ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Data Perspective-aware Process View Updates

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Bernd Mertesz
Bernd.Mertesz@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Jens Kolb

2012

„Data Perspective-aware Process View Updates“
Fassung vom 23. Oktober 2012

Satz: PDF-LaTeX 2ε

Zusammenfassung

Komplexe und unübersichtliche Geschäftsprozessen stellen Unternehmen heutzutage vor neue Probleme: Für den einzelnen Bearbeiter ist es schwer seinen Anteil im gesamten Geschäftsprozess zu erkennen. Prozesssichten lösen dieses Problem, indem sie Bearbeitern eine individuelle Sicht auf den Geschäftsprozess ermöglichen. Bearbeitern soll es aber ebenfalls möglich sein, auf Ausnahmesituationen zu reagieren und Änderung in den Geschäftsprozess einzubringen. Prozesssichten machen spezielle Änderungsoperationen notwendig, die die durchgeführte Änderung in der Prozesssicht des Bearbeiters in den Geschäftsprozess überträgt.

Diese Arbeit beschäftigt sich mit dem Erzeugen von Prozesssichten aus bestehenden Prozessschemata und stellt dabei neue datenbasierte Erzeugungsoperationen vor. Fortführend werden Änderungsoperationen für Prozesssichten vorgestellt, die die Änderungen auf das Prozessschema der Prozesssicht übertragen. Diese Änderungsoperationen werden dahingehend untersucht, ob sie die Datenfluss- oder Kontrollflusskorrektheit des Prozessschema zerstören können. Schließlich werden Migrationsregeln vorgestellt, die die Änderungen an einer Prozesssichten auf alle anderen Prozesssichten des gleichen Prozessschemas übertragen.

Die Ergebnisse der Untersuchung zeigen, dass es Fälle gibt in denen einige Änderungsoperationen die Datenflusskorrektheit zerstören. Man muss bei der Anwendung dieser Änderungsoperationen auf diese Fälle prüfen, um die Datenflusskorrektheit der Prozessschemata sicherzustellen. Mit dieser Prüfung ermöglichen die Änderungsoperationen einem Bearbeiter eine Prozesssicht zu verändern, ohne die Datenfluss- oder Kontrollflusskorrektheit des Prozessschemas zu zerstören.

Inhaltsverzeichnis

1 Einleitung

In Unternehmen gibt es eine Vielzahl von Geschäftsprozessen, an denen verschiedene Bearbeiter, Abteilungen und Partner beteiligt sind. Die Komplexität dieser einzelnen Geschäftsprozesse ist dabei hoch, weil sie aus vielen verschiedenen Aktivitäten bestehen und komplexe Kontrollstrukturen besitzen. Ein Prozessschema, das diese Geschäftsprozesse abbildet, wird häufig zu Dokumentationszwecken angefertigt. Dabei ergeben sich Prozessschemata, die gedruckt DIN A0 weit überschreiten (auch „Wandtapete“genannt). Darunter leidet die Übersichtlichkeit und für den einzelnen Bearbeiter ist es schwer seinen Anteil im gesamten Prozess zu erkennen. Das Prozessschema ist zusätzlich häufig eine sehr technische Darstellung und enthält Aktivitäten, die zum Beispiel automatisch auf Datenbanken zugreifen oder von anderen System ausgeführt werden. Für den einzelnen Bearbeiter im Prozess spielen solche Aktivitäten aber nur eine geringe Rolle und führen dazu, dass für ihn die Übersicht im Gesamtprozess verloren geht.
Die Lösung dieses Problems sind Prozesssichten. Prozesssichten entstehen aus Geschäftsprozessen, in dem mehrere Aktivitäten zu einer abstrakten Aktivität zusammengefasst oder einzelne Aktivitäten ausblendet werden. Der Geschäftsprozess an sich wird dabei nicht verändert. Prozesssichten ermöglichen es auch individuelle Anforderungen der verschiedenen Bearbeiter an die Visualisierung des Geschäftsprozesses zu berücksichtigen. Zum Beispiel muss sich ein Manager einen abstrakten Überblick über den Gesamtprozess verschaffen können, ohne dass für ihn die Details relevant sind. Mit Prozesssichten kann der Geschäftsprozess soweit zusammengefasst werden, dass auf einen Blick der Stand des Prozesses ersichtlich ist. Jeder Bearbeiter erhält dabei seine eigene Prozesssicht, damit er eine genaue Übersicht über seine zu erledigenden Aufgaben hat.
Ein Beispiel eines komplexen Prozesses, an dem viele unterschiedliche Bearbeiter beteiligt sind, ist zum Beispiel der Entwicklungsprozess eines Automobils: Fahrzeugkonstrukteure, Produktionsplaner, Einkaufsagenten, Marketingplaner, IT-Entwickler, Manager und viele weitere Personen sind daran beteiligt. Es genügt in diesem Fall nicht jedem dieser Bearbeiter ein und dieselbe Prozesssicht anzubieten, denn jeder Bearbeiter hat andere

Fragestellungen und benötigt dafür eine andere Perspektive. Zum Beispiel hat ein Marketingplaner wenig Interesse an den exakten Aktivitäten die zur Produktion nötig sind.
In [2] wird intensiv auf die Erzeugung von Prozesssichten und die Anpassung der Prozesssichten an die Bedürfnisse der Benutzer eingegangen. Moderne Unternehmen benötigen jedoch nicht nur auf die Bedürfnisse der Bearbeiter angepasste Prozesssichten, sondern die zugrundeliegenden Prozesse müssen höchst flexibel angepasst werden können, um auf Ausnahmen reagieren zu können. Damit ergibt sich ein neues Problem: Bearbeiter sollen Änderungen in den Prozess einbringen können, obwohl sie durch ihre eingeschränkte Prozesssicht gar nicht den Überblick über den gesamten Prozess haben können. So kann z.B. der Produktionsplaner, der in den Entwicklungsprozess eines Automobils involviert ist, den Produktionsablauf ändern. Diese Änderung führt er dabei auf seiner Prozesssicht durch. Sie muss aber auf den zugrundeliegenden Prozess übertragen werden, damit die Prozesssichten der anderen Bearbeiter ggf. aktualisiert werden können und sie dadurch diese Änderung erfahren.
In dieser Arbeit werden datenzentrierte Operationen vorgestellt, die es ermöglichen spezifische Prozesssichten aus einem Prozessschema zu erstellen. In [2] werden diese Operationen auf Aktivitäten angewandt und als Resultat kann es dabei vorkommen, dass ebenfalls Datenelemente ausgeblendet oder zusammengefasst werden. Die in dieser Arbeit vorgestellten Operationen werden direkt auf Datenelemente angewendet. Außerdem führt die Arbeit Änderungsoperationen, die auf einer Prozesssicht ausgeführt werden, ein. Die vorgestellten Änderungsoperationen beschränken sich auf die Manipulation von bestehenden Datenelementen sowie dem Einfügen von neuen Datenelementen. Zusätzlich wird bei der Ausführung von Änderungsoperationen die Korrektheit des Prozesses analysiert. Dafür werden Korrektheitslevel eingeführt, die Aufschluss geben, ob sich die Prozesssicht in dieser Form korrekt ausführen lässt.
Diese Arbeit wird im Rahmen des *proView-Projekt* durchgeführt [1, 3, 4, 5]. Die Arbeit ist folgendermaßen aufgebaut: Kapitel 2 legt die Grundlage für die weitere Arbeit. Dazu wird das Prozessschema sowie das Prozesssichtschema definiert und Operationen vorgestellt, die im weiteren Verlauf benutzt werden. Im Kapitel 3 werden Operationen vorgestellt, um aus einem Prozess eine Prozesssicht bezüglich seiner Datenelemente zu erzeugen. Dabei wird der Vorgang erst durch die Operation in Pseudocode beschrieben und danach anhand von Beispielen anschaulich dargestellt. In Kapitel 4 werden die Änderungsoperationen vorgestellt, die es ermöglichen bestehende Datenelemente des Prozesses zu manipulieren bzw. neue Datenelemente einzufügen. In Kapitel 5 wird das Korrektheitslevel eines Pro-

zesses definiert und die Änderungsoperationen dahingehend untersucht, ob sie das Korrektheitslevel eines Prozesses verändern. In Kapitel 6 werden die Ergebnisse nochmal zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

In diesem Kapitel werden wichtige Begriffe und formale Grundlagen eingeführt, die für die Betrachtung von Prozesssichten benötigt werden. Neben der formalen Definitionen eines Prozess und einer Prozesssicht, werden Hilfsoperationen vorgestellt, die für die Änderungsoperationen in Kapitel 4 benötigt werden.

2.1 Prozessschema und Prozesssicht

Das *Prozessschema* basiert auf [2, S.45] und ist um Datenelemente und Datenflusskanten erweitert (siehe Definition 1).Die Menge N beinhaltet alle Knoten. Die Menge D enthält dabei alle *Datenelemente*. Die Menge E beinhaltet alle Kanten und setzt sich aus der Menge der *Kontrollflusskanten* CE und der Menge der *Datenflusskanten* DE zusammen. In der Menge der Datenflusskanten kann nochmals zwischen *Lesekanten* und *Schreibkanten* unterschieden werden. EC ordnet jeder Kontrollflusskante optional eine Transitionsbedingung zu (z.B. bei ausgehenden Kanten eines $XORSplit$). NT ordnet jedem Knoten einen Knotentyp $NT(n)$ zu, dabei wird zwischen Aktivitäten und Strukturknoten unterschieden. ET ordnet jeder Kante einen Kantentyp zu.

Definition 1. *Ein Prozessschema ist ein Tupel P = (N,D,E,EC,NT,ET). Dabei ist*

- *N die Menge der Konten,*
- *D die Menge der Datenelemente,*
- *E =* $CE \cup DE$ *die Kantenmenge von* P*.* $CE \subseteq N \times N$ *bezeichnet die Menge der Kontrollflusskanten und* $DE \subseteq N \times D$ *die Menge der Datenflusskanten. Wobei* $dr = (d \in D, n \in N)$ *eine Lesekante ist und* $dw = (n \in N, d \in D)$ *eine Schreibkante.*
- $EC : E \rightarrow Conds \cup \{TRUE\}$ *ordnet Kontrollflusskanten optional eine Transitionsbedingung zu.*

- $NT : N \rightarrow NodeTypes$ *ordnet jedem Knoten* $n \in N$ *einen Knotentyp* $NT(n)$ *zu. Die Menge an Knotentypen in* N *ist gegeben durch* $NodeTypes = \{Activity, ANDSplit, ANDJoin, XORSplit, XORJoin, LoopSplit, LoopJoin\}$. *Dadurch kann* N *in die disjunkte Mengen aus Aktivitäten A* $(NT(n) = Activity$ *für* $n \in N)$ *und Strukturknoten S* $(NT(n) \neq Activity$ *für* $n \in N)$ *zerlegt werden, d.h.* $N = A \dot{\cup} S$.

- $ET : E \rightarrow EdgeTypes$ *ordnet jeder Kante einen Kantentyp zu, wobei* $EdgeTypes = \{ControlFlow, DataFlow\}$

$$ET(e) = \begin{cases} ControlFlow & e \in CE \\ DataFlow & e \in DE \end{cases}$$

Abbildung 2.1 zeigt ein Beispiel eines Prozessschemas und seine verschiedenen Bestandteile (siehe Definition 1).

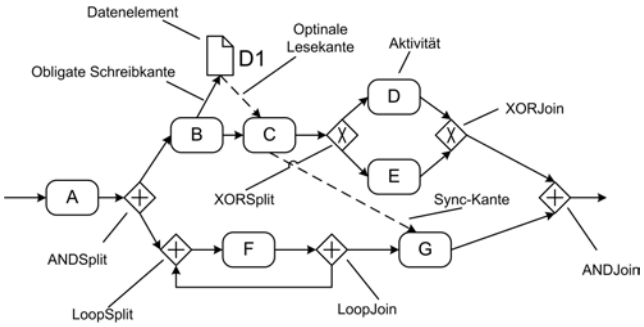

Abbildung 2.1: Beispiel eines Prozessschemas

Eine Prozesssicht ist eine Abstraktion eines bestehendes Prozessschema. Sie wird zur Visualisierung von großen Prozessschemata verwendet. Eine Prozesssicht basiert auf einem Prozessschema P. Durch Anwendung einer Reihe von *Prozesssichtsoperationen*, wie zum Beispiel der Reduktion einer Aktivität, wird die Prozesssicht erzeugt. Die *Prozesssichtsoperationen* werden durch die Menge OP repräsentiert (siehe Definition 2). Die *Prozesssichtsoperationen RedActivity*, *AggrSESE* und *AggrComplBranches* werden in [7] ausführlich definiert. In Kapitel 3 folgt eine Vorstellung anhand von Beispielen. Die *Prozesssichtsoperationen RedDataElement* und *AggrDataElements* hingegen werden in Kapitel 3 ausführlich definiert.

Definition 2. *Eine Prozesssicht V wird beschrieben durch die Erzeugungsmenge $CS_v = (P, OP, PS)$. Dabei ist*

- $P = (N, D, E, EC, NT, ET)$ *das Prozesschema, auf welchem die Prozesssicht basiert,*
- $OP = \{Op_1, ..., Op_n\}$ *die Menge an Prozesssichtsoperationen, welche auf P angewendet werden um die Prozesssicht: $V = \{Op_1 \circ ... \circ Op_n\}$ mit $Op_i \in \{RedActivity, AggrSESE, AggrComplBranches, RedDataElement, AggrDataElements\}$ zu erzeugen,*
- $PS = \{PS_1, ..., PS_m\}$ *die Menge an Parametern und dazugehörigen Parameterwerten , die für die spezifische Prozesssicht definiert sind.*

Abbildung 2.2 zeigt ein Beispiel einer Erzeugung einer Prozesssicht für den Prozess P. Dabei werden die Prozesssichtsoperationen $RedActivity(P, A)$, $AggrSESE(P, \{B, C, D\})$ und $RedDataElement(P, D2)$ verwendet.

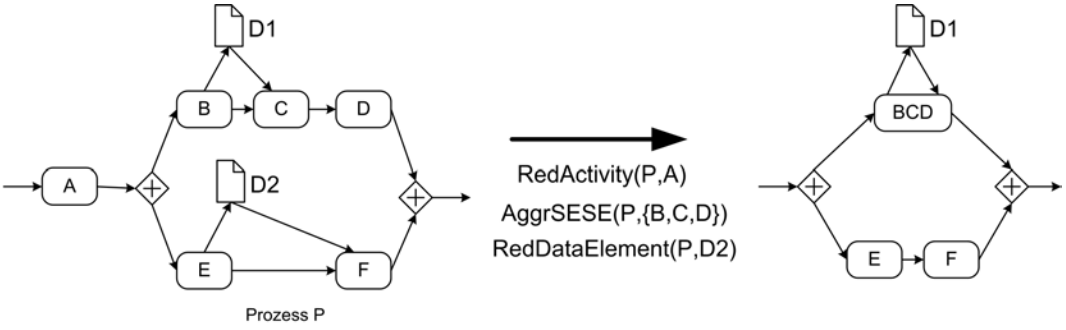

Abbildung 2.2: Erzeugung einer Prozesssicht

In diesem Abschnitt wurde das Prozessschema (siehe Definition 1) und die Prozesssicht (siehe Definition 2) definiert und vorgestellt. In Abbildung 2.1 wurden anhand eines Prozessschemas die verschiedenen Elemente eines Prozessschemas vorgestellt und in Abbildung 2.2 wurde gezeigt wie eine Prozesssicht aus einem Prozessschema entstehen kann. Das Prozessschema und die Prozesssicht bilden die Grundlage für alle Operationen in dieser Arbeit.

2.2 Hilfsoperationen

Im folgenden werden *Hilfsoperationen* definiert, auf die etwa in Änderungsoperationen zugegriffen wird: Die ersten beiden Funktionen beziehen sich auf das Prozessschema (siehe Definition 3).

Definition 3. *Die Operation CET ordnet jeder Kontrollflusskante in einem Prozessschema $P = (N, D, E, EC, NT, ET)$ einen entsprechenden Typ zu und die Operation DET beschreibt für jede Datenflusskante, welcher Typ von Datenzugriff durch sie repräsentiert wird. Der Typ von Datenzugriff wird durch ein Tupel (x, y) beschrieben. x gibt an, ob es um eine Lese- bzw. Schreibkante handelt. y gibt an, ob der Datenzugriff immer erfolgt (obligat), optional ist (optional) oder nie erfolgt (never). Sie sind definiert durch die Abbildungen*

- $CET : CE \rightarrow \{ControlFlowEdge, LoopEdge\}$
- $DET : DE \rightarrow \{r, w\} \rightarrow \{always, optional, never\}$

Die folgenden Operationen werden für die Änderungsoperationen benötigt (siehe Kapitel 4). Wenden wir die Operation *CPMDataElement* auf ein aggregiertes Datenelement d in der Prozesssicht V an, dann gibt diese Operation alle Datenelemente zurück, aus denen das aggregierte Datenelement besteht (siehe Definition 4).

