ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Implementierung einer Komponente zur Erstellung und Visualisierung von Prozesssichten

Masterabschlussarbeit an der Universität Ulm

Vorgelegt von:
Manuel Ihlenfeld
manuel.ihlenfeld@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Betreuer:
Jens Kolb

2011

Fassung 29. Dezember 2011

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Grafische Darstellungen von umfangreichen Geschäftsprozessen können sehr unübersichtlich für einzelne Zielgruppen sein. Nicht jede Zielgruppe muss aber alle Details des Prozesses sehen. Sichten auf einen Geschäftsprozess blenden Teile des Prozesses aus oder fassen sie zusammen.

Im Zuge dieser Arbeit wurde eine Komponente zum Erstellen und Visualisieren solcher Prozesssichten prototypisch implementiert. Es werden die Operationen zur Sichten-Erzeugung, Vereinfachungs-Operatoren und Modifikationsoperationen realisiert und beschrieben.

Inhaltsverzeichnis

1 Einleitung

Prozessorientierte Informationssysteme bieten Computerunterstützung für arbeitsteilige Geschäftsprozesse. Diese Prozesse können über lange Zeiträume laufen und abteilungsübergreifend viele Beteiligte haben. In großen Organisationen können Geschäftsprozesse mit hunderten oder tausenden Aktivitäten sehr umfangreich sein, die in einer mehr oder weniger strengen Reihenfolge erledigt werden müssen. Prozessorientierte Informationssysteme steuern und überwachen die Einhaltung dieser Reihenfolge und stellen den Beteiligten je nach Rolle anstehende Aktivitäten zur Auswahl.

Eine grafische Darstellung eines Geschäftsprozesses zeigt die Reihenfolgebeziehungen zwischen den einzelnen Aktivitäten. Ist ein Geschäftsprozess sehr umfangreich, kann jedoch die grafische Darstellung unübersichtlich und speziell für Benutzer ohne technischen Hintergrund schwer verständlich werden. Je nach *Benutzergruppe* bestehen unterschiedliche Anforderungen an die grafische Darstellung des Geschäftsprozesses. Beispielsweise wünscht ein *Prozessbeteiligter*, der nur einen Teil der Aktivitäten bearbeitet (z.B. ein Kundengespräch und anschließende Dateneingabe), eine Darstellung, in der nur die für ihn relevanten Arbeitsschritte gezeigt werden. Der *Prozessmodellierer*, der den Prozess modelliert, sollte zumindest alle fachlich relevanten Aktivitäten sehen können. Dagegen sollte der *IT-Systementwickler*, der das prozessorientierte Informationssystem implementiert, neben den fachlichen auch alle technischen Aktivitäten (z.B. Datenbankzugriffe) sehen können. Ein *Prozessverantwortlicher* hingegen wünscht eine eher abstrakte Darstellung, in der beispielsweise Gruppen von Aktivitäten abstrahiert werden (z.B. alle bereits abgeschlossenen Aktivitäten). Eine solche Darstellung, in der gewisse Aktivitäten ausgeblendet oder zusammengefasst sind, ist eine abstrakte *Sicht auf den Geschäftsprozess* und wird als *Prozesssicht* bezeichnet. Anforderungen an solch eine Visualisierung von Geschäftsprozessen werden zum Beispiel in [1] anhand mehrerer Fallstudien ermittelt.

Die Visualisierung von Geschäftsprozessen basiert in dieser Arbeit auf Prozessgraphen, bestehend aus Knoten in Form von Aktivitäten und Kontrollflusselementen, und Kanten in Form von Reihenfolgebeziehungen zwischen den Aktivitäten. Alternative Darstellungen für Geschäftsprozesse sind zum Beispiel das Gantt-Diagramm, in dem zeitliche Zusammenhänge innerhalb eines Prozesses gut erkennbar, aber die Abhängigkeiten zwischen Aktivitäten schwer darstellbar sind, oder eine tabellarische Auflistung der Aktivitäten mit ihren

Bearbeitern. Eine Sammlung vielfältiger Muster zur Darstellung von Geschäftsprozessen findet sich in [2].

Das Geschäftsprozessmodell selbst, das *alle* Aktivitäten und Details eines Geschäftsprozesses enthält, wird im Folgenden auch als zentrales Prozessmodell (*CPM*, Central Process Model) bezeichnet. Prozesssichten auf das zentrale Prozessmodell werden auch als View bezeichnet. Prozesssichten können nicht nur auf dem zentralen Prozessmodell basieren, sondern auch auf anderen Sichten. Abbildung 1.1 zeigt eine solche View-Hierarchie. *View 1* und *View 2* sind Sichten auf das zentrale Prozessmodell, während *View 2.1* und *View 2.2* wiederum Sichten auf *View 2* sind.

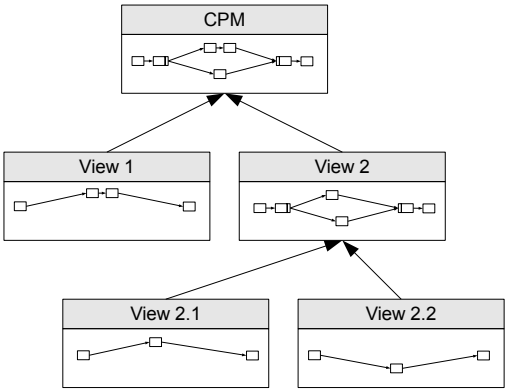

Abbildung 1.1: View-Hierarchie

Geschäftsprozesse sind nicht statisch, das heißt sie werden im Laufe der Zeit verändert. Das passiert, wenn sich Rahmenbedingungen wie beispielsweise die Marktsituation oder die Gesetzeslage ändert, aber auch bei der Optimierung von Geschäftsprozessen oder einfach, wenn in einer konkreten Prozessinstanz eine bisher nicht vorgesehene Ausnahme auftritt, die ein Abweichen vom bisherigen Vorgehen in einem Prozess erfordert [3].

Es ist aber auch möglich, dass nicht nur *ein* Prozessmodellierer den Prozess gestaltet, sondern mehrere Personen oder Rollen an der Modellierung beteiligt sind. Jede dieser Personen kann eine eigene Sicht auf das zentrale Prozessmodell haben, in der gewisse Aktivitäten ausgeblendet oder aggregiert sind. Möchte eine solche Person eine Aktivität über ihre Sicht *einfügen*, soll diese Änderung möglichst automatisch an das zentrale Prozessmodell und andere Sichten in der View-Hierarchie propagiert werden.

Ohne weiteres ist eine automatische Änderungspropagation jedoch nicht möglich. Wird zum Beispiel in View 2.1 (Abb. 1.1) eine Aktivität eingefügt, kann die Einfüge-Position nicht eindeutig einer Position im zentralen Prozessmodell zugeordnet werden. Ebenso ist nicht klar, ob die eingefügte Aktivität zum Beispiel in View 2.2 angezeigt werden soll, obwohl sie in einem Bereich liegt, der in View 2.2 vollständig ausgeblendet ist.

Um trotzdem automatisch das zentrale Prozessmodell und die anderen Sichten *aktualisieren* zu können, werden für das CPM und jede Sicht Parameter festgelegt, die steuern, wie eine Änderung an das CPM und andere Sichten übertragen wird.

Ziel der Arbeit ist es, im Rahmen des *proView*-Projekts eine Komponente prototypisch zu implementieren, die zum einen das Erstellen und Visualisieren von Prozesssichten ermöglicht, und zum anderen Einfüge-Operationen samt Änderungspropagation mithilfe von Parametern realisiert. Bei der Sichtenerzeugung oder -änderung können unvorteilhafte Graphen entstehen. Deshalb sollen Operationen realisiert werden, die Prozessgraphen vereinfachen.

Die Sichtenerzeugung basiert auf den Elementar-Operationen aus dem Proviado-Ansatz [4, 5], ebenso wie die Vereinfachungs-Operationen. Die Algorithmen für die Änderungspropagtion wurden im Rahmen des *proView*-Projekts an der Universität Ulm entwickelt. Als Grundlage für die Implementierung dient die Prozess-Designer-Komponente des Mobile-Services-Prototyps [6].

In Kapitel 2 werden die Grundlagen, die für das weitere Verständnis der Arbeit wichtig sind, eingeführt. Kapitel 3 beschreibt die Vereinfachungs-Operatoren, die bei der Erzeugung von Sichten notwendig sind. Kapitel 4 führt Operatoren und Algorithmen ein zur Aktualisierung des zentralen Prozessmodells und der restlichen Sichten für den Fall, dass in einer Sicht eine Änderung eingeführt wurde. Schließlich werden in Kapitel 5 verwendete Algorithmen und Aspekte des Prototyps gezeigt. Kapitel 6 fasst die Ergebnisse der Arbeit abschließend zusammen.

2 Grundlagen

Die Darstellung von Geschäftsprozessen basiert auf einem formalen Modell, dem *Prozessschema*. Das Prozessschema wird in Kapitel 2.1 definiert. Darauf aufbauend werden in Kapitel 2.2 *Prozesssichten* definiert, die durch Ausblenden oder Zusammenfassen von Aktivitäten erzeugt werden. *Operatoren zur Sichtenerzeugung* werden in Kapitel 2.3 beschrieben. Um konsistente Prozessschemas und -sichten zu gewährleisten, können Aktivitäten über Sichten nicht in beliebiger Art und Weise eingefügt werden, sondern nur unter Verwendung dreier Operatoren. Diese *Einfüge-Operatoren* werden in Kapitel 2.4 konkretisiert.

2.1 Prozessschema

Grundlegend für die Erstellung von Prozesssichten ist das *Prozessschema*. Ein Prozessschema besteht aus Knoten in Form von Aktivitäten oder Gateway-Knoten und Kontrollflussabhängigkeiten. Definition 1 basiert auf der Definition der wohlstrukturierten, azyklischen Kontrollflussgraphen [7], unterscheidet sich aber darin, dass mehrfache Kanten zwischen zwei Knoten existieren können, die durch eindeutige Nummern voneinander unterschieden werden. Prozessschemas sind in dieser Arbeit auf den Kontrollfluss beschränkt, der Datenfluss wird nicht behandelt.

Definition 1. (Prozessschema).

Ein *Prozessschema* ist ein Tupel $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$:

N ist eine Menge von Knoten.

$\mathbb{E} \subset N \times N \times \mathbb{N}$ setzt je zwei Knoten zueinander über gerichtete Kanten in Relation und ordnet jeder Kante eine eindeutige Kantennummer zu. Die Nummerierung der Kanten wird benötigt, um mehrfache direkte Kanten zwischen Split- und Join-Knoten identifizieren zu können. $\mathbb{E}$ ist eine *Mehrfachkantenmenge*; die Elemente von $\mathbb{E}$ werden als *Kontrollflusskanten* bezeichnet.

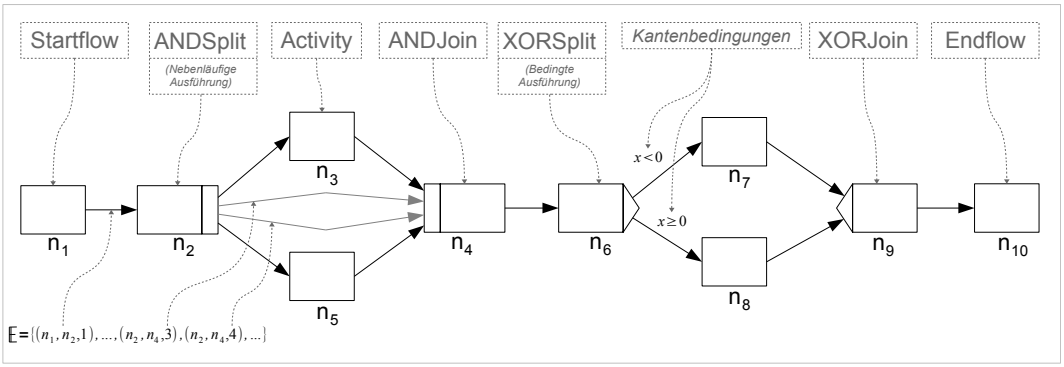

Abbildung 2.1: Beispiel für ein Prozessschema; annotiert mit Knotentypen (oben) und ausschnittsweise der Kantenmenge (links unten).

$\mathrm{EC} : \mathbb{E} \to \mathrm{Cond} \cup \{\mathrm{TRUE}\}$ ordnet jeder Kontrollflusskante eine *Übergangsbedingung* zu, wobei Cond eine Menge von Übergangsbedingungen ist. Mögliche Schreibweisen sind $\mathrm{EC}(e) \equiv \mathrm{EC}(n_1, n_2, k) \equiv \mathrm{EC}((n_1, n_2, k))$.

$\mathrm{NT} : N \to \{\mathrm{Startflow}, \mathrm{Endflow}, \mathrm{Activity}, \mathrm{ANDSplit}, \mathrm{ANDJoin}, \mathrm{XORSplit}, \mathrm{XORJoin}\}$ ordnet jedem Knoten aus N seinen Knotentyp zu.

Knoten $n \in N$, für die gilt $\mathrm{NT}(n) = \mathit{Activity}$ werden als *Aktivitäten* bezeichnet, Knoten mit $\mathrm{NT}(n) \in \{\mathrm{ANDSplit}, \mathrm{XORSplit}\}$ als *Split-Knoten* und Knoten mit $\mathrm{NT}(n) \in \{\mathrm{ANDJoin}, \mathrm{XORJoin}\}$ als *Join-Knoten*. Knoten, die Split- oder Join-Knoten sind, werden auch unter dem Oberbegriff *Gateway-Knoten* zusammengefasst. $A = \{a \in N | \, \mathrm{NT}(a) = \mathrm{Activity}\}$ die *Aktivitätenmenge* von P.

Die Mengen der Knoten N und Kanten $\mathbb{E}$ müssen einen *blockstrukturierten* Graphen bilden. Das heißt, für den Graph $(N, \mathbb{E})$ muss gelten, dass alle Pfade, die von einem ANDSplit-Knoten bzw. XORSplit-Knoten n_1 ausgehen, in einem korrespondierenden ANDJoin-Knoten bzw. XORJoin-Knoten n_2 münden *und* alle in n_2 mündenden Pfade müssen n_1 beinhalten. Alle Knoten $n \in N$ mit $\mathrm{NT}(n) = \mathrm{Activity}$ müssen jeweils genau eine ein- und ausgehende Kante haben. Split-Knoten müssen genau eine eingehende und mindestens eine ausgehende Kante haben. Join-Knoten müssen entsprechend mindestens eine eingehende und genau einen ausgehende Kante besitzen.

Ist n_1 der korrespondierende Split-Knoten zum Join-Knoten n_2 ordnen die Funktionen Split und Join die Gateway-Knoten n_1 und n_2 einander zu: $\mathrm{Split} : N \to N : \mathrm{Split}(n_2) = n_1$ bzw. $\mathrm{Join} : N \to N : \mathrm{Join}(n_1) = n_2$. Die Knoten n_1 und n_2 bilden einen *Block*. Ein *AND-Block* ist

ein Block mit $(\mathrm{NT}(n_1), \mathrm{NT}(n_2)) = (\mathrm{ANDSplit}, \mathrm{ANDJoin})$. Analog dazu ist ein *XOR-Block* ein Block mit $(\mathrm{NT}(n_1), \mathrm{NT}(n_2)) = (\mathrm{XORSplit}, \mathrm{XORJoin})$.

In jedem Prozessschema muss jeweils genau ein Knoten enthalten sein, der keinen Vorgänger bzw. Nachfolger hat. Dieser wird als *Startknoten* bzw. *Endknoten* bezeichnet. Der Startknoten $s \in N$ bzw. Endknoten $e \in N$ hat den Knotentyp $\mathrm{NT}(s) = \mathrm{Startflow}$ bzw. $\mathrm{NT}(e) = \mathrm{Endflow}$.

Es gilt außerdem: $\forall (n_1, n_2, k) \in \mathbb{E} : \mathrm{NT}(n_1) \neq XORSplit \Rightarrow \mathrm{EC}(n_1, n_2, k) = \mathrm{TRUE}$. Das heißt, nur Kanten, die von einem XORSplit-Knoten ausgehen, können eine Übergangsbedingung $\neq \mathrm{TRUE}$ haben.

Schließlich muss die Disjunktion aller ausgehenden Kanten eines XORSplit-Knotens TRUE ergeben: $\bigvee_{(n_1,n_2,k)\in\mathbb{E}} \mathrm{EC}(n_1, n_2, k) = TRUE$, $\forall n_1 \in N : \mathrm{NT}(n_1) = XORSplit$.

Abbildung 2.1 zeigt beispielhaft ein Prozessschema. In diesem sind die Knoten n_2 und n_4 durch *zwei* Kanten verbunden mit den Nummern 3 und 4. Die Erweiterung der Kantenmenge für Mehrfachkanten ist für die Definition der Vereinfachungs-Operatoren in Kapitel 3 relevant.

Die jeweilige Semantik von Startflow, Endflow, Activity, ANDSplit, ANDJoin, XORSplit, XORJoin wird hier nicht definiert, weil der Ausführungsaspekt in dieser Arbeit nicht berücksichtigt werden soll. Es lässt sich aber zusammenfassen, dass ANDSplit- und ANDJoin-Knoten eine nebenläufige Ausführung und XORSplit- und XORJoin-Knoten eine bedingte bzw. alternative Ausführung kapseln. Die Prozessausführung wird beim Startknoten gestartet und wird beendet, wenn der Endknoten der nächste auszuführende Knoten ist. Knoten vom Typ Activity ist ein sogenanntes *Aktivitätentemplate* zugeordnet, das festlegt, was bei der Ausführung einer Aktivität passieren soll (beispielsweise ein Formular anzeigen oder ein Datenbankzugriff).

In Definition 2 werden einige Bezeichnungen und Operatoren vereinbart für den formalen Umgang mit den Relationen eines Prozessschemas.

Definition 2. ($\bullet n$, $n\bullet$, Pfad, $n_1 \prec n_2$, $n_1 \preceq n_2$, $\star n$ und $n\star$).

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema und $n, n_1, n_2 \in N$.

Wenn n_1 und n_2 zueinander in der Relation $(n_1, n_2, k) \in \mathbb{E}$ stehen, gilt: $\bullet n_2 = n_1$ (*Vorgänger*) und $n_1\bullet = n_2$ (*Nachfolger*).

Ein *Pfad* zwischen n_1 und n_2 ist eine Folge von Kanten aus $\mathbb{E}$ der Form $((n_1, m_1, k_1) \in \mathbb{E}, (m_1, m_2, k_2) \in \mathbb{E}, \ldots, (m_l, n_2, k_l) \in \mathbb{E})$ mit $l \in \mathbb{N}$.

$n_1 \prec n_2$ gilt genau dann, wenn mindestens ein Pfad von n_1 nach n_2 existiert.

$n_1 \preceq n_2$ gilt genau dann, wenn $n_1 \prec n_2$ oder $n_1 = n_2$.

$\star n = \{m | m \in N \wedge m \prec n\}$ ist die *Menge aller direkten und indirekten Vorgänger* von n.

$n \star = \{m | m \in N \wedge n \prec m\}$ ist die *Menge aller direkten und indirekten Nachfolger* von n.

Falls aus dem Kontext nicht ersichtlich ist, auf welche Kantenmenge $\mathbb{E}$ sich die Operatoren beziehen, kann diese mit Klammern annotiert werden: $(\bullet n_1)_{\mathbb{E}}$, $(n_2 \bullet)_{\mathbb{E}}$, $(n_1 \prec n_2)_{\mathbb{E}}$, $(n_1 \preceq n_2)_{\mathbb{E}}$, $(\star n)_{\mathbb{E}}$ und $(n \star)_{\mathbb{E}}$.

Die Definitionen 3-6 helfen beim Umgang mit mehrfachen Kanten zwischen zwei Knoten. Der Term *einfache Kantenmenge* E von $\mathbb{E}$ wird in Fällen verwendet, wenn sich eine Aussage auf *alle* Kanten zwischen zwei Knoten bezieht oder nur *eine* Kante zwischen zwei Knoten existiert. E entspricht der üblichen Kantenmenge (ohne Kantennummern).

Definition 3. (einfache Kantenmenge, einfaches Prozessschema).

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema mit der Mehrfachkantenmenge $\mathbb{E} \subset N \times N \times \mathbb{N}$.

Dann ist $E = \{(n_1, n_2) | (n_1, n_2, k) \in \mathbb{E}\} \subset N \times N$ die *einfache Kantenmenge* von $\mathbb{E}$.

Ferner ist $P' = (N, E, \mathrm{EC}, \mathrm{NT})$ ein *einfaches Prozessschema.*

Die Anzahl der Kanten zwischen zwei Knoten wird von der Funktion η berechnet. Wenn beispielsweise $\mathbb{E} = \{(n_1, n_2, 1), (n_1, n_2, 2)\}$ eine Mehrfachkantenmenge ist (n_1 und n_2 sind Knoten), dann gilt $\eta(n_1, n_2) = 2$.

Definition 4. (η).

Sei $\mathbb{E} \subseteq N \times N \times \mathbb{N}$ eine Mehrfachkantenmenge mit der einfachen Darstellung E.

Dann ist $\eta_{\mathbb{E}} : E \to \mathbb{N} : \eta_{\mathbb{E}}((n_1, n_2)) = |\{(n_1, n_2, k) | (n_1, n_2, k) \in \mathbb{E}\}|$ die Anzahl der Kanten zwischen den Knoten n_1 und n_2. Mögliche *Schreibweisen* sind $\eta_{\mathbb{E}}(e) \equiv \eta_{\mathbb{E}}((n_1, n_2)) \equiv \eta_{\mathbb{E}}(n_1, n_2)$.

Wird auf *alle* Kanten zwischen zwei Knoten Bezug genommen, bildet die Funktion $\mathrm{all}_{\mathbb{E}}$ ein Knotenpaar auf die Menge aller Kanten zwischen den Knoten ab (Definition 5).

Definition 5. ($\mathrm{all}_{\mathbb{E}}$ und $(n_1, n_2)_{\mathbb{E}}$)

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema und $2^{\mathbb{E}}$ die Potenzmenge von $\mathbb{E}$.

Dann ist $\mathrm{all}_{\mathbb{E}} : N \times N \to 2^{\mathbb{E}} : \mathrm{all}_{\mathbb{E}}(n_1, n_2) = \{(n_1, n_2, k) | (n_1, n_2, k) \in \mathbb{E}\}$ die Menge aller Kanten zwischen n_1 und n_2. Eine mögliche *Schreibweise* ist $(n_1, n_2)_{\mathbb{E}} = \mathrm{all}_{\mathbb{E}}(n_1, n_2)$.

Gibt es nur eine einfache Kante zwischen zwei Knoten, ordnet id der Kante ihre Kantennummer zu, während single einer Kante, die nur durch ihre Quelle und Senke identifiziert wird (ohne Kantennummer), die um ihre Kantennummer erweiterte Kante aus $\mathbb{E}$ zuordnet.

Definition 6. (id, single, $\langle n_1, n_2 \rangle_{\mathbb{E}}$).

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema mit der einfachen Darstellung E für $\mathbb{E}$ und $\eta_{\mathbb{E}}(n_1, n_2) = 1$ (das heißt, genau eine Kante zwischen den beiden Knoten) für $(n_1, n_2, k) \in \mathbb{E}$.

$\mathrm{id} : E \to \mathbb{N} : \mathrm{id}(n_1, n_2) = k$ ordnet der Kante $e = (n_1, n_2)$ ihre Kantennummer zu. Mögliche *Schreibweisen* sind $\mathrm{id}(e) \equiv \mathrm{id}((n_1, n_2)) \equiv \mathrm{id}(n_1, n_2)$.

$\mathrm{single}_{\mathbb{E}} : E \to \mathbb{E} : single_{\mathbb{E}}((n_1, n_2)) = (n_1, n_2, k)$ ergänzt eine Kante, die nur durch Vorgänger und Nachfolger identifiziert wird, um ihre zugehörige Kantennummer. Mögliche *Schreibweisen* sind $\langle e \rangle_{\mathbb{E}} \equiv \langle (n_1, n_2) \rangle_{\mathbb{E}} \equiv \langle n_1, n_2 \rangle_{\mathbb{E}} \equiv single_{\mathbb{E}}((n_1, n_2))$.

Zur Erhaltung der Blockstruktur bei Sichtenerzeugungs- und -modifikations-Operationen wird ferner das Konzept des SESE-Fragments benötigt – einem Teilgraphen mit genau einem Ein- und Ausgang. Definition 7 legt die genaue Beduteung fest.

Definition 7. (SESE).

