalb der Schleife hinzu (Abb. 9 a).

Aufgrund der großen Datenmengen sind Historien nur in Verbindung mit horizontaler und vertikaler Partitionierung sinnvoll. Bei horizontaler Partitionierung kommt allerdings erschwerend hinzu, dass (Ausführungs-) Historien chronologisch sortiert sind. Dadurch kann es bei Schritten in parallelen Zweigen zu Verschränkungen der Historieneinträge kommen, d. h. nicht alle Einträge eines Knotens stehen hintereinander. Eine horizontale Partitionierung erfordert aber, dass die Daten verschiedener Knoten getrennt werden können.

Die Verschränkung von Historieneinträgen verhindert auch den effizienten Zugriff auf Historiendaten. Zwar lassen sich die Zustände der wichtigsten Knoten, nämlich die der aktiven Knoten, relativ schnell rekonstruieren, da diese am Ende einer chronologisch sortieren Historie stehen. Allerdings lassen sich andere Knotenzustände nur durch zusätzliche Indexstrukturen, vergleichbar denen bei Adjazenzlisten (siehe Abschnitt 3.2), effizient bestimmen. Sind die Historieneinträge zusätzlich verschränkt, kann der Knotenzustand nur durch sequentielle Iteration der Historieneinträge nach (oder vor, je nach Realisierung des Index) dem referenzierten Historieneintrag erfolgen.

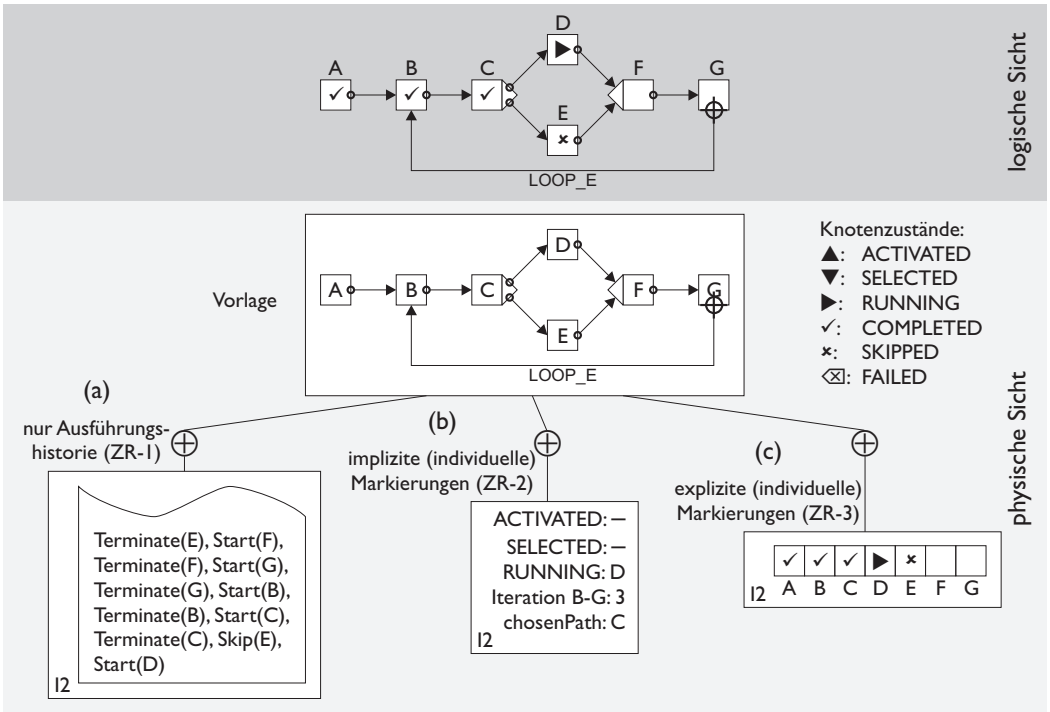

Abbildung 9: Realisierungsmöglichkeiten von Knotenzuständen

Variante ZR-2 (Implizite Markierungen, vgl. Abb. 9 b) Bei impliziten Markierungen besitzen nur aktive Knoten eine Markierung, also alle aktuell ausgeführten oder zur Ausführung anstehenden Aktivitäten. Die Rekonstruktion der Zustände anderer Knoten erfolgt über die Vorgänger-/Nachfolgerbeziehung zu einem aktiven Knoten. Da es keine isolierten Knoten in einer Vorlage gibt, existiert diese Beziehung immer. Vorgängerknoten sind abgeschlossen oder wurden abgewählt, Nachfolgerknoten stehen zukünftig noch zur Ausführung an. Nicht möglich ist die Zustandsrekonstruktion bei Metamodellen mit impliziten Schleifen (z. B. Petrinetze). Bei diesen Modellen gibt es keinen speziellen Kantentyp für Schleifen. Stattdessen werden Schleifen einfach durch das entsprechende Setzen von normalen (Vorwärts-) Kanten realisiert. Hierdurch lassen sich jedoch keine eindeutigen Vorgänger-/Nachfolgerbeziehungen bestimmen: Ein Knoten innerhalb eines Kantenzyklus ist sowohl Nachfolger als auch Vorgänger von sich selbst sowie von allen anderen Knoten des Zyklus. Eine eindeutige Vorgänger-/Nachfolgerbeziehung ist für fortschrittliche Funktionalität (Schemaevolution, instanzspezifische Änderungen) aber unabdingbar. Bei expliziten Schleifen wird zur Zustandsrekonstruktion ein Iterationszähler mit verwaltet. Zudem werden die bei alternativen Verzweigungen gewählten Pfade gespeichert (Abb. 9 b). Eine detaillierte Beschreibung des Rekonstruktions-Algorithmus findet sich in Anhang A.

Der Laufzeitaufwand der Zustandsrekonstruktion bei impliziten Markierungen ist linear zur Anzahl der Instanzknoten, sofern keine Historienzugriffe erfolgen. Für einfache Funktionen ist dies ausreichend, jedoch nicht für die Verträglichkeitsprüfung bei Schemaevolution. Hier ist der Algorithmus für jeden von der Änderung betroffenen Knoten jeder Instanz einmal auszuführen, was die Laufzeit $O(a * n * m)$ ergibt. Diese hängt von der Anzahl der aktiven Knoten (a), der Anzahl von der Änderung betroffener Knoten (n) sowie der Anzahl zu untersuchender Instanzen (m) ab. Dies ist sehr problematisch, da von einer Schemaevolution potentiell sehr viele Instanzen betroffen sind.

Variante ZR-2 benötigt insgesamt wenig Speicher. Dies beschränkt sich auf die Zustände aktiver Knoten (insgesamt ca. 3-4 Byte pro Instanz), gewählte Pfade bei alternativen Verzweigungen (ca. 3 Byte pro Instanz) sowie einen Zähler pro Schleife (ca. 2 Byte pro Instanz). Implizite Zustände bieten trotzdem nur eine geringfügig schlechtere Unterstützung für das Weiterschalten im Vergleich zu Variante ZR-1. Beim Beenden von Aktivitäten müssen diese lediglich aus der Liste der aktiven Instanzknoten entfernt und ggf. die nach dem Weiterschalten neu aktivierten Knoten hinzugefügt werden. Beim Weiterschalten ist ähnlich wie bei ZR-1 die Rekonstruktion von Zuständen nötig, was jedoch weniger Speicher benötigt. Für Schemaevolution ist ZR-2 schlecht geeignet. Der Laufzeitaufwand der Zustandsrekonstruktion wird aufgrund der Abhängigkeit sowohl von der Knoten- als auch der Instanzanzahl sehr groß.

Variante ZR-3 (Explizite Markierungen, vgl. Abb. 9 c) Jede Aktivität – auch wenn sie bereits abgeschlossen, abgewählt oder noch nicht ausgeführt worden ist – erhält eine explizite Markierung, wodurch der Rekonstruktionsaufwand für Knotenzustände wegfällt. Auch die Entscheidungen bei alternativen Verzweigungen und Schleifen brauchen nicht verwaltet zu werden, da sich diese aus den expliziten Markierungen ergeben. Bei expliziten Schleifen werden bei Variante ZR-3 die Knotenzustände des Schleifenkörpers vor erneutem Schleifendurchlauf zurückgesetzt. Die Markierungen spiegeln damit nur den jeweils aktuellen bzw. letzten Schleifendurchlauf wider. Vorherige Iterationen können aus der Historie rekonstruiert werden, was aber nur sehr selten nötig ist.

Da die Menge der möglichen Knotenzustände bei allen Prozess-Metamodellen begrenzt ist (z. B. 8 mögliche Zustände pro Knoten in ADEPT [26]), sind wenige Bit pro Knoten für die Speicherung des Zustands ausreichend (ca. 20-40 Byte pro Instanz). Dies ist mehr als bei Variante ZR-2, jedoch bedeutend weniger als eine vollständige Historie benötigt. Variante ZR-3 ist für das Weiterschalten gut geeignet. Im Gegensatz zu den beiden anderen Varianten müssen zwar mehr Markierungen geändert werden, jedoch ist die Bestimmung von Knotenzuständen sehr einfach, da sie nicht rekonstruiert werden müssen. Dasselbe gilt für Schemaevolution. Verträglichkeitsprüfungen sind sehr effizient ausführbar, die Zustandsanpassungen erfolgen direkt bei den entsprechenden Knoten. Aufgrund der Änderung des expliziten Knotenzustands kann auch ermittelt werden, wel-

che Auswirkungen dies auf die Historie hat (z. B. Hinzufügen/Löschen eines Eintrags), ohne dass auf diese lesend zugegriffen werden muss. Variante ZR-3 benötigt zudem wesentlich weniger Primärspeicher als ZR-1, was ein schnelles Ein-/Auslagern aller Zustandsdaten ermöglicht.

Tabelle 3: Vergleich verschiedener Zustandsrepräsentationen

	ZR-1 Ausführungs-historie	ZR-2 Implizite Markierungen	ZR-3 Explizite Markierungen
Speicherbedarf	-	++	+
Zugriff/Rekonstruktion	-	O	++
Partitionierung	O	O	++

Tabelle 3 fasst die Bewertung der Zustandsrepräsentationen zusammen.

3.4 Weitere Aspekte

Bei der Repräsentation von Vorlagen- und Instanzdaten können noch weitere Aspekte berücksichtigt werden, die maßgeblichen Einfluss auf die Performanz besitzen. Eine starke Verbesserung der Zugriffszeiten lässt sich durch Optimierung der häufig benötigten Funktion zur Bestimmung, ob zwei Knoten in Vorgänger-/Nachfolgerbeziehung zueinander stehen, erreichen. Im Normalfall erfordert dies ein Durchlaufen des Graphen mit einem Aufwand von $O(n)$. Durch eine topologische Sortierung der Knoten genügt ein einfacher Vergleich mit Aufwand $O(1)$. Diese kann wie bereits erwähnt (siehe Abschnitt 3.2) bei Prozessmodellen mit expliziten Schleifenkanten erzeugt werden. Hierzu werden Schleifenkanten bei der Bestimmung der Sortierung einfach ignoriert. Da die topologische Sortierung nur pro Vorlage und nicht pro Instanz zu speichern ist, wird nur wenig zusätzlicher Speicherplatz benötigt. Bei den meisten Prozess-Metamodellen ist eine topologische Sortierung auch möglich, da sie auf Schleifen ganz verzichten oder explizite Schleifenkonstrukte besitzen.

Im Gegensatz dazu sind Blockstrukturen nur in wenigen Prozess-Metamodellen vorhanden (z. B. BPEL, ADEPT). Durch Bereitstellung der Blockstrukturen als zusätzliche Metainformation, sind weitere Optimierungen möglich. Dadurch können nicht benötigte Blöcke (also ganze Knotenmengen) als abstrakte Knoten aggregiert werden, was beispielsweise das Durchlaufen eines Graphen beschleunigt. Auch für die Realisierung der Blockstrukturen gibt es mehrere Möglichkeiten:

Variante BS-1 (Blockstrukturen als Objekte) Intern kann eine Prozessvorlage eines blockbasierten Metamodells durch eine Hierarchie von Blöcken realisiert werden (Abb. 10). Dabei sind Blöcke mit Subprozessen vergleichbar. Die Speicherrepräsentation unterscheidet sich signifikant von Metamodellen ohne Blockstrukturen. In dieser reinen Form sind Blöcke aber nicht realisierbar, da selbst blockbasierte Metamodelle oft noch Strukturen außerhalb der regelmäßigen Blockstruktur erlauben (z. B. Links in BPEL [1], Synchronisationskanten in ADEPT [27] – Abb. 10). Zudem benötigt die Strukturinformation für die Blöcke sehr viel Speicher gegenüber einer flachen Graphstruktur.

Variante BS-2 (Blockstrukturen als Relation): Eine einfache Möglichkeit, Blöcke mit einer flacher Graphrepräsentation zu realisieren, ist die Verwendung eines speziellen Kantentyps. Kanten dieses Typs bestehen zwischen Blockanfangs-/Blockendknoten (Abb. 11). Hierdurch sind ähnlich weitreichende Optimierungen wie bei Variante BS-1 möglich, beispielsweise das Überspringen ganze Blöcke. Trotzdem bleibt die interne Datenrepräsentation weiterhin einfach und einheitlich.