Definition 4. *Die Operation CPMDataElement liefert für ein aggregiertes Datenelement d aus der Prozesssicht V alle Datenelemente zurück, aus denen das Datenelement d besteht. Die Prozesssicht V wird dabei durch die Erzeugungsmenge $CS_v = (P, OP, PS)$ (siehe Definition 2) beschrieben.*

$$CPMDataElement(d, V) = \begin{cases} d & d \in D \\ D_a & \exists Op_i : D_i \xrightarrow{Op_i} a \end{cases}$$

Die nächsten beiden Operationen beziehen sich auf die Datenflusskanten. Bei der Zusammenfassung von mehreren Datenelementen für eine Prozesssicht, werden dabei häufig auch die Datenflusskanten der Datenelemente zusammengefasst. Die erste Operation wird auf eine Lesekante in der Prozesssicht angewendet und liefert alle Lesekanten des Prozessschemas zurück, die in der Prozesssicht in dieser Lesekante zusammengefasst sind. Analog bezieht sich die zweite Operation auf Schreibkanten (siehe Definition 5).

Definition 5. *Die Operation CPMDataEdge liefert für eine Datenflusskante aus der Prozesssicht alle Datenflusskanten des Prozessschemas zurück, die in der Prozesssicht in dieser Lesekante zusammengefasst sind. Bei den Menge D_V und N_V handelt es sich um die Menge der Datenelemente bzw. Aktivitäten der Prozesssicht V.*

$$CPMDataEdge((d \in D_V, n \in N_V), V) =$$
$$\{e = (d', n') \in DE | n' \in CPMNode(n, V), d' \in CPMDataElement(d, V)\}$$

$$CPMDataEdge((n \in N_V, d \in D_V), V) =$$
$$\{e = (n', d') \in DE | n' \in CPMNode(n, V), d' \in CPMDataElement(d, V)\}$$

Die Prozessgraphen sind blockstrukturiert und lassen sich in *SESE*-Blöcke einteilen. Die Abkürzung *SESE* steht für *Single Entry Single Exit*. Ein *SESE*-Block ist ein Subgraph, der exakt einen Ein- und Ausgang besitzt. Abbildung 2.3 zeigt Beispiele für *SESE*-Blöcke in einem Prozessschema. Darin sehen wir eine weitere Eigenschaft von SESE-Blöcken: SESE-Blöcke können beliebig ineinander geschachtelt werden.

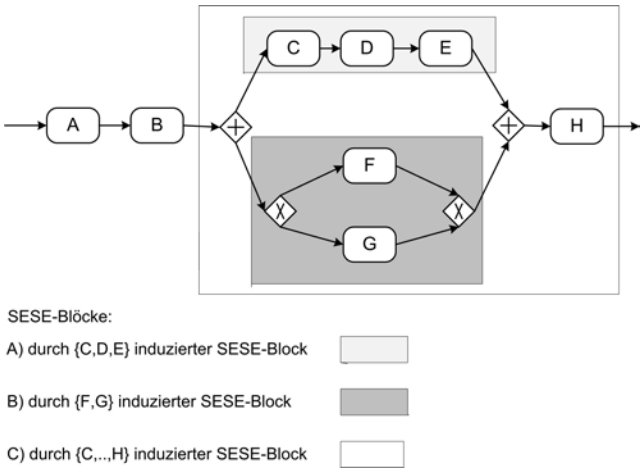

Abbildung 2.3: Beispiel für *SESE*-Blöcke

Die vorgestellten Hilfsoperationen werden für die Änderungsoperationen in Kapitel 4.1 und Erzeugungsoperationen in Kapitel 3.2 benötigt.

2.3 Prozesskorrektheit

Die Korrektheit einer Prozesssicht ermitteln wir mit Hilfe von Regeln. Diese Regeln teilen sich auf in Regeln für den Kontrollfluss (siehe [6, Seite 74ff]) und Regeln für den Datenfluss (siehe [6, Seite 121ff]). Kontrollflussregel *KF-1* fordert, dass es genau eine Start- und eine Endaktivität gibt. Die Startaktivität ist die einzige Aktivität im Prozessschema, die keinen direkten Vorgänger besitzt. Die Endaktivität ist die einzige Aktivität, die keinen direkten Nachfolger besitzt (siehe Abbildung 2.4). Kontrollflussregel *KF-2* verlangt, dass jede Aktivität abgesehen von der Start- und Endaktivität mindestens einen direkten Vorgänger und einen direkten Nachfolger besitzt (siehe Abbildung 2.4).

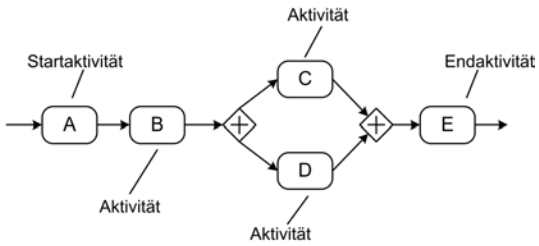

Abbildung 2.4: Beispiel für Kontrollflussregel *KF-1* und *KF-2*

KF-3 macht die Blockstruktur zur Bedingung. Das heißt, dass jede Aufsplittung des Kontrollflusses mit einem Split-Knoten $\in \{ANDSplit, XORSplit, LoopSplit\}$ wieder symmetrisch mit dem passenden Join-Knoten $\in \{ANDJoin, XORJoin, LoopJoin\}$ zusammengeführt werden muss.

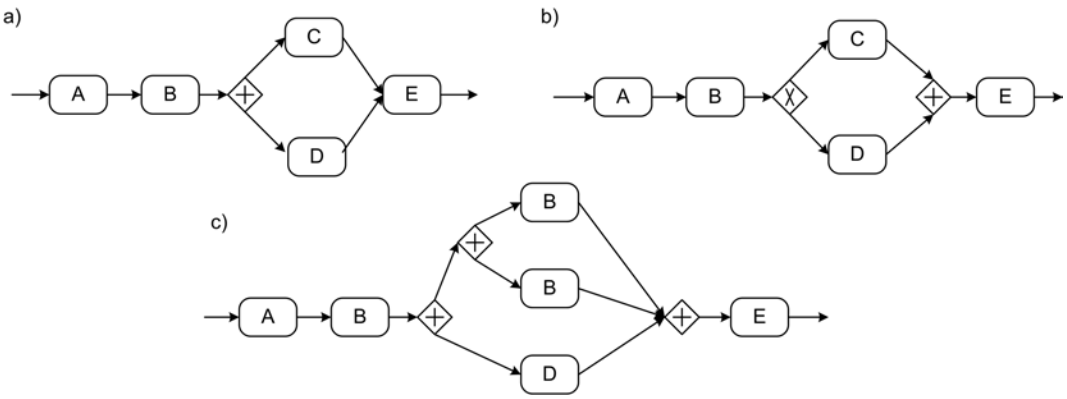

Abbildung 2.5: Beispiele für unzulässige Kontrollflussstrukturen

Abbildung 2.5 zeigt Beispiele von unzulässigen Kontrollflüssen. Der Kontrollfluss im Prozessschema in Abbildung 2.5 a) ist unzulässig, weil auf ein $ANDSplit$ kein $ANDJoin$ folgt. Es ist nicht zulässig, die beiden Verzweigungspfade in einer Aktivität enden zu lassen. In Abbildung 2.5 b) wird ein $XORSplit$ mit einem $ANDJoin$ abgeschlossen. Das ist ebenfalls unzulässig, weil jeder Split-Knoten mit seinem Pendant aus der Menge der Join-Knoten abgeschlossen werden muss. Es ist ebenfalls unzulässig mehrere $ANDSplit$ mit nur einem $ANDJoin$ abzuschließen (siehe Abbildung 2.5 c)). Für jeden Split-Knoten muss einen dazu passenden Join-Knoten geben.
KF-4 fordert Zyklenfreiheit bezüglich Kontrollfluss- und Sync-Kanten. D.h. Zyklen bzw. Schleifen dürfen nur mit $LoopSplit$-Aktivitäten und $LoopJoin$-Aktivitäten erstellt werden. Abbildung 2.6 a) zeigt ein Beispiel für eine unzulässige Schleife und Abbildung 2.6 b) für eine zulässige Schleife.

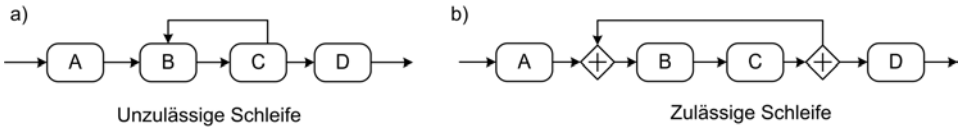

Abbildung 2.6: Beispiele für unzulässige und zulässige Schleifen

KF-5 stellt Bedingungen für die Verwendung von Sync-Kanten auf, um die korrekte Ausführung sicher zustellen. Die Quell- und Zielaktivität der Sync-Kante muss auf unterschiedlichen Verzweigungspfaden liegen. Die Sync-Kante von Aktivität B nach Aktivität D in Abbildung 2.7 ist zulässig, weil die Aktivitäten auf unterschiedlichen Verzweigungspfaden liegen. Die Sync-Kante von Aktivität G nach Aktivität E ist nicht zulässig, weil die Aktivitäten auf dem gleichen Verzweigungspfad liegen und es zu einer Verklemmung kommt. Die Aktivität E wartet mit ihre Ausführung bis die Aktivität G ausgeführt wurde. Da aber Aktivität G im Prozessschema hinter Aktivität E liegt wird das hier nie eintreten. Das Prozessschema kann nicht mehr ausgeführt werden.

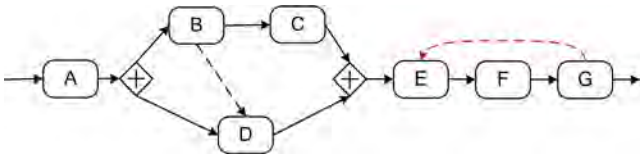

Abbildung 2.7: Beispiele für Sync-Kanten

KF-6 stellt Regeln für die Verwendung von strikten Sync-Kanten auf. Sie hat aber keinen Einfluss, weil im Prozessschema (siehe Definition 1) nicht zwischen strikten und einfachen Sync-Kanten unterschieden wird. Deswegen fließt *KF-6* nicht in die Bewertung der Prozesskorrektheit ein. *KF-7* stellt Regeln auf für die Verwendung von Schleifenknoten und -kanten. Eine Schleife besitzt einen Anfangsknoten $LoopSplit$ und einen Endknoten $LoopJoin$. Dabei ergibt sich wieder einen Blockstrukturierung wie in Kontrollflussregel *KF-3* gefordert. $LoopSplit$ ist über eine Kontrollflusskante mit $LoopJoin$ verbunden (siehe Abbildung 2.6 b)). *KF-8* fordert die Vermeidung von Zyklen in rekursiven Aufrufsbeziehungen. Da es im Prozessschema keine Subprozesse gibt, hat diese Regeln keinen Einfluss auf die Prozesskorrektheit.
Neben den Kontrollflussregeln gibt es drei Datenflussregeln: *DF-1* fordert die sichere Versorgung von Eingabeparametern. Ein Datenelement muss zwingend geschrieben werden, bevor es gelesen werden kann (siehe Abbildung 2.8).

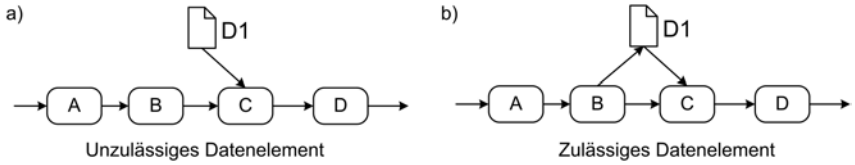

Abbildung 2.8: Beispiel für unzulässiges und zulässiges Datenelement

DF-2 verlangt die Vermeidung von parallelen Schreibzugriffen. In einer AND-Kontrollstruktur darf ein Datenelement nur auf einem Verzweigungspfad geschrieben werden. Dadurch steht für nachfolgende Aktivitäten immer eindeutig fest, welcher Wert des Datenelements gelesen werden soll. Abbildung 2.9 zeigt ein Beispiel eines parallelen Schreibzugriffs. Das Datenelement $D1$ wird von den Aktivitäten B und D parallel geschrieben. Es kann der Fall auftreten, dass Aktivität D das Datenelement $D1$ schreibt und bevor Aktivität E Datenelement $D1$ lesen kann, wird Datenelement $D1$ von Aktivität B überschrieben. Aktivität E ließt dann den falschen Wert.

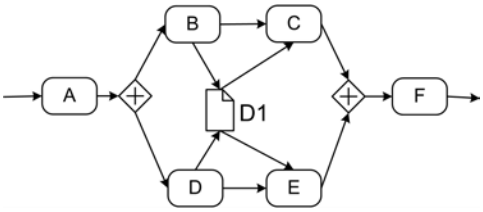

Abbildung 2.9: paralleler Schreibzugriff auf das Datenelement $D1$

DF-3 fordert schließlich die Vermeidung von direkt aufeinanderfolgenden Schreibzugriffen. Dieser Fall muss nicht unbedingt ausgeschlossen werden, jedoch sollte der Bearbeiter darauf hingewiesen werden. In Schleifen kann es vorkommen, dass auf einen Schreibvorgang direkt wieder ein Schreibvorgang folgt.
Die Kontroll- und Datenflussregeln ermöglichen es die Korrektheit von Prozesssichten anhand fester Kriterien festzustellen, dazu definieren wir mit Hilfe der Kontroll- und Datenflussregeln Korrektheitslevel (siehe Kapitel 5). Die Korrektheitslevel benötigen wir bei der Definition der Änderungsoperationen in Kapitel 4.1 und in Kapitel 5, wenn wir die Änderungsoperationen auf ihre Auswirkungen auf die Korrektheit der Prozesssichten untersuchen.

2.4 Das proView-Projekt

Wie in der Einleitung angemerkt, findet diese Arbeit im Rahmen des *proView*-Projekts statt. Die Idee und das Grundkonzept der Migration basiert auf dem *proView*-Projekt, welches in [4] näher beschrieben ist. Das Grundkonzept des *proView*-Projekt ist, dass es ein zentrales Prozessmodel, welches dem Prozessschema in Definition 1 entspricht, gibt. Alle Prozesssichten werden vom zentralen Prozessmodel abgeleitet. Eine Prozesssicht wird beschrieben durch eine Erzeugungsmenge $CS_v = (P, OP, PS)$, die aus einem Prozessschema P besteht, eine Menge an Prozesssichtoperationen OP und einer Menge an Parametern PS und dazugehörigen Parameterwerten, die für die spezifische Prozesssicht definiert sind. Eine Prozesssicht im *proView*-Projekt erstellen wir, indem wir zuerst die Prozesssichtoperationen OP auf das eigentliche Prozessschema anwenden, im nächsten Schritt die erhaltene Struktur vereinfachen und im letzten Schritt die Prozesssicht noch weiter anpassen. Es gibt unterschiedliche Erzeugungsmengen für unterschiedliche Prozesssichten. Ändert ein Bearbeiter etwas an seiner Prozesssicht, soll diese Änderung auch bei den Prozesssichten der anderen Bearbeiter durchgeführt werden. Dazu gehen wir in folgenden Schritten vor: Zuerst findet eine Änderung mit einer der Änderungsoperationen aus Kapitel 4.1 statt. Die Änderungsoperation führt zu einer Aktualisierung des zentralen Prozessmodells. Das Ergebnis vereinfachen wir, z.B. werden leere Verzweigungspfade entfernt. Im letzten Schritt migrieren wir die anderen assoziierten Prozesssichten, was durch die Aktualisierung der Menge der Prozesssichtenoperationen OP geschieht und damit die Prozesssicht auf den neuen Stand bringt.

Das Konzept des *proView*-Projekts spielt eine wichtige Rolle bei der Migration von Prozesssichten und Aktualisierung der Prozesssichten. Die Migration der anderen Prozesssichten der anderen Bearbeiter ist wichtig, damit es keine inkonsistenten Prozesssichten verschiedener Bearbeiter gibt und die Bearbeiter auf dem neusten Stand sind.

3 Operationen zur Prozesssichten-Erzeugung

Kapitel 3 stellt Operationen vor, die zur Erstellung von Prozesssichten genutzt werden. In Kapitel 3.1 werden die Operationen *RedActivity*, *AggrSESE* und *AggrComplBranches* vorgestellt [2, 7]. Kapitel 3.2 stellt die Operationen *RedDataElement* und *AggrDataElements* ausführlich vor.

