Effiziente Verträglichkeitsprüfung und automatische Migration von Workflow-Instanzen bei der Evolution von Workflow-Schemata

Stefanie Rinderle, Manfred Reichert, Peter Dadam
Universität Ulm
Abt. Datenbanken und Informationssysteme
D-89069 Ulm
E-Mail:{rinderle,reichert,dadam}@informatik.uni-ulm.de

Zusammenfassung

Sollen Workflow-Management-Systeme (WfMS) in umfassender Weise für die rechnerbasierte Verwaltung und Steuerung von Geschäftsprozessen einsetzbar sein, müssen die von ihnen verwalteten Workflow-Schemata und -Instanzen bei Bedarf rasch anpassbar sein. Dabei müssen die auf Basis eines (alten) Workflow-Schemas erzeugten Workflow-Instanzen auch nach dessen Änderung ungestört weiterlaufen können, etwa durch Bereitstellung geeigneter Versionskonzepte. Sehr viel schwieriger wird es, wenn die angewandten Schemaänderungen – wo gewünscht und semantisch möglich – auch auf die bereits (vielleicht in großer Zahl) laufenden Workflow-Instanzen übertragen werden sollen. Dies bei Bedarf zu können – und zwar ohne Inkonsistenzen oder Fehler zu verursachen – ist aber ungemein wichtig, wenn WfMS breit und umfassend einsetzbar sein sollen. In diesem Beitrag wird erstmals ein Ansatz zur effizienten Prüfung auf Migrierbarkeit von WF-Instanzen bei WF-Schemaänderungen vorgestellt. Dabei werden auch fortschrittliche WF-Beschreibungskonstrukte (z. B. Schleifen) in die Betrachtungen einbezogen. Außerdem wird aufgezeigt, wie der Benutzer durch eine effiziente und automatische Migration von Workflow-Instanzen optimal unterstützt werden kann. Bisher sind in diesem Kontext noch viele Fragen offen, sei es in kommerziellen Systemen oder in der Workflow-Literatur. An den Kernteilen dieses Beitrags, wie effiziente Überprüfbarkeit bei Migration von Workflow-Instanzen, umfassende Unterstützung wichtiger Modellaspekte und Benutzerfreundlichkeit durch automatische Anpassungen bei Workflow-Instanzmigrationen, führt kein Weg zu einem umfassenden Änderungsmanagement vorbei.

1 Einleitung

Für Unternehmen gewinnt die elektronische Unterstützung ihrer Geschäftsprozesse zunehmend an Bedeutung. Sowohl für traditionelle Anwendungssysteme (z. B. ERP-Systeme) als auch für Anwendungen im sich rasch entwickelnden E-Business-Bereich (z. B. E-Procurement, Supply Chain Management) wird von Anwenderseite deshalb in verstärktem Maße eine aktive Prozessunterstützung gewünscht [22, 45]. Dasselbe trifft auf Technologien zur unternehmensweiten und -übergreifenden Anwendungsintegration (Enterprise Application Integration) zu [37]. Prozessorientierte Informationssysteme sollen die Durchführung unternehmensweiter Abläufe aktiv koordinieren, Anwendungssysteme prozessorientiert integrieren, Benutzer ablaufbezogen unterstützen und informieren, den Fortgang der Prozesse überwachen und ihren Verlauf lückenlos dokumentieren [27, 39, 43].

Sollen Geschäftsprozesse in solch umfassender Weise elektronisch unterstützt werden, muss beachtet werden, dass prozessorientierte Anwendungen in der Praxis sehr viel häufiger angepasst werden müssen als dies für funktionsorientierte Informationssysteme der Fall ist [17, 30]. Anpassungen können erforderlich werden, wenn neue gesetzliche Bestimmungen in Kraft treten, optimierte oder neu gestaltete Prozesse implementiert werden sollen [23] oder auf ein verändertes Marktgeschehen reagiert werden muss [22]. Wichtig dabei ist, dass die notwendigen Prozessänderungen bei Bedarf sehr rasch, ggf. sogar innerhalb weniger Tage oder Stunden, erfolgen können. Idealerweise sollte auch die Möglichkeit bestehen, die bereits laufenden ,,alten" Prozesse – soweit gewünscht und semantisch sinnvoll – auf die neue Abwicklung umzustellen.

