Speichereffiziente Repräsentation instanzspezifischer Änderungen in Prozess-Management-Systemen

Ulrich Kreher, Manfred Reichert

Ulmer Informatik-Berichte

Nr. 2010-02
März 2010

Speichereffiziente Repräsentation instanzspezifischer Änderungen in Prozess-Management-Systemen

Ulrich Kreher, Manfred Reichert

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

Zusammenfassung

Neben Funktionen für die Steuerung und Verwaltung von Prozessen muss ein Prozess-Management-System (PMS) auch eine gewisse Flexibilität für Endbenutzer bieten. So sollte es beispielsweise möglich sein, zur Laufzeit fallspezifisch und flexibel vom vordefinierten Prozess abzuweichen, d. h. die betreffende Prozessinstanz strukturell zu modifizieren. Entsprechende Ad-hoc-Änderungen dürfen jedoch weder zu Lasten der Robustheit des PMS noch auf Kosten der Systemperformanz gehen, insbesondere wenn eine große Zahl von Instanzen verwaltet werden muss. Robustheitsaspekte im Zusammenhang mit der Unterstützung von Flexibilität des PMS sind bereits in mehreren Arbeiten theoretisch untersucht worden. In diesem Beitrag untersuchen wir, wie Flexibilität in Prozess-Management-Systemen systemintern realisiert werden kann und wie dies möglichst performant bewerkstelligbar ist. Dazu diskutieren wir verschiedene Realisierungskonzepte sowie einige Implementierungsvarianten für Änderungen auf Prozessen und bewerten diese sowohl qualitativ als auch quantitativ. Eine der vorgestellten Implementierungsvarianten ist aktuell im ADEPT2-PMS umgesetzt.

1 Einleitung

Prozess-Management-Systeme (PMS) erlauben durch die Trennung von Prozess- und Anwendungslogik eine flexible Komposition von Anwendungsdiensten [22]. Diese Flexibilität wird jedoch nicht nur a-priori bei der Prozessmodellierung bzw. -erstellung benötigt, sondern auch zur Laufzeit [12, 15]. Sie ist bereits in mehreren Arbeiten untersucht worden [18, 3, 31], jedoch konzentrieren sich diese überwiegend auf theoretische Aspekte wie der Korrektheit dynamischer Änderungen [16, 17] oder der Semantik angebotener Änderungsmuster [29, 27]. Außerdem wird meist nur eine Art von Prozessadaptionen betrachtet, d. h. entweder Änderungen einzelner Prozessinstanzen oder Änderungen an Prozessvorlagen und deren Propagation auf laufende Prozessinstanzen. Für eine hohe Benutzerakzeptanz ist jedoch die Unterstützung beider Änderungsarten und vor allen Dingen deren Kombination unabdingbar. Dies ist im adaptiven Prozess-Management-System (PMS) ADEPT2 der Fall [7, 25, 23, 20]. Neben umfassenden Anforderungen an das dynamischen Änderungen zugrundeliegende theoretische Modell ergeben sich durch Kombination beider Änderungsarten auch weitreichende Konsequenzen für deren praktische Umsetzung. In diesem Beitrag untersuchen wir, wie diese praktische Umsetzung konkret aussehen kann. Aufbauend auf der effizienten Repräsentation von Vorlagen- und (unveränderten) Prozessinstanzen [10] gilt unser Hauptaugenmerk der Effizienz der Realisierung.

1.1 Problembeschreibung

In einem Prozess-Management-System werden *Prozessvorlagen* verwaltet [10]. Dabei wird die *Prozessstruktur* häufig graphbasiert mittels Knoten und Kanten repräsentiert [14, 13, 18, 16, 1]. Knoten repräsentieren Aktivitäten, die innerhalb eines Prozesses durchgeführt werden müssen, Kanten

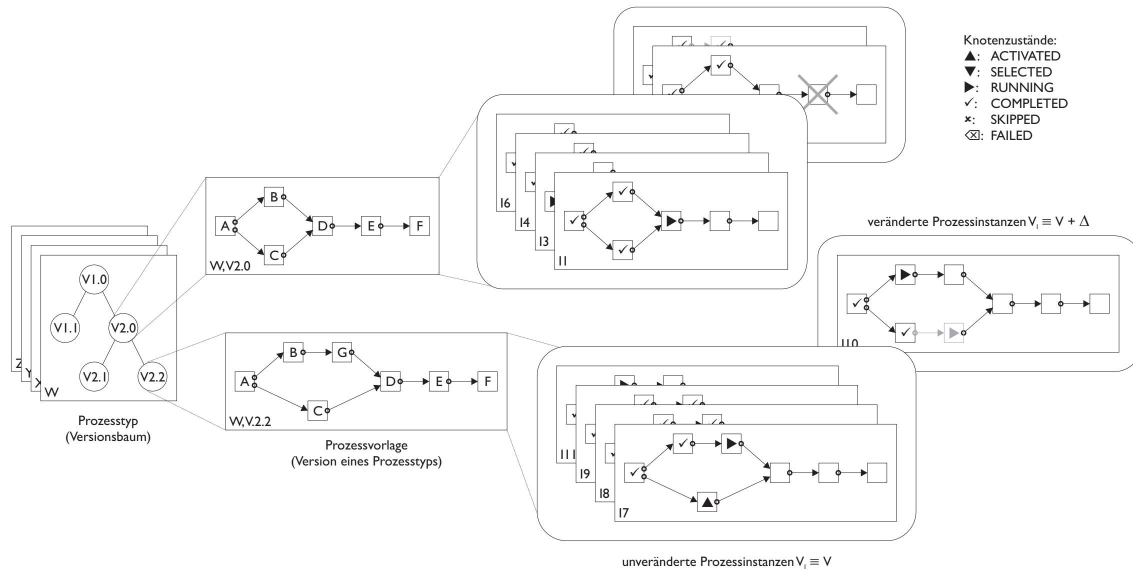

Abbildung 1: Verwaltung von Prozessgraphen in einem adaptiven PMS (logische Sicht)

entsprechen (Reihenfolge-)Abhängigkeiten zwischen diesen Aktivitäten. Eine konkrete Prozessausführung erfolgt auf Grundlage einer *Prozessinstanz*, die eine Prozessvorlage mit Zustandsinformationen versieht. Diese Zustandsinformationen spiegeln sich in *Knotenzuständen* wider, wobei die Gesamtheit der Zustände aller Knoten einer Prozessinstanz den jeweils aktuellen *Instanzzustand* definiert. Ferner wird die Ausführung jeder Prozessinstanz in *Ausführungs-* und *Datenhistorien* protokolliert [2, 9, 28]. Prozessvorlagen werden zur einfacheren Verwaltung in *Prozesstypen* kategorisiert (s. Abb. 1)

Flexibilität wird von einem PMS insbesondere durch die rasche und fehlerfreie Änderbarkeit von Prozessen unterstützt [21, 6]. Dies betrifft einerseits *Schemaevolution*, also das Ändern von Prozessvorlagen (sog. *Vorlagenänderungen*) und ggf. die Propagation dieser Änderungen auf laufende Prozessinstanzen (*Migration von Instanzen*) [24, 30], andererseits die Möglichkeit, ausgewählte laufende Prozessinstanzen jederzeit individuell ändern zu können (sog. *Instanzänderungen* oder *instanzspezifische Änderungen*, siehe Instanz I10 rechts in Abb. 1) [18]. Beide Aspekte bilden unabhängige Dimensionen des „Problemraums“ der effizienten Repräsentation von Vorlagen- und Instanzdaten. Eine weitere Dimension bilden die verschiedenen Prozesstypen innerhalb eines PMS (links in Abb. 1). Allerdings kann davon abstrahiert werden, da Änderungen auf Vorlagenebene sinnvollerweise nur innerhalb der Prozessvorlagen eines Prozesstyps (d. h. eines Versionsbaums) durchgeführt werden und die Prozesstypen damit voneinander unabhängig sind.

Wie in [10] diskutiert, ist eine weitere Dimension durch die Speicherart gegeben, in der Prozessdaten repräsentiert werden. Gemeint ist die Speicherrepräsentation in Primär- und Sekundärspeicher, wobei die Repräsentation der Prozessdaten jeweils unterschiedlich sein kann. Damit ergibt sich die in Abb. 2 gezeigte Struktur. Während wir die Bereiche I und II bereits in [10] diskutiert haben, untersuchen wir in diesem Beitrag den Bereich III sowie zusätzlich die Dimension der Vorlagen(-änderungen) der Bereiche I bis III. Der Bereich IV ist Gegenstand zukünftiger Arbeiten; prinzipiell lassen sich die Überlegungen zur Repräsentation von unveränderten Prozessinstanzen in Primär- und Sekundärspeicher (I und II) aber auf veränderten Instanzen (III und IV) übertragen.

1.2 Anforderungen

Die Anforderungen an die effiziente Unterstützung instanzspezifischer Änderungen sind mit den Anforderungen bei der Repräsentation von Vorlagen und unveränderten Instanzen vergleichbar [10]. Zusätzlich müssen beide Änderungsarten in Kombination miteinander unterstützt werden.

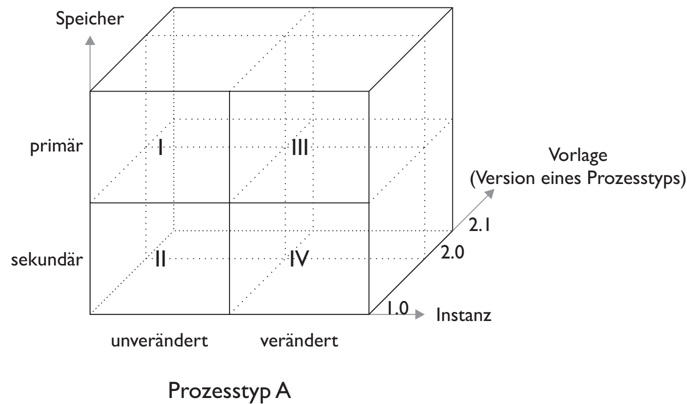

Abbildung 2: Problemdimensionen bei der Verwaltung von Prozessdaten

Wir zeigen, dass dies Auswirkungen auf die Repräsentation unveränderter Instanzen (Bereich I und II) hat. Zu beachten ist, dass instanzspezifische Änderungen in bestimmten Einsatzszenarien eines PMS den Normalfall bilden [5, 4] und damit eine performante Repräsentation unabdingbar ist.

Die Effizienz der Repräsentation von Vorlagen- und Instanzdaten spiegelt sich zum einen im Primärspeicherbedarf der Prozessdaten und zum anderen in der Laufzeit verschiedener Funktionen wider. Hierzu zählen insbesondere das Weiterschalten im Prozess (nach Beendigung von Aktivitäten) sowie die transitive Bestimmung von Vorgängerknoten, d. h. die Ermittlung von Knoten, die direkt oder indirekt Vorgänger eines bestimmten Knotens sind.

Eine weitere wichtige Anforderung ist die weitgehende Transparenz von instanzspezifischen Änderungen für diese Funktionen. Die entsprechenden Datenstrukturen sollten intern so dargestellt werden, dass für Zugriffe kein Unterschied zwischen einer unveränderten und einer veränderten Instanz besteht. Dies soll natürlich nur gelten, sofern der Unterschied für den Zugriff nicht relevant ist, wie beispielsweise bei der Schemaevolution veränderter Instanzen. Dies erleichtert die Implementierung erheblich, da dieselben Algorithmen sowohl für unveränderte als auch veränderte Prozessinstanzen anwendbar bleiben. Auch die Partitionierung von Prozessdaten [10] ist für die Unterstützung von Flexibilität sehr hilfreich. Auch dies muss weitgehend transparent realisiert werden. Des weiteren sollten auch Anfragen von Kollektionen von Instanzen möglichst unverändert und effizient auch mit instanzspezifischen Änderungen funktionieren.

Dieser Beitrag gliedert sich wie folgt: In Kapitel 2 stellen wir Änderungsoperationen auf Prozessen und deren Anwendung im Rahmen von Schemaevolution, instanzspezifischer Änderungen sowie die Anwendung einer Schemaevolution mit geänderten Instanzen. Kapitel 3 diskutiert verschiedene Konzepte für die Realisierung von Prozessänderungen. Eines dieser Realisierungskonzepte wird in Kapitel 4 in zwei Implementierungsvarianten in konkrete Datenstrukturen umgesetzt. Anschließend werden diese beiden Implementierungsvarianten sowie die weiteren Realisierungskonzepte in Kapitel 5 quantitativ verglichen. Hierzu stellen wir die entsprechende Testumgebung und den Ablauf der Messungen vor, bevor wir die Messergebnisse ausführlich diskutieren und interpretieren. Abschließend folgen in Kapitel 6 eine Zusammenfassung und ein Ausblick.

2 Grundlagen

Für die interne Repräsentation von Änderungen an Prozessvorlagen und -instanzen ist es wichtig, wie entsprechende Prozessanpassungen generell durchgeführt werden und welche Auswirkungen die Änderungen in Bezug auf spätere Zugriffe auf systeminterne Datenstrukturen haben, z. B. Anfragen nach den aktuell laufenden Schritte eines Prozesses. In diesem Abschnitt stellen wir zuerst einige generelle Aspekte von Änderungen vor, die für das weitere Verständnis des Aufsat-

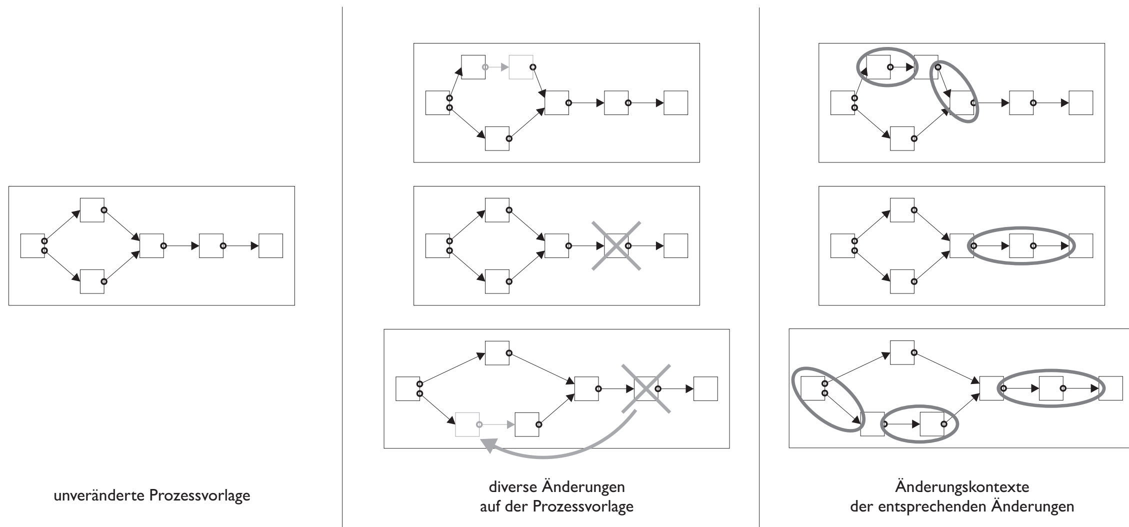

Abbildung 3: Änderungskontexte beim Einfügen, Löschen und Verschieben von Knoten

zes wichtig sind. Anschließend gehen wir im Detail auf die vom ADEPT2-System unterstützen Prozessänderungen ein. Zum einen sind dies Änderungen auf Prozessvorlagen und ggf. deren Propagation auf zugehörige Prozessinstanzen (Schemaevolution), zum anderen Änderungen einzelner Prozessinstanzen (instanzspezifische Änderungen).