3.1 RedActivity, AggrSESE und AggrComplBranches

Bei der Anwendung von *Reduktionsoperationen* auf Aktivitäten in einem Prozessschema, werden die entsprechenden Aktivitäten aus dem Prozessschema entfernt. Wenn die reduzierte Aktivität eine Datenflusskante zu ein Datenelement besitzt, wird diese ebenfalls entfernt. Anwendung findet die Reduktion, wenn nicht benötigte Details aus dem Prozessschema ausgeblendet werden sollen. Ein einfaches Beispiel für solch eine Reduktion zeigt Abbildung 3.1. Die Operation *RedActivity(C,V)* reduziert die Aktivität C aus dem Prozessschema CPM der Prozesssicht V und ersetzt diese sowie ihre eingehenden und ausgehenden Kontrollflusskanten durch eine neue Kontrollflusskante. Die Datenflusskante $E2$ wird dabei ebenfalls reduziert, weil sie mit der Aktivität C verbunden war.

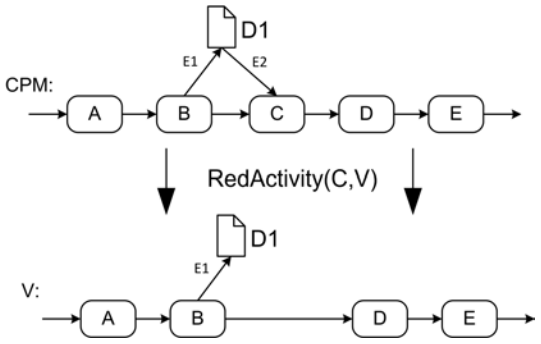

Abbildung 3.1: Reduktion der Aktivität C in einer Sequenz

Die Reduktion von komplexeren Prozessfragmenten geschieht durch wiederholtes Anwenden von der Operation *RedActivity*. Bei der Reduktion von den Aktivitäten B und C aus dem Prozessschema CPM in Abbildung 3.1 erfolgt das durch das Anwenden von *RedActivity(B,V)* und *RedActivity(C,V)* (siehe Abbildung 3.2).

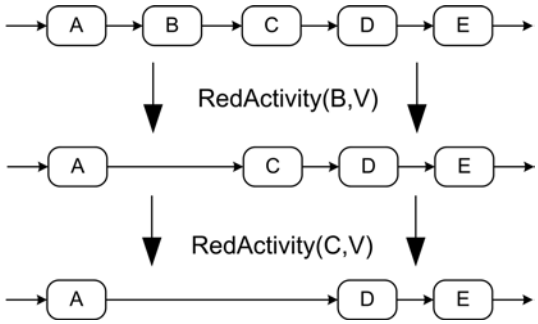

Abbildung 3.2: Reduktion mehrerer Aktivitäten

Ein weiteres Beispiel für die Reduktion eines komplexen Prozessfragmentes, ist die Reduktion in einer AND-Kontrollstruktur (siehe Abbildung 3.3). Mit Vereinfachungsoperationen (siehe [8]) kann die erhaltene Prozesssicht noch weiter vereinfacht werden.

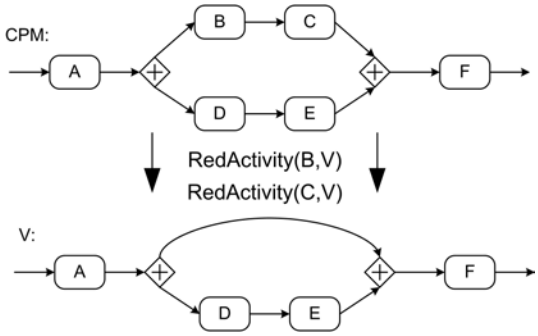

Abbildung 3.3: Reduktion in AND-Kontrollstruktur

Aggregationsoperationen fassen dagegen mehrere Aktivitäten zu einer neuen abstrakten Aktivität zusammen. Die Operation *AggrSESE(*N_a, V*)* fasst einen SESE-Block, bestehend aus den Aktivitäten in der Menge N, zu einer abstrakten Aktivität zusammen. Datenflusskanten die mit den Aktivitäten aus der Menge N verbunden sind, werden ebenfalls zusammengefasst. Lesen, zum Beispiel zwei Aktivitäten aus der Menge N das gleiche Datenelement d, so hat die neue abstrakte Aktivität nur noch eine Lesekante zum Datenelement d. In Abbildung 3.4 bilden die Aktivitäten B, C und D einen SESE-Block. Dieser SESE-Block

wird durch die Operation *AggrSESE($\{B, C, D\}, V$)* zu der abstrakten Aktivität N zusammengefasst. Die Aktivitäten B und C lesen beide das Datenelement $D1$, deshalb werden die beiden Datenflusskante $E2$ und $E3$ in Prozesssicht V zu einer Datenflusskante $E2E3$ zusammengefasst.

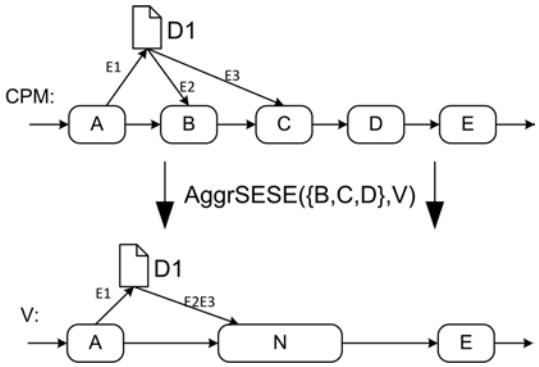

Abbildung 3.4: Aggregation eines SESE-Blocks

Die Operation *AggrComplBranches(N_a, V)* aggregiert Aktivitäten, die mehrere Verzweigungspfade umfassen zu einer abstrakten Aktivität. Hinsichtlich der Datenelementen verhält sich *AggrComplBranches(N_a, V)* wie die Operation *AggrSESE(N_a, V)*. In Abbildung 3.5 werden die Verzweigungspfade, die aus den Aktivitäten B und C, sowie den Aktivitäten D und F bestehen, zu einer neuen abstrakten Aktivität aggregiert. Das Ergebnis ist ein Verzweigungspfad, der die neue Aktivität $BCDF$ enthält.

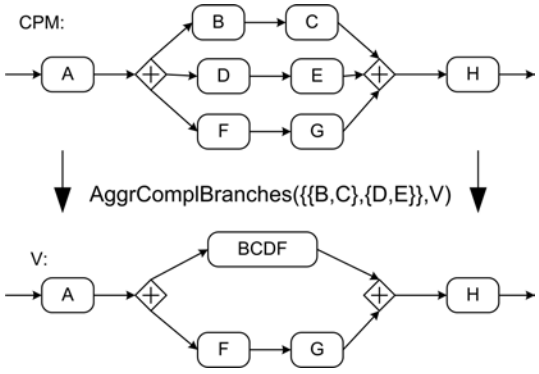

Abbildung 3.5: Aggregation zweier Verzweigungspfade

Die Operationen *RedActivity(C,V)*, *AggrSESE(N_a, V)* und *AggrComplBranches(N_a, V)* beziehen sich auf Aktivitäten in Prozessschemata und gehen nur im geringen Maße auf Da-

tenelement und Datenflusskanten in Prozessschemata ein. Ausführliche Definitionen der Operationen findet sich in [2, 7].

3.2 RedDataElement und AggrDataElements

Im Folgenden werden die Operationen *RedDataElement* und *AggrDataElements* vorgestellt. Im Gegensatz zu den Operationen aus Kapitel 3.1 werden diese nicht direkt auf Aktivitäten, sondern auf Datenelemente angewendet.
Bei *RedDataElement(x,P)* wird ein Datenelement x und alle mit ihm verbundenen Datenflusskanten aus dem Prozessschema P entfernt (siehe Algorithmus 1). Als Eingaben erhält der Algorithmus das Prozessschema P und das Datenelement x, das reduziert werden soll. Zeile 4 und 5 bestimmen die Mengen $E1$ und $E2$, die alle Lese- bzw. Schreibkanten enthalten, die mit x und einer beliebigen Aktivität in P verbunden sind. In Zeile 6 werden diese Kanten aus der Menge der Kanten des Prozesses entfernt und in Zeile 7 wird das zu löschende Datenelement aus der Menge der Datenelemente entfernt.

Algorithm 1 RedDataElement(x,P)

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $x \in D$: data element to be reduced

4: $E1 = \{de = (n, x) | n \in N\}$
5: $E2 = \{de = (x, n) | n \in N\}$
6: $E' = E \backslash (E1 \cup E2)$
7: $D' = D \backslash \{x\}$
8: **return** $V = (N, D', E', EC, NT, ET)$

Abbildung 3.6 zeigt an einem Beispiel die Anwendung des Operators *RedDataElement(x,P)*, in dem das Datenelement $D1$ aus dem Prozessschema CPM in der Prozesssicht V reduziert wird. Dabei werden die mit $D1$ verbundenen Datenflusskanten $E1$ und $E2$ ebenfalls reduziert.

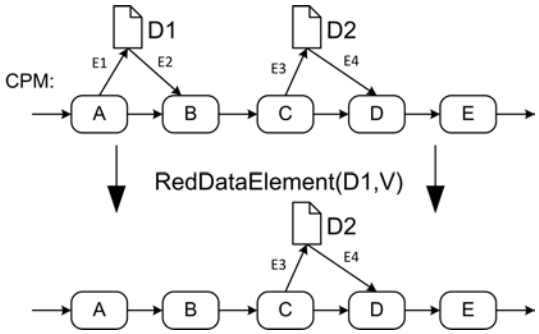

Abbildung 3.6: Reduzieren des Datenelements $D1$

Soll mehr als ein Datenelement entfernt werden, wird die Operation *RedDataElement* wiederholt angewendet und entspricht somit dem Vorgehen von *RedActivity* (siehe Kapitel 3.1).

Bei der Operation *AggrDataElements(S,P)* werden mehrere Datenelemente zu einem komplexen Datenelement (oder auch *Business Object* genannt) zusammengefasst (siehe Algorithmus 2). Als Eingaben erhält die Operation das Prozessschema P und die Menge S von Datenelementen, die zusammengefasst werden sollen. DEW beschreibt die Menge aller Schreibkanten, die eine Verbindung zwischen einem Datenelement aus der Menge S und einer Aktivität im Prozessschema P haben. Analog für die Menge DER, die die entsprechenden Lesekanten beinhaltet. Zu Beginn der Operation erstellen Zeile 6 und 7 das neue abstrakte Datenelement d_{new} und fügen es der Menge der Datenelemente D' hinzu. Zusätzlich entfernt Zeile 7 alle Datenelemente, die zusammengefasst werden, aus der Menge der Datenelemente D. In den Zeilen 8 bis 18 findet das Löschen der ursprünglichen Schreibkanten und das Setzen der neuen Schreibkanten zwischen d_{new} und entsprechenden Aktivitäten statt. Für jede ursprüngliche Schreibkante überprüft Zeile 9, ob die neue Schreibkante zum neuen komplexen Datenelement d_{new} schon gesetzt wurde. Wenn nicht, erstellt Zeile 15 die Schreibkante und daraufhin löscht Zeile 16 die ursprüngliche Kante. Wenn diese schon gesetzt wurde, wird zuerst überprüft, ob es sich bei der ursprünglichen Kante um eine obligate Kante handelt. Falls sie *obligat* ist, wird das Attribut der neuen Kante auf *obligat* gesetzt. Sobald in einer Menge von zusammenzufassenden Kanten eine *obligate* enthalten ist, ist die zusammengefasste Kante eine *obligate* Kante. Zum Schluss löscht Zeile 13 noch die alte Kante. Das gleiche Vorgehen wird bei Lesekanten angewendet.

Algorithm 2 AggrDataElements(S,P)

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: S : set of data elements, to aggregate
4: $DEW = \{de = (n, s) | s \in S, n \in N\}$ all writing edges, to aggregate
5: $DER = \{de = (s, n) | s \in S, n \in N\}$ all reading edges, to aggregate

6: create aggregated data element d_{new}
7: $D' := (D \backslash S) \cup \{d_{new}\}$
8: $E' := E$
9: **for all** $dew = (n, s) \in DEW$ **do**
10: **if** $\exists (n, d_{new}) \in E'$ **then**
11: **if** $DET(dew) = (w, obligat)$ **then**
12: $DET((n, d_{new})) := (w, obligat)$
13: **end if**
14: $E' = E' \backslash \{dew\}$
15: **else**
16: $E' = E' \cup \{(n, d_{new})\}$ with $DET((n, d_{new})) := DET(dew)$
17: $E' = E' \backslash \{dew\}$
18: **end if**
19: **end for**
20: **for all** $der = (s, n) \in DER$ **do**
21: **if** $\exists (d_{new}, n) \in E'$ **then**
22: **if** $DET(der) = (r, obligat)$ **then**
23: $DET((d_{new}, n)) := (r, obligat)$
24: **end if**
25: $E' = E' \backslash \{der\}$
26: **else**
27: $E' = E' \cup \{(d_{new}, n)\}$ with $DET((n, d_{new})) := DET(der)$
28: $E' = E' \backslash \{der\}$
29: **end if**
30: **end for**
31: **return** $V = (N, D', E', EC, NT, ET)$

Das Vorgehen zum Aggregieren von Datenelementen zeigt das Beispiel in Abbildung 3.7: In Abbildung 3.7 a) sollen die Datenelemente $D1$, $D2$ und $D3$ aus Prozessschema CPM mit Hilfe von *AggrDataElements(*$\{D1, D2, D3\}, V$*)* aggregiert werden. Zuerst werden die Datenelemente $D1$, $D2$ und $D3$ entfernt und das neue komplexe Datenelement $D1D2D3$ eingefügt (siehe Algorithmus 2, Zeilen 6-7). Die ursprünglichen Datenflusskanten bleiben vorerst unverändert (siehe Abbildung 3.7 b)). Dann beginnt man damit die Schreibkanten von $D1D2D3$ zu erstellen (siehe Algorithmus 2, Zeilen 8-18). Als erstes wird Kante $E1$ betrachtet: Es existiert noch keine neue Kante von Aktivität A nach Datenelement $D1D2D3$ (siehe Zeile 9). Deswegen wird die neue Kante $E1\_neu$ erstellt und Kante $E1$ gelöscht. Das Kantenattribut von $E1\_neu$ wird dabei von $E1$ übernommen (siehe Zeile 15-16, Abbildung 3.7 c)). Danach wird die Schreibkante $E3$ betrachtet. Da bereits eine Kante von Aktivität A nach Datenelement $D1D2D3$ existiert (siehe Algorithmus 2, Zeile 9), wird die Lesekante $E3$ entfernt (siehe Zeile 13). Da $E1\_neu$ bereits obligat ist, ändert sich das Kantenattribut nicht (siehe Zeile 10). Abbildung 3.7 d) zeigt das Ergebnis dieser Schritte. Als nächstes wird die Lesekante $E5$ betrachtet. Es existiert keine Kante von Aktivität C nach Datenelement $D1D2D3$ (siehe Algorithmus 2, Zeile 9), d.h. es wird die neue Kante $E2\_neu$ erstellt und $E5$ entfernt (siehe Zeile 15-16, Abbildung 3.7 e)). Nachdem alle Schreibkanten zusammengefasst sind, werden nun die Lesekanten betrachtet (siehe Zeilen 19-29): Es existiert noch keine Kante von Datenelement $D1D2D3$ nach Aktivität B (siehe Zeile 20). In den Zeilen 26-27 von Algorithmus 2 wird die Kante $E3\_neu$ erstellt und Kante $E2$ entfernt (siehe Abbildung 3.7 f)). Die nächste Kante ist $E4$: Es existiert noch keine Kante von Datenelement $D1D2D3$ nach Aktivität D (siehe Zeile 20), deswegen wird die Kante $E4\_neu$ erstellt und $E4$ entfernt. In Zeilen 26-27 im Algorithmus 2 erhält $E4\_neu$ das Kantenattribut von $E4$ (siehe Abbildung 3.7 g)). Nun verbleibt nur noch die Kante $E6$: Es existiert bereits eine Kante von Datenelement $D1D2D3$ nach Aktivität D (siehe Zeile 20). Sie ist jedoch optional, wohingegen $E6$ obligat ist(siehe Zeile 21). Deswegen wird das Attribut von $E4\_neu$ zu obligat geändert (siehe Zeile 22) und $E6$ anschließend gelöscht (siehe Zeile 24). Abbildung 3.7 h) zeigt abschließend die finale Prozesssicht V.