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema und $X \subset N$. Der durch X induzierte Teilgraph P' ist ein *SESE-Fragment* (Single Entry, Single Exit) genau dann, wenn P' verbunden ist und genau eine eingehende Kante e_1 und eine ausgehende Kante e_2 hat, die ihn mit P

verbindet. Der Knoten aus X, der mit der eingehenden Kante e_1 (ausgehenden Kante e_2) verbunden ist, wird als *Eingangsknoten* (*Ausgangsknoten*) bezeichnet.

Ebenfalls zur Erhaltung der Blockstruktur ist das kleinste gemeinsame SESE-Fragment einer Menge von Knoten relevant bzw. dessen Ein- und Ausgang.

Definition 8. ($\mathrm{LeastCommonSESE}(X)$, $\mathrm{LeastCommonEntry}(X)$, $\mathrm{LeastCommonExit}(X)$)

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema und $X \subset N$.

$Y = \mathrm{LeastCommonSESE}(X)$ ist die Menge der Knoten aus N, die das *kleinste gemeinsame SESE-Fragment* bilden, dass alle Knoten aus X enthält. Das heißt, es gibt kein SESE-Fragment $Y' \subset N$ mit $|Y'| < |Y|$, das alle Knoten aus X enthält.

Ferner ist $s_1 = \mathrm{LeastCommonEntry}(X)$ der *Eingangsknoten* von $\mathrm{LeastCommonSESE}(X)$ und $s_2 = \mathrm{LeastCommonExit}(X)$ ist der *Ausgangsknoten* von $\mathrm{LeastCommonSESE}(X)$.

In Kapitel 5 zeigen die Algorithmen 5.3 und 5.4 jeweils eine Möglichkeit, um das kleinste gemeinsame SESE-Fragment bzw. den kleinsten gemeinsamen SESE-Ausgangsknoten zu bestimmen (der Algorithmus für den Eingangsknoten ist analog dazu).

2.2 Prozesssichten

Eine Prozesssicht (siehe Definition 9) basiert auf einem Prozessschema (oder einer anderen Prozesssicht) und blendet ausgewählte Aktivitäten aus oder fasst sie zu abstrakten Aktivitäten zusammen. Das Ausblenden geschieht durch *Reduktion* von Aktivitäten, Zusammenfassen durch *Aggregation*.

Definition 9. (Prozesssicht).

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema.

Eine *Prozesssicht* ist definiert als $V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ mit der Knotenmenge N' und Kantenmenge $\mathbb{E}' \subset N' \times N' \times \mathbb{N}$, die durch Reduktion oder Aggregation von Knoten aus dem Prozessschema P abgeleitet sind.

$\mathrm{EC}' : \mathbb{E}' \to \mathrm{Cond} \cup \{\mathrm{TRUE}\}$ gibt die dafür angepassten Übergangsbedinungen an (die nötige Anpassung wird bei den Vereinfachungs-Operatoren in Kapitel 3 beschrieben; ohne Vereinfachungen ist $\mathrm{EC}'(e) = \mathrm{EC}(e)\, \forall e \in \mathbb{E}'$).

$\mathrm{NT}' : N' \to \{\mathrm{Startflow}, \mathrm{Endflow}, \mathrm{Activity}, \mathrm{ANDSplit}, \mathrm{ANDJoin}, \mathrm{XORSplit}, \mathrm{XORJoin}, \mathrm{AggregationNode}\}$ ordnet den Knoten ihren entsprechenden Typ zu. Ergänzend zu den Knotentypen in Definition 1 wird der Typ AggregationNode verwendet für Aggregationsknoten, in denen in N vorhandene Knoten zu einem Aggregationsknoten zusammengefasst werden (die *Komponenten* des Aggregationsknotens).

$C \subset \{\alpha \in N' | \, \mathrm{NT}'(\alpha) = \mathrm{AggregationNode}\} \times \{c \in N | \, \mathrm{NT}(c) \neq \mathrm{AggregationNode}\} \subset N' \times N$ (*Aggregationskorrespondenz* oder *Korrespondenz*) ist eine injektive Relation, die jedem Aggregationsknoten α seine Komponentenknoten c aus dem Prozessschema zuordnet. Das heißt, für alle $(\alpha, c) \in C$ gilt, dass α ein Aggregationsknoten ist und c dagegen ein Knoten aus N. Komponentenknoten können niemals selber Aggregationsknoten sein, was bei View-Hierarchien mit mehr als einer Ebene von Bedeutung ist, da dort Aggregationsknoten aus einer darüberliegenden Ebene aggregiert werden können. Die Aggregations-Operation *AggregateSESE* wird in Kapitel 2.3.2 so definiert, dass bei der Aggregation von Aggregationsknoten immer deren Komponentenknoten übernommen werden statt des Aggregationsknotens („Flachklopfen“).

Für die Struktur des Graphs $(N', \mathbb{E}')$ gelten schließlich dieselben Bedingungen wie für Prozessschemas in Definition 1. Für Aggregationsknoten gelten in diesem Kontext die gleichen Bedingungen wie für Aktivitäten.

Im Folgenden werden in Definition 10 zwei Relationen eingeführt, die die Korrespondenz zwischen Knoten mit beliebigem Typ aus einem Prozessschema und einer Prozesssicht formal beschreiben. Das heißt, die Korrespondenz *aller* Knotentypen, im Vergleich zur Aggregationskorrespondenz C, die nur die Korrespondenz zwischen Aggregations- und Komponentenknoten beinhaltet. Knoten, die nicht in einer Aggregationsbeziehung stehen, werden eins zu eins einander zugeordnet, außer im Falle der Reduktion, in dem der entsprechende Schema-Knoten keinem Knoten aus der Sicht zuordenbar ist.

Definition 10. ($\uparrow_V^P$ und $\downarrow_V^P$).

Sei $V = V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ eine Prozesssicht auf das Prozessschema $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$.

$\uparrow_V^P \subset N' \times N$ (*„up“*) ist eine injektive Relation, die jedem Knoten aus der Prozesssicht den oder die (im Falle von Aggregationsknoten) korrespondierenden Knoten aus dem ursprünglichen Prozessschema zuordnet:

$$\uparrow_V^P = \{(n', n') | n' \in N' : \mathrm{NT}'(n') \neq \mathrm{AggregationNode}\} \cup C$$

$\downarrow_V^P \subset N \times N'$ (*„down“*) ist eine surjektive Relation, die jedem Knoten aus dem ursprünglichen Prozessschema eventuell (wegen Reduktion) den korrespondierenden Knoten aus einer Prozesssicht zuordnet:

$$\downarrow_V^P = \{(n', n') | n' \in N' : \mathrm{NT}'(n') \neq \mathrm{AggregationNode}\} \cup \{(n, \alpha) | (\alpha, n) \in C\}$$

$\uparrow_V^P [n']$ ist eine *Schreibweise* für $\{n | (n', n) \in \uparrow_V^P\}$. (Diese Menge enthält entweder einen oder *mehrere* Knoten (falls *aggregiert*)).

$\downarrow_V^P [n]$ ist eine *Schreibweise* für $\{n' | (n, n') \in \downarrow_V^P\}$. (Diese Menge enthält entweder einen oder *keinen* Knoten (falls *reduziert*)). Falls $\downarrow_V^P [n] \neq \emptyset$ ist auch die Schreibweise $n' = \downarrow_V^P [n]$ zulässig (wobei dann für n' gilt: $\exists! n' \in N' \exists! n \in N : (n, n') \in \downarrow_V^P [n]$). Einfach ausgedrückt ist $\downarrow_V^P [n]$ dann der zu n korrespondierende Knoten.

$\uparrow_V^P$ bietet eine Sichtweise auf V, in der alle Aggregationsknoten durch ihre Komponentenknoten ersetzt sind. Umgekehrt bietet $\downarrow_V^P$ eine Sichtweise auf P, in der alle Komponentenknoten durch ihre Aggregationsknoten (aus V) ersetzt sind. Man beachte aber, dass *AggregateSESE* (Kapitel 2.3.2) die Gateway-Knoten auslässt. Das bedeutet, die Aggregationskorrespondenz C enthält keine Gateway-Knoten, wodurch auch $\uparrow_V^P$ keine Gateway-Knoten enthält und somit $\uparrow_V^P [\alpha]$ nicht die Umkehroperation von *AggregateSESE* ist. In Kapitel 2.3.3 wird die eigentliche Umkehroperation *UnaggregateSESE* beschrieben, welche Gateway-Knoten mithilfe des ursprünglichen Prozessschemas rekonstruiert.

Nützlich erweist sich die Kombination von $\mathrm{LeastCommonEntry}$ bzw. $\mathrm{LeastCommonExit}$ mit $\uparrow_V^P$: ist n' ein Aggregationsknoten in einer Sicht, entspricht $\mathrm{LeastCommonEntry}(\uparrow_V^P [n'])$ dem Eingangsknoten und $\mathrm{LeastCommonExit}(\uparrow_V^P [n'])$ dem Ausgangsknoten des mit n' korrespondierenden SESE-Fragments im Prozessschema. Ist n' kein Aggregationsknoten, entsprechen beide einfach dem korrespondierenden Knoten aus dem Prozessschema. Abbildung 2.2 zeigt diesen Sachverhalt beispielhaft. Beachtet werden sollte dabei auch, dass zwar das SESE-Fragment $n_1 \ldots n_4$ zu α zusammengefasst wird – das mit α korrespon-

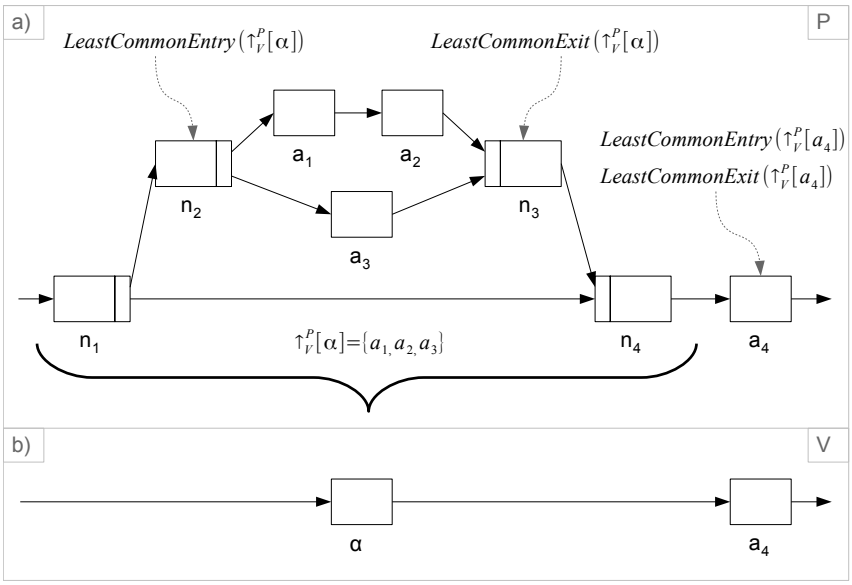

Abbildung 2.2: Beispiel für LeastCommonEntry und -Exit

dierende kleinste gemeinsame SESE-Fragment aber $n_2 \dots n_3$ ist, da α nur mit der Menge $\{a_1, a_2, a_3\}$ korrespondiert.

Um bei Mehrfachkantenmengen das Hinzufügen und Entfernen von Kanten zu erleichtern – vor allem mit Blick auf Kantennummern – werden in Definition 11 die Hilfsfunktionen $\mathrm{AddEdges}$ und $\mathrm{RemoveEdges}$ definiert, die es erlauben, eine Menge von Kanten ohne Kantennummern zu einer Mehrfachkantenmenge hinzuzufügen bzw. daraus zu entfernen, ohne dass die Kantennummern explizit angegeben werden müssen.

Definition 11. ($\mathrm{AddEdges}$, $\mathrm{RemoveEdges}$ und $\oplus$ bzw. $\ominus$).

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema und $F = \{(n_1, m_1), \dots, (n_k, m_k) | k \in \mathbb{N}\} \subseteq N \times N$ eine Menge von Kanten.

$\mathrm{AddEdges}(\mathbb{E}, F)$ fügt die in F enthaltenen Kanten zu $\mathbb{E}$ hinzu, wobei die Kanten um ihre Kantennummer erweitert werden:

$$\mathrm{AddEdges}(\mathbb{E}, F) = \mathbb{E} \cup \{(n_1, m_2, j+1), \dots, (n_k, m_k, j+k)\}.$$

Dabei ist $j = \max\limits_{(p,q,i) \in \mathbb{E}} i$ die größte in $\mathbb{E}$ vorkommende Kantennummer.

$\mathrm{RemoveEdges}(\mathbb{E}, F)$ entfernt *alle* durch F definierten Kanten aus $\mathbb{E}$:

$$\mathrm{RemoveEdges}(\mathbb{E}, F) = \mathbb{E} \setminus \mathrm{all}_{\mathbb{E}}(n_1, m_1) \setminus \ldots \setminus \mathrm{all}_{\mathbb{E}}(n_k, m_k).$$

$\oplus$ und $\ominus$ sind Operatorschreibweisen für $\mathrm{AddEdges}$ bzw. $\mathrm{RemoveEdges}$:

$$\begin{aligned} \mathbb{E} \oplus F &= \mathrm{AddEdges}(\mathbb{E}, F) \\ \mathbb{E} \ominus F &= \mathrm{RemoveEdges}(\mathbb{E}, F) \end{aligned}$$

Nachdem Prozesssichten und zugehörige Hilfsmittel definiert wurden, können im Folgenden Operatoren zur Erzeugung von Sichten auf ein Prozessschema festgelegt werden.

2.3 Operatoren zur Sichtenerzeugung

Sichten werden durch Reduktion und Aggregation von Aktivitäten erzeugt. Die elementaren Operatoren dafür sind $\mathrm{ReduceActivity}$ und $\mathrm{AggregateSESE}$ [8]. $\mathrm{UnaggregateSESE}$ ist der Umkehroperator von $\mathrm{AggregateSESE}$. Diese drei Operatoren werden in den Kapiteln 2.3.1-2.3.3 beschrieben. In Kapitel 2.3.4 werden weitere Operatoren zur Sichtenerzeugung angesprochen, die u.a. auf den genannten Elementar-Operatoren aufbauen können.

Im Folgenden sei $V = V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ eine Prozesssicht auf das Prozessschema $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$.

2.3.1 ReduceActivity

$\mathrm{ReduceActivity}(a)$ blendet die Aktivität a aus einer Sicht aus, indem die Aktivität samt ein- und ausgehender Kante aus der Prozesssicht entfernt und durch eine neue Kante ersetzt wird. Abbildung 2.3 veranschaulicht diesen Vorgang grafisch. Wenn $a \in N' : \mathrm{NT}'(a) \in \{\mathrm{Activity}, \mathrm{AggregationNode}\}$ die zu reduzierende Aktivität ist, ergibt sich die neue Knotenmenge N'' in $V(P)$ durch:

$$N'' = N' \setminus \{a\}.$$

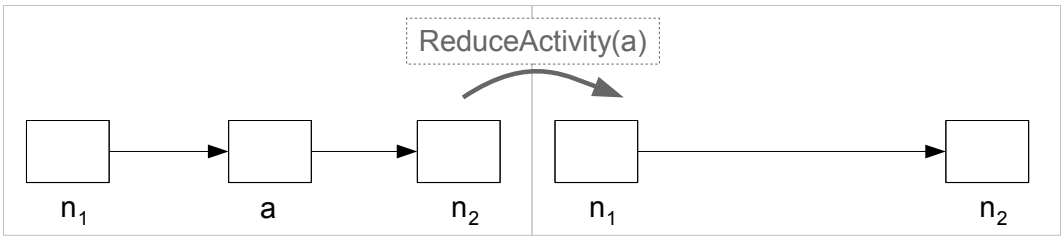

Abbildung 2.3: Reduktion einer Aktivität

Die neue Kantenmenge $\mathbb{E}''$ entsteht durch Entfernen der ein- und ausgehenden Kanten von a und Hinzufügen einer Kante, die den Vorgänger und Nachfolger von a verbindet:

$$\mathbb{E}'' = \mathbb{E}' \ominus \{(\bullet a, a), (a, a\bullet)\} \oplus \{(\bullet a, a\bullet)\}.$$

Eine eventuell vorhandene Kantenbedingung muss übernommen werden (für alle anderen Kanten bleibt $\mathrm{EC}'' = \mathrm{EC}'$):

$$\mathrm{EC}''(\langle \bullet a, a\bullet \rangle) = \mathrm{EC}'(\langle \bullet a, a \rangle).$$

Falls a ein Aggregationsknoten ist, das heißt $\mathrm{NT}'(a) = \mathrm{AggregationNode}$, muss die Korrespondenz zwischen a und seinen Komponentenknoten entfernt werden:

$$C' = C \setminus \{(a, c) | (a, c) \in C \wedge c \in N\}.$$

2.3.2 AggregateSESE

$\mathrm{AggregateSESE}(S) \rightarrow \alpha$ fasst die Aktivitäten des SESE-Fragments S zu einem einzigen Aggregationsknoten zusammen. In der Prozesssicht werden *alle* Knoten des SESE-Fragments samt den sie verbindenden Kanten entfernt, und durch einen Aggregationsknoten ersetzt, indem alle Knoten $n \in S$ mit $\mathrm{NT}'(n) = \mathrm{Activity}$ zusammengefasst werden. Insbesondere werden *Gateway-Knoten* ausgelassen (welche von UnaggregateSESE mithilfe des CPMs rekonstruiert werden können, siehe Kapitel 2.3.3).

Sei α der Aggregationsknoten mit $NT''(\alpha) = \mathrm{AggregationNode}$, durch den S ersetzt wird, und $s_1, s_2 \in N'$ der Vorgänger- bzw. Nachfolgerknoten des SESE-Fragments S. Die Kno-

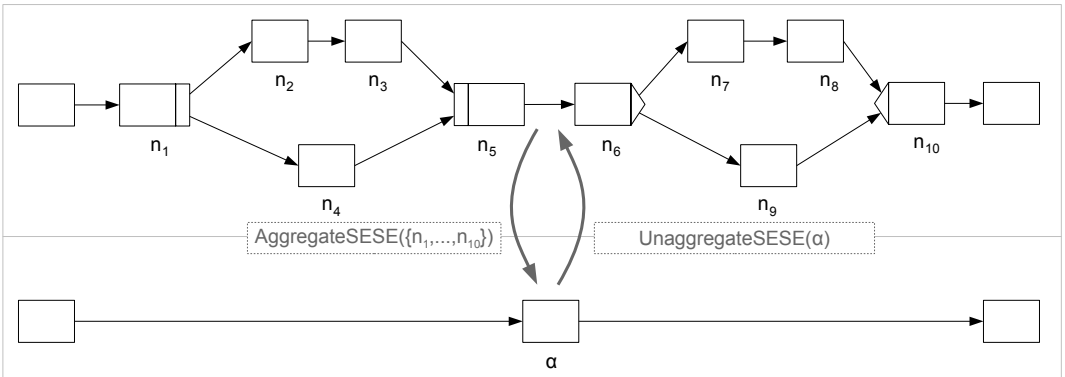

Abbildung 2.4: Aggregation eines SESE-Fragments

tenmenge N' wird angepasst zu N'':

$$N'' = N' \setminus S \cup \{\alpha\}.$$

Aus der Kantenmenge werden die Kanten des SESE-Fragments entfernt und die ein- und ausgehende Kante von α eingefügt:

$$\mathbb{E}'' = \mathbb{E}' \setminus \{(n_1, n_2, k) \in \mathbb{E}' : (n_1 \in S \vee n_2 \in S)\} \oplus \{(s_1, \alpha), (\alpha, s_2)\}.$$

Die Kantenbedingung der eingehenden Kante s_1 von S wird übernommen (für alle anderen Kanten bleibt $\mathrm{EC}'' = \mathrm{EC}'$):

$$\mathrm{EC}''(\langle s_1, \alpha\rangle_{\mathbb{E}''}) = \mathrm{EC}'(\langle s_1, s_1\bullet\rangle_{\mathbb{E}'}).$$

Wenig komplexer ist die Änderung der Aggregationskorrespondenz C, weil auch berücksichtigt werden muss, dass S neben Aktivitäten auch Aggregationsknoten enthalten kann. Im letzteren Fall werden statt des Aggregationsknotens seine Komponentenknoten übernommen („Flachklopfen“).

$$\begin{aligned} C' = C \setminus \{&(\alpha_S, c) | \alpha_S \in S : \mathrm{NT}'(\alpha_S) = \mathrm{AggregationNode} \wedge (\alpha_S, c) \in C\} \\ &\cup \{(\alpha, s) | s \in S : \mathrm{NT}'(s) = \mathrm{Activity}\} \\ &\cup \{(\alpha, c) | (\alpha_S, c) \in C : \mathrm{NT}'(\alpha_s) = \mathrm{AggregationNode}\} \end{aligned}$$

Alternativ lässt sich dies mithilfe der Relation $\uparrow_V^P$ ausdrücken, welche dem Knoten in einer Sicht die korrespondierenden Knoten des Prozessschemas zuordnet – insbesondere auch

jedem Aggregationsknoten seine Komponentenknoten (siehe Definition 10):

$$C' = C \setminus \{(\alpha_S, c) | (\alpha_S, c) \in C : \alpha_S \in S\} \\ \cup \{(\alpha, s) \in \uparrow_V^P [s] | s \in S : \mathrm{NT}'(s) \in \{\mathrm{Activity}, \mathrm{AggregationNode}\}\}.$$

Die Umkehroperation von $\mathrm{AggregateSESE}$ wird anschließend in Kapitel 2.3.3 beschrieben.

2.3.3 UnaggregateSESE

$\mathrm{UnaggregateSESE}(\alpha)$ ist die Umkehroperation von $\mathrm{AggregateSESE}$ und ersetzt in einer Prozesssicht einen Aggregationsknoten $\alpha \in N' : \mathrm{NT}(\alpha) = \mathrm{AggregationNode}$ durch das entsprechende SESE-Fragment. Weil $\mathrm{AggregateSESE}$ die Gateway-Knoten und auch die Kanten des aggregierten SESE-Fragments nicht im Aggregationsknoten speichert, muss die Struktur des aggregierten SESE-Fragments rekonstruiert werden. Das ist deshalb möglich, weil im Prozess*schema* die ursprüngliche Struktur des aggregierten Bereichs noch enthalten ist. Das heißt, $\mathrm{UnaggregateSESE}$ rekonstruiert mithilfe Strukturinformation aus dem Prozessschema den aggregierten Bereich. Der Algorithmus für $\mathrm{UnaggregateSESE}$ geht in zwei Schritten vor: im ersten Schritt werden alle aggregierten Aktivitäten (*Komponentenknoten*) sowie alle dafür nötigen Gateway-Knoten aus dem CPM in die Sicht übernommen. Im zweiten Schritt werden alle Kanten zwischen diesen Knoten wiederhergestellt. Weil eventuell unnötige Gateway-Knoten übernommen werden, ist es angebracht, danach den Graphen zu vereinfachen und dabei die unnötigen Gateway-Knoten zu entfernen (siehe Kapitel 3).

Das gesamte zu rekonstruierende SESE-Fragment (inklusive der Gateway-Knoten) entspricht dem *kleinsten gemeinsamen SESE-Fragment* der Komponentenknoten (siehe Algorithmus 5.3 für die Ermittlung des kleinsten gemeinsamen SESE-Fragments). Ein- und Ausgangsknoten dieses kleinsten gemeinsamen SESE-Fragments legen die Grenzen des Bereichs fest, der rekonstruiert werden muss. Insbesondere ist es möglich, dass Ein- oder Ausgangsknoten Gateway-Knoten sind. Algorithmus 5.7 zeigt in Kapitel 5 den konkreten Ablauf der Rekonstruktion des SESE-Fragments.

2.3.4 Weitere Sichtenerzeugungs-Operatoren

In dieser Arbeit sind die Operatoren zur Sichtenerzeugung auf solche beschränkt, die die Blockstruktur eines Prozessschemas erhalten, vor allem mit Blick auf das Einfügen von Aktivitäten in Sichten und das Propagieren der Einfügung zum CPM und an weitere Sichten. Generell können bei der Sichtenbildung aber auch Operationen erwünscht sein, die

weniger strenge Bedingungen an die Strukturerhaltung stellen. Konkret ist in dieser Arbeit die Aggregations-Operation auf das Aggregieren eines SESE-Fragments beschränkt. Beispielsweise werden in [8] Aggregations-Operationen beschrieben, die es erlauben, eine Auswahl an Knoten zu aggregieren, die kein SESE-Fragment bilden.