Aufgrund der relationalen Natur von Blockstrukturen, kann man ihre Repräsentation bezüglich derselben Aspekte vergleichen wie die Repräsentation von Kanten (Tabelle 4). Auch hierbei

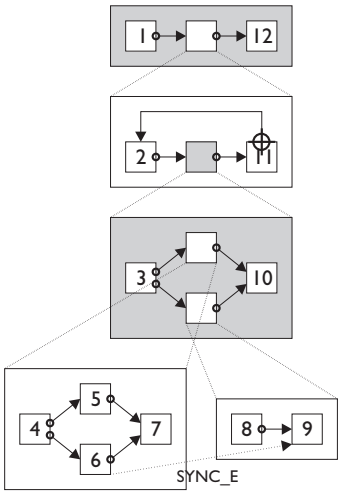

Abbildung 10: Realisierung von Blockstrukturen als Objekte

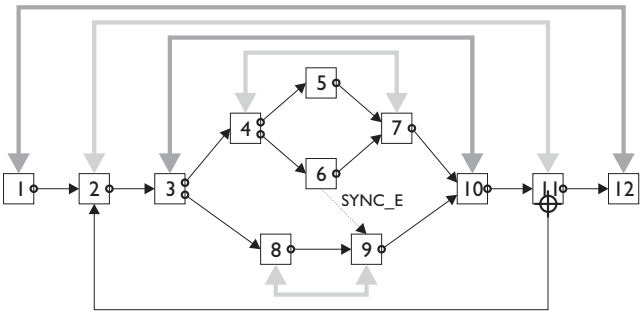

Abbildung 11: Topologische Sortierung und Realisierung von Blockstrukturen mittels Kanten

Tabelle 4: Vergleich verschiedener Repräsentationen für Blockstrukturen

	BS-1 Blöcke als Objekte	BS-2 Blöcke als Relationen
Speicherbedarf	-	+
Realisierung von Attributen	O	O
Zugriff	-	+
Partitionierung	O	+

benötigt die Realisierung mittels expliziter Objekte sehr viel Speicherbedarf gegenüber einer relationalen Speicherung. Dies liegt an den zusätzlichen (weitgehend leeren) Knoten, die in den entsprechenden Elternblöcken benötigt werden. Diese sind auch der Grund dafür, dass der Zugriff bei Variante BS-1 nicht so effizient möglich ist wie bei Variante BS-2: Kindblöcke referenzieren den entsprechenden Knoten im Elternblock und dieser den entsprechenden Kindblock. Beim Durchlaufen des Graphen über mehrere Blöcke hinweg muss deshalb bei jeder Blockgrenze eine Referenz aufgelöst werden. Dies ist bei Variante BS-2 nicht notwendig. Hier können die Blockrelationen sogar komplett ignoriert werden, wenn sie nicht benötigt werden.

Ähnliches gilt auch für die Auswirkungen auf die Paritionierung bei vorhandenen realisierten Blockstrukturen. Während die relationale Repräsentation sich bzgl. Partitionierung gleich verhält wie die die Repräsentation von Kanten und somit keine separate Behandlung benötigt, muss die Objektrepräsentation bei der Partitionierung explizit berücksichtigt werden, da sie selbst bereits eine (horizontale) Partitionierung des Graphen darstellt.

Sollen Blockstrukturen mit zusätzlichen Attributen versehen werden, etwa der Blocktyp oder Zeitinformationen wie die maximale Dauer des Blocks, so ist dies bei beiden Varianten in vergleichbarer Weise möglich: Sowohl Blockobjekte als auch Blockkanten können mit zusätzlichen Attributen versehen werden. Wobei dies im Detail von der konkreten Umsetzung als Objekt bzw. Relation/Kante abhängt.

4 Clusterung von Instanzen

Bisher haben wir individuelle Instanzmarkierungen als Realisierungsmöglichkeit von Knotenzuständen betrachtet. Alternativ dazu können Zustandscluster verwendet werden. Dabei werden Knotenzustände nicht individuell für jede Instanz gespeichert, sondern Instanzen einer Vorlage aufgrund ihres aktuellen Zustands gruppiert. Für jede Gruppe („Zustandscluster“) werden nur einmal die entsprechenden Knotenzustände verwaltet. Dadurch kann einerseits Speicherplatz für Markierungen eingespart werden, andererseits sind Optimierungen (insbesondere für Schemaevolution) möglich.

4.1 Konzept der Clusterung

Jede Instanz einer Prozessvorlage kann sich nur in endlich vielen Zuständen befinden. Die Zustandsmenge wird durch die Anzahl der Knoten der Vorlage sowie den definierten Knotenzuständen beschränkt. Eine weitere Einschränkung ergibt sich durch die Semantik des Prozessmodells. So ist es ein ungültiger Zustand, wenn der Nachfolger einer laufenden Aktivität bereits als beendet markiert ist. Der sich daraus ergebende Zustandsraum einer Vorlage kann als Graph dargestellt werden, der alle erreichbaren Zustände als Knoten sowie die möglichen Zustandsübergänge dazwischen als Kanten repräsentiert (Abb. 12). Statt individuellen Markierungen für jede Instanz werden nun Instanzen aufgrund ihres aktuellen Zustands dem entsprechenden Knoten im Zustandsgraph zugeordnet. Dies spart dann Speicherplatz, wenn von der entsprechenden Prozessvorlage mehr Instanzen gleichzeitig existieren, als Instanzzustände möglich sind. Dann befinden sich mehrere Instanzen im selben Zustand, der aber nur einmal gespeichert wird. Die konkrete Repräsentation für Zustandsknoten ist wie bei Vorlagenkopien orthogonal zu anderen Aspekten und kann deshalb

mittels expliziter oder impliziter Zustände erfolgen (vgl. Abschnitt 3.3.2). Im folgenden gehen wir von einer Realisierung mittels expliziten Markierungen aus.

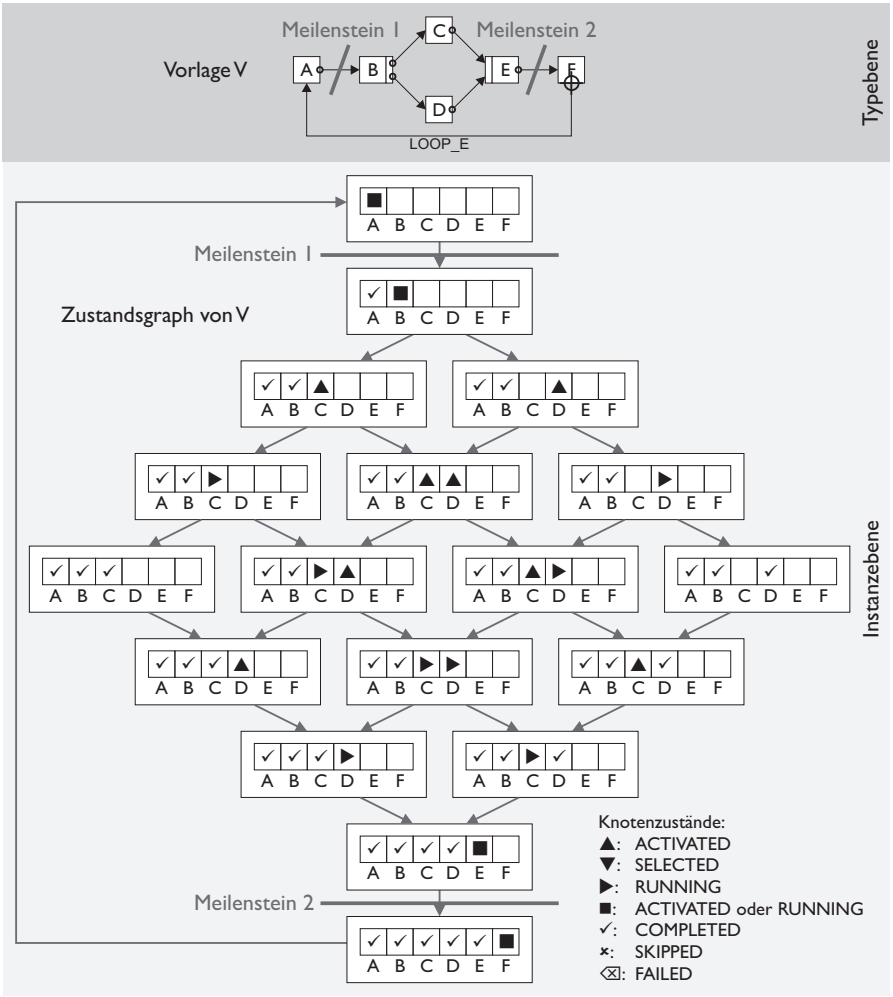

Abbildung 12: Zustandsgraph für Vorlage V (optimiert)

Abb. 12 zeigt den Zustandsgraph einer Vorlage mit Parallelverzweigung. Da der vollständige Zustandsgraph nur logisch vorhanden sein muss, ergeben sich Realisierungsvarianten. So kann der Zustandsgraph zur Modellierzeit oder erst bei Instanzierung einer Prozessvorlage erstellt werden. Letzteres benötigt weniger Speicher, verschlechtert aber die Laufzeit. Die Erstellung zur Modellierzeit ermöglicht schnelles Weiterschalten und eine effiziente Realisierung der Schemaevolution. Nachteilig ist der große Speicherbedarf, der vor allem bei Verzweigungen aufgrund der kombinatorischen Vielfalt besteht. Bei Parallelverzweigungen wächst er exponentiell zur Anzahl der Teilzweige[1], bei alternativen Verzweigungen exponentiell zur Anzahl alternativer Verzweigungen[2]. Explizite Schleifen führen lediglich zu einem oder mehreren Rücksprüngen im Zustandsgraphen (siehe Abb. 12) ohne Auswirkungen auf den Speicherbedarf.

4.2 Anwendungsfälle

Die Clusterung von Instanzen hat signifikante Auswirkungen auf Zustandsänderungen und -überprüfungen bei Instanzen. Beispiele hierfür sind die Prüfung von Instanzzuständen bezüglich ihrer

[1] $O(l^b)$ (l: Knoten im längsten Pfad einer Verzweigung, b: Teilzweige der Verzweigung)
[2] $O(b^a)$ (b: Teilzweige pro Verzweigung, a alle alternativen Verzweigungen im Schema

Verträglichkeit bei Schemaevolution, die anschließende Propagation von Änderungen bei Schemaevolution sowie das Weiterschalten bei der normalen Prozessausführung.

4.2.1 Verträglichkeitsprüfung bei Schemaevolution

Die Clusterung von Instanzen erlaubt eine wesentliche Verbesserung der Verträglichkeitsprüfung bei Schemaevolution. Dabei müssen nicht mehr alle Instanzen inidividuell untersucht werden, es reicht, wenn die Veträglichkeit einmalig für jeden Cluster bestimmt wird. Hierdurch ergibt sich bei großer Anzahl von Instanzen bereits eine Effizienzsteigerung gegenüber der Verträglichkeitsprüfung für einzelne Instanzen.

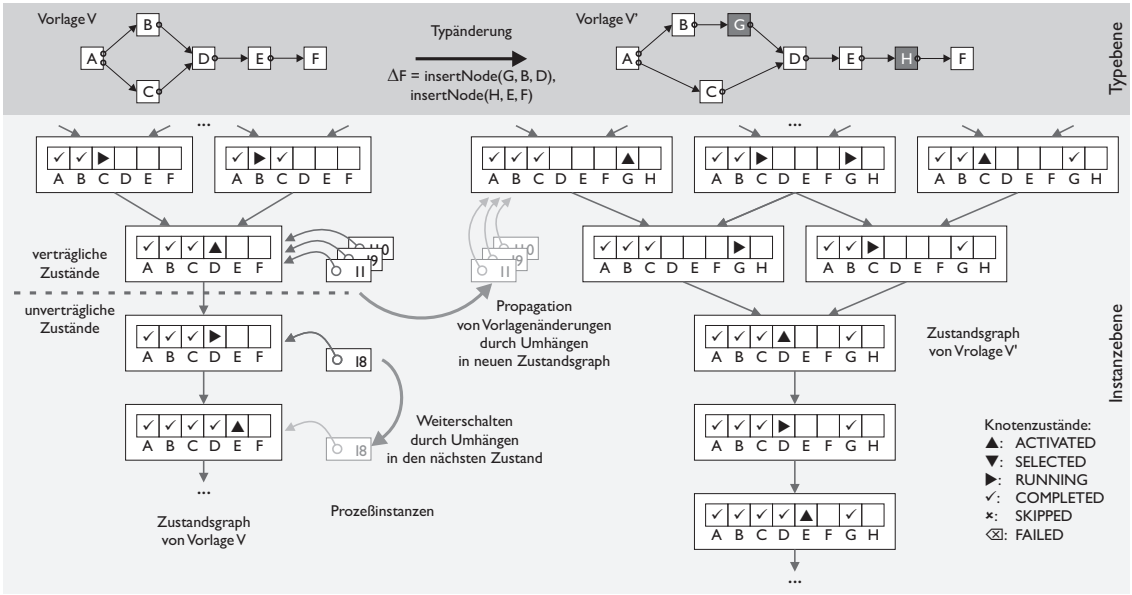

Abbildung 13: Zustandsclusterung von Instanzen

Aufgrund der Abhängigkeiten der einzelnen Cluster untereinander, kann die Verträglichkeitsprüfung noch weiter optimiert werden. Abhängig von der Art durchgeführter Vorlagenänderungen lassen sich Zustandscluster ohne aufwendige, individuelle Prüfungen jedes Clusters in drei Klassen einteilen: verträgliche, unverträgliche und zu überprüfende Cluster. Cluster, die nur aktive Knoten vor dem Änderungsbereich besitzen, sind verträglich. Cluster, deren aktive Knoten dem Änderungsbereich folgen, sind unverträglich. Für die Cluster mit aktiven Knoten im Änderungsbereich ist eine genaue Verträglichkeitsprüfung notwendig. Abb. 13 verdeutlicht dies. Aufgrund der Struktur des Zustandsgraphen und der Art durchgeführter Vorlagenänderungen gibt es in diesem Beispiel keine zu überprüfenden Cluster. Die Unterteilung in verträgliche und unverträgliche Cluster lässt sich anhand eines „Pivot"-Zustandsübergangs vornehmen. Instanzen in Cluster vor diesem Pivot-Zustandsübergang sind verträglich und können sofort der Änderungspropagation unterworfen werden, Instanzen in einem Cluster nach einem Pivot-Zustandsübergang sind unverträglich und verbleiben auf der ursprünglichen Vorlage (Abb. 13 links). Im allgemeinen Fall lassen sich die relevanten Zustandsübergänge durch Analyse durchgeführter Vorlagenänderungen bestimmen, wobei meist auf aufwendige Verträglichkeitsprüfungen verzichtet werden kann.