2.1 Änderungsoperationen

Um die korrekte Durchführbarkeit von Änderungen auf Prozessen zu gewährleisten (Instanz I10 in Abb. 1), definiert das Änderungsrahmenwerk des ADEPT2-Systems eine umfassende theoretische Grundlage [21, 19]. Hierzu gehört insbesondere eine wohldefinierte Menge an *Änderungsoperationen* (z. B. Einfügen, Löschen oder Verschieben eines Knotens (Abb. 3) oder Einfügen eines Verzweigungs-/Schleifenblocks). Diese Änderungen können verschieden komplex sein, bauen letztlich aber alle auf einfachen *Änderungsprimitiven* auf. Letztere beschränken sich auf das Erzeugen, Ändern und Löschen einzelner Knoten und Kanten des jeweiligen Prozessgraphen. Höherwertige Änderungsoperationen fassen mehrere Änderungsprimitive so zusammen, dass bei Anwendung der Änderungsoperation auf einen Prozessgraphen dieser immer von einer gültigen Struktur in eine andere gültige Struktur überführt wird. Dazu müssen Einschränkungen bezüglich der Prozessgraphstruktur (z. B. Einhaltung von Blockstruktur, keine isolierten Knoten) eingehalten werden. Darüber hinaus gibt es im Fall einer Prozessinstanz weitere Einschränkungen bezüglich des Prozesszustands. Beispielsweise müssen alle Vorgänger eines laufenden Prozessschritts entweder abgeschlossen oder „abgewählt“ worden sein, und alle Nachfolger müssen sich noch in ihrem Initialzustand befinden. Änderungsoperationen bei deren Durchführung dies nicht gewährleistet ist, dürfen erst gar nicht angewandt werden. So ist etwa das Einfügen eines neuen Knotens vor einem bereits abgeschlossenen Schritt nicht zulässig. Wir sprechen in einem solchen Fall von der *Unverträglichkeit* der Instanz mit den aus der Änderung resultierenden Prozessvorlage.

Mehrere Änderungsoperationen können in einer *Änderungstransaktion* zusammengefasst werden. Dadurch können beispielsweise komplexe Änderungen atomar durchgeführt werden. Ein Beispiel hierfür ist das Ersetzen eines Knotens durch einen anderen Knoten, was durch die beiden Änderungsoperationen Einfügen und Löschen realisiert werden kann. Strukturell ist der resultierende Prozess nach der Anwendung jeder einzelnen Änderungsoperation korrekt, die semantische Konsistenz ist jedoch erst nach Abschluss der Änderungstransaktion hergestellt.

Änderungsoperationen betreffen immer einen bestimmen Bereich eines Prozessgraphen, den sogenannten *Änderungskontext*. Dieser ist je nach Änderungsoperation unterschiedlich. So umfasst der Änderungskontext beim (seriellen) Einfügen eines Knotens die zwei Knoten zwischen

denen der neue Knoten eingefügt wird; beim Verschieben von Knoten umfasst der Änderungskontext die Kante, die die ursprüngliche Position des verschobenen Knotens „überspringt“, die zwei Knoten zwischen denen der verschobene Knoten eingefügt wird sowie den verschobenen Knoten selbst (Abb. 3). Der Änderungsbereich markiert damit die Stellen, an denen die Graphstruktur in der internen Datenrepräsentation angepasst werden muss, um die entsprechenden Änderungen zu realisieren.

Ebenso wie die Ausführung einer Prozessinstanz in einer Ausführungs- und Datenhistorie protokolliert wird (z. B. aus Gründen der Nachvollziehbarkeit) [10], werden alle Änderungen einer Prozessvorlage oder -instanz in einer *Änderungshistorie* protokolliert [26]. Diese beinhaltet Informationen über Änderungstransaktionen, angewandte Änderungsprimitive sowie Änderungskontexte. Ähnlich wie Knotenmarkierungen bei Prozessinstanzen eine konsolidierte Sicht auf die Ausführungshistorie ermöglichen, bietet die interne Repräsentation von Änderungen eine konsolidierte Sicht auf die Änderungshistorie.

2.2 Schemaevolution und instanzspezifische Änderungen

Im Prozess-Management-System ADEPT2 kann eine vollständige Menge an Änderungsoperationen sowohl auf Prozessvorlagen als auch auf Prozessinstanzen angewandt werden. Im ersten Fall enstehen Prozessvorlagen, die eine neue Version des entsprechenden Prozesstyps bilden und die vorherigen Prozessvorlagen ablösen. Die Änderungen, die auf der ursprünglichen Prozessvorlage vorgenommen worden sind, können im Rahmen einer Schemaevolution auch auf bereits laufende Prozessinstanzen der ursprünglichen Vorlage propagiert werden [24]. Dabei werden diese Prozessinstanzen noch während ihrer Ausführung auf die neue Prozessvorlage migriert, sofern dies gewünscht und möglich ist. Konkret sind mehrere Randbedingungen zu beachten. Beispielsweise ist es zwecks Gewährleistung einer korrekten Prozessausführung nicht gestattet, Änderungen auf eine laufende Prozessinstanz zu propagieren, wenn die betroffene Instanz den entsprechenden Änderungskontext bereits durchlaufen hat. Wäre dies erlaubt, würde durch die Schemaevolution nachträglich die Vergangenheit der Prozessausführung verändert bzw. „manipuliert“ (vgl. Instanz 2 in Abb. 4).

Im Fall von instanzspezifischen Änderungen gelten die durchgeführten Änderungen nur für eine einzelne laufende Prozessinstanz, während bei einer Schemaevolution bzw. Instanzmigration die Änderungen potentiell für alle Instanzen einer Prozessvorlage gelten. Dementsprechend muss in der internen Repräsentation von Instanzdaten gewährleistet werden, dass die entsprechenden Änderungen nur für die betroffene Prozessinstanz gelten. In diesem Fall sind die der Instanz zugrundeliegenden Prozessstrukturen Änderungen unterworfen und entsprechen damit nicht mehr den durch die entsprechende Prozessvorlage bereitgestellten Prozessstrukturen.

Wie bei der Schemaevolution ist es auch bei instanzspezifischen Änderungen nicht zulässig, die Vergangenheit zu manipulieren, d. h. hier gelten weitgehend diesselben Korrektheitskriterien für die Anwendung von Änderungsoperationen. Allerdings ist die Reihenfolge für die Überprüfung sowie die Durchführung der Änderungen etwas anders. Wir werden dies nun im Detail vorstellen (vgl. Abb. 4)

Ablauf einer Schemaevolution

Der erste Schritt einer Schemaevolution ist die Erzeugung einer neuen Prozessvorlage V' basierend auf einer bereits existierenden Prozessvorlage V (im Sinne von Klonen). Die neu erzeugte Prozessvorlage wird anschließend einer Menge von Änderungen ΔF unterworfen. Dabei wird vor Anwendung einer jeden Änderungsoperation strukturelle Verträglichkeit der Instanz mit der resultierenden (Zwischen-)Vorlage geprüft (z. B. Einhaltung der Blockstruktur) und ggf. die Struktur der Prozessvorlage entsprechend geändert.

Sind alle gewünschten Änderungen der Prozessvorlage erfolgt und auf V' angewandt worden, können die Änderungen auf laufende Instanzen von V propagiert werden. Dazu wird zuerst überprüft, ob ΔF bezüglich des aktuellen Instanzzustands verträglich ist, d. h. ob der Änderungsbereich von ΔF bereits abgeschlossene Teile der Instanz betrifft. Ist dies der Fall, kann die Instanz nicht

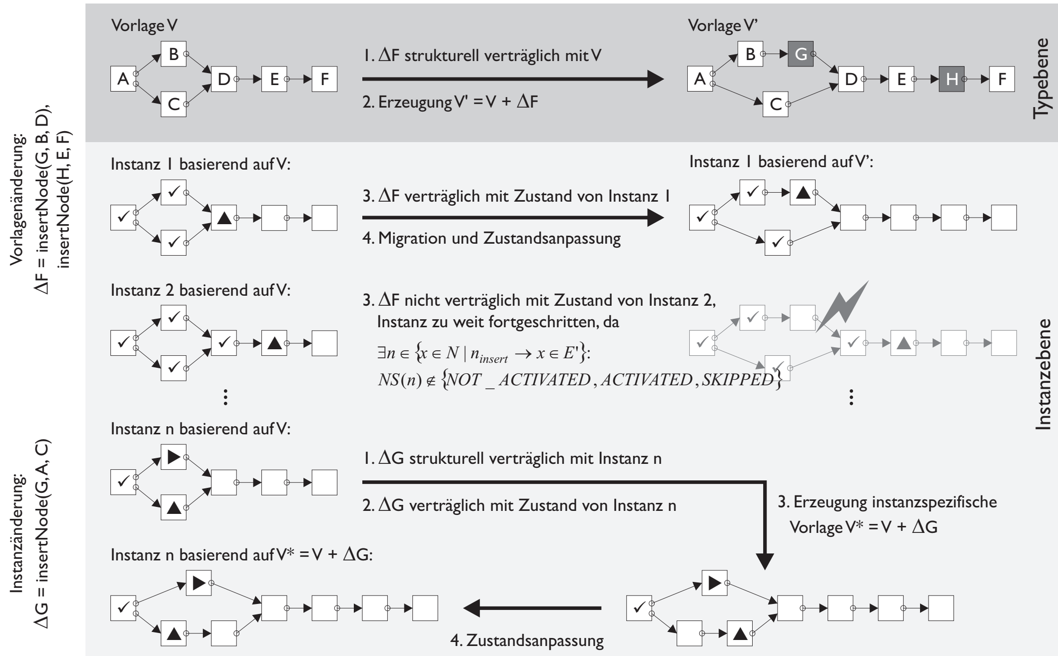

Abbildung 4: Vorlagen- und Instanzänderung (logische Sicht)

auf die neue Vorlage V' migriert werden und muss auf der alten Prozessvorlage V verbleiben, d. h. auf ihrer Grundlage zu Ende geführt werden. Ist ΔF dagegen bezüglich des Zustands verträglich, wird die Instanz auf die neue Vorlage migriert, d. h. sie erfährt logisch diesselben strukturellen Änderungen ΔF wie ihre ursprüngliche Vorlage V. Anschließend müssen an der Instanz ggf. noch Zustandsanpassungen vorgenommen werden, etwa wenn durch ΔF ein neuer Knoten vor einem aktuell ausführbaren, aber noch nicht ausgeführten Knoten eingefügt wird (vgl. Instanz 1 in Abb. 4). In diesem Fall wird der neue Knoten als ausführbar markiert und dessen Nachfolger auf den Initialzustand zurückgesetzt [18, 17].

Ablauf einer instanzspezifischen Änderung

Bei Anwendung einer Änderungstransaktion ΔG auf einer Prozessinstanz finden diesselben Überprüfungen statt, allerdings in anderer Reihenfolge. Da eine Instanz einen Zustand besitzt, wird sofort nach Feststellung der strukturellen Verträglichkeit auch die Verträglichkeit bezüglich des Zustands geprüft. Erst anschließend werden strukturelle Änderungen und ggf. eine Zustandsanpassung durchgeführt (vgl. Instanz n in Abb. 4). Die strukturellen Änderungen führen dann zu einer Vorlage V^*, die nur für die veränderte Instanz gilt. Daneben werden alle Überprüfungen bei instanzspezifischen Änderungen für jede Änderungsoperation sofort durchlaufen. Dies erlaubt, bei jeder Änderungsoperation sofort sowohl auf Struktur- als auch auf Zustandsverträglichkeit zu überprüfen und ggf. die Änderungsoperation sofort zurückzuweisen. Bei einer Schemaevolution dagegen wird jede Überprüfungsphase jeweils für eine Menge von Änderungsoperationen (also die entsprechende Änderungstransaktion) ausgeführt, bevor die nächste Überprüfungsphase stattfindet. Dies liegt daran, dass die Änderungen auf einer Vorlage eingebracht werden und dort nur strukturelle Verträglichkeit geprüft werden kann; Zustandsverträglichkeit lässt sich erst bei der Propagation auf betroffene Instanzen testen.

Kombination von Schemaevolution und instanzspezifischen Änderungen

Lässt man sowohl Schemaevolution als auch instanzspezifische Änderungen zu, wie dies im ADEPT2-System der Fall ist, können auch bereits veränderte Prozessinstanzen von einer Sche-

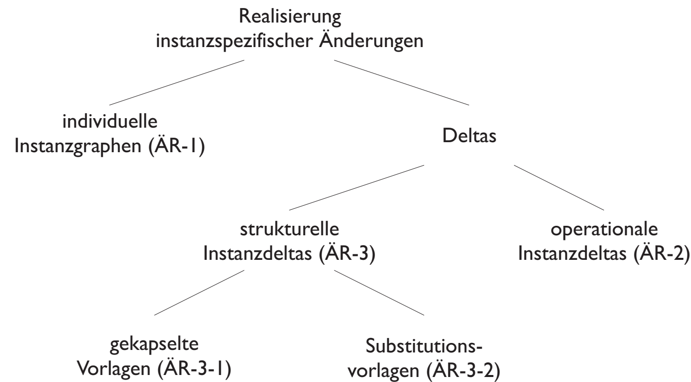

Abbildung 5: Klassifikation der Realisierungskonzepte instanzspezifischer Änderungen

maevolution betroffen sein [23, 20, 25]. In diesem Fall müssen die Änderungen auf der Vorlage sowie die instanzspezifischen Änderungen miteinander verglichen werden. So ist es beispielsweise denkbar, dass die instanzspezifischen Änderungen genau den Änderungen der Schemaevolution entsprechen, weil bei der Instanz diese Änderungen bereits vorweggenommen wurden. Änderungen auf Prozessvorlagen und -instanzen können auch echte Teilmengen voneinander sein oder sich überlappen [23]. In allen Fällen ist es das Ziel, die Instanz auf die neue Prozessvorlage zu migrieren und die instanzspezifischen Änderungen entsprechend anzupassen. Durch diese Anpassung wird eine neue Menge an instanzspezifischen Änderungen erzeugt, die nicht mehr auf der ursprünglichen Prozessvorlage sondern auf der geänderten Prozessvorlage beruhen.

