View-Unterstützung in Prozess-Management-Systemen

Diplomarbeit

vorgelegt von

Achim Klotz

1. Gutachter: Dr. Manfred Reichert

2. Gutachter: Prof. Dr. Peter Dadam

Ulm, Oktober 2004

Kurzfassung

Sichten (englisch *Views*) spielen in Datenbanksystemen seit langem eine wichtige Rolle, wenn es um die Datenunabhängigkeit von Anwendungssystemen oder Sicherheitsaspekte geht. Im Blickpunkt dieser Arbeit ist die Fragestellung, inwieweit sich Views auch auf Schemata und Instanzen in Prozess-Management-Systemen anwenden lassen. Zu diesem Zweck werden zunächst für Prozess-Views verschiedene Anwendungsfälle (z.B. Kapselung, Komplexitätsverminderung, Effizienz und Verbergung von Informationen) identifiziert. Je nach Anwendungsfall sind verschiedene Operationen auf Views nötig. Diese lassen sich in Operationen für reine Lesezugriffe sowie Operationen für Schema- und Instanzänderungen unterteilen. Zur View-Bildung können insbesondere zwei grundlegende Methoden zur Anwendung kommen: Die Zusammenfassung mehrerer Aktivitäten zu einem neuen virtuellen Schritt (Graphaggregation) oder das Verbergen von einzelnen Aktivitäten bzw. Prozesselementen (Graphreduktion). Aufbauend auf diesen Methoden und den für Views definierten Operationen werden Algorithmen für die Viewerzeugung sowie verschiedene mögliche Methoden für die Verwaltung und Speicherung von Views vorgestellt und diskutiert. Schließlich wird auf die Problematik der Aktualisierung von Views bei Prozess-Änderungen und die Möglichkeit von Schemaänderungen über eine View im Detail eingegangen. Beide Aspekte sind für die Realisierung adaptiver Prozess-Management-Systeme fundamental.

Inhalt

1 Motivation

In relationalen Datenbanksystemen spielen Views bzw. Sichten schon lange eine wichtige Rolle [1]. Mit ihrer Hilfe ist es möglich, virtuelle Relationen zu erzeugen, auf die Anwender, ebenso wie auf Basis-Relationen zugreifen können. Beispielsweise können mehrere Basis-Relationen zu einer virtuellen Relation zusammengefasst oder einzelne Spalten einer Relation ausgeblendet werden.

Neben Datenbanksystemen spielen in der heutigen Zeit auch Prozess- bzw. Workflow-Management-Systeme [2, 3] eine immer wichtigere Rolle. Mit ihrer Hilfe kann nicht nur die Datenverwaltung, sondern auch die Modellierung, Ausführung und Überwachung von Geschäftsprozessen rechnergestützt erfolgen.

Prozesse sind in der Regel wesentlicher komplexer zu handhaben als Relationen in Datenbanken. Trotzdem gibt es für Prozess-Management-Systeme bisher kaum Mechanismen für die Definition und Bildung von Sichten. Dabei wird die Möglichkeit, komplexe Prozesse vereinfachen bzw. Teile eines Prozesses ausblenden zu können, bei zunehmender Verbreitung von Prozess-Management-System eine immer wichtigere Rolle spielen. Je nach Anwendungsfall sind nur Teilaspekte eines Prozesses von Interesse; die Darstellung des Gesamtprozesses mit all seinen Einzelheiten dagegen ist für normale Benutzer oftmals zu unübersichtlich und aus Sicherheitsgründen nicht immer gewünscht. In dieser Arbeit werden mögliche Anwendungsfälle und Verfahren zur Viewbildung für Prozesse eingehend untersucht.

1.1 Prozess-Management-Systeme

Mit Hilfe von Prozess-Management-Systemen (PMS) ist eine rechnergestütze Modellierung, Analyse, Ausführung und Überwachung von Geschäftsprozessen möglich. Grundidee ist, dass Prozesse nicht mehr implizit in Programmen codiert und unveränderbar vorliegen, sondern dass sie explizit beschrieben und im PMS hinterlegt werden. Dieser Ansatz hat mehrere Vorteile: Durch graphische Darstellung können die Prozesse verständlich und auf hoher semantischer Ebene veranschaulicht werden. Schwachstellen sind leichter auffindbar. Durch die explizite Modellierung ist es grundsätzlich auch möglich, Prozesse unter verschiedenen Fragestellungen zu analysieren sowie diese nachträglich zu ändern (auch auf Instanzebene). Dies gestaltet sich bei einer im Programmcode „versteckten“ Ablauflogik sehr viel schwieriger möglich. Auch eine Überwachung des Prozesszustandes laufender Instanzen sowie Auswertungen auf einer Kollektion von Instanzen (sog. *Process Performance Management*) sind möglich.

Für die Modellierung von Prozessen bedarf es eines Metamodells. Dieses legt fest, welche Art von Prozessen zulässig und welche Operationen für sie möglich sind. Mit Hilfe eines geeigneten Metamodells ist es auch möglich, Aussagen über Prozess-Eigenschaften zu treffen, z.B. darüber ob ein Prozess in jedem Fall terminiert oder ob Deadlocks auftreten können. Ein Beispiel für ein solches Metamodell ist das ADEPT Basismodell [4], das auch für die in dieser

Arbeit vorgestellten Algorithmen und Beispiele verwendet wird. Die angestellten Überlegungen lassen sich aber leicht auf andere Metamodelle übertragen.

1.2 Sichtenunterstützung in Prozess-Management-Systemen

Heutige Prozess-Management-Systeme bieten kaum Unterstützung für die Erzeugung und Verwaltung von Views. Prozesse und Prozess-Instanzen werden stets komplett dargestellt, und es gibt keine Möglichkeit, Teile auszublenden oder in aggregierter Form zusammenzufassen. Dabei gibt es auch für Prozess-Management-Systeme zahlreiche Anwendungsfälle für Views. Manche Informationen dürfen z.B. bestimmten Nutzer-Gruppen aus Sicherheitsgründen nicht zugänglich sein, andere sind im jeweiligen Anwendungskontext nicht von Interesse. Reale Prozesse können derart komplex sein (und aus hunderten von Einzelschritten bestehen), dass es für einen Bearbeiter oder Prozessdesigner schlicht unverzichtbar wird, Teile des Prozesses zusammenzufassen und nur die für die gerade anstehende Aufgabe relevanten Ausschnitte im Detail darzustellen.

Dabei ist das von vielen Prozess-Metamodellen unterstütze Konzept der Subprozess-Definition und -Bildung bei weitem nicht ausreichend. Bei diesem Konzept kann für einen einzelnen (Aktivitäten-) Knoten ein ganzer Subprozess hinterlegt werden und es ist damit möglich, die Komplexität des Graphen zu reduzieren. Diese Vereinfachung durch hierachische Anordnung der Prozessmodelle ist aber sehr statisch und muss teilweise bereits bei der Modellierung erfolgen. Eine kontextabhängige Vereinfachung und Zusammenfassung bestimmter Prozessteile für einen bestimmten Benutzer oder abhängig von einem bestimmten Prozesszustand ist damit nicht möglich.

Die Möglichkeiten, die eine flexible Viewbildung bietet werden in dem in Abbildung 1 gezeigten Beispiel deutlich. Als Beispielprozess dient der (vereinfachte) Produktionsprozess eines PC-Herstellers. Dabei geht eine Bestellung für einen PC ein, die Produktion wird vorbereitet, die Gehäuse werden vom Hersteller selbst gefertigt und alle sonstigen Teile von verschiedenen Lieferanten bezogen. Anschließend werden die Teile zusammengebaut, kontrolliert, verpackt und schließlich versandt. Schon an diesem einfachen Beispiel wird deutlich, dass es in einem Prozess viele verschiedene Beteiligte gibt, die sich für verschiedene Teilaspekte des Prozesses interessieren, teilweise auch abhängig vom jeweiligen Berarbeitungskontext.

In diesem Beispiel wird eine View für den Kunden, der einen neuen PC in Auftrag gibt, gebildet. Für ihn ist es wichtig, dass er eine neue Bestellung aufgeben kann und Informationen zum Status der Produktion und zum Datum des Versands erhält. Details des Produktionsprozesses selbst und der daran Beteiligten interessieren ihn nicht und der Hersteller wird diese internen Abläufe in der Regel auch nicht nach außen bekannt machen. Ganz anders muss eine View für den Produktionsleiter der Gehäuseproduktion aussehen. Für ihn ist vor allem der gesamte Produktionsablauf, der für die Herstellung eines Gehäuses notwendig ist,

interessant. Nachfolgende oder parallele Tätigkeiten, wie Versand und Beschaffung anderer Komponenten, sind für ihn dagegen von untergeordneter Bedeutung. Wieder eine andere View benötigt der Lieferant der CPUs. Für ihn ist vor allem die Beschaffungsaktivität von Interesse, die anderen Lieferanten dürfen ihm dagegen nicht bekannt gemacht werden.

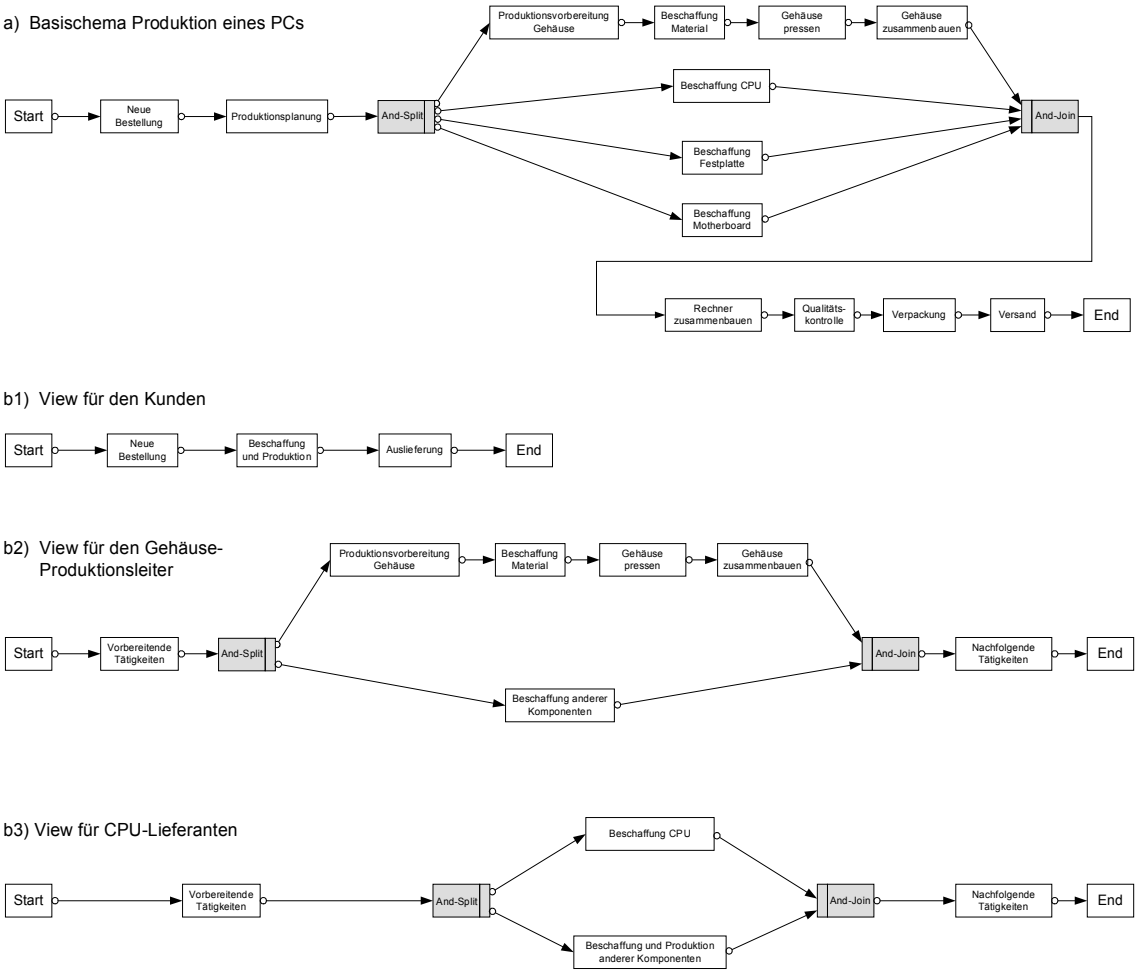

Abbildung 1: Beispiel eines Prozesses mit verschiedenen Views

1.3 Aufgabenstellung und Gliederung

In dieser Diplomarbeit sollen verschiedene Aspekte von Prozess-Sichten sowie ihrer Definition, Verwaltung und Nutzung behandelt werden. Zunächst werden relevante Anwendungsfälle für Views und notwendige Operationen identifiziert. Danach sollen verschiedene Arten von Views und Algorithmen für die View-Bildung entwickelt werden. Neben der Erzeugung von Views sollen Möglichkeiten für ihre Speicherung und Verwaltung aufgezeigt werden. Ein wichtiges

Thema ist die Fragestellung, inwieweit es möglich ist, über Views auch Aktualisierungen auf die zugrunde liegenden Basisprozesse zu propagieren. Umgekehrt stellt sich die Frage, wie in der Gegenrichtung bei Änderungen am Basisprozess, die Views entsprechend aktualisiert werden können (soweit sinnvoll und möglich). Beide Fragestellungen wurden in bisherigen Arbeiten nicht untersucht.

In Kapitel 2 erörtern wir verschiedene Anwendungsfälle für die Verwendung von Views. In Kapitel 3 betrachten wir dann die notwendigen Operationen für die Handhabung dieser Anwendungsfälle. In Kapitel 4 erfolgt eine Klassifizierung für verschieden Arten von Views in Abhängigkeit der jeweils betroffenen Elemente eines Prozessgraphen. Für die verschiedenen Arten von Views werden in Kapitel 5 geeignete Algorithmen zur View-Bildung vorgestellt. Kapitel 6 diskutiert verschiedene Möglichkeiten für die Erzeugung von Views, Kapitel 7 vergleicht typische Ansätze für die View-Speicherung und -Verwaltung. In Kapitel 8 wird auf die Problematik der Aktualisierung von Views eingegangen. Kapitel 9 vergleicht die in dieser Arbeit entwickelten Konzepte mit anderen Ansätzen. Die Diplomarbeit schließt mit einer kurzen Zusammenfassung in Kapitel 10.

2 Anwendungsfälle für Prozess-Views

Um Aussagen über Eigenschaften von Prozess-Views treffen zu können, ist es sinnvoll, mögliche Anwendungsfälle zu unterscheiden. Viele davon sind ähnlich der Verwendung von Views in Datenbanken. Jedoch gibt es für Prozess-Management-Systeme noch einige weitere Anwendungsszenarien, die bei Datenbanken nicht vorhanden sind, bei denen eine View-Bildung essentiell ist.

2.1 Grundlagen

Geschäftsprozesse, wie im Beispiel aus Abbildung 1 dargestellt, werden mit Hilfe von gerichteten Graphen dargestellt. Dabei stellen die einzelnen Knoten jeweils Aktivitäten (bzw. Schritte) im Prozess dar, die Abfolge der einzelnen Schritte (Kontrollfluss) wird mit Hilfe von Pfeilen (Kontrollflusskanten) dargestellt. Zusätzlich zu einer seriellen Abfolge von Schritten können mit Hilfe spezieller Knoten (z.B. AND-Split, AND-Join, XOR-Split und XOR-Join) auch Verzweigungen[1] und Schleifen dargestellt werden. Neben dem Kontrollfluss ist für Prozesse auch der Datenfluss relevant. Datenelemente bzw. Prozessvariablen werden dabei mit Hilfe von speziellen Datenknoten und der Fluss von Daten zwischen den einzelnen Schritten mit Hilfe von Pfeilen (Datenflusskanten) dargestellt. Jedes Element in einem Prozessgraphen kann ein oder mehrere Attribute haben, z.B. bei Aktivitäten können dies die benötigte Dauer oder die Kosten für die Durchführung einer Aktivität sein.

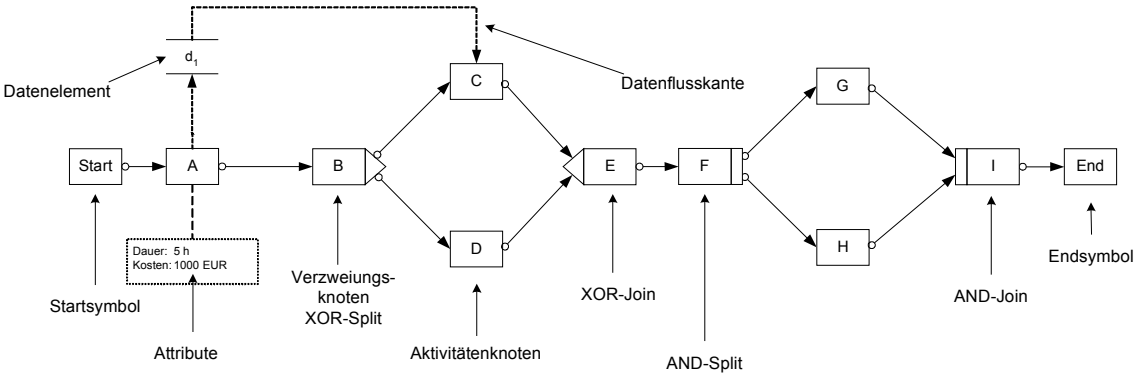

Abbildung 2: Beispiel für einen Prozess und die einzelnen Elemente

Mit Hilfe von Aktivitäten, Kontrollflusskanten und Datenflusselementen kann eine Prozessvorlage, ein sog. Schema bzw. Prozess-Schema definiert werden. Wenn ein Prozess nun real ausgeführt werden soll, können mit Hilfe des Schemas ein oder mehrere (Prozess-) Instanzen gebildet werden. Im Unterschied zu einem Schema ist bei einer Instanz für die

[1] Hierbei kann es sich entweder um parallele Verzweigungen handeln, bei denen verschiedene Pfade parallel durchlaufen werden oder alternative Verzweigungen, bei denen nur ein Pfad durchlaufen wird, während die nicht selektierten Pfade abgewählt werden.

einzelnen Knoten (je nach Metamodell auch für die Kanten) ein Zustand vorhanden. So kann jeder Aktivitätenknoten sich z.B. in einem der Zuständen *nicht aktiviert*, *aktiviert*, *laufend*, *beendet* oder *abgebrochen* befinden.

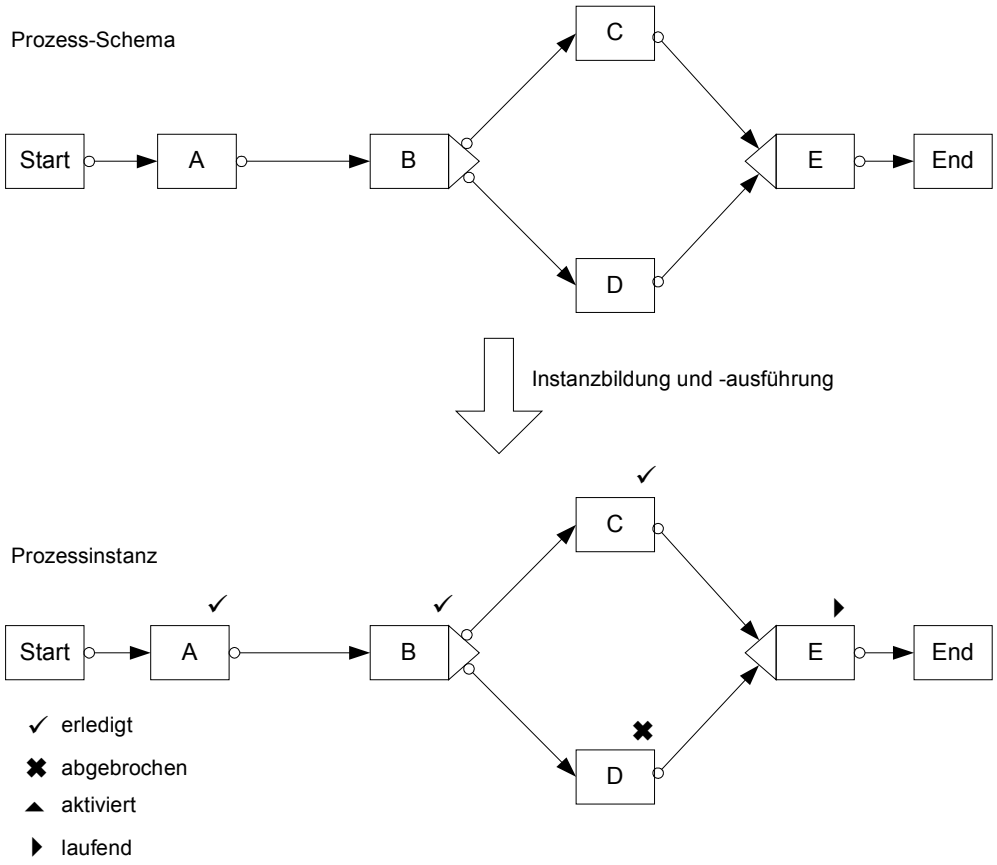

Abbildung 3: Schema und daraus gebildete Instanz

Views können sowohl für Schemata als auch für Instanzen gebildet werden. Dabei wird der Graph für den eine View gebildet wird als *Basisprozess* bezeichnet. Je nach Art des Basisprozesses kann er auch *Basisschema* oder *Basisinstanz* genannt werden. Die Aktivitäten des Basisprozesses bezeichnen so genannte Basisschritte oder Basisaktivitäten. Eine View, die im Gegensatz zu einem Basisprozess einen virtuellen Prozess darstellt, kann sowohl Basisschritte als auch virtuelle Schritte (bzw. virtuelle Aktivitäten) enthalten. Ein virtueller Schritt wird auch abstrakter Schritt genannt. Basisschritte werden direkt aus dem Basisprozess übernommen, während virtuelle Schritte erst bei der Erzeugung der View entstehen und z.B. mehrere Basisschritte zusammenfassen.

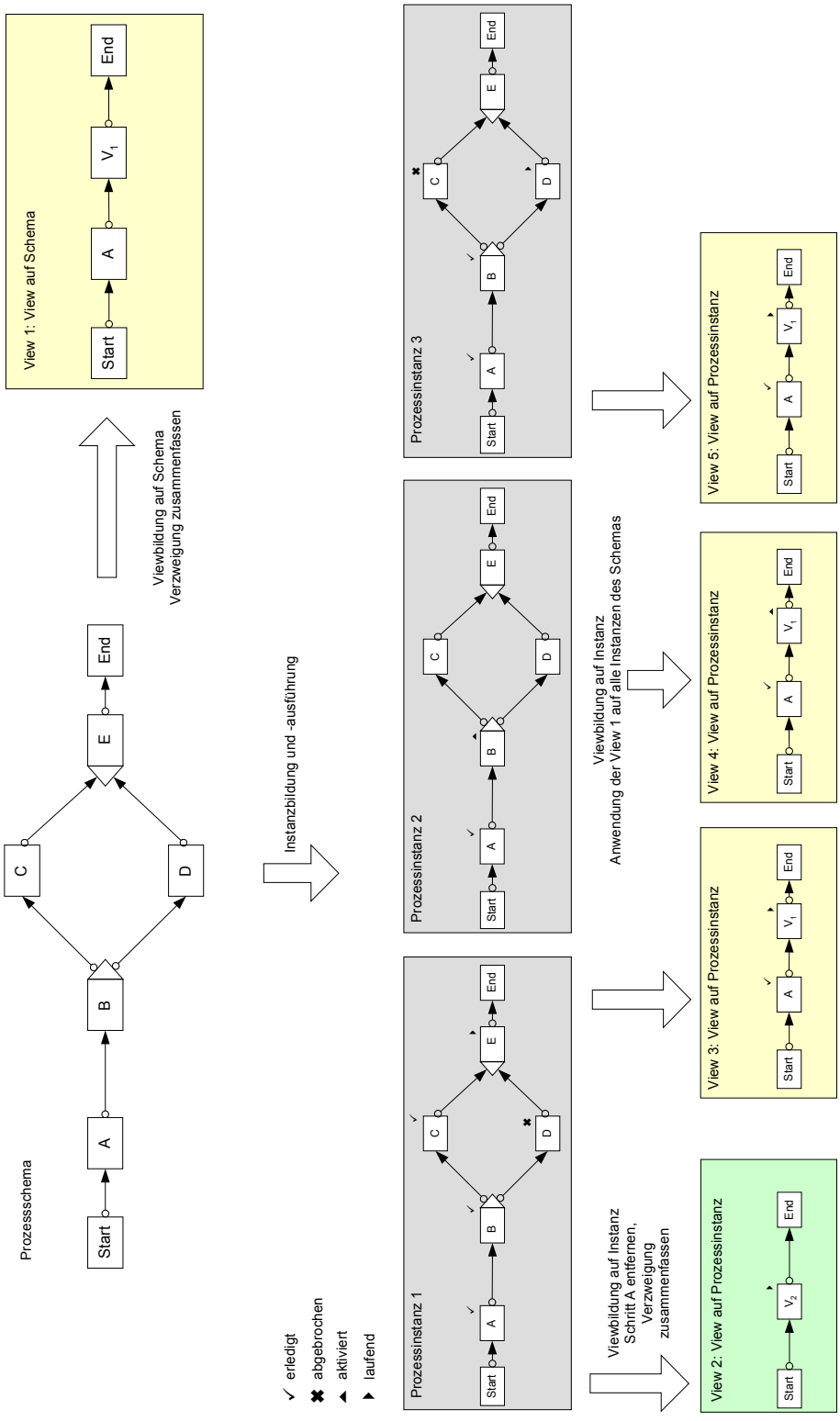

Abbildung 4: Verschiedene Möglichkeiten für die Viewbildung

Views können, wie im Beispiel aus Abbildung 4 exemplarisch dargestellt sowohl für Schemata als auch für Instanzen definiert werden. Eine View, die für ein Schema vorliegt, kann auch für die auf diesem Schema laufenden Instanzen genutzt werden, zumindest solange diese nicht individuell modifiziert worden sind.

Eine weitere Möglichkeit der Viewbildung bei Prozess-Management-Anwendungen ist die Zusammenfassung mehrerer Prozessgraphen (Integration von Prozessen) in einem Graphen, wie in den Beispielen aus Abbildung 20 auf Seite 32 (Zusammenfassung mehrere Instanzen in einem Graph) und Abbildung 21 auf Seite 34 (Verknüpfung mehrerer Schemata in einem Graph) gezeigt.

Wenn auf einen Basisprozess eine View gebildet wird, dessen Graph ein gültiger Prozessgraph nach der Definition des Metamodells für den Basisprozess ist, wird die View als Struktur erhaltend bezeichnet.

2.2 Anwendungsfälle

Die Anwendungsfälle für Views lassen sich in vier Hauptkategorien einteilen: Komplexitätsreduzierung, Verbergen von Informationen (Sicherheit), Effizienz und Kapselung:

1. **Vereinfachung komplexer Prozesse**
   Vereinfachte bzw. abstrakte Prozessdarstellungen sind für das Monitoring nützlich. Damit können Benutzern Sichten angeboten werden, die nur die für sie relevanten Prozessteile anzeigt:

   - Bezogen auf eine bestimmte Ressource (z.B. einen bestimmten Bearbeiter) sollte ein Überblick möglich sein. Es sind nur die Schritte relevant, die dieser Ressource zugeordnet sind, alle anderen sollen entweder ausgeblendet oder zu wenigen Knoten, die besonders gekennzeichnet sind, zusammengefasst werden.
   - Monitoring laufender Prozesse: Bereits erledigte bzw. abgewählte Prozesspfade sind oft von sekundärem Interesse, d.h. sie können bei der Prozessvisualisierung weggelassen bzw. vereinfacht werden. Auch aktive Prozessteile können zwecks besserer Übersichtlichkeit zusammengefasst werden. Wichtig ist, dass auch für zusammengefasste Aktivitäten ein „abstrakter Zustand“ gezeigt werden kann.
   - Überblick zu einer Kollektion von Prozess-Instanzen (bzw. Ausschnitte). Wenn z.B. in einem Unternehmen viele Prozesse, die auf dem gleichen Schema beruhen (z.B. Antrag für einen Kredit), gleichzeitig laufen, können diese Prozesse für die Visualisierung auf einen einzigen Prozessgraph abgebildet werden.
   - Bildung von Views für system- und organisationsübergreifende Prozesse. Verschiedene Prozesse mehrerer Beteiligter werden zu einem virtuellen Prozess zusammengefasst. Wenn ein Hersteller z.B. die Einzelteile für die Produktion von

mehreren verschiedenen Lieferanten bezieht, können der Herstellungsprozess und die Prozesse der Lieferanten über eine View zu einem Prozess integriert werden.

In einigen dieser Anwendungsfälle ist sekundär, ob die strukturelle Integrität der Prozesse erhalten bleibt oder nicht. In solchen Fällen dienen Views nur dazu, sich einen Überblick über einen Prozess zu verschaffen.
Unter gewissen Voraussetzungen können Sichten aber auch beim Prozessentwurf und bei Prozessänderungen (auch laufender Instanzen) verwendet werden:

- Wenn ein komplexer Prozess entworfen wird (oder ein bereits vorhandener geändert), sollte man Teile des Prozesses bei Bedarf zusammengefasst darstellen können, um eine bessere Übersichtlichkeit beim Entwurf zu erreichen
- Ad-Hoc Änderungen eines laufenden Prozesses: Wenn ein bestimmter Ausschnitt einer Prozess-Instanz geändert werden soll, kann ein Großteil des Prozesses (z.B. die Teile, die schon beendet oder die für die Änderung nicht relevant sind) ausgeblendet werden.
- In der Praxis treten oft mehrere dieser Anwendungsfälle gleichzeitig auf. Beispielsweise hat ein Nutzer oftmals nur eine eingeschränkte Sicht auf einen Prozess, auf der er auch Änderungen definiert.
  In diesen Anwendungsfällen muss gewährleistet sein, dass über die Views die Änderungsoperationen durchgeführt und diese auf den Basisprozess propagiert werden kann.

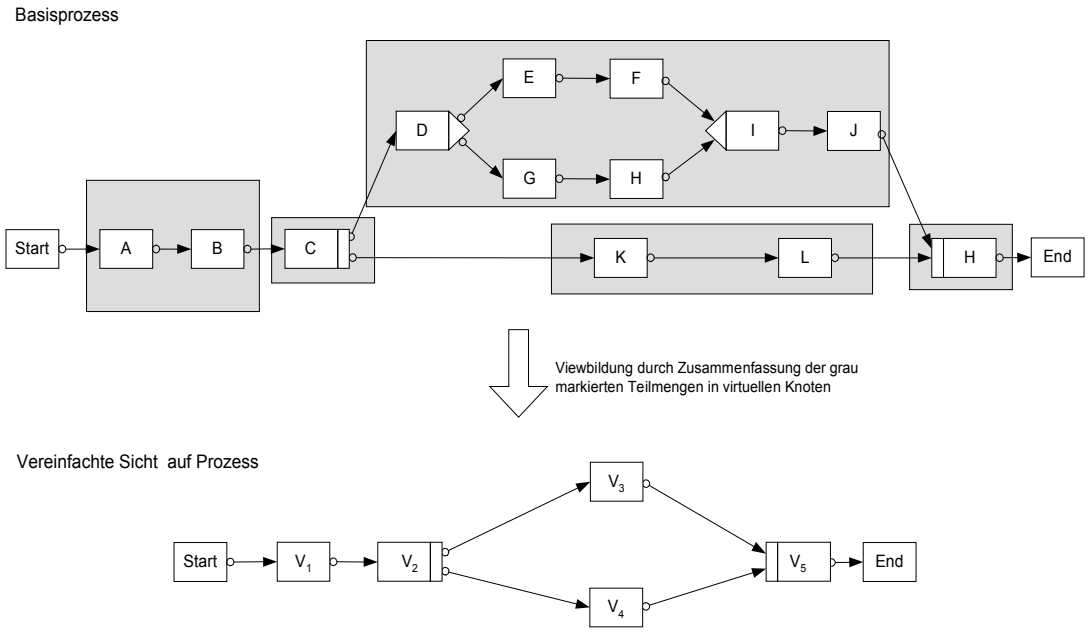

Abbildung 5: Viewbildung zur Komplexitätsverminderung

2. **Sicherheitsaspekte, Verbergen von Prozessinternas**
   Bestimmte Prozessdaten sind für einen Prozess-Beteiligten oftmals irrelevant oder sollen aus Sicherheitsgründen nicht angezeigt werden:
   - Interorganisationelle Workflows: Verschiedene Unternehmen arbeiten zusammen (Lieferant, Hersteller). Um ihre jeweiligen Workflows zu verknüpfen, werden Views benötigt, die geheime Informationen über Prozessinternas (z.B. Herstellungsablauf, involvierte Partnerunternehmen) verbergen, es dem jeweiligen Partner jedoch trotzdem erlauben, vernünftige Statusinformationen zum Gesamtprozess über die View zu erhalten bzw. auch Daten über die Views austauschen zu können
   - Innerhalb eines Unternehmens sollen den Bearbeitern, die an einem Prozess beteiligt sind, für bestimmte Aktivitäten nicht alle Attribute angezeigt werden. So sollten z.B. die Kosten oder die geplante Dauer eines Prozessschrittes nur bestimmten Rollen (bzw. entsprechenden Benutzern) sichtbar gemacht werden. Genauso könnten Teilgraphen (was passiert bei einem bestimmten Schritt im Fehlerfall) verborgen, oder Schritte, die für einen Bearbeiter nicht wichtig sind, zusammengefasst werden.

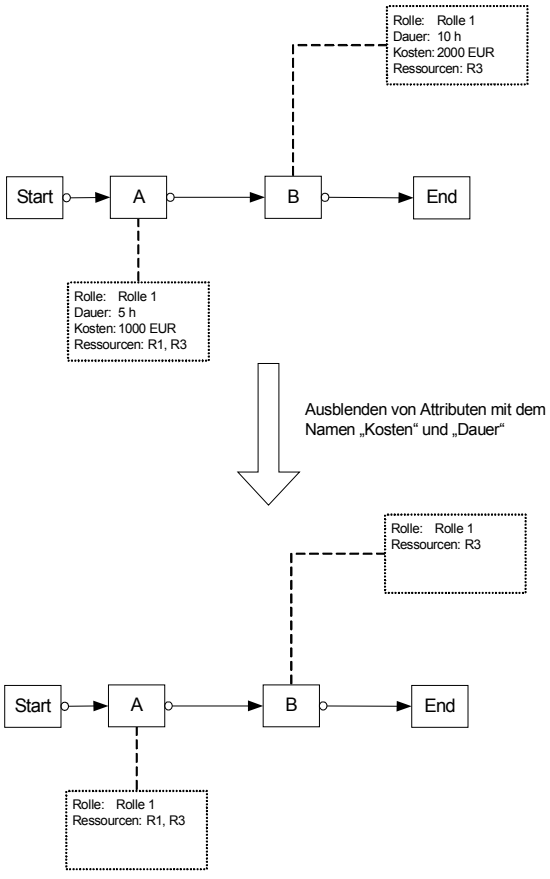

Abbildung 6: Verbergen sicherheitsrelevanter Informationen

Hier ist es notwendig, korrekte Informationen über den Prozesszustand des Basisprozesses auch über die View darzustellen, und Instanzen auch über die View abändern zu können.

3. **Effizienzaspekte**
   Kompakte bzw. abstrahierte Prozessrepräsentationen haben nicht nur für Bearbeiter den Vorteil der besseren Übersichtlichkeit, sondern sie können auch in der Prozess-Engine für höhere Effizienz sorgen:

   - Wenn es möglich ist, effizient Views zur Verfügung zu stellen, kann die Prozess-Engine bestimmte Operationen auf einfacheren Prozessmodellen durchführen. So könnte es z.B. sinnvoll sein, bereits beendete oder abgewählte Zweige, die für die Verwaltung von Prozess-Instanzen unwichtig sind, wegzulassen. Genauso könnte man Aktivitäten, die erst in ferner Zukunft aktiv werden, für eine gewisse Zeit ausblenden.
   - Wenn an Clients Information zu reduzierten Views und nicht zum kompletten Basisprozess geschickt wird, kann die zu transferierende Datenmenge erheblich reduziert werden. Damit läßt sich einerseits Datenkommunikation eingesparen, andererseits muss auch der Client nicht soviel Rechen- und Speicherkapazität zur Verfügung stellen.

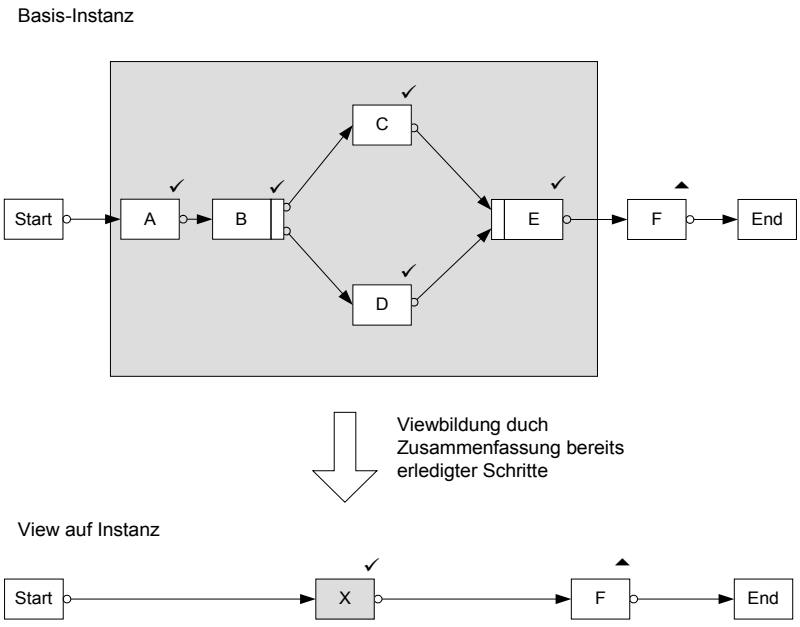

Abbildung 7: Verkleinerung einer Prozess-Instanz

- Bei interorganisationellen Workflows spielt, neben der Sicherheit, auch die Effizienz eine Rolle. Wenn viele gemeinsam durchgeführte Prozesse parallel ablaufen, macht es Sinn, nur die für die einzelnen Partner relevanten Teile zu übertragen.

Hierbei werden, je nach Anwendungsfall, verschiedene Anforderungen an die strukturelle Korrektheit von Views gestellt. Wenn nur Visualisierungsdaten an Clients geschickt werden, ist die strukturelle Integrität der Prozesse unwichtig, bezüglich der Verwendung von Views in der Prozess-Engine ist sie dagegen relevant.

4. **Kapselung von Prozessen**
   Wenn für ein Prozess-Management-System auch externe Anwendungsprogramme angesteuert werden, kann mit Hilfe von Views eine bessere Unabhängigkeit von Prozessen und Anwendungsprogrammen erreicht werden:

   - Zwei Unternehmen arbeiten über Views ihrer Prozesse zusammen. Ändert nun ein Unternehmen seinen internen Prozess (*Private Process*), lässt die View (d.h. den *Public Process*) aber gleich, muss das andere Unternehmen seinen Prozess nicht verändern.
   - Irgendeine Applikation (z.B. eine alte Anwendung) geht von einem ganz speziellen Prozessgraphen aus. Sie nutzt dabei einen bestimmten Basisprozess. Wenn dieser Prozess geändert werden soll, ohne dass man die Anwendung anpassen muss, kann man eine View auf dem geänderten Basisprozess definieren. Dabei entspricht die Struktur der View dem Schema des Basisprozesses vor der Änderung. Dann kann der Basisprozess umbenannt werden und die View mit dem vorherigen eindeutigen Bezeichner des Basisprozesses versehen werden. Somit ist der Prozess von der Anwendung durch die View gekapselt und die Anwendung kann ohne Änderung weiterverwendet werden.

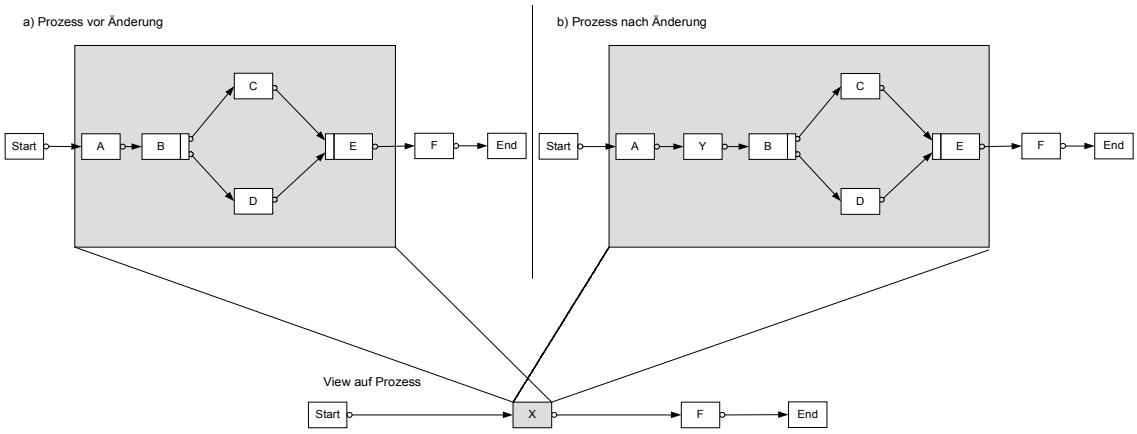

Abbildung 8: Kapselung eines Prozesses mit Hilfe einer View

Oftmals können sich mehrere der hier vorgestellten Anwendungsfälle überschneiden. So kann es z.B. einerseits erforderlich sein, gewisse Informationen aus Sicherheitsgründen auszublenden, aber andererseits sollen andere Information zur Komplexitätsverminderung noch zusätzlich ausgeblendet werden.

3 Operationen auf Views

Im vorherigen Kapitel haben wir aufgezeigt, für welche Anwendungsfälle Views benötigt werden. Je nach Anwendungsfall ist es notwendig, dass mehr oder weniger Operationen auf der jeweiligen View durchführbar sind. Diese lassen sich in drei grundsätzliche Kategorien einteilen: Leseoperationen, Instanzänderung und Schemaänderung. Alle drei Operationsarten werden in den folgenden Abschnitten genauer erläutert.

Anwendungsfall	Lesen	Instanz	Schema	Bemerkung
Komplexität vermindern	ja	ja	ja	Bei Views, bei denen die Komplexität eines Prozessen vermindert werden soll, sind alle Operationen sinnvol, da es nur darum geht, für den Benutzer nicht relevante Teile auszublenden.
Information verbergen / Sicherheitsaspekte	ja	ja	ja	Hier gilt das Gleiche wie für den obigen Anwendungsfall, nur dass es hier darum geht, Teile auszublenden, die der Benutzer nicht sehen darf.
Effizienzaspekte	ja	ja	nein	Hier geht es insbesondere um die effiziente Bearbeitung von Instanzen in der Prozess-Engine. Dabei ist vor allem das Lesen und Ändern von Daten und Zustands-informationen notwendig.
Kapselung	ja	ja	nein	Programme greifen nur über eine View auf Instanzen zu, dabei werden keine Schemaänderungen durchgeführt. Der Sinn der View-Bildung soll ja gerade die Unabhängigkeit von Schemaänderungen sein. Bei einer Schemaänderung über die View verändert sich auch die View und eine Kapselung ist nicht mehr gegeben.

Tabelle 1: Notwendige Operationen für die verschiedenen Anwendungsfälle

3.1 Leseoperationen

Die View dient lediglich dem Monitoring oder der Überwachung, ohne das der darunter liegende Basisprozess geändert wird. Solche Views können sowohl für Prozess-Schemata als auch -Instanzen gebildet werden. Hierbei ist die strukturelle Korrektheit der Views, je nach Anwendungsfall, unwichtig. Solange Informationen in der View vorhanden sind, können diese

auch gelesen werden. So ist es möglich, dass ein Bearbeiter lediglich die von ihm durchgeführten Schritte in einem Prozess sehen will, an deren Reihenfolgebeziehungen aber nicht interessiert ist. Sollen auch Informationen über Prozesszustände zur Verfügung gestellt werden, muss bei virtuellen Knoten, die mehrere Basisaktivitäten zusammenfassen, ein abstrakter Zustand generiert werden. Hierbei erfolgt mit Hilfe einer Funktion eine Abbildung der Zustände der Basisaktivitäten auf einen abstrakten Zustand.

3.2 Instanzänderung

Im vorherigen Abschnitt wurden Leseoperation auf Prozess-Schemata und -Instanzen erläutert. Bei Prozess-Instanzen kann über Views auch der Zustand der Instanz geändert werden. So kann z.B. ein Bearbeiter auch über eine (vereinfachende) View, die nur die von ihm durchgeführten Schritte anzeigt, Dateneingaben machen und damit Prozessvariablen ändern oder auch durch Weiterschalten eines Schrittes den eigentlichen Prozesszustand ändern. Damit dies möglich ist, muss die View die benötigten Elemente des Basisprozesses enthalten. Je nach Anwendungsfall können dies Teile des Datenflusses oder Aktivitäten, die bearbeitet werden sollen, sein.

3.3 Schemaänderung

Eine Prozesssicht wird dazu verwendet, um das zugrunde liegende Schema direkt zu ändern. Dazu müssen verschiedene Bedingungen erfüllt sein:

1. Die View-Bildung muss strukturerhaltend erfolgt sein, d.h. auch der durch die View dargestellte Prozess muss eine gültige Prozessbeschreibung gemäß dem verwendeten Metamodell sein.
2. Die View Bildung muss auch kantenerhaltend sein. Bei Änderungen (Einfügen, Löschen, Verschieben) von Knoten und Kanten muss immer klar sein, auf welche Knoten und Kanten des Basisprozesses Bezug genommen wird, damit die Änderungen dort ebenfalls erfolgen können. Es dürfen in der View also keine Kanten vorhanden sein, die durch das Zusammenfallen mehrerer Kanten im Basisprozess entstanden sind.

4 Arten zur Erzeugung von Views

In den beiden vorherigen Kapitel wurde beschrieben, für welche Anwendungsfälle Views benötigt werden und welche Operationen über Views möglich sein sollten.

Views müssen sich einerseits für den speziellen Anwendungsfall eignen, anderseits auch die Bedingungen für die dann notwendigen Operationen erfüllen. Grundsätzlich lassen sich Views auf zwei verschiedene Arten erzeugen, einerseits durch **Aggregation** (Zusammenfassen mehrerer Elemente in einem neuen virtuellen Element), andererseits durch das Entfernen von Elementen (**Reduktion**).

Je nach Art der View-Erzeugung kann noch unterschieden werden, welcher Arten von Elementen eines Prozessgraphen betrachten werden. Es ergibt sich dann die auf der folgenden Seite in Abbildung 9 dargestellten Einteilung. Bei der Aggregation muss zwischen der Aggregation innerhalb eines Prozessgraphen und der Aggregation mehrerer Prozessgraphen, z.B. die Zusammenfassung mehrerer Instanzen des gleichen Schemas, unterschieden werden. Innerhalb eines Prozessgraphen können Aktivitäten, Attribute und Datenlemente aggregiert werden. Die Aggregation von Attributen und Datenelementen ist dabei nur für bereits aggregierte Aktivitäten sinnvoll. Somit können für einen virtuellen Knoten auch virtuelle Attribute und Datenelemente erzeugt werden. Bezüglich der Reduktion gibt es das Entfernen von Aktivitäten und den zugehörigen Kontrollflusskanten das Enfernen von Attributen sowie das Entfernen von Datenlementen (mit zugehörigen Datenflusskanten).

Grundsätzlich sind die zwei Methoden zur Viewerzeugung komplementär zueinander und können miteinander kombiniert werden. Bei der Aggregation können jedoch nur zusammenenhängende Teilgraphen aggregiert werden. Dies kann durch eine vorherigen Reduktion nicht mehr möglich sein, wenn dabei Teilgraphen entfernt wurden. Sinnvoll ist jedoch die Anwendung der Entfernung von Attributen und Datenelementen auch bei aggregierten Views, da damit geheime Informationen ausgeblendet werden können.

Besonders von Bedeutung ist die Aggregation von Aktivitäten und das Entfernen von Teilen des Kontrollflusses, da hiervon abhängt, welche Operationen auf den Views unterstützt werden. Für die Änderung von Instanzen ist es notwendig, dass einzelne Aktivitäten erhalten bleiben, deshalb kommt dort die Aggregation von Aktivitäten nicht in Frage, sondern lediglich das Entfernen nicht benötigter Aktivitäten. Bei der Änderung des Schemas ist es wichtig, dass die Struktur erhalten bleibt. Mit einer Aggregation ist dies besser durchführbar, da dort immer Teilmengen, die einen zusammenhängenden Teilgraphen bilden, aggregiert werden können. Diese aggregierten Teilmengen bilden zusammen wieder eine gültige Struktur.

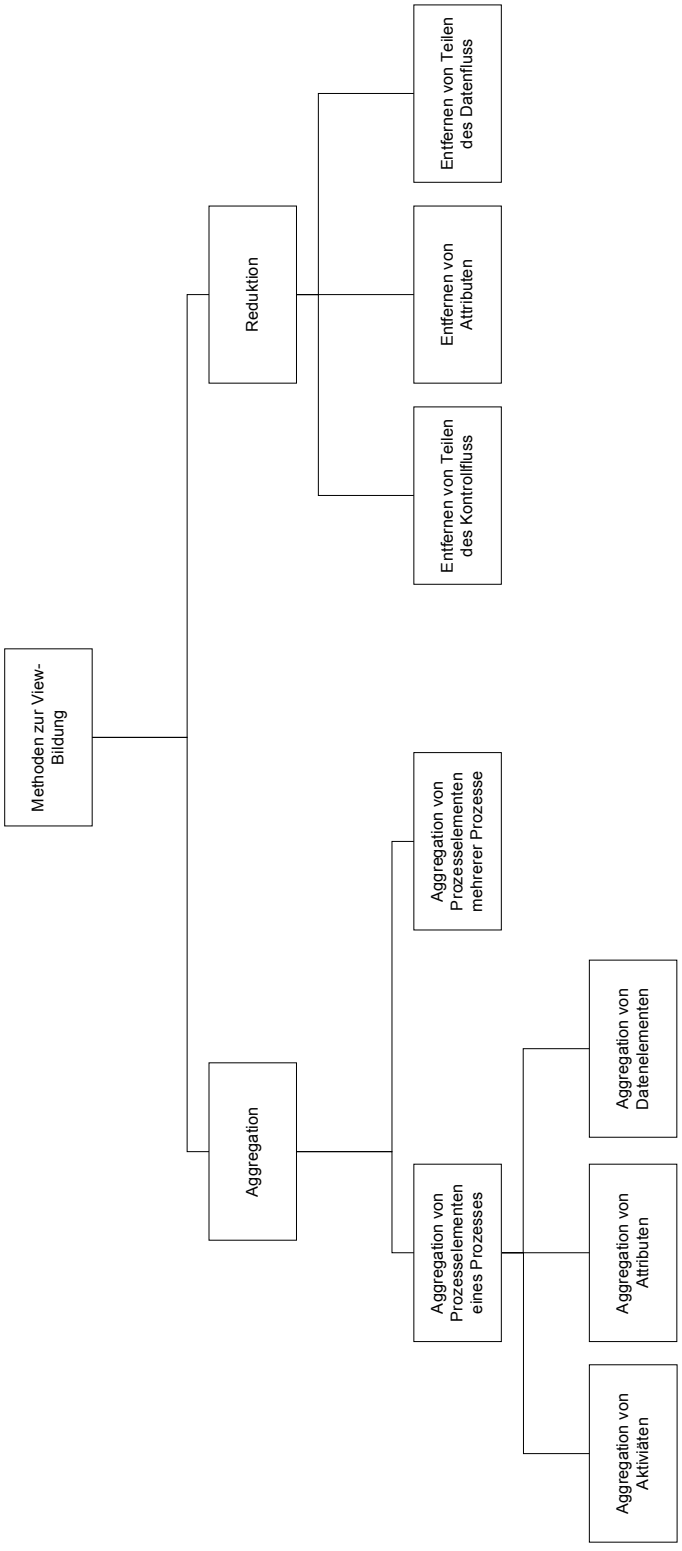

Abbildung 9: Methoden zur Viewbildung

4.1 Graphreduktion

Bei der Graphreduktion werden Elemente des Basisprozesses entfernt und, je nach Art des entfernten Elements, weitere Elemente begleitend gelöscht oder adaptiert. Beim Löschen von Aktivitäten etwa werden assoziierte Kontrollflusskanten und alle dem Knoten zugeordneten Attribute mit gelöscht. Das gleiche gilt für Datenelemente. Lediglich das Entfernen von Attributen zieht weitere Schritte nach sich.