4.2.2 Änderungspropagation bei Schemaevolution

Alle bezüglich einer Schemaevolution veträglichen Instanzen können die bei der Evolution durchgeführten Änderungen erfahren. Das heißt, die strukturellen Änderungen und die ggf. notwendigen Zustandsanpassungen, etwa das Einfügen eines Knotens vor einem bereits aktivierten Knoten, werden auf die entsprechenden Instanzen angewandt. Auch hier ist bietet die Instanzclusterung

erhebliches Optimierungspotential: Sowohl die strukturellen Änderungen als auch die Zustandsanpassung werden einmal für jeden Cluster vorgenommen. Dabei entsteht eine Zuordnung zwischen den Clustern der alten und denen der neuen Prozessvorlage. Die Änderungspropagation besteht dann nur noch darin, Instanzen eines verträglichen Clusters direkt in den entsprechenden Cluster der neuen Prozessvorlage umzuhängen (vgl. Abb. 13).

4.2.3 Weiterschalten

Beim Weiterschalten ist wie bei der Schemaevolution keine individuelle Anpassung von Knotenzuständen notwendig. Stattdessen wird eine Instanz lediglich konform zu den Kanten im Zustandsgraph in einen nachfolgenden Zustandscluster umgehängt (Abb. 13 links). Existieren mehrere mögliche Nachfolger-Zustände (z. B. bei Parallelverzweigungen) muss überprüft werden, welcher Knoten eine Zustandsänderung erfahren hat. Insgesamt ergibt sich damit nur eine geringe Verbesserung gegenüber dem Weiterschalten bei individuellen Knotenzuständen. Neben Schemaevolution ist die Clusterung von Instanzen sehr vorteilhaft für Anfragen bezüglich des Zustands von Instanzkollektionen. Diese werden beispielsweise zu Monitoring-Zwecken oder für Analysen laufender Prozesse (*Process Performance Management*) benötigt [13].

4.3 Varianten

Aufgrund der großen Zustandsräume von Prozessvorlagen benötigt ein vollständiger Zustandsgraph sehr viel Speicher, insbesondere beim Auftreten von Verzweigungen. In diesem Abschnitt untersuchen wir verschiedene Varianten, die diese Situation verbessern können, indem die Anzahl der Cluster grundsätzlich oder zu einem bestimmten Zeitpunkt reduziert wird.

4.3.1 Variante IC-1 (Zustandsaggregation)

Bei Variante IC-1 werden verschiedene Zustände eines einzelnen Prozessknotens zusammengefasst, deren Unterscheidung selten oder überhaupt nicht benötigt wird. Dies gilt beispielsweise für die Zustände ACTIVATED und RUNNING in Abb. 12. Beide Zustände werden im Rahmen einer Schemaevolution nur dann unterschieden, wenn vor dem entsprechenden Knoten ein weiterer Knoten eingefügt wird: Im Zustand ACTIVATED ist dies möglich, im Zustand RUNNING nicht. Durch diese Aggregation wird der Speicherbedarf für den Zustandsgraphen signifikant reduziert, allerdings erhöht sich der Speicherbedarf für Instanzen, da dort die Unterscheidung der Zustände erfolgen muss. Für eine große Anzahl von Instanzen lohnt sich diese Aggregation nicht, da die Unterscheidung außerhalb des Clusters in diesem Fall mehr Speicher benötigt als im Cluster.

4.3.2 Variante IC-2 (Meilensteine)

Daneben können auch mehrere aufeinanderfolgende Prozessknoten zusammengefasst und innerhalb des Zustandsgraphs als ein Prozessknoten behandelt werden. Am einfachsten geht dies, indem bei blockbasierten Prozessmodellen (vgl. Abschnitt 3.4) alle Knoten eines Blocks zusammengefasst und stattdessen im Zustandsgraph ein einzelner abstrakter Knoten den gesamten Block repräsentiert. Diese Variante IC-2 (Meilensteine) ist jedoch nicht nur auf Blöcke sondern auf beliebige Mengen aufeinanderfolgender Prozessknoten anwendbar. Dadurch erhält man „Meilensteine“ innerhalb des Zustandsgraphs, die am Ende einer solchen Knotenmenge plaziert sind und die eine einfache Fortschrittskontrolle des Prozesses ermöglichen (Abb. 12).

Diese Meilensteine stellen Instanzzustände wesentlich gröber dar als (vollständige) Zustandsgraphen, was zur Einsparung von Speicherplatz insbesondere im Kontext von Verzweigungen führt. Aus Konsistenzgründen sollten Meilensteine nicht innerhalb einer Verzweigung liegen und diese unterteilen. Durchtrennt ein Meilenstein z. B. die Kanten B-D und C-E in der Vorlage aus Abb. 12, besitzt er nicht immer einen definierten Zustand. Befindet sich Knoten C im Zustand ACTIVATED und Knoten D im Zustand RUNNING, so ist der Meilenstein teilweise abgeschlossen und teilweise noch nicht. In diesem Fall ergibt sich keine Verbesserung gegenüber anderen Realisierungsmöglichkeiten für Zustände.

Durch die Vergröberung des Zustandsgraphen mittels Meilensteinen verringert sich neben dem Speicherbedarf auch der Nutzen bei Schemaevolution. So ist mit Meilensteinen keine effiziente Zustandsanpassung mehr möglich. Lediglich die Verträglichkeitsprüfung gestaltet sich einfacher als bei individuellen Knotenmarkierungen, da eine Klasseneinteilung analog den Pivot-Zustandsübergängen möglich ist. Je größer jedoch die Knotenmenge zwischen zwei Meilensteinen ist, um so größer wird die Klasse der zu überprüfenden Cluster, und damit auch die Anzahl der durchzuführenden Verträglichkeitsprüfungen.

Für die effiziente Unterstützung der Schemaevolution bei gleichzeitig geringem Speicherbedarf empfiehlt sich deshalb ein hybrides Verfahren als Kombination von Meilensteinen und expliziten Markierungen. Dabei werden die Markierungen der Knoten zwischen dem „aktiven Meilenstein“ und dem vorherigen Meilenstein explizit verwaltet. Die Zustände anderer Knoten ergeben sich implizit aus den Zuständen der jeweiligen Meilensteine. Dies ist vergleichbar mit der Realisierung mittels impliziter Markierungen (ZR-2, 3.3.2).

4.3.3 Variante IC-3 (Späte Clustererzeugung)

Der Speicherbedarf von Zustandsclustern kann auch dadurch verringert werden, indem nur solche Zustandscluster zur Laufzeit realisiert werden, die mindestens eine Instanz beinhalten. Dann belegen leere Cluster, denen zu einem Zeitpunkt keine Instanz zugeordnet ist, keinen Speicher. Dadurch wird jedoch das Weiterschalten eines Prozesses sehr aufwendig. So ist jedesmal vor „Umhängen“ einer Instanz zu überprüfen, ob der „Nachfolgecluster“ bereits existiert oder erst neu erzeugt und in den (unvollständigen) Zustandsgraphen eingefügt werden muss. Diese Verschlechterung der Laufzeit beim Weiterschalten zugunsten des Speicherbedarfs ist insofern problematisch, als dass Weiterschalten eine häufig benötigte Funktion ist.

Tabelle 5: Vergleich von Instanzclusterungsvarianten

	IC-1 Zustands-aggregation	IC-2 Meilensteine	IC-3 Späte Cluster-erzeugung
Speicherbedarf	O	++	+
Schemaevolution	+	--	-
Weiterschalten	+	O	-

Tabelle 5 bewertet alle vorgestellten Varianten zur Instanzclusterung bezüglich des Speicherbedarfs, der Unterstützung für Schemaevolution sowie des Weiterschaltens. Insgesamt bietet die Instanzclusterung großes Optimierungspotential. Für jede Instanz sind nur noch die Werte (bzw. Zustände) der im Prozess erzeugten Daten individuell zu speichern sowie Referenzen auf die Vorlage und den entsprechenden Zustandscluster. Hierdurch wird einerseits Schemaevolution effizient möglich, andererseits kann bei großer Anzahl Instanzen pro Vorlage Speicherplatz eingespart werden. Aufgrund der Größe vollständiger Zustandsgraphen ist dies jedoch im konkreten Anwendungsszenario zu untersuchen. Tabelle 6 (siehe S. 26) ermöglicht einen Vergleich des Speicherbedarfs von Instanzclustern gegenüber individuellen Knotenzuständen für verschiedene Mengengerüste. Ein guter Kompromiss sind Meilensteine mit expliziten Knotenmarkierungen.

5 Quantitativer Vergleich der Repräsentationsvarianten

Um die einzelnen Repräsentationsvarianten umfassend bewerten zu können, ist ein quantitativer Vergleich des (Primär-)Speicherbedarfs beim Einsatz in einem PMS unabdingbar. Dabei beschränken wir uns auf strukturelle Instanz- sowie Zustandsdaten. Im Prozess erzeugte Daten sind nicht Gegenstand der Betrachtung. Diese sind unabhängig von der Repräsentation von Kontrollflussstrukturen sowie Zustandsinformationen und stellen somit bei den folgenden Betrachtungen bezüglich des Speicherbedarfs lediglich einen konstanten Wert dar, von dem abstrahiert werden kann.

Der Speicherbedarf (M_{total}) hängt von zahlreichen Faktoren ab, beispielsweise der Anzahl der Vorlagen im System, der Komplexität der Vorlagen oder der Anzahl der Instanzen im System. Da diese zudem komplexe Abhängigkeiten besitzen können, werden im folgenden nur die wichtigsten Faktoren berücksichtigt und einige vereinfachende Annahmen getroffen. So nehmen wir an, dass alle Prozessvorlagen gleich groß und gleich aufgebaut sind, d. h. sie besitzen diesselbe Anzahl an parallelen (und alternativen) Verzweigungen, wobei diese wiederum alle gleich viele und gleich lange Zweige besitzen. Des weiteren wird angenommen, dass pro Knoten diesselben Einträge in der Ausführungshistorie existieren, d. h. dass jeder Knoten diesselben Zustände durchläuft und dieselben Zustandsübergänge protokolliert werden.

5.1 Vorlagenkopien mit expliziten Markierungen

Der Speicherbedarf von Vorlagenkopien mit expliziten Markierungen (VG-1 und ZR-3) ergibt sich aus der Anzahl der Instanzen im System ($N_{Instances}$) multipliziert mit dem Speicherbedarf einer Instanz. Der Speicherbedarf einer Instanz wiederum berechnet sich aus der Knotenanzahl einer Instanz (und damit auch einer Vorlage (N_{Nodes})) multipliziert mit der Summe aus Speicherbedarf für die statischen Informationen eines Knotens (z. B. ID, Bearbeiterzuordnungsregel; d. h. dem Speicherbedarf eines Vorlagenknotens (M_{Node})) sowie die Zustandsinformationen ($M_{NodeState}$) und die weiteren dynamischen Informationen ($M_{InstanceNode}$), z. B. Bearbeiter eines Knotens, Zeitinformationen, Iterationszähler.

$$M_{total} = N_{Instances} * N_{Nodes} * (M_{Node} + M_{NodeState} + M_{InstanceNode}) \tag{1}$$

5.2 Vorlagenreferenzen mit Ausführungshistorie

Der benötigte Speicher bei der Verwendung von Vorlagenreferenzen (VG-2) ergibt sich aus der Anzahl der Vorlagen im System ($N_{Temeplates}$) multipliziert mit dem Speicherbedarf für eine Vorlage. Dieser wiederum entspricht der Anzahl der Knoten (N_{Nodes}) multipliziert mit der Speichergröße der statischen Knoteninformationen (M_{Node}).