Für die interne Repräsentation von Änderungen ist somit neben der Unterscheidung zwischen Änderungen an Prozessvorlagen sowie Änderungen an einzelnen Prozessinstanzen auch die Vergleichbarkeit beider Änderungsarten wichtig. Andernfalls ist es nicht möglich, diese in Kombination miteinander durch ein PMS zu unterstützen.

3 Realisierungskonzepte für instanzspezifische Änderungen

In Bezug auf die interne Repräsentation der einer bestimmten Instanz zugrundeliegenden Prozessstruktur, unterscheiden wir zwischen Vorlagenkopie (Variante VG-1) und Vorlagenreferenz (Variante VG-2) [10]. Bei der Vorlagenkopie werden intern für jede Prozessinstanz eigene Datenstrukturen für die Abbildung der jeweiligen Prozessstruktur angelegt, während diese bei Vorlagenreferenzen nur einmal pro Vorlage existieren und von den entsprechenden Prozessinstanzen ledigilch referenziert werden. Offensichtlich benötigen Vorlagenreferenzen wesentlich weniger Speicher und sind deshalb für die Realisierung der Datenstrukturen für Prozessvorlagen und -instanzen vorzuziehen. Mit dieser Repräsentationsform lassen sich allerdings instanzspezifische Änderungen nicht direkt realisieren. Diese Art von Änderungen müssen auf einer Prozessvorlage durchgeführt werden, die nur für die jeweils abzuändernde Prozessinstanz gilt (*instanzspezifische Prozessvorlage*). In den folgenden Abschnitten stellen wir alternative Ansätze dafür vor, wie auf Grundlage von Vorlagenreferenzen und sinnvollen Erweiterungen instanzspezifische Änderungen realisiert werden können.

Die einfachste Realisierungsvariante für die Repräsentation instanzspezifischer Änderungen ist die Verwendung individueller Instanzgraphen (Variante ÄR-1). Diese entsprechen einer normalen Vorlagenkopie (Variante VG-1), allerdings besitzt eine Instanz nur dann einen individuellen, von einer Vorlage abgeleiteten Instanzgraph (und keine einfache Referenz wie bei Variante VG-2), wenn die Prozessinstanz Änderungen unterworfen worden ist. Der individuelle Instanzgraph selbst repräsentiert eine vollständige Prozessvorlage wie bei Variante VG-1.

Alternativ lassen sich instanzspezifische Änderungen auch mit einer Vorlagenreferenz (Variante VG-1) realisieren, indem neben der Referenz die Unterschiede (Deltas) zwischen der durch die Änderungen resultierenden instanzspezifischen Vorlage und der ursprünglichen Vorlage gespeichert

werden. Dies führt dann nicht zu einer vollständigen (geänderten) Prozessvorlage und somit nicht zu einem explizit gespeicherten individuellen Instanzgraphen. Werden die zusätzlich zur Vorlagenreferenz existierenden Änderungen geeignet repräsentiert, kann die resultierende instanzspezifische Vorlage bei Bedarf erzeugt werden.

Die Realisierung der Deltas wiederum kann auf zwei Arten erfolgen: Die instanzspezifisch angewandten Änderungsoperationen können direkt als Operationen (z. B. als Änderungshistorie) repräsentiert werden (Variante ÄR-2, sog. operationale Instanzdeltas), oder es werden die aus den Änderungsoperationen resultierenden Graphstrukturen gespeichert (Variante ÄR-3, sog. strukturelle Instanzdeltas). Die geänderten Graphstrukturen werden intern durch Mengen von hinzugefügten, geänderten (hierzu zählen auch verschobene Knoten) und gelöschten Knoten und Kanten repräsentiert. Auch hier kann man wieder zwei Realisierungsarten unterscheiden: Gekapselte Vorlagen (Variante ÄR-3-1), die implizit und transparent zwischen geänderten Prozessstrukturen und der ursprünglichen Vorlage unterscheiden, sowie Substitutionsvorlagen (Variante ÄR-3-2), bei denen explizit zwischen geänderten Graphstrukturen und ursprünglicher Vorlage unterschieden wird (Abb. 5).

In diesem Kapitel werden die Varianten ÄR-1, ÄR-2 und ÄR-3 im Detail vorgestellt und miteinander verglichen. Wie erwähnt, entsprechen die Anforderungen an diese Varianten im Kern wieder den Anforderungen an Repräsentationen unveränderter Instanzen (vgl. [10]): niedriger Speicherbedarf, horizontale und vertikale Partitionierbarkeit, weitgehende Transparenz für Algorithmen, instanzübergreifende Anfragen (Sekundärspeicher) und Unterstützung von Schemaevolution [20]. Die Varianten ÄR-3-1 und ÄR-3-2 für die Realisierung von strukturellen Instanzdeltas werden in Kapitel 4 ausführlich diskutiert.

3.1 Variante ÄR-1: Individuelle Instanzgraphen

Eine naheliegende Lösung zur Realisierung von Instanzänderungen bieten individuelle Instanzgraphen. Dabei wird für die Durchführung von Änderungen eine vollständige Kopie der zugrundeliegenden Vorlage angelegt (vgl. [10], Variante VG-1), auf der dann die entsprechenden Änderungen auch durchgeführt werden. Die Instanz referenziert anschließend das neu erstellte, instanzspezifische Vorlagenobjekt (Abb. 6). Im Vergleich zu Vorlagenkopien (bei unveränderten Instanzen [10]) wird hier jedoch nur im Fall von instanzspezifischen Änderungen eine Vorlagenkopie angelegt, nicht dagegen für unveränderte Instanzen. Letztere referenzieren jeweils die ursprüngliche Prozessvorlage.

Zugriffe auf die internen Datenstrukturen erfolgen bei unveränderten als auch bei veränderten Instanzen gleich, d. h. in beiden Fällen wird bei Zugriffen auf strukturelle Informationen die Referenz auf die für die Instanz gültige Vorlage aufgelöst und die Anfrage entsprechend auf der Vorlage verarbeitet. Hierdurch ist eine vollständige Transparenz von Instanzänderungen gegeben.

Individuelle Instanzgraphen werden intern wie vollwertige Vorlagen verwaltet. Das Anlegen eines individuellen Instanzgraphen ist weitgehend identisch mit dem Ändern einer Prozessvorlage vor einer Schemaevolution, d. h. die Vorlage wird geklont und anschließend der Klon direkt strukturell geändert. Im Gegensatz zu einer Prozessvorlage kann ein individuelle Instanzgraph jedoch nicht (neu) instanziert werden, und es gibt exakt eine darauf laufende Instanz.

Problematisch an diesem Konzept ist die Verwendung der gleichen Konstrukte, nämlich Vorlagenobjekte, sowohl für Instanz- als auch für Vorlagenänderungen. Ein Zusammentreffen beider Änderungsarten, also die Änderung der ursprünglichen Vorlage einer individuell geänderten Instanz, lässt sich dabei nur sehr schwer realisieren. Beim Umhängen (verträglicher) laufender Instanzen auf eine neue Vorlage[1], werden geänderte Instanzen nicht berücksichtigt, da sie nicht mehr die ursprüngliche Vorlage sondern nunmehr einen individuellen Instanzgraphen referenzieren. Solche Instanzen können nur migriert werden, indem die Vorlagenänderungen der Schemaevolution auch auf dem individuellen Instanzgraphen durchgeführt werden, oder indem dieser komplett verworfen und, basierend auf einer Kopie der neuen Vorlage, entsprechend neu aufgebaut wird.

[1] Gemeint ist die Migration von Instanzen von einer alten Prozessvorlage auf eine neue Prozessvorlage bei Schemaevolution [24].

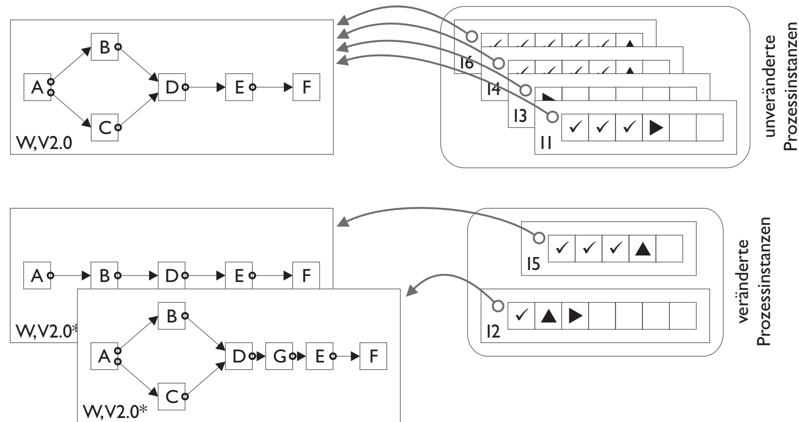

Abbildung 6: Änderungen repräsentiert durch individuelle Instanzgraphen (ÄR-1)

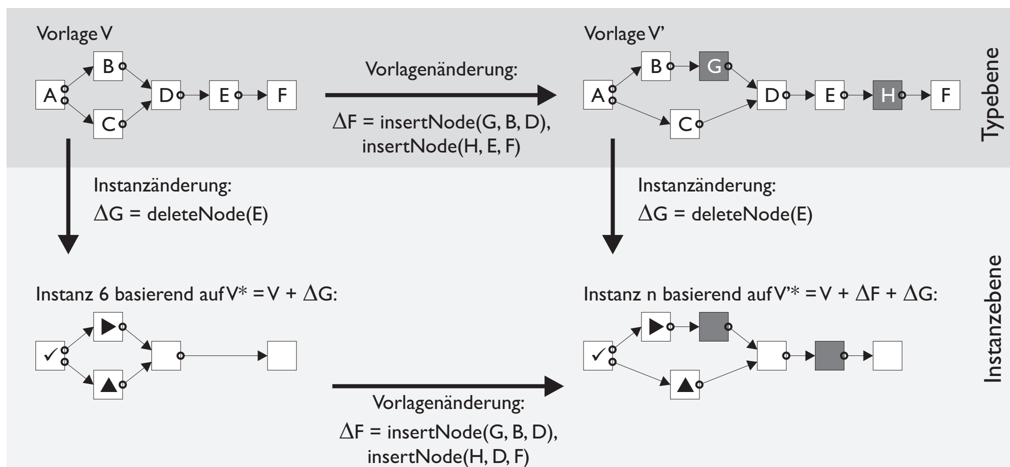

Abbildung 7: Schemaevolution mit ind. Instanzgraphen und angepasstem Änderungskontext

Beide Varianten zur Durchführung der Schemaevolution bei vorhandenen individuellen Instanzgraphen sind sehr aufwendig, da in jedem Fall Änderungen wiederholt werden müssen: dies betrifft entweder die Änderungen zum Zeitpunkt der instanzspezifischen Änderung oder die Vorlagenänderungen (Abb. 7). Dies ist besonders kritisch, da für die korrekte Durchführung der Migration einer geänderten Instanz in jedem Fall die Vorlagen- und Instanzänderungen zuvor noch auf Überlappungen und komplementäre Änderungen verglichen werden müssen. Dazu wird eine entsprechende Änderungshistorie benötigt, da die zu wiederholenden Änderungsoperationen entsprechend angepasst werden müssen: Beispielsweise kann ein Knoten auf Instanzebene gelöscht sein, hinter dem auf Vorlagenebene eine Änderung vorgenommen wird. Damit existiert der Kontext der entsprechenden Änderung auf Vorlagenebene nicht mehr auf Instanzebene. Dies erfordert die Neuermittlung des Kontexts für die Änderungsoperation auf Instanzebene. Beim Zusammenführen der Änderungen auf Vorlagen- und Instanzebene können Änderungsoperationen auch komplett entfallen, etwa wenn auf Instanzebene vorhandene Änderungen auch auf der Vorlage durchgeführt worden sind.

Beispielsweise wurde in Instanz 6 in Abb. 7 der Knoten E gelöscht. Wird auf Vorlagenebene ein Knoten H zwischen Knoten E und Knoten F eingefügt, so muss bei Propagation dieser Vorlagenänderungen auf Instanz n der Änderungskontext angepasst werden. Aus dem Einfügen von Knoten H zwischen den Knoten E und F wird ein Einfügen des Knotens H zwischen Knoten D (dem Vorgängerknoten von E) und Knoten F.

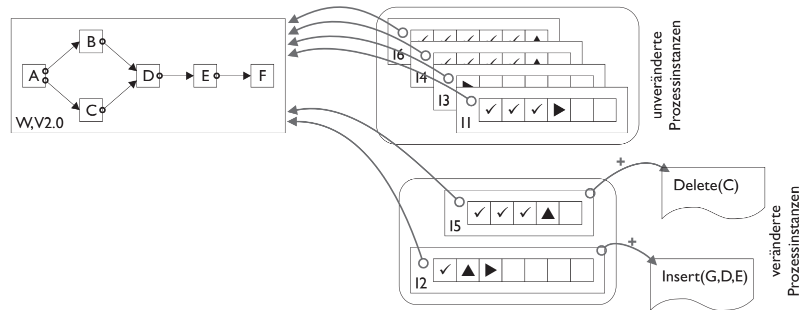

Abbildung 8: Änderungen mittels operationaler Instanzdeltas (ÄR-2)

Insgesamt sind individuelle Instanzgraphen sehr einfach in der Realisierung und bieten trotzdem eine vollständige Transparenz von Instanzänderungen. Es wird jedoch relativ viel Speicherplatz benötigt, da die Erstellung vollständiger Vorlagenkopien auch Redundanzen bezüglich der Vorlageninformationen mit sich bringt. Noch schwerer wiegt jedoch der sehr große Laufzeitaufwand für die Durchführung einer Schemaevolution für geänderte Instanzen, da beide Änderungsarten intern gleich realisiert werden und kein direkter Vergleich der Änderungen möglich ist.