4.1.1 Ausblenden von Attributen

Das Ausblenden von Attributen kann mit allen anderen Methoden zur Viewbildung kombiniert werden. Zudem kann es, in Abhängigkeit von Eigenschaften des Prozesses, automatisiert erfolgen, z.B. Entfernen aller Attribute, deren Wert größer X ist und die einen bestimmten Bezeichner haben. Durch das Entfernen von Attributen können geheime Informationen (z.B. Kosten, Dauer) versteckt werden.

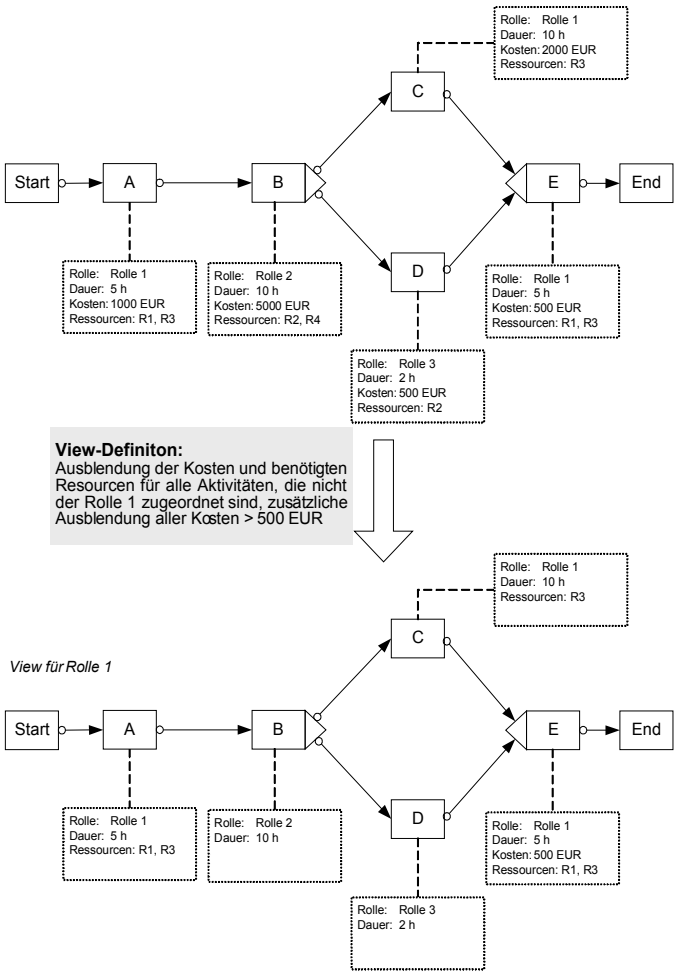

Abbildung 10: Verbergen von Attributen

Im Beispiel aus Abbildung 10 werden für eine Prozess in dem für jeden Knoten vier Attribute definiert sind, die Attribute „Kosten“ und „Ressourcen“ abhängig von gewissen Bedingungen („Kosten“ größer 500 EUR, „Ressourcen“ wenn der entsprechende Aktivitätenknoten nicht der Rolle 1 zugeordnet ist) ausgeblendet.

4.1.2 Entfernen Datenflusselementen

Auch das Entfernen von Datenflusselementen (Datenlemente und zugehörige Datenflusskanten) kann mit allen anderen Methoden zur Viewerzeugung kombiniert werden. Jedoch muss beachtet werden, welche Operationen später auf die View anwendbar sein sollen. Sollen Schemaänderungen auf Grundlage von Views definiert werden können (mit anschließender Propagation der Änderung auf das Schema des Basisprozesse) darf z.B. der Datenfluss nicht entfernt werden.

Wenn Datenknoten entfernt werden, müssen wie im Beispiel aus Abbildung 11 die zugehörigen Datenflusskanten auch entfernt werden.

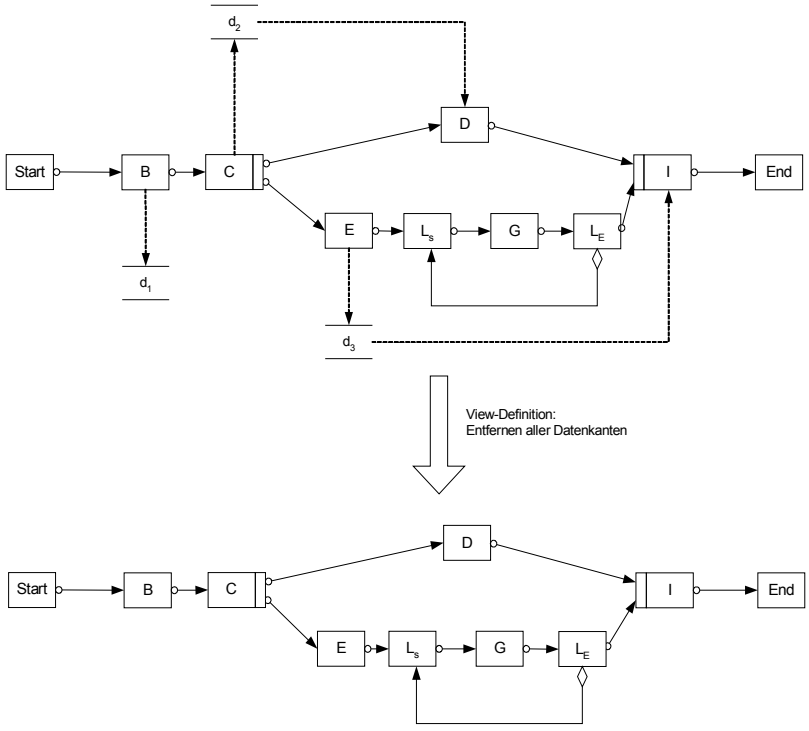

Abbildung 11: Ausblenden Datenfluss

4.1.3 Entfernen von Teilen des Kontrollflusses

Beim Entfernen von Kontrollflusselementen werden (meist) Aktivitäten aus dem Prozess gelöscht. Für eine Darstellung dieser Teilsicht müssen ein-/ausmündende Kontrollkanten des

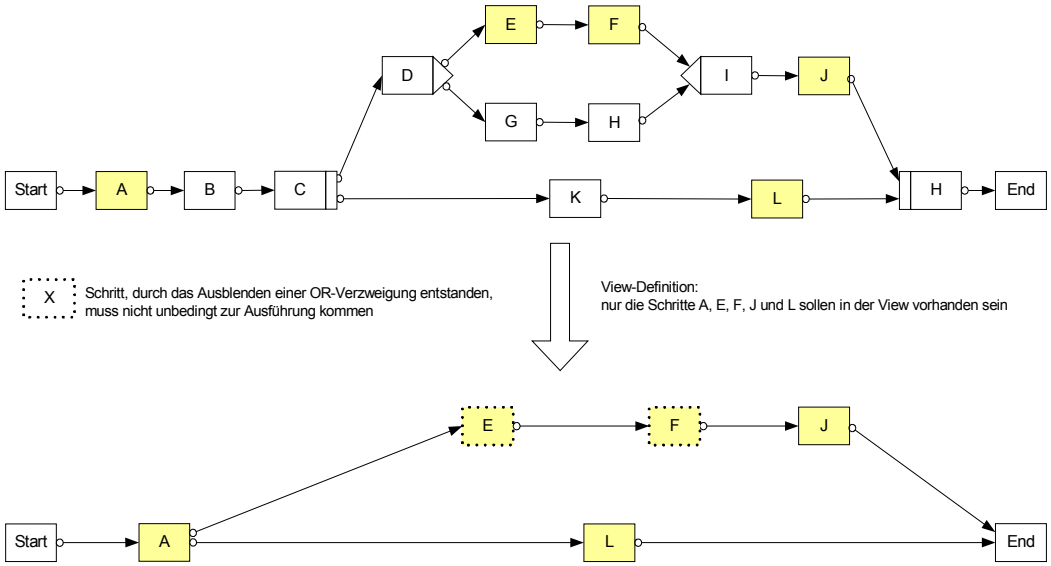

Abbildung 12: Viewbildung aufgrund der Auswahl relevanter Schritte

betreffenden Knotens nach dessen Löschung mit den verbleibenden Knoten verbunden werden (z.B. durch Zusammenfassen ein-/ausmündenden Kanten) damit wieder ein zusammenhängender Graph entsteht. Eine View, die mit dieser Methode gebildet wird, hat folgende Eigenschaften:

- Sie ist aktivitätenerhaltend, da nicht gelöschte Aktivitäten unverändert in der View vorhanden sind.
- Sie ist nicht kantenerhaltend, da die Kanten gelöschter Aktivitäten mit anderen Kanten „verschmolzen“ werden müssen.
- Je nach verwendetem Algorithmus kann die Sicht strukturerhaltend sein oder nicht (wenn nur für die Struktur unbedeutende Schritte entfernt werden).
- Sie eignet sich grundsätzlich für die Instanzenänderungen, nicht aber für die Schemaänderungen, da die View aktivitätenerhaltend aber nicht strukturerhaltend ist.
- Eine View-auf-View Bildung ist möglich (es können weitere Schritte gelöscht werden).

Durch Entfernen von Aktivitäten können Views für verschiedene Anwendungsfälle erzeugt werden. Eine Möglichkeit ist z.B. die Erzeugung von Views zwecks effizienter Bearbeitung von Prozessausschnitten in der Prozess-Engine (z.B. durch Ausblenden bereits beendeter Schritte). Nach demselben Prinzip kann auch eine View erzeugt werden, die für einen bestimmten Bearbeiter nur die für ihn relevanten Schritte[2] (wie im Beispiel aus Abbildung 12) des

[2] Dies können z.B. nur die Schritte sein, die einem Bearbeiter zugewiesen sind, oder Schritte deren Attribute einen bestimmten Wert haben.

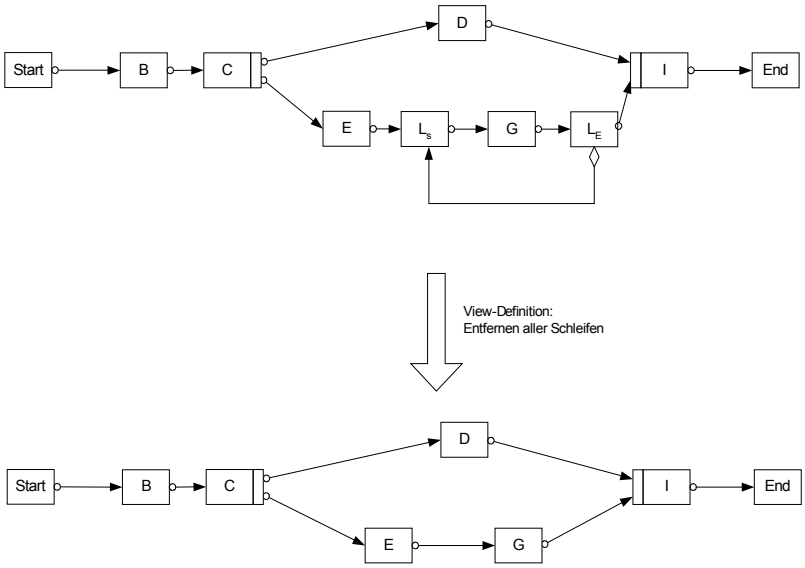

Abbildung 13: Ausblenden von Schleifen

Prozesses anzeigt. Insbesondere sind über die View auch Instanzänderungen möglich, indem z.B. ein Bearbeiter über die View einen Schritt weiterschaltet.

4.2 Graphaggregation

Bei der Aggregation von Elementen werden Teilmengen eines bestimmten Typs von Knoten (Aktivitäten, Attribute oder Datenelemente) oder ganzer Prozesse gebildet, diese Teilmengen werden dann durch einen virtuellen Knoten oder einen virtuellen Prozess ersetzt. Hierbei lassen sich vier Kategorien unterscheiden:

- Aggregation von Aktivitäten
- Aggregation von Attributen,
- Aggregation von Datenflusselementen
- Aggregation von Prozess-Instanzen

Im Gegensatz zur Graphreduktion ist zu beachten, dass die Aggregation von Attributen und Datenflusselementen nur begleitend mit der Aggregation von Aktivitäten sinnvoll ist. Dabei können dann für einen aggregierten Knoten auch die Aktivitäten sinnvoll, da nur dann die Attribute bzw. Datenelemente der in dem aggregierten Knoten enthaltenden Basisknoten zusammengefasst werden können.

4.2.1 Aggregation von Aktivitäten

Eine Teilmenge von Knoten wird durch einen virtuellen Knoten ersetzt. Dabei müssen die Knoten der Teilmenge mit allen Kontrollflusskanten, die als Start- und Endknoten nur Knoten

innerhalb der Teilmenge haben, einen zusammenhängenden Graphen bilden. Anschließend müssen für diesen abstrakten Knoten neue Kanten erzeugt werden, welche die ein- bzw. ausgehenden Kanten der Start- und Endknoten[3] der aggregierten Knotenmenge ersetzen.

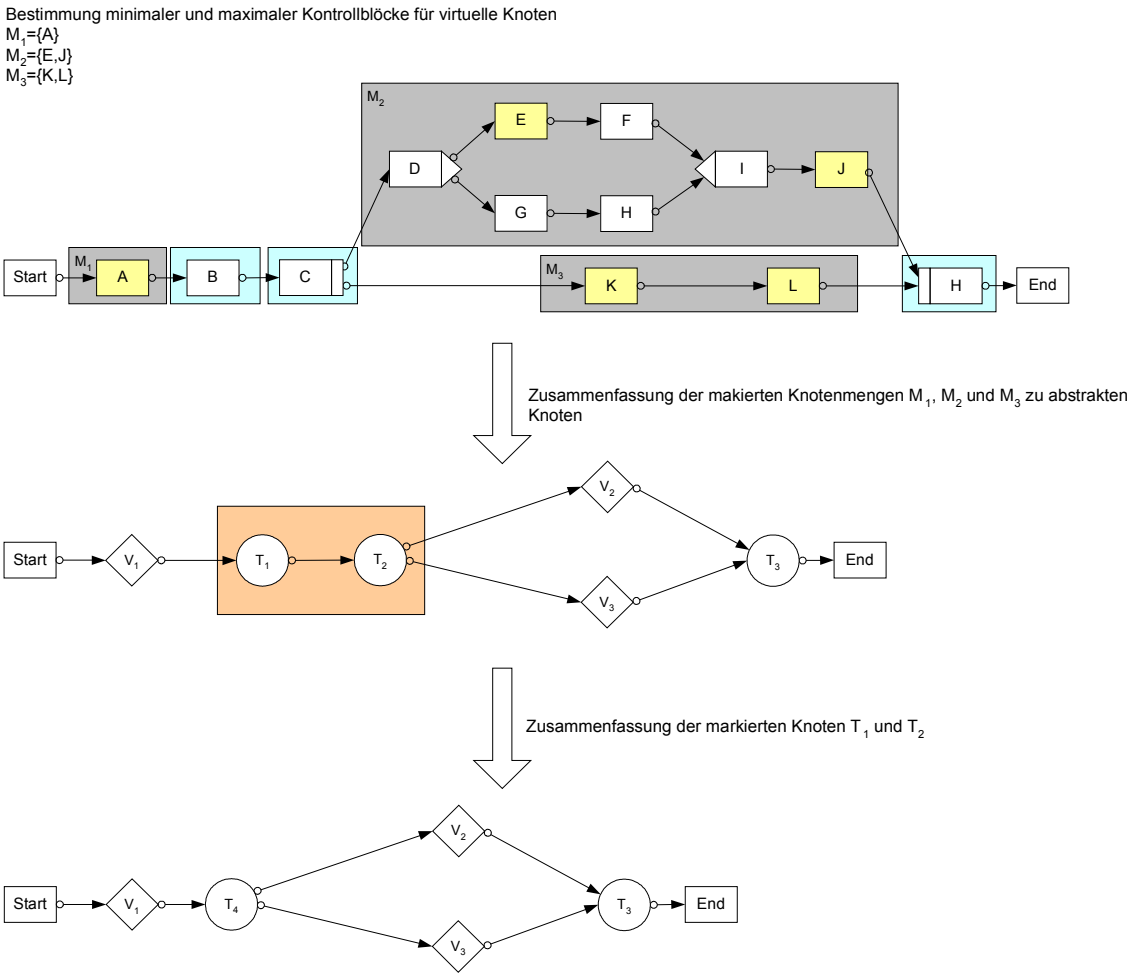

Abbildung 14: Beispiel für die Aggregation von Aktivitäten

Im Beispiel aus Abbildung 14 werden die Knoten-Teilmengen M_1, M_2 und M_3 zu den virtuellen Knoten V_1, V_2 und V_3 zusammengefasst. Anschließend erfolgt eine weiter Zusammenfassung der Knoten T_1 und T_2.

Die Eigenschaften einer mittels Aggregation erzeugten View hängen im wesentlichen vom verwendeten Algorithmus ab:

[3] Dies sind Knoten, deren Vorgänger- bzw. Nachfolgerknoten nicht in der Teilmenge liegen.

- Wenn immer nur gültige Teilstrukturen (z.B. komplette Verzweigung) des Basisprozesses (z.B. Kontrollblöcke bei blockstrukturierten Graphen) aggregiert werden, ist die View strukturerhaltend.
- Wichtig für spätere View- oder Schemaänderungen ist die eindeutige Zuordbarkeit von Kanten der View zu korrespondierenden Kanten des Basisprozesses. Damit ist es möglich genau zu bestimmen, zwischen welchen Knoten im Basisschema eine Änderungsoperation durchgeführt werden soll. Hierzu ist es notwendig, dass keine Kanten zusammenfallen. Dies kann z.B. dadurch erreicht werden, dass die aggregierte Teilmenge zu jedem anderen Knoten in der View jeweils nur eine ein-/ausgehende Kante besitzt. Dann sind die in den virtuellen Knoten ein-/ausgehenden Kanten eindeutig den Kanten des Basisprozesses zuordbar.
- Eine View-auf-View Bildung ist immer möglich, es können auch virtuelle Knoten aggregiert werden.

Die Viewbildung mittels Aggregation ermöglicht verschiedene Anwendungsfälle. So kann die Komplexität des Prozessgraphen verringert werden. In Kombination mit der Aggregation von Attributen können zudem Informationen verborgen werden. Schließlich kann durch eine solche View auch das Schema der View und infolgedessen das zugrunde liegende Basisschema geändert werden, vorausgesetzt die Viewerzeugung erfolgte strukturerhaltend.

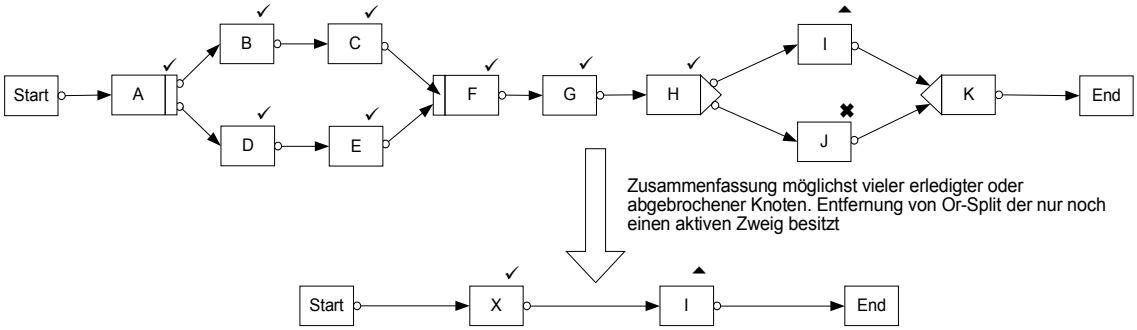

Abbildung 15: Beispiel für Verkleinerung einer Prozess-Instanz

Es ist auch möglich, solche Views zwecks Effizienzsteigerung in der Prozess-Engine einzusetzen (typischerweise in Kombination mit einer Graphreduktion. Wir kommen hierauf in Abschnitt 5.3.8 zurück).

4.2.2 Aggregation von Attributen

Wenn mehrere Knoten, wie im vorherigen Abschnitt beschrieben, aggregiert werden, sind die Informationen zu Attributen dieser einzelnen Knoten in der View nicht mehr verfügbar. Der Grund ist, dass diese Knoten durch einen virtuellen Knoten ersetzt werden. Sollen diese

Informationen in „verdichteter Form“ auch für den virtuellen Knoten zur Verfügung stehen, ist es notwendig, die Werte der Attribute auf geeignete Art zusammenzufassen und dann als virtuelle Attribute an dem erzeugten virtuellen Knoten zur Verfügung zu stellen. Wenn z.B. für jede Aktivität die Kosten für ihre Ausführung als Attribut zur Verfügung stehen, können für einen virtuellen Knoten die Ausführungskosten der aggregierten Aktivitäten addiert werden und die Summe als virtuelles Attribut für den virtuellen Knoten verfügbar gemacht werden.

Ein Problem stellt in diesem Kontext die Behandlung der Attribute von Knoten innerhalb von alternativen Zweigen dar, da hier nicht im Voraus bekannt ist, welcher Zweig durchlaufen wird. Wenn wie im Beispiel im vorherigen Absatz Kosten von Aktivitäten aufsummiert werden sollen, ergibt sich bei alternativen Verzweigungen das Problem, wie die Kosten in den verschiedenen Zweigen für die Gesamtsumme gewichtet werden sollen. Hier kann nur mit Hilfe von Heuristiken oder der statistischen Auswertung bereits durchlaufener Instanzen eine vernünftige Aggregation durchgeführt werden. Diese Problematik besteht natürlich nicht, wenn die View auf bereits durchlaufene Pfade einer Prozess-Instanz angewendet wird.

Je nach Datentyp der Attribute sind verschiedene Möglichkeiten zur Aggregation und Transformation denkbar. Dabei versteht man unter Aggregation die Abbildung mehrerer Attributwerte (z.B. Dauer von Aktivitäten) auf einen Attributwert des virtuellen Knotens mit Hilfe einer Abbildungsfunktion, bei der Transformation wird hingegen eine Wert durch eine Abbildungsfunktion in einen anderen Wert überführt:

- Aggregation von mehreren Attributwerten:
  - Bei numerischen Typen können die üblichen, von Datenbanken her bekannten Aggregationsfunktionen (SUM, AVG, MIN, MAX, usw.) verwendet werden.
  - Bei Rollenzuordnungen kann eine Mengenbildung vorgenommen werden. Es wird dann die Menge aller Rollenzuordnungen (der zusammengefassten Knoten) als aggregiertes Ergebnis dargestellt wird. Wurden z.B. zwei Aktivitäten aggregiert, bei der die eine Aktivität von Rolle A und die andere von Rolle B ausgeführt wird, kann dann als aggregierte Attributwert die Menge {Rolle A, Rolle B} verwendet werden.
  - Bei Attributen, die einen endlichen Wertebereich besitzen (z.B. Aktivitätenzustand bei Prozess-Instanzen), kann eine Aufsummierung der Zustände erfolgen (vgl. Beispiel aus Abbildung 20). Falls es möglich ist, für die einzelnen Werte eine Priorisierung vorzunehmen, können die verschiedenen Werte, wie bei numerischen Attributen, auf einen Wert abgebildet werden. Wenn z.B. für eine Zustandsschritt nur die Werte „beendet“ oder „nicht beendet“ möglich sind, wird der Wert des aggregierten Attributs auf „beendet“ gesetzt, wenn alle aggregierten Knoten den Attributwert „beendet“ haben, ansonsten wird der Wert „nicht beendet“ gewählt. In Abschnitt 5.3.6

beschreiben wir, wie eine solche Mappingfunktion für die Zustände von Prozess-Instanzen aussieht.

- Transformation eines aggregierten Attributwertes:
  - Sollen auf einer Sicht zwar Informationen über die Attributwerte bekannt gemacht werden, jedoch genaue Daten verborgen bleiben, ist es möglich, im Anschluss an eine Aggregation noch eine Transformation der Attributwerte durchzuführen. So kann z.B. bei dem Attribut Kosten erst eine Durchschnittsbildung erfolgen, und dann eine Transformation in einen anderen Wertebereich. Liegen die Kosten z.B. unter 5.000 EUR, dann könnte das entsprechende Attribut zum Wert „günstig“ transformiert werden, zwischen 5.000 und 10.000 EUR zum Wert „normal“ und für Kosten größer als 10.000 EUR zum Wert „teuer“. Neben dem Verbergen von Informationen hat diese Art der Transformation auch den Vorteil, dass sich damit die Übersichtlichkeit für Views verbessert wird.
  - Falls die Attributwerte hierarchisch (z.B. bei Organisationsmodellen in Unternehmen) angeordnet sind, können Vorkommen mehrerer Werte auf einer Ebene durch den Wert der darüberliegenden Stufe ersetzt werden. Gibt es z.B. in einem Benutzerverwaltungssystem eine Rolle 1, mit darunterliegenden spezielleren Rollen A, B und C, und sind alle Aktiviäten eines aggregierten Knoten den Rollen A oder B zugeordnet, so kann man den Wert des Attribut statt auf die Menge der Rollen A und B (siehe vorherigen Absatz) auch auf den Wert der darüber liegenden gemeinsamen Rolle 1 setzen.

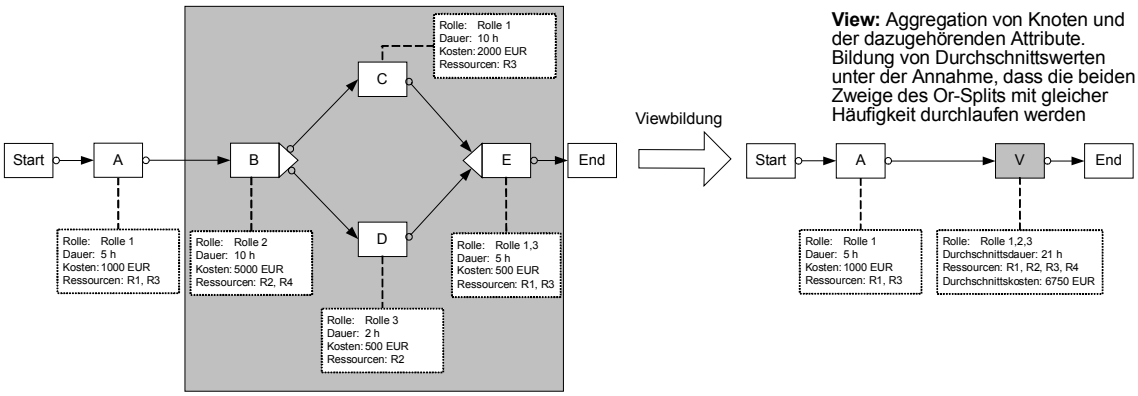

Abbildung 16: Knoten und Attributaggregation

4.2.3 Aggregation von Datenelementen

Datenflüsse sind bei der Aggregation von Aktivitäten in der View zunächst nicht sichtbar. Ob und in welcher Form sie für eine virtuelle Aktivität dargestellt werden, hängt vom jeweiligen

Anwendungsfall ab. Einige Möglichkeiten zur Aggregation von Datenflüssen werden in [5] beschrieben. Grundsätzlich müssen folgende Aspekte beachtet werden:

- **Fall 1:** Der Datenfluss bleibt innerhalb der virtuellen Aktivität (d.h. Datenflusskanten haben nur mit Aktivitäten aus der aggregierten Knotenmenge eine Verbindung). Wenn es um Änderungen von Prozess-Schemata geht, können solche Datenflüsse komplett verborgen werden, müssen also nicht angezeigt werden. Im Beispiel aus Abbildung 17 werden zwei virtuellen Knoten erzeugt. Die Datenelemente d_1 und d_2 werden jeweils nur von Aktivitäten innerhalb eines der virtuellen Knoten geschrieben und gelesen und können deshalb ausgeblendet werden.

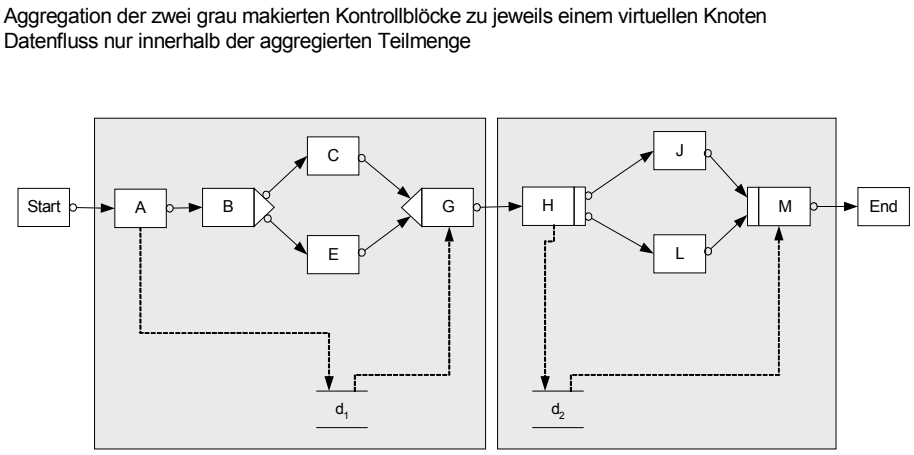

Abbildung 17 :Datenfluss innerhalb einer aggregierten Teilmenge

- **Fall 2:** Es sind auch Aktivitäten außerhalb der virtuellen Aktivität betroffen. Abhängig von der „Richtung“ des Datenflusses, können deshalb Aktivitäten außerhalb der virtuellen Aktivität von der: Dateneingabe und -ausgabe von Aktivitäten innerhalb einer virtuellen Aktivität abhängig sein. Dieser Datenfluss muss dann auch in der View vorhanden sein, insbesondere wenn auf dieser View später Schemaänderungen durchführbar sein sollen. Im Beispiel aus Abbildung 18 wird das Datenelement d_2 von einer Aktivität des einen virtuellen Knoten beschrieben und von einer Aktivität des anderen virtuellen Knoten gelesen.

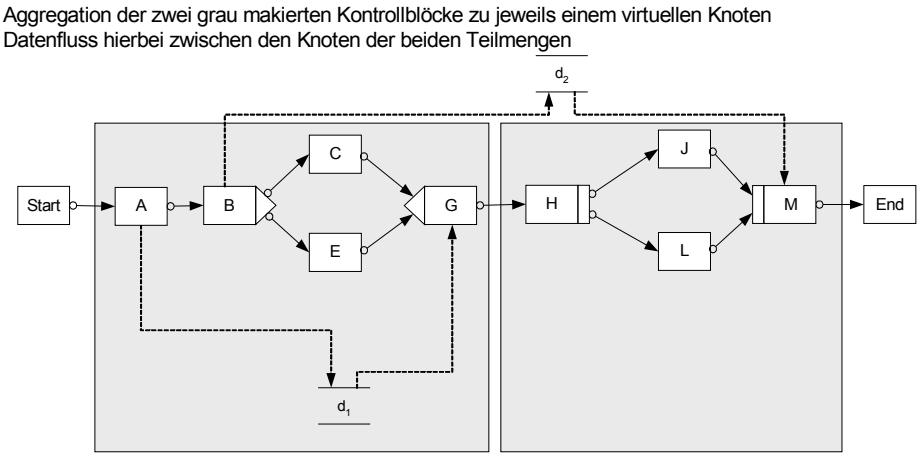

Abbildung 18: Datenfluss zwischen zwei aggregierten Teilmengen

Für die Aggregation von Datenelementen gibt es folgende Möglichkeiten:

- Auch für den virtuellen Knoten werden alle Datenelemente angezeigt. Dies gilt für Elemente, die sowohl innerhalb als auch außerhalb der virtuellen Aktivität liegen. Damit können detaillierte Informationen über den Datenfluss auch für aggregierte Aktivitäten geliefert werden, worunter aber die Übersichtlichkeit leiden kann, vor allem wenn es viele Datenein-/Datenausgaben innerhalb einer virtuellen Aktivität gibt. Im Beispiel in Abbildung 19 wird im Graphen aus Abbildung 19a) der gesamte Datenfluss zwischen und innerhalb der virtuellen Knoten X und Y dargestellt. An diesem Beispiel wird deutlich, dass bei aggregierten Aktivitäten, die viele Basisaktivitäten beinhalten, die Darstellung des kompletten Datenfluss schnell unübersichtlich wird, da dann mehr Datenelemente als (virtuelle) Aktivitätenknoten vorhanden sind.
- Es werden die Datenelemente, deren assoziierte Kanten nur innerhalb der aggregierten Aktivitätenmenge liegen, verborgen. Die dann verbleibenden Datenflussteile sind ausreichend, um das Kontrollfluss-Schema auch über die View ändern zu können (siehe Kapitel 8.3). Ein Problem ergibt sich, wenn über die View auch das Datenschema geändert werden soll, da – je nach aggregierter Aktivitätenmenge – mehrere Datenflusskanten zusammenfallen können (z.B. wenn das gleiche Datenelement in unterschiedlichen Verzweigungen geschrieben oder gelesen wird). In einem solchen Fall ist es nicht mehr möglich, zu bestimmen, welche Datenflusskante geändert werden soll. Ein weiteres Problem ergibt sich dadurch, dass bei einer Schreiboperation nicht zwingend ein Datenelement einer virtuellen Aktivität geschrieben werden muss. Dies ist z.B. der Fall, wenn die Datenflusskante nur von einem Zweig einer XOR-Verzweigung zu dem entsprechendem Datenelement führt. In der View sieht es dann so aus, als ob das Datenelement immer vom virtuelle Knoten geschrieben wird. Hier gibt es die

Möglichkeit, mit Hilfe eine Funktion zu bestimmen, ob ein Element auf jeden Fall von einem virtuellen Knoten beschrieben wird (obligat). Das Ergebnis kann dann in der View z.B. graphisch verdeutlicht werden (z.B. durch gestrichelte Datenflusskanten, falls Elemente nur in Zweigen einer XOR-Verzweigung geschrieben werden). In Abbildung 19 ist dies im Beispiel b) dargestellt. Dort ist nur noch der Datenfluss zwischen den virtuellen Knoten X und Y sichtbar, die Datenflusskante die von einem Knoten innerhalb eines Teilzweiges einer XOR-Verzweigung ausgeht, ist gesondert markiert.

- Der Datenfluss könnte noch weiter eingeschränkt werden, in dem nur noch die Elemente angezeigt werden, die zum Starten einer virtuellen Aktivität notwendig sind bzw. die nach Beendigung der virtuellen Aktivität als Ausgabe (interne Knoten) feststehen. Mehrere Datenelemente können, die den gleichen Quell- und Zielaktivitätenknoten haben, können dabei durch einen virtuelles Datenelement ersetzt werden. Damit ist keine Schemaänderung auf der View mehr realisierbar, es sind jedoch Informationen über den Datenfluss beim Start und Ende einer virtuellen Aktivität möglich. Dies wird in Abbildung 19 im Beispiel unter c) demonstriert. Der komplette Datenfluss zwischen X

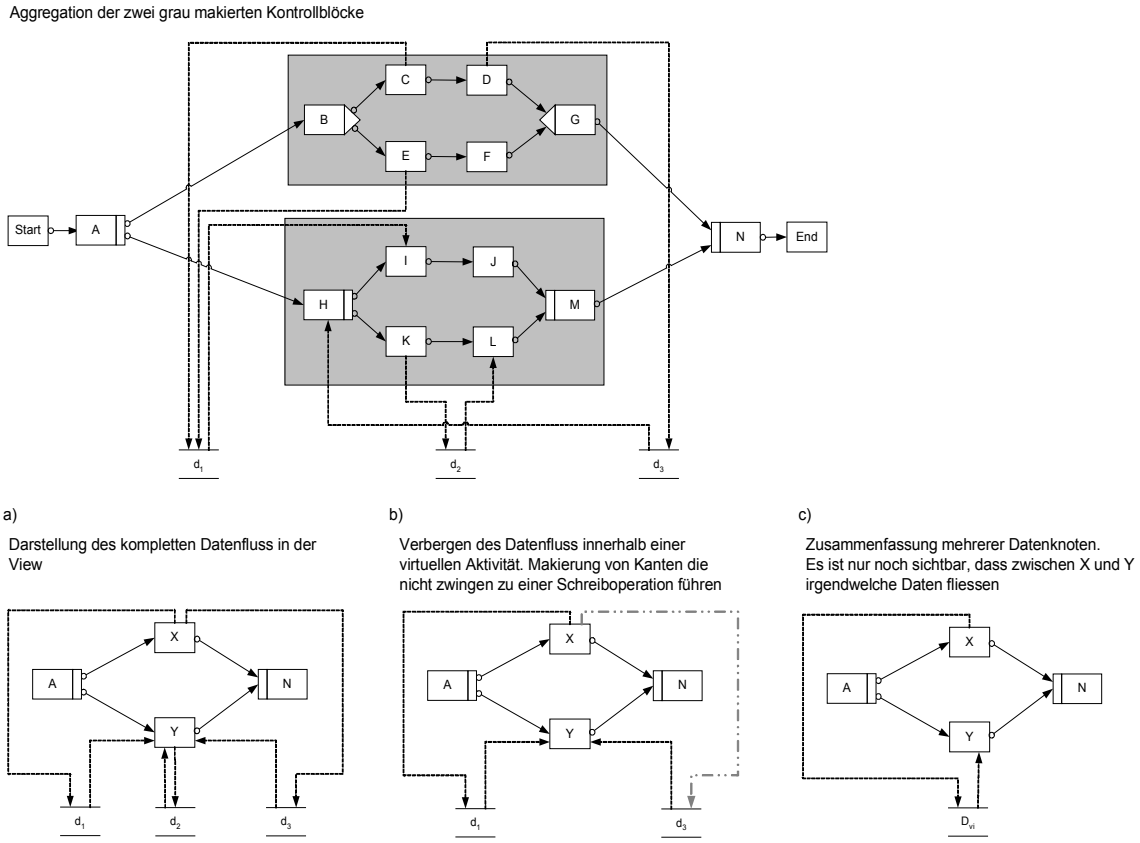

Abbildung 19: Verschiedene Möglichkeiten der Datenflussaggregation

und Y wird nur noch durch ein virtuelles Datenelement D_{vi} dargestellt. Es ist nicht mehr erkennbar, welche konkreten Daten von X nach Y geliefert werden.

- Welche Datenelemente angezeigt und verborgen werden, kann auch manuell festgelegt werden, um z.B. spezifische Informationen in einer View zu erhalten.

4.2.4 Aggregation von Prozess-Instanzen

Bei der Aggregation von Prozess-Instanzen wird eine Sicht gebildet, die mehrere Instanzen auf einen Prozessgraph abbildet. Damit kann z.B. eine Gesamtübersicht zu laufenden Prozessen eines bestimmten Typs erstellt werden. Zustände der einzelnen Aktivitäten können aufsummiert werden. Attributwerte können, wie in Verbindung mit der Aggregation von Attributen beschrieben[4], aggregiert werden. Eine solche View hat folgende Eigenschaften:

- Sie eignet sich nicht zur Instanzenänderung, da mehrere Instanzen zusammengefasst wurden
- Auf sie können anschließend mit Hilfe anderer Methoden weitere View-Bildungen angewendet werden.
- Durch das Zusammenfassen mehrerer Instanzen in einer View können durch Schemaänderungen, die auf der View durchgeführt werden, eine ganze Menge von Instanzen auf einmal geändert werden.

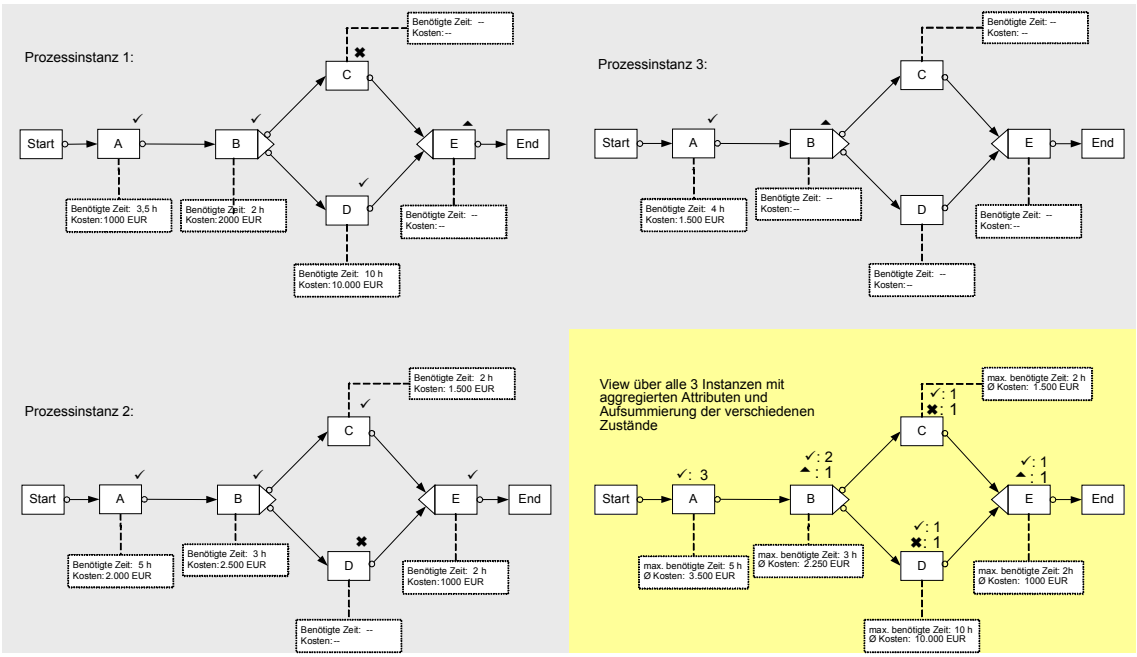

Abbildung 20: Zusammenfassung mehrerer Prozess-Instanzen

[4] Hier jetzt aber für die gleichen Aktivitäten in verschiedenen Instanzen.

Im Beispiel in Abbildung 20 wird für drei Prozess-Instanzen gleichen Typs eine View gebildet. Der Graph der View ist der gleiche wie im Fall der einzelnen Instanzen. Für jede Aktivität werden Zustände und Attribute in aggregierter Form dargestellt. Grundlage hierfür ist die Verwendung einer geeigneten Aggregationsfunktion (siehe Abschnitt 4.2.2). Dabei werden für Attribute eine Durchschnitts- und Maximumsbildung sowie für Zustände eine Aufsummierung vorgenommen.

4.3 Verknüpfung mehrerer Prozesse in einer View

Mit Hilfe der Aggregation und Reduktion von Graphen lassen sich für einen Prozess unterschiedliche Views für verschiedene Anwendungsfälle definieren.

Ein weiterer Anwendungsfall ist die Verknüpfung mehrerer Prozesse in einer View. Diese beschreibt dann z.B. einen virtuellen (interorganisationellen) Workflow, bei dem ein Hersteller seinen Prozess mit denen seiner Lieferanten zusammenführen will. Dabei sind die vorgestellten Ansätze der Aggregation und Reduktion nicht mehr ausreichend, da dort jeweils ein Graph verändert wurde, jetzt aber mehrere Graphen zu einem zusammengefasst werden müssen.
Der resultierende Graph sollte wieder ein gültiger Prozess im Sinne des verwendeten Metamodells, mit denen die Basisprozesse definiert werden, sein. Nur dann können auf diesem Graph wieder verschiedene Views gebildet werden. Demzufolge darf es auch im resultierenden Graphen jeweils nur einen Start-/Endknoten geben, und die verschiedenen Teilprozesse in der View haben dann als Start- bzw. Endknoten normale Aktivitätenknoten von anderen Teilprozessen. Außer dass die Start-/Endknoten der Prozesse durch normale Aktivitätenknoten anderer, in der View verwendeten Prozesse, ersetzt werden, können auch Aktivitäten verschiedener Prozesse zusammenfallen und in der View durch einen einzelnen Knoten repräsentiert werden.

Für die Integration mehrerer Prozesse ist es notwendig, die Semantik der einzelnen Prozesse und Aktivitäten zu kennen. Deshalb ist es schwierig, einen Algorithmus für die automatische Generierung solcher integrierten Views anzugeben. Eine Integration muss deshalb von einem Prozess- bzw. View-Designer manuell durchgeführt werden, indem er z.B. angibt, auf welchen Aktivitäten ein anderer Prozess folgt oder welche Aktivitäten verschiedener Prozesse zusammenfallen. Dabei können in der View auch neue Knoten eingefügt werden, die in den ursprünglichen Prozessen nicht vorhanden waren, z.B. zusätzliche Verzweigungsknoten für den Fall dass verschiedene Teilprozesse parallel ablaufen können.

Anhand des Beispiels aus Abbildung 21 wird dies deutlich. Dort werden drei Prozess-Schemata in einer View zusammengefasst. In der View läuft Prozess P_2 parallel zu den Prozessen P_1 und P_3 ab, deshalb sind bei der View virtuelle Verzweigungsknoten eingefügt worden. Prozess P_1 und P_3 überlappen sich in dem Beispiel und P_3 wird als Teilzweig in der Verzweigung von P_1 statt eines einzelnen Knoten eingefügt.

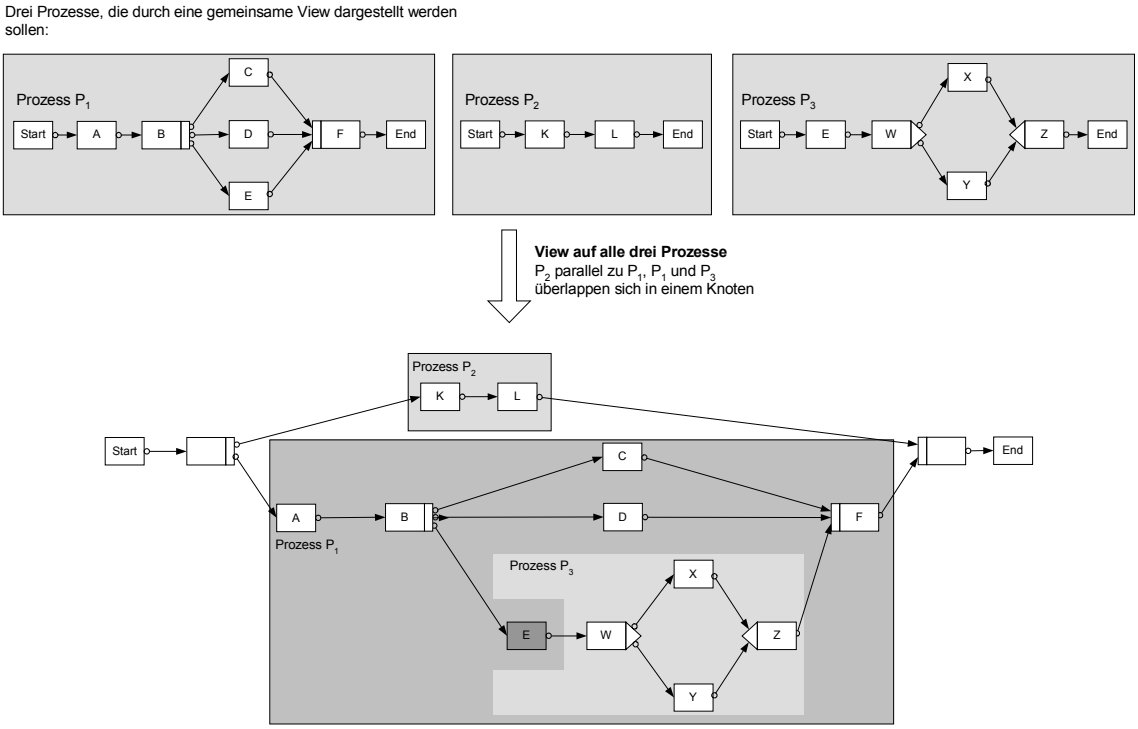

Abbildung 21: View über 3 Prozesse

Sobald eine solche View definiert wurde, ist es wieder möglich, mit Hilfe von Aggregation oder Reduktion des (jetzt einzelnen) Graphen weitere Views für verschiedene Anwendungsfälle zu erzeugen (View-auf-View). Damit kann auch überprüft werden, ob die integrierte View noch die Struktur der einzelnen Teilprozesse wiedergibt. Dies erfolgt, indem für jeden Teilprozess eine View auf der integrierten View gebildet wird, die nur die Knoten des Teilprozesses enthält. Diese Views müssen strukturell, bis auf durch die Integration zusätzlich entstandene Verzweigungsknoten, genau dem jeweiligen Teilprozess entsprechen.

5 View-Bildung

In diesem Kapitel wird gezeigt wie man für Prozessgraphen Views definieren kann. Anschließend werden Algorithmen für die Erzeugung verschiedener View-Arten vorgestellt.

5.1 Modell für Prozess-Schema und Prozess-Instanz

Kontrollflussgraphen sind Graphen mit spezieller Erweiterung. Diese Graphen besitzen unterschiedliche Knoten- und Kantentypen, sind also attributiert. Auf Instanzebene sind sie zusätzlich noch um Zustandsmarkierungen angereichert. Für die Beschreibung eines Prozess-Schemas oder einer Prozess-Instanz wird ein formales Modell benötigt. In dieser Arbeit wird dazu das ADEPT-Metamodell [4, 6] verwendet. Ein Kontrollfluss-Schema ist dort wie folgt definiert:

Definiton 1 (Kontrollfluss-Schema CFS)
Ein Kontrollfluss-Schema (*control flow schema)* CFS = (*N, E, D, NT, NP,* V_{out}*,* V_{in}*, DP, EC, Template*) ist ein Tupel mit:

- *N* ist eine endliche Menge von (Aktivitäten-)Knotenbezeichnern.
- $E \subseteq N \times N \times EdgeTypes$ ist ein endliche Menge von gerichteten Kanten denen ein Typ aus der Menge EdgeTypes := {`CONTROL_E, LOOP_E, SOFT_SYNC, STRICT_SYNC, FAILURE_E, PRIORITY_E`} zugeordnet ist. Mit `CONTROL_E` werden Kontrollflusskanten bezeichnet, mit `LOOP_E` Kanten zwischen Loop-Knoten mit `SOFT_SYNC` und `STRICT_SYNC` Synchronisationskanten, `FAILURE_E` und `PRIORITY_E` sind Spezialkanten für Priorisierung und Fehlerbehandlung.
- Abbildung $NT : N \rightarrow NodeTypes$ die jedem Knoten aus N einen Knotentyp aus der Menge NodeTypes := {`STARTFLOW, ENDFLOW, ACTIVITY, NULL, STARTLOOP, ENDLOOP`} zuordnet. `STARTFLOW` und `ENDFLOW` sind die Start- bzw. Endknoten eines Prozessgraphen, `ACTIVITY` ist ein normaler Aktivitätenknoten, `NULL` bezeichnet die Null-Aktivität und mit `STARTLOOP` und `ENDLOOP` werden die Start- und Endknoten einer Schleife bezeichnet.
- Abbildung $NP : N \rightarrow Priority$ die jedem Knoten aus N einen Knotenpriorität aus der Menge Priority := {`REGULAR, EXCEPTIONAL`} zuordnet.
- Abbildung $V_{out} := N \rightarrow OutBehaviour$ die jedem Knoten aus N eine Ausgangssemantik aus der Menge OutBehaviour := {`ONE_Of_ONE, One_Of_ALL, ALL_Of_ALL, NONE`} zuordnet. Bei `ONE_Of_ONE` hat ein Knoten eine ausgehende Kante, bei `One_Of_ALL` handelt es sich um einen XOR-Splitknoten und bei `ALL_Of_ALL` um einen AND-Splitknoten.