Eine solche Prozessunterstützung findet man bei heutigen betrieblichen Informationssystemen entweder gar nicht oder aber sie ist höchst unflexibel implementiert. Häufig ist die Ablaufsteuerung direkt in den Anwendungsprogrammen codiert, was die Anwendungsentwicklung sehr aufwendig und fehlerträchtig gestaltet. Mit der Einführung jedes neuen Prozesstyps ist zusätzliche Programmierarbeit verbunden, auch spätere Änderungen der Prozesse verursachen hohe Kosten für das Customizing der Software. Für die IT-Abteilungen, die bereits heute mit einer hohen Wartungshypothek leben müssen, ergeben sich dadurch massive Probleme.

Eine vielversprechende Technologie zur Realisierung prozessorientierter Informationssysteme bieten Workflow-Management-Systeme (WfMS) [3, 27, 35, 39, 43]. Charakteristisch für WfMS ist die Trennung von Prozesslogik und Anwendungscode, d. h. die Ablauflogik der unterstützten Arbeitsprozesse (engl. *Workflow* (WF)) wird dem WfMS explizit durch (graphische) Modellierung der Prozesse bekannt gemacht und nicht im Programmcode ,,versteckt". Zu diesem Zweck muss für jeden zu unterstützenden *Prozess-Typ* ein *WF-Schema* (*WF-Modell*) erstellt und im System hinterlegt werden.[1] Ein solches WF-Schema beschreibt die verschiedenen Aspekte eines Arbeitsprozesses, sofern sie für die Ablaufsteuerung relevant sind (z. B. Kontroll- und Datenflüsse, Bearbeiterzuordnungen oder Ausnahmebehandlungen). Des weiteren können einzelnen *WF-Schritten* (sog. *Aktivitäten*) Anwendungsprogramme zugeordnet werden, die dann

[1] Im Allgemeinen kann es zu einem Prozess-Typ mehrere WF-Schemata geben, die unterschiedlichen Versionen entsprechen (siehe später).

während der WF-Bearbeitung zur Ausführung kommen. Ausgehend von einem solchen WF-Schema können neue *WF-Instanzen* erzeugt werden, die vom WfMS über ihre komplette Lebensdauer – ggf. also über Tage, Wochen oder Monate – hinweg gesteuert werden. Die Ausführung, Verwaltung und Überwachung der WF-Instanzen übernimmt das WfMS. Es koordiniert die Durchführung der WF-Schritte, bietet anstehende Aktivitäten den Anwendern über *Arbeitslisten* an, startet die zugehörigen Anwendungsprogramme (mit den entsprechenden Daten) und überwacht die fristgerechte Durchführung von Aktivitäten.

Die beschriebene Trennung von Prozesslogik und Anwendungscode bietet mehrere Vorteile: Die Durchführung der Ablaufsteuerung durch das WfMS vereinfacht die Anwendungsentwicklung erheblich. Durch die explizite Beschreibung der Prozesslogik kann das zukünftige Systemverhalten vorab evaluiert werden, wodurch sich Entwurfsfehler vor der Implementierung der eigentlichen Anwendungskomponenten entdecken lassen. Aus denselben Gründen sind spätere Prozessänderungen und dadurch notwendig werdende Anpassungen der Anwendungsysteme – zumindest vom Prinzip her – sehr viel einfacher durchführbar. Wurde bei der Implementierung der Anwendungsfunktionen sorgfältig vorgegangen, so kann etwa die Reihenfolge der Aktivitäten geändert oder es können neue Schritte hinzugenommen werden, ohne dass hiervon die bereits existierenden Programmbausteine betroffen sind.