3.2 Variante ÄR-2: Operationale Instanzdeltas

Operationale Instanzdeltas unterscheiden sich in zwei Aspekten von individuellen Instanzgraphen: Einerseits werden nur Unterschiede gegenüber der ursprünglichen Vorlage verwaltet (Deltas). Dadurch kann auf die ursprüngliche Vorlage nicht verzichtet werden, es bleibt aber der Bezug zu ihr bestehen. Andererseits werden Instanzänderungen nicht als Graphstrukturen, sondern als Änderungsoperationen gespeichert. Dies entspricht (einem Teil) der Änderungshistorie (Abb. 8).

Gegenüber individuellen Instanzgraphen ist die Durchführung von Instanzänderungen mit operationalen Instanzdeltas sehr effizient: Neben der für zahlreiche Funktionen benötigten Änderungshistorie müssen keine zusätzlichen Datenstrukturen erzeugt und verwaltet werden. Dafür müssen vor Zugriffen auf geänderte Instanzen erst die entsprechenden Änderungen durchgeführt werden. Dies kann beispielsweise auf einer temporären Kopie der ursprünglichen Vorlage erfolgen, wodurch Auswirkungen auf andere, unveränderte Instanzen verhindert werden. Dies ist mit Variante ÄR-1 vergleichbar, wobei der individuelle Instanzgraph nur bei Bedarf erzeugt wird, was insgesamt jedoch noch immer sehr aufwendig sein kann (Abb. 9).

Große Vorteile besitzt die Realisierung operationaler Instanzdeltas im Zusammenhang mit der Realisierung von Schemaevolution. Da die Instanzänderungen direkt in Form von Operationen vorliegen, kann der Vergleich mit den entsprechenden Vorlagenänderungen sehr effizient erfolgen. Dabei können bei Bedarf relativ einfach die Kontexte von Änderungsoperationen in den operationalen Instanzdeltas geändert oder ganze Änderungsoperationen entfernt werden. Anschließend können die geänderten Instanzen ohne zusätzliche strukturellen Anpassungen auf die neue Vorlage umgehängt werden; die Instanzdeltas wurden bereits bei dem Vergleich entsprechend angepasst.
Während operationale Instanzdeltas sehr effizient Instanz- und Vorlagenänderungen ermöglichen, gestalten sich Zugriffe auf die zugrundeliegenden Prozessstrukturen problematisch. Dabei muss jedesmal die resultierende instanzspezifische Vorlage erzeugt werden, um benötigte Informationen (z. B. Vorgänger/Nachfolger eines Knotens) ermitteln zu können, da diese Information nicht direkt aus der Änderungshistorie ermittelt werden kann bzw. nur mit einem sehr hohen Aufwand (Abb. 9). Ohne diesen gravierenden Nachteil wären operationale Instanzdeltas sehr gut für die Realisierung instanzspezifischer Änderungen geeignet. Sie benötigen zudem keinerlei zusätzlichen Speicherplatz, da eine Änderungshistorie zu Dokumentationszwecken in jedem Fall erforderlich ist.

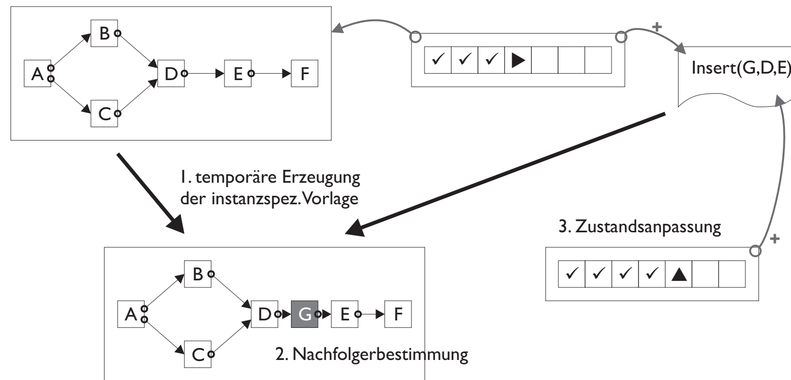

Abbildung 9: Weiterschalten mit Rekonstruktion der instanzspezifischen Vorlage

3.3 Variante ÄR-3: Strukturelle Instanzdeltas

Strukturelle Instanzdeltas bewegen sich zwischen individuellen Instanzgraphen und operationalen Instanzdeltas. Dabei werden wie bei operationalen Instanzdeltas nur Instanzänderungen und keine vollständigen Vorlagen verwaltet. Änderungen liegen jedoch nicht operational sondern wie bei invidiuellen Instanzgraphen als realisierte Graphstrukturen vor. Dazu wird bei Instanzänderungen der von der Änderung betroffene Bereich (Änderungskontext) der ursprünglichen Vorlage kopiert und anschließend geändert. Bei Zugriffen substituiert diese Kopie (das *Subsitutionsobjekt*) den entsprechenden Bereich der ursprünglichen Vorlage.

In Abb. 10 ist bei Instanz I5 der Knoten C gelöscht worden. Da hierbei eine Verzweigung aufgelöst worden, müssen durch das Löschen der Knotentyp der Knoten A und D geändert werden, wodurch der Änderungskontext die Knoten A und C sowie durch Wegfall der Parallelität auch Knoten B umfasst. Dieser Änderungskontext wurde entsprechend der Änderungsoperation manipuliert, d. h. Knoten C wurde gelöscht, und der Kontext bildet nun das Substitutionsobjekt. Durch die Überlagerung des Substitutionsobjekts auf die entsprechenden Knoten der ursprünglichen Vorlage entsteht die für I5 gültige instanzspezifische Vorlage.

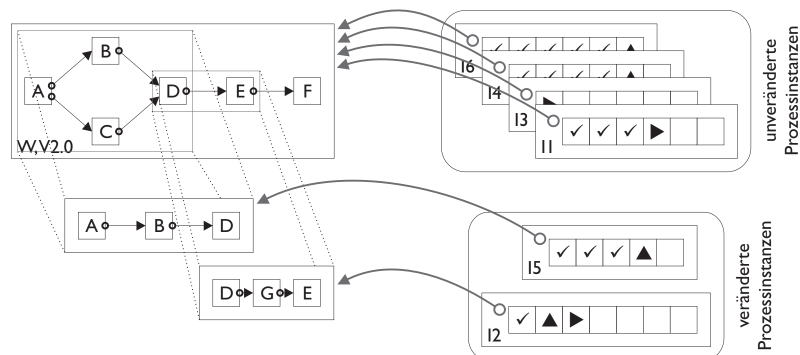

Abbildung 10: Änderungen mittels strukturellen Instanzdeltas (ÄR-3)

Damit vereinen strukturelle Instanzdeltas die Vorteile individueller Instanzgraphen und operationaler Instanzdeltas. Einerseits kann die instanzspezifische Vorlage bei Bedarf schnell erzeugt werden und ermöglicht anschließend direkte Zugriffe auf die Graphstrukturen der instanzspezifischen Vorlage. Andererseits ist der Speicherbedarf sehr gering, da gegenüber der ursprünglichen Vorlage keine redundanten Informationen verwaltet werden.

Konzeptionell sind zwei Arten der Substitution möglich: *einfache* und *hierarchische* (also mehrfache) *Substitution* (Abb. 11). Bei der einfachen Substitution existiert für eine geänderte Instanz jeweils nur ein Substitutionsobjekt, auch wenn die entsprechende Instanz mehrmals geändert wor-

den ist. Dagegen führt bei hierarchischer Substitution jede Änderungstransaktion zu einem eigenen Substitutionsobjekt, das die vorherigen überlagert. In Abb. 11 wurde Knoten C gelöscht und ein Knoten G zwischen Knoten D und E hinzugefügt. Links sieht man die resultierende einfache Substitution, rechts die hierarchische Substitution. Die resultierende instanzspezifische Vorlage ergibt sich jeweils aus der Überlagerung aller Substitutionsobjekte in der Reihenfolge ihrer Erzeugung.

Hierarchische Subsitution hat den Vorteil, dass die Substitutionsobjekte zum Zeitpunkt der Änderung sehr schnell erzeugt werden können. Dabei ist irrelevant, ob bereits Substitutionsobjekte exisitieren oder nicht. Eine Änderung beruht immer auf der zum Änderungszeitpunkt gültigen (logischen) Vorlage einer Instanz. Dies führt jedoch potentiell zu einem höheren Speicherbedarf, da ggf. redundante Informationen verwaltet werden. Dies ist beispielsweise der Fall, wenn durch komplementäre Änderungen vorherige Änderungen aufgehoben werden oder ein Änderungskontext für mehrere Substitutionsobjekte gilt (Knoten D in Abb. 11). In so einem Fall ist ein konsolidiertes Substitutionsobjekt speichereffizienter.

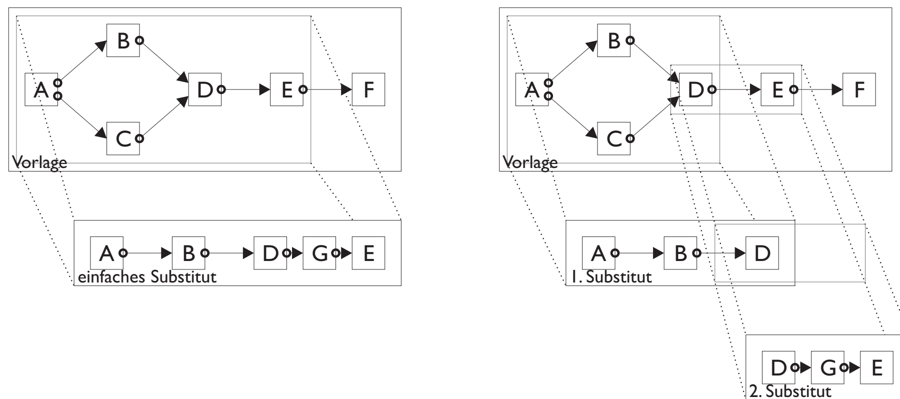

Abbildung 11: Einfache und hierarchische Substitution bei strukturellen Instanzdeltas

Genau dies ist die Funktionsweise der einfachen Substitution. Hierbei wird bei einer Änderung unterschieden, ob sie sich auf eine bereits vorhandene Änderung bezieht und dann zu einer Modifikation des Substitutionsobjekts führt, oder ob eine Änderung einen unveränderten Bereich der ursprünglichen Vorlage betrifft und das Substitutionsobjekt entsprechend erweitert wird. Damit ist die Durchführung von Änderungen aufwendiger als bei hierarchischer Substitution, es wird jedoch weniger Speicher benötigt und der Zugriff auf die instanzspezifische Vorlage ist wesentlich effizienter, da nicht mehrere Subsitutionsobjekte unterschieden und überlagert werden müssen. Insgesamt ist die einfache Substitution deshalb vorzuziehen.

3.4 Qualitativer Vergleich der Realisierungskonzepte

Tabelle 1 zeigt einen qualitativen Vergleich der vorgestellten Realisierungskonzepte bezüglich verschiedener Anforderungen. Am kritischsten sind der Primärspeicherbedarf sowie die Laufzeit der verschiedenen Realisierungen. Beide Aspekte werden wir in Kapitel 5 noch detailliert anhand von Messungen untersuchen.

Der Primärspeicherbedarf bei Variante ÄR-1 ist insgesamt sehr hoch. Insbesondere bei geringfügigen Änderungen, etwa beim Löschen eines einzelnen Knotens, wird viel redundante Information im Vergleich zur ursprünglichen Prozessvorlage gespeichert. Diesen Nachteil vermeiden operationale und strukturelle Instanzdeltas, d. h. hier wird signifikant weniger Speicherplatz benötigt. Weitgehend umgekehrt verhält es sich bezüglich der Laufzeit. Variante ÄR-2 ist ineffizient, da die Änderungen nicht strukturell vorliegen und erst rekonstruiert werden müssen. Der Aufwand hängt dabei von der Art und der Komplexität der Änderungen ab. Einfache Änderungen, wie das Löschen eines Knotens, sind noch relativ effizient handhabbar. Jedoch wird das Laufzeitverhalten bei mehreren komplexen Änderungen schnell problematisch. Wesentlich besser verhält sich dagegeben

Tabelle 1: Bewertung der Realisierungskonzepte instanzspezifischer Änderungen

	ÄR-1 Individuelle Instanzgraphen	ÄR-2 Operationale Instanzdeltas	ÄR-3 Strukturelle Instanzdeltas
Primärspeicherbedarf	-	+	+
Laufzeit	++	--	+
Transparenz	++	-	O
Partitionierung	+	-	O
Schemaevolution	-	++	O
Anfragen an Kollektionen von Instanzen	++	-	+

Unterstützung:
++: sehr gut
+: gut
O: neutral
-: schlecht
--: sehr schlecht

Variante ÄR-3, hier liegen die Änderungen schon in Graphstrukturen vor. Es muss bei Zugriffen lediglich noch eine Substitution vorgenommen werden. Dagegen unterscheidet sich der Zugriff auf instanzspezifische Vorlagen bei Variante ÄR-1 nicht vom Zugriff auf Vorlagen unveränderter Prozessinstanzen. Dies ist bezüglich der Laufzeit am effizientesten.

Auch die weiteren untersuchten Anforderungen werden von den Realisierungskonzepten sehr unterschiedlich unterstützt. Bei Variante ÄR-1 sind Änderungen vollständig transparent und deshalb mit der Realisierung unveränderter Instanzvorlagen vergleichbar. Sie bietet daher bezüglich Transparenz, Partitionierung und Anfragen an Kollektionen von Instanzen gute bis sehr gute Unterstützung (vgl. [10]). Dagegen ist eine Partitionierung von Variante ÄR-2 schwierig, da aus der Operation selbst nicht sofort hervorgeht, welche Bereiche von ihrer Anwendung betroffen sind. Bei Variante ÄR-3 ist der Änderungskontext dagegen bekannt, die Substitutionsobjekte bilden jedoch nicht unbedingt einen zusammenhängenden Prozessgraphen, weshalb sich die Partitionierung etwas aufwendiger als bei unveränderten Vorlagen gestaltet. Ähnlich verhält es sich bei Anfragen an Kollektionen von Instanzen. Diese sind bei Variante ÄR-2 sehr schwierig in der Umsetzung, da die Auswirkungen einzelner Operationen nicht absehbar sind. Bei Variante ÄR-3 muss für Anfragen die Substitution nur vorgenommen werden, wenn sich die Anfrage auf einen geänderten Bereich einer Vorlage bezieht.