- Abbildung $V_{in} := N \rightarrow InBehaviour$ die jedem Knoten aus N eine Eingangssemantik aus der Menge InBehaviour := {ONE_Of_ONE, One_Of_ALL, ALL_Of_ALL, NONE} analog zur Menge OutBehaviour zuordnet. Bei One_Of_ALL handelt es sich um einen XOR-Joinknoten und bei ALL_Of_ALL um einen AND-Joinknoten.
- Abbildung $DP : N \rightarrow D$ die jedem XOR-Splitknoten aus N einen Entscheidungsparameter aus der Menge der Datenlemente D zuordnet.
- Abbildung $EC : E \rightarrow EdgeCode \cup \{UNDEFINED\}$ die jeder Kante eine Wert aus der Menge EdgeCode zuordnet.
- Abbildung $Template : N \rightarrow Templates \cup \{NULL\}$ die jedem Knoten eine Vorlage aus der Menge Templates zuordnet. Null-Aktivitäten wird NULL zugeordnet.

Mit Hilfe der obigen Definition kann der Kontrollfluss für einen Prozess modelliert werden. Damit ein Tupel CFS ein gültiger Prozessgraph ist, müssen folgenden Eigenschaften gelten (hier nur informell dargestellt):

- Der durch die Knotenmenge N sowie die Kontrollfluss- und Sync-Kanten (Kanten e ∈ E mit ET(e) = CONTROL_E oder ET(e) = STRICT_SYNC oder ET(e) = SOFT_SYNC) induzierte Teilgraph ist azyklisch. Deshalb kommt es bei der Ausführung nicht zu Deadlocks.
- Für jeden Splitknoten gibt es genau einen entsprechenden Joinknoten (das gleich gilt für Loop-Start- und Loop-End-Knoten).
- CFS ist blockstrukturiert, d.h. Kontrollblöcke wie Sequenzen, Verzweigungen und Schleifen können ineinander verschachtelt sein, dürfen sich jedoch nicht überlappen

Weitere nützliche Restriktionen werden in [4] beschrieben. Wir verzichten an dieser Stelle auf eine umfassende Darstellung.

In Abbildung 22 ist ein Beispiel für ein Kontrollfluss-Schema P gegeben. Der Graph besteht aus der Knotenmenge N = {Start, A, B, C, D, E, End}. Alle Kanten haben den Typ CONTROL_E. Die Kante zwischen Start und A etwa wird durch das Tupel (Start, A, CONTROL_E) repräsentiert. Bis auf die Verzweigungsknoten (B und E) der XOR-Verzweigung haben alle Knoten die Eingangs- und Ausgangssemantik ONE_Of_ONE. Der Knotentyp ist bis auf die Start- und Endknoten jeweils ACTIVITY, für den Startknoten ist der Typ NT(Start) = STARTFLOW und entsprechend für den Endknoten NT(End) = ENDFLOW.

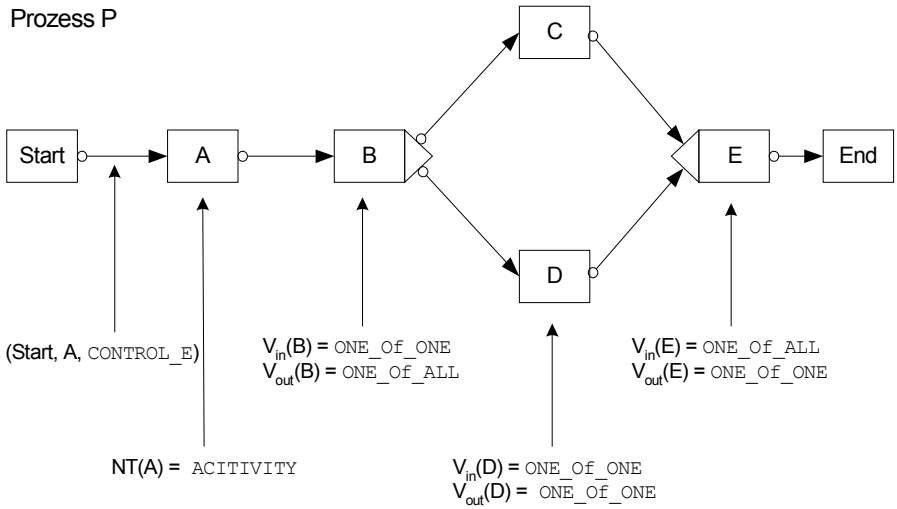

Abbildung 22: Beispiel für CFS

Der Datenfluss für ein Kontrollfluss-Schema kann mit Hilfe eines Datenfluss-Schemas *DFS* definiert werden:

Definiton 2 (Datenschema DFS)
Sei durch CFS ein Kontrollfluss-Schema gegeben. Eine Datenflusskante wird durch das Tupel dfc=(d,x,par,access_mode) beschrieben. Dabei ist $d \in D$, $x \in N$, $par \in InParams^x \cup OutParams^x$, und acces_mode $\in \{$`read`, `write`$\}$.
$InParams^x$ sei dabei die dem Knoten x zugeordneten Menge der Eingabeparameter. Ein Parameter p sei dabei (vereinfacht) ein Tupel $p= (p_{name}, dom^p)$, d.h. jeder Parameter habe einen eindeutigen Bezeichner p_{name} und einen Wertebereich dom^p. Analoges gilt für $OutParams^x$, hierbei werden Ausgabeparameter für den Knoten x beschrieben.
Mit Hilfe access_mode wird angegeben ob mit der entsprechenden Kante schreiben (`write`) oder lesend (`read`) auf das Datenelement d zugegriffen wird. Mit Hilfe einer Datenflusskante werden Ein- oder Ausgabeparameter eines Knotens mit einem Datenelement verküpft.
Die Menge aller Datenflusskanten für ein Kontrollfluss-Schema CFS wird als Datenschema (data flow schema) DFS bezeichnet.

Mit Hilfe der vorherigen Definitionen lässt sich jetzt der Kontrollfluss und der Datenfluss für ein Prozess-Schema definieren. Eine Prozess-Instanz für eine Prozess-Schema wird folgendermaßen definiert:

Definiton 3 (Prozess-Instanz basierend auf einem Kontrollfluss-Schema)
Eine Prozess-Instanz $CFS_{instance}$ wird durch das Tupel $CFS_{instance}$ = (CFS, M^{CFS}, $Val^{CFS,}$) beschrieben. Dabei ist:

- CFS = (N, E, D, ...) das Kontrollfluss-Schema auf dem die Instanz $CFS_{instance}$ basiert
- M^{CFS}=(NS^{CFS}, ES^{CFS}) enthält die Knoten und Kantenmarkierungen für $CFS_{instance}$.
  Dabei ist $NS^{CFS} : N \rightarrow NodeStates$ eine Abbildung die jedem Knoten aus N einen der Zustand aus der Menge NodeStates := {NOT_ACTIVATED, ACTIVATED, SELECTED, STARTED, COMPLETED, SKIPPED} zuweist und $ES^{CFS} : N \rightarrow EdgeStates$ eine Abbildung die jeder Kante aus E einen Zustand aus der Menge EdgeStates := {NOT_SIGNALED, TRUE_SIGNALED, FALSE_SIGNALED} zuweist.
- Val^{CFS} ist eine Funktion die jedes Datenlement aus der Menge D auf den aktuellen Wert von D, bzw. `UNDEFINED` falls dieser noch nicht gesetzt wurde, abbildet

Mit Hilfe obiger Definition lassen sich jetzt für ein Prozess-Schema beliebig viele Instanzen bilden. Eine genauere Erklärung der *NodeStates* und der Zustandsübergänge erfolgt in Abschnitt 5.3.6.

5.2 Erweiterung des Grundmodells

Eine Möglichkeit, Views für Graphen mit Hilfe einer Algebra zu definieren wird in [7] beschrieben. Die dortigen Operationen eignen sich jedoch nur bedingt für die Erzeugung von Views auf Prozessgraphen, da keine Knoten- und Kantentypen unterschieden werden, und somit z.B. keine besondere Behandlung von Verzweigungen oder Schleifen erfolgt.

Bei der Viewbildung geht es generell darum Teilmengen der oben genannten Elemente des ADEPT-Modells zu verbergen (oder in bestimmten Anwendungsfällen zu löschen) bzw. durch einen Platzhalter zu ersetzen. Eine View kann sowohl auf einem Schema als auch auf einer Instanz gebildet werden. Dafür muss das obige formale Modell um weitere Abbildungsfunktionen erweitert werden. Die Grundidee ist dabei, für jedes Element eines Prozessgraphen zwei zusätzliche Markierungen einzuführen, mit denen festgelegt werden kann, ob ein Element sichtbar und ob es virtuell (d.h. erst durch Viewbildung bzw. Aggregation enstanden) ist oder nicht. Diese zwei Markierungen sind orthogonal zueinander, so dass es problemlos möglich ist, auch „Views-auf-Views“ zu bilden.

Definiton 4 (Attribute von Knoten)
Es gebe eine Menge A von Attributen. Ein Element a ∈ A sei durch folgendes Tupel definiert:

a = (name, dom^a)

Dabei ist *name* ein eindeutiger Bezeichner und *dom^a* der für a gültige Wertebereich
Jedem Knoten n ∈ N kann für jedes Attribut a ein Wert value ∈ *dom^a* zugewiesen werden:

$$ATTRIBUTE : N \times A \rightarrow v \cup UNDEFINED$$
$$mit\ v \in dom^A$$

Diese Abbildungsfunktion wird verwendet, um einem Knoten Attribute mit Werten zuweisen zu können. Dabei kann es sich um bereits vorhandene Attribute handeln (z.B. Ein-/ Ausgabeparameter einer View, einer Vorlage zugeordnete Rollen, oder bei der Viewbildung neu berechnete Attribute).

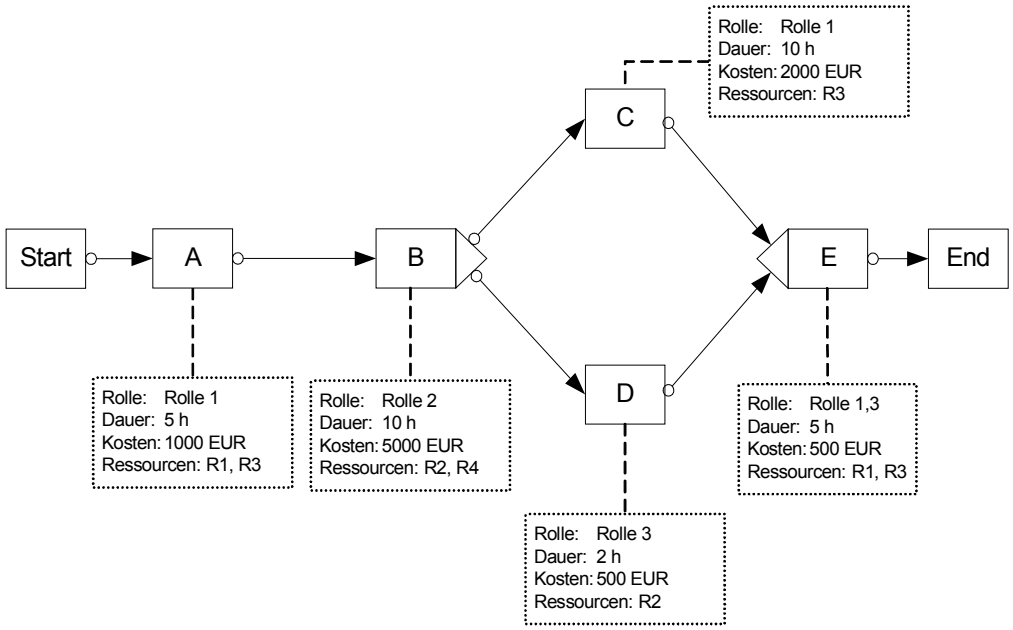

Abbildung 23: Prozessgraph mit attributierten Aktivitäten

Abbildung 23 zeigt ein Beispiel für die Zuweisung von Attributen zu Aktivitätenknoten. Im dargestellten Fall sind vier Attribute[5] für eine Aktivität definiert:

- a_1 = ("Rolle", {Rolle 1, Rolle 2, Rolle 3})

[5] In der obigen Definition wurde auf eine Zuweisung von Einheiten für die einzelnen Attribute, wie in der Abbildung sichtbar, verzichtet, da diese für die weitere Betrachtung nicht relevant sind.

- a_2 = ("Dauer", $\mathbb{R}^+$)
- a_3 = ("Kosten", $\mathbb{R}^+$)
- a_4 = ("Ressourcen", {R1, R2, R3})

Im Beispiel wurden jetzt jedem Knoten über die Abbildung ATTRIBUTE ein Attributwert für die einzelnen Attribute zugewiesen. Mit ATTRIBUTE(A,a_3) := 1000 wird z.B. dem Knoten A für das Attribut „Kosten“ der Wert 1000 zugeordnet.

Das Modell des vorherigen Abschnitt mit der obigen Erweiterung hat jetzt folgende Eigenschaften:

- Der Prozess wird als gerichteter Graph mit Knoten (diese stellen Aktivitäten dar) und Kanten (diese stellen den Kontroll- und Datenfluss dar) repräsentiert.
- Wir unterscheiden zwischen Graphen für ein Schema (ohne Zustand, Vorlage für Instanzen) und für Instanzen (mehrere Graphen, des gleichen Typs, können verschiedene Zustände haben)
- Knoten von Instanzen haben genau einen Zustand
- Knoten und Kanten können verschiedene Typen haben (Knoten: Aktivitäten, Split, Join, Daten Ein-/Ausgaben; Kanten: Kontrollfluss, Schleife, Datenfluss, Fehlerkanten)
- Knoten und Kanten haben Attribute
  - Konstante (für alle Instanzen gleiche) vs. variable Attribute
  - Beispiele für Knotenattribute: Kosten, Dauer, zugeordnete Ressourcen (Bearbeiter, Arbeitsmittel)
  - Beispiele für Kantenattribute: Kanten z.B. Priorität, Dauer, etc.

Zusätzlich werden alle in ADEPT definierten Änderungsprimitive und Algorithmen als gegeben vorausgesetzt.

Definiton 5 (Sichtbarkeit von Graphelementen)

$$VISIBLE : x \rightarrow \{TRUE , FALSE\}$$
$$mit\ x \in N \cup E \cup D \cup DFS$$

$$VISIBLE\_ATTR : N \times A \rightarrow \{TRUE , FALSE\}$$

Mit Hilfe der Abbildung *VISIBLE* können Graphelemente als ausgeblendet markiert werden, mit Hilfe von *VISIBLE_ATTR* kann für einen Knoten ein Attribut als ausgeblendet markiert werden.

Mit Hilfe dieser Abbildung ist es möglich, für jedes Element aus einer der Mengen N, E, D und DFS festzulegen, ob es in der View sichtbar sein soll oder nicht. Damit ist es möglich, die Viewbildung direkt auf dem Basisgraphen durchzuführen, da keine Kanten oder Knoten gelöscht werden. Die Viewbildung kann aber auch auf Kopien stattfinden, in denen dann alle nicht sichtbaren Elemente gelöscht werden. Welche Varianten verwendet wird, hängt vom konkreten Anwendungsfall ab (siehe auch Kapitel 7), im weiteren wird die Viewbildung direkt am Basisgraphen durchgeführt.

Ausgehend von obiger Abbildungsfunktion definieren wir das ein Kontrollflussschema CFS_{vi} und Datenflussschema DFS_{vi} eine View, die jeweils nur sichtbare Elemente enthält, und somit eine Sicht auf die Ursprungsschemata *CFS* und *DFS* repräsentiert.

Definiton 6 (Kontrollfluss-Schema und Datenschema dass eine View repräsentiert)
Gegeben sei ein ein Kontrollfluss-Schema CFS und zugehöriges Datenschema DFS. Mit Hilfe der Abbildung VISIBLE kann jetzt ein Kontrollfluss-Schema CFS_{vi} und ein Datenschema DFS_{vi} definiert werden, dass nur noch sichtbare Elemente enthält:

$$CFS_{vi} = (N_{vi}, E_{vi}, D_{vi}, NT_{vi}, NP_{vi}, V_{vi}^{out}, V_{vi}^{in}, DP_{vi}, EC_{vi}, Template_{vi})$$

mit

$$N_{vi} \subseteq N, \quad E_{vi} \subseteq E, D_{vi} \subseteq D$$

$$\forall n \in N \text{ gilt}: \quad VISIBLE(n) = TRUE \quad \Leftrightarrow \quad n \in N_{vi}$$
$$\forall e \in E \text{ gilt}: \quad VISIBLE(e) = TRUE \quad \Leftrightarrow \quad e \in E_{vi}$$
$$\forall d \in D \text{ gilt}: \quad VISIBLE(d) = TRUE \quad \Leftrightarrow \quad d \in D_{vi}$$

$$NT_{vi}: \quad N_{vi} \rightarrow NodeTypes \text{ mit } NT_{vi}(n) = NT(n)$$
$$NP_{vi}: \quad N_{vi} \rightarrow Priority \text{ mit } NP_{vi}(n) = NP(n)$$
$$V_{vi}^{out}: \quad N_{vi} \rightarrow OutBehaviour \text{ mit } V_{vi}^{out}(n) = V^{out}(n)$$
$$V_{vi}^{in}: \quad N_{vi} \rightarrow InBehaviour \text{ mit } V_{vi}^{in}(n) = V^{in}(n)$$
$$DP_{vi}: \quad N_{vi} \rightarrow D_{vi} \cup \{\texttt{UNDEFINED}\} \text{ mit } DP_{vi}(n) = DP(n)$$
$$EC_{vi}: \quad E_{vi} \rightarrow EdgeCode \cup \{\texttt{UNDEFINED}\} \text{ mit } EC_{vi}(e) = EC(e)$$
$$Template_{vi}: \quad N_{vi} \rightarrow Templates \text{ mit } Template_{vi}(n) = Template(n)$$

$$DFS_{vi} \subseteq DFS$$
$$\forall dfc \in DFS \text{ gilt}: \quad VISIBLE(dfc) = TRUE \quad \Leftrightarrow \quad dfc \in DFS_{vi}$$

$$ATTRIBUTE_{vi}: \quad NA \rightarrow dom$$
$$\text{mit } NA \subseteq N_{vi} \times A$$
$$\forall y \in N_{vi} \times A \text{ gilt}: VISIBLE\_ATTR(y) = TRUE \quad \Leftrightarrow \quad y \in NA$$

Hiermit können jetzt Primitive für das Verbergen von Graphelementen und Attributen definiert werden (E steht dabei für Erklärung, V für Vorbedingung und S für Semantik:

SetNodeAttributeValue(CFS, nodeLabel, a,value)
E: Setzt für den Knoten nodeLabel das Attribut a auf value.
V: nodeLabel ∈ N, a ∈ A, value ∈ dom^a
S: ATTRIBUE(nodeLabel, a):= value

HideElements(CFS, DFS, elementSet)
E: Verbirgt alle Element die in elementSet enhalten sind
V: elementSet ⊆ N ∪ E ∪ D ∪ DFS
S: ∀ e ∈ elementSet:
VISIBLE(e) := FALSE

HideNodeAttributes(CFS, nodeLabel, attributeSet)
E: Verbirgt für den Knoten nodeLabel alle in attributeSet enthaltenen Attributen
V: nodeLabel ∈ N
attributeSet ⊆ A
S: ∀ a ∈ attributeSet:
VISIBLE(n, a) := FALSE

Mit Hilfe der obigen Definitionen können jetzt Teile eines Graphen entfernt werden. Für die Viewbildung ist es jedoch auch notwendig, dass neue virtuelle Elemente (z.B. ein Knoten der mehrere andere Knoten zusammenfasst) hinzugefügt werden können.

Definiton 7 (Makierung als virtuelles Element)

$$VIRTUAL : x \rightarrow \{TRUE , FALSE\}$$
$$mit\ x \in N \cup E \cup D \cup DFS$$

$$VIRTUAL\_ATTR : N \times A \rightarrow \{TRUE , FALSE\}$$

Mit dieser Abbildungsfunktion ist es möglich, Elemente des Basisgraphen durch ein neues Element zu ersetzen bzw. mehrere Elemente zusammenzufassen. Das Beispiel aus Abbildung 24 verdeutlicht mögliche Kombinationen von VISIBLE- und VIRTUAL-Attributen. Dort gibt es einen Basisprozess, der aus den drei Schritten A, B und C besteht. Die Schritte B und C werden bei einer Viewbildung durch einen virtuellen Schritt V_1 ersetzt. Deshalb sind die Knoten sowie ein- und ausgehende Kanten von B und C auf VISIBLE=`FALSE` gesetzt, das Attribut VIRTUAL ist für diese Elemente ebenfalls `FALSE`, da es sich um Knoten des Basisprozesses handelt. Da der virtuelle Knoten V_1 für die View erzeugt worden ist, sind hier

sowohl VISIBLE als auch VIRTUAL auf `TRUE` gesetzt. Der Knoten A aus dem Basisprozess wird allerdings in die View unverändert übernommen, d.h. für A gilt VISIBLE=`TRUE` und VIRTUAL=`FALSE`.

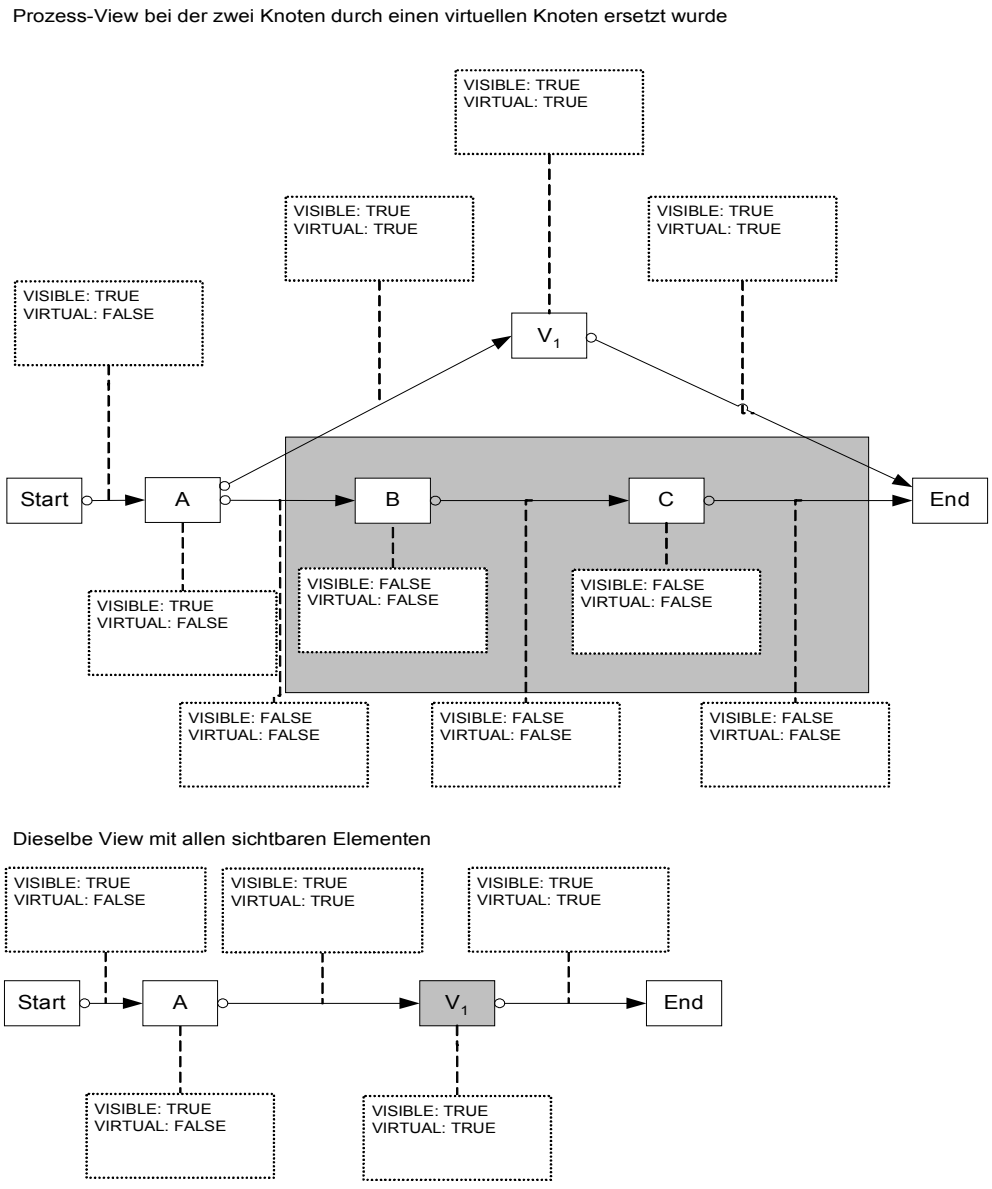

Abbildung 24: Beispiel für eine View mit VISIBLE- und VIRTUAL-Attributen

Ein virtuelles Element ist im Gegensatz zu einem Basis-Element kein Bestandteil eines real ablaufenden Prozesses, sondern steht als Platzhalter für ein oder mehrere Basiselemente. Um eine Element als VIRTUAL zu markieren, werden folgende Primitive definiert:

SetVirtual(CFS, DFS, elementSet)
E: Markiert alle Element aus elementSet als virtuelles Element V: elementSet $\subseteq$ N $\cup$ E $\cup$ D $\cup$ DFS S: $\forall$ e $\in$ elementSet: VIRTUAL(e) := TRUE
SetNodeAttributesVirtual(CFS,n,attributeSet)
E: Markiert alle Attribute aus attributeSet für den Knoten n als virtuell V: n $\in$ N attributeSet $\subseteq$ A S: $\forall$ a $\in$ attributeSet: VIRTUAL(n, a): = TRUE

Zum Einfügen virtueller Knoten und Kanten müssen lediglich die jeweiligen Einfügeoperationen des ADEPT-Basismodells (mit anschließender Markierung als virtueller Knoten) ausgeführt werden:

AddVirtualNodes(CFS, DFS, nodeLabels)
AddNodes(CFS, nodeLabels); SetVirtual(CFS, DFS, nodeLabels);
AddVirtualEdges(CFS, DFS, edgeSet)
AddEges(CFS, edgeSet); SetVirtual(CFS, DFS, edgeSet);
AddVirtualDataElements(CFS, DFS, dataLabels, dom, defaultVal)
AddDataLabels(CFS, DFS, dataLabels, dom, defaultVal); SetVirtual(CFS, DFS, dataLabels);
AddVirtualDataEdge(CFS, DFS, dataEdges)
AddDataEdges(CFS, DFS, dataEdges); SetVirtual(CFS, DFS, dataEdges);
AddVirtualAttribute(CFS, DFS, nodeLabel, attribute, value)
SetNodeAttributeValue(CFS, nodeLabel, attribute, value); SetNodesAttributesVirtual(CFS, nodeLabel, {attribute});

Für einen Graph mit virtuellen Knoten kann nun eine Teilmenge definiert werden, die nur Elemente des Basisgraphen enthält.

Definiton 8 (Kontrollfluss-Schema und Datenschema die das Basisschema einer View repräsentieren)
Gegeben sei ein ein Kontrollfluss-Schema CFS und zugehöriges Datenschema DFS. Sind in diesem Schema auch virtuelle Elemente enthalten, die während einer Viewbildung entstanden sind, so seien das Basis-Kontrollfluss-Schema CFS_{base} und Basis-Datenschema DFS_{base} wie folgt definiert:

$$CFS_{base} = (N_{base}, E_{base}, D_{base}, NT_{base}, NP_{base}, V^{out}_{base}, V^{in}_{base}, DP_{base}, EC_{base}, Template_{base})$$

mit

$$N_{base} \subseteq N, \quad E_{base} \subseteq E, D_{base} \subseteq D$$

$$\forall n \in N \text{ gilt}: \; VIRTUAL(n) = FALSE \quad \Leftrightarrow \quad n \in N_{base}$$
$$\forall e \in E \text{ gilt}: \; VIRTUAL(e) = FALSE \quad \Leftrightarrow \quad e \in E_{base}$$
$$\forall d \in D \text{ gilt}: \; VIRTUAL(d) = FALSE \quad \Leftrightarrow \quad d \in D_{base}$$

$$NT_{base}: \; N_{base} \rightarrow NodeTypes \text{ mit } NT_{base}(n) = NT(n)$$
$$NP_{base}: \; N_{base} \rightarrow Priority \text{ mit } NP_{base}(n) = NP(n)$$
$$V^{out}_{base}: \; N_{base} \rightarrow OutBehaviour \text{ mit } V^{out}_{base}(n) = V^{out}(n)$$
$$V^{in}_{base}: \; N_{base} \rightarrow InBehaviour \text{ mit } V^{in}_{base}(n) = V^{in}(n)$$
$$DP_{base}: \; N_{base} \rightarrow D_{base} \cup \{\texttt{UNDEFINED}\} \text{ mit } DP_{base}(n) = DP(n)$$
$$EC_{base}: \; E_{base} \rightarrow EdgeCode \cup \{\texttt{UNDEFINED}\} \text{ mit } EC_{base}(e) = EC(e)$$
$$Template_{base}: \; N_{base} \rightarrow Templates \text{ mit } Template_{base}(n) = Template(n)$$

$$DFS_{base} \subseteq DFS$$
$$\forall dfc \in DFS \text{ gilt}: \; VIRTUAL(dfc) = FALSE \quad \Leftrightarrow \quad dfc \in DFS_{base}$$

$$ATTRIBUTE_{base}: \; NA \rightarrow dom$$
$$\text{mit } NA \subseteq N_{base} \times A$$
$$\forall y \in N_{base} \times A \text{ gilt}: VIRUTAL\_ATTR(y) = FALSE \quad \Leftrightarrow \quad y \in NA$$

5.3 Algorithmen für die View-Bildung

Aufbauend auf den im vorigen Abschnitt definierten Änderungsprimitiven definieren wir nun Algorithmen für die Bildung von Views. Dazu geben wir zuerst Operationen für das Verbergen einzelner Elemente an und auf diesen aufbauend, komplexe Operationen für die Zusammenfassung von Knoten. Abschließend stellen wir die eigentlichen Algorithmen für die Bildung von Views vor.

5.3.1 Verbergen einzelner Elemente

Die folgenden Grundoperationen dienen dem Verbergen bzw. Ersetzen einzelner Graphelemente. Sie werden von den im Anschluss vorgestellten Algorithmen zur View-Bildung verwendet. Einige dieser Operationen, z.B. zum Verbergen von Attributen, können auch direkt zur View-Bildung herangezogen werden. Für andere Operationen (z.B. HideActivity) gilt dies nicht, da das Ergebnis der Operation kein gültiger Graph ist, und die Anwendung deshalb nur im Zusammenspiel mit anderen Operationen Sinn macht.

HideAttribute(CFS, X, A)
input:
CFS: WF-Graph, für den eine View gebildet werden soll
X: Aktivität, für die ein Attribut weggeblendet werden soll
A: Attribut das weggeblendet werden soll
begin:
HideNodeAttributes(CFS,X,{A});
end

Diese Operation verbirgt für eine Aktivität X das Attribut A. In Abbildung 25 etwa wird für die Aktivität X das Attribut „Kosten“ verborgen.

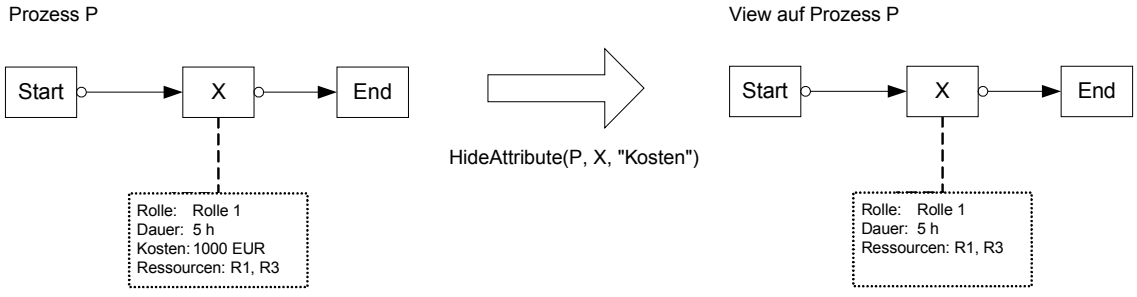

Abbildung 25: Beispiel für HideAttribute

Mit der folgenden Funktion kann in einem Prozessgraph ein Datenelement mit allen ein-/ ausgehenden Datenflusskanten ausgeblendet werden.

HideDataNode(CFS, DFS, dataNode)
input:
CFS, DFS: WF-Graph, für den eine View gebildet werden soll
dataNode: Datenknoten, der ausgeblendet werden soll
begin:
HideElements(CFS, DFS, {dataNode});
E_{in} := Menge aller in dataNode einmündenden Datenkanten
E_{out} := Menge aller in dataNode ausgehenden Datenkanten
HideElements(CFS, DFS, $E_{in} \cup E_{out}$);
end

Im Beispiel in Abbildung 26 werden die drei Datenelemente d_1, d_2 und d_3 ausgeblendet.

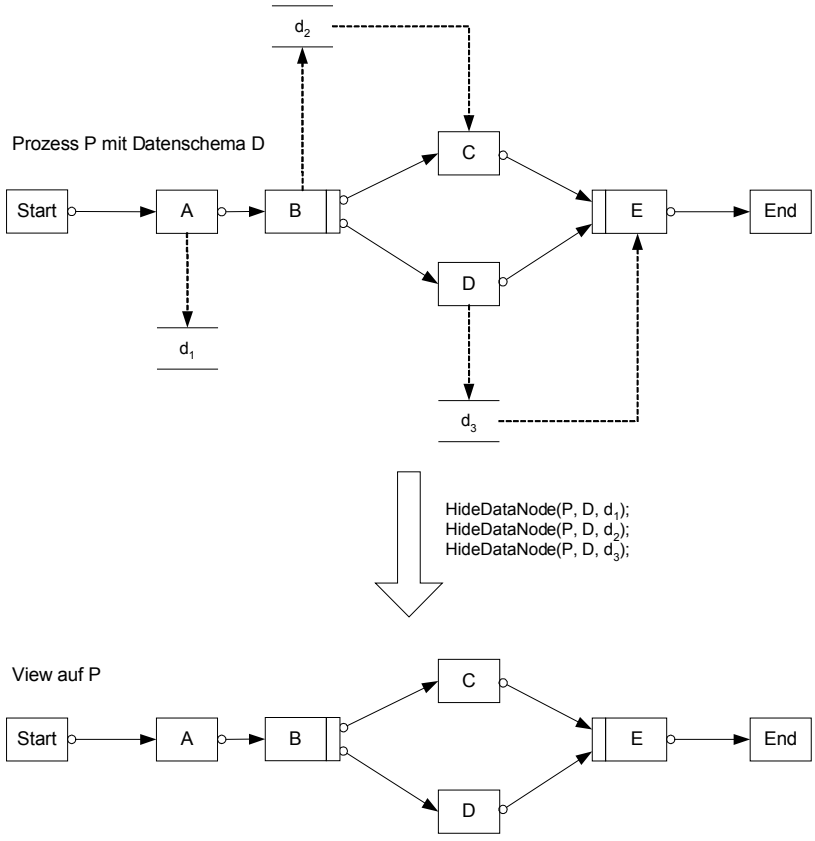

Abbildung 26: Beispiel für HideDataNode

Mit folgendem Algorithmus kann ein Aktivitätenknoten ausgeblendet werden.

```
HideActivity(CFS, DFS, X)
  // verbergen von Aktivitätenknoten und aller in den Knoten ein- und
  // ausgehenden Kanten
  input:
      CFS, DFS: WF-Graph, bei dem eine Aktivität ausgeblendet werden sollen
      X: Aktivität, die ausgblendet werden sollen
  begin:
      HideElements(CFS, DFS, {X});

      // Kontrollflusskanten verbergen
      edges : = ∅;
      forall e ∈ E with Id_source(e)= X or Id_dest(e)=X do
         edges := edges ∪ {e};
      done

      // Datenflusskanten verbergen
      forall d ∈ DFS with Id_source(d)= X  do
         edges := edges ∪ {d};
         D:= Id_dest(e); // zugehöriges Datenelement
         data_edges:= ∅;
         forall  y ∈ DFS with (Id_source(y)=D or Id_dest(y)=D)  and VISIBLE(y)=TRUE do
              data_edges:= date_edges∪ {y};
         done
         if |data_edges| = 1 then
              HideElements(CFS, DFS, {D});
      done
      forall d ∈ DFS with Id_dest(d)= X  do
         edges := edges ∪ {d};
         D:= Id_source(e); // zugehöriges Datenelement
         data_edges:= ∅;
         forall  y ∈ DFS with (Id_source(y)=D or Id_dest(y)=D)  and VISIBLE(y)=TRUE do
              data_edges:= date_edges∪ {y};
         done
         if |data_edges| = 1 then
              HideElements(CFS, DFS, {D});
      done
      HideElements(CFS, DFS, edges);
      // Attribute verbergen
      forall a ∈ A with ATTRIBUTE(X,a) ≠ UNDEFINED do
         HideAttribute(CFS, X, a);
      done
  end
```

Die beiden obigen Operationen entsprechen dem logischen Löschen eines Knotens aus einer View. Für den Knoten und alle ein-/ausgehenden Kanten wird das Attribut VISIBLE auf `FALSE` gesetzt. Assoziierte Datenflusskanten werden ebenfalls verborgen. Falls die verborgene Kante die letzte sichtbare eines Datenelementes ist, wird diese ebenfalls verborgen. Somit ist gewährleistet, das keine Datenelemente ohne sichtbaren Datenfluss in der View vorhanden sind.

Aufbauend auf *HideActivity* können mit folgendem Algorithmus auch die assoziierten Kontrollflusskanten „kurzgeschlossen" werden.

HideActivityBypassEdge(CFS, DFS, X)

input:
CFS, DFS: WF-Graph, bei dem eine Aktivität ausgeblendet werden sollen
X: Aktivität, die ausgeblendet werden sollen
begin:
// nur für Aktivitätenknoten die genau eine sichtbare ein- und ausgehenden
// Kante haben
//
// verbergen des Knoten und Kurzschließen der sichtbaren ein- mit der
// sichtbaren ausgehenden Kontrollflusskante
HideActivity(CFS, DFS, X);
e_{in} := in den Knoten X eingehende Kontrollflusskante mit VISIBLE(e_{in})=`TRUE`;
e_{out} := von dem Knoten X ausgehende Kontrollflusskante mit VISBLE(e_{out})=`TRUE`;
e_{new}:= ($Id_{source}(e_{in})$, $Id_{dest}(e_{out})$, `CONTROL_E`);
AddVirtualEdges(CFS, DFS, {e_{new}});
end

Diese Operation setzt direkt auf der vorherigen auf. Sie erzeugt eine virtuelle Kante, die von dem Vorgänger- zu dem Nachfolgerknoten der ausgeblendeten Aktivität geht. Die Aktivität wird also nicht nur logisch gelöscht, sondern mit Hilfe der neu erzeugten Kante „überbrückt". Diese Operation bildet somit die Grundlage für den in Abschnitt 5.3.4 vorgestellten Graphreduktionsalgorithmus.

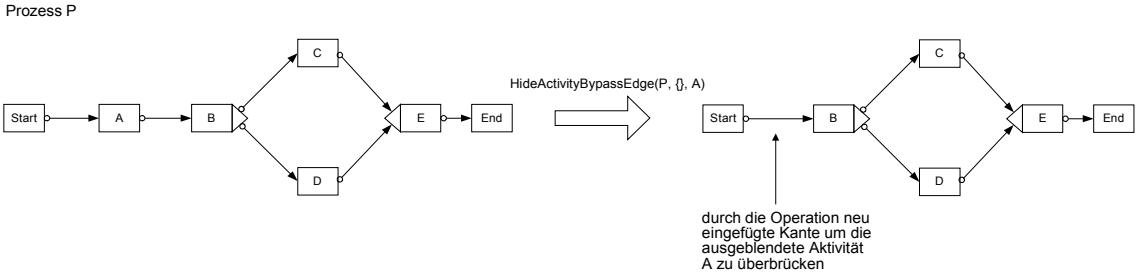

Abbildung 27: Beispiel für HideActivityBypassEdge

Im Beispiel aus Abbildung 27 wird die Aktivität A ausgeblendet. Begleitend werden die assoziierten Kanten ebenfalls verborgen und durch eine virtuelle Kante zwischen dem Startknoten und Knoten E ersetzt.

Mit folgender Operation kann ein Aktivitätenknoten versteckt und durch einen virtuellen Knoten ersetzt werden.

ReplaceActivity(CFS, DFS, X)
// Ersetzten einer Aktivität und aller ein- und ausgehenden Kanten durch eine
// virtuelle Aktivität mit virtuellen Kanten
input:
CFS, DFS: WF-Graph, bei dem eine Aktivität ausgeblendet werden sollen
X: Aktivität, die durch einen virtuelle Aktivität ersetzt werden soll
output:
Label der neu erzeugten virtuellen Aktivität
begin:
HideActivity(CFS, DFS, X);
n_{new}:= GenerateUniqueNodeLabel(CFS);
AddVirtualNodes(CFS, DFS, {n_{new}});
$V_{out}^{n_{new}} := V_{out}^{X}$; $V_{in}^{n_{new}} := V_{in}^{X}$; $NT(n_{new}) := NT(X)$; $NP(n_{new}) := NP(X)$;
$DP(n_{new}) := DP(X)$; $Template(n_{new}) := Template(X)$;
forall e ∈ E **with** Id_{source}(e)= X **and** VISIBLE(e) = `TRUE` **do**
e_{new}:= (n_{new}, Id_{dest}(e),ET(e));
AddVirtualEdges(CFS, DFS,{e_{new}});
done
forall e ∈ E **with** Id_{dest}(e)= X **and** VISIBLE(e) = `TRUE` **do**
e_{new}:= (Id_{source}(e),n_{new},ET(e));
AddVirtualEdges(CFS, DFS, {e_{new}});
done
// Datenflusskanten können entsprechend den Kontrollflusskanten ersetzt werden
return n_{new};
end

Es handelt sich hierbei um eine Kopieroperation, die einen Knoten und dessen Kanten verbirgt und dafür einen neuen virtuellen Knoten und zugehörige Kanten erzeugt. Diese Operation wird für den Graphreduktionsalgorithmus benötigt, um Join- und Split-Knoten vorziehen zu können.

5.3.2 Knotenaggregation

Mit Hilfe der folgenden beiden Operationen kann eine Menge von Knoten (und deren zugehörige Attribute) durch einen virtuellen Knoten ersetzt werden. Diese Operationen können direkt zur Viewbildung verwendet werden (z.B. um einen Verzweigungsblock durch einen einzigen Schritt zu ersetzen). Des weiteren bilden sie auch die Basis für den nachfolgend vorgestellten Graphaggregationsalgorithmus.

AggregateActivities(CFS, DFS, n_{start}, n_{end})

// Zusammenfassung aller zwischen n_{start} und n_{end} liegenden Knoten durch einen
// virtuellen Knoten. Dabei muss durch n_{start} und n_{end} ein gültiger Kontrollblock[6]
// gegeben sein oder es darf nur ein einziger Knoten (n_{start}) gegeben sein

input:
CFS, DFS: WF-Graph, bei dem eine Aktivität aggregiert werden soll
n_{start}: Startknoten des Kontrollblocks
n_{end}: Endknoten des Kontrollblocks oder `UNDEFINED`

output:
n_{new}: neu erzeugter virtueller Knoten

begin:
if ($n_{end} \neq$ `UNDEFINED` **and** MinBlock(n_{start}, n_{end}) $\neq$ (n_{start}, n_{end})) **then abort**;
n_{new}:= GenerateUniqueNodeLabel(CFS);
AddVirtualNodes(CFS, DFS, {n_{new}});
a_{start}:=(startNode, N); a_{end}:=(endNode, N);
AddVirtualAttribute(CFS, n_{new}, a_{start}, n_{start});
AddVirtualAttribute(CFS, n_{new}, a_{end}, n_{end});
M:= succ_c*(n_{start}) $\cap$ pred_c*(n_{end}) $\cup$ {n_{start}} $\cup$ {n_{end}} \ `UNDEFINED;`
forall n $\in$ M **do**
// Alle ausgehenden Kontrollflusskanten des Blocks kopieren und auf den neu
// erzeugten Knoten setzen
forall e $\in$ E **with** Id_{source}(e)= n **and** Id_{dest}(e) $\notin$ M and VISIBLE(e) = TRUE **do**
e_{new}:= (n_{new}, Id_{dest}(e), ET(e));
AddVirtualEdges(CFS, DFS, {e_{new}});
done
// Alle eingehenden Kontrollflusskanten des Blocks kopieren und auf den neu
// erzeugten Knoten setzen
forall e $\in$ E **with** Id_{dest}(e)= n **and** Id_{source}(e) $\notin$ M and VISIBLE(e) = TRUE **do**
e_{new}:= (Id_{source}(e), n_{new}, ET(e));
AddVirtualEdges(CFS, DFS, {e_{new}});
done
HideActivity(CFS, DFS, n); *// Alle Knoten und Kanten des Blocks verbergen*
done forall n $\in$ M
// Anschließend Ein- und Ausgangssemantik von n_{start} und n_{end} entsprechend
// auf n_{new} abbilden
if ($n_{end} \neq$ `UNDEFINED`) **then** *// es handelt sich um einen ganzen Block*
$V_{out}^{n_{new}} := V_{out}^{n_{end}}$; $V_{in}^{n_{new}} := V_{in}^{n_{start}}$
else *// es handelt sich nur um einen Knoten*
$V_{out}^{n_{new}} := V_{out}^{n_{start}}$; $V_{in}^{n_{new}} := V_{in}^{n_{start}}$
endif
return n_{new};

end

[6] Dabei handelt es sich um einen zusammenhängenden Teilgraphen mit einem eindeutigen Start- und Endknoten. Eventuelle Verzweigungs- oder Loopblöcke sind in dem Teilgraphen vollständig enthalten. Siehe auch [4].

Der obige Algorithmus fasst die durch n_{start} und n_{end} definierte Knotenmenge zusammen, indem sämtliche Elemente (Kanten und Knoten) innerhalb dieser Menge verborgen und durch einen neuen Knoten ersetzt werden. Alle in den Block ein- und ausgehenden Kontrollflusskanten werden für den neu erzeugten Knoten kopiert und anschließend mit diesem Knoten verbunden. Der Algorithmus klammert den Datenfluss aus (dieser wird einfach verborgen). Es ist jedoch leicht möglich, die Datenflusskanten entsprechend den Kontrollflusskanten zu kopieren.

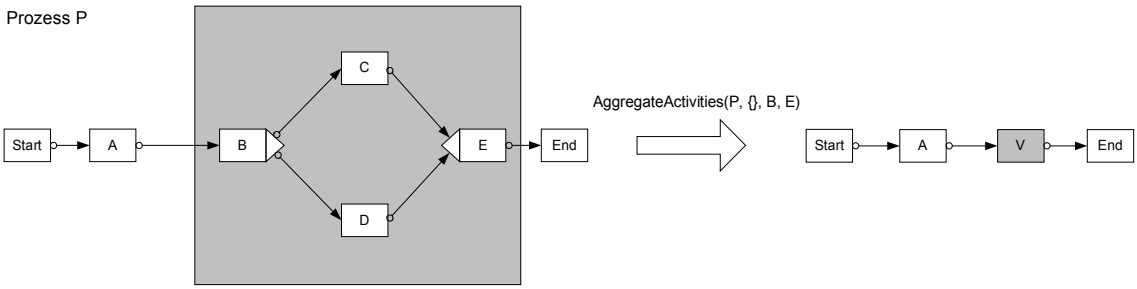

Abbildung 28: Beispiel für AggregateActivities

In Abbildung 28 wird die graumarkierte Teilemenge (Knoten B, C, D und E) aggregiert und durch einen virtuellen Knoten V ersetzt. Auch assoziierten Kanten der Start- und Endknoten des virtuellen Schrittes (Knoten B und E) wurden ausgeblendet und durch neue Kanten für V ersetzt.

Mit folgender Operation können neben Aktivitäten auch die zugehörigen Attribute aggregiert werden.

AggregateActivitiesAndAttributes(CFS, DFS, n_{start}, n_{end}, AF, A, UF)

// Zusammenfassung aller zwischen n_{start} und n_{end} liegenden Knoten durch einen
// virtuellen Knoten und Aggregation der übergebenen Attribute
// Dabei muss durch n_{start} und n_{end} ein gültiger Kontrollblock
// gegeben sein.

input:
CFS, DFS: WF-Graph, bei dem eine Aktivität aggregiert werden soll
n_{start}: Startknoten des Kontrollblocks
n_{end}: Endknoten des Kontrollblocks
AF: Menge der gültigen Aggregationsfunktionen (z.B. SUM, MAX, MIN, AVG)
A: Menge von Attributen, für die aggregierte Attribute erzeugt werden sollen
UF: Abbildung, die jedem Attribut eine Aggregationsfunktion zuordnet
UF: A → AF

output:
n_{new}: neu erzeugter virtueller Knoten

begin:
if (MinBlock(n_{start}, n_{end}) ≠ (n_{start}, n_{end})) **then abort**;
else
n_{new}:= AggregateActivities(CFS,DFS, n_{start}, n_{end});
forall a ∈ A **do**
f:= UF(a); *//Aggreagtionsfunktion für das aktuelle Attribut, f ∈ AF*
B:= ∅; *//B ist eine Bag, die Attributwerte aufnimmt*
forall n ∈ (succ_c*(n_{start}) ∩ pred_c*(n_{end}) ∪ {n_{start}} ∪ {n_{end}}) **do**
B:= add(B, ATTRIBUTE(n,a); *//Wert des Attributes in B aufnehmen*
done
AddVirtualAttribute(CFS, n_{new}, a, f(B));
endif
return n_{new};
end

In Abbildung 29 wird, wie im vorherigen Beispiel, eine Menge von Knoten (B, C, D und E) aggregiert. Zusätzlich werden ebenfalls die in diesem Beispiel für jeden Knoten vorhandenen Attribute „Kosten“ und „Dauer“ noch mit Hilfe einer Aggregationsfunktion (Durchschnitts- bzw. Summenbildung) für den virtuellen Knoten Attribute zur Verfügung gestellt. Bei der XOR-Verzweigung wird für die Summenbildung der Zweig (hier Knoten D) mit dem maximalen Wert verwendet. Bei der Durchschnittsbildung wird von einer Wahrscheinlichkeit für die Ausführung eines Zweiges von 50% ausgegangen.