Ferner sind für die praktische Anwendung durch Benutzer Sichtenerzeugungs-Operatoren notwendig, die eine höhere Abstraktion bieten. Zum einen ist $\mathrm{ReduceActivity}$ auf eine einzige Aktivität beschränkt. Praktischerweise sollte es auch möglich sein, mehrere Aktivitäten auf einmal zu reduzieren. Durch sukzessive Anwendung von $\mathrm{ReduceActivity}$ auf die einzelnen Aktivitäten lässt sich jedoch eine solche Operation realisieren. Zum anderen ist es sinnvoll, Operationen anzubieten, die nicht auf der manuellen Selektion von zu reduzierenden oder zu aggregierenden Aktivitäten basieren, sondern auf Basis von Prädikaten Benutzer-spezifisch Aktivitäten automatisch ausblenden oder zusammenfassen. Knoten würden mit einem solchen Prädikat anhand ihrer Eigenschaften ausgewählt werden. Zum Beispiel könnte ein Benutzer wünschen, dass alle technischen Aktivitäten ausgeblendet werden oder nur Aktivitäten angezeigt werden sollen, an denen dieser Benutzer beteiligt ist. Ein weiteres Beispiel ist das Ausblenden oder Zusammenfassen aller bereits ausgeführten Aktivitäten. $\mathrm{ReduceActivity}$ und $\mathrm{AggregateSESE}$ bilden die Basis für solche Operationen. [8] zeigt das Zusammenspiel von Operationen, die auf verschiedenen Abstraktionsebenen angesiedelt sind.

2.4 Einfüge-Operatoren

Im Folgenden werden die grundlegenden Operatoren *InsertSerial*, *InsertParallel* und *InsertConditional* definiert, mit denen *Aktivitäten* in Prozessschemas und Sichten eingefügt werden können. Diese werden auch in der prototypischen Implementierung verwendet, um Prozesse zu modellieren (siehe Kapitel 5.2). Die Operatoren sind so definiert, dass die Bedingungen für die Struktur von Prozessschemas in Definition 1 eingehalten werden. Das heißt, es können damit nur wohlstrukturierte, azyklische Prozesse modelliert werden.

Die Einfüge-Operatoren werden hier für Prozessschemas definiert. Sie sind aber auch auf Prozesssichten anwendbar. Allerdings müssen bei letzteren konsequent die Aktualisierungs-Operatoren aus Kapitel 4 angewendet werden, um Prozessschema und Sichten konsistent zu halten.

Für alle drei Operatoren sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ das Prozessschema, in das die *Aktivität* a eingefügt werden soll. E sei die einfache Kantenmenge von $\mathbb{E}$.

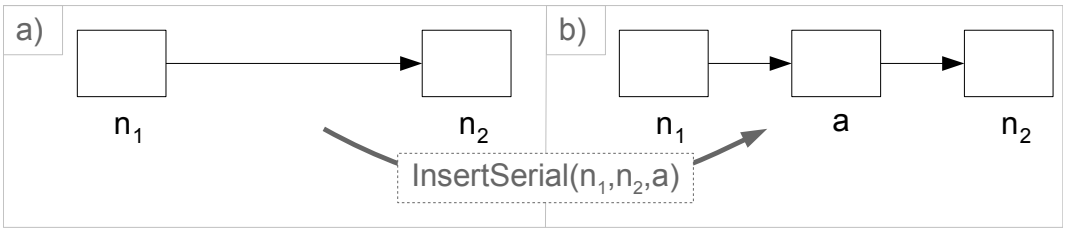

Abbildung 2.5: Serielle Einfügung.

2.4.1 InsertSerial

$\mathrm{InsertSerial}(n_1, n_2, a)$ fügt die *Aktivität* a zwischen den Knoten n_1 und n_2 ein. Vorbedingung ist $(n_1, n_2) \in E$, das heißt, n_2 muss ein direkter Nachfolger von n_1 sein. Abbildung 2.5 veranschaulicht die serielle Einfügung.

Durch Anwendung von $\mathrm{InsertSerial}$ wird die Aktivität in die Knotenmenge eingefügt: $N' = N \cup \{a\}$. Die Kante (n_1, n_2) wird ersetzt durch zwei neue Kanten, die a mit n_1 und n_2 verbinden: $\mathbb{E}' = \mathbb{E} \ominus \{(n_1, n_2)\} \oplus \{(n_1, a), (a, n_2)\}$. Außerdem wird eine eventuell vorhandene Kantenbedingung übernommen: $\mathrm{EC}'(\langle n_1, a\rangle_{\mathbb{E}'}) = \mathrm{EC}(\langle n_1, n_2\rangle_{\mathbb{E}})$. Schließlich wird der Knotentyp von a spezifiziert: $\mathrm{NT}'(a) = \mathrm{Activity}$. Für alle anderen Kanten ist $\mathrm{EC}' \equiv \mathrm{EC}$ und für alle anderen Knoten bleibt $\mathrm{NT}' \equiv \mathrm{NT}$.

Neben seriellen Einfügungen sind parallele und konditionale Einfügungen möglich, die im Folgenden beschrieben werden.

2.4.2 InsertParallel und InsertConditional

$\mathrm{InsertParallel}(S, a)$ fügt eine *Aktivität* a mit einem *AND-Block* parallel zu einem SESE-Fragment S ein. $\mathrm{InsertConditional}(S, a)$ fügt a mit einem *XOR-Block* alternativ zu S ein. Abbildung 2.6 zeigt diese Operationen beispielhaft. Teilabbildung 2.6 b) zeigt das ursprüngliche Prozessschema P mit dem SESE-Fragment S, der s_1 als Eingangs- und s_2 als Ausgangsknoten hat. Da beide Operatoren die selbe resultierende Graphstruktur erzeugen, werden sie hier gemeinsam beschrieben. Alternative Signaturen für diese Operatoren sind $\mathrm{InsertParallel}(s_1, s_2, a)$ bzw. $\mathrm{InsertConditional}(s_1, s_2, a)$ und $\mathrm{InsertParallel}(s_1, s_2, a, n_3, n_4)$ bzw. $\mathrm{InsertConditional}(s_1, s_2, a, n_3, n_4)$ mit den folgenden Defintionen für s_1, s_2, n_3 und n_4:

Sei s_1 der Eingangs- und s_2 der Ausgangsknoten des SESE-Fragments S und seien n_1 und n_2 der Vorgänger bzw. Nachfolger des SESE-Fragments, das heißt $n_1\bullet = s_1$ und $s_1\bullet = n_2$. Seien n_3 und n_4 die Gateway-Knoten, die für die Einfügung von a miteingefügt werden.

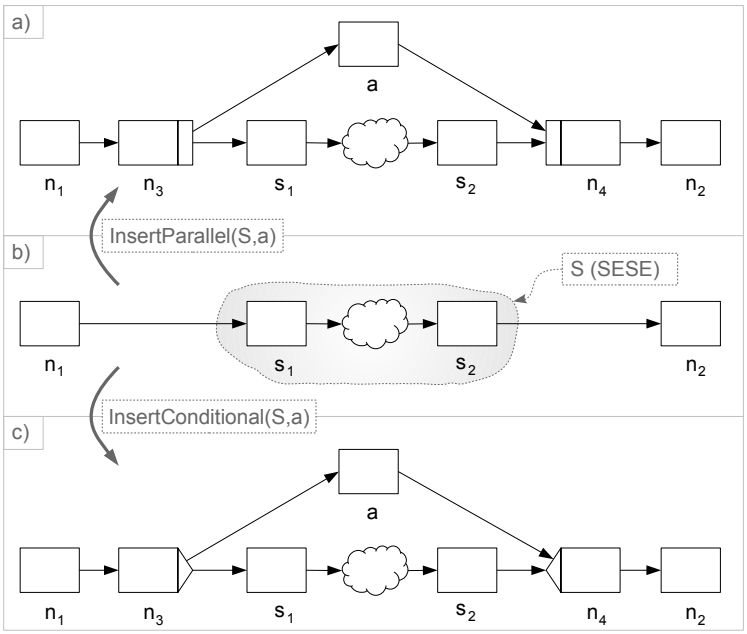

Abbildung 2.6: Paralleles (a) und bedingtes (c) Einfügen.

Für $\mathrm{InsertParallel}$ gilt:

$$(\mathrm{NT}'(n_3), \mathrm{NT}'(n_4)) = (\mathrm{ANDSplit}, \mathrm{ANDJoin}),$$

während für $\mathrm{InsertConditional}$

$$(\mathrm{NT}'(n_3), \mathrm{NT}'(n_4)) = (\mathrm{XORSplit}, \mathrm{XORJoin})$$

ist. Als nächstes werden die Knoten- und Kantenmenge angepasst:

$$\begin{aligned} N' &= N \cup \{n_1, n_2, a\} \\ \mathbb{E}' &= \mathbb{E} \ominus \{(n_1, s_1), (s_2, n_2)\} \\ &\quad \oplus \{(n_1, n_3), (n_3, s_1), (n_3, a), (a, n_4), (s_2, n_4), (n_4, n_2)\}. \end{aligned}$$

Schließlich muss eine eventuell vorhandene Kantenbedingung der eingehenden Kante des SESE-Fragments übernommen werden:

$$\mathrm{EC}'(\langle n_1, n_3 \rangle_{\mathbb{E}'}) = \mathrm{EC}(\langle n_1, s_1 \rangle_{\mathbb{E}}).$$

Bei den Parametern von $\mathrm{InsertConditional}$ ist die Kantenbedingung für den eingefügten Zweig, der a enthält nicht mit inbegriffen. Diese Bedingung der Kante (n_3, a) – die vom neuen XORSplit-Knoten n_3 ausgeht – wird separat im Anschluss definiert. Eventuell müssen weitere Kantenbedingungen angepasst werden (zum Beispiel bei der Kante (n_3, s_1) zwischen dem neuen XORSplit-Knoten und dem SESE-Zweig S).

Bilden s_1 und s_2 (Anfangs- bzw. Endknoten des SESE-Fragments) selbst bereits einen AND- oder XOR-Block, werden dieser und der neu eingefügte Block in einem nachfolgenden Vereinfachungsschritt zu einem einzigen Block verschmolzen. Dieser wird in Kapitel 3 beschrieben (SimplifyMultipleBlocksAND und SimplifyMultipleBlocksXOR). Der selbe Fall tritt ein, wenn die umgebenden Knoten n_1 und n_2 bereits einen AND- oder XOR-Block bilden. Beim Verschmelzen der XOR-Blöcke werden auch die Übergangsbedingungen der Kanten verknüpft. Daher ist es sinnvoll, die Übergangsbedingungen erst *nach* der Vereinfachung zu spezifizieren, weil diese Reihenfolge in manchen Fällen zu einfacheren Kantenbedingungen führt.

2.5 Zusammenfassung

In diesem Kapitel wurden Prozessschemas und Prozesssichten definiert zusammen mit nützlichen Begriffen und Funktionen. Des Weiteren wurden die Operatoren $\mathrm{ReduceActivity}$ zum Ausblenden und $\mathrm{AggregateSESE}$ zum Zusammenfassen von Aktivitäten mit der Umkehroperation $\mathrm{UnaggregateSESE}$ definiert neben den Operatoren zum seriellen, parallelen und alternativen Einfügen von Aktivitäten. $\mathrm{InsertSerial}$ fügt eine Aktivität zwischen zwei Knoten ein. $\mathrm{InsertParallel}$ und $\mathrm{InsertConditional}$ fügen einen Split- und einen Join-Knoten vor bzw. nach einem SESE-Fragment ein, um eine Aktivität parallel bzw. alternativ zu diesem Fragment einfügen zu können.

Es wurde ein vereinfachtes Prozessmodell gewählt. Im Vergleich zu praxistauglichen Prozessmodellen (siehe zum Beispiel [9]) werden beispielsweise Konstrukte für die *Wiederholung* von Abläufen, zur *Synchronisation* von Kontrollflüssen, zur *Fehlerbehandlung* und zur Modellierung des *Datenflusses* in dieser Arbeit nicht behandelt.

Umfangreicher wird das Modell jedoch durch Mehrfachkanten, die bei der Vereinfachung der Prozessgraphen von Bedeutung sind. Um die Arbeit mit Mehrfachkanten zu erleichtern, wurden einige Hilfsdefinitionen eingeführt.

3 Operatoren zur strukturellen Vereinfachung

Bei der Reduktion von Aktivitäten in Sichten können leere Zweige oder überflüssige Split- und Join-Knoten zurückbleiben. In Abbildung 3.1 ist dieser Vorgang beispielhaft gezeigt. Durch Reduktion der Aktivität a_2 bleibt ein leerer Zweig zwischen den Knoten n_1 und n_2 zurück. Wird die Aktivität a_5 reduziert, hat der Block der Knoten n_3 und n_4 nur noch *einen* nichtleeren Zweig. In diesem Fall sind die Gateway-Knoten n_3 und n_4 unnötig. Ein Block mit nur einem Zweig ist ein *einfacher Block*. Nach der Reduktion von a_6 ist der Block der Knoten n_6 und n_7 direkt mit dem umgebenden Block (n_5 und n_8) verbunden. Die Vereinfachungs-Operatoren vereinfachen Prozesssgraphen, indem sie unnötige Kontrollfluss-Elemente *entfernen* (im Beispiel die leeren Zweige zwischen n_1 und n_2 bzw. n_3 und n_4 sowie die Gateway-Knoten n_3 und n_4) oder *zusammenfassen* (die Gateway-Knoten n_6 und n_7 können im Beispiel mit den Gateway-Knoten n_5 und n_8 zusammengefasst werden).

Alle Vereinfachungsoperatoren sind über Parameter steuerbar. Das heißt, ein Parameter steuert jeweils, ob und wie eine Vereinfachungs-Operation angewendet wird. Die Bezeichnungen *SimplifyEmptyBranchesAND*, *SimplifyEmptyBranchesXOR*, *SimplifyMultipleBlocksAND*, *SimplifyMultipleBlocksXOR*, *SimplifyBlocksAND* und *SimplifyBlocksXOR* sind zugleich die Namen der elementaren Operatoren und die Namen der entsprechenden Parameter, die den jeweiligen Operator steuern. Bei Parametern mit den möglichen Werten *Yes* und *No* bedeuted *Yes* die entsprechende Vereinfachung anzuwenden. *No* heißt, diese zu deaktivieren. *SimplifyMultipleBlocksAND* bzw. *-XOR* und *SimplifyBlocksAND* bzw. *-XOR* werden unter dem Oberbegriff *Block-Vereinfachungs-Operatoren* zusammengefasst.

Die genannten Operatoren sind für sich jeweils auf einen einzelnen Block anwendbar und betrachten dabei diesen Block und teilweise den umgebenden Block. Ein Algorithmus, der den ganzen Prozessgraphen vereinfachen soll, wendet die Vereinfachungs-Operatoren nach dem *Inside-Out-Prinzip* auf die einzelnen Blöcke an; das heißt, die innersten Blöcke werden zuerst betrachtet und nach und nach die umgebenden. Dabei muss eine Auswertungsreihenfolge eingehalten werden: es müssen zuerst die leeren Zweige entfernt werden (SimplifyEmptyBranchesAND bzw. -XOR) bevor die Gateway-Knoten einfacher Blöcke beseitigt werden können (SimplifyBlocksAND bzw. -XOR), weil durch das Entfernen leerer Zweige weitere einfache Blöcke entstehen können. Beispielsweise hat in Abbildung 3.1 b) der Block der Gateway-Knoten n_3 und n_4 erst *nach* dem Entfernen der Kante (n_3, n_4) nur

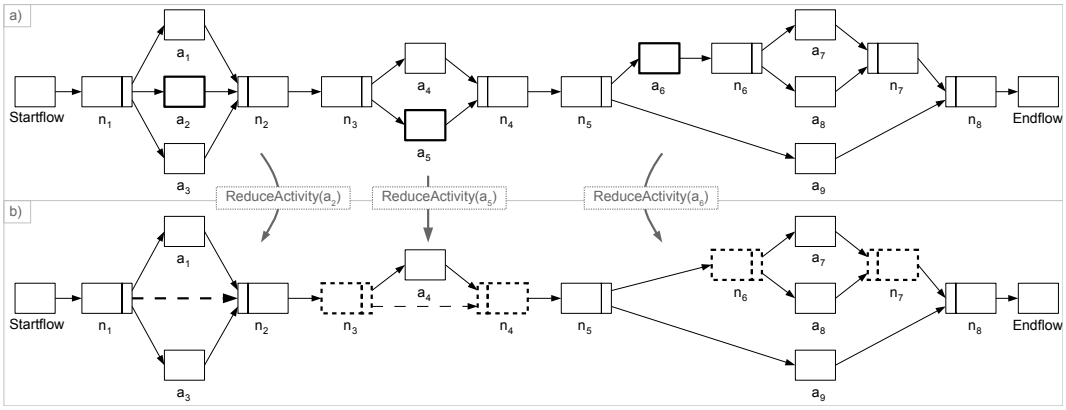

Abbildung 3.1: Durch Reduktion der Aktivitäten a_2, a_5 und a_6 bleiben leere Zweige und unnötige Gateway-Knoten zurück.

noch einen Zweig). Die elementaren Vereinfachungs-Operatoren werden zu einem höherwertigen Operator $\mathrm{Simplify}(P)$ zusammengefasst, der unter Verwendung des beschriebenen Inside-Out-Algorithmus das Prozessschema (oder die Sicht) P vereinfacht.

3.1 Voraussetzungen

Im Folgenden sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema mit der Knotenmenge N, der Kantenmenge $\mathbb{E}$, der Kantenbedingungsfunktion EC und der Knotentypenfunktion NT. E sei die einfache Kantenmenge von $\mathbb{E}$.

Die Vereinfachungs-Operatoren werden für Prozessschemas definiert, sind aber auch auf Prozesssichten anwendbar, da die Aggregationskorrespondenz C bei der Vereinfachung keine Rolle spielt. Aggregationsknoten werden bei der Vereinfachung so behandelt wie Aktivitäten. Ferner ist ein *leerer Zweig* eine Kante $(a_1, a_2, k) \in \mathbb{E}$, für die gilt: $(\mathrm{NT}(a_1), \mathrm{NT}(a_2)) \in \{(\mathrm{ANDSplit}, \mathrm{ANDJoin}), (\mathrm{XORSplit}, \mathrm{XORJoin})\}$, also eine Kante, die einen Split- direkt mit einem Join-Knoten verbindet.

3.2 SimplifyEmptyBranchesAND

SimplifyEmptyBranchesAND entfernt leere Zweige in AND-Blöcken und wird durch den gleichnamigen Parameter gesteuert. Mögliche Parameterwerte sind $\{\mathrm{None}, \mathrm{All}\} \cup \mathbb{N}$. *None* (*All*) bedeuted, dass keine (nach Möglichkeit alle) leeren Zweige in AND-Blöcken entfernt

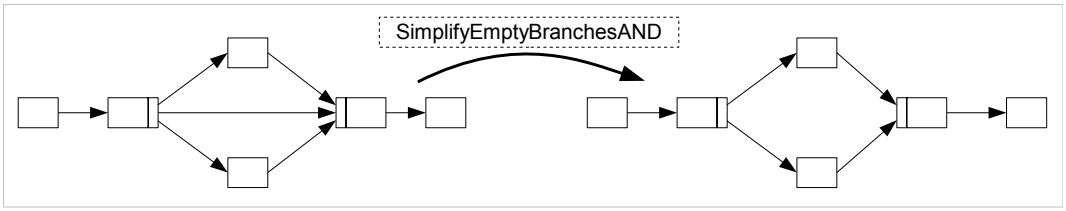

Abbildung 3.2: Beispiel für *SimplifyEmptyBranchesAND*.

werden. Eine positive Ganzzahl gibt dagegen an, wie viele leere Zweige nach Möglichkeit beibehalten werden sollen. Abbildung 3.3 a) zeigt ein Beispiel mit dem Parameterwert All.

Hat ein AND-Block mit dem Split-Knoten $p \in N$ und dem Join-Knoten $q \in N$ insgesamt $m = |\{(p, r, k)|(p, r, k) \in \mathbb{E}\}|$ Zweige von denen $n = \eta_{\mathbb{E}}(p, q)$ leere Zweige sind ($n < m$), dann gelten die Vereinfachungen aus Tabelle 3.1.

SimplifyEmptyBranchesAND	Anzahl entfernter leerer Zweige p
None	$p = 0$
All	$p = \begin{cases} n-1 & n = m \\ n & sonst \end{cases}$
$l \in \mathbb{N}$	$p = \begin{cases} n-1 & n = m \\ n & n < l \\ l & sonst \end{cases}$

Tabelle 3.1: Anzahl entfernter leerer Zweige.

Man beachte, dass bei ausschließlich leeren Zweigen mindestens eine Kante beibehalten wird, damit der Graph verbunden bleibt. *SimplifyBlocksAND* entfernt anschließend einen solchen Block, wenn nur noch eine Kante vorhanden ist (siehe Kapitel 3.4.1).

3.3 SimplifyEmptyBranchesXOR

SimplifyEmptyBranchesXOR fasst leere Zweige in XOR-Blöcken zu einem einzigen leeren Zweig zusammen (siehe Abbildung 3.3). Mögliche Parameterwerte für *SimplifyEmptyBranchesXOR* sind $\{\mathrm{None}, \mathrm{All}\}$. *None* bedeuted, keine leeren Zweige in XOR-Blöcken zusammenzufassen, während bei *All* alle leeren Zweige zu einem einzigen leeren Zweig zusammengefasst werden.

Hat ein XOR-Block mit den Gateway-Knoten p und q als Split- bzw. Join-Knoten $n = \eta_{\mathbb{E}}(p, q)$ leere Zweige, dann werden alle diese leeren Zweige $e_1, \ldots, e_n \in \mathbb{E}$ durch einen einzigen

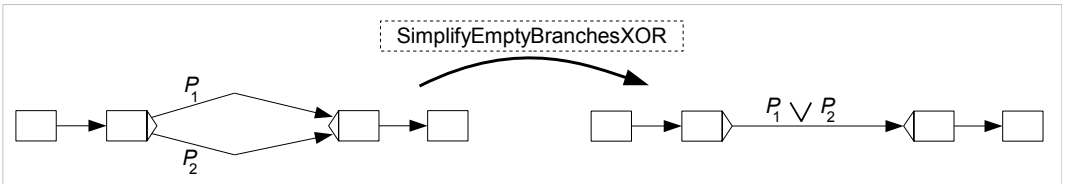

Abbildung 3.3: Beispiel für *SimplifyEmptyBranchesXOR*.

leeren Zweig e ersetzt:

$$\mathbb{E}' = \mathbb{E} \setminus \{e_1, \ldots, e_n\} \cup \{e\}.$$

Die Übergangsbedingung des neuen Zweigs e erhält man durch Disjunktion der Übergangsbedinungen der zusammengefassten leeren Zweige:

$$\mathrm{EC}'(e) = \bigvee_{k=1}^{n} \mathrm{EC}(e_k)$$

Für alle anderen Übergangsbedingungen bleibt EC' gleich EC.

Nach dem Entfernen bzw. Zusammenfassen leerer Zweige können die Gateway-Knoten einfacher Blöcke entfernt und mehrfache Blöcke zusammengefasst werden. Diese Vereinfachungen werden in den Kapiteln 3.4 und 3.5 gezeigt.

3.4 SimplifyMultipleBlocksAND und -XOR

Liegt ein Block direkt in einem Block gleichen Typs, entfernen *SimplifyMultipleBlocksAND* bzw. *SimplifyMultipleBlocksXOR* die jeweils inneren Gateway-Knoten und verbinden die Zweige des inneren Blocks mit dem umgebenden Block. Beispielsweise bilden in Abbildung 3.1 b) n_6 und n_7 den inneren Block und die Knoten n_5 und n_8 den äußeren Block. Erst nach der Reduktion von a_6 ist der innere Block direkt mit dem äußeren Block verbunden.