Große Auswirkungen hat die Wahl des Realisierungskonzepts für instanzspezifische Änderungen auf die Durchführung einer Schemaevolution. Hierfür ist Variante ÄR-2 am besten geeignet, da der Vergleich der Vorlagen- und Instanzänderungen direkt erfolgen kann. Wird anschließend die Migration durchgeführt, müssen die instanzspezifischen Änderungen (ebenso wie die entsprechende Änderungshistorie) allerdings angepasst werden, da sie dann relativ zur neuen Prozessvorlage benötigt werden. Da Variante ÄR-2 direkt über die Änderungshistorie realisiert ist, müssen neben der Änderungshistorie keine weiteren Datenstrukturen für die Repräsentation der instanzspezifischen Änderungen angepasst werden. Dagegen sind sowohl bei Variante ÄR-1 als auch bei Variante ÄR-3 zusätzlich zur Manipulation der Historie strukturelle Anpassungen nötig. Diese gestalten sich bei strukturellen Deltas (ÄR-3) jedoch einfacher, da die enstprechenden Datenstrukturen direkt die Unterschiede zwischen der ursprünglichen Vorlage und der instanzspezifischen Vorlage repräsentieren, während bei Variante ÄR-1 eine geänderte Instanz unabhängig von der ursprünglichen Vorlage ist. Durch diese Unabhängigkeit können die Unterschiede nur durch einen Vergleich der vollständigen Objekte abgeleitet werden.

Im folgenden betrachten wir vor allem Variante ÄR-3 (strukturelle Instanzdeltas) näher, da diese qualitativ einen guten Kompromiss darstellt. Insbesondere der Primärspeicherbedarf und die relativ geringe Laufzeit überzeugen.

4 Implementierungsvarianten

Nachdem wir im vorherigen Kapitel das Realisierungskonzept der strukturellen Instanzdeltas als gute Lösung für die Repräsentation instanzspezifischer Änderungen identifiziert haben, untersuchen und vergleichen wir in diesem Kapitel verschiedene Implementierungsvarianten hierzu. Konkret sind drei Möglichkeiten denkbar (vgl. Abb. 5, Abb. 12):

- *Temporäre Realisierung*: Vor einem Zugriff wird die vollständige logische Vorlage physisch realisiert, wodurch die Zugriffe (z. B. beim Weiterschalten im Prozess) wie bei unveränderten Instanzen funktionieren
- *Substitutionsvorlagen*: Hierbei wird vor jedem Zugriff auf einen Knoten oder eine Kante überprüft, ob das entsprechende Element ein Bestandteil der strukturellen Instanzdeltas ist oder nicht. Dem entsprechend findet der Zugriff dann auf den Instanzdeltas (eben den Substitutionsvorlagen) oder auf der ursprünglichen Vorlage statt.
- *Gekapselte Vorlagen*: Jeder Zugriff auf einen Knoten oder eine Kante wird sowohl auf der ursprünglichen als auch auf den strukturellen Instanzdeltas durchgeführt. Die Vereinigung beider Ergebnisse bildet anschließend das eigentliche Resultat des Zugriffs.

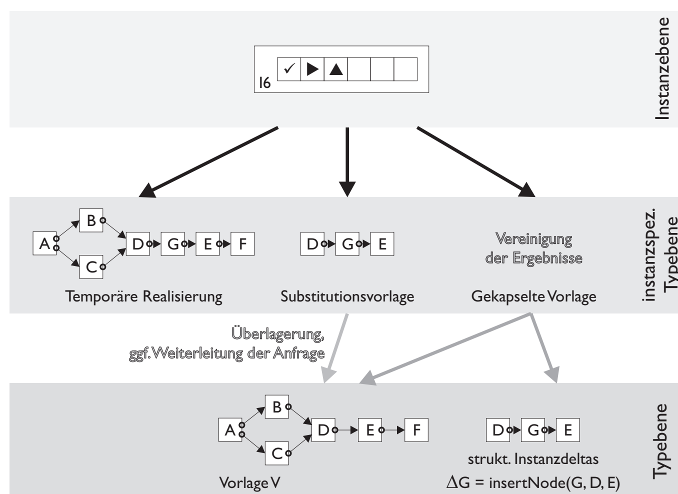

Abbildung 12: Ermittlung der Prozessstruktur bei den Implementierungsvarianten

Da bei Zugriffen nicht immer die gesamte Vorlage benötigt wird, sondern oftmals nur einzelne Teile des Prozessgraphen, ist die temporäre Realisierung der vollständigen Vorlage unnötig aufwendig, sowohl bezüglich Speicherbedarf als auch hinsichtlich der Laufzeit für die Rekonstruktion. Im folgenden werden wir deshalb nur Substitutionsvorlagen und gekapselte Vorlagen weiter betrachten und ihre Funktionsweise anhand repräsentativer Beispiele verdeutlichen. Abschließend folgt ein qualitativer Vergleich der Implementierungsvarianten. Ein quantitativer Vergleich schließt sich in Kapitel 5.

4.1 Variante ÄR-3-1: Substitutionsvorlagen

Implementierungsvariante ÄR-3-1 (Substitutionsvorlagen) ist dadurch gekennzeichnet, dass eine geänderte Instanz zwei zugrundeliegende Vorlagen besitzt und beide explizit referenziert: die ursprüngliche Vorlage sowie das Substitutionsobjekt (*Teilvorlage*) – bei einfacher Substitution existiert höchstens ein Subsitutitonsobjekt, bei hierarchischer Substitution wären es mehrere (vgl. Abb. 11). Zusätzlich ist bei der Instanz hinterlegt, welches Objekt für welchen Bereich bzw. welche Knoten gültig ist (Abb. 13). Da bei unveränderten Instanzen nur eine gültige Vorlage existiert, kann hier auf diese Information verzichtet werden.

Um die Funktionsweise von Substitutionsvorlagen zu verdeutlichen, werden wir im folgenden die konkrete Realisierung einer unveränderten Instanz, einer Instanz mit gelöschten Knoten, einer Instanz mit einem hinzugefügten Knoten sowie eine Instanz, die beide Änderungen der beiden anderen Instanzen enthält, erläutern. Der gelöschte Knoten ist dabei so gewählt, dass daraus eine komplexe Änderung resultiert, da durch das Löschen des Knotens ein leerer Zweig in einer Verzweigung entsteht. Da dies kein gültiges Prozessmodell darstellt, soll der leere Zweig im Rahmen

derselben Änderungsoperation gleich mitgelöscht werden. Dies verdeutlicht die Realisierung von Substitutionsvorlagen bei komplexen Änderungen. Der zweite Fall stellt eine einfache Änderung dar; dabei wird ein Knoten in einer Sequenz eingefügt. Im dritten Fall ist die Vereinigung der Änderungen aus Fall 1 und 2 und stellt eine komplexe Änderung dar, die sowohl die vorderen als auch die hinteren Teile der Prozessvorlage umfasst. Das Verschieben von Knoten wird nicht explizit betrachtet, da dies durch die Verknüpfung von Löschen und Hinzufügen realisiert wird. Die aus diesen Änderungen resultierenden vollständigen Vorlagen sind in Abb. 6 links unten zu sehen.

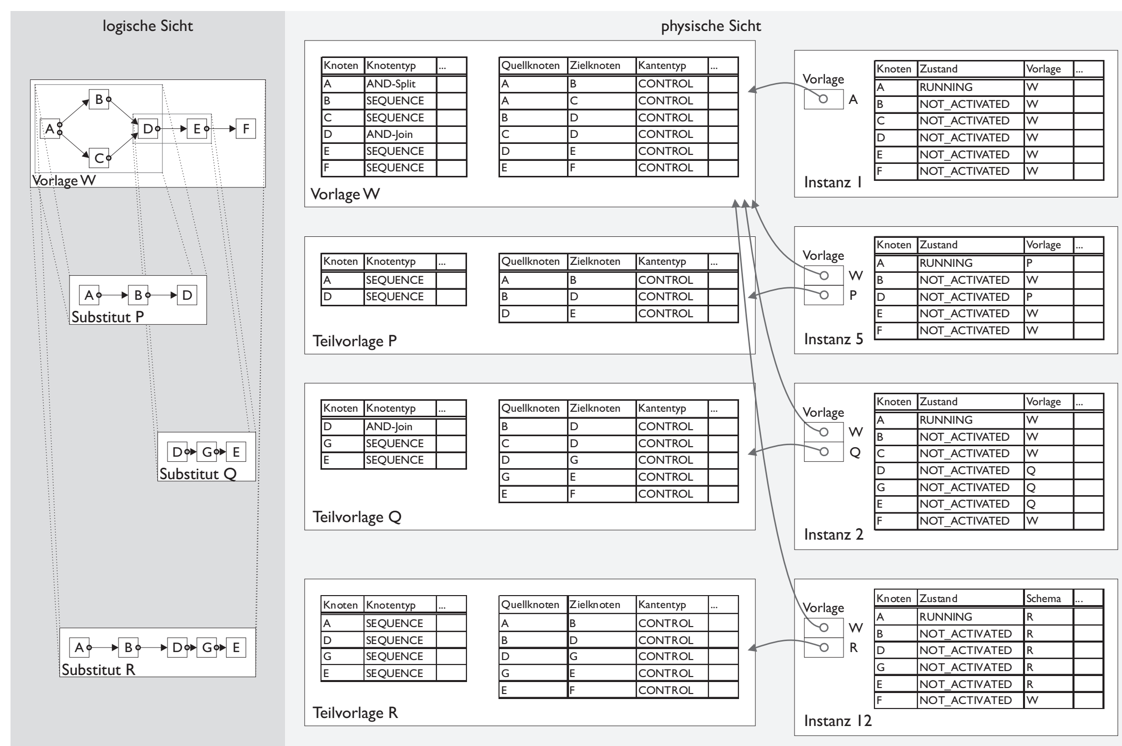

Abbildung 13: Realisierung von Substitutionsvorlagen (ÄR-3-1)

Abb. 13 zeigt die logische und die physische Sicht auf eine unveränderte Instanz (Instanz 1) und zwei geänderte Instanzen (Instanzen 5 und 2) sowie die entsprechenden Substitute (logische Sicht) bzw. (Teil-)Vorlagen (physische Sicht). In Instanz 5 wurde Knoten C zusammen mit dem entsprechenden Teilzweig einer Verzweigung gelöscht, was sich in Substitut P widerspiegelt (linke Seite von Abb. 13). Für die Realisierung dieser Änderung wurden alle betroffenen Knoten mit sämtlichen Attributen sowie alle ein-/ausgehenden Kanten von der ursprünglichen Vorlage auf eine neue Teilvorlage P kopiert und dort geändert. Ein Knoten ist von einer Änderung betroffen, wenn seine Attribute oder die Menge seiner ein-/ausgehenden Kanten modifiziert wird.

Beim Löschen von Knoten C entsteht ein leerer Zweig, der auch entfernt wird. Dadurch werden die Knoten A und D modifiziert. Diese ändern ihren Typ von Verzweigungs- bzw. Vereinigungsknoten zu einfachen Sequenzknoten[2]. Die Kanten A-B, B-D und D-E werden nicht modifiziert, sie werden aber bei Teilvorlage P benötigt, um den Übergang vom Substitut P zur usprünglichen Vorlage W zu realisieren (sog. *Kontextkanten*). Ohne die Kante D-E würde die für den Knoten D gültige Teilvorlage P keinen Nachfolger für D liefern. Das liegt daran, dass die Information über die jeweils gültige (Teil-)Vorlage in Instanzobjekten (rechte Spalte in Abb. 13) nur für Knoten existiert, nicht aber für Kanten.

Nach Erzeugung der Substitutionsvorlage wird noch das Instanzobjekt angepasst (Abb. 13,

[2]Dabei ist es unerheblich, ob es sich um parallele oder alternative Verzweigungen handelt.

rechts in der Mitte). Dazu wird Teilvorlage P als gültige Vorlage bei Knoten A und D hinterlegt. Knoten C wird aus der Zustandsliste gelöscht und ist damit nicht mehr erreichbar. Es existieren zwar noch die Kanten A-C und C-D in Vorlage W, diese werden jedoch nicht mehr berücksichtigt, da Zugriffe auf Knoten A und D ausschließlich über die Teilvorlage P erfolgen, die diese Kanten aber nicht besitzt.

Beim Hinzufügen eines neuen Knotens G bei Instanz 2 werden neben Knoten G genau die Knoten auf Teilvorlage Q kopiert, deren Attribute bzw. ein-/ausgehende Kanten modifziert werden. Dies sind im vorliegenden Fall Knoten D und F. Zusätzlich werden noch die zugehörigen Kanten (B-D, C-D, E-F) sowie die neu hinzugekommenen Kanten (D-G, G-E) übernommen. Analog zum Löschen von Knoten C bei Instanz 5 muss die Kante D-E der Vorlage W bei Instanz 2 nicht weiter berücksichtigt werden, da sie nach den entsprechenden Anpassungen im Instanzobjekt nicht mehr erreicht werden kann. Diese Anpassungen beschränken sich auf das Einfügen des Knotens G in die Zustandsliste sowie das Hinterlegen von Teilvorlage Q bei den Knoten D, G und E (Abb. 13, rechts unten).

Bei einem Zugriff auf eine geänderte Instanz muss vor jedem einzelnen Knotenzugriff auf die (logische) instanzspezifische Vorlage überprüft werden, welche (Teil-)Vorlage für den entsprechenden Knoten gültig ist. Beispielsweise verläuft die Bestimmung der transitiven Nachfolger von Knoten B in Instanz 2 folgendermaßen: Zuerst wird die für B gültige Vorlage bestimmt. In der Zustandstabelle ergibt sich hierfür Vorlage W (3. Spalte). In Vorlage W wiederum ergibt sich aufgrund der Kontrollkante B-D der Knoten D als Nachfolger von B. Für den Knoten D ergibt sich aus Instanz 2 Teilvorlage Q. Aus dieser wiederum wird aufgrund der Kontrollkante D-G der Knoten G als Nachfolger von D bestimmt. Auch für Knoten G gilt, wie in Instanz 2 hinterlegt, Teilvorlage Q. Hieraus lässt sich der Nachfolger E ermitteln. Über die Kontextkante E-F von Teilvorlage Q ergibt sich der Knoten F als Nachfolger von E. Für Knoten F gilt die ursprüngliche Vorlage W. Da dort Knoten F der Endknoten ist, gibt es keine weiteren Nachfolgerknoten mehr.

An diesem Beispiel wird deutlich, wie die Kontextkanten den Übergang zwischen Vorlage W und Teilvorlage Q ermöglichen, also die Kanten, die in einen geänderten Bereich ein- und ausgehen. Diese sind zu diesem Zweck sowohl in der ursprünglichen Vorlage als auch im Substitutionsobjekt hinterlegt.