Die Zustands- und weitere dynamische Information (z. B der Bearbeiter eines Prozessschritts) werden durch die Ausführungshistorien zur Verfügung gestellt (ZR-1). Der hierfür benötigte Speicherplatz berechnet sich aus der Anzahl der verwalteten Instanzen ($N_{Instances}$) multipliziert mit der Größe einer Ausführunghistorie. Wir nehmen an, dass alle Historieneinträge gleich viel Speicherplatz benötigen (M_{Entry}). Die Anzahl der Historieneinträge entspricht der Anzahl der Knoten (N_{Nodes}) multipliziert mit der Anzahl der Historieneinträge pro Knoten ($N_{EntriesPerNode}$). Letztere ergibt sich aufgrund der Zustandsübergänge, die in der Ausführungshistorie protokolliert werden. Dies unterscheidet sich von der Gesamtzahl der Zustände im Metamodell (N_{States}), da ein Knoten im Normalfall nicht alle im Modell definierten Zustände durchläuft (z. B. abgeschlossen, abgebrochen, übersprungen).

$$\begin{aligned} M_{total} &= N_{Temeplates} * N_{Nodes} * M_{Node} \\ &+ N_{Instances} * N_{Nodes} * N_{EntriesPerNode} * M_{Entry} \end{aligned} \tag{2}$$

5.3 Vorlagenreferenzen mit impliziten Markierungen

Wie bei Vorlagenreferenzen mit Ausführungshistorie müssen bei Vorlagenreferenzen mit impliziten Markierungen (VG-2 und ZR-2) die Vorlagenobjekte nur einmal im Speicher bereitgestellt werden. Der Speicherbedarf berechnet sich wieder aus der Anzahl der Vorlagen multipliziert mit der mit der Anzahl der Knoten multipliziert mit dem benötigten Knotenspeicherplatz ($N_{Temeplates} * N_{Nodes} * M_{Node}$). Zusätzlich wird pro Instanz Speicherplatz für die impliziten Markierungen benötigt. Dieser hängt von der Struktur der zugrundeliegenden Vorlage ab (z. B. der Anzahl und Komplexität paralleler und alternativer Verzweigungen). Im Gegensatz zu den anderen Repräsentationsvarianten hängt der Speicherbedarf nicht von der Knotenanzahl ab.

Vereinfachend wird in der Berechnung ein konstanter Speicherbedarf für diese Informationen angenommen ($M_{ActiveNodes}$). Zusätzlich werden auch hier für jeden Instanzknoten dynamische Informationen benötigt ($M_{InstanceNode}$). Bei impliziten Markierungen werden jedoch Iterationszähler nur einmal pro Instanz und Schleife hinterlegt, sie sind somit bei $M_{ActiveNodes}$ berücksichtigt (im Vergleich mit anderen Varianten) und müssen deshalb vom Speicherbedarf eines Instanzknotens abgezogen werden ($M_{InstanceNode} - M_{IterationCount}$). Multipliziert man dies mit der Anzahl Knoten einer Instanz (N_{Nodes}) so ergibt sich der zusätzliche Speicherbedarf pro Instanz, der für dynamische Informationen (z. B. Zeitinformationen) benötigt wird.

$$\begin{aligned} M_{total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Instances} * (M_{ActiveNodes} + N_{Nodes} * (M_{InstanceNode} - M_{IterationCount})) \end{aligned} \tag{3}$$

5.4 Vorlagenreferenzen mit expliziten Markierungen

Vorlagenreferenzen mit expliziten Markierungen (VG-2 und ZR-3) benötigen neben dem Speicherplatz für die Vorlagen ($N_{Templates} * N_{Nodes} * M_{Node}$) Speicher für die Zustandsrepräsentation und die dynamischen Knotenfinormationen jeder Instanz (z. B. Zustände, Bearbeiter eines abgeschlossenen Schritts,...). Dieser entspricht der Anzahl der Knoten (N_{Nodes}) multipliziert mit dem Speicherbedarf für die Zustands- und Instanzknotenrepräsentation ($M_{NodeState} + M_{InstanceNode}$).

$$\begin{aligned} M_{total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Instances} * N_{Nodes} * (M_{NodeState} + M_{InstanceNode}) \end{aligned} \tag{4}$$

5.5 Zustandsclusterung mit expliziten Markierungen

Die Berechnung des Speicherbedarfs des Zustandsgraphen hängt in komplexer Weise von der Anzahl und Größe paralleler und alternativer Verzweigungen einer Prozessvorlage ab. Im folgenden betrachten wir deshalb zuerst sequentielle Prozesse, dann Prozesse mit alternativen Verzweigungen und anschließend Prozesse mit Parallelität.

Bei sequentiellen Prozessen ergibt sich der Speicherbedarf für den Zustandsgraphen durch die Anzahl der Zustandsknoten ($N_{StateNodes}$) multipliziert mit dem Speicherbedarf eines Zustandsknotens ($M_{StateNodes}$). Der Speicherbedarf eines Zustandsknoten umfasst die Zustandsliste (Abb. 12) und ergibt sich aus der Anzahl der Prozessknoten (N_{Nodes}) multipliziert mit dem Speicherbedarf für die Zustandsinformationen ($M_{NodeState}$). Die Anzahl der Zustandsknoten entspricht der Anzahl der Vorlagenknoten (N_{Nodes}) multipliziert mit der Anzahl der möglichen Zustände im verwendeten Metamodell (N_{States}) abzüglich des Speicherbedarfs für Initial- und Endzustand. Die Subtraktion ist nötig, da (in einem sequentiellen Prozess) der Initial- und der Endzustand implizit sind. Alle Vorgänger eines aktiven Knotens sind beendet, alle Nachfolger sind noch im Initialzustand.

$$\begin{aligned} MSeq_{StateNodes} &= NSeq_{StateNodes} * M_{StateNodes} \\ &= N_{Nodes} * (N_{States} - 2) * M_{NodeState} \end{aligned} \tag{5}$$

Diese Formel gilt nur für sequentielle Prozesse und Prozesse mit Schleifen, nicht aber für Prozesse mit alternativen oder parallelen Verzweigungen . Schleifen führen dabei lediglich zu Rücksprüngkanten im Zustandsgraphen und haben keine weiteren Auswirkungen (vgl. Abb. 12). Alternative Verzweigungen dagegen erhöhen den Speicherbedarf abhängig von der Anzahl der Zweige ($N_{Branches}(i)$) einer alternativen Verzweigung i, der Anzahl der Verzweigungen im gesamten Prozess sowie der Position einer Verzweigung im Prozess:

Bei jeder alternativen Verzweigung wird nur ein Zweig ausgewählt durchlaufen, alle anderen Zweige (bzw. deren Prozessknoten) bekommen den Zustand „übersprungen“. Der ausgewählte Zweig selbst durchläuft die normalen Zustandsübergänge, die die Prozessknoten des Zweiges ohne alternative Verzweigung durchlaufen würden. Da vorab nicht entschieden werden kann, welcher Zweig ausgewählt wird, ergibt sich für jede alternative Verzweigung auch eine Verzweigung im Zustandsgraphen, wobei deren Zweiganzahl mit der der alternativen Verzweigung im Prozess

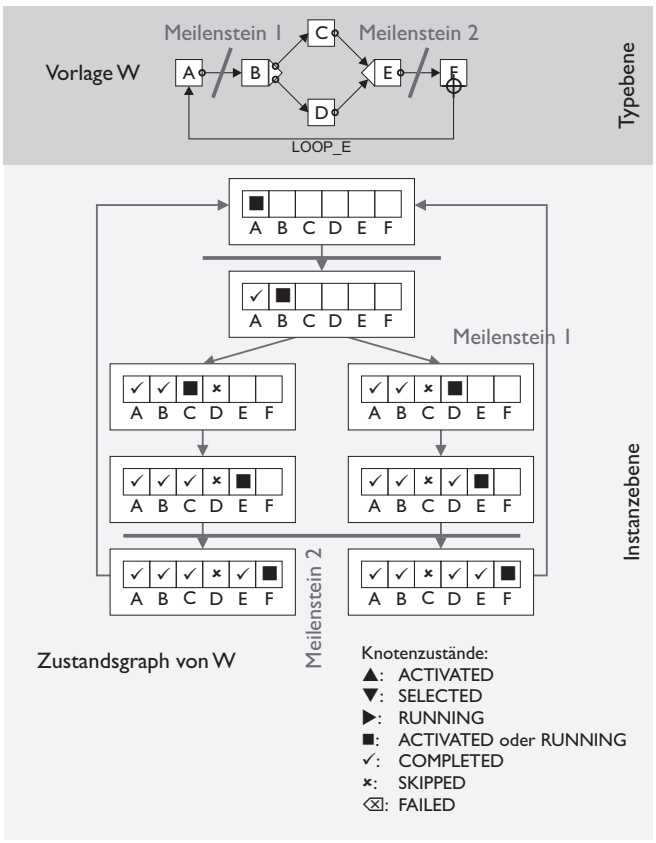

Abbildung 14: Zustandsgraph für Vorlage W mit alternativer Verzweigung (optimiert)

übereinstimmt. Allerdings führt der Vereinigungsknoten ($join_i$) einer alternativen Verzweigung nicht zu einer Vereinigung im Zustandsgraphen. Stattdessen müssen alle Zustandsknoten der Prozessknoten nach dem Vereinigungsknoten an jeden Zweig im Zustandsgraphen angefügt werden (vgl. Knoten F in Ausführung in Abb. 12). Andernfalls würde die Information, welcher Zweig ausgeführt wurde, im Zustandsgraph verlorengehen. Damit sind alle Zustandsknoten nach einer alternativen Verzweigung mehrfach vorhanden, nämlich einmal pro Zweig.

Dementsprechend führen mehrere (nicht verschachtelte) alternative Verzweigungen im Prozessgraph jeweils zu einer weiteren Aufteilung der vorhandenen Zweige im Zustandsgraphen; diese Verzweigungen sind verschachtelt. Deshalb ist die Formel für die Anzahl der Zustandsknoten einer alternativen Prozessverzweigung rekursiv:

$$NAlt_{StateNodes}(x,y) = \begin{cases} (N_{Nodes}(x, join_i) * (N_{States} - 2)) & , join_i \in nodes(x,y) \\ +N_{Branches}(i) * NAlt_{StateNodes}(join_i, y) & \\ & \\ NSeq_{StateNodes}(x,y) & , otherwise \\ = (N_{Nodes}(x,y) + N_{Nodes}(y,y)) * (N_{States} - 2) & \end{cases} \tag{6}$$

Dabei gibt $N_{Nodes}(x,y)$ die Anzahl der Prozessknoten zwischen x (einschließlich) und y (ausschließlich) an, $nodes(x,y)$ steht für die Menge der Prozessknoten von x bis y (einschließlich).

Der erste Teil der Formel berechnet die Anzahl der Zustandsknoten rekursiv. Sie ist anzuwenden, wenn sich zwischen den Prozessknoten x und y ein Vereinigungsknoten einer alternativen Verzweigung befindet. Existiert keine alternative Verzweigung, so gilt der zweite Teil der Formel. Dieser entspricht dem sequentiellen Fall.

Im Gegensatz zu alternativen Verzweigungen sind die einzelnen Zweige bei parallelen Verzweigungen vollständig unabhängig, die Zustände der einzelnen Zweige können beliebig kombiniert werden. Am Ende der Verzweigung findet eine Vereinigung aller parallelen Zweige statt. Da die Berechnung des Speicherbedarfs für parallele Verzweigungen sehr komplex ist, machen wir folgende vereinfachende Annahmen:

Die parallelen Verzweigungen aller Prozesse besitzen gleich viele Zweige ($N_{Branches}$), außerdem besitzt jeder Zweig diesselbe Anzahl an Knoten ($N_{NodesPerBranch}$). Zunächst berechnen wir die Anzahl der Zustandsknoten eines parallelen Verzweigungsblocks. Anschließend erweitern wir die Berechnung so, dass sich die Anzahl der Zustandsknoten für einen ganzen Prozess mit mehreren parallelen Verzweigungen ergibt.

Für die Anzahl der Zustandsknoten in einem parallelen Verzweigungsblock gilt:

$$NParBlock_{StateNodes} = (N_{NodesPerBranch} * (N_{States} - 2))^{N_{Branches}} \tag{7}$$

Die Anzahl der Zustandsknoten berechnet sich aus der Anzahl der Knoten eines Zweigs multipliziert mit der Anzahl der möglichen Zustände des Metamodells. Da die Zustände der einzelnen Zweige unabhängig voneinander sind und jeder Knotenzustand eines Zweigs mit jedem anderen Knotenzustand eines anderen Zweigs kombiniert werden kann, muss die ermittelte Anzahl ($N_{NodesPerBranch} * (N_{States} - 2)$) mit der Anzahl der Zweige der parallelen Verzweigung potenziert werden.

Um die Gesamtzahl an Zustandsknoten für einen ganzen Prozess mit mehreren parallelen Verzweigungen zu bestimmen, muss die Anzahl an Zustandsknoten für einen parallelen Block mit der Anzahl an parallelen Blöcken ($N_{ParallelBlocks}$) multipliziert werden – dies beruht auf der Annahme, dass jeder parallele Block gleich viele Knoten enthält. Zu den Zustandsknoten in parallelen Verzweigungen müssen die Zustandsknoten der umgebenden sequentiellen Prozessknoten addiert werden. Deren Anzahl ergibt sich durch die Anzahl aller Knoten im Prozess (N_{Nodes}) abzüglich

der Knoten in parallelen Verzweigungen ($N_{NodesPerBranch} * N_{Branches} * N_{ParallelBlocks}$).

$$\begin{aligned} NPar_{StateNodes} &= (N_{Nodes} - (N_{NodesPerBranch} * N_{Branches} * N_{ParallelBlocks})) * (N_{States} - 2) \\ &+ N_{ParallelBlocks} * (N_{NodesPerBranch} * (N_{States} - 2))^{N_{Branches}} \end{aligned} \tag{8}$$

Der gesamte Speicherbedarf für Zustandsclusterung hängt von der Anzahl an Vorlagen ($N_{Templates}$) ab. Für jede Vorlage wird die Vorlage selbst sowie ein Zustandsgraph verwaltet. Der Speicherbedarf für die Vorlagen berechnet sich wie bei den anderen Konzepten aus der Vorlagenanzahl multipliziert mit der Anzahl an Prozessknoten multipliziert mit dem Speicherbedarf eines (Vorlagen-)Knotens.