1.1 Problembeschreibung

Sollen WfMS in umfassender Weise für die rechnerbasierte Verwaltung und Steuerung von Geschäftsprozessen einsetzbar sein, müssen die von ihnen verwalteten WF-Schemata und WF-Instanzen bei Bedarf rasch anpassbar sein. Solche Änderungen können einen WF-Typ (bzw. sein Schema) als Ganzes oder auch nur einzelne Instanzen betreffen. Bei Änderungen auf WF-Typebene wird man in der Regel fordern, dass die auf Basis des alten WF-Schemas erzeugten Instanzen auch nach Änderung dieses Schemas ungestört weiterlaufen können. Dies lässt sich z. B. durch geeignete Versionskonzepte erreichen. Ihre Anwendung soll sicherstellen, dass zukünftige WF-Instanzen gemäß dem neuen Schema ausgeführt werden, während die bereits aktiven Instanzen dieses Typs weiterhin nach dem alten Schema weitergeführt werden können. Dieser einfache Ansatz mag für Prozesse mit kurzer Dauer ausreichend sein, wirft aber im Zusammenhang mit langlaufenden Prozessen, wie sie z. B. im Krankenhaus- oder Engineering-Bereich [7, 17] auftreten, Probleme auf. Bei dem dann resultierenden ,,Mix" von Instanzen alter und neuer Form muss ggf. für längere Zeit ein Durcheinander in Produktion und/oder Service-Leistungen in Kauf genommen werden. Abgesehen davon ist eine Fortführung der Prozesse auf Grundlage des alten Schemas aus verschiedenen Gründen nicht immer akzeptabel, etwa wenn dadurch gesetzliche Vorschriften oder Geschäftsregeln des Unternehmens (z. B. Behandlungsrichtlinien eines Krankenhauses) verletzt werden. Aus denselben Gründen scheiden auch einige andere der in der Literatur diskutierten Strategien aus, etwa dass eine Änderung von Prozessmodellen erst dann stattfinden kann, wenn alle laufenden Service- oder Produktionsprozesse abgeschlossen sind (man versuche dies mal bei einem Unternehmen) oder dass laufende Prozesse gestoppt und später per Hand zurückgesetzt werden müssen.

Aus diesen Gründen besteht von Anwenderseite der Wunsch, die auf WF-Schemaebene festgelegten Änderungen – wo sinnvoll und möglich – auch auf die bereits (vielleicht in großer Zahl) laufenden WF-Instanzen zu übertragen. Wir sprechen in diesem Zusammenhang auch von der *Propagation* einer WF-Schemaänderung auf laufende WF-Instanzen bzw. von der *Migration* dieser („alten") WF-Instanzen auf das geänderte WF-Schema. Dies bei Bedarf zu können, und zwar ohne es dadurch auf Instanzebene in der Folge zu Inkonsistenzen oder Fehlern kommt, ist ungemein wichtig, wenn ein WfMS breit und umfassend einsetzbar sein soll. Heutige auf dem Markt verfügbare WfMS bieten hierfür keine ausreichende Unterstützung. Sie erlauben es entweder gar nicht, die Änderungen eines WF-Schemas auf bereits laufende WF-Instanzen zu übertragen (z. B. MQSeries Workflow) oder aber dies kann in der Folge zu Inkonsistenzen oder gar Systemabstürzen führen (z. B. Staffware) [38]. Dieser Mangel ist ein wesentlicher Grund für die immer noch recht geringe Verbreitung dieser Systeme.

Für die Abänderung von WF-Schemata und für die Migration von WF-Instanzen auf das neue WF-Schema sind u.a. die folgenden Eigenschaften zu fordern:

- *Vollständigkeit:* Es müssen alle Aspekte eines Prozessmodells, die von Änderungen betroffen sein können, berücksichtigt werden. Dies schließt auch fortschrittliche Kontrollflusselemente (z. B. Schleifen, hierarchische Prozessmodelle) und Datenflüsse mit ein.

