Effiziente Überprüfung semantischer Korrektheit in adaptiven Prozess-Management-Systemen

Universität Ulm
Fakultät für Informatik und Ingenieurwissenschaften
Abteilung Datenbanken und Informationssysteme (DBIS)

Diplomarbeit

Vorgelegt von

ZHOU, Hao

1. Gutachter: Prof. Dr. P. Dadam
2. Gutachter: Dr. S. Rinderle

2006

Kurzfassung.

Es ist bekannt, dass die Sicherung der syntaktischen Korrektheit ein wichtiger Meilenstein der Entwicklung von adaptiven Prozess-Management-System (PMS) ist. Aber Sicherung der syntaktischen Korrektheit allein ist nicht genug. Ebenfalls ist die Sicherung der semantischen Korrektheit wichtig für das adaptive Prozess-Management-System. Semantische Constraints lassen sich nicht direkt in dem Prozess-Metamodell repräsentieren. Aber ebenfalls kann die semantische Korrektheit durch die Änderungsoperationen zerstört werden. Bevor man sie wieder erstellen kann, muss die semantischen Konflikte zuerst identifiziert werden. Bei Schema-Modellierung und -Verifikation muss dann neben die syntaktische Korrektheit die semantische Korrektheit auch sichergestellt. Was wir in dieser Diplomarbeit behandeln ist genau die Überprüfung von semantischer Korrektheit im adaptiven Prozess-Management-System, vorausgesetzt, dass die syntaktische Korrektheit immer gewährleistet ist. Wir werden die Semantische Constraints kennen lernen und die Szenarien für die semantische Überprüfung analysieren. Und danach werden wir implementierungsnahen Algorithmen für die semantische Überprüfung entwickeln. Dabei wird der Effizienzaspekt berücksichtigt. Da die Redundanz in den grundlegenden Algorithmen bei der Überprüfung großer Menge von Prozessen sehr viel kosten kann, werden sie möglicherweise vermieden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Wegen der Globalisierung und der immer schnelleren Entwicklung der Technik ist der Markt heutzutage viel flexibler als früher. Die Fähigkeit, auf die Änderung in dem Markt schnell reagieren zu können, gilt als die Kernkompetenz eines Unternehmens. Als häufig verwendete Unternehmenssoftware müssen Prozess-Management-Systeme(PMS) entsprechend auch anpassungsfähig sein. Bei nicht-adaptiven PMS sind alle zur Laufzeit möglichen Ausführungsvarianten bereits zur Modellierungszeit bestimmt [4]. Eine spätere Änderung ist sehr schwer, wenn überhaupt möglich, was nicht wünschenswert ist. Deswegen stellen adaptive PMS einen wesentlichen Fortschritt dar. Damit ist man nun jedoch mit neuen Anforderungen konfrontiert, nämlich die konsequente Gewährleistung von sowohl syntaktischer als auch semantischer Korrektheit in adaptiven PMS. Die Überprüfung und Sicherung von syntaktischer Korrektheit sind schon in anderen Arbeiten gelöst werden. In dieser Arbeit geht es um die Überprüfung von semantischer Korrektheit.

1.1 Adaptive Prozess-Management-Systeme

Adaptive Prozess-Management-Systeme sind eine neue mächtige Variante von Prozess-Management-Systemen, wobei Änderungen zur Laufzeit sowohl auf Instanz als auch auf Schemaebene vollständig unterstützt werden sollen. Das System muss in der Lage sein, die syntaktische und semantische Korrektheit selbstständig zu überprüfen, um inkonsistente Zustände zu vermeiden. Folgende sind die zulässigen Änderungen in einem adaptiven PMS.

- Änderungen an Instanzen
  In adaptive Prozess-Management-Systeme soll ermöglicht sein, die laufende Instanz zu ändern, damit das System auf unerwartete Ereignisse, die mit dem originalen Prozess nicht gelöst werden können, reagieren kann. Unter verzerrte Instanzen versteht man die laufenden Instanzen, auf denen Ad-hoc-Änderung gemacht wird. Eine echtzeitige Änderung an der Instanz ist im adaptiven PMS erlaubt. Aber eine Ad-hoc-Änderung an der Instanz in dem bereits abgelaufenen Bereich ist syntaktisch falsch und ist deswegen auch verboten.

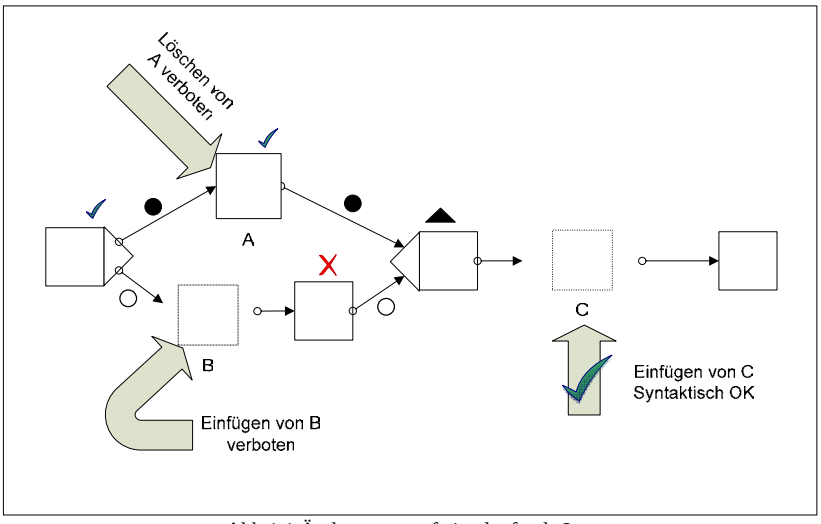


Abb 1.1 Änderungen auf eine laufende Instanz

Wie Abbildung 1.1 zeigt, die Operationen Löschen von A und Einfügen von B verboten sind, weil Sie keine Wirkung haben können. Die Aktivität A wurde schon ausgeführt. Die Aktion kann man nicht durch die Löschoperation zurücksetzen. Die Aktivität B würde sich in dem abgelaufenen Teil befindet, wenn sie eingefügt würde. Sie wird daher nie aktiviert. Deswegen sind solche Operationen syntaktisch falsch und verboten. Die Operation Einfügen von C erzeugt keinen syntaktischen Konflikt, nur die semantische Korrektheit dieser Operation bleibt noch zu überprüfen.

- Schemaevolution und Instanzmigration
  Wenn Die Änderungen des Schemas nennt man auch Schemaevolution. Wenn es laufende Instanzen gibt, die vor den Änderungen auf dem Schema basieren, werden wir versuchen die Schemaänderungen an diesen Instanzen zu übertragen und sie auf dem neuen Schema beruhen zu lassen [4,7]. Diese Anpassung wird auch Instanzmigration genannt.

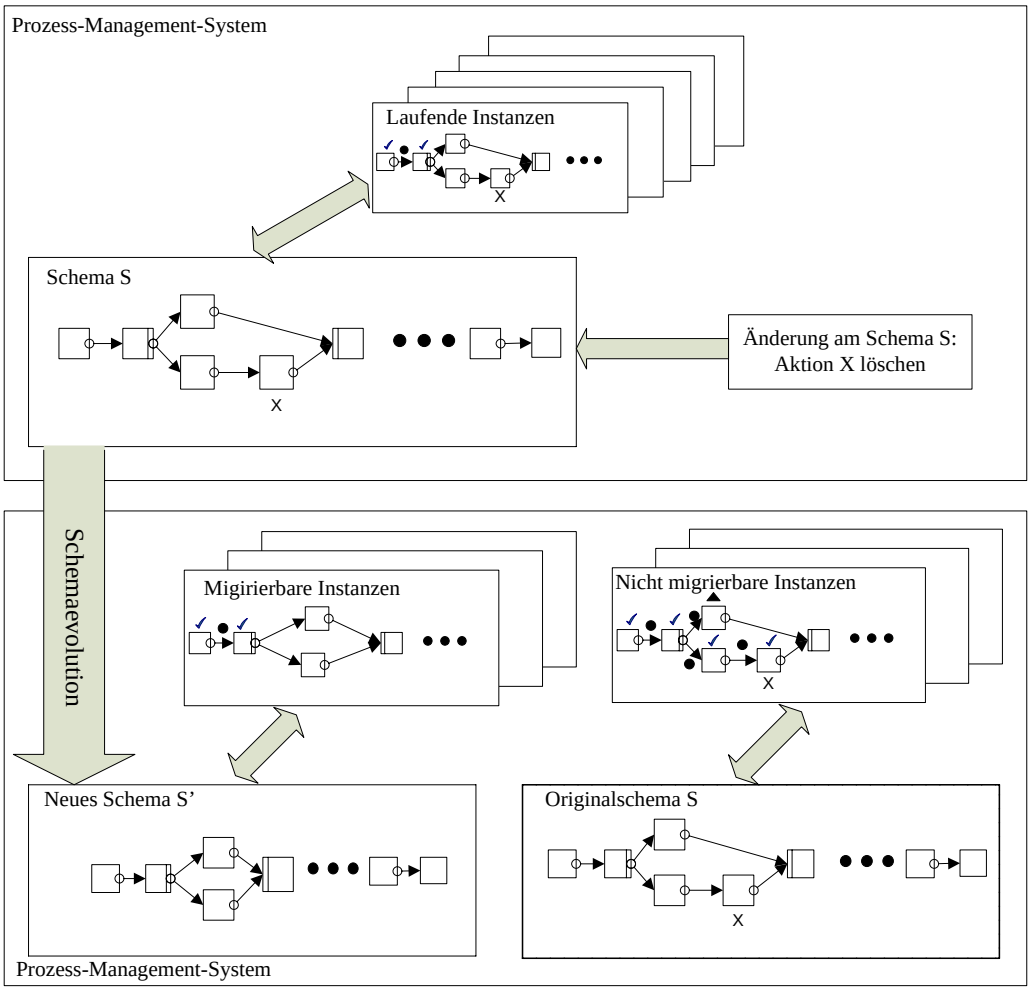


Abb. 1.2 Schemaevolution

Wie die Abbildung 1.2 zeigt, durch das Löschen von X ergibt sich ein neues Schema S', das die Aktivität X nicht mehr enthält. Da unterschiedliche Instanzen unterschiedliche Ausführungszustände haben und auch individuell modifiziert sein können, können u.U. nicht alle Instanzen migriert werden. Die in Abbildung 1.2 gezeigte nicht migrierbare Instanz würde beispielsweise einen syntaktischen Konflikt erzeugen, wenn sie migriert würde. Die in der Instanz schon ausgeführte Aktivität X würde bei der Anpassung an das neue Schema (Migration) gelöscht werden, was syntaktisch falsch und verboten ist. Auch wenn die Instanz mit dem neuen Schema S' syntaktisch verträglich ist, kann sie semantisch nicht verträglich sein. Deswegen muss man die semantische Korrektheit für die Migration überprüfen. Die nicht migrierbaren Instanzen beruhen nach der Schemaevolution nach wie vor auf dem Originalschema S.

Bei den Änderungsszenarien in adaptiven PMS können semantische Konflikte erzeugt werden. Deswegen muss die semantische Korrektheit für sie überprüft werden. Die Operationen, die semantische Konflikte erzeugen, sind semantisch falsch und sollen abgelehnt werden. Bei Schemaevolution wird es anhand der semantischen Überprüfung festgestellt, welche Instanzen semantisch migrierbar und welche nicht sind.

1.2 Semantische Prozessverifikation

Syntaktische Konflikte wie Deadlocks, nicht erreichbare Zustände oder Lesen von nicht geschriebenen Daten lassen sich durch die syntaktische Überprüfung ausschließen. Allerdings beschreiben Prozesse in der Regel Sachverhalte aus der realen Welt. Da gibt es normalerweise viele andere Constraints zu gewährleisten. Beispielsweise kann es ein Lager geben, das man entweder mit Rohstoff_1 oder Rohstoff_2 füllen kann. Die zwei Aktivitäten A und B in Abbildung 1.3 stehen jeweils für die Aktion „das Lager mit Rohstoff1 füllen“ und „das Lager mit Rohstoff2 füllen“. Sie sind offensichtlich semantisch nicht verträglich, weil die nacheinander Ausführung von den beiden Aktionen ohne das Lager zu leeren, in der Tat keinen Sinn macht. D.h. nur eine von den beiden Aktionen kann jeweils ausgeführt werden.

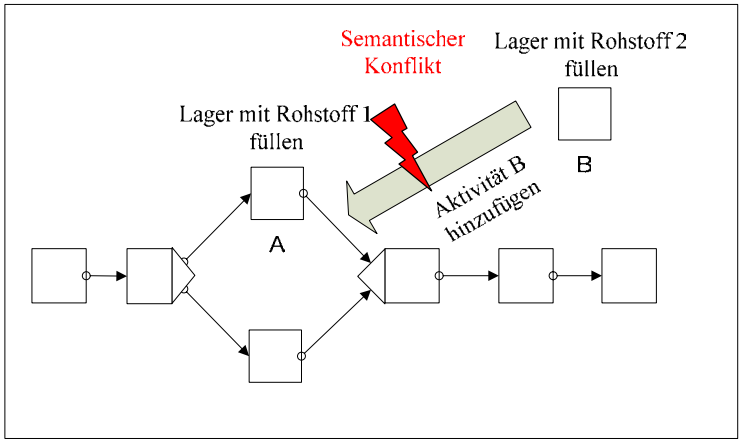


Abb. 1.3 Semantische Konflikt

Die Einfügeoperation soll einen Konflikt auslösen. Aber ein solches semantisches Constraint lässt sich nicht syntaktisch im Prozess formulieren. Ebenfalls kann man die Überprüfung von einem solchen Konflikt nicht die Person, die das System verwendet, überlassen. Weil die realen Geschäftsprozesse in der Regel sehr komplex sind, kann man sich nicht darauf verlassen, dass die Bearbeiter alle semantischen Constraints in Kopf haben und bei allen Operationen immer auf die semantische Korrektheit achten. Die semantische Korrektheit eines Prozesses wird besonders gefährdet, wenn Prozesse von vielen Bearbeitern geändert werden [3]. Man kann sich vorstellen, dass es ein zweiter Bearbeiter, der keiner Designer ist und nichts von den ausgeführten Aktivitäten weiß, eine Aktivität B in den Prozess hinzufügen möchte. Dadurch verursachte Konflikte sind nicht manuell zu vermeiden.

Allerdings können wir das Problem mit der Hinterlegung eines semantischen Constraints lösen. Das Constraint soll dafür stehen, dass die Aktivitäten A und B nie zusammen ausgeführt werden. Wir nennen diese Beziehung die Ausschlussbeziehung. Neben der Ausschlussbeziehung gibt es noch andere semantische Constraints, die die Ausführungen von Aktivitäten auf einem Prozess festlegen können. Genaueres dazu kommt noch später in Kapitel 2.

Wir brauchen entsprechend ein Verifikationsmodul, das die Erfüllung und Einhaltung dieser Constraints, nämlich die semantische Korrektheit des Prozesses, selbstständig überprüft. Nur wenn die semantischen Konflikte durch das Verifikationsmodul vermieden werden, ist das adaptive PMS einsatzbereit.

1.3 Effizienzaspekte

Bei der Überprüfung semantischer Korrektheit ist die Effizienz der grundlegenden Algorithmen und des darauf basierenden Programm unerlässlich. Da die realen Geschäftsprozesse sehr komplex sein können, können die echtzeitigen Änderungsoperationen wegen der semantischen Überprüfung zu viel kosten, was das PMS viel schwächer und unpraktisch macht. Die Situation ist besonders kritisch bei der Schemaevolution. Da können zugleich Tausende von Instanzen in Betrieb sein. Eine Änderung des Originalschemas betrifft sie alle bei der Migration. Die semantische Verträglichkeit muss unter Umstände auch tausende Mal überprüft werden, bevor die Migration zustande kommen kann. Daher müssen die Algorithmen zur semantischen Überprüfung effizient sein.

Um die Effizienz zu verbessern, können wir auch die vorhandenen Informationen ausnutzen. Wenn beispielsweise nur eine Löschoperation auf die Instanz ausgeführt wird, braucht man danach keine Ausschlussbeziehungen zu überprüfen, weil dadurch keine neue Aktivität in den Prozess vorkommt. Es wird daher auch keine semantischen Constraints von Ausschlussbeziehung verletzt (genauer dazu in Kapitel 5). Durch angemessene Verwertung von nützlichen Informationen lassen sich die überflüssige Aktionen stark reduzieren. Deshalb sollen die Informationen zuerst analysiert werden, bevor die Überprüfung stattfindet.

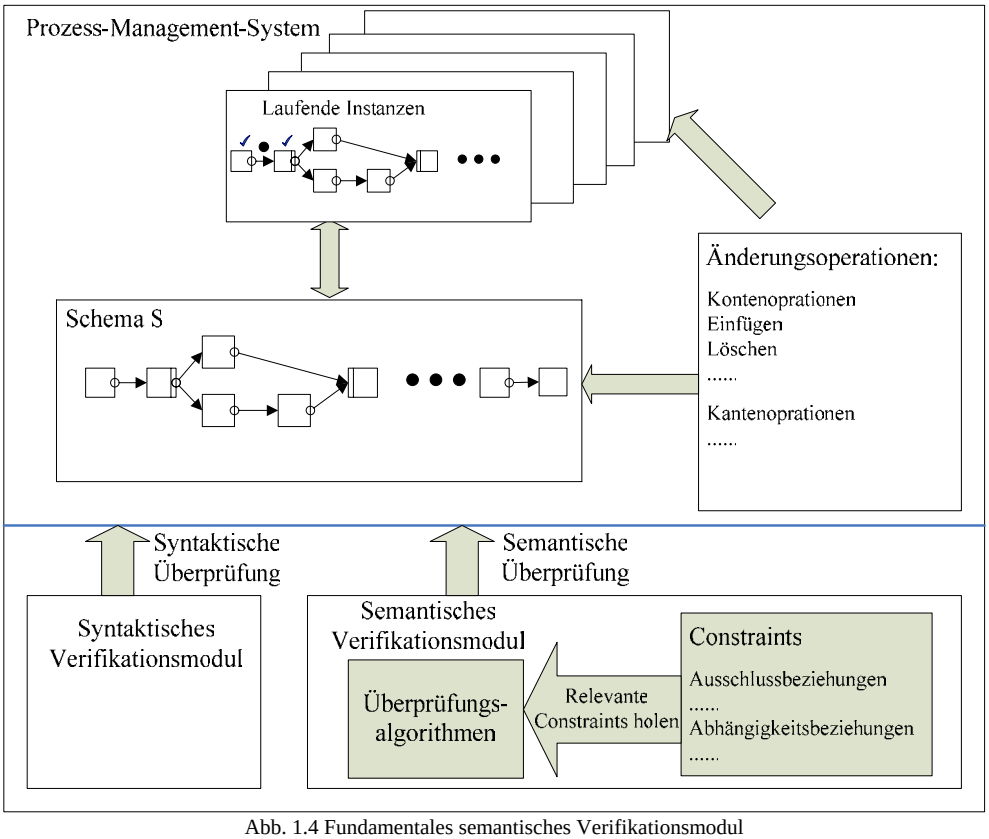


Abb. 1.4 Fundamentales semantisches Verifikationsmodul

In Abbildung 1.4 wird das Konzept der semantischen Überprüfung verdeutlicht. Die Semantischen Constraints werden als Regeln in dem PMS gespeichert. Bei der semantischen Überprüfung werden jeweils die relevanten Constraints in der Regelbasis identifiziert und dann verifiziert. Das adaptive PMS in Abbildung 1.4 hat zwei Verifikationsmodule, das syntaktische und das semantische Verifikationsmodule. Ein Vorteil von dieser Architektur ist, dass die semantische Korrektheit und die syntaktische Korrektheit eines Prozesses immer separat überprüft werden. Damit hat das adaptive PMS eine klare Struktur, was günstig für die Implementierung und spätere Entwicklung des Systems sein kann.

1.4 Aufbau der Arbeit

In diesem Kapitel haben wir die effiziente Überprüfung von semantischer Korrektheit im adaptiven PMS motiviert.

Anschließend werden in Kapitel 2 die notwendigen Grundlagen vorgestellt, wie z.B. das grundlegende Prozess-Metamodell und die Definitionen der semantischen Constraints, damit wir notwendigen Vorkenntnisse haben, um das Problem der semantischen Überprüfung zu verstehen, und eine Lösung dafür entwickeln zu können, geschaffen werden.

In Kapitel 3 werden wir deutlich machen, wann ein Prozess semantisch überprüft werden muss und was wir bei der Überprüfung besonders betrachten müssen. Wir werden die Szenarien für die semantische Überprüfung nach der Schwierigkeit stufen, damit verdeutlicht wird, mit welchen Problemen wir konfrontiert sind. Dabei wird jeweils versucht, eine naive Vorgehensweise vorzustellen. Zugleich werden wir an den Optimierungsmöglichkeiten für die naive Vorgehensweise denken.

Anhand der Analyse in Kapitel 3 werden wir in Kapitel 4 die Algorithmen für die semantische Überprüfung, schrittweise herleiten. Mit den Algorithmen werden wir alle semantischen Constraints mit einem Durchlauf des Prozesses überprüfen. Sie werden optimiert und an alle Überprüfungsszenarien angepasst.

In Kapitel 5 werden die semantischen Wirkungen der Änderungsoperationen zur Effizienz der Überprüfung analysiert. Anhand der vorhandenen Informationen, wie z.B. der Operationstyp und die betroffene Aktivitäten, wird versucht, die zu überprüfenden semantischen Constraints einzuschränken.

Zum Schluss in Kapitel 6 werden wir eine Zusammenfassung machen. Im Ausblick werden wir uns mit neuen Anforderungen bei der semantischen Überprüfung befassen. Dabei wird auch ermittelt, welche entsprechenden Erweiterungsmöglichkeiten die Algorithmen, die wir in dieser Arbeit entwickelt haben, besitzen.

Kapitel 2

Grundlagen

In diesem Kapitel werden wir die für diese Arbeit relevanten Grundlagen vorstellen. Zuerst lernen wir das Prozess-Metamodell kennen, weil es als Basis zur Modellierung des Prozesses dient. Alle Prozess-Schemata, Instanzen und Änderungsoperationen beruhen darauf. Um das Konzept entwerfen oder verstehen zu können, muss man zuerst die Eigenschaften des Prozess-Metamodells kennen lernen. Danach werden die vorhandenen, derzeit schon klar definierten semantischen Constraints einbezogen. Dabei werden wir deutlich machen, was semantische Korrektheit bedeutet. Anschließend werden wir die für die Überprüfungen relevanten Operationen im adaptiven PMS vorstellen. Zum Schluss des Kapitels wird noch die Delta-Schicht, die zur Effizienz der Durchführung von Änderungsoperationen eingesetzt wird, beleuchtet.

2.1 Verwendetes Prozess-Metamodell

In dieser Arbeit verwenden wir ADEPT als Prozess-Metamodell, alle Instanzen und Schemata werden in ADEPT formuliert. ADEPT gehört zu blockstrukturierten Metamodellen [11]. Die Kontrollstrukturen (Sequenz, Verzweigung, Schleife, ...) werden auf Blöcke mit eindeutiger Start-/Endaktivität abgebildet.

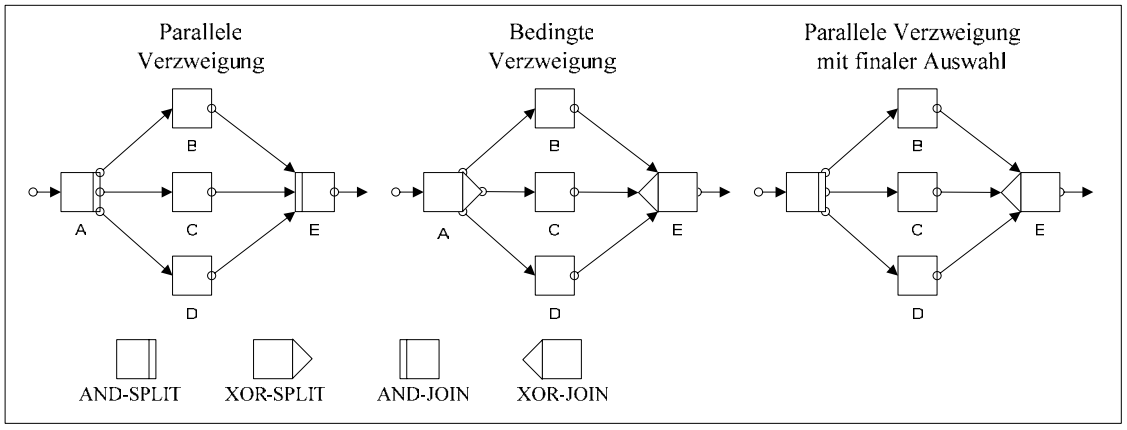


Abb. 2.1 Verzweigungen in ADEPT

Zu den Verzweigungen gehören: AND-Split / AND-Join; AND-Split / XOR-Join; XOR-Split / XOR-Join. Die Strukturen zeigt die Abb. 2.1. Die Kombination AND-Split / OR-Join ist zwar in ADEPT erlaubt, aber das Konzept ist nicht ausgereift. Deswegen wird sie in dieser Arbeit nicht berücksichtigt.

Die Abbildung 2.2 zeigt ein Workflow-Modell in ADEPT. Hier kann man deutlich sehen, dass die Blockstrukturen sich nicht überschneiden dürfen, aber beliebig ineinander verschachtelt sein können. Die blockstrukturierte Workflow-Modellierung hat die Vorteile im Vergleich zu unstrukturierten Modellen (z.B. Petri-Netzen): Sie unterstützt Benutzer besser bei der Erstellung, Analyse und Änderung von Workflow-Modellen. Sie ist übersichtlicher und weniger Konfliktträchtig.

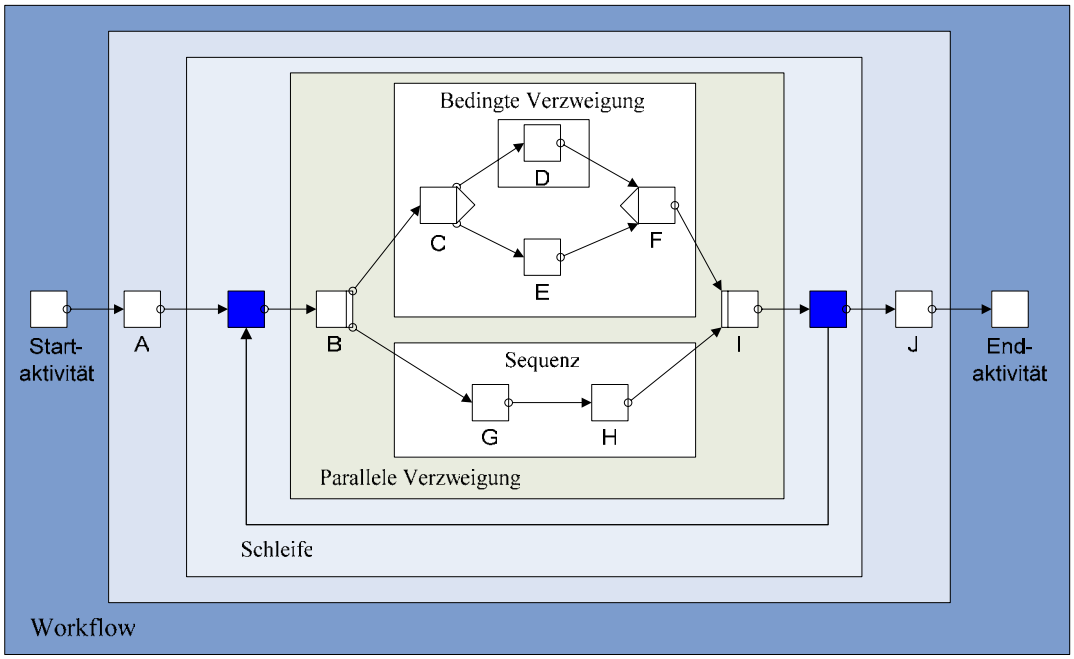


Abb. 2.2 Ein ADEPT-Prozess [11]

Wie in Abbildung 2.2 gezeigt, sind Schleifen in ADEPT erlaubt. Noch eine Sache lässt sich nicht übersehen: die Synchronisationskante(Sync-Kante). Sie wird verwendet, um die Ausführung von Knoten paralleler Teilzweige zu synchronisieren. Damit kann man die Reihenfolge der Ausführungen von Aktivitäten auf unterschiedlichen Zweigen eines Blocks bestimmen.

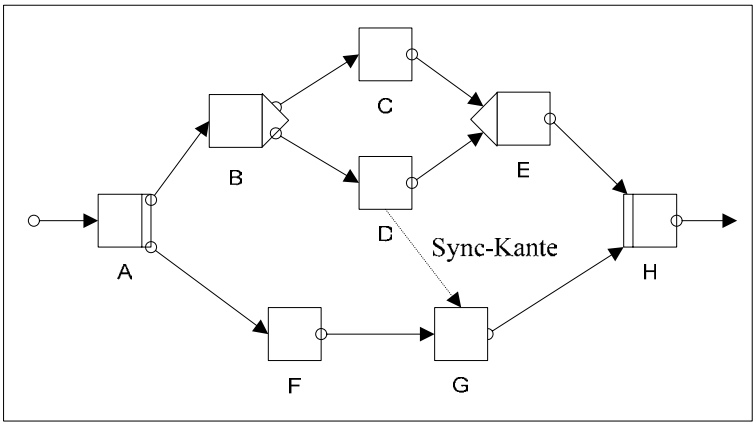


Abb. 2.3 Synchronisationskante

Die in Abbildung 2.3 gezeigte Sync-Kante (D, G) beschreibt, dass G frühestens dann aktiviert bzw. ausgeführt werden kann, wenn D entweder zuvor erfolgreich beendet wurde oder wenn feststeht, dass D nicht mehr zur Ausführung kommt.

In ADEPT gibt es ebenfalls Formulierungsmittel für die Variablen und Datenfluss zwischen Aktivitäten. Sie sind aber für die Überprüfung semantischer Korrektheit momentan unwichtig, werden deshalb hier nicht vorgestellt.

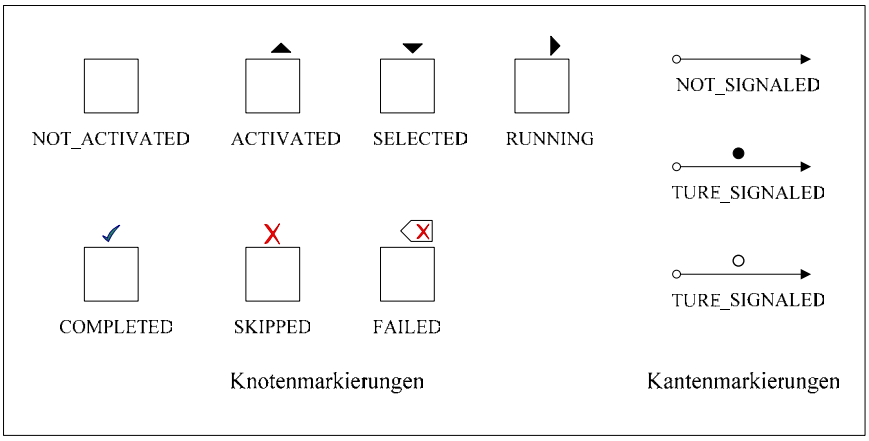


Abb. 2.2 Markierungen in ADEPT

In Abbildung 2.4 werden alle Markierungen dargestellt, die zur Markierung des Ausführungszustandes von Knoten und Kanten von Instanzen verwendet werden. Eine ausgeführte Aktivität kann im Laufe der Ausführung mit NOT_ACTIVATED, ACTIVATED, RUNNING und COMPLETED markiert werden. Die Aktivitäten, die von Benutzer ausgewählt werden, werden mit SELECTED markiert. NOT_ACTIVATED bedeutet, dass eine Aktivität noch nicht ausgeführt wird. Wenn eine Aktivität mit ACTIVATED markiert ist, kann sie ausgeführt werden. Mit RUNNING markierte Aktivitäten werden gerade ausgeführt. Nach der erfolgreichen Ausführung wird eine Aktivität mit COMPLETED markiert. Mit SKIPPED markierte Aktivitäten werden bei der Ausführung der Instanz nicht mehr ausgeführt. Genauere Erklärung dafür gibt es in [11]. Für semantische Überprüfung sind die Markierungen NOT_ACTIVATED, COMPLETED und SKIPPED wichtig.

2.2 Semantische Constraints und semantische Korrektheit

Die Semantik im adaptiven PMS ist ein neues Forschungsgebiet. Wir können die Constraints in zwei Gruppen teilen: die Abhängigkeitsbeziehung und die Ausschlussbeziehung. Jedes Constraint wird als ein Tupel *(Type, Source, Target, Position)* repräsentiert. Parameter *Type* kann entweder den Wert *dependsOn* oder *excludes* annehmen, wobei *dependsOn* für die Abhängigkeitsbeziehung und *excludes* für die Ausschlussbeziehung steht. *Source* und *Target* stehen für die zwei Aktivitäten, die ein Constraint betrifft. *Position* bestimmt die Reihenfolgebeziehung von den *Source*- und *Target*-Aktivitäten. Wir haben drei Werte für die *Position*: *pre*, *post* und *notSpecified*. *Pre* bedeutet, dass die *Target*-Aktivität vor der *Source*-Aktivität ist; post bedeutet, dass die *Target*-Aktivität nach der *Source*-Aktivität ist. Und *notSpecified* besagt nur, dass für das Constraint die Reihenfolge von den *Source* und *Target*-Aktivitäten irrelevant ist [1,3]. Beispielsweise beschreibt das Constraint *(excludes, A, B, pre)* die Semantik, dass nach der Ausführung von der Aktivität A in einer Instanz die Aktivität B nicht mehr zur Ausführung kommen darf. Mit zwei Typen und drei Positionen haben wir insgesamt sechs Arten von semantischen Constraints.

Für ein Schema oder eine Instanz kann eine Menge von Constraints hinterlegt werden. Wenn alle diese Constraints auf dem Schema oder auf der Instanz erfüllt sind, ist das Schema oder die Instanz semantisch korrekt. Deswegen überprüfen wir, ob alle Constraints eines Prozesses erfüllt sind oder nicht, um die semantische Korrektheit des Prozesses festzustellen.

Wir werden jetzt für jedes Constraint klar machen, genau wann es auf einem Prozess erfüllt ist.

1. **(dependsOn, A, B, post)**
   Das Constraint *(dependsOn, A, B, post)* ist erfüllt über einen Prozess P, wenn es keine Ausführungsreihenfolge von P gibt, in der A ausgeführt wird und B nicht nach A ausgeführt wird. D.h. in jeder möglichen Ausführungsreihenfolge von P wird die Aktivität B immer ausgeführt, nachdem A ausgeführt wurde.

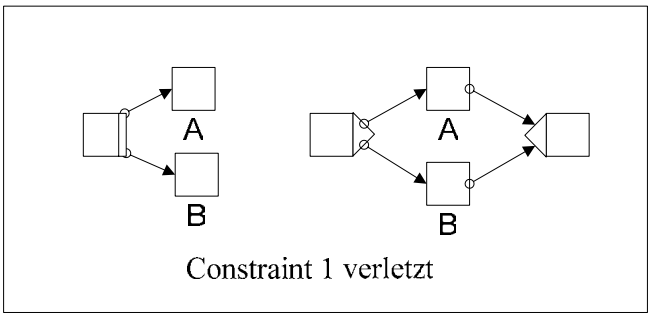


Abb. 2.5 Nicht erfülltes Constraint 1

Im Abbildung 2.5 werden zwei Szenarien dargestellt, in denen das Constraint *(dependsOn, A, B, post)* verletzt ist. Wenn A und B sich auf zwei unterschiedlichen Zweigen eines UND-Blocks befinden, werden sie zwar immer beide ausgeführt, aber die Reihenfolgebeziehung A-B ist nicht gewährleistet. B kann nämlich vor A ausgeführt werden. Wenn B nicht immer nach A ausgeführt wird, ist das Constraint verletzt. Durch die Abbildung 2.6. wird dies nochmal verdeutlicht.

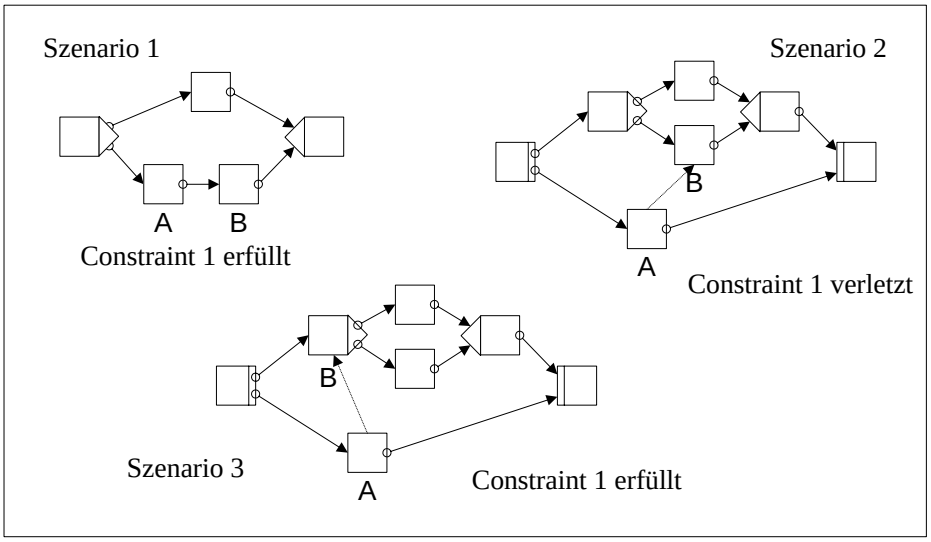


Abb. 2.6 XOR-Block und Constraint 1

2. (dependsOn, A, B, pre)

Analog wie beim ersten Constraints: Das Constraint *(dependsOn, A, B, pre)* ist erfüllt über einen Prozessvorlage P, wenn es keine Ausführungsreihenfolge von P gibt, in der A ausgeführt wird und B nicht vor A ausgeführt wird. D.h. die Aktivität A kann nur nach der Ausführung von B ausgeführt werden. Dabei ist nämlich die Reihenfolge B-A zu sichern.

3. (dependsOn, A, B, notSpecified)

Wie gesagt, *notSpecified* als *Position* bedeutet, dass die Ausführungsreihenfolge von den *Source*- und *Target*-Aktivitäten für das Constraint irrelevant ist. Deswegen ist das Constraint *(dependsOn, A, B, notSpecified)* über einen Prozess P erfüllt, wenn es keine Ausführungsreihenfolge von P gibt, in der A vorkommt und B nicht vorkommt.

Der Unterschied zwischen Constraint 3 und Constraint 1 und 2 liegt darin, dass neben der Reihenfolgebeziehung A-B und B-A auch A parallel zu B sein kann. Es wird nur, wie in Abbildung 2.7 gezeigt, die Konstellation von A und B ausgeschlossen, wenn die beiden sich auf zwei unterschiedlichen Zweigen eines XOR-Blocks befinden.

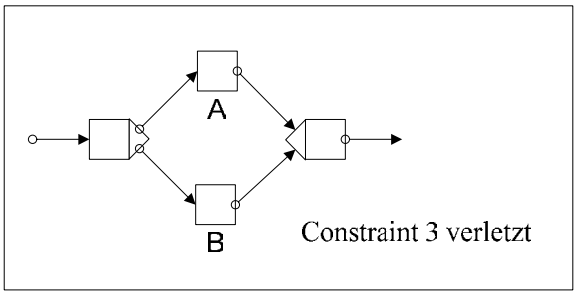


Abb. 2.7 XOR-Split und nicht erfülltes Constraint 3

4. **(excludes, A, B, post)**
   Das Stichwort *excludes* steht für die Ausschlussbeziehung. Das Constraint *(excludes, A, B, post)* ist erfüllt über einen Prozess P, wenn es keine Ausführungsreihenfolge von P gibt, in der B nach A vorkommt. D.h. B darf nicht nach A ausgeführt werden.

   Für die Ausschlussbeziehung ist das Constraint bereits nicht erfüllt, wenn es auch nur die Möglichkeit gibt, dass die ungewünschte Ausführungsreihenfolge vorkommen kann. In Abbildung 2.8 wird B nicht immer nach A ausgeführt, weil B sich in einem XOR-Block befindet und der XOR-Zweig von B zur Laufzeit nicht immer genommen wird. Trotzdem ist das Constraint 4 dabei verletzt. D.h. es muss gesichert werden, dass die ungewünschte Ausführungsreihenfolge nie vorkommt.

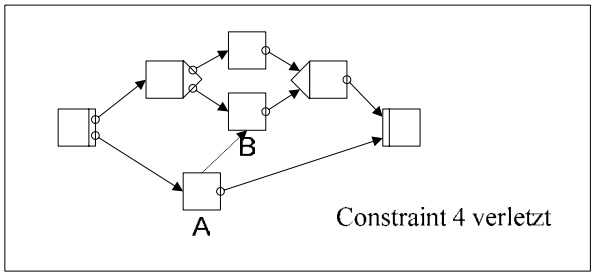


Abb. 2.8 Nicht erfüllende A-B Folgebeziehung

5. **(excludes, A, B, pre)**
   Das Constraint *(excludes, A, B, pre)* ist erfüllt über einen Prozess P, wenn es keine Ausführungsreihenfolge von P gibt, in der B vor A vorkommt. D.h. B darf nicht vor A ausgeführt werden.

6. **(excludes, A, B, notSpecified)**
   Mit dem Constraint 6 wird es einfach verboten, dass A und B beide ausgeführt werden, egal wie die Ausführungsreihenfolge ist. D.h. ist das Constraint *(excludes, A, B, notSpecified)* erfüllt über einen Prozess P, wenn es keine Ausführungsreihenfolge von P gibt, in der sowohl B als auch A vorkommt.

Formale Definitionen zu den semantischen Constraints sind in [13] zu finden.

2.3 Änderungsoperationen

Die Änderungsoperationen wurden bereits in der Diplomarbeit von Martin Jurisch [8] definiert. In dieser Arbeit werden diese Definitionen als Standard verwendet. Nach der Definition gibt es einfache und komplexe Änderungsoperationen in unterschiedlichen semantischen Ebenen.

2.3.1 Änderungsprimitiven und semantisch höhere Operationen

Die Änderungsprimitiven sind das Fundament aller Änderungsoperationen. Sie sind die Operationen, die die Mengen der Knoten, Kanten und Datenelemente eines Prozesses direkt manipulieren, wie z.B. das Einfügen eines Knotens [8]. Wegen der syntaktischen Sicherheit wird es verboten, dass die

Änderungsprimitiven direkt von den Benutzern verwendet werden. Den gewünschten Änderungseffekt muss man durch die darauf bauenden semantisch höheren Operationen erzielen.

Die semantisch höheren Operationen werden in der Literatur [8] definiert. Diese einfachen und komplexen Änderungsoperationen bauen auf den Änderungsprimitiven auf und können unmittelbar vom Benutzer aufgerufen werden. Für die syntaktische Korrektheit des Graphen sorgen dabei die Änderungsoperationen selbst.

2.3.2 Einfache vs. komplexe Änderungsoperationen

Je nach der Zusammensetzung der jeweiligen Operationen werden die semantisch höheren Operationen in zwei Gruppen geteilt: die einfachen und komplexen Änderungsoperationen. Wenn eine Operation lediglich auf Änderungsprimitiven aufbaut, gehört sie zur Gruppe der einfachen Änderungsoperationen. Wenn sie sich dagegen aus einfachen oder wiederum aus komplexen Änderungsoperationen zusammensetzt, ist sie eine komplexe Änderungsoperation. Im Anhang A der Literatur [8] befinden sich Tabellen von allen einfachen und komplexen Änderungsoperationen mit Erklärungen. Die Ausführbarkeit einer komplexen Änderungsoperation ergibt sich direkt aus der Ausführbarkeit derjenigen Operationen, aus denen diese zusammengesetzt ist. Ebenfalls ergibt sich die semantische Korrektheit einer komplexen Änderungsoperation aus der semantischen Korrektheit der Komponentenoperation. Deswegen werden wir uns in dieser Arbeit auf die semantische Überprüfung von einfachen Operationen konzentrieren, vor allem die Operation: *insertNode*, *deleteNode* und *moveNodes*.

- *insertNode(newNode, pred, succ)*: Mit der Operation insertNode kann man eine neue Aktivität seriell zwischen zwei direkt aufeinander folgenden Aktivitäten einfügen. Die Parameter *pred* und *succ* können auch Verzweigungs- und Vereinigungsknoten sein. Dabei muss ein leerer Teilzweig vorhanden sein, um die Operation syntaktisch zu ermöglichen.

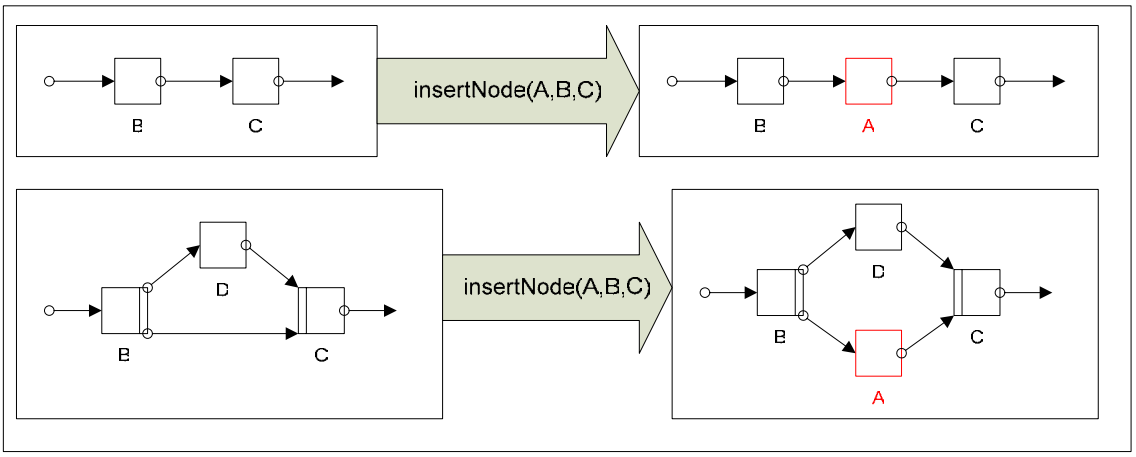


Abb. 2.9 Operation *insertNode*

- *deleteNode(node)*: Mit der Operation kann man eine Aktivität aus dem Prozess löschen. Dabei werden alle in dieser Aktivität eingehenden und von ihr ausgehenden (Sonder-)Kanten mitgelöscht.

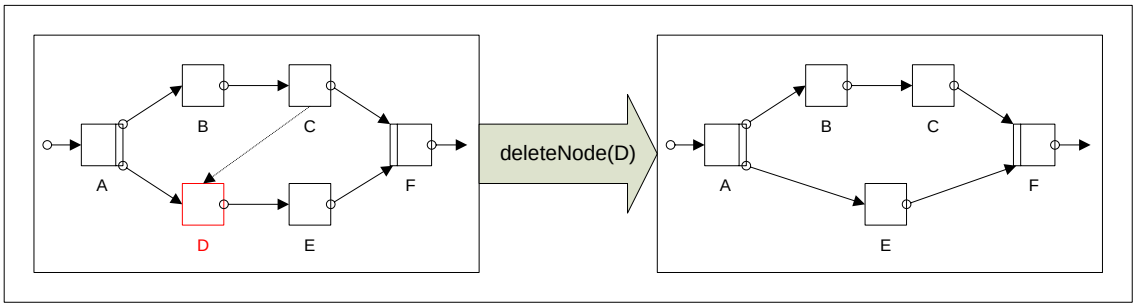


Abb. 2.10 Operation *deleteNode*

Für die semantische Überprüfung ist es lediglich wichtig, ob eine Sync-Kante mitgelöscht wird. Dagegen sind die eventuell mitgelöschten Datenflusskanten für die semantische Korrektheit des Prozesses irrelevant.

- *moveNodes(first, last, pred, succ)*: Durch die Operation werden die Aktivitäten, die sich zwischen *first* und *last*(inkl.) befinden, zwischen *pred* und *succ* neu positioniert.

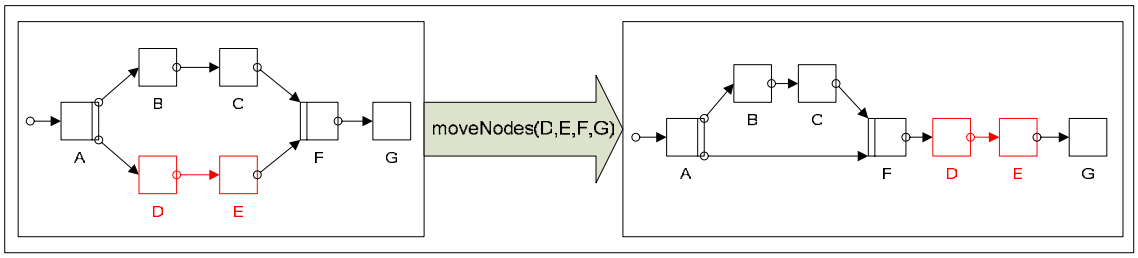


Abb. 2.11 Operation *moveNodes*

Es gibt noch weitere einfache Änderungsoperationen, wie z.B. *createSyncEdge*, *insertEmptyBranch* usw. Die semantischen Wirkungen von diesen Operationen werden wir noch in Kapitel 5 genauer ermitteln.

2.4 Die Delta-Schicht

Um die Adaptivität zu ermöglichen werden in adaptiven PMS viele Techniken eingesetzt, die in einem nicht-adaptiven PMS nicht erforderlich sind. Eine davon wird speziell zur Speicherung der Modifikation angewendet. Für die Überprüfung semantischer Korrektheit muss man sie zuerst genau kennen lernen: die Delta-Schicht.

Anders als Änderungen auf einem Schema werden die Ad-hoc-Änderungen nicht direkt auf die Instanzen materialisiert. Weil die verzerrten Instanzen immer auf das Originalschema beruhen, würde hier die direkte Materialisierung der Änderung ein neues Problem hervorrufen, dass die Instanzen nicht mehr auf dem Originalschema beruhen können, weil sie strukturell nicht identisch sind. Es ist möglich, eine Kopie von dem Schema zu machen und die Instanzänderungen auch darauf zu materialisieren und dann die verzerrte Instanz auf dieses neue Schema umzubiegen. Aber das ist sehr aufwendig und die ursprüngliche Prozesstyp–Zugehörigkeit geht dadurch verloren: Die verzerrten Instanzen beruhen nicht mehr auf dem ursprünglichen Schema. Ohne zusätzliche Vormerkungen werden diese Instanzen bei einer späteren Schemaevolution nicht mehr berücksichtigt [2].

Stattdessen gibt es hier eine elegantere Lösung, eine Zwischensicht zur Speicherung der Änderungen einzusetzen, die so genannte Delta-Schicht. Wie der Name sagt, in der Delta-Schicht wird nicht die verzerrte Instanz sondern nur Ausschnitte gespeichert, die von den Änderungsoperationen modifiziert werden.

Die Abbildung 2.12 verdeutlicht das Konzept der Delta-Schicht. Es wurde eine Änderungsoperation *insertNode(A23, A2, A3)* auf der Instanz I3 ausgeführt. Hier wird es in der Delta-Schicht von I3 beispielsweise nur die A2, A3 und die neu eingefügte A23 gespeichert. A2 und A3 braucht man um die Position der neuen Aktivität A2 in I3 zu bestimmen. Bei den Löschoperation und Verschiebeoperation wird es ebenfalls die zwei Aktivitäten vor und nach dem modifizierten Teil in der Delta-Schicht zur Positionierung gespeichert.