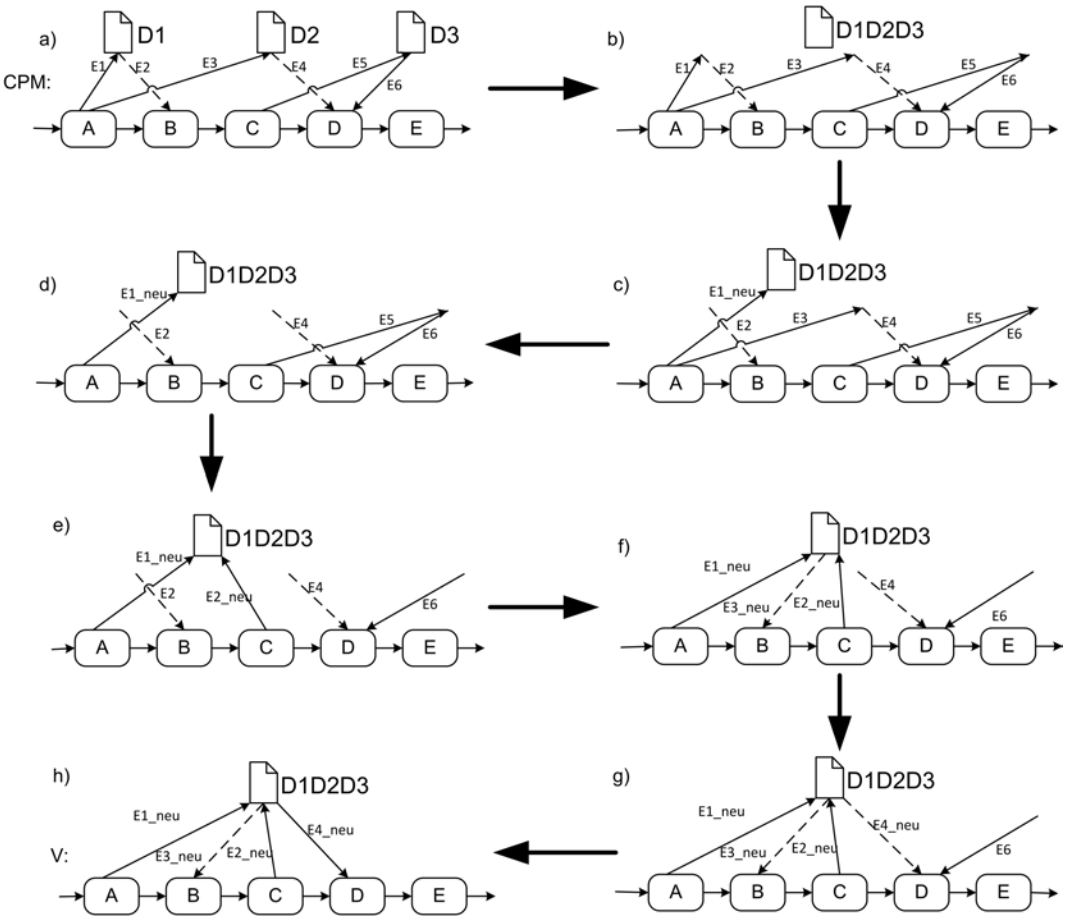

Abbildung 3.7: Beispiel für *AggrDataElements*

Die Anwendung der Operationen *RedDataElement* und *AggrDataElements* erstellen aus einem Prozessschema eine neue Prozesssicht. Im Gegensatz zu den Operationen aus Kapitel 3.1 werden sie direkt auf Datenelemente angewendet. Prozesssichten bilden die Grundlage der Änderungsoperationen aus Kapitel 4.

4 Aktualisierung von Prozesssichten

Kapitel 4.1 stellt die Operationen zur Änderung der Prozesssichten vor. Außerdem werden die Migrationsregeln vorgestellt (siehe Kapitel 4.2), die es ermöglichen Änderungen in andere Prozesssichten, die vom gleichen Prozessschema abstammen, zu übertragen. Diese Übertragung ist wichtig, damit andere Bearbeiter über Änderungen informiert werden und ihre Prozesssicht aktuell gehalten wird.

4.1 Änderungsoperationen

4.1.1 Löschen eines Datenelements: DeleteDataElement

Die Operation *DeleteDataElement(P,V,d)* löscht Datenelement d aus der Prozesssicht V, die vom Prozessschema P abstammt (siehe Algorithmus 3). Neben dem Datenelement d werden alle mit diesem verbundenen Lese- und Schreibkanten gelöscht. Ein Datenelement in der Prozesssicht V kann dabei ein komplexes Datenelement sein, d.h. es ist eine Aggregation von mehreren Datenelement aus dem Prozessschema P (siehe Kapitel 3.2). Bei der Aggregation von Datenelementen im Prozessschema werden häufig die entsprechenden Datenflusskanten ebenfalls aggregiert (siehe Kapitel 3.2). Sollte es sich beim Datenelement d um ein komplexes Datenelement handeln, werden bei der Anwendung von *DeleteDataElement(P,V,d)* alle Datenelemente und ihre verbundenen Datenflusskanten im Prozessschema P gelöscht, die zu d aggregiert wurden.
Als Eingabe erhält die Operation das Prozessschema P, die Prozesssicht V und das zu löschende Datenelement d. Die Menge DW enthält alle Schreibkanten von d in der Prozesssicht (siehe Algorithmus 3, Zeile 5); die Menge DR alle Lesekanten von d (siehe Zeile 6). Die Menge $D1$ besteht aus allen Datenelementen aus Prozessschema P, die möglicherweise im Datenelement d aggregiert sind (siehe Zeile 7), ansonsten nur aus d selbst. Die Menge $E1$ besteht aus allen Datenflusskanten im Prozessschema, die in einer Schreibkante von d aggregiert wurden (siehe Zeile 8). Analog für die Lesekanten in $E2$ (siehe Zeile 9). Zeile 10 entfernt alle Lese- und Schreibkanten schließlich aus der Menge der Kanten im Prozessschema und Zeile 11 die entsprechenden Datenelemente im Prozessschema P, die in d zusammengefasst wurden.

Algorithm 3 DeleteDataElement(P,V,d)

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $V = (N_V, D_V, E_V, EC_V, NT_V, ET_V)$
4: $d \in D_V$: data element to be deleted

5: $DW = \{dw = (n, d) | n \in N_V, d \in D_V\}$ all writing edges of d
6: $DR = \{dr = (d, n) | d \in D_V, n \in N_V\}$ all reading edges of d
7: $D1 = CPMDataElement(d, V)$
8: $E1 = \{de | de \in CPMDataEdge(dw, V) : \forall dw \in DW\}$
9: $E2 = \{de | de \in CPMDataEdge(dr, V) : \forall dr \in DR\}$
10: $E' = E \backslash (E1 \cup E2)$
11: $D' = D \backslash D1$
12: **return** $P' = (N, D', E', EC, NT, ET)$

Abbildung 4.1 zeigt das Beispiel einer Anwendung von *DeleteDataElement*. Abbildung 4.1 a) zeigt das Prozessschema *CPM*. Aus *CPM* wird mit Hilfe der Operationen *AggrDataElements(*$\{D1, D2\}, V$*)* und *AggrSESE(*$\{A, B, C, D, E\}, V$*)* die Prozesssicht V erstellt (siehe Abbildung 4.1 b)). Auf V wird die Operation *DeleteDataElement(*CPM*,*V*,*$D1D2$*)* angewendet. Das führt dazu, dass alle Datenelemente, die in *D1D2* in der Prozesssicht V zusammengefasst sind, aus Prozessschema CPM in Abbildung 4.1 a) gelöscht werden. Außerdem werden alle Kanten gelöscht, die durch Kanten $E1E3$ und $E2E4$ im Prozessschema CPM aus Abbildung 4.1 a) zusammengefasst wurden (siehe Abbildung 4.1 c)).

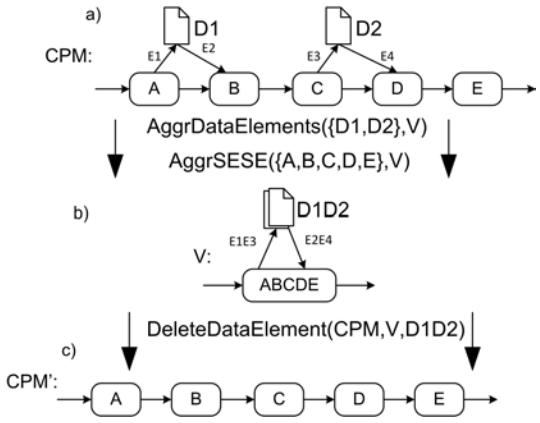

Abbildung 4.1: Prozessschema CPM und Prozesssicht V

4.1.2 Löschen einer Kante: DeleteEdge

DeleteEdge(P,V,e) löscht Datenflusskante e aus Prozesssicht V. Im Prozessschema P werden dazu alle korrespondieren Datenflusskanten gelöscht, die durch die Datenflusskanten e in der Prozesssicht V repräsentiert werden. Die Operation erhält als Eingabe Prozessschema P, eine auf P basierende Prozesssicht V und eine zu löschende Datenflusskante e. Zuerst wird in Zeile 5 im Algorithmus 4 die neue Kantenmenge E' definiert, die initial der ursprünglichen Kantenmenge E entspricht. Die Datenflusskante e ist mit dem Datenelement d in der Prozesssicht V verbunden. Die Menge D_{aggr} besteht aus allen Datenelementen des Prozessschema, die im Datenelement d in der Prozesssicht aggregiert sind (siehe Zeile 6). In Zeile 7 wird überprüft, ob man versucht die letzte Schreibkante eines Datenelements zu löschen, das aber noch an anderer Stelle gelesen wird. Denn nach Datenflussregel *DF-1* muss ein Datenelement, das von einer Aktivität obligat gelesen wird, von mindestens eine davor liegende Aktivität obligat geschrieben werden (siehe Kapitel 2.3). In diesem Fall wird das Korrektheitslevel der Prozesssicht reduziert und die Prozesssicht ist nicht mehr korrekt (siehe Abbildung 4.2 für ein Beispiel).

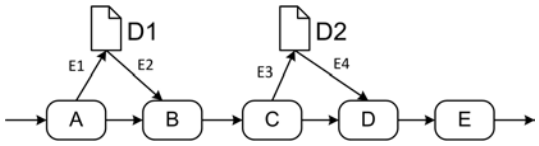

Abbildung 4.2: Fehler beim Löschen einer Schreibkante

Löscht man die Kante $E1$ bzw. $E3$ in Abbildung 4.2 und greift Aktivitäten B oder D auf das Datenelement zu, dann besitzen diese keinen validen Wert. Sollte die Prüfung in Zeile 5 jedoch negativ verlaufen, werden alle Kanten, die durch e repräsentiert werden in den Zeilen 10-12 aus der neuen Kantenmenge E' gelöscht.

Algorithm 4 DeleteEdge(P,V,e)

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $V = (N_V, D_V, E_V, EC_V, NT_V, ET_V)$
4: $e = (n \in N_V, d \in D_V) : e \in E_V$ Edge to be deleted

5: $E' := E$
6: $D_{aggr} := CPMDataElement(d, V)$

7: **if** $(DET_V(e) = (w, always) \wedge \exists dr = (d1 \in D_{aggr}, n1 \in N)) \wedge \nexists dw = (n2 \in N, d1 \in D_{aggr}) | n2 \neq n)$ **then**
8: *reduces correction level*
9: **else**
10: **for all** $de \in CPMDataEdge(e, V)$ **do**
11: $E' = E' \backslash \{de\}$
12: **end for**
13: **end if**
14: **return** $P' = (N, D, E', EC, NT, ET)$

Der Ablauf der Operation *DeleteEdge* wird im folgenden noch exemplarisch veranschaulicht. Dazu zeigt Abbildung 4.3 a) das Prozessschema CPM auf dessen Prozesssicht V die Operation *DeleteEdge* angewendet wird. Aus dem Prozessschema CPM aus Abbildung 4.3 a) wird mit den Operationen *AggrDataElements*($\{D1, D2\}, V$) und *AggrSESE*($\{A, B, C, D, E\}, V$) die Prozesssicht V erstellt (siehe Abbildung 4.3 b)). Nun wird auf die Schreibkante $E2E4$ in der Prozesssicht V die Operation *DeleteEdge(CPM,V,E2E4)* angewandt. Das hat zur Folge, dass alle die Datenflusskanten im Prozessschema CPM gelöscht werden, die durch die Schreibkante in der Prozesssicht repräsentiert wurden (siehe Abbildung 4.3 c))

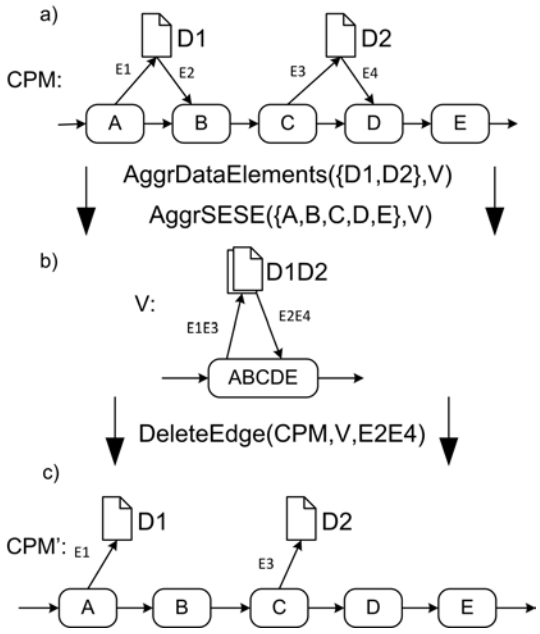

Abbildung 4.3: Anwendung von *DeleteEdge* bei Sequenz

Ein weiteres Beispiel zeigt die Anwendung von *DeleteEdge(*P,V,e*)* in Abbildung 4.4 bei einer AND-Kontrollstruktur. Mit Hilfe der Operationen *AggrDataElements(*$\{D1, D2\}$,V*)* und *AggrComplBrachnes(*$\{\{A, B, C\}, \{D, E\}\}$,$V$*)* wird aus dem Prozessschema CPM aus Abbildung 4.4 a) die Prozesssicht V in Abbildung 4.4 b) erstellt. Auf V wird die Operation *DeleteEdge(CPM,V,E2E4)* angewendet und damit die Lesekante $E2E4$ gelöscht. Dadurch werden die Datenflusskanten im Prozessschema CPM gelöscht, die in der Datenflusskante $E2E4$ in der Prozesssicht V aggregiert wurden (siehe Abbildung 4.4 c)). Das Löschen bei einer Prozesssicht, die eine XOR-Kontrollstruktur darstellt, verläuft analog zum Löschen bei einer AND-Kontrollstruktur.

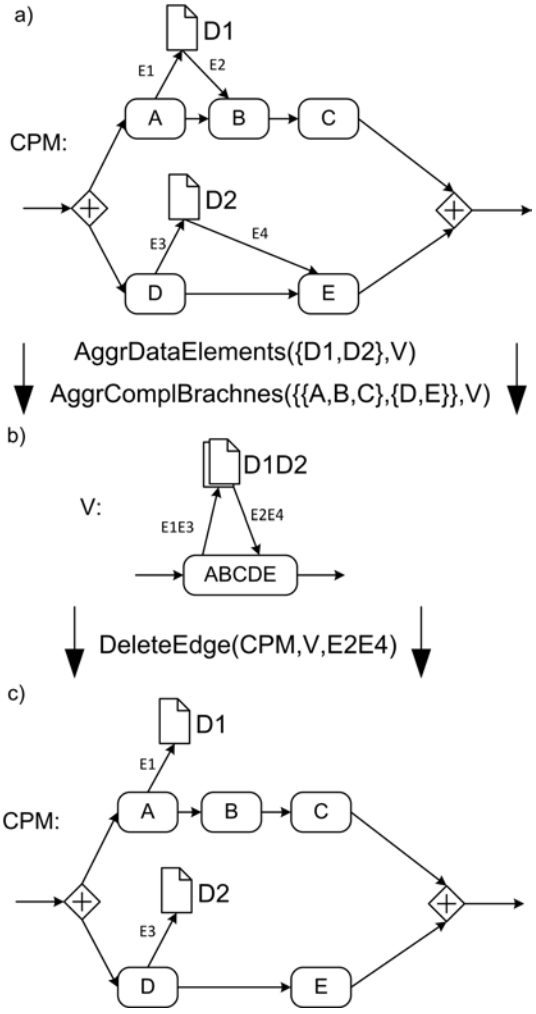

Abbildung 4.4: Anwendung von *DeleteEdge* bei einer AND-Kontrollstruktur

4.1.3 Ändern eines Kantenattributs: ChangeAttribute

Die Operation *ChangeAttribute(P,V,e,t_{new})* ändert das Attribut (z.b. von *obligat* auf *optional*) einer Datenflusskante e in der Prozesssicht V. Alle entsprechenden Datenkanten im Prozessschema P, die durch diese Datenkante e aus der Prozesssicht V repräsentiert werden, erhalten ebenfalls das neue Attribut t_{new}. Die Operation hat als Eingabe ein Prozessschema P, eine auf P basierende Prozesssicht V, die Datenflusskante e deren Attribute geändert werden soll sowie das neue Kantenattribut t_{new}. Die Datenflusskante e ist in der Prozesssicht mit der Aktivität n und dem Datenelement d verbunden. Zu Beginn wird in Algorithmus 5 in Zeile 6 die Menge D_{aggr} ermittelt, die alle Datenelemente aus dem Prozessschema P, welche im Datenelement d aggregiert sind enthält. In Zeile 7 im Algorithmus 5 wird überprüft, ob es sich bei der zu ändernden Datenkante um eine obligate Schreibkante handelt, deren Kantenattribut auf optional gesetzt werden soll und ob das geschriebene Datenelement noch im weiteren Prozessverlauf obligat gelesen wird. Dabei kann es wieder zu einem unterversorgten Datenelement kommen, was laut Datenflussregel *DF-1* nicht vorkommen darf und das Korrektheitslevel reduziert (siehe Kapitel 2.3). Sollte die Prüfung in Zeile 7 im Algorithmus 5 aber negativ verlaufen, wird in den Zeilen 10-12 das Kantenattribut von allen Kanten im Prozessschema P, die durch die Kante e in der Prozesssicht repräsentiert wird, in t_{new} geändert.