- *Korrektheit/Konsistenz:* Eine WF-Instanz darf nur dann auf das geänderte WF-Schema migriert werden, wenn es dadurch in der Folge zu keinen unerwünschten Effekten (z. B. Verklemmungen) kommt. Hierfür muss durch Definition eines geeigneten, allgemeingültigen Korrektheitsbegriffs ein stabiles Fundament geschaffen werden.

- *Koexistenz (von Instanzen alter und neuer Form):* WF-Instanzen, die nicht „verträglich" mit dem neuen WF-Schema sind und die deshalb nicht migriert werden können, sollten „ungestört" weiterlaufen können.

- *Anpassungen im laufenden Betrieb*: Die Propagation von Änderungen eines WF-Schemas auf („alte") WF-Instanzen muss auch im laufenden Betrieb möglich sein.

- *Benutzerfreundlichkeit durch automatische Anpassungen*: Bei der Migration von WF-Instanzen sind (aufwendige) Benutzerinteraktionen zu vermeiden. Dies würde zum einen zu erheblichen Verzögerungen bei der Ausführung der WF-Instanzen führen, zum anderen wäre der Modellierer bzw. Prozessverantwortliche mit dieser Aufgabe vielfach überfordert.

- *Effizienz:* Die auf Instanzebene im Zusammenhang mit Migrationen notwendigen Korrektheitsüberprüfungen und Zustandsanpassungen müssen, selbst bei grosser Anzahl von WF-Instanzen, effizient durchführbar sein. Dabei ist zu beachten, dass sich WF-Instanzen zum Zeitpunkt der Änderungspropagation in unterschiedlichen Zuständen befinden können. Es muss deshalb für jede WF-Instanz zustandsabhängig überprüft werden, ob eine Änderungspropagation möglich ist oder nicht.

Obwohl bereits sehr umfangreich, ist diese Liste noch unvollständig. Weitere ebenfalls sehr wichtige Anforderungen betreffen den Umgang mit nicht migrierbaren WF-Instanzen, die Einbeziehung von Semantik-Aspekten im Zusammenhang mit Verträglichkeitsprüfungen, das Zusammenspiel von Änderungen auf WF-Typ- und WF-Instanz-Ebene sowie Änderungen anderer Bestandteile prozessorientierter Informationssysteme (z. B. Evolution des Organisationsmodells [32]). Auf diese erweiterten Aspekte, die von uns ebenfalls untersucht werden, wird in diesem Beitrag aus Platzgründen nicht eingegangen. Abschließend sei erwähnt, dass die obigen Eigenschaften für die Anwendbarkeit und Praktikabilität eines Lösungsansatzes essentiell sind. Dennoch klammern die in der WF-Literatur diskutierten Vorschläge meist einen oder mehrere dieser Aspekte aus. Dem entsprechend eingeschränkt ist ihre praktische Anwendbarkeit.

1.2 Beitrag

Im ADEPT-Projekt [6, 17, 40] arbeiten wir seit mehreren Jahren an der Entwicklung einer Technologie, die es ermöglichen soll, unternehmensweite Geschäftsprozesse zu modellieren, zu steuern, zu überwachen und sie auf einfache Weise – bei Bedarf auch im laufenden Betrieb – flexibel zu verändern. In unseren bisherigen Veröffentlichungen zu adaptivem Workflow-Management haben wir uns mit Ad-hoc-Änderungen einzelner WF-Instanzen und mit der Modellierung planbarer Abweichungen befasst [10, 17, 42]. Schwerpunkte bildeten dabei unter anderem die Definition geeigneter Änderungsoperationen (einschließlich Graphtransformationsregeln und -vereinfachungsregeln) sowie Korrektheits-, Skalierbarkeits- und Implementationsaspekte.