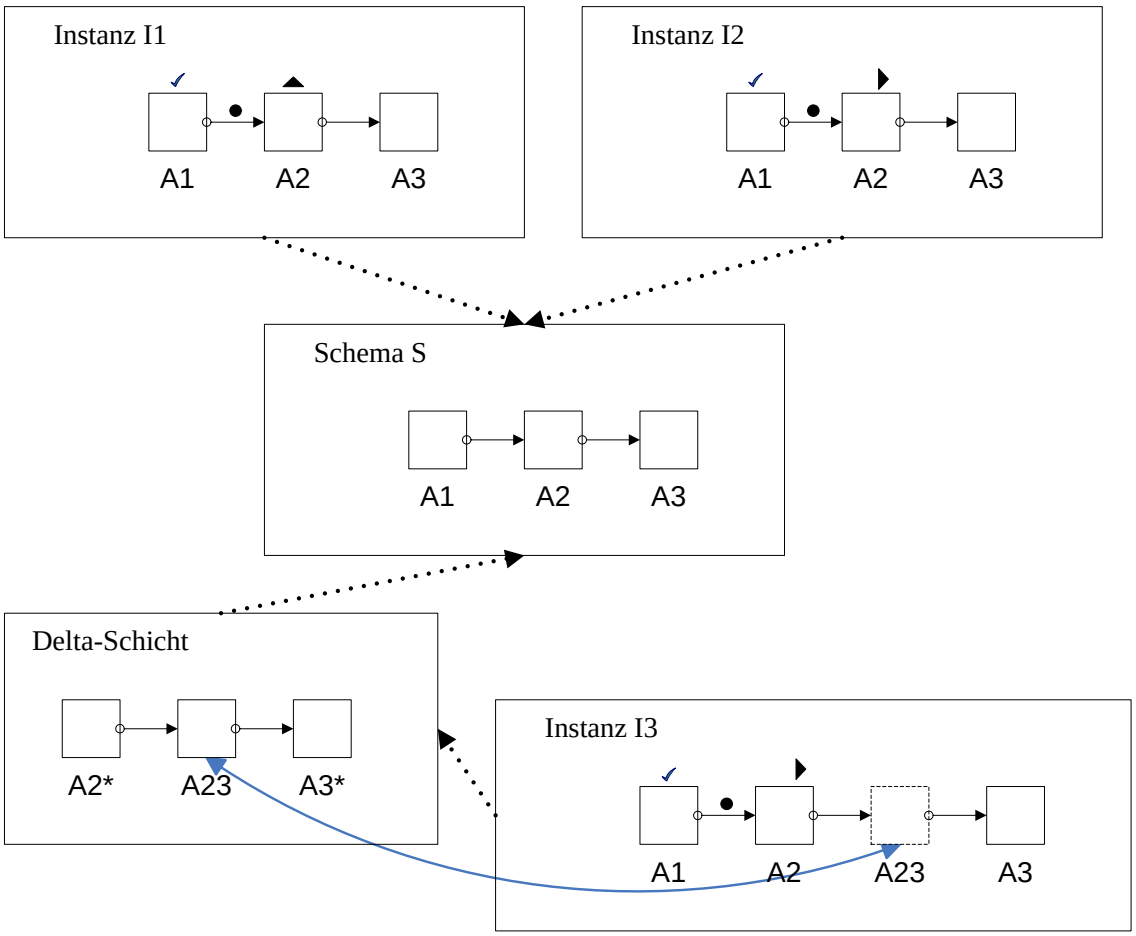


Abb. 2.12 Konzept der Delta-Schicht

Die logische Sichte der unverzerrten Instanzen I1 und I3, lassen sich durch die Ausführungszustände und das Schema repräsentieren. Die logische Schicht von I3 setzt sich dagegen aus dem Ausführungszustand, dem Originalschema und der Delta-Schicht zusammen.

Wie in Abbildung 2.12 angedeutet, beruhen die unverzerrten Instanzen I1 und I2 direkt auf das Schema S. Die verzerrte Instanz I3 basiert dagegen auf die Delta-Schicht und die Delta-Schicht setzen sich am Schema S. Dadurch wird die ursprüngliche Prozesstyp–Zugehörigkeit beibehalten. Die Instanz I3 gilt immer als von Typ S. Für die Frage zur Laufzeit, „Gib mir alle direkten Nachfolgeaktivitäten von Aktivität A!“, wird es bei modifizierter Instanz immer zuerst in der Delta-Schicht nach geeigneten Aktivitäten gesucht. Hier wird es beispielsweise als direkten Nachfolgeaktivitäten von A1 in I3 nicht die A2 sondern die A2* in der Delta-Schicht zurückgeliefert. Nach der Ausführung von A3* in der Delta-Schicht geht es zurück zu dem Schema S. So wird die Modifikation auch tatsächlich realisiert.

Die Beziehung zwischen Delta-Schicht und Instanz ist nicht unbedingt eins zu eins. Instanzen vom selben Typ mit den gleichen modifizierten Teilen können sich eine Delta-Schicht teilen. Umgekehrt kann eine Instanz auch mehrere Delta-Schichten besitzen. Zusammen mit dem Originalschema bilden sie das Laufzeitschema für diese Instanz [10]. Das werden wir hier werde mehr tiefer eingehen. Für die Überprüfung semantischer Korrektheit reicht schon das Verständnis über das grundlegende Konzept der Delta-Schicht.

Kapitel 3

Änderungsszenarien

Es gibt mehrere Situationen im adaptiven PMS, in denen die semantische Korrektheit überprüft werden muss. Wir werden in diesem Kapitel diese Szenarien klassifizieren und separat detailliert betrachten. Die Besonderheiten bei jedem Szenario müssen grundsätzlich analysiert werden. Die Vorteile und Nachteile von verschiedener Vorgehensweisen der Überprüfung werden so aufgelistet, dass später das beste Konzept ausgewählt werden kann und Algorithmen entwickelt werden können.

Nach der Schwierigkeit der Überprüfung lassen sich die Szenarien wie folgt bestufen.

- Verifikation eines ganzen Schemas
  Im adaptiven PMS muss es möglich sein, neues Schema zu importieren. Bevor man das neue Schema in einen adaptiven PMS richtig einsetzen kann, muss es zuerst verifiziert werden. D.h. es muss sichergestellt werden, dass das Schema mit den Constraints, die spezifisch für das Schema sind, verträglich ist.

- Modellierung / Änderung des Prozess-Schemas
  Es werden einzelne Änderungsoperationen einer nach dem anderen auf einem (Teil-)Schema ausgeführt. Die Aufgabe hier ist die Semantische Korrektheit des neuen Schemas zu überprüfen und zu sichern.

- Ad-hoc-Änderung an Instanzen
  Die Ermöglichung echtzeitiger Änderung gilt als die Kernkompetenz des adaptiven PMS. Aber eine anwendbare Ad-hoc-Änderung muss zuerst semantisch Konfliktfrei sein. Für die Überprüfung von semantischer Korrektheit einer Ad-hoc-Änderung ist der Ausführungszustand der aktuellen Instanz relevant.

- Migration
  Bei Schemaevolution versucht man, die laufenden Instanzen auf dem neuen Schema zu migrieren. Um zu bestimmen, ob die verzerrten Instanzen auf dem neuen Schema beruhen können, muss sowohl die syntaktische und als auch die semantische Verträglichkeit zwischen der Instanz und dem neuen Schema überprüft werden. Die syntaktische Überprüfung wurde bereits gelöst. Wir beschäftigen uns in dieser Arbeit mit der semantischen Überprüfung.

Es gibt ein paar wichtige Fragen für jede Gruppe zu beantworten, z.B. soll die Überprüfung nach jeder Änderungsoperationen ausgeführt oder eher einmalig nach eine Reihe von Operationen? Gibt es für diese Gruppe von Fällen etwas besonders zu berücksichtigen? Wie kann man die Effizienz der semantischen Überprüfung für das Szenario verbessern? usw. Dabei gehen wir immer davon aus, dass die zu überprüfenden Prozesse bereits syntaktisch überprüft werden und syntaktisch korrekt sind. Wir können uns syntaktisch falsche Prozesse vorstellen können, für die die semantische Überprüfung sehr schwierig oder gar nicht möglich ist.

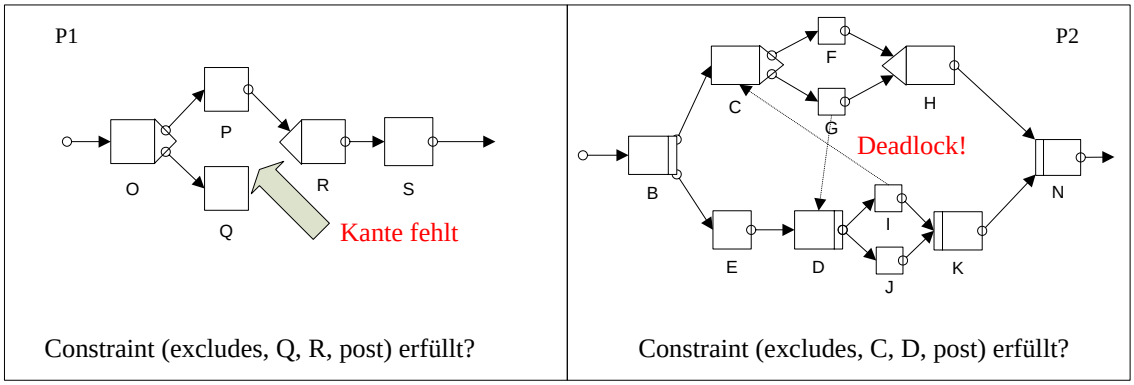


Abb. 3.1 Syntaktisch falsche Schemata

Die Abbildung 3.1 liefert einen Hinweis auf das Problem der semantischen Überprüfung an syntaktisch falschen Prozessen. In dem Prozess P1 fehlt eine Kante zwischen den Aktivitäten Q und R. Hierbei ist unklar, ob die fehlende Kante im Nachhinein noch hinzugefügt wird. Wenn die Kante hinzugefügt wird, ist das Constraint *(excludes, Q, R, post)* verletzt. Es ist aber auch möglich, dass die Aktivität Q bei Schemamodellierung überflüssig da gelassen ist. Wenn die Aktivität Q nach der syntaktischen Überprüfung gelöscht wird, ist das Constraint *(excludes, Q, R, post)* mit nicht vorkommender Source-Aktivität erfüllt. Deswegen kann angesichts der Situation keine Aussage getroffen werden, ob das Constraint erfüllt ist oder nicht.

In P2 wird ein Deadlock durch die beiden Sync-Kanten zwischen I und C und zwischen G und D erzeugt. Zur Laufzeit wartet die Aktivität C auf die Beendung der Aktivität I, ein Nachfolger von D. Mittlerweile wartet D auf die Beendung der G, ein Nachfolger von C. Die beide Aktivitäten C und D werden nie zur Ausführung kommen. Je nachdem, welche Sync-Kante gelöscht wird, ergeben sich unterschiedliche Resultate für die semantische Korrektheit. Falls die Sync-Kante zwischen G und D gelöscht wird, wird das Constraint *(excludes, C, D, post)* erfüllt. Wenn die Sync-Kante zwischen I und C gelöscht wird, wird das Constraint verletzt.

Alles im allem ist es schwierig und auch nicht so sinnvoll, eine Aussage über die semantische Korrektheit der Prozesse, die syntaktisch inkorrekt sind, zu machen. Daher beschränken sich die Untersuchungen dieser Arbeit auf syntaktisch bereits erfolgreich verifizierte Prozesse.

3.1 Verifikationen eines ganzen Schemas

Bei diesem Szenario wird ein ganzes Schema zu überprüfen geliefert. In diesem Abschnitt geht es darum, die Besonderheiten bei der Verifikation eines ganzen Schemas zu analysieren und die Vorgehensweise der semantischen Überprüfung zu untersuchen.

3.1.1 Anwendungsfälle

In den adaptiven PMS soll es möglich sein, Schemata, die in einem Modellierungswerkzeug ohne semantische Korrektheitschecks modelliert wurden, zu importieren. Dabei muss man aufpassen, dass das importierte Schema nicht direkt verwendet werden kann, obwohl es syntaktisch korrekt ist. Vor der Verwendung muss es sichergestellt werden, dass das Schema auch mit den Constraints, die zu dem Schema gehören, verträglich ist. D.h. die externen Schemata müssen in dem adaptiven PMS semantisch überprüft werden. Die Schemata müssen ebenfalls überprüft werden, falls neue Constraints für das Schema hinzugefügt werden (vgl. Abbildung 3.2).

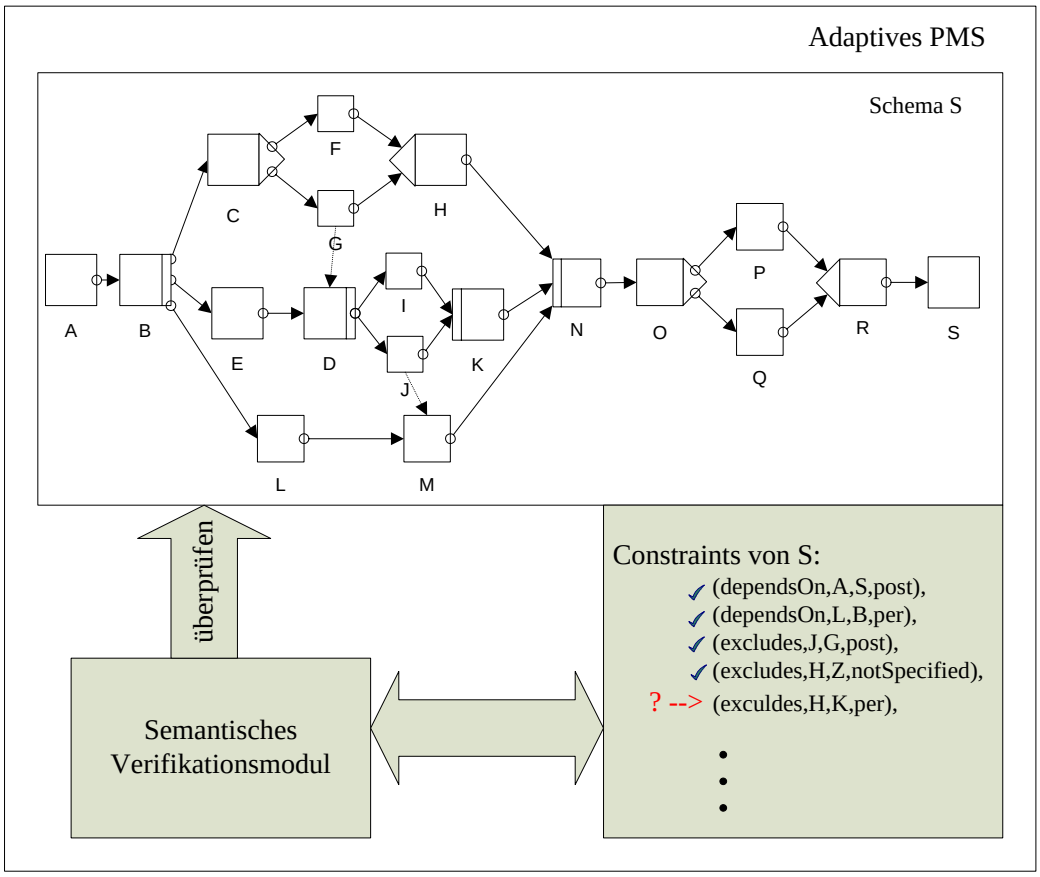


Abb. 3.2 Verifikation eines Prozessschemas

3.1.2 Besonderheiten

Bei Verifikation eines ganzen Schemas muss normalerweise die Verträglichkeit zwischen dem Schema und jedem zu dem Schema gehörenden Constraint überprüft werden. Diese Constraints müssen für jeden möglichen Ausführungspfad erfüllt sein. Aus einem Schema mit XOR-Block kann man Instanzen mit unterschiedlichen Ausführungspfaden erzeugen (vgl. Abbildung 3.3). Bei der Verifikation eines Schemas muss sichergestellt werden, dass alle möglichen Instanzen des Schemas semantisch korrekt sind [3]. Erst dann ist das Schema korrekt.

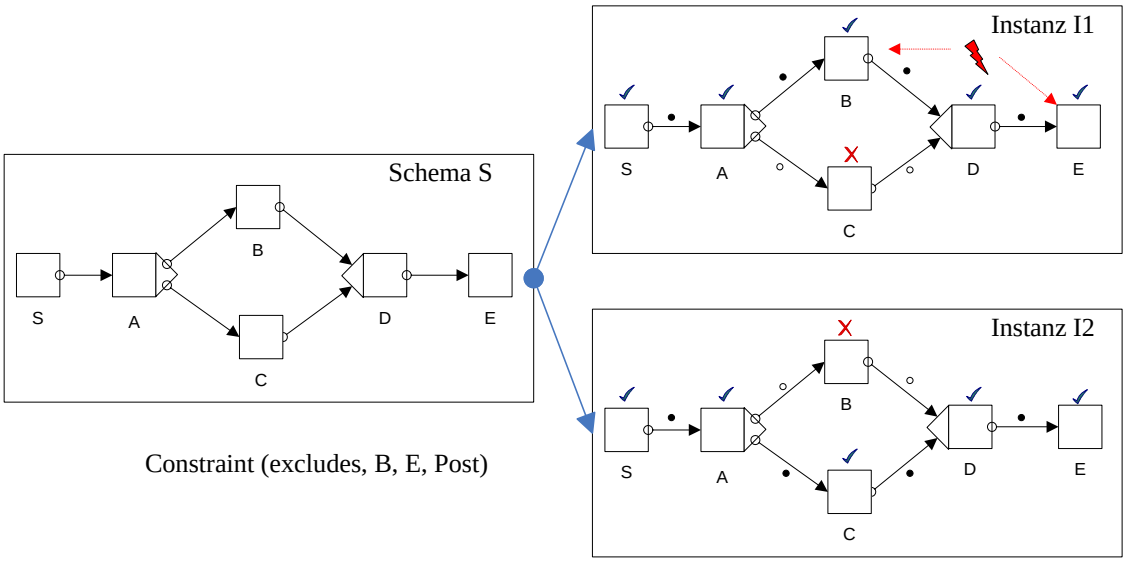


Abb. 3.3 Instanzen mit unterschiedlichen Ausführungspfade

Obwohl die Aktivitäten B und C sich in einem XOR-block befinden und zur Laufzeit nicht jedes mal ausgeführt werden, müssen sie beide bei der Verifikation eines Schemas als die direkten Nachfolger von A und die direkten Vorgänger von D betrachtet werden. Für XOR-Blöcke mit mehr als zwei Zweigen gilt dasselbe. Das Schema S in Abbildung 3.3 ist semantisch inkorrekt, weil das Constraint *(excludes, B, E, post)* verletzt ist. Es kann aus S die semantisch inkorrekte Instanz I1 erzeugt werden, obwohl die andere mögliche Instanz I2 korrekt ist.

3.1.3 Naive Vorgehensweise und Optimierungsmöglichkeiten

Die naive Vorgehensweise für die Verifikation des Schemas kann wie folgt aussehen: Es wird sowohl die Source-Aktivität als auch die Target-Aktivität vom Constraint in dem ganzen Schema gesucht. Bei der naiven Vorgehensweise kann man das Schema nach der Source-Aktivität ein Mal durchsuchen. Wenn sie nicht gefunden wird, bleibt das Constraint unberührt und gilt als erfüllt. Wenn sie gefunden wird, dann durchsucht man noch einmal das Schema für die Target-Aktivität. Je nachdem, ob die Target-Aktivität vorkommt oder nicht, kann man eine Aussage über die semantische Korrektheit anhand des Reihenfolgeverhältnisses von den beiden Aktivitäten treffen. Mit Pseudocode lässt sich dieser einfache Algorithmus wie folgt beschreiben:

Definitionen:

C : Menge aller Constraints auf das Schema S.
Succ*(A) : Menge aller direkten oder indirekten Vorgänger von A bzgl. normaler Kontrollkanten und Sync-Kanten. [9]
Pred*(A) : Menge aller direkten oder indirekten Nachfolgeraktivität von A bzgl. normaler Kontrollkanten und Sync-Kanten. [9]
Tasks(S) : Funktion, die alle Aktivitäten in S zurückgibt.
Source(c) : Funktion, die die *Source-Aktivität* vom Constraint c zurückgibt.
Target(c) : Funktion, die die *Target-Aktivität* vom Constraint c zurückgibt.
Position(c): Funktion, die den Parameter *Position* von das Constraint c zurückgibt.
Type(c): Funktion, die den Parameter *Type* von das Constraint c zurückgibt.
KBlock(M) : Funktion, die den minimalen Kontrollblock zurückgibt, der alle Aktivitäten der Menge *M* umschließt. Es gibt drei Returnwerte: XOR-Block, UND-Block oder *Null*. *Null* steht dafür, dass keine Aktivitäten in der Menge M sich in eine Block-Struktur befinden.
XOR-Zweig(A, B): boolesche Funktion, die zurückgibt, ob die beiden Aktivität A und B sich in demselben Zweig eines XOR-Block befinden.

Code:

```
For each c ∈ C do
// für jedes zu überprüfende Constraint
{
      if Source(c) ∉ Tasks(S) then
// Methode aufrufen und Schema S einmal durchlaufen, um die Source-Aktivität
// zu finden.

                Ausgabe c erfüllt
// suche das nächste Constraint
      else
      {
        if Position(c) = notSpecified
        {   if Type(c) = dependsOn
          {       if Target(c) ∉ Tasks(S) then
                // Die Target-Aktivität kommt nicht vor.
                        c nicht erfüllt → Konflikt gefunden!
                else
                    if KBlock( {Target(c)} )=XOR-Block und
                    // Target(c) wird nicht jedes Mal ausgeführt.
                       NOT(XOR-Zweig(A, B))
                    // Source(c) und Target(c) befinden sich nicht in demselben
```

// Zweig eines XOR-Blocks
c nicht erfüllt → Konflikt gefunden!
else
Ausgabe c erfüllt }

if Type(c) = excludes
{ if Target(c) ∉ Tasks(S) then
// Die Target-Aktivität kommt nicht vor.
Ausgabe c erfüllt
else
if XOR-Zweig(A, B)
// Source(c) und Target(c) befinden sich in demselben Zweig
// eines XOR-Blocks
c nicht erfüllt → Konflikt gefunden!
else
Ausgabe c erfüllt }}

if Position(c) = pre
{ if Type(c) = dependsOn
{ if Target(c) ∉ Tasks(S) then
// Die Target-Aktivität kommt nicht vor.
c nicht erfüllt → Konflikt gefunden!
else
if Target(c) ∉ Succ*(A) oder
((KBlock({Target(c)})=XOR-Block) and
(NOT(XOR-Zweig(A, B))
// Target(c) ist keine Vorgänger der Source(c) oder wird nicht
// jedes mal vor der Source(c) ausgeführt .
c nicht erfüllt → Konflikt gefunden!
else
Ausgabe c erfüllt }

if Type(c) = excludes
{ if Target(c) ∉ Tasks(S) then
// Die Target-Aktivität kommt nicht vor.
Ausgabe c erfüllt
else
if Target(c) ∈ Succ*(A) oder
KBlock({Target(c)})=UND-Block
// Target(c) ist ein Vorgänger von Source(c) oder Target(c) und
// Source(c) befinden sich in zwei unterschiedlichen Zweig eines
// UND-Blocks
c nicht erfüllt → Konflikt gefunden!
else
Ausgabe c erfüllt }}

if Position(c) = post
{ if Type(c) = dependsOn
{ if Target(c) ∉ Tasks(S) then
// Die Target-Aktivität kommt nicht vor.
c nicht erfüllt → Konflikt gefunden!
else
if Target(c) ∉ Pred*(A) oder
((KBlock({Target(c)})=XOR-Block) and

```
                                        (NOT(XOR-Zweig(A, B))
            // Target(c) ist keine Vorgänger der Source(c) oder wird nicht
            // jedes mal nach der Source(c) ausgeführt .
                c nicht erfüllt → Konflikt gefunden!
            else
                Ausgabe c erfüllt          }

if Type(c) = excludes
{   if Target(c) ∉ Tasks(S) then
        // Die Target-Aktivität kommt nicht vor.
            Ausgabe c erfüllt
        else
            if (Target(c) ∈ Pred*(A)) oder
                                    ((KBlock( {Target(c)} )=UND-Block)
            // Target(c) ist ein Nachfolger von Source(c) oder Target(c) und
            // Source(c) befinden sich in zwei unterschiedlichen Zweig eines
            // UND-Blocks
                c nicht erfüllt → Konflikt gefunden!
            else
                Ausgabe c erfüllt          }}
                                                    }}
```

Als Optimierung können wir die Constraints, deren Source-Aktivität nicht im Prozess vorkommt, statt nach einem erfolglosen Durchlauf des Prozesses, bereits vor der Überprüfung ausschließen. Es ist auch nicht notwendig für jedes zu überprüfende Constraint das Schema ein paar Mal durchsuchen. Wir können die semantische Überprüfung mehrere Constraints in einem Durchlauf integrieren. In Kapitel 4 wird einen Algorithmus vorgestellt, der dazu fähig ist.

3.2 Modellierung / Änderung eines Prozess-Schemas

In diesem Abschnitt werden wir die Modellierung / Änderung eines Prozess-Schemas diskutieren.

3.2.1 Anwendungsfälle

Als Verlaufsmodell für die Instanzen sind die Schemata im PMS die grundlegende Basis. Der Realität entsprechend ändern sich die realen Geschäftprozesse, die das PMS verwaltet, auch stetig. Falls die Änderungen dauerhaft sind, z.B. wegen einer Gesetzesänderung, ist es sinnvoll nicht jede einzelne Instanz anzupassen, sondern die Änderung einmalig am Schema vorzunehmen und die Schemaänderung auf die Instanzen zu propagieren. Deswegen muss es in einem adaptiven PMS möglich sein, sowohl Schemata zu modellieren als auch vorhandene Schemas zu ändern. Für das geänderte Schema oder das neu modellierte Schema muss die semantische Korrektheit überprüft werden, um die problemlose Ausführung der Instanzen zu gewährleisten. Reine Schemaänderungen sollen keine Instanzen betreffen, d.h. es gibt entweder keine laufenden Instanzen, die auf dem zu ändernden Schema basieren, oder sie werden nicht berücksichtigt. Die Migration der Instanzen aufs geänderte Schema wird in Abschnitt 3.4 behandelt.

3.2.2 Besonderheiten

In diesem Abschnitt werden wir die Besonderheiten der semantischen Überprüfung bei Schemaänderung und –modellierung ermitteln.

3.2.2.1 Unfertige Schemata

Bei der Schemamodellierung muss man die unvollständigen Schemata überprüfen. Manchmal ist es nicht zu verlangen, dass ein Teilschema auch alle semantischen Constraints erfüllt. Insbesondere gilt dies bei Abhängigkeitsbeziehungen.

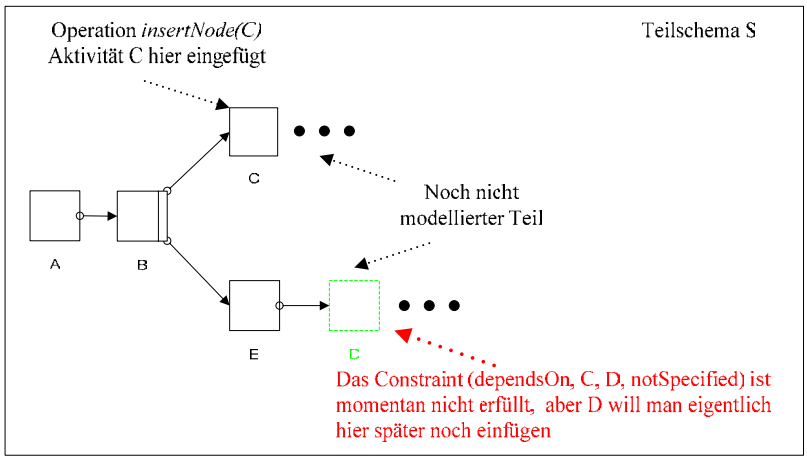


Abb. 3.4 Schemamodellierung

Abbildung 3.4 zeigt ein Teilschema S, wobei das Constraint *(dependsOn, C, D, notSpecified)* nicht erfüllt ist. Jedoch kann dieser semantische Konflikt temporär sein, d.h. das Constraint lässt sich mit weiterer Modellierung eventuell wieder erfüllen. Da das Schema schrittweise modelliert wird, lassen sich die Aktivitäten, von denen anderen Aktivitäten abhängig sind, nicht auf einmal in das Schema einfügen. In diesem Beispiel würde die Target-Aktivität D im Nachhinein noch in dem Schema eingefügt werden, damit wird das Constraint erfüllt. Die Änderungsoperationen, die eine Abhängigkeitsbeziehung verletzt, sollen daher bei Schemamodellierung auf keinen Fall direkt verboten oder ablehnen. Vernünftiger wäre es hier, eine Warnung mit den verletzten Constraints zur Erinnerung auszugeben. Dagegen ist die unerfüllte Ausschlussbeziehung eher ein Konflikt, der durch eine semantisch inkorrekte Änderungsoperation erzeugt wird.

3.2.2.2 Verletzte und unerfüllte Constraint

In Allgemein ist ein verletztes Constraint selbstverständlich unerfüllt. Aber hier möchte ich das Wort „unerfüllt“ speziell für die Constraints verwenden, die zwar momentan nicht erfüllt, aber noch erfüllt werden kann. Das Constraint *(dependsOn, C, D, notSpecified)* in Abbildung 3.4 gehört beispielsweise zu dieser Gruppe, weil es sich später durch die Operation *InsertNode(D, E, …)* wieder erfüllen lässt. Wie in dem Abschnitt 3.2.2.1 erwähnt, soll es für die unerfüllten Constraints nur Warnungen ausgegeben werden. Die Operation, die die Unerfüllung verursacht, wie z.B. die Operation *InsertNode(C, B, …)* auf dem Teilschema S in Abbildung 3.4 soll nicht abgelehnt wird.

Dagegen sind die verletzten Constraints nicht mehr mit weiterer Modellierung zu erfüllen. Die Abbildung 3.5 illustriert ein Teilschema bei Modellierung, wo das Constraint *(excludes, A, C, post)* verletzt ist. Das Constraint lässt sich durch keine weitere Modellierungsoperationen erfüllen außerdem die Modellierungsoperation, die den Konflikt verursacht zurückzusetzen, d.h. die Aktivität C wieder zu löschen. Deswegen kann man die Änderungsoperationen, die ein oder mehrere Constraints verletzen, direkt ablehnen.

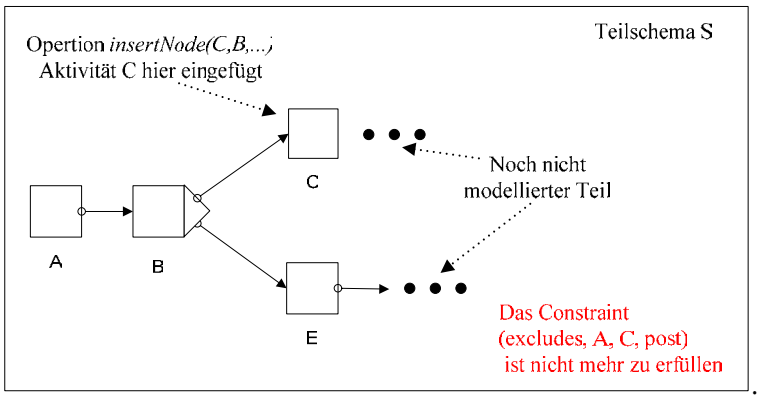


Abb. 3.5 Verletztes Constraints

Jetzt ist man mit der Frage konfrontiert: Die Änderungsoperationen, die Constraints verletzen oder unerfüllt machen, sollen unterschiedlich behandelt werden. Wie kann man es dann unterschieden? Vielleicht hat man die Antwort dafür durch die Abbildungen 3.4 und 3.5 bereits erraten, wenn es um eine Ausschlussbeziehung geht, ist das nicht erfüllt Constraint „verletzt“. Wenn es um eine Abhängigkeitsbeziehung geht, ist das nicht erfüllt Constraint „unerfüllt“.

Definition: die unerfüllte und verletzte Constraints

- Mit dem Wort „**unerfüllt**“ beschreiben wir die Constraints, die bei Schemamodellierung nicht erfüllt sind und eine Abhängigkeitsbeziehung repräsentieren.

- Mit dem Wort „**verletzt**“ beschreiben wir die Constraints, die bei Schemamodellierung nicht erfüllt sind und eine Ausschlussbeziehung repräsentieren.

Der Grund liegt darin, dass wenn eine Ausschlussbeziehung nicht eingehalten wird, mindestens ein „Pfad“ zwischen den zwei unverträglichen Aktivitäten existiert, wie z.B. das Pfad A->B->C in Abbildung 3.5 existiert. Das gilt auch, wenn die Aktivität A nicht jedes Mal zur Laufzeit ausgeführt wird, d.h. sie befindet sich wie in Abbildung 3.6 in einem XOR-Block.

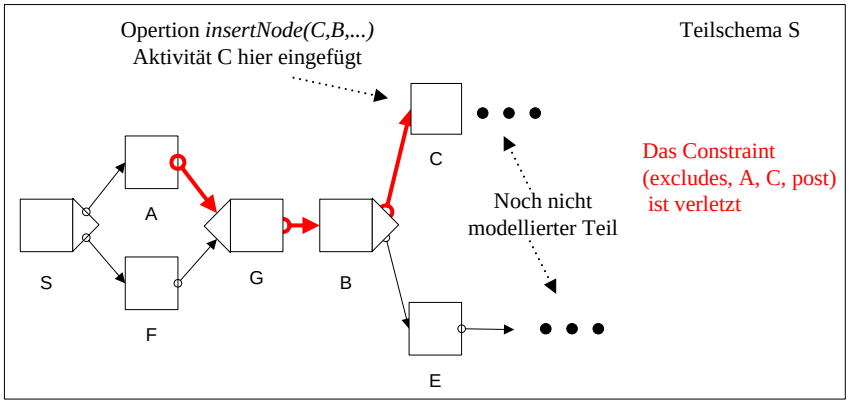


Abb. 3.6 Unverträgliche Aktivitäten in XOR-Block

Trotzdem kann sich zur Laufzeit eine Ausführungsspur A->G->B->C ergeben, wobei das Constraint *(excludes, A, C, post)* verletzt ist. Das Constraint wird erfüllt, nur wenn der Pfad „zerstört“ wird. Deswegen kann man die Änderungsoperationen, die eine Verletzung der Ausschlussbeziehung verursachen, direkt ablehnen. Im Beispiel ist sie die Operation *insertNode(C)*. Es ist leichter, die Ausführung der semantisch inkorrekten Änderungsoperationen zu verhindern, als später die semantischen Konflikte zu lösen.

Für die Abhängigkeitsbeziehung besteht dagegen bevor der Beendung der Modellierung immer die Möglichkeit, die unerfüllten Constraints durch Einfügen von der Target-Aktivität (*insertNode(D)* in Abbildung 3.4) oder Löschen von der Source-Aktivität zu erfüllen. Die Änderungsoperationen, die eine Verletzung der Abhängigkeitsbeziehung verursachen, sollen nicht verboten werden. Stattdessen könnte dem Bearbeiter eine dauerhafte Warnung mit den unerfüllten Constraints und den fehlenden Aktivitäten anzeigen (vgl. Abbildung 3.7):

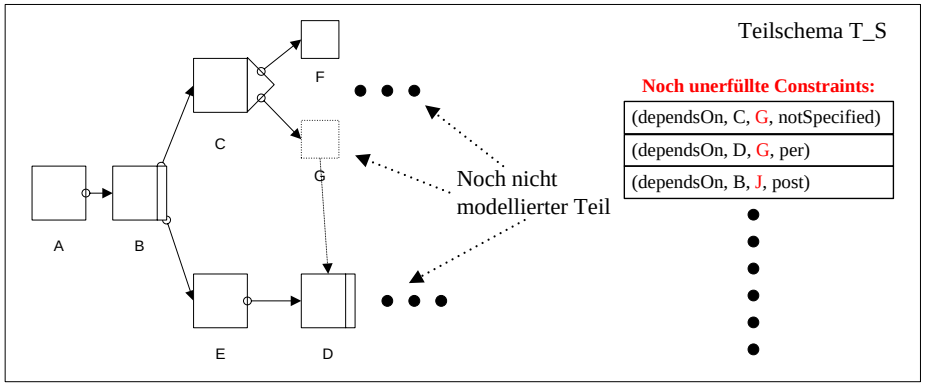


Abb. 3.7 Warnungen für die unerfüllten Constraints

3.2.3 Naive Vorgehensweise und Optimierungsmöglichkeiten

Die semantische Überprüfung bei Modellierung oder Änderung des Prozess-Schemas ist prinzipiell dieselbe wie die Verifikation eines ganzen Schemas. Als Überprüfungsverfahren ist die in dem Abschnitt 3.1.3 beschriebene naive Lösung hier ebenfalls einsetzbar.

Das Verfahren ist zwar richtig, aber offensichtlich sehr ineffizient. Dabei werden zu viele überflüssige Aktionen durchgeführt. Als Optimierungsmöglichkeiten machen wir folgende Vorschläge:

Zunächst lässt sich die Anzahl der zu überprüfenden Constraints stark reduzieren. Eine einzelne Änderungsoperation betrifft normalerweise nicht alle Constraints, sondern nur einen Teil davon. Je nach der Operation kann man die Menge von Constraints, die durch diese Operation gefährdet sein können, einschränken. Dabei bieten die Parameter in der Operation nützliche Informationen. Das Constraint *(dependsOn, A, B, pre)* kann beispielsweise nicht durch die Operation *insertNode(B, X, Y)* verletzt werden. Wir müssen daher hierbei nicht wie bei Verifikation eines ganzen Schemas alle Constraints des Schemas überprüfen sondern nur die Constraints, die durch die Modellierungsoperation oder die Änderungsoperation potenziell verletzt werden können, überprüfen. Das werden wir später in Kapitel 5 noch genauer ermitteln. Weil dadurch nicht jedes Mal alle Constraints des Schemas neu überprüft werden, muss es bei Schemamodellierung aufgepasst werden, ob die unerfüllten Constraints durch die neue Operation wieder erfüllt werden. Wenn ja, soll die Warnungsliste der unerfüllten Constraints entsprechend aktualisiert werden.

Die Anzahl der Durchläufe auf dem Schema kann auch reduziert werden. Es muss nicht für jede Constraint das Schema ein oder sogar zwei Mal durchsuchen. Mit einem eleganten Algorithmus kann man mit einem Durchlauf des Schemas alle Constraints überprüfen. Die Entwicklung der Algorithmen werden wir in Kapitel 4 behandeln.

Weitere denkbare Verbesserung wäre, die semantische Korrektheit nicht für jede Änderungsoperation zu überprüfen sondern für eine Menge von Operationen, um die eventuelle Redundanz zu vermeiden. Es gibt aber semantisch unverträgliche Operationen, die allein semantisch richtig auf dem ungeänderten Schema sind, aber zusammen semantische Konflikte erzeugen werden. Und es ist aufwendig, die problematischen Operationen im Nachhinein zu finden, wenn eine Menge von Operationen wegen semantischen Konflikts abgelehnt wird. Dann muss man vielleicht wieder jede einzelne Änderungsoperation überprüfen. Ebenfalls gilt es für die Ad-hoc-Änderung an Instanzen. Das werden wir noch in Abschnitt 3.3.2.2 genauer mit Beispielen veranschaulichen.

3.3 Ad-hoc-Änderung an Instanzen

Im diesen Abschnitt werden Ad-hoc-Änderungen an Instanzen analysiert. Eine semantisch inkorrekte Operation an einer Instanz kann semantische Konflikte erzeugen. Um diese zu vermeiden, muss die semantische Korrektheit der Ad-hoc-Änderung durch die Überprüfung gewährleistet werden.

3.3.1 Anwendungsfälle

Die Wichtigkeit der Adaptivität für ein PMS wurde bereits in Kapitel 1 erklärt. Die allgemeinen Änderungen, die alle Instanzen betreffen, kann man durch Änderung des Schemas anpassen. Damit gilt die Änderung auch bei jeder neuen Instanz. Aber oft gibt es auch spezielle Anforderungen von einzelnem Kunden, wobei nur eine oder einige Instanzen betroffen sind. Beispielsweise kann sein, dass bei Durchführung eines Projektes einige neue Arbeiten wegen eines externen Grund zusätzlich getan werden muss oder einige geplante Arbeiten nicht mehr getan werden soll. Dann muss die Instanz, die durch das adaptive PMS verwaltet wird, auch entsprechend geändert werden. Da will man statt des Schemas nur eine einzelne Instanz zur Laufzeit ändern.

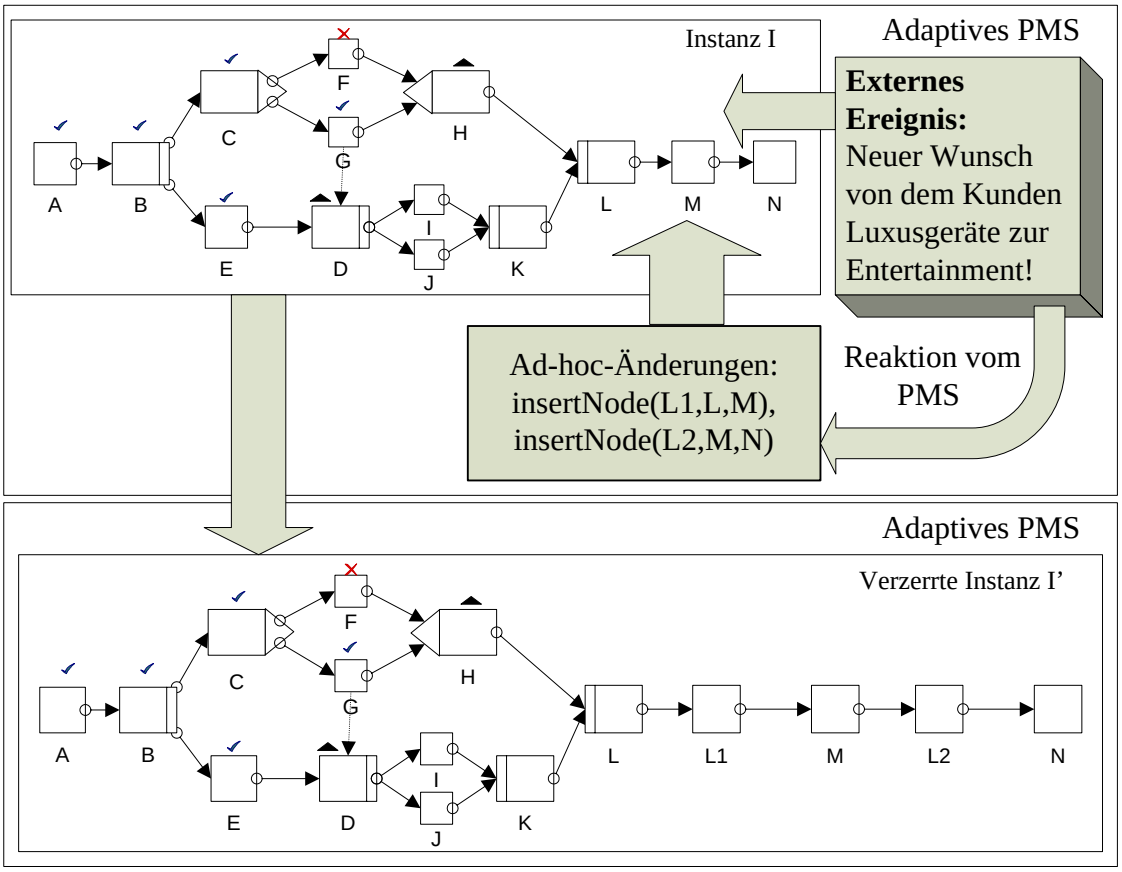


Abb. 3.8 Ad-hoc-Änderungen

Die Abbildung 3.8 illustriert ein Beispiel für die Anwendung der Ad-hoc-Änderungen. Die gezeigt Geschäftsprozesse kann beispielsweise die Vorgänge der Produktion eines Schiffs darstellen (stark vereinfacht). Es wurde ein Vertrag für den Bau des Schiffs geschlossen, dabei werden die Details von dem Unternehmen und dem Kunden zusammen festgelegt. Entsprechend erzeugte das Unternehmen die Produktionsprozesse, die in der Instanz I gezeichnet wurde, mittels eines adaptiven PMS. Aber nach dem Anfang des Aufbaus bekommt das Unternehmen neue Anforderungen von dem Kunden, dass es zusätzlich noch ein paar Luxusgeräte zur Entertainment eingebaut werden soll. Der Einbau ist mit dem Hauptbildprozess unabhängig und stört den Vorgang auch nicht. Deshalb werden die Anforderungen von dem Unternehmen akzeptiert. Es wird geplant, dass die Geräte nach der Befertigung des Hauptteils des Schiffs eingebaut werden sollen. Die Änderungen muss man entsprechend durch Änderungsoperationen an der Instanz realisieren. Hier werden die zwei Aktivität L1, L2 für Einbau der neuen Luxusgeräte vor dem Ende des Prozess eingefügt. Nach der Ad-hoc-Änderungen bekommt man die verzerrte Instanz I'.

Mit dem Wort „**verzerrt**“ beschreiben wir die Instanzen, die durch Ad-hoc-Änderungen modifiziert sind. Mit dem Wort „**unverzerrt**“ beschreiben wir die Instanzen, die strukturell mit dem Originalschema identisch sind [2,7]. Die verzerrte Instanz von I wird als I' gezeichnet.

Aber solche Änderung kann Problem auslösen. Die neuen Aktivitäten L1, L2 können unverträglich mit vorherigen Aktivitäten sein, z.B. das Constraint *(excludes, H, L1, post)* wird durch die Ad-hoc-Änderungen verletzt. Um solche Konflikte zu vermeiden, muss man nach den Ad-hoc-Änderungen die semantische Korrektheit des Prozesses überprüfen.

3.3.2 Besonderheiten

In diesem Abschnitt versuchen wir, die Besonderheiten der Ad-hoc-Änderungen bei semantischer Überprüfung zu ermitteln.

3.3.2.1 Toleranz gegenüber Änderungsoperationen

Im Gegensatz zu Änderungen an einem Prozessschema sind bei Durchführung der Ad-hoc-Änderung sowohl das Schema als auch den Ausführungszustand von der Instanz zu berücksichtigen. Der Ausführungszustand ist für die Überprüfung semantischer Korrektheit relevant, weil dadurch eventuell mehr Operationen an der Instanz anwendbar sein können. Das passiert, wenn ein Zweig eines XOR-Blocks gewählt wird und damit alle anderen Zweige abgewählt werden.

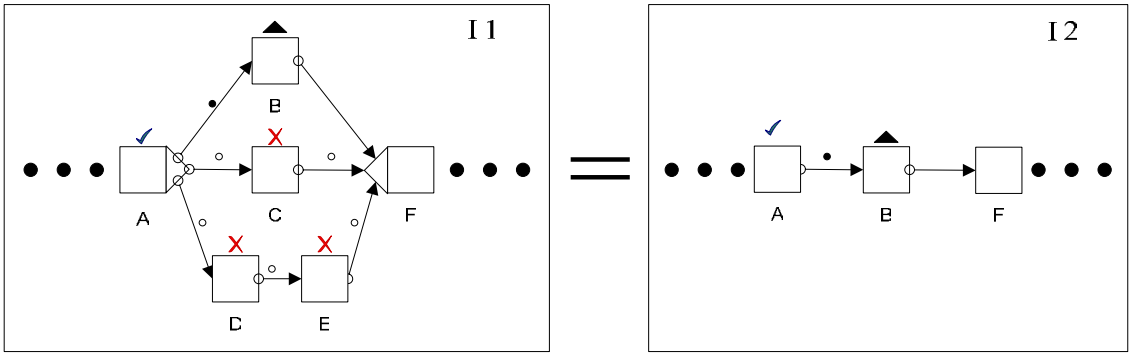


Abb. 3.9 Semantisch gleiche Teilprozesse

Die zwei Teilprozesse in Abbildung 3.9 sind bei der semantischen Überprüfung gleichwertig. Die abgewählten Teile werden dabei nicht mehr berücksichtigt. Nehmen wir an, dass es ein Constraint *(excludes, D, G, post)* gibt. Trotzdem kann man die Aktivität G hinter F in der Instanz I1 hinzufügen. Das wäre auf dem Schema nicht möglich, dabei wird D in dem unteren XOR-Zweig eine Verletzung der Ausschlussbeziehung auslösen.

Die Abbildung 3.9 liefert einen Hinweis darauf, dass Instanzen eventuell toleranter gegenüber Änderungsoperationen als das entsprechende Schema sein können. Die Ursache dafür sind die Ausführungszustände der Instanzen, genauer gesagt die mit SKIPPED bewerteten Aktivitäten in XOR-Blöcke. Die Aktivitäten C, D und E werden mit SKIPPED bewertet, weil die entsprechenden XOR-Zweige abgewählt sind. Abbildung 3.10 verdeutlicht die Situation nochmal, wenn dieselbe Änderungsoperation auf dem Schema und auf einer Instanz unterschiedliche Ergebnisse hervorruft.

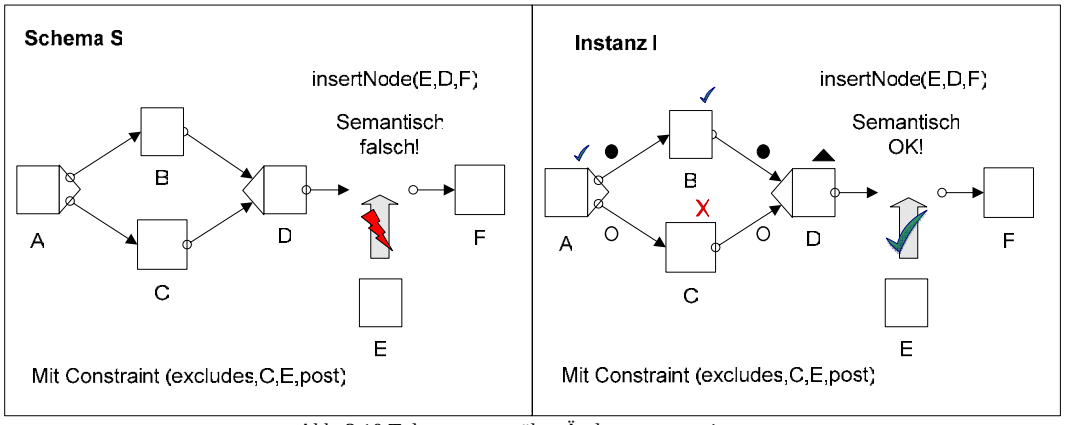


Abb. 3.10 Toleranz gegenüber Änderungsoperationen

Die Einfügeoperation *InsertNode(E, D, F)* ist erlaubt auf der Instanz, wird aber auf dem Schema abgelehnt. Der Grund dafür liegt darin, dass die Source-Aktivität von dem Constraint die Aktivität C in der Instanz I mit SKIPPED bewertet wurde. Bei der Änderung auf dem Schema muss man dagegen sicherstellen, dass für jeden möglichen Ausführungspfad von dem Schema kein semantischer Konflikt auftauchen kann, weil jeder Ausführungspfad zur Laufzeit möglich ist (vgl. Abschnitt 3.1.2). Es kann beispielsweise eine Instanz von Typ S geben, die den unteren Zweig vom XOR-Block über die Aktivität C nimmt. Dann erzeugt die eingefügte Aktivität E zusammen mit C einen semantischen Konflikt. Das Constraint *(excludes, B, E, post)* wird dadurch verletzt.

Es gibt noch ein anderes mögliches Szenario, wo die Instanzen toleranter gegenüber Änderungsoperationen als das Schema sein können.

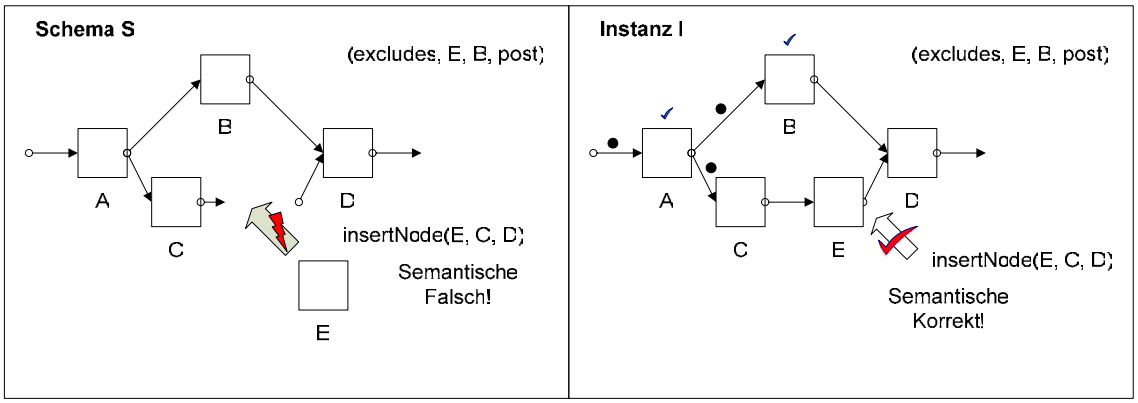


Abb. 3.11 Tolerantere Instanz

In Abbildung 3.11 ist die Operation *insertNode(E, C, D)* nur an der Instanz I erlaubt, aber nicht auf dem Schema S. Das liegt darin, dass die Source-Aktivität B in der Instanz I bereits ausgeführt ist, und damit nicht mehr nach die neu hinzugefügte Aktivität E ausgeführt werden kann. Das Constraint *(excludes, E, B, post)* wird daher nicht durch die Operation verletzt. Aber beim Schema S steht die Ausführungsreihenfolge von B und E nicht fest. Nach der Operation kann es bei einer Instanz des Schemas B nach E ausgeführt werden. Das Constraint ist daher verletzt durch die Operation.

In Allgemein gilt es, dass falls ein Schema semantisch konfliktfrei ist, sind alle darauf basierenden unverzerrten Instanzen semantisch korrekt. Eine Änderungsoperation ist mit Sicherheit anwendbar auf der unverzerrten Instanz, wenn sie auf dem Schema anwendbar ist. Andersherum gilt dies aber nicht. Eine auf der Instanz anwendbare Änderungsoperation ist nicht unbedingt zulässig auf dem Schema. Die Instanz ist sozusagen toleranter gegenüber den Änderungsoperationen. Als eine wichtige Eigenschaft werden wir den Toleranz-Unterschied zwischen der Instanz und dem Schema bei späterer Analyse in Abschnitt 3.4 noch verwenden.

3.3.2.2 Ausführung mehrerer Änderungsoperationen

Wie bei der Änderung des Schemas kann die einmalige Ausführung mehrerer Änderungsoperationen mit Überprüfung auf dem ungeänderten Prozess problematisch sein. Wir nehmen hierzu das Beispiel aus der Abbildung 3.8. Es werden zwei Aktivitäten L1, L2 hintereinander in dem Prozess hinzugefügt. In einem solchem Fall muss man aufpassen, dass sich eine semantisch inkorrekte verzerrte Instanz ergeben kann, wenn die semantisch Überprüfung für alle Operationen unabhängig auf dem ungeänderten Prozess gemacht wird, um die Ad-hoc-Änderungen auf einmal ausführen zu können. Nehmen wir an, dass es ein Constraint *(excludes, L1, L2, post*) gibt. Die Abbildung 3.12 zeigt die Überprüfungen für die zwei Operationen auf der ungeänderten Instanz.