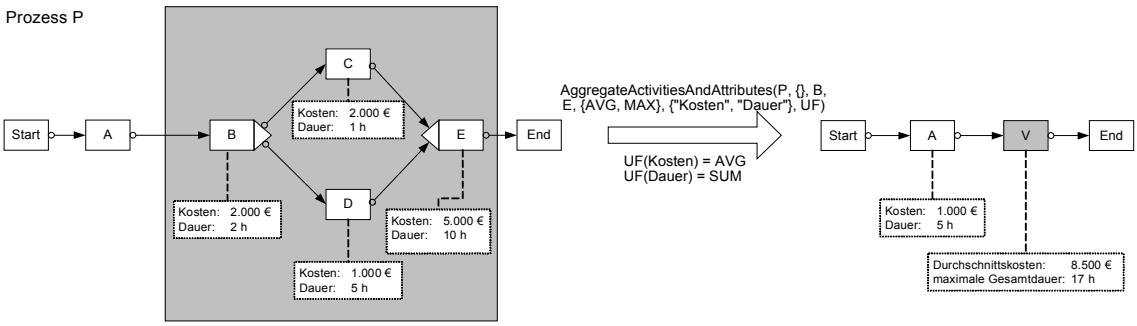

Abbildung 29: Beispiel für die Anwendung des Algorithmus AggregateActivitiesAndAttributes

Dieses Beispiel verdeutlicht auch, dass die Aggregation von Attributen nur für bereits (zumindest zum Teil) beendete Instanzgraphen „korrekt“ ausgeführt werden kann, da für XOR-Verzweigungen im Voraus nicht klar ist, welcher Teilzweig zur Ausführung kommt. XOR-Verzweigungen werden von obigem Algorithmus nicht gesondert behandelt. Um dafür sinnvolle Berechnungen durchführen zu können, müssten am XOR-Split für die einzelnen Teilzweige Ausführungswahrscheinlichkeiten bekannt sein. Das Problem in der Praxis ist, diese Ausführungswahrscheinlichkeiten zu bestimmen. Dies könnte z.B. dadurch erfolgen, dass ein Prozessmodellierer die Wahrscheinlichkeiten vorgibt oder indem Analyse bereits durchlaufener Prozess-Instanzen entsprechende Wahrscheinlichkeiten für zukünftige Prozess-Instanzen abschätzt.

5.3.3 Einfache Algorithmen zur View-Bildung

Basierend auf den obigen Grundoperationen können einfache Algorithmen für die Bildung von Views definiert werden. Es folgen beispielhaft zwei davon, mit denen der Datenfluss sowie sämtliche Schleifen in einer View entfernt werden können. Es können weitere Algorithmen angegeben werden, die nach dem gleichen Schema funktionieren (Verbergen sämtlicher Attribute, Verzweigungen durch einen Knoten ersetzen, usw.). Hierauf wird an dieser Stelle verzichtet.

```
HideLoops(CFS)
input:
      CFS: WF-Graph, bei dem die Schleifen ausgeblendet werden sollen
  begin:
      // verbergen sämtlicher Schleifen Knoten und Kanten
      forall n ∈ N with NT(n) = STARTLOOP or NT(n) = ENDLOOP do
            HideActivityBypassEdge(CFS, DFS, n);
      done
  end
```

Ein Beispiel für die Anwendung dieses Algorithmus gibt die Abbildung 13 auf Seite 24. Dort wird die vorhandenen Schleifenknoten verborgen und die Kanten entsprechend vorgezogen. Mit der folgenden Operation kann der gesamte Datenfluss für ein Schema verborgen werden.

HideDataFlow(CFS, DFS)
input:
CFS, DFS: WF-Graph, für den eine View gebildet werden soll
begin:
// verbergen des kompletten Datenflusses
HideElements(CFS, DFS, D);
HideElements(CFS, DFS, DFS);
done
end

Im Beispiel aus Abbildung 30 wird der gesamte Datenfluss für einen Prozess P ausgeblendet.

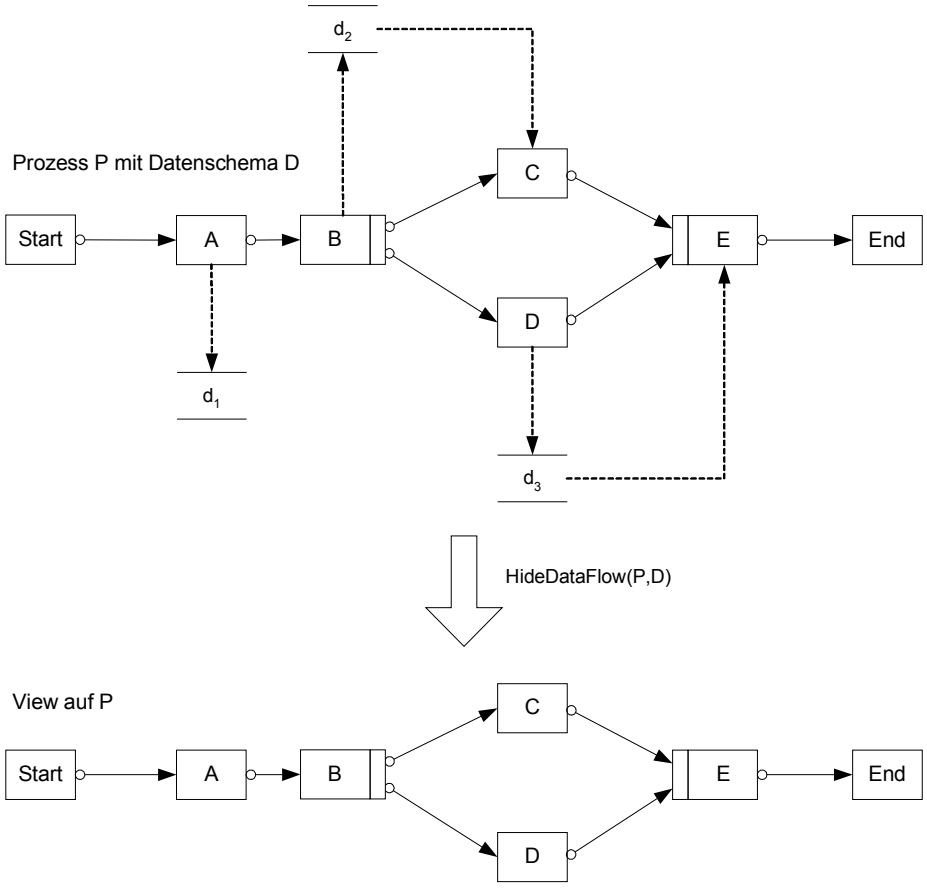

Abbildung 30: Beispiel für HideDataFlow

5.3.4 Graphreduktionalgorithmus

In Kapitel 4 wurden zwei Arten der Viewbildung unterschieden: Graphreduktion (Löschen von Aktivitäten) und Graphaggregation (Zusammenfassen von Aktivitäten). Aufbauend auf den im vorherigen Abschnitt definierten Operationen stellen wir jetzt einen Algorithmus für die Graphreduktion vor. Bei diesem Algorithmus ist der Ansatz folgender: Es ist nur eine bestimmte Menge von Knoten für eine View von Interesse. Dem Algorithmus wird dann diese Menge von Knoten übergeben. Es wird dann der Graph – soweit möglich und unter Beibehaltung von Verzweigungsstrukturen[7] – reduziert (durch Löschen von Aktivitäten). Ein ähnlicher Algorithmus wird auch in [8] zur Analysierung der Korrektheit von Prozessgraphen verwendet.

Die Arbeitsweise dieses Algorithmus soll hier an einem Beispiel illustriert werden. Ein Prozessgraph bestehe aus der eine Menge von Knoten {Start, A, B, C, D, E, F, G, H, I, J, K, L, M, End} (siehe Abbildung 31). Für des Graph soll nun eine graphreduzierte View gebildet werden soll.

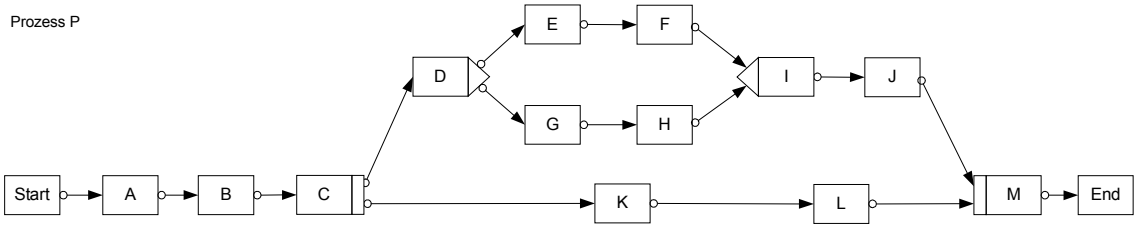

Abbildung 31: Basisprozess für Graphreduktion

Schritt 1: Mit Hilfe einer Auswahlfunktion wird ermittelt, welche Knoten erhalten bleiben sollen (z.B. alle Aktivitäten die einem bestimmten Bearbeiter zugeordnet sind, alle Aktivitäten die bestimmte Kosten verursachen oder eine bestimmte Ressource benötigen usw.). In dem Beispiel werden die Knoten A, E, F, J und L als relevante Knoten für die View ausgewählt. Daher wird der Algorithmus folgendermaßen aufgerufen: *ReduceGraph(P, {}, {A, E, F, J, L})*

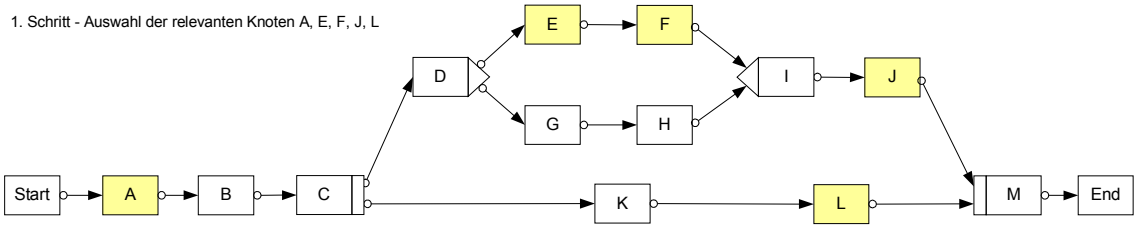

Abbildung 32: Auswahl relevanter Knoten

[7] Durch das Beibehalten von Verzweigungsstrukturen kann in der View weiterhin unterschieden werden, ob Aktivitäten sequentiell oder parallel ausgeführt werden.

Schritt 2: Entfernung aller Knoten, deren Ein-/Ausgangsgrad jeweils 1 ist und die nicht relevante Knoten sind. Die eingehende Kante dieser Knoten wird mit der jeweils ausgehenden „verschmolzen“. In dem Beispiel werden die Knoten B, G, H und K ausgeblendet. Übrig bleibt der in Abbildung 33 sichtbare Graph.

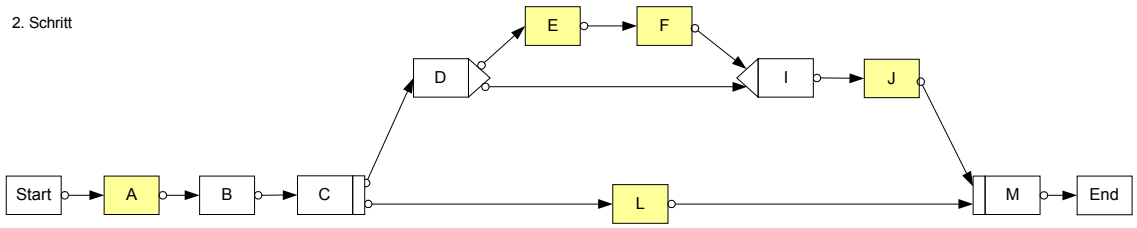

Abbildung 33: Entfernung nicht relevanter Knoten

Schritt 3: Entfernung der Kanten, die direkt von einem XOR/AND-Split zu einem XOR/AND-Join zeigen. Falls dadurch ein XOR/AND-Block keine Kanten mehr hat, diesen Block entfernen, eingehende Kante des Split-Knoten mit ausgehender Kante des Join-Knotens kurzschließen. Im Beispiel wird in diesem Schritt die Kante zwischen D und I entfernt (siehe Abbildung 33).

Schritt 4: Entfernung von AND-Blöcken und XOR-Blöcken, die jetzt nur noch einen Pfad haben. Markierung der verbliebenen Knoten eines XOR-Blockes als „optional“, um deutlich zu machen, dass es sich um Knoten in dem Teilpfad einer XOR-Verzweigung handelte. Im Beispiel ist hiervon die XOR-Verzweigung zwischen D und I betroffen. Die Kante zwischen D und I wurde im vorherigen Schritt entfernt, damit hat der Block nur noch einen sichtbaren Pfad, und die Verzweigungsknoten D und I können entfernt und die Kanten zu den Vorgänger- (D) bzw. Nachfolger-Knoten (J) angepasst werden. Da es sich um einen XOR-Block handelt werden die Knoten des verbliebenen Pfades (Knoten E und F) noch mit einer besonderes Markierung versehen, um deutlich zu machen, dass sie nicht immer zu Ausführung gelangen müssen.

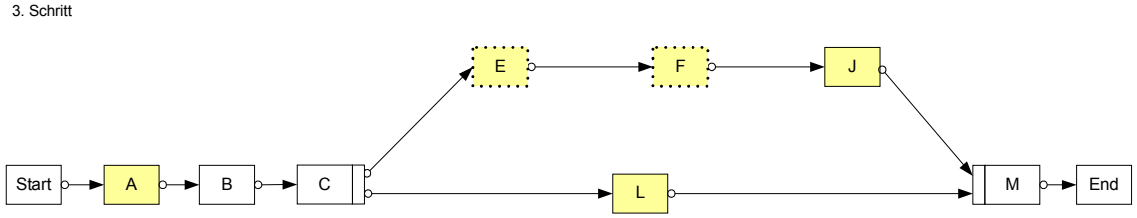

Abbildung 34: Entfernung von nicht benötigten Split- und Join-Knoten

Schritt 5: Die View enthält jetzt nur noch relevante Knoten sowie Verzweigungsknoten. Sie kann noch weiter vereinfacht werden, indem die Split-Knoten zu dem direkten Vorgänger vorgezogen werden. Dies ist dann möglich, wenn der direkte Vorgänger selbst kein Split-Knoten ist, da sonst die Information über Verzweigungsstrukturen in der View verloren gehen würden. Auch wird ein Split-Knoten nicht mit einem vorausgehenden Join-Knoten vereinigt, da sonst Aktualisierungen von Views (siehe Kapitel 8) nicht mehr einfach durchgeführt werden können. Deshalb werden AND-Splits entfernt und deren ausgehende Kanten vorgezogen, wenn sie nur einen Vorgängerknoten mit nur einer ein- und ausgehenden Kante haben. Der Vorgänger wird dabei durch eine Kopie ersetzt, da für den Originalknoten nicht die Ausgangssemantik geändert werden darf. Für XOR/AND-Joins, die nur einen Nachfolger mit nur einer ein- und ausgehenden Kante, gilt dasselbe wie für AND-Splits, und sie können auf die gleiche Weise ersetzt werden. Im Beispiel werden deshalb der AND-Split Knoten C entfernt und die Kanten zum Vorgängerknoten A vorgezogen (dabei wird jedoch A durch eine Kopie ersetzt, da sich die Ein- und Ausgangssemantik von Basisaktivitäten durch die Viewbildung nicht ändern soll). Entsprechendes gilt für den zugehörigen Join-Knoten M.

Ist in der View eine Unterscheidung zwischen Knoten, die in einem XOR-Zweig liegen[8] und anderen Knoten unwichtig können auch XOR-Splitknoten vorgezogen werden. Dabei sollte der Knoten an dem die Verzweigung dann stattfindet nicht auf die XOR-Ausgangssematik sondern auf die AND-Ausgangssemantik gesetzt werden. Durch das Vorziehen des XOR-Splitknoten müssen nicht alle Knoten innerhalb dieser neuen Verzweigung auch im Basisprozess innerhalb einer XOR-Verzweigung liegen. Es kann durch das Vorziehen von XOR-Split also eine „Vermischung“ von Knoten aus XOR- und AND-Verzweigungsblöcken in der View resultieren.

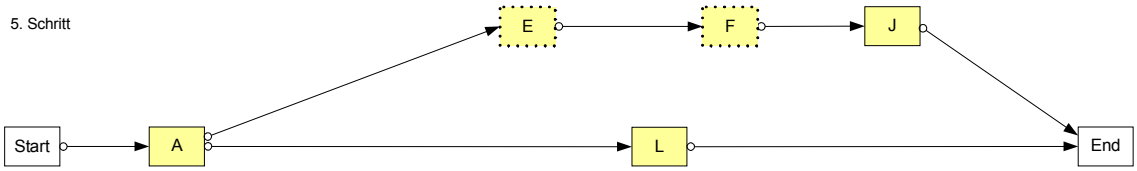

Abbildung 35: Vorziehen von Join- und Splitknoten

[8] Diese Knoten müssen deshalb nicht unbedingt zur Ausführung gelangen

Im Folgenden jetzt eine formale Beschreibung des am Beispiel erläuterten Algorithmus.

ReduceGraph(CFS, DFS, nodeLabels)

input:

CFS, DFS: WF-Graph, für den eine View gebildet werden soll
nodeLabels: Knoten, die erhalten bleiben sollen

begin:

EC:= Menge aller Kanten in CFS mit EdgeType `CONTROL_E`;
// Verbergen aller Knoten mit Ein- und Ausgangsgrad 1
forall n ∈ N \ nodeLabels **with** VISIBLE(n) = TRUE **and**
V_{in}^{n} = `ONE_Of_ONE` **and** V_{out}^{n} = `ONE_Of_ONE` **do**
HideActivityBypassEdge(CFS, DFS, n);
done
// Entfernung der durch den letzten Schritt entstandenen virtuellen Kanten,
// die direkt von eine Split- zu einem Join-Knoten gehen
EC:= Menge aller Kanten in CFS mit EdgeType `CONTROL_E`;
forall e ∈ EC **with** VIRTUAL(e) = `TRUE` **and**
$V_{out}^{Id_{source}(e)}$ = `ONE_Of_ALL` ∨ `ALL_Of_ALL` **and**
$V_{in}^{Id_{det}(e)}$ = `ONE_Of_ALL` ∨ `ALL_Of_ALL` **do**
DeleteEdges(CFS, {e});
// Entfernung aller Split- und Join-Knoten, die durch den vorherigen
// Schritt keine sichtbaren Kanten mehr haben,
// und anschließendes Kurzschließen der Kante
if (**forall** x∈EC **with** (Id_{source}(x) = Id_{source}(e)) VISIBLE(x) =`FALSE`) **then**
HideElements(CFS, DFS, {Id_{source}(e), Id_{dest}(e)});
e_{split} := einmündende Kontrollkante von Id_{source}(e);
e_{join} := ausgehende Kontrollkante von d_{dest}(e);
HideElements(CFS, DFS, {e_{split}, e_{join}});
e_{new}:= (Id_{source}(e_{split}), Id_{dest}(e_{join}), `CONTROL_E`);
AddVirtualEdges(CFS, DFS, {e_{new}});
endif
done

```
forall n ∈ SplitNodes^CFS \ (nodeLabels ∩ SplitNodes^CFS)
 with VISIBLE(n)=TRUE do
   vE:= Menge aller von n ausgehenden Kanten e  mit VISBILE(e) = TRUE;
   //Entfernung alle Verzweigungen, die jetzt nur noch einen sichtbaren
   // Pfad haben
   if (|vE| = 1) then
       // Ausblenden des Split-Knotens und Kurzschließen der Kanten
       HideActivityBypassEdge(CFS, DFS, n);
       // Ausblenden des zugehörigen Join-Knotens und dessen Kanten
       HideActivityByPassEdge(CFS, DFS, join^CFS(n));
       if ( V^n_out = ONE_Of_ALL ) then
          // es handelt sich um eine Or-Verzweigung,
          // deshalb Markierung der innerhalb gelegenen Knoten
          forall j ∈ (succ_c*(n) ∩ pred_c*(join^CFS(n)))
             with VISIBLE(j) = TRUE do
                attr_or:= (or_attribut,{TRUE, FALSE});
                //Attribut mit einem boolschen Wertebereich
                SetNodeAttributeValue(CFS, j, attr_or, TRUE);
                SetNodeAttributesVirtual(CFS, j, {attr_or});
          done
       endif
   endif (|vE| = 1)
   predN := direkter Vorgänger von n;
   if (V^predN_out = ONE_Of_ONE and V^predN_in = ONE_Of_ONE) then
       // Vorziehen von Split-Knoten mit einem Vorgänger mit nur einer
       // ein- und ausgehenden Kante
       HideActivity(CFS, DFS, n);
       predN := ReplaceActivity(CFS, DFS, predN);
       V^predN_out := V^N_out;
       forall edge ∈ EC with Id_source(edge) =  n do
          HideEdges(CFS, DFS, {e});
          e_new:= (predN, Id_dest(edge), CONTROL_E);
          AddVirtualEdges(CFS, DFS, {e_new});
       done
   endif
```

```
        succJ := direkter Nachfolger von join^CFS(n);
        if ( V_in^succJ = ONE_Of_ONE and V_out^predN = ONE_Of_ONE) then
            // Nachziehen von Join-Knoten mit einem Nachfolger mit nur einer
            // eingehenden Kante
            HideActivityBypassEdge(CFS, DFS, join^CFS(n));
            succJ:= ReplaceActivity(CFS, DFS, succJ);
              V_in^succJ := V_in^join_CFS ;
            forall edge ∈ EC with Id_dest(edge) = join^CFS(n) do
               HideEdges(CFS, DFS, {e});
               e_new:= (Id_source(edge), succJ, CONTROL_E);
               AddVirtualEdges(CFS, DFS, {e_new});
            done
        endif ( V_in^succJ = ONE_Of_ONE )
    done forall n ∈ SplitNodes^CFS \ (nodeLabels ∩ SplitNodes^CFS)
end
```

Wir fassen die Vor- und Nachteile dieses Graphreduktionsalgorithmus zusammen:

Vorteile :

- Relativ leicht und effizient implementierbar.
- Geeignet um sich zu einem Schema einen Überblick über wichtige bzw. relevante Schritte zu verschaffen.
- Prozess-Instanzen können mit Views, die durch obigen Algorithmus generiert wurden, kontrolliert und geändert werden (Bearbeiter sieht nur seine Schritte, Kapselung eines Prozess für eine Anwendung mit Hilfe der Schritte für die diese Anwendung benötigt wird).
- Es ist eine View-auf-View Bildung möglich, da weitere Knoten weggelassen werden können.

Nachteile:

- Die Blockstruktur wird zerstört, deshalb nur noch eingeschränkte Nutzbarkeit der View.
- Nicht geeignet in der Prozess-Engine für Effizienssteigerungen, da Prozessteile komplett gelöscht werden.
- Nicht geeignet, um über die View das zugrunde liegende Schema zu ändern, da Kanten zusammenfallen bzw. gelöscht werden.

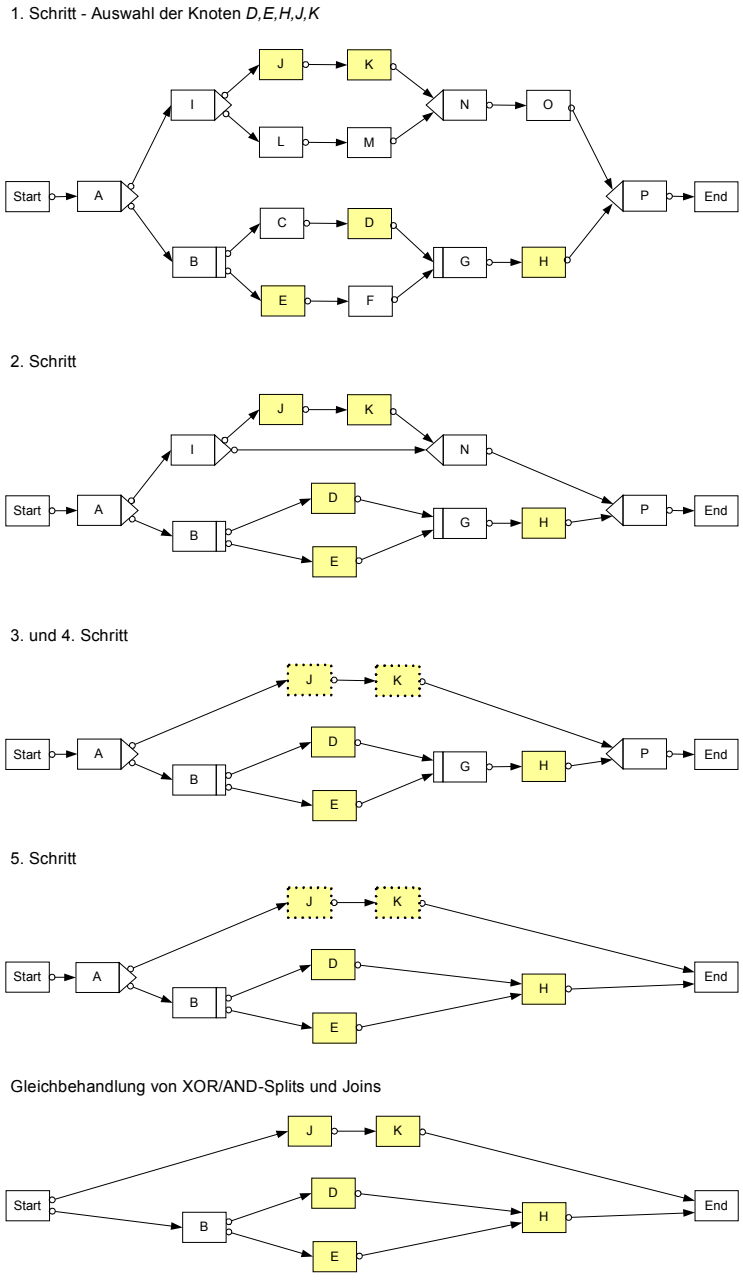

Abbildung 36: Weiteres Beispiel für View-Bildung durch Graphreduktion

Ein weiteres Beispiel für die Viewbildung mit Hilfe des Graphreduktionsalgorithmus liefert Abbildung 36. Der Algorithmus wird mit ReduceGraph(P, {}, {E, D, H, J, K}) aufgerufen (1. Schritt). Im 2. Schritt werden alle nicht relevanten Aktivitäten mit einer ein- und ausgehenden Kante (Knoten C, F, L, M und O) entfernt. Anschließend werden im 3. Schritt die Kante zwischen I und J entfernt, danach im 4. Schritt die Verzweigungsknoten I und J (sie

haben hier nur noch einen sichtbaren Zweig). Für den 5. Schritt sind in der Abbildung zwei Alternativen angegeben. Bei der ersten werden ein AND-Verzweigungsknoten soweit möglich entfernt (hier Knoten G), bei der zweiten Alternative auch die XOR-Knoten.

5.3.5 Graphaggregationsalgorithmus

Der Algorithmus des vorherigen Abschnittes hat den Nachteil, dass die dabei erzeugten Views nicht strukturerhaltend sind, und er sich deshalb für viele Anwendungsfälle[9] nicht eignet. Ein Algorithmus, der strukturerhaltend ist, sollte folgende Bedingungen erfüllen:

- Beibehaltung der Blockstruktur: Damit sind auf Views prinzipiell alle Operationen möglich, die auch auf Basisprozessen möglich sind (z.B. erneute Viewbildung, Schemaänderungen).
- Beibehaltung der Reihenfolge von Aktivitäten: Diese Anforderung entspricht weitgehend der in [9] aufgestellten Anforderung. Wenn ein Schritt im Basisprozess es erfordert, dass ein anderer Schritt vorher beendet wird, sollte das auch für die View gelten. Wenn andererseits ein Schritt nicht vor einem anderen abhängt, sollte er in der View auch nicht davon abhängen.

Für die Bildung strukturerhaltender Views dürfen Schritte nicht einfach gelöscht, sondern müssen durch im Modell gültige Elemente ersetzt werden. Dieser Ansatz führt zu einem Graphaggregationsalgorithmus, in dem mehrere Schritte eines Basisprozesses (die zusammen einen Kontrollblock bilden) durch einen virtuellen Schritt ersetzt werden.

Für den nachfolgenden Algorithmus müssen noch Nachfolger- und Vorgängerfunktionen für sichtbare Elemente, analog zu den in [4] beschriebenen Nachfolger- und Vorgängerfunktionen[10], definiert werden.

---

[9] z.B. für Schemänderungen über eine View

[10] Dies sind *succ* (direkte Nachfolger einer Aktivität), *c_succ* (direkte Nachfolger wenn nur Kontrollflusskanten betrachtet werden), *pred* und *c_pred* (entsprechende Funktionen für die direkten Vorgänger).

Definiton 9 (sichtbare direkte Vorgänger/Nachfolger eines Knoten)

Sei n (N,E, ...) ein KF-Graph. Dann gilt für ein n ∈ N:

(1): $succ\_vis: N \rightarrow 2^N$ *mit*
$succ\_vis(n) := succ(n) \cap N_{vi}$

(2): $c\_succ\_vis: N \rightarrow 2^N$ *mit*
$c\_succ\_vis(n) := c\_succ(n) \cap N_{vi}$

(3): $pred\_vis: N \rightarrow 2^N$ *mit*
$pred\_vis(n) := pred(n) \cap N_{vi}$

(4): $c\_pred\_vis: N \rightarrow 2^N$ *mit*
$c\_pred\_vis(n) := c\_pred(n) \cap N_{vi}$

Es werden für die bereits vorhanden direkten Vorgänger- und Nachfolgerfunktionen neue Varianten definiert, die in der Ergebnismenge nur diejenigen Knoten x der Ursprungsfunktionen enthalten, für die VISIBLE(x) = TRUE gilt.

Mit Hilfe dieser Funktionen kann jetzt ein Graphaggregationsalgorithmus angegeben werden. Auf eine genaue Darstellung des Algorithmus wird an dieser Stelle verzichtet, diese erfolgt im Anhang. Die einzelnen Schritte des Algorithmus werden hier informell anhand eines Beispiels beschrieben. Aufgerufen wird der Algorithmus mit *AggregateGraph(CFS, DFS, nodeSubsets)*, wobei *CFS* und *DFS* das Kontrollfluss-Schema und Datenschema des Prozessgraphen angeben und *NodeSubsets* eine Menge von Aktivitätenteilmengen[11] ist.

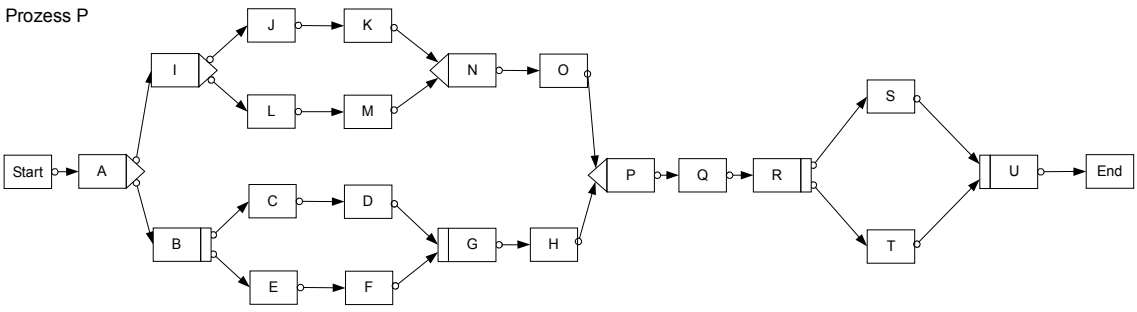

Abbildung 37: Basisprozess für Graphaggregation

Für das folgende Beispiel wird der Prozess auf Abbildung 37 verwendet.

Schritt 1: Bestimmung der Knotenmengen, deren Elemente zusammengefasst werden sollen. Dies kann entweder manuell oder auf Grundlage von Knotenattribute erfolgen. Im

[11] Dabei sind in jeder Teilmenge Aktivitäten enthalten, die in der View zu einer virtuellen Aktivität zusammengefasst werden sollen.

Beispiel aus Abbildung 38 dienen zu diesem Zweck die zwei Teilmengen {J, O} und {D}. Der Algorithmus wird also mit *AggregateGraph(P, {}, {{J, O}, {D}})* aufgerufen

Schritt 2: Für diese Knotenmengen erfolgt jeweils die Bestimmung der minimalen Kontrollblöcke. Dies entspricht der Menge von Knoten, die jeweils einen neuen virtuellen Knoten bilden. Hier werden im Beispiel die beiden Teilmengen M_1 = {I, J, K, L, M, N, O} und M_2 = {D} gebildet.

Die Schnittmengen sämtlicher auf dieser Art bestimmten Mengen müssen leer sein, ansonsten ist die Viewbildung nicht möglich.

Eventuell können solche überlappenden Mengen auch zu einer Menge, und damit zu einem virtuellen Schritt zusammengefasst werden. Damit ist in jedem Fall eine Viewbildung möglich. Dies führt aber im Extremfall dazu, dass die Prozess-View nur noch aus einem einzigen Schritt besteht.

Schritt 3: Die jetzt verbleibenden Knoten werden durch die maximal möglichen Kontrollblöcke, die keine der vorher bestimmten Knoten enthalten, zusammengefasst. Dies kann z.B. folgendermaßen geschehen:

Es wird aus der Menge der verbliebenen Knoten ein beliebiges Element ausgewählt und damit eine neue Menge gebildet. Zu dieser neuen Menge wird zum Start- und Endknoten (der bisher gebildeten Teilmenge[12]) der direkte Vorgänger bzw. Nachfolger hinzugenommen. Anschließend wird überprüft, ob der dann induzierte minimale Kontrollblock keine bereits verwendeten Knoten enthält. Dies wird solange wiederholt, bis kein Knoten mehr hinzugefügt werden kann.

Hierbei gilt eine Menge, die nur einen einzigen Split-, Join- oder Loop-Knoten enthält, auch als gültiger Kontrollblock.

Die somit ermittelten Knotenmengen bilden weitere virtuelle Knoten. In Abbildung 38 sind dies die Mengen {A}, {B}, {C}, {E, F}, {G}, {H}, {P} und {Q, R, S, T, U}.

[12] Dabei handelt es sich entweder um einen einzelnen Knoten oder einen Kontrollblock, deshalb gibt es immer einen eindeutigen Start- und Endknoten.

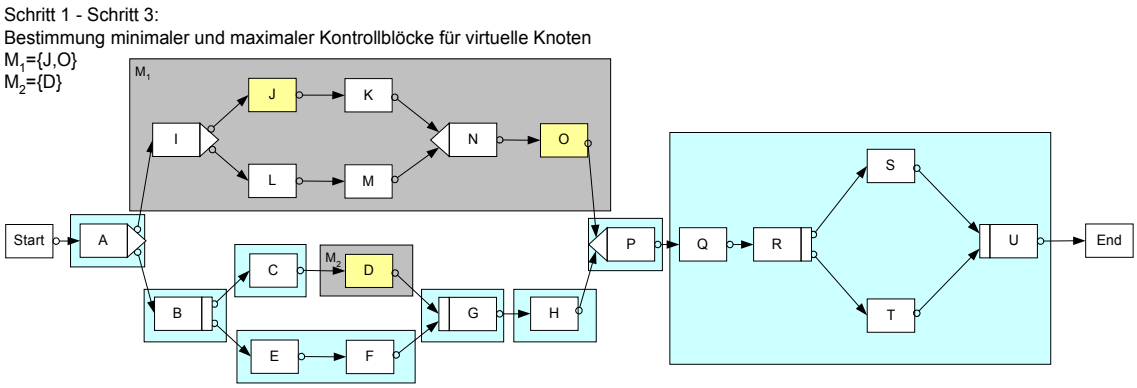

Abbildung 38: Bestimmung minimaler und maximaler Kontrollblöcke

Schritt 4: Die in Schritt 3 erzeugten Knoten können noch weiter zusammengefasst werden: Angenommen ein Split Knoten hat einen direkten Vorgänger, der seinerseits kein Split- oder Join-Knoten ist. Dann kann dieser Split-Knoten mit diesem Vorgänger zusammengefasst werden, ohne die Blockstruktur zu zerstören. Dasselbe gilt für Join-Knoten mit genau einem direkten Nachfolger, der kein Split- oder Join-Knoten ist.

Im Beispiel aus Abbildung 39 wurden die in den vorherigen Schritten 1 – 3 erzeugten virtuellen Knoten zwecks besserer Übersichtlichkeit als Raute (Knoten für relevanten Teilmengen) und Kreis (restliche virtuelle Knoten) dargestellt. Die Bedingungen für Schritt 4 sind für die zwei (virtuellen) Verzweigungsknoten T_5 und T_7 erfüllt. Diese können mit ihren direkten Nachfolgern T_6 bzw. T_8 zusammengefasst und aggregiert werden.

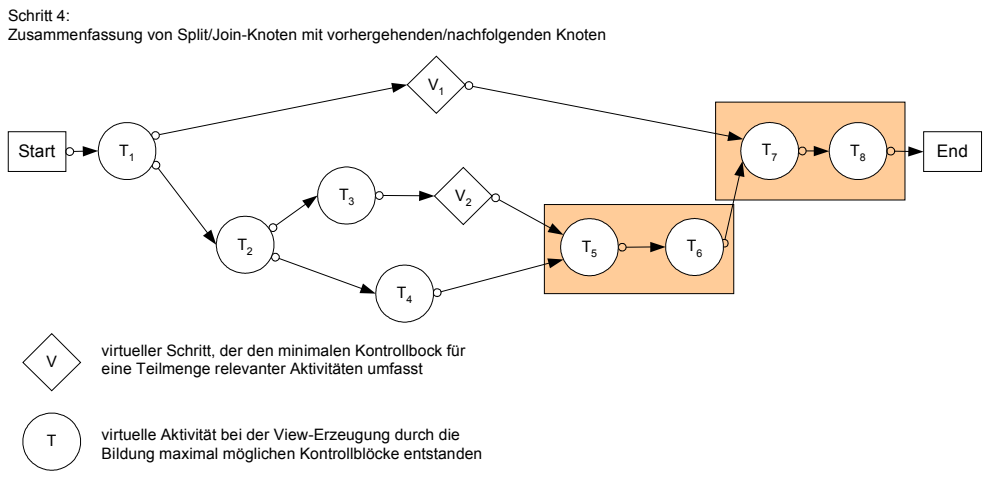

Abbildung 39: Zusammenfassung von Split/Join-Knoten

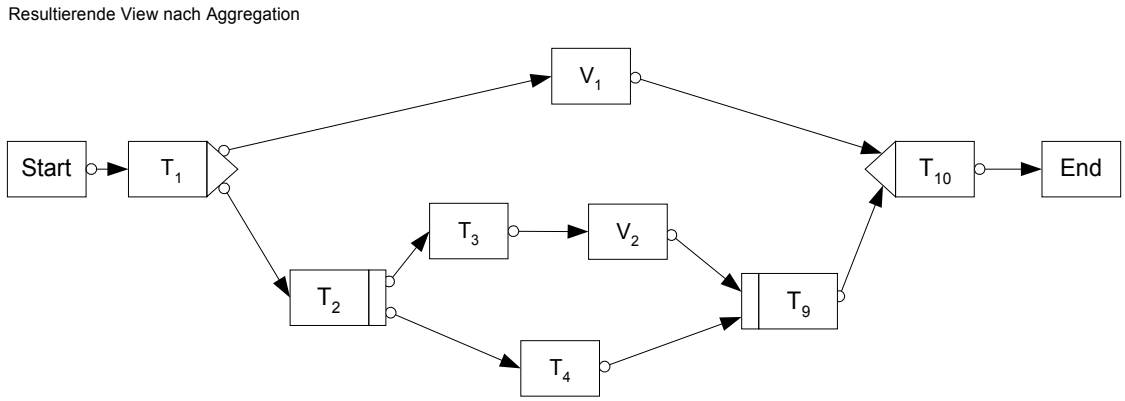

Abbildung 40: View nach Graphaggregation

Das Ergebnis des Aufrufs *AggregateGraph(P, {}, {{J, O}, {D}})* ist in Abbildung 40 dargestellt. Die Knoten T_5 und T_6 werden zum Knoten T9 und die Knoten T_7 und T_8 zum Knoten T_{10} aggregiert.

Über Views, die mit diesem Algorithmus erzeugt werden, kann das zugrunde liegende Basisschema geändert werden. Grund ist, dass jeder virtuelle Knoten durch Ersetzung eines Kontrollblocks entstanden ist und mit Hilfe der Attribute a_{start} und a_{end} immer die entsprechenden Knoten[13] des Basisprozess bestimmt werden können. Damit können bei Änderungsoperation immer die Knoten und Kanten des Basisprozesses bestimmt und die Änderungen dort durchgeführt werden.

Im Beispiel aus Abbildung 41 und 42 wird zwischen T_1 und V_1 über die View einer neuer Knoten X eingefügt. Da die Kante zwischen den virtuellen Knoten T_1 und V_1 eindeutig einer Kante im Basisschema zuordbar ist, ist es im Basisschema auch möglich, die Änderung zwischen den Knoten A und I durchzuführen. Dem Vorteil, dass der obige Algorithmus blockerhaltend ist und damit Schemaänderungen über eine View möglich werden, steht der Nachteil gegenüber, dass die Erzeugung einer View aufwändiger als mit dem Reduktionsalgorithmus ist (da hier viele neue virtuelle Knoten erzeugt werden). Zudem kann die resultierende View, je nach Graph und ausgewählter Knotenmenge, immer noch sehr komplex sein, da nicht relevante Knoten auch in der View noch vorhanden sind.

[13] Bei Views-auf-Views ist eventuell ein mehrfaches Lesen dieser Attribute notwendig, da auch Start-/ Endknoten aggregierte Knoten sein können.

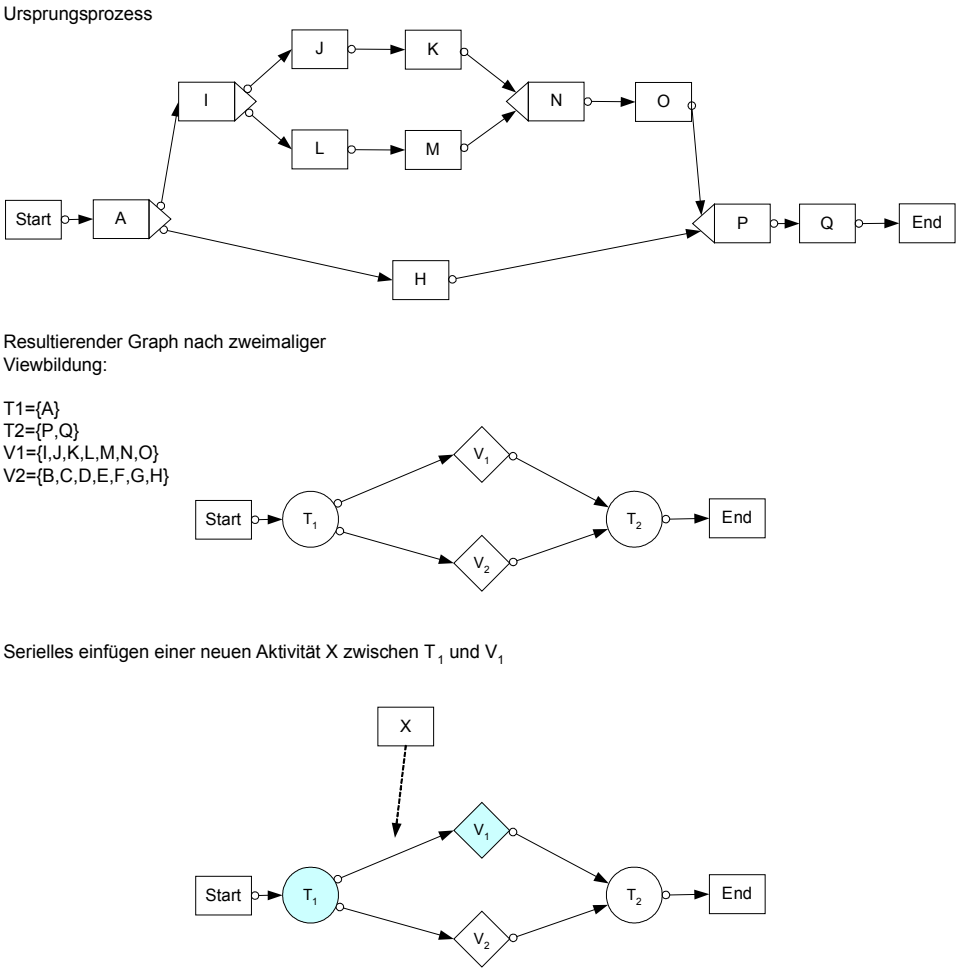

Abbildung 41: Schemaänderung über View

Beim obigen Algorithmus werden für die relevanten Knotenmengen jeweils die minimalen Kontrollblöcke gebildet. Dies kann dazu führen, dass irrelevante Schritte in der View enthalten bleibe, da keine großen Kontrollblöcke mehr gebildet werden können. Dies ist vor allem dann der Fall, wenn relevanten Knotenmengen einen minimalen Kontrollblock bilden, der innerhalb einer Verzweigung oder Loop-Struktur liegt. Dann müssen die Verzweigungsknoten (bzw. Loop-Knoten) als einzelne Knoten auch in der View erhalten bleiben, sie können dann bestenfalls noch mit dem direkten Vorgänger bzw. Nachfolger des Verzweigungsknotens vereinigt werden.

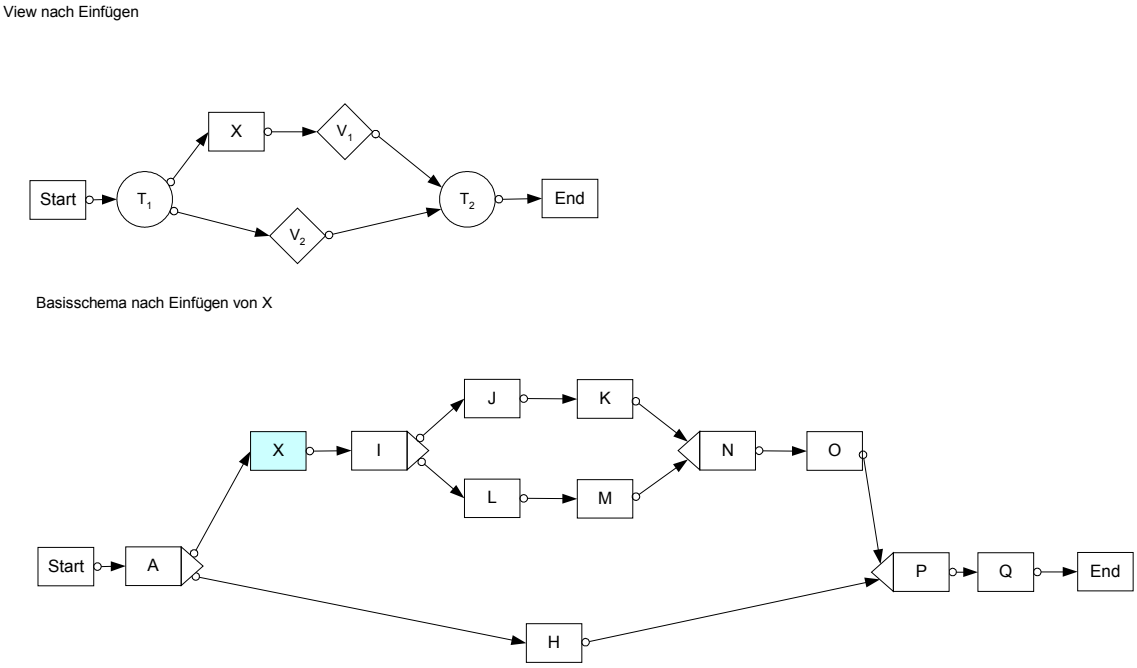

Abbildung 42: Resultierende View und Basisprozess

Kompaktere Views resultieren, indem obiger Algorithmus abgewandelt wird. Dazu werden im 2. Schritt nicht die minimalen Kontrollblöcke bestimmt, sondern größtmögliche Kontrollblöcke dergestalt, dass sie sich nicht mit Kontrollblöcken anderer virtueller Knoten überschneiden. Dazu muss festgelegt werden, in welcher Reihenfolge und wie weit pro Schritt die einzelnen Mengen erweitert werden.

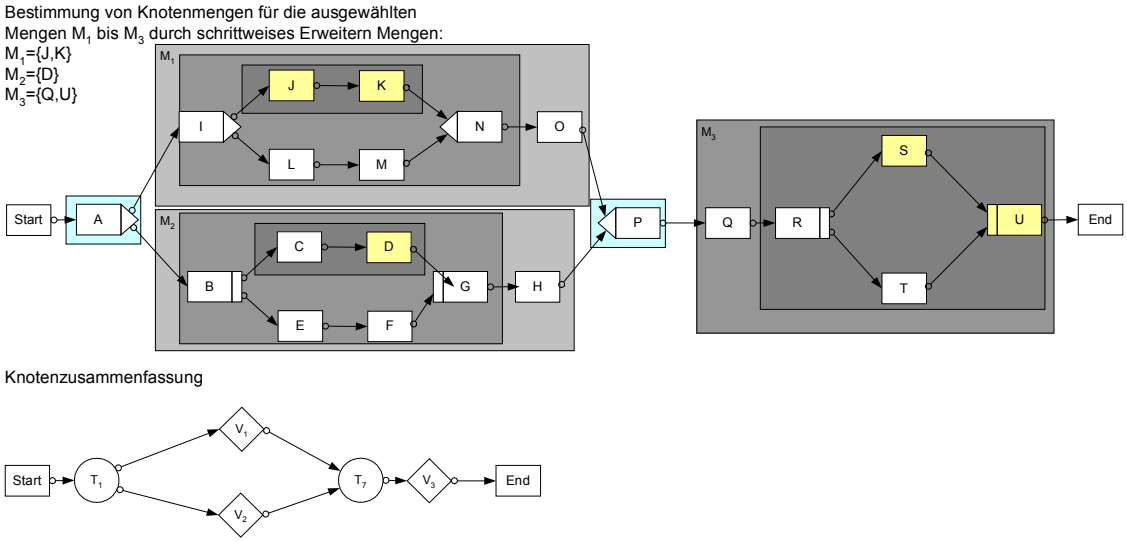

Abbildung 43: Viewbildung mit abgewandeltem Algorithmus

Im Beispiel aus Abbildung 43 wird jede Menge nacheinander solange wie möglich um einen Stufe erweitert. Eine Erweiterung um eine Stufe erfolgt dabei Hinzufügen des direkten Vorgängers/Nachfolgers des Start/Endknotens des bisherigen minimalen Kontrollblocks und anschließende Neubestimmung des minimalen Kontrollblock.