Algorithm 5 ChangeAttribute(P,V,e,t_{new})

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $V = (N_V, D_V, E_V, EC_V, NT_V, ET_V)$
4: $e = (n \in N_V, d \in D_V) : e \in E_V$ edge to be changed
5: $t_{new} \in \{obligat, optional\}$ new edge attribute

6: $D_{aggr} := CPMDataElement(d, V)$
7: **if** $DET_V(e) = (w, obligat) \wedge t_{new} = optional \wedge \exists dr \in E : dr = (d1 \in D_{aggr}, n1 \in N) \in$
$\wedge DET(dr) = (r, obligat)$ **then**
8: *reduces correction level*
9: **else**
10: **for all** $de \in CPMDataEdge(e, V)$ **do**
11: $DET(de) := (*, t_{new})$
12: **end for**
13: **end if**

Im Folgenden wird die Operation *ChangeAttribute(P,V,e,t_{new})* exemplarisch vorgestellt. In Abbildung 4.5 wird der Fehlerfall vorgestellt auf den Algorithmus 5 in Zeile 7 prüft. Mit Hilfe der Operationen *AggrDataElements($\{D1, D2\}$,V)* und *AggrSESE($\{A, B, C, D, E\}$,V)* wird aus dem Prozessschema CPM aus Abbildung 4.5 a) die Prozesssicht V erstellt (siehe Abbildung 4.5 b)). Nun wird mit der Operation *ChangeAttribute(CPM,V,E1E3,optional)* das Attribut der Schreibkante $E1E3$ von *obligat* zu *optional* geändert. Dadurch werden die Kantenattribute von den Datenflusskante $E1$ und $E3$ im Prozessschema CPM zu *optional* geändert. Der Datenfluss im Prozessschema in Abbildung 4.5 c) ist damit nicht mehr korrekt. Die Datenelemente $D1$ und $D2$ werden nur optional mit Daten versorgt, aber immer gelesen.

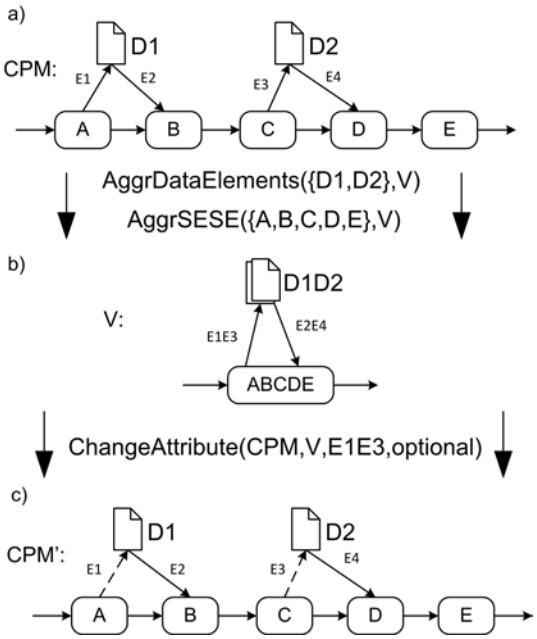

Abbildung 4.5: Beispiel für den Fehlerfall in Algorithmus 5

4.1.4 Hinzufügen einer Datenflusskante: AddEdge

Die Operation *AddEdge(P,V,e,AddEdgeMode)* fügt eine Datenflusskante in Prozesssicht V anhand des Parameters *AddEdgeMode* im Prozessschema P ein. Der Parameter wird benötigt, um aus den vielen Möglichkeiten die Datenflusskante in Prozessschema P einzufügen eine auszuwählen.

Die Operation hat als Eingabe das Prozessschema P, die auf P basierende Prozesssicht V, die einzufügende Kante e sowie den Parameterwert für *AddEdgeMode*, was den Modus des Einfügens in das eigentliche Prozessschema festlegt. *AddEdgeMode* besteht aus dem Modus zum Einfügen einer Lesekante, gefolgt vom Modus zum Einfügen einer Schreibkanten.
Zuerst wird in Algorithmus 6 überprüft, ob es sich bei der einzufügenden Datenkante um eine Lese- oder Schreibkante handelt. Eine Lesekante wird in Algorithmus 6 in den Zeilen 7-49 behandelt und eine Schreibkante in den Zeilen 50-82.
Bei einer Lesekante wird in Zeile 8 in Algorithmus 6 die Menge der Aktivitäten N_{CPM} des Prozessschemas bestimmt, die in der lesenden Aktivität n_v der Prozesssicht aggregiert sind. Danach wird in Zeile 9 jedes Datenelements d im Prozessschema P bestimmt, das im gelesenen Datenelement d_v der Prozesssicht aggregiert ist. Jedes dieser Datenelemente d wird einzeln in der Schleife betrachtet (siehe Zeilen 9-48). In Zeile 10 wird mit Hilfe der Funktion $CPMFirstWritingEdge(P, d)$ die erste Schreibkante des Datenelement d im Prozessschema P bestimmt. Dazu durchläuft man das Prozessschema von der Startaktivität zur Endaktivität bis man die erste Aktivität gefunden hat, die d schreibt. Anschließend wird unterschieden welcher Modus des Einfügens in *AddEdgeMode* gewählt wurde. Bei $AddEdgeMode = EARLY\_*$ in Algorithmus 6, Zeile 11 wird mit der Hilfsfunktion $first(N_{CPM})$, die erste Aktivität n_{first} bezogen auf den Kontrollfluss aus der Menge der Aktivitäten N_{CPM} bestimmt. Sollte die erste schreibende Aktivität n_1 innerhalb der Menge N_{CPM} liegen, wird n_{first} neu gesetzt. Dann ist n_{first} der direkte Nachfolger von n_1, der mit der Hilfsfunktion $succ(n_1)$ bestimmt wird. Mit dieser Überprüfung in Zeile 13-15 wird verhindert, dass die Lesekante vor der ersten Schreibkante gesetzt wird und damit die Datenflusskorrektheit zerstört. Handelt es sich bei n_{first} um eine Aktivität wird die Lesekante gesetzt (Zeile 21). Handelt es sich bei n_{first} um keine Aktivität, z.B. um ein AND-Gateway, wird die Lesekante an alle Nachfolger dieses Knotens gesetzt (Zeile 17-19).
Bei $AddEdgeMode = LATE\_*$ in den Zeilen 24-35, wird die letzte Aktivität bezogen auf den Kontrollfluss in der Menge N_{CPM} mit der Hilfsfunktion $last(N_{CPM})$ bestimmt. Handelt es dabei um eine Aktivität wird die Lesekante gesetzt. Handelt es sich um keine Aktivität (z.B. um einen $ANDJoin$) werden mit der Hilfsfunktion $pre(n_{last})$ alle Vorgänger von n_{last} bestimmt. Die Hilfsfunktion $CPMAreSamePath(n_2, n_1)$ überprüft für jeden Vorgänger n_2, ob er auf dem gleichen Verzweigungspfad ist wie n_1. Nur dann wird die Lesekante eingefügt. Würde man die Kante auf einem anderen Verzweigungspfad einfügen, kann es bei einer AND-Kontrollstruktur vorkommen, dass versucht wird das Datenelement zu lesen,

bevor es geschrieben wurde.
Bei $AddEdgeMode = ALL\_*$ in Zeilen 36-47 wird unterschieden, ob die erste schreibende Aktivität innerhalb von N_{CPM} liegt oder nicht. Falls nicht, wird für jede Aktivität innerhalb von N_{CPM} eine Schreibkante gesetzt (siehe Zeilen 43-45). Im anderen Fall werden für alle Aktivitäten nach der ersten schreibenden Aktivität n_1 Schreibkante gesetzt (siehe Zeilen 38-41).
Bei einer Schreibkante wird die Menge der Aktivitäten N_{CPM} aus dem Prozessschema bestimmt, die in der schreibenden Aktivität n_v aus der Prozesssicht aggregiert wurden (siehe Zeile 51). Wie bei der Lesekante wird jedes Datenelement d, welches in d_v aggregiert wurde einzeln betrachtet. Mit der Hilfsfunktion $CPMFirstReadingEdge(P, d)$ wird die erste Lesekante des Datenelements d im Prozessschema P bestimmt. Danach wird unterschieden welcher Modus in *AddEdgeMode* gewählt wurde.
Bei $AddEdgeMode = *\_EARLY$ in den Zeilen 54-65 wird mit Hilfe der Funktion $first($ $N_{CPM})$ die erste Aktivität n_{first} aus der Menge N_{CPM} bestimmt. Eine Besonderheit bildet beim Einfügen einer Schreibkante, die AND-Kontrollstruktur (siehe Zeile 56). Diese Operation kann das Korrektheitslevel reduzieren, weil beim Einfügen die Prozesskorrektheit nicht sichergestellt werden kann, wenn die Schreibkante in allen Verzweigungspfaden eingefügt wird. Darum wird keine Schreibkante zu Beginn oder an Ende einer AND-Kontrollstruktur eingefügt. Auf diesen Fall wird noch in einem Beispiel eingegangen (siehe Abbildung 4.12 und 4.13). Wenn es sich um keine AND-Kontrollstruktur handelt, wird die Schreibkante analog zu den vorherigen Fällen eingefügt.
Bei $AddEdgeMode = *\_LATE$ in den Zeilen 66-80 wird zuerst überprüft, ob die erste lesende Aktivität n_1 Teil der Menge N_{CPM} ist. Sollte dem so sein, wird n_{last} auf den Nachfolger dieser Aktivität gesetzt. Danach wird die Schreibkante gesetzt (siehe Zeilen 72-79).

Algorithm 6 AddEdge(P,V,e,*AddEdgeMode*)

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $V = (N_V, D_V, E_V, EC_V, NT_V, ET_V)$
4: $e \in E_V$ edge to be added
5: AddEdgeMode $\in \{EARLY\_EARLY, LATE\_LATE, LATE\_EARLY, ALL\_EARLY\}$

6: $E' := E$
7: **if** DET(e)=(r,*) **then**
8: $N_{CPM} = CPMNode(V, n_v)$ with $e = (d_v, n_v)$

```
9:  for all d ∈ CPMDataElement(d_v, V) do
10:   (n_1, d_1) = CPMFirstWritingEdge(P, d)
11:   if AddEdgeMode=EARLY_* then
12:     n_first = first(N_CPM)
13:     if n_1 ∈ N_CPM then
14:       n_first = succ(n_1)
15:     end if
16:     if NT(n_first) ≠ Activity then
17:       for all n_2 ∈ succ(n_first) do
18:         E' = E' ∪ {(d, n_2)}
19:       end for
20:     else
21:       E' = E' ∪ {(d, n_first)}
22:     end if
23:   end if
24:   if AddEdgeMode=LATE_* then
25:     n_last = last(N_CPM)
26:     if NT(n_last) ≠ Activity then
27:       for all n_2 ∈ pre(n_last) do
28:         if CPMAreSamePath(n_2, n_1) then
29:           E' = E' ∪ {(d, n_2)}
30:         end if
31:       end for
32:     else
33:       E' = E' ∪ {(d, n_last)}
34:     end if
35:   end if
36:   if AddEdgeMode=ALL_* then
37:     if n_1 ∈ N_CPM then
38:       while succ(n_1) ∈ N_CPM do
39:         E' = E' ∪ {(d, succ(n_1))}
40:         n_1 = succ(n_1)
41:       end while
42:     else
```

43: **for all** $n_3 \in N_{CPM}$ **do**
44: $E' = E' \cup \{(d, n_3)\}$
45: **end for**
46: **end if**
47: **end if**
48: **end for**
49: **end if**
50: **if** DET(e)=(w,*) **then**
51: $N_{CPM} = CPMNode(V, n_v)$ with $e = (n_v, d_v)$
52: **for all** $d \in CPMDataElement(d_v, V)$ **do**
53: $(d_1, n_1) = CPMFirstReadingEdge(P, d)$
54: **if** AddEdgeMode=*_EARLY **then**
55: $n_{first} = first(N_{CPM})$
56: **if** $NT(n_{first}) \neq ANDSplit$ **then**
57: **if** $NT(n_{first}) \neq Activity$ **then**
58: **for all** $n_2 \in succ(n_{first})$ **do**
59: $E' = E' \cup \{(n_2, d)\}$
60: **end for**
61: **else**
62: $E' = E' \cup \{(n_{first}, d)\}$
63: **end if**
64: **end if**
65: **end if**
66: **if** AddEdgeMode=*_LATE **then**
67: $n_{last} = last(N_{CPM})$
68: **if** $n_1 \in N_{CPM}$ **then**
69: $n_{last} = pre(n_1)$
70: **end if**
71: **if** $NT(n_{last}) \neq ANDJoin$ **then**
72: **if** $NT(n_{last}) \neq Activity$ **then**
73: **for all** $n_2 \in pre(n_{last})$ **do**
74: $E' = E' \cup \{(n_2, d)\}$
75: **end for**
76: **else**

77: $E' = E' \cup \{(n_{last}, d)\}$
78: **end if**
79: **end if**
80: **end if**
81: **end for**
82: **end if**
83: **return** $P' = (N, D, E', EC, NT, ET)$

Nun werden Beispiele vorgestellt, wie sich die verschiedenen Parameterwerte von *AddEdgeMode* auf die verschiedenen Kontrollstrukturen auswirken. Danach wird noch ein Beispiel für den Ablauf des Algorithmus 6 gegeben. Abbildung 4.6 zeigt das Prozessschema CPM und die Prozesssicht V, die aus dem Prozessschema CPM mit Hilfe der Operation *AggrSESE(*$\{A, B, C, D, E, F\}$*,V)* erstellt wird. In der Prozesssicht V wird die Lesekante $E2$ mit der Operation AddEdge(CPM,V,$E2$,*AddEdgeMode*) eingefügt.

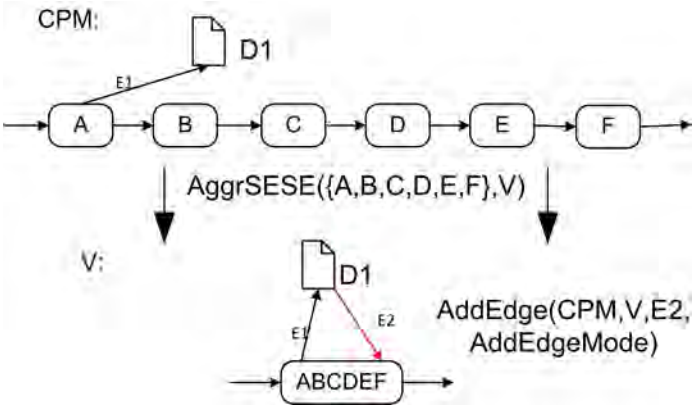

Abbildung 4.6: Prozessschema CPM und die daraus erstellte Prozesssicht V

Je nachdem welcher Wert im Parameter *AddEdgeMode* gewählt wurde, wird die Lesekante in unterschiedlichen Varianten in das Prozessschema CPM eingefügt. Abbildung 4.7 a) zeigt die *EARLY-Variante*, Abbildung 4.7 b) die *LATE-Variante* und Abbildung 4.7 c) die *ALL-Variante* beim Einfügen einer Lesekante in eine Sequenz von Aktivitäten im Prozessschema.