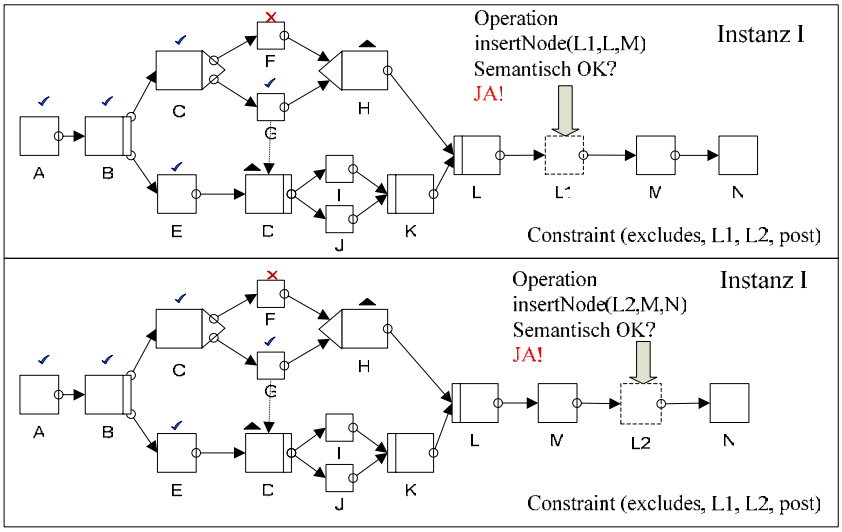


Abb. 3.12 Separate Überprüfungen

Es wird bei den semantischen Überprüfungen kein Problem gefunden, weil sie immer auf der ungeänderten Instanz beruht. Die zwei Operationen sind zwar beide semantisch richtig auf dem Originalschema, aber die eingefügten Aktivitäten sind in der Reihenfolge semantisch unverträglich.

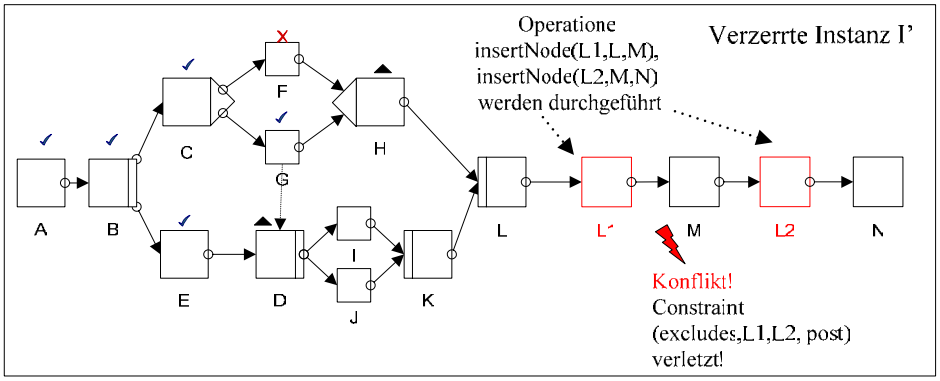


Abb. 3.13 Unentdeckter Konflikt

So ist die verzerrte Instanz I' aus der Abbildung 3.13 semantisch inkorrekt. Aber nach dem Ergebnis unabhängiger Überprüfungen werden die beiden Einfügeoperationen als semantisch korrekt festgestellt und ausgeführt. Das ist nicht die einzige Möglichkeit für semantisch unverträgliche Operationen. Man kann sich beliebig viele Fälle vorstellen. Abbildung 3.14 stellt ein anderes Beispiel dar.

Auf der Instanz I in Abbildung 3.14 sind die Operationen 1 und 2 beide semantisch richtig. Diese zwei Änderungsoperationen sind ebenfalls semantisch unverträglich. Aber der semantische Konflikt lässt sich durch unabhängige Überprüfungen nicht entdecken. Die separaten Überprüfungen führen auch hier zu unerwünschten Ergebnissen: Verletzung des Constraints *(dependsOn, G, E, notSpecified)*. Die Konflikte, die durch unverträgliche Operationen ausgelöst werden, sind nämlich vielfältig.

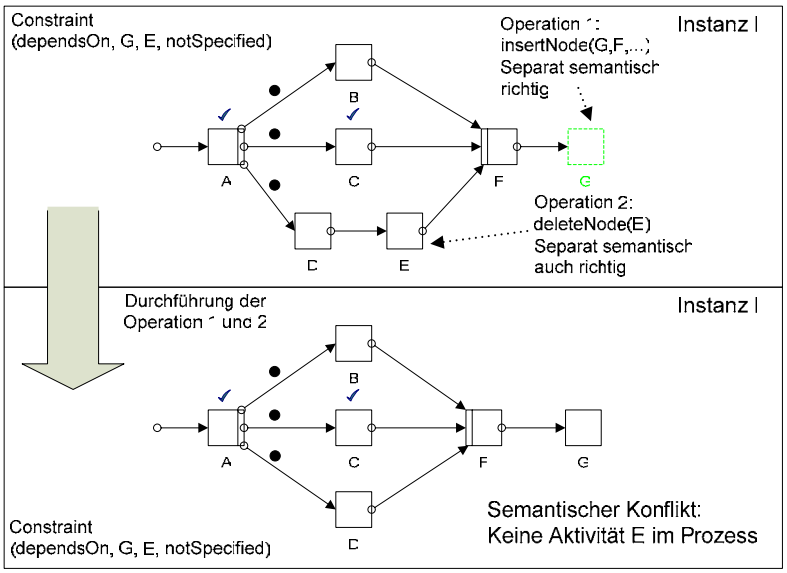


Abb. 3.14 Unverträgliche Operationen

Andersherum ist es ebenfalls möglich, dass durch Ad-hoc-Änderungen neue Operationen an der Instanz ermöglicht werden. Die Operation *insertNode(D)* in Abbildung 3.15 ist an der Instanz I nicht verwendbar, weil dadurch das Constraint *(excludes, B, D, post)* verletzt wird. Jedoch nach der Ausführung von der Operation *deleteNode(B)*, wird sie zulässig. Aber mit unabhängigen Überprüfungen der Operationen an der unverzerrten Instanz I wird die Operation *insertNode(D)* nach der Operation 1 auch als semantisch falsche Operation direkt abgelehnt.

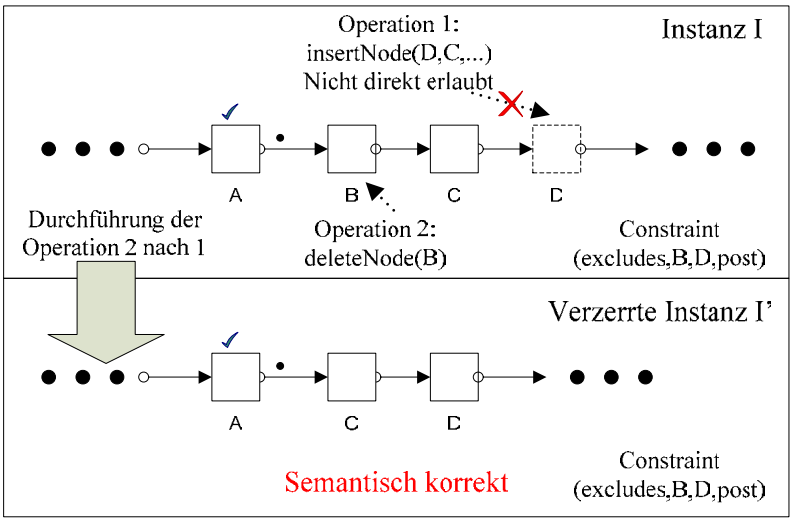


Abb. 3.15 Die neu ermöglichte Operation

Das Problem lässt sich dadurch lösen, dass wir die Überprüfung immer auf Grundlage des Zwischenergebnisses durchgeführt wird, d.h. die Überprüfung für eine Änderungsoperation immer auf der verzerrten Instanz vornehmen, an der die vorherigen Änderungsoperationen bereits ausgeführt wurden (vgl. Abbildung 3.16). Eventuelle semantische Konflikte werden dadurch entdeckt und die neu ermöglichten Operationen werden dadurch auch zugelassen.

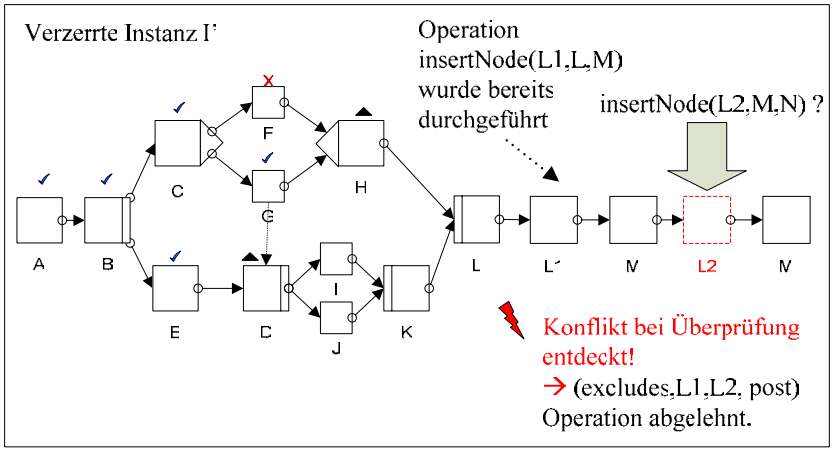


Abb. 3.16 Effektive Überprüfung

Der semantische Konflikt in Abbildung 3.12, der durch unabhängige Überprüfungen nicht entdeckt werden kann, wird in Abbildung 3.16 entdeckt, weil die semantische Wirkung der Operation 1 dabei berücksichtigt wird.

3.3.3 Naive Vorgehensweise und Optimierungsmöglichkeiten

Für die Ad-hoc-Änderung kann man das vorgestellte naive Überprüfungsverfahren für die Änderungen des Schemas einsetzen. Aber zusätzlich muss man dabei die semantische Wirkung der Ausführungszustand berücksichtigen. Für die Ad-hoc-Änderung ist das Überprüfungsverfahren ebenfalls zu aufwendig. Es besteht gleicherweise zwei Optimierungsmöglichkeiten:

1. Die zu überprüfenden Constraints lassen sich anhand der Operationen reduzieren. Wie man das macht, wird im 5. Kapitel beschrieben.

2. Wenn es mehrere Constraints zu überprüfen gibt, soll nicht für jedes Constraint die Instanz ein Mal durchlaufen werden. Die Überprüfungen kann man in eins Durchlauf integrieren.

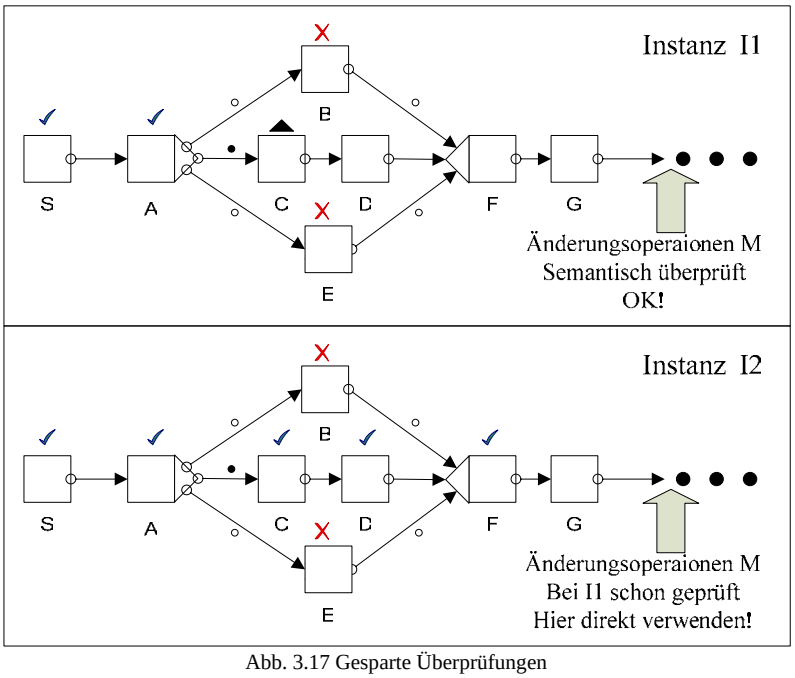


Abb. 3.17 Gesparte Überprüfungen

Für Instanzen desselben Typs, die momentan dieselbe Ausführungsspur haben, braucht man für dieselben Änderungsoperationen die semantische Korrektheit nur ein Mal zu überprüfen. Die Abbildung 3.17

illustriert die Situation. Instanzen I1 und I2 sind vom selben Typ. Für die Änderungsoperationen M reicht eine einmalige semantische Überprüfung. Dabei müssen die Ausführungszustände von den beiden Instanzen nicht unbedingt gleich sein. Hier ist die Ausführung von Instanz I2 ein bisschen weiter fortgeschritten als die von I1. Es muss aber derselbe Zweig bei allen XOR-Blöcken gewählt werden. Ansonsten können die Instanzen semantisch nicht einheitlich behandelt werden.

Die Instanzen I1 und I2 in Abbildung 3.18 lassen sich bei semantischer Überprüfung nicht einheitlich behandeln, weil sie unterschiedliche Ausführungspfade haben.

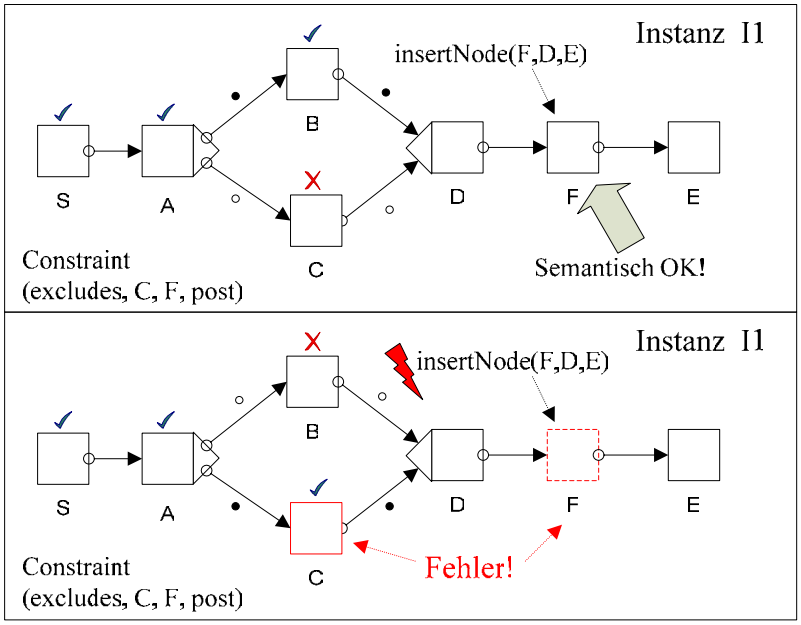


Abb.3.18 Instanzen mit unterschiedlichen Ausführungspfade

Übrigens darf die Änderungsoperation M den bereits ausgeführten Teil von der Instanz nicht betreffen. Hier lassen sich beispielsweise die Aktivitäten D und F in der Instanz I2 von der Abbildung 3.17 nicht modifizieren, was bei der Instanz I1 eventuell erlaubt ist. Jedoch werden dadurch nur syntaktische Fehler erzeugt, aber keine semantische Konflikte.

3.4 Schemaevolution und Instanzmigration

Die Schemaevolution wird als das schwierigste Szenario für die semantische Überprüfung kategorisiert. Mit semantischer Überprüfung möchten wir hierbei feststellen, welche Instanzen migrierbar sind, d.h. semantisch verträglich mit dem neuen Schema sind, und welche nicht. Da die Schemaevolution Tausende von Instanzen betreffen kann, ist die Effizienz der Überprüfung hierbei besonders wichtig.

3.4.1 Anwendungsfälle

In dem Abschnitt 1.1 wurde der Begriff Schemaevolution bereits grob erklärt. Bei Schemaevolution wird versucht, die Änderung des Schemas auf die Instanzen zu übertragen. Dieser Vorgang wird auch Instanzmigration genannt.

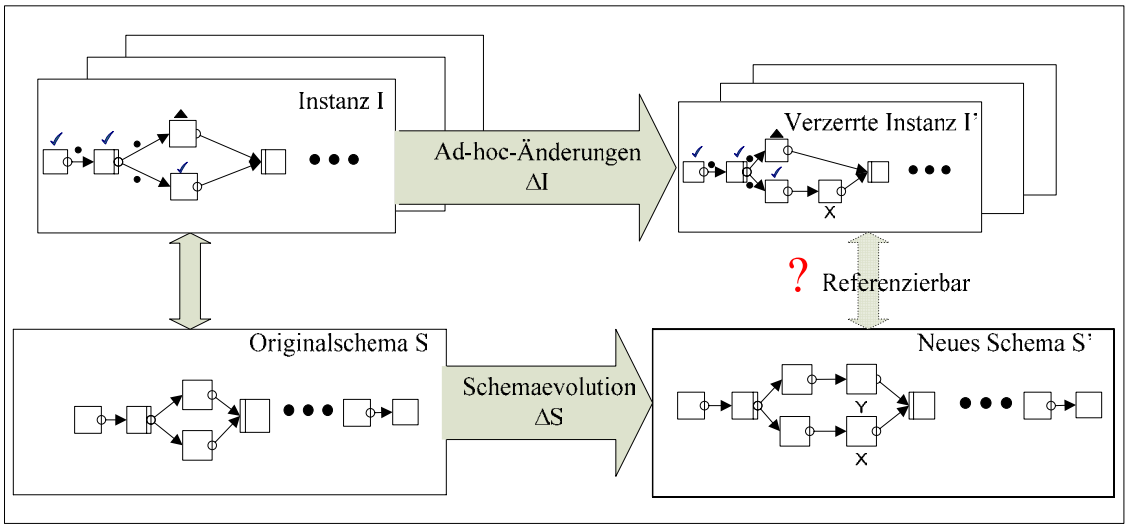


Abb. 3.19 Verzerrte Instanzen und geändertes Schema

Abbildung 3.19 zeigt das Problem, das durch die Schemaevolution erzeugt wird. Nach einer Schemaänderung ist zu bestimmen, ob die entsprechenden Instanzen nach dem neuen Schema S' ablaufen können. Die nicht migrierbaren Instanzen werden sich wie vor der Schemaevolution auf das Originalschema S beruhen und problemlos zu Ende laufen. Alle Instanzen, die nach den Änderungen gestartet werden, basieren auf dem neuen Schema S'.

Das Problem ist, dass beim adaptiven PMS durch Ad-hoc-Änderungen verzerrte Instanzen erzeugt werden. Wenn jetzt eine Schemaänderung gemacht wird, reicht die syntaktische Verträglichkeit allein nicht als das Kriterium für die Migrierbarkeit der Instanzen. Es kann syntaktisch mit dem neuem Schema S' verträgliche aber semantisch nicht verträgliche Instanzen geben.

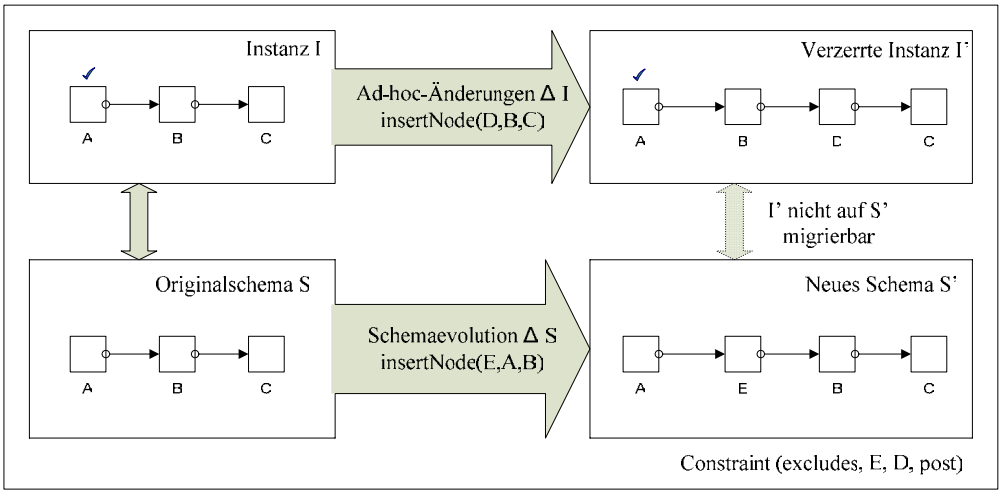


Abb. 3.20 Eine nicht migrierbare Instanz

Die verzerrte Instanz I' in Abbildung 3.20 ist syntaktisch verträglich mit dem neuen Schema S'. Die ausgeführte Aktivität A wird nicht berührt, wenn die Migration stattfinden würde. Aber dadurch wird eine semantisch inkorrekte Instanz erzeugt. Das Constraint *(excludes, E, D, post)* wird durch die Migration verletzt. Abbildung 3.21 stellt die durch Migration erzeugte semantisch inkorrekte Instanz I dar.

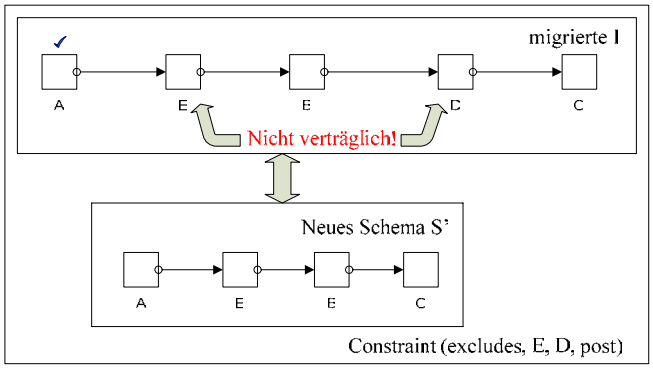


Abb. 3.21 Die migrierte I'

Deswegen ist die Überprüfung der semantischen Verträglichkeit zwischen den verzerrten Instanzen und das geändertes Schema bei Schemaevolution unerlässlich. Erst nach dem Bestanden der Überprüfung ist die Referenz der semantisch migrierbaren Instanzen auf das neue Schema umzuhängen und die Migration der entsprechenden Instanzen durchzuführen.

3.4.2 Besonderheiten

In diesem Abschnitt möchten wir klar machen, was wir bei Instanzmigration überprüfen. Dann werden wir die Instanzen anhand der Schemaänderungen ΔS und der instanzspezifischen Änderungen (Ad-hoc-Änderungen) ΔI klassifizieren. Obwohl die semantische Überprüfung bei Instanzmigration unerlässlich ist, kann man sich in vielen Fällen den Aufwand gefahrlos sparen. Um diesen Vorteil ausnutzen zu können, werden wir in Abschnitt 3.4.2.2 diese Klassen von Instanzen analysieren.

3.4.2.1 Die virtuelle migrierte Instanz

Für die drei vorherigen Szenarien haben wir immer eine Basis für die semantische Überprüfung zur Verfügung: das geänderte (Teil-)Schema oder die verzerrte Instanz. Bei der Überprüfung semantischer Migrierbarkeit einer Instanz ist es anders. Hier möchten wir die Schemaänderungen auf die Instanzen übertragen. Deswegen sind hier sowohl die Schemaänderungen als auch die Instanzen wichtig für die Überprüfung. Was wir hier sichern wollen, ist die semantische Korrektheit der Instanz nach der Migration, beispielsweise die Korrektheit der I'' in Abbildung 3.22 (Mit I'' zeichnen wir die migrierte verzerrte Instanz.). Jedoch kann die Migration erst nach der semantischen Überprüfung stattfinden, d.h. die migrierte Instanz haben wir für die semantische Überprüfung nicht zur Verfügung. Wir können aber eine virtuelle migrierte Instanz so erzeugen: Sie setzt sich aus zwei Teilen zusammen: das geänderte Schema S' und die verzerrte Instanz I'. Nehmen wir die ausgeführten Teile von der verzerrter Instanz I' direkt. In Abbildung 3.22 sind dies die Aktivitäten {S, A, B, C, E} und alle bewerteten Kanten. Als nicht ausgeführte Teile dienen die entsprechenden Teile von dem geänderten Schema als Basis. Das entspricht den Aktivitäten {D, G, F, L, H} und die Kanten zwischen ihnen in Abbildung 3.22. Auf dieser Basis sollen noch die Ad-hoc-Änderungen angewendet werden, deren Effekten die nicht ausgeführten Teile betreffen. Im Beispiel ist sie die Operation *deleteNode(L)*. Dazu können wir einfach die Delta-Schicht von der verzerrten Instanz I' einsetzen. In unserem Beispiel erhält man durch die Delta-Schicht die Information, dass H der direkte Nachfolger von F ist. Damit wird L dazwischen gelöscht und die auszuführende Ad-hoc-Änderung *deleteNode(L)* vorgenommen.

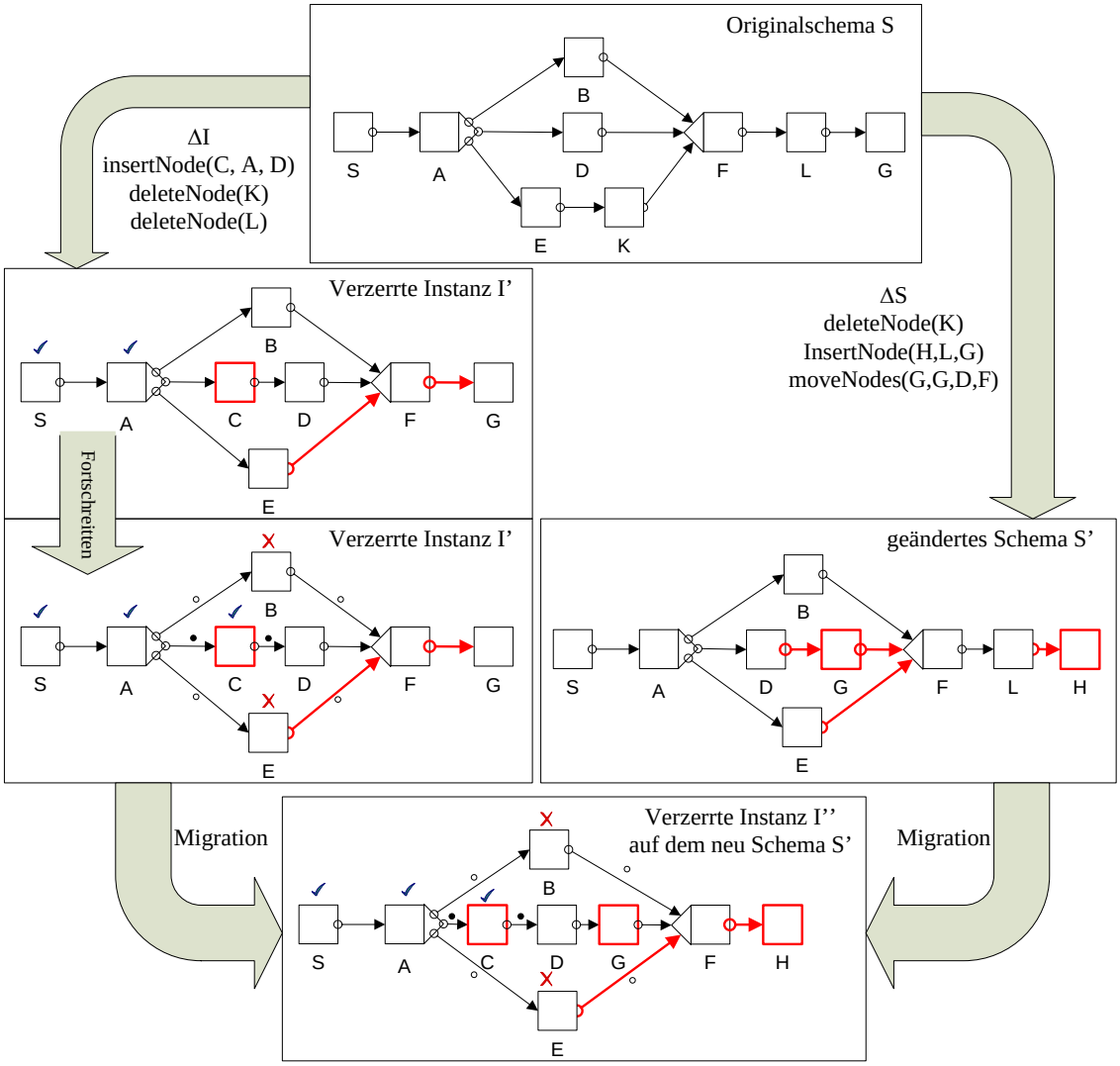


Abb. 3.22 Eine virtuelle migrierte Instanz

So werden sowohl die Ad-hoc-Änderungen ΔI als auch die Schemaänderungen ΔS berücksichtigt. Die dadurch erzeugte virtuelle migrierte Instanz ist identisch mit I'', die Instanz nach der Migration. Weil wir uns auf syntaktische migrierbare Instanzen beschränken (siehe Abschnitt 3.4.2.2), ist gewährleistet, dass die so erzeugte virtuelle Instanz syntaktisch korrekt ist.

3.4.2.2 Kategorien der Instanzen

Wie bei allen vorherigen Szenarien, ist die syntaktische Korrektheit bei der Schemaevolution auch gewährleistet. D.h. die semantische Überprüfung wird nach der syntaktischen Überprüfung durchgeführt. Wir können die Instanzen in zwei Gruppen verzerrte und nicht verzerrte, unterteilen. Nach der syntaktischen Überprüfung ergeben sich syntaktisch migrierbare und nicht migrierbare Instanzen.

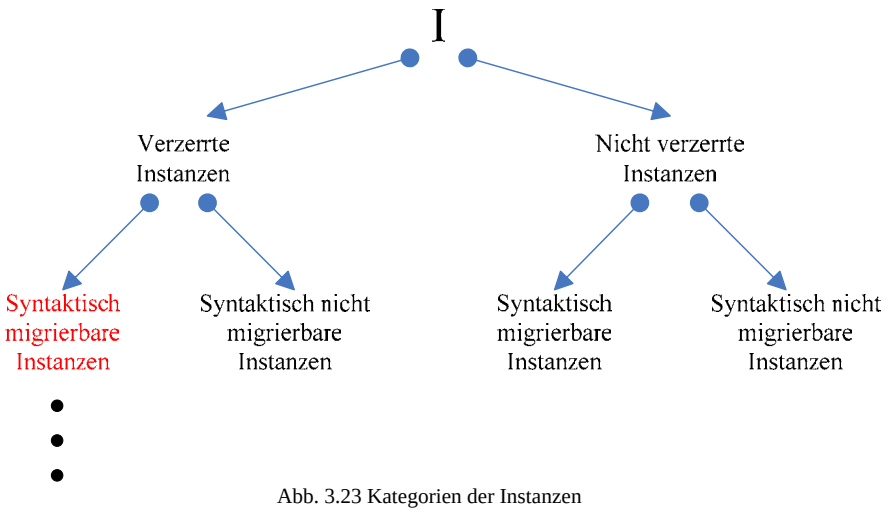


Abb. 3.23 Kategorien der Instanzen

Die Abbildung 3.23 zeigt eine grobe Kategorisierung der Instanzen. Das große I oben steht für die Menge aller Instanzen. Dazu gehören die verzerrten und nicht verzerrten Instanzen. Für die syntaktisch nicht migrierbaren Instanzen ist die semantische Überprüfung nicht notwendig. Für sie wird wegen syntaktischer Unverträglichkeit sowieso keine Migration stattfinden. Sie werden nach wie vor auf dem Originalschema zu Ende laufen. Die syntaktisch migrierbare, nicht verzerrten Instanzen sind mit Sicherheit semantisch migrierbar, weil sie ungeändert und damit strukturell identisch mit dem Originalschema sind. D.h. die Schemaänderungen ΔS sind in jedem Fall auf sie übertragbar, weil gegenüber dem entsprechenden Schema Instanzen nur mehr Änderungen erlauben können (siehe Abschnitt 3.3.2.1) [13]. Jetzt bleiben nur die verzerrten Instanzen, die syntaktisch migrierbar sind, übrig. Jedoch lassen sich diese Instanzen weiter unterteilen.

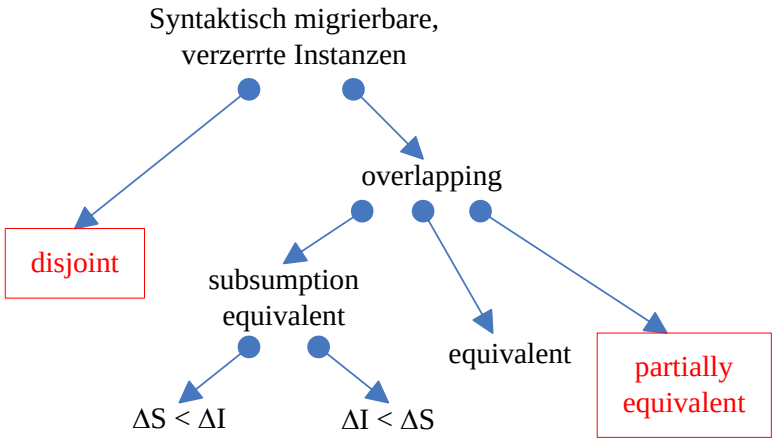


Abb. 3.24 Weitere Unterteilungen (nach [8,7])

Abbildung 3.24 stellt eine weitere nützliche Unterteilung für die semantische Überprüfung dar. Sie werden anhand des Ergebnisses syntaktischer Überprüfung gemacht. Die Zweige *disjoint, overlapping, equivalent* usw. sind Beziehungen zwischen den instanzspezifischen Änderungen ΔI und die auf S angewendeten Änderungsoperationen ΔS. Die Beziehung zwischen ΔI und ΔS wird durch syntaktische Überprüfung bestimmt. Diese Klassifikation ist ein Nebenprodukt der syntaktischen Überprüfung [8]. Diese Klasseneinteilung kann dazu verwendet werden, um die für die semantische Verifikation ohnehin irrelevanten Instanzen zu identifizieren [13].
Zur Klasse *disjoint* gehören die Instanzen, bei denen die Auswirkungen von ΔI nicht mit den Auswirkungen von ΔS überlappen. Formal: $\Delta I \cap \Delta S = \varnothing$. Die in Abbildung 3.20 dargestellten ΔI und ΔS sind disjunkt. Da sieht man, dass die verzerrte Instanz I' und das geänderte Schema S' semantisch unverträglich sind. Deswegen kann man sich die semantische Überprüfung hier nicht sparen.

Zur Klasse overlapping gehören alle Instanzen, bei denen die Auswirkungen von ΔI mit den Auswirkungen von ΔS überlappen [8]. Formal: $\Delta I \cap \Delta S \neq \varnothing$. Je nach dem Grad der Überlappung lassen sich die Instanzen dieser Klasse in die Klassen *equivalent, subsumption equivalent* und *partially equivalent* weiter unterteilen.

1. Zur Klasse *equivalent* gehören die Instanzen, bei denen ΔI exakt die gleichen Auswirkungen wie ΔS haben. Hier sind die verzerrten Instanzen mit dem geänderten Schema strukturell identisch. Damit ist die Migrierbarkeit gewährleistet. Für die Klasse *equivalent* braucht man deshalb keine Verträglichkeit zu überprüfen [13].

2. Zur Klasse *partially equivalent* gehören die Instanzen, bei denen ΔI und ΔS zwar gemeinsame Auswirkungen haben, aber jeweils noch weitere eigene Änderungen besitzen. Die ΔI und ΔS in Abbildung 3.21 sind in der Klasse *partially equivalent.* Hierbei muss man die semantische Korrektheit überprüfen, weil es außer den gemeinsamen Änderungsoperationen noch disjunkte Änderungsoperationen ausgeführt wird [13]. Man kann sich vorstellen, dass im Beispiel aus Abbildung 3.19 es jeweils eine Aktivität F an Ende der Instanz und des Schemas hinzugefügt wird. Die Schemaänderungen und die Instanzänderungen sind jetzt in der Klasse *partially equivalent*, aber die Instanz bleibt wie vorher semantisch nicht migrierbar.

3. Die Klasse *subsumption equivalent* ist zweigeteilt. Der eine Teil umfasst Instanzen, bei denen die Schemaänderungen aus ΔS die gleichen Auswirkungen besitzen wie ein Teil der Instanzänderungen aus ΔI. ΔI besitzen aber noch zusätzliche eigene Änderungen, die nicht in ΔS enthalten sind. D.h. die Auswirkungen von ΔS sind eine Teilmenge der Auswirkungen von ΔI. Dies wird in Abbildung 3.23 mit $\Delta S < \Delta I$ bezeichnet. Bei dem anderen Teil der *subsumption equivalent* Instanzen ist die Beziehung zwischen ΔI und ΔS genau umkehrt. Hier haben die ΔS gegenüber ΔI noch zusätzliche Auswirkungen auf S. Dies wird analog mit $\Delta I < \Delta S$ bezeichnet.

   - Für den Fall $\Delta S < \Delta I$ hat die Migration keinerlei Einfluss auf die verzerrten Instanzen, weil die Schemaänderungen ΔS in der Instanzänderungen ΔI enthalten sind. Die verzerrten Instanzen bleiben unverändert nach der Migration. Nur die Delta-Schichten sollen entsprechend geändert werden [8]. Aber es wird dadurch keinen semantischen Konflikt erzeugt [13].

   - Für $\Delta I < \Delta S$ werden die Änderungsoperation ΔS\ΔI gegenüber der verzerrten Instanz I' zusätzlich auf dem Schema vorgenommen. Außerdem sind die verzerrte Instanz I' und das geänderte Schema S' strukturell identisch. Deshalb stellt man sich die Frage, ob die Änderungsoperationen ΔS\ΔI an I' übertragbar sind. Wie in Abschnitt 3.3.2.1 geschrieben, die Instanz kann nur toleranter gegenüber Änderungsoperation als das Schema sein. Die Operationen ΔS\ΔI ist deswegen semantisch anwendbar an der I' [13]. Zum Verständnis können wir ein Zwischenschema ZS so erzeugen, dass wir die Änderungsoperationen ΔI auf dem Originalschema S ausführen. ZS lässt sich durch die Änderungsoperationen ΔS\ΔI zu S' ändern (Der Beweis für die Realisierbarkeit befindet sich im Anhang A). ZS ist strukturell identisch mit I'. Daher kann man die Fragestellung in einer neuen umwandeln: Überprüfung der semantischen Migrierbarkeit einer unverzerrten Instanz I' bei Schemaevolution von ZS zu S'.

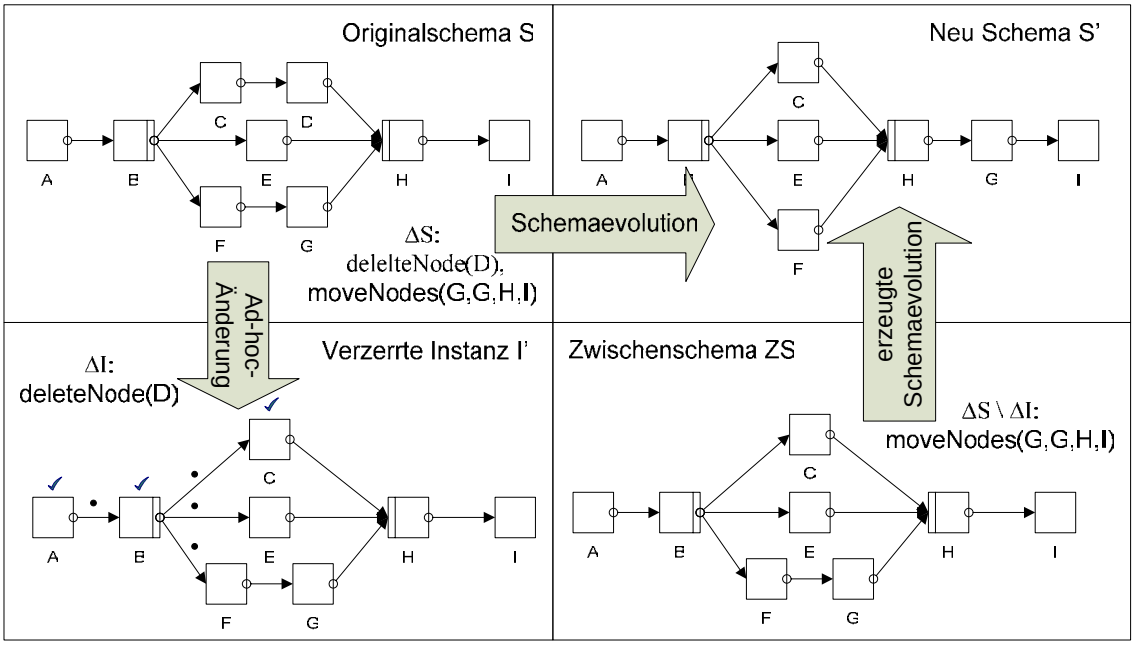


Abb. 3.25 Das Zwischenschema

Die Abbildung 3.25 verdeutlicht das Konzept vom Zwischenschema. ZS ist das Zwischenschema für die verzerrte Instanz I' bei der Schemaevolution von S zu S'. I' ist gegenüber ZS unverzerrt. Sie kann bei der Schemaevolution von ZS zu S' als syntaktisch migrierbare unverzerrte Instanz betrachtet. Daher ist Sie auf S' migrierbar.

Werfen wir noch einen Blick auf die Abbildung 3.24. Es ist jetzt deutlich zu sehen, dass nur disjunkte Operationen semantische Konflikte erzeugen können. Für die Klasse *partially equivalent* ist die semantische Überprüfung auch erforderlich, weil es außer den gemeinsamen Änderungsoperationen noch disjunkte Änderungsoperationen gibt. Mit dieser Klassifizierung können viele Instanzen von der Überprüfung ausgeschlossen werden. Zusammenfassend müssen nach der syntaktischen Überprüfung nur Instanzen der Klassen *disjoint* und *partially equivalent* auf semantische Migrierbarkeit überprüft werden.

3.4.2.3 Problemumstellung für die Klasse partially equivalent

Die semantische Überprüfung für die Klasse *partially equivalent* können wir auf der Überprüfung für die Klasse *disjoint* zurückführen, weil die gemeinsamen Operationen von ΔI und ΔS in der Klasse *partially equivalent* keine Rolle bei der semantischen Überprüfung spielen.

Als ein Beweis dazu, können wir nochmal die Zwischenschema-Methode vom Abschnitt 3.4.2.2 verwenden. Falls ΔG die gemeinsamen Operationen von ΔI und ΔS sind, kann man sie auf das Originalschema S anwenden, um das Zwischenschema ZS zu erzeugen. Entsprechend sollen die ΔG an der Originalinstanz I angewendet werden, um eine Zwischeninstanz ZI zu erzeugen. (Beweis für die Realisierbarkeit im Anhang B). Dann ist das Problem so transformiert: Überprüfung der semantischen Migrierbarkeit von ZI bei Schemaevolution von ZS zu S', wobei die neuen instanzspezifischen Änderungen ΔI' gleich ΔI\ΔG und die neuen Schemaänderungen ΔS' gleich ΔS\ΔG sind (vgl. Abbildung 3.26). Dabei sind die ΔI' und ΔS' für die umgewandelte Fragestellung in der Klasse *disjoint*. Das neuen Schema S' und die verzerrten Instanz I' der neuen Fragestellung sind die gleich wie die von der alten Fragestellung. Deswegen sind die zu überprüfenden semantischen Migrierbarkeiten von den beiden Fragestellungen ebenfalls gleich.

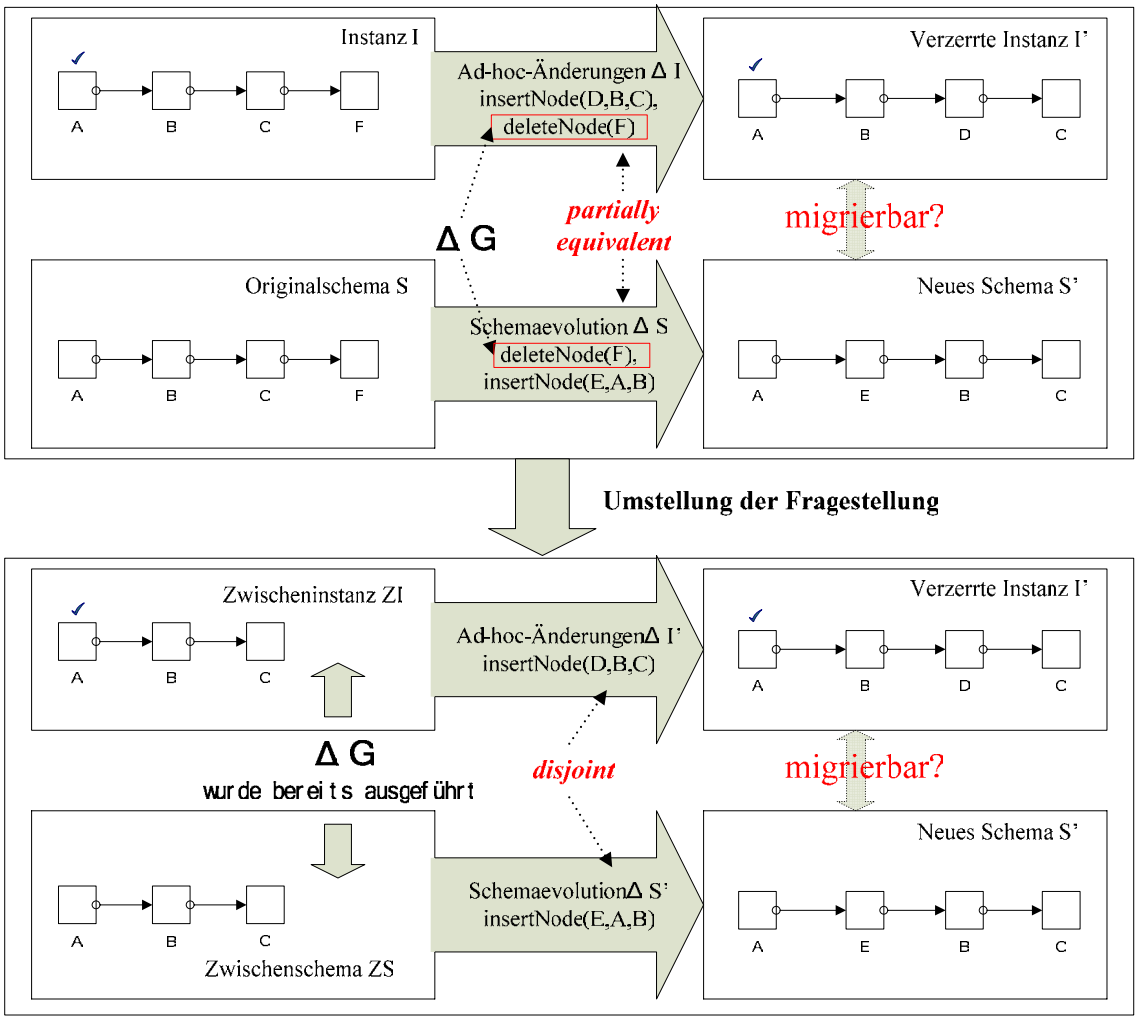


Abb. 3.26 Frageumstellung

Daher können wir die gemeinsamen Änderungsoperationen ΔG ignorieren, und die Überprüfung für die Klasse *partially equivalent* in der Überprüfung für die Klasse *disjoint* transformieren.

3.4.3 Vorgehensweise und Optimierungsmöglichkeiten

In diesen Abschnitt wird die Vorgehensweise der semantischen Überprüfung für die Schemaevolution ermittelt und optimiert.

3.4.3.1 Direkte Überprüfung der virtuellen migrierten Instanz

Da wir bereits eine virtuelle migrierte Instanz als Basis der semantischen Überprüfung erzeugten (in Abschnitt 3.4.2.1), können wir sie semantisch überprüfen. Wenn sie semantisch korrekt ist, ist die entsprechende Instanz migrierbar. Die virtuelle migrierte Instanz zu überprüfen ist nichts anderes als die Überprüfung einer verzerrten Instanz nach Ad-hoc-Änderungen. Wir nehmen das Beispiel aus der Abbildung 3.22.

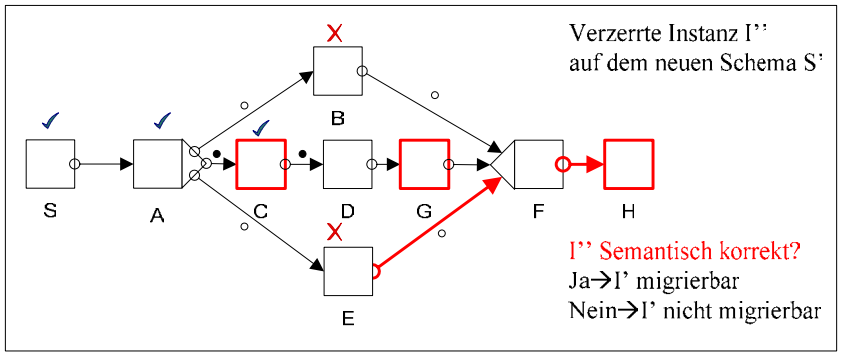


Abb. 3.27 Direkte Überprüfung der virtuellen migrierten Instanz

Bei Instanzmigration wird versucht, die Schemaänderungen ΔS auf die Instanz zu übertragen. Für unser Beispiel aus der Abbildung 3.22 sind sie *deleteNode(K)*, *InsertNode(H, L, G)* und *moveNodes(G, G, D, F)*. So ist das Problem auf Überprüfung für die Ad-hoc-Änderungen an Instanz zurückgeführt.

Wie in Abschnitt 3.2.3 geschrieben, durch eine bestimmte Änderungsoperation können nicht alle sondern nur Constraints von bestimmen Arten verletzt werden (genauere Analyse in Kapitel 5). Deswegen brauchen wir dabei nur die Constraints, die durch ΔS gefährdet werden, zu überprüfen [13].

3.4.3.2 Die Wechselwirkung von ΔS und ΔI

Für die Schemaevolution ist die in Abschnitt 3.4.3.1 vorgestellte Vorgehensweise nicht effizient genug, da alle Constraints, die durch ΔS gefährdet werden, überprüft werden müssen. Jedoch haben wir bei Schemaevolution zusätzliche Bedingungen, die wir ausnutzen können, um den Aufwand zu reduzieren. Wir betrachten nun die Klasse *disjoint*, wobei die Instanz und das Schema schon geändert werden. d.h.:

1. Die instanzspezifischen Änderungen ΔI sind semantisch und syntaktisch korrekt auf der Instanz I.

2. Die Änderungsoperationen ΔS sind semantisch und syntaktisch korrekt auf dem Schema S.

Und wir müssen nur die syntaktisch migrierbare Instanzen überprüfen, d.h.:

3. ΔS sind syntaktisch korrekt an der unverzerrten Instanz I. Wegen besserer semantischer Toleranz von Instanz, sind ΔS auch semantisch korrekt an der unveränderten Instanz I.

Bei der Migration wird versucht, die Schemaänderung auf die verzerrte Instanz I' zu übertragen. Für die Klasse *disjoint* bedeutet das, ΔS an I' anzuwenden. Jetzt wissen wir schon, dass sowohl ΔS und ΔI korrekt an I sind. D.h. wenn man allein ΔS oder ΔI an I ausführen, wird kein Constraint verletzt. ΔS und ΔI müssen nicht mehr wie bei der Vorgehensweise in Abschnitt 3.4.3.1 überprüft werden. Aber wenn beide ΔS und ΔI an I ausgeführt werden, kann semantische Konflikten dadurch erzeugt werden [13].

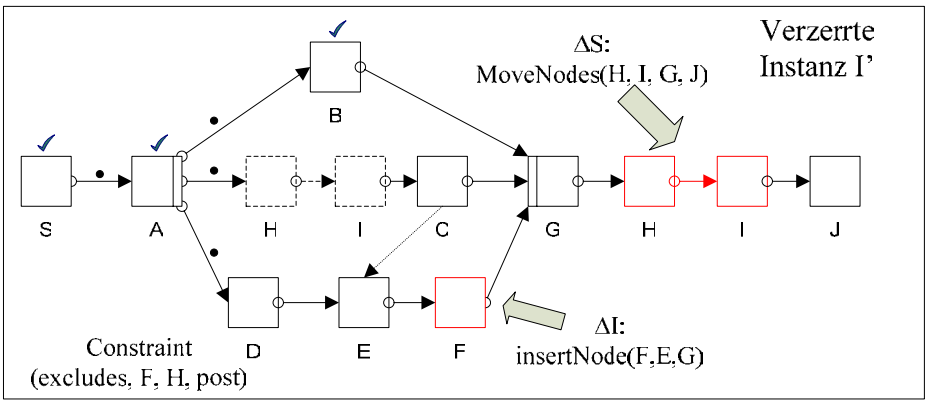


Abb. 3.28 Die Wechselwirkung von ΔS und ΔI

Die Abbildung 3.28 illustriert ein Beispiel. Die ΔI und ΔS in der Abbildung sind beide separat semantisch richtig an der Instanz. Aber wenn sie zusammen ausgeführt werden, wird das Constraint *(excludes, F, H, post)* verletzt. Das nennen wir die Wechselwirkung von ΔI und ΔS.