Der Speicherbedarf eines Knotenzustands ($M_{NodeState}$) multipliziert mit der Anzahl an Vorlagenknoten (N_{Nodes}) ergibt den Speicherbedarf eines Zustandsknotens. Multipliziert man diesen mit der Anzahl an Zustandsknoten ($N^*_{StateNodes}$ – abhängig von der Prozessstruktur, d h. $NSeq_{StateNodes}$, $NAlt_{StateNodes}$ und $NPar_{StateNodes}$) und das resultierende Ergebnis wiederum mit der Vorlagenanzahl, erhält man den Speicherbedarf für die Zustandsgraphen aller Vorlagen.

Daneben benötigen bei Zustandsclusterung auch Instanzen Speicherplatz. Diese beinhalten nicht mehr die eigentlichen Zustandsinformationen sondern nur Informationen, die bei jeder Instanz unterschiedlich sind (z. B. Bearbeiter abgeschlossener Schritte). Pro Instanz wird hierfür die Anzahl der Knoten multipliziert mit dem Speicherbedarf der dynamischen Informationen an ($N_{Nodes} * M_{InstanceNode}$). Die Zustandsinformation wird mittels einer Referenz auf den enstprechenden Knoten im Zustandsgraphen verwaltet. Diese benötigt auch etwas Speicher ($M_{Reference}$).

$$\begin{aligned} M_{total} &= N_{Templates} * N_{Nodes} * M_{Node} \\ &+ N_{Templates} * N^*_{StateNodes} * (N_{Nodes} * M_{NodeState}) \\ &+ N_{Instances} * (M_{Reference} + N_{Nodes} * M_{InstanceNode}) \end{aligned} \tag{9}$$

5.6 Beispiel

Basierend auf diesen Überlegungen zeigt Tabelle 6 beispielhaft den Speicherbedarf zwischen Vorlagenkopien (VG-1) und Vorlagenreferenzen (VG-2) sowie den verschiedenen Realisierungsvarianten für Knotenzustände (ZR-1, ZR-2, ZR-3) anhand einiger Mengengerüste für einfache (2 parallele Verzweigungen pro Prozessvorlage, 2 parallele Zweige pro Verzweigung, 3 Knoten pro Zweig) und parallele Prozesse (4 parallele Verzweigungen pro Prozessvorlage, 3 parallele Zweige pro Verzweigung, 6 Knoten pro Zweig). Neben der Komplexität der Prozesse variiert die Anzahl und das Verhältnis von Prozessvorlagen und Instanzen.

Tabelle 6: Speicherbedarf für versch. Realisierungsvarianten von Knotenzuständen

		einfache Prozesse 10 Vorl. à 20 Knoten, 100 Instanzen	parallele Prozesse 10 Vorl. à 100 Knoten, 100 Instanzen	einfache Prozesse 200 Vorl. à 20 Knoten, 100.000 Instanzen	parallele Prozesse 200 Vorl. à 100 Knoten, 100.000 Instanzen
Vorlagenkopien (expl. Markierungen) (VG-1)		512.000 Byte	2.560.000 Byte	472.800.000 Byte	2.364.000.000 Byte
Vorlagen-referenzen	Ausführungshistorie (ZR-1)	360.000 Byte	1.800.000 Byte	320.800.000 Byte	1.604.000.000 Byte
	impl. Markierungen (ZR-2)	108.000 Byte	532.000 Byte	68.800.000 Byte	336.000.000 Byte
	expl. Markierungen (ZR-3)	112.000 Byte	560.000 Byte	72.800.000 Byte	364.000.000 Byte
Instanzcluster mit expl. Markierungen (ZR-3)		174.400 Byte	55.958.400 Byte	72.480.000 Byte	1.462.560.000 Byte

Auffallend ist der hohe Speicherbedarf bei Vorlagenkopien mit expliziten Markierungen in allen Fällen. Trotz der im Vergleich zu Vorlagenreferenczen mit Ausführungshistorie kompakten Speicherung von Zuständen benötigen Vorlagenkopien mit expliziten Markierungen fast den doppelten Speicherbedarf. Dies liegt an der hohen Redundanz der strukturellen Informationen. Je mehr Instanzen pro Vorlage existieren, umso größer wird die Differenz. Auch die Vorlagenreferenzen mit Ausführungshistorie haben im Vergleich zu den anderen Varianten mit Vorlagenreferenzen einen relativ hohen Speicherbedarf. Somit fallen diese beiden keine ernsthaften Realisierungsvarianten – zumindest in der hier vorgestellten Form.

Den niedrigste Speicherbedarf haben Vorlagenreferenzen mit impliziten und expliziten Markierungen. Aufgrund des geringen Unterschieds können beide Varianten bezüglich der Speicherbedarfs als gleichwertig angesehen werden und somit die benötigte Laufzeit den Ausschlag für die eine oder die andere Variante gibt.

Der Speicherbedarf von Instanzcluster hängt erwartungsgemäß in sehr hohem Maß von der Komplexität der Prozesse ab – mehr als dies bei den anderen Realisierungsvarianten der Fall ist. Sie verhalten sich in gewisser Hinsicht umgekehrt wie Vorlagenkopien (mit expliziten Markierungen): Je mehr Instanzen pro Vorlage existieren, umso geringer wird der Speicherbedarf. Sie empfehlen sich in der unoptimierten Form deshalb nur für sehr einfache Prozesse oder für Einsätze, in denen es sehr viele Instanzen bei wenigen (einfachen) Vorlagen gibt. Aufgrund der erwarteten sehr guten Laufzeit bei Schemaevolution wird diese Realisierungsvariante noch Gegenstand weiterer Untersuchungen sein. Ein wichtiges Ziel dabei ist die Verringerung der Abhängigkeit zur Komplexität von Prozessvorlagen.

6 Physische Speicherstrukturen

Bisher haben wir die logische Repräsentation von Graphstrukturen, Zustandsdaten und weiteren Metadaten diskutiert. In diesem Abschnitt betrachten wir die konkrete Repräsentation der Vorlagen- und Instanzdaten. Ihre physische Repräsentation kann in Primär- und Sekundärspeicher unterschiedlich sein (z. B. objektorientiert und relational). Dann ist beim Ein-/Auslagern eine Transformation der Daten nötig (Abb. 15). Aus Performanzgründen empfiehlt sich, diese möglichst einfach zu gestalten. Durch Verwendung von *Stored Procedures* der DBMS können Transformationen eingespart werden, da keine Einlagerung in den Primärspeicher notwendig ist. Stored Procedures sind jedoch funktional nicht so einfach in der Handhabung wie Programmiersprachen und wesentlich schwieriger zu warten.

Ob auf Daten im Primär- oder Sekundärspeicher zugegriffen wird, hängt von der Zugriffsart ab. Werden Daten häufig benötigt (z. B. Propagation von Vorlagenänderungen), erfolgt dies am besten im Primärspeicher. Dagegen erfolgt die Abfrage oder Änderung einzelner Knotenattribute idealerweise direkt im Sekundärspeicher. Dies ist wesentlich effizienter, da die zum Ein-/Auslagern notwendige Transformation und Rücktransformation entfallen. Um die vom PMS bereitgestellten Funktionen sinnvoll nutzen zu können, ist es unabdingbar, dass für den Aufrufer einer Funktion weitestgehend transparent ist, in welchem Speicher Daten manipuliert werden.

6.1 Primärspeicher

Im Primärspeicher gibt es drei Arten von Speicherstrukturen für die physische Realisierung von Entitäten (z. B. Instanzen, Knoten): unstrukturierte Daten (z.B. untypisierte Container), strukturierte Daten (z. B. Objekte) und semi-strukturierte Daten (z. B. Arrays).

Variante PS-1 (Unstrukturierte Daten) Unstrukturierte Daten sind für die Speicherverwaltung des Betriebssystems bzw. der Laufzeitumgebung (z. B. JRE) rein binäre Objekte, d. h. sie werden ausschließlich in den zugreifenden Funktionen interpretiert. Dort ist auch die Objektstruktur hinterlegt (z. B. Speicherstelle des Datums und Offset des Attributs). Problematisch sind variabel große Attribute, etwa die als Zeichenkette bei einer Aktivität hinterlegte Bearbeiterzuordnung. Sie verhindern den direkten Zugriff auf Attribute, da die Position im Bitstrom variiert. Deshalb muss immer die gesamte Entität eingelesen werden, was viel Zeit kostet. Zudem wird hierdurch auch die Partitionierung erschwert.

Die Verwendung unstrukturierter Daten ist vergleichbar mit manueller Speicherverwaltung. Diese ist nur in Hochleistungsumgebungen sinnvoll, in denen Performanz oberste Priorität hat. Insbesondere müssen massive Probleme bei der Wartung in Kauf genommen werden. In allen anderen Fällen empfiehlt es sich, die Arbeit dem Compiler oder der Laufzeitumgebung zu überlassen und strukturierte Daten zu verwenden. Die automatische Speicherveraltung besitzt heutzutage

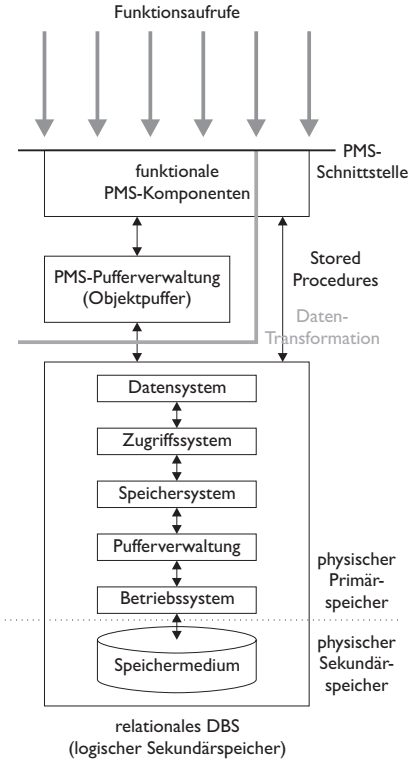

Abbildung 15: Datenzugriffe in PMS

aufgrund de hohen Entwicklungsstands nur geringe Auswirkungen auf Speicherbedarf und Laufzeitperformanz.

Variante PS-2 (Programmiersprachenobjekte) Aufgrund der Verbreitung von OO-Programmiersprachen bieten sich Objekte für die Realisierung von Entitäten an. Entitätsattribute werden als Objektvariablen direkt als Programmiersprachen-Konstruke realisiert. Dadurch ist ein schneller Zugriff ohne zusätzlichen algorithmischen Aufwand möglich. Es fällt jedoch zusätzlicher Speicherbedarf für Verwaltungsinformationen an. So benötigt ein Objekt in Java initial 8 Byte für den Objektheader. Dies ist bei vielen kleinen Objekten (z. B. ein Objekt pro Kante) problematisch; je kleiner ein Objekt ist, desto höher ist sein initialer Speicherbedarf. Programmiersprachenobjekte lassen sich zudem schlecht vertikal partitionieren. Dies ist mittels statischer Festlegung einzelner Objektpartitionen möglich. z. B. mittels Proxy-Objekten. Sie belegen wiederum zusätzlich Speicher für Objektheader.

Variante PS-3 (Semi-strukturierte Daten) Semi-strukturierte Daten bilden einen Kompromiss zwischen unstrukturierten Daten und Programmiersprachenobjekten. Dabei werden gleichartige Daten (z. B. bestimmtes Attribut einer Entität) zusammen im Speicherbereich abgelegt, z. B. mittels Array. Während bei Variante PS-2 jeweils eine Entität in einem zusammenhängenden Speicherbereich zu finden ist, ist bei Variante PS-3 in einem zusammenhängenden Speicherbereich (Array) genau ein Attribut aller Entitäten abgelegt. Der Zugriff auf entsprechende Attributwerte erfolgt ausschließlich über die ID, was in den meisten Fällen ausreicht. Findet ein Zugriff bezüglich eines bestimmten Attributwerts statt (z. B. Suche nach allen Knoten in bestimmtem Zustand), so kann dies durch die Bereitstellung von Indexstrukturen und -verfahren (z. B. Hashing) sinnvoll unterstützt werden.

Für die Repräsentation der in einem Hochleistungs-PMS anfallenden Daten sind semi-strukturierte Daten, also ein- oder mehrdimensionale Arrays aufgrund der guten Partitionierbarkeit und des

einfachen Ein-/Auslagerns am besten geeignet. Gewöhnlich gibt es nur wenige dynamische Änderungen, wodurch die Datenstrukturen stabil bleiben. Die Eignung für die Repräsentation im Primärspeicher hängt jedoch auch von den Speicherstrukturen im Sekundärspeicher ab. Da aufwendige Datentransformationen schnell zu einem Flaschenhals werden können, sind z. B. semistrukturierte Daten im Primärspeicher bei gleichzeitiger Verwendung einer OO-Datenbank nicht sinnvoll. Vorteile von semi-strukturierten Daten gegenüber Programmiersprachenobjekten sind einerseits der geringere Speicherbedarf, andererseits die strikte Trennung zwischen Daten und Funktionen. Unsere Erfahrungen mit der Implementierung des ADEPT2-Prototyps haben gezeigt, dass sich dadurch und durch die gute Partitionierbarkeit das Ein-/Auslagern vereinfacht. Ein Nachteil an semi-strukturierten Daten ist jedoch die mangelnde Dynamik der Strukturen. So müssen die IDs der Entitäten jederzeit stabil bleiben, was dazu führt, dass beim Löschen leere Einträge in den Speicherbereichen entstehen.