Wir behandeln zuerst die gemeinsamen Aspekte der beiden elementaren Operatoren, bevor wir in Kapitel 3.4.1 und 3.4.2 die beiden Operatoren spezifischer beschreiben. Im Folgenden seien $a_1, a_2, b_1, b_2 \in N$. *Äußerer Block* bezeichne den Block (a_1, a_2), *innerer Block* bezeichne den Block (b_1, b_2), verbunden durch die Kanten $E_{\text{Block}} = \{(a_1, b_1), (b_2, a_2)\} \subset E$. Außerdem seien $c_1, \ldots, c_n \in N$ die jeweils ersten Knoten der Zweige des inneren Blocks und $d_1, \ldots, d_n \in N$ jeweils die letzten Knoten der Zweige des inneren Blocks (sie-

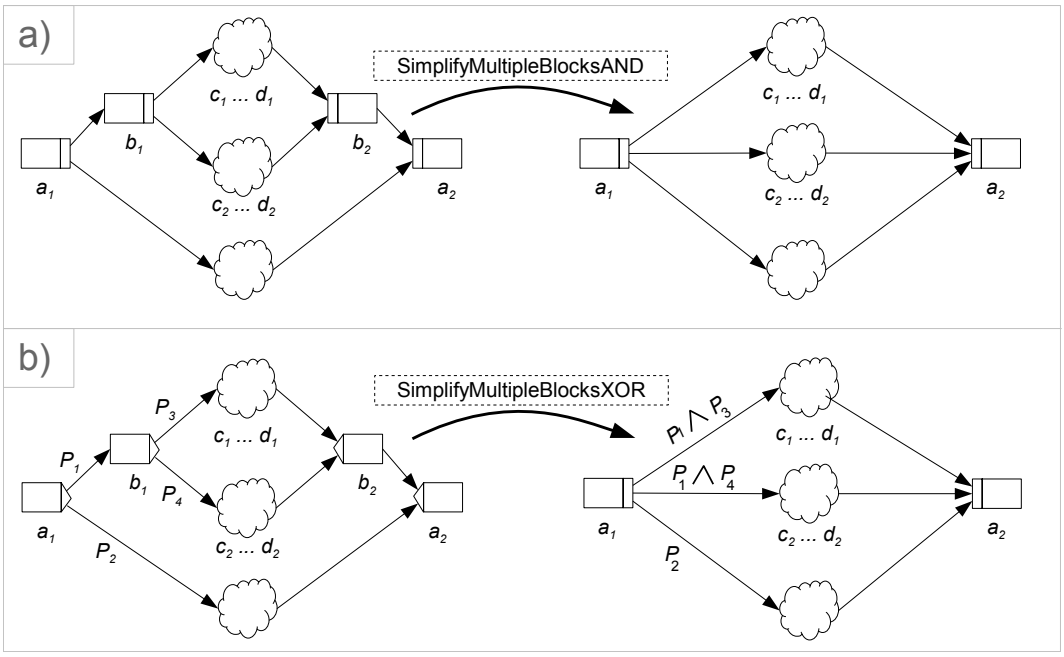

Abbildung 3.4: Vereinfachung mehrfacher Blöcke.

he Abbildung 3.4). Ferner soll gelten: $E_{\text{Anfang}} = \{(b_1, c_1), \ldots, (b_1, c_n)\} \subset E$ und $E_{\text{Ende}} = \{(d_1, b_2), \ldots, (d_n, b_2)\} \subset E$ mit $c_1 \preceq d_1, \ldots, c_n \preceq d_n$.

Bei beiden Operatoren ergibt sich die neue Knotenmenge N' des Prozessschemas mit:

$$N' = N \setminus \{b_1, b_2\}$$

und die neue Kantenmenge $\mathbb{E}'$ folgendermaßen:

$$\mathbb{E}' = \mathbb{E} \ominus E_{\text{Block}} \ominus E_{\text{Anfang}} \ominus E_{\text{Ende}} \oplus \{(a_1, c_1), \ldots, (a_1, c_n)\} \oplus \{(d_1, a_2), \ldots, (d_n, a_2)\}.$$

Die weiteren Aspekte werden in den folgenden Kapiteln 3.4.1 und 3.4.2 getrennt betrachtet. Angewendet werden beide Operatoren auf den inneren Block (b_1, b_2) – entsprechend dem Inside-Out-Prinzip des *Simplify*-Algorithmus.

3.4.1 SimplifyMultipleBlocksAND

Liegen zwei AND-Blöcke direkt ineinander, entfernt *SimplifyMultipleBlocksAND* den Split- und Join-Knoten des inneren Blocks und verbindet die Zweige des inneren Blocks mit dem umgebenden Block (siehe Abbildung 3.4 a)). Mögliche Werte des zugehörigen Parameters sind $\{\mathrm{Yes}, \mathrm{No}\}$.

Für *SimplifyMultipleBlocksAND* gilt: $\mathrm{NT}(a_1) = \mathrm{NT}(b_1) = \mathrm{ANDSplit}$ und $\mathrm{NT}(a_2) = \mathrm{NT}(b_2) = \mathrm{ANDJoin}$. Ansonsten genügt die oben genannte Änderung der Knoten- und Kantenmenge. Eine Anpassung der Übergangsbedingungen EC ist nicht nötig, weil die von einem ANDSplit-Knoten ausgehenden Kanten immer TRUE als Übergangsbedingung haben.

3.4.2 SimplifyMultipleBlocksXOR

Liegen zwei XOR-Blöcke direkt ineinander, entfernt *SimplifyMultipleBlocksXOR* den Split- und Join-Knoten des inneren Blocks, verbindet die Zweige des inneren Blocks mit dem umgebenden Block und passt die Verzweigungsbedingungen entsprechend an (siehe Abbildung 3.4 b)). Mögliche Werte des zugehörigen Parameters sind $\{\mathrm{Yes}, \mathrm{No}\}$.

Für *SimplifyMultipleBlocksXOR* gilt: $\mathrm{NT}(a_1) = \mathrm{NT}(b_1) = \mathrm{XORSplit}$ und $\mathrm{NT}(a_2) = \mathrm{NT}(b_2) = \mathrm{XORJoin}$. Zusätzlich zu der oben genannten Änderung der Knoten- und Kantenmenge müssen die Kantenbedingungen angepasst werden. Die geänderten Kantenbedingungen erhält man durch Konjunktion der ursprünglichen Bedingungen:

$$\begin{aligned} \mathrm{EC}'(\langle a_1, c_1\rangle_{\mathbb{E}'}) &= \mathrm{EC}(\langle a_1, b_1\rangle_{\mathbb{E}}) \wedge \mathrm{EC}(\langle b_1, c_1\rangle_{\mathbb{E}}) \\ &\vdots \\ \mathrm{EC}'(\langle a_1, c_n\rangle_{\mathbb{E}'}) &= \mathrm{EC}(\langle a_1, b_1\rangle_{\mathbb{E}}) \wedge \mathrm{EC}(\langle b_1, c_n\rangle_{\mathbb{E}}) \end{aligned}$$

Durch *SimplifyMultipleBlocksXOR* werden die Kantenbedinungen umfangreicher. Darunter kann für den Benutzer die Verständlichkeit einer Prozesssicht leiden, weshalb es sich bei ohnehin schon komplizierten Kantenbedingungen anbieten kann, *SimplifyMultipleBlocksXOR* auszuschalten (Parameterwahl *No*).

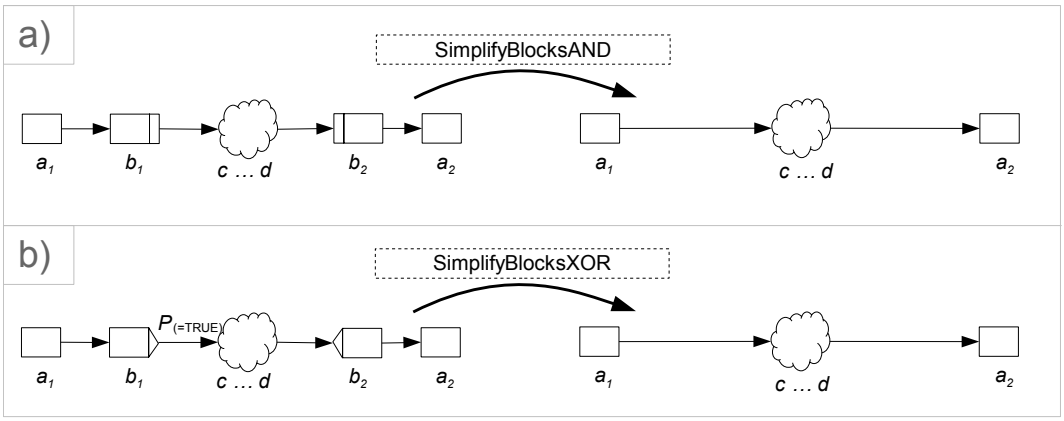

Abbildung 3.5: Entfernen einfacher Blöcke

3.5 SimplifyBlocksAND und -XOR

SimplifyBlocksAND und *SimplifyBlocksXOR* entfernen bei AND- und XOR-Blöcken mit nur einem Zweig den Split- und Join-Knoten aus dem zugehörigen Prozessschema (siehe Abbildung 3.5). Mögliche Parameterwerte sind jeweils $\{\mathrm{Yes}, \mathrm{No}\}$.

Seien $b_1, b_2 \in N$ die Gateway-Knoten eines Blocks. *SimplifyBlocksAND* wird auf diesen Block angewendet, wenn $(\mathrm{NT}(b_1), \mathrm{NT}(b_2)) = (\mathrm{ANDSplit}, \mathrm{ANDJoin})$ ist, während *SimplifyBlocksXOR* auf Blöcke mit $(\mathrm{NT}(b_1), \mathrm{NT}(b_2)) = (\mathrm{XORSplit}, \mathrm{XORJoin})$ angewendet wird. Beide nehmen unabhängig vom Typ der Gateway-Knoten die selben Änderungen am Prozessschema vor. Sie sind aber getrennt durch die beiden Parameter steuerbar.

Außerdem seien $a_1, a_2, c, d \in N$ Knoten mit $\{(a_1, b_1), (b_1, c), (d, b_2), (b_2, a_2)\} \subset E$ und $c \preceq d$. Durch Anwendung von *SimplifyBranchesAND* bzw. *-XOR* werden b_1 und b_2 reduziert, das heißt die neue Knotenmengen N' ist gegeben durch:

$$N' = N \setminus \{b_1, b_2\}$$

und die neue Kantenmenge $\mathbb{E}'$ ergibt sich aus

$$\mathbb{E}' = \mathbb{E} \ominus \{(a_1, b_1), (b_1, c), (d, b_2), (b_2, a_2)\} \oplus \{(a_1, c), (d, a_2)\}.$$

Im Falle eines XOR-Blocks mit $(\mathrm{NT}(b_1), \mathrm{NT}(b_2)) = (\mathrm{XORSplit}, \mathrm{XORJoin})$ ist zu beachten, dass für die Übergangsbedingung der ersten Kanten des einzelnen Zweigs $\mathrm{EC}(\langle b_1, c \rangle) =$

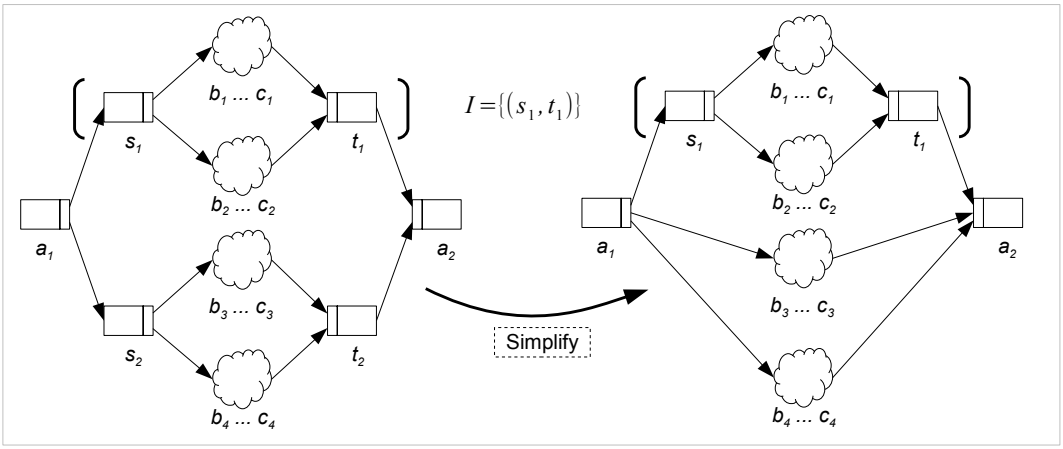

Abbildung 3.6: Bereichsweise Vereinfachung unter Verwendung eines *IgnoreSimplification*-Bereichs (s_1, t_1).

TRUE gelten muss (eine andere Bedingung würde bei der Ausführung des Schemas zu einem Deadlock führen).

Diese beiden Vereinfachungen können auch angewendet werden, wenn zwischen den Gateway-Knoten nur ein leerer Zweig ist. In diesem Fall ist $c = b_1$ und $d = b_2$.

3.6 Bereichsweise Vereinfachung

Eine Möglichkeit, die Vereinfachung gezielt zu steuern, sind *IgnoreSimplification*-Bereiche. Diese steuern nicht einzelne Vereinfachungs-Operatoren, sondern schalten in ausgewählten SESE-Fragmenten die gesamte Vereinfachung aus.

Ein einzelner IgnoreSimplification-Bereich wird spezifiziert durch den Ein- und Ausgangsknoten s bzw. t eines SESE-Fragments im Prozessschema. Für jedes Prozessschema und jede Sicht kann eine Menge $I = \{(s_1, t_1), \dots, (s_n, t_n)\} \subseteq N \times N$ solcher IgnoreSimplification-Bereiche festgelegt werden ($n \in \mathbb{N}$). Die SESE-Fragmente $(s_1, t_1), \dots, (s_n, t_n)$ dürfen sich nicht überlappen.

Der $\mathrm{Simplify}$-Algorithmus fährt beim jeweiligen SESE-Ausgangsknoten t_i fort, falls ein Knoten $p \in N : s_i \preceq p \preceq t_i$ angetroffen wird, der sich in einem der IgnoreSimplification-Bereiche befindet ($i \in \{1, \dots, n\}$).

Mithilfe der IgnoreSimplification-Bereiche lassen sich in ausgewählten Bereichen von Prozesssichten Einfüge-Positionen erhalten, die andernfalls nicht zur Verfügung stehen wür-

den, wenn in anderen Bereichen trotzdem eine Vereinfachung gewünscht ist. In Abbildung 3.6 ist es zum Beispiel im oberen Zweig von a_1 möglich, nach der Vereinfachung eine Aktivität zwischen a_1 und s_1 einzufügen. Insbesondere wird damit die Aktivität vor b_1 *und* b_2 eingefügt. Dagegen ist es nicht ohne weiteres möglich, nach der Vereinfachung eine Aktivität *vor* b_3 *und* b_4 einzufügen. (In Kapitel 5 wird ein Aktualisierungs-Algorithmus gezeigt, der den Block (s_2, t_2) in einer Sicht rekonstruiert bei geeigneter Parameterwahl – in diesem Fall $\mathrm{InsertSerial} = \mathrm{Early}$. Dieser Parameter unterliegt jedoch im Allgemeinen nicht der Kontrolle des Benutzers, der die Sicht erstellt hat (über welche die Aktivität eingefügt wird), sondern ist dem ursprünglichen Prozess*schema* zugeordnet. In einem solchen Fall ist die bereichsweise Vereinfachung sinnvoll.)

3.7 Zusammenfassung

Die Vereinfachungs-Operatoren erlauben es, Prozesssichten die aus *Reduktionen* von Aktivitäten hervorgehen, kompakter und dadurch noch übersichtlicher für den Benutzer darzustellen. Durch *Aggregation* von SESE-Fragmenten entstehen Sichten, bei denen die beschriebenen Vereinfachungs-Operatoren keine Anwendung finden, weil die Struktur des Graphens dabei nicht derart verändert wird, dass eine Vereinfachung möglich ist. Eine weitere Anwendung findet die Vereinfachung bei den Einfüge-Operationen (siehe Kapitel 4), da sich diese einfacher gestalten, wenn zum Beispiel beim Einfügen von Split- und Join-Knoten nicht darauf geachtet werden muss, dass diese eventuell schon direkt von einem Block des selben Typs (AND- oder XOR-Block) umgeben sind (siehe SimplifyMultipleBlocksAND bzw. -XOR in Kapitel 3.4).

Allerdings schafft die Vereinfachung auch ein Problem: mögliche Einfügepositionen können durch die Vereinfachung verloren gehen. Das heißt, beim Einfügen einer Aktivität in der vereinfachten Sicht kann die zugehörige Position im CPM nicht mehr eindeutig zugeordnet werden. Abhilfe schaffen teilweise die Parameter der Aktualisierungs-Operatoren (siehe Kapitel 4). In Situationen, in denen die Mehrdeutigkeit durch Parameter nicht auflösbar ist, wäre eine Möglichkeit, dem Benutzer den entsprechenden Teilgraph aus dem CPM anzuzeigen und dort die genaue Position auswählen zu lassen. Nachteilig ist allerdings, dass die Aktualisierung dann nicht mehr vollständig automatisch stattfindet, was vor allem bei einem umfangreichen Prozessschema schwierig für einen Benutzer sein kann, weil dieser eventuell nicht über den Wissensstand des Prozess-Designers bezüglich des Gesamtprozesses verfügt und daher die Entscheidung gar nicht treffen kann, während der Prozess-Designer die Parameter für die Aktualisierungs-Operationen beim CPM selbst festlegen kann. Auf Seiten des Prozess-Designers ergibt sich dann aber das Problem, dass dieser womöglich

die CPM-Parameter nicht im Voraus für alle Einfügungen passend festlegen kann. Daraus folgt, dass manche Mehrdeutigkeiten beim Einfügen nur durch Abstimmung zwischen Sichten-Erstellern und Prozess-Designern aufgelöst werden können.

4 Aktualisierungs-Operatoren

Wird eine Aktivität in einer Sicht eingefügt, muss diese Änderung an das CPM und andere Sichten propagiert werden. Im CPM sind immer alle Aktivitäten sichtbar, während in Sichten – je nach Parametereinstellung – *eingefügte* Aktivitäten reduziert oder aggregiert werden. Das heißt, wenn in der zu aktualisierenden Sicht die einzufügende Aktivität in einen Bereich fällt, in dem die umgebenden Aktivitäten reduziert beziehungsweise aggregiert sind, muss diese gegebenenfalls ebenso aggregiert bzw. reduziert werden. Mit den Aktualisierungs-Parametern kann bei jeder Sicht kann gesteuert werden, ob eine eingefügte Aktivität in einem solchen Bereich *auch* reduziert bzw. aggregiert wird. Zudem ist die Zuordnung von Einfüge-Positionen im CPM nach Reduktionen oder Aggregationen nicht immer eindeutig, weshalb durch Parameter gesteuert werden kann, wo Aktivitäten eingefügt werden sollen, wenn eine eindeutige Zuordnung nicht möglich ist. Abbildung 4.1 zeigt beispielhaft diese Situation, in der nach dem Einfügen der Aktivität a_5 in einem reduzierten Bereich (Abbildung 4.1 d)) *mehrere* Zuordnungen der Einfüge-Position möglich sind, zum Beispiel vor a_2 oder nach a_3 (Abbildung 4.1 c)).

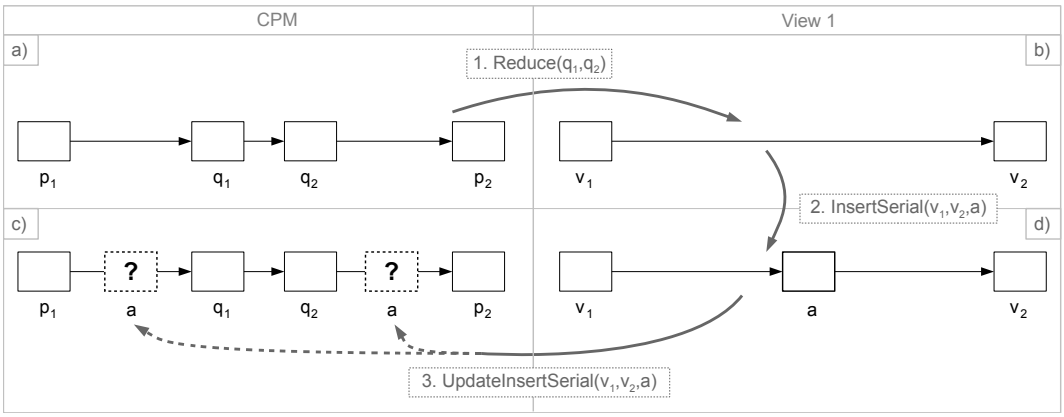

Abbildung 4.1: Beispiel für die Mehrdeutigkeit nach dem Einfügen einer Aktivität in einer Sicht.

Der Gesamtablauf beim Einfügen einer Aktivität in eine Prozesssicht ist folgender: nach dem Einfügen der Aktivität in einer Prozesssicht wird zuerst das CPM aktualisiert und ab-

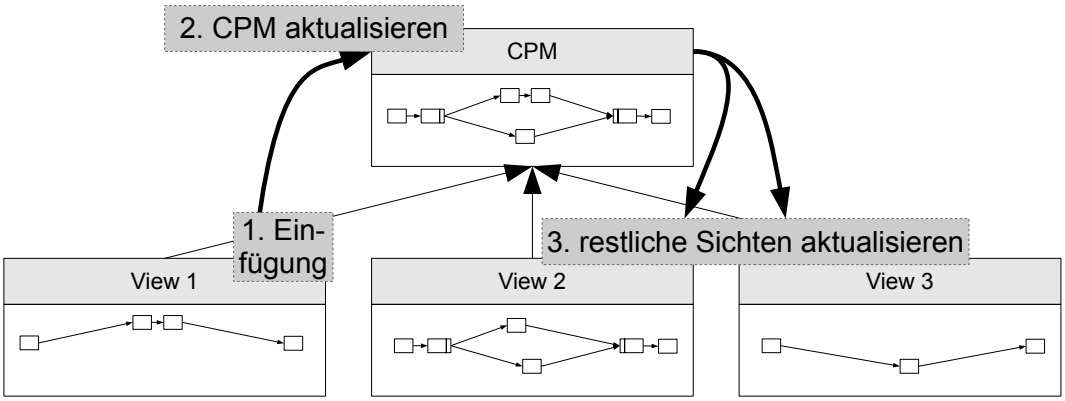

Abbildung 4.2: Gesamtablauf bei der Aktualisierung

schließend alle anderen Sichten (außer der Sicht, in der die Aktivität ursprünglich eingefügt wurde). Abbildung 4.2 veranschaulicht diesen Vorgang grafisch.

Eine wichtige Bedingung beim Einfügen oder Löschen von Aktivitäten über Prozesssichten ist, dass Prozesssichten und -schema das gleiche Ausführungsverhalten haben sollen. Das bedeuted, die möglichen Ausführungsreihenfolgen der Aktivitäten müssen in Sichten und Schema die selben sein. Diese Bedingung wäre beispielsweise verletzt, wenn die beiden Aktivitäten a_1 und a_2 im Prozessschema seriell aufeinanderfolgen und in einer Sicht in umgekehrter Reihenfolge vorkommen, das heißt, a_2 vor a_1. Die Einhaltung dieser Bedingung ist also gewährleistet, wenn die Ordnung der Aktivitäten im Schema und den Sichten erhalten bleibt. Folgende Definition beschreibt diesen Sachverhalt formal (angelehnt an [8]):

Definition 12. (Strenge Ordnungserhaltung)

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema mit der Aktivitätenmenge $A \subset N$ und $V = V(P)$ eine zugehörige Prozesssicht. V ist genau dann *streng ordnungserhaltend*, wenn für alle Aktivitäten $a_1, a_2 \in A$ mit den korrespondierenden Aktivitäten $\downarrow_V^P [a_1] \neq \emptyset$ und $\downarrow_V^P [a_2] \neq \emptyset$ gilt: $a_1 \preceq a_2 \Rightarrow \downarrow_V^P [a_1] \preceq \downarrow_V^P [a_2]$.