Wir müssen jetzt nur die Constraints überprüfen, die durch die Wechselwirkung verletzt werden können. Wir werden es in Kapitel 5 ermitteln, genau welche Constraints durch die Wechselwirkung verletzt werden können. Es ist aber klar, dass diese Constraints eine Teilmenge von der Schnittmenge von den Constraints, die durch ΔI an der Originalinstanz gefährdet und auch eine Teilmenge von den Constraints, die durch ΔS auf dem Originalschema gefährdet sind [13].

Formal:
C(ΔI) : alle Constraints, die durch ΔI an der Originalinstanz verletzt werden können.
C(ΔS) : alle Constraints, die durch ΔS auf dem Originalschema verletzt werden können.
C(W) : alle Constraints, die durch die Wechselwirkung von ΔI und ΔS an der virtuellen migrierten Instanz verletzt werden können

$$C(W) \subseteq C(\Delta I) \cap C(\Delta S)$$

Mit der Kontrollierung der Wechselwirkung kann man den Aufwand der Überprüfung reduzieren. Um die Verbesserung zu verdeutlichen, verwenden wir nochmal das Beispiel in Abbildung 3.28, jedoch mit zusätzlichen Constraints.

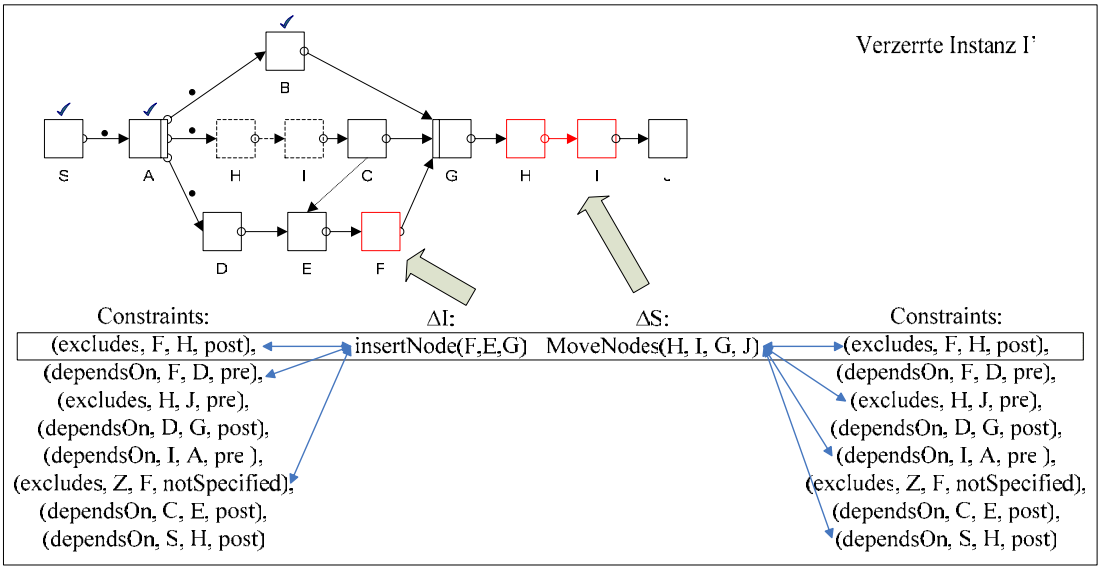


Abb. 3.29 Verbesserung durch Analyse der Wechselwirkung

Für das Beispiel von Abbildung 3.29 müssen ohne Analyse der Wechselwirkung vier Constraints überprüft werden: |C(ΔS)| = 4. Mit Analyse der Wechselwirkung brauchen wir dagegen nur das gemeinsame Constraint (*excludes, F, H, post)* zu überprüfen (|C(W)| = 1).

3.5 Zusammenfassung

In diesem Kapitel haben wir alle vier Verifikationsszenarien eines adaptiven PMS analysiert. Dabei wurden naive Vorgehensweisen für die semantische Überprüfung und mögliche Optimierungsstrategien ermittelt. Es wird bislang zwei Arten von Optimierungsmöglichkeiten genannt [13]:

1. Reduzierung der zu überprüfen Constraints.
2. Verbesserung des Verfahrens zur semantischen Überprüfung. (vgl. Abbildung 3.30)

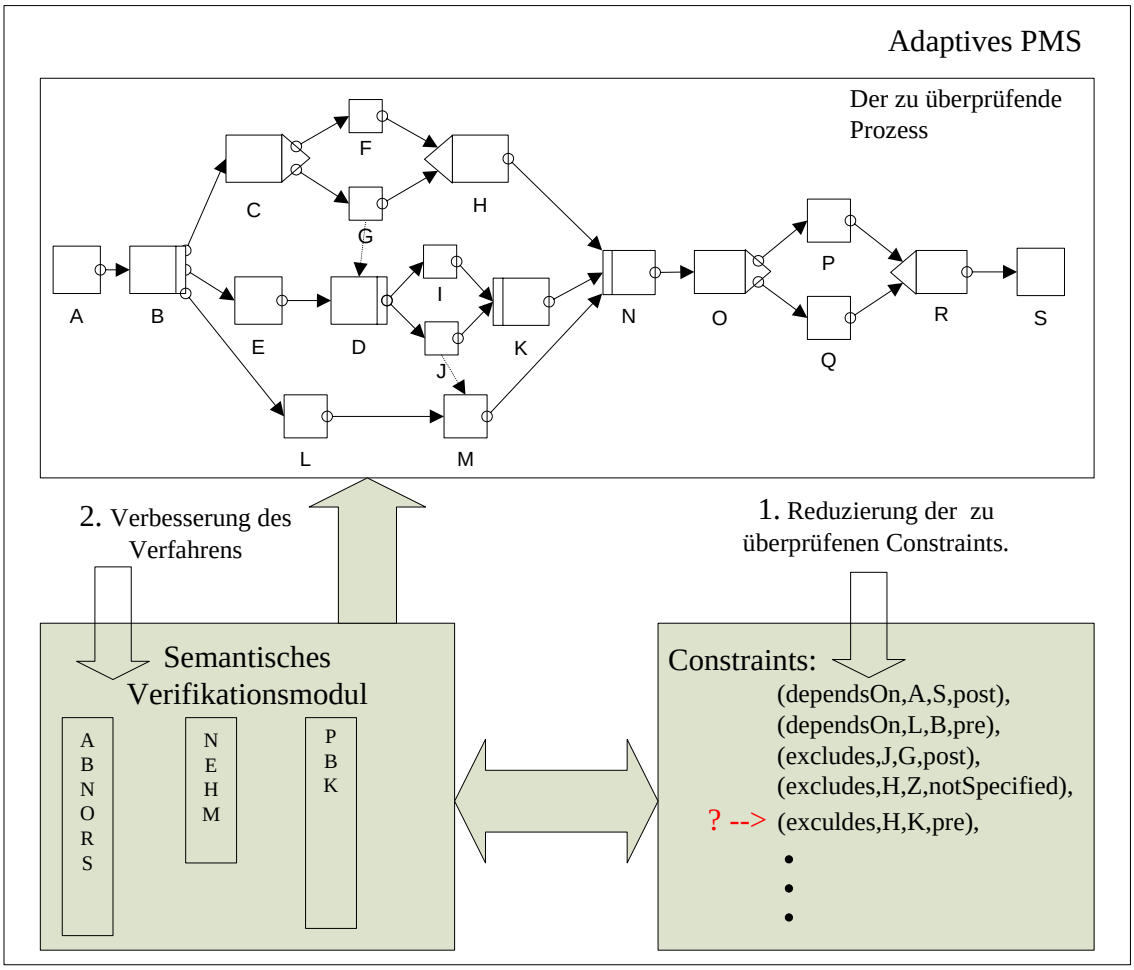


Abb. 3.30 Optimierungsmöglichkeit

In Kapitel 4 wird ein optimiertes Überprüfungsverfahren vorgestellt. Und in Kapitel 5 werden wir versuchen, die zu überprüfend Constraints einzuschränken. Wir werden die Constraints, die durch eine bestimmte Änderungsoperation gefährdet werden, identifizieren. Für die Schemaevolution werden wir die Wechselwirkung von Schemaänderungen und Instanzänderungen analysieren.

Kapitel 4

Ansätze zur semantischen Verifikation

In diesem Kapitel werden wir die elegante Vorgehensweise für die semantische Überprüfung kennen lernen. Dabei beschäftigen wir uns nur mit den Prozessen, in denen jede Aktivität höchstens einmal auftauchen kann. Wir werden zuerst eine naive Vorgehensweise für die serielle Struktur betrachten, um die Grundidee des Algorithmus zu verstehen. Ausgehend davon wird ein optimierter Ansatz Schritt für Schritt anhand von Beispielen hergeleitet. Danach werden wir den Algorithmus an Block-Strukturen anpassen, damit er allgemein einsetzbar wird.

4.1 Grundidee und naive Vorgehensweise

Die semantische Überprüfung entspricht im Wesentlichen der Suche nach bestimmten Verhältnissen oder Reihenfolgen von Aktivitäten in einem Prozess. Für die Abhängigkeitsbeziehung muss die in dem Constraint definierte Reihenfolge gefunden werden, z.B. für das Constraint *(dependsOn, A, B, post)* muss die Aktivität B nach A gefunden werden. Für die Ausschlussbeziehung ist die Suche analog, nur soll hier die im Constraint spezifizierte Reihenfolge nicht vorkommen. Ansonsten liegt ein semantischer Konflikt vor.

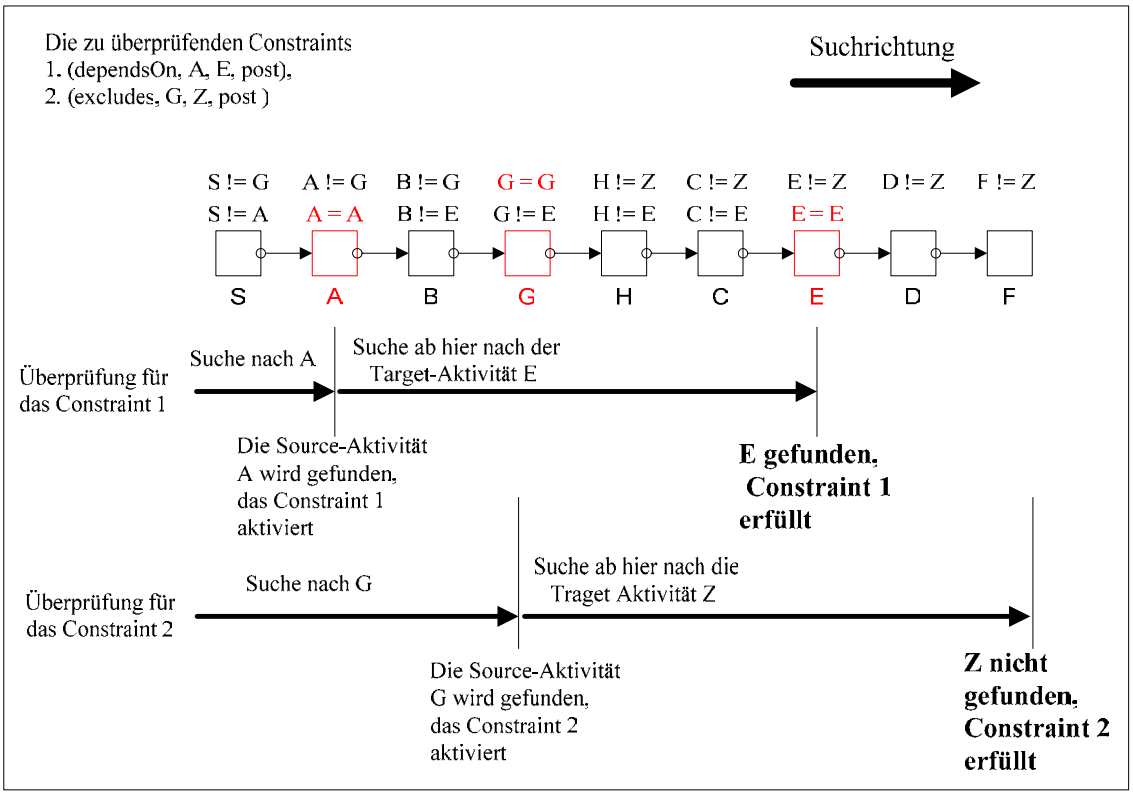


Abb. 4.1 Die Grundidee des Lösungsansatzes

Basierend auf der oben geschriebenen Grundidee können wir einen naiven Algorithmus für die semantische Überprüfung entwerfen. Dabei wird jedes Constraint separat überprüft. Wir suchen zuerst nach der Source-Aktivität in dem Prozess. Dazu vergleichen wir die Aktivitäten im Prozess mit der

Source-Aktivität. Wenn die Source-Aktivität nicht vorkommt, gilt das Constraint als erfüllt. Dann ist es unnötig nach der Target-Aktivität zu suchen. Wenn die Source-Aktivität gefunden wird, gilt das Constraint als aktiviert. Ab dann muss nach der Target-Aktivität gesucht werden. Die Abbildung 4.1 zeigt die semantische Überprüfung für zwei Constraints mit der naiven Vorgehensweise. In dem Beispiel sind die beiden Constraints erfüllt. Bei der naiven Vorgehensweise muss für jedes Constraint den Prozess mindestens einmal durchlaufen. Das ist für den praktischen Einsatz zu aufwendig. In Abbildung 4.1 wird deutlich, dass die Vergleiche der Aktivitäten gleichzeitig in einem Durchlauf gemacht werden können. Die erste Verbesserung der Vorgehensweise beruht auf dieser Überlegung.

4.2 Gekoppelte Überprüfung von Constraints

Wir betrachten zuerst nur die Constraints, deren *Position* gleich *post* ist. Wir beschränken uns dabei zunächst auf Prozesse mit rein sequentieller Struktur, d.h. Prozesse ohne Verzweigungen. Für andere Constraints und Strukturen werden wir den Algorithmus später noch vervollständigen.

4.2.1 Grundidee

Wir wollen nun nicht für jedes Constraint den Prozess einmal durchlaufen sondern die Überprüfung mit möglichst wenigen Durchläufen machen, damit der Aufwand für die Verifikation möglichst klein ist. Wir werden die zu suchenden Aktivitäten in Mengen hinzufügen und die Suche in einem Durchlauf integrieren. Für die Übersichtlichkeit der Beispiele werden wir weiterhin die Mengen in Beispielen als Listen darstellen.

Wir erzeugen eine Menge L1 für die Source-Aktivitäten aller zu überprüfenden Constraints. Wie man die Menge der zu überprüfenden Constraints (d.h. potentiell verletzten Constraints) bestimmt, wird in Kapitel 5 vorgestellt. Bei der Überprüfung werden die Aktivitäten im Prozess mit den Elementen in der Menge L1 verglichen. Wenn eine Source-Aktivität vorkommt, sollen die entsprechenden Constraints aktiviert und die Source-Aktivität aus der Menge gelöscht werden. Danach suchen wir nach der Target-Aktivität. Da die Abhängigkeitsbeziehung und die Ausschlussbeziehung unterschiedlich behandelt werden sollen, fügen wir die zu suchenden Target-Aktivitäten der Abhängigkeitsbeziehung in die Menge L2 und die von der Ausschlussbeziehung in die Menge L3. Ab der zweiten Aktivität im Prozess vergleichen wir sie mit den Elementen von Menge L2 und Menge L3. Wenn die Target-Aktivität im Prozessdurchlauf gefunden wird, wird sie sofort aus der Menge gelöscht. Für die Abhängigkeitsbeziehung gilt dann das entsprechende Constraint als erfüllt. Für die Ausschlussbeziehung ist dagegen ein semantischer Konflikt gefunden. D.h. die Menge L1, L2 und L3 werden bei der Überprüfung dynamisch modifiziert.

Wenn nach dem ganzen Durchlauf die Menge L2 nicht leer ist, bedeutet dies, dass es Target-Aktivität(en) von Abhängigkeitsbeziehungen gibt, die nicht in dem gewünschten Bereich vom Prozess aufgetaucht ist (sind). Die entsprechenden Constraints sind dann verletzt. Abbildung 4.2 illustriert die Überprüfung mit Mengen. Bei der Überprüfung der Aktivität A wird E als Target-Aktivität vom Constraint 1 in L2 eingefügt. Sie wird danach gefunden und aus L2 gelöscht. Daher ist das Constraint 1 erfüllt. Das Constraint 3 ist dagegen verletzt, weil D in L3 gefunden wird. Da Constraint 3 für Ausschlussbeziehung steht.

Für einen semantisch korrekten Prozess soll die Menge L2 nach der Überprüfung leer sein, d.h. alle Target-Aktivitäten von den aktivierten Abhängigkeitsbeziehungen sollen gefunden werden. Und die Menge L3 soll nicht bei der Überprüfung verkleinert werden, d.h. keine Target-Aktivität von den aktivierten Ausschlussbeziehungen soll im Prozess auftauchen.

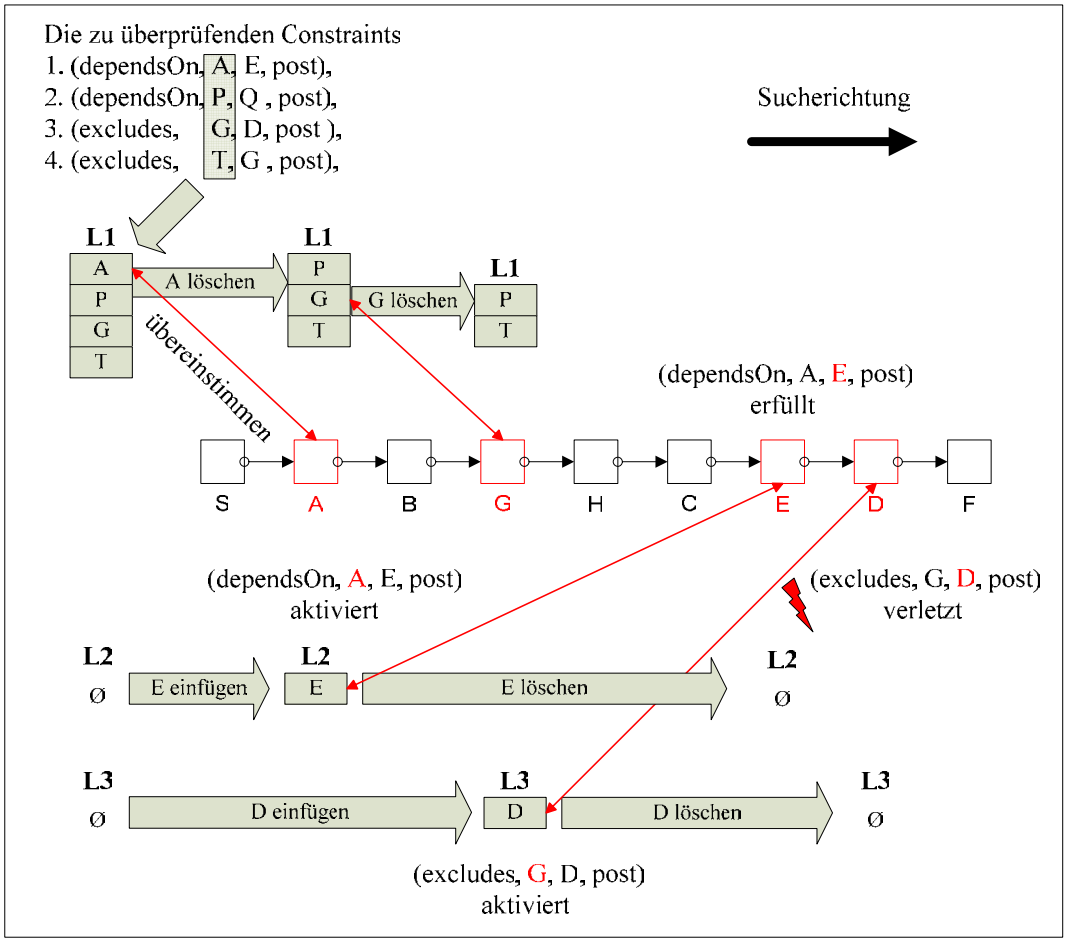


Abb. 4.2 Integrierte Überprüfung mit Menge

Als Optimierung kann man die Menge der zu suchenden Source-Aktivitäten verringern, indem eine Menge L1_MIN so erzeugen wird: Eine Menge L1_ALL bei der Schemamodellierung erzeugen, die alle Source-Aktivitäten irgendeines Constraints, die tatsächlich im Prozess vorkommen, enthält. Und vor der Überprüfung erzeugen wir die Menge L1_MIN als Schnittmenge von L1_ALL und L1.

$$L1\_MIN = L1\_ALL \cap L1$$

Dadurch bleiben nur die Source-Aktivitäten, die tatsächlich in Prozesse vorkommen und sich in einen zu überprüfende Constraint befinden, in L1_MIN. Die im Prozess nicht vorkommenden Source-Aktivitäten werden dadurch ausgeschlossen. Statt L1 vergleichen wir jetzt die Aktivitäten beim Prozessdurchlauf mit den Elementen in der Menge L1_MIN, um die Source-Aktivität zu finden. D.h. die Suche nach Source-Aktivitäten, die ohnehin nicht im Prozess vorkommen, wird gespart. Die entsprechenden Constraints, deren Source-Aktivitäten nicht im Prozess vorkommen, gelten per Definition als erfüllt (siehe Abschnitt 2.2). Sie müssen daher nicht mehr überprüft werden.

Für unser Beispiel in Abbildung 4.2, würde L1_MIN zwei Elemente weniger als L1 enthalten: die Aktivitäten P und T. Sie kommen nicht in dem Prozess damit auch nicht in der Menge L1_ALL vor. Das Constraint *(dependsOn, P, Q, post)*, das von der Änderungsoperation *deleteNode(Q)* gefährdet werden könnte, und das Constraint *(excludes, T, G, post)*, das von der Änderungsoperation *insertNode(G, B, H)* gefährdet werden könnte, sind automatisch erfüllt. Durch Einsatz der Menge L1_ALL kann die Überprüfung dieser beiden Constraints gespart werden.

Beim Einfügen und Löschen von semantisch relevanten Aktivitäten (d.h. die Aktivitäten, die Source-Aktivität oder Target-Aktivität irgendeines Constraints sind.) muss die Menge L1_ALL entsprechend geändert werden. Bei den Löschoperationen suchen wir die gelöschten Aktivitäten in der Menge L1_ALL.

Wenn sie in der Menge ist, löschen wir sie gleichzeitig aus der Menge. Für die Einfügeoperationen bei Schemaänderungen, überprüfen wir, ob die eingefügten Aktivitäten semantisch relevant und Source-Aktivitäten sind. Wenn ja und sie nicht in der Menge L1_ALL sind, fügen wir sie in der Menge hinzu. Für die Ad-hoc-Änderungen an der Instanz können wir einfach die eingefügten Aktivitäten in L1_ALL hinzufügen, ohne sie zu identifizieren. Die semantisch irrelevanten Aktivitäten und die Aktivitäten, die keine Source-Aktivitäten sind, werden bei Erzeugung der Menge L1_MIN als die Schnittmenge von L1_ALL und L1 ausgeschlossen.

Mit Einsetzung von L1_MIN wird der Überprüfungsaufwand reduziert. Und wir können sicherstellen, dass die Source-Aktivitäten der zu überprüfenden Constraints auf jedem Fall im Prozess vorkommen werden. Diese Voraussetzung ist nützlich für den Überprüfungsalgorithmus. Als Nachteil muss man zur Laufzeit die L1_ALL pflegen und die L1 erzeugen. Jedoch ist dieser Aufwand relativ klein.

4.2.2 Rückwärtssuche

Wir haben bislang nur die Constraints überprüft, deren Parameter *Position* gleich *post* sind. Dies liegt darin: Wir laufen den Prozess bisher immer vorwärts durch. Wenn wir alle Constraints in einem Durchlauf überprüfen, werden Konflikte wegen der Suchrichtung vorkommen (vgl. Abbildung 4.3).

Für die Constraints, deren Parameter *Position* gleich *post* sind, ist die Target-Aktivität nach der Source-Aktivität zu suchen. Für die Constraints, deren Parameter *Position* gleich *pre* sind, ist dagegen die Menge der Vorgänger der Source-Aktivität der Suchbereich. Für die Constraints, deren Parameter *Position* gleich *notSpecified* sind, sind die beiden Suchrichtungen relevant.

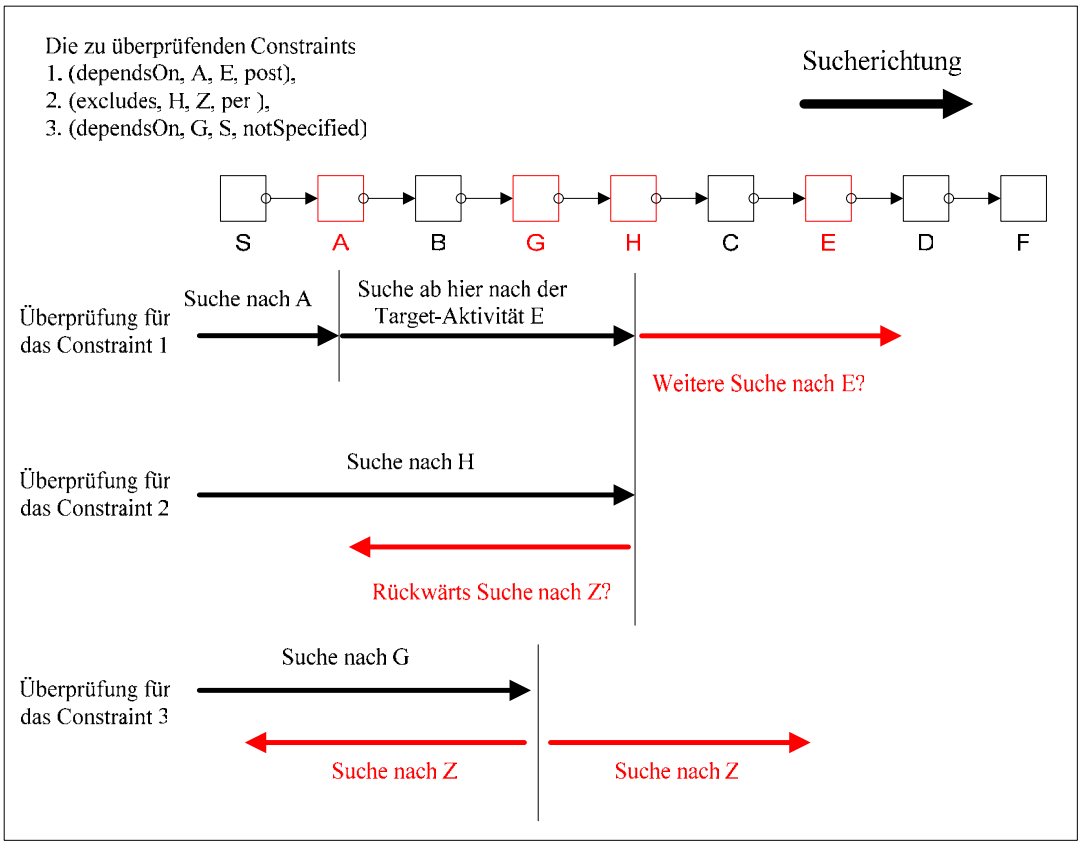


Abb. 4.3 Konflikt wegen der Suchrichtung

Wir können die Constraints, deren Parameter *Position* gleich *pre* sind, getrennt von den Constraints, deren Parameter *Position* gleich *post* sind, überprüfen. Wir können für sie den Prozess einmal rückwärts durchlaufen. Dabei ist das Suchverfahren gleich (vgl. Abbildung 4.4).

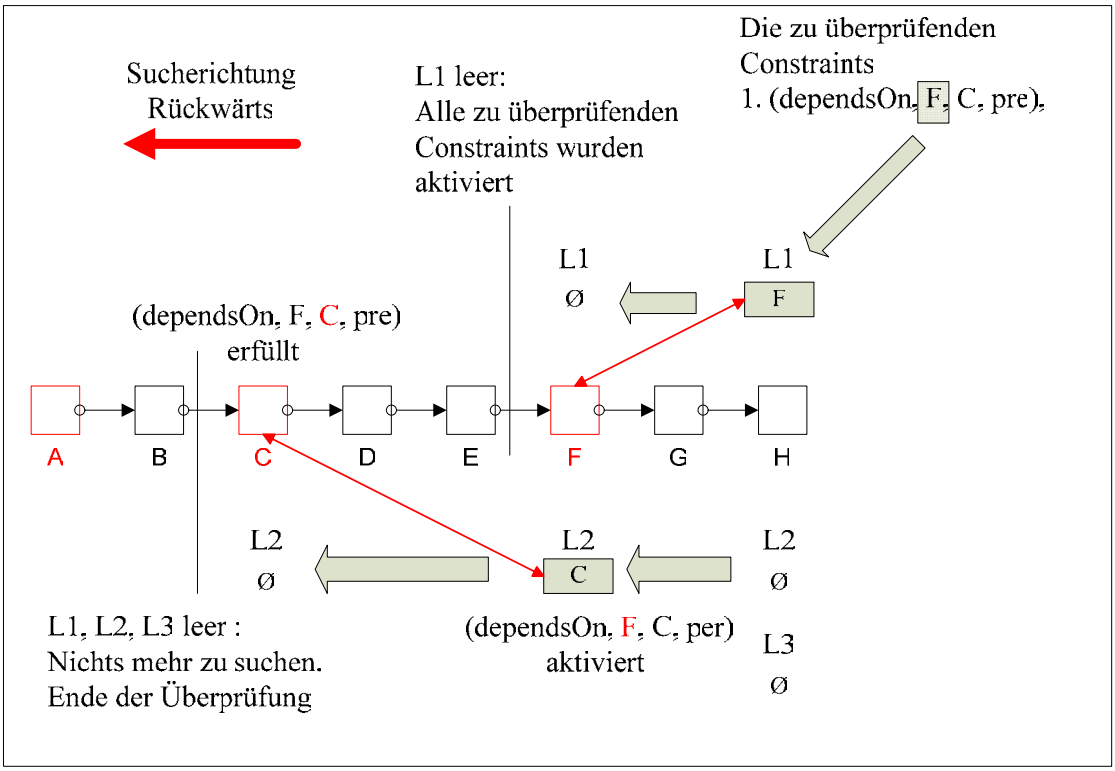


Abb. 4.4 Rückwärtssuche

Bei Rückwärtssuche wird zuerst die letzte Aktivität mit den Elementen in L1_MIN verglichen. Dann gehen wir zu der vorletzten Aktivität. Ab der vorletzten Aktivität vergleichen wir sie auch mit den Elementen von L2 und L3. Die Änderungen von den Mengen L2 und L3 sind identisch wie bei Vorwärtssuche. In dem Beispiel aus der Abbildung 4.4 wird die Überprüfung nicht bis zu der ersten Aktivität durchgeführt sondern bis die Aktivität C, weil da alle drei Mengen schon leer sind und es nichts mehr zu suchen gibt.

4.2.3 Überprüfung in beiden Richtungen

In Abbildung 4.3 haben wir bereits gesehen, dass für Constraints, deren Parameter *Position* gleich *notSpecified* sind, die beiden Suchrichtungen relevant sind. Diese Constraints können wir sowohl bei Vorwärtssuche als auch bei Rückwärtssuche überprüfen. Wir überprüfen sie zuerst wie Constraints mit *Position post* bei Vorwärtssuche, dann eventuell nochmal wie Constraints mit *Position pre* bei Rückwärtssuche. Die umgekehrte Reihenfolge, d.h. zuerst Überprüfung wie Constraints mit *Position pre* bei Rückwärtssuche und dann wie Constraints mit *Position post* bei Vorwärtssuche, ist auch richtig.

Manchmal reicht für die Überprüfung dieser Constraints die Suche in einer Richtung. Für den Fall, dass die Source-Aktivität bei der Suche in einer Richtung nicht gefunden wird, sollen die Constraint bei der Suche in der zweiten Richtung ignoriert werden, weil die Source-Aktivität da auch nicht vorkommen wird. Für die Abhängigkeitsbeziehungen sind die Constraints bereits erfüllt, wenn die Target-Aktivitäten bei der Suche in einer Richtung gefunden werden. Für die Ausschlussbeziehungen sind die Constraints verletzt, wenn die Target-Aktivitäten bei der Suche in einer Richtung gefunden werden. Für diesen Fall brauchen wir die Constraints nicht mehr bei der Suche in der zweiten Richtung nochmal zu überprüfen.

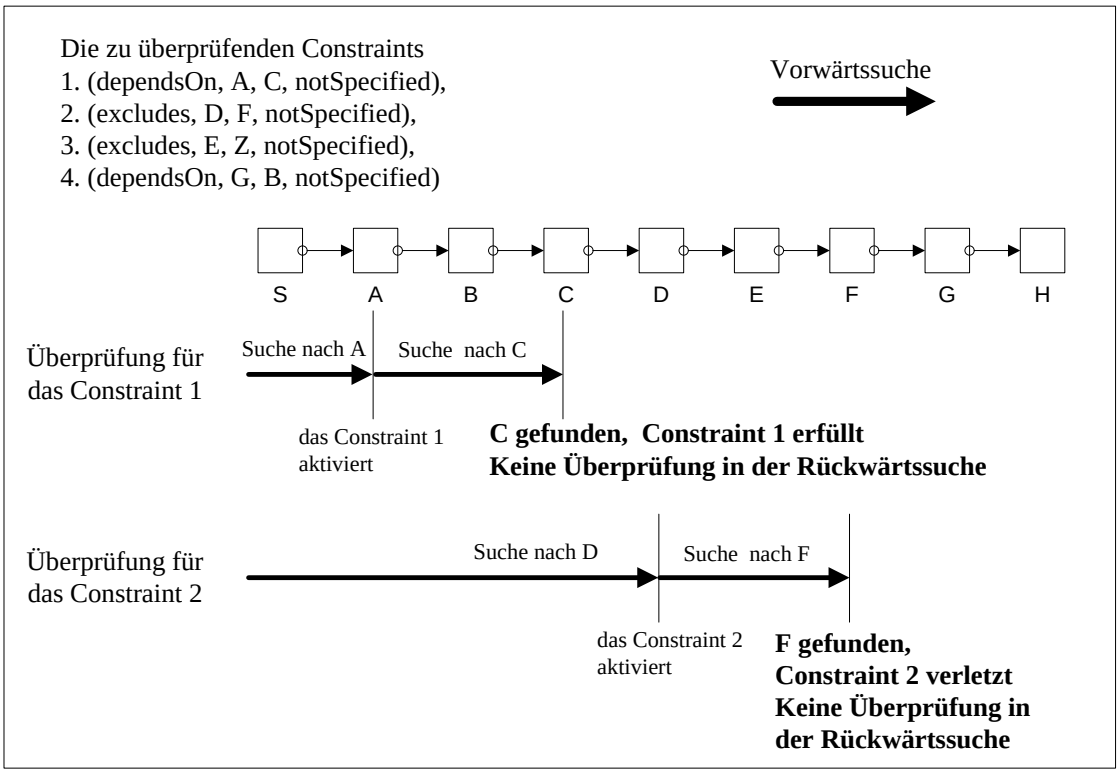


Abb. 4.5 Suche in der ersten Richtung

Für die Überprüfung von den Constraints 1 und 2 in Abbildung 4.5 reicht schon die Vorwärtssuche aus. Dagegen müssen wir die Constraints 3 und 4 in beiden Suchrichtungen überprüfen, weil die Target-Aktivitäten nicht durch Vorwärtssuche gefunden werden können (vgl. Abbildung 4.6).

Jetzt stellen wir uns die Frage: Wenn wir das Konzept der Menge L1_ALL einsetzen, werden alle Elemente / Source-Aktivitäten in der Menge L1_MIN im Prozess vorkommen. D.h. die zu überprüfenden Constraints werden früh oder später alle aktiviert. Wieso suchen wir nicht direkt nach den Target-Aktivitäten von Constraints mit *Position notSpecified*, ohne nach den Source-Aktivitäten zu suchen, wenn die Reihenfolge der beiden Aktivitäten unwichtig ist? Dies liegt darin: Für Prozesse mit nur serieller Struktur, wie die Prozesse in bisherigen Beispielen, wäre das OK, da jede Aktivität ausgeführt wird. Aber für Prozesse mit Verzweigungen muss man noch zusätzlich die Ausführungen von den Aktivitäten kontrollieren. Darauf werden wir in Abschnitt 4.3 genauer eingehen.

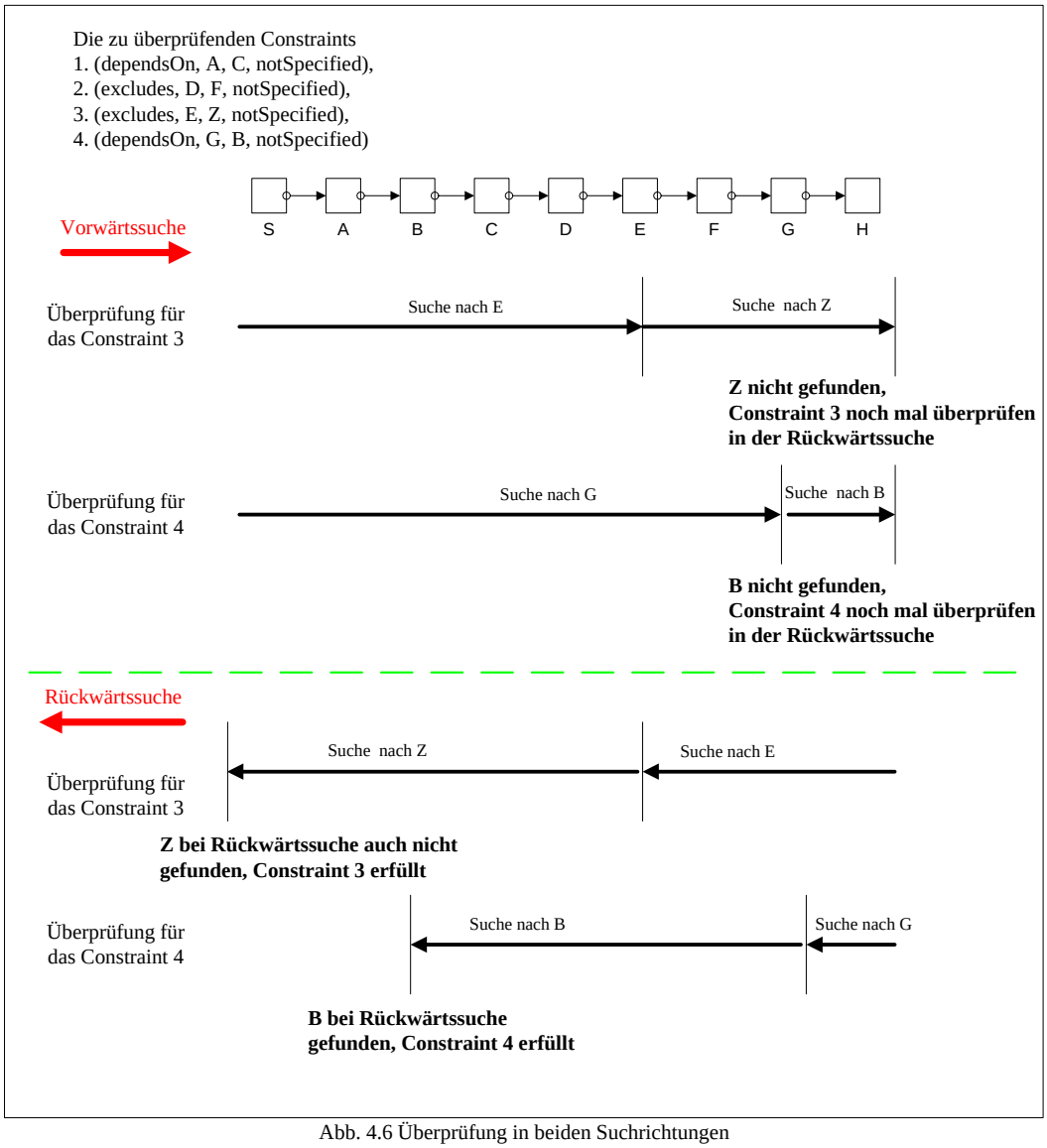


Abb. 4.6 Überprüfung in beiden Suchrichtungen

4.3 Die fortgeschrittene Vorgehensweise

Mit bisheriger Vorgehensweise brauchen wir zwei Durchläufe des Prozesses um alle Constraints überprüfen zu können. Jedoch mit weiterer Entwicklung des Algorithmus können wir alle Constraints in einem Durchlauf des Prozesses überprüfen. Das Problem der Suchrichtung (siehe Abschnitt 4.2.2) können wir dadurch lösen, dass wir für die Constraints mit *Position* gleich *pre* oder *notSpecified* in einem Durchlauf gleichzeitig nach den Source-Aktivitäten und den Target-Aktivitäten suchen. Weil wir das Konzept der Menge L1_MIN einsetzen, gibt es keine Target-Aktivitäten, die nicht gesucht werden müssen. Eventuell können wir sogar allein mit den zuerst gefundenen Target-Aktivitäten eine Aussage über das Constraint auf dem Prozess machen. Deswegen ist diese Änderung keine Verschlechtung des Überprüfungsalgorithmus.

Für die neue Vorgehensweise verwenden wir wie vorher drei Mengen: L1 für die Source-Aktivitäten, L2 für die Target-Aktivitäten der Constraints mit *Type = dependsOn*, L3 für die Target-Aktivitäten der Constraints mit *Type = excludes*. Jedoch werden die Aktivitäten einer Menge nicht mehr dynamisch geändert. Alle Source-Aktivitäten und Target-Aktivitäten werden bei Initialisierung in der entsprechenden Menge eingefügt, d.h. wir erzeugen drei vollständigen Mengen für alle zu überprüfenden Constraints (vgl. Abbildung 4.7).

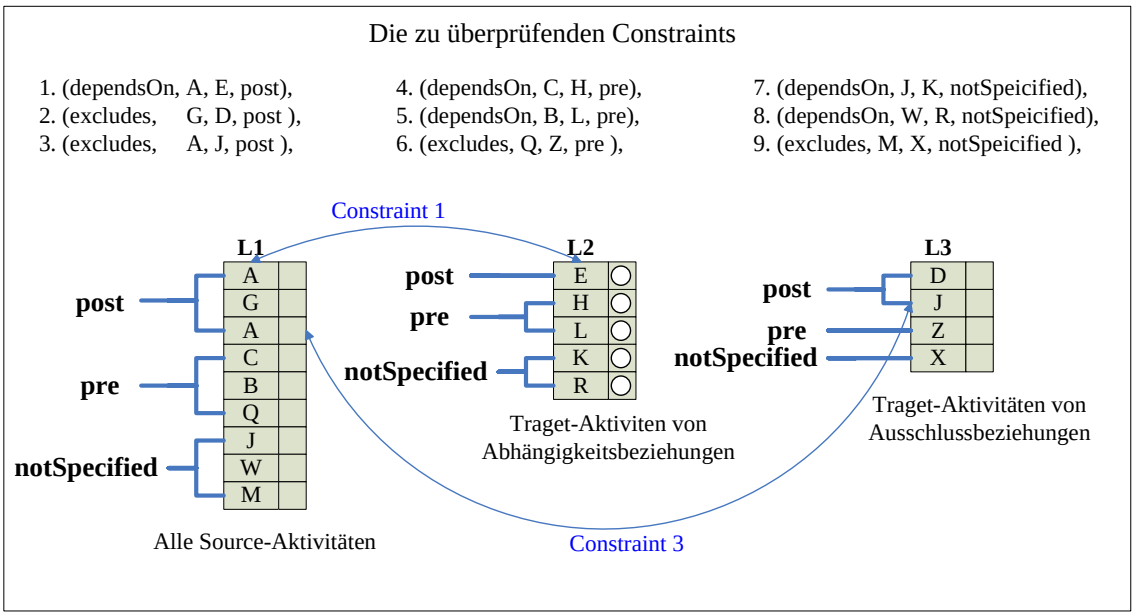


Abb. 4.7 Der Ausgangszustand der drei Mengen

In Abbildung 4.7 wird veranschaulicht, dass jede Menge aus 3 Teilen besteht. Wir verwenden für die Constraints mit unterschiedlicher *Position post, pre oder notSpecified* unterschiedliche Überprüfungskonzepte. Wenn eine Aktivität in mehreren Constraints vorkommt, soll sie entsprechend in der Menge auch mehrmals vorkommen. Es muss möglich sein, dieselbe Aktivität für unterschiedliche Constraints zu unterscheiden. In Abbildung 4.7 kommt die Aktivität A beispielsweise für Constraint 1 und 3 zwei Mal in L1 vor.

Für jede Aktivität in der Menge gibt es Speicherplätze für Markierungen. Bei der Überprüfung werden wir nun statt der Aktivitäten diese Markierungen dynamisch ändern, um den Zustand eines Constraints auf dem Prozess bei der Überprüfung zu zeigen. Als Ausgangszustand sind die Target-Aktivitäten in L2 bei Initialisierung schon markiert. Das dient der einheitlichen Feststellung von semantischen Konflikten. Wenn ein Constraints der Abhängigkeitsbeziehung bei der Überprüfung als erfüllt festgestellt wird, wird die Markierung für die Target-Aktivität gelöscht. Am Ende der semantischen Überprüfung wollen wir den Zustand erreichen, dass sowohl in L2 als auch in L3 nur die Target-Aktivitäten eines verletzten Constraints eine Markierung besitzen. Wir verwenden diese Markierungen, um festzustellen, ob ein Constraint auf dem Prozess verletzt ist oder nicht. Deswegen nennen wir sie die Verletzungsmarkierung. So kann man den Ausgangszustand von L2 und L3 verstehen: Vor der Überprüfung ist keine Traget-Aktivität gefunden, daher sind alle Constraints der Abhängigkeitsbeziehung zu dem Zeitpunkt „temporär“ verletzt. Entsprechend sind die Target-Aktivitäten in L2 mit der Verletzungsmarkierung markiert. Dagegen sind alle Constraints der Ausschlussbeziehung mit nicht vorgekommen Target-Aktivitäten „temporär“ erfüllt. Bei der Überprüfung werden wir die Zustände von L2 und L3 Schritt für Schritt zum richtigen Überprüfungsergebnis ändern.

Für ein einheitliches Überprüfungsergebnis, nämlich die Markierungszustände von L2 und L3 nach der Überprüfung, ist die Verwendung von L1_MIN erforderlich. Ansonsten werden die Constraints der Abhängigkeitsbeziehung als verletzt festgestellt, wenn weder die Source-Aktiviät noch die Target-Aktivität im Prozess vorkommen. Da die entsprechende Verletzungsmarkierung der Target-Aktivitäten in L2 nicht gelöscht werden, wenn die Target-Aktivitäten bei der Überprüfung nicht gefunden wird. Allerdings werden alle Source-Aktivitäten, die im Prozess vorkommen, bei der Überprüfung markiert. Ohne L1_MIN können wir am Ende der Überprüfung die Verletzungsmarkierungen der Target-Aktivitäten in L2 und L3 löschen, wenn die entsprechenden Source-Aktivitäten in L1 nicht markiert sind. Nach dieser Nachbearbeitung wäre das Überprüfungsergebnis wieder korrekt.

Die auf ADEPT basierenden Prozesse lassen sich in drei Gruppen mit steigender Überprüfungsschwierigkeit einteilen: die serielle Struktur, einfache Blöcke und gemischt verschachtelte

Blöcke. Mit einfachen Block meinen wir einen einzelnen, nicht verschachtelten XOR oder UND-Block. Zur Gruppe gemischt verschachtelte Blöcke gehören alle anderen Block-Strukturen, die komplexer sind (vgl. Abbildung 4.8).

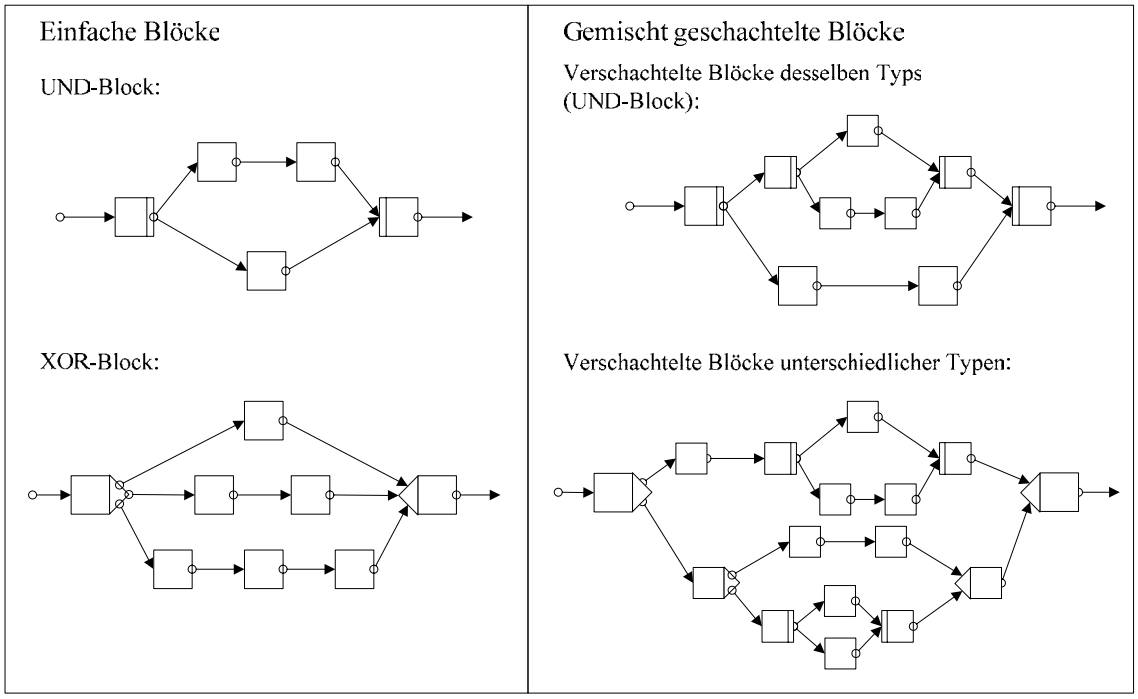


Abb. 4.8 Einfache Blöcke und gemischt verschachtelte Blöcke

Entsprechend haben wir ein hierarchisches System von Algorithmen und Markierungen mit drei Stufen entwickelt. Damit können wir alle Constraints mit einmal vorwärts Durchlaufen des Prozesses überprüfen.

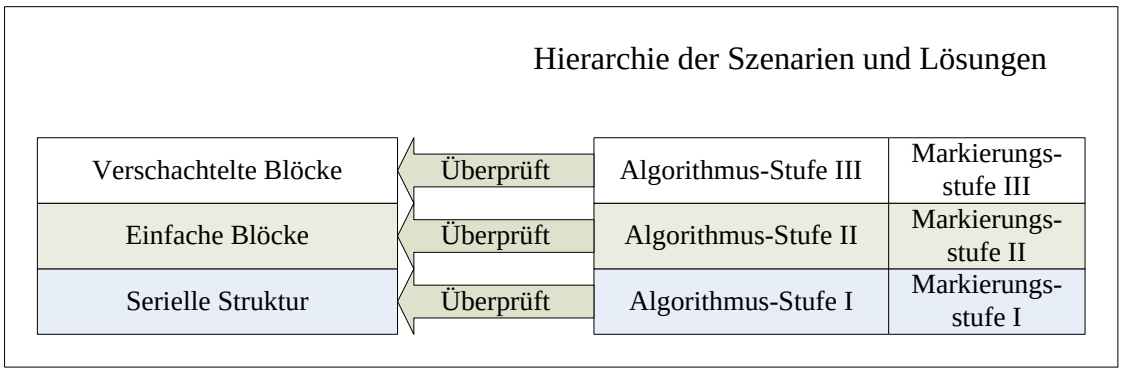


Abb. 4.9 Die Stufen des Lösungssystem

Wie Abbildung 4.9 zeigt, bauen die drei Stufen aufeinander auf. Der Algorithmus oberer Stufe enthält die Logik des Algorithmus unterer Stufe. Und die Markierungen werden mit Steigerung der Stufe immer komplexer. Die Markierung von Stufe I besteht lediglich aus der Verletzungsmarkierung. Für die Überprüfung serieller Struktur reicht dies schon aus. Bei der semantischen Überprüfung eines Prozesses werden wir die Algorithmus abwechselnd für entsprechende Struktur verwenden (vgl. Abbildung 4.10).

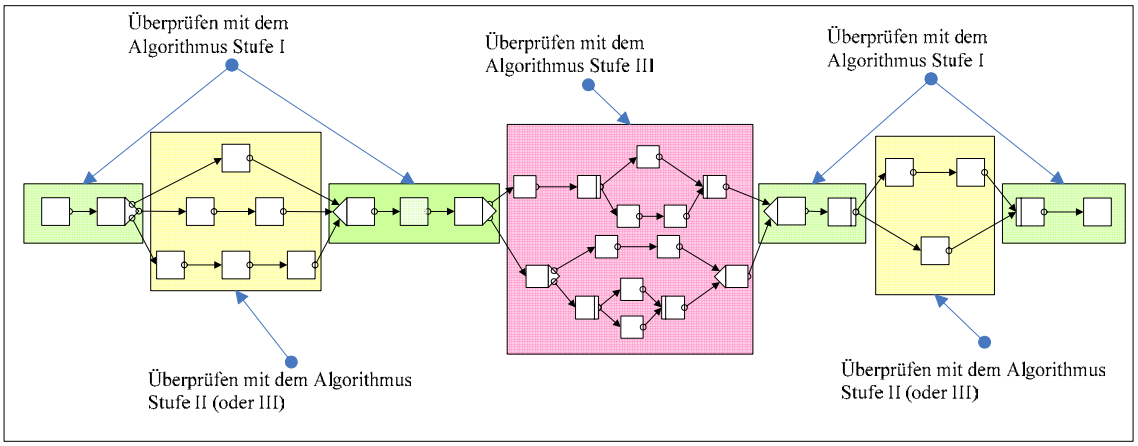


Abb. 4.10 Verwendung der Algorithmen

Allerdings basieren die Algorithmen der drei Stufen auf denselben Mengen L1, L2 und L3. Wir werden den Algorithmus Stufe für Stufe ausbauen. Weiterhin werden wir die Abkürzungen S-A für Source-Aktivität und T-A für Target-Aktivität verwenden.