Mit dem vorgestellten Algorithmus ist es nicht möglich, einen in der Praxis häufig auftretenden Fall für die Viewbildung zu unterstützen. So ist es oftmals wünschenswert, bei einem Verzweigungsblock nur einen Zweig in die View zu übernehmen, und die anderen Zweige durch einen aggregierten Knoten zu ersetzen, wie im Beispiel aus Abbildung 1 auf Seite 5 dargestellt. Da der obige Algorithmus jedoch immer Knoten auf Kontrollblock-Ebene aggregiert, ist dies nicht möglich, obwohl eine solche View mit aggregierten Teilzweigen auch strukturerhaltend ist. Man kann jedoch mit einfachen Mitteln einen Algorithmus implementieren, der vor dem eigentlichen Aggregationsalgorithmus mehrere Teilzweige zu einem Knoten zusammenfasst. Die einzelnen Knoten desjenigen Teilzweiges, den man erhalten möchte, werden dann anschließend als einzelne Teilmenge (jeder Knoten ist das einzige Element einer Teilmenge) in *NodeSubsets* dem Aggregationsalgorithmus übergeben.

In der Praxis kann bei obigem Algorithmus ein Problem auftreten, wenn sich die minimalen Kontrollblöcke mehrerer relevanter Knotenmengen überlappen. Der Algorithmus vereinigt dann diese Menge, was im Extremfall dazu führen kann, dass die resultierende View nur noch aus einem Knoten besteht, wie Abbildung 44 verdeutlicht.

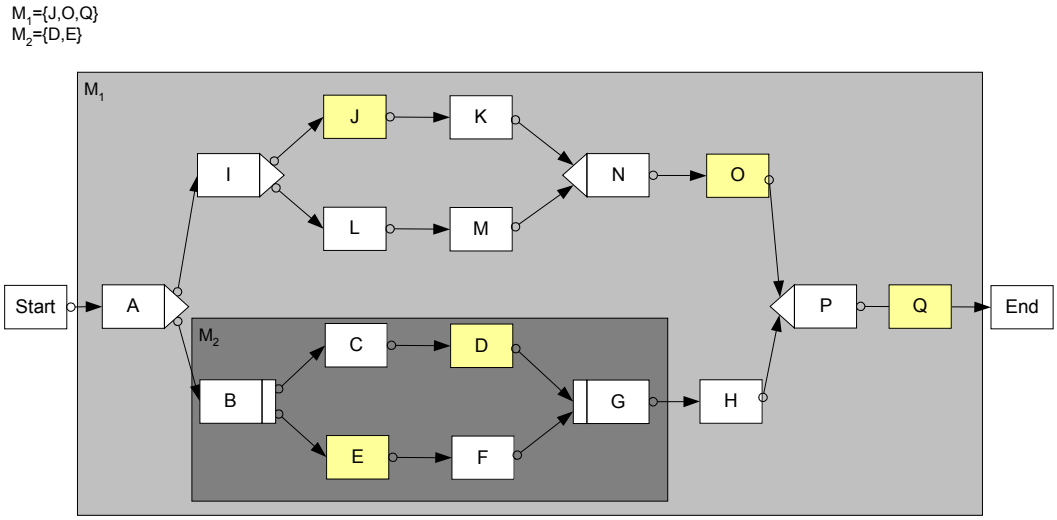

Abbildung 44: überlappende Kontrollblöcke für M_1 und M_2

Da eine durch den Graphaggregationsalgorithmus erzeugte View strukturerhaltend ist, kann auch eine View-auf-View Bildung erfolgen. Durch die Speicherung der Start- und Endknoten von aggregierten Kontrollblöcken ist es auch möglich, einen mehrstufig erzeugte View wieder auf den Basisgraphen zurückzuführen. Im Beispiel aus Abbildung 45 wird diese View-auf-

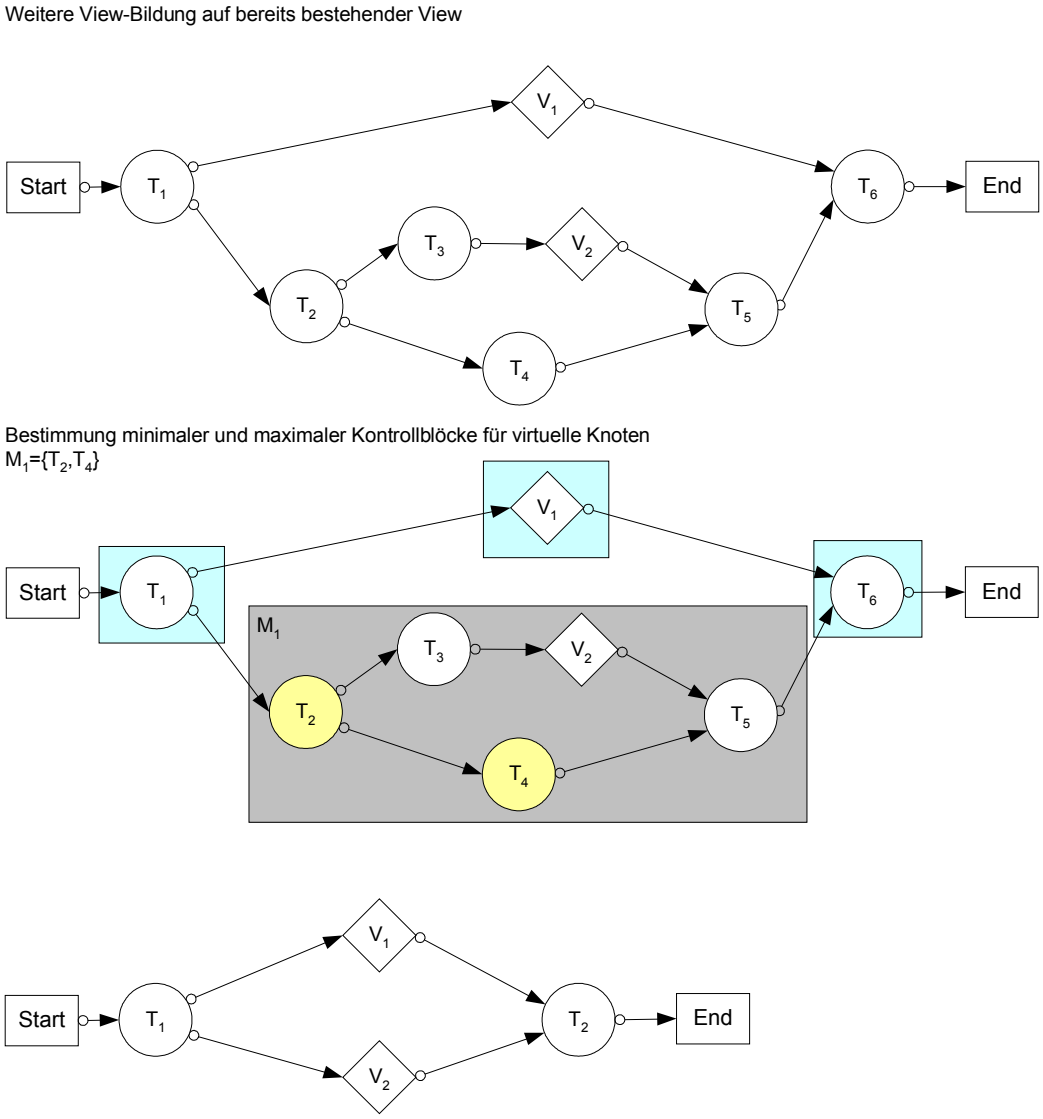

Abbildung 45: View-auf-View

View Bildung demonstriert. Für eine bereits existierende View (entspricht dem Beispiel aus Abbildung 40) wird eine weitere Viewbildung durch zusammenfassen des Verzweigungsblocks zwischen den Knoten T_2 und T_5 durchgeführt.

Der vorgestellte Algorithmus beachtet bei der View-Bildung keine Synchronisationskanten. Wenn über eine View das Schema geändert werden soll, ist es aber wichtig, auch diese einzubeziehen. Dabei dürfen Sync-Kanten zwischen zwei virtuellen Knoten nicht verschmolzen oder weggeblendet werden, da sich ansonsten die Ausführungssemantik der View ändert, wenn ein virtueller Knoten verschoben wird.

Deshalb muss bei der Viewbildung nach der Bildung der Knotenmengen (minimale Kontrollblöcke für die relevanten Knotenmengen, maximale Kontrollblöcke für die restlichen Knoten) für die virtuellen Knoten noch Folgendes beachtet werden:

1. Knotenmengen, die keine Sync-Kanten bzw. nur Sync-Kanten enthalten, deren Quell- und Endknoten in der gleichen Menge liegen, müssen nicht beachten werden.
2. Für die verbleibenden Knotenmengen muss jetzt paarweise überprüft werden, dass es nur eine Sync-Kante von der einen Menge zu der anderen Menge gibt. Ansonsten müssen die Mengen geändert werden (z.B. indem die zwei Mengen miteinander verschmolzen werden und anschließend der minimale Kontrollblock für die vereinigte Menge neu bestimmt wird).

Eine View, für die obige Bedingungen erfüllt sind, stellt alle Sync-Kanten zwischen virtuellen Knoten dar. Es werden keine Snyc-Kanten verschmolzen und es gibt keine Zyklen zwischen zwei Knoten (z.B. zwei Sync-Kanten, von denen eine zwei Knoten in die eine Richtung und die andere die gleichen Knoten in der umgekehrten Richtung verbindet). Für eine Sicht mit Sync-Kanten gibt es bei Schemaänderungen trotzdem gewisse Einschränkungen. Wenn Sync-Kanten mit virtuellen Knoten verbunden werden, ist nicht klar, auf welche Aktivitäten des Basisschemas sich die Kanten beziehen sollen. Sync-Kanten dürfen deshalb nicht gelöscht werden, oder es muss sichergestellt werden, dass sie nur für Basisaktivitäten geändert werden dürfen, oder aber sich immer auf den Start- oder Endknoten einer virtuellen Aktivität beziehen.

5.3.6 Erzeugung eines abstrakten Zustands für aggregierte Knoten

Wird für einen Prozess-Instanz eine aggregierte View gebildet, ist es wichtig auch für die einzelnen Knoten in der View Zustandsinformationen bereitzustellen. Damit kann der Nutzer auch über die View entsprechende Informationen über den Prozesszustand erhalten, z.B. darüber welche Teile eines Prozesses bereits beendet wurden.

Wenn Knoten zusammengefasst und durch einen virtuellen Schritt ersetzt werden, muss bei Anwendung der View ein geeignetes Mapping für die Prozesszustände erfolgen. Jeder Knoten, der innerhalb eines virtuellen Knoten abstrahiert wird, hat einen Zustand. Diese Zustände müssen mit Hilfe einer Funktion auf einen abstrakten Zustand abgebildet werden. Dies kann ähnlich wie unter [10] erfolgen, wo eine Zustandpriorisierung stattfindet, d.h. bestimmte Zustände überlagern andere Zustände.

Die Zustände seien wie unter [4] definiert. Dabei gibt es, wie im Statechart in Abbildung 46 ersichtlich, vier Hauptzustände (der in [4] zusätzlich definierte Zustand *ARCHIVED* wird hier nicht betrachtet).

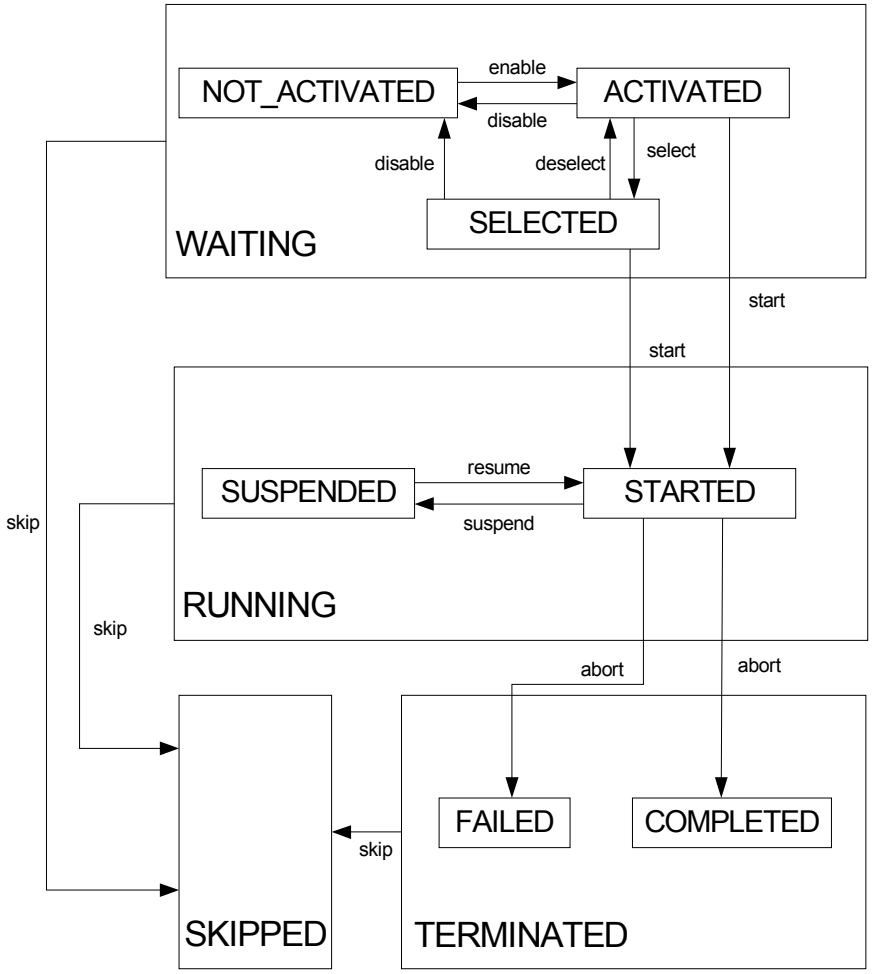

Abbildung 46: Statechart für die Zustände eine Aktivitäteninstanz

Die vier Hauptzustände haben folgende Bedeutung:

- *WAITING*
  Im Zustand *WAITING* ist eine Aktivität noch nicht gestartet und wartet auf ihre Ausführung.
  Initial befindet sich eine Aktivität im Zustand *NOT_ACTIVATED*. Sobald die Bedingungen für ihre Ausführung erfüllt sind, wechselt der Zustand zu *ACTIVATED*. Wenn eine Aktivität im Zustand *ACTIVATED* von einem Benutzer ausgewählt wird, wechselt der Zustand in *SELECTED*. Solange eine Aktivität nicht gestartet wurde (Zustand *RUNNING*) kann sie wieder deaktiviert werden, sie befindet sich dann wieder im Zustand *NOT_ACTIVATED*.

- *RUNNING*
  Wird eine Aktivität aktuell ausgeführt, befindet sie sich im Zustand *RUNNING*.
  Von den Zuständen *SELECTED* (für interaktive Aktivitäten) und *ACTIVATED* (nicht interaktive Aktivitäten) kann der Zustand einer Aktivität sofort oder durch Interaktion eines Benutzer in den Zustand *STARTED* wechseln. Ein Aktivität kann dabei zwischenzeitlich auch unterbrochen werden, in diesem Fall befindet sie sich dann im Zustand *SUSPENDED*.
- *TERMINATED*
  Wird eine Aktivität ausgeführt und anschließend beendet, befindet sie sich im Zustand *TERMINATED*.
  Vom Zustand *STARTED* ausgehend gelangt eine Aktivität entweder in den Zustand *COMPLETED,* wenn sie erfolgreich beendet wird, oder aber in den Zustand *FAILED,* wenn die aktuelle Bearbeitung scheitert.
- *SKIPPED*
  Von allen drei Hauptzuständen kann eine Aktivität in den Zustand *SKIPPED* wechseln. Dies ist dann der Fall, wenn die Bedingungen für die Ausführung der Aktivität nicht mehr gegeben sind. Wird z.B. in einer XOR-Verzweigung ein Zweig ausgewählt wurde, werden die Aktivitäten aller anderen Zweige auf *SKIPPED* gesetzt.

Mit Hilfe der oben definierten Zustände kann jetzt eine Abbildungsfunktion definiert werden.

- Gegeben: Knotenmenge, deren Elemente sich jeweils in einem bestimmten Zustand befinden, und die zusammen einen virtuellen Knoten bilden. Für diesen virtuellen Knoten soll auch ein Zustand vorhanden sein. Zustände der einzelnen Knoten seien wie oben definiert.
- Definition der Mapping-Regeln:

  1. Priorität von Running-Knoten. Wenn sich mindestens ein Knoten im Zustand *RUNNING* befindet, ist der Zustand des virtuellen Knoten ebenfalls *RUNNING*.
     Dabei gilt: *STARTED* > *SUSPENDED*, also wenn sich mindestens ein Knoten im Zustand *STARTED* befindet, so befindet sich auch der virtuelle Knoten im Zustand *STARTED*.
     In Abbildung 47 wird der Zustand des virtuellen Knotens V auf *STARTED* gesetzt, da sich ein Knoten (hier Knoten B) des Basisprozesses sich in diesem Zustand. Alle anderen Knoten dieses Beispiels sind entweder im Zustand *COMPLETED* oder *NOT_ACTIVATED* (alle anderen Knoten ohne Zustandssymbol)

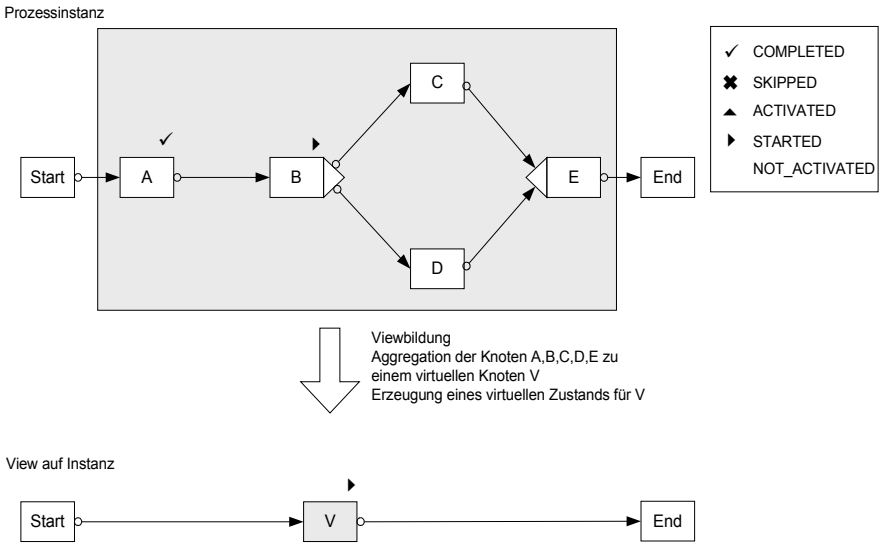

Abbildung 47: Zustandsmapping für RUNNING-Knoten

2. Falls obige Bedingung nicht erfüllt ist, haben Knoten im Zustand *WAITING* Vorrang. Falls ein Knoten sich im Zustand *WAITING* befindet, so weist auch der virtuelle Knoten diesen Zustand auf. Es gilt: SELECTED > *ACTIVATED* > *NOT_ACTIVATED*.

   Im Beispiel aus Abbildung 48 wird ein Zweig der XOR-Verzweigung (Konten C) ausgewählt (Zustand *ACTIVATED)* und deshalb alle anderen (Knoten D) auf

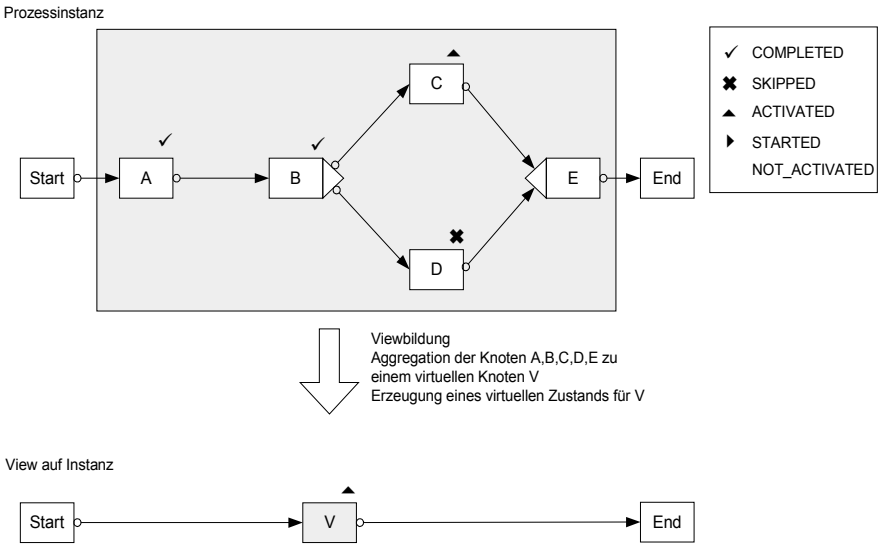

Abbildung 48: Zustandsmapping für WAITING-Knoten

SKIPPED gesetzt. Der virtuelle Knoten wird, da sich zwei Knoten[14] im Zustand *WAITING* befinden, auf den Zustand *ACTIVATED* gesetzt. Dabei hat der Zustand von Knoten C nach obiger Regel eine höhere Priorität als der Zustand von Knoten E.

3. Falls keine der obigen zwei Bedingungen zutreffen befinden sich alle Knoten der Menge in einem der Zustände *FAILED, COMPLETED* oder *SKIPPED*. Wenn durch die Viewbildung gewährleistet ist, dass jeder virtuelle Knoten genau einen Start- und Endknoten im Basisprozess hat[15], kann der Zustand des virtuellen Knotens einfach durch den Zustand des Endknotens (des Basisprozesses) dargestellt werden.

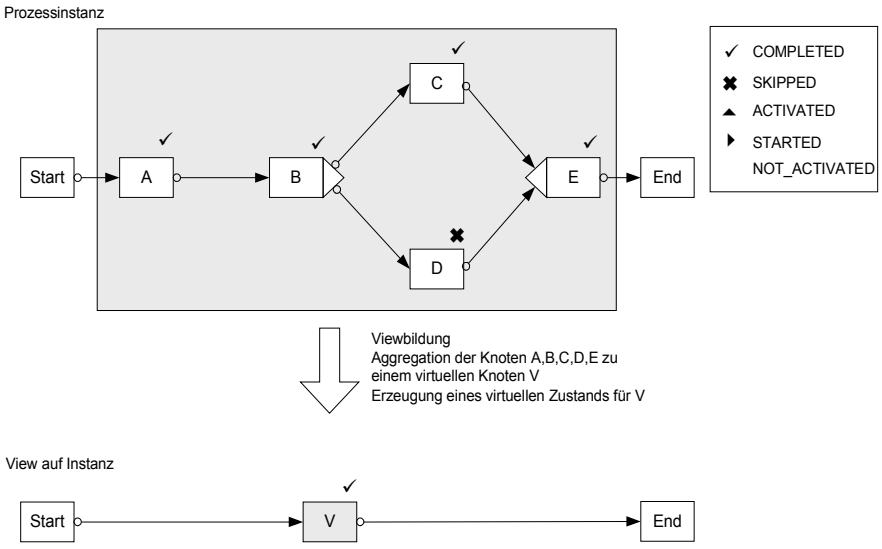

Abbildung 49: Zustandsmapping für COMPLETED- und SKIPPED-Knoten

Am Beispiel aus Abbildung 49 wird dies deutlich. Alle Knoten innerhalb der aggregierten Teilmenge befinden sich im Zustand *TERMINATED* oder *SKIPPED* – deshalb wird der Zustand des virtuellen Knotens auf den Zustand des Endknoten (im Beispiel Knoten E) des betreffenden virtuellen Knotens gesetzt, in diesem Fall auf *COMPLETED.*

[14] C im Zustand *ACTIVATED* und D im Zustand *NOT_ACTIVATED.*
[15] Dies ist bei dem in Abschnitt 5.3.5 vorgestellten Algorithmus der Fall.

Formal lassen sich die im vorherigen Absatz formulierten 3 Regeln folgendermaßen ausdrücken:

Definiton 10 (Zustandsmapping für virtuelle Knoten)
Sei $CFS_{instance}$ eine Prozess-Instanz und CFS_{vi} eine durch Graphaggregation gebildete Sicht auf $CFS_{instance}$.
Sei ferner NodeStates := {NOT_ACTIVATED, ACTIVATED, SELECTED, STARTED, SUSPENDED, COMPLETED, FAILED, SKIPPED} eine Menge von Zuständen. Mit Hilfe der Abbildung $NS: N \rightarrow NodeStates$ weist jeder Aktivität aus N ein Zustand aus der Menge NodeStates zugeordnet.
Sei V ein durch Graphaggregation entstandener virtueller Knoten in CFS_{vi}. Es sei die Menge M, $M \subset N_{instance}$, die durch den virtuellen Knoten V zusammengefasste Teilmenge an Basisaktivitäten aus $CFS_{instance}$.
Dann sei eine Funktion $MapStates: 2^N \times N \rightarrow NodeStates$ zur Abbildung der Zustände aus M für einen virtuellen Knoten V auf einen Zustand $z \in$ NodeStates wie folgt definiert:

MapStates(M, V)
input:
M: Teilmenge von Basisaktivitäten einer Prozess-Instanz
V: virtueller Knoten, der die Teilmenge M aggregiert
output:
virtualState: virtueller Zustand für V, virtualState ∈ NodeStates
begin:
// Regel 1:
// virtueller Knoten wird auf Zustand RUNNING gesetzt, STARTED > SUSPENDED
if ∃ n∈M mit NS(n) = STARTED **then return** STARTED
else if ∃ n∈M mit NS(n) = SUSPENDED **then return** SUSPENDED
// Regel 2:
// virtueller Knoten wird auf WAITING gesetzt,
// NOT_ACTIVATED > ACTIVATED > SELECTED
else if ∃ n∈M mit NS(n) = SELECTED **then return** SELECTED
else if ∃ n∈M mit NS(n) = ACTIVATED **then return** ACTIVATED
else if ∃ n∈M mit NS(n) = NOT_ACTIVATED **then return** NOT_ACTIVATED
// Regel 3:
// falls bisher keine Bedingung erfüllt wurde sind alle Knoten aus M im Zustand
// TERMINATED oder SKIPPED, gib Zustand des Endknotens zurück
else
n_{end} := ATTRIBUTE(V,a_{end}); *// Endknoten der virtuellen Aktivität*
return NS(n_{end}); *// Zustand des Endknotens*
end

Je nach Festlegung der Mapping Funktion (siehe vorherigen Abschnitt) können die Zustandsänderungen innerhalb eines virtuellen Knoten (Zusammenfassung mehrerer Basisknoten) in einer anderen Reihenfolge erfolgen als in einem Basisknoten (in einem Basisknoten kann z.B. auf *RUNNING* lediglich einer Zustände *TERMINATED* oder *SKIPPED* folgen, in einem virtuellen Knoten kann nach *RUNNING* jedoch auch wieder *WAITING* kommen, da intern um einen Schritt weitergeschaltet wurde). Wenn dieses Verhalten nicht erwünscht ist, muss die Mapping-Funktion entsprechend angepasst werden.

Eine Möglichkeit dazu wäre folgende (Aggregation von Zuständen):

1. Wenn alle Knoten sich im Zustand *NOT_ACTIVATED* befinden, so ist auch der virtuelle Knoten im Zustand *NOT_ACTIVATED*. Es befinden sich alle Knoten einer aggregierten Teilmenge im Zustand NOT_ACTIVATED, wenn sich der Startknoten in diesem Zustand befindet. Wenn der Startknoten nicht aktiviert wurde, gilt dies selbstverständlich für alle darauf folgenden Knoten.
2. Wenn sich mindestens ein Knoten der Menge im Zustand *WAITING* oder *RUNNING* befindet, aber nicht im Zustand *NOT_ACTIVATED*, wird der Zustand des virtuellen Knotens auf *NOT_COMPLETED* gesetzt. Dies sei ein neu definierter Zustand nur für virtuelle Knoten, mit dem ausgedrückt wird, das mindestens eine Aktivität innerhalb des virtuellen Knoten aktiviert ist oder bereits läuft. Angewendet auf die obigen Beispiele aus Abbildung 47 und 48 würde der virtuelle Knoten V jeweils auf den Zustand *NOT_COMPLETED* gesetzt werden, da sich in der aggregierten Teilmenge ein Knoten in den Zuständen *RUNNING* bzw. *ACTIVATED* befindet.
3. In allen anderen Fällen gilt das gleiche wie in der im vorherigen Absatz vorgestellten Mapping-Funktion unter Punkt 3.

Formal kann diese geänderte (durch Einführung des *NOT_COMPLETED* Zustands) Mapping-Funktion folgendermaßen angegeben werden:

```
MapStates(M, V)
  input:
      M: Teilmenge von Basisaktivitäten einer Prozess-Instanz
      V: virtueller Knoten, der die Teilmenge M aggregiert
  output:
      virtualState: virtueller Zustand für V, virtualState ∈ NodeStates ∪
      {NOT_COMPLETED}
  begin:
      // Regel 1
      if  ∀ n ∈M gilt: NS(n) =NOT_ACTIVATED     then return NOT_ACTIVATED
      // Regel 2:
      else if  ∃ n∈M mit NS(n) = SELECTED  or NS(n) = ACTIVATED
         then return NOT_COMPLETED
      // Regel 3:
      else
         n_end := ATTRIBUTE(V,a_end);  // Endknoten der virtuellen Aktivität
         return NS(n_end);  // Zustand des Endknotens
  end
```

5.3.7 Automatische Generierung von Views

Mit Hilfe der in den vorherigen Abschnitten vorgestellten Algorithmen ist es möglich, auf Graph-Ebene Views zu bilden. Allerdings stellt sich die Frage, wie die für die View-Bildung notwendigen initialen Knotenmengen bestimmt werden können, ohne dass diese von einem Anwender manuell ausgewählt werden müssen. Wünschenswert wäre es, wenn aufgrund bestimmter Eigenschaften (z.B. Wert von Knotenattributen) eines Prozesses (oder einer Prozess-Instanz) diese Knotenmengen, und damit auch die View automatisch ermittelt, bzw. erzeugt werden könnte.

Einen Ansatz dazu liefert [11]. Dort werden für jede Aktivität (*Task*) der Relevanzgrad für eine Rolle bestimmt und anschließend Knotenmengen gebildet, deren Relevanzgradsumme einen gewissen Schwellenwert (*Threshold*) nicht überschreitet. Mit Hilfe des Schwellenwertes kann also die Granularität der View festgelegt werden. Dieser Ansatz mit einem Relevanzgrad hat, bezogen auf Rollen, den Vorteil, dass nicht ein feste Rolle für die Viewbildung angegeben werden muss, sondern durch den Relevanzgrad auch Vertreterregelungen (die dann z.B. eine niedrigere Relevanz als die eigentlich zugeordnete Rolle haben) berücksichtigt werden können.

Dieser Ansatz kann auch für die obigen Algorithmen verwendet und etwas verallgemeinert werden. Anstatt nur für Rollen kann genauso für beliebige Attribute oder auch eine Kombination von Attributen ein Relevanzgrad bestimmt werden. Dabei sei der Relevanzgrad f_r

eine Abbildung $f_r: a_1 \times a_2 \times ... \times a_n \rightarrow \mathbb{R}$, die einen oder mehrere Attributwerte a_1 bis a_n auf eine reelle Zahl r abbildet, wobei je größer r, desto größer sei auch der Relevanzgrad. Möglich ist auch eine normierte Relevanzfunktion $f_r: a_1 \times a_2 \times ... \times a_n \rightarrow [0,1]$, die die Attributwerte auf ein Intervall zwischen 0 (keine Relevanz) und 1 (höchste Relevanz) abbildet. Allerdings steht der Bestimmung des Relevanzgrades in der Praxis ein Problem dar, da oftmals nicht bekannt ist, wie oft z.B. eine Aktivität von einem bestimmten Nutzer oder Rolle (Rollenrelevanz) durchgeführt wird oder wie lange eine Aktivität dauert (Relevanz nach Dauer einer Aktivität).

Für die Bestimmung des Relevanzgrades gibt es mehrere Möglichkeiten:

1. Ein Relevanzgrad wird für ein oder mehrere Attribute manuell vom Viewdesigner vorgegeben. Dies ist nur sinnvoll, wenn die Angabe eines Relevanzgrades einfacher durchführbar ist, als die manuelle Auswahl der Knotenmengen (sinnvoll z.B. bei sehr komplexen Prozessen, bei denen jedoch die Relevanz einfach festgelegt werden kann, z.B. Attributwert a vorhanden, $f_r(a)=1$, ansonsten $f_r(a)=0$)
2. Bei manchen Attributen könnte direkt der Wert des Attributs als Relevanzgrad dienen, z.B. beim Attribut Kosten. So kann aufgrund des Attributs Kosten eine View gebildet werden, in der die einzelnen virtuellen Knoten der View Aktivitäten des Basisprozesses zusammenfassen, die in der Summe einen bestimmten Schwellenwert an Kosten nicht überschreiten.
3. Für andere Attribute (z.B. bei Rollen) könnte eine Auswertung von bereits durchlaufenen Prozess-Instanzen oder auch zusätzlich verfügbare Informationen (bei Rollen oder Benutzern z.B. Zugriffsberechtigungen) dazu verwendet werden, einen Relevanzgrad zu bestimmen.

Sobald für jede Aktivität ein Relevanzgrad bestimmt wurde, ist es leicht möglich, für den Graphreduktionsalgorithmus eine Menge von relevanten Knoten zu bestimmen: Die Menge enthält einfach alle Knoten, deren Relevanzgrad über einem vorher festgelegten Schwellenwert liegt.

Mit Hilfe des Relevanzgrades ist es auch möglich, Knotenmengen für den Graphaggreationsalgorithmus zu bestimmen. Idee ist, dass man viele Knoten, die nur eine geringe Relevanz haben in der View in einem Knoten zusammenfassen kann und Knoten mit hoher Relevanz auch in der View genauer (indem z.B. nur 2 Knoten zusammengefasst werden) darstellt. Dabei kann die Stärke der Viewbildung über den Schwellenwert, bis zu dem Aktivitäten zusammengefasst werden, gesteuert werden. Dafür ist noch eine Funktion $F_r: A \rightarrow \mathbb{R}$ notwendig, die für eine Menge von Aktivitäten A den maximal möglichen Relevanzgrad $r \in \mathbb{R}$ berechnet. Dafür wird einfach die Summe der Relevanzgrade der

einzelnen Aktivitäten berechnet, bei XOR-Verzweigungen wird nur der Zweig mit dem höchsten Relevanzgrad berücksichtigt.

```
GetNodeSet(CFS, DFS, threshold)
  input:
      CFS, DFS: WF-Graph, für den eine View gebildet werden soll
      threshold: Schwellenwert, der nicht überschritten werden soll
  output:
      nodeSubsets: Menge, die Teilmengen von N enhält:  x∈NodeSubsets ⇒ x⊆N
  begin:
      tmp:= N; // tmp enthält alle noch nicht verwendeten Knoten
      nodeSubsets := ∅; //Menge an ausgewählten Blöcken
      while (tmp ≠ ∅) do
         P:= Queue, die alle Knoten n aus tmp enthält für die gilt: c_pred(n) ∩ tmp = ∅;
         while (P ≠ ∅) do
            x:= dequeue(P);
            nodeSet := {x};
            tmp:= tmp – {x};
            // Für alle direkten Nachfolger von x prüfen, ob der dadurch entstehende
            // Kontrollblock noch unter dem Schwellenwert bleibt
            Q:= ∅; //Queue für die Aufnahme der direkten Nachfolger von s
            forall s in (c_succ(x) ∩ tmp) do enqueue(Q, s); done
            while (Q ≠ ∅) do
               q:= dequeue(Q);
               (x_start, x_end) = MinBlock(CFS, x, q);
               nodeBlock := succ_c*(x_start) ∩ pred_c*(x_end) ∪ {x_start} ∪ {x_end};
               if (F_r(nodeBlock ∪ nodeSet) <= threshold) then
                  forall s in (c_succ(x_end)) do enqueue(Q,s);
                  nodeSet:= nodeSet ∪ nodeBlock;
                  tmp:= tmp \ nodeBlock;
                  forall s in nodeBlock do
                     remove(Q,s); //Entferne bereits erledigte Knoten aus Q
                  done
               endif
            done while (Q ≠ ∅)
            nodeSubsets := nodeSubsets ∪ {nodeSet};
         done while (P ≠ ∅)
      done while (tmp ≠ ∅)
      return nodeSubsets;
  end
```

Der Algorithmus arbeitet folgendermaßen: Ausgehend vom Startknoten wird immer mit den direkten Nachfolgern jeweils eine Menge *nodeSet* gebildet und anschließend für dessen direkte

Nachfolger der minimalen Kontrollblock bestimmt. Anschließend wird überprüft, ob für die Menge der im Kontrollblock enthaltenen Aktivitäten und der bisherigen Menge *nodeSet* der Schwellenwert *threshold* noch unterschritten ist. Falls dies der Fall ist, werden alle Aktivitäten aus dem Kontrollblock zur Menge *nodeSet* hinzugefügt. Diese Prozedur wird solange wiederholt, bis sämtliche Knoten (tmp = ∅) zu *nodeSubsets* hinzugefügt wurden. Falls für manche Aktivitäten die Relevanz größer als der Schwellenwert ist, wird dieser Knoten unverändert in die View übernommen.

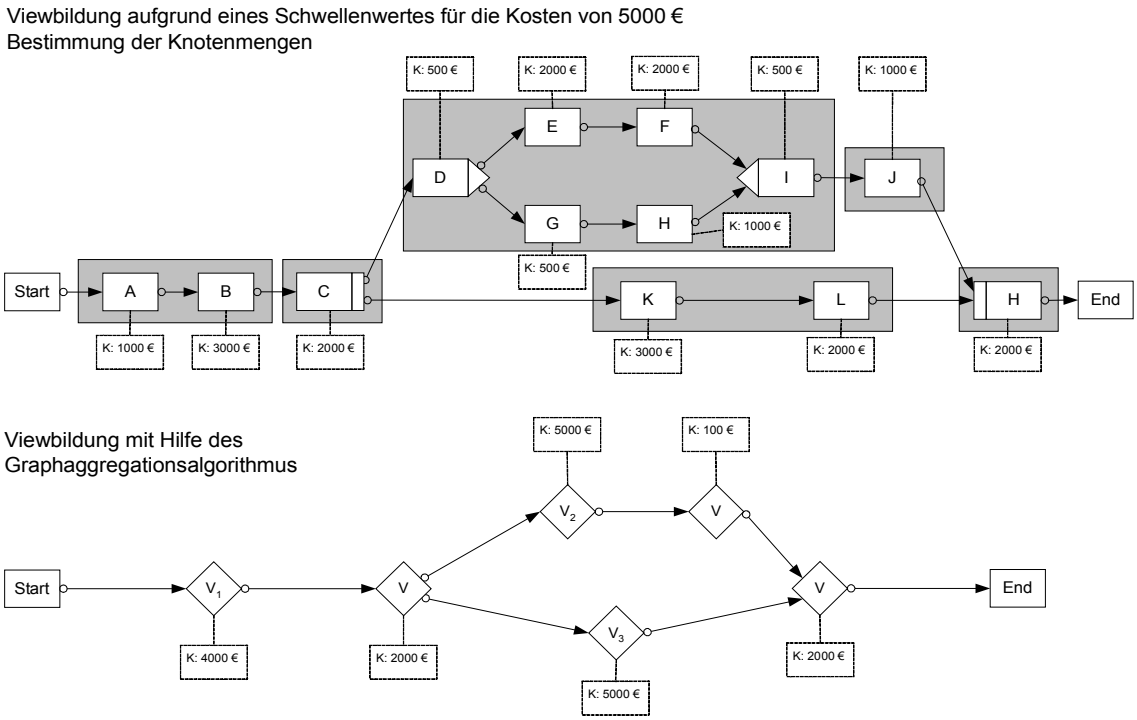

Abbildung 50: Viewbildung aufgrund von Kosten

Im Beispiel aus Abbildung 50 werden relevante Knotenmengen aufgrund der Kosten für eine Aktivität mit eine Schwellenwert von 5.000 EUR, beginnend beim Startknoten, bestimmt. Die Viewbildung erfolgt mit dem Graphaggregationsalgorithmus.

Ein Problem des obigen Algorithmus ist die Abhängigkeit vom Start und der Richtung der Aggregation. Eine View hat eine andere Struktur, wenn die Knotenmengen von Start- zum Endknoten bestimmt werden, als wenn dies vom End- zum Startknoten geschieht. Wünschenswert wäre eine Lösung, bei der eine minimale Anzahl (es gibt keine andere View mit weniger Knoten, bei der die einzelnen aggregierten Knoten unter dem Schwellenwert bleiben) von Knoten übrig bleibt. Eine Möglichkeit, um eine möglichst ideale Aggregation zu erzielen, ist das mehrfache Durchführen der Aggregation, mit verschiedenen Richtungen und Startknoten, um dann als Resultat die View mit den wenigsten Knoten zu verwenden.

5.3.8 Entfernung und Aggregation von Knoten bei Prozess-Instanzen

Bei der Bildung von Views auf Instanzen der Zustands[16] der Instanz zugrunde gelegt werden. Die Auswahl der zu entfernenden Knoten wird dabei auf Grundlage des *Zustands* der Knoten getroffen (z.B. „alle erledigten und abgebrochenen Knoten entfernen“). Dies Art der Viewbildung ist vor allem für zwei Anwendungsfälle sinnvoll:

- Unterstützung von Views in der Prozess-Engine, etwa um durch einfachere Prozessgraphen eine effizientere Bearbeitung von Prozessen zu ermöglichen
- Views für den Endbenutzer: Wenn es bei sehr großen, komplexen Prozessen aus bestimmten Gründen nicht möglich oder erlaubt ist, Knoten wie in den obigen Beispielen zu entfernen (durch Reduktion oder Aggregation), kann durch ein Entfernen bereits erledigter Knoten dennoch eine Vereinfachung des Graphen erzielt werden.

Die Viewbildung kann sehr ähnlich dem in Abschnitt 5.3.5 vorgestelltem Aggregationsalgorithmus erfolgen, wobei jedoch auch Teile des Graphen entfernt werden. Hierbei ist das Ziel, möglichst alle beendeten oder übersprungenen Knoten zu entfernen oder zusammenzufassen, jedoch die Blockstruktur zu erhalten.

1. Bildung größtmöglicher Kontrollblöcke, die nur Knoten enthalten, die nicht mehr zur Ausführung gelangen können[17]. Dies kann wie oben beschrieben durch schrittweises Erweitern der einzelnen Mengen um den direkten Nachfolger und anschließende Neubestimmung des minimalen Kontrollblocks erfolgen. Die so bestimmten Mengen werden durch jeweils einen Knoten ersetzt, der den Zustand des Endknoten der Menge erhält.
2. Entfernung sämtlicher jetzt noch vorhandener AND/XOR-Verzweigungen, bei denen nur noch ein Zweig aktiv ist (die Knoten alle anderen Zweige sind mit *TERMINATED* oder *SKIPPED* markiert).

---

[16] Dies ist ähnlich dem vorherigen Abschnitt, wo die View-Bildung aufgrund von Attributwerten erfolgte.

[17] Wenn sich die Knoten im Zustand *TERMINATED* oder *SKIPPED* befinden.

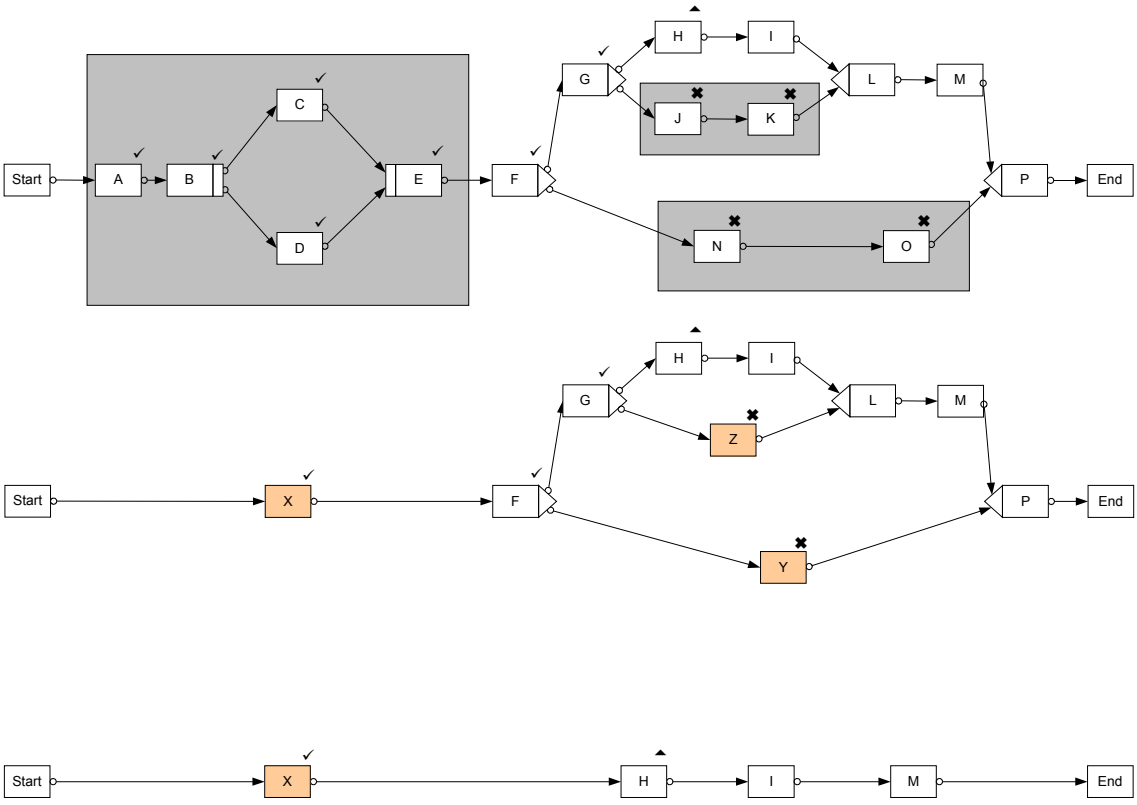

Abbildung 51: Zusammenfassung von erledigten Schritten, und anschließende Elimination von Verzweigungen, die nur noch einen aktiven Zweig besitzen

Der Algorithmus kann auch verwendet werden, um Knoten, die erst in ferner Zukunft zur Ausführung gelangen, zusammenzufassen. Dabei ist eine Funktion notwendig, die im 1. Schritt die Menge der in Frage kommenden Knoten bestimmt. Anstatt Kontrollblöcke für erledigte und abgebrochene Knoten zu bilden, kann dies z.B. auch für Knoten, die frühestens in x Zeiteinheiten in der Zukunft zur Ausführung gelangen, erfolgen.

5.3.9 Zusammenfassung mehrerer Prozess-Instanzen

Wenn es viele Instanzen gleichen Typs gibt, kann durch ihre Abbildung auf einen Graphen ein Überblick geschaffen werden.
Die Konstruktion einer solchen View ist einfach:

- Knotenattribute können auf beliebige (je nach Typ des Attributs mehr oder weniger sinnvolle) Weise aggregiert werden (Summe, Durchschnitt, Maximum, Minimum usw.). Siehe hierzu auch Abschnitt 4.2.2.
- Für jeden Knoten wird die Anzahl der jeweiligen Zustände, in denen sich die Knoten der einzelnen Instanzen befinden, angegeben.

Ein Beispiel zeigt Abbildung 20. Wichtig ist dabei, dass alle Instanzen vom gleichen Typ sind. Wurden von manchen Instanzen während der Laufzeit das Schema geändert, ist eine Zusammenfassung nur dann möglich, falls es gelingt, ähnlich wie in der Objektorientierten Programmierung für alle Instanzen einen gemeinsamen „Supertyp“ zu bestimmen, und dann Knoten der „Untertypen“ auf einen Knoten des „Supertyps“ zu mappen, ähnlich wie es in [12] und [13] für Petrinetze beschrieben wird.

5.4 View-Definition aus Benutzersicht

Mit Hilfe der in den vorherigen Abschnitten vorgestellten Algorithmen ist es möglich, Views zu erzeugen. Für Anwender sollten Views möglichst einfach definierbar sein. Denkbar wäre z.B. die View-Definition mit Hilfe eines graphischen Editors. Für die Definition von Views ist es jedoch sinnvoll, wenn man zwischen der eigentlichen View-Engine und graphischen Tools eine Schicht setzt, in der Views definiert werden können. Die View-Definition kann dabei ähnlich wie bei relationalen Datenbanksystemen mit Hilfe einer DDL (*Data Definition Language)* definiert werden. Folgende Elemente müssen dabei in einer DDL angegeben werden:

- Name der View
- Schema/Prozess-Instanz für die View gebildet werden soll
- Angabe ob View mittels Graphreduktion oder Graphaggregation erzeugt werden soll
- Angabe relevanter Knoten bzw. relevanter Knotenmengen
- Bei Knotenmengen: Angabe eines Namen für neu erzeugten virtuellen Knoten

Eine Query könnten dann z.B. folgendermaßen aussehen:

```
CREATE (AGGREATED | REDUCED) VIEW ViewName AS
SELECT (Node1 | SET(Node11 , Node12 , ..., Node1n) ) AS VirtualNodeName1,
       (Node2 | SET(Node21 , Node22 , ..., Node2n) ) AS VirtualNodeName2,
       ...,
       (Noden | SET(Noden1 , Noden2 , ..., Nodenn) ) AS VirtualNodeNamen
(REMAINING AS VirtualNodeName (ASC | DESC) )
FROM   ProcessName | ViewName
```

Mit Hilfe dieser DDL lassen sich Views für den Kontrollfluss definieren. Bei der Graphaggregation können, neben den Knoten für die relevanten Knotenmengen noch virtuelle Knoten für die restlichen in dem Prozessgraph vorhandenen Knoten entstehen. Um diese fortlaufend mit einem Namen und einer Nummer benennen zu können kann die *REMAINING AS* Klausel verwendet werden.

Beispiel für eine Definition zur Erzeugung einer graphreduzierten View (Abbildung 35):

```
CREATE REDUCED VIEW EXAMPLE_VIEW_1 AS
SELECT A, E, F, J, L
FROM EXAMPLE_PROCESS
```

Beispiel für eine Definition zur Erzeugung einer aggregierten View (Abbildung 37):

```
CREATE AGGREGATED VIEW EXAMPLE_VIEW_2 AS
SELECT SET(J,O) AS V1, SET(D) AS V2
REMAINING AS T ASC
FROM EXAMPLE_PROCESS
```

Es fehlt eine Möglichkeit, für Knoten einer View auch noch Angaben über Attribute und den Datenfluss dieser Knoten festlegen zu können. Dafür muss für jeden Knoten die Möglichkeit bestehen, die Darstellung seiner Attributen und der assoziierten Datenlemente festzulegen.

```
FOR             NodeName IN ViewName | ViewName
SHOW ATTRIBUTE  VirtualAttributeName1 AS AggregateFunction1(AttributeName1)
...
SHOW ATTRIBUTE  VirtualAttributeNamen AS AggregateFunctionn(AttributeNamen)

HIDE ATTRIBUTE  AttributeName1
...
HIDE ATTRIBUTE  AttributeNamen

SHOW DATANODE   VirtualDataNodeName1 FROM (DataNode11 , ..., DataNode1n)
...
SHOW DATANODE   VirtualDataNodeNamen FROM (DataNoden1 , ..., DataNodenn)
CONCEAL INTERNAL DATAFLOW | CONCEAL ENTIRE DATAFLOW
```

Mit Hilfe dieser DDL können jetzt für einen virtuellen Knoten (der durch Graphaggregation entstanden ist) oder für eine ganze View virtuelle Attribute mit Hilfe einer Aggregationsfunktion und den Attributen der Basisaktivitäten Knotens definiert werden oder einzelne Attribute ausgeblendet werden. Für graphreduzierte Views können mit Hilfe von *HIDE ATTRIBUTE* einzelne Attribute ausgeblendet werden. Mehrere Datenflussknotenkönnen mit *SHOW DATANODE* zu einem virtuellen Knoten zusammengefasst werden, oder es kann nur der ein- und ausgehende Datenfluss angezeigt (*CONCEAL INTERNAL DATAFLOW*), oder der Datenfluss komplett verborgen werden (*CONCEAL ENTIRE DATAFLOW*).