6.2 Sekundärspeicher

Auch im Sekundärspeicher kann zwischen unstrukturierten (z. B. BLOB) und strukturierten Daten (z. B. relationale, normalisierte Daten) unterschieden werden. Strukturierte Daten im Sekundärspeicher sind nur dann sinnvoll, wenn sie auch im Primärspeicher verwendet werden. Für die Umkehrung gilt dies nicht. So kann es günstig sein, strukturierte Daten im Primärspeicher entitätenweise zu serialisieren und unstrukturiert als BLOB im Sekundärspeicher abzulegen.

Unstrukturierte Daten im Sekundärspeicher können schnell gespeichert und ausgelesen werden. Jedoch sind keine externen Zugriffe und damit auch keine direkten Anfragen an den Sekundärspeicher (z. B. mittels SQL) möglich. Alle Zugriffe müssen über die vom PMS bereitgestellte Schnittstelle erfolgen. Eine Partitionierung unstrukturierter Daten im Sekundärspeicher ist nicht möglich. Diese müssen komplett ausgelesen und ggf. im Primärspeicher partitioniert werden. Relationale Datenbanken bieten eine ausgereifte Technik für die Speicherung strukturierter Daten. Sie ermöglichen eine einfache Transformation der Datenstrukturen zwischen Primär- und Sekundärspeicher. In vielen (relationalen) DBMS gibt es Unterstützung für *Stored Procedures*, so dass Funktionen direkt im Sekundärspeicher ausgeführt werden können. Zudem sind mittels SQL effiziente Anfragen gegen alle Daten und damit auch gegen Kollektionen von Instanzen möglich. Mit relationalen Datenstrukturen im Sekundär- und Programmiersprachenobjekte im Primärspeicher stellt sich das Problem des *Impedance Mismatch*. Eine große Schwierigkeit besteht dabei in der unterschiedlichen Realisierung von Referenzen im Primär- und Sekundärspeicher. Dies lässt sich nur mittels einer zentralen Komponente im PMS lösen, die alle zu einem Zeitpunkt existierenden Primärspeicherreferenzen kennt und die Sekundärspeicherreferenzen entsprechend transformieren kann.

Eine einfache Möglichkeit aufwendiges Ein-/Auslagern zu optimieren und die Transformation zwischen der Datenrepräsentation im Primär- und Sekundärspeicher zu vermeiden, ist die Pufferung der Daten zwischen Primär- und Sekundärspeicher (vgl. Abb. 15). Dabei ist zu beachten, dass ein DBMS intern schon Daten puffert. Eine Pufferverwaltung basierend auf Seiten und Segmenten wie im DBMS ist daher für das PMS nicht sinnvoll. Statt dessen sollten mit dem PMS-Puffer geeignete Objekte (z. B. Knoten, Kanten, Zustandsdaten, Historiendaten) verwaltet werden. Aufgrund unterschiedlicher Größen dieser Objekte, ist der Puffer flexibel zu gestalten, was eine aufwendige Pufferverwaltung erfordert. Das Konzept ist mit ORDBMS vergleichbar, in denen auch Objektpuffer verwendet werden. Diese werden entweder zusätzlich zu Seiten- und Segmentpuffer verwendet oder sie ersetzen diese [39]. Zudem ist eine Kooperation von DBMS- und PMS-Puffer nötig, damit Daten nicht redundant in beiden Puffern vorliegen. Die Redundanz kann z. B. dadurch vermieden werden, dass alle Vorlagen im Primärspeicher oder im PMS-Puffer gehalten werden und nur die Instanzdaten bei Bedarf von Sekundärspeicher eingelesen werden.

7 Zusammenfassung und Ausblick

Um PMS breit einsetzen zu können, müssen sie eine umfassende Funktionalität bieten, gleichzeitig aber auch performant arbeiten. In diesem Beitrag wurden Aspekte der effizienten Speicherrepräsentation von Vorlagen- und Instanzdaten untersucht. Dabei wurde deutlich, dass sich Entwurfsentscheidungen signifikant auf die Performanz auswirken, insbesondere aufgrund des Umfangs an zu berücksichtigenden funktionalen Anforderungen. Daneben ist die Art der repräsentierten Daten (Knoten, Kanten, Zustände,...) relevant.

Um eine umfassende Funktionalität realisieren zu können, ist eine „Metamodell-nahe" Repräsentation unabdingbar. Hierfür bieten sich interpretierte Graphen an. Diese sparen aufgrund der Partitionierbarkeit Speicherplatz ein. Knoten von Vorlagen werden am besten als eigene Objekte, Kanten als Adjazenzliste realisiert. Für Instanzen empfiehlt sich, das entsprechende Vorlagen nur zu referenzieren und für jeden Knoten eine explizite Markierung zu verwalten. Gegebenenfalls kann es sinnvoll sein, weitere Metadaten bereitzustellen, z. B. zu topologischer Ordnung oder expliziter Blockstruktur. Großes Optimierungspotential bietet die Clusterung von Instanzen, womit sich bei großer Instanzzahl pro Vorlage viel Speicherplatz sparen lässt. Dies ist Gegenstand zukünftiger Forschung, ebenso wie Aspekte der Repräsentation im Sekundärspeicher in Verbindung mit effizienter Pufferverwaltung.

Eine wichtige Funktion von PMS wurde im Rahmen dieser Arbeit nicht berücksichtigt: Instanzspezifische Änderungen, also Änderungen an individuellen Instanzen. Dies ist mit eines der Hauptanliegen bei der Implementierung von ADEPT2. Auch hier ist das Ziel, möglichst wenig Speicherplatz für die bereitzustellenden Daten zu verwenden und trotzdem effiziente Zugriffe zu ermöglichen. Weitere Performanzaspekte bei der Realisierung fortschrittlicher Funktionen werden in ADEPT2 berücksichtigt, etwa Sperrkonzepte und Synchronisation konkurrierender Zugriffe auf Daten (z. B. beim Weiterschalten) im Mehrbenutzerbetrieb sowie die Nutzung multipler Threads im System.

Literatur

[1] *Web Services Business Process Execution Language Version 2.0.* Technischer Bericht, OASIS, April 2007.

[2] *Business Process Modeling Notation (BPMN).* Technischer Bericht Version 1.2, OMG, January 2009.

[3] Aalst, W.M.P. van der, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm und A.J.M.M. Weijters: *Workflow mining: A Survey of Issues and Approaches.* Data & Knowledge Engineering, 47(2):237–267, 2003.

[4] Aalst, W.M.P. van der und K. Van Hee: *Workflow Management: Models, Methods, and Systems.* MIT Press, 2004.

[5] Bassil, S., M. Benyoucef, R.K. Keller und P. Kropf: *Addressing Dynamism in E-negotiations by Workflow Management Systems.* In: *13th International Workshop on Database and Expert Systems Applications (DEXA)*, Seiten 655–659, Aix-en-Provence, 2002.

[6] Bassil, S., R.K. Keller und P. Kropf: *A Workflow-Oriented System Architecture for the Management of Container Transportation.* Lecture Notes in Computer Science, LNCS 3080:116–131, June 2004.

[7] Baumgarten, B.: *Petri-Netze – Grundlagen und Anwendungen.* Spektrum Akademischer Verlag, 1996.

[8] Casati, F., S. Ceri, B. Pernici und G. Pozzi: *Workflow Evolution.* Data & Knowledge Engineering, 24(3):211–238, 1998.

[9] Cormen, T.H., C.E. Leiserson, R.L. Rivest und C. Stein: *Introduction to Algorithms.* MIT Press, 3 Auflage, September 2009.

[10] Dadam, P. und M. Reichert: *The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support – Challenges and Achievements.* Computer Science – Research and Development, 23(2):81–97, May 2009.

[11] Dogaç, A., L. Kalinichenko, T. Özsu und A. Sheth (Herausgeber): *Workflow Management Systems and Interoperability*, Band 164 der Reihe *NATO ASI Series F: Computer and System Sciences*, Istanbul, 1998. Springer.

[12] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.

[13] Grigori, D., F. Casati, M. Castellanos, U. Dayal, M. Sayal und M.-C. Shan: *Business Process Intelligence.* Computers in Industry, 53(3):321–343, 2004. Process / Workflow Mining.

[14] Heimann, P., G. Joeris, C.A. Krapp und B. Westfechtel: *DYNAMITE: Dynamic Task Nets for Software Process Management.* In: *Proceedings of the 18th ICSE*, Seiten 331–341, Berlin, 1996.

[15] Karbe, B.: *Flexible Vorgangssteuerung mit ProMInanD.* Seiten 117–133, 1994.

[16] Karbe, B., N. Ramsperger und P. Weiss: *Support of Cooperative Work by Electronic Circulation Folders.* ACM SIGOIS Bulletin, 11(2-3):109–117, 1990.

[17] Kloppmann, M., D. König, F. Leymann, G. Pfau und D. Roller: *Enabling Technology: Ein J2EE-basiertes Business Process Management System zur Ausführung von BPEL-und Web Service-basierten.* IT-Information Technology, 46(4):184–192, 2004.

[18] Kochut, K., J. Arnold, A. Sheth, J. Miller, E. Kraemer, B. Arpinar und J. Cardoso: *IntelliGEN: A Distributed Workflow System for Discovering Protein-Protein Interactions.* Distributed and Parallel Databases, 13(1):43–72, 2003.

[19] Lenz, R und M. Reichert: *IT Support for Healthcare Processes – Premises, Challenges, Perspectives.* Data & Knowledge Engineering, 61(1):39–58, 2007.

[20] Leymann, F.: *Web Services Flow Language (WSFL 1.0).* IBM Technical White Paper, IBM, 2001.

[21] Leymann, F. und W. Altenhuber: *Managing Business Processes as an Information Resource.* IBM Systems Journal, 33(2):326–348, 1994.

[22] Mangan, P. und S. Sadiq: *On Building Workflow Models for Flexible Processes.* Australian Computer Science Communications, 24(2):103–109, 2002.

[23] Müller, D., J. Herbst, M. Hammori und M. Reichert: *IT Support for Release Management Processes in the Automotive Industry.* Lecture Notes in Computer Science, LNCS 4102:368–377, October 2006.

[24] Müller, R., U. Greiner und E. Rahm: *AgentWork: A Workflow System Supporting Rule-based Workflow Adaptation.* Data & Knowledge Engineering, 51(2):223–256, 2004.

[25] Muth, P., D. Wodtke, J. Weissenfels, G. Weikum und A. Kotz Dittrich: *Enterprise-Wide Workflow Management Based on State and Activity Charts.* In: *NATO Advanced Science Institutes (ASI), Series F: Computer and Systems Sciences* [11], Seiten 281–303.

[26] Reichert, M.: *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* Dissertation, Universität Ulm, Juli 2000.

[27] Reichert, M. und P. Dadam: *ADEPTflex — Supporting Dynamic Changes of Workflows Without Losing Control.* Journal of Intelligent Information Systems, 10(2):93–129, 1998.

[28] Reichert, M. und P. Dadam: *Geschäftsprozessmodellierung und Workflow-Management – Konzepte, Systeme und deren Anwendung.* Industrie Management, 16(3):23–27, 2000.

[29] Reichert, M. und P. Dadam: *Realizing Adaptive Proceess-aware Information Systems with ADEPT2.* Technical Report 13, University of Ulm, Ulm, September 2008.

[30] Reichert, M., P. Dadam, S. Rinderle-Ma, M. Jurisch, U. Kreher und K. Göser: *PRIMIUM – Process Innovation for Enterprise Software*, Kapitel Architectural Principles and Components of Adaptive Process Management Technology, Seiten 81–97. Lecture Notes in Informatics (LNI). Koellen-Verlag, 2009.

[31] Reichert, M., S. Rinderle und P. Dadam: *ADEPT Workflow Management System: Flexible Support for Enterprise-wide Business Processes.* Lecture Notes in Computer Science, LNCS 2678:370–379, 2003.

[32] Reichert, M., S. Rinderle und P. Dadam: *On the Common Support of Workflow Type and Instance Changes under Correctness Constraints.* Lecture Notes in Computer Science, LNCS 2888:407–425, 2003.

[33] Reichert, M., S. Rinderle, U. Kreher und P. Dadam: *Adaptive Process Management with ADEPT2.* In: *Proc. International Conference on Data Engineering (ICDE)*, Seiten 1113–1114, Tokyo, 2005.

[34] Rinderle, S., U. Kreher, M. Lauer, P. Dadam und M. Reichert: *On Representing Instance Changes in Adaptive Process Management Systems.* In: *First IEEE Workshop on Flexibility in Process-aware Information Systems (ProFlex)*, Nummer 1, Seiten 297–302, Manchester, 2006.

[35] Rinderle, S., M. Reichert und P. Dadam: *Correctness Criteria for Dynamic Changes in Workflow Systems — A Survey.* Data & Knowledge Engineering, 50(1):9–34, 2004.

[36] Rinderle, S., M. Reichert und P. Dadam: *Flexible Support of Team Processes by Adaptive Workflow Systems.* Distributed and Parallel Databases, 16(1):91–116, 2004.

[37] Rinderle, S., M. Reichert und P. Dadam: *On Dealing with Structural Conflicts between Process Type and Instance Changes.* Lecture Notes in Computer Science, LNCS 3080:274–289, June 2004.

[38] Rinderle, S., M. Reichert, M. Jurisch und U. Kreher: *On Representing, Purging, and Utilizing Change Logs in Process Management Systems.* Lecture Notes in Computer Science, LNCS 4102:241–256, October 2006.