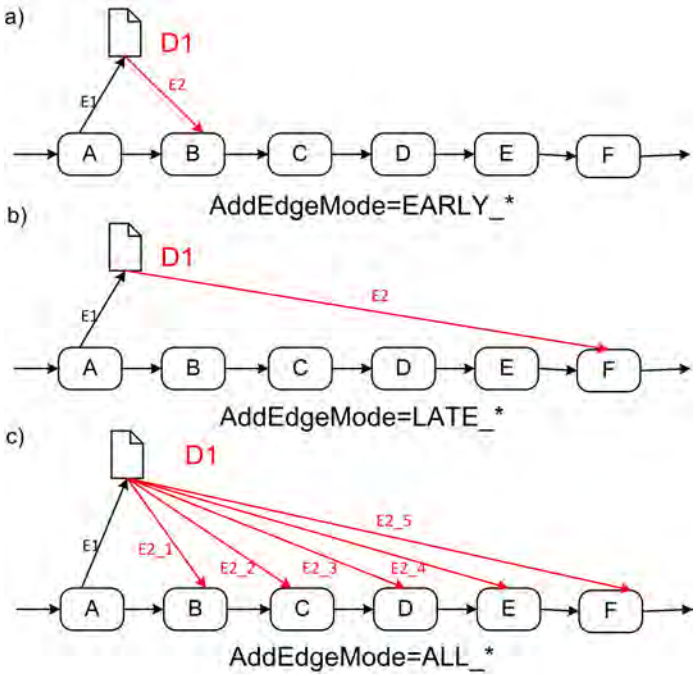

Abbildung 4.7: Darstellung der verschiedenen Modi einer Lesekante bei einer Sequenz

Das Einfügen einer Schreibkante bei einer Sequenz von Aktivitäten wird in den Abbildung 4.8 und 4.9 veranschaulicht. Abbildung 4.8 zeigt das Prozessschema CPM und die Prozesssicht V, die aus dem Prozessschema CPM durch Anwenden der Operation *AggrSESE*($\{A, B, C, D, E\}$,V) entsteht. In der Prozesssicht V wird die Schreibkante $E1$ mit Hilfe der Operation AddEdge(CPM,V,$E1$,*AddEdgeMode*) eingefügt.

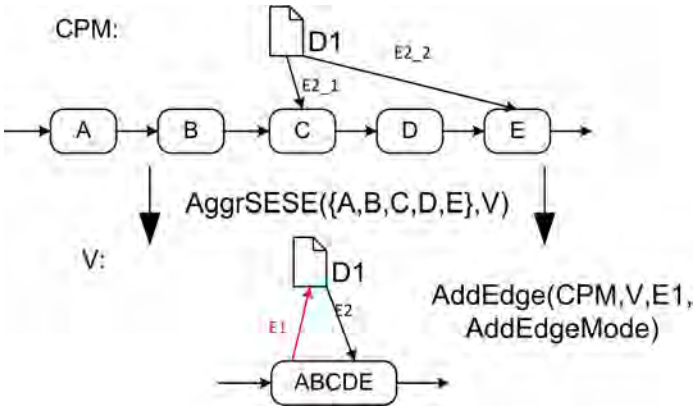

Abbildung 4.8: Prozessschema CPM und die daraus erstellte Prozesssicht V

Je nach gewählten Modus im Parameter *AddEdgeMode* wird die Schreibkante in unterschiedlichen Varianten eingefügt. Abbildung 4.9 a) zeigt die *EARLY-Variante* und Abbildung 4.9 b) die *LATE-Variante*. Eine *ALL-Variante* wie bei der Lesekante ist nicht möglich, weil laut Datenflussregel *DF-3* keine Schreibzugriffe auf Daten nacheinander erfolgen dürfen (siehe Kapitel 2.3).

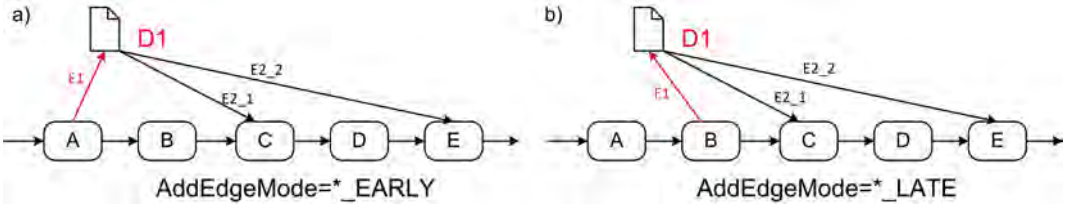

Abbildung 4.9: Darstellung der verschiedenen Modi einer Schreibkante bei einer Sequenz

Das Einfügen von Datenkanten in eine AND-Kontrollstruktur unterscheidet sich vom Einfügen in eine Sequenz. Eine AND-Kontrollstruktur besteht aus mehreren parallel zueinander ablaufenden Sequenzen. Diese Besonderheit muss beim Einfügen von Datenkanten berücksichtigt werden.
Abbildung 4.10 zeigt das Prozessschema CPM und die Prozesssicht V, die aus dem Prozessschema CPM durch Anwendung der Operation *AggrComplBranches*($\{\{A, B, C\}, \{D, E, F\}\}$,$V$) und *AggrDataElement*($\{D1, D2\}$,$V$) entsteht. In der Prozesssicht V wird die Lesekante $E2$ mit AddEdge(CPM,V,$E2$,*AddEdgeMode*) eingefügt.

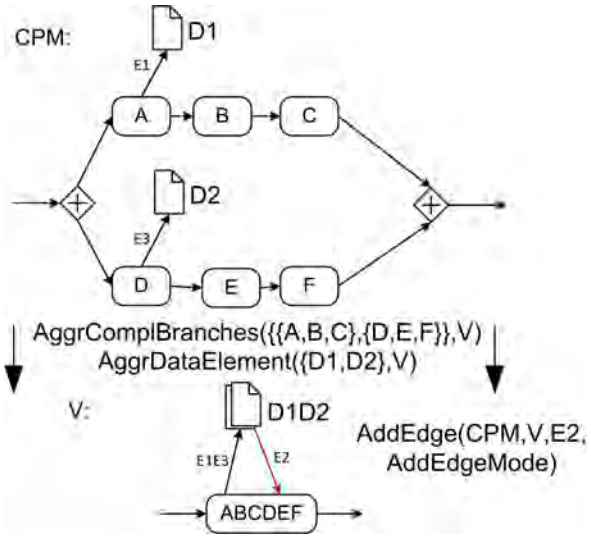

Abbildung 4.10: Prozessschema CPM und die daraus erstellte Prozesssicht V

Der Parameter *AddEdgeMode* bestimmt die Variante, mit der die Lesekante eingefügt wird, wobei hier jeder Verzweigungspfad der AND-Kontrollstruktur als einzelne Sequenz betrachtet wird. Die Lesekante wird auf jedem Verzweigungspfad eingefügt auf dem das Datenelement auch geschrieben wird. Sonst besteht die Gefahr von Lesezugriffen, ohne dass das Datenelement bereits durch einen Schreibzugriff versorgt wurde. Abbildung 4.11 a) zeigt die *EARLY-Variante*, Abbildung 4.11 b) die *LATE-Variante* und Abbildung 4.11 veranschaulicht die *ALL-Variante* beim Einfügen in einer AND-Kontrollstruktur.

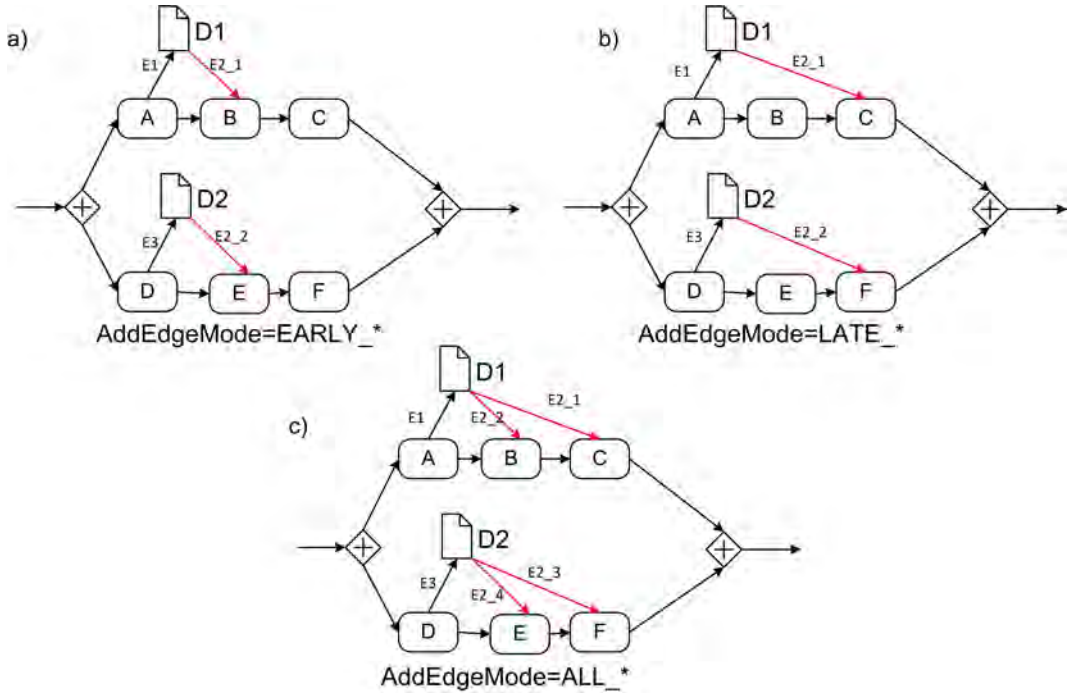

Abbildung 4.11: Varianten beim Einfügen einer Lesekante in einer AND-Kontrollstruktur

Wie diskutiert, kann das Einfügen einer Schreibkante bei einer AND-Kontrollstruktur die Datenflusskorrektheit zerstören und das Korrektheitslevel reduzieren, weil man sehr viele Möglichkeiten hat die Schreibkante einzufügen. Da alle Verzweigungspfade gleichzeitig ablaufen können, kann nicht sicher gestellt werden, dass zum Beispiel Daten überschrieben werden, die noch gebraucht werden. Darum ist es für die Prozesskorrektheit nicht vertretbar, diese Operation durchzuführen. Außerdem gibt es sehr viele Möglichkeiten die Schreibkanten einzufügen. Es ist schwer zu entscheiden, welche von diesen Möglichkeiten der Bearbeiter meint. Abbildung 4.12 und Abbildung 4.13 zeigen Beispiele dazu.
Abbildung 4.12 zeigt das Prozessschema CPM und die Prozesssicht V, die durch die Ope-

rationen *AggrComplBranches*($\{\{A, B, C\}, \{D, E, F\}\}$,$V$) und *AggrDataElement*($\{D1, D2\}$, V) aus dem Prozessschema CPM erstellt wurde. Die Schreibkante $E1$ wird mit AddEdge(CPM,V,$E1$,*AddEdgeMode*) eingefügt.

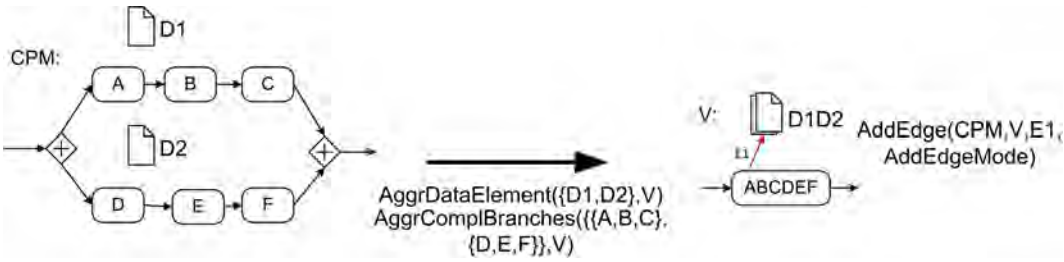

Abbildung 4.12: Prozessschema CPM und die daraus erstellte Prozesssicht V

Es gibt nun verschiedene Möglichkeiten die Schreibkante $E1$ einzufügen: Man kann sie wie in Abbildung 4.13 a) einfügen und dabei annehmen, dass jeweils ein Datenelement auf dem einen und ein Datenelement auf dem anderen Verzweigungspfad geschrieben werden soll. Ebenso kann man die Schreibkante wie in Abbildung 4.13 b) einfügen und dabei annehmen, dass beide Datenelemente auf dem unteren Verzweigungspfad geschrieben werden sollen. Es ist auch möglich, dass beide Datenelemente auf dem oberen Pfad geschrieben werden sollen. Diese Möglichkeiten beim Einfügen der Schreibkante sind neben der Gefahr, die Prozesskorrektheit zu verletzten, der Grund, dass keine Schreibkanten in eine AND-Verknüpfung einfügt werden.

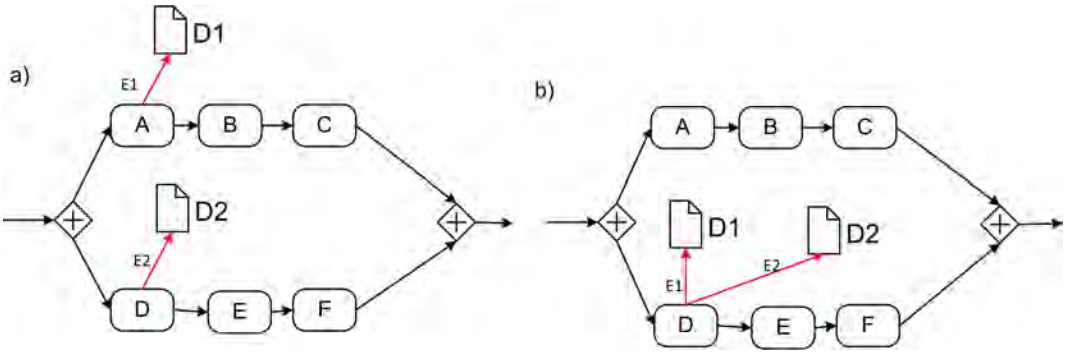

Abbildung 4.13: Möglichkeiten beim Einfügen einer Schreibkante in einer AND-Kontrollstruktur mit dem Modus *EARLY*

Das Einfügen von Lesekanten in XOR-Kontrollstrukturen verläuft analog zu AND-Kontrollstrukturen, nur ist es ebenso möglich Schreibkanten einzufügen, weil immer nur ein Ver-

zweigungspfad ausgeführt wird. Es ist nicht möglich, dass ein Verzweigungspfad ein Datenelement überschreibt, das noch nicht in einem anderen Verzweigungspfad gelesen wurde. Abbildung 4.14 zeigt das Prozessschema CPM und die Prozesssicht V, die mit den Operationen *AggrComplBranches*($\{\{A, B, C\}, \{D, E, F\}\}$,$V$) und *AggrDataElement*($\{D1, D2\}$,$V$) aus dem Prozessschema CPM erstellt wurde. In der Prozesssicht V wird die Schreibkante $E1$ mit der Operation *AddEdge*(CPM,V,$E1$,*AddEdgeMode*) eingefügt.

Abbildung 4.14: Prozessschema CPM und die daraus erstellte Prozesssicht V

Abbildung 4.15 zeigt die verschiedenen Varianten beim Einfügen einer Schreibkante in eine XOR-Kontrollstruktur. Abbildung 4.15 a) zeigt die *LATE_*-* und Abbildung 4.15 b) die *EARLY_*-Variante*. Jedes Datenelement wird auf jedem Verzweigungspfad geschrieben. Eine XOR-Kontrollstruktur hat den Vorteil, dass immer nur ein Verzweigungspfad ausgeführt wird. Damit kann ein Datenelement nicht auf einem anderen Verzweigungspfad überschrieben werden.

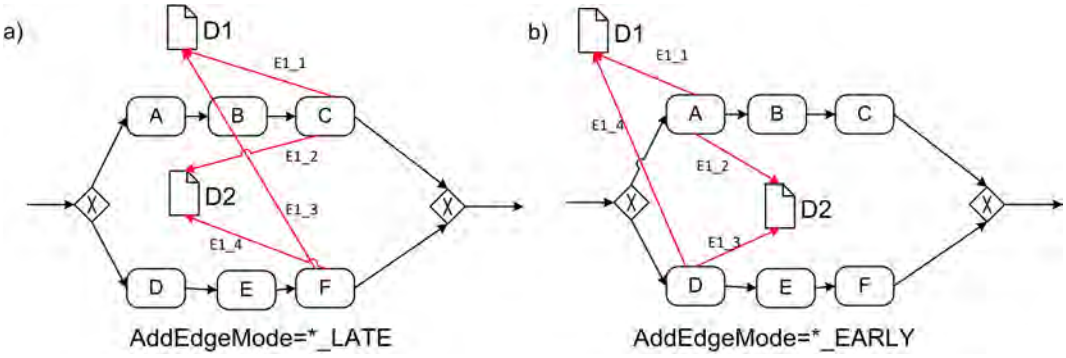

Abbildung 4.15: *LATE*- und *EARLY*-Variante beim Einfügen einer Schreibkante in eine XOR-Kontrollstruktur

Der Ablauf der Operation *AddEdge*(P,V,e,*AddEdgeMode*) wird nun in einem vollständigen Beispiel veranschaulicht. Abbildung 4.16 zeigt das Prozessschema CPM und die

Prozesssicht V, die durch Anwenden der Operationen *AggrComplBranches(C,D,V)* und *AggrSESE(A,B,CD,E,V)* aus dem Prozessschema CPM entsteht. Mit Hilfe der Operation *AddEdge(CPM,V,$E2$,EARLY_*)* wird die Lesekante $E2$ eingefügt.