Für den Algorithmus der mit Hilfe eines Schwellenwertes die relevanten Knotenmengen automatisch bestimmt, ist es auch möglich eine solche View mit Hilfe einer DDL zu definieren:

```
CREATE (AGGREATED | REDUCED) VIEW ViewName AS
FROM   ProcessName | ViewName
WHERE  threshold
```

Dabei wird mit Hilfe eine Prädikats *threshold* der Schwellenwert festgelegt, der sich auf ein- oder mehrere Attribute oder Rollendefinitionen beziehen kann.

Für das Beispiel aus Abbildung 50 auf Seite 82 würde eine Definition folgendermaßen aussehen:

```
CREATE AGGREGATED VIEW EXAMPLE_VIEW AS
FROM EXAMPLE_PROCESS
WHERE Kosten <= 5000 EUR
```

Wenn eine solche DDL definiert ist, können darauf graphische Tools aufsetzen, die die Benutzereingaben in eine Definition für die DDL umsetzen. Die Definition muss dann von einem Queryprozessor in einen oder mehrere Aufrufe der in den vorherigen Abschnitten vorgestellten Algorithmen umgesetzt werden.

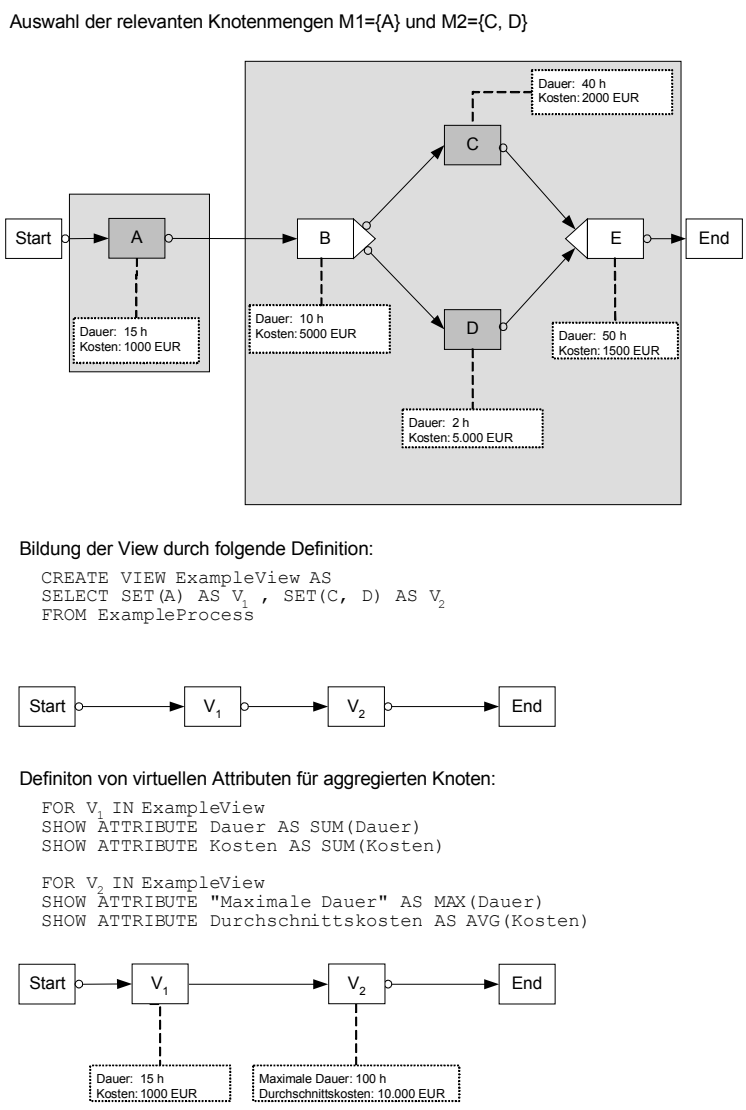

Abbildung 52: Beispiel einer View-Bildung mit Hilfe einer DDL-Definition

Im Beispiel aus Abbildung 52 werden für einen einfachen Prozess erst eine View für zwei relevante Knotenmengen gebildet und anschließend noch Attribute eingeblendet. Da V_1 nur aus einer Basisaktivität besteht, kann hier auf die Angabe einer Aggregationsfunktion verzichtet werden.

Ein weiteres Beispiel für die Aggregation und das Verbergen von Datenelemente wird in Abbildung 53 verdeutlicht. Im Beispiel a) werden die Datenelemente d_1 und d_2 zu dem virtuellen Datenelement D_{vi} zusammengefasst, im Beispiel b) wird für den virtuellen Knoten V_1 der interne Datenfluss entfernt, in diesem Fall ist dass das Datenelement d_3 und d_4.

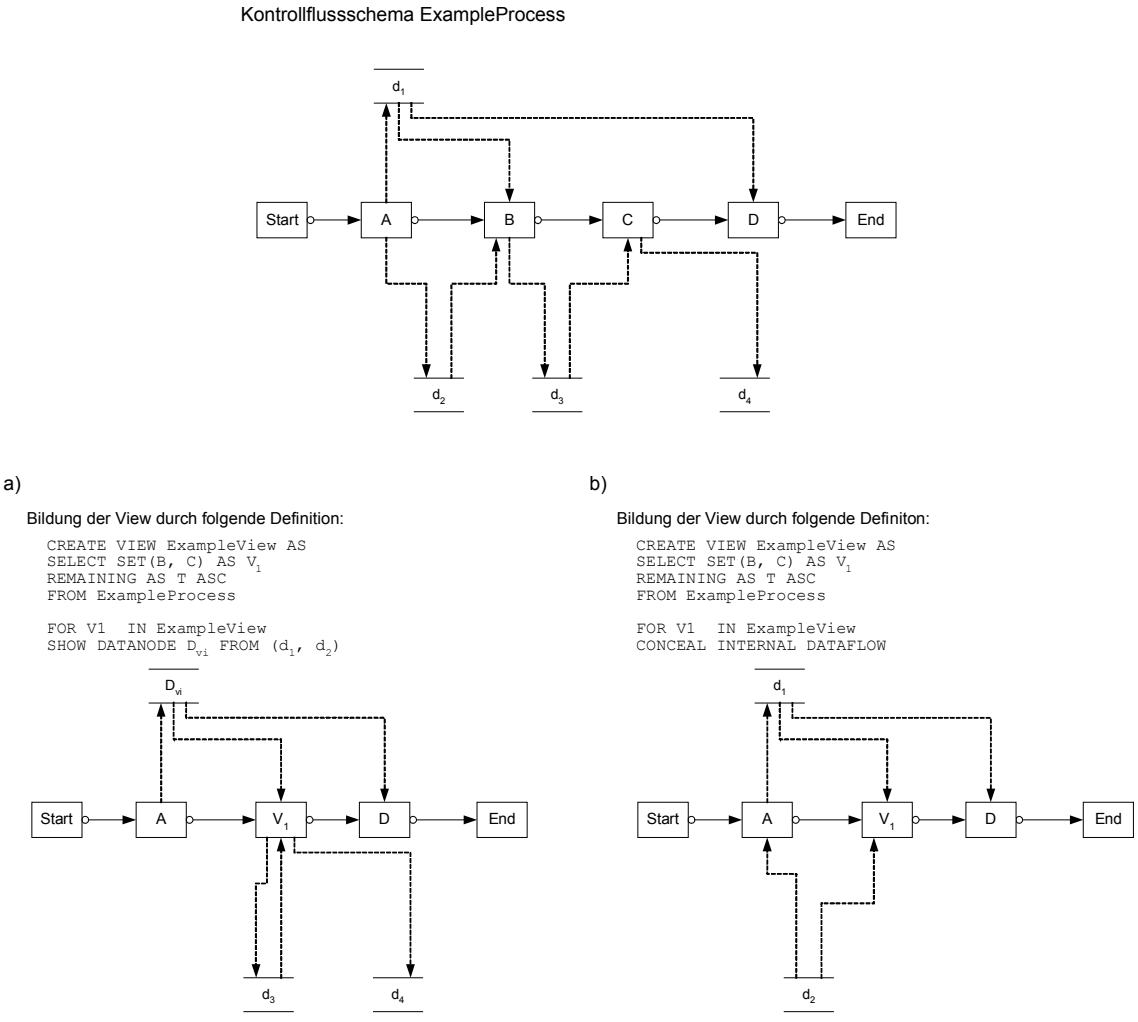

Abbildung 53: Aggregation und Ausblenden von Datenelementen

6 Komponenten zur View-Erzeugung

Im vorherigen Kapitel haben wir gezeigt, wie Views definiert und mit Hilfe von Algorithmen erzeugt werden können. In einer Client-Server Architektur stellt sich die Frage, von welcher Komponente die Views erzeugt und gespeichert werden sollen.

Views können grundsätzlich sowohl im Server als auch auf dem Client, wie in Abbildung 54 verdeutlicht, erzeugt werden. Die Vor- und Nachteile dieser beiden Möglichkeiten sollen im Folgenden aufgezeigt werden.

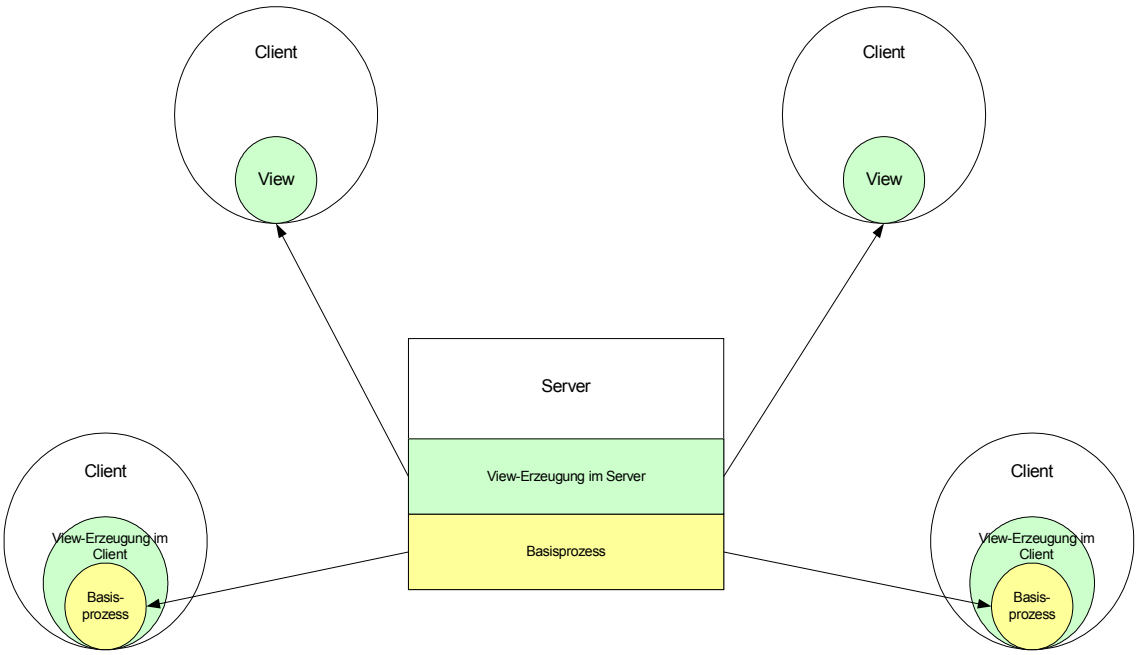

Abbildung 54: View-Erzeugung im Server oder Client

6.1 View-Erzeugung im Server

Views werden bei dieser Variante im Server erzeugt und dort verwaltet. Die Clients erhalten dann, je nach Anwendungsfall, keinen Zugriff auf die Basisprozesse sondern nur Zugriff auf die jeweiligen Views. Vor allem wenn Views aus Sicherheitsgründen gebildet werden, ist die serverseitige Erzeugung notwendig, damit sicherheitsrelevante Informationen nicht nach außen gelangen können.

Die Erzeugung und Verwaltung von Views erfordert die Existenz einer View-Engine im Server, damit effizient auf die Views zugegriffen werden und auch die Aktualisierung von Views und Basisprozesse zeitnah geschehen kann.

Vorteile:

- Es ist weniger Datenkommunikation zwischen Client und Server notwendig. Views können in kompakter Form[18] zu den Clients geschickt werden.
- Die View-Verwaltung erfolgt zentral auf dem Server. Ein Administrator kann sich jederzeit einen Überblick über vorhandene Views verschaffen, und View-Änderungen leichter durchführen.
- View-Updates (z.B. wenn sich der zugehörige Basisprozess geändert hat) lassen sich im Server einfacher und schneller durchführen, da dafür keine Kommunikation mit dem Client notwendig ist.
- Strukturelle Korrektheit von Views kann im Server besser sichergestellt werden, da ein direkter Zugriff auf die zugehörigen Basisprozesse möglich ist, oder die Views sogar direkt mit den Basisprozessen verknüpft sind (siehe hierzu auch Abschnitt 7).
- Kapselung von Prozessen durch Views im Server: Anwendungsprogramme können auf Basis einer View entworfen werden und müssen auch bei Änderung des Basisprozesses im Normalfall nicht geändert werden.
- Das Verbergen von sicherheitsrelevanten Informationen ist deutlich einfacher durchführbar, da Informationen zu Berechtigungen, die üblicherweise über ein User-Management-System zur Verfügung gestellt werden, im Server sowieso vorhanden sein müssen, so dass keine sicherheitsrelevanten Informationen mehr an den Client gesendet werden müssen.
- Views können von einem Prozess- bzw. View-Modellierer vordefiniert und dann im Server erzeugt werden. Solche Views sind auch für Nutzer mit geringer Erfahrung im Umgang mit Prozessen oder View-Definitionen nutzbar.

Nachteile:

- Die View-Erzeugung führt zu einer höheren Rechenlast auf dem Server, vor allem wenn es viele verschiedene Views auf den gleichen Basisprozess gibt. Aufwändig ist auch die Aktualisierung der Views, wenn sich der Basisprozess ändert. Zusätzlich kann die Rechnerlast sich noch weiter unnötig erhöhen, wenn viele Views, die nur für einen kurzen Zeitraum (z.B. Monitoring, Ad-Hoc Änderung) benötigt werden, immer noch im Server vorhanden sind und weiterhin aktualisiert werden.
- Das Problem der Aktualisierung von Views ist auch bei der Erzeugung von Views im Server noch nicht vollständig gelöst, da View-Änderungen den Clients weiterhin bekannt gemacht werden müssen. Dies ist aber kein View-spezifisches Problem, diese Problemstellung tritt auch ohne Verwendung von Views bei geänderten Prozessen auf.

[18] durch physische Entfernung nicht sichtbarer Elemente

- Views können nicht flexibel eingesetzt werden, da sie nur auf dem Server erzeugt werden können und dafür üblicherweise besondere Rechte vorhanden sein müssen. Wenn ein View lediglich erzeugt wird, um sich einen Überblick über einen Prozess zu verschaffen, oder bei einer Änderung vorher nicht relevante Teile auszublenden, so ist der Ansatz der serverseitigen View-Erzeugung nicht besonders geeignet.

6.2 View-Erzeugung im Client

Aus Sicht des Servers gibt es bei dieser Variante den Begriff der View nicht. Der Server arbeitet lediglich mit den Basisprozessen und stellt diese den Clients zur Verfügung. Die Clients sind dann für eine geeignete View-Erzeugung zuständig. Die View-Bildung findet im Client statt. Dieser Ansatz macht aus den oben genannten Gründen nur Sinn, wenn die View nicht zum Verbergen sicherheitsrelevanter Informationen generiert wird.

Vorteile:

- Der Server muss sich nicht zusätzlich um die rechenintensive Verwaltung und Aktualisierung von Views kümmern. Es wird keine spezielle View-Engine benötigt, und der Rechenaufwand für die Erzeugung der Views verteilt sich auf die verschiedenen Clients.
- Die Arten von Views, die erzeugt werden können, sind nicht von den Fähigkeiten des Servers abhängig. Auch für Server, die keine Möglichkeit zur View-Bildung vorsehen, können mit Hilfe spezieller Clients (mit der Fähigkeit zur View-Bildung), Views auf Prozesse für verschiedene Anwendungsfälle (z.B. Prozessmonitoring) genutzt werden.
- Clients sind für die Viewbildung wesentlich flexibler einsetzbar als ein Server, da kein besonderer Zugang für den Server und die nötigen Berechtigungen vorhanden sein müssen. Besonders beim Monitoring oder bei Ad-Hoc-Änderungen ist eine lokale View-Erzeugung deutlich schneller durchführbar, da die View nicht erst im Server erzeugt, und dann vom Client neu geladen werden muss.

Nachteile:

- Durch die Übertragung der gesamten Basisprozesse mit möglicherweise vielen Attributen wird eine höhere Datenkommunikation verursacht.
- Wenn die gleiche View von mehreren Clients benötigt wird, muss sie von jedem Client neu berechnet werden
- Für die View-Erzeugung und Verwaltung müssen die Clients deutlich komplexere Aufgaben durchführen und brauchen damit deutlich mehr Speicher und Rechenleistung.
- Es entsteht eine Entkopplung von Views und zugehörigen Basisprozessen. Falls der Basisprozess geändert wird, und dies dem Client nicht schnell genug bekannt gemacht

wird, können ungültige[19] Views entstehen. Dann treten Fehler auf, wenn versucht wird über die ungültigen Views Änderungen auf den Basisprozessen an der Prozess-Engine zu propagieren. Je nach Anwendungsfall sind sogar Sperrmechanismen für die Änderung von Prozessen notwendig, die auch die korrekte Aktualisierung von Views sicherstellen.

- Die Wartung von Views ist durch die Verteilung auf viele Clients deutlich aufwändiger.

6.3 Mischformen

Je nach Anwendungsfall ist die View-Bildung auf dem Server oder Client zu bevorzugen. Deshalb liegt es nahe, sofern die Systemarchitektur es zulässt, einen gemischten Ansatz zu wählen um Views sowohl im Server als auch im Client erzeugen zu können. Für welchen Anwendungsfall eine Erzeugung im Server oder im Client besser ist, hängt vor allem von der notwendigen räumlichen und zeitlichen Nähe für die View mit dem zugehörigen Basisprozess ab. Eine Erzeugung im Server ist sinnvoll für:

- das Verbergen von Informationen aus Sicherheitsgründen
- die Realisierung interorganisationellen Workflows
- vereinfachte Darstellung von Prozessen für verschiedene Bearbeiter
- die Kapselung von Prozessen

Und für folgende Anwendungsfälle ist die Erzeugung im Client besser:

- Monitoring und Überwachung
- Ad-Hoc Änderungen

Viele dieser Anwendungsfälle können auch gleichzeitig auftreten, so dass hier eine kombinierte View-Erzeugung sinnvoll ist.

[19] In dem Sinne, dass sie keine Sicht mehr auf den Basisprozess darstellen, weil sie z.B. Strukturen enthalten, die im Basisprozess nach einer Änderung gar nicht mehr vorhanden sind.

Vorteile:

- Die Vorteile der obigen Ansätze werden vereint. Views sind einerseits sehr flexibel einsetzbar und andererseits können auch Sicherheitsaspekte beachtet werden.
- Durch Kombination der View-Erzeugung im Server und im Client ergeben sich noch zusätzliche Möglichkeiten. So kann zum Beispiel im Server eine View erzeugt werden, die gewisse Informationen ausblendet (Attribute oder Teilgraphen). Der Benutzer des Client vereinfacht dann diese View noch weiter, um einen besseren Überblick zu haben.

Nachteile:

- Je nach Anwendungsfall resultiert durch die Kombination der beiden Ansätze ein erhöhter Speicher- und Rechenbedarf je nach Anwendungszweck sowohl auf dem Server als auch dem Client. Zudem benötigen sowohl der Server als auch der Client Mechanismen zur View-Erzeugung und Verwaltung.
- Die Wartung von Views ist noch aufwändiger als bei der reinen Client-Lösung. Views sind sowohl im Server vorhanden als auch auf den Clients verteilt, und können sogar noch voneinander abhängen.
- Durch die Verteilung der Views und der möglichen View-auf-View-Bildung müssen jetzt im Server und den Clients auch aufwändigere Synchronisationsmechanismen vorhanden sein, um die Existenz korrekter Views zu gewährleisten.

Ein pauschale Aussage, wo Views am besten erzeugt werden sollten, lässt sich nicht treffen. Welche Komponente dafür zum Einsatz kommt, hängt von Anwendungsfall, vorhandener Hardware und Anzahl und Komplexität der vorhandenen Prozesse ab.

7 Speicherung von Views

In Kapitel 5 wurde beschrieben, wie Views durch Verwendung der zwei Attribute VISIBLE und VIRTUAL definiert und mit Hilfe verschiedener Algorithmen erzeugt werden können. Dabei handelt es sich lediglich um ein logisches Konzept, es wurde noch nicht geklärt, wie solche Views dann physisch gespeichert werden.
Für die Speicherung und Verwaltung von Views gibt es verschiedene Möglichkeiten:

1. Eine View wird als Sequenz von Anweisungen (z.B. durch die unter Kapitel 5 beschriebenen Algorithmen) zur Erzeugung der View gespeichert. Die berechneten Views werden lediglich im Hauptspeicher gehalten. Bei jeder Veränderung der Basisinstanz oder des Basisschemas muss die View neu berechnet werden. Vorteilhaft ist die einfachere Verwaltung von Views, es müssen lediglich die jeweiligen Anweisungen gespeichert und verwaltet werden. Nachteilig ist der große Rechenaufwand, vor allem wenn es viele verschiedene Views auf einen Basisprozess gibt oder auch wenn Views-auf-Views gebildet werden.
2. Die View wird einmal berechnet und dann als Kopie zum Prozess gehalten (*Materialized View*). Bei Änderungen des Basisprozesses müssen die Kopien entsprechend aktualisiert werden, wobei hier auch verschiedene Aktualisierungsstrategien zum Einsatz kommen können (View wird erst dann aktualisiert wenn auf sie wieder zugegriffen wird oder in bestimmten Zeitabständen)
   Die Speicherung kann auf verschiedene Arten erfolgen:

   - **Speicherung der Views als Kopien des Basisgraphen:** Der Basisgraph wird kopiert und anschließend auf der Kopie die Viewbildung durchgeführt. Dabei können je nach Anwendungsfall, die verborgenen Teile des Graphen auch physisch gelöscht werden (kompaktifizierte Views). Dies ist z.B. notwendig, wenn aus Sicherheitsgründen Prozessinternas verborgen werden sollen und nur kompaktifizierte Views den Clients zur Verfügung gestellt werden. Nachteil kompaktifizierter Views ist die Schwierigkeit, anschließende Änderungen des Basisprozesses auf die View zu propagieren, da durch das Löschen von verborgenen Schritten die Verbindung zum Basisprozess verloren geht und bei einer Änderung Teile des Basisprozesses erst wieder in die View kopiert werden müssen.
     Vorteil dieser Art der Speicherung ist die Unabhängigkeit der Views vom Basisprozess. Es ist je nach Anwendungsfall keine direkte Verbindung mit dem Basisprozess notwendig (z.B. wenn man die View lediglich verwendet, um sich einen Überblick über einen komplexen Prozess zu verschaffen).
     Nachteilig ist der hohe Speicherbedarf, da für jeden View eine Kopie des Basisgraphen gespeichert wird; und bei View-auf-View Bildung kann die

Aktualisierung einer View sehr lange dauern, da zuerst die zugrundeliegenden (kompaktifizierten) Views geändert werden müssen.
Abbildung 55 zeigt ein Beispiel für zwei verschiedene Views die auf demselben Basisschema P definiert sind. Die Views werden jeweils als Kopie zum eigentlichen Prozess gespeichert, und die bei der Viewbildung neu erzeugten Kanten und Knoten (wie der neu erzeugte Knoten V_1 in der View 1) werden nur in den Kopien hinzugefügt.

- **Speicherung als Attribut am Basisgraphen:** Die Viewbildung wird direkt am Basisgraphen durchgeführt, eine View ist dann durch die Elemente, deren VISIBLE Attribut auf TRUE gesetzt ist, definiert. Diese Art der Speicherung kann noch erweitert werden, indem für jede View eigene VISIBLE und VIRTUAL Attribute verwendet werden, und somit auch mehrere Views für einen Basisgraph gebildet werden können. Vorteil dieser Art der Speicherung ist der geringe Speicherbedarf, da pro Knoten lediglich einige zusätzliche Attribute gespeichert werden. Durch die direkte Speicherung am Basisgraph ist auch immer eine direkte Verknüpfung der View mit dem Basisschema gegeben, was View-Updates erleichtert. So ist es z.B. möglich, wenn eine View V_1 auf Basis einer View V_2 gebildet wird, bei einer notwendigen Aktualisierung von V_1 dies direkt durch die Verbindung mit dem Basisprozess zu machen, ohne dass V_2 vorher aktualisiert werden muss. Ein Nachteil ist die Aufblähung des Basisgraphen, die bei häufigen Zugriffen und Operationen auf den Basisprozess zu Perfomanzeinbußen führen kann.
  In Abbildung 56 zeigt für das Basisschema aus Abbildung 55 die selben gebildet. Hier werden diese jedoch nicht als separate Kopien des Basisprozesses, sondern nur durch unterschiedliche Attribute direkt an dem Basisprozess gespeichert. Dies hat zur Folge, dass bei der Viewbildung neu erzeugte Knoten und Kanten auch in das Basisschema eingefügt werden, und dann die Attribute für diese neuen Elemente entsprechend gesetzt werden müssen.
- Denkbar sind auch Mischformen dieser beiden Ansätze, bei denen z.B. Teile der View als Kopie gehalten und andere Teile durch Attribute des Basisprozesses repräsentiert werden.

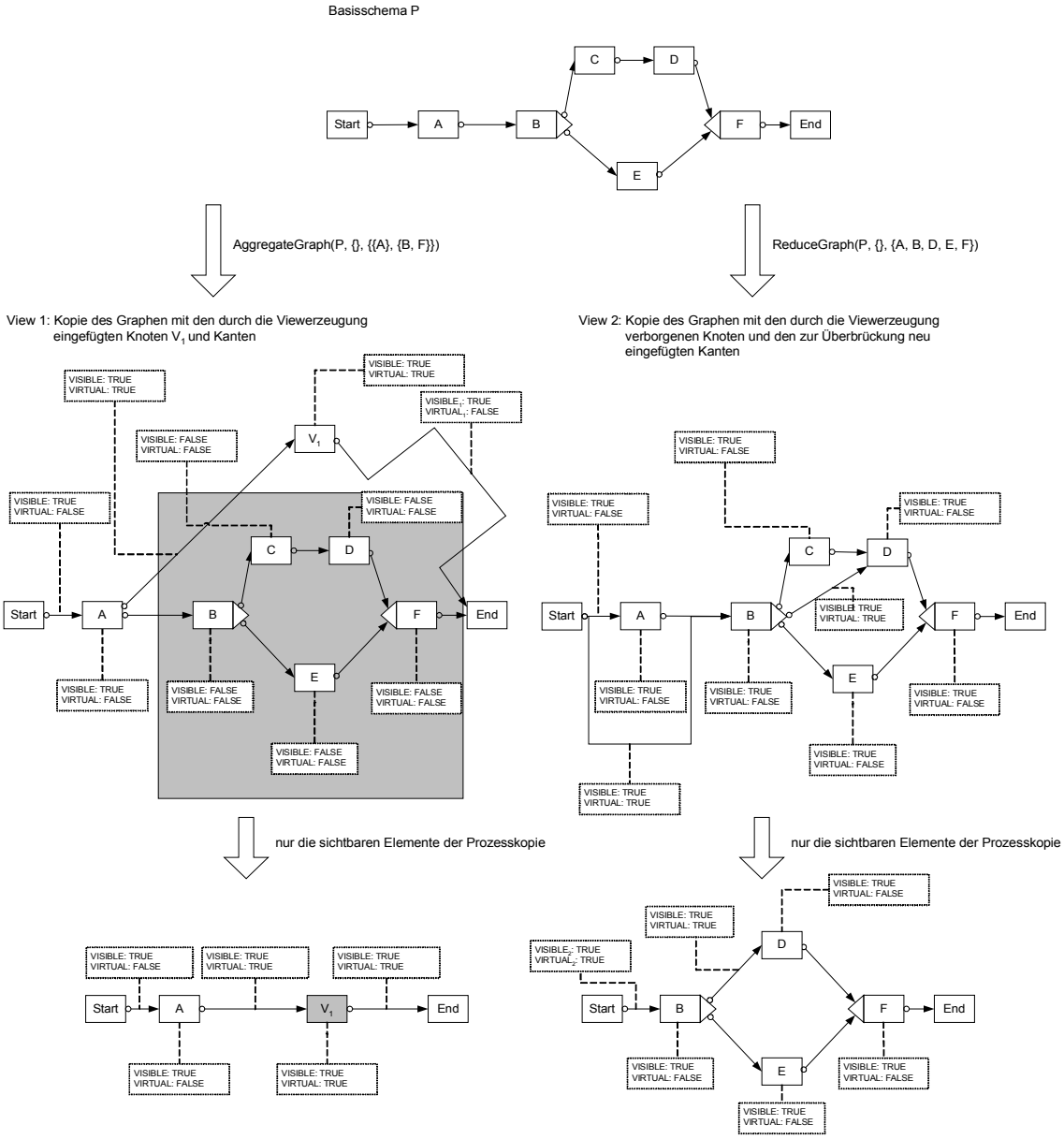

Abbildung 55: View-Speicherung als Kopie

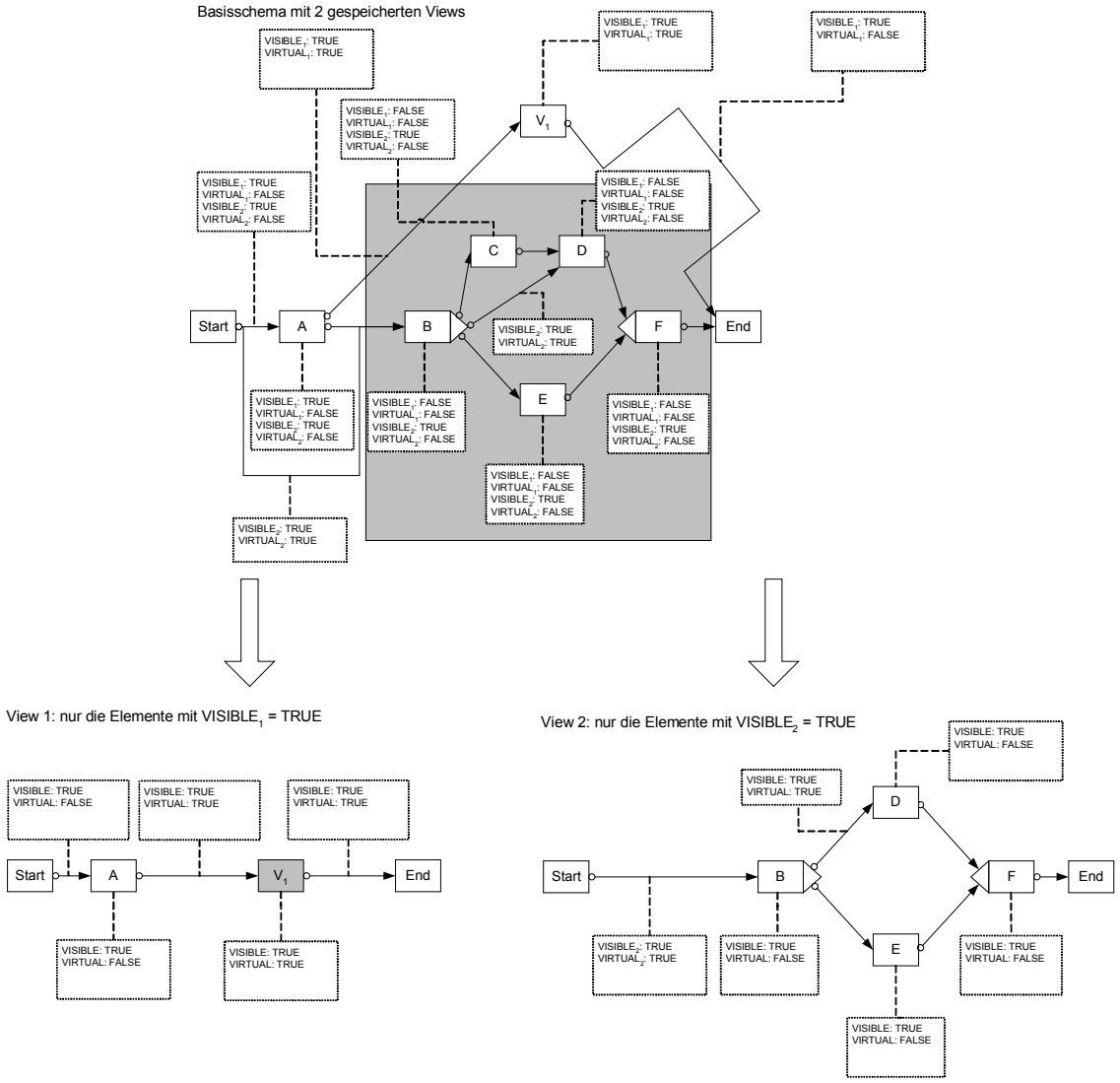

Abbildung 56: Speicherung von zwei Views im Basisschema

8 Aktualisierung von Views

Bei materialisierten Views stellt sich, ebenso wie bei Views in Datenbanken ([14, 15, 16]), die Frage, wie bei Änderungen des Basisprozesses die Views möglichst effizient aktualisiert werden können. Dabei muss bei Prozessen zwischen Änderungen des Schemas und Änderungen des Prozesszustands (d.h. Weiterschalten im Prozess) unterschieden werden.

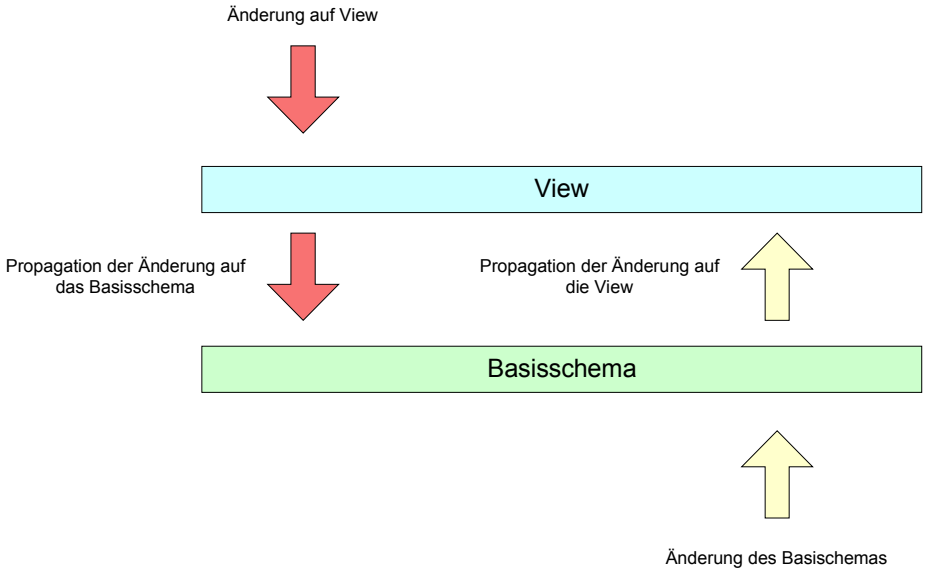

Abbildung 57: Aktualisierung von Views

Wichtig ist die Unterscheidung in welche Richtung die Änderungen propagiert werden sollen. Änderungen können entweder für das Basisschema erfolgen, und müssen dann auf die Views propagiert werden, oder es werden Änderungen über eine View auf das Basisschema (und anschließend auf alle anderen für dieses Schema existierenden Views) propagiert.

In den folgenden Abschnitten wird auf die Änderungsoperationen des ADEPT-Metamodells Bezug genommen. Ein detaillierte Beschreibung dieser Operationen erfolgt in [4], an dieser Stelle erfolgt lediglich eine informelle Zusammenfassung der wichtigsten Basisoperationen:

- **deleteActivity:** Mit dieser Operation wird eine Aktivität im Schema entweder durch eine NULL-Aktivität ersetzt, oder zusammen mit den assoziierten Kanten gelöscht. Im zweiten genannten Fall wird dann eine neue Kante zwischen den Vorgänger(n) und Nachfolger(n) des gelöschten Knotens eingefügt.
- **serialInsert:** Hiermit kann eine neue Aktivität zwischen zwei direkt aufeinander folgenden Aktivitätenknoten eingefügt werden.
- **parallelInsert:** Mit parallelInsert kann ein Knoten parallel zu einem bestehenden Kontrollblock (auch einzelne Aktivität) eingefügt werden. Es wird vor dem Startknoten

des Kontrollblocks ein AND-Split- und nach dessen Endknoten ein AND-Joinknoten eingefügt. Der neue Knoten wird dann als paralleler Verzweigungspfad zum bestehenden Kontrollblock eingefügt.

- **branchInsert:** Mit branchInsert kann in einem alternativen Verzweigungsblock ein Knoten als neuer Teilpfad eingefügt werden.
- **moveActivity:** Damit kann eine Aktivität im Schema von einer Position an eine andere verschoben werden.

8.1 Änderung von Prozesszustand und -attributen

Zustandsänderungen lassen sich einfach auf vorhandene Views propagieren: Wenn eine View durch Verbergen von Graphteilen entstanden ist, müssen lediglich die Knoten aktualisiert werden, deren Zustand sich geändert hat und die in der View sichtbar sind. Der Zustand des Knotens in der View wird dabei auf den Zustand der Basisaktivität gesetzt. Bei Views, die durch Graphaggregation (siehe Seite 63) entstanden sind, muss der virtuelle Knoten bestimmt werden, der die Aktivitäten, deren Zustand sich geändert hat einschließt und dann entsprechend den Regel aus Abschnitt 3.1 der Zustand des virtuellen Knotens angepasst werden.

Bei Änderungen des Prozesszustands kann sich nicht nur der Zustand von Aktivitäten ändern, sondern auch der Wert von Attributen (z.B. wenn eine Aktivität den Wert eines Datenknotens ändert). Dabei muss für einen im Basisprozess geänderten Attributwert für die View diejenigen virtuellen Attribute neu bestimmt werden, die das geänderte Attribut durch Aggregation enthalten. Anschließend muss der Wert der virtuellen Attribute mit Hilfe der für das jeweilige Attribut gültigen Aggregations- und Transformationsfunktion neu bestimmt werden.

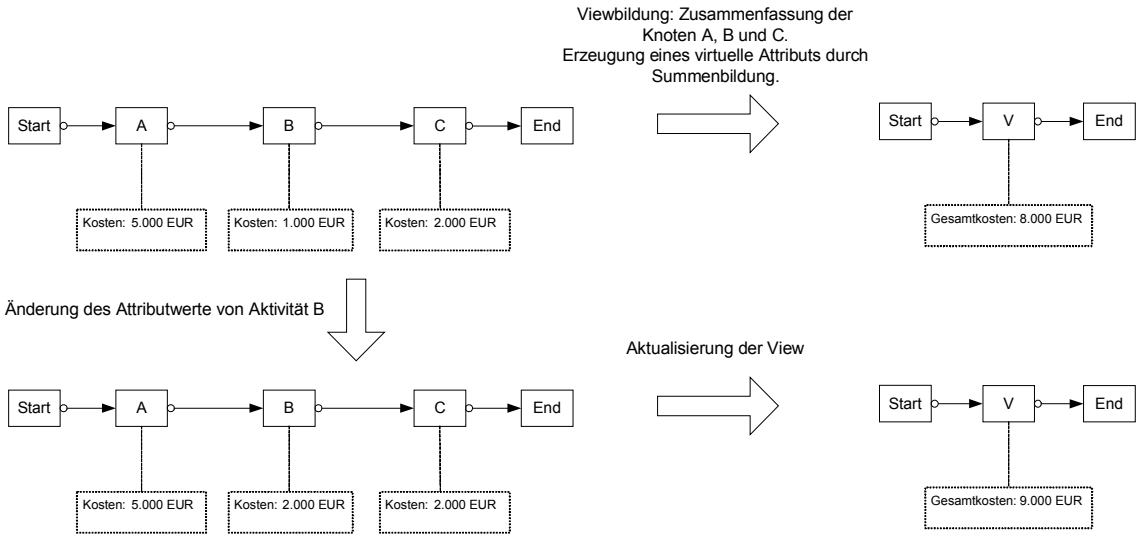

Abbildung 58: Aktualisierung eines virtuellen Attributs

In Abbildung 58 wird die Aktualisierung eines Attributs an einem Beispiel verdeutlicht. Für die drei Aktivitäten A,B und C ist jeweils das Attribut „Kosten“ definiert. Diese drei Knoten werden zu einem virtuellen Knoten zusammengefasst und für diesen Knoten das virtuelle Attribut „Gesamtkosten“ definiert. Als Aggregationsfunktion wird die Summenbildung verwendet. Nach einer Änderung eines Attributwerts im Schema (Kosten von Aktivität B werden erhöht) wird das virtuelle Attribut des Knotens V entsprechend angepasst.

8.2 Änderung des Schemas

Wie erwähnt, ist eine Prozess- bzw. Schemänderung über Views nur sinnvoll, wenn die jeweilige View durch Graphaggregation gebildet wurde.

Umgekehrt kann auch für Views, die durch Graphaggregation oder Graphreduktion entstanden sind, das Schema des zugrunde liegenden Basisprozesses geändert, und deshalb ein Update der (materialisierten) View (ggf. auch eine Anpassung der View-Definition) notwendig werden. Wenn die Views als Kopien (ohne Löschung verborgener Elemente) gehalten werden, können sämtliche Änderungsoperationen direkt auf den Views, genauso wie auf den Basisprozessen, durchgeführt werden. Grund ist, dass alle Elemente des Basisprozesses auch in der View vorhanden sind. Falls die Views durch Attribute des Basisprozesses repräsentiert werden, entfällt dieser Schritt, da hier nur der Basisgraph vorhanden ist.

Die Aktualisierung von Views infolge eines geänderten Basisschemas kann, je nach Bereich und Umfang der Änderungen, sehr aufwändig sein. Hauptgrund ist, dass nach jeder Änderung des Basisschemas auch die zugehörigen Views aktualisiert werden müssen. Deshalb muss stets abgewogen werden, ob es nicht günstiger ist, nach einer Änderung die materialisierte View zurückzusetzen und neu aufzubauen. Insbesondere wenn viele Änderungen in Folge durchgeführt werden, kann durch einen kompletten Neuaufbau der (materialisierten) View eine verzögerte Aktualisierung deutlich effizienter erfolgen, als wenn bei jeder einzelnen Änderung auch die View aktualisiert wird. Die durch mehrere Aktualisierungen notwendigen Operationen können in der Summe aufwändiger sein wie die Operationen zum Neuaufbau einer View. Deshalb ist es sinnvoll, solange es der konkrete Anwendungsfall[20] der View zulässt, erst nach mehreren Änderungen des Basisschema eine Aktualisierung der View durch einen Neuaufbau durchzuführen.

Bei der Aktualisierung müssen Views, die durch Graphreduktion oder Graphaggregation entstanden sind, unterschieden werden.

8.2.1 Aktualisierung graphreduzierter Views

Bei der Aktualisierung (infolge von Änderungen des Basisschemas) einer graphreduzierten View, ist das Problem, dass die View kein gültiger Prozessgraph im Sinne des Basismodells mehr sein muss und deshalb auf dieser View auch keine Operationen des Basismodells

[20] Wenn es im Anwendungsfall Views nicht immer sofort aktualisiert werden müssen.

ausgeführt werden können. Es muss beachtet werden, dass in der View virtuelle Kanten zwischen Knoten auftreten, zwischen denen es im Basisprozess keine Verbindung gibt. Als Beispiel hierfür diene Abbildung 36 auf Seite 62, wo z.B. in der View zwischen den Knoten E und H eine Kante vorhanden ist, die es im Basisschema nicht gibt).
Zwecks bessere Strukturierung des Problems unterscheiden wir zwischen folgenden drei Operationen auf dem Basisschema (und ihrer Propagierung auf Views):

- **Löschen von Aktivitäten**
  Werden im Basisschema keine in der View sichtbaren Knoten gelöscht, ist nichts zu tun. Bei einer Löschung sichtbarer Aktivitäten werden Knoten und Kanten physikalisch gelöscht, und nicht wie bei der Viewbildung nur als nicht sichtbar markiert. Als Ergebnis einer Löschung von Knoten ergeben sich virtuelle Kanten, die als Start- oder Endknoten den oder die gelöschten Knoten haben. Es muss unterschieden werden, ob im Basisprozess lediglich ein einzelner Knoten oder ein ganzer Kontrollblock gelöscht wurde.
  Wenn im Basisschema ein einzelner Knoten gelöscht wird, werden bei der Löschoperation die assoziierten Kanten des Knotens mit entfernt. Für die Aktualisierung der View müssen diese Kanten jedoch noch zur Verfügung stehen, damit die Vorgänger und Nachfolger des gelöschten Knotens identifiziert werden können. Dies kann z.B. dadurch geschehen, dass die Kanten nicht physikalisch entfernt werden, sondern nur mittels eines Attributs als gelöscht markiert werden. Eine andere Möglichkeit ist, dass vor dem Löschen (virtuelle) Kopien der Kanten für die zu aktualisierenden Views erzeugt werden. Im Folgenden wird davon ausgegangen, dass die Kanten auch nach dem Löschen einer Aktivität noch in der View zur Verfügung stehen. Bei Löschung eines einzelnen Knotens sind in der View jetzt Kanten enthalten, die keinen Vorgänger oder keinen Nachfolger mehr haben. Hier können jetzt einfach die verbliebenen Kanten[21] „kurzgeschlossen“ werden. Mit dem Begriff „kurzschließen“ sind hierbei das physikalische Löschen der assoziierten Kanten des gelöschten Knotens sowie das Ersetzen durch eine virtuelle Kante zwischen Vorgängern und Nachfolgern des gelöschten Knoten gemeint. Durch die Konstruktion der Views[22] ist sichergestellt, dass es entweder nur eine ausgehende oder nur eine eingehende virtuelle Kante in den gelöschten Knoten gibt, und somit das Kurzschließen problemlos möglich ist.
  Im Beispiel aus Abbildung 59 wird im Basisschema P der Knoten B gelöscht. Da dieser Knoten auch in der für Schema P definierten View sichtbar ist, muss die View nach der Änderung entsprechend angepasst werden. Nach Löschung sind in der View zwei ungültige Kanten vorhanden: Die Kante zwischen A und B wurde in der View direkt aus

[21] Diese haben den nicht mehr vorhanden Knoten als Quell- oder Zielaktivität.
[22] Ein Knoten ist entweder einer normaler Knoten oder ein Split- oder Join-Knoten, jedoch kein Split- und Join-Knoten gleichzeitig.

dem Basisschema übernommen. Hinzu kommt und die virtuelle Kante zwischen C und End, die im Basisschema nicht vorhanden ist, und die bei der View-Bildung erzeugt wurde. Für die Aktualisierung der View müssen diese Kanten jetzt gelöscht und durch eine neue virtuelle Kanten zwischen A und End ersetzt werden.

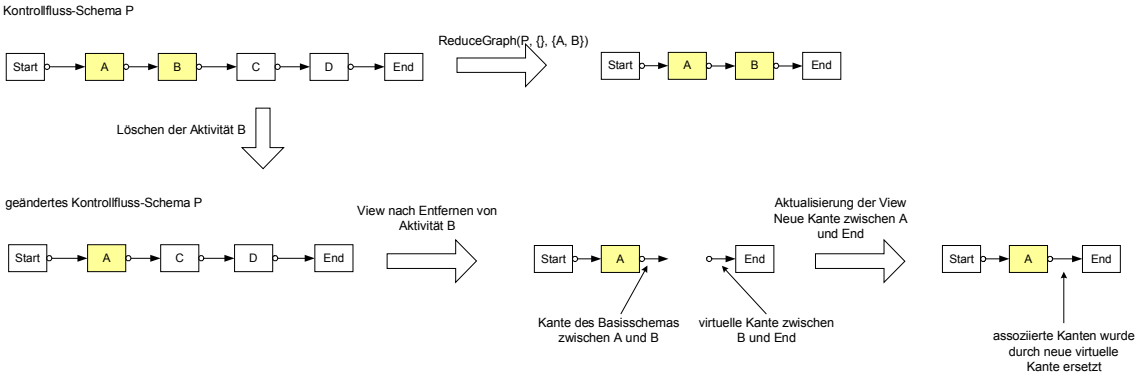

Abbildung 59: Aktualisierung nach Löschen einer Aktivität

Wird im Basisprozess ein ganzer Kontrollblock gelöscht, gibt es anschließend in der View mehrere Kanten, die (wie beim Löschen einer einzelnen Aktivität) entweder keinen Quell- oder Zielknoten mehr haben. Zusätzlich kann es „freistehende“ Kanten geben, die weder einen Quell- noch eine Zielknoten haben. Freistehende Kanten können ignoriert und (physikalisch) gelöscht werden; die dann verbleibenden Kanten werden wie bei der Löschung eines einzelnen Knotens behandelt und „kurzgeschlossen“.
Die Aktualisierung einer View im Anschluss an das Löschen eines Kontrollblocks wird in Abbildung 60 verdeutlicht. Der Kontrollblock zwischen den Knoten B und D wird im Basisschema gelöscht. Dadurch entsteht in der View eine freistehende Kante, die ursprünglich die zwei gelöschten Knoten B und D verbunden hat. Zur Aktualisierung der View wird erst die freistehende Kante gelöscht und anschließend die verbleibenden Kanten kurzgeschlossen.

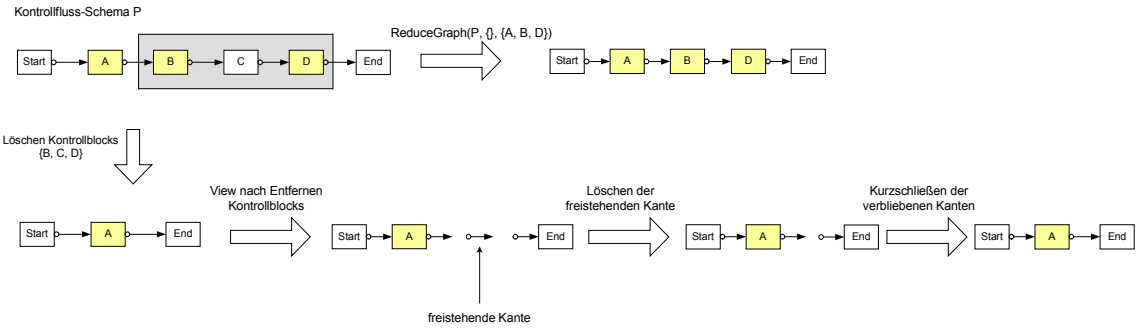

Abbildung 60: Aktualisierung nach Löschung eines Kontrollblocks

In beiden Fällen kann es vorkommen, dass durch das Löschen und der anschließenden Aktualisierung der View ein Zweig eines Verzweigungsblock entfernt wird. Wenn die überbrückte Kante also infolge der Aktualisierung direkt von einem Split- zu einem Join-Knoten geht, kann diese Kante auch weggelassen werden. Wenn die Split- und Join-Knoten jetzt nur noch einen sichtbaren Pfad haben können sie, falls nicht in der Menge der relevanten Knoten (siehe Abschnitt 5.3.4) enthalten, ebenfalls entfernt werden. Im Beispiel aus Abbildung 61 wird aus einem Basisschema der Knoten E innerhalb eines Verzweigungsblocks gelöscht. Dies hat zu Folge, dass in der View die Kanten zwischen A und End kurzgeschlossen werden. Da A und End in der View jeweils Verzweigungsknoten sind, kann diese Kante wieder gelöscht werden.