[39] Saake, G. und A. Heuer: *Datenbanken: Implementierungstechniken.* MITP-Verlag, 1999.

[40] Sheth, A. und K.J. Kochut: *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems.* In: *NATO Advanced Science Institutes (ASI), Series F: Computer and Systems Sciences* [11], Seiten 35–60.

[41] Wächter, H. und A. Reuter: *The ConTract Model.* In: Elmagarmid, A.K. (Herausgeber): *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[42] Wagenknecht, A. und U. Rüppel: *Improving Resource Management in Flood Response With process Models and webGIS.* In: *Proceedings of the 16th TIEMS Annual Conference 2009*, Seiten 141–151, Istanbul, June 2009.

[43] Weber, B., M. Reichert, W. Wild und S. Rinderle-Ma: *Providing Integrated Life Cycle Support in Process-Aware Information Systems.* Int'l Journal of Cooperative Information Systems (IJCIS), 18(1):115–165, 2009.

[44] Weber, B., S. Rinderle-Ma und M. Reichert: *Change Support in Process-Aware Information Systems – A Pattern-Based Analysis.* Technical Report 76, University of Twente, Enschede, 2007.

[45] Weber, B., S. Sadiq und M. Reichert: *Beyond Rigidity – Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-aware Information Systems.* Computer Science – Research and Development, 23(2):47–65, May 2009.

[46] Weske, M.: *Object-Oriented Design of a Flexible Workflow Management System.* Lecture Notes in Computer Science, LNCS 1475:119–130, September 1998.

A Zustandsbestimmung bei impliziten Zuständen

Definitionen [26]:

- Knotenzustand:
  $NS : Knoten\ N \longmapsto \{NOT\_ACTIVATED, ACTIVATED, SELECTED, RUNNING, COMPLETED, SKIPPED, FAILED\}$
- Knotentyp:
  $NT : Knoten\ N \longmapsto \{STARTFLOW, ENDFLOW, ACTIVITY, NULL, OR\_SPLIT, OR\_JOIN, AND\_SPLIT, AND\_JOIN, STARTLOOP, ENDLOOP\}$
- Menge der direkten Vorgänger-/Nachfolgerknoten:
  $pred : Knoten\ N \longmapsto \{Knoten\ N\},\ succ : Knoten\ N \longmapsto \{Knoten\ N\}$
- Menge der transitiven Vorgänger-/Nachfolgerknoten:
  $pred^* : Knoten\ N \longmapsto \{Knoten\ N\},\ succ^* : Knoten\ N \longmapsto \{Knoten\ N\}$

Gegeben:

- Instanz I, basierend auf Vorlage CFS
- Menge der aktiven Knoten von I:
  $A^I = \{n \mid NS(n) \in \{ACTIVATED, SELECTED, RUNNING\}\}$
- Menge der Teilzweige in alternativen Verzweigungspfade in CFS:
  $P^{CFS} = \{Z_i \mid Z_i = (succ^*(z_i) \cup \{z_i\}) \cap pred^*(join^{CFS}(s_j))\}, s_j \in SplitNodes^{CFS} \wedge NT(s_j) = OR\_SPLIT, z_i \in succ(s_j)\}$
- Menge der gewählten Pfade alternativer Verzweigungen bei der bisherigen Ausführung von I:
  $C^I = \{Z_i \mid \forall n \in Z_i : NS(n) = COMPLETED, Z_i \in P^{CFS}\}$

Gesucht:

- Knotenzustand von Knoten x: $NS(x)$

Algorithmus:

- $x \in A^I \Rightarrow NS(x) \in \{ACTIVATED, SELECTED, RUNNING\}$
  ist x ein aktiver Knoten, so wird sein Zustand explizit verwaltet und ist damit bekannt, Algorithmus abbrechen
- $x \in succ^*(n), n \in A^I \Rightarrow NS(x) = NOT\_ACTIVATED$
  ist x ein Nachfolger eines aktiven Knotens, so ist sein Zustand $NOT\_ACTIVATED$
- $x \in pred^*(n), n \in A^I \Rightarrow NS(x) \in \{COMPLETED, SKIPPED\}$
  ist x ein Vorgänger eines aktiven Knotens, so sind weitere Überprüfungen notwendig
  - $x \notin C^I \wedge x \in P^{CFS} \Rightarrow NS = SKIPPED$
    befindet sich x in einem Teilzweig, jedoch nicht in einem ausgeführten Teilzweig, so ist sein Zustand $SKIPPED$
  - $NS(x) = COMPLETED$
    in allen anderen Fällen befindet sich x im Zustand $COMPLETED$

Anmerkung:

- Explizite Schleifenkanten werden bei diesem Algorithmus ignoriert. Die rekonstruierten Knotenzustände geben innerhalb einer Schleife den Zustand beim letztmaligen (bzw. dem aktuellen) Schleifendurchlauf an. Dies reicht aus, wenn zusätzliche Iterationszähler in den Zustandsinformationen einer Instanz mitgeführt werden (s. 3.3.2).

B Verwendete Abkürzungen für qualitativen Vergleich

$join_i$: Vereinigungsknoten der Verzweigung i

$M_{ActiveNodes}$: Speicherbedarf (pro Instanz) für Informationen zur Verwaltung aller aktiven Knoten
Bei Variante ZR-2 (s. Abschnitt 3.3.2) werden nur die aktiven Knoten einer Instanz sowie einige weitere Informationen zur vollständigen Rekonstruktion des Instanzzustands gespeichert.

M_{Entry}: Speicherbedarf für einen Eintrag in der Ausführungshistorie

$M_{InstanceNode}$: Speicherbedarf für die Repräsentation eines Instanzknotens

$M_{IterationCount}$: Speicherbedarf für den Iterationszähler für einen Instanzknoten

M_{Node}: Speicherbedarf für einen Knoten einer Prozessvorlage

$M_{NodeState}$: Speicherbedarf für die Repräsentation eines Knotenzustands

$M_{Reference}$: Speicherbedarf für die Referenz von einer Instanz zu dem ihren Zustand repräsentierenden Knoten im entsprechenden Zustandsgraphen

$M_{StateNodes}$: Speicherbedarf eines Knotens eines Zustandsgraphen

M_{total}: Gesamter Speicherbedarf für alle Vorlagen- und Instanzdaten

$MSeq_{StateNodes}$: Speicherbedarf eines (Teil-)Zustandsgraphen für sequentielle Prozessknoten

$nodes(x, y)$: Menge der Prozessknoten von x bis y (einschließlich)

$N_{Branches}$: Anzahl der Zweige einer parallelen Verzweigung (konstant)

$N_{Branches}(i)$: Anzahl der Zweige der alternativen Verzweigung i

$N_{EntriesPerNode}$: Anzahl der Einträge in der Ausführungshistorie pro Knoten
Da ein Knoten mehrere Zustände durchläuft werden mehrere Einträge in der Ausführungshistorie protokolliert.

$N_{Instances}$: Anzahl aller Prozessinstanzen im System unabhängig von der jeweiligen Prozessvorlage

N_{Nodes}: Anzahl der Knoten pro Prozessvorlage

$N_{Nodes}(x, y)$: Anzahl der Prozessknoten zwischen x (einschließlich) und y (ausschließlich)

$N_{NodesPerBranch}$: Anzahl der Knoten eines Zweigs einer parallelen Verzweigung (konstant)

$N_{ParallelBlocks}$: Anzahl der parallelen Blöcke in einer Prozessvorlage

$N^*_{StateNodes}$: Anzahl aller Knoten eines Zustandsgraphen unter Berücksichtigung der Struktur der entsprechenden Prozessvorlage

$N_{StateNodes}(x, y)$: Anzahl der Knoten eines Zustandsgraphen für alle Prozessknoten zwischen x (einschließlich) und y (ausschließlich)

N_{States}: Anzahl aller Zustände des verwendeten Prozessmetamodells

$N_{Templates}$: Anzahl aller Prozessvorlagen im System

$NAlt_{StateNodes}(x, y)$: Anzahl aller Knoten eines Zustandsgraphen für alle Prozessknoten zwischen x (einschließlich) und y (ausschließlich) einer Prozessvorlage mit mindestens einer alternativen Verzweigung

$NPar_{StateNodes}$: Anzahl aller Knoten eines Zustandsgraphen für Prozessknoten in einer Sequenz

$NParBlock_{StateNodes}$: Anzahl aller Knoten eines Zustandsgraphen für Prozessknoten in einer parallelen Verzweigung

$NSeq_{StateNodes}$: Anzahl aller Knoten eines Zustandsgraphen für Prozessknoten in einer Sequenz

$NSeq_{StateNodes}(x, y)$: Anzahl aller Knoten eines Zustandsgraphen für alle Prozessknoten in einer Sequenz zwischen x (einschließlich) und y (ausschließlich)

Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm

Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity

91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03* *Alfons Geser*
Relative Termination

91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP

91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions

91-06* *Uwe Schöning*
Recent Highlights in Structural Complexity Theory

91-07* *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit

91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk, A. Ogiwara, U. Schöning, R. Silvestri, T. Thierauf*
Reductions for Sets of Low Information Content

92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simulataneous Evaluation of Noncircular Attribute Grammars

92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04* *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions

92-05* *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy

92-06* *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics

92-07* *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

92-08* *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets

92-09* *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network

92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function

92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes

92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets

92-13 *Alfons Geser*
On a monotonic semantic path ordering

92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager

93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model

93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions

93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System

93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP

94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers

94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems

94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model

94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages

94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software

94-10 *Axel Dold*
Formalisierung schematischer Algorithmen

94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits

94-12 *Rainer Schuler*
On Average Polynomial Time

94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14 *Wolfram Schulte, Ton Vullinghs*
Linking Reactive Software to the X-Window System

94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen

94-16 *Robert Regn*
Verteilte Unix-Betriebssysteme

94-17 *Helmuth Partsch*
Again on Recognition and Parsing of Context-Free Grammars:
Two Exercises in Transformational Programming

94-18 *Helmuth Partsch*
Transformational Development of Data-Parallel Algorithms: an Example

95-01 *Oleg Verbitsky*
On the Largest Common Subgraph Problem

95-02 *Uwe Schöning*
Complexity of Presburger Arithmetic with Fixed Quantifier Dimension

95-03 *Harry Buhrman,Thomas Thierauf*
The Complexity of Generating and Checking Proofs of Membership

95-04 *Rainer Schuler, Tomoyuki Yamakami*
Structural Average Case Complexity

95-05 *Klaus Achatz, Wolfram Schulte*
Architecture Indepentent Massive Parallelization of Divide-And-Conquer Algorithms

95-06 *Christoph Karg, Rainer Schuler*
Structure in Average Case Complexity

95-07 *P. Dadam, K. Kuhn, M. Reichert, T. Beuter, M. Nathe*
ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen

95-08 *Jürgen Kehrer, Peter Schulthess*
Aufbereitung von gescannten Röntgenbildern zur filmlosen Diagnostik

95-09 *Hans-Jörg Burtschick, Wolfgang Lindner*
On Sets Turing Reducible to P-Selective Sets

95-10 *Boris Hartmann*
Berücksichtigung lokaler Randbedingung bei globaler Zieloptimierung mit neuronalen Netzen am Beispiel Truck Backer-Upper

95-12 *Klaus Achatz, Wolfram Schulte*
Massive Parallelization of Divide-and-Conquer Algorithms over Powerlists

95-13 *Andrea Mößle, Heiko Vogler*
Efficient Call-by-value Evaluation Strategy of Primitive Recursive Program Schemes

95-14 *Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
A Generic Specification for Verifying Peephole Optimizations

96-01 *Ercüment Canver, Jan-Tecker Gayen, Adam Moik*
Formale Entwicklung der Steuerungssoftware für eine elektrisch ortsbediente Weiche mit VSE

96-02 *Bernhard Nebel*
Solving Hard Qualitative Temporal Reasoning Problems: Evaluating the Efficiency of Using the ORD-Horn Class

96-03 *Ton Vullinghs, Wolfram Schulte, Thilo Schwinn*
An Introduction to TkGofer

96-04 *Thomas Beuter, Peter Dadam*
Anwendungsspezifische Anforderungen an Workflow-Mangement-Systeme am Beispiel der Domäne Concurrent-Engineering

96-05 *Gerhard Schellhorn, Wolfgang Ahrendt*
Verification of a Prolog Compiler - First Steps with KIV

96-06 *Manindra Agrawal, Thomas Thierauf*
Satisfiability Problems

96-07 *Vikraman Arvind, Jacobo Torán*
A nonadaptive NC Checker for Permutation Group Intersection

96-08 *David Cyrluk, Oliver Möller, Harald Rueß*
An Efficient Decision Procedure for a Theory of Fix-Sized Bitvectors with Composition and Extraction

96-09 *Bernd Biechele, Dietmar Ernst, Frank Houdek, Joachim Schmid, Wolfram Schulte*
Erfahrungen bei der Modellierung eingebetteter Systeme mit verschiedenen SA/RT–Ansätzen

96-10 *Falk Bartels, Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
Formalizing Fixed-Point Theory in PVS

96-11 *Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
Mechanized Semantics of Simple Imperative Programming Constructs

96-12 *Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß*
Generic Compilation Schemes for Simple Programming Constructs

96-13 *Klaus Achatz, Helmuth Partsch*
From Descriptive Specifications to Operational ones: A Powerful Transformation Rule, its Applications and Variants

97-01 *Jochen Messner*
Pattern Matching in Trace Monoids

97-02 *Wolfgang Lindner, Rainer Schuler*
A Small Span Theorem within P

97-03 *Thomas Bauer, Peter Dadam*
A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration

97-04 *Christian Heinlein, Peter Dadam*
Interaction Expressions - A Powerful Formalism for Describing Inter-Workflow Dependencies

97-05 *Vikraman Arvind, Johannes Köbler*
On Pseudorandomness and Resource-Bounded Measure

97-06 *Gerhard Partsch*
Punkt-zu-Punkt- und Mehrpunkt-basierende LAN-Integrationsstrategien für den digitalen Mobilfunkstandard DECT

97-07 *Manfred Reichert, Peter Dadam*
$ADEPT_{flex}$ - Supporting Dynamic Changes of Workflows Without Loosing Control

97-08 *Hans Braxmeier, Dietmar Ernst, Andrea Mößle, Heiko Vogler*
The Project NoName - A functional programming language with its development environment

97-09 *Christian Heinlein*
Grundlagen von Interaktionsausdrücken

97-10 *Christian Heinlein*
Graphische Repräsentation von Interaktionsausdrücken

97-11 *Christian Heinlein*
Sprachtheoretische Semantik von Interaktionsausdrücken

97-12 *Gerhard Schellhorn, Wolfgang Reif*
Proving Properties of Finite Enumerations: A Problem Set for Automated Theorem Provers

97-13 *Dietmar Ernst, Frank Houdek, Wolfram Schulte, Thilo Schwinn*
Experimenteller Vergleich statischer und dynamischer Softwareprüfung für eingebettete Systeme

97-14 *Wolfgang Reif, Gerhard Schellhorn*
Theorem Proving in Large Theories

97-15 *Thomas Wennekers*
Asymptotik rekurrenter neuronaler Netze mit zufälligen Kopplungen

97-16 *Peter Dadam, Klaus Kuhn, Manfred Reichert*
Clinical Workflows - The Killer Application for Process-oriented Information Systems?