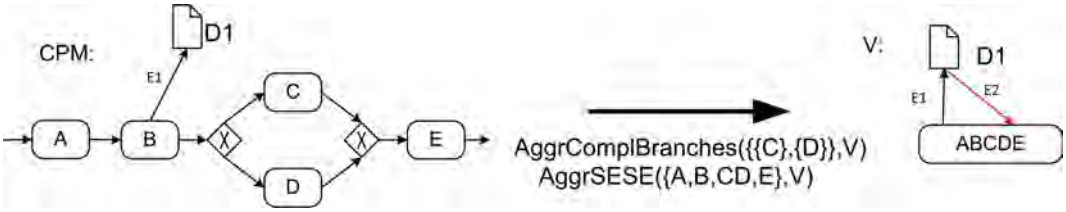

Abbildung 4.16: Prozessschema CPM und die daraus erstellte Prozesssicht V

In Algorithmus 6 wird zuerst die Menge der Aktivitäten N_{CPM} bestimmt, aus denen die Aktivität $ABCDE$ besteht (siehe Zeile 8). Dann wird in Zeile 10, die erste schreibende Kante für das Datenelement $D1$ bestimmt. In unseren Fall ist $E1$ die erste Schreibkante für das Datenelement $D1$. Da der Einfügemodus $AddEdgeMode = EARLY\_*$ ist, wird n_{first} bestimmt (siehe Zeile 12). In unseren Fall ist $n_{first} = A$, da aber die Prüfung in Zeile 13 ergibt, dass die erste schreibende Aktivität Teil von N_{CPM} ist, wird n_{first} neu gesetzt. Knoten n_{first} ist nun der $AND\text{-}Split$. Die Prüfung in Zeile 16 ergibt, dass n_{first} keine Aktivität ist und damit die Lesekante $E2$ und $E3$ an die nachfolgenden Aktivitäten von n_{first} gesetzt werden (siehe Abbildung 4.17). Damit ist das Einfügen der Kante abgeschlossen.

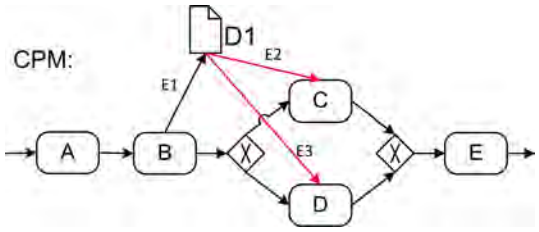

Abbildung 4.17: Mit AddEdgeMode=EARLY_* eingefügte Lesekante in das Prozessschema CPM

4.1.5 Einfügen eines Datenelements: InsertDataElement

Die Operation *InsertDataElement(P,V,S,d_{new},InsertDataMode)* fügt ein neues Datenelement d_{new} mit einer oder mehreren Datenkanten in die Prozesssicht V und das Prozessschema P ein. Als Eingaben hat die Operation ein Prozessschema P, eine auf P basierende Prozesssicht V, eine Menge an Datenkanten S, die das neue Datenelement mit Aktivitäten verbindet, das einzufügende Datenelement d_{new} und den Modus für das Einfügen der Datenkanten *InsertDataMode*. Zuerst wird in Zeile 7 im Algorithmus 7 das neue Datenelement d_{new} der Menge der Datenelemente D hinzugefügt und in Zeile 8 in Algorithmus 7 wird die neue Datenmenge D' in das Tupel des Prozessschema P aufgenommen. Ab Zeile 9 wird auf AddEdge zugegriffen, um die Datenkanten einzufügen. *InsertDataMode* legt dabei fest, ob die Lesekanten oder Schreibkanten zuerst eingefügt werden. Bei $InsertDataMode = *\_EARLY$ muss man immer die Schreibkanten zuerst einfügen, damit man sich beim Einfügen der Lesekante an der Schreibkante orientieren kann und es so nicht möglich ist die Lesekante vor der ersten Schreibkante zu setzen. Gerade anders herum gestaltet es sich bei $InsertDataMode = LATE\_LATE$. Die Lesekanten müssen zuerst einfügt werden, damit man sich beim Einfügen der Schreibkante an der Lesekante orientieren kann und es nicht möglich ist die Schreibkante nach der ersten Lesekante zu setzen.

Algorithm 7 InsertDataElement(P,V,S,d_{new},InsertDataMode)

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $V = (N_V, D_V, E_V, EC_V, NT_V, ET_V)$
4: $S \subset E_V$: edge(s) to be inserted
5: $d_{new} \in D_V$ new data element
6: InsertDataMode $\in \{EARLY\_EARLY, LATE\_LATE, LATE\_EARLY, ALL\_EARLY\}$
//read mode followed by write mode
7: $D' = D \cup \{d_{new}\}$
8: $P' = (N, D', E, EC, NT, ET)$
9: **if** InsertDataMode=*_EARLY **then**
10: **for all** $dw \in S \wedge DET(dw) = (w, *)$ **do**
11: AddEdge(P',V,dw,*InsertDataMode*)
12: **end for**
13: **for all** $dr \in S \wedge DET(dr) = (r, *)$ **do**

14: AddEdge(P',V,dr,*InsertDataMode*)
15: **end for**
16: **end if**
17: **if** InsertDataMode=LATE_LATE **then**
18: **for all** $dr \in S \wedge DET(dr) = (r, *)$ **do**
19: AddEdge(P',V,dr,*InsertDataMode*)
20: **end for**
21: **for all** $dw \in S \wedge DET(dw) = (w, *)$ **do**
22: AddEdge(P',V,dw,*InsertDataMode*)
23: **end for**
24: **end if**

4.1.6 Erweitern eines bestehendes Datenelements: ExpandDataElement

Die Operation *ExpandDataElement(P,V,D',d_{new})* erweitert ein bestehendes Datenelement aus der Prozesssicht (d.h. es werden dem komplexen Datenelement weitere Attribute, z.B. ein Integer, hinzugefügt). Die Menge D' enthält alle Datenelemente, die als *Vorbild* für das neue Datenelement dienen sollen. *Vorbild* bedeutet in diesem Fall, dass das neue Datenelement von allen Aktivitäten gelesen und geschrieben wird, die auch die Datenelemente aus D' schreiben oder lesen. Als Eingaben erhält die Operation zusätzlich das Prozessschema P, eine auf P basierende Prozesssicht V und das neue Datenelement d_{new}.

Algorithm 8 ExpandDataElement(P,V,D',d_{new})

1: **Input:**
2: $P = (N, D, E, EC, NT, ET)$
3: $V = (N_V, D_V, E_V, EC_V, NT_V, ET_V)$
4: $d_{new} \in D$ new data element

5: $D'' := D \cup \{d_{new}\}$
6: $E' := E$
7: **for all** $d_{old} \in D'$ **do**
8: **for all** $x \in CPMDataEdge((d_{old}, n1 \in N), V)$ **do**
9: $E' = E' \cup \{(d_{new}, n1)\}$
10: **end for**

11: **for all** $y \in CPMDataEdge((n2 \in N, d_{old}), V)$ **do**
12: $E' = E' \cup \{(n2, d_{new})\}$
13: **end for**
14: **end for**
15: **return** $P' = (N, D'', E', EC, NT, ET)$

In den Zeile 8-10 in Algorithmus 8 wird für jede Lesekante x eines Datenelements aus der Menge D', eine neue Lesekante für das Datenelement d_{new} gesetzt. Analog für Schreibkanten in Zeilen 11-13. Bei der Erstellung der Datenflusskanten kann es passieren, dass das Korrekheitslevel der Prozesssicht reduziert wird. Auf das Korrektheitslevel wird in Kapitel 5 näher eingegangen.
Die Operationen *DeleteDataElement*(P,V,d), *DeleteEdge*(P,V,e), *ChangeAttribute*(P,V,e,t_{new}), *AddEdge*, *InsertDataElement*(P,V,S,d_{new},*InsertDataMode)* und *ExpandDataElement*(P,V,D',d_{new}) aus Kapitel 4.1 ermöglichen eine Veränderung der Prozesssichten und ihren zugrundeliegenden Prozessschemata. Der nächste Schritt ist die Veränderungen auf andere Prozesssichten zu übertragen, die auf dem selben Prozessschema basieren wie die veränderte Prozesssicht. Auf diese Migration wird in Kapitel 4.2 eingegangen.

4.2 Migrationsregeln

Die Migration ist ein wichtiges Konzept im proView-Projekt (siehe Kapitel 2.4). Sie führen die Aktualisierung der Prozesssichten bei der Anwendung einer Änderungsoperation durch. Die Migration der anderen Prozesssichten der anderen Bearbeiter ist wichtig, damit es keine inkonsistenten Prozesssichten verschiedener Bearbeiter gibt und die Bearbeiter auf dem neusten Stand sind.
Für alle Migrationsfälle gilt: Wenn durch eine Änderung die Menge an Datenelementen in der AggrDataElements-Operation leer ist, wird die gesamte Operation gelöscht. Wenn das Datenelement gelöscht wird das reduziert wird, wird die RedDataElement-Operation ebenfalls gelöscht.

M1: DeleteDataElement:	$\exists AggrDataElements(V1, D_1) = Op_1$ with $d \in D_1, Op_1 \in Op \Rightarrow Op_1 := AggrDataElements(V1, D_1 \backslash \{d\})$
M2: DeleteDataElement:	$\exists RedDataElement(V1, d) = Op_1, Op_1 \in Op \Rightarrow Op' = Op \backslash \{Op_1\}$
M3: ExpandDataElement:	$\exists AggrDataElements(V1, D_1) = Op_1$ with $d1 \in D_1; Op_1 \in Op \Rightarrow Op_1 := AggrDataElements(V1, D_1 \cup \{x\})$
M4: ExpandDataElement:	$\exists RedDataElement(V1, d1) = Op_1; Op_1 \in Op \Rightarrow Op' = Op \cup \{RedDataElement(V1, x)\}$
M5: InsertDataElement:	let d be the new dataelement:$\nexists RedActivity(n, V1) = Op_1$ with $\exists (n, d) \in DE \vee \exists (d, n) \in DE; Op_1 \in Op$ InsertDataMode=SHOW $\Rightarrow Op = Op$ InsertDataMode=HIDE $\Rightarrow Op = Op \cup RedDataElement(V1, d)$

Tabelle 4.1: Übersicht Migrationsregeln

Wie man in der Tabelle 4.1 sieht hat nicht jede Änderungsoperation ihre eigene Migrationsregel. Die Operationen *DeleteEdge(P,V,e)*, *ChangeAttribut(P,V,e,t_{new})* und *AddEdge* benötigen keine Migrationsregeln, da sie nur Datenflusskanten behandeln. Es gibt keine Prozesssichtoperationen in OP in der Erzeugungsmenge $CS_v = (P, OP, PS)$, die direkt auf Datenflusskanten angewendet wird. Datenflusskanten werden immer nur als Ergebnis einer anderen Prozesssichtoperation wie zum Beispiel *AggrDataElements(S,P)* zusammengefasst. Darum reicht es bei diesen Änderungsoperationen die Prozesssichten neu zu erzeugen, was das proView-Konzept automatisch vorsieht.
Sei bei Migrationsregel $M1$ d das gelöschte Datenelement. Nachdem die Operation *Delete DataElement(P,V,d)* ausgeführt wurde, wird dieses Datenelement aus der Menge der aggregierten Datenelemente D_1 in den Prozesssichtoperationen $AggrDataElements(V1, D_1)$ gelöscht. Bei der Migrationsregel $M2$ die ebenfalls bei *DeleteDataElement(P,V,d)* angewendet wird, werden alle Prozesssichtoperationen $RedDataElement(V1, d)$ aus der Menge der Prozesssichtoperationen OP in den verschiedenen Erzeugungsmengen $CS_v = (P, OP, PS)$ gelöscht. Für die Operation *ExpandDataElement(P,V,D',d_{new})* existieren ebenfalls zwei Migrationsregeln. Bei Migrationsregel $M3$ wird das neue Datenelement x, dass sich beispielsweise wie Datenelement $d1$ soll, in allen Erzeugungsmengen in die zu aggregierende Menge D_1 eingefügt, in denen Datenelement $d1$ aggregiert wird. Bei Migrations-

regel $M4$ wird das neue Datenelement x in allen Erzeugungsmengen reduziert, in denen auch $d1$ reduziert wird. Bei Migrationsregel $M5$ wird unterschieden ob das neue Datenelement d von einer Aktivität gelesen oder geschrieben wird, die der Benutzer sieht. Das erkennt man daran, ob es in der Menge der Prozesssichtoperationen OP in der Erzeugungsmenge $CS_v = (P, OP, PS)$ eine Reduktionsregel für dieses Aktivität gibt oder nicht. Wenn es gelesen oder geschrieben wird, von einer Aktivität die der Benutzer sieht, dann hat man die Wahl, ob man es anzeigen lassen will oder reduzieren will. Diese Wahl wird durch den Parameter *InsertDataMode* getroffen, der die Werte *SHOW* und *HIDE* haben kann. Existiert eine Reduktionsregel in der Menge der Prozesssichtoperationen OP für die schreibende oder lesende Aktivität, wird das Datenelement immer reduziert.

5 Korrektheitsaspekte von Prozesssichten

Kapitel 5.1 führt mit Hilfe der in Kapitel 2.3 vorgestellte Kontroll- und Datenflussregeln Korrektheitslevel ein. In Kapitel 5.2 werden die Änderungsoperationen aus Kapitel 4.1 dahingehend überprüft, ob sie das Korrektheitslevel einer Prozesssicht herabstufen können.

5.1 Korrektheitslevel

Das proView-Projekt erlaubt die Ausführung von ausgewählten Prozesssichten auf verschiedenen Business Process Management Systemen. Die Laufzeitinformationen werden im zentralen Prozessschema gesammelt und in allen verbundenen Prozesssichten angezeigt. Damit das Ausführen von Prozesssichten möglich ist, muss eine Prozesssicht Korrektheitsaspekte einhalten, die in verschiedenen Korrektheitsleveln definiert sind. Es gibt drei Korrektheitslevel: *KL-1:No Update*, *KL-2:Update* und *KL-3:Execution-Ready*. Hat eine Prozesssicht Korrektheitslevel *KL-3:Execution-Ready*, kann die Prozesssicht ausgeführt werden oder durch Änderungsoperationen verändert werden (siehe Kapitel 4.1). Hat eine Prozesssicht Korrektheitslevel *KL-2:Update*, kann die Prozesssicht nicht ausgeführt werden, weil ein korrekter Datenfluss nicht vorhanden ist, aber durch Änderungsoperationen verändert werden. Bei Korrektheitslevel *KL-1:No Update* kann die Prozesssicht weder ausgeführt noch geändert werden, weil weder ein korrekter Datenfluss noch ein korrekter Kontrollfluss vorhanden ist.
Eine Prozesssicht hat ein Korrektheitslevel, wenn sie die spezifischen Regeln, die das Korrektheitslevel definiert, erfüllt. Die Regeln teilen sich auf in Regeln für den Kontroll- und Datenfluss und wurden in Kapitel 2.3 vorgestellt. In Tabelle 5.1 ist übersichtlich dargestellt, welche Regeln für welches Korrektheitslevel gelten müssen. Zusammengefasst müssen bei Korrektheitslevel 3 alle Regeln gelten, damit die Datenflusskorrektheit und Kontrollflusskorrektheit gesichert ist. Davon ausgenommen sind für alle Korrektheitslevel die Kontrollflussregeln *KF-6* und *KF-8*, da sie keinen Einfluss auf das Prozessschema haben. Für

Korrektheitslevel 2 muss nur die Kontrollflusskorrektheit sichergestellt werden. Für Korrektheitslevel 1 wiederum muss nur Kontrollflussregel KF-1 gelten.