4.3.1 Überprüfung für serielle Strukturen

Die semantische Korrektheit der seriellen Struktur ist relativ einfach zu bestimmen. Wir betrachten zuerst die möglichen Verhältnisse zwischen S-A und T-A in seriellen Strukturen.

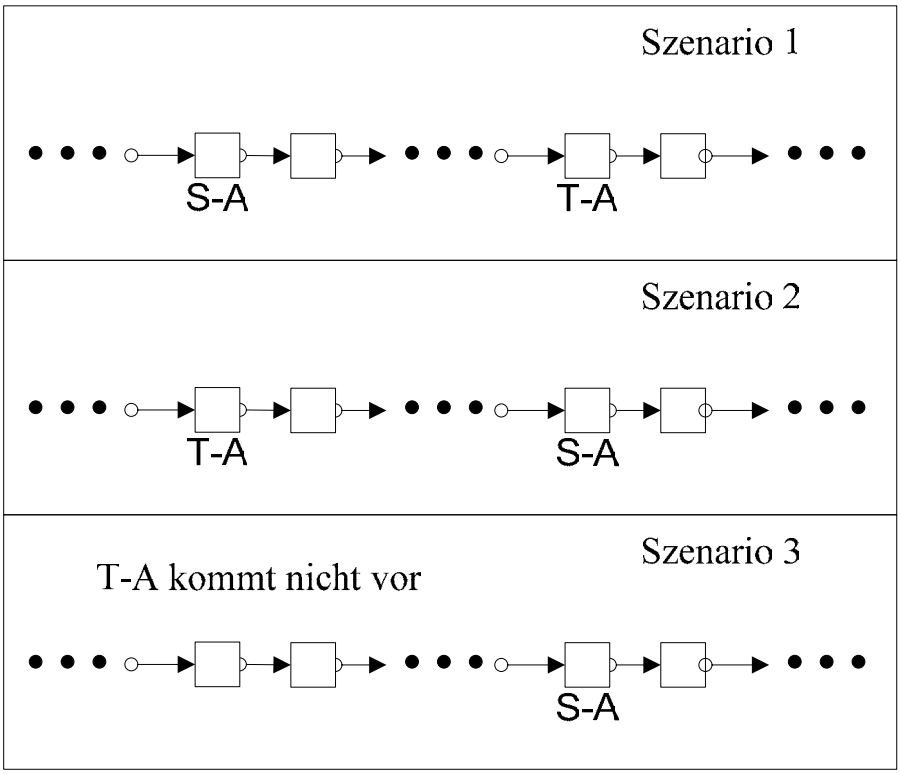
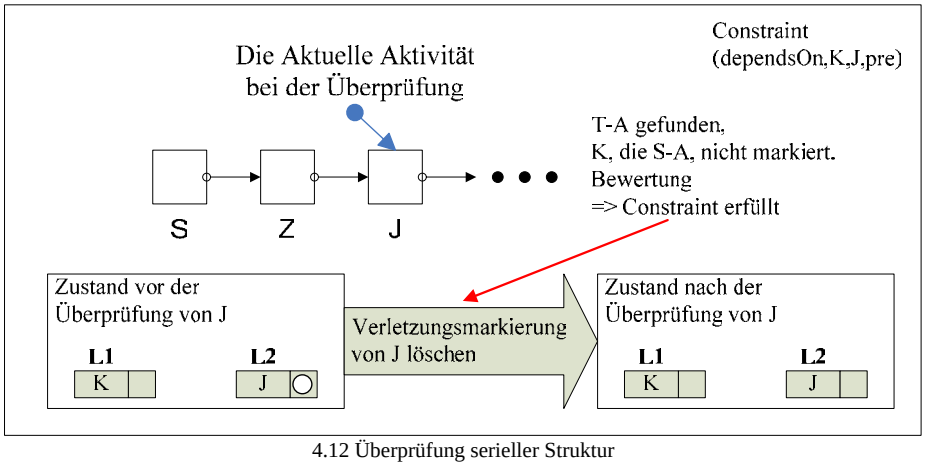


Abb. 4.11 Die drei möglichen Szenarien für serielle Prozessstrukturen

Wie die Abbildung 4.11 zeigt, gibt es für die serielle Struktur insgesamt drei mögliche relevante Konstellationen von S-A und T-A. Das Szenario 1 steht für den Fall, dass S-A vor T-A vorkommt. Das Szenario 2 steht für T-A vor S-A und das Szenario 3 steht für allein vorkommende S-A. Die Situation, dass nur die T-A vorkommt, wird durch die Verwendung der Menge L1_MIN ausgeschlossen (Wie gesagt, ohne L1_MIN bräuchten wir eine Nachbearbeitung des Überprüfungsergebnisses).

Vorher haben wir die Grundidee der fortgeschrittenen Vorgehensweise bereits einigermaßen verraten: Bei der Überprüfung werden wir jede gefundene S-A markieren und wenn eine T-A gefunden wird, können wir anhand des Markierungszustands der entsprechenden S-A feststellen, ob das Constraint auf dem Prozess erfüllt ist oder nicht. Dann können wir den Markierungszustand der gefundenen T-A angemessen ändern. D.h. wir machen die Bewertung eines Constraints immer zu dem Zeitpunkt, wo wir die T-A im Prozess finden. Wenn die T-A nicht im Prozess vorkommt, ist das Überprüfungsergebnis wegen des Ausgangszustands auch richtig. Mit Verwendung der Menge L1_MIN wissen wir schon, dass jede S-A in L1 früh oder später im Prozess vorkommen wird. Deswegen können wir eine Aussage über ein Constraint mit Sicherheit machen, wenn wir T-A des Constraints im Prozess mit nur serieller Struktur zuerst finden, weil die S-A bestimmt irgendwo nachher auftauchen wird (vgl. Abbildung 4.12). Dies wird bei Prozessen mit Block Struktur nicht gewährleistet. Deswegen passt das Grundkonzept der seriellen Struktur besonders gut. Darauf basiert der Algorithmus von Stufe I, der in Folgenden näher beschrieben wird.



4.12 Überprüfung serieller Struktur

Wir laufen den Prozess vorwärts durch. Dabei wird für jede Aktivität des Prozesses überprüft, ob sie in irgendeiner Menge vorkommt. Wenn ein Element in irgendeiner Menge bei der Überprüfung gefunden wird, ändern wir die Verletzungsmarkierungen der relevanten Aktivitäten. Die sechs Arten von Constraints werden unterschiedlich behandelt. Aber es gibt eine gemeinsame Operation für alle Constraints und alle Strukturen: Wenn ein Element von L1 im Prozess gefunden wird, markieren wir es. D.h. jede vorgekommene S-A wird markiert.

Für die Constraints mit *Position notSpecified* ist die Reihenfolge von S-A und T-A irrelevant, die Überprüfung für sie ist die einfachste. Daher betrachten wir zuerst den Algorithmus für sie, anschließend für die Constraints *Position post* und am Ende für die Constraints mit *Position pre*. Um den Algorithmus in Pseudocode zu beschreiben, definieren wir folgende Funktionen:

Definitionen:

Wir haben insgesamt sechs Arten von Constraints, jede enthält eine S-A und eine T-A. Deswegen haben wir zwölf Funktionen, um die Rolle(n) einer Aktivität zu bestimmen.

***A* :** Die Menge von allen Aktivitäten.

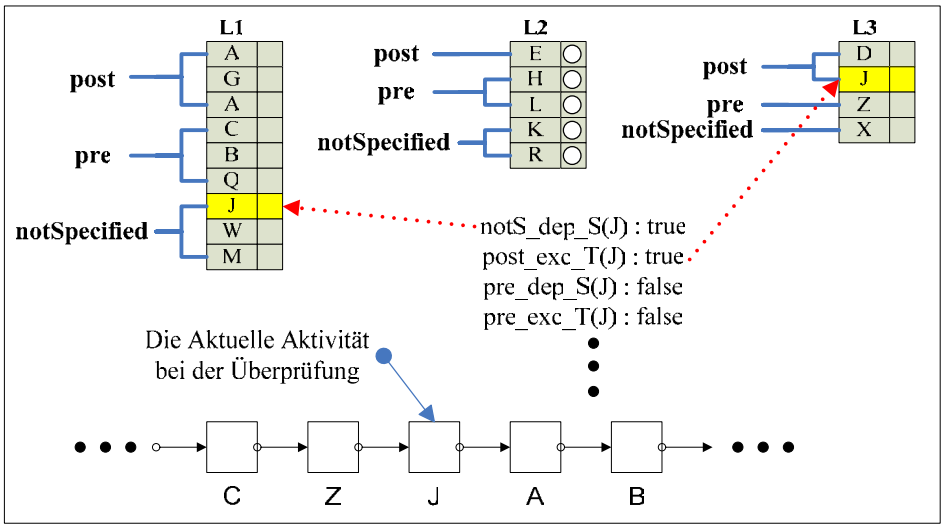
***X* :** X sei eine Aktivität, $X \in A$

notS_dep_S(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *notSpecified*-Teil von L1 mit einer T-A in L2 ist. D.h. ob es ein Constraint *(dependsOn, X, Y, notSpecified)* mit $Y \in A$ in der zu überprüfenden Constraintmenge gibt.

notS_dep_T(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *notSpecified*-Teil von L2 ist. D.h. ob es ein Constraint *(dependsOn, Y, X, notSpecified)* mit $Y \in A$ in der zu überprüfenden Constraintmenge gibt.

notS_exc_S(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *notSpecified*-Teil von L1 mit einer T-A in L3 ist. D.h. ob es ein Constraint *(excludes, X, Y, notSpecified)* mit $Y \in A$ in der zu überprüfenden Constraintmenge gibt.

notS_exc_T(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *notSpecified*-Teil von L3 ist. D.h. ob es ein Constraint *(excludes, Y, X, notSpecified)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
post_dep_S(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *post*-Teil von L1 mit einer T-A in L2 ist. D.h. ob es ein Constraint *(dependsOn, X, Y, post)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
post_dep_T(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *post*-Teil von L2 ist. D.h. ob es ein Constraint *(dependsOn, Y, X, post)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
post_exc_S(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *post*-Teil von L1 mit einer T-A in L3 ist. D.h. ob es ein Constraint *(excludes, X, Y, post)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
post_exc_T(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *post*-Teil von L3 ist. D.h. ob es ein Constraint *(excludes, Y, X, post)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
pre_dep_S(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *pre*-Teil von L1 mit einer T-A in L2 ist. D.h. ob es ein Constraint *(dependsOn, X, Y, pre)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
pre_dep_T(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *pre*-Teil von L2 ist. D.h. ob es ein Constraint *(dependsOn, Y, X, pre)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
pre_exc_S(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *pre*-Teil von L1 mit einer T-A in L3 ist. D.h. ob es ein Constraint *(excludes, X, Y, pre)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.
pre_exc_T(X) : boolesche Funktion, die überprüft, ob die Aktivität X ein Element von dem *pre*-Teil von L3 ist. D.h. ob es ein Constraint *(excludes, Y, X, pre)* mit Y ∈ A in der zu überprüfenden Constraintmenge gibt.

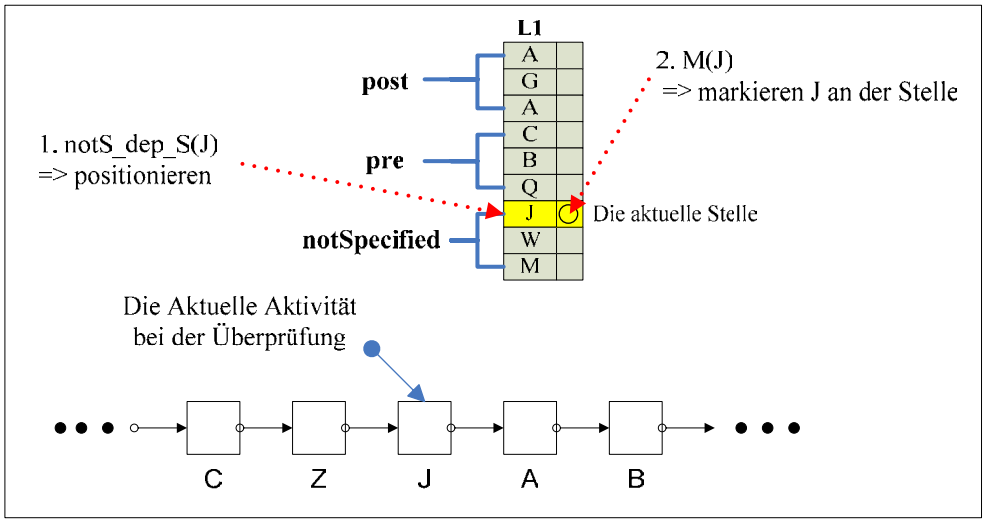


4.13 Werte der Funktionen

Mit diesen zwölf Funktionen können wir bei der Überprüfung die Positionen der aktuellen Aktivität in der drei Menge L1, L2 und L3 genau bestimmen. Die Abbildung 4.13 illustriert ein Beispiel für die Funktionswerte.

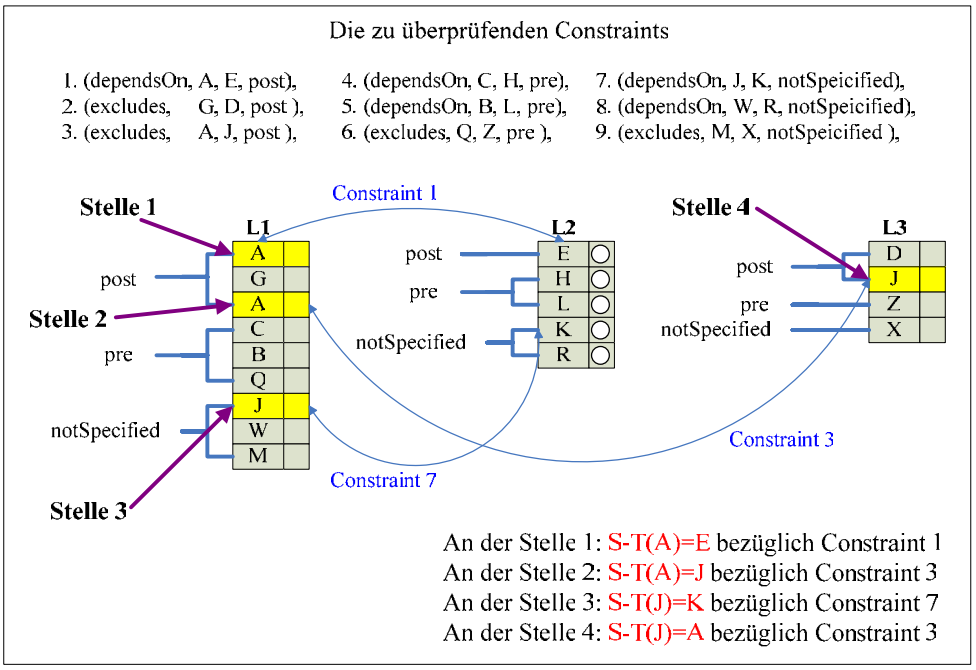
MARKIEREN(X) **:** Funktion, die Aktivität X an der Stelle, die durch die Funktionen ***notS_dep_S(X), notS_dep_T(X), notS_exc_S(X), notS_exc_T(X), post_dep_S(X), post_dep_T(X), post_exc_S(X),***

post_exc_T(X), pre_dep_S(X), pre_dep_T(X), pre_exc_S(X), pre_exc_T(X) festgestellt wird, mit dem Verletzungsmarkierung markiert. Die Verletzungsmarkierung können wir mit einer booleschen Variable *m* implementieren. Und als Markieren setzen wir den Wert von der Variablen auf *TRUE* (*m = true*), damit eine eventuelle redundante Markierung keinen Fehler erzeugen würde (vgl. Abbildung 4.14).



4.14 Die Wirkung der Funktion Markieren(X)

LÖSCHEN(X) **:** Funktion, die Verletzungsmarkierung von X an der aktuellen Stelle in der drei Menge, die durch die zwölf Funktionen wie für die Funktion Markieren(X) bestimmt wird, aufgehoben (*m = false*).
MIT_MARKIERUNG(X) **:** Funktion, die überprüft, ob die Aktivität X an der aktuellen Stelle mit einer Verletzungsmarkierung markiert ist.



4.15 Die Wirkung der Funktion S-T(X)

S-T(X) **:** Funktion, die relevante Aktivität für X an der aktuellen Stelle zurückliefert. D.h. Wenn X die S-A ist, holt die Funktion die entsprechende T-A und wenn X die T-A ist, liefert die Funktion die S-A zurück. Die Verbindungen zwischen S-A und T-A müssen daher da sein, um deutlich machen zu können, dass ein paar von S-A und T-A zu einem Constraint gehören (vgl. Abbildung 4.15).

Next(X) : Funktion, die die nächste Aktivität im Prozess holt.

Mit den Funktionen sieht den Algorithmus der Stufe I wie folgt aus:

Code:

```
// Für alle Aktivität in dem Prozess
for (X != End-Aktivität)
{
// Überprüfung für die Constraints mit Position notSpecified:
// Jede gefundene S-A wird markiert
        if notS_dep_S(X) M(X);
// Ein Constraint des Typs (dependsOn, Y, X, notSpecified) ist erfüllt, wenn die T-A
// vorkommt. Die Markierungszustand der S-A, nämlich ob S-A vorgekommen oder
// noch nicht, ist hier irrelevant. Mit gefundener T-A können wir die
// Verletzungsmarkierung daher sofort löschen.
        if notS_dep_T(X) LÖSCHEN(X);
// Ein Constraint des Typs (excludes, Y, X, notSpecified) ist verletzt, wenn doch nur
// die T-A auf serieller Struktur vorkommt.
        if notS_exc_S(X) M(X);
        if notS_exc_T(X) M(X);

// Überprüfung für die Constraints mit Position post:
// Die Constraints des Typs (dependsOn, X, Y, post) ist verletzt, wenn die T-A vor S-A gefunden wird.
// Deswegen überprüfen wir den Markierungszustand der S-A, bevor wir die Markierung von T-A
// aufheben.
        if post_dep_S(X) M(X);
        if post_dep_T(X)
                if MIT_MARKIERUNG(S-T(X))
                        LÖSCHEN(X);
// Für die Constraints des Typs (excludes, Y, X, post) wird ebenfalls für die gefundene
// T-A überprüft, ob die S-A bereits markiert, nämlich vorgekommen, ist. Wenn ja
// entspricht dies dem Szenario 2 in Abbildung 4.10. Das Constraint ist erfüllt und wir
// heben die Verletzungsmarkierung der T-A auf.
        if post_exc_S(X) M(X);
        if post_exc_T(X)
                if MIT_MARKIERUNG(S-T(X))
                        M(X);

// Überprüfung für die Constraints mit Position pre:
// Für die Constraints des Typs (dependsOn, Y, X, post) soll die T-A vor der S-A
// gefunden werden. Deswegen überprüfen wir, ob die S-A schon markiert ist. Wenn
// nicht, ist das Constraint erfüllt.
        if pre_dep_S(X) M(X);
        if pre_dep_T(X)
                if NOT(MIT_MARKIERUNG(S-T(X)))
                        LÖSCHEN(X);
// Ein Constraint des Typs (excludes, Y, X, post) ist verletzt, wenn die T-A zuerst
// gefunden wird.
        if pre_exc_S(X) M(X);
        if pre_exc_T(X)
                if NOT(MIT_MARKIERUNG(S-T(X)))
                        M(X);

// Hole die nächste Aktivität im Prozess.
        Next(X);
}
```

In Abbildung 4.16 werden die Modifikationen der Markierung bei der Überprüfung nach dem Algorithmus von Stufe I des Prozess veranschaulicht. Das Constraint 2 *(excludes, A, J, post)* ist verletzt auf dem Prozess. Entsprechend besitzt J in L3 nach der Überprüfung die Verletzungsmarkierung.

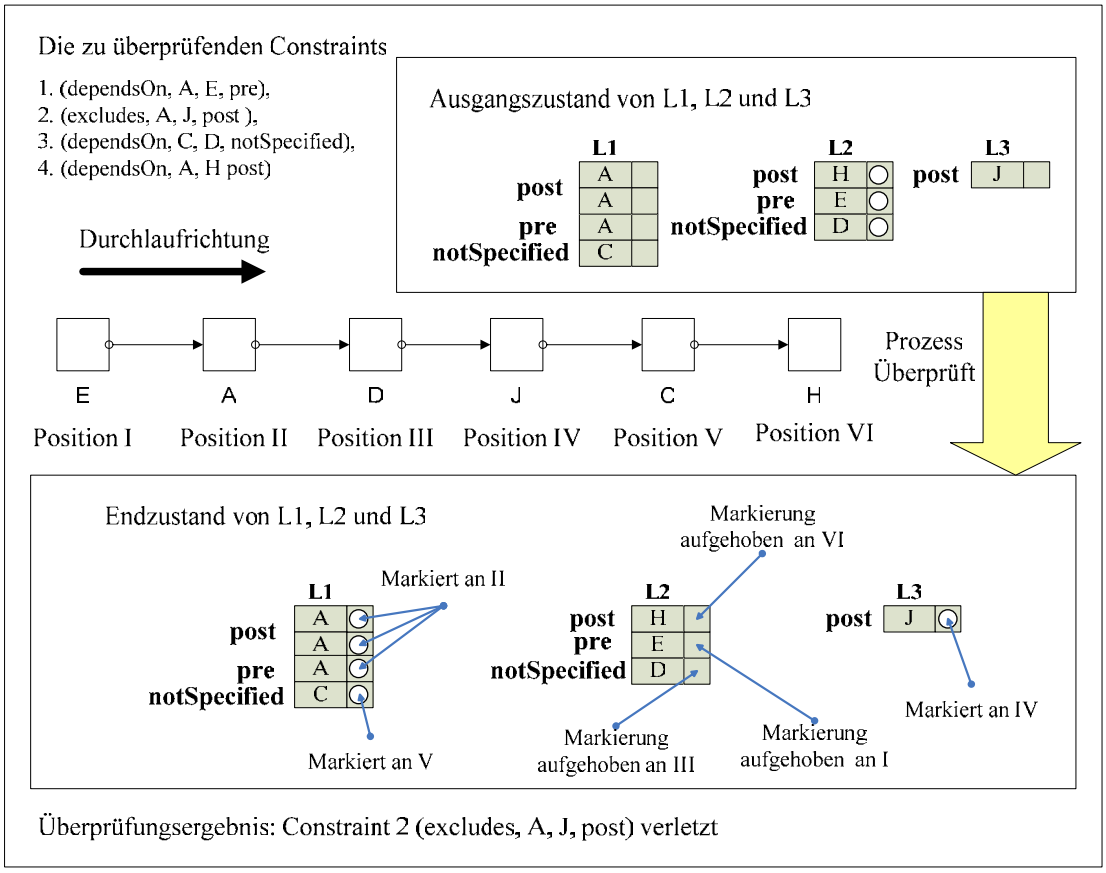


Abb. 4.16 Verwendung des Algorithmus von Stufe I

4.3.2 Überprüfung für einfache Blöcke

Für Prozesse mit Block-Strukturen können wir den Algorithmus von Stufe I nicht direkt einsetzen. Weil die Aktivitäten in einem XOR-Block nicht immer ausgeführt werden und die Ausführungsreihenfolge von Aktivitäten in einem UND-Block nicht fest ist.

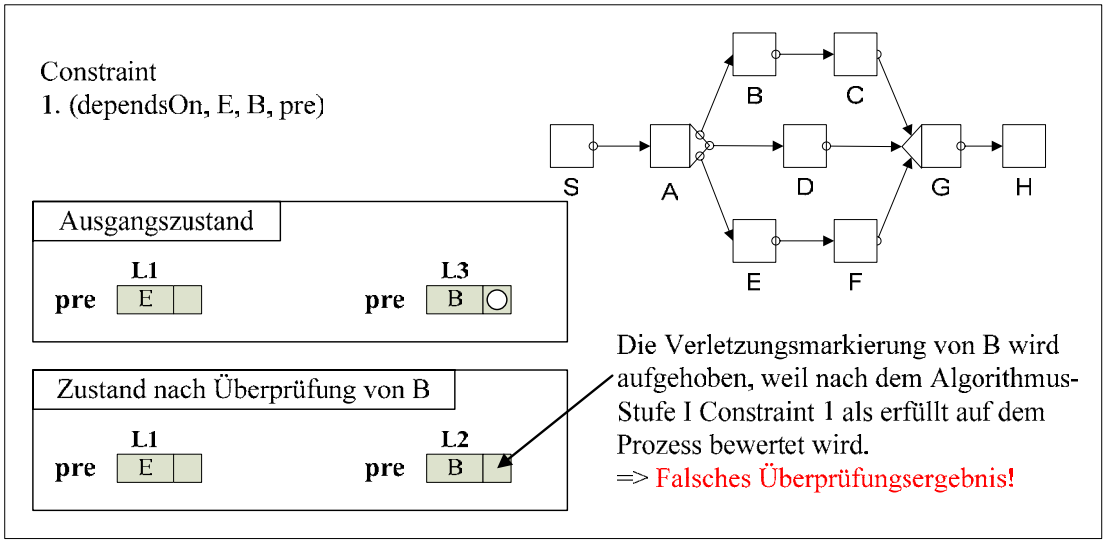
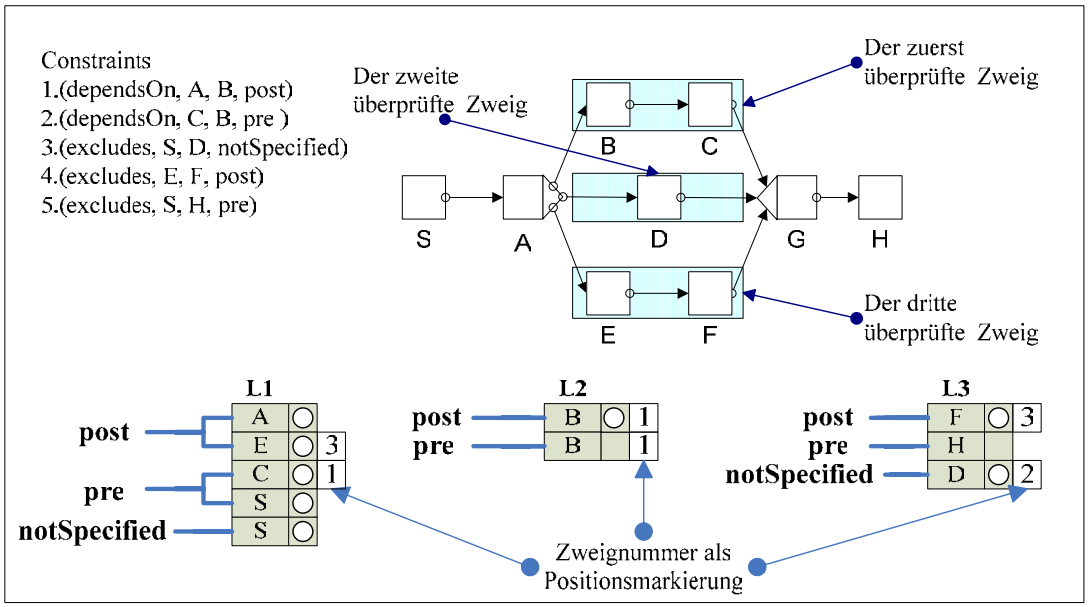


Abb. 4.17 Verwendung des Algorithmus von Stufe I an einem XOR-Block

Abbildung 4.17 illustriert ein Beispiel, wo die Anwendung des Algorithmus von Stufe I auf einem Prozess mit einem XOR-Block zu einem falschen Ergebnis führt. Auf serieller Struktur können wir an der Stelle der B feststellen, dass E nach B vorkommt wird und dass das Constraint *(dependsOn, E, B, pre)* auf dem Prozess erfüllt ist. Aber in einem XOR-Block ist diese Schlussfolgerung falsch. Dafür erweitern wir den Algorithmus von Stufe I zum Algorithmus von Stufe II. Im Folgenden wird dieser Algorithmus Schritt für Schritt hergeleitet.

Es gibt zwei Arten von Blöcken in ADEPT: XOR-Blöcke und UND-Blöcke. Entsprechend ist der Algorithmus von Stufe II zweigeteilt, um sie unterschiedlich zu behandeln. Für die Aktivitäten innerhalb eines Blocks verwenden wir außer Verletzungsmarkierung noch Positionsmarkierung. Für einfache Blöcke nummerieren wir die Zweige eines Blocks und markieren jede Aktivität innerhalb eines Blocks mit einer Zweignummer als Positionsmarkierung. Damit können wir bei Bewertung eines Constraints deutlich machen, wo sich eine Aktivität im Block befindet und wie die Konstellation zwischen S-A und T-A ist. Allerdings müssen wir nicht vor der Überprüfung des XOR-Blocks, wie z.B. an der Stelle von A im Prozess in Abbildung 4.18, die Zweige des XOR-Blocks nummerieren und die Positionsmarkierung erzeugen. Dafür ist nämlich keine Vorarbeit notwendig.

Mit unserem Algorithmus können die Zweige eines Blocks in beliebiger Reihenfolge überprüft werden. Die Aktivitäten auf dem Zweig, der zuerst überprüft wird, markieren wir mit Zweignummer 1, wenn sie bei der Überprüfung gelesen werden. Die auf den zweiten überprüften Zweig markieren wir mit 2, usw. In Abbildung 4.18 wird eine Möglichkeit von Positionsmarkierungen der Aktivitäten eines XOR-Blocks gezeigt, wobei die Zweige von oben nach unten überprüft werden. Die Markierung von Stufe II setzt sich aus Verletzungsmarkierung und Zweignummer als Positionsmarkierung zusammen. Wir werden weiterhin die Zweige eines Blocks wie in Abbildung 4.18 immer von oben nach untern überprüfen. Die Überprüfungsreihenfolge von den Aktivitäten ist {B, C, D, E, F}. Wir erzeugen eine *Integer*-Variable *p*, um die Zweignummer (als Positionsmarkierung bei Markierung von Stufe II) zu speichern. Jedoch müssen die Aktivitäten, die sich nicht innerhalb eines Blockes befinden, nach wie vor nur mit Verletzungsmarkierung markiert werden (vgl. Abbildung 4.18). Für sie nehmen wir an, dass *p = Null* ist. Wenn die Zweignummern von zwei Aktivitäten gleich sind, wissen wir, dass sie sich auf demselben Zweig des Blocks befinden. Wenn die Zweignummern ungleich sind, befinden sie sich dann auf zwei unterschiedlichen Zweig des Blocks. Nach Überprüfung des ganzen Blocks löschen wir alle Positionsmarkierungen.



4.18 Markierung von Stufe II für Aktivitäten in Block

4.3.2.1 Überprüfung eines XOR-Blocks

Sync-Kanten können nur zwischen Zweigen eines UND-Blocks auftreten. Das macht die Überprüfung für den XOR-Block einfacher als die für den UND-Block. Wir betrachten daher zuerst den Überprüfungsalgorithmus für einen einfachen XOR-Block. Er ist prinzipiell derselbe wie der für die serielle Struktur. Falls wir T-A nach S-A finden, können wir anhand der Positionsmarkierung die Konstellation der beiden feststellen. Damit ist die entsprechende Bewertung des Constraint auf jedem Fall richtig. Aber wenn T-A bei der Überprüfung zuerst gefunden wird, ist eine sofortige korrekte Aussage über die Erfüllung des Constraints nicht immer gewährleistet, weil die Position von S-A nicht mehr eindeutig wie auf der seriellen Struktur ist. Eventuell müssen wir die Bewertung eines Constraints korrigieren, wenn wir S-A nach T-A in dem Block finden.

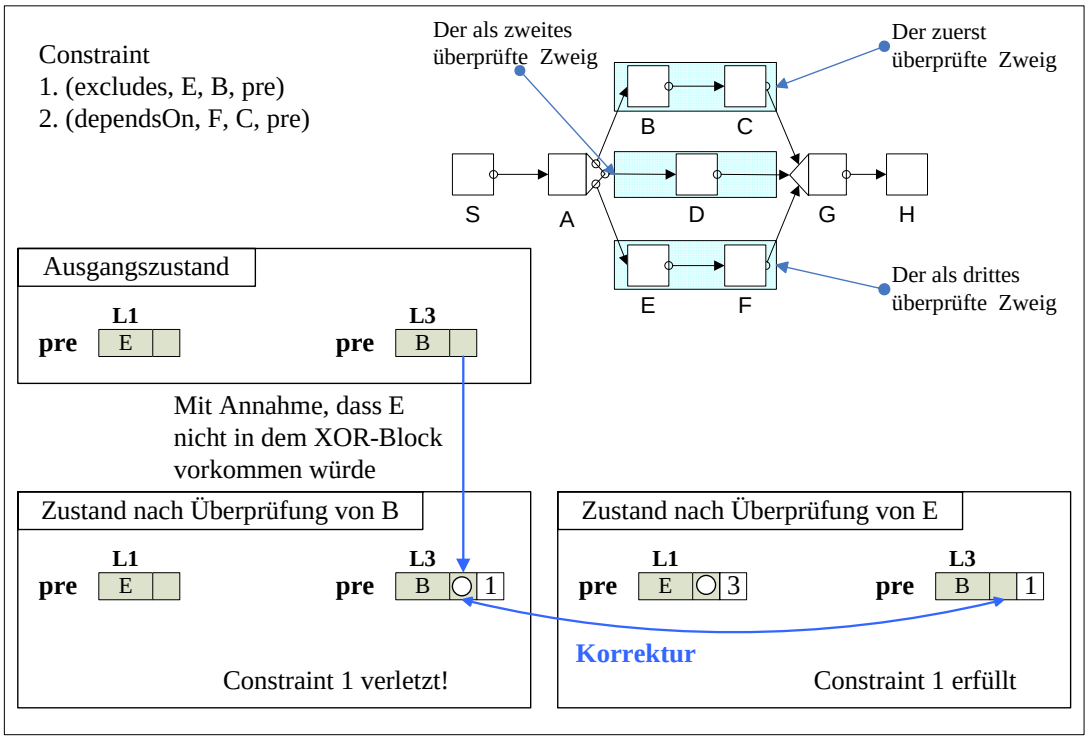


Abb. 4.19 Markierung für die zu erst gefundene T-A im XOR-Block

Das Problem von mehrfachen Möglichkeiten der Position von S-A lösen wir dadurch: Wenn wir eine T-A in einem XOR-Block mit bis dahin nicht vorgekommener S-A finden, markieren wir die T-A an dieser Stelle so, als ob die S-A nicht in dem XOR-Block sondern nach dem XOR-Block vorkommen würde. Wenn die S-A aber tatsächlich nachher im XOR-Block gefunden wird, korrigieren wir die Markierung der T-A, falls nötig. Unter Umständen führen eine S-A innerhalb eines XOR-Blocks und eine S-A nach dem XOR-Block zu demselben Resultat. Da ist dann keine Korrektur notwendig. In Abbildung 4.19 wird B bei der Überprüfung auf ihrer Stelle mit der Verletzungsmarkierung markiert, weil unter der Annahme, dass E nach dem Block vorkommt, das Constraint *(excludes E, B, pre)* verletzt wäre. Allerdings wird E nachher auf einem anderen Zweig des XOR-Blocks gefunden. Bei der Konstellation ist das Constraint erfüllt. Entsprechend wird die Verletzungsmarkierung von B bei Überprüfung von E wieder aufgehoben. Für Constraint 2 macht es keinen Unterschied, ob F innerhalb des XOR-Blocks oder nach dem gefunden werden. Das Constraint *(dependsOn, F, C, pre)* wäre in beiden Fällen verletzt. Dabei ist die initiale Markierung von C unter der erwähnten Annahme richtig. Bei der Überprüfung von F müssen wir sie nicht mehr korrigieren.

Für die Überprüfung eines XOR-Blocks sind insgesamt acht Konstellationen von S-A und T-A relevant (vgl. Abbildung 4.20). Die Beziehung, die durch die Paare von Aktivitäten S-A1 und T-A1 in Abbildung 4.20 dargestellt wird, nennen wir S-T-1. Und die Beziehung zwischen S-A2 und T-A2 nennen wir S-T-2, usw. Bei S-T-1 wurde S-A vor dem XOR-Block auf serieller Struktur überprüft und markiert. Wenn wir nun die T-A im XOR-Block finden, können wir an der Stelle mit Sicherheit eine richtige Aussage über das entsprechende Constraint machen. Bei S-T-2 wurde die Beurteilung über das Constraint bereits an der Stelle von T-A vor dem XOR-Block gemacht. Sie ist ebenfalls richtig, weil es keinen Unterschied macht,

ob die S-A nach T-A in Block-Struktur oder auf serieller Struktur vorkommt. Die S-A bleibt semantisch effektiv.

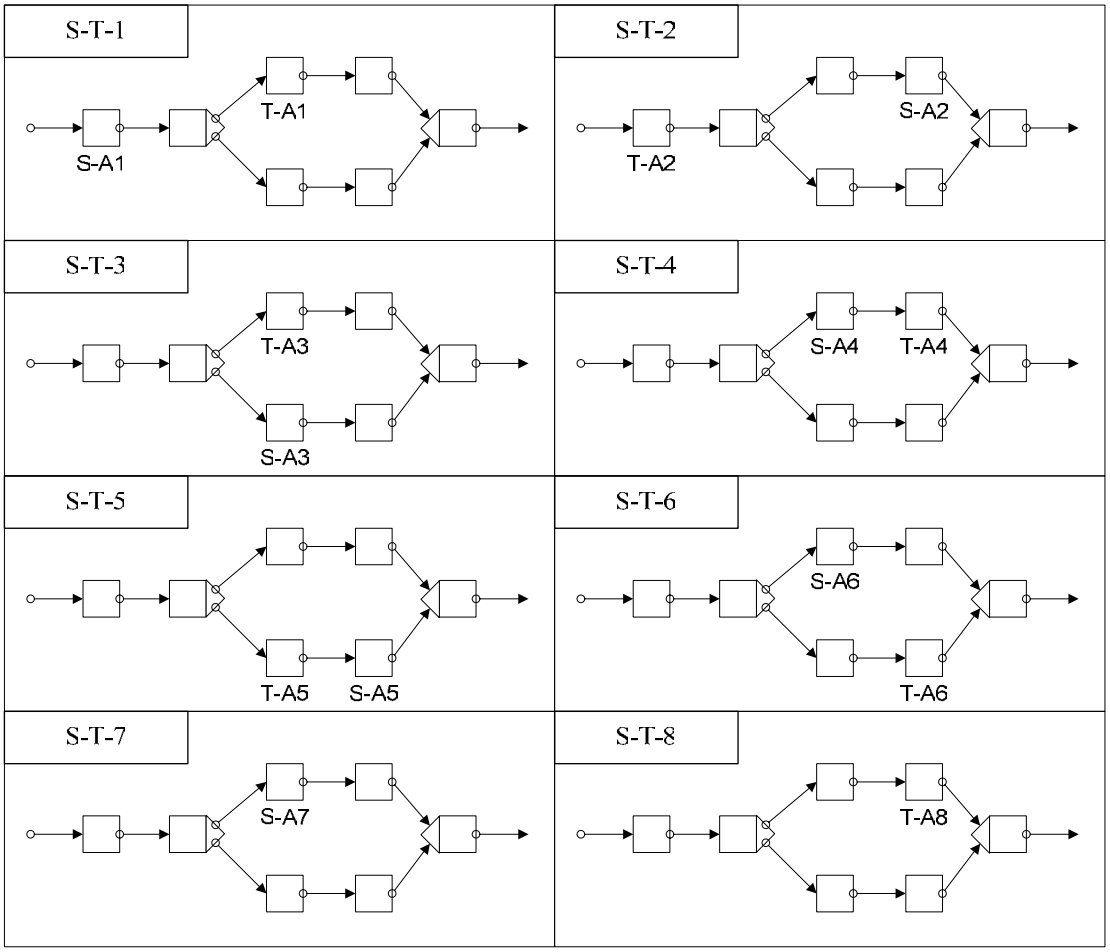


Abb. 4.20 Die mögliche Verhältnis von S-A und T-A mit einem XOR-Block

Bei S-T-4 und S-T-5 befinden sich die S-A und T-A auf demselben Zweig im XOR-Block. Da sind die Zweignummern von den beiden gleich. Für diese Aktivitäten ist die semantische Überprüfung analog wie bei serieller Struktur. Trotzdem werden wir in dem folgenden Pseudocode für den Algorithmus von Stufe II die entsprechenden Operationen für diese Aktivitäten direkt ausgeben. Bei S-T-4 wird T-A nach S-A gefunden. D.h. die Bewertung eines Constraint, sobald T-A gefunden wird, ist im Szenario S-T-4 immer richtig. Dagegen müssen wir die Bewertung eines Constraint, die beim Finden der T-A stattfindet, im Szenario S-T-5 wieder korrigieren, wenn S-T-8 (wie wir annehmen, dass S-A nicht im XOR-Block vorkommen würde) und S-T-5 zu unterschiedlichen Überprüfungsergebnis führen sollen.

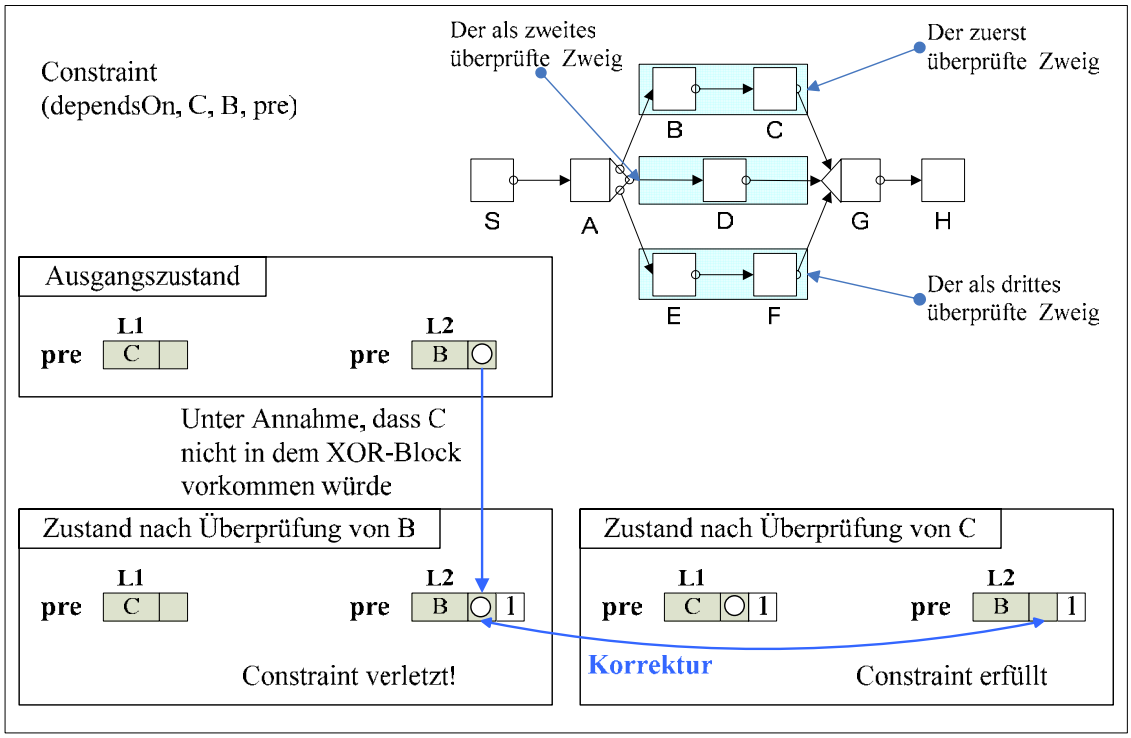


Abb. 4.21 Korrektur für S-T-5

In Abbildung 4.21 ist das Constraint *(dependsOn, C, B, pre)* unter der Annahme, dass C (die S-A) nicht in dem XOR-Block vorkommen würde (S-T-8), verletzt. Aber C wurde im späteren Verlauf der Überprüfung auf demselben Zweig des XOR-Blocks gefunden (S-T-5). Nun können mit Sicherheit wir feststellen, dass das Constraint erfüllt ist. Daher korrigieren wir die Markierung von B (die T-A).

Wenn die Zweignummern von der S-A und T-A ungleich sind, wissen wir, dass die beiden Aktivitäten sich auf zwei unterschiedlichen Zweigen befinden. S-T-3 und S-T-6 in Abbildung 4.20 stellen die Situationen dar. Der Unterschied zwischen den beiden liegt darin, ob S-A oder T-A bei der Überprüfung des XOR-Blocks zuerst gefunden wurde (Überprüfungsreihenfolge: von oben nach unten). Bei S-T-3 brauchen wir eventuell auch eine Korrektur der Bewertung eines Constraint (Markierung der T-A). Bei S-T-7 und S-T-8 treten jeweils nur die S-A oder die T-A im XOR-Block auf.

Im Folgenden wird der Algorithmus zur Verifikation von einfachen Blöcken im Pseudocode dargestellt. Für die Überprüfung eines einfachen Blocks brauchen wir zusätzlich die folgenden Funktionen:

Definitionen:

M_POSITIONSM(X) : Funktion, die die Aktivität X auf jede Stellen in L1 L2 und L3 mit Positionsmarkierung markiert.

BRANCHNR(X) : Funktion, die den Wert der Variable *p* (die Zweignummer) von Aktivität X zurückliefert.

Next_IN_BLOCK(X) : Funktion, die die nächste Aktivität auf dem Zweig holt. Am Ende eines Zweigs holt sie die erste Aktivität des nächsten zu überprüfenden Zweigs. Wenn alle Aktivitäten des Blocks geholt wurden, liefert die Funktion *Null* zurück.

L_POSITIONSM : Funktion, die alle Positionsmarkierung in L1, L2 und L3 aufhebt.

Damit sieht der Algorithmus von Stufe II für XOR-Block in Pseudocode so aus:

Code:

```
// Für jede Aktivität im Block
Next_IN_BLOCK(X);
For (X!=Null)
{
// Jede Aktivität im Block wird mit Positionsmarkierung markiert.
        M_POSITIONSM(X);
// Überprüfung für die Constraints mit Position notSpecified:
```

```
// Jede gefundene S-A wird markiert
        if notS_dep_S(X)
        {       M(X);
// Wenn die T-A auf denselben Zweig im Block bereits gefunden wurde, ist das Constraint
// (dependsOn, X, Y, notSpecified) erfüllt (S-T-5 in Abbildung 4.20). Aber an der Stelle von T-A wurde
// angenommen, dass S-A nicht in dem XOR-Block vorkommen würde. Und das Constraint wurde da als
// verletzt bewertet. Daher korrigieren wir den Markierungszustand der T-A. Hierbei keine Korrektur
// für S-T-3 nötig.
                if (BRANCHNR(X)==BRANCHNR(S-T(X)))
                        LÖSCHEN(S-T(X));                        // Korrektur für S-T-5
        }
// Das Constraint ist ebenfalls erfüllt bei S-T-4. Wenn die S-A außerhalb des Blocks (S-T-1 und S-T-8)
// oder auf ein anderen Zweig (S-T-6) vorkommt, ist das Constraint verletzt. Dafür ist keine Operation
// notwendig, da die T-A bei Initialisierung bereits markiert wurde.
        if notS_dep_T(X)
                If (BRANCHNR(X)==BRANCHNR(S-T(X)))
                        LÖSCHEN(X);                             // S-T-4
// Ein Constraint des Typs (excludes, X, Y, notSpecified) ist erfüllt, wenn die T-A auf ein anderen Zweig
// des XOR-Blocks (S-T-3) vorkommt. Die S-A und T-A werden dann nie beide an einer Instanz beide
// ausgeführt. In diesen Fall korrigieren wir die Markierung der T-A.
        if notS_exc_S(X)
        {       M(X);
                if ((BRANCHNR(S-T(X)!=Null) AND (BRANCHNR(X)!=BRANCHNR(S-T(X))))
                        LÖSCHEN(S-T(X));                // Korrektur für S-T-3
        }
// Analog für die gefundene T-A.
        if notS_exc_T(X)
        {       M(X);
                if ((BRANCHNR(S-T(X)!=Null) AND (BRANCHNR(X)!=BRANCHNR(S-T(X))))
                        LÖSCHEN(X);                     // S-T-6
        }

// Überprüfung für die Constraints mit Position post:
// Für die Constraints des Typs (dependsOn, X, Y, post) ist keine Korrektur an der Stelle von S-A nötig,
// weil das Constraint wie beim Ausgangszustand verletzt ist, wenn die S-A nach T-A gefunden würde,
        if post_dep_S(X) M(X);
// Wenn die T-A nach S-A auf demselben Zweig gefunden wird, ist das Constraint (dependsOn, X, Y, post)
// erfüllt (S-T-4). Bei allen anderen Konstellationen von S-A und T-A in Abbildung 4.20 ist das
// Constraint verletzt.
        if post_dep_T(X)
                if (BRANCHNR(X)==BRANCHNR(S-T(X)))              // S-T-4
                        LÖSCHEN(X);
// Ein Constraint (excludes, X, Y, post) ist bei S-T-1 und S-T-4 verletzt. Hierbei ist keine Korrektur nötig.
        if post_exc_S(X) M(X);
        if post_exc_T(X)
        {       if (((MIT_MARKIERUNG (S-T(X)))AND (BRANCHNR(S-T(X))==Null)))
                                                                // S-T-1
                OR (BRANCHNR(X)==BRANCHNR(S-T(X)))              // S-T-4
                        M(X);
        }
// Überprüfung für die Constraints mit Position pre:
// Bei S-T-2 und S-T-5 ist das Constraint (dependsOn, X, Y, pre) erfüllt. Bei S-T-2 wurde die
// Verletzungsmarkierung von T-A bereits bei der Überprüfung von T-A auf serieller Struktur aufgehoben.
        if pre_dep_S(X)
        {       M(X);
                if (BRANCHNR(X)==BRANCHNR(S-T(X)))              // Korrektur für S-T-5
                        LÖSCHEN(S-T(X));
        }
```

```
// Es sind keine Operationen für eine gefundene T-A des Constraints (dependsOn, Y, X, pre) in einem
// XOR-Block notwendig. Bei allen Fällen werden wir die T-A nach wie vor markiert bleiben lassen.
// Ein Constraint (excludes, X, Y, pre) ist verletzt bei S-T-5 und S-T-8. Bei S-T-3 wurde T-A zuerst
// markiert. Wir müssen daher die Verletzungsmarkierung von T-A aufheben, wenn S-A im XOR-Block
// vorkommt.
        if pre_exc_S(X)
        {       M(X);
                if (BRANCHNR(S-T(X))!=BRANCHNR(X))      // Korrektur für S-T-3, entspricht
                        LÖSCHEN(S-T(X));                // dem Beispiel in Abb. 4.18
        }
        if pre_exc_T(X)
                if (NOT(MIT_MARKIERUNG(S-T(X))))        // S-T-5 und S-T-8
                        M(X);

// Hole die nächste Aktivität im XOR-Block.
        Next_IN_BLOCK(X);
}
// Alle Positionsmarkierungen aufheben.
L_POSITIONSM;
```

Für die Aktivitäten, die sich außerhalb eines XOR-Blocks befinden, sind alle S-A in dem XOR-Block semantisch effektiv. Ebenfalls sind die allein vorkommenden T-A (d.h. die S-A nicht vorgekommen sind) von Constraints mit *Type = excludes* im XOR-Block semantisch effektiv. Dagegen sind die allein vorkommenden T-A von Constraints mit *Type = dependsOn* im XOR-Block semantisch ineffektiv (siehe Abschnitt 3.1.2). Nach unseren Algorithmus werden nach der Überprüfung eines XOR-Blocks die richtigen Markierungszustände von L1, L2 und L3 vorliegen. Die S-A werden bei Überprüfung eines XOR-Blocks nach wie vor markiert, und die allein gefundene T-A werden so markiert, als ob die entsprechenden S-A nicht im XOR-Block vorkommen würden. Deswegen ist keine Nachbearbeitung der Markierung in L1, L2 und L3 nach der Überprüfung eines XOR-Blocks notwendig.

Wir zeigen im Folgenden ein Beispiel, wobei die Überprüfungsschritte eines XOR-Blocks veranschaulicht werden. In Abbildung 4.22 wird die Überprüfung der Aktivitäten {B, C, D} auf dem obersten Zweig des XOR-Blocks dargestellt. Vor der Überprüfung von B wurden S und A auf der seriellen Struktur bereits mit dem Algorithmus von Stufe I überprüft und als S-A markiert.