Instanz 12 schließlich vereinigt das Löschen von Knoten C und das Hinzufügen von Knoten G. Wie aus Abb. 13 ersichtlich, gilt hier für die meisten Knoten die Teilvorlage R. Dies liegt daran, dass die ursprüngliche Vorlage relativ klein und damit vorderer und hinterer Teil der Prozessvorlage dicht beieinander liegen. Mit einer größeren Vorlage gälte für mittlere Knoten durchaus noch Vorlage W. Die Abbildung macht auch deutlich, dass die Knoten- und Kantenmengen in Teilvorlage R nicht ganz die Vereinigungen der Knoten- und Kantenmengen von Teilvorlage P und Teilvorlage Q sind. Knoten D ist von beiden Änderungen betroffen, da er Kontextknoten ist. Naheliegendeweise existiert er in Teilvorlage R aber nur einmal. Außerdem gibt es durch das Löschen von Knoten C in Teilvorlage R die Kontextkante C-D nicht, und auch die Kante D-E existiert nicht, da sie durch die Kante D-G ersetzt wird.kaum signifikante

4.2 Variante ÄR-3-2: Gekapselte Vorlagen

Bei Implementierungsvariante ÄR-3-2 (gekapselte Vorlagen) besitzt das Substitutionsobjekt (*Wrapper*) diesselbe Schnittstelle wie eine vollständige Vorlage. Dies entspricht den Software-Entwurfsmustern *Decorator* bzw. *Wrapper* [8]. Dadurch besteht für eine Instanz kein Unterschied zwischen Zugriffen auf Vorlage und Wrapper, was zu vollständiger Transparenz führt (Abb. 14). Die in einem Wrapper gekapselten instanzspezifischen Änderungen werden direkt bei Zugriffen berücksichtigt. Hierzu wird eine Anfrage sowohl vom Vorlagenobjekt als auch vom Wrapper bearbeitet und die beiden Teilergebnisse anschließend vereinigt.

Bei Durchführung instanzspezifischer Änderungen werden wie bei Implementierungsvariante ÄR-3-1 (Substitutionsvorlagen) alle Knoten mit Änderungen in das Substitutionsobjekt (in diesem Fall der Wrapper) übernommen. Bei gekapselten Vorlagen findet jedoch keine Substitution ganzer Graphbereiche statt, sondern es werden einzelne Knoten und Kanten ersetzt.

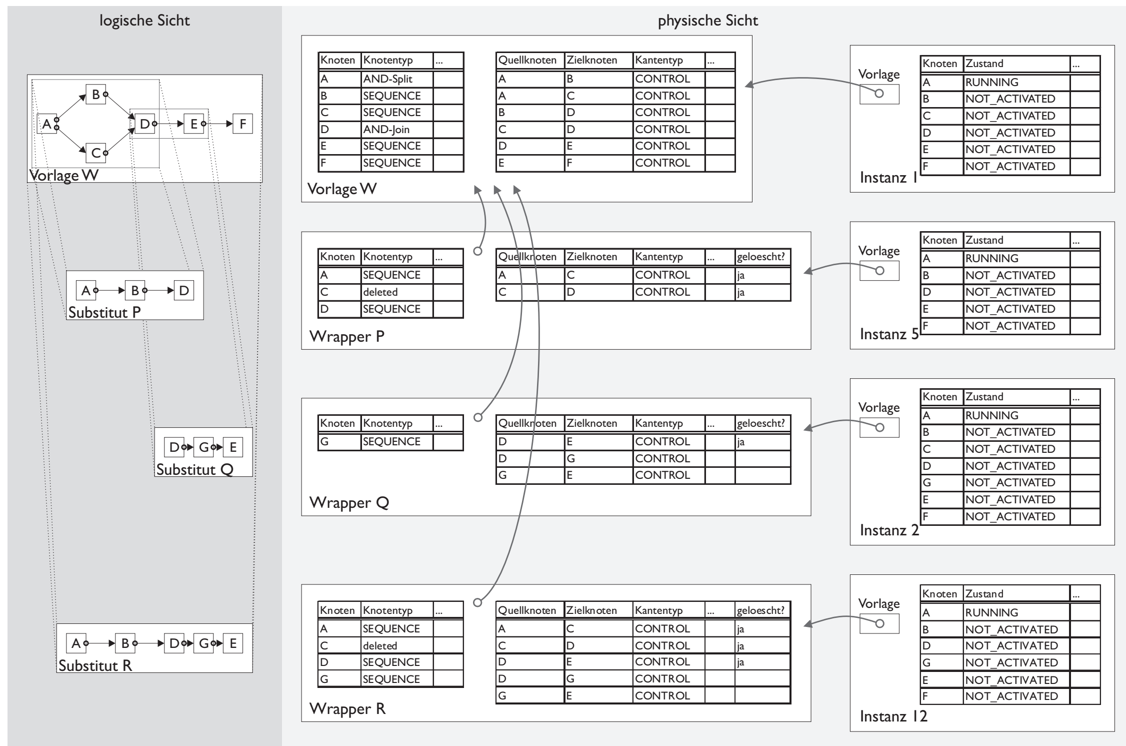

Abbildung 14: Realisierung von gekapselten Vorlagen (ÄR-3-2)

Beim Löschen von Knoten C in Instanz 5 werden die Knoten A und D in den Wrapper P übernommen und jeweils die Änderung des Knotentyps vermerkt (siehe resultierende Datenstrukturen in Abb. 14, Wrapper B) . Zusätzlich werden die gelöschten Knoten und Kanten (im Beispiel Knoten C und Kanten A-C und C-D) explizit verwaltet und damit auch in Wrapper B übernommen. Dafür gibt es im Gegensatz zu Variante ÄR-3-1 (Substitutionsvorlagen) jedoch keine Redundanzen zwischen ursprünglicher Vorlage und Wrapper (z. B. Kanten A-B und B-D), da keine Kontextkanten benötigt werden. In Instanz 5 wird abschließend noch Knoten C aus der Zustandstabelle entfernt (Abb. 14, rechts in der Mitte). Auch bei den Instanzobjekten wird bei Variante ÄR-3-2 weniger Speicherplatz benötigt als bei Implementierungsvariante ÄR-3-1, da nur eine Vorlage referenziert wird und bei Knoten keine Unterscheidung bezüglich der gültigen (Teil-)Vorlage benötigt wird. In Abb. 14 spiegelt sich dies im Fehlen der entsprechenden Spalte wider.

Das Hinzufügen eines Knotens ist bei gekapselten Vorlagen einfacher als bei Implementierungsvariante ÄR-3-1 (vgl. Abb. 13 und Abb. 14). Um den neuen Knoten G zwischen D und E in Instanz 2 einzufügen, wird er zusammen mit den ein-/ausgehenden Kanten (D-G, G-E) in Wrapper Q hinterlegt. Zusätzlich wird noch die ursprüngliche Kante D-E als gelöscht markiert. Die Änderung schließt mit der Aufnahme von G in die Zustandstabelle von Instanz 2 (Abb. 14, rechts unten).

Die Bestimmung der transitiven Nachfolger von Knoten B der Instanz 2 werden wie bei Implementierungsvariante ÄR-3-1 iterativ bestimmt. Dazu werden in jedem Schritt die (direkten) Nachfolger des jeweiligen Knotens sowohl mittels der ursprünglichen Vorlage (Vorlage W) als auch mittels Substitutionsobjekt (Wrapper Q) bestimmt und die einzelnen Ergebnisse entsprechend zusammengefasst. Aufgrund von Vorlage W und Kante B-D ergibt sich der Knoten D als Nachfolger von B, während Wrapper Q keine von B ausgehende Kontrollkante besitzt und somit auch keinen Nachfolgerknoten für B liefert. Aus beiden Informationen ergibt sich Knoten D als Nachfolger von B. Als Nachfolger von D wiederum liefert Vorlage W den Knoten E, während Wrapper Q den Knoten G als Nachfolger liefert und zusätzlich die Information, dass die Kante D-E gelöscht ist und somit Knoten E nicht mehr länger Nachfolger von D ist. Somit ergibt sich Knoten G als

alleiniger Nachfolger von D. Dessen Nachfolgerknoten wiederum ist der Vorlage W nicht bekannt, wohingegen Wrapper Q den Knoten E ermittelt, der damit der korrekte Nachfolger von G ist. Der Nachfolger von Knoten E wird wie der Nachfolger von Knoten B ermittelt: Vorlage W liefert Knoten F, Wrapper Q kennt keinen Nachfolgerknoten, womit F als (letzter) Nachfolger bestimmt ist.

Im Gegensatz zu ÄR-3-1 sind bei der Anwendung beider Änderungen bei Instanz 12 die Knoten- und Kantenmengen von Wrapper R die echte Vereinigung der Knoten- und Kantenmengen von Wrapper P und Wrapper Q. Dies liegt daran, dass bei gekapselten Vorlagen auch gelöschte Knoten und Kanten explizit vorgehalten werden, während diese bei ÄR-3-1 (Substitutionsvorlagen) gleich entsprechend entfernt werden. Letztere ist damit jedoch etwas aufwendiger zu implementieren. Wie in Abb. 14 ersichtlich, haben umfangreichere Änderungen auch hier keine Auswirkungen auf die Menge der zu verwalteten Daten für die Repräsentation von Änderungen. Das heißt, obwohl sowohl der vordere Teil als auch der hintere Teil der Prozessvorlage von Änderungen betroffen ist, wird die Repräsentation des (logischen) Substitutionsobjekts (in diesem Fall Substitut R) nicht größer als wenn die Änderungen einen zusammenhängenden Teil der Prozessvorlage beträfen.

Charakteristisch für Variante ÄR-3-2 ist, dass nicht eine nicht nur an die gültige (Teil-)Vorlage geht, sondern dass sie sowohl von der ursprüngliche Vorlage als auch vom Wrapper bearbeitet wird. Die ermittelten Ergebnisse werden anschließend entsprechend vereinigt, wobei das Ergebnis vom Wrapper höhere Priorität hat. Diese Vereinigung ist potentiell aufwendig, da es sich dabei um Mengenoperationen handelt. Allerdings sind die ermittelten Knotenmengen meistens nicht besonders groß: Bei Prozessvorlagen sind die meisten Knoten sequentiell angeordnet, was zu zweielementigen Mengen bei der Bestimmung direkter Vorgänger-/Nachfolgerknoten führt – ein Knoten von der ursprünglichen Vorlage und ein Knoten vom Wrapper. Lediglich bei Verzweigungs-/Vereinigungsknoten werden die Mengen größer, aber auch hier sind mehr als 2*4 Elemente in der Menge sehr selten. Dadurch bleiben die notwendigen Mengenoperationen effizient.

Insbesondere für sukzessive Zugriffe ist mit Variante ÄR-3-2 aber eine Verbesserung sehr leicht möglich, indem temporär die logische instanzspezifische Vorlage realisiert wird. Hierzu werden einfach die Vereinigung der gesamten Knoten- und Kantentabellen von Vorlage und Wrapper erzeugt und die im Wrapper als gelöscht markierten Objekte enfernt. Dadurch ergibt sich direkt auf Ebene der Datenstrukturen von Knoten- und Kantenmengen das resultierende vollständige logische Schema, ohne dass Änderungsoperationen aufwendig nachgespielt werden müssen.

4.3 Qualitativer Vergleich der Implementierungsvarianten

Sowohl Implementierungsvariante ÄR-3-1 (Substitutionsvorlagen) als auch Variante ÄR-3-2 (gekapselte Vorlagen) realisieren strukturelle Instanzdeltas. Es gibt jedoch signifikante Unterschiede bezüglich der verwalteten Information und dem Ablauf von Zugriffen. Dies äußert sich auch im Speicherbedarf und in der Zugriffszeit. So ist bei Variante ÄR-3-1 vor jedem Zugriff auf Vorlageninformationen jeweils ein zusätzlicher Zugriff auf die geänderte Instanz notwendig, um die gültige (Teil-)Vorlage zu bestimmen. Anschließend kann die gewünschte Information direkt bestimmt werden. Bei Variante ÄR-3-2 sind mehrere Zugriffe nötig, um die jeweilige Information in der Vorlage und im Wrapper zu bestimmen und um die so bestimmten Ergebnisse anschließend geeignet zusammenzufassen, also zusätzliche Knoten aus dem Wrapper der Ergebnismenge hinzuzufügen und im Wrapper als gelöscht markierte Knoten zu entfernen. Wie erwähnt, ist das Zusammenfassen jedoch nicht allzu zeitaufwendig, da die bei einem Wrapper-/Vorlagenzugriff bestimmten Knoten- und Kantenmengen nicht sehr groß sind. Dies liegt daran, dass Prozesse meistens eine eher sequentielle Struktur besitzen. Zudem kann die Situation durch horizontale Partitionierung weiter verbessert werden.

Implementierungsvariante ÄR-3-2 benötigt gegenüber Variante ÄR-3-1 weniger Speicherplatz, da nur die Änderungen der ursprünglichen Vorlage verwaltet werden und damit keine Redundanzen existieren wie etwa Kontextkanten. Es reicht sogar aus, nur die wirklich geänderten Attribute eines Knotens bzw. einer Kante zu speichern und nicht das komplette Knoten-/Kantenobjekt bereitzustellen. Das Vermerken von gelöschten Objekten bei Variante ÄR-3-2 erfordert zwar zu-

sätzlichen Speicherbedarf, jedoch ist dieser nur wenig höher als bei Variante ÄR-3-1 (vgl. Abb. 13, Teilvorlage P und Abb. 14, Wrapper P). Zudem wird die Information über gelöschte Objekte für den Vergleich von Vorlagenänderungen mit Instanzänderungen bei Schemaevolution benötigt. Bei Variante ÄR-3-1 muss diese erst rekonstruiert werden.

Für Implementierungsvariante ÄR-3-2 spricht auch die vollständige Transparenz und die einfache Erzeugung der logischen instanzspezifische Vorlage. Durch die Kapselung ist es sehr leicht möglich, bei großer Anzahl von Änderungen einen Wrapper durch einen vollständigen individuellen Instanzgraphen zu ersetzen (vgl. Realisierungskonzept ÄR-1, Abschnitt 3.1).

5 Quantitativer Vergleich

Zwecks quantitativer Vergleichbarkeit haben wir die vorgestellten Realisierungskonzepte für individuelle Instanzgraphen, operationale Instanzdeltas und strukturelle Instanzdeltas in den beiden Implementierungsvarianten Substitutionsvorlagen und gekapselte Vorlagen prototypisch implementiert. Anhand der vorgenommenen Implementierungen werden Messungen bezüglich Speicherbedarf und Laufzeit durchgeführt, wobei die Messungen von der Anzahl der Instanzänderungen abhängen.