	Korrektheitslevel 1: *No Update*	Korrektheitslevel 2: *Update*	Korrektheitslevel 3: *Execution-Ready*
KF-1	✓	✓	✓
KF-2	×	✓	✓
KF-3	×	✓	✓
KF-4	×	✓	✓
KF-5	×	✓	✓
KF-7	×	✓	✓
DF-1	×	×	✓
DF-2	×	×	✓
DF-3	×	×	✓

Tabelle 5.1: Übersicht über die Korrektheitslevel

5.2 Korrektheitsaspekte der Änderungsoperationen

Die Änderungsoperationen *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* und *ExpandDataElement* aus Kapitel 4.1 können das Korrektheitslevel einer Prozesssicht heruntersetzten und damit zum Beispiel eine Prozesssicht nicht mehr ausführbar machen. Betrachten wir nun die einzelnen Änderungsoperationen:
DeleteDataElement aus Abschnitt 4.1.1 ändert das Korrektheitslevel einer Prozesssicht nicht. Löscht man ein Datenelement in einer Prozesssicht, die das Korrektheitslevel *KL-3:Execution-Ready* besitzt, wird das Datenelement vollständig mit all seinen verbundenen Datenkanten gelöscht. Es bleibt also kein Datenelement zurück, das zum Beispiel nicht geschrieben wird.
Tabelle 5.2 zeigt eine Übersicht über die Änderung des Korrektheitslevel bei der Operation *DeleteDataElement*. Tabelle 5.2 wird von Zeile nach Spalte gelesen. Hat die Prozesssicht *KL-1: No Update* kann keine Änderungsoperation auf sie angewendet werden und die Prozesssicht verbleibt in *KL-1: No Update*. Bei der Anwendung einer Änderungsoperation auf eine Prozesssicht in *KL-2: Update*, kann die Prozesssicht nicht in *KL-1: No Update* herabgesetzt werden. Für *KL-2: Update* sind nur Kontrollflussregeln von Bedeutung und diese

können nicht durch die Änderungsoperation *DeleteDataElement* verletzt werden, weil sie sich nur auf Datenelemente bezieht. Deswegen verbleibt eine Prozesssicht in *KL-2: Update* bei der Anwendung von *DeleteDataElement* immer in *KL-2: Update*. Befindet sich die Prozesssicht in *KL-3: Execution-Ready* kann sie durch die Operation *DeleteDataElement* nicht in *KL-1: No Update* heruntergesetzt werden, weil wieder keine Kontrollflussregeln durch *DeleteDataElement* verletzt werden können. Ebenfalls kann die Prozesssicht nicht in *KL-2: Update* herabgesetzt werden, weil das Datenelement immer vollständig mit allen verbundenen Datenkanten gelöscht wird. Bei der Anwendung von *DeleteDataElement* verbleibt eine Prozesssicht in *KL-3: Execution-Ready* immer in *KL-3: Execution-Ready*.

		Korrektheitslevel nach Änderungsoperation		
		KL-1: No Update	KL-2: Update	KL-3: Execution-Ready
Kl vor Ops	KL-1: No Update	×		
	KL-2: Update	×	✓	
	KL-3: Execution-Ready	×	×	✓

Tabelle 5.2: Herabsetzung der Korrektheitslevel bei der Operation *DeleteDataElement*

Bei der Operation *DeleteEdge* aus Abschnitt 4.1.2 kann als Nebeneffekt zur eigentlichen Anwendung das Korrektheitslevel heruntergesetzt werden. Löscht man beispielsweise die letzte verbleibende Schreibkante eines Datenelements, obwohl dieses an einer anderen Stelle noch gelesen wird , ist die Datenflussregel *DF-1* nicht mehr erfüllt. Die Prozesssicht hat dann nur noch ein Korrektheitslevel *KL-2:Update* und ist damit nicht mehr ausführbar (siehe Abbildung 4.2).
Ähnlich ist bei der Operation *ChangeAttribute* aus Abschnitt 4.1.3. Ändert man das Kantenattribut einer Schreibkante von obligat zu optional und gibt es eine obligate Lesekante für das Datenelement, dass von der Schreibkante geschrieben wird, dann ist wiederum Datenflussregel *DF-1* nicht mehr erfüllt. Ein obligate Lesezugriff erfordert immer einen obligaten Schreibzugriff (Abbildung 4.5 zeigt diesen Fall exemplarisch).
Bei der Operation *AddEdge* aus Abschnitt 4.1.4 kann es beim Einfügen einer Schreibkante in einer AND-Kontrollstruktur passieren, dass es zu einem parallelen Schreibzugriff auf ein Datenelement kommt. Abbildungen 5.1 und 5.2 zeigen diesen Fall exemplarisch. Abbildung 5.1 zeigt das Prozessschema CPM und die Prozesssicht V, die durch Anwendung der Operation $AggrComplBranches(\{\{A, B, C\}, \{D, E, F\}\}, V)$ auf das Prozessschema

CPM erstellt wurde. Mit Hilfe der Operation $AddEdge(CPM, V, E1, AddEdgeMode)$ wird die Schreibkante $E1$ eingefügt.

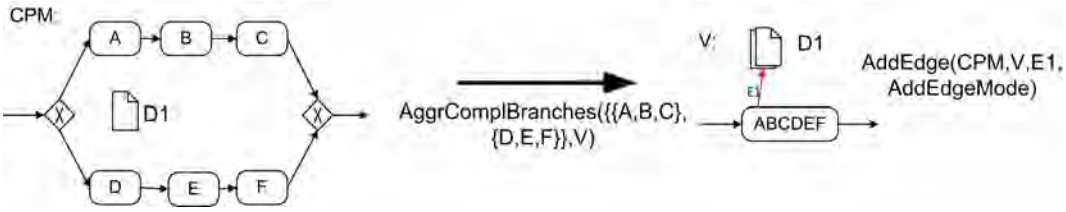

Abbildung 5.1: Prozessschema CPM und die daraus erstellte Prozesssicht V

Das Datenelement $D1$ wird von den Aktivitäten A und D parallel geschrieben. Das widerspricht Datenflussregel *DF-2* und ist ein weiterer Grund, wieso in Algorithmus 6 überprüft wird, ob die Schreibkante in eine AND-Kontrollstruktur eingefügt wird. Gleiches gilt für die Operation *InsertDataElement* aus Abschnitt 4.1.5, da die Operation größtenteils auf *AddEdge* basiert.

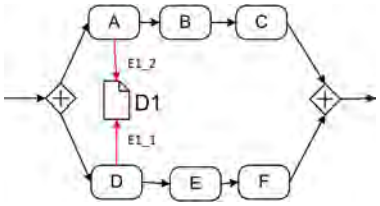

Abbildung 5.2: Ergebnis der Operation

Bei der Operation *ExpandDataElement* kann es passieren, dass sich in dieser Menge der *Vorbilder* Datenelemente befinden, die von direkt aufeinanderfolgenden Aktivitäten geschrieben werden. Das neue Datenelement wird dann von zwei direkt aufeinanderfolgenden Aktivitäten geschrieben. Dies verletzt Datenflussregel *DF-3*. Ebenso kann diese Menge Aktivitäten enthalten die sich auf verschiedenen Pfaden in einer AND-Kontrollstruktur befinden. Das hat bei Anwendung der Operation eine Verletzung von Datenflussregel *DF-2* zur Folge. Abbildung 5.3 und 5.4 zeigen exemplarisch einen solchen Fall.
Abbildung 5.3 zeigt das Prozessschema CPM und die Prozesssicht V, die durch Anwendung der Operationen *AggrComplBranches*($\{\{A, B, C\}, \{D, E, F\}\}$,$V$) und *AggrDataElement*($\{D1, D2\}, V$) aus dem Prozessschema CPM erstellt wird. Mit Hilfe der Operation *ExpandDataElement*(CPM,V,$\{D1, D2\}$,$D3$) wird das komplexe Datenelement $D1D2$ er-

weitert. Das neue Datenelement $D3$ soll sich dabei wie die beiden bestehen Datenelement $D1$ und $D2$ verhalten.

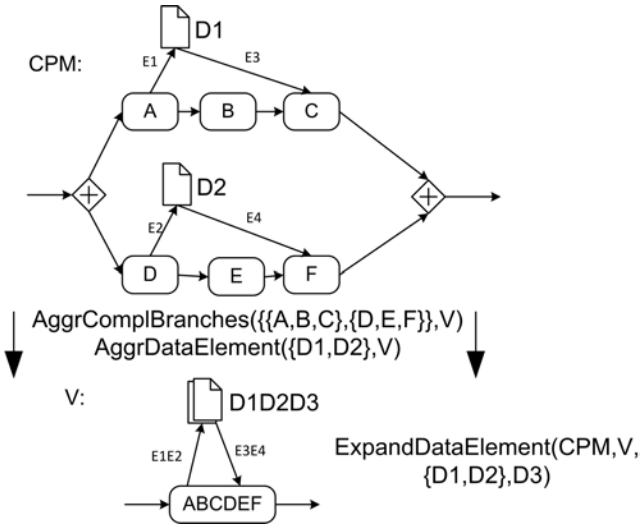

Abbildung 5.3: Prozessschema CPM und die daraus erstellte Prozesssicht V

Die Anwendung der Operation *ExpandDataElement(*CPM*,*V*,*$\{D1, D2\}$*,*$D3$*)* hat zur Folge, dass das Datenelement $D3$ von allen Aktivitäten gelesen und geschrieben wird, die auch $D1$ und $D2$ lesen oder schreiben. Deswegen wird $D3$ durch die Datenkanten $E5$ und $E6$ von den Aktivitäten A und D parallel geschrieben. Damit ist aber Datenflussregel *DF-2* verletzt, die parallele Schreibzugriffe verbietet (siehe Abbildung 5.4).

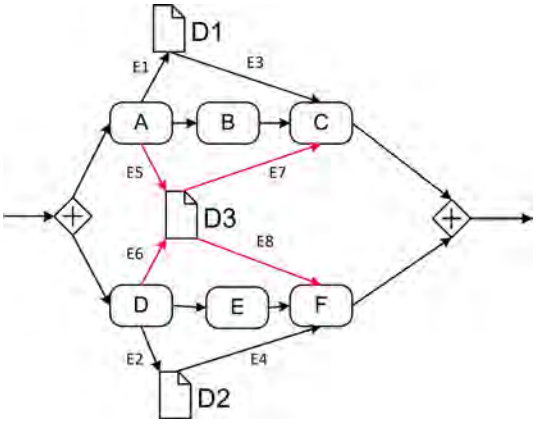

Abbildung 5.4: Ergebnis der Operation *ExpandDataElement(*CPM*,*V*,*$\{D1, D2\}$*,*$D3$*)*

Tabelle 5.3 zeigt eine Übersicht über die Änderungen am Korrektheitslevel einer Prozesssicht, die als Nebeneffekt zur Anwendung von *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* oder *ExpandDataElement* auftreten können. Es werden nur die Fälle betrachtet in denen das Korrektheitslevel herabgesetzt wird. Tabelle 5.3 wird von Zeile nach Spalte gelesen. Auf eine Prozesssicht mit *KL-1:No Update* können keine Änderungsoperationen angewendet werden, deswegen verbleibt die Prozesssicht im Korrektheitslevel *KL-1:No Update*. Die Operationen *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* und *ExpandDataElement* können eine Prozesssicht mit Korrektheitslevel *KL-2:Update* nicht in Korrektheitslevel *KL-1:No Update* herabstufen, weil nur Kontrollflussregeln für *KL-2:Update* von Bedeutung sind. Die Operationen nehmen keine Änderung am Kontrollfluss vor, sondern beschränken sich auf Änderungen am Datenfluss. Bei der Anwendung der Operationen *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* oder *ExpandDataElement* verbleibt eine Prozesssicht in *KL-2: Update* immer in *KL-2: Update*. Eine Prozesssicht in Korrektheitslevel *KL-3:Execution-Ready* kann durch die Operationen *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* oder *ExpandDataElement* in *KL-2:Update* herabgestuft werden.

		Korrektheitslevel nach Änderungsoperation		
		KL-1: No Update	KL-2: Update	KL-3: Execution-Ready
Kl vor Ops	KL-1: No Update	×		
	KL-2: Update	×	✓	
	KL-3: Execution-Ready	×	✓	✓

Tabelle 5.3: Herabsetzung der Korrektheitslevel bei den Operation *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* und *ExpandDataElement*

Die Änderungsoperationen *DeleteEdge*, *ChangeAttribute*, *AddEdge*, *InsertDataElement* und *ExpandDataElement* aus Kapitel 4.1 können das Korrektheitslevel von Prozesssichten heruntersetzten und dazu führen, dass die Prozesssichten nicht mehr ausführbar sind. Sollte die Ausführbarkeit der Prozesssichten immer gewährleistet sein, muss man bei der Anwendung der Operationen auf die in diesem Kapitel vorgestellten Fehlerfälle prüfen und ggf. die Operation zurückweisen.

6 Schlussfolgerung

Heutzutage sind in modernen Unternehmen eine Vielzahl an unterschiedlichen Partnern, Abteilungen und Bearbeitern an Geschäftsprozessen beteiligt. Die immer weiter wachsende Komplexität dieser Geschäftsprozesse erschwert es, dem einzelnen Bearbeiter seine benötigten Daten schnell und einfach zugänglich zu machen. Individuelle Prozesssichten für den einzelnen Nutzer lösen dieses Problem und ermöglichen es den Prozess an die Bedürfnisse des jeweiligen Betrachters anzupassen. Die in Abschnitt 3.2 vorgestellten Operationen ermöglichen es individuelle Prozesssichten zu erstellen. Sie beziehen sich speziell auf Datenelemente und sind dazu geeignet Datenelemente vollständig zu reduzieren oder mehrere Datenelemente in einem komplexen Datenelement zu aggregieren. Die in Abschnitt 3.1 vorgestellten Operationen beziehen sich auf Aktivitäten. Sie können Aktivitäten reduzieren oder mehrere Aktivitäten zusammenfassen. In [2] finden sich noch mehr Operationen als die in Abschnitt 3.1 vorgestellten.
Die Wettbewerbsfähigkeit von moderne Unternehmen hängt heute entscheidend davon ab, wie flexibel sie auf Veränderungen reagieren können. Die in Kapitel 4 vorgestellten Änderungsoperationen ermöglichen es, die Prozesssichten zu verändern und diese Änderungen auf das ursprüngliche Prozessschema zu übertragen. Die Bearbeiter können damit auf Sonderfälle oder Ausnahmen reagieren. Die Operationen sind robust und erkennen Sonderfälle, die die Prozesskorrektheit gefährden.
Ein weiterer wichtiger Schritt für die Flexibilität sind die Migrationsregeln aus Abschnitt 4.2. Sie übertragen die Änderungen durch die Änderungsoperationen in die anderen Prozesssichten, die vom gleichen Prozessschema abstammen wie die veränderte. Dadurch wird jede Prozesssicht immer aktuell gehalten.
Das Ziel des Korrektheitslevel für Prozesssichten ist es, die Prozesssichten ausführbar zu machen. Wie in Kapitel 5 dargestellt, können einige Änderungsoperationen das Korrektheitslevel herunter setzen und damit die Prozesssicht ihrer Ausführbarkeit berauben.
Zusammenfassend bieten, die in dieser Arbeit vorgestellten Operationen, nicht nur die Möglichkeit der individuellen Prozessvisualisierung, sondern ermöglichen die Änderung dieser

Prozessvisualisierungen und die Übertragung dieser Änderungen auf andere Prozessvisualisierungen.

Literaturverzeichnis

[1] *proView Projektseite*. `http://www.uni-ulm.de/in/iui-dbis/forschung/projekte/proview-project.html`. – 25.09.2012

[2] BOBRIK, Ralph: *Konfigurierbare Visualisierung komplexer Prozessmodelle*, Universität Ulm, Diss., 2008

[3] KOLB, Jens ; KAMMERER, Klaus ; REICHERT, Manfred: Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In: *Demo Track of the 10th Int'l Conf on Business Process Management (BPM'12)*, 2012

[4] KOLB, Jens ; KAMMERER, Klaus ; REICHERT, Manfred: Updatable Process Views for User-centered Adaption of Large Process Models. In: *10th Int'l Conference on Service Oriented Computing (ICSOC'12)*, Springer, 2012

[5] KOLB, Jens ; REICHERT, Manfred ; WEBER, Barbara: Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In: *S-BPM ONE 2012*, Springer, April 2012 (CCIS 284), 237–251

[6] REICHERT, Manfred: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*, Universität Ulm, Diss., 2000

[7] REICHERT, Manfred ; KOLB, Jens ; BOBRIK, Ralph ; BAUER, Thomas: Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In: *27th ACM Symposium On Applied Computing (SAC'12), 9th Enterprise Engineering Track (EE'12)*, ACM Press, March 2012, 1653–1660

[8] WEBER, Barbara ; REICHERT, Manfred ; MENDLING, Jan ; REIJERS, Hajo: Refactoring Large Process Model Repositories. In: *Computers in Industry* 62 (2011), June, Nr. 5, S. 467–486

Name: Bernd Mertesz Matrikelnummer: 699008

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Bernd Mertesz