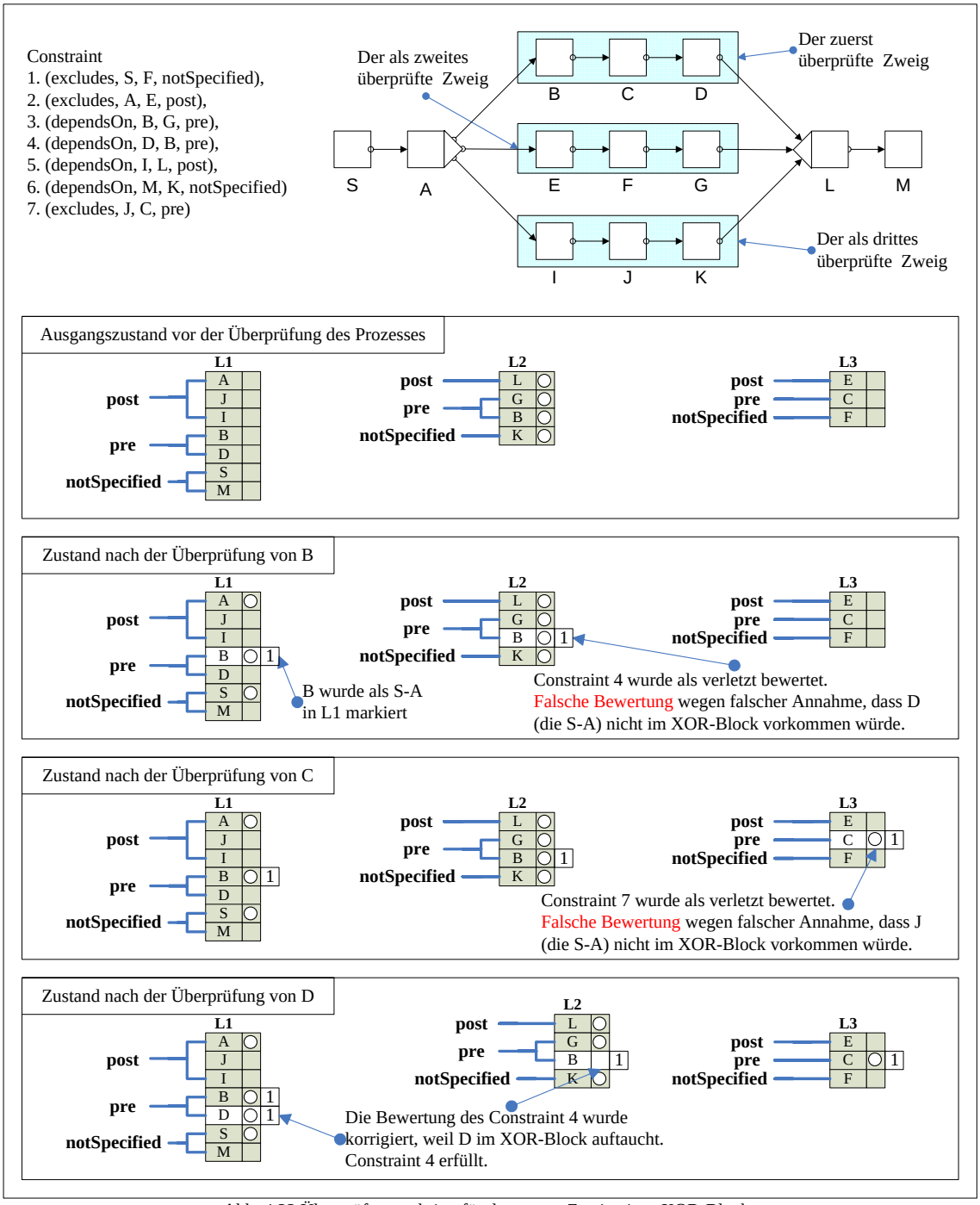


Abb. 4.22 Überprüfungsschritte für den ersten Zweig eines XOR-Blocks

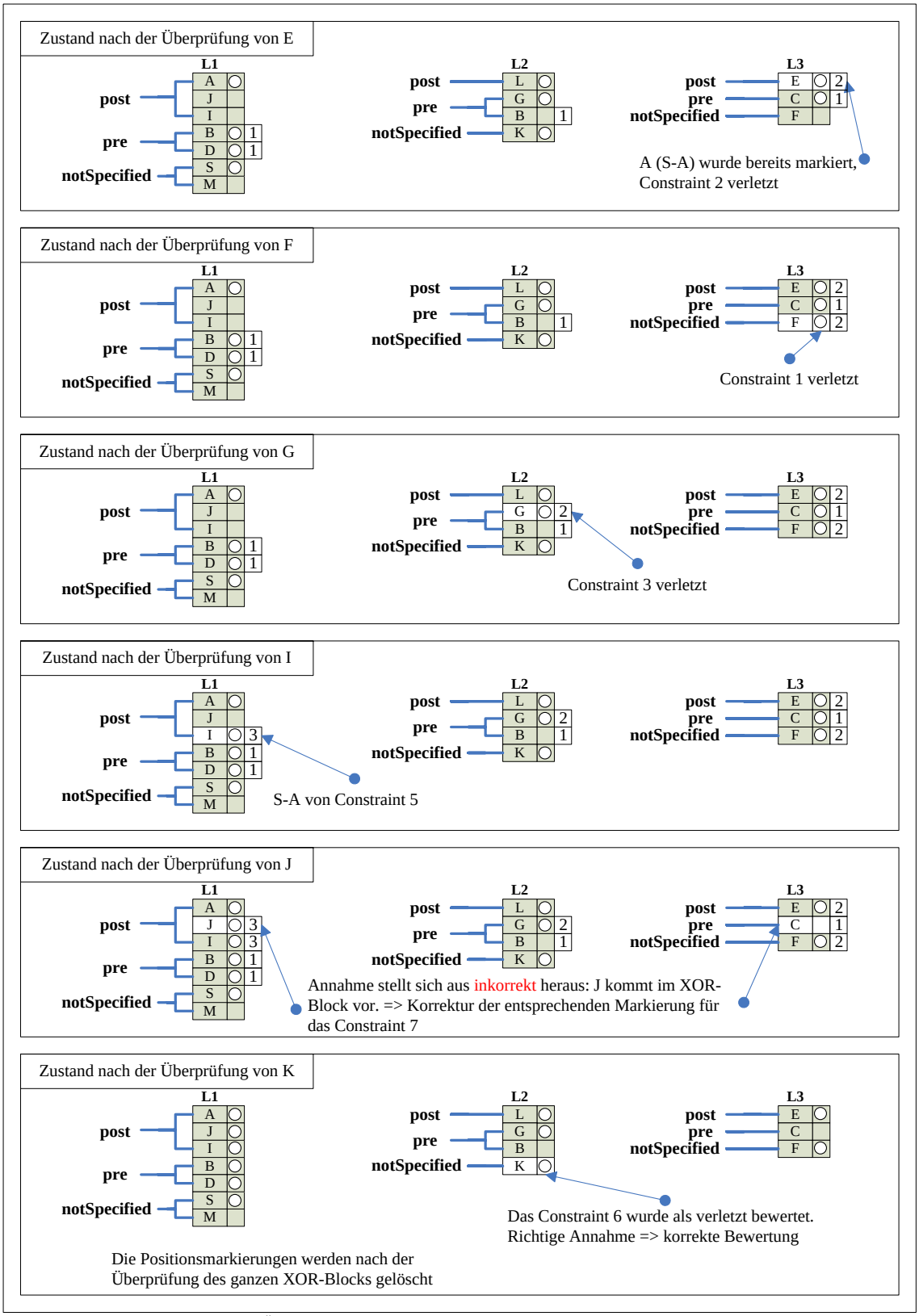


Abb. 4.23 Überprüfungsschritte für die übrigen zwei Zweige des XOR-Blocks

4.3.2.2 Überprüfung eines UND-Blocks

Die Überprüfung eines UND-Blocks ist prinzipiell ähnlich zu der Überprüfung eines XOR-Blocks. Allerdings müssen wir hierbei zusätzlich die Sync-Kanten bei der Überprüfung berücksichtigen. Wie die Aktivitäten auf serieller Struktur werden die Aktivitäten in einem UND-Block immer ausgeführt. Allerdings lässt sich die Ausführungsreihenfolge von Aktivitäten auf unterschiedlichen Zweigen eines UND-Blocks nicht genau festlegen. Um eine Synchronisation paralleler Zweige herzustellen, gibt es das Konzept der Sync-Kante. Bevor wir den Überprüfungsalgorithmus für UND-Blöcke kennen lernen, werden wir zunächst die semantische Wirkung der Sync-Kante analysieren. Wie in Abschnitt 2.1 geschrieben, die Wirkung von Sync-Kanten ist nichts anderes als die Ausführungsreihenfolge der Aktivitäten auf zwei unterschiedlichen Zweigen eines UND-Blocks teilweise festzulegen. In Abbildung 4.24 wird durch die Sync-Kante zwischen J und G festgestellt, dass J (damit auch I) vor G (damit auch vor H) ausgeführt wird. Und die Sync-Kante hat keinen Einfluss auf den anderen Zweig {B, C, D}.

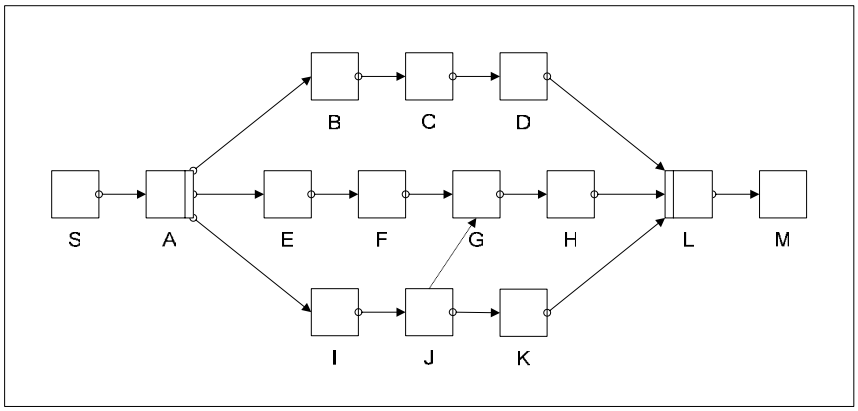


Abb. 4.24 Sync-Kante in einem einfachen UND-Block

Satz:

Durch Sync-Kanten wird kein Constraint verletzt. Jedoch kann es Constraints geben, die durch Sync-Kanten erfüllt werden.

Das begründen wir wie folgt: Für die Constraint mit *Position notSpecified* ist die Ausführungsreihenfolge von S-A und T-A irrelevant. Daher hat Sync-Kante keine Wirkung auf sie. Die übrigen Constraints sind verletzt, wenn die Ausführungsreihenfolge von S-A und T-A nicht fest ist. Wir können die gewünschte Ausführungsreihenfolge für die Abhängigkeitsbeziehung nicht sichern und wir können die ungewünschte Ausführungsreihenfolge für die Ausschlussbeziehung nicht ausschließen. Wenn wir da die Ausführungsreihenfolge von S-A und T-A mit einer Sync-Kante richtig feststellen, können wir jedoch die Constraints erfüllen. Beispielsweise wäre das Constraint *(dependsOn, I, H, post)* verletzt auf dem Prozess in Abbildung 4.24, wenn die Sync-Kante zwischen J und G nicht existierten, weil dann H vor I ausgeführt werden kann. Aber das Constraint ist mit der Sync-Kante erfüllt. Für die Ausschlussbeziehung ist es analog. Das Constraint *(excludes, J, G, pre)* wäre ohne die Sync-Kante verletzt. Aber mit der Sync-Kante kann G nur nach J ausgeführt werden. Das Constraint ist deswegen durch die Sync-Kante erfüllt.

Daher werden wir bei der Überprüfung eines UND-Blocks zuerst die Sync-Kanten ignorieren. Nach der Überprüfung des UND-Blocks ohne Sync-Kanten, allerdings korrigieren wir anhand der Sync-Kante das Überprüfungsergebnis vor dem Löschen der Positionsmarkierungen. Dabei werden die Verletzungsmarkierungen der T-A von Constraints, die durch Sync-Kante erfüllt sind, aufgehoben. Es ist auch möglich, die Überprüfung für die Sync-Kante in der Überprüfung des UND-Blocks zu integrieren. Jedoch wird der Algorithmus dadurch komplexer und unübersichtlich. Deswegen machen wir die Überprüfung zuerst separat.

Wir betrachten zuerst den Algorithmus für UND-Block ohne Sync-Kante. Wie bei der Überprüfung eines XOR-Blocks: Wenn wir eine T-A in einem UND-Block mit nicht vorgekommener S-A finden, markieren wir sie an der Stelle so, als ob die S-A nicht in dem UND-Block sondern nach dem UND-Block vorkommen würde. Wenn die S-A nachher im UND-Block gefunden wird, korrigieren wir die Markierung der T-A (vgl. Abbildung 4.25). Damit ist ebenfalls für die Überprüfung von UND-Block, außer der Korrektur anhand der Sync-Kante, keine Nachbearbeitung der Markierungszustände von L1, L2

oder L3 für die Aktivitäten nach dem UND-Block erforderlich. In Abbildung 4.25 wird die Verletzungsmarkierung von B bei der Überprüfung auf ihrer Stelle aufgehoben. Unter der Annahme, dass E nach dem Block vorkommt, wäre das Constraint *(dependsOn, E, B, pre)* erfüllt. Wenn E nicht in dem UND-Block vorkommen würde, wäre diese Markierung von B korrekt. Allerdings wird E nachher in einem anderen Zweig des UND-Blocks gefunden. Dann ist die Ausführungsreihenfolge von B und E nicht mehr fest. Entsprechend markieren wir B wieder, um zu zeigen, dass das Constraint verletzt ist.

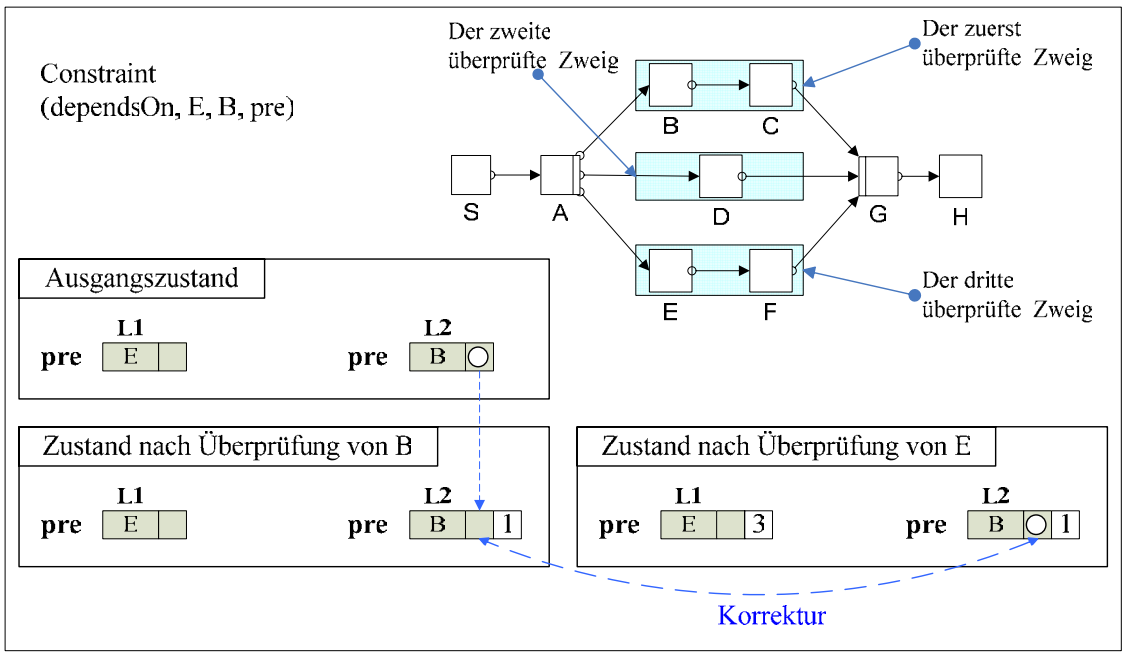


Abb. 4.25 Markierung für die zu erst gefundene T-A in einem UND-Block

Für einen UND-Block gibt es wie bei XOR-Block ebenfalls acht mögliche Konstellationen von S-A und T-A (vgl. Abb. 4.26). Aber anders als bei XOR-Blöcken benötigen wir für S-T-5 bei dem Überprüfungsverfahren für UND-Blöcke keine Korrektur, weil S-A sowohl bei S-T-5 und als auch bei S-T-8 (die Konstellation, die wir für zuerst gefundene T-A in einem UND-Block annehmen) nach T-A vorkommen wird. Und in einem UND-Block wird die T-A immer ausgeführt. D.h. S-T-5 und S-T-8 sind semantisch gleichgültig. Deshalb brauchen wir für die Überprüfung für UND-Blöcke nur bei S-T-3 an eine eventuelle Korrektur der Markierungszustand der T-A zu denken. Im folgenden Pseudocode ist mit S-T-1 die Positionen von S-A-1 und T-A-1 in Abbildung 4.26 gemeint und S-T-2 für S-A-2 und T-A-2 in Abbildung 4.26, usw.

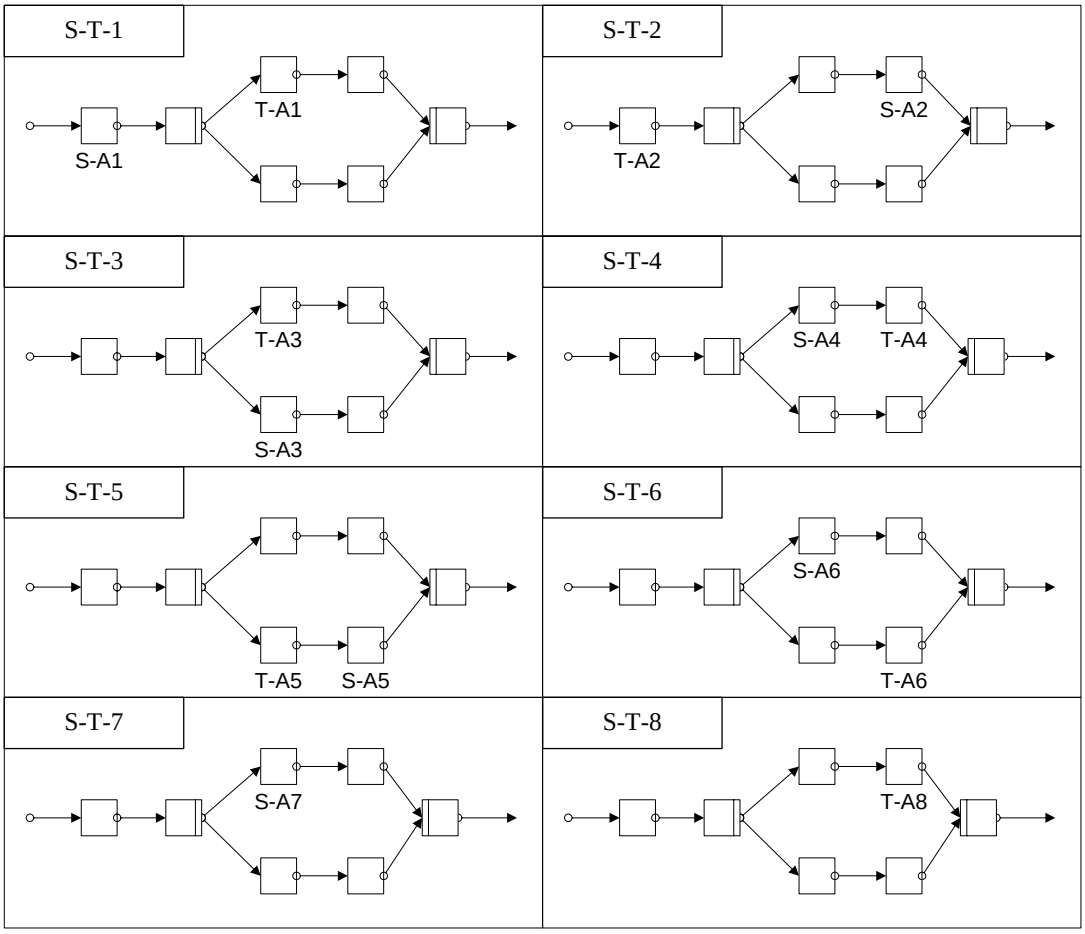


Abb. 4.26 Mögliche Konstellationen von S-A und T-A in einem UND-Block

Der Algorithmus von Stufe II für UND-Block ohne Sync-Kante sieht im Pseudocode wie folgt aus:

Definitionen:

SYNC-KANTE(X) : boolesche Funktion, die überprüft, ob X die Quell-Aktivität einer Sync-Kante ist. Die Aktivität J in Abbildung 4.25 ist beispielsweise die Quell-Aktivität der Sync-Kante (J, G).

SPEICHERN_S-K(X) : Funktion, die X in der Menge S, eine Menge von Quell-Aktivitäten von Sync-Kante, speichert.

Code:

```
// Für jede Aktivität im Block
Next_IN_BLOCK(X);
For (X!=Null)
{
// Jede Aktivität im Block wird mit Positionsmarkierung markiert.
        M_POSITIONSM(X);
// Wenn X die Quellaktivität einer Sync-Kante ist, speichern wir sie für spätere Überprüfung für
// Sync-Kanten in einer neue Menge S.
        if SYNC-KANTE(X) SPEICHERN_S-K(X);

// Überprüfung für die Constraints mit Position notSpecified:
// Weil die Aktivitäten in UND-Block wie auf serielle Struktur immer ausgeführt werden und die
// Ausführungsreihenfolge für Constraint mit Position notSpecified irrelevant ist, ist die Überprüfung für
// sie in einem UND-Block dieselbe wie die für serielle Struktur.
        if notS_dep_S(X) M(X);
        if notS_dep_T(X) LÖSCHEN(X);
```

```
        if notS_exc_S(X) M(X);
        if notS_exc_T(X) M(X);

// Überprüfung für die Constraints mit Position post:
// Markierung von S-A für die Constraints des Typs (dependsOn, X, Y, post)
        if post_dep_S(X) M(X);
// Wenn die T-A nach S-A auf denselben Zweig gefunden wird (S-T-4), oder wenn S-A vor dem
// UND-Block vorkommt (S-T-1), ist das Constraint (dependsOn, X, Y, post) erfüllt.
        if post_dep_T(X)
                if (BRANCHNR(X)==BRANCHNR(S-T(X)))          // S-T-4
                OR ((MIT_MARKIERUNG(S-T(X)))AND(BRANCHNR(S-T(X)==Null)))
                        LÖSCHEN(X);                         // S-T-1
// Ein Constraint (excludes, X, Y, post) ist bei S-T-1, S-T-3, S-T-4 und S-T-6 verletzt:
        if post_exc_S(X)
        {       M(X);
                if (BRANCHNR(X)!=BRANCHNR(S-T(X)))AND(BRANCHNR(S-T(X))!=Null))
                        M(S-T(X));                          // Korrektur für S-T-3
        }
        if post_exc_T(X)
        {       if (BRANCHNR(X)!=BRANCHNR(S-T(X)))AND(BRANCHNR(S-T(X)!=Null))
                                                            // S-T-6
                OR (BRANCHNR(X)==BRANCHNR(S-T(X)))
                                                            // S-T-4
                OR ((BRANCHNR(S-T(X)==Null)AND(MIT_MARKIERUNG(S-T(X))))
                                                            // S-T-1
                        M(X);
        }

// Überprüfung für die Constraints mit Position pre:
// Bei S-T-2, S-T-5 und S-T-8 ist das Constraint (dependsOn, X, Y, pre) erfüllt. Bei S-T-2 wurde die
// Verletzungsmarkierung von T-A jedoch bereits bei der Überprüfung von T-A aufgehoben.
        if pre_dep_S(X)
        {       M(X);
                if (BRANCHNR(X)!=BRANCHNR(S-T(X)))AND(BRANCHNR(S-T(X)!=Null))
                        M(S-T(X));                          // Korrektur für S-T-3, vgl. Abb. 4.21
        }
        if pre_dep_T(X)
                if NOT(MIT_MARKIERUNG(S-T(X)))              // S-T-5,S-T-3 und S-T-8
                        LÖSCHEN(X);
// Ein Constraint (excludes, X, Y, pre) ist bei S-T-2, S-T-3, S-T-5, S-T-6 und S-T-8 verletzt. Bei S-T-2
// wurde T-A bei der Überprüfung der seriellen Struktur bereits markiert.
        if pre_exc_S(X) M(X);
        if pre_exc_T(X)
                if (NOT(MIT_MARKIERUNG(S-T(X))))            // S-T-3, S-T-5 und S-T-8
                OR (BRANCHNR(X)!=BRANCHNR(S-T(X)))AND(BRANCHNR(S-T(X)!=Null))
                                                            // S-T-6
                        M(X);

// Hole die nächste Aktivität im UND-Block.
        Next_IN_BLOCK(X);
// Die Positionsmarkierungen werden erst nach der Überprüfung von Sync-Kante aufgehoben.
}
```

Im Überprüfungsergebnis nach dem Algorithmus ist die semantische Wirkung der Sync-Kanten noch nicht berücksichtigt. Nun betrachten wir den Überprüfungsalgorithmus für die Sync-Kanten. Dabei muss nicht der ganze UND-Block überprüft werden, weil die Sync-Kante die Ausführungsreihenfolge von den Aktivitäten auf zwei UND-Zweige nur teilweise bestimmt. In Abbildung 4.27 steht die Ausführungsreihenfolge zwischen {E, F} und {K} nach wie vor nicht fest. Für die Aktivitäten {I, J}

werden die Aktivitäten {G, H} durch die Sync-Kante (J, G) als Nachfolger festgestellt. Analog können die Aktivitäten {C, D} wegen der Sync-Kante (G, C) nur nach {I, J} und {E, F, G} ausgeführt werden. Die Menge der Aktivitäten, die sich zwischen der Spilt-Aktivität des UND-Blocks und der Quell-Aktivität der Sync-Kante befinden, nennen wir die Vorgängermenge der Sync-Kante, wie z.B. {I, J} bezüglich der Sync-Kante (J, G) und {E, F, G} bezüglich der Sync-Kante (G, C) in Abbildung 4.27. Und die Menge der Aktivitäten, die durch Sync-Kante(n) als Nachfolger von den Aktivitäten in der Vorgängermenge festgelegt werden, nennen wir die Nachfolgermenge, wie z.B. {G, H, C, D} bezüglich der Sync-Kante (J, G) und {C, D} bezüglich der Sync-Kante (G, C) in Abbildung 4.27.

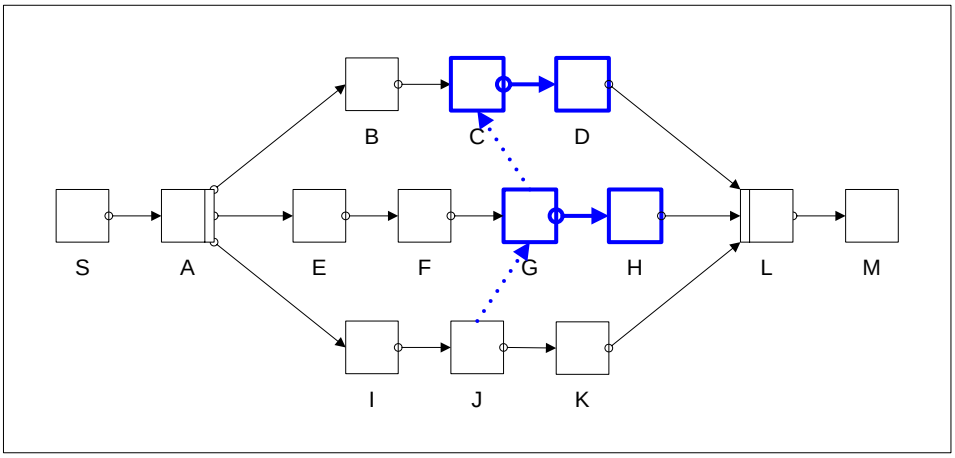


Abb. 4.27 Zwei korrelierende Sync-Kanten

Die Nachfolgermenge einer Sync-Kante enthält auch die Aktivitäten, die durch eine andere Sync-Kante als Nachfolger der Aktivitäten in der Vorgängermenge bestimmt werden. In unserem Beispiel sind dies die Aktivitäten {C, D}. Die Aktivitäten {C, D} werden bei unseren Algorithmus als Nachfolger von zwei unterschiedlichen Vorgängermengen {I, J} und {E, F, G} zweimal überprüft.

Bei Überprüfung einer Sync-Kante suchen wir nach bestimmten Konstellationen von Aktivitäten. Wie gesagt, die Sync-Kante hat keinen Einfluss auf die Constraints mit *Position = notSpecified*. Die anderen Constraints werden erfüllt, wenn die gewünschte Ausführungsreihenfolge von S-A und T-A durch die Sync-Kanten festgestellt oder die ungewünschte Ausführungsreihenfolge dadurch ausgeschlossen wird.

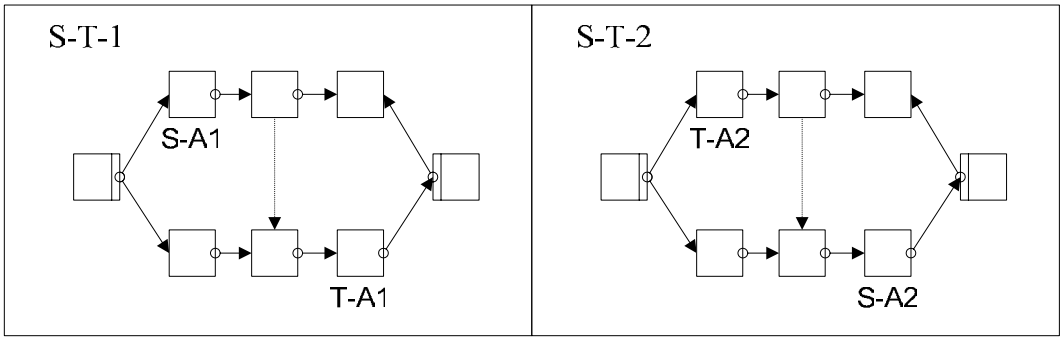


Abb. 4.28 Szenarien mit Sync-Kanten

Abbildung 4.28 stellt die Szenarien dar. Bei S-T-1 werden die Constraints von Typen *(dependsOn, S-A, T-A, post)* und *(excludes, S-A, T-A, pre)* erfüllt, weil mit der Sync-Kante die T-A nur nach S-A ausgeführt werden kann. Analog werden bei S-T-2 die Constraints von Typen *(dependsOn, S-A, T-A, pre)* und *(excludes, S-A, T-A, post)* erfüllt. Wir überprüfen daher, ob diese Szenarien vorgekommen sind oder nicht. Dafür suchen wir die T-A der Constraints von Typen *(dependsOn, S-A, T-A, post)* und *(excludes, S-A, T-A, pre)* in der Nachfolgermenge einer Sync-Kante. Wenn sie gefunden wird, überprüfen wir, ob die entsprechenden S-A sich in der Vorgängermenge dieser Sync-Kante befinden. Dazu sind die noch nicht gelöschten Zweignummern unerlässlich. Ebenfalls suchen wir die S-A der Constraints von Typen *(dependsOn, S-A, T-A, pre)* und *(excludes, S-A, T-A, post)* in der Nachfolgemenge einer Sync-Kante und die entsprechenden T-A in der Vorgängermenge. Wenn wir solche Konstellationen von S-A und T-A

finden, löschen wir die Verletzungsmarkierung von den entsprechenden T-A, um das Überprüfungsergebnis zu korrigieren. Da wir uns nur dafür interessieren, ob sich eine Aktivität in einer der Zwei Mengen einer Sync-Kante befindet, ist die Überprüfungsreihenfolge der Aktivitäten in der Nachfolgermenge irrelevant.

Definitionen:

NEXT_S(X) : Funktion, die die nächste Quell-Aktivität einer Sync-Kante in der Menge S, die wir bei der Überprüfung des UND-Blocks erzeugt haben, zurückgibt. Wenn alle Aktivitäten in der Menge S zurückgegeben wurden, liefert die Funktion *Null* zurück.

NEXT_N(X) : Funktion, die die nächste zu überprüfende Aktivität in der Nachfolgermenge einer Sync-Kante holt. Zu der Nachfolgermenge gehören alle Aktivitäten in dem UND-Block, die durch Sync-Kanten als Nachfolger von den Aktivitäten in der Vorgängermenge festgestellt werden. Wie gesagt, wenn im Teilzweig nach der aktuellen zu überprüfenden Sync-Kante eine Quell-Aktivität einer neuen Sync-Kante gefunden wird, müssen die Nachfolger der neuen Sync-Kante auch als Nachfolger der aktuelle zu überprüfende Sync-Kante betrachtet und überprüft werden.

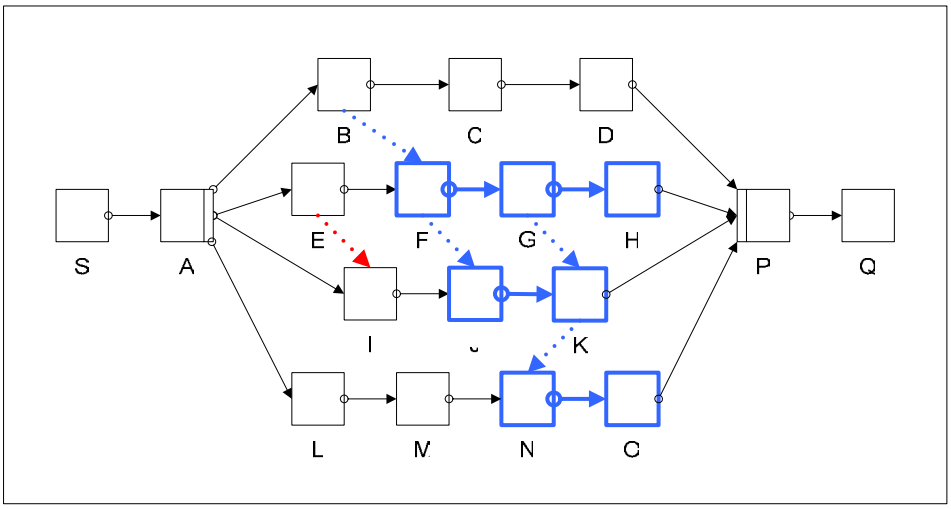


Abb. 4.29 Bestimmung der Nachfolgermenge

In Abbildung 4.29 gibt es drei Sync-Kanten zwischen den zwei mittleren Zweigen. Für die Überprüfung der Sync-Kante (B, F) ist die Sync-Kante (E, I) irrelevant, weil die Quell-Aktivität E sich nicht in der Nachfolgermenge der Sync-Kante (B, F) befindet. Zu der Nachfolgermenge der Sync-Kante (B, F) gehören die Aktivitäten {F, G, H, J, K, N, O}. Die Aktivitäten {N, O} sind wegen der Sync-Kanten (K, N) und (F, J) (oder (G, K)) auch Elemente in der Nachfolgermenge der Sync-Kante (B, F). Wenn wir die nächste zu überprüfende Aktivität bei der Überprüfung echtzeitig anhand des Prozess bestimmen, können die „parallelen“ Sync-Kanten (die Sync-Kanten zwischen zwei gleichen Zweigen, wie z.B. (F, J) und (G, K) in Abbildung 4.29) zur redundanten Überprüfung führen. In unserem Beispiel können die Aktivitäten {K, N, O} zweimal als Nachfolger der Sync-Kante (B, F) überprüft werden. Weil die Überprüfungsreihenfolge von den Aktivitäten in der Nachfolgermenge irrelevant ist, können wir das Problem dadurch lösen, dass wir zuerst eine vollständige Nachfolgermenge erzeugen. Bei Aufruf der Funktion ***NEXT_N(X)*** holen wir einfach ein beliebiges Element aus der Menge. Wenn alle Elemente abgearbeitet wurden, liefert die Funktion *Null* zurück.

VORG(X) : Funktion, die überprüft ob sich die Aktivität X in der Vorgängermenge der aktuell zu überprüfenden Sync-Kante befindet.

Wenn wir die Überprüfung der Sync-Kante separat nach der Überprüfung eines UND-Blocks machen, müssen wir den Pfad im UND-Block, der abhängig von der Sync-Kante ist, einmal durchzulaufen, um die Vorgängermenge und die Nachfolgermenge zu bestimmen. Zur Überprüfung der Sync-Kante (F, C) in Abbildung 4.30, suchen wir zuerst die Spilt-Aktivität des UND-Blocks. In Abbildung 4.30 ist sie die Aktivität A. Dann laufen wir den Teilzweig zwischen der Spilt-Aktivität und der Quell-Aktivität einmal durch und nehmen die Aktivitäten dazwischen (die Quell-Aktivität einschließend) als Elemente der Vorgängermenge. Dabei müssen wir zuerst den Zweig, auf dem sich die Quell-Aktivität der Sync-kante befindet, finden. Wir können die Zweignummer von der Quell-Aktivität der Sync-Kante und der ersten

Aktivität auf einen UND-Zweig vergleich. Für die Sync-Kante (F,J) gilt *BRANCHNR(F) = BRANCHNR(E)*. Die Vorgängermenge von (F, C) ist daher {E, F}.

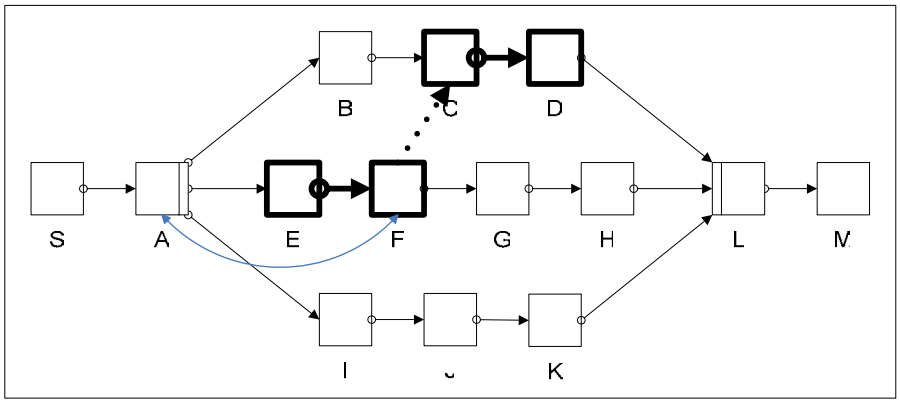


Abb. 4.30 Der relevante Pfad der Sync-Kante (F, C)

Die Elemente zwischen der Ziel-Aktivität der Sync-Kante und der Join-Aktivität des UND-Blocks gehören zu der Nachfolgermenge. Beispielsweise ist {C, D} die Nachfolgermenge der Sync-Kante (F, C) in Abbildung 4.30. Wenn darin wieder Quell-Aktivität einer anderen Sync-Kante auftaucht, zählen die Elemente der Nachfolgermenge der neuen Sync-Kante auch zu Nachfolger der aktuell zu überprüfenden Sync-Kante. Deswegen müssen sie auch in der Nachfolgermenge der aktuell zu überprüfenden Sync-Kante enthalten sein (vgl. Abbildung 4.29).

Der Pseudocode des Algorithmus von Stufe II für die Überprüfung der Sync-Kante sieht wie folgt aus.

```
// Für die erste Sync-Kante
NEXT_S(X);
// Überprüfung aller Sync-Kanten
FOR (X!=Null)
{
// Hole die erste zu überprüfende Aktivität in der Nachfolgermenge.
        NEXT_N(X);
// Überprüfung aller Aktivitäten in der Nachfolgermenge der aktuell überprüft Sync-Kante
FOR (X!=Null)
{
// Überprüfung für S-T-1 für die Constraints von Typen (dependsOn, S-A, T-A, post)
        if post_dep_T(X)
                if (VORG(S-T(X)))
                        LÖSCHEN(X);
// Überprüfung für S-T-1 für die Constraints von Typen (excludes, S-A, T-A, pre)
        if pre_exc_T(X)
                if (VORG(S-T(X)))
                        LÖSCHEN(X);
// Überprüfung für S-T-2 für die Constraints von Typen (dependsOn, S-A, T-A, pre)
        if pre_dep_S(X)
                if (VORG(S-T(X)))
                        LÖSCHEN(S-T(X));
// Überprüfung für S-T-2 für die Constraints von Typen ((excludes, S-A, T-A, post)
        if post_exc_S(X)
                if (VORG(S-T(X)))
                        LÖSCHEN(S-T(X));
// Holt das nächste Element in der Nachfolgermenge.
        NEXT_N(X);
}
// Die nächste Sync-Kante
NEXT_S(X)
}
```

// Nach der Überprüfung der Sync-Kanten löschen wir die Positionsmarkierungen (die Zweignummer)
L_POSITIONSM;

Mit der Überprüfung von Sync-Kanten ist unser Algorithmus für einen einfachen UND-Block nun vollständig. Wir zeigen im Folgenden ein Beispiel, mit dem die Überprüfungsschritte eines UND-Blocks veranschaulicht werden.

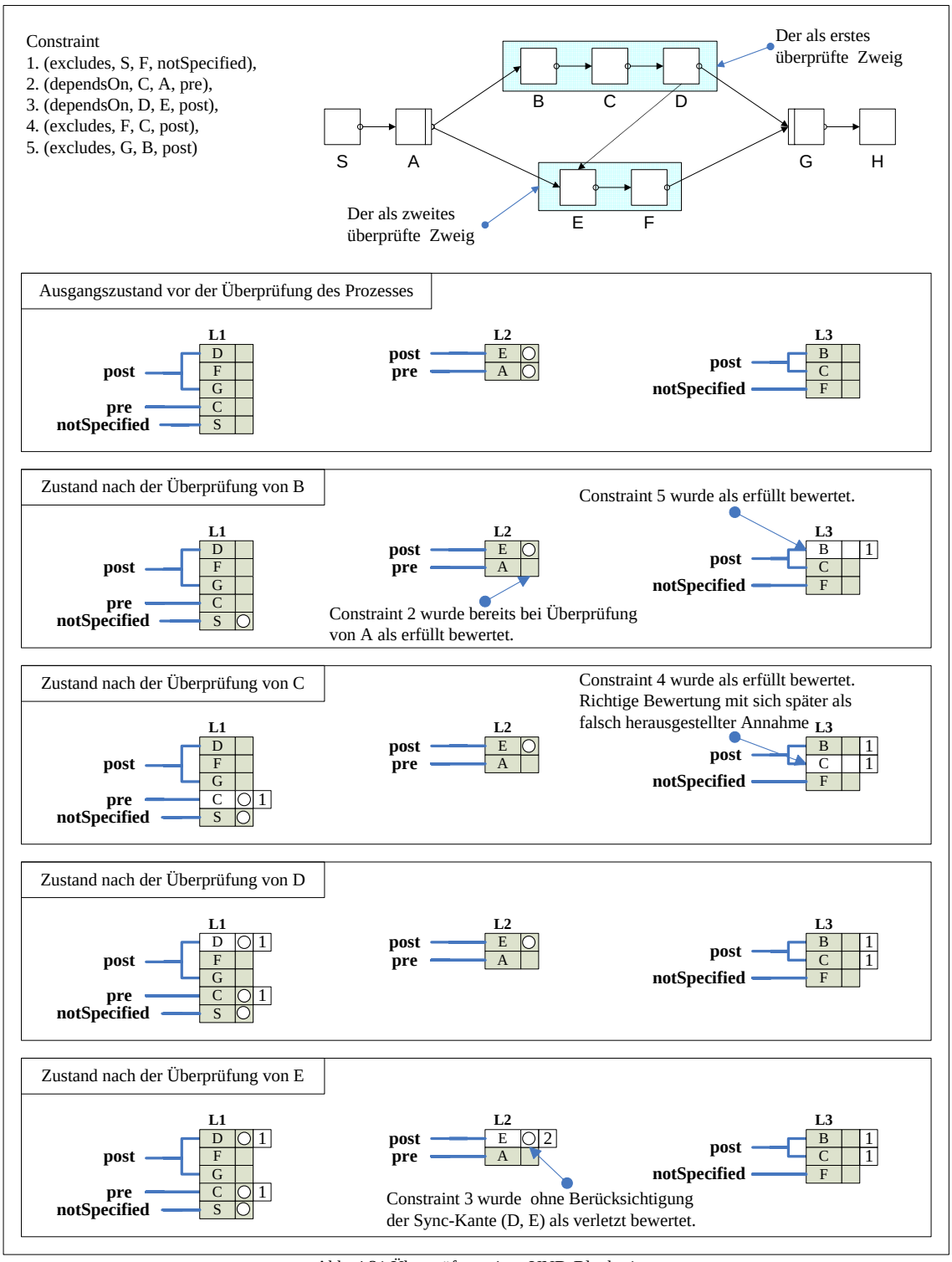


Abb. 4.31 Überprüfung eines UND-Blocks 1

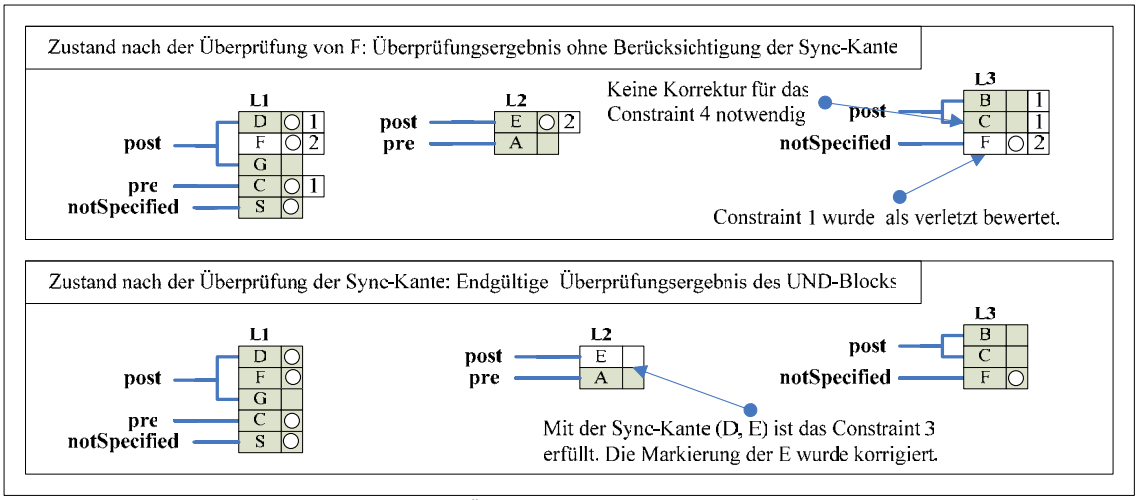


Abb. 4.32 Überprüfung eines UND-Blocks 2

4.3.3 Überprüfung für gemischt geschachtelte Blöcke

Zur Gruppe gemischt verschachtelte Blöcke gehören alle Block-Strukturen außer den einfachen Blöcken. Wir wollen in diesem Abschnitt einen allgemein gültigen Algorithmus für alle Block-Strukturen entwickeln: der Algorithmus von Stufe III. In komplexen Block-Strukturen mit UND-Block können auch 4für sie lässt sich ebenfalls nach der Überprüfung reiner Block-Struktur ohne Sync-Kanten machen. Deshalb überprüfen wir sie auch separat im Nachhinein.

Bei den gemischt verschachtelten Blöcken gibt es unendliche Möglichkeiten von Konstellationen von S-A und T-A, weil die Blöcke beliebig verschachtelt sein können. Es ist daher unmöglich, alle Szenarien wie bisher aufzulisten. Allerdings werden wir sie in fünf Gruppen kategorisieren. Aber zuerst müssen wir die Aktivitäten in den komplexen Block-Strukturen positionieren. Dazu reichen die Zweignummern allein offensichtlich nicht mehr aus. Deshalb erzeugen wir mit einer Erweiterung der Positionierungsmarkierung Markierungen von Stufe III.

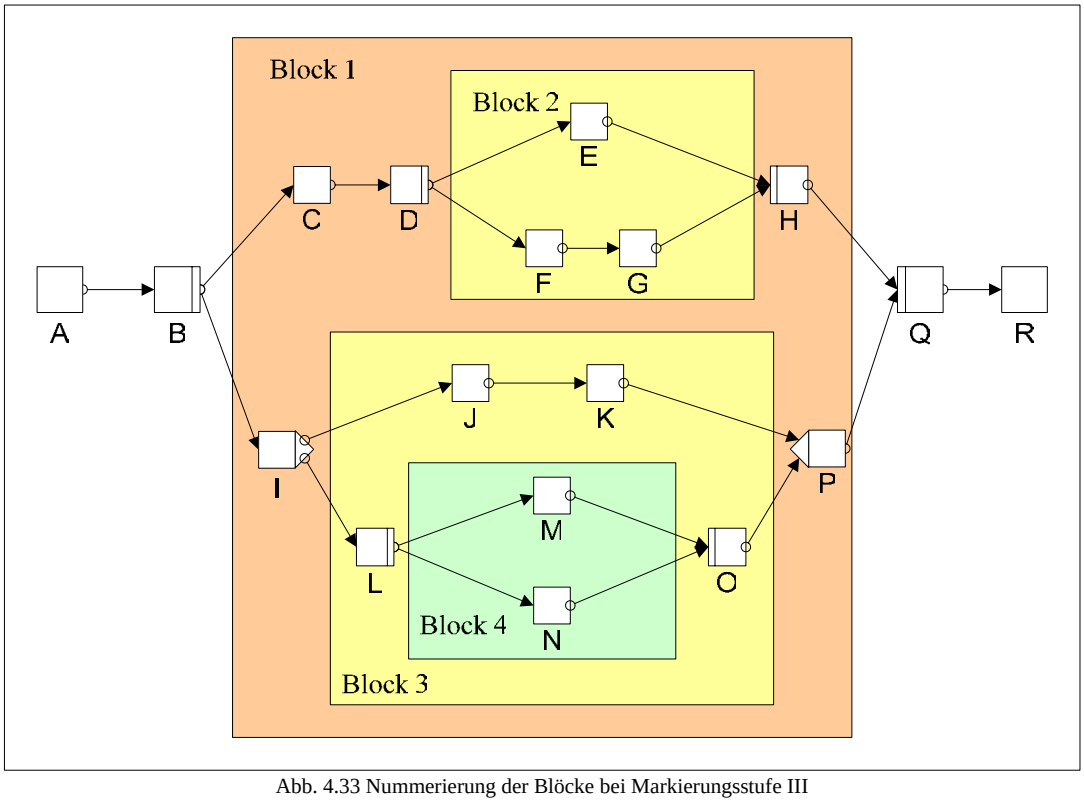


Abb. 4.33 Nummerierung der Blöcke bei Markierungsstufe III

Die Positionierungsmarkierung von Markierungsstufe III ist eine Liste von Tupel des Typs *(Blocknummer, Blocktyp, Zweignummer)*. Zu dem Parameter *Blocktyp* gehören zwei Stichwörter: *U* für UND-Block und *X* für XOR-Block (Dies kann man mit einer booleschen Variable implementieren). Die Erzeugung der *Zweignummer* ist wie bei einfachen Blöcken. Wir zählen die Zweige eines neuen Blocks immer erneut von eins an. Die Nummerierung der Blöcke in komplexen Block-Strukturen ist abhängig von der Überprüfungsreihenfolge. Der Block wird mit *Blocknummer* eins nummeriert, wenn seine Split-Aktivität bei der Überprüfung einer komplexen Block-Struktur zuerst gelesen wird. Der Block mit der als zweites gelesenen Split-Aktivität wird mit 2 nummeriert, usw. Abbildung 4.33 zeigt die Nummerierung der Blöcke, wenn wir die Zweige des Blocks von oben nach unten überprüfen.

Bei der Überprüfung werden wir zuerst einen Block vollständig überprüfen, dann weiter gehen. Die Namen der Aktivitäten in Abbildung 4.33 geben diese Überprüfungsreihenfolge wieder. Allerdings bleibt die Überprüfungsreihenfolge der Zweige eines Blocks bei unserem Algorithmus nach wie vor beliebig wählbar. Im Prinzip können wir die Aktivitäten in komplexer Block-Struktur auch in beliebiger Reihenfolge überprüfen. Jedoch ist es günstig für die Erzeugung der Positionsmarkierungen, dass wir immer zuerst einen Block vollständig überprüfen.