Die in Kapitel 2.3 beschriebenen Operatoren $\mathrm{ReduceActivity}$ und $\mathrm{AggregateSESE}$ erzeugen streng ordnungserhaltende Prozesssichten. Im Folgenden werden die nötigen Aktualisierungen des CPMs und anderer Prozesssichten für die Einfüge-Operatoren aus Kapitel 2.4 (serielles, paralleles und bedingtes Einfügen) beschrieben. Die Aktualisierungen für paral-

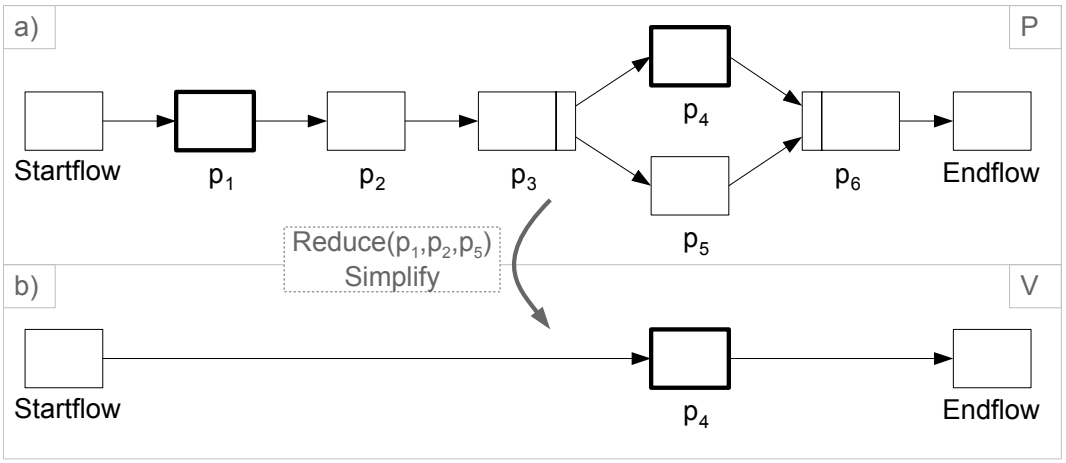

Abbildung 4.3: $\mathrm{VisibleSuccessor}(p_1, P, V) = p_4$

leles und bedingtes Einfügen sind sich ähnlich, weshalb diese gemeinsam in einem Kapitel beschrieben werden.

Falls in einer Prozesssicht eine Reihe von Knoten reduziert wurden, verwenden die Aktualisierungs-Algorithmen den ersten Vorgänger bzw. Nachfolger des Knotens, der auch in der Prozesssicht vorhanden ist. Abbildung 4.3 zeigt ein Beispiel für diesen Sachverhalt.

Definition 13. ($\mathrm{VisibleSuccessor}(p_1, P, V)$ und $\mathrm{VisiblePredecessor}(p_1, P, V)$)

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema, $V = V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ eine zugehörige Prozesssicht und $p_1 \in N : \mathrm{NT}(p_1) \neq \mathrm{Endflow}$. Seien $p_2, \ldots, p_n \in N$ mit $n \in \mathbb{N}$ alle direkten und indirekten Nachfolger von p bis zum Endknoten $p_n \in N : \mathrm{NT}(p_n) = \mathrm{Endflow}$ in topologischer Sortierreihenfolge:

$$p_1 \mapsto p_2 \mapsto \ldots \mapsto p_n.$$

Für einen Teil dieser Folge gelte:

$$\downarrow_V^P [p_1] = \emptyset, \ldots, \downarrow_V^P [p_{k-1}] = \emptyset \text{ und } \downarrow_V^P [p_k] \neq \emptyset$$

für ein $k \leq n$. Dann ist $\mathrm{VisibleSuccessor}(p_1, P, V) = \downarrow_V^P [p_k]$ der erste *sichtbare Nachfolger* von p_1 in der Sicht V. Analog ist $\mathrm{VisiblePredecessor}(p_1, P, V)$ der erste *sichtbare Vorgänger* von p in der Sicht V (das heißt, von p_1 ausgehend in topologischer Sortierreihenfolge zum Startknoten). Beispiel: in Abbildung 4.3 ist p_4 der sichtbare Nachfolger von p_1.

Folgende Definition hilft dabei, den *passenden* Vorgänger bzw. Nachfolger eines Gateway-Knotens auszuwählen.

Definition 14. ($\mathrm{BranchPredecessor}(n_1, n_2)$ und $\mathrm{BranchSuccessor}(n_1, n_2)$)

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema mit der einfachen Darstellung E von $\mathbb{E}$. Dann sind $\mathrm{BranchPredecessor}$ und $\mathrm{BranchSuccessor} : N \times N \to N$ definiert als:

$$\mathrm{BranchPredecessor}(n_1, n_2) = \begin{cases} m\bullet = n_1 : m \in N \wedge n_2 \preceq m & n_1 \text{ ist Join-Knoten und (*)} \\ \bullet n_1 & n_1 \text{ ist Aktivität, Endflow-} \\ & \text{oder Split-Knoten} \end{cases}$$

$$\mathrm{BranchSuccessor}(n_1, n_2) = \begin{cases} m \in N : \bullet m = n_1 \wedge m \preceq n_2 & n_1 \text{ ist Split-Knoten und (*)} \\ n_1\bullet & n_1 \text{ ist Aktivität, Startflow-} \\ & \text{oder Join-Knoten} \end{cases}$$

(*) bedeuted, dass n_2 im Block von n_1 enthalten sein muss und nicht einer der beiden Gateway-Knoten dieses Blocks sein darf. $\mathrm{BranchPredecessor}(n_1, n_2)$ gibt *einen* der Vorgänger von n_1 an. Ist n_1 ein Join-Knoten, gibt es in der Regel keinen eindeutigen Vorgänger. In diesem Fall wird n_2 zu Hilfe genommen. n_2 muss ein Knoten in einem der Zweige des Join-Knotens n_1 sein. Der letzte Knoten des Zweiges, der in n_1 mündet und n_2 enthält, ist der *passende* Vorgänger. Da alle anderen Knotentypen (außer dem Startknoten) nur einen Vorgänger haben, kann dieser direkt verwendet werden. Das heißt, in diesem Fall wird n_2 ignoriert und $\mathrm{BranchPredecessor}(n_1, n_2)$ ist einfach der direkte Vorgänger von n_1. Im Beispielschema in Abbildung 4.4 wäre $\mathrm{BranchPredecessor}(n_1, a_1) = a_2$ und $\mathrm{BranchPredecessor}(a_2, a_1) = n_2$. Analog dazu gibt $\mathrm{BranchSuccessor}$ *einen der* oder *den* Nachfolger an.

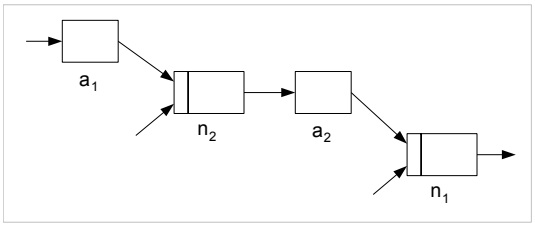

Abbildung 4.4: Beispielschema

Nachdem diese Hilfsfunktionen definiert wurden, können nun die jeweiligen Aktualisierungs-Operatoren beschrieben werden. Im Folgenden sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ und $V_k = V_k(P) = (N_k, \mathbb{E}_k, \mathrm{EC}_k, \mathrm{NT}_k, C_k)$ zugehörige Sichten ($k \in \{1, \ldots, n\}$ mit $n \in \mathbb{N}$). Sichten, die in der View-Hierarchie nicht direkt unter dem zentralen Prozessmodell CPM angeordnet sind

(„Sichten auf Sichten“) werden beim Update so behandelt, wie direkte Sichten auf das zentrale Prozessmodell. E und E_k sind jeweils die einfachen Darstellungen von von $\mathbb{E}$ bzw. $\mathbb{E}_k$. $V = V_1$ sei die Sicht, in der die Aktivität a eingefügt wird. Knoten aus P werden mit p_i bezeichnet, Knoten aus V mit v_i, um die Herkunft eines Knotens zu verdeutlichen ($i \in \mathbb{N}$).

4.1 UpdateInsertSerial

Seien $v_1, v_2 \in N_1 : (v_1, v_2) \in E_1$ die beiden Knoten, zwischen denen die Aktivität a eingefügt werden soll. Das heißt, auf die Sicht V wird $\mathrm{InsertSerial}(v_1, v_2, a)$ angewendet. $\mathrm{UpdateInsertSerial}(v_1, v_2, a)$ fügt die Aktivität im CPM ein und veranlasst die Aktualisierung der anderen Sichten.

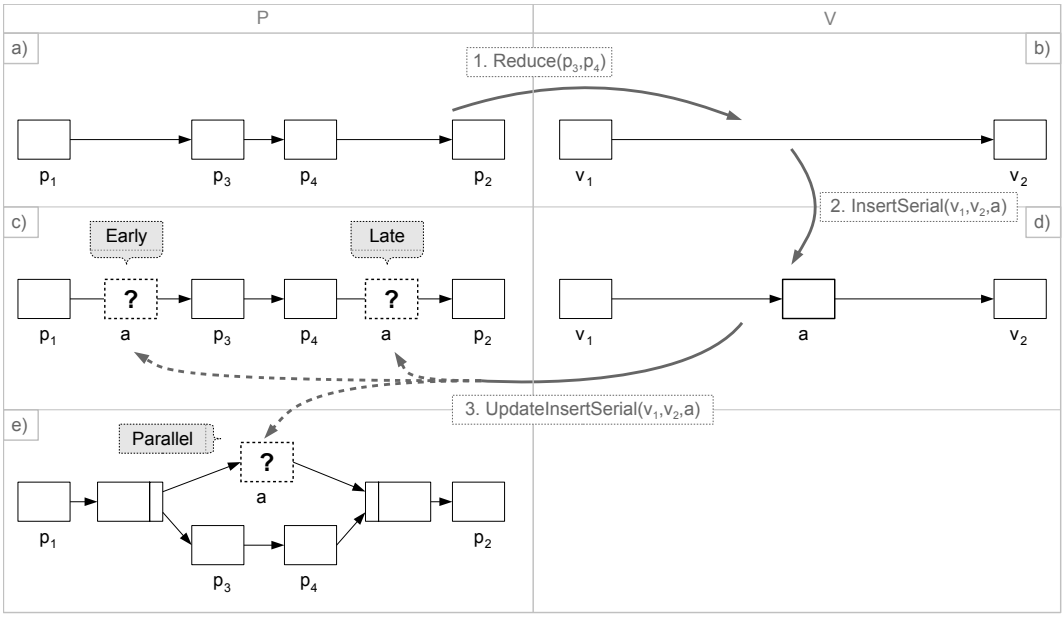

Abbildung 4.5: Beispiel für *UpdateInsertSerial* ($v_1 \equiv p_1, v_2 \equiv p_2$).

Falls sich im CPM der Kontext von dem in der Sicht unterscheidet, das heißt, zwischen den zu v_1 und v_2 korrespondierenden Knoten befindet sich im Schema ein Teilgraph, der in der Sicht fehlt, wird $\mathrm{UpdateInsertSerial}$ durch den Parameter $\mathrm{InsertSingle} \in \{\mathrm{Early}, \mathrm{Late}, \mathrm{Parallel}\}$ gesteuert. Abbildung 4.5 zeigt zum einen den unterschiedlichen Kontext – in Abbildung 4.5 b) + d) fehlen p_3 und p_4 – und veranschaulicht zum anderen die durch den Parameter $\mathrm{InsertSingle}$ möglichen Einfüge-Positionen. Early fügt die Aktivität *vor* dem fehlenden Bereich ein, Late danach und $\mathrm{Parallel}$ parallel dazu.

Es folgt der Algorithmus für $\mathrm{UpdateInsertSerial}(v_1, v_2, a)$ (siehe Algorithmus 4.1). Gesucht sind der jeweils korrespondierende Vorgänger und Nachfolger von a in P. Falls der Vorgänger von a in V ein Aggregationsknoten ist, wird dessen Ausgangsknoten verwendet. Analog wird beim Nachfolger der Eingangsknoten verwendet (man beachte, dass $\mathrm{LeastCommonEntry}$ und $\mathrm{LeastCommonExit}$ keine Effekt haben, falls der angegebene Knoten kein Aggregationsknoten ist):

$$\begin{aligned} p_1 &= \mathrm{LeastCommonExit}(\uparrow_V^P [v_1]) \\ p_2 &= \mathrm{LeastCommonEntry}(\uparrow_V^P [v_2]) \end{aligned}$$

Um den Kontext von p_1 und p_2 mit dem in der Sicht zu vergleichen, wird der Nachfolger von p_1 und der Vorgänger von p_2 benötigt. Sollte p_1 ein Split-Knoten oder p_2 ein Join-Knoten sein, wird der jeweils andere Knoten verwendet, um den passenden Zweig zu finden.

$$\begin{aligned} p_3 &= \mathrm{BranchSuccessor}(p_1, p_2) \\ p_4 &= \mathrm{BranchPredecessor}(p_2, p_1) \end{aligned}$$

Abschließend werden entsprechend Algorithmus 4.1 verschiedene Fälle unterschieden und entsprechende Änderungen veranlasst. Der Fall, dass sich der Kontext unterscheidet, wird daran erkannt, dass der Nachfolger von p_1 (nämlich p_3) ein anderer ist als der zu v_2 korrespondierende Knoten (nämlich p_2).

Algorithmus 4.1 $\mathrm{UpdateInsertSerial}(v_1, v_2, a)$

$P, v_1, v_2, a, p_1, p_2, p_3, p_4$ `siehe Text`
`Wenn` $p_3 \neq p_2$
  `Falls` InsertSingle = Early
    $\mathrm{InsertSerial}(p_1, p_3, a)$
    $\mathrm{UpdateViewsInsertSerial}(p_1, p_3, a)$ **(siehe Kapitel 4.1.1)**
  `Falls` InsertSingle = Late
    $\mathrm{InsertSerial}(p_4, p_2, a)$
    $\mathrm{UpdateViewsInsertSerial}(p_4, p_2, a)$
  `Falls` InsertSingle = Parallel
    `Wenn` $p_3 \ldots p_4$ `keine SESE-Fragment ist`
      `Fehler:` InsertSingle=Parallel `nicht möglich`
        (siehe Kapitel 4.1.2)

```
        Sonst
            InsertParallel(p3, p4, a)
            Simplify(P)
            UpdateViewsInsertParallel(p3, p4, a)  (siehe Kapitel 4.2.1)
    Sonst
        InsertSerial(p1, p2, a)
        UpdateViewsInsertSerial(p1, p2, a)
```

4.1.1 UpdateViewsInsertSerial

Nachdem das CPM aktualisiert wurde, können nun die restlichen Sichten aktualisiert werden. $\mathrm{UpdateViewsInsertSerial}(p_1, p_2, a)$ wendet den in diesem Abschnitt beschriebenen Algorithmus auf jede der verbleibenden Prozesssichten $W = V_k$ $(k = 2, \ldots, n)$ an und wird gesteuert durch die folgenden Parameter:

$$\begin{aligned} \mathrm{ChangeInReductionComplete} &\in \{\mathrm{Reduce}, \mathrm{Show}\} \\ \mathrm{ChangeInAggregateComplete} &\in \{\mathrm{Aggregate}, \mathrm{Unaggregate}\}. \end{aligned}$$

Der Parameter $\mathrm{ChangeInReductionComplete}$ kommt zur Anwendung, wenn in der Sicht der Bereich, in dem die Einfügung stattfinden soll, komplett reduziert wurde. Der Wert Reduce bedeuted dann, die einzufügende Aktivität *auch* zu reduzieren (das heißt, die Aktivität *nicht* einzufügen), während bei Show die Aktivität zwischen den verbleibenden Knoten eingefügt wird.

Der Parameter $\mathrm{ChangeInAggregateComplete}$ kommt zur Anwendung, wenn der entsprechende Bereich, in dem die Änderung stattgefunden hat, komplett aggregiert wurde. Wurde als Parameter-Wert $\mathrm{Aggregate}$ gewählt, wird die einzufügende Aktivität zum Aggregationsknoten hinzugefügt. $\mathrm{Unaggregate}$ bewirkt hingegen, dass der Aggregationsknoten durch das entsprechende SESE-Fragment aus dem Prozessschema ersetzt und a anschließend in dem nun vorhandenen Bereich eingefügt wird.

Die Parameter $\mathrm{ChangeInReductionComplete}$ und $\mathrm{ChangeInAggregateComplete}$ überschneiden sich jedoch in einer Situation. Falls in der Sicht zuerst p_1 und p_2 reduziert wurden, und dann deren umgebende Knoten (mindestens der Vorgänger von p_1 und der Nachfolger von p_2) zu *einem* Knoten aggregiert wurden, liegt die mögliche Einfüge-Position in einem Aggregationsknoten. In dem Fall wird zuerst überprüft, ob $\mathrm{ChangeInReductionComplete} = \mathrm{Show}$ ist. Wenn ja, hängt die weitere Vorgehensweise von $\mathrm{ChangeInAggregateComplete}$ ab (wie

im vorigen Absatz beschrieben). Andernfalls ($\mathrm{ChangeInReductionComplete} = \mathrm{Reduce}$) wird die Aktivität reduziert.

Algorithmus 4.2 basiert darauf, dass die Block-Vereinfachungs-Operatoren ausgeschaltet sind (siehe Kapitel 3). Das heißt, die Parameter *SimplifyBlocksAND*, *-XOR*, *SimplifyMultipleBlocksAND* und *-XOR* haben jeweils den Wert No. Andernfalls können fehlerhafte Prozessschemas entstehen. Das passiert, wenn die genannten Vereinfachungs-Operatoren Gateway-Knoten entfernen, die bei späteren Einfügungen wieder benötigt werden. In Kapitel 5 wird ein erweiterter Algorithmus für $\mathrm{UpdateViewsInsertSerial}$ gezeigt, der Gateway-Knoten rekonstruieren kann, die bei der Vereinfachung entfernt wurden.

Abbildung 4.6 zeigt Beispiele für $\mathrm{UpdateViewsInsertSerial}$. Abbildung 4.6 a) zeigt einen Ausschnitt aus einem CPM. In der linken Hälfte sind verschiedene Sichten auf das CPM *vor* dem Update gezeigt, in der rechten Hälfte die selben Sichten *nach* dem Update mit den jeweiligen Parametereinstellungen für *ChangeInReductionComplete* und *ChangeInAggregationComplete* (grau hinterlegt). Die Sichten V1.1, V2.1 und V3.1 sind Sichten auf die jeweils darüberliegende Sicht V1, V2 bzw. V3.

Algorithmus 4.2 $\mathrm{UpdateViewsInsertSerial}(p_1, p_2, a)$

a Aktivität, die in die Sicht W eingefügt werden soll und bereits im Prozessschema P zwischen p_1 und p_2 eingefügt wurde.
$w_1 = \downarrow_W^P [p_1]$
$w_2 = \downarrow_W^P [p_2]$
Wenn $w_1 = \emptyset$ und $w_2 = \emptyset$ **(beide reduziert, Abb. 4.6 b+c)**
  Falls $\mathrm{ChangeInReductionComplete} = \mathrm{Reduce}$
    Fertig. **(Abb. 4.6 b)**
  Falls $\mathrm{ChangeInReductionComplete} = \mathrm{Show}$
    $w_3 = \mathrm{VisiblePredecessor}(p_1, P, W)$
    $w_4 = \mathrm{VisibleSuccessor}(p_2, P, W)$
    Wenn $w_3 = w_4$ **(Umgebung aggr., Abb. 4.6 c)**
      Falls $\mathrm{ChangeInAggregateComplete} = \mathrm{Aggregate}$
        Füge (w_1, a) zu C_k hinzu
        (in Aggregationsknoten w_1 einfügen)
      Falls $\mathrm{ChangeInAggregateComplete} = \mathrm{Unaggregate}$
        $\mathrm{UnaggregateSESE}(w_1)$
        $\mathrm{InsertSerial}(\mathrm{VisiblePredecessor}(p_1, P, W), \mathrm{VisibleSuccessor}(p_2, P, W), a)$
        $\mathrm{Simplify}(W)$

Sonst **($w_3 \neq w_4$)**

InsertSerial(w_3, w_4, a) **(Abb. 4.6 b)**

Sonst, wenn $w_1 = w_2$ **(beide aggregiert, Abb. 4.6 h)**

Falls ChangeInAggregateComplete = Aggregate

Füge (w_1, a) zu C_k hinzu **(in Aggregationsknoten w_1 einfügen)**

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE(w_1)
InsertSerial($\downarrow_W^P [p_1], \downarrow_W^P [p_2], a$)
Simplify(W)

Sonst, wenn $w_1 \neq \emptyset$ und $w_2 \neq \emptyset$ **(keine Änderung)**

InsertSerial(w_1, w_2, a)

Sonst **(einer von beiden reduziert, Abb. 4.6 d-g])**

Falls $w_1 = \emptyset$ **(Vorgänger reduziert, Abb. 4.6 f+g)**

Wenn VisiblePredecessor$(p_1, P, W) = w_2$ **(in Aggr. red., Abb. 4.6 g)**

Falls ChangeInAggregateComplete = Aggregate

Füge (w_1, a) zu C_k hinzu

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE(w_2)
InsertSerial(VisiblePredecessor(p_1, P, W), $\downarrow_V^P [p_2], a$)
Simplify(W)

Sonst

InsertSerial(VisiblePredecessor(p_1, P, W), w_2, a) **(Abb. 4.6 f)**

Falls $w_2 = \emptyset$ **(Nachfolger reduziert, Abb. 4.6 d+e)**

Wenn VisibleSuccessor(p_2, P, W) $= w_1$ **(in Aggr. red., Abb. 4.6 e)**

Falls ChangeInAggregateComplete = Aggregate

Füge (w_1, a) zu C_k hinzu

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE(w_1)
InsertSerial($\downarrow_V^P [p_1]$, VisibleSuccessor(p_2, P, W), a)
Simplify(W)

Sonst

InsertSerial(w_1, VisibleSuccessor(p_2, P, W), a) **(Abb. 4.6 d)**

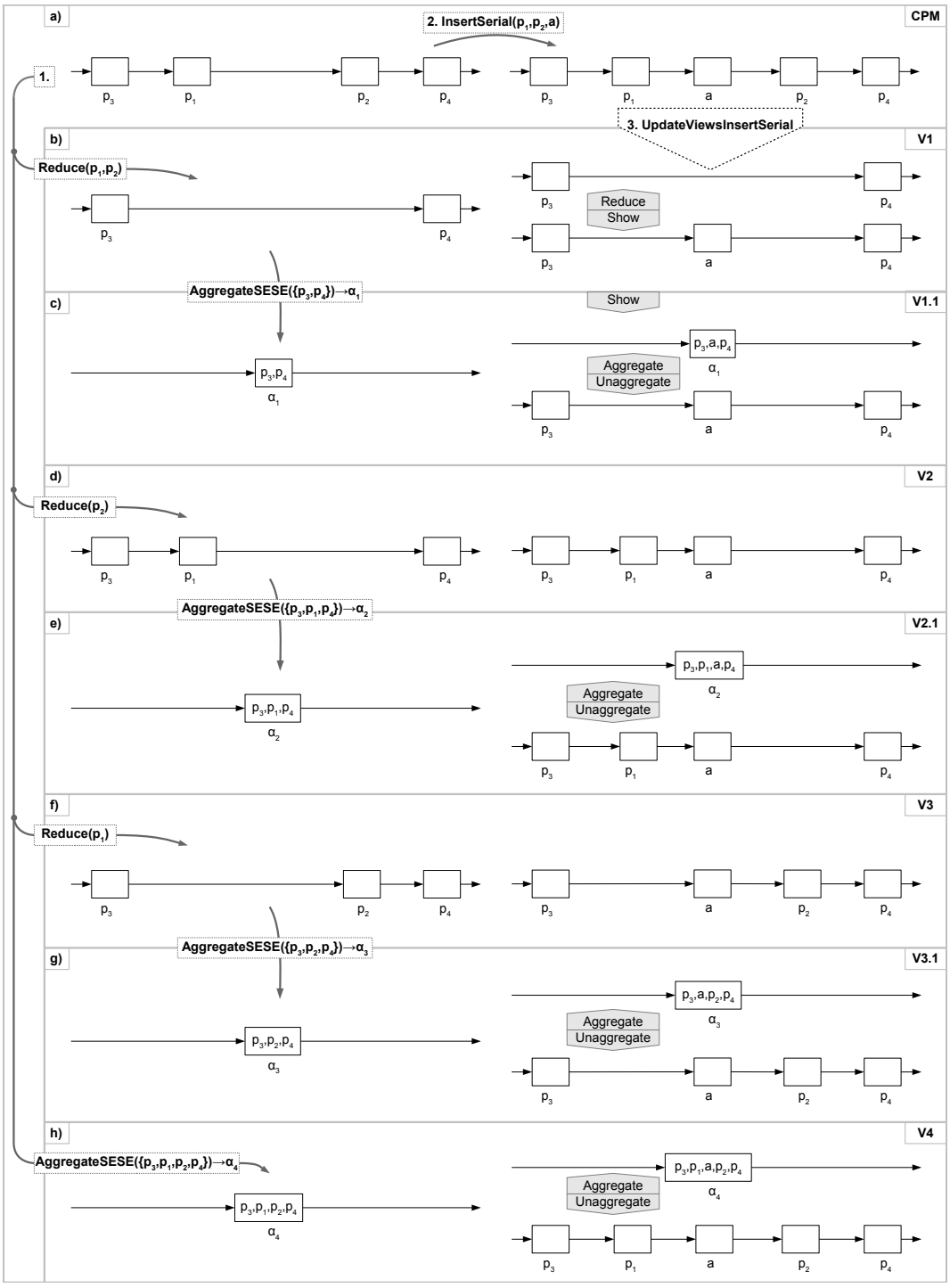

Abbildung 4.6: Beispiele für $\mathrm{UpdateViewsInsertSerial}(p_1, p_2, a)$.