97-17 *Mohammad Ali Livani, Jörg Kaiser*
EDF Consensus on CAN Bus Access in Dynamic Real-Time Applications

97-18 *Johannes Köbler,Rainer Schuler*
Using Efficient Average-Case Algorithms to Collapse Worst-Case Complexity Classes

98-01 *Daniela Damm, Lutz Claes, Friedrich W. von Henke, Alexander Seitz, Adelinde Uhrmacher, Steffen Wolf*
Ein fallbasiertes System für die Interpretation von Literatur zur Knochenheilung

98-02 *Thomas Bauer, Peter Dadam*
Architekturen für skalierbare Workflow-Management-Systeme - Klassifikation und Analyse

98-03 *Marko Luther, Martin Strecker*
A guided tour through *Typelab*

98-04 *Heiko Neumann, Luiz Pessoa*
Visual Filling-in and Surface Property Reconstruction

98-05 *Ercüment Canver*
Formal Verification of a Coordinated Atomic Action Based Design

98-06 *Andreas Küchler*
On the Correspondence between Neural Folding Architectures and Tree Automata

98-07 *Heiko Neumann, Thorsten Hansen, Luiz Pessoa*
Interaction of ON and OFF Pathways for Visual Contrast Measurement

98-08 *Thomas Wennekers*
Synfire Graphs: From Spike Patterns to Automata of Spiking Neurons

98-09 *Thomas Bauer, Peter Dadam*
Variable Migration von Workflows in *ADEPT*

98-10 *Heiko Neumann, Wolfgang Sepp*
Recurrent V1 – V2 Interaction in Early Visual Boundary Processing

98-11 *Frank Houdek, Dietmar Ernst, Thilo Schwinn*
Prüfen von C–Code und Statmate/Matlab–Spezifikationen: Ein Experiment

98-12 *Gerhard Schellhorn*
Proving Properties of Directed Graphs: A Problem Set for Automated Theorem Provers

98-13 *Gerhard Schellhorn, Wolfgang Reif*
Theorems from Compiler Verification: A Problem Set for Automated Theorem Provers

98-14 *Mohammad Ali Livani*
SHARE: A Transparent Mechanism for Reliable Broadcast Delivery in CAN

98-15 *Mohammad Ali Livani, Jörg Kaiser*
Predictable Atomic Multicast in the Controller Area Network (CAN)

99-01 *Susanne Boll, Wolfgang Klas, Utz Westermann*
A Comparison of Multimedia Document Models Concerning Advanced Requirements

99-02 *Thomas Bauer, Peter Dadam*
Verteilungsmodelle für Workflow-Management-Systeme - Klassifikation und Simulation

99-03 *Uwe Schöning*
On the Complexity of Constraint Satisfaction

99-04 *Ercument Canver*
Model-Checking zur Analyse von Message Sequence Charts über Statecharts

99-05 *Johannes Köbler, Wolfgang Lindner, Rainer Schuler*
Derandomizing RP if Boolean Circuits are not Learnable

99-06 *Utz Westermann, Wolfgang Klas*
Architecture of a DataBlade Module for the Integrated Management of Multimedia Assets

99-07 *Peter Dadam, Manfred Reichert*
Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications. Paderborn, Germany, October 6, 1999, GI–Workshop Proceedings, Informatik ’99

99-08 *Vikraman Arvind, Johannes Köbler*
Graph Isomorphism is Low for ZPP^{NP} and other Lowness results

99-09 *Thomas Bauer, Peter Dadam*
Efficient Distributed Workflow Management Based on Variable Server Assignments

2000-02 *Thomas Bauer, Peter Dadam*
Variable Serverzuordnungen und komplexe Bearbeiterzuordnungen im Workflow-Management-System ADEPT

2000-03 *Gregory Baratoff, Christian Toepfer, Heiko Neumann*
Combined space-variant maps for optical flow based navigation

2000-04 *Wolfgang Gehring*
Ein Rahmenwerk zur Einführung von Leistungspunktsystemen

2000-05 *Susanne Boll, Christian Heinlein, Wolfgang Klas, Jochen Wandel*
Intelligent Prefetching and Buffering for Interactive Streaming of MPEG Videos

2000-06 *Wolfgang Reif, Gerhard Schellhorn, Andreas Thums*
Fehlersuche in Formalen Spezifikationen

2000-07 *Gerhard Schellhorn, Wolfgang Reif (eds.)*
FM-Tools 2000: The 4th Workshop on Tools for System Design and Verification

2000-08 *Thomas Bauer, Manfred Reichert, Peter Dadam*
Effiziente Durchführung von Prozessmigrationen in verteilten Workflow-Management-Systemen

2000-09 *Thomas Bauer, Peter Dadam*
Vermeidung von Überlastsituationen durch Replikation von Workflow-Servern in ADEPT

2000-10 *Thomas Bauer, Manfred Reichert, Peter Dadam*
Adaptives und verteiltes Workflow-Management

2000-11 *Christian Heinlein*
Workflow and Process Synchronization with Interaction Expressions and Graphs

2001-01 *Hubert Hug, Rainer Schuler*
DNA-based parallel computation of simple arithmetic

2001-02 *Friedhelm Schwenker, Hans A. Kestler, Günther Palm*
3-D Visual Object Classification with Hierarchical Radial Basis Function Networks

2001-03 *Hans A. Kestler, Friedhelm Schwenker, Günther Palm*
RBF network classification of ECGs as a potential marker for sudden cardiac death

2001-04 *Christian Dietrich, Friedhelm Schwenker, Klaus Riede, Günther Palm*
Classification of Bioacoustic Time Series Utilizing Pulse Detection, Time and Frequency Features and Data Fusion

2002-01 *Stefanie Rinderle, Manfred Reichert, Peter Dadam*
Effiziente Verträglichkeitsprüfung und automatische Migration von Workflow-Instanzen bei der Evolution von Workflow-Schemata

2002-02 *Walter Guttmann*
Deriving an Applicative Heapsort Algorithm

2002-03 *Axel Dold, Friedrich W. von Henke, Vincent Vialard, Wolfgang Goerigk*
A Mechanically Verified Compiling Specification for a Realistic Compiler

2003-01 *Manfred Reichert, Stefanie Rinderle, Peter Dadam*
A Formal Framework for Workflow Type and Instance Changes Under Correctness Checks

2003-02 *Stefanie Rinderle, Manfred Reichert, Peter Dadam*
Supporting Workflow Schema Evolution By Efficient Compliance Checks

2003-03 *Christian Heinlein*
Safely Extending Procedure Types to Allow Nested Procedures as Values

2003-04 *Stefanie Rinderle, Manfred Reichert, Peter Dadam*
On Dealing With Semantically Conflicting Business Process Changes.

2003-05 *Christian Heinlein*
Dynamic Class Methods in Java

2003-06 *Christian Heinlein*
Vertical, Horizontal, and Behavioural Extensibility of Software Systems

2003-07 *Christian Heinlein*
Safely Extending Procedure Types to Allow Nested Procedures as Values
(Corrected Version)

2003-08 *Changling Liu, Jörg Kaiser*
Survey of Mobile Ad Hoc Network Routing Protocols)

2004-01 *Thom Frühwirth, Marc Meister (eds.)*
First Workshop on Constraint Handling Rules

2004-02 *Christian Heinlein*
Concept and Implementation of C+++, an Extension of C++ to Support User-Defined Operator Symbols and Control Structures

2004-03 *Susanne Biundo, Thom Frühwirth, Günther Palm(eds.)*
Poster Proceedings of the 27th Annual German Conference on Artificial Intelligence

2005-01 *Armin Wolf, Thom Frühwirth, Marc Meister (eds.)*
19th Workshop on (Constraint) Logic Programming

2005-02 *Wolfgang Lindner (Hg.), Universität Ulm , Christopher Wolf (Hg.) KU Leuven*
2. Krypto-Tag – Workshop über Kryptographie, Universität Ulm

2005-03 *Walter Guttmann, Markus Maucher*
Constrained Ordering

2006-01 *Stefan Sarstedt*
Model-Driven Development with ACTIVECHARTS, Tutorial

2006-02 *Alexander Raschke, Ramin Tavakoli Kolagari*
Ein experimenteller Vergleich zwischen einer plan-getriebenen und einer leichtgewichtigen Entwicklungsmethode zur Spezifikation von eingebetteten Systemen

2006-03 *Jens Kohlmeyer, Alexander Raschke, Ramin Tavakoli Kolagari*
Eine qualitative Untersuchung zur Produktlinien-Integration über Organisationsgrenzen hinweg

2006-04 *Thorsten Liebig*
Reasoning with OWL - System Support and Insights –

2008-01 *H.A. Kestler, J. Messner, A. Müller, R. Schuler*
On the complexity of intersecting multiple circles for graphical display

2008-02 *Manfred Reichert, Peter Dadam, Martin Jurisch,l Ulrich Kreher, Kevin Göser, Markus Lauer*
Architectural Design of Flexible Process Management Technology

2008-03 *Frank Raiser*
Semi-Automatic Generation of CHR Solvers from Global Constraint Automata

2008-04 *Ramin Tavakoli Kolagari, Alexander Raschke, Matthias Schneiderhan, Ian Alexander*
Entscheidungsdokumentation bei der Entwicklung innovativer Systeme für produktlinien-basierte Entwicklungsprozesse

2008-05 *Markus Kalb, Claudia Dittrich, Peter Dadam*
Support of Relationships Among Moving Objects on Networks

2008-06 *Matthias Frank, Frank Kargl, Burkhard Stiller (Hg.)*
WMAN 2008 – KuVS Fachgespräch über Mobile Ad-hoc Netzwerke

2008-07 *M. Maucher, U. Schöning, H.A. Kestler*
An empirical assessment of local and population based search methods with different degrees of pseudorandomness

2008-08 *Henning Wunderlich*
Covers have structure

2008-09 *Karl-Heinz Niggl, Henning Wunderlich*
Implicit characterization of FPTIME and NC revisited

2008-10 *Henning Wunderlich*
On span-P^{cc} and related classes in structural communication complexity

2008-11 *M. Maucher, U. Schöning, H.A. Kestler*
On the different notions of pseudorandomness

2008-12 *Henning Wunderlich*
On Toda's Theorem in structural communication complexity

2008-13 *Manfred Reichert, Peter Dadam*
Realizing Adaptive Process-aware Information Systems with ADEPT2

2009-01 *Peter Dadam, Manfred Reichert*
The ADEPT Project: A Decade of Research and Development for Robust and Fexible Process Support
Challenges and Achievements

2009-02 *Peter Dadam, Manfred Reichert, Stefanie Rinderle-Ma, Kevin Göser, Ulrich Kreher, Martin Jurisch*
Von ADEPT zur AristaFlow® BPM Suite – Eine Vision wird Realität "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen

2009-03 *Alena Hallerbach, Thomas Bauer, Manfred Reichert*
Correct Configuration of Process Variants in Provop

2009-04 *Martin Bader*
On Reversal and Transposition Medians

2009-05 *Barbara Weber, Andreas Lanz, Manfred Reichert*
Time Patterns for Process-aware Information Systems: A Pattern-based Analysis

2009-06 *Stefanie Rinderle-Ma, Manfred Reichert*
Adjustment Strategies for Non-Compliant Process Instances

2009-07 *H.A. Kestler, B. Lausen, H. Binder H.-P. Klenk. F. Leisch, M. Schmid*
Statistical Computing 2009 – Abstracts der 41. Arbeitstagung

2009-08 *Ulrich Kreher, Manfred Reichert, Stefanie Rinderle-Ma, Peter Dadam*
Effiziente Repräsentation von Vorlagen- und Instanzdaten in Prozess-Management-Systemen

Ulmer Informatik-Berichte
ISSN 0939-5091

Herausgeber:
Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
89069 Ulm