Ziel des vorliegenden Beitrags ist die Entwicklung eines umfassenden und theoretisch fundierten Ansatzes für WF-Typänderungen und deren Progagation auf bereits laufende WF-Instanzen. Dies erfordert zum einen die Definition von (formalen) Kriterien zur effizienten Überprüfung der Verträglichkeit von WF-Instanzen mit dem neuen WF-Schema, zum anderen sind geeignete Mechanismen für die Migration verträglicher WF-Instanzen auf das neue WF-Schema vonnöten. Die zu entwickelnde Lösung muss dabei den im vorangehenden Abschnitt skizzierten Anforderungen genügen. Insbesondere muss die Propagation von Schemaänderungen auch bei einer großen Zahl von WF-Instanzen effizient und korrekt bewerkstelligt werden können.

Zu diesem Zweck werden von uns allgemeingültige, formale Kriterien definiert, auf deren Grundlage feststellbar ist, ob eine gegebene WF-Instanz mit dem aus einer Änderung hervorgehenden WF-Schema verträglich ist oder nicht. Wir werden uns dabei an existierenden Vorschlägen orientieren, diese aber – soweit sinnvoll und nötig – geeignet erweitern. Zu diesem Zweck diskutieren wir, inwieweit existierende Ansätze unvollständig oder zu restriktiv sind. Auf Grundlage der definierten Verträglichkeitskriterien untersuchen wir dann systematisch für verschiedene Klassen von Schemaänderungen (z. B. additive und subtraktive Änderungen), welche Auswirkungen sie auf die Verträglichkeit von WF-Instanzen haben bzw. welche formalen Voraussetzungen zu stellen sind, um Verträglichkeit mit dem neuen Schema zusichern zu können. Ein wichtiges Anliegen stellt dabei die effiziente Überprüfbarkeit dieser Voraussetzungen dar.

Zur besseren Differenzierung stellen wir diese Betrachtungen für verschiedene Klassen von

WF-Graphen an. Dies dient zum einen dem besseren Verständnis des entwickelten Lösungsansatzes, zum anderen wird dadurch eine bessere Einordnung existierender WF-Metamodelle möglich. Beispielsweise zeigen wir, welche zusätzlichen Fragestellungen sich in Verbindung mit der Unterstützung von Schleifen ergeben und warum bisherige Ansätze hier zu kurz springen. Auch andere wichtige Modellaspekte, wie Datenflüsse oder hierarchische Workflows, werden in diese Betrachtungen mit einbezogen.

Die effiziente Prüfung auf Verträglichkeit ist jedoch nur eine Seite der Medaille. Ebenso wichtig ist natürlich die Durchführung der Migrationen selbst. Gerade hier zeigt sich, ob ein gewählter Lösungsansatz für die Praxis tauglich ist und ob er auch bei großer Zahl von WF-Instanzen anwendbar bleibt. Wir diskutieren in diesem Zusammenhang zunächst prinzipielle Lösungsansätze und stellen dann ein Verfahren vor, mit dem sich die für einzelne WF-Instanzen notwendigen Zustandsanpassungen automatisch vornehmen lassen. Dieses Verfahren ist so konstruiert, dass einerseits der Umfang der erforderlichen Zustandsneubewertungen auf ein Minimum reduziert wird, andererseits im Anschluss an eine Migration wieder ein korrekter und konsistenter Ausführungszustand resultiert. Schließlich diskutieren wir, wie im Fall nicht migrierbarer WF-Instanzen verfahren werden kann. Auch hier ergeben sich im Zusammenhang mit Schleifen wieder interessante Aspekte, wie sie von bisherigen Ansätzen nicht aufgegriffen worden sind.