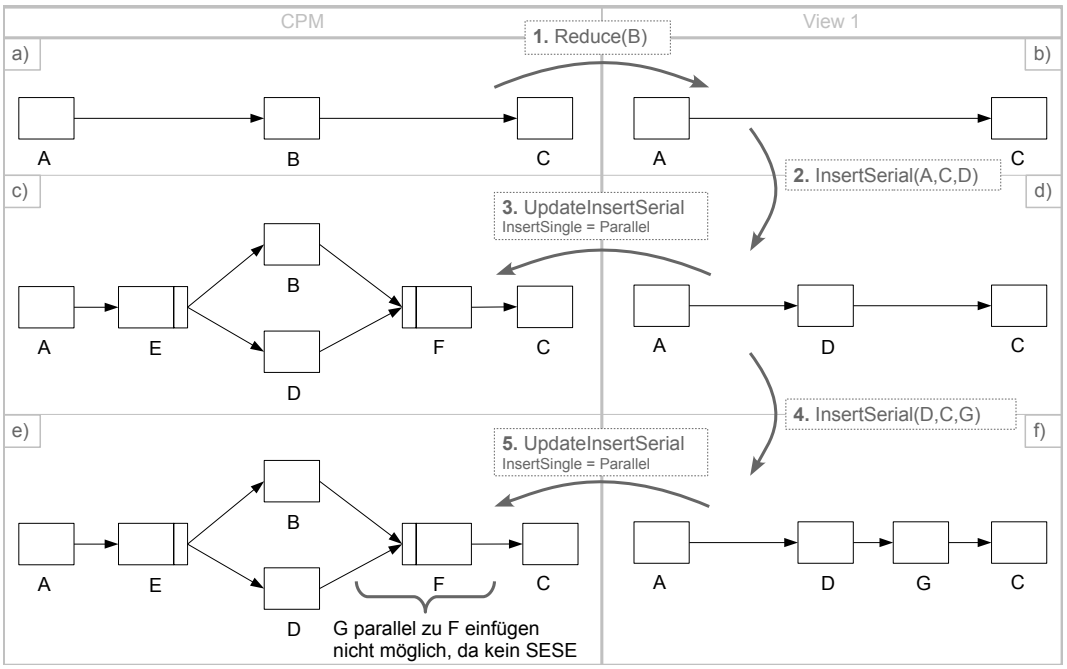

Abbildung 4.7: Skizzierung des Problems mit dem Parameter *InsertSingle = Parallel*.

4.1.2 InsertSingle-Problematik

Wenn für *UpdateInsertSerial* der Parameterwert $\mathrm{InsertSingle} = \mathrm{Parallel}$ eingestellt ist, gibt es einen speziellen Fall, in der eine paralle Einfügung im Schema nicht möglich ist, die aus einer seriellen Einfügung in einer Sicht resultiert.

Der Parameter $\mathrm{InsertSingle}$ steuert die Einfüge-Position im CPM falls in der Sicht, in der die Einfügung getätigt wurde, zwischen zwei Knoten ein Bereich reduziert wurde, der im CPM vorhanden ist. Mit dem Parameterwert $\mathrm{Parallel}$ wird die einzufügende Aktivität parallel zu diesem Bereich eingefügt. Falls dieser Bereich kein SESE-Fragment bildet, ist eine parallele Einfügung nicht möglich.

Abbildung 4.7 skizziert das Auftreten dieses Falls. Das Problem tritt beim Übergang von Abbildung 4.7 d) zu 4.7 f) auf. Die Aktivität G wird zwischen D und C eingefügt. Im CPM ist der Bereich $D, \ldots, C$ kein SESE-Fragment. G müsste also parallel zu F eingefügt werden, was zur Folge hätte, dass der resultierende Graph nicht mehr wohlstrukturiert ist.

Der Algorithmus für *UpdateInsertSerial* erkennt diesen Fall. Der implementierte Prototyp (siehe Kapitel 5) macht in diesem Fall die Einfügung in der Sicht rückgängig und benachrichtigt den Benutzer darüber.

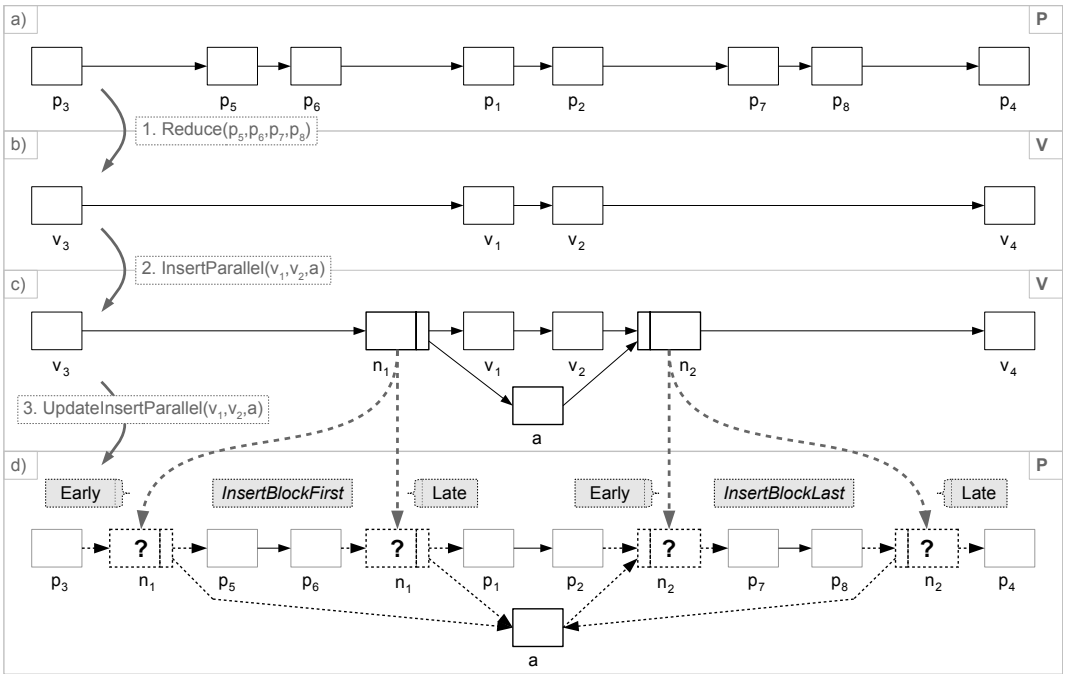

Abbildung 4.8: Beispiel für *UpdateInsertParallel*. Oben und unten das Schema P, dazwischen die beiden Zustände der Sicht V mit $(v_1, \ldots, v_4) \equiv (p_1, \ldots, p_4)$.

4.2 UpdateInsertParallel, UpdateInsertConditional

Wird in einer Sicht eine Aktivität parallel oder bedingt eingefügt unter Verwendung der Einfüge-Operatoren $\mathrm{InsertParallel}$ bzw. $\mathrm{InsertConditional}$, führen die entsprechenden Aktualisierungs-Operatoren $\mathrm{UpdateInsertParallel}$ bzw. $\mathrm{UpdateInsertConditional}$ die nötige Anpassung des CPMs durch und veranlassen abschließend die Aktualisierung der restlichen Sichten.

Seien $v_1, v_2 \in N_1$ der Eingangs- bzw. Ausgangsknoten des SESE-Fragments, zu dem die Aktivität a parallel eingefügt wurde. Es wurde also $\mathrm{InsertParallel}(v_1, v_2, a)$ auf die Sicht V angewendet. Daraufhin fügt $\mathrm{UpdateInsertParallel}(v_1, v_2, a)$ die Aktivität im CPM ein und veranlasst die Aktualisierung der restlichen Sichten. Die Aktualisierungen für bedingtes Einfügen mit $\mathrm{InsertConditional}$ sind analog. Daher wird $\mathrm{UpdateInsertConditional}$ nicht gesondert beschrieben.

Unterscheiden sich die Bereiche vor v_1 bzw. nach v_2 ist die Einfüge-Position für den Split- bzw. Join-Knoten im CPM nicht eindeutig. Um eine eindeutige Einfüge-Position zu bestimmen, greift *UpdateInsertParallel* auf die Parameter $\mathrm{InsertBlockFirst}, \mathrm{InsertBlockLast} \in$

$\{\mathrm{Early}, \mathrm{Late}\}$ zurück. *InsertBlockFirst* bestimmt die Position des Split-Knotens, während *InsertBlockLast* die Position des Join-Knotens festlegt. Der Wert *Early* bedeuted in diesem Fall, den Knoten vor dem in der Sicht reduzierten Bereich einzufügen, der Wert *Late* entsprechend danach (siehe Abbildung 4.8). Effektiv bedeuted $\mathrm{InsertBlockFirst} = \mathrm{Early}$, dass die Aktivität a zusätzlich parallel zum im CPM reduzierten Bereich vor v_1 eingefügt wird (analog für $\mathrm{InsertBlockLast} = \mathrm{Late}$). Im Rest dieses Abschnitts wird der Algorithmus für $\mathrm{UpdateInsertParallel}(v_1, v_2, a)$ bzw. $\mathrm{UpdateInsertConditional}(v_1, v_2, a)$ beschrieben.

Seien n_1 der Split-Knoten und n_2 der Join-Knoten, mit denen a in die Sicht eingefügt wurde:

$$\begin{aligned} n_1 &= (\bullet a)_{\mathbb{E}_1} \\ n_2 &= (a\bullet)_{\mathbb{E}_1} \end{aligned}$$

Bei *UpdateInsertParallel* sind n_1 und n_2 AND-Knoten, bei *UpdateInsertConditional* XOR-Knoten. p_1 und p_2 bezeichnen die zu v_1 und v_2 korrespondierenden Knoten:

$$\begin{aligned} p_1 &= \mathrm{LeastCommonEntry}(\uparrow_V^P [v_1]) \\ p_2 &= \mathrm{LeastCommonExit}(\uparrow_V^P [v_2]) \end{aligned}$$

$\mathrm{LeastCommonEntry}$ bzw. $\mathrm{LeastCommonExit}$ sind nötig für den Fall, dass v_1 oder v_2 Aggregationsknoten sind. v_3 ist der Vorgänger von v_1 und v_4 der Nachfolger von v_2 *vor* der parallelen (bedingten) Einfügung in V. p_3 und p_4 sind die dazu korrespondierenden Knoten (ebenfalls wieder unter Berücksichtigung von Aggregationsknoten):

$$\begin{aligned} p_3 &= \mathrm{LeastCommonExit}(\uparrow_V^P [v_3]) \\ p_4 &= \mathrm{LeastCommonEntry}(\uparrow_V^P [v_4]) \end{aligned}$$

Nun ist einfach zu erkennen, ob sich zwischen p_3 und p_1 bzw. p_2 und p_4 Knoten befinden, die in der Sicht reduziert wurden. (*BranchSuccessor* und *BranchPredecessor* werden verwendet für den Fall, dass p_3 ein Split- bzw. p_4 ein Join-Knoten ist). Algorithmus 4.3 beschreibt das weitere Vorgehen.

Algorithmus 4.3 UpdateInsertParallel(v_1, v_2, a)

$P, v_1, v_2, a, p_1, p_2, p_3, p_4$ `siehe Text`
`Setze` $q_1 := p_1$ `und` $q_2 := p_2$
`Wenn` $p_3 \neq \bullet p_1$ **(Vorbereich reduziert)**

$$\text{Ändere } q_1 := \begin{cases} \text{BranchSuccessor}(p_3, p_1) & \text{InsertBlockFirst} = \text{Early} \\ p_1 & \text{InsertBlockFirst} = \text{Late} \end{cases}$$

`Wenn` $p_2 \bullet \neq p_4$ **(Nachbereich reduziert)**

$$\text{Ändere } q_2 := \begin{cases} p_2 & \text{InsertBlockLast} = \text{Early} \\ \text{BranchPredecessor}(p_4, p_2) & \text{InsertBlockLast} = \text{Late} \end{cases}$$

InsertParallel(q_1, q_2, a, n_1, n_2)
Simplify(P)
UpdateViewsInsertParallel(q_1, q_2, a)

Im Fall von UpdateInsertConditional wird am Ende von Algorithmus 4.3 *InsertConditional* und *UpdateViewsInsertConditional* verwendet.

4.2.1 UpdateViewsInsertParallel, UpdateViewsInsertConditional

UpdateViewsInsertParallel(q_1, q_2, a) und UpdateViewsInsertConditional$(q_1, q_2 a)$ aktualisieren die verbleibenden Sichten indem auf jede Sicht $W = V_k$ $(k = 2 \ldots n)$ Algorithmus 4.4 angewendet wird. Beide sind neben den von *UpdateViewsInsertSerial* auch verwendeten Parametern:

$$\begin{aligned} \text{ChangeInReductionComplete} &\in \{\text{Reduce}, \text{Show}\} \\ \text{ChangeInAggregateComplete} &\in \{\text{Aggregate}, \text{Unaggregate}\} \end{aligned}$$

abhängig von zwei weiteren Parametern:

$$\begin{aligned} \text{ChangeInReductionPartly} &\in \{\text{Expand}, \text{Show}\} \\ \text{ChangeInAggregatePartly} &\in \{\text{Expand}, \text{Decrease}\}. \end{aligned}$$

Der Parameter ChangeInReductionComplete legt fest, ob eine Aktivität eingefügt wird, falls die Einfüge-Positionen für die Gateway-Knoten in einem vollständig reduzierten Bereich

liegen. Der Parameterwert Reduce bedeuted, die Aktivität auch zu reduzieren. Der Parameterwert Show dagegen fügt die Aktivität ein und zwar seriell zwischen dem nächsten sichtbaren Vorgänger und Nachfolger.

Der Parameter $\mathrm{ChangeInReductionPartly}$ bestimmt, ob eine Aktivität eingefügt wird, wenn die Einfüge-Positionen von einem oder beiden Gateway-Knoten in reduzierten Bereichen liegen. Expand dehnt den reduzierten Bereich auf die einzufügenden Aktivität aus, das heißt, sie wird auch reduziert. Dagegen bewirkt der Parameterwert Show, dass die Aktivität parallel bzw. konditional zum zwischen den Einfüge-Positionen verbliebenen Bereich eingefügt wird.

Mit dem Parameter $\mathrm{ChangeInAggregateComplete}$ wird in dem Fall, dass die Einfüge-Positionen beider Gateway-Knoten in *einem* aggregierten Bereich liegen, entschieden, ob eine Aktivität in den Aggregationsknoten eingefügt wird (d.h. Parameterwert $\mathrm{Aggregate}$), oder ob der Aggregationsknoten durch das entsprechende SESE-Fragment aus dem CPM ersetzt wird, bevor die Einfügung der Aktivität stattfindet (d.h. Parameterwert $\mathrm{Unaggregate}$).

Vom Parameter $\mathrm{ChangeInAggregatePartly}$ hängt ab, ob ein aggregierter Bereich ausgedehnt oder verkleinert wird, falls die Einfüge-Position von einem oder beiden Gateway-Knoten in aggregierten Bereichen liegen. Mit dem Parameterwert Expand wird der Aggregationsbereich soweit ausgedehnt, dass das gesamte ursprüngliche SESE-Fragment (zu dem a parallel eingefügt wurde in V) – abzüglich eventuell reduzierter Knoten – umfasst wird. Mit dem Parameterwert $\mathrm{Decrease}$ wird eingestellt, dass ein aggregierter Bereich vor der Einfügung durch das entsprechende SESE-Fragment aus dem CPM ersetzt wird.

Algorithmus 4.4 präzisiert die Verwendung der genannten Parameter. Die Knoten q_1 und q_2 sind der Eingangs- bzw. Ausgangsknoten des SESE-Fragments zu dem die Aktivität a parallel bzw. konditional mit den Gateway-Knoten $n_1 = (\bullet a)_{\mathbb{E}}$ (Split) und $n_2 = (a\bullet)_{\mathbb{E}}$ (Join) im CPM eingefügt wurde (in Kapitel 4.2.2 wird begründet, warum Vorgänger und Nachfolger von a verwendet werden müssen). Voraussetzung ist, dass die Block-Vereinfachungs-Operationen ausgeschaltet sind, das heißt, die Parameter *SimplifyBlocksAND*, *-XOR*, *SimplifyMultipleBlocksAND* und *-XOR* haben den Parameterwert *No*. Bei *UpdateViewsInsertConditional* muss an den entsprechenden Stellen *InsertConditional* statt *-Parallel* verwendet werden.

Algorithmus 4.4 $\mathrm{UpdateViewsInsertParallel}(q_1, q_2, a)$

$P, W, q_1, q_2, a, n_1, n_2$ `siehe Text`
$w_1 = \downarrow_W^P [q_1]$
$w_2 = \downarrow_W^P [q_2]$

Wenn w_1=$\emptyset$ und w_2=$\emptyset$ und von den Knoten in P im Block $n_1 \ldots n_2$ sind in W alle reduziert **(SESE-Fragment $q_1 \ldots q_2$ reduziert)**

Falls ChangeInReductionComplete = Reduce

Fertig.

Falls ChangeInReductionComplete = Show

Setze $w_3 :=$ VisiblePredecessor(n_1, P, W)

Setze $w_4 :=$ VisibleSuccessor(n_2, P, W)

Wenn $w_3 = w_4$ **(Vorgänger und Nachfolger aggregiert)**

Falls ChangeInAggregateComplete = Aggregate

Füge (w_3, a) zu C_k hinzu

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE(w_3)

InsertSerial(VisiblePredecessor(n_1, P, W), VisibleSuccessor$(n_2, P, W), a)$

Simplify(W)

Sonst

InsertSerial(w_3, w_4, a)

Sonst, wenn $w_1 = \emptyset$ oder $w_2 = \emptyset$ **(q_1 und/oder q_2 reduziert)**

Falls ChangeInReductionPartly = Expand

Fertig.

Falls ChangeInReductionPartly = Show

Setze $w_3 := w_1$ und $w_4 := w_2$

Wenn $w_1 = \emptyset$

Ändere $w_3 :=$ VisibleSuccessor(q_1, P, W)

Wenn $w_2 = \emptyset$

Ändere $w_4 :=$ VisiblePredecessor(q_2, P, W)

InsertParallel(w_3, w_4, a, n_1, n_2)

Simplify(W)

Sonst, wenn $w_1 = w_2$ und w_1 ist Aggregationsknoten **(aggregiert)**

Falls ChangeInAggregateComplete = Aggregate

Füge (w_1, a) zu C_k hinzu

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE(w_1)

InsertParallel$(\downarrow_W^P [q_1], \downarrow_W^P [q_2], a, n_1, n_2)$

Simplify(W)

Sonst, wenn w_1 und/oder w_2 ist Aggregationsknoten
(einer oder beide aggregiert (getrennt))

Falls $\mathrm{ChangeInAggregatePartly} = \mathrm{Expand}$

$w_1 \ldots w_2$ bilden ein SESE-Fragment S
$\alpha = \mathrm{AggregateSESE}(S)$
Füge (α, a) zu C_K hinzu

Falls $\mathrm{ChangeInAggregatePartly} = \mathrm{Decrease}$

Wenn w_1 ein Aggregationsknoten ist

$\mathrm{UnaggregateSESE}(w_1)$

Wenn w_2 ein Aggregationsknoten ist

$\mathrm{UnaggregateSESE}(w_2)$

$\mathrm{InsertParallel}(\downarrow_W^P [q_1], \downarrow_W^P [q_2], a, n_1, n_2)$
$\mathrm{Simplify}(W)$

Sonst **(keine Änderung bei q_1 und q_1)**

$\mathrm{InsertParallel}(w_1, w_2, a, n_1, n_2)$

4.2.2 CPM-Vereinfachung vor View-Update

Nach dem Einfügen einer Aktivität in eine Sicht wird zuerst das CPM und dann die anderen Sichten aktualisiert. Die Vereinfachung des CPMs muss *vor* der Aktualisierung der anderen Sichten stattfinden, weil sonst eventuell in den aktualisierten Sichten Gateway-Knoten verbleiben, die nach der Vereinfachung des CPMs nicht mehr im CPM vorhanden sind. Dieser Fall darf aber nicht eintreten, weil es zu jedem Knoten in einer Sicht einen korrespondierenden Knoten im CPM geben muss. Der Prototyp verhindert diesen Fall, indem das CPM immer vor den View-Updates vereinfacht wird.

Abbildung 4.9 veranschaulicht diesen Fall. In der linken Spalte ist das CPM zu sehen, in der rechten eine Sicht, die durch die Reduktion von *E* entstanden ist. Schritt 3 sei der Effekt einer Aktualisierung des CPMs nach dem Einfügen der Aktivität *I* in einer weiteren Sicht (die hier aus Platzgründen nicht dargestellt ist). Kern der Abbildung ist, dass nach dem View-Update (Abbildung 4.9 d)) die Gateway-Knoten *H* und *J* in View 1 noch enthalten sind, während diese beiden Knoten im CPM nach der Vereinfachung (Abbildung 4.9 e)) nicht mehr enthalten sind. Abbildung 4.9 f) zeigt das erwünschte Ergebnis des Updates *nach* der CPM-Vereinfachung.

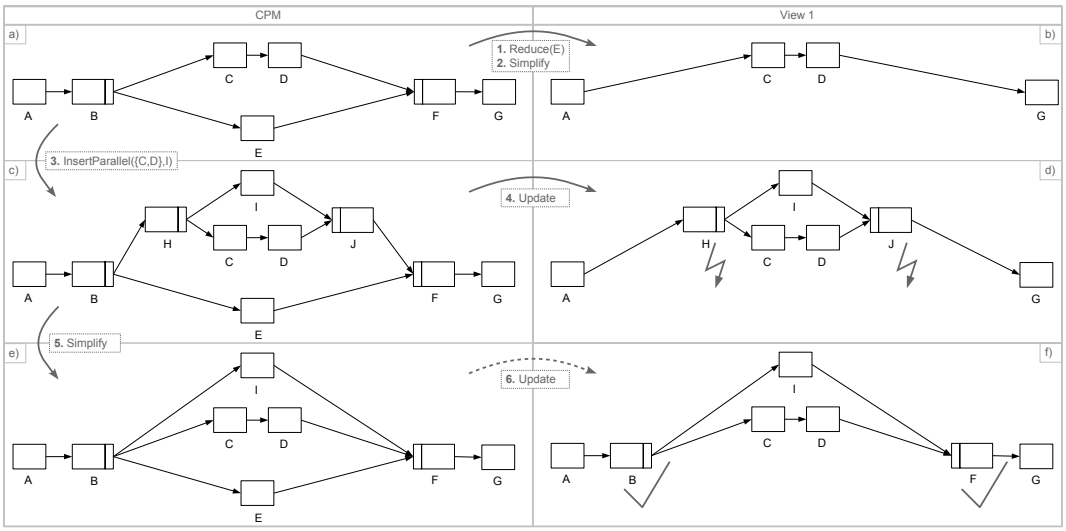

Abbildung 4.9: Beispiel für den Effekt, der auftritt, wenn das CPM *nach* der Aktualisierung der anderen Sichten vereinfacht wird.

Bei den View-Updates muss dann berücksichtigt werden, dass nicht die für das CPM-Update neu eingefügten Gateway-Knoten (bzw. deren korrespondierende Knoten) verwendet werden dürfen, sondern der Vorgänger und der Nachfolger des im CPM neu eingefügten Knotens. In Abbildung 4.9 wären dies *B* und *F* statt *H* und *J*.

Eine Alternative wäre gewesen, auch ein Update für die Simplify-Operation auf den Sichten durchzuführen, indem die Gateway-Knoten der Sichten an die des CPMs angepasst werden, was jedoch mehr Implementierungs- und Rechenaufwand bedeuted hätte.

4.3 Zusammenfassung

Das Einfügen einer Aktivität geschieht im vorgestellten Konzept in drei Schritten. Als erstes wird die Aktivität in einer Sicht eingefügt, anschließend das zentrale Prozessmodell und zum Schluss die weiteren Sichten aktualisiert. Insgesamt sieben Parameter (siehe Tabelle 5.1) steuern das Vorgehen in den Fällen, in denen in Prozesssichten Aktivitäten reduziert oder aggregiert wurden, um resultierende Mehrdeutigkeiten aufzulösen.