Für den quantitativen Vergleich der Realisierungskonzepte ziehen wir Messungen im Rahmen dieser Arbeit rein theoretischen Betrachtungen vor. Einerseits sollen die Einflüsse der Laufzeitumgebung (etwa die laufzeitkritische Durchführung von Mengenoperationen) mit in die Messung eingehen, andererseits soll die Laufzeit möglichst konkret ermittelt werden und nicht nur abstrakt (etwa mittels O-Notation). Bevor wir die Messergebnisse vorstellen und interpretieren, erläutern wir zuerst die Testumgebung, in der wir die Messungen durchgeführt haben, sowie den Ablauf der Messungen.

5.1 Testumgebung

Die Implementierung der Realisierungskonzepte erfolgt in Java, die Messungen wurden auf einer Java-Laufzeitumgebung von Sun (32 Bit) durchgeführt. Als Basis-Datentypen für die Datenstrukturen werden `boolean` (8 Bit-Datentyp), `byte` (8 Bit-Datentyp), `short` (16 Bit-Datentyp), Referenzen (32 Bit-Datentyp), Zeichenketten (Unicode, jeweils 16 Bit pro Zeichen) sowie Arrays dieser Basis-Datentypen verwendet. Prozessvorlagen, geänderte Prozessvorlagen (d. h. individuelle Instanzgraphen, Teilvorlagen und Wrapper) und Prozessinstanzen sind jeweils als eigene Javaobjekte realisiert. Prozessvorlagen besitzen Arrays vom Datentyp `short` als Objektvariablen für Knotentypen, Kanten-Quellknoten, Kanten-Zielknoten und Kantentypen. Prozessinstanzen besitzen eine Referenz auf das entsprechende Prozessvorlagenobjekt sowie ein Array vom Datentyp `byte` für die Knotenzustände. Der Index der Arrays wird als Knoten-ID bzw. Kanten-ID interpretiert, der Inhalt an der Stelle im Array ist der entsprechende Attributwert.

Wird eine Prozessinstanz geändert, wird das referenzierte Prozessvorlagenobjekt durch ein eigenes Objekt des entsprechenden Realisierungskonzepts ersetzt. Dieses referenziert dann die ursprüngliche Vorlage und ist damit der Instanz und der ursprünglichen Vorlage zwischengeschaltet. Dies gilt nicht für ÄR-1 (individuelle Instanzgraphen), hier wird eine neues normales Vorlagenobjekt erzeugt, das die ursprüngliche Vorlage komplett ersetzt. Allerdings werden in der Implementierung hier zusätzlich die gelöschten Knoten mit im ersetzenden Vorlagenobjekt abgelegt. Dies ist für diese Änderungsrepräsentation eigentlich nicht notwendig, es erleichtert aber die Ermittlung der Unterschiede zwischen dem individuellen Instanzgraphen und der ursprünglichen Vorlage.

Bei ÄR-3-1 (Substitutionsvorlagen) wird ein Vorlagenobjekt erzeugt, das die Objektvariablen einer normalen Vorlage zur Aufnahme neuer und geänderter Prozessstrukturen besitzt und zusätzlich ein Array, welches das Vorlagenobjekt (entweder das Substitutionsobjekt `this` oder die ursprüngliche Vorlage) referenziert, das für einen bestimmten Knoten gültig ist. Das Objekt für gekapselte Vorlagen (ÄR-3-2) benötigt dieses Array nicht, und hat dementsprechend nur eine Referenz auf das ursprüngliche Vorlagenobjekt. Dafür besitzt diese Objekt neben den Arrays für die

hinzugefügten und geänderten Prozessstrukturen zwei Arrays vom Datentyp `boolean` für gelöschte Knoten und Kanten.

Bei der Implementierung von ÄR-2 (operationale Instanzdeltas) gibt es für jede durchgeführte Änderung ein eigenes Änderungsobjekt. Deren Objektvariablen entsprechen den für die jeweilige Änderungsoperation notwendigen Informationen, z. B. eine Knoten-ID vom Typ `short` für das Änderungsobjekt zum Löschen eines Knotens. Die Änderungsobjekte werden in einem Array bei der entsprechenden Instanz referenziert. Vor einem Zugriff auf die Prozessinstanz werden die Änderungsoperationen neu auf die ursprüngliche Vorlage angewandt und damit temporär ein individueller Instanzgraph erstellt. Dieser gilt dabei immer nur für eine Anfrage, z. B. transitive Vorgängerbestimmung oder eine Zustandsänderung eines Knotens. Das heißt, der individuelle Instanzgraph wird einmal für die komplette transitive Vorgängerbestimmung erzeugt, aber jedes Mal für eine Zustandsänderung eines Knotens.

5.2 Messungsablauf

Ausgangspunkt der Messungen ist eine unveränderte Instanz. Die entsprechende Prozessvorlage besitzt 100 Knoten, die sequentiell angeordnet sind. Hiervon wird der Speicherbedarf für die Datenstrukturen gemessen, die für die Repräsentation der instanzspezifischen Änderungen notwendig sind. Der Speicherplatz für die Instanzdaten (z. B. Zustandsinformation) sowie der von der ursprünglichen Vorlage belegte Speicher sind nicht in der Messung enthalten, da diese unabhängig von der gewählten Änderungsrepräsentation sind. Der Speicherbedarf wird absolut angegeben.

Außerdem wird die Zeit für die Bestimmung aller Vorgänger des letzten Knotens der Sequenz sowie die Zeit für das Weiterschalten eines Knotens ermittelt. Um Laufzeitunterschiede einzelner Messsungen zu minimieren, wird die Vorgängerbestimmung 10.000 Mal durchgeführt und alle Ergebnisse arithmetisch gemittelt. Auch das Weiterschalten wird 10.000 Mal durchgeführt. Allerdings wird zusätzlich jeder Knoten der Prozessinstanz weitergeschaltet, d. h. jeder Knoten der Instanz wird pro Durchlauf einmal „gestartet“ und einmal „beendet“. Das Ergebnis wird ansschließend durch die Anzahl der Knoten (100) dividiert, um somit den Mittelwert für das Weiterschalten eines Knotens dieser Instanz zu erhalten, bevor auch hier der Mittelwert über alle 10.000 gemessenen Zeiten gebildet wird. Die Implementierung der Prozessinstanz basiert auf Vorlagenreferenzen mit expizliten Markierungen [10]. Die transitive Vorgängerbestimmung ist ein komplexer Datenzugriff. Dieser erfolgt iterativ, wobei in jedem Schritt die jeweiligen Vorgänger eines Knotens ermittelt und der Ergebnismenge hinzugefügt werden. Dabei dürfen die Vorgänger jedes Knotens nur einmal ermittelt werden, um nicht in eine Endlosschleife zu geraten.

Die Zeitmessungen bei der Vorgängerbestimmung und beim Weiterschalten werden jeweils in Relation zur Laufzeit einer unveränderten Instanz gesetzt. Das heißt, ohne Änderungen ist der relative Zeitbedarf sohwohl für das Weiterschalten als auch für die transitive Vorgängerbestimmung jeweils 1.

Da die Messungen bezüglich der Anzahl an Änderungen ermittelt werden sollen, wird nach dem Durchlaufen aller Messungen wie eben beschrieben bei jedem Realisierungskonzept eine Instanzänderung durchgeführt. Diese beinhaltet das Hinzufügen eines Knotens in die Sequenz sowie das Entfernen eines bereits vorhandenen Knotens. Dadurch bleibt die Anzahl der Knoten der Prozessvorlage über alle Messungen konstant. Bei jedem Durchlauf „wandert“ jedoch ein Knoten von der ursprünglichen Prozessvorlage auf die geänderte Prozessvorlage. Dies erfolgt für jeden Knoten der ursprünglichen Sequenz einmal, so dass am Ende zwar diesselben Graphstrukturen wie zu Beginn des Tests existieren, diese jedoch ausschließlich über Änderungen erzeugt wurden und die ursprüngliche Vorlage komplett überlagert ist. Damit sind pro Realisierungskonzept 100 Messdurchläufe nötig. Hierdurch erhalten wir die gewünschten Messergebnisse pro Realisierungskonzept abhängig von der Anzahl an durchgeführen Änderungen.

Die Art der Änderung ist weitgehend vernachlässigbar. Lediglich bei ÄR-2 und bei ÄR-3-1 sind bei komplexeren Änderungsoperationen größere Änderungskontexte notwendig, so dass der Speicherbedarf etwas höher sein wird als beim vorliegenden Messungsablauf.

5.2.1 Speicherbedarf

Abb. 15 zeigt den absoluten Speicherbedarf der verschiedenen Implementierungen in Abhängigkeit von der Anzahl der vorgenommenen Instanzänderungen. Am Speicherbedarf von 0 Byte bei 0 Änderungen mit operationalen Instanzdeltas wird deutlich, dass sich der gemessene Speicherbedarf nur auf die Repräsentation von instanzspezifischen Änderungen bezieht.

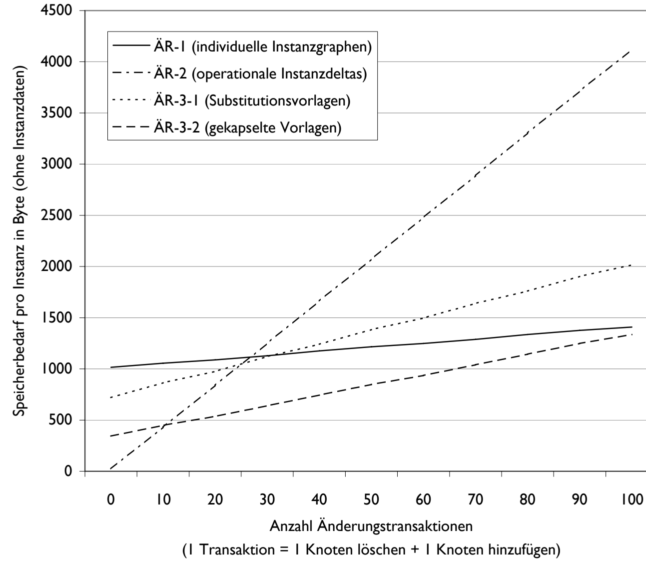

Abbildung 15: Speicherbedarf für Realisierungskonzepte

Sehr auffällig an der Messung ist der schnelle Anstieg bei Realisierungskonzept ÄR-2 (operationale Instanzdeltas), obwohl man intuitiv hierfür eigentlich mit weniger Speicherbedarf als bei den anderen Realisierungskonzepten rechnet. Das liegt in dieser Implementierung daran, dass für jede durchgeführte Änderungsoperation ein eigenes Änderungsobjekt angelegt wird, was relativ viel Speicher für Objektmetainformationen benötigt. Eine andere, potentiell bessere Realisierungmöglichkeit ist, Zeichenketten anstelle von Änderungsobjekten zu verwenden (z. B. „DeleteNode B“), vergleichbar einem Änderungsprotokoll. Dies führte jedoch in der Implementierung überraschenderweise zu einem noch schnelleren Anstieg des Speicherbedarfs. Der Grund hierfür könnte in der relativ aufwendigen Repräsentation von Zeichenketten in Java (2 Byte pro Zeichen) liegen. Deshalb ist Realisierungskonzept ÄR-2, wie in Abb. 15 zu sehen, nur bis zu einer mittleren Anzahl an Änderungen für die Repräsentation geeignet.

Realisierte Graphstrukturen (Realisierungskonzept ÄR-1, Implementierungsvarianten ÄR-3-1 und ÄR-3-2) benötigen ab einer mittleren Anzahl von Änderungen wesentlich weniger Speicherplatz als operationale Instanzdeltas: Implementierungsvariante ÄR-3-2 ist im vorliegenden Szenario ab 10 Änderungstransaktionen (d. h. 10 Einfügungen und 10 Löschungen) besser, ab 25 Änderungen ist Implementierungsvariante ÄR-3-1 und ab 30 Änderungen sogar Realisierungskonzept ÄR-1 (vollständige individuelle Instanzgraphen) besser als Implementierungsvariante ÄR-3-1. Der Speicherbedarf für die Implementierungsvarianten ÄR-3-1 (Substitutionsvorlagen) und ÄR-3-2 (gekapselte Vorlagen) besitzt eine vergleichbare Steigung. Die Implementierungsvariante ÄR-3-1 ist aber aufgrund der redundanten Information (z. B. Kontextknoten) insgesamt größer.

Realisierungskonzept ÄR-1 (individuelle Instanzgraphen) ist erwartungsgemäß für wenige Änderungen sehr groß. Sie benötigt jedoch bei steigender Anzahl von Änderungen nur wenig zusätzlichen Speicherbedarf. Dieser Anstieg ist ausschließlich darauf zurückzuführen, dass gelöschte Knoten bei dieser Implementierung explizit verwaltet werden, d. h. sie werden als gelöscht markiert. Kanten werden dagegen physisch gelöscht. Würde man auch Knoten physisch löschen, wäre der Speicherbedarf von individuellen Instanzgraphen in dem verwendeten Anwendungsfall konstant, da pro Änderungstransaktion ein Knoten hinzugefügt und einer entfernt wird.

5.2.2 Laufzeit (Zugriffe)

Abb. 16 zeigt das Ergebnis der Messungen für das Weiterschalten abhängig von der Anzahl Änderungen für die verschiedenen Änderungsrepräsentationen jeweils relativ zum Weiterschalten bei einer unveränderten Prozessinstanz. Abb. 17 zeigt die Ergebnisse für die Bestimmung aller transitiven Vorgänger des letzten Knotens der getesteten Knotensequenz.