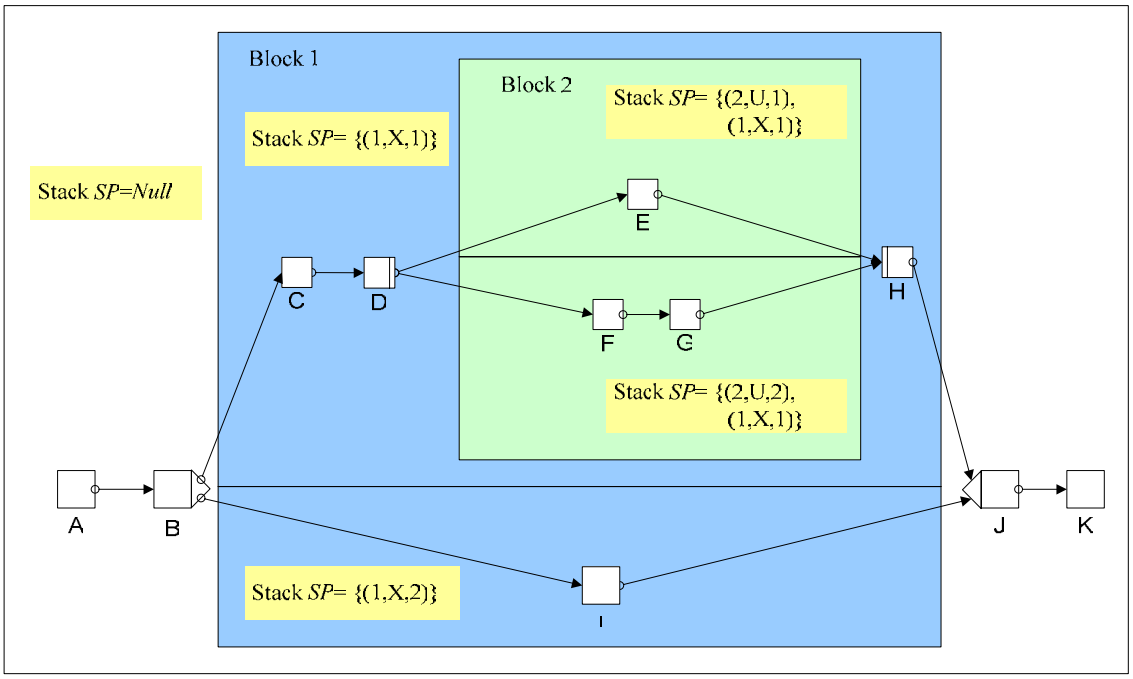


Abb. 4.34 Positionsmarkierungen bei Markierungsstufe III

Wie bei der Überprüfung der einfachen Blöcke, erzeugen wir die Positionsmarkierung einer Aktivität in komplexer Block-Struktur nicht im Voraus, sondern erst wenn wir die Aktivität bei der Überprüfung lesen. Die Positionsmarkierung einer Aktivität in komplexer Block-Struktur enthält die Informationen aller Blöcke, die diese Aktivität umschließen. Abbildung 4.34 illustriert ein Beispiel, um die dynamische Erzeugung der Positionsmarkierungen bei der Überprüfung zu verdeutlichen. Wir erzeugen einen globalen Stack *SP* als Speicher der Positionsmarkierungen. Bei Eintritt in einem neuen Block fügen wir ein Tupel mit *Blocknummer, Blocktyp* von dem Block und *Zweignummer* des aktuell zu überprüfenden Zweigs in dem Stack ein. Wenn wir den zu überprüfenden Zweig wechseln, ändern wir gleichzeitig die *Zweignummer* vom obersten Tupel in *SP*. Bei Austritt eines Blocks löschen wir das Tupel aus *SP*. Die Tupellist in Stack *SP* entspricht der Positionsmarkierung der aktuell zu überprüfenden Aktivität. Beispielsweise haben die Aktivitäten F und G in Abbildung 4.34 die gleichen Positionsmarkierungen: *{(2,U,2), (1,X,1)}*, weil sie sich in demselben Block befinden. In einer so erzeugten Positionsmarkierung steht das Tupel eines äußeren Blocks mit Sicherheit unterhalb des Tupel eines inneren Blocks im Stack. Diese Reihenfolge ist essentiell für die spätere Analyse.

Mit dem Markierungssystem können wir jetzt die Konstellationen von zwei Aktivitäten in komplexer Block-Struktur feststellen. Durch Analyse der Positionsmarkierungen von den S-A und T-A eines

Constraints in komplexer Block-Struktur lassen sich die Szenarien in fünf Gruppen klassifizieren. Wir suchen zuerst das oberste paar von Tupeln, die die gleiche *Blocknummer* und *Blocktyp* besitzen, in den Positionsmarkierungen von den zwei Aktivitäten. Wir nennen die zwei Tupeln *OT (oberste Tupel).* Alle Tupeln in der Positionsmarkierung, die sich unterhalb vom *OT* befinden, sind für die semantische Überprüfung des Constraints irrelevant.

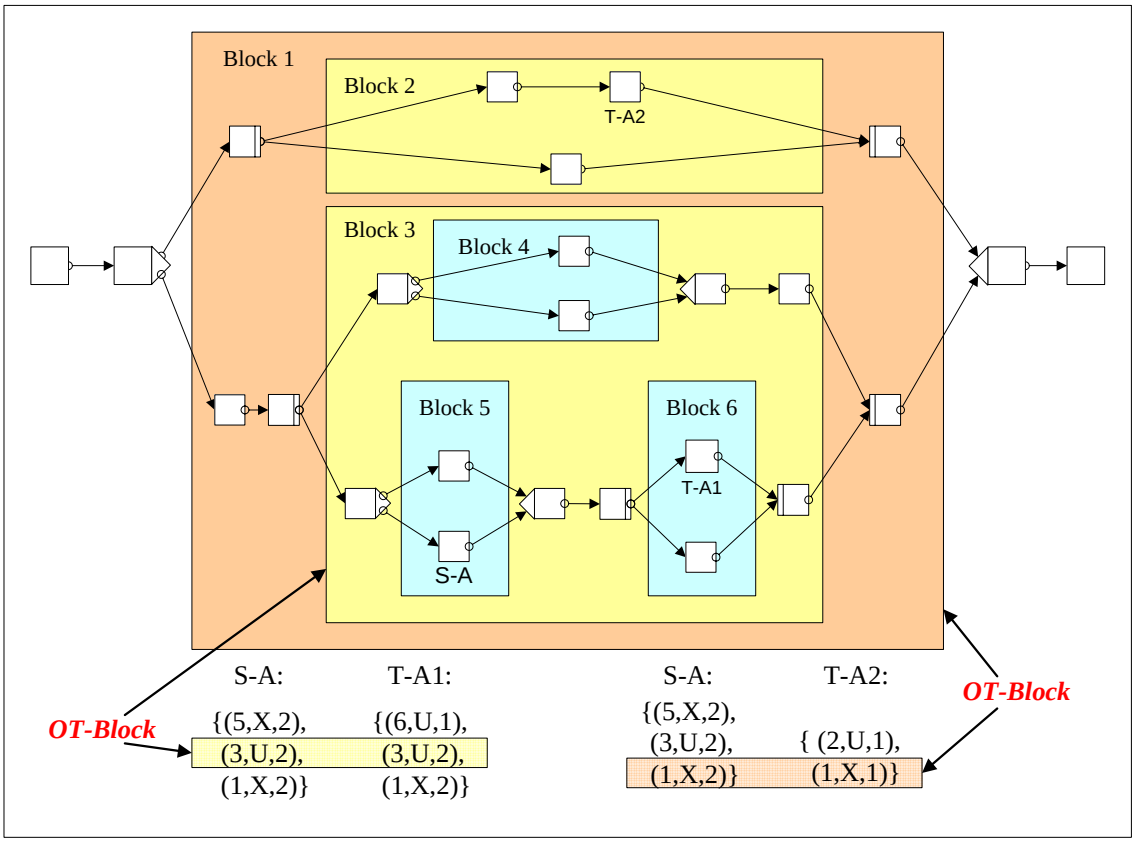


Abb. 4.35 Bestimmung der *OT*

In Abbildung 4.35 besitzt die S-A die Positionsmarkierung *{(5,X,2),(3,U,2),(1,X,2)}* und die T-A1 wird mit *{(6,U,1),(3,U,2),(1,X,2)}* markiert. Die *OT* für die beide sind gleich: das Tupel *(3, U, 2)*. Allerdings können die zwei *OT* unterschiedliche Zweignummern besitzen, wie z.B. die *OT* von S-A und T-A2. Dabei ist *OP* von S-A *(1,X,2)* und *OP* von T-A2 ist *(1,X,1)*. Durch Abbildung 4.35 wird veranschaulicht, dass die *OT* dem innersten umschließenden Block von den zwei Aktivitäten entspricht (Block 3 für S-A und T-A1 und Block 1 für S-A und T-A2). Wir nennen den Block der *OT-Block*. Daher sind die Tupeln unterhalb *OT* in der Positionsmarkierung irrelevant bei der Überprüfung. Sie entsprechen nämlich den Blöcken, die außerhalb vom *OT-Block* sind, wie z.B. der Block 1 für S-A und T-A1.

Jetzt betrachten wir die fünf Szenarien von zwei Aktivitäten innerhalb komplexer Block-Struktur. Wenn das *OT* das oberste Tupel der Positionsmarkierung bei den beiden Aktivitäten ist, befinden sich die zwei Aktivitäten dann direkt auf Zweigen desselben Blocks. Die Block-Struktur außerhalb oder innerhalb des *OT-Block* ist für die semantische Überprüfung des entsprechenden Constraints irrelevant.

Wir unterscheiden den Fall anhand des Parameters *Blocktyp* von *OT*. *OT* mit *U* als *Blocktyp* (d.h. der *OT-Block* ein UND-Block ist) bedeutet, dass S-A und T-A sich direkt auf Zweigen eines UND-Blocks befinden, wie z.B. F und G oder C und I in Abbildung 4.33. Dabei ist die Überprüfung analog wie die Überprüfung von zwei Aktivitäten in einem einfachen UND-Block (vgl. Szenarien 1 in Abbildung 4.36).

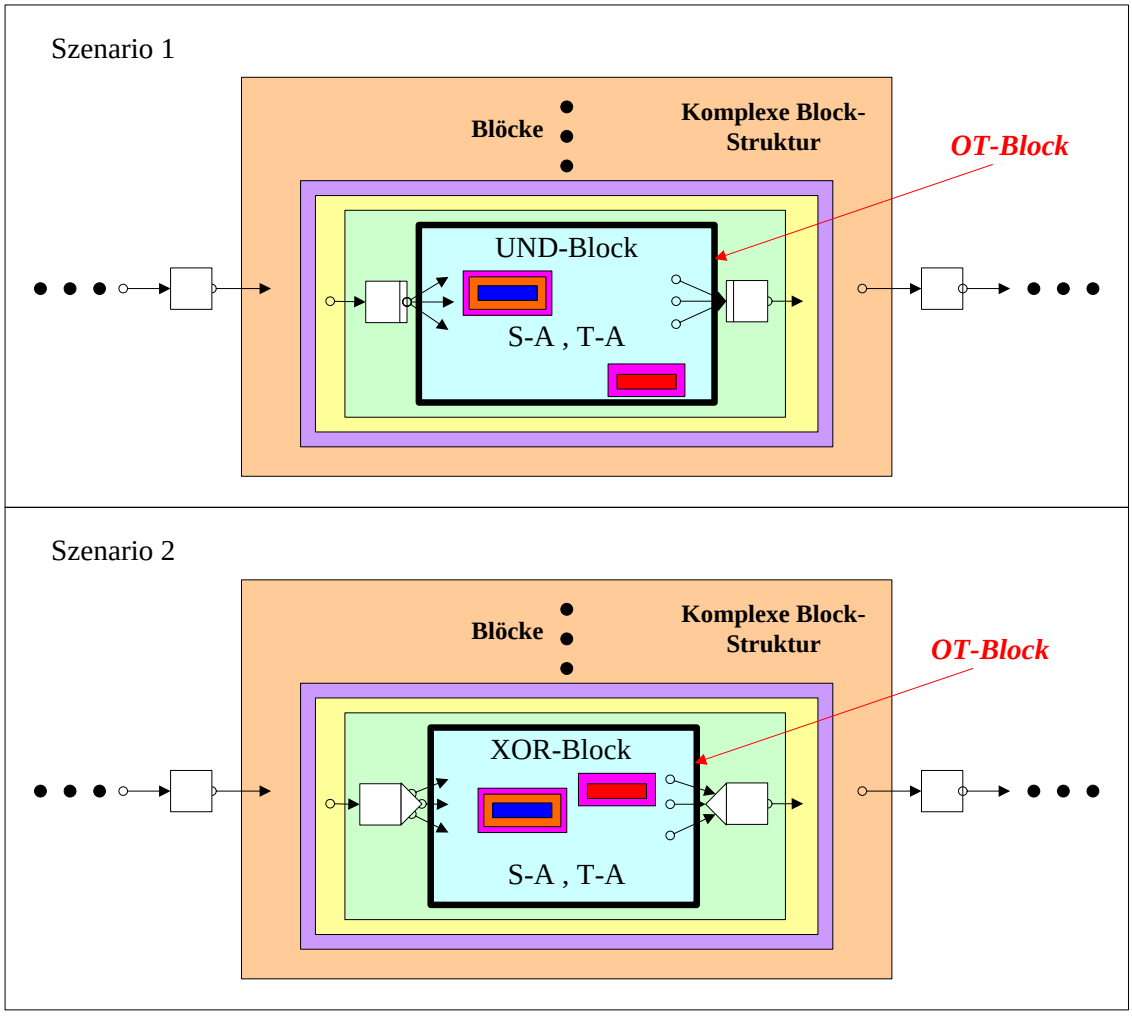


Abb. 4.36 Szenario 1 und 2: einziger relevanter Block: *OT-Block*

Wenn das *OT* das oberste Tupel der Positionsmarkierung bei den beiden Aktivitäten ist und *OT X* als *Blocktyp* (d.h. der *OT-Block* ein XOR-Block ist) haben, befinden sich dann die zwei Aktivitäten direkt auf Zweigen eines XOR-Blocks, wie z.B. C und I in Abbildung 4.34. Analog kann man bei der Überprüfung alle anderen Blöcke außer dem *OT-Block* ignorieren (vgl. Szenarien 2 in Abbildung 4.36). Dabei ist die Überprüfung analog wie die Überprüfung von zwei Aktivitäten in einem einfachen XOR-Block.

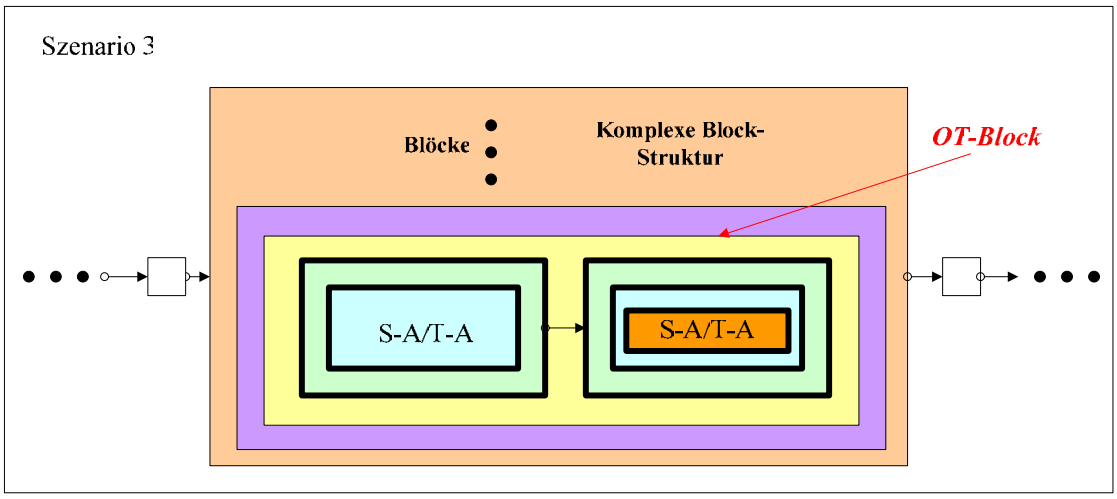


Abb. 4.37 Szenario 3: Serielle Block-Strukturen für S-A und T-A

Wenn *OT* nicht das oberste Tupel der beiden Positionsmarkierung sind, befinden sich die beiden Aktivitäten nicht direkt auf Zweigen eines Blocks. In dem Fall vergleichen wir die Zweignummern der *OT*. Falls sie gleich sind (d.h. die beiden *OT* völlig identisch sind), befinden sich die zwei Aktivitäten dann in zwei separaten Blockstrukturen, die sich aufeinander folgend auf denselben Zweig von *OT-Block* befinden, wie z.B. S-A und T-A1 in Abbildung 4.35 (vgl. Szenario 3 in Abbildung 4.37). Um die Ausführungen von T-A beim Szenario 3 zu kontrollieren, ermitteln wir jeweils die Tupeln, die oberhalb von *OT* in den Positionsmarkierung von T-A stehen. Diese Tupeln nennen wir *TOOT (Tupeln oberhalb von OT)*. In Abbildung 4.37 werden die Blöcke, die *TOOT* entsprechen, mit fetten Linien bezeichnet. Wir überprüfen dabei, ob ein Tupel mit *Blocktyp=X* in *TOOT* vorkommt, d.h. ob ein XOR-Block, die die Aktivität umschließt, innerhalb von *OT-Block* vorkommt.

Hierzu illustrieren wir ein Beispiel. In Abbildung 4.38 wird M mit {(4,U,2),(3,X,2),(1,U,2)} und S mit {(5,U,2),(1,U,2)} markiert. Dabei sind die *OT* von den beiden Aktivitäten gleich: (1,U,2). Die *TOOT* von M sind {(4,U,2),(3,X,2)} und die *TOOT* von S ist {(5,U,2)}. In *TOOT* von M wird ein Tupel mit *Blocktyp=X* auftaucht: das Tupel (3,X,2). Deswegen wissen wir, dass M nicht jedes Mal ausgeführt wird, wenn S ausgeführt wird. Das Constraint *(dependsOn, S, M, pre)* wäre daher verletzt auf dem Prozess.

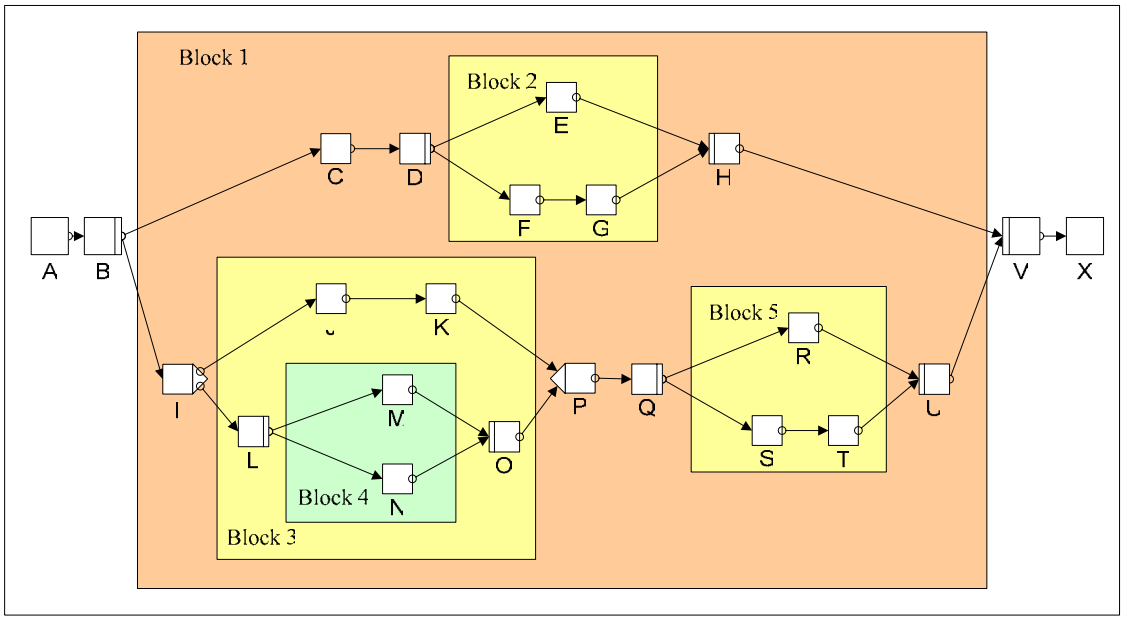


Abb. 4.38 Beispiel für die Szenarien

Zum Szenario 3 gehören auch die Fälle, wo *TOOT* einer Aktivität von den beiden leer ist, d.h. ein *OT* befindet sich oberst in der Positionsmarkierung. Strukturell bedeutet es, dass eine von den beiden Aktivitäten sich direkt auf den Zweig von *OT-Block* befindet, wie z.B. Q und N in Abbildung 4.38, wobei Q direkt auf den Zweig von *OT-Block* befindet. Wir wissen daher diese Aktivität immer ausgeführt wird, wenn die andere Aktivität ausgeführt wird. Eben besitzt leer *TOOT* kein Tupel mit *Blocktyp=X*.

Wenn *OT* nicht das oberste Tupel der beiden Positionsmarkierung sind und die Zweignummern der *OT* ungleich sind, befinden die zwei Aktivitäten dann in zwei separaten Block-Strukturen, die sich auf zwei unterschiedlichen Zweige eines Blocks befinden. In dem Fall betrachten wir die Parameters *Blocktyp* von *OT*.

Wenn *Blocktyp=X* befinden sich die zwei Aktivitäten in zwei separaten Block-Strukturen, die sich auf zwei unterschiedlichen Zweigen eines XOR-Blocks befinden (vgl. Abbildung 4.39). Bei der Konstellation können die beiden Aktivitäten nie zusammen ausgeführt werden. Daher brauchen wir die *TOOT* von den beiden Aktivitäten bei dem Szenario 4 nicht zu analysieren.

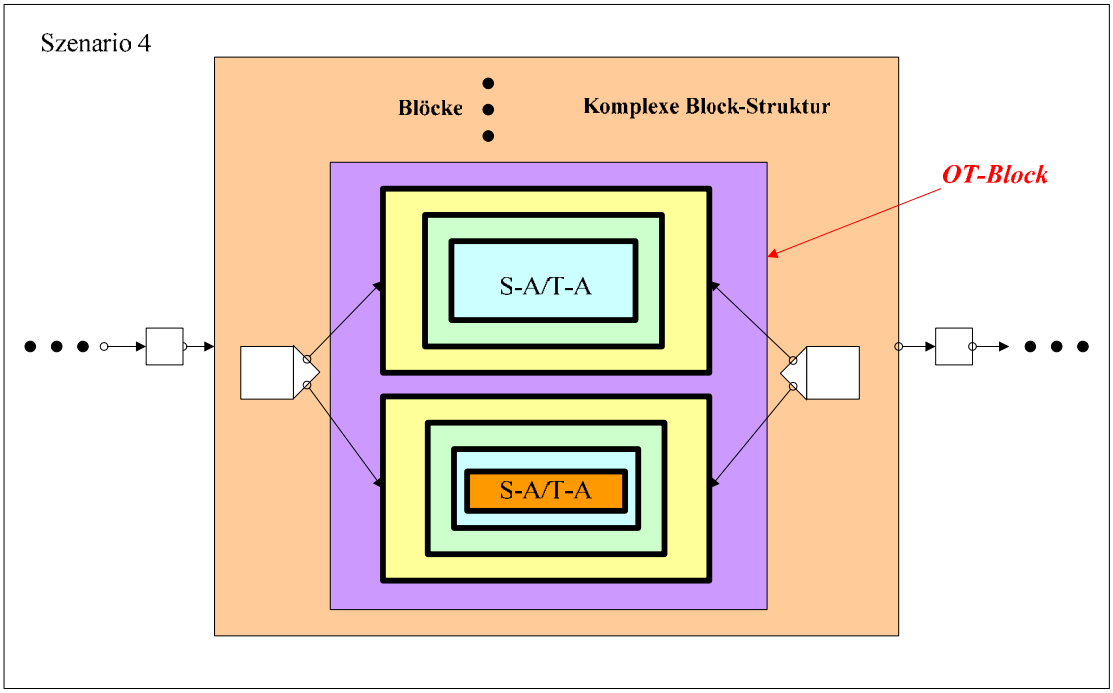


Abb. 4.39 Szenario 4: umschließender XOR-Block

Wenn *Blocktyp=U* befinden sich die zwei Aktivitäten in zwei separaten Block-Strukturen, die sich auf zwei unterschiedlichen Zweige eines UND-Blocks befinden (vgl. Abbildung 4.40). Beim Szenario 5 müssen wir wie beim Szenario 3 die *TOOT* von T-A analysieren, um die Ausführung von T-A zu kontrollieren. Dabei suchen Wir ebenfalls nach Tupeln mit *Blocktyp=X* in *TOOT*, jedoch nur für T-A von Constraints von Abhängigkeitsbeziehung. Wenn S-A und T-A eines Constraints von Ausschlussbeziehung diese Konstellation besitzen, ist das Constraint verletzt, egal ob T-A immer ausgeführt wird oder nicht.

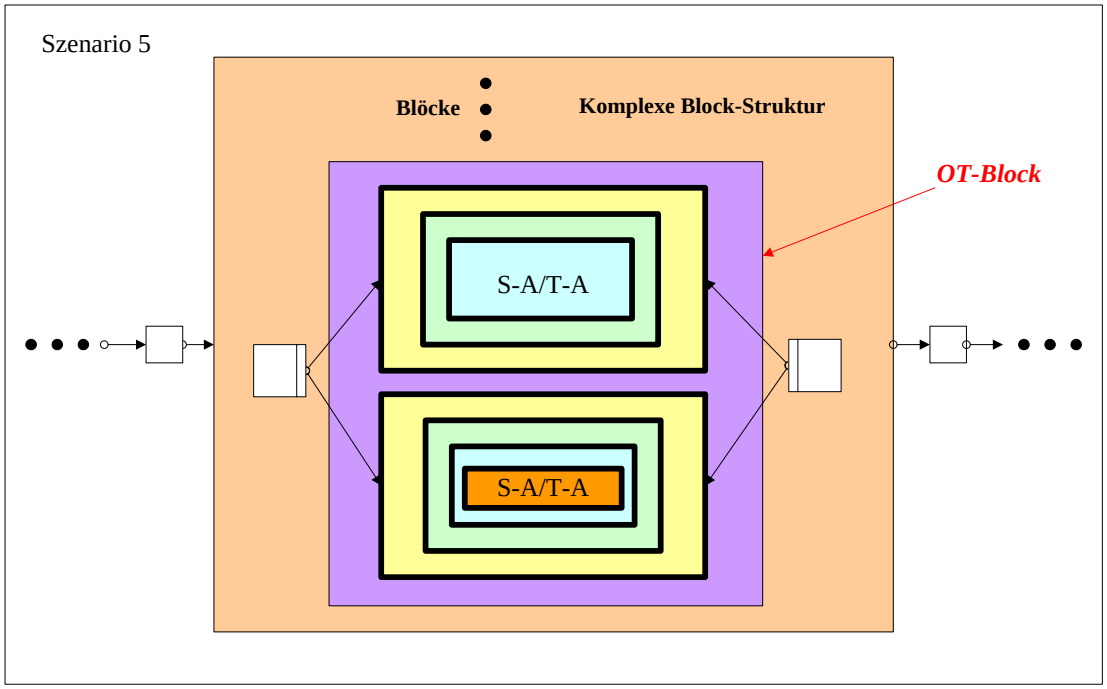


Abb. 4.40 Szenario 5: umschließender UND-Block

Die fünf Szenarien stehen für Konstellationen von S-A und T-A, die beide in komplexer Block-Struktur vorkommen. Bei den Szenarien 1 und 2 befinden die zwei Aktivitäten direkt auf den Zweigen desselben

Blocks. Wir vergleichen hier die Zweignummern von *OT*, um festzustellen, ob S-A und T-A sich auf demselben Zweig befinden. Beim Szenario 3 tauchen die S-A und T-A nacheinander in zwei separaten Block-Strukturen auf. Hier müssen wir die *TOOT* analysieren, bevor eine Bewertung des Constraints gemacht werden können. Beim Szenario 4 werden die beiden Aktivitäten nie zusammen ausgeführt. Constraints für Ausschlussbeziehung sind dabei erfüllt. Dagegen sind Constraints für Abhängigkeitsbeziehung beim Szenario 4 verletzt. Beim Szenario 5 könnten die beiden Aktivitäten ausgeführt werden. Um die Ausführungen zu kontrollieren, überprüfen wir beim Szenario 5 ebenfalls, ob ein Tupel mit *Blocktyp=X* in *TOOT* von T-A vorkommt.

Selbstverständlich kann auch nur eine von S-A und T-A in komplexer Block-Struktur auftauchen. Es gibt vier Szenarien dafür (vgl. 4.41).

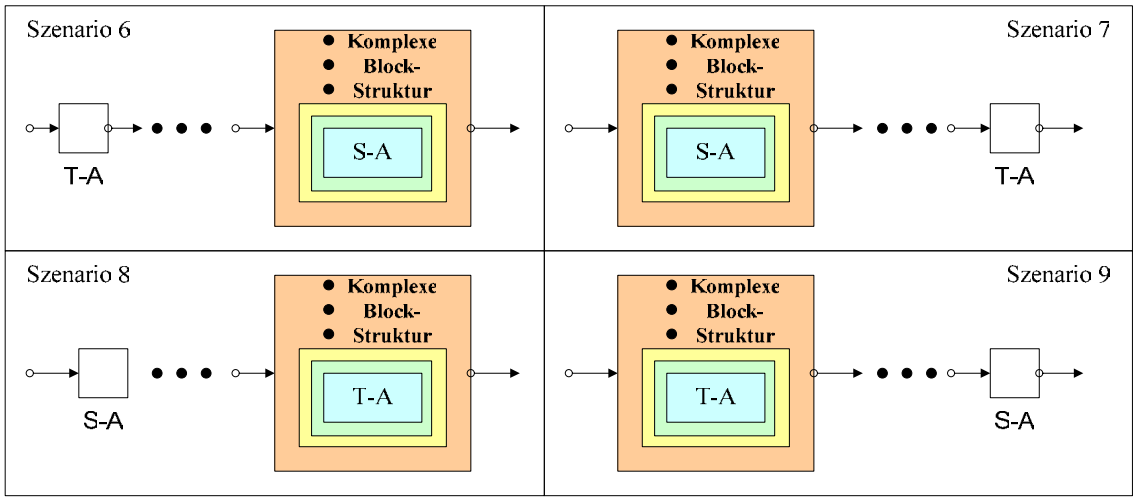


Abb. 4.41 Szenarien für einzelne Aktivität in komplexer Block-Struktur

Für allein in komplexer Block-Struktur vorkommende S-A (Szenario 6 und 7 in Abbildung 4.41), machen wir nichts anderes als die S-A zu markieren. Die Bewertung des Constraints machen wir an der Stelle der T-A. Wie bei einfachen XOR-Blöcken, ist eine allein vorkommende T-A (Szenario 8 und 9 in Abbildung 4.41) von Ausschlussbeziehung in komplexer Block-Struktur semantisch effektiv. Dagegen ist eine allein vorkommende T-A von Abhängigkeitsbeziehung effektiv, nur wenn kein Tupel mit *Blocktyp=X* in der Positionsmarkierung auftaucht, d.h. wenn sie sich nicht in einem XOR-Block befindet und damit immer ausgeführt wird.

Der Algorithmus von Stufe III ist eine Verallgemeinerung des Algorithmus von Stufe II. Die Grundideen von den beiden sind dieselben. Wir treffen die Beurteilung über ein Constraint, wenn wir die T-A finden. Wenn dabei die S-A noch nicht vorgekommen (nicht markiert) ist, markieren wir die T-A an der Stelle so, als ob die S-A nicht in der ganzen komplexen Block-Struktur vorkommen würde (wie Szenario 9 in Abbildung 4.41). Wenn die S-A nachher aber darin gefunden würde, korrigieren wir den Markierungszustand der T-A, falls nötig.

Definitionen:

M_POSITIONSM3(X) : Funktion, die die Aktivität X in komplexer Block-Struktur auf jede Stellen in L1 L2 und L3 mit Positionsmarkierung von Markierungsstufe III markiert.
POSITIONSM3(X) : Funktion, die die Positionsmarkierung von Markierungsstufe III (eine Liste von Tupeln) von Aktivität X zurückliefert. Für die Aktivitäten die keine Positionsmarkierung besitzen, liefert sie *Null* zurück.
Next_IN_KBS(X) : Funktion, die die nächste zu überprüfende Aktivität X in der komplexen Block-Struktur holt (Überprüfungsreihenfolge vgl. Abbildung 4.33). Wenn alle Aktivitäten in der komplexen Block-Struktur überprüft wurden, liefert die Funktion *Null* zurück.
AUSFÜHRUNG_XOR(TL) : boolesche Funktion, die Überprüft, ob in der Tupellist ***TL*** ein Tupel mit *Blocktyp=X* auftaucht.
T : Variable, die ein Tupel speichert.
TOOT(X) : Funktion, die *TOOT* von der Aktivität ***X*** zurückliefert.

Konstellation(X,Y) : Funktion, die die Konstellationen der Aktivitäten X, Y, die sich beide in komplexer Block-Struktur befinden, feststellt. Sie liefert 1 zurück, wenn die Konstellation von X und Y wie beim Szenario 1 in Abbildung 4.36 ist. Sie liefert 2 für das Szenario 2 in Abbildung 4.36 zurück. Und Funktionswerte 3, 4, 5 stehen für Szenario 3, 4, 5.

Definierte Funktionen:

SYNC-KANTE(X) : boolesche Funktion, die überprüft, ob X die Quell-Aktivität einer Sync-Kante ist.

SPEICHERN_S-K(X) : Funktion, die X in der Menge S, eine Menge von Quell-Aktivitäten von Sync-Kante, speichert.

L_POSITIONSM : Funktion, die alle Positionsmarkierung in L1, L2 und L3 aufhebt.

Geänderte Funktion:

BRANCHNR(X) : Funktion, die die *Zweignummer* in dem obersten Tupel von Positionsmarkierung der Aktivität ***X*** zurückliefert. Wir verwenden diese Funktion, um festzustellen, ob zwei Aktivitäten sich beim Szenario 1 oder 2 direkt auf demselben Zweig eines Blocks befinden.

Der Algorithmus von Markierungsstufe III für komplexe Block-Struktur ohne Sync-Kante sieht im Pseudocode wie folgt aus:

Code:

```
// Überprüfung jeder Aktivität in komplexer Block-Struktur
Next_IN_KBS(X);
For (X!=Null)
{
// Jede Aktivität komplexer Block-Struktur wird mit Positionsmarkierung von Stufe III markiert.
        M_POSITIONSM3(X);
// Wenn X die Quellaktivität einer Sync-Kante ist, speichern wir sie für spätere Überprüfung für
// Sync-Kanten in einer Menge S.
        if SYNC-KANTE(X) SPEICHERN_S-K(X);

// Überprüfung für Constraints von Typ (dependsOn, X, Y, notSpecified):
// Operationen für S-A des Constraints
        if notS_dep_S(X)
        {       // Jede S-A wird markiert
                M(X);
        // Korrektur dafür, dass S-A nach T-A in der komplexen Block-Struktur gefunden wurde.
                if (POSITIONSM3(S-T(X))!=Null)
                {
                        SWITCH (Konstellation(X,S-T(X)))
                        { // Korrektur fürs Szenario 1, das Constraint ist dabei erfüllt
                          CASE 1: LÖSCHEN(S-T(X)); break;
                          // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                          // erfüllt beim Szenario 2.
                          CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))
                                                LÖSCHEN(S-T(X)); break;
                          // Kontrollieren der Ausführung von T-A beim Szenario 3.
                          CASE 3: if (NOT(AUSFÜHRUNG_XOR(TOOT(S-T(X))))
                                                LÖSCHEN(S-T(X));break;
                          // Constraint verletzt beim Szenario 4 => keine Operation
                          CASE 4: break;
                          // Kontrollieren der Ausführung von T-A beim Szenario 5.
                          CASE 5:
                        if (NOT(AUSFÜHRUNG_XOR(TOOT(S-T(X))))
                                                LÖSCHEN(X);break;
                        }
                }
        }
// Operationen für T-A des Constraints
```

```
if notS_dep_T(X)
{       // Wurde S-A schon gefunden?
        if NOT(MIT_MARKIERUNG(S-T(X)))
        {       // S-A wird noch nicht gefunden => Annahme: Szenario 9 in Abbildung 4.41
                // Wird T-A immer ausgeführt?
                if NOT(AUSFÜHRUNG_XOR(POSITIONSM3(X))
                // T-A wird immer ausgeführt => Constraint erfüllt
                        LÖSCHEN(X);
                // T-A wird nicht immer ausgeführt => Constraint verletzt (keine Operation)
        }
        // S-A schon gefunden.
        else
        {       // Befindet sich S-A vor der komplexen Block-Struktur (Szenario 8)?
                if BRANCHNR(S-T(X)==Null))
                // Wird T-A immer ausgeführt?
                        if NOT(AUSFÜHRUNG_XOR(POSITIONSM3(X))
                        // T-A wird immer ausgeführt => Constraint erfüllt
                                LÖSCHEN(X);
                        // T-A wird nicht immer ausgeführt
                        // => Constraint verletzt, keine Operation
                // S-A befindet sich in der komplexen Block-Struktur
                else
                SWITCH (Konstellation(X,S-T(X)))
                { // Constraint erfüllt beim Szenario 1
                  CASE 1: LÖSCHEN(X); break;
                  // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                  // erfüllt beim Szenario 2.
                  CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))LÖSCHEN(X);break;
                  // Kontrollieren der Ausführung von T-A beim Szenario 3.
                  CASE 3: if (NOT(AUSFÜHRUNG_XOR(TOOT(X))))LÖSCHEN(X);break;
                  // Constraint verletzt beim Szenario 4 => keine Operation
                  CASE 4: break;
                  // Kontrollieren der Ausführung von T-A beim Szenario 5.
                  CASE 5: if (NOT(AUSFÜHRUNG_XOR(TOOT(X))))LÖSCHEN(X);break;
                }
        }
}

// Überprüfung für Constraints von Typ (excludes, X, Y, notSpecified):
        if notS_exc_S(X)
        {       // Jede S-A wird markiert
                M(X);
// Korrektur dafür, dass S-A nach T-A in der komplexen Block-Struktur gefunden wurde.
                if (POSITIONSM3(S-T(X))!=Null)
                {
                        SWITCH (Konstellation(X,S-T(X)))
                        { // Constraint verletzt beim Szenario 1
                          CASE 1: M(S-T(X)); break;
                          // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                          // verletzt beim Szenario 2.
                          CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))
                                        M(S-T(X));break;
                          // Constraint verletzt beim Szenario 3
                          CASE 3: M(S-T(X)); break;
                          // Constraint erfüllt beim Szenario 4 => keine Operation
                          CASE 4: break;
                          // Constraint verletzt beim Szenario 5
                          CASE 5: M(S-T(X)); break;
```

```
                    }
                }
            }
            if notS_exc_T(X)
            {       // Wurde S-A bereits gefunden?
                    if NOT(MIT_MARKIERUNG(S-T(X)))
                    {       // S-A wird noch nicht gefunden => Annahme: Szenario 9 in Abbildung 4.41
                            // T-A und S-A können zusammen ausgeführt werden => Constraint verletzt
                            M(X);
                    }
                    // S-A schon gefunden.
                    else
                    {       // Befindet sich S-A vor der komplexen Block-Struktur (Szenario 8)?
                            if BRANCHNR(S-T(X)==Null))
                                    // => Constraint verletzt beim Szenario 8
                                    M(X);
                            // S-A befindet sich in der komplexen Block-Struktur
                            else
                            SWITCH (Konstellation(X,S-T(X)))
                            { // Constraint verletzt beim Szenario 1
                              CASE 1: M(X); break;
                              // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                              // verletzt beim Szenario 2.
                              CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))
                                              M(X);break;
                              // Constraint verletzt beim Szenario 3
                              CASE 3: M(X); break;
                              // Constraint erfüllt beim Szenario 4 => keine Operation
                              CASE 4: break;
                              // Constraint verletzt beim Szenario 5
                              CASE 5: M(X); break;
                            }
                    }
            }

// Überprüfung für Constraints von Typ (dependsOn, X, Y, post):
            if post_dep_S(X)
            {       // Jede S-A wird markiert
                    M(X);
                    // Keine Korrektur notwendig
            }
            if post_dep_T(X)
            {       // Wurde S-A schon gefunden?
                    if NOT(MIT_MARKIERUNG(S-T(X)))
                    {       // S-A wird noch nicht gefunden => Annahme: Szenario 9 in Abbildung 4.41
                            // => Constraint verletzt (keine Operation)
                    }
                    // S-A schon gefunden.
                    else
                    {       // Befindet sich S-A vor der komplexen Block-Struktur (Szenario 8)?
                            if BRANCHNR(S-T(X)==Null))
                            // Wird T-A immer ausgeführt?
                                    if NOT(AUSFÜHRUNG_XOR(POSITIONSM3(X))
                                    // T-A wird immer ausgeführt => Constraint erfüllt
                                            LÖSCHEN(X);
                                    // T-A wird nicht immer ausgeführt
                                    // => Constraint verletzt, keine Operation
                            // S-A befindet sich in der komplexen Block-Struktur
```

```
                    else
                    SWITCH (Konstellation(X,S-T(X)))
                    { // Wenn die Zweignummern von S-A und T-A gleich sind, ist das
                      // Constraint erfüllt beim Szenario 1.
                     CASE 1: if (BRANCHNR(X)==BRANCHNR(S-T(X)))LÖSCHEN(X);break;
                      // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                      // erfüllt beim Szenario 2.
                     CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))LÖSCHEN(X);break;
                      // Kontrollieren der Ausführung von T-A beim Szenario 3.
                      CASE 3: if (NOT(AUSFÜHRUNG_XOR(TOOT(X))))LÖSCHEN(X);break;
                      // Constraint verletzt beim Szenario 4 => keine Operation
                      CASE 4: break;
                      // Constraint verletzt beim Szenario 5 => keine Operation
                      CASE 5: break;
                    }
            }
    }

// Überprüfung für Constraints von Typ (excludes, X, Y, post):
    if post_exc_S(X)
    {       // Jede S-A wird markiert
            M(X);
            // Korrektur für das Szenario 1 und das Szenario 5.
            // Wenn dabei T-A vor S-A vorkommt, ist das Constraint verletzt.
            if (POSITIONSM3(S-T(X))!=Null)
                    if ((Konstellation(X,S-T(X))==1) ODER (Konstellation(X,S-T(X))==5))
                            M(S-T(X));
    }
    if post_exc_T(X)
    {       // Wurde S-A bereits gefunden?
            if NOT(MIT_MARKIERUNG(S-T(X)))
            {       // S-A wird noch nicht gefunden => Annahme: Szenario 9 in Abbildung 4.41
                    // Constraint dabei erfüllt => keine Operation
            }
            // S-A schon gefunden.
            else
            {       // Befindet sich S-A vor der komplexen Block-Struktur (Szenario 8)?
                    if BRANCHNR(S-T(X)==Null))
                            // => Constraint verletzt beim Szenario 8
                            M(X);
                    // S-A befindet sich in der komplexen Block-Struktur
                    else
                    SWITCH (Konstellation(X,S-T(X)))
                    { // Constraint verletzt beim Szenario 1
                      CASE 1: M(X); break;
                      // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                      // verletzt beim Szenario 2.
                      CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))M(X);break;
                      // Constraint verletzt beim Szenario 3
                      CASE 3: M(X); break;
                      // Constraint erfüllt beim Szenario 4 => keine Operation
                      CASE 4: break;
                      // Constraint verletzt beim Szenario 5
                      CASE 5: M(X); break;
                    }
            }
    }
```

```
// Überprüfung für Constraints von Typ (dependsOn, X, Y, pre):
        if pre_dep_S(X)
        {       // Jede S-A wird markiert
                M(X);
        // Korrektur dafür, dass S-A nach T-A in der komplexen Block-Struktur gefunden wurde.
                if (POSITIONSM3(S-T(X))!=Null)
                {
                        SWITCH (Konstellation(X,S-T(X)))
                        { // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                          // erfüllt beim Szenario 1.
                          CASE 1: if (BRANCHNR(X)==BRANCHNR(S-T(X)))
                                                        LÖSCHEN(S-T(X)); break;
                          // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                          // erfüllt beim Szenario 2.
                          CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X))) LÖSCHEN(S-T(X));
                                                        break;
                          // Kontrollieren der Ausführung von T-A beim Szenario 3.
                          CASE 3: if (NOT(AUSFÜHRUNG_XOR(TOOT(S-T(X))))
                                                        LÖSCHEN(S-T(X));break;
                          // Constraint verletzt beim Szenario 4 => keine Operation
                          CASE 4: break;
                          // Constraint verletzt beim Szenario 5 => keine Operation
                          CASE 5: break;
                        }
                }
        }
        if pre_dep_T(X)
        {       // Wurde S-A schon gefunden?
                if NOT(MIT_MARKIERUNG(S-T(X)))
                {       // S-A wird noch nicht gefunden => Annahme: Szenario 9 in Abbildung 4.41
                        // Wird T-A immer ausgeführt?
                        if NOT(AUSFÜHRUNG_XOR(POSITIONSM3(X))
                        // T-A wird immer ausgeführt => Constraint erfüllt
                                LÖSCHEN(X);
                        // T-A wird nicht immer ausgeführt => Constraint verletzt (keine Operation)
                }
                // S-A schon gefunden.
                else
                {
                        // Falls T-A nach S-A gefunden wird,
                        // ist das Constraint verletzt => keine Operation
                }
        }

// Überprüfung für Constraints von Typ (excludes, X, Y, pre):
        if pre_exc_S(X)
        {       // Jede S-A wird markiert
                M(X);
        // Korrektur dafür, dass S-A nach T-A in der komplexen Block-Struktur gefunden wurde.
                if (POSITIONSM3(S-T(X))!=Null)
                {
                        SWITCH (Konstellation(X,S-T(X)))
                        { // Constraint verletzt beim Szenario 1
                          CASE 1: M(S-T(X)); break;
                          // Wenn die Zweignummern von S-A und T-A gleich sind, ist das Constraint
                          // verletzt beim Szenario 2.
                          CASE 2: if (BRANCHNR(X)==BRANCHNR(S-T(X)))M(S-T(X));break;
                          // Constraint verletzt beim Szenario 3
```

```
                    CASE 3: M(S-T(X)); break;
                    // Constraint erfüllt beim Szenario 4 => keine Operation
                    CASE 4: break;
                    // Constraint verletzt beim Szenario 5
                    CASE 5: M(S-T(X)); break;
                  }
            }
      }
      if pre_exc_T(X)
      {     // Wurde S-A bereits gefunden?
            if NOT(MIT_MARKIERUNG(S-T(X)))
            {     // S-A wird noch nicht gefunden => Annahme: Szenario 9 in Abbildung 4.41
                  // Constraint verletzt beim Szenario 9
                  M(X)
            }
            // S-A schon gefunden.
            else
            {     // Befindet sich S-A vor der komplexen Block-Struktur (Szenario 8)?
                  if BRANCHNR(S-T(X)==Null))
                  {
                        // Constraint erfüllt => keine Operation
                  }
                  // S-A befindet sich in der komplexen Block-Struktur
                  else
                  SWITCH (Konstellation(X,S-T(X)))
                  { // Constraint verletzt, wenn T-A nach S-A auf ein anderen Zweig des UND-
                    // Block (OT-Blocks) beim Szenario 1 gefunden wird.
                    CASE 1: if (BRANCHNR(X)!=BRANCHNR(S-T(X)))
                                    M(X); break;
                      // Constraint erfüllt beim Szenario 2 => keine Operation
                    CASE 2: break;
                      // Constraint erfüllt beim Szenario 3 => keine Operation
                    CASE 3: break;
                    // Constraint verletzt beim Szenario 4
                    CASE 4: M(X); break;
                      // Constraint erfüllt beim Szenario 5 => keine Operation
                    CASE 5: break;
                  }
            }
      }

// Hole die nächste Aktivität in der komplexen Block-Struktur
      Next_IN_KBS(X);
// Die Positionsmarkierungen werden erst nach der Überprüfung von Sync-Kante aufgehoben.
}
```

In komplexer Block-Struktur mit UND-Block können auch Sync-Kanten auftreten. Wie bei einfachen UND-Blöcke lässt sich die Überprüfung der Sync-Kanten in komplexer Block-Struktur separat nach der Überprüfung der reinen komplexen Block-Struktur machen, weil die semantischen Wirkungen von Sync-Kante in einem einfachen UND-Block und in komplexer Block-Struktur fast gleich sind. Es gibt nur einen einzigen Unterschied zwischen den beiden: Die Aktivitäten in einem einfachen UND-Block werden immer ausgeführt. In komplexer Block-Struktur ist dies nicht gewährleistet. Daher müssen wir für Constraints von Abhängigkeitsbeziehung bei komplexer Block-Struktur überprüfen, ob T-A immer ausgeführt wird, wenn S-A ausgeführt wird. D.h. erst wenn *AUSFÜHRUNG_XOR(TOOT(T-A)) == false*, kann ein Constraint von Abhängigkeitsbeziehung durch Sync-Kante erfüllt werden.

In Abbildung 4.42 ist das Constraint 1 *(dependsOn, O, E, post)* trotz der Sync-Kante (O,E) verletzt auf dem Prozess, weil E sich in ein XOR-Block, der innerhalb vom *OT-Block* (UND-Block (A,R) im Beispiel)

ist, befindet. Sie wird nicht immer ausgeführt, wenn O ausgeführt wurde (*AUSFÜHRUNG_XOR(TOOT(E)) == true*). Dagegen ist das Constraint 2 *(dependsOn, E, M, pre)* wegen der Sync-Kante erfüllt, weil *AUSFÜHRUNG_XOR(TOOT(M)) == false*. Die Ausführung von T-A des Constraint von Ausschlussbeziehung ist dabei irrelevant. Es gilt *AUSFÜHRUNG_XOR(TOOT(E)) == true*. Trotzdem ist das Constraint 3 *(excludes, O, E, pre)* erfüllt. Weil die ungewünschte Ausführungsreihenfolge von O und E durch die Sync-Kante ausgeschlossen ist.

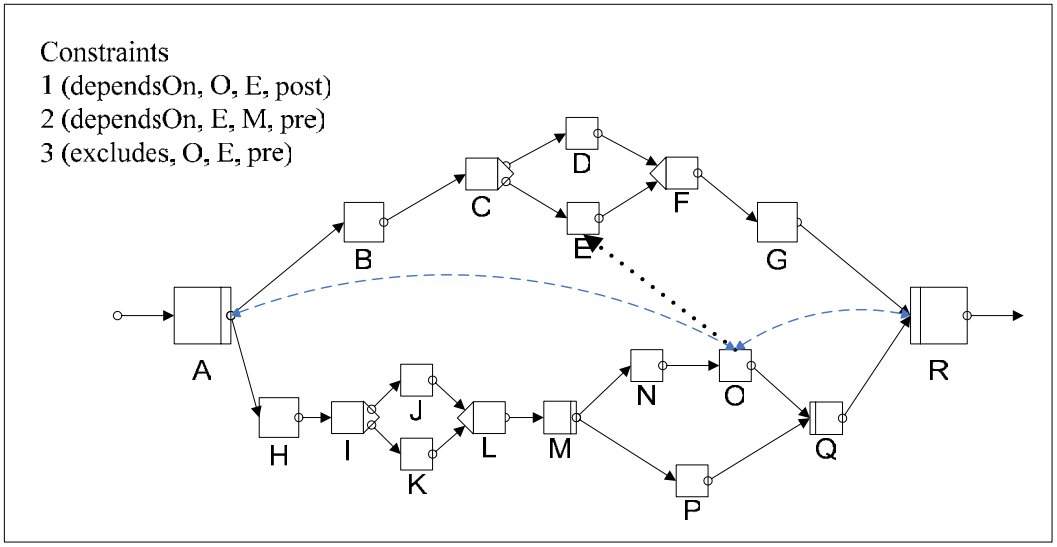


Abb. 4.42 Sync-Kante in komplexer Block-Struktur

Definierte Funktionen:

NEXT_S(X) : Funktion, die die nächste Quell-Aktivität einer Sync-Kante in der Menge S, die wir bei der Überprüfung des UND-Blocks erzeugt haben, zurückgibt. Wenn alle Aktivitäten in der Menge S abgearbeitet wurden, liefert die Funktion *Null* zurück.

NEXT_N(X) : Funktion, die die nächste zu überprüfende Aktivität in der Nachfolgermenge einer Sync-Kante holt.

VORG(X) : Funktion, die überprüft ob sich die Aktivität X in der Vorgängermenge der aktuell zu überprüfenden Sync-Kante befindet.

Die Bestimmung der Vorgängermenge und der Nachfolgermenge ist analog wie bei einfachen UND-Blöcken. Dabei müssen wir wissen, zu welchen UND-Block eine Sync-Kante gehört. Die Sync-Kante in Abbildung 4.42 gehört beispielsweise zum UND-Block (A,R).

Der Pseudocode des Algorithmus von Stufe II für die Überprüfung der Sync-Kante sieht wie folgt aus.

```
// Für die erste Sync-Kante
NEXT_S(X);
// Überprüfung aller Sync-Kanten
FOR (X!=Null)
{
// Hole die erste zu überprüfende Aktivität in der Nachfolgermenge.
        NEXT_N(X);
// Überprüfung aller Aktivitäten in der Nachfolgermenge der aktuell überprüft Sync-Kante
FOR (X!=Null)
{
// Korrektur für die Constraints von Typen (dependsOn, S-A, T-A, post)
        if post_dep_T(X)
                if (VORG(S-T(X)))
                        if (NOT(AUSFÜHRUNG_XOR(TOOT(X)))
                                LÖSCHEN(X);
// Korrektur für die Constraints von Typen (excludes, S-A, T-A, pre)
        if pre_exc_T(X)
```

```
                    if (VORG(S-T(X)))
                        LÖSCHEN(X);
// Korrektur für die Constraints von Typen (dependsOn, S-A, T-A, pre)
        if pre_dep_S(X)
                    if (VORG(S-T(X)))
                        if (NOT(AUSFÜHRUNG_XOR(TOOT(S-T(X))))
                            LÖSCHEN(S-T(X));
// Korrektur für die Constraints von Typen ((excludes, S-A, T-A, post)
        if post_exc_S(X)
                    if (VORG(S-T(X)))
                        LÖSCHEN(S-T(X));
// Holt das nächste Element in der Nachfolgermenge.
        NEXT_N(X);
}
// Die nächste Sync-Kante
NEXT_S(X)
}
// Nach der Überprüfung der Sync-Kanten löschen wir die Positionsmarkierungen
L_POSITIONSM;
```

Mit der Überprüfung von Sync-Kanten ist der Algorithmus von Stufe III vollständig. Wenn wir die einfachen Blöcke und die verschachtelten Blöcke bei der Überprüfung an der Stelle der Split-Aktivität nicht unterscheiden können, können wir nur die Algorithmus von Stufe III implementieren und für alle Block-Strukturen verwenden. Wie gesagt, sie ist eine Verallgemeinerung des Algorithmus von Stufe II und für alle Block-Strukturen einsetzbar.