Der vorgegebene Gesamtablauf und die Wahl der Kontrollflusskonstrukte (beispielsweise keine Synchronisationskanten) schränkt die Verwendung der Parameter jedoch ein. Die Wahl $\mathrm{InsertSingle} = \mathrm{Parallel}$ ist unter diesen Voraussetzungen nicht für vollständig automatisierte Aktualisierungen geeignet, da in einem reduzierten Bereich nur die erste Einfügung

erfolgreich sein wird. Bei allen weiteren Einfügungen in diesem Bereich muss der Benutzer eingreifen. Mit einem veränderten Gesamtablauf und weiteren Kontrollflusskonstrukten könnte diese Problem gelöst werden.

Werden Vereinfachung-Operatoren auf die Sichten angewendet, sind aufwändigere Algorithmen für die Aktualisierung der Sichten nötig. Es müssen nämlich nicht nur Reduktionen und Aggregationen berücksichtigt werden, sondern auch die Effekte der Block-Vereinfachungs-Operatoren (SimplifyBlocksAND bzw. -XOR und SimplifyMultipleBlocksAND bzw. -XOR). Diese Fälle müssen von den Algorithmen erkannt und die bei der Vereinfachung entfernten Gateway-Knoten teilweise wiederhergestellt werden.

5 Implementierungsaspekte

In Kapitel 5.1 werden für die Implementierung wichtige Algorithmen eingeführt. In Kapitel 5.2 werden wichtige Aspekte des Prototyps gezeigt – die Realisierung der Parameter für das zentrale Prozessmodell und für Sichten; ein Prozesssichten-Log, dass eine Historie der Operationen auf Prozesssichten auflistet; die bereichsweise Vereinfachung (Ignore Simplification) sowie das automatische Layout der Prozessgraphen.

5.1 Algorithmen

Im Folgenden werden Algorithmen beschrieben, auf die im Laufe der vorigen Kapitel Bezug genommen wurde und die für die Implementierung von Prozesssichten unverzichtbar sind. Es werden jeweils wohlgeformte, azyklische Graphen für Prozessschemas und -sichten entsprechend der Definitionen 1 und 9 vorausgesetzt (siehe Kapitel 2).

5.1.1 Sichtbare Vorgänger und Nachfolger

Sind die Block-Vereinfachungen *SimplifyMultipleBlocksAND* und *-XOR* ausgeschaltet, hat in einer Sicht jede reduzierte Aktivität jeweils genau einen sichtbaren Vorgänger und Nachfolger. Werden dagegen diese beiden Block-Vereinfachungen durchgeführt, kann eine reduzierte Aktivität jedoch mehrere sichtbare Vorgänger oder Nachfolger haben. In diesem Fall müssen alle sichtbaren Vorgänger bzw. Nachfolger in *einem* gemeinsamen Block enthalten sein. Außerdem kann ein Knoten dann *entweder* genau einen sichtbaren Vorgänger und einen oder mehrere sichtbare Nachfolger haben *oder* umgekehrt einen oder mehrere sichtbare Vorgänger und genau einen sichtbaren Nachfolger. Das bedeuted insbesondere, dass die Funktionen $\mathrm{VisiblePredecessor}$ und $\mathrm{VisibleSuccessor}$ (siehe Definition 13) in einem solchen Fall nicht eindeutig sind. In Definition 15 werden daher die Funktionen $\mathrm{VisiblePredecessors}$ und $\mathrm{VisibleSuccessors}$ definiert, die *alle* sichtbaren Vorgänger bzw. Nachfolger beinhalten.

Definition 15. ($\mathrm{VisiblePredecessors}$ und $\mathrm{VisibleSuccessors}$)

Sei $P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ ein Prozessschema und $V = V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ eine Sicht. Sei $p_0 \in N$ in V reduziert, das heißt, $\downarrow_V^P [p_0] = \emptyset$. Dann sind die *Mengen der sichtbaren Vorgänger* bzw. *Nachfolger* von p_0 in V definiert als

$$\begin{aligned}\mathrm{VisiblePredecessors}(p_0, P, V) = \{ \downarrow_V^P [p] | p \in (\star p_0)_{\mathbb{E}} \text{ mit } \downarrow_V^P [p] \neq \emptyset \text{ und} \\ \text{es existiert kein Pfad von } p \text{nach } p_0 \\ \text{der einen Knoten } q \text{enthält mit } \downarrow_V^P [q] \neq \emptyset\} \\ \mathrm{VisibleSuccessors}(p_0, P, V) = \{ \downarrow_V^P [p] | p \in (p_0 \star)_{\mathbb{E}} \text{ mit } \downarrow_V^P [p] \neq \emptyset \text{ und} \\ \text{es existiert kein Pfad von } p_0 \text{nach } p \\ \text{der einen Knoten } q \text{enthält mit } \downarrow_V^P [q] \neq \emptyset\}.\end{aligned}$$

Dabei ist $\star p_0$ bzw. $p_0 \star$ die Menge aller direkten und indirekten Vorgänger bzw. Nachfolger von p_0 (siehe Definition 2 in Kapitel 2).

Algorithmus 5.1 zeigt eine Möglichkeit, die sichtbaren Nachfolger eines Knotens zu bestimmen. Der Algorithmus sucht rekursiv mithilfe der Funktion $\mathrm{FindVisibleSuccessors}$ in Blöcken nach sichtbaren Nachfolgern. Algorithmus 5.2 beschreibt den konkreten Ablauf von $\mathrm{FindVisibleSuccessors}$. Die Algorithmen für $\mathrm{VisiblePredecessors}$ bzw. $\mathrm{FindVisiblePredecessors}$ sind analog dazu und werden daher nicht weiter beschrieben.

Algorithmus 5.1 $\mathrm{VisibleSuccessors}(p_0, P, V) \rightarrow S$

$P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ `Prozessschema`
$V = V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ `Sicht`
$p_0 \in N : \mathrm{NT}(p) \neq$ Endflow `der Knoten dessen sichtbare Nachfolger gesucht sind`
$S \subset N'$`: sichtbare Nachfolger von` p_0
`Wenn` $\mathrm{NT}(p_0) =$ Endflow

    `Fehler: Endflow-Knoten hat keinen Nachfolger.`

`Sonst`

    $S :=$ $\mathrm{FindVisibleSuccessors}(p, \mathtt{nil}, p, P, V)$

Algorithmus 5.2 FindVisibleSuccessors$(p, q, p_0, P, V) \rightarrow U$

$P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ Prozessschema
$V = V(P) = (N', \mathbb{E}', \mathrm{EC}', \mathrm{NT}', C)$ Sicht
$p \in N : \mathrm{NT}(p) \neq \mathrm{Endflow}$ der Knoten dessen sichtbare Nachfolger gesucht sind
$q \in N$ Join-Knoten, bei dem die Suche beendet werden soll
$p_0 \in N$ Knoten, dessen sichtbare Nachfolger initial gesucht sind (siehe Algorithmus 5.1)
$S \subset N'$: in V sichtbare Nachfolger von p
E einfache Kantenmenge von $\mathbb{E}$
Initialisiere $n = p$
Wiederhole solange $n \neq q$

    Wenn $(n \neq p_0$ und $\downarrow_V^P [n] \neq \emptyset)$ oder $\mathrm{NT}(n) = \mathrm{Endflow}$
        $S = \{\downarrow_V^P [n]\}$
        Fertig.
    Falls $\mathrm{NT}(n) \in \{\mathrm{Startflow}, \mathrm{Activity}, \mathrm{ANDJoin}, \mathrm{XORJoin}\}$
        Setze $n := n\bullet$
    Falls $\mathrm{NT}(n) \in \{\mathrm{ANDSplit}, \mathrm{XORSplit}\}$
        Initialisiere $S = \emptyset$
        Für alle Nachfolger r von n $((n, r) \in E)$
            $S := S \cup \mathrm{FindVisibleSuccessors}(r, \mathrm{Join}(n), p_0, P, V)$
        Wenn $S \neq \emptyset$
            Fertig.
        Setze $n := \mathrm{Join}(n)$

5.1.2 Kleinstes gemeinsames SESE-Fragment

Entsprechend Definition 8 berechnet Algorithmus 5.3 den Ein- und Ausgangsknoten des kleinsten gemeinsamen SESE-Fragments einer Menge von Knoten M. Diese Knotenmenge M ist üblicherweise eine Teilmenge der Knoten des kleinsten gemeinsamen SESE-Fragments. Zum Beispiel kann M nur die Aktivitäten eines SESE-Fragments enthalten, aber keine Gateway-Knoten (wie im Fall von $\mathrm{UnaggregateSESE}$; siehe Kapitel 2). Liegen diese Aktivitäten zum Beispiel in einem Zweig hintereinander in verschiedenen Blöcken oder in einem Block in verschiedenen Zweigen kann ein Gateway-Knoten der Ein- bzw. Ausgangsknoten sein. Liegen die Knoten in verschiedenen Zweigen *eines* Blocks, entspricht $\mathrm{LeastCommonSESE}(M)$ dem kleinsten gemeinsamen Block der Knoten in M.

Zur Berechnung des kleinsten gemeinsamen SESE-Fragments bzw. des Ein- und Ausgangsknotens greift Algorithmus 5.3 auf die Algorithmen für den kleinsten gemeinsamen SESE-Eingang bzw. -Ausgang zurück ($\mathrm{LeastCommonEntry}$ bzw. $\mathrm{LeastCommonExit}$). Algorithmus 5.4 beschreibt die Vorgehensweise zur Bestimmung von $\mathrm{LeastCommonExit}$. Ein Algorithmus für $\mathrm{LeastCommonEntry}$ ist analog dazu und wird nicht gesondert beschrieben.

Algorithmus 5.3 $\mathrm{LeastCommonSESE}(M) \rightarrow (n_1, n_2)$

$P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ `Prozessschema`
$M \subset N : |M| > 1$ `Menge von Knoten, für die das kleinste gemeinsame SESE-Fragment berechnet werden soll,` M `darf nicht den Start- oder Endknoten enthalten`
$n_1, n_2 \in N$ `Ein-/Ausgang des kleinsten gemeinsamen SESE-Fragments`
`Setze` $n_1 := \mathrm{LeastCommonEntry}(M)$
`Setze` $n_2 := \mathrm{LeastCommonExit}(M)$

Der kleinste gemeinsame SESE-Ausgang $\mathrm{LeastCommonExit}(M)$ einer Knotenmenge $M \subset N$ kann mit Algorithmus 5.4 berechnet werden (der Eingang $\mathrm{LeastCommonEntry}$ wird analog bestimmt).

Die Berechnung erfolgt in zwei Phasen. In der ersten Phase ($\mathrm{FirstSweepForward}$) wird – beginnend bei einem beliebigen Knoten aus M – der Graph in Vorwärtsrichtung traversiert bis zum End-Knoten. In der zweiten Phase ($\mathrm{SweepForward}$) werden die restlichen Knoten aus M verarbeitet durch Traversieren des Graphs beginnend bei jedem dieser restlichen Knoten bis zu einem Knoten, der bereits verarbeitet wurde. Der Ablauf der beiden Phasen ist durch die Algorithmen 5.5 und 5.6 definiert (für $\mathrm{LeastCommonEntry}$ sind $\mathrm{FirstSweepBackward}$ und $\mathrm{SweepBackward}$ analog; der Graph wird dann entgegen der Kantenrichtung traverisert).

Jeder Knoten des Prozessgraphens wird dabei höchstens einmal betrachtet (die Menge $done$ enthält die zu einem Zeitpunkt bereits bearbeiteten Knoten). Wird in der zweiten Phase ein Knoten angetroffen, der bereits bearbeitet wurde, muss der Graph für diesen Knoten nicht weiter traversiert werden. Daher hat Algorithmus 5.4 den Berechnungsaufwand $O(|N|)$.

Algorithmus 5.4 $\mathrm{LeastCommonExit}(M) \rightarrow n_2$

$P = (N, E, \mathrm{EC}, \mathrm{NT})$ einfaches Prozessschema
$M \subset N : M \neq \emptyset$ Menge von Knoten, für die der kleinste gemeinsame SESE-Ausgang berechnet werden soll
$n_2 \in N$ kleinster gemeinsamer SESE-Ausgang von M
Falls $|M| = 1$

    Dann ist $\{n_2\} = M$, Fertig.

$todo = M$ Menge der noch zu bearbeitenden Knoten
$done = \emptyset$ Menge der erledigten Knoten
J_1 Menge von Join-Knoten, die für mindestens einen Knoten aus M relevant sind
J_2 Menge von Join-Knoten, die für mindestens zwei Knoten aus M relevant oder selbst in M enthalten sind
$tailList$ Liste von Aktivitäten oder Join-Knoten, inital leer
$\mathrm{FirstSweepForward}(todo, done, J_1, J_2, tailList)$
Solange $todo \neq \emptyset$ wiederhole

    $\mathrm{SweepForward}(todo, done, J_1, J_2)$

Setze exit := *letzter* Knoten n aus $tailList$ mit

    $\mathrm{NT}(n) = \mathrm{Activity}$ oder $n \in J_2$

Der Algorithmus $\mathrm{FirstSweepForward}$ bestimmt initial die Mengen J_1 (Join-Knoten, die relevant für einen Knoten aus der $todo$-Menge sind) und J_2 (Join-Knoten, die relevant für mindestens zwei Knoten sind) und die Liste $tailList = (n_1, \ldots, n_i, \ldots, n_k)$, deren letzter Eintrag n_i mit $\mathrm{NT}(n_i) = \mathrm{Activity}$ oder $n_i \in J_2$ der gesuchte SESE-Ausgang ist.

Algorithmus 5.5 $\mathrm{FirstSweepForward}(todo, done, J_1, J_2, tailList)$

$M, N, \mathrm{NT}, todo, done, J_1, J_2, tailList$ siehe Algorithmus 5.4
$n \in N : n =$ wähle beliebigen Knoten aus $todo$

```
Wiederhole
    Falls NT(n) ∈ {Activity, Startflow}
        Wenn n ∈ todo
            Verschiebe n von todo nach done
            Hänge n an das Ende von tailList an
    Falls NT(n) ∈ {ANDSplit, XORSplit}
        Sei j = Join(n)
        Für jeden Knoten m im Block n...j (inklusive n und j)
            Wenn m ∈ todo
              Verschiebe m von todo nach done
        Wenn mindestens ein Knoten verschoben wurde
            Füge j in J2 ein
            Hänge j an das Ende von tailList an
        Setzte n := j
    Falls NT(n) ∈ {ANDJoin, XORJoin}
        Wenn n ∈ todo
            Verschiebe n von todo nach done
            Füge n in J2 ein
        Füge n in J1 ein
        Hänge n an das Ende von tailList an
    Falls NT(n) = Endflow
        Fertig.
    Für die nächste Wiederholung setze n := n•
```

SweepForward wird für alle verbleibenden Knoten aus der Menge $todo$ verwendet und traversiert den Prozessgraphen beginnend bei einem aus der Menge $todo$ ausgewählten Knoten in Vorwärtsrichtung bis ein Knoten angetroffen wird, der in der Menge $done$ oder J_1 enthalten ist. Ist ein Join-Knoten bereits in J_1 enthalten, bedeuted dies, dass mindestens zwei unterschiedliche Zweige dieses Join-Knotens Knoten aus M enthalten.

Algorithmus 5.6 SweepForward(todo, done, J_1, J_2)

```
M, N, NT, todo, done, J1, J2, tailList  siehe Algorithmus 5.4
n ∈ N : n = wähle beliebigen Knoten aus todo
```

```
Wiederhole
    Falls NT(n) ∈ {Activity, Startflow}
        Wenn n ∈ done
            Fertig.
        Wenn n ∈ todo
            Verschiebe n von todo nach done
    Falls NT(n) ∈ {ANDSplit, XORSplit}
        Sei j = Join(n)
        Setze containsDone := FALSE
        Setze containsTodo := FALSE
        Für jeden Knoten m im Block n...j (inklusive n und j)
            Wenn m ∈ done
              Setze containsDone := TRUE
              Fahre bei (1) fort
            Wenn m ∈ todo
              Setze containsTodo := TRUE
              Verschiebe m von todo nach done
        (1):
        Wenn containsDone = TRUE oder containsTodo = TRUE
            Füge j in J2 ein
        Wenn containsDone = TRUE
            Fertig.
        Setzte n := j
    Falls NT(n) ∈ {ANDJoin, XORJoin}
        Wenn n ∈ todo
            Verschiebe n von todo nach done
        Wenn n ∈ J1
            Füge n in J2 ein
            Fertig.
        Sonst
            Füge n in J1 ein
    Für die nächste Wiederholung setze n := n•
```

5.1.3 UnaggregateSESE

Das Ersetzen eines Aggregationsknotens durch das aggregierte SESE-Fragment geschieht mithilfe von Algorithmus 5.7. $\mathrm{UnaggregateSESE}$ wird zum Beispiel verwendet, wenn eine Einfügung in einem aggregierten Bereich stattfindet und der Parameter $\mathrm{ChangeInAggregateComplete}$ den Wert $\mathrm{Unaggregate}$ hat.

Im ersten Schritt werden die aggregierten Aktivitäten und *alle* Gateway-Knoten im kleinsten gemeinsamen SESE-Fragment der Aktivitäten wiederhergestellt. Im zweiten Schritt werden die Kanten zwischen diesen Knoten eingefügt. Gateway-Knoten, die durch Reduktionen (vor der Aggregation) überflüssig geworden sind, können bei Bedarf durch eine anschließende Vereinfachung mit der $\mathrm{Simplify}$-Operation entfernt werden.

Algorithmus 5.7 $\mathrm{UnaggregateSESE}(\alpha)$

$P = (N, \mathbb{E}, \mathrm{EC}, \mathrm{NT})$ Prozessschema, E einfache Kantenmenge von $\mathbb{E}$
$V = V(P) = (N', E', \mathrm{EC}', \mathrm{NT}', C)$ Prozesssicht
$\alpha \in N' : \mathrm{NT}'(\alpha) = \mathrm{AggregationNode}$ Aggregationsknoten, der durch das aggregierte SESE-Fragment ersetzt werden soll
$(\mathrm{entry}, \mathrm{exit}) = \mathrm{LeastCommonSESE}(\uparrow_V^P [\alpha])$
Komponentenknoten *und* Gateway-Knoten rekonstruieren, dabei reduzierte Knoten auslassen:
$M = \emptyset$
Für jeden Knoten n zwischen entry und exit (beide inklusive)
    Wenn n ein Gateway-Knoten ist oder $n \in \uparrow_V^P [\alpha]$
        Füge n zu M hinzu
Setze $N' := N' \cup M$
Wiederherstellen der Kanten:
Für jeden eingefügten Knoten $n_1 \in M$
    Wenn $n_1 \neq \mathrm{exit}$
        Für jede von n_1 ausgehende Kante $e = (n_1, n_2) \in E$
            $n_1' = \downarrow_V^P [n_1]$
            Falls der Nachfolger von n_1 in der Sicht reduziert wurde, verwende den sichtbaren Nachfolger:

$$n_2' = \begin{cases} \downarrow_V^P [n_1\bullet] & \downarrow_V^P [n_1\bullet] \neq \emptyset \\ \mathrm{VisibleSuccessor}(n_1, P, V) & \text{sonst} \end{cases}$$

            $\mathbb{E}' := \mathbb{E}' \oplus \{(n_1', n_2')\}$

$\mathbb{E}' := \mathbb{E}' \oplus \{(\bullet\alpha, \downarrow_V^P [\text{entry}]), (\downarrow_V^P [\text{exit}], \alpha\bullet)\}$
Entferne α aus N'
Entferne die Kanten $(\bullet\alpha, \alpha)_{\mathbb{E}'}$ und $(\alpha, \alpha\bullet)_{\mathbb{E}'}$ aus $\mathbb{E}'$

5.1.4 Aktualisierung mit Block-Vereinfachungen

Der in Kapitel 4 gezeigte Algorithmus 4.2 zur Aktualisierung von Sichten bei einer seriellen Einfügung basiert darauf dass die Block-Vereinfachungs-Operationen ausgeschaltet sind. Algorithmus 5.8 erweitert diesen Algorithmus um die Fälle, in denen bei der Vereinfachung Gateway-Knoten entfernt wurden, die zum ordnungserhaltenden Aktualisieren aber wieder benötigt werden und daher rekonstruiert werden müssen.

Der Fall, dass *SimplifyMultipleBlocksAND* oder *-XOR* Gateway-Knoten entfernt haben, wird daran erkannt, dass ein Knoten *mehrere* sichtbare Vorgänger oder Nachfolger hat. Algorithmus 5.8 greift dann auf Algorithmus 5.9 ($\text{ReconstructMergedBlock}$) zurück, um die Gateway-Knoten wiederherzustellen.

Das Entfernen von Gateway-Knoten bei einfachen Blöcken durch *SimplifyBlocksAND* oder *-XOR* ist daran zu erkennen, dass sich zwischen dem sichtbaren Vorgänger und Nachfolger eines Knotens mindestens ein Knoten befindet, der im zentralen Prozessmodell nicht direkt zwischen diesen beiden Knoten liegt.

In Algorithmus 5.8 sind diese beiden Fälle jeweils mit der zu kompensierenden Block-Vereinfachungs-Operation gekennzeichnet.