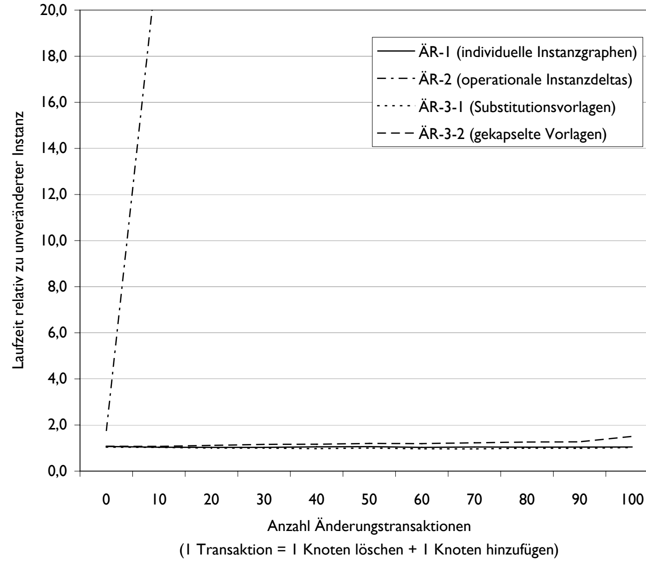

Abbildung 16: Zugriffsdauer beim Weiterschalten

Sehr auffällig ist in beiden Fällen die hohe Laufzeit von Realisierungskonzept ÄR-2. Insbesondere beim Weiterschalten (vgl. Abb. 16) ist der Anstieg bezüglich der Anzahl der Änderungen sehr ausgeprägt. Dies liegt insgesamt an der aufwendigen temporären Rekonstruktion der aus den Änderungen resultierenden logischen Vorlage. Da diese im Falle des Weiterschaltens vor jedem Zugriff erfolgt, während sie bei der Vorgängerbestimmung nur einmal erfolgt, fällt der Aufwand für die Rekonstruktion gegenüber dem des Zugriffs hier wesentlich mehr ins Gewicht als bei der Vorgängerbestimmung.

Bei Realisierungskonzept ÄR-1 gibt es sowohl beim Weiterschalten als auch bei der transitiven Vorgängerbestimmung kaum Unterschiede in der Laufzeit verglichen mit einer unveränderter Instanz. Dies ist einleuchtend, da der Zugriff beides Mal identisch abläuft, d. h. es gibt keinerlei Unterschied zwischen dem Zugriff auf eine uneränderte Instanz und einer veränderten Instanz mit individuellem Instanzgraphen.

Auffällig ist, dass auch Implementierungsvariante ÄR-3-1 ein zu Realisierungskonzept ÄR-1 vergleichbares Laufzeitverhalten besitzt. Dies zeigt, dass die Bestimmung der gültigen Vorlage vor jedem Knotenzugriff keine wesentlichen Auswirkungen hat. Der leichte Anstieg der Zugriffszeit sowohl bei individuellen Instanzgraphen als auch im Zusammenhang mit Substitutionsvorlagen bei größerer Anzahl an Änderungen kann mit dem höheren Speicherbedarf erklärt werden. Bei beiden Verfahren steigt dieser gegenüber einer unveränderten Instanz an (vgl. Abschnitt 5.2.1), wodurch die Laufzeitumgebung (z. B. Speicherverwaltung) beeinträchtigt wird. Dieser wiederum beeinflusst indirekt die Laufzeit.

Bei Implementierungsvariante ÄR-3-2 ist der Zugriff etwas langsamer als bei Implementierungsvariante ÄR-3-1 und Realisierungskonzept ÄR-1, da potentiell aufwendigere Mengenoperationen nötig werden, z. B. zum Filtern von gelöschten Knoten. Mit steigender Anzahl an Änderungen werden diese Operationen noch aufwendiger, da die zu durchsuchende Knoten- und Kantenmenge größer wird, was den relativen Anstieg gegenüber den beiden schnelleren Verfahren erklärt.

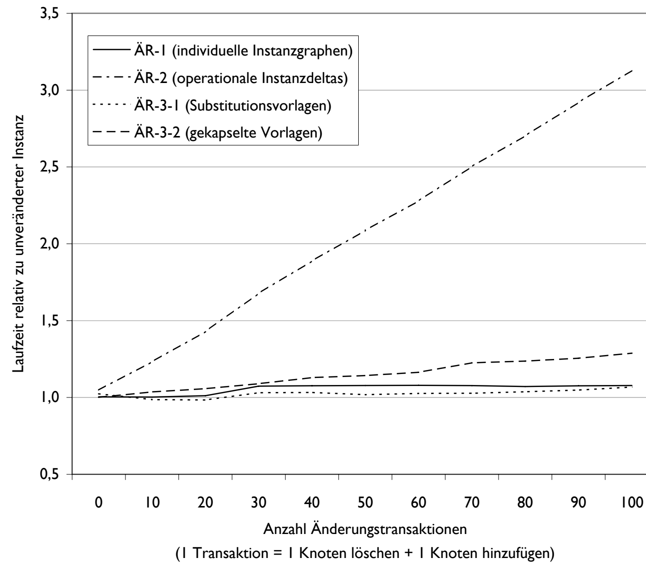

Abbildung 17: Zugriffsdauer bei der transitiven Vorgängerbestimmung

5.3 Fazit

Die Realisierung instanzspezifischer Änderungen mittels Realisierungskonzept ÄR-2 (operationale Instanzdeltas) ist aufgrund der Zugriffszeit und des Speicherbedarfs nicht für die Repräsentation im Primärspeicher geeignet. In Form einer Änderungshistorie im Sekundärspeicher sind sie absolut notwendig, z. B. für Schemaevolution [20] und das Nachvollziehen von Änderungen.

Aufgrund der Zugriffszeiten ist Implementierungsvariante ÄR-3-1 eine sehr gutes Konzept bei geringem Speicherbedarf. Die prototypische Implementierung offenbarte jedoch eine sehr komplexe und damit fehleranfällige Realisierung. Insbesondere die Verwaltung der Kanten hat sich als problematisch herausgestellt, da bei zu ändernden oder zu löschenden Kanten vorab nicht immer festgestellt werden kann, ob die Kante bereits Bestandteil des Substitutionsobjekts oder nur in der ursprünglichen Vorlage vorhanden ist. Im ersten Fall muss sie ggf. physisch gelöscht werden, im zweiten Fall nicht, da dies Auswirkungen auf alle Instanzen der entsprechenden Vorlage hätte. Bei komplexen Änderungen ist zudem eine Verschlechterung des Speicherbedarfs zu erwarten, da ein größerer Änderungskontext zu speichern ist.

Realisierungskonzept ÄR-1 benötigt bei wenigen Änderungen vergleichsweise viel Speicherplatz, weshalb sich diese Variante erst ab einer Vielzahl von Instanzänderungen empfiehlt. Dies lässt sich sehr gut in Kombination mit gekapselten Vorlagen realisieren, da diese den Wechsel zur Repräsentation mittels individueller Instanzgraphen zur Laufzeit ermöglichen, d. h. sobald die Änderungen eine bestimmte Grenze überschreiten, kann bei der Implementierung von Variante ÄR-3-2 nahtlos zu eienr Implementierung des Realisierungskonzepts ÄR-1 gewechselt werden. Variante ÄR-3-2 benötigt zudem am wenigsten Speicherplatz (bis 100 einfache Änderungen). Sie hat zwar etwas schlechtere Zugriffszeiten als Implementierungsvariante ÄR-3-1 und Realisierungskonzept ÄR-1, die Zugriffszeiten sind jedoch immer noch sehr effizient. Zudem ist Variante ÄR-3-2 einfach zu implementieren.

6 Zusammenfassung und Ausblick

Flexibilität ist für ein breit einsetzbares Prozess-Management-System unabdingbar. Dies darf jedoch nicht auf Kosten der Performanz gehen. In diesem Beitrag wurde untersucht, wie sich umfassende Flexiblität, nämlich sowohl Änderungen an einzelnen laufenden Prozessinstanzen als auch die Propagation von Änderungen von Prozessvorlagen auf laufende Instanzen effizient realisie-

ren lassen. Besonderes Augenmerk wurde dabei auch auf ein effizientes Zusammenspiel beider Änderungsarten gelegt. Dazu wurden verschiedene Realisierungskonzepte und Implementierungsvarianten umfassend diskutiert und miteinander verglichen. Neben einem qualitativen Vergleich wurden auch eine prototypische Implementierung vorgenommen, die anhand von Messungen einen quantitativen Vergleich ermöglicht. Die gewonnenen Erkenntnisse wurden bereits in einem Softwarewerkzeug für Schemaevolution und instanzspezifische Änderungen umgesetzt [11].

Die in diesem Beitrag diskutieren Realisierungskonzepte und Implementierungsvarianten bilden eine ausgezeichnete Ausgangsbasis für eine weitergehende Untersuchung. Dabei sollen weitere Varianten im Detail untersucht und bewertet werden. Hierzu zählen die temporäre Realisierung der logischen instanzspezifischen Vorlage für komplexe Funktionen mit häufigen Zugriffen auf die Vorlage sowie hybride Verfahren, etwa der Wechsel zwischen gekapselten Vorlagen (ÄR-3-2) und individuellen Instanzgraphen (ÄR-1). Außerdem können die Auswirkungen von Ein-/Auslagern aus bzw. in den Sekundärspeicher untersucht werden; hierbei spielt die absolute Größe der Datenstrukturen eine wichtige Rolle. Ebenso sind Varianten für die Repräsentation von Änderungen im Sekundärspeicher sowie die Durchführung einer Schemaevolution ohne und mit instanzspezifischen Änderungen Gegenstand weiterer Forschung.

Literatur

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

[2] W.M.P. van der Aalst, 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.

[3] M. Adams, A.H.M. ter Hofstede, D. Edmond, und W.M.P. van der Aalst: *Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows.* In: *On The Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Band 4275 der Reihe *LNCS*, Seiten 291–308, Montpellier, November 2006. Springer.

[4] S. Bassil, 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.

[5] S. Bassil, R.K. Keller, und P. Kropf: *A Workflow-Oriented System Architecture for the Management of Container Transportation.* In: *Business Process Management*, Band 3080 der Reihe *LNCS*, Seiten 116–131, Potsdam, June 2004. Springer.

[6] P. Dadam 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.

[7] P. Dadam, M. Reichert, S. Rinderle, M. Jurisch, H. Acker, K. Göser, U. Kreher, und M. Lauer: *Towards Truly Flexible and Adaptive Process-Aware Information Systems.* In: *Information Systems and e-Business Technologies*, Band 5 der Reihe *LNBIP*, Seiten 72–83, Klagenfurt, Austria, April 2008. Springer.

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

[9] M. Kloppmann, 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.

[10] U. Kreher, M. Reichert, S. Rinderle-Ma, und P. Dadam: *Effiziente Repräsentation von Vorlagen- und Instanzdaten in Prozess-Management-Systemen.* Technical Report UIB-2009-08, University of Ulm, Ulm, 2009.

[11] M. Lauer, S. Rinderle, und M. Reichert: *Repräsentation von Schema- und Instanzobjekten in adaptiven Prozess-Management-Systemen.* In: *GI-Jahrestagung*, Nummer 2, Seiten 555–560, Ulm, 2004.

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

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

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

[15] D. Müller, J. Herbst, M. Hammori, und M. Reichert: *IT Support for Release Management Processes in the Automotive Industry.* In: *Business Process Management*, Band 4102 der Reihe *LNCS*, Seiten 368–377, Vienna, October 2006. Springer.

[16] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, und A. Kotz Dittrich: *Enterprise-Wide Workflow Management Based on State and Activity Charts.* In: A. Dogaç, 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*, Seiten 281–303, Istanbul, 1998. Springer.

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

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

[19] M. Reichert und P. Dadam: *Enabling Adaptive Process-aware Information Systems with ADEPT2.* In: *Handbook of Research on Business Process Modeling*, Seiten 173–203. Hershey, New York, New York, 2009.

[20] M. Reichert, S. Rinderle, und P. Dadam: *On the Common Support of Workflow Type and Instance Changes under Correctness Constraints.* In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, Band 2888 der Reihe *LNCS*, Seiten 407–425, Catania, 2003. Springer.

[21] M. Reichert, S. Rinderle-Ma, und P. Dadam: *Flexibility in Process-aware Information Systems.* In: *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, Band 5460 der Reihe *LNCS*, Seiten 115–135, Eindhoven, 2009. Springer.

[22] M. Reichert und D. Stoll: *Komposition, Choreographie und Orchestrierung von Web Services: Ein Überblick.* EMISA-Forum, 24(2):21–32, 2004.

[23] S. Rinderle, M. Reichert, und P. Dadam: *Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications.* In: *On The Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, Band 3290 der Reihe *LNCS*, Seiten 101–120, Agia Napa, Cyprus, 2004. Springer.

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

[25] S. Rinderle, M. Reichert, und P. Dadam: *On Dealing with Structural Conflicts between Process Type and Instance Changes.* In: *Business Process Management*, Band 3080 der Reihe *LNCS*, Seiten 274–289, Potsdam, Germany, June 2004. Springer.

[26] S. Rinderle, M. Reichert, M. Jurisch, und U. Kreher: *On Representing, Purging, and Utilizing Change Logs in Process Management Systems.* In: *Business Process Management*, Band 4102 der Reihe *LNCS*, Seiten 241–256, Vienna, October 2006. Springer.

[27] S. Rinderle-Ma, M. Reichert, und B. Weber: *On the Formal Semantics of Change Patterns in Process-aware Information Systems.* In: *Proceedings of the 27th Int'l Conference on Conceptual Modeling (ER'08)*, Band 5231 der Reihe *LNCS*, Seiten 279–293, Barcelona, October 2008. Springer.

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

[29] B. Weber, M. Reichert, und S. Rinderle-Ma: *Change Patterns and Change Support Features – Enhancing Flexibility in Process-Aware Information Systems.* Data & Knowledge Engineering, 66(3):438–466, 2008.

[30] M. Weske: *Object-Oriented Design of a Flexible Workflow Management System.* In: *Advances in Databases and Information Systems (ADABIS)*, Band 1475 der Reihe *LNCS*, Seiten 119–130, Poznan, September 1998. Springer.

[31] M. Weske: *Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System.* In: R. Sprague (Herausgeber): *Proc. 34th Hawaii International Conference on System Sciences (HICSS-34).* IEEE Computer Society Press, 2001.

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

2009-09 *Dammertz, Holger, Alexander Keller, Hendrik P.A. Lensch*
Progressive Point-Light-Based Global Illumination

2009-10 *Dao Zhou, Christoph Müssel, Ludwig Lausser, Martin Hopfensitz, Michael Kühl, Hans A. Kestler*
Boolean networks for modeling and analysis of gene regulation

2009-11 *J. Hanika, H.P.A. Lensch, A. Keller*
Two-Level Ray Tracing with Recordering for Highly Complex Scenes

2010-01 *Hariolf Beth, Frank Raiser, Thom Frühwirth*
A Complete and Terminating Execution Model for Constraint Handling Rules

2010-02 *Ulrich Kreher, Manfred Reichert*
Speichereffiziente Repräsentation instanzspezifischer
Änderungen in Prozess-Management-Systemen

Ulmer Informatik-Berichte
ISSN 0939-5091

Herausgeber:
Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
89069 Ulm