4.3.4 Anwendung des Verifikationsansatzes

Bei der Überprüfung eines Prozesses verwenden wir die Algorithmen von den drei Stufen abwechselnd für entsprechende Struktur (vgl. Abbildung 4.10). Wenn wir Prozesse ohne Ausführungszustand überprüfen, sind die Algorithmen direkt zu verwenden, wie z.B. bei Verifikation eines ganzen Schemas oder bei Schemaänderungen. Für Prozesse mit Ausführungszustand (Prozessinstanzen) ignorieren wir die abgewählten Zweige in XOR-Blöcke und alle Aktivitäten darauf.

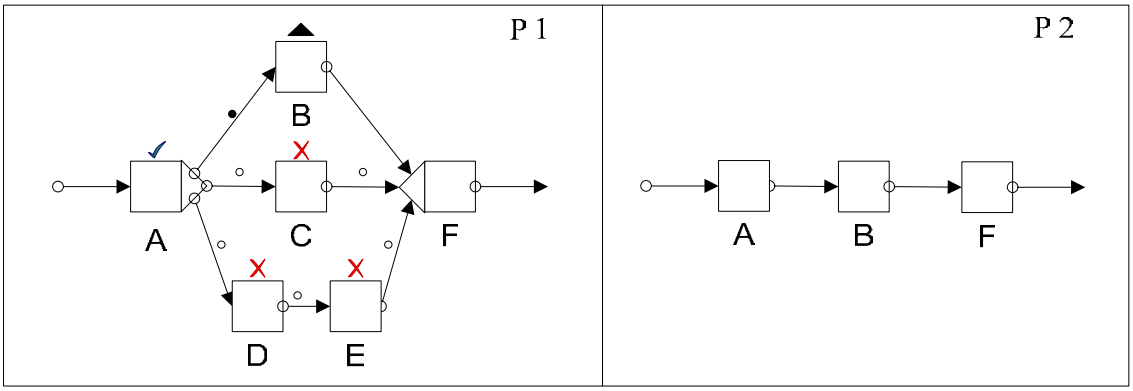


Abb. 4.43 Überprüfung von Prozess mit Ausführungszustand

Die mit „SKIPPED“ bewerteten Aktivitäten {C, D, E} auf dem P1 in Abbildung 4.43 werden bei der Überprüfung als nicht vorgekommen betrachtet, d.h. bei der Überprüfung betrachten wir P1 wie P2.

Bei Erzeugung der Menge L1_ALL (Konzept in Abschnitt 4.2.1) werden die mit „SKIPPED“ bewerteten Aktivitäten ebenfalls ignoriert, weil sie nicht mehr als S-A semantisch effektiv sind. Das Constraint *(excludes, D, F, post)* auf dem Prozess P1 in Abbildung 4.43 ist beispielsweise erfüllt und muss nicht überprüft werden. Deswegen bei erzeugen der Menge L1_ALL für eine Instanz, suchen wir S-A nur in der Menge der Aktivitäten, die nicht mit „SKIPPED“ bewertet sind. Damit enthält die Menge L1_MIN (L1_MIN = L1_ALL ∩ L1) nur die S-A, die tatsächlich im Prozesse vorkommen und sich in einem zu überprüfenden Constraint befinden und semantisch effektiv sind.

In Abschnitt 3.3.2.1 haben wir noch eine andere Situation genannt, wo die Instanz toleranter gegenüber Änderungsoperationen als das Schema sein kann (vgl. Abbildung 3.10). Es geht um die bereits ausgeführten Aktivitäten in einem UND-Block. Wenn zur Laufzeit einige Aktivitäten in einem UND-Block schon ausgeführt sind, ist die Ausführungsreihenfolge von allen Aktivitäten in dem UND-Block teilweise festgestellt.

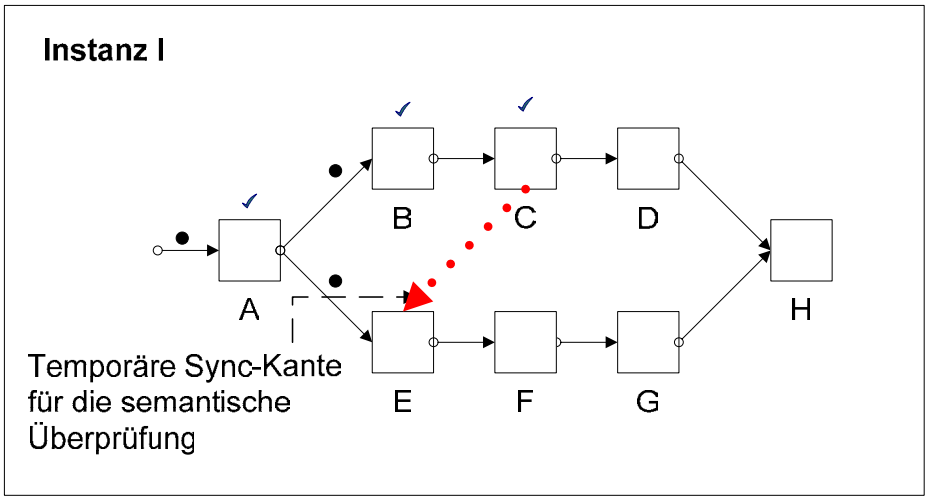


Abb. 4.44 Temporäre Sync-Kante für die semantische Überprüfung

In Abbildung 4.44 können die ausgeführten Aktivitäten B, C nicht mehr nach den Aktivitäten auf den anderen UND-Zweig ({E, F, G}) ausgeführt werden. Constraints wie z.B. *(dependsOn, F, C, pre)* sind dabei erfüllt. Wenn wir die Algorithmen direkt verwenden und die Markierung für „ausgeführte“ ignorieren, werden diese Constraints als verletzt bewertet. Weil die semantische Wirkung der Ausführungszustand hierbei gleich wie Sync-Kanten sind, können wir das Problem dadurch lösen, dass wir temporäre Sync-Kanten in solchen UND-Block hinzufügen, und zwar ausgehend von der letzten ausgeführten Aktivität auf einem Zweig zu der ersten noch nicht ausgeführten Aktivität auf einem anderen Zweig, z.B. die Sync-Kante (C, E) im Beispiel. Es kann notwendig sein, mehrere temporäre Sync-Kanten, die Deadlock erzeugen können, als Hilfsmitteln für die semantische Überprüfung hinzugefügt werden müssen. Wir verwenden die Überprüfungsalgorithmen für diese Sync-Kanten. Die eventuell falschen Überprüfungsergebnisse werden dadurch korrigiert. Nach der semantischen Überprüfung löschen wir wieder diese Sync-Kanten.

4.4 Optimierungen

In diesem Abschnitt werden wir versuchen, die Vorgehensweise zu verbessern. Der überflüssige Aufwand bei der Überprüfung soll vermieden werden und der Algorithmus soll effizienter werden.

4.4.1 Ignorieren semantisch irrelevanter Aktivitäten

Als eine Optimierung kann man die semantisch irrelevanten Aktivitäten, d.h. die Aktivitäten, die weder Source-Aktivität noch Target-Aktivität von irgendeinem Constraint sind, bei der semantischen Überprüfung ignorieren.

4.4.1.1 Markierung der semantischen irrelevanten Aktivitäten

Dazu kann man alle semantisch irrelevanten Aktivitäten für einen Prozess markieren. Die Abbildung 4.45 zeigt die Entdeckung eines semantischen Konflikts, wobei die semantisch irrelevanten Aktivitäten bereits markiert sind.

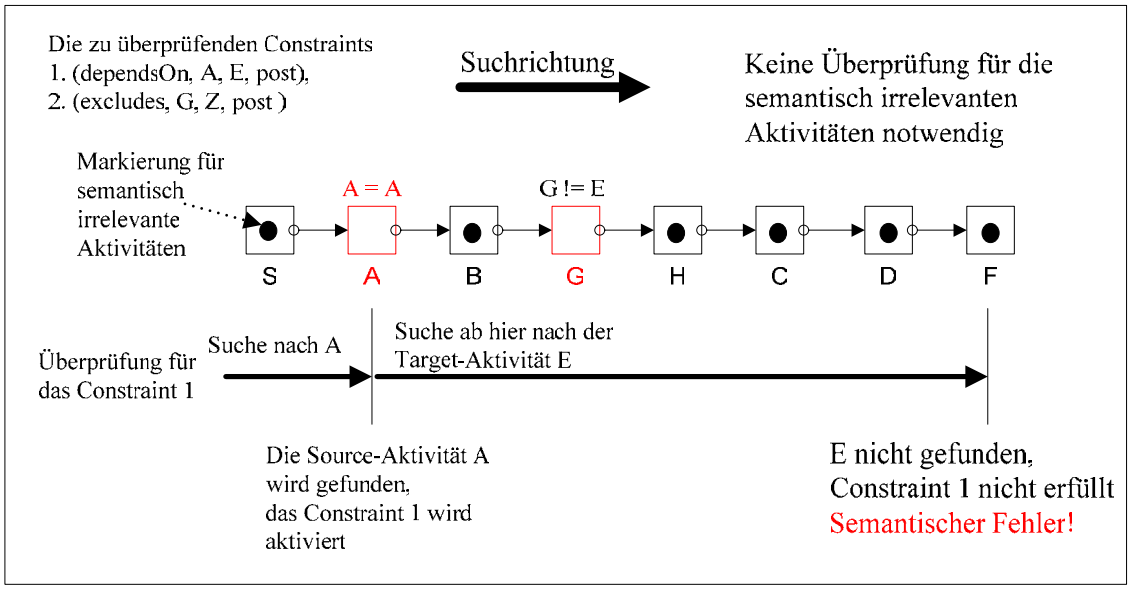


Abb. 4.45 Entdeckung eines semantischen Konflikt

Es wird verdeutlicht, dass durch Markierung der irrelevanten Aktivitäten viele Operationen gespart werden können. Statt acht bleiben nur zwei Aktivitäten zu kontrollieren. Die Verbesserung ist sehr erheblich, dagegen ist der Aufwand um die Aktivität zu markieren relativ klein und zwar einmalig für ein Schema und alle darauf basierenden Instanzen. Wir können beispielsweise ein Array aller möglichen Aktivitäten eines Schemas erzeugen. Für semantisch relevante und irrelevante Aktivitäten werden unterschiedliche Werte zugewiesen, z.B. mit einer booleschen Variable mit Werte *True* für die semantisch relevanten Aktivitäten und *False* für die irrelevanten.

4.4.1.2 Links zwischen semantisch relevanten Aktivitäten

Wenn die semantisch irrelevanten Aktivitäten markiert wurden, brauchen sie bei der Überprüfung nicht mehr überprüft werden. Trotzdem müssen wir jede Aktivität lesen, um zu wissen, welche markiert sind und welche nicht. Wenn nun zusätzlich noch eine Verbindung, wie z.B. ein semantischer Link, zwischen zwei nachfolgenden semantisch relevanten Aktivitäten gebildet wird, können die semantisch irrelevanten Aktivitäten bei der Überprüfung übersprungen werden (vgl. Abbildung 4.46).

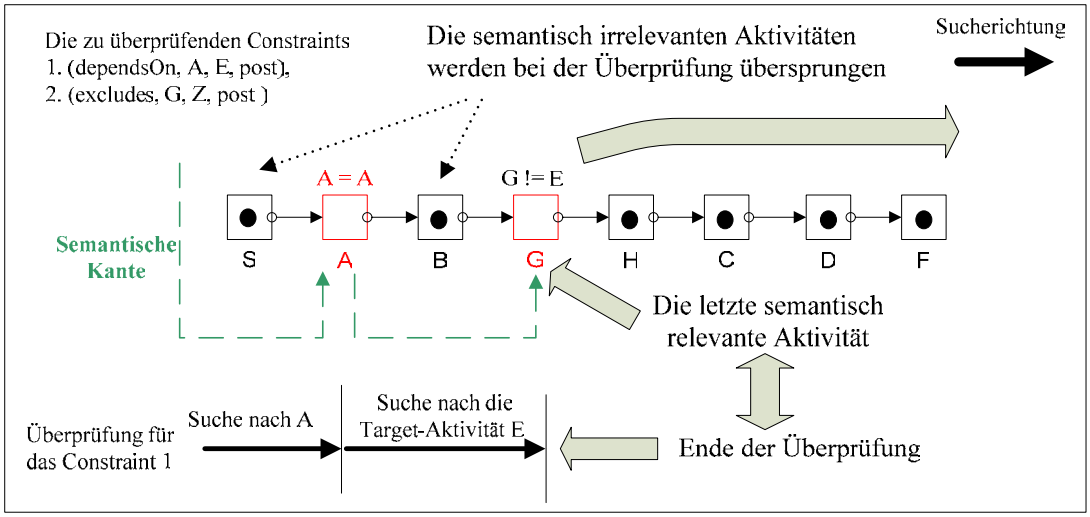


Abb. 4.46 Einsatz von semantischen Kanten

Dadurch werden die semantisch irrelevanten Aktivitäten bei der Überprüfung tatsächlich ignoriert. Für die Überprüfung sind die Links günstiger als die Markierungen, aber entsprechend muss man dann den Pflegeaufwand übernehmen. Anders als die Markierung, die unabhängig von den Änderungsoperationen

ist, müssen die semantische Links nach den Änderungsoperationen, die die semantisch relevanten Aktivitäten modifizieren, aktualisiert oder ggf. neu hergestellt werden.

4.4.2 Vorschlag für die Implementierung

Bei unserem Algorithmussystem suchen wir jede zu überprüfende Aktivität in der drei Menge L1, L2 und L3. Weil die drei Mengen vollständig bezüglich der zu überprüfenden Constraints sind, können die Funktionen ***notS_dep_S(X), notS_dep_T(X)*** usw. aufwendig sein. Dies können wir dadurch lösen, dass wir statt der drei Mengen ein Array implementieren. Das Array ist nur eine andere Repräsentation der zu überprüfenden Constraints (vgl. Abbildung 4.47).

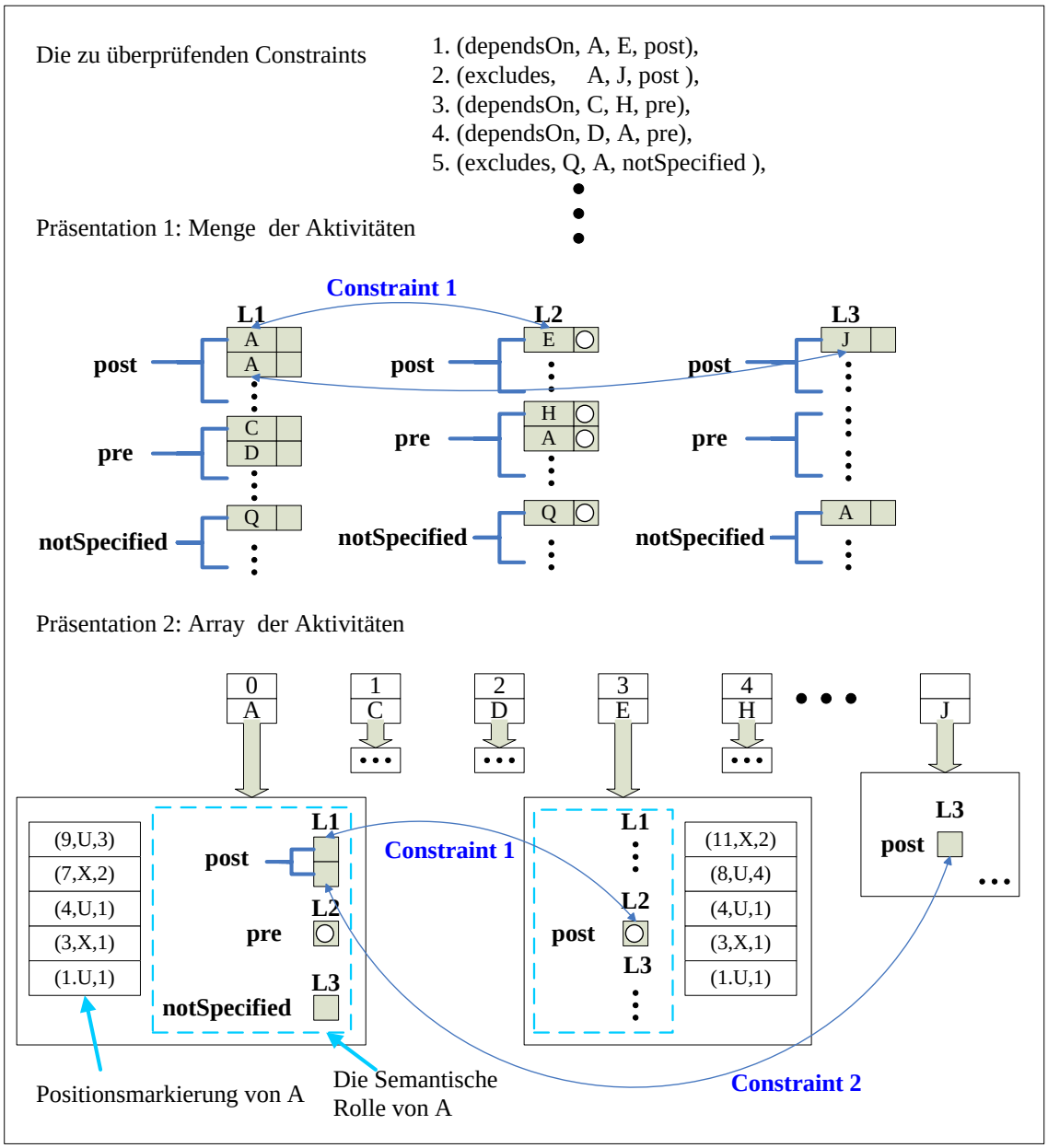


Abb. 4.47 Implementierung mit einem Array

Das Element des Array ist ein Objekt mit zwei Teilen. Einen Teil verwenden wir um die Positionsmarkierung einer Aktivität zu speichern, der andere Teil ist für die semantische Rolle der Aktivität. Im Beispiel ist A die S-A von Constraints 1 und 2 und die T-A von Constraints 4 und 5. Entsprechend taucht sie zweimal in L1 und jeweils einmal in L2 und L3 auf. Deswegen gibt es in dem Objekt von A vier Speicherplätze für die Verletzungsmarkierungen. Dabei muss deutlich gemacht werden,

für welche semantische Rolle sie stehen. D.h. die Informationen post/pre/notSpecified und L1/L2/L3 müssen da sein. Ebenfalls brauchen wir die Verbindungen zwischen S-A und T-A. In der Abbildung wird nur die Verbindungen für Constraints 1 und 2 gezeichnet. Jedoch müssen wir die Verbindungen zwischen S-A und T-A für alle zu überprüfende Constraints implementieren (für die Funktion *S-T(X)*). Als Alternative können wir auch direkt die entsprechenden Constraints in dem Objekt speichern.

Wenn wir ein solches vollständiges Array bevor der Überprüfung erzeugen, können wir zur Laufzeit mit einem Zugriff auf dem Array die semantische Rolle einer Aktivität feststellen. Dann brauchen wir nicht mehr eine Aktivität in der drei Menge L1, L2 und L3 zu suchen. Je nachdem, welche Rolle eine Aktivität spielt, rufen wir einfach direkt die entsprechenden Methoden auf. Die Aktivität A in Abbildung 4.47 wird beispielsweise zweimal als S-A und zweimal als T-A überprüft. Wenn eine Aktivität nicht in dem Array ist, dann ist sie irrelevant für die Überprüfung. D.h. der Effekt der Markierungen für semantische irrelevante Aktivitäten (Abschnitt 4.3.1.1) wird durch die Implementierung des Array auch realisiert.

4.4.3 Weitere Optimierungsmöglichkeiten

Für die Algorithmen gibt es noch ein paar nicht grundsätzliche Optimierungsmöglichkeiten. Beispielsweise lässt sich die Überprüfung einer Sync-Kante in die Überprüfung des UND-Blocks integrieren, damit die Bestimmung von Vorgängermengen einigermaßen weniger aufwendig werden kann. Wegen drei Gründe haben wir dies in dem Konzept nicht gemacht.

1. Das Konzept soll systematisch und verständlich sein.
2. Das Konzept soll vollständig und weiter entwickelbar sein.
3. Die Verbesserung des Konzepts entspricht nicht unbedingt einer Verbesserung bei der Implementierung.

Jedoch bei Implementierung gibt es solche Einschränkung nicht. Da kann man die Algorithmen kompakter machen.

4.5 Zusammenfassung

In Kapitel 4 haben wir Algorithmen zur semantischen Überprüfung entwickelt. Am Anfang haben wir festgestellt, dass die semantische Überprüfung im Wesentlichen der Suche nach bestimmten Verhältnissen oder Reihenfolgen von Aktivitäten in einem Prozess Entspricht. Entsprechend haben wir eine naive Vorgehensweise entwickelt. Mit Einsetzung des mengenbasierten Verfahrens wird es ermöglicht, dass alle Constraints mit höchstens zwei Durchläufen eines Prozesses zu überprüfen. Das ist aber immer noch nicht effizient genug. Deswegen haben wir ein Markierungssystem als Hilfsmittel entwickelt, um die Überprüfung aller Constraints in einen Durchlauf integrieren zu können. Damit können wir gleichzeitig S-A und T-A bei der Überprüfung beobachten. Entsprechend werden die Algorithmen zur Überprüfung unterschiedlicher Strukturen eingeteilt. Die Algorithmen sind nach der Optimierungen und Anpassungen effizient und können bei allen vier Szenarien, die in Kapitel 3 vorgestellt wurden, eingesetzt werden. Es wurden auch Implementierungsmöglichkeiten der Algorithmen dargestellt, die das Programm gegenüber dem Pseudocode in der Arbeit einigermaßen vereinfachen kann.

Kapitel 5

Semantische Wirkungen von Änderungsoperationen

Wie in Abschnitt 3.5 geschrieben, wir können die Effizienz der semantischen Überprüfung durch Optimierung des Überprüfungsverfahrens verbessern. Außerdem können wir noch die zu überprüfende Constraints reduzieren. Eine Änderungsoperation kann nicht alle Constraints verletzen, sondern nur eine Teilmenge davon gefährden. In diesem Kapitel werden wir die semantische Wirkung der Änderungsoperationen bestimmen. Wir werden die Constraints, die potentiell durch eine Änderungsoperation verletzt werden, identifizieren. Mit dieser Einschränkung brauchen wir zur Laufzeit, wie z.B. nach ein paar Ad-hoc-Änderungen an einer Instanz, nicht mehr alle Constraints sondern nur die relevanten Constraints für die Änderungsoperationen zu überprüfen. Für Schemaevolution werden wir noch die Wechselwirkung von den Instanzänderungen ΔI und den Schemaänderungen ΔS analysieren, um die zu überprüfenden Constraints weiter einzuschränken. Diese Analyse wurde bereits in [13] getan, hier wird sie nur weiter entwickelt.

5.1 Operation insertNode

Durch die Operation *insertNode(A, B, C)* wird eine neue Aktivität A in einem Prozess eingefügt. Alle Constraints von dem Prozess mit A als S-A werden dadurch auf dem Prozess aktiviert. Und die Constraints für Ausschlussbeziehungen mit A als T-A werden auch von der Operation gefährdet. Deswegen sind folgende Constraints, falls sie in der Constraintmenge für den Prozess enthalten sind, nach der Operation zu überprüfen [13]:

Sei X eine beliebige Aktivität, dann sind folgende Constraints potentiell verletzt:

- *(dependsOn, A, **X**, post)* (***X*** steht für eine beliebige Aktivität);
- *(dependsOn, A, **X**, pre)*;
- *(dependsOn, A, **X**, notspecified)*;
- *(excludes, A, **X**, post);*
  Wenn es ein Constraint gibt mit ***X*** = C, soll die Operation direkt abgelehnt werden.
- *(excludes, A, **X**, pre);*
  Wenn es ein Constraint gibt mit ***X*** = B, soll die Operation direkt abgelehnt werden.
- *(excludes, A, **X**, notspecified);*
  Wenn es ein Constraint gibt mit ***X*** = B oder C, soll die Operation direkt abgelehnt werden.
- *(excludes, **X**, A, post);*
  Wenn es ein Constraint gibt mit ***X*** = B, soll die Operation direkt abgelehnt werden.
- *(excludes, **X**, A, pre);*
  Wenn es ein Constraint gibt mit ***X*** = C, soll die Operation direkt abgelehnt werden.
- *(excludes, **X**, A, notspecified);*
  Wenn es ein Constraint gibt mit ***X*** = B oder C, soll die Operation direkt abgelehnt werden.

Die Constraints in der Menge $C_{insertNode(A, B, C)}$ werden durch die Operation *insertNode(A, B, C)* potentiell verletzt, mit

$C_{insertNode(A, B, C)}$ = {(*Type*, *Source*, *Target*, *Position*) | (*Source*=*A*) ODER (*Type*=*excludes* UND *Target*=*A*)}

Die Constraints für Abhängigkeitsbeziehung mit A als Target-Aktivität bleiben bei Ausführung der Einfügeoperation unberührt. Alle Constraints, die A nicht als Parameter haben, sind selbstverständig für die Operation irrelevant.

5.2 Operation deleteNode

Für die Operation *deleteNode(A)* sind die Constraints mit A als Source-Aktivität irrelevant. Weil ein Constraint als automatisch erfüllt gilt, wenn die S-A nicht vorkommt ist. Durch die Löschoperation wird nur die Constraints von Abhängigkeitsbeziehung, die A als T-A haben, gefährdet [13]. Wenn die zu löschende Aktivität die Quelle oder Ziel-Aktivität einer Sync-Kante ist, wird die Sync-Kante durch die Operation mitgelöscht (siehe Abbildung 2.10). In dem Fall müssen zusätzlich die Constraints, die durch Löschen der Sync-kanten gefährdet werden, auch überprüft werden.

Die Constraints in der Menge $C_{deleteNode(A)}$ werden durch die Operation *deleteNode(A)* potentiell verletzt, mit

$C_{deleteNode(A)} = \{(Type, Source, Target, Position) \mid (Type= dependsOn \text{ UND } Target=A)\} (\cup \; C_{deleteSyncEdge})$

5.3 Operation moveNodes

Die Ausführung der Operation *moveNodes* ist gleich wie die aufeinander folgende Ausführung von den entsprechenden Operationen *deleteNode* und *insertNode.* Daher sind die durch die Operation *moveNodes* gefährdet Constraints einfach die Union der gefährdet Constraints von der entsprechenden *deleteNode* und *insertNode* Operationen [13]. Es gilt nämlich für die Operation *moveNodes(A, A, B, C),* die nur eine Aktivität A verschieben:

$C_{moveNodes(A, A, B, C)} = C_{deleteNode(A)} \cup \; C_{insertNode(A, B, C)}$

5.4 Operation createSyncEdge

Wir haben bereits festgestellt, dass kein Constraint durch eine Sync-Kante verletzt werden kann. Deswegen ist die Operation *createSyncEdge* semantisch immer korrekt. Jedoch können Constraints geben, die durch die Operation *createSyncEdge* erfüllt werden (siehe Abschnitt 4.3.2.2). Dies kann bei der Schemamodellierung relevant sein. Dabei kann man durch die Operation die temporär nicht erfüllten Constraints erfüllen (siehe Abschnitt 3.2.2).

5.5 Operation deleteSyncEdge

Um die durch die Operation *deleteSyncEdge* gefährdete Constraints einzuschränken, können wir die Aktivitäten in der Block-Struktur analysieren. Jedoch ist dieses Vorgehen zu aufwendig und nicht notwendig. Hierbei können wir einfach mit dem Überprüfungsverfahren, das in Abschnitt 4.3 vorgestellt wurde, die zu löschende Sync-Kante überprüfen. Die Überprüfung betrifft nicht den ganzen Prozess sondern nur der relevante Pfad der zu löschenden Sync-Kante (siehe Abbildung 4.30). Anhand der Überprüfung können wir feststellen, welche Constraints wegen der Sync-Kante erfüllt sind. Sie sind eben die Constraints, die durch die Operation *deleteSyncEdge* verletzt werden. Wir haben bereits festgestellt, dass Sync-Kanten keinerlei Einfluss auf die Constraints mit *Position = notSpecified* haben, weil für diese Constraints die Ausführungsreihenfolge von S-A und T-A irrelevant ist. Und Sync-Kanten machen nichts anderes als die Ausführungsreihenfolge von Aktivitäten festzulegen. Die übrigen Constraints werden durch die Operation *deleteSyncEdge* verletzt, wenn die in Abbildung 4.28 gezeigten Szenarien vorkommen. Durch die Operation *deleteSyncEdge* werden die Constraints $C_{deleteSyncEdge(X,Y)}$ verletzt, mit:
Vor(X,Y): Vorgängermenge der Sync-Kante (X,Y)
Nach(X,Y): Nachfolgermenge der Sync-Kante (X,Y) (vgl. Abschnitt 4.3)

$C_{deleteSyncEdge(X,Y)}$= {(*Type, Source, Target, Position*) |
(*Type= dependsOn* UND *Position=post* UND *Source*$\in$ Vor(X,Y) UND *Target*$\in$ Nach(X,Y)) ODER
(*Type= excludes* UND *Position=pre* UND *Source*$\in$ Vor(X,Y) UND *Target*$\in$ Nach(X,Y)) ODER
(*Type= dependsOn* UND *Position=pre* UND *Source*$\in$ Nach(X,Y) UND *Target*$\in$ Vor(X,Y)) ODER
(*Type= excludes* UND *Position=post* UND *Source*$\in$ Nach(X,Y) UND *Target*$\in$ Vor(X,Y)) }

5.6 Operation insertEmptyBranch

Durch die Operation *insertEmptyBranch* wird ein neuer leerer Teilzweig in einen einfachen Block eingefügt. Wenn man mit der Operation einen leeren Teilzweig in einen UND-Block hinzufügt, bleibt die Menge der möglichen Ausführungsreihenfolgen unverändert, d.h. die Operation ist in dem Fall semantisch wirkungslos. Wenn es um einen XOR-Block geht, wird aus der Operation eine neue Ausführungsreihenfolge entstehen, in der keine Aktivität in dem XOR-Block ausgeführt wird.

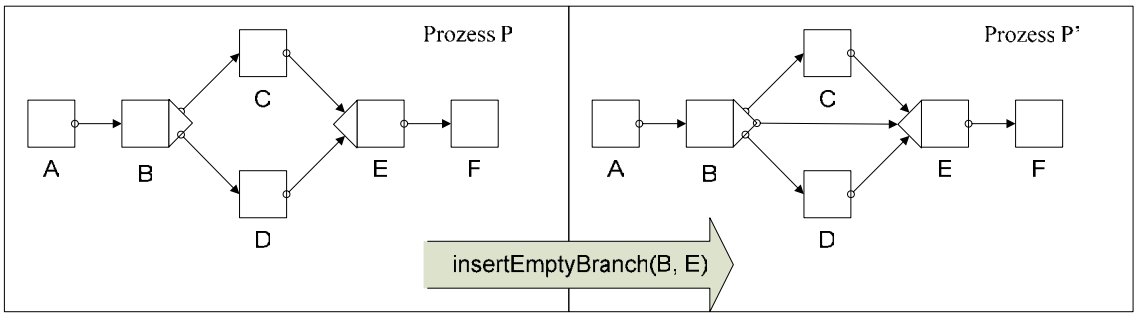


Abb. 5.1 Operation insertEmptyBranch

Für den Prozess P in Abbildung 5.1 gibt es zwei mögliche Ausführungsreihenfolgen {A, B, C, E, F} und {A, B, D, E, F}. Nach der Operation *insertEmptyBranch(B,E)* entsteht eine neue Möglichkeit: {A, B, E, F}. Allerdings wird durch diese Änderung kein Constraint verletzt. Weil die Aktivitäten in einem XOR-Block nicht immer ausgeführt werden, wird kein Constraint wegen dieser Aktivitäten erfüllt. Deswegen ist die Operation *insertEmptyBranch* immer semantisch richtig.

5.7 Operation deleteEmptyBranch

Die Operation *deleteEmptyBranch* löscht einen leeren Teilzweig in einem einfachen Block, wobei mindestens ein Teilzweig in dem Block übrig bleiben muss. Diese Operation kann man als die Gegenoperation von *insertEmptyBranch* betrachten. Die Operation *insertEmptyBranch* ist in einem einfachen UND-Block semantisch ineffektiv. Ebenfalls ist die Operation *deleteEmptyBranch*, weil die Menge der möglichen Ausführungsreihenfolgen des Prozesses nicht durch die Operation verändert wird. In einen XOR-Block ist die Operation *deleteEmptyBranch* ebenso immer semantisch richtig, weil es dadurch nur ein Ausführungspfad wenig wird. In einem spezialen Fall kann die Operation Constraints erfüllen.

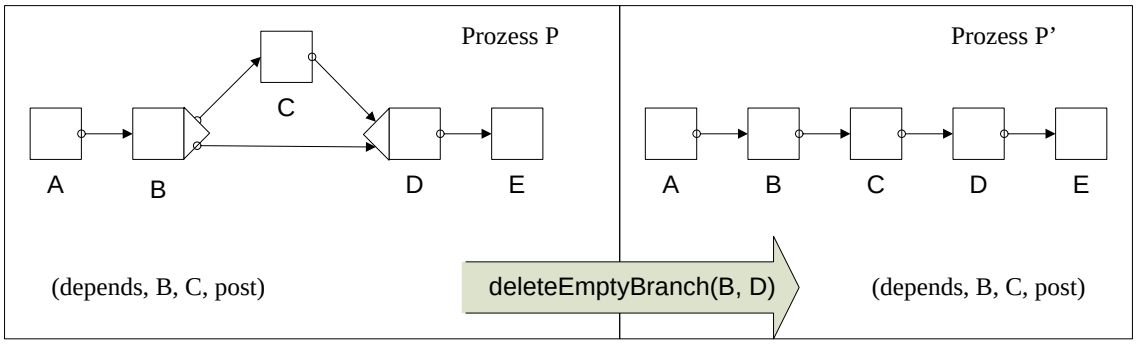


Abb. 5.2 Semantisch effektive Operation *deleteEmptyBranch*

Abbildung 5.2 illustriert ein Beispiel, wobei das Constraint *(dependsOn, B, C, post)* durch die Operation *deleteEmptyBranch* erfüllt wird. Das passiert nur, wenn nach der Operation *deleteEmptyBranch* nur ein Teilzweig in dem XOR-Block übrig bleibt. Da wird der XOR-Block automatisch zu Sequenz umstrukturiert. Die Aktivitäten auf dem letzten XOR-Zweig (C im Beispiel) werden nun immer ausgeführt und als T-A von Abhängigkeitsbeziehung gültig.

5.8 Wechselwirkung von Änderungsoperationen bei der Schemaevolution

In Abschnitt 3.4.3.2 haben wir festgestellt, dass es unnötig ist, bei Schemaevolution alle Constraints, die durch die instanzspezifischen Änderungen ΔI oder die Schemaänderungen ΔS gefährdet werden, zu überprüfen. Die Constraints, die tatsächlich durch die Migration verletzt werden können, sind nur eine Teilmenge davon ($C(W) \subseteq C(\Delta I) \cap C(\Delta S)$). C(W) bezeichnet die Constraints, die durch die Wechselwirkung von ΔI und ΔS gefährdet werden. Im Folgenden werden wir die Constraints, die durch Wechselwirkung verletzt werden können, in Tabelleform ausgeben [13].

Wenn mit der Operation *deleteNode* zusätzlich Sync-Kante mitgelöscht wird, können auch Constraints geben, die druch die Kombination von der Operation *deleteSyncEdge* und eine andere Operation verletzt werden. Dazu difinieren wir eine Menge von die Constraints $C_{Welchselwirkung}$ mit:

Sei X eine Aktivität
$C_{Welchselwirkung}(X)$ = {(*Type*, *Source*, *Target*, *Position*) |
(*Source*=X ODER *Target*=X) UND (*Position* != *notSpecified*)}

Für zwei *deleteNode* Operationen gibt es einen speziellen Fall, wobei Constraints durch die Welchselwirkung verletzt werden können. Das passiert, wenn durch die zwei *deleteNode* Operationen, die sich getrennt in ΔI und ΔS befinden, zwei parallele Sync-Kanten im denselben UND-Block löschen (vgl. Abbildung 5.3).

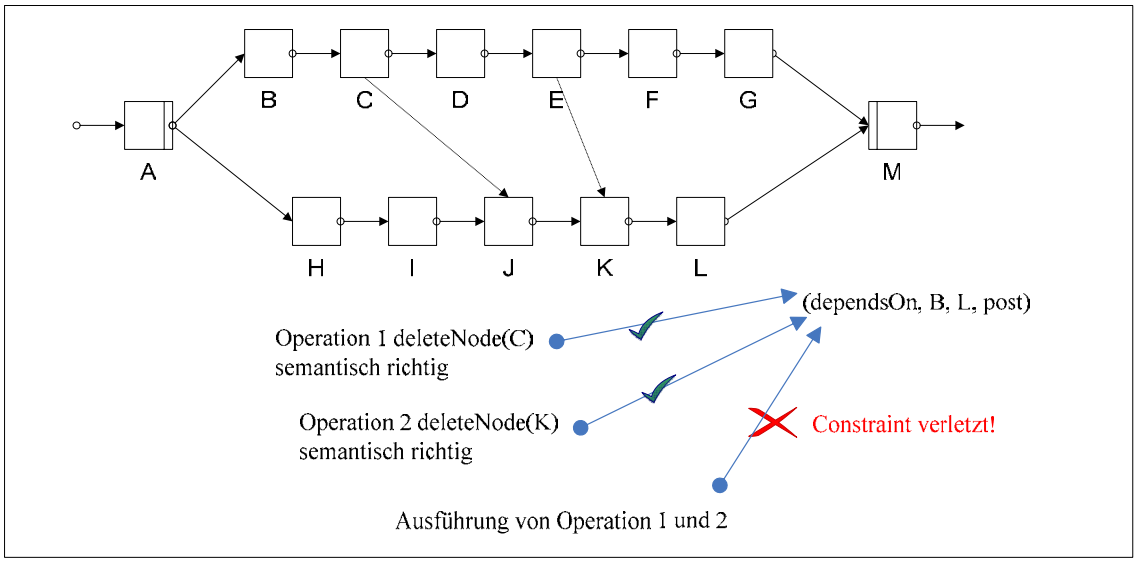


Abb 5.3 Welchselwirkung von zwei *deleteNode* Operationen

Die dadurch verletzten Constraints $C_{del\text{-}del}$ brauchen mindestens eine Sync-Kante von den beiden gelöschten, um die Ausführungsreihenfolge von S-A und T-A festzulegen, d.h.

$$C_{del\text{-}del} = C_{deleteSyncEdge(Sync\text{-}Kante1)} \cap C_{deleteSyncEdge(Sync\text{-}Kante2)}$$

Das gilt auch für die *moveNodes* Operationen, die Sync-Kanten durch Verschieben der Aktivitäten ebenfalls löschen oder modifizieren können.

	insertNode(A,B,C)	deleteNode(A)	moveNodes(A,A,B,C)
insertNode(D,E,F)	(excludes, A, D, pre/post/notSpecified) (excludes, D, A, pre/post/notSpecified)	(dependsOn, D, A, pre/post/notSpecified) $C_{Welchselwirkung}(D)$*	(excludes, A, D, pre/post/notSpecified) (excludes, D, A, pre/post/notSpecified) (dependsOn, D, A, pre/post/notSpecified) $C_{Welchselwirkung}(D)$*
deleteNode(B)	(dependsOn, A, B, pre/post/notSpecified) $C_{Welchselwirkung}(A)$*	$C_{del\text{-}del}$ **	(dependsOn, A, B, pre/post/notSpecified) $C_{Welchselwirkung}(A)$* $C_{del\text{-}del}$ **
moveNodes(D,D,E,F)	(excludes, A, D, pre/post/notSpecified) (excludes, D, A, pre/post/notSpecified) (dependsOn, A, D, pre/post/notSpecified) $C_{Welchselwirkung}(A)$*	(dependsOn, D, A, pre/post/notSpecified) $C_{Welchselwirkung}(D)$* $C_{del\text{-}del}$ **	(excludes, A, D, pre/post/notSpecified) (excludes, D, A, pre/post/notSpecified) (dependsOn, A, D, pre/post/notSpecified) (dependsOn, D, A, pre/post/notSpecified) $C_{Welchselwirkung}(D)$* $C_{Welchselwirkung}(A)$* $C_{del\text{-}del}$ **

Abb. 5.4 Die durch Wechselwirkung der Änderungsoperationen gefährdeten Constraints [13]
* nur wenn es Sync-Kanten gibt, die durch die Operation geändert oder gelöscht werden.
** nur beim spezialen Fall, dass zwei parallele Sync-Kanten getrennt von ΔS und ΔI modifiziert werden

Wenn beispielsweise ΔI gleich *deleteNode(A)* ist und ΔS gleich *insertNode(D,E,F)* ist, und A weder Quelle-Aktivität noch Ziel-Aktivität einer Sync-Kante ist, werden nur die Constraints *(dependsOn, D, A, pre), (dependsOn, D, A, post) und (dependsOn, D, A, notSpecified)* gefährdet.

Kapitel 6

Zusammenfassung und Ausblick

Im dieser Arbeit haben wir uns mit der semantischen Überprüfung für die einfachen Constraints befasst. Es wurde alle Überprüfungsszenarien: Verifikationen eines ganzen Schemas, Modellierung/Änderung eines Prozess-Schemas, Ad-hoc-Änderungen an Instanzen, Schemaevolution und Instanzmigration, ermittelt. Wir haben sie analysiert und die unterschiedlichen Besonderheiten für jedes Szenario festgestellt. Anhand dessen haben wir die Algorithmen für die semantische Überprüfung schrittweise entwickelt. Sie sind systematisch und effizient. Jedoch ist das Gebiet der semantischen Überprüfung damit nicht vollständig behandelt. Es gibt noch weitere offene Probleme zu lösen, wie z.B. semantische Überprüfung der Prozesse mit redundanten Aktivitäten und Verifikation von komplexeren Constraints. Entsprechend sind die Algorithmen, die in der Arbeit vorgestellt wurden, vielseitig weiter entwickelbar. Für die Prozesse, in der eine Aktivität mehrmals auftauchen darf, können wir das in Abschnitt 4.4.2 vorgestellte Array variieren. Statt eines Objekts für jede Aktivität als Element des Array können wir eine Liste von Objekten für die Aktivität an unterschiedlichen Stellen als Element des Arrays implementieren. Je nach der Definition kann man anhand des Erfüllungszustandes der Aktivitäten eine Aussage über die Erfüllung des Constraints machen. Für komplexere Constraints könnten wir eventuell mehr als drei Mengen brauchen. Da müssen die Algorithmen auch angepasst werden.

Anhang

Anhang A Beweis für die Realisierbarkeit des Zwischenschemas

Das Zwischenschema ZS wird durch Ausführung der Änderungsoperationen ΔI auf dem Originalschema S erzeugt. Im Allgemeinen sind die Instanzänderungen nicht unbedingt auch auf Schema semantisch richtig. ΔI bildet jedoch eine Teilmenge von ΔS und wird früher oder später auf dem Schema S ausgeführt. Die Frage ist bloß, ob ΔI direkt auf dem ungeändert Schema S anwendbar ist? Gibt es Änderungsoperationen von ΔS, die vor ΔI auf S ausgeführt werden müssen? Das passiert aber nur, wenn ein oder mehrerer XOR-Zweig(e) in der Instanz mit SKIPPED bewertet wurde(n). D.h. die Instanz ist jetzt toleranter gegenüber Änderungsoperationen als das Schema. Nehmen wir hierzu das Beispiel aus dem Abschnitt 3.3.2.1.

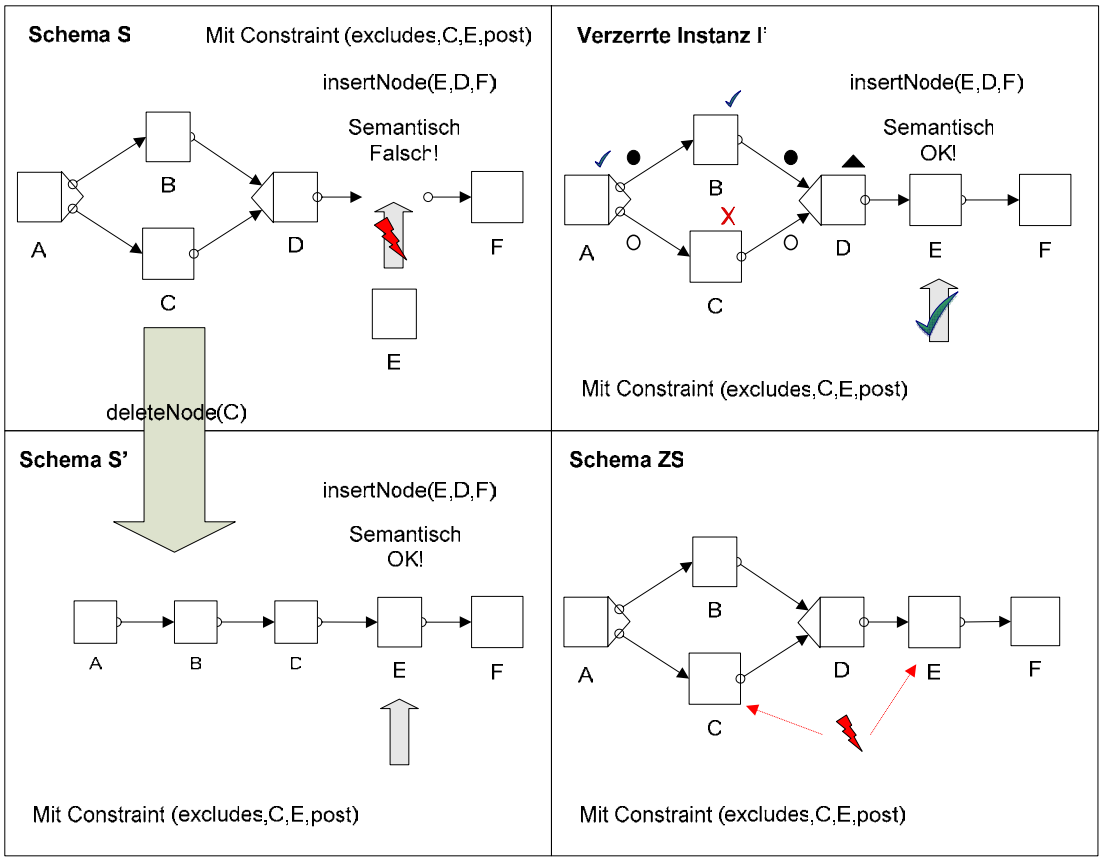


Abb. B1 die notwendige Operation

Hier ist ΔI die Operation *insertNode(E, D, F)*. Sie kann erst nach der Operation *deleteNode(C)* an S angewendet werden. Und es wird nie ein Zwischenschema ZS geben, was semantisch inkorrekt ist.

Wieso ist denn das Zwischenschema ZS für unser Szenario erzeugbar? Hier ist die Voraussetzung essentiell. Es geht um syntaktisch migrierbare Instanzen (vgl. Abbildung 3.22). Mit der Voraussetzung werden die Operationen, die die mit SKIPPED bewerteten XOR-Zweige modifizieren, ausgeschlossen. Beispielsweise kann die Operation *deleteNode(C)* in Abbildung B1 nicht in ΔS enthalten sein. Ansonsten sind S' und I' syntaktisch unverträglich [10]. Der ausgeführte Teil von I' wird durch ΔS geändert. In der Abbildung ist I' beispielsweise syntaktisch auf S' nicht migrierbar. Das widerspricht der Voraussetzung.

Wenn die ausgeführten XOR-Blöcke unberührt bleiben, wird es keine erforderlichen Operationen von ΔS geben, die vor ΔI auf dem Schema ausgeführt werden müssen. D.h. man kann die ΔI direkt am S

anwenden. So ist das Zwischenschema ZS für syntaktisch migrierbare Instanzen mit Sicherheit erzeugbar. Formal:

Beweis:

Voraussetzung: I syntaktisch migrierbar

⇒ ∀ s ∈ ΔS | s modifiziert die ausgeführten Teile von I' nicht.

ΔI direkt auf I anwendbar

⇒ ∄ s ∈ ΔS\ΔI mit s eine erforderliche Operation, die vor ΔI am dem Schema angewendet werden muss.

⇒ ΔI lässt sich direkt an S anwenden.

⇒ Das Zwischenschema ZS ist bestimmt erzeugbar.

Anhang B Beweis für die Realisierbarkeit der Zwischeninstanz

In Anhang A haben wir bewiesen, dass für den Fall $\Delta I < \Delta S$ das Zwischenschema ZS durch ΔI auf dem Originalschema S erzeugt werden können. Für die klasse *partially equivalent* sind die Instanzänderungen ΔI aber nicht ganz in den Schemaänderungen ΔS enthalten. Es gibt gemeinsame Änderungsoperationen ΔG von ΔI und ΔS, d.h.

$$\Delta G \subset \Delta I \text{ und } \Delta G \subset \Delta S$$

Jetzt wollen wir beweisen, dass die Zwischeninstanz ZI durch Anwendung von ΔG auf der Instanz erzeugbar ist. Falls ZI erzeugbar ist, kann man sie als eine verzerrte Instanz I' betrachten. Dabei ist die Instanzänderungen $\Delta I'$ gleich ΔG, was in ΔS enthalten sind, nämlich $\Delta I' < \Delta S$. D.h., wie im Anhang A bereits bewiesen, das Zwischenschema ZS ist hierbei auch erzeugbar.

Beweis:

Es gibt eine Änderungsoperation A mit $A \in \Delta I \backslash \Delta G$
Nehmen wir an, A sei eine erforderliche Operation, die vor ΔG an der Instanz I ausgeführt werden muss.

Da $\Delta G \subset \Delta S$ und
Die Instanz kann nur toleranter gegenüber Änderungsoperationen sein als das Schema.

⇒ **Bevor ΔG auf dem Schema S angewendet werden können, muss es zuerst auch die Operation A am S angewandet werden.**

⇒ **$A \in \Delta S$**

⇒ **$A \in \Delta G$**

aber Voraussetzung: $A \in \Delta I \backslash \Delta G$

⇒ **Widerspruch!**

⇒ **Es gibt keine Operation A die $\in \Delta I \backslash \Delta G$ und vor ΔG an der Instanz I ausgeführt werden muss.**

⇒ **ΔG lässt sich direkt an I anwenden.**

⇒ **Die Zwischeninstanz ZI ist erzeugbar.**

Beweis im Anhang A

⇒ **Das Zwischenschema ZS ist erzeugbar.**

Literaturverzeichnis

[1] Michael Nahler: *Semantische Konflikte in adaptiven Prozess-Management-Systemen.* Diplomarbeit, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2005.

[2] M. Lauer, S. Rinderle, M. Reichert: *Repräsentation von Schema und Instanzobjekten in adaptiven Prozess-Management-Systemen.* Proc Informatik '04, Bd. 2, Ulm (2004), S. 555-560.

[3] Linh Thao Ly, Stefanie Rinderle, and Peter Dadam: *Semantic Correctness In Adaptive Process Management Systems*. Dept.DBIS, University of Ulm.

[4] Stefanie Rinderle, and Peter Dadam: *Schemaevolution in Workflow-Management-Systemen*, Universität Ulm, Fakultät für Informatik, Abteilung DBIS

[5] Stefanie Rinderle, Manfred Reichert, and Peter Dadam: *Evaluation of Correctness Criteria for Dynamic Workflow Changes.* Universität Ulm, Fakultät für Informatik, Abteilung DBIS

[6] Manfred Reichert, Stefanie Rinderle, and Peter Dadam: *On the Common Support of Workflow Type and Instance Changes under Correctness Constraints.* Universität Ulm, Fakultät für Informatik, Abteilung DBIS

[7] Stefanie Rinderle: *Schemaevolution in Process Management Systems.* Dissertation, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[8] Martin Jurisch: *Konzeption eines Rahmenwerkes zur Erstellung und Modifikation von Prozessvorlagen und –instanzen.* Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2006.

[9] Manfred Reichert: *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* Universität Ulm, Fakultät für Informatik, Abteilung DBIS

[10] Markus Lauer: *Effiziente Realisierung von Prozess-Schemaevolution in Hochleistungs- Prozess-Management-Systemen*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[11] Prof. Dr. Peter Dadam, Dr. Stefanie Rinderle: *Workflow-Management-Systeme.* Skript zur Vorlesung Workflow, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2005.

[12] U. Kreher, H. Acker: *Spezifikation einer Änderungsschnittstelle für blockorientierte Prozessgraphen*. Internes Arbeitspapier, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[13] Semantische Korrektheit in adaptive PMS, Interner Bericht, Institut für Datenbanken und Informationssysteme, Universität Ulm (in Bearbeitung)

Erklärung

Hiermit erkläre ich, dass ich die Diplomarbeit „Effiziente Überprüfung semantischer Korrektheit in adaptiven Prozess-Management-System“ vollkommen selbständig geschrieben habe. Außer den Literaturen in der gegebnen List wurde keine andere Literatur verwendet.

ZHOU, Hao