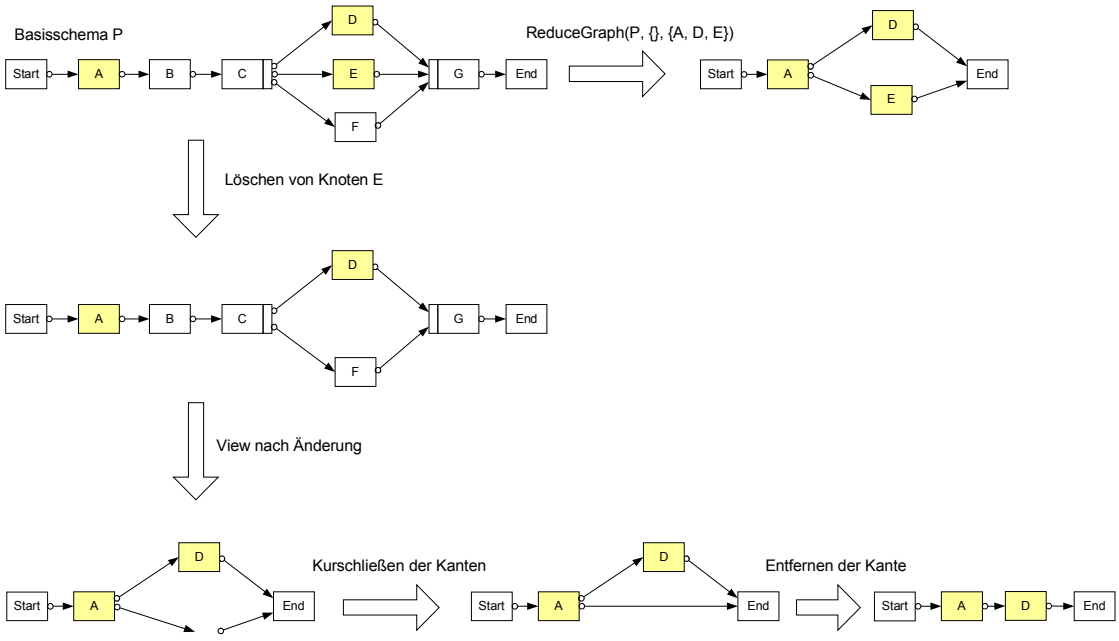

Abbildung 61: Aktualisierung nach Löschen einer Aktivität

- **Einfügen von Aktivitäten**
  Beim Einfügen von Aktivitäten stellt sich die Frage, ob die neu eingefügten Knoten für die View relevant sind oder nicht. Dazu ist es wichtig zu wissen, nach welchen Kriterien die View ursprünglich erzeugt wurde. Wenn eine View mit Hilfe eines Relevanzgrades, wie in Kapitel 5.3.7 beschrieben, erzeugt worden ist und die Informationen über den Schwellenwert und die Attribute für die Relevanzgradbildung noch bekannt sind, kann sehr leicht entschieden werden, ob ein Schritt auch in der View sichtbar sein soll oder nicht. Auch wenn eine View aufgrund eines bestimmten Attributwertes (z.B. bestimmte Rolle als Bearbeiter) gebildet wird, kann die Sichtbarkeit neuer Knoten einfach bestimmt werden. Ansonsten muss der Benutzer entscheiden, ob der im Basisschema eingefügte

Knoten auch in der View sichtbar sein soll. In Abbildung 62 wird dies an einem Beispiel verdeutlicht. Eine graphreduzierte View wird für einen Bearbeiter, dem die „Rolle 1“ zugeordnet ist, gebildet. Es werden alle der „Rolle 1“ zugeordneten Aktivitäten in der View angezeigt. Da die neu eingefügte Aktivität X auch der „Rolle 1“ zugeordnet ist, ist diese für die View relevant, und sollte deshalb durch eine Aktualisierung der View in der View-Darstellung sichtbar werden.

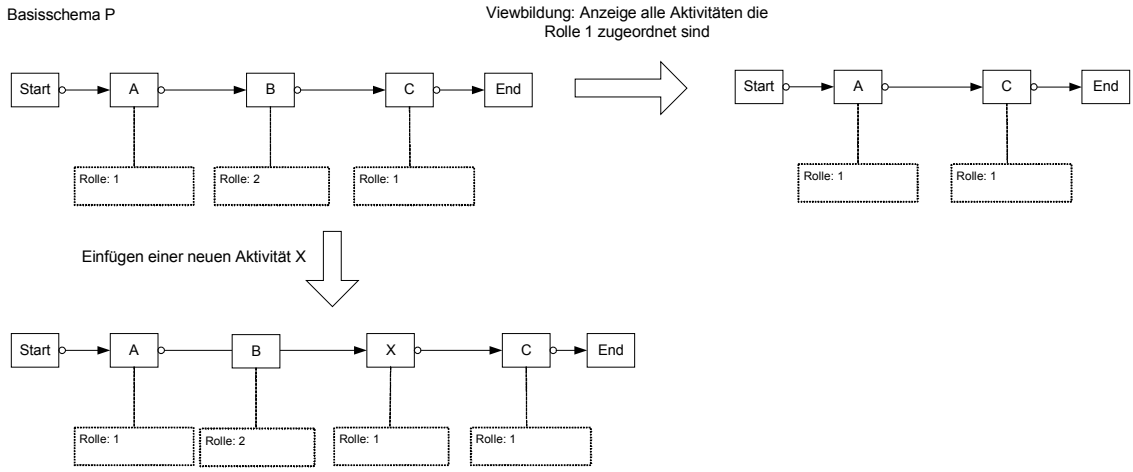

Abbildung 62: Relevanz eingefügter Knoten für eine View

Wird ein einzelner Knoten, der auch in der View sichtbar sein soll, seriell eingefügt, müssen zuerst die Knoten ermittelt werden, die in der View die direkten Vorgänger und Nachfolger des neu eingefügten Knotens sind. Wird die View mit Hilfe des Graphreduktionsalgorithmus (siehe Abschnitt 5.3.4) erzeugt, ist in jedem Fall sichergestellt, dass es einen eindeutigen sichtbaren Vorgänger oder Nachfolger gibt[23], da bei der View-Bildung keine Split- und Join-Knoten in der View miteinander vereinigt werden. Ein neu eingefügter Knoten hat mehrere sichtbare Nachfolger, wenn der direkt sichtbare Vorgänger in der View ein Split-Knoten ist. Entsprechendes gilt, wenn der direkt sichtbare Nachfolger ein Join-Knoten ist. In Abbildung 63 werden Beispiele für mehrere sichtbare Vorgänger oder Nachfolger verdeutlicht. Der Knoten X wird jeweils eingefügt, die grau markierten Knoten sind auch in der View sichtbar. In Beispiel a) ist Knoten A der Splitknoten in der View, und Knoten X hat die sichtbaren Nachfolger C und D. In Beispiel b) ist E der Join-Knoten in der View und X hat die sichtbaren Vorgänger C und D.

[23] Es kann auch sowohl einen eindeutigen Vorgänger als auch einen eindeutigen Nachfolger geben.

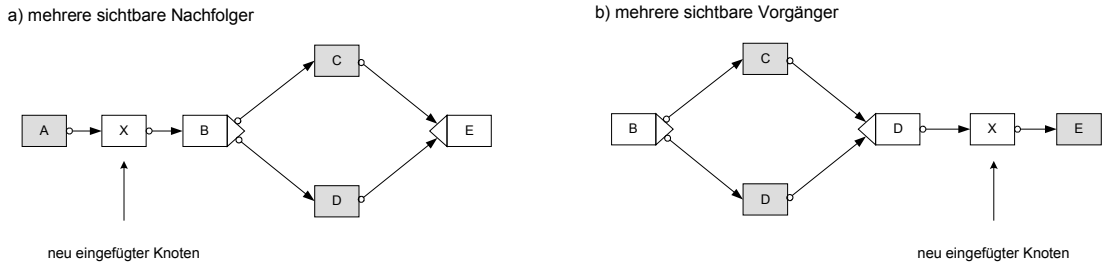

Abbildung 63: mehrere sichtbare Vorgänger und Nachfolger

Für einen neu eingefügten Knoten kann der eindeutige, sichtbare Vorgänger folgendermaßen bestimmt werden: Es wird von dem neu eingefügten Knoten solange der direkte Vorgänger betrachtet, bis entweder direkt der sichtbare Vorgänger gefunden wird, oder man auf einen Join-Knoten[24] trifft. Für diesen Fall gibt es drei Möglichkeiten:

- Innerhalb des durch den Join-Knoten (und zugehörigem Split-Knoten) induzierten Verzweigungsblocks gibt es keine sichtbaren Knoten. Daraus folgt, dass der komplette Verzweigungsblock in der View ausgeblendet wird. Dadurch kann jetzt von dem zum Join-Knoten zugehörigen Split-Knoten aus weiter der eindeutige sichtbare Vorgänger gesucht werden. Im Beispiel a) aus Abbildung 64 ist im Verzweigungsblock zwischen B und D kein sichtbarer (grau markierter) Knoten vorhanden. Deshalb wird die Suche bei Aktivität B fortgesetzt, der eindeutige sichtbare Vorgänger des neuen Knotens X ist A.
- Es gibt genau einen sichtbaren Knoten innerhalb des durch den Join-Knoten induzierten Verzweigungsblocks. Dieser Knoten ist dann der direkt sichtbare Vorgänger, da in der View die Verzweigungsknoten und alle anderen Knoten in den verschiedenen Verzweigungspfaden ausgeblendet wurden. Im Beispiel b) aus Abbildung 64 ist D der einzig sichtbare Knoten in dem Verzweigungsblock, und dies ist auch der eindeutige Vorgänger von X.
- Es gibt mehr als einen sichtbaren Vorgänger im Verzweigungsblock. Damit kann es keinen eindeutigen direkten Vorgänger geben, es muss nach einem eindeutigen direkten Nachfolger gesucht werden. Im Beispiel c) aus Abbildung 64 etwa sind zwei Knoten (C und D) im Verzweigungsblock sichtbar. Deshalb gibt es keinen eindeutigen sichtbaren Vorgänger, es muss ein eindeutiger sichtbarer Nachfolger (hier Knoten E) gesucht werden.

Analog kann nach einem eindeutig sichtbaren Nachfolger gesucht werden, nur müssen hier jeweils die durch Split-Knoten (statt der durch Join-Knoten), induzierten

[24] Nur bei Join-Knoten kann es mehr als einen direkten Vorgänger geben.

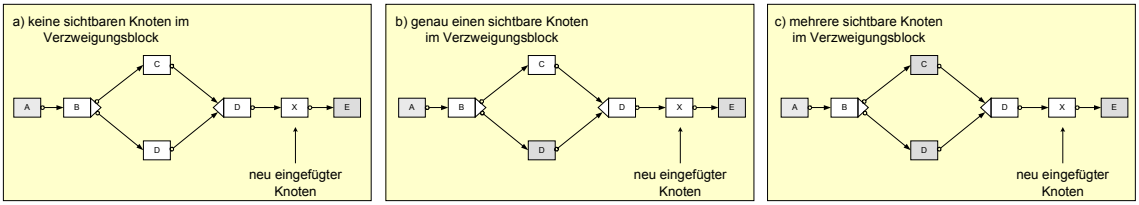

Abbildung 64: Verschiedene Variante für sichtbare Vorgänger in Verzweigungsblöcken

Verzweigungsblöcke, betrachtet werden. Wurde ein eindeutiger sichtbarer Vorgänger (bzw. Nachfolger) gefunden, sind dessen direkte sichtbaren Nachfolger (bzw. Vorgänger) auch die direkten sichtbaren Nachfolger (bzw. Vorgänger) des neu eingefügten Knotens. Die Menge der direkt sichtbaren Vorgänger sei mit N_{pred} und die der direkt sichtbaren Nachfolger mit N_{succ} bezeichnet. Im Beispiel aus Abbildung 65 ist für den neu eingefügten Knoten X die Menge der direkt sichtbaren (grau markierter Knoten) Vorgänger $N_{pred} = \{A\}$ und die der Nachfolger $N_{succ} = \{C, D\}$.

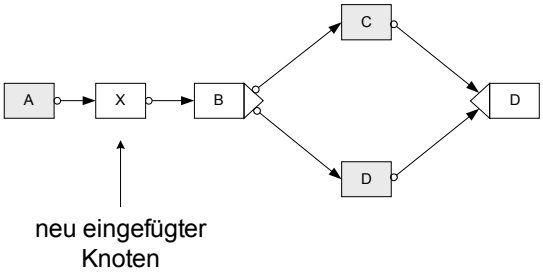

Abbildung 65: Sichtbare Vorgänger- und Nachfolgermengen

Zur Aktualisierung der View werden jetzt die Kanten zwischen N_{pred} und N_{succ} gelöscht und durch neue virtuelle Kanten zwischen N_{pred} und dem neuen Knoten bzw. zwischen dem neuen Knoten und N_{succ} ersetzt. Dabei werden für jede Kante zwischen einem Knoten P aus N_{pred} und einem Konten S aus N_{succ} zwei virtuelle Kanten eingefügt. Eine davon hat dabei P als Quell- und den neuen Knoten als Zielaktivität, die andere den neuen Knoten als Quell- und den Knoten S als Zielaktivität. Im Beispiel aus Abbildung 66 wird zwischen A und B ein neuer Knoten X eingefügt. Der eindeutig sichtbare Vorgänger ist A und die sichtbaren Nachfolger sind C und D. Zur Aktualisierung werden die Kanten zwischen A, C und D gelöscht, und neue virtuelle Kanten zwischen A und X bzw. zwischen X, C und D eingefügt.

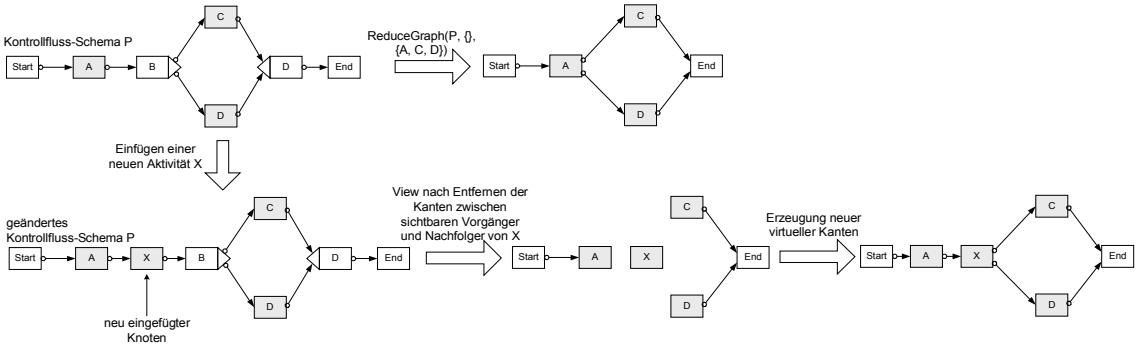

Abbildung 66: Aktualisierung nach dem Einfügen einer Aktivität

Wenn ein ganzer Kontrollblock neu eingefügt wird, bzw. durch eine Einfügeoperation (z.B. *parallelInsert*) ein ganzer Kontrollblock betroffen ist, werden zuerst alle im Kontrollblock vorhandenen sichtbaren Knoten ausgeblendet. Daran anschließend kann für alle im Kontrollblock enthaltenen relevanten Knoten eine neue Subview gebildet werden. Mit Subview ist dabei folgendes gemeint: Man fasst den eingefügten Kontrollblock als eigenständigen Prozess auf. Ein Kontrollblock hat einen eindeutigen Start- und Endknoten, dies sind dann die Start- und Endknoten des Prozessgraphen. Für diesen Prozessgraphen kann jetzt mit dem Graphreduktionalgorithmus eine normale View-Bildung für die relevanten Knoten durchgeführt werden. Durch den Graphreduktionsalgorithmus ist sichergestellt, dass es für diese Subview immer einen eindeutig sichtbaren Start- und Endknoten gibt. Anschließend werden, wie beim Einfügen einer einzelnen Aktivität, die Mengen N_{pred} für den sichtbaren Startknoten und N_{succ} für den sichtbaren Endknoten des Kontrollblocks bestimmt. Zwischen den Knoten aus N_{pred} und dem sichtbaren Startknoten des Kontrollblocks werden neue virtuelle Kanten eingefügt, dass gleiche gilt für den sichtbaren Endknoten des Kontrollblocks und den Knoten der Menge N_{succ}. Die vorher sichtbaren Kanten zwischen dem sichtbaren Vorgänger des Startknotens und dem sichtbaren Nachfolger des Endknotens werden gelöscht. Im Beispiel aus Abbildung 67 wird ein Knoten X parallel zu dem Kontrollblock zwischen Knoten B und G eingefügt. Für den durch das Einfügen entstandenen Kontrollblock wird dann eine Subview gebildet und mit dieser die View entsprechend aktualisiert.

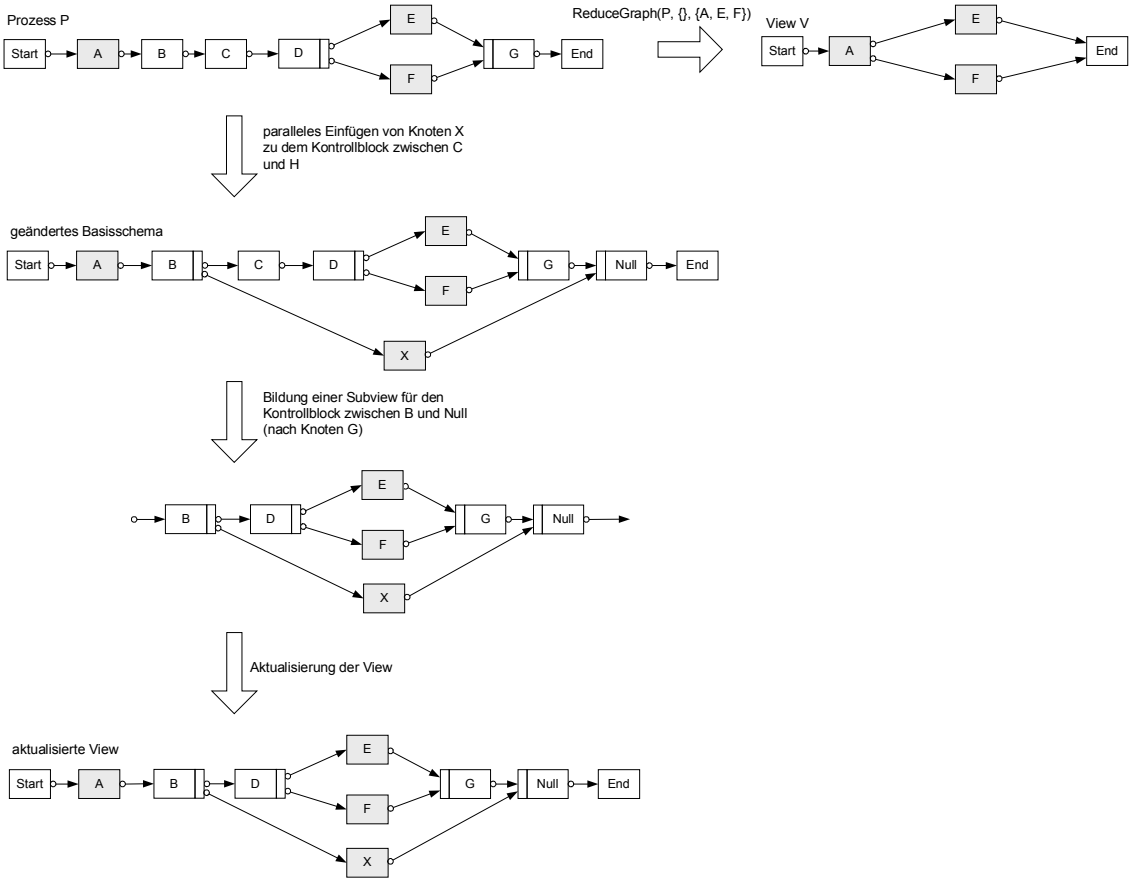

Abbildung 67: Aktualisierung nach Einfügen einer Aktivität

- **Verschieben von Aktivitäten**
  Prinzipiell kann das Verschieben von Aktivitäten beim View-Update wie Löschen und anschließendes Einfügen der Aktivitäten mit Hilfe der oben beschriebenen Operationen durchgeführt werden. Da beim Verschieben von Aktivitäten lediglich Kanten „umgehängt“ werden, ist es auch möglich, die jeweils sichtbaren Vorgänger und Nachfolger vor und nach dem „Umhängen“ zu identifizieren, und anschließend die virtuellen Kanten entsprechend anzupassen

8.2.2 Aktualisierung graphaggregierter Views

Da aggregierte Views strukturerhaltend sind, sind Aktualisierungen hier deutlich leichter durchzuführen. Wir diskutieren hier wieder entlang der verschiedenen Änderungsoperationen.

- **Löschen von Aktivitäten**
  Es muss zwischen verschiedenen Löschoperationen unterschieden werden:

– *Löschen einer einzelnen Aktivität oder eines Kontrollblocks innerhalb eines virtuellen Schrittes*. Hier muss die View nicht aktualisiert werden, da die Start- und Endknoten aller virtuellen Schritte immer noch gültig sind. Wie im Beispiel aus Abbildung 68 deutlich wird, ist beim Löschen des Knotens B, der sich innerhalb des virtuellen Schrittes V befindet, keine Aktualisierung der View notwendig.

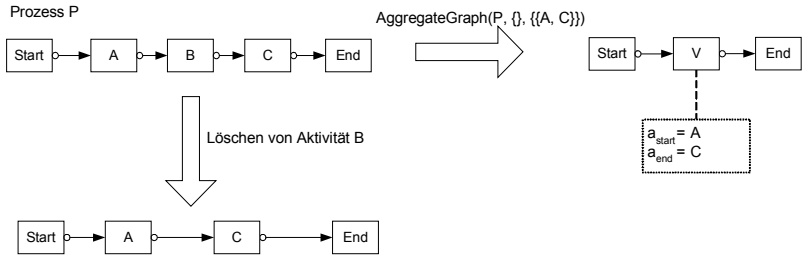

Abbildung 68: Löschen einer Aktivität innerhalb eines virtuellen Schrittes

– *Eine Aktivität, die Start- oder Endknoten einer virtuellen Aktivität ist, wird gelöscht.* Dies kann nur eine einzelne Aktivität, also kein Split- oder Join-Knoten sein, da sonst die Löschoperation nicht gültig wäre. Der gelöschte Knoten hat also genau einen direkten Nachfolger (wenn es sich um einen Startknoten handelt) oder einen direkten Vorgänger (bei einem Endknoten). Damit kann der Start- bzw. Endknoten der virtuellen Aktivität einfach angepasst werden, indem der entsprechende Attributwert (v_{start} bzw. v_{end}) auf diesen Vorgänger bzw. Nachfolger gesetzt wird.

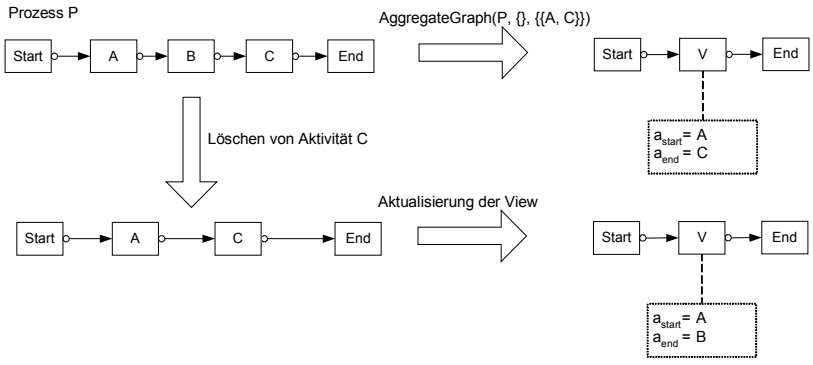

Abbildung 69: Löschen einer Start-Aktivität eines virtuellen Knotens

– *Löschen eines Kontrollblocks, der einen virtuellen Schritt vollständig umfasst*[25] *oder sogar auf mehrere (aufeinander folgende) virtuelle Schritte verteilt ist*. In diesem Fall müssen auch virtuelle Schritte gelöscht werden. Bei allen virtuelle Knoten, bei denen

[25] Wenn Aktivitäten bei der Aggregation ohne Änderung übernommen wurden, können diese auch als virtueller Schritt aufgefasst werden, der dann nur diese eine Aktivität beinhaltet.

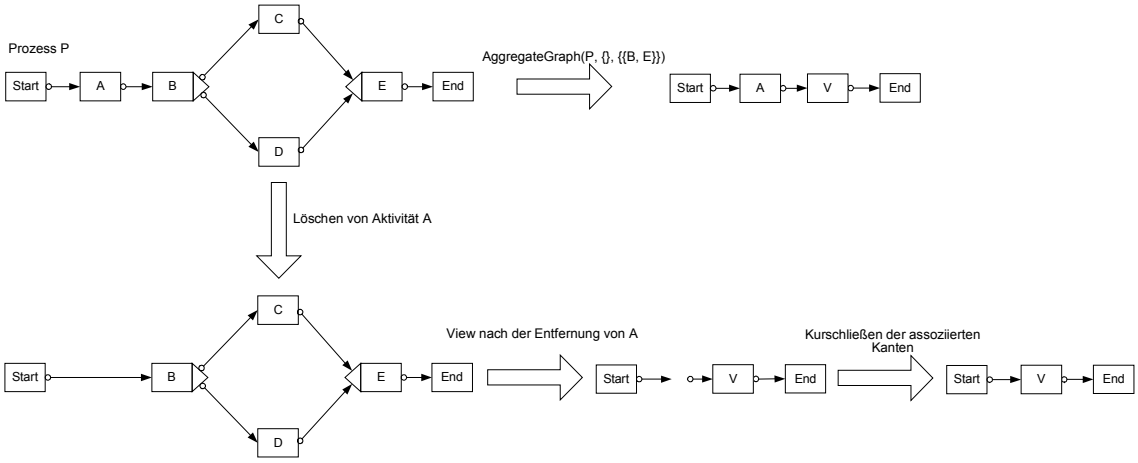

Abbildung 70: Löschen eines virtuellen Knotens

sowohl der Start- als auch der Endknoten gelöscht wurde, ist der gesamte aggregierte Kontrollblock gelöscht worden, also kann auch der virtuelle Schritt entfernt werden. Für alle anderen muss der Start- und Endknoten, wie im vorherigen Punkt, entsprechend angepasst werden.

Jetzt sind, wie bei den graphreduzierten Views, noch mehrere, mit den gelöschten virtuellen Knoten assoziierte virtuelle Kanten vorhanden. Die freistehenden Kanten können gelöscht werden, die verbleibenden werden, wie bei der Aktualisierung graphreduzierter Views „kurzgeschlossen". Im Beispiel aus Abbildung 70 wird ein Knoten, der direkt (also durch einen einzelnen virtuellen Schritt repräsentiert wird) in der View vorhanden ist, gelöscht. In der View gibt es nach dem Entfernen des virtuellen Schrittes Kanten ohne Quell- bzw. Zielknoten, diese werden wie bei der Aktualisierung graphreduzierte Views „kurzgeschlossen".

Ein komplexeres Beispiel zeigt Abbildung 71. Hier wird ein ganzer Kontrollblock gelöscht, der sich in der View über vier virtuelle Knoten erstreckt. Zwei der virtuellen Knoten werden in der View komplett entfernt, es entstehen dadurch freistehende Kanten. Bei den verbleibenden Knoten V_1 und V_4 müssen die Attributwerte für die Start- und Endknoten entsprechend angepasst werden. Anschließend werden noch die Kanten zwischen V_1 und V_4 kurzgeschlossen.

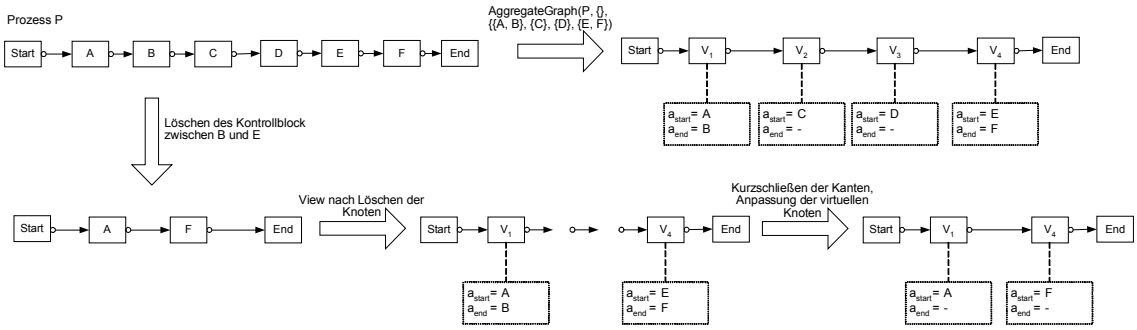

Abbildung 71: Löschen mehrere virtueller Knoten

- **Einfügen von Aktivitäten**
  Hierbei stellt sich, wie bei der Graphreduktion erwähnt, die Frage, ob neu eingefügte Knoten in der View sichtbar sein sollen oder nicht. Dies kann z.B. über die Relevanz oder die anderen beschriebenen Möglichkeiten geschehen. Im Gegensatz zur Graphreduktion muss bei Views, die durch Graphaggregation erzeugt werden, eine Aktivität auch dann beachtet werden, wenn sie nicht sichtbar sein soll. Dies bedeutet nämlich, dass sie in einem bereits vorhandenen virtuellen Schritt mit aggregiert werden muss.
  Es sind ähnliche Fälle zu unterscheiden wie beim Löschen von Knoten:

  – *Serielles Einfügen einer Aktivität die innerhalb der Menge eines virtuellen Knotens liegt:* An der View muss nichts geändert werden, da alle virtuellen Schritte noch gültige Start- und Endknoten haben. Falls der neu eingefügte Knoten als einzelner Knoten in der View sichtbar sein soll, muss der virtuelle Knoten gesplittet werden.
    Seien n_{start} und n_{end} der Start- und Endknoten des virtuellen Schrittes, und n_{pred} und n_{succ} die eindeutigen direkten Vorgänger bzw. Nachfolger des neu eingefügten Knotens. Dann wird der virtuelle Schritt gelöscht, und durch drei neue ersetzt. Der erste hat als Startknoten n_{start} und als Endknoten n_{pred}. Der zweite neue virtuelle Knoten beinhaltet die neu eingefüge Aktivität und der dritte Knoten hat als Startknoten n_{succ} und als Endknoten n_{end}. Anschließend müssen noch Kanten zwischen den neuen virtuellen Schritten eingefügt werden. Abbildung 72 verdeutlicht diese beiden Alternativen. Ein neuer Knoten wird zwischen B und C eingefügt. Im ersten Fall bleibt der virtuelle Knoten unverändert mit A als Start- und C als Endknoten. Im zweiten Fall wird der virtuelle Knoten aufgetrennt, und X ist als einzelner virtueller Schritt auch in der View vorhanden.

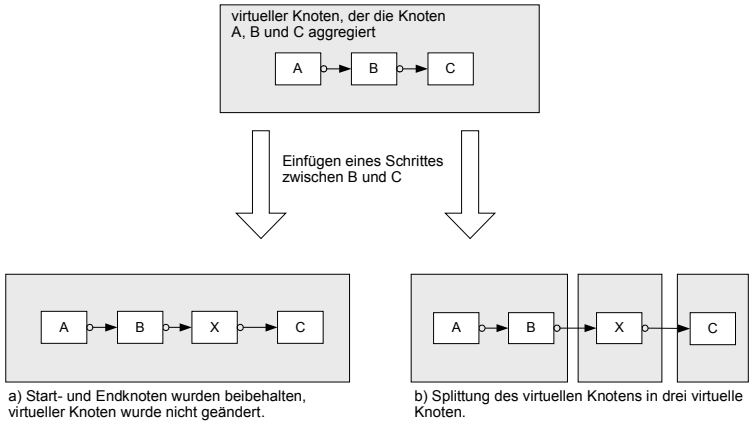

Abbildung 72: Einfügen einer Aktivität innerhalb eines virtuellen Knotens

– *Serielles Einfügen einer Aktivität die nicht innerhalb des Kontrollblocks eines virtuellen Schrittes liegt*. Eine solch neu eingefügte Aktivität liegt zwangsläufig zwischen zwei virtuellen Schritten. Soll sie nicht sichtbar sein, wird sie einfach einer virtuellen Aktivität hinzugefügt, also entweder der neue Endknoten des virtuellen Vorgängers oder der neue Startknoten des virtuellen Nachfolgers. Soll der Schritt sichtbar sein, wird er einfach in der View durch einen virtuellen Schritt ersetzt. In Abbildung 73 werden diese beiden Alternativen verdeutlicht. Im Fall a) wird der neue Knoten X zu dem ersten virtuellen Knoten hinzugefügt, im Fall b) wird er als einzelner Knoten auch in die View dargestellt.

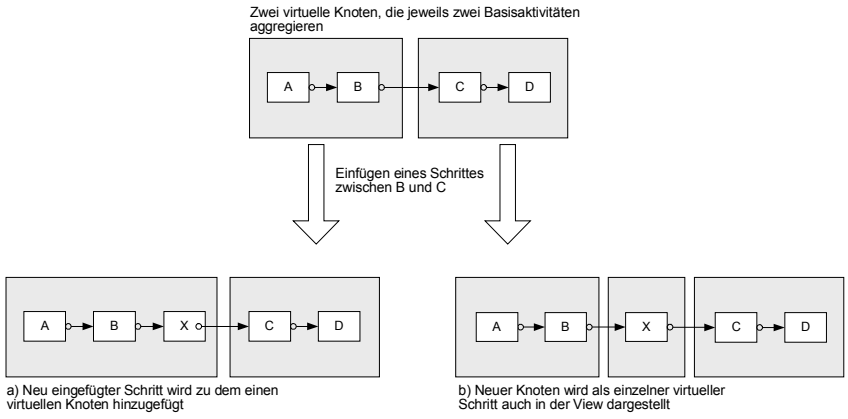

Abbildung 73: Einfügen einer Aktivität zwischen zwei virtuellen Knoten

– *Paralleles Einfügen einer Aktivität*. Wenn die entsprechenden Split- und Join-Knoten innerhalb eines virtuellen Knoten liegen, muss nichts geändert werden. D.h. die View ist immer noch gültig. Ein Splitten eines virtuellen Schrittes zur gesonderten

Darstellung des eingefügten Schrittes ist bei einer solchen Einfügeoperation aufwändig, da hier unter Umständen sehr viele neue virtuelle Knoten entstehen können. Es soll deshalb hier nicht weiter betrachtet werden.

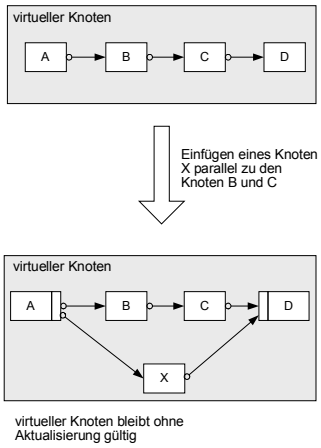

Abbildung 74: Paralleles Einfügen innerhalb eines virtuellen Knotens

In Abbildung 74 wird parallel zu den Knoten B und C ein neuer Knoten X eingefügt. Da beim Einfügen nur Knoten und Kanten innerhalb des virtuellen Schrittes betroffen sind, bleibt der virtuelle Knoten nach dem Einfügen von X ohne Aktualisierung gültig. Eine andere Möglichkeit ist, das der parallel eingefügte Schritt parallel zu ein oder mehreren virtuellen Schritten eingefügt wurde. Hier gibt es zwei Ansätze zur Aktualisierung. Entweder wird der gesamte betroffene Block, vom Split- bis zum Join-Knoten des neu eingefügte Schrittes zu einem neuen virtuellen Knoten aggregiert sowie alle dazwischen liegenden virtuellen Knoten entfernt, oder die eingefügte Aktivität und Verzweigungsknoten werden auch in die View übernommen und durch virtuelle Schritte ersetzt. In Abbildung 75 werden die beiden Alternativen an einem Beispiel verdeutlicht. Parallel zu zwei virtuellen Knoten wird eine neue Aktivität X eingefügt. Im Fall a) werden die virtuellen Knoten entfernt, und der gesamte betroffene Kontrollblock durch einen virtuellen Knoten ersetzt. Im Fall b) bleiben die bisherigen Knoten erhalten, und für die neue Aktivität X und die Verzweigungsknoten werden neuen virtuelle Knoten und Kanten erzeugt.

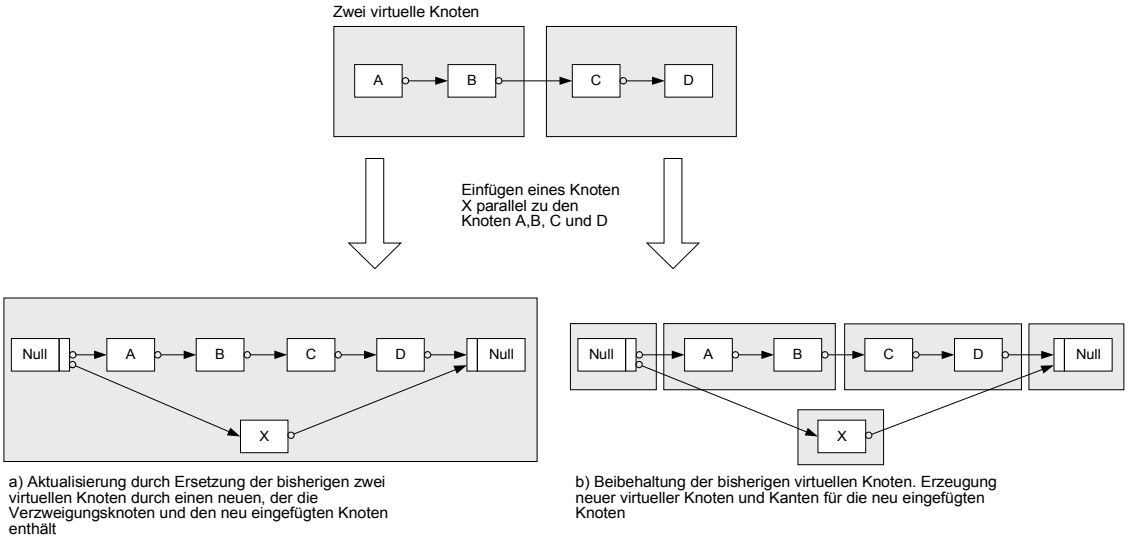

Abbildung 75: Alternativen zur Aktualisierung beim parallelen Einfügen

– *Einfügen einer neuen Aktivität in einer Verzweigung (branchInsert)*. Falls die betroffenen Verzweigungsknoten innerhalb eines virtuellen Schrittes liegen, muss wie beim parallelen Einfügen nichts aktualisiert werden. Sind die Verzweigungsknoten auch separat in der View vorhanden, kann der neu eingefügte Schritt auch einfach in die View übernommen und durch einen neuen virtuellen Schritt dargestellt werden. Im Beispiel aus Abbildung 76 wird zwischen den Verzweigungsknoten A und E eine neue Aktivität X eingefügt. Diese wird anschließend in der View durch einen neuen virtuellen Knoten sichtbar gemacht.

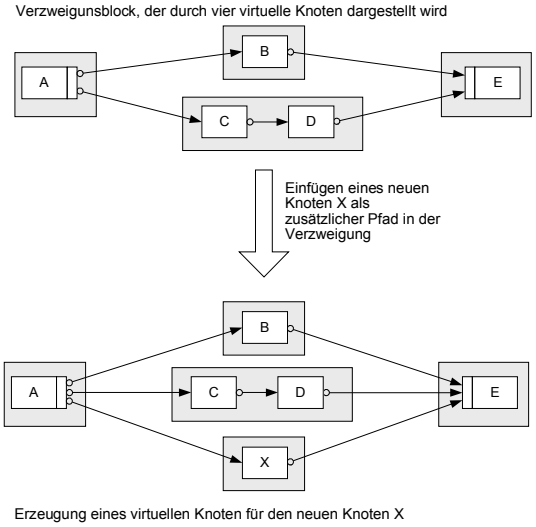

Abbildung 76: Einfügen einer neuen Aktivität mit branchInsert

– *Einfügen von Kontrollblöcken*. Hier gilt das für einzelne Aktivitäten Gesagte, nur dass jeweils die Start- und Endknoten betrachtet werden müssen, und der Kontrollblock vorher noch durch eine Subview-Bildung[26] verkleinert und für die View angepasst werden kann. Im Beispiel aus Abbildung 77 wird zwischen den Knoten B und C ein neuer Kontrollblock, bestehend aus den Knoten X, Y und Z, eingefügt. Für diesen Kontrollblock wird eine Subview-Bildung (Konten X und Y werden aggregiert, Z wird durch einen virtuellen Knoten ersetzt) durchgeführt, anschließend wird die View aktualisiert.

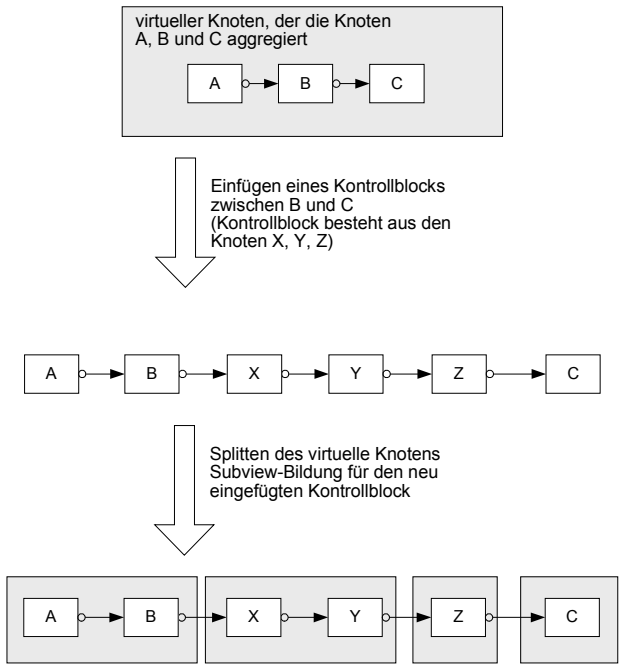

Abbildung 77: Einfügen eines Kontrollblocks

Im Beispiel aus Abbildung 78 wird das serielle Einfügen einer neuen Aktivität X verdeutlicht. Im Basisschema P wird die neue Aktivität X zwischen Aktivität A und B eingefügt. Diese neue Aktivität soll in der für P gebildeten View nicht als seperate Aktivität sichtbar sein. Deshalb wird sie in die virtuelle Aktivität V_2 aufgenommen und bildet dort dann den neuen Startknoten.

[26] Diese wird hier im Gegensatz zu Abschnitt 8.2.1 mit Hilfe des Graphaggregationsalgorithmus durchgeführt.

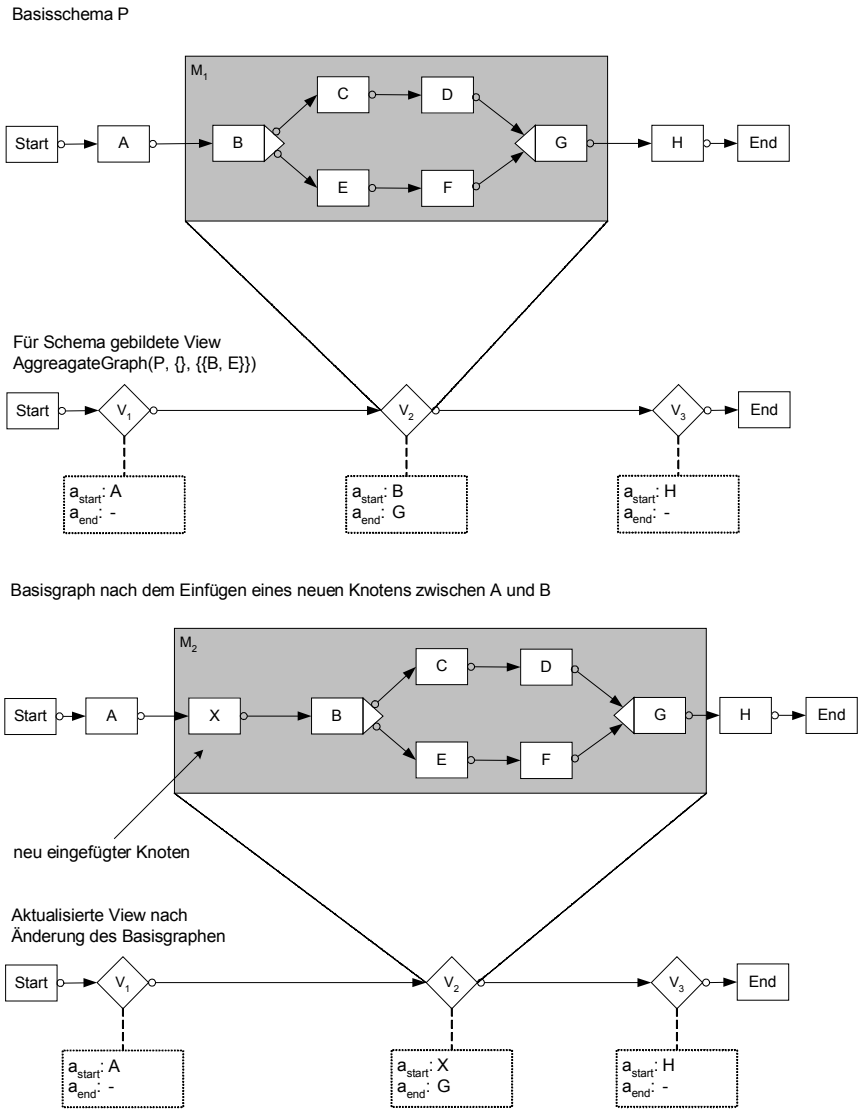

Abbildung 78: Aktualisierung nach dem Einfügen einer Aktivität

8.3 Schemaänderungen über Views

Für viele Anwendungsfälle ist es notwendig, über eine View auch das Basisschema oder das Schema der zugrundeliegenden Prozess-Instanz (Ad-Hoc Änderungen) ändern zu können. Dies ist z.B. der Fall, wenn eine View dazu dient, einem Prozess-Modellierer nur bestimmte Teile eines Prozesse detailliert darzustellen, und den Rest zwecks Komplexitätsreduzierung auszublenden bzw. zu vereinfachen. Noch bedeutsamer wird diese Möglichkeit im Zusammenhang mit der Unterstützung von Ad-Hoc-Änderungen, die üblicherweise von Fachanwendern und nicht Computerexperten vollzogen werden müssen.

8.3.1 Schemaänderung über aggregierte Views

Bei einer Schemaänderung geht es meist um das Einfügen, Löschen und Verschieben von Aktivitäten und Datenknoten. Damit die Operationen und Algorithmen des Basismodells auch für Views anwendbar bleiben, ist es notwendig, dass eine View selbst wieder eine gültige Prozessbeschreibung (gemäß der Definition des ADEPT-Basismodells) darstellt. Werden Schemaänderungen auf Grundlage der Operationen des Basismodells vorgenommen, kommen also nur Views in Frage, die durch einen der in Abschnitt 5.3.5 beschriebenen Algorithmen erzeugt worden sind.

Bei Schemaänderungen über Views müssen verschiedene Aspekte beachtet werden: Um die nachfolgenden Diskussionen nicht zu überfrachten, fokussieren wir zunächst auf Kontrollflussaspekte. Um das zugrunde liegende Basisschema ändern zu können, muss bei Anwendung einer Änderungsoperation auf die View feststehen, auf welche Knoten und Kantenmengen des Basisschemas sich diese beziehen. Eine Schemaänderung kann dann wie folgt durchgeführt werden:

1. Auf der View wird eine der ADEPT-Änderungsoperationen z.B. [*serial*, *parallel*, *branch*] *Insert*, *deleteActivity* oder *moveActivity* mit den jeweils notwendigen Start- und Endknoten aufgerufen
2. Für die Durchführung dieser Operationen gibt es zwei Alternativen möglich:

   - Die Änderungen werden direkt auf der View ausgeführt, und anschließend auf das Basisschema übertragen. Dazu müssen alle in der View neu hinzugekommenen sowie geänderten Knoten und Kanten betrachtet werden, um dann die im Basisschema betroffenen Knoten zu bestimmen[27] sowie die Kanten und Knoten auch dort einzufügen.
   - Es werden direkt für die den Änderungsoperationen übergebenen Parameter die entsprechenden Knoten im Basisschema bestimmt, und die Änderungsoperationen dann direkt auf dem Basisschema durchgeführt. Anschließend kann die View, wie im vorherigen Abschnitt beschrieben, aktualisiert werden.

Im Beispiel aus Abbildung 79 werden diese beiden Alternativen verdeutlicht. In einer aggregierten View soll zwischen den Knoten V_1 und V_2 ein neuer Knoten X eingefügt werden. Im Fall a) wird die Änderung direkt auf der View durchgeführt (und dann erst anschließend auf das Basisschema übertragen), im Fall b) werden die entsprechenden Knoten im Basisschema (hier die Knoten A und B) bestimmt und die Änderung dort durchgeführt.

---

[27] Jeder virtuelle Knoten hat einen eindeutigen Start- und Endknoten im Basisschema, folglich kann für jede hinzugefügte und geänderte Kante der entsprechende Start- und Endknoten im Basisschema bestimmt werden.