Algorithmus 5.8 $\text{UpdateViewInsertSerial}(P, V, p_1, p_2, a)$

$P = (N, \mathbb{E}, \text{EC}, \text{NT})$ Prozessschema
$V = V(P) = (N', \mathbb{E}', \text{EC}', \text{NT}', C)$ zu aktualisierende Sicht
a ist die einzufügende Aktivität
$p_1, p_2 \in N$ Knoten, zwischen den a eingefügt wurde
Wenn $\downarrow_V^P [p_1] = \emptyset$ und $\downarrow_V^P [p_2] = \emptyset$ **(beide reduziert)**

Falls $\text{ChangeInReductionComplete} = \text{Reduce}$
Fertig.
Falls $\text{ChangeInReductionComplete} = \text{Show}$
$R = \text{VisiblePredecessors}(p_1, P, V)$
$S = \text{VisibleSuccessors}(p_2, P, V)$
Wenn $|R| = 1$ und $|S| > 1$ **(SimplifyMultipleBlocksAND/-XOR)**

Sei $\{r\} = R$ **(r muss Split sein)**
$(m_1, m_2) = \text{ReconstructMergedBlock}(V, P, r, \text{Join}(r), S)$
$\text{InsertSerial}(r, m_1, a)$

Sonst, wenn $|R| > 1$ und $|S| = 1$ **(SimplifyMultipleBlocksAND/-XOR)**
Sei $\{s\} = S$ **(s muss Join sein)**
$(m_1, m_2) = \text{ReconstructMergedBlock}(V, p, \text{Split}(s), s, R)$
$\text{InsertSerial}(m_2, s, a)$

Sonst, wenn $|R| = 1$ und $|S| = 1$
Seien $\{r\} = R$ und $\{s\} = S$
Wenn $r = s$ **(aggregiert)**
Falls ChangeInAggregateComplete = Aggregate
Füge (r, a) zu C hinzu
Falls ChangeInAggregateComplete = Unaggregate
$\text{UnaggregateSESE}(r)$
$\text{InsertSerial}(\text{VisiblePredecessor}(p_1, P, V), \text{VisibleSuccessor}(p_2, P, V), a)$
Sonst
Setze $q_1 := \text{BranchSuccessor}(\text{LeastCommonExit}(\uparrow_V^P [r]), p_1)$
Setze $q_2 := \text{BranchPredecessor}(\text{LeastCommonEntry}(\uparrow_V^P [s]), p_2)$
Setze $v_1 := \begin{cases} \downarrow_V^P [q_1] & \downarrow_V^P [q_1] \neq \emptyset \\ \text{VisibleSuccessor}(q_1, P, V) & \text{Sonst} \end{cases}$
Setze $v_2 := \begin{cases} \downarrow_V^P [q_2] & \downarrow_V^P [q_2] \neq \emptyset \\ \text{VisibleSuccessor}(q_2, P, V) & \text{Sonst} \end{cases}$
Wenn $s \neq v_1$ **(SimplifyBlocksAND,-XOR)**
$(m_1, m_2) = \text{LeastCommonSESE}(\{p_1, p_2\} \cup \uparrow_V^P [v_1] \cup \uparrow_V^P [v_2])$
(m_1 und m_2 sind Split- bzw. Join-Knoten des kleinsten gemeinsamen Blocks)
Falls $\text{NT}(m_1) = \text{ANDSplit}$
$\text{InsertParallel}(v_1, v_2, a, m_1, m_2)$
Falls $\text{NT}(m_1) = \text{XORSplit}$
$\text{InsertConditional}(v_1, v_2, a, m_1, m_2)$
Sonst
$\text{InsertSerial}(r, s, a)$

Sonst, wenn $\downarrow_V^P [p_1] = \downarrow_V^P [p_2]$ **(beide aggregiert)**

Falls ChangeInAggregateComplete = Aggregate

Füge $(\downarrow_V^P [p_1], a)$ zu C hinzu

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE($\downarrow_V^P [p_1]$)
InsertSerial($\downarrow_V^P [p_1], \downarrow_V^P [p_2], a$)

Sonst, wenn $\downarrow_V^P [p_1] \neq \emptyset$ und $\downarrow_V^P [p_2] \neq \emptyset$ **(ohne Änderung)**

InsertSerial($\downarrow_V^P [p_1], \downarrow_V^P [p_2], a$)

Sonst **(Vorgänger und/oder Nachfolger red.)**

R = VisiblePredecessors(p_1, P, V)
S = VisibleSuccessors(p_2, P, V)
Wenn $\downarrow_V^P [p_1] \neq \emptyset$ und $|S| > 1$ **(SimplifyMultipleBlocksAND/-XOR)**

(m_1, m_2) = ReconstructMergedBlock($V, P, \downarrow_V^P [p_1]$, Join($\downarrow_V^P [p_1]$), S)
InsertSerial($\downarrow_V^P [p_1], m_1, a$)

Sonst, wenn $\downarrow_V^P [p_2] \neq \emptyset$ und $|R| > 1$ **(SimplifyMultipleBlocksAND/-XOR)**

(m_1, m_2) = ReconstructMergedBlock(V, P, Split($\downarrow_V^P [p_2]$)$, \downarrow_V^P [p_2], R$)
InsertSerial($m_2, \downarrow_V^P [p_2], a$)

Sonst

Setze $v_1 := \begin{cases} \downarrow_V^P [p_1] & \downarrow_V^P [p_1] \neq \emptyset \\ r \in R & \text{Sonst } (|R| = 1!) \end{cases}$

Setze $v_2 := \begin{cases} \downarrow_V^P [p_2] & \downarrow_V^P [p_2] \neq \emptyset \\ s \in S & \text{Sonst } (|S| = 1!) \end{cases}$

Wenn $v_1 = v_1$ **(aggregiert)**

Falls ChangeInAggregateComplete = Aggregate

Füge (v_1, a) zu C hinzu

Falls ChangeInAggregateComplete = Unaggregate

UnaggregateSESE(v_1)

Setze $w_1 := \begin{cases} \downarrow_V^P [p_1] & \downarrow_V^P [p_1] \neq \emptyset \\ \text{VisiblePredecessor}(p_1, P, V) & \text{Sonst} \end{cases}$

Setze $w_2 := \begin{cases} \downarrow_V^P [p_2] & \downarrow_V^P [p_2] \neq \emptyset \\ \text{VisibleSuccessor}(p_2, P, V) & \text{Sonst} \end{cases}$

InsertSerial(w_1, w_2, a)

Sonst

$\text{InsertSerial}(\downarrow_V^P [p_1], \downarrow_V^P [p_2], a)$

$\text{Simplify}(V)$

Algorithmus 5.9 beschreibt nun den Algorithmus $\text{ReconstructMergedBlock}$, die je einen Split- und Join-Knoten wiederherstellt für eine gegebene Menge von sichtbaren Vorgängern oder Nachfolgern.

Algorithmus 5.9 $\text{ReconstructMergedBlock}(V, P, n_1, n_2, U') \rightarrow (m_1, m_2)$

$P = (N, \mathbb{E}, \text{EC}, \text{NT})$ Prozessschema
$V = V(P) = (N', \mathbb{E}', \text{EC}', \text{NT}', C)$ Sicht
$n_1, n_2 \in N'$: n_1 Split-Knoten, n_2 Join-Knoten
$U' \subset N'$ Menge von sichtbaren Vorgängern oder Nachfolgern
$U \subset N : U = \{\uparrow_V^P [u'] | u' \in U'\}$
$(m_1, m_2) = \text{LeastCommonSESE}(U)$ **(m_1 und m_2 sind Split- bzw. Join-Knoten des kleinsten gemeinsamen *Blocks* von U)**

Füge m_1 und m_2 zu N' hinzu
$\mathbb{E}' := \mathbb{E}' \oplus \{(n_1, m_1), (m_2, n_2)\}$
Setze $\text{EC}'(n_1, m_1) := \text{EC}(n_1, \text{BranchSuccessor}(n_1, m_1))$
Für alle $v \in \text{VisibleSuccessors}(m_1, P, V)$

  Entferne $(n_1, v)_{\mathbb{E}'}$ aus $\mathbb{E}'$
  $\mathbb{E}' := \mathbb{E}' \oplus (m_1, v)$
  Setze $\text{EC}'(m_1, v) := \text{EC}(m_1, \text{BranchSuccessor}(m_1, v))$

Für alle $v \in \text{VisiblePredecessors}(m_2, P, V)$

  Entferne $(v, n_2)_{\mathbb{E}'}$ aus $\mathbb{E}'$
  $\mathbb{E}' := \mathbb{E}' \oplus (v, m_2)$

Parameter	Werte	Verwendet von
InsertSingle	Early, Late, Parallel	UpdateInsertSerial
InsertBlockFirst	Early, Late	UpdateInsert-
InsertBlockLast		Parallel, -Conditional
ChangeInReductionComplete	Reduce, Show	UpdateViewsInsert-
ChangeInReductionPartly	Expand, Show	Serial, -Parallel, -Conditional
ChangeInAggregateComplete	Aggregate, Unaggregate	
ChangeInAggregatePartly	Expand, Decrease	
SimplifyEmptyBranchesAND	None, All, $k \in \mathbb{N}$	Simplify
SimplifyEmptyBranchesXOR	None, All	
SimplifyBlocksAND	Yes, No	
SimplifyBlocksXOR		
SimplifyMultipleBlocksAND		
SimplifyMultipleBlocksXOR		

Tabelle 5.1: Parameter, mögliche Werte und Verwendung.

5.2 Aspekte des Prototyps

In diesem Abschnitt werden grundlegende Aspekte des Prototyps gezeigt. Der Prototyp dieser Arbeit basiert auf dem Mobile-Services-Prototyp [6]. Mit diesem konnten bereits einzelne, unabhängige Prozessgraphen modelliert und dargestellt werden. Die Modellierung basierte beim Mobile-Services-Prototyp im Wesentlichen auf dem Einfügen einzelner Knoten und Kanten. Im Zuge dieser Arbeit wurde diese Art der Modellierung ersetzt durch die Verwendung der strukturellen Operationen *InsertSerial*, *InsertParallel* und *InsertConditional* (siehe Kapitel 2). Dadurch ist zum einen sichergestellt, dass ausschließlich wohlgeformte, azyklische Prozessgraphen modelliert werden können. Zum anderen sind diese Operationen bedingt durch das Einfügen von Aktivitäten über Sichten. Abbildung 5.4 zeigt das Hauptfenster des Prototyps. Das Hauptfenster beinhaltet die Darstellung des zentralen Prozessmodells sowie der Sichten neben der Parameterliste (siehe Kapitel 5.2.1) und einer Anzeige für ein Log.

5.2.1 Parameter und deren Vererbung

Die Parameter des zentralen Prozessmodells und der Sichten können jeweils in der *Parameterliste* (siehe Abbildung 5.4 am linken Rand) oder dem *Parametereditor* (siehe Abbildung 5.1) eingestellt werden.

Grundsätzlich werden Parameter vom zentralen Prozessmodell an die darauf basierenden Sichten vererbt. Ebenso erben Sichten auf Sichten die Parameter der übergeordneten

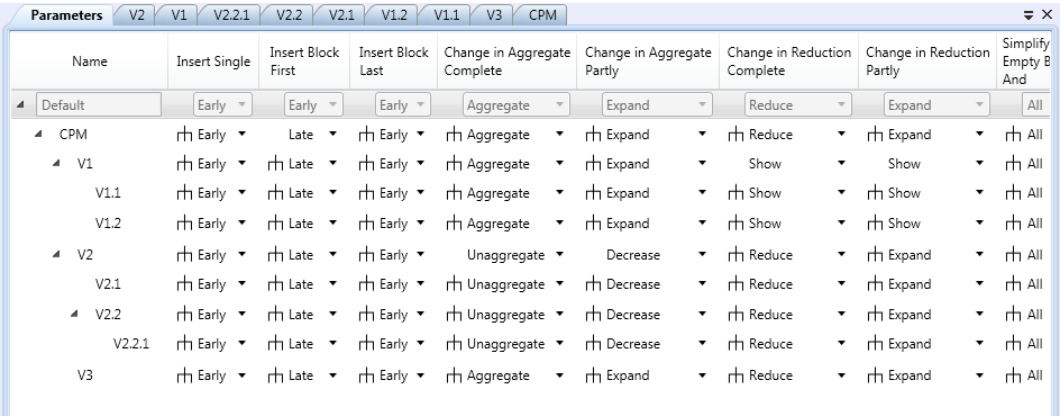

Abbildung 5.1: Der Parametereditor zeigt die View-Hierarchie in der ersten Spalte. In den restlichen Spalten können die Parameter für jede Sicht eingestellt werden. Anhand des ⫛-Symbols lässt sich die Parameter-Vererbung erkennen.

Sicht, auf der sie basieren. Sei beispielsweise P das zentrale Prozessmodell mit der Sicht $V_1 = V(P)$. W_1 sei wiederum eine Sicht auf V_1, das heißt, $W_1 = V(V_1)$. Dann erbt V_1 die Parameter von P und W_1 die Parameter von V_1. Alle Parameter können jedoch bei jeder Sicht *überschrieben* werden. Im Prototyp ist ein Standard-Parameter-Satz hinterlegt, von dem jedes zentrale Prozessmodell seine Parameter erbt. Das heißt, jeder Parameter hat einen von *proView* vorgegebenen Default-Wert. In Abbildung 5.1 in der ersten Zeile ist diese Menge mit *Default* bezeichnet. Das ⫛-Symbol zeigt an, dass ein Parameterwert geerbt ist. In Tabelle 5.1 sind alle Parameter mit ihren möglichen Werten zusammengefasst.

5.2.2 Bereichsweise Vereinfachung

Die Vereinfachung der Prozesssichten schafft zusätzliche Mehrdeutigkeiten beim Aktualisieren des zentralen Prozessmodells. Durch das Zusammenfassen oder Entfernen von Gateway-Knoten (*SimplifyBlocksAND*, *-XOR*, *SimplifyMultipleBlocksAND* und *-XOR*) können Einfüge-Positionen vor oder hinter Gateway-Knoten nicht eindeutig einer Position im CPM zugeordnet werden. Lässt sich diese Mehrdeutigkeit mit den Einfüge-Parametern *InsertSingle* und *InsertBlockFirst* bzw. *-Last* nicht hinreichend genau auflösen, kann die Vereinfachung bereichsweise ausgeschaltet werden. Der Bereich, in dem keine Vereinfachung stattfinden soll, muss ein SESE-Fragment sein und wird als Ignore-Simplification-Bereich bezeichnet (siehe Kapitel 3). Im Prototyp wird ein solcher Bereich von einem Klammerpaar umschlossen (siehe Abbildung 5.2). In einem Ignore-Simplification-Bereich bleiben

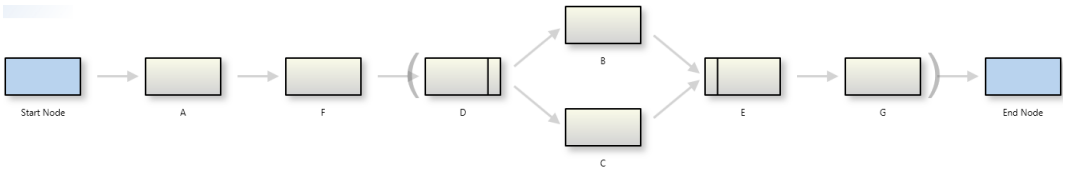

Abbildung 5.2: Die Klammer um das SESE-Fragment D bis G zeigt an, dass in diesem Bereich keine Vereinfachung stattfinden soll (Ignore Simplification).

alle Gateway-Knoten erhalten. Dadurch ist es möglich, die Einfüge-Position von Aktiviäten genauer zu wählen. (Durch Reduktionen können dennoch Mehrdeutigkeiten entstehen).

5.2.3 Sichten-Log

Im Sichten-Log sind alle bisherigen Sichtenerzeugungs-, Einfüge- und Vereinfachungs-Operationen in einer Historie aufgelistet in der Reihenfolge ihrer Anwendung. Die Einträge bilden eine Hierarchie, weil eine Operation weitere Operationen zur Folge haben kann. Zum Beispiel hat eine Einfügung die Aktualisierung des CPMs und der restlichen Sichten und verschiedene Vereinfachungen zur Folge. Mithilfe des Logs lässt sich die Entstehung einer Sicht durch Reduktionen und Aggregationen nachvollziehen ebenso wie die Effekte der Aktualisierungen und Vereinfachungen.

5.2.4 Layout

Für den Prototyp wurde ein automatisches Layout der Prozessgraphen realisiert. Die Knoten des zentralen Prozessmodells werden automatisch blockweise angeordnet (*Block-Layout*). Der Layout-Algorithmus ermittelt in einer ersten Phase den Platzbedarf jedes Blocks und ordnet die Knoten dann mit dieser Information in einer zweiten Phase blockweise an. Die vertikale Reihenfolge von Zweigen in Blöcken kann vom Benutzer festgelegt werden, indem der Benutzer die vertikale Position der Zweige anpasst. Der Block-Layout-Algorithmus behält die vom Benutzer vorgegebene Reihenfolge der Zweige bei.

Prozesssichten imitieren grundsätzlich das Layout des zentralen Prozessmodells indem für eine Sicht die Position der Knoten im CPM übernommen werden, nachdem das Block-Layout des CPMs berechnet wurde. Diese Imitation des Layouts unterstützt den Benutzer bei der visuellen Zuordnung von korrespondierenden Knoten zwischen Sichten bzw. dem CPM (siehe Abbildung 5.4). Optional können bei jeder Sicht die Knoten auch mit dem Block-Layout angeordnet werden.

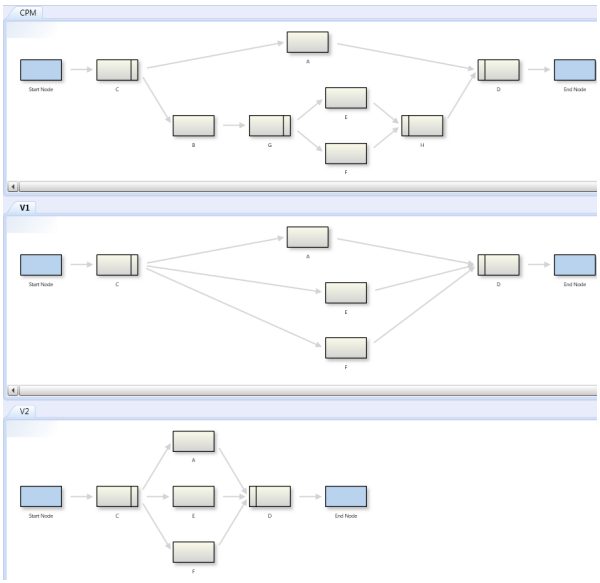

Abbildung 5.3: Beispiele für das automatische Layout. Das CPM (oben) hat ein Block-Layout. In den Sichten V1 und V2 wurde jeweils der selbe Knoten reduziert. V1 (mitte) imitiert das Layout des CPMs während V2 (unten) ein Block-Layout hat.

5.3 Zusammenfassung

In diesem Kapitel wurden für die Implementierung wichtige Algorithmen eingeführt. Diese dienen der Bestimmung der Menge der sichtbaren Vorgänger bzw. Nachfolger und des kleinsten gemeinsamen SESE-Fragments bzw. dessen Eingangs- und Ausgangs-Knoten. Außerdem wurde der Algorithmus für $\mathrm{UnaggregateSESE}$ gezeigt, sowie eine Erweiterung des Algorithmus zur Aktualisierung von Sichten bei seriellen Einfügungen. Schließlich wurden wichtige Aspekte des Prototypen vorgestellt.

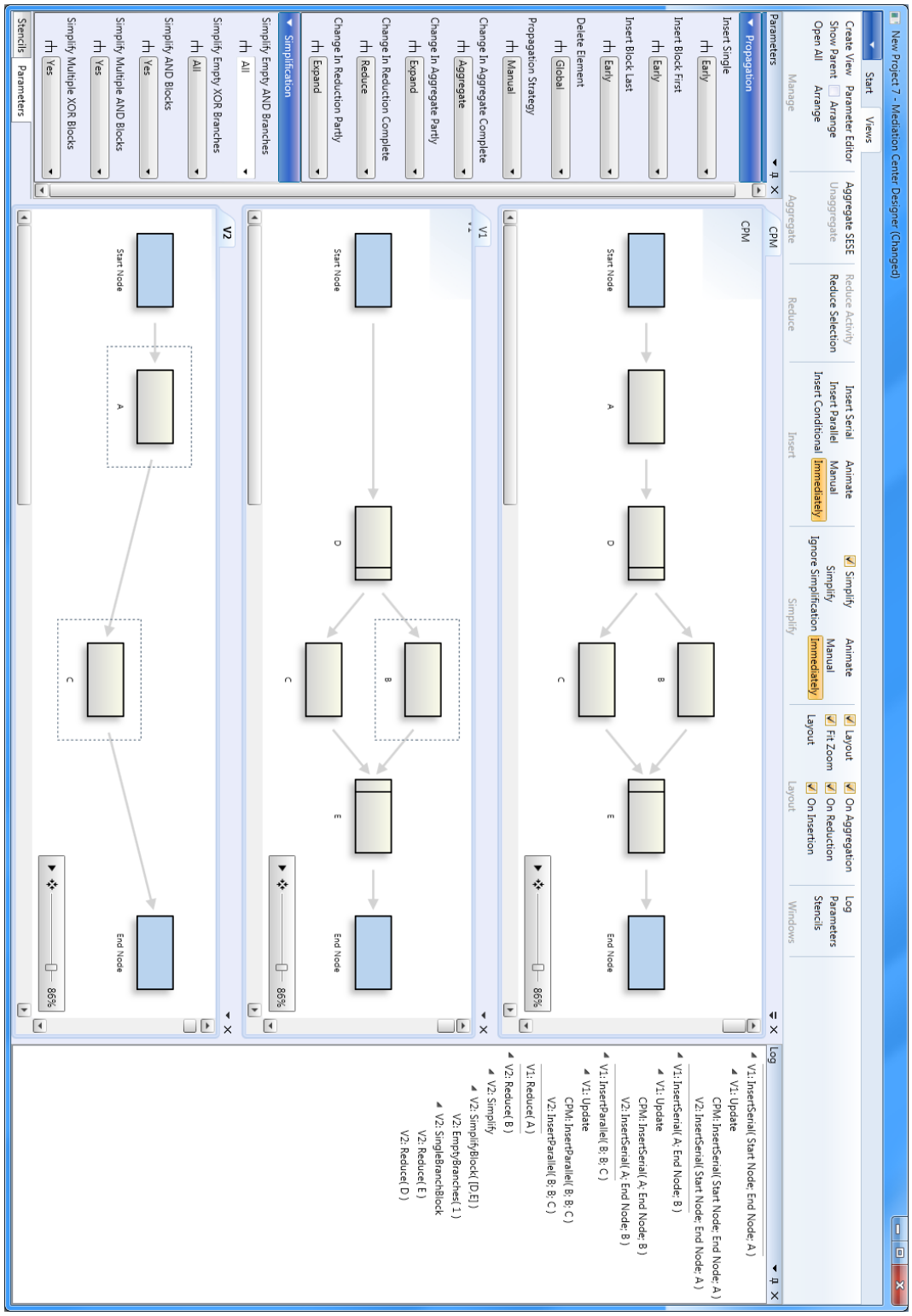

Abbildung 5.4: Das Hauptfenster des Prototyps. Zu sehen sind das CPM und zwei Sichten zwischen der Parameterliste links und dem Log rechts.

6 Zusammenfassung

In dieser Arbeit wurde im Rahmen des *proView*-Projekts eine Komponente zur Erzeugung und Visualisierung von Prozesssichten sowie Vereinfachungs- und Modifikations-Operationen prototypisch implementiert. Das gewählte Prozessmodell basiert auf wohlstrukturierten, azyklischen Prozessgraphen. Nicht behandelt wurden Konstrukte zur Wiederholung oder Synchronisation von Abläufen sowie der Datenfluss.

Eine Prozesssicht entsteht dabei durch Reduktion und Aggregation von Aktivitäten. Unnötige Gateway-Knoten und mehrfache Kanten zwischen zwei Knoten werden durch Vereinfachungs-Operatoren entfernt. Die Vereinfachungs-Operatoren werden durch Parameter gesteuert.

Modifikationen an einem Prozessschema können über Sichten vorgenommen werden. Realisiert wurden Modifikations-Operatoren zur seriellen, parallelen und konditionalen Einfügung von Aktivitäten. Ist eine Einfüge-Position im zentralen Prozessmodell nicht eindeutig zuordenbar, wird mithilfe von Parametern entschieden, welches die geeignete Position ist. Ebenso legen Parameter für die Aktualisierung weiterer Sichten fest, ob eingefügte Aktivitäten in einem reduzierten oder aggregierten Bereich ebenfalls reduziert bzw. aggregiert werden. Jede Sicht hat dazu ihren eigenen Satz von Parametern, die von einer übergeordneten Sicht oder dem zentralen Prozessmodell geerbt werden können.

Es hat sich herausgestellt, dass die Aktualisierung eines Prozessschemas mit dem gewählten Prozessmodell nicht vollständig automatisierbar ist, wenn der Parameter $\mathrm{InsertSingle}$ den Wert $\mathrm{Parallel}$ hat.

Weil bei der Vereinfachung von Sichten Gateway-Knoten entfernt werden können, gestalten sich die Algorithmen zur Aktualisierung der Sichten bei Einfügungen wesentlich komplizierter und damit fehleranfälliger. Vermutlich würde ein anderer Gesamtablauf zu einfacheren Algorithmen führen. Eine Möglichkeit wäre – nach Auswahl der Einfüge-Position in einer Sicht – eine Aktivität zuerst im zentralen Prozessmodell einzufügen, und dann *alle* Sichten durch Anwendung aller bisherigen Reduktions- und Aggregationsoperationen neu zu erzeugen und abschließend eine Vereinfachung der Sichten durchzuführen. Bei jeder weiteren Einfügung müssten die Sichten ebenfalls neu erzeugt werden.

Literaturverzeichnis

[1] R. Bobrik, M. Reichert, and T. Bauer, “Requirements for the Visualization of System-Spanning Business Processes,” in *In DEXA’05*, 2005, pp. 948–954.

[2] D. Schumm, F. Leymann, and A. Streule, “Process Viewing Patterns,” in *Proceedings of the 14th IEEE International EDOC Conference, EDOC 2010, 2529 October 2010, Vitória, Brazil*. IEEE Computer Society, 2010, pp. 89–98.

[3] “ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control,” *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, vol. 10, no. 2, pp. 93–129, March 1998.

[4] R. Bobrik, M. Reichert, and T. Bauer, “View-Based Process Visualization,” in *5th Int’l Conf. on Business Process Management (BPM’07)*. Springer, September 2007, pp. 88–95.

[5] R. Bobrik, “Konfigurierbare Visualisierung komplexer Prozessmodelle,” Dissertation, Universität Ulm, 2008.

[6] R. Pryss, J. Tiedeken, U. Kreher, and M. Reichert, “Towards Flexible Process Support on Mobile Devices,” in *Proc. CAiSE’10 Forum - Information Systems Evolution*. Springer, 2010, pp. 150–165.

[7] M. Reichert, “Dynamische Ablaufänderungen in Workflow-Management-Systemen,” Dissertation, Universität Ulm, 2000.

[8] M. Reichert, J. Kolb, R. Bobrik, and T. Bauer, “Enabling Personalized Visualization of Large Business Processes through Parameterizable Views,” in *27th ACM Symposium On Applied Computing (SAC’12), 9th Enterprise Engineering Track*. ACM Press, 2012.

[9] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, “Workflow Patterns,” *Distributed and Parallel Databases*, vol. 14, pp. 5–51, 2003.

Name: Manuel Ihlenfeld Matrikelnummer: 529951

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis vom 16.10.2009 beachtet habe.

Ulm, den .....

Manuel Ihlenfeld