Im nachfolgenden Kapitel fassen wir wichtige Grundlagen zusammen, die für das weitere Verständnis erforderlich sind. In Kapitel 3 definieren wir erstmals formale Kriterien für die Verträglichkeit von WF-Instanzen mit einem geänderten Schema. Wir zeigen, unter welchen Voraussetzungen diese Kriterien erfüllt sind und wie sich diese effizient überprüfen lassen. Kapitel 4 behandelt dann die Migration von verträglichen WF-Instanzen auf ein geändertes Schema und diskutiert Strategien für den Umgang mit nicht migrierbaren Fällen. Eine Einordnung des Beitrags und eine Abrenzung zu verwandten Arbeiten findet sich in Kapitel 5. Der Beitrag schließt mit einer Zusammenfassung und einem Ausblick auf zukünftige Arbeiten.

2 Grundlagen

Den in diesem Beitrag angestellten Betrachtungen legen wir das von uns entwickelte ADEPT Workflow-Metamodell [42, 44] zugrunde. Dieses Metamodell ist einerseits ausdrucksmächtig genug, um damit reale Prozesse abbilden zu können, andererseits sind die beschreibbaren WF-Modelle aufgrund ihrer Struktureigenschaften (siehe unten) sowohl für den Modellierer als auch für den Benutzer gut verständlich [17]. Modelliert werden können alle relevanten Aspekte eines Arbeitsprozesses, wie Kontroll- und Datenfluss, Bearbeiterzuordnungen, zeitliche Abhängigkeiten oder planbare Abweichungen. Darüberhinaus hat sich das ADEPT-Metamodell im Zusammenhang mit anderen wichtigen Aspekten, wie statischen und dynamischen Korrektheitseigenschaften von WF-Modellen [44], Ad-hoc-Änderungen einzelner WF-Instanzen [42] oder Partitionierung von WF-Schemata und verteilte WF-Ausführung [6] sehr gut bewährt. Auch das Zusammenspiel dieser Aspekte haben wir unter Verwendung des ADEPT-Metamodells bzw. des darauf basierenden ADEPT WfMS eingehend untersucht[10, 26]. Wir werden im Folgenden darlegen, dass das ADEPT-Metamodell auch für die Evolution von WF-Schemata und für die in diesem Zusammenhang notwendigen Anpassungen auf WF-Instanzebene gut geeignet ist.

In diesem Abschnitt fassen wir zunächst wichtige Eigenschaften des ADEPT-Metamodells zusammen. Sie sind für das weitere Verständnis dieser Arbeit notwendig und bilden die Voraussetzung für die präzise Formulierung formaler Aussagen zur Propagation von Schemaänderungen auf laufende WF-Instanzen. An geeigneter Stelle werden wir darauf hinweisen, inwieweit die präsentierten Ergebnisse auch auf andere WF-Metamodelle übertragbar sind bzw. unabhängig vom ADEPT-Metamodell betrachtet werden können.

2.1 WF-Modellierung und -Ausführung

Die Spezifikation eines Arbeitsprozesses erfolgt durch graphische Modellierung des WF-Schemas. Dieses legt unter anderem die Ausführungsreihenfolgen und -bedingungen der einzelnen Aktivitäten, d. h. den Kontrollfluss zwischen ihnen, fest. Intern werden solche Kontrollflüsse bei unserem Ansatz durch attributierte WF-Graphen repräsentiert, die sich durch die Unterscheidbarkeit von Knoten-/Kantentypen auszeichnen. Wie wir noch sehen werden, erleichtert dies zum einen die (effiziente) Analysierbarkeit der WF-Modelle, zum anderen ergeben sich auch für die interpretative Ausführung der WF-Graphen gewisse Vorteile. Zur formalen Repräsentation eines WF-Schemas S und zur präzisen Definition von WF-Schemaänderungen verwenden wir eine mengenbasierte Darstellung S = (N, E, ...), wobei N die Knotenmenge und E die Kantenmenge von S beschreibt.

Zur Differenzierung verschiedener Arten von Kontrollkanten (zwischen Aktivitäten) wird jeder Kante e (intern) ein Kantentyp $ET(e)$ aus der Menge `EdgeTypes` = {`