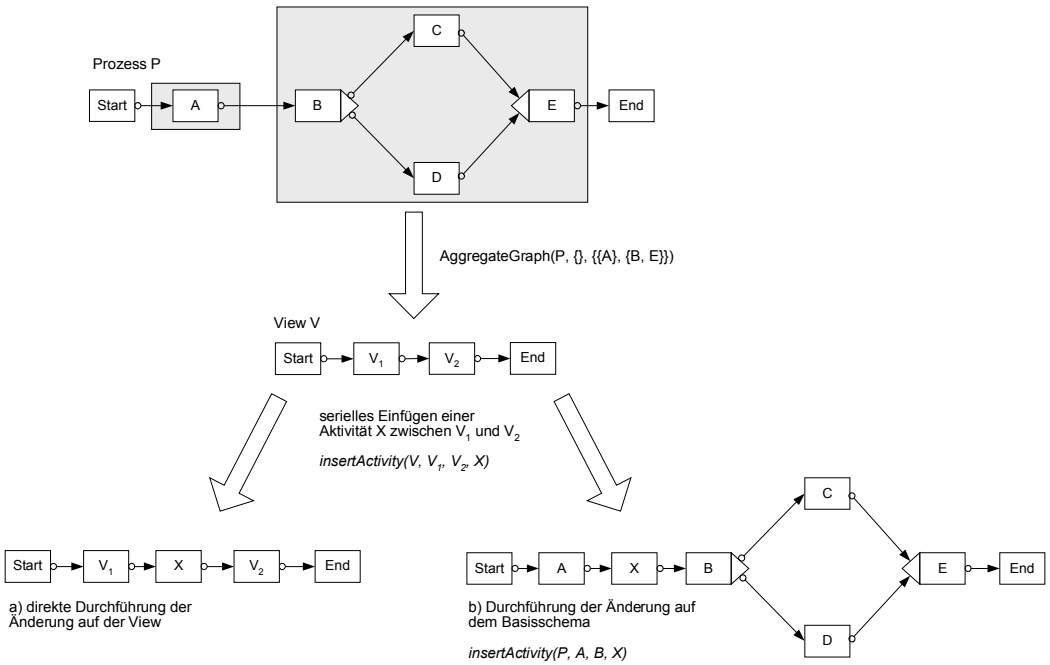

Abbildung 79: Alternativen für die Durchführung von Schemaänderungen

Ein Problem ergibt sich wenn Synchronisationskanten von den Änderungsoperationen betroffen sind. So werden z.B. bei der ADEPT-Operation *insertBetweenNodeSets* etwa neue Sync-Kanten in den Prozess eingefügt. Das kann dazu führen, dass die View dann nicht mehr für weitere Änderungsoperationen verwendet werden kann, weil z.B. zwei Sync-Kanten zwischen zwei virtuellen Knoten zusammenfallen. Aus diesem Grund eignet sich die erstgenannte Alternative (Ausführen der Änderungsoperation auf der View und anschließendes Nachziehen der Änderungen auf dem Basisschema) nur für einfache Operationen ohne Änderung von Sync-Kanten; in der View können bereits Sync-Kanten zwischen zwei Knoten vorhanden sein, so dass es dann nicht möglich ist parallel eine neue Sync-Kante mit gleichen Start- und Endknoten einzufügen.

Neben Kontrollflussaspekten ist auch eine Betrachtung des Datenflusses bei Schemaänderungen notwendig. Manche Aktivitäten hängen davon ab, dass der Wert eines gelesenen Datenelementes vorher gesetzt wurde. Damit eine solche Aktivität in einer View eingefügt werden kann, muss aus der View hervorgehen, ob das entsprechende Datenelement in jedem Fall von einer virtuellen Aktivität geschrieben wird (siehe hierzu auch Abschnitt 4.2.3). Ansonsten ist der Datenfluss für Änderungsoperationen die Aktivitäten betreffen unproblematisch.

Das gleiche Problem wie bei Sync-Kanten ergibt sich, wenn der Datenfluss selbst geändert werden soll, da durch Aggregation von Aktivitäten auch mehrere Datenflusskanten zusammenfallen können bzw. bei neu eingefügten Kanten nicht klar ist, mit welcher Aktivität

des Basisschemas ein Datenelement verknüpft werden soll. Eine Möglichkeit zur Umgehung dieses Problems ist es sich beim Verschieben oder Neuanlegen einer Datenflusskante immer auf den Start- oder Endknoten innerhalb einer virtuellen Aktivität zu beziehen, oder Änderungsoperationen bezogen auf den Datenfluss nur für Basisaktivitäten zuzulassen.

8.3.2 Schemaänderung über graphreduzierte Views

Graphreduzierte Views haben gegenüber Views die durch eine Graphaggregation entstanden sind den Vorteil, dass bei der View-Bildung, neben den für die View relevanten Aktivitäten, weniger zusätzliche Knoten in die View übernommen werden müssen. Der Grund dafür ist, dass keine Rücksicht auf den Erhalt der Struktur genommen werden muss. Leider resultiert hieraus der Nachteil, dass Operationen zur Schema-Änderung auf Grundlage des Basismodells nicht angewendet werden können. Jedoch ist es wünschenswert, auf diesen graphreduzierten, von der Struktur her sehr einfachen Views dem Benutzer ebenfalls Änderungen zu ermöglichen.

Eine Möglichkeit, dies zu realisieren, ist die Definition von Änderungsoperationen die auf der View angewendet werden können und anschließende Übersetzung dieser Änderungsoperationen in geeignete Operationen des Basismodells. So können für eine graphreduzierte View folgende Operationen definiert werden:

- Löschen einer Aktivität
- Einfügen einer Aktivität zwischen zwei anderen Aktivitäten der View

Diese Operationen lassen sich sehr leicht in notwendige Operationen auf dem Basismodell übersetzen:

```
DeleteViewActivity(CFS, DFS, VCFS, X)
  input:
      CFS, DFS: WF-Graph auf dem die View basiert
      VCFS: View WF-Graph
      X: Aktivität in der View die gelöscht werden soll
  begin:
      DeleteActivity(CFS,DFS, X);
  end
```

Die Operation für das Löschen lässt sich am einfachsten auf das Basismodell übertragen, da die entsprechende Aktivität (die eventuell in der View anders benannt ist) einfach im Basismodell gelöscht wird und anschließend die View neu gebildet wird.

InsertActivityBetweenViewActivities(CFS, DFS, VCFS, nodeLabel, template, N_1, N_2)
input:
CFS, DFS: WF-Graph auf dem die View basiert
VCFS: View WF-Graph
nodeLabel: Bezeichner für die neue Aktivität
template: Vorlage für die neue Aktivität
N_1: Aktivität nach der die neue Aktivität X eingefügt werden soll
N_2: Aktivität vor der die neue Aktivität X eingefügt werden soll
begin:
n_{start}:= Startknoten des minimalen Kontrollblocks der N_1 und N_2 enthält;
n_{end}:= Endknoten des minimalen Kontrollblocks der N_1 und N_2 enthält;
parallelInsert(CFS, n_{start}, n_{end}, nodeLabel, template);
insertSyncEdge(CFS, N_1, nodeLabel, `STRICT_SYNC_E`);
insertSyncEdge(CFS, nodeLabel, N_2, `STRICT_SYNC_E`);
end

Da in der graphreduzierten View durch das Verbergen von Knoten zwischen N_1 und N_2 mehrere nicht sichtbare Knoten im Basisprozess vorhanden sein können, wird ein neuer Knoten einfach parallel zum minimalen Kontrollblock, der N_1 und N_2 enthält, eingefügt und anschließend das Ausführen der neuen Aktivität nach N_1 und vor N_2 mit Hilfe zweier Sync-Kanten sichergestellt.

Abbildung 80 zeigt dazu ein Beispiel. Zudem wird eine besondere Problematik bei dieser Art von Änderung über eine View deutlich: Wenn nach erfolgreicher Änderung des Basisprozesses die View aktualisiert wird, sieht diese meist anders aus, als dies der Benutzer erwartet. Wenn der Benutzer eine neue Aktivität X zwischen B und D einfügt, wird ein serielles Einfügen, wie in dem dritten Graphen aus Abbildung 80 erwartet. Tatsächlich ergibt sich durch das parallele Einfügen jedoch als View nach einer Aktualisierung der untere Graph in Abbildung 80. Dieses Problem könnte man eventuell dadurch umgehen, indem auf dem View-Graphen noch Vereinfachungsregeln angewendet werden und man dann einfachere Views erzeugen kann.

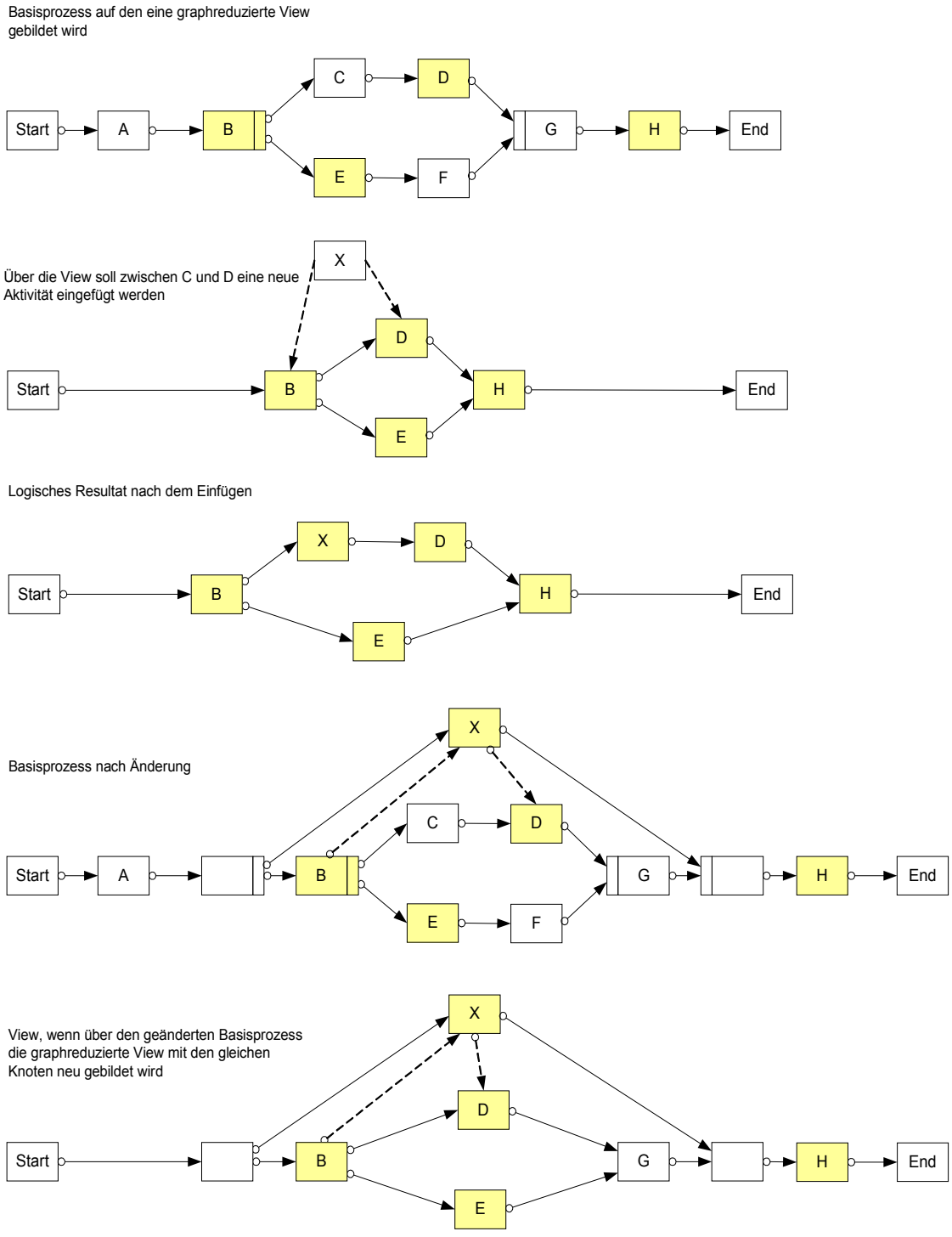

Abbildung 80: Schemaänderung über eine graphreduzierte View

8.4 Zusammenfassung

Um Views im praktischen Einsatz verwenden zu können, ist es auf jeden Fall notwendig, dass auch ihre Aktualisierung unterstützt wird. So ist die Aktualisierung von Views nach Änderungen des Basisschemas für alle denkbaren Anwendungsfälle wichtig. Je nach Art der View und der durchgeführten Änderung auf dem Basisschema kann diese Aktualisierung aufwändig sein, so dass auch immer abgewogen werden muss, ob ein Neuaufbau der View einfacher durchführbar ist. In allen Anwendungsfällen, in denen über eine View auch das Schema geändert werden soll, muss dies zusätzlich unterstützt werden. Dies ist jedoch, je nach Art der View, mit Einschränkungen verbunden, vor allem bei graphreduzierten Views oder wenn Sync-Kanten bei der Änderung beteiligt sind.

Für die Aktualisierung von Views könnten weitergehende Untersuchungen durchgeführt werden:

- Die formale Definition der hier größtenteils nur informell vorgestellten Aktualisierungsoperationen.
- Die Möglichkeit, effizient zu ermitteln, wann eine Aktualisierung oder der Neuaufbau einer View besser ist.
- Vereinfachung von graphreduzierten Views, die durch Schemaänderungen, wie in Abschnitt 8.3.2 beschrieben, verändert wurden.

9 Verwandte Arbeiten

In diesem Kapitel werden einerseits Ansätze für Views in Datenbanken und Prozess-Management-Systemen verglichen, und andererseits ein kurzer Überblick über vorhandene Ansätze für Prozess-Views gegeben.

9.1 Vergleich Views in Datenbanken und Prozessen

Views sind ein von Datenbanken bekanntes Konzept. Views in Datenbanken und bei Prozessen haben einerseits viele Gemeinsamkeiten, andererseits unterscheiden sie sich aber auch in vielen Punkten. Es sollen hier die wichtigsten Punkte miteinander verglichen werden.

1. **Anwendungsfälle**
   In Datenbanken werden Views vor allem aus folgenden Gründen verwendet:

   - Sicherheitsaspekte (Verbergen geheimer Informationen)
   - Kapselung von Daten (Programme greifen nur über Views auf Daten zu)
   - Verringern von Komplexität (Ausblenden von Spalten, Zusammenfassung von Relationen mit Joins)
   - Verkleinerung der zu übertragenden Datenmenge, wenn Clients auf die Datenbank zugreifen
   - Effizienzgründe, z.B. materialisierte Views in Data-Warehouse-Systemen

   Alle diese Anwendungsfälle spielen auch bei Views für Prozesse eine Rolle. Da Views in Datenbanken oftmals aus verschiedenen Relationen zusammengesetzt sind, ist es bei Datenbank-Views nicht möglich, über die View das Schema der Relationen zu ändern. Auch die Durchführung von Updates auf Tupeln über Views ist meistens nur eingeschränkt möglich. Ein möglicher Ansatz für den Update von Tupeln wird z.B. in [17] behandelt.
2. **Definition von Views**
   Bei Prozessen wird die View durch Attribute (VISIBLE, VIRTUAL) an den einzelnen Graphelementen gespeichert. Wichtig hierbei ist, das es eine Unterscheidung zwischen dem Schema- und Instanzgraphen gibt. Views können sowohl für ein Schema als auch für eine Instanz gebildet werden. Bei Datenbanken gibt es diese Unterscheidung nicht, Views können nur für bestehende Relationen und bereits vorhandene Views gebildet werden. Dabei werden Views einfach durch eine normale Query definiert.
3. **Arten von Views**
   Bei Views für Prozesse können zwei Arten für die Viewbildung verwendet werden: Graphreduktion und Graphaggregation. Je nach Art der View-Bildung kann die View dann für verschiedene Anwendungsfälle verwendet werden. Da in Datenbanken die

Views durch eine normale Query definiert werden, sind für eine View die Kombination aller Möglichkeiten einer Query (Selektion, Projektion, Joins, Mengenoperationen und Aggregation auf Relationen) nutzbar.

4. **Operationen auf Views**
   Bei Prozess-Views sind, je nach Art der Viewbildung, Leseoperationen, Instanz- und Schemaänderungen möglich. Damit können grundsätzlich alle Operationen auf Prozessen auch über Views durchgeführt werden. Im Gegensatz dazu werden Views in Datenbanken vor allem für Leseoperationen genutzt, da für die meistens Anwendungsfälle auch nur Leseoperationen benötigt werden. Teilweise sind auch bei Datenbanken die Daten in den Relationen über Views änderbar.
5. **Algorithmen zur Viewbildung**
   Da Prozesse normalerweise durch Graphen repräsentiert werden, sind für die Erzeugung von Views Graphalgorithmen notwendig. Mit diesen Algorithmen kann ein Graph dann so verändert werden, dass die benötigten Informationen dann in dem View-Graphen zur Verfügung stehen. Bei Datenbanken sind für die Erzeugung von Views keine speziellen Algorithmen notwendig, da eine View einfach durch eine normale Query definiert wird.
6. **Erzeugung von Views**
   Die Erzeugung von Views kann für Prozesse je nach Anwendungsfall im Client oder im Server erfolgen, teilweise auch auf beiden Seiten. Da für die Erzeugung von Datenbank-Views sämtliche für die View benötigten Relationen verfügbar sein müssen erfolgt hier die View-Erzeugung normalerweise im Server.
7. **Speicherung und Verwaltung von Views**
   Bei Prozessen können die Views als Sequenz von Anweisungen (Aufruf der Algorithmen zur Viewbildung), als Attribute (VISIBLE und VIRTUAL) am Prozessgraphen oder als komplette Prozesskopie (die durch die Viewbildung dann geändert wird) gespeichert werden. Bei den Datenbanken gibt es vor allem zwei Möglichkeiten für die View-Speicherung: entweder wird die Query für die Viewbildung unter einem Namen gespeichert und kann mit diesem Namen wie eine normale Relation angesprochen werden, oder die View wird in einer neuen Relation als M*aterialized View* gespeichert.
8. **Aktualisierung von Materialized Views**
   Bei der Aktualisierung von materialisierten Sichten stellt sich sowohl bei Prozess-Views als auch bei Views in Datenbanken die Frage, wie die Views anschließend effizient aktualisiert werden können. In beiden Fällen muss abgewogen werden, ob es effizienter ist, einen Update der View durchzuführen oder ob man die View neu aufbaut.

9.2 Prozess-Views

Im Bereich des Prozess-Managements gibt es bereits erste Ansätze, die sich mit Teilaspekten von Prozess-Views beschäftigen. Diese Ansätze lassen sich grob in drei Kategorien unterteilen:

- Bildung einer Sicht auf organisationsübergreifende Prozesse, d.h. die Verknüpfung von Prozesse verschiedener Parteien und die Darstellung des virtuellen Gesamtprozesses durch eine View (d.h. einen logischen Prozessgraphen).
- Repräsentation ähnlicher Prozess-Schemata durch ein abstraktes Super-Schema (vergleichbar mit Konzepten der Vererbung aus objektorientierten Programmiersprachen).
- Die Ersetzung mehrere Basisaktivitäten durch eine virtuelle Aktivität, d.h. Viewbildung mit Hilfe der Graphaggregation.

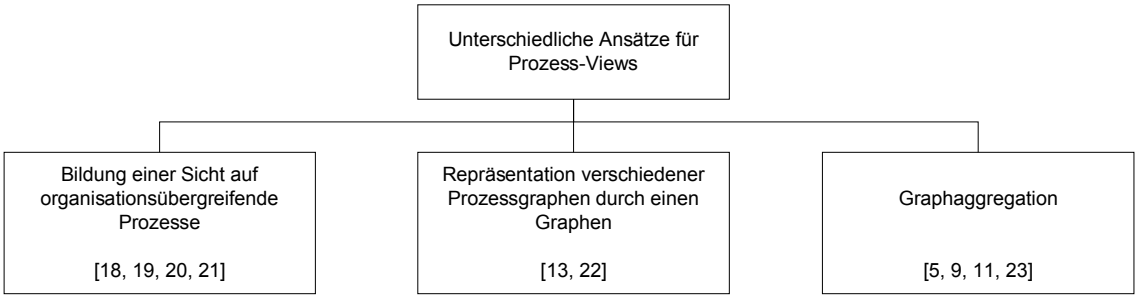

Abbildung 81: Ansätze für Prozess-Views

In *Workflow View Driven Cross-Organizational Interoperability in a Web Service Environment* [18] werden Views als Mechanismus für die Realisierung von organisationsübergreifenden Workflows verwendet. Dabei wird eine View als strukturell korrekte Teilmenge einer Workflow Definition mit Hilfe einer View Definition Language definiert. Der Kontrollfluss zwischen zwei an einem interorganisationellen Workflow Beteiligten findet dann mit Hilfe von Web Services statt. Die Views müssen manuell definiert werden, es werden keine Aussagen für Algorithmen zur Viewerzeugung oder für die Verwaltung und Speicherung von Views getroffen.

Ein ähnlicher Ansatz wird in dem Artikel *View-based Contracts in an E-service Cross-Organizational Workflow Environment* [19] verfolgt. Eine View wird als korrekte Teilmenge eines Workflow Graphen definiert. Einzelnen Aktivitäten können ausgeblendet oder umbenannt werden. Mit Hilfe einer Rechtevergabe kann für den Datenfluss in der View festgelegt werden, wer ein Datum lesen oder schreiben darf. Mit Hilfe dieser Views können dann interorganisationelle Workflows aufgebaut werden. Dafür definieren zwei Partner ihre jeweiligen Views, die an bestimmten Aktivitäten miteinander verknüpft werden können. Wenn

nur die Kanten zwischen den Views betrachtet werden, entsteht ein bipartiter Graph. Diese Kanten zwischen den Views werden mit Hilfe von *Contracts* ausgehandelt.

Bei *Modelling inter-organizational processes with process model fragments* [20] geht es ebenfalls um die Bildung von interorganisationellen Workflows. Es wird davon ausgegangen, dass mehrere Beteiligte verschiedene geografisch verteilte Prozesse haben, die für eine Zusammenarbeit miteinander verbunden werden sollen. Um dies realisieren zu können wird der Begriff des *Process Fragments* eingeführt. Mit Hilfe dieser Prozess-Fragmente kann ein Prozess in mehrere unabhängige Teile unterteilt werden. Diese Fragmente können dann mit Hilfe von *Events* miteinander verbunden werden. Ein Fragment löst einen Event aus und dadurch wird ein anderes Fragment (bzw. die erste Aktivität in diesem Fragment) gestartet. Daten werden zwischen Aktivitäten mit Hilfe von *Documents*, die Daten eines beliebigen Typs enthalten können, ausgetauscht. Jedes Fragment hat eine festgelegte Schnittstelle (welche Events und Daten benötigt werden) und ist somit gegenüber Änderungen gekapselt. Mit Hilfe dieser Fragmente können jetzt verschiedene Prozesse zu einem interorganisationellen Workflow miteinander verbunden werden. In diesem Ansatz finden sich mehrere Aspekte, die auch bei Prozess-Views eine Rolle spielen. Einerseits gibt es ein Kapselung der einzelnen Fragmente, die Schnittstelle stellt somit gewissermaßen eine Sicht auf ein Fragment dar. Andererseits kann durch das Verbinden mehrerer Fragmente durch Events eine View auf mehrere Prozessteile oder Prozesse erzeugt werden.

Bei *Finding Trading Partners to Establish Ad-hoc Business Processes* [21] handelt es sich um einen Ansatz mit Statecharts als Metamodell. Das Grundmodell wird um *Roles* erweitert. Views werden jetzt auf Basis der Rollen erstellt, indem für eine Rolle nicht relevante Events (beteiligte Rolle ist nicht Sender oder Empfänger) und Aktivitäten ausgeblendet werden. Mit Hilfe dieser Views können jetzt Partner gefunden werden, deren Prozesse mit dem eigenen Prozess kompatibel sind. Dieser Ansatz ist ähnlich der Graphreduktion, indem für eine Rolle irrelevante Schritte ausgeblendet werden. Die Auswahl relevanter Schritte basiert ausschließlich auf Rollen und ist daher relativ inflexibel.

In *Generic Workflow Models: How To Handle Dynamic Change and Capture Management Information?* [13] geht es um Prozesse, die mit Hilfe von Petrinetzen beschrieben werden. Hierbei handelt es sich um einen Ansatz, der dem der Vererbung in der objektorientierten Programmierung entspricht. Es gibt einerseits eine horizontale Dimension – Generalisierung und Spezialisierung – andererseits eine vertikale Dimension in Form von „ist Teil von“ und (bzw. „enthält“) für Prozesse, die mit Hilfe des *Generic Workflow Models* dargestellt werden. Konkret kann es mehrere Varianten eines Prozesses geben, die z.B. durch Ad-Hoc Änderung entstanden sein können. Damit man sich über mehrere Instanzen einen Überblick verschaffen kann, können diese Varianten mit Hilfe einer Funktion nach dem Prinzip des „größten gemeinsamen Teilers“ bzw. „kleinsten gemeinsamen Vielfachen“ auf ein Petrinetz abgebildet werden. Diese Petrinetze stellt dann eine abstrakte Sicht auf alle Prozesse dar. Bei dieser Art

von View handelt es sich im Prinzip um eine Aggregation von Prozess-Instanzen, wobei hier auch Instanzen mit unterschiedlichen Schemata auf eine Sicht gemappt werden können.

Auch in *The P2P Approach to Interorganizational Workflows* [22] werden Prozesse mit Hilfe von speziellen Petrinetzen – so genannten W*orkflow Nets* – beschrieben. Dabei wird zuerst ein *Public Workflow* definiert, der sämtliche Schritte eines interorganisationellen Workflows enthält. Anschließend wird dieser Prozess partitioniert, so dass die einzelnen Partitionen jeweils nur die Schritte eines der Beteiligten enthalten. Es gibt jetzt einen Kontrollfluss zwischen den einzelnen Partnern. Dann wird, wie in [13] dargelegt, eine Vererbungsbeziehung für Prozesse eingeführt. Ein Prozess kann ein „Subprozess" eines „Oberprozess" sein, wenn er das gleiche Verhalten wie der „Oberprozess" zeigt. Es werden Regeln definiert, wie zu einem Prozess neue Schritte hinzugefügt werden, so dass die Vererbungsbeziehung erhalten bleibt. Die einzelnen vorher definierten Partitionen, auch *Public Part* genannt, können jetzt mit den vorher definierten Regeln erweitert werden und es entstehen damit *Private Workflows*. Die *Private Workflows* können von den jeweiligen Beteiligten beliebig geändert werden, solange die Vererbungsbeziehungen erhalten bleiben. Jetzt ist der *Public Workflow*, der den Kontrollfluss zwischen den Beteiligten darstellt, zu einer View auf den *Overall Workflow* (Gesamtworkflow, der aus allen *Private Workflows* besteht) geworden. Zusätzlich ist eine weitere Viewbildung durch die Abbildung verschiedene Varianten eines Prozesses auf ein Petrinetz wie in [13] möglich. Im Prinzip handelt es sich bei dem in [22] vorgestellten Viewkonzept um eine Graphaggregation, die jedoch nicht für bestehende Prozesse ein oder mehrere Views erzeugt, sondern die View entsteht bei der Top-Down Modellierung „nebenbei". Es ist fraglich, ob diese Art der Prozessmodellierung in der Praxis realistisch ist, da Views oftmals für bereits bestehende Prozesse benötigt werden und nicht bereits beim Prozessentwurf interorganisationeller Prozesse berücksichtigt werden können.

Die Grundlage zu dem umfangreichsten Ansatz zu Prozess-Views liefert *Workflow Modeling for Virtual Processes: an Order-Preserving Process-View Approach* [9]. Als Basismodell dienen hier Aktivitätennetze (der Workflow Coalition). Views werden als abstrakte bzw. virtuelle Prozesse gesehen, die durch Graphaggregation von Aktivitäten bestehender Prozesse erzeugt werden. Für eine virtuelle Aktivität müssen folgende Regeln erfüllt sein:

- Eine virtuelle Aktivität besteht aus Basisaktivitäten oder aus anderen virtuellen Aktivitäten. Bei Schleifen darf eine virtuelle Aktivität entweder alle Basisaktivitäten der Schleife enthalten oder nur Teile davon. Sie darf nicht Teile innerhalb und Teile außerhalb der Schleife enthalten.
- Atomarität: eine virtuelle Aktivität wird als atomare Einheit ausgeführt. Eine virtuelle Aktivität gilt als begonnen, wenn einer ihrer enthaltenen Aktivitäten begonnen wurde und ist beendet, wenn ALLE enthaltenen Akivitäten beendet wurden. Der Start einer

virtuellen Aktivität bedeutet, das alle vorhergehenden virtuelle Aktivitäten beendet worden sind.

- Views müssen die Reihenfolge der Ausführung von Aktivitäten innerhalb des Basisprozesses erhalten.

Nach Definition virtueller Aktivitäten wird der Begriff der *Essential Activities* eingeführt, der den relevanten Kontenmengen des Graphaggregationsalgorithmus (siehe Abschnitt 5.3.5) entspricht, also eine Menge von Schritten bezeichnet, die für den Viewmodellierer in einer virtuellen Aktivität enthalten sein müssen. Mit Hilfe des Algorithmus *VAGenerator*[28] ist es dann möglich, eine minimale Menge von Aktivitäten und den dazugehörenden Abhängigkeiten zu bestimmen, die die angegebenen *Essential Activities* enthalten, und die den obigen drei Regeln genügt. Der Kontrollfluss zwischen den virtuellen Aktivitäten kann danach einfach durch Iterieren über alle Kontrollflusskanten des Basisprozesses bestimmt werden (es gibt eine virtuelle Abhängigkeit zwischen zwei virtuellen Aktivitäten, wenn es mindestens eine Abhängigkeit zwischen zwei Basisaktivitäten in den zwei virtuellen Aktivitäten gibt). Der Artikel enthält damit einen umfassenden Ansatz für die algorithmische Erzeugung von Views mit Hilfe von Graphaggregation.

In *Business-to-Business Workflow Interoperation Based on Process-Views* [5] wird der obige Ansatz um zusätzliche Aspekte erweitert. Dabei wird einerseits der Begriff des *State Mapping* eingeführt, mit dem es möglich ist, für eine virtuelle Aktivität auch einen Zustand anzugeben. Gleichzeitig ist es möglich, über diesen *Abstract State* auch eine Zustandspropagation auf den Basisprozess durchzuführen. Dabei wird versucht, die Änderung des abstrakten Zustands der virtuellen Aktivität so auf den Basisprozess abzubilden, dass wieder ein korrektes *State Mapping* entsteht. In [5] wird aber nicht erklärt, wie das genau funktionieren soll. Auch ist damit nicht möglich, alle Arten von Zustandsänderungen zu propagieren, da bei Zustandsänderungen der Basisaktivitäten der Zustand des virtuellen Schrittes trotzdem gleich bleiben (z.B. bei einer Sequenz von Schritten) kann. Zusätzlich wird noch ein *Data Mapping* eingeführt, mit dem Teile des Datenfluss verborgen (z.B. innerhalb einer virtuellen Aktivität) und Rechte für das Schreiben und Lesen von Daten vergeben werden können. Auch können Aggregationen und Transformationen auf Daten durchgeführt werden. Dann wird noch der Begriff des *Integrated Process* eingeführt, bei dem es um die Zusammenfassung mehrerer Prozesse in einem virtuellen Prozess für die Bildung einer Sicht auf interorganisationelle Workflows geht.

In *Discovering Role-Relevant Process-Views for Recommending Workflow Information* [11] wird zusätzlich noch ein Algorithmus vorgestellt, mit dessen Hilfe es möglich ist, automatisiert Views zu erzeugen. Die relevanten Knotenmengen für den *VAGenerator* müssen dabei nicht

[28] prüft im wesentlichen ob mit den Vorgängerknoten und Nachfolgerknoten der Menge der *Essential Activities* eine Verletzung der Ordnung verbunden ist, und nimmt diese Knoten dann in die virtuelle Aktivität auf

mehr angegeben werden, sondern werden abhängig von der Rolle, für die eine Sicht erzeugt werden soll, automatisch bestimmt. Dazu werden mit Hilfe eines Relevanzgrades (eine Aktivität hat für eine bestimmte Rolle ein hohe oder niedrige Relevanz) für die einzelnen Aktivitäten, vor der Durchführung des eigentlichen Algorithmus *VAGenerator,* die relevanten Knotenmengen bestimmt. Insgesamt liefern die obigen drei Artikel einen umfassenden Ansatz für die Erzeugung von Views mittels Graphaggregation. Nicht behandelt wird die Viewbildung mit Hilfe von Graphreduktion, welche Voraussetzungen und Möglichkeiten für die Änderung von Prozess-Schemata über Views bestehen und wie bei Schemaänderungen Views aktualisiert werden können.

In *The Use of Systemic Methodologies In Workflow Management Systems* [23] wird schließlich der Ansatz der obigen drei Artikel dazu verwendet Views zu generieren. Es geht in diesem Artikel lediglich darum, mit Hilfe verschiedener Methoden (*Problem Structuring Methodology, Metasystems Approach* und *Strategic Assumption Surfacing & Testing*), die relevanten Aktivitäten für eine Viewbildung zu bestimmen.

Zusammenfassend lässt sich sagen, dass in den vorhandenen Ansätzen die Notwendigkeit für Views auf Prozessen erkannt wird. Jedoch liegt der Fokus hierbei vor allem auf die View-Unterstützung bei interorganisationellen Prozessen [18, 19, 20, 21]. Es werden auch Methoden für eine Viewerzeugung mit Hilfe der Graphaggregation [9] und der Graphreduktion [21] in den verschiedenen Ansätzen vorgestellt. Was bisher jedoch nur teilweise behandelt wurde ist die Aktualisierung von Views nach Schema- oder Instanzänderungen – lediglich in [13] und [22] ist dies eingeschränkt möglich, wird dort jedoch nicht detailliert behandelt. Überhaupt noch nicht erkannt wurde die Bedeutung von Views zur Unterstüztung beim Prozess-Entwurf und für Ad-Hoc-Änderungen mit Hilfe vereinfachter abstrakter Prozessgraphen. Die für diese Anwendungsfälle notwendige Unterstützung von Schemaänderungen über Views wird in keinem der Artikel behandelt.

10 Zusammenfassung

Die steigende Bedeutung einer rechnergestützten Prozess-Verwaltung mit Hilfe von Prozess-Management-Systemen führt zu immer komplexeren Prozessen und der Notwendigkeit eine umfassenden Prozess-Unterstützung auch organisationsübergreifende Szenarien. Dadurch wird auch die Unterstützung von Views für verschiedene Anwendungsfälle immer wichtiger. Die Anwendungsfälle lassen sich in vier Kategorien einteilen:

- Vereinfachung komplexer Prozesse zwecks Definition von Prozessänderungen oder einfach nur für das Monitoring und die Überwachung von Prozessen. Dabei können für einen Benutzer unwichtige Teile ausgeblendet werden.
- Verbergen sicherheitsrelevanter Informationen vor bestimmten Benutzern. Dies ist sowohl innerhalb eines Unternehmens wichtig (nicht jeder Mitarbeiter darf sämtliche Prozessinternas sehen) als auch bei interorganisationellen Prozessen.
- Kapselung von Prozessen und damit verbunden die Möglichkeit, Prozesse ändern zu können ohne das von dem Prozess abhängige Prozesse oder Anwendungen mit angepasst werden müssen.
- Effizienzsteigerung durch die Reduzierung von Prozessgraphen. Damit ist z.B. in der Prozess-Engine eine effizientere Bearbeitung (z.B. in Verbindung mit dem Ein-/ Auslagern von Schema- und Instanzdaten) möglich.

Für die verschiedenen Anwendungsfälle werden verschiedene Operationen auf Views benötigt. Diese lassen sich in Operationen für den lesenden Zugriff auf Views, Operationen für Änderungen des Zustandes einer Prozess-Instanz und Operationen für die Änderung von Prozessgraphen unterscheiden.

Für die Unterstützung dieser verschiedenen Operationen müssen verschiedene Methoden zur View-Erzeugung unterschieden werden. Grundsätzlich unterscheiden wir die Methoden: Bildung von Views durch das Zusammenfassen mehrerer Aktivitäten zu einem virtuellen Schritt (Graphaggregation) und das Verbergen von Aktivitäten (Graphreduktion). Dabei kann eine Graphaggregation strukturerhaltend erfolgen, wodurch auch Schemaänderungen über eine View möglich werden. Neben den Aktivitäten können auch andere Graphelemente, wie Attribute oder Datenelemente, in die Viewbildung einbezogen, und damit z.B. bestimmte Datenelemente verborgen oder Attribute aggregiert werden.

Aufbauend auf dem ADEPT-Metamodell [4, 6] ist es mit Hilfe der zusätzlichen Attribute VISIBLE und VIRTUAL möglich, Views zu definieren. Dabei kann ein Knoten durch Setzen von VISIBLE=`FALSE` verborgen werden und ein in der View erzeugter virtueller Knoten kann durch VIRTUAL=`TRUE` von den Basisaktivitäten unterschieden werden. Mit diesen Attributen und zusätzlichen Änderungsprimitiven können jetzt Algorithmen für die Graphreduktion und Graphaggregation definiert werden. Neben den Algorithmen für die Viewbildung werden noch

zusätzliche Algorithmen für die automatische Generierung von Views und dem Zustandsmapping für virtuelle Knoten in Prozess-Instanzen formuliert.

Neben der Viewbildung ist auch die Fragestellung, wo die View-Erzeugung stattfindet, relevant. Views können sowohl im Server als auch im Client erzeugt werden. Beide Alternative haben ihre Vor- und Nachteile. Bei der Speicherung im Server können Views zeitnah aktualisiert werden, und es werden bereits kompaktere Graphen an die Clients gesendet. Nachteilig ist jedoch die erhöhte Rechenlast im Server, wenn viele Views verwaltet werden müssen. Bei der View-Erzeugung im Client können Views flexibel und unabhängig von den Fähigkeiten des Servers erzeugt werden. Nachteil sind die notwendige Leistungsfähigkeit des Clients sowie Entkopplung von Views und zugehörigem Basisprozess.

Auch für die Speicherung von Views gibt es zwei verschiedene Möglichkeiten: Speicherung als Sequenz von Anweisungen oder als Kopie des Prozesses (*Materialized View)*. Bei einer Speicherung als Kopie kann noch unterschieden werden, ob der komplette Graph oder lediglich die entsprechenden VISIBLE und VIRTUAL Attribute kopiert werden. Beide Verfahren haben Vor- und Nachteile bezüglich Speicherbedarf und der Durchführung von Aktualisierungen.

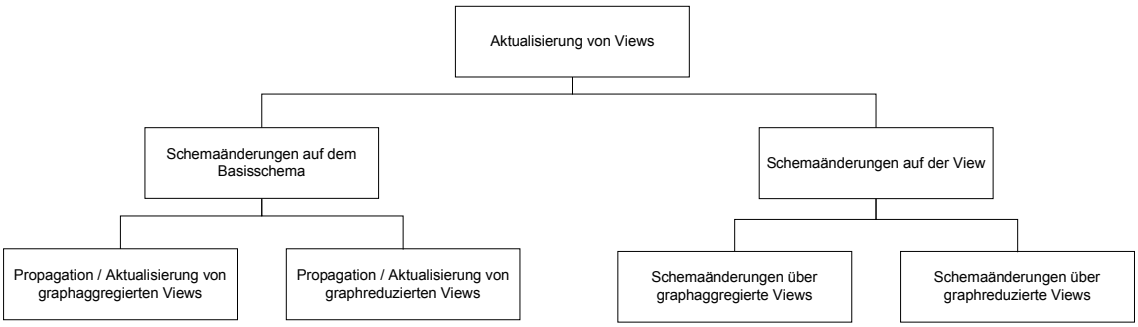

Abbildung 82: Verschiedene Varianten für die Aktualisierung von Views

Sowohl Prozess-Schemata als auch Prozess-Instanzen können sich ändern. Wenn auf den jeweiligen Prozessgraphen Views definiert sind, müssen diese entsprechend aktualisiert werden. Bei Instanzänderungen (Schritt weiterschalten, Datenelement schreiben) ist diese Änderung einfach durchzuführen, es muss lediglich der virtuelle Knoten der die betreffende Basisaktivität bzw. das betreffende Datenlement enthält, angepasst werden. Bei Schemaänderungen (Löschen, Einfügen und Verschieben von Aktivitäten) muss zwischen Views, die durch Aggregation und die per Reduktion entstanden sind, unterschieden werden. Die betroffenen sichtbaren Elemente in der View müssen dann identifiziert und für Änderungen entsprechend angepasst werden.

Auch über Views kann das Schema des zugrundeliegenden Basisprozess im Prinzip geändert werden. Für Views die durch Graphaggregation entstanden sind, ist dies relativ einfach möglich, da die Viewbildung strukturerhaltend erfolgt und deshalb auch die View einen

gültigen Prozessgraphen repräsentiert. Hier können für die View die gleichen Änderungsoperationen, die auch für den Basisprozess definiert sind, verwendet werden. Für graphreduzierte Views ist dies nicht möglich, da hier die Viewbildung nicht strukturerhaltend erfolgt. Dort können nur einfache Operationen für die View-Änderung definiert werden, die dann auf dem Basisprozess durch mehrere Basisoperationen umgesetzt werden. Dabei ergibt sich die Problematik, dass nach der Änderung des Basisschemas und anschließender Aktualisierung der View (siehe vorherigen Absatz), die View eine andere Struktur hat als dies ein Benutzer durch Ausführung der für die View definierten Änderungsoperation erwarten würde. In Abbildung 82 sind die verschiedenen Varianten für die Aktualisierung von Views noch einmal in einer Übersicht zusammengefasst.

Mit dieser Arbeit wurden erstmals sämtliche Aspekte, die für Prozess-Views relevant sind behandelt. Dabei wurden sowohl Anwendungsfälle als auch die Möglichkeiten zur Definition, Erzeugung, Verwaltung und Speicherung verschiedener View-Arten betrachtet.

Literaturverzeichnis

[1] Ramiz Elmasri, Shamkant B. Navathe: Fundamentals of Database Systems, Fourth Edition, Addison Wesley (2002)

[2] Frank Leymann, Dieter Roller: Production Workflow, Prentice Hall (2000)

[3] Wil van der Aalst, Kees Van Hee: Workflow Management: Models, Methods, and Systems (Cooperative Information Systems), The MIT Press (2002)

[4] Manfred Reichert: Dynamische Ablaufänderungen in Workflow-Management-Systemen, Dissertation, Universität Ulm (2000)

[5] Duen-Ren Liu, Minxin Shen: Business-to-Business Workflow Interoperation Based on Process-Views, Journal of Decision Support Systems 38(3): 399-419 (2004)

[6] Stefanie Rinderle, Manfred Reichert, Peter Dadam: Flexible Support Of Team Processes By Adaptive Workflow Systems, Distributed and Parallel Databases 16(1): 91-116 (2004)

[7] Alejandro Gutiérrez, Philippe Pucheral, Hermaun Steffen, Jean-Marc Thevenin: Database Graph Views : A Practical Model to Manage Persistent Graphs, Proceedings of the 20th VLDB Conference, Santiago, Chile: 391-402 (1994)

[8] Wasim Sadiq and Maria E. Orlowska: Analyzing Process Models Using Graph Reduction Techniques, Information Systems 25(2): 117-134 (2000)

[9] Duen-Ren Liu and Minxin Shen: Workflow Modeling for Virtual Processes: an Order-Preserving Process-View Approach, Information Systems 28(6): 505-532 (2003)

[10] Minxin Shen and Duen-Ren Liu: Coordinating Interorganizational Workflows based on Process-Views, Proceedings of the 12th International Conference on Database and Expert Systems Applications (DEXA'01), LNCS 2113, München: 274-283 (2001)

[11] Minxin Shen, Duen-Ren Liu: Discovering Role-Relevant Process-Views for Recommending Workflow Information, Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA'03), LNCS 2736, Prag: 836-845 (2003)

[12] W.M.P. van der Aalst: Inheritance of Interorganizational Workflows: How to agree or disagree without loosing control?, Information Technology and Management 4 (4), Trento, Italien: 345-389 (2003)

[13] W.M.P. van der Aalst: Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information, International Conference on Cooperative Information Systems: 115-126 (1999)

[14] Hui Wang, Maria Orlowska, Weifa Liang: Efficient Refreshment of Materialized Views With Multiple Sources, Proceedings of the eighth international conference on Information and knowledge management, Kansas City, Missouri: 375 - 382 (1999)

[15] Martin Staudtl, Christoph Quix and Manfred A. Jeusfeld: View Maintenance And Change Notification For Application Program Views, Proceedings of the 1998 ACM symposium on Applied Computing, Atlanta: 220 - 225 (1998)

[16] Ashish Gupta and Inderpal Singh Mumick: Maintenance of Materialized Views: Problems, Techniques and Applications, IEEE Data Engineering Bulletin 18(2) (1995)

[17] Stavros S. Cosmadakis, Christos H. Papadimitriou: Updates of Relational Views, Journal of the ACM (JACM) 31(4): 742-760 (1984)

[18] Dickson K.W. Chui, Shing-Chi Cheung, Kamalakar Karlapalem, Qing Li and Sven Till: Workflow View Driven Cross-Organizational Interoperability in a Web Service Environment, Ch. Bussler et al. (Eds.): WES 2002, LCNS 2512: 41-56 (2002)

[19] Eleanna Kafeza, Dickson K.W. Chiu and Irene Kafeza: View-based Contracts in an E-service Cross-Organizational Workflow Environment, Technologies for E-Services (TES), Rom: 74-88 (2001)

[20] Frank Lindert, Wolfgang Deiters: Modelling inter-organizational processes with process model fragments, GI Workshop Informatik 99, Paderborn (1999)

[21] Andreas Wombacher and Bendick Mahleko: Finding Trading Partners to Establish Ad-hoc Business Processes, DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California: 339-355 (2002)

[22] W.M.P. van der Aalst, M. Weske: The P2P Approach to Interorganizational Workflows, Proceedings of the 13th International Conference, CAiSE 2001, Interlaken, Schweiz: 140-156 (2001)

[23] Nikitas A. Assimakopoulos and Apostolos E. Lydakis: The Use Of Systemic Methodologies In Workflow Management, Proceedings of the 47th Annual Meeting of the International Society for the Systems Sciences Hersonissos, Kreta (2003)

Anhang: Graphaggregationsalgorithmus zu Abschnitt 5.3.5

AggregateGraph(CFS, DFS, NodeSubsets)

input:

CFS, DFS: WF-Graph, für den eine View gebildet werden soll

NodeSubsets: Menge, die Teilmengen von N enhält: $x \in NodeSubsets \Rightarrow x \subseteq N$

begin:

// Bildung von Blöcken für die jeweils gewählten Knoten

selectedBlocks := ∅; *//Menge an ausgewählten Blöcken*

selectedNodes := ∅; *// Menge an ausgewählten Knoten*

forall x in NodeSubsets **do**

(x_{start}, x_{end}) := MinBlock(CFS, x);

block := ∅;

forall n ∈ (succ_c*(x_{start}) ∩ pred_c*(x_{end}) ∪ {x_{start}} ∪ {x_{end}})

with VISIBLE(n)=`TRUE` **do**

block:= block ∪ {n};

selectedNodes:= selectedNodes ∪ {n};

done

selectedBlocks := selectedBlock ∪ {{block}};

done

// Blöcke die sich überlappen zusammenfassen

forall b_1 ∈ selectedBlocks, b_2 ∈ selectedBlocks **with** $b_1 \neq b_2$ **do**

if ($b_1 \cap b_2 \neq \emptyset$) **then**

b_{new}:= $b_1 \cup b_2$;

selectedBlocks:= selectedBlocks -{ b_1} -{ b_2};

selectedBlocks:= selectedBlocks ∪ {{b_{new}}};

endif

done

// Blöcke durch einen virtuellen Knoten ersetzen

forall b ∈ selected Blocks **do**

b_{start}:= Startknoten des Kontrollblocks b;

b_{end}:= Endknoten des Kontrollblocks b;

AggregateActivities(CFS, DFS, b_{start}, b_{end});

end

```
// Für die verbliebenen Knoten maximale Kontrollblöcke bilden
M:= enthält alle x ∈ N \ (selectedNodes ∪{n_start}∪{n_end}); VISBIBLE(x)=TRUE;
    // n_start und n_end sind die Start- und Endknoten von CFS
while (M ≠ ∅) do
   x:= beliebiges Element aus M;
   y:= UNDEFINIED;
   Q:= ∅;
   // Für alle direkten Nachfolger von x prüfen, ob der dadurch entstehende
   // Kontrollblock noch keine der bisherigen Knoten verwendet
   forall s in (c_succ(x) ∩ M) do enqueue(Q, s); done
   while (Q ≠ ∅) do
      q:= dequeue(Q);
      (x_start, x_end) = MinBlock(CFS, x, q);
      if ([succ_c*(x_start) ∩ pred_c*(x_end) ∪ {x_start} ∪ {x_end}] ∩ selectedNodes = ∅)
        then
          y:= x_end;
          forall s in (c_succ(y) ∩ M) do enqueue(Q, s); done
      endif
   done while (Q ≠ ∅)
   // Für alle direkten Vorgägner von x prüfen, ob der dadurch entstehende
   // Kontrollblock noch keine der bisherigen Knoten verwendet
   if (y = UNDEFINIED) then y:= x;
   forall p ∈ (c_pred(x) ∩ M) do enqueue(Q, p); done
   while (Q ≠ ∅) do
      q:= dequeue(Q);
      (x_start, x_end) = MinBlock(CFS, q, y);
      if ([succ_c*(x_start) ∩ pred_c*(x_end) ∪ {x_start} ∪ {x_end}] ∩ selectedNodes = ∅)
        then
          x:= x_start;
          forall p ∈ (c_pred(x) ∩ M) do enqueue(Q, p); done
      endif
   done while (Q ≠ ∅)
   // Für den ermittelten Kontrollblock einen virtuellen Knoten einfügen
   if (x ≠ y) then
      M:= M \ (succ_c*(x) ∩ pred_c*(y) ∪ {x} ∪ {y});
      selectedNodes:= selectedNodes ∪ (succ_c*(x) ∩ pred_c*(y) ∪ {x} ∪ {y});
      AggregateActivities(CFS, DFS, x, y);
   else
      M:= M \ {x}; selectedNodes:= selectedNodes ∪ {x};
      AggregateActivities(CFS, DFS, x, UNDEFINIED);
   endif
done while (M ≠ ∅)
```

```
      // Sämtliche Knoten des Basisprozesses wurden jetzt durch virtuelle Knoten
      // ersetzt (bis auf Start- und Endknoten von CFS)
      // Zusammenfassen von Split-Knoten mit direktem Vorgänger
      forall n∈N with VISIBLE(n) = TRUE
       and |c_succ_vis(n)| > 1 and |c_pred_vis(n)| =1 do
         predN:= direkter Vorgänger von n;
         if ( V_in^predN = V_out^predN = ONE_Of_ONE ) then
              // folgende Anweisungen wie bei AggregateActivities(...)
            SetNodeAttributeValue(CFS, n, a_start, ATTRIBUTE(predN,a_start));
            HideElements(CFS, DFS, {predN});
            // eingehende Kante von predN ausblenden und ersetzen
            forall e in E with Id_dest(e) = predN do
               HideElements(CFS, DFS, {e});
               e_new:= (Id_source(e), n, CONTROL_E);
               AddVirtualEdges(CFS, DFS, {e_new});
            done
         endif
      done
      // Zusammenfassen von Join-Knoten mit direktem Nachfolger
      forall n∈N with VISIBLE(n) = TRUE
       and |c_pred_vis(n)| > 1 and |c_succ_vis(n)| =1 do
         succN:= direkter Nachfolger von n;
         if ( V_in^succN = V_out^succN = ONE_Of_ONE ) then
            SetNodeAttributeValue(CFS, n, a_end, ATTRIUTE(succN,a_end));
            HideElements(CFS, DFS, {succN});
            // ausgehende Kante von succN ausblenden und ersetzen
            forall e in E with Id_source(e) = succN do
               HideElements(CFS, DFS, {e});
               e_new:= (n, Id_dest(e), CONTROL_E);
               AddVirtualEdges(CFS, DFS, {e_new});
            done
         endif
      done
  end
```

„Ich versichere, dass ich meine Diplomarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.
Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen."

Ulm, den 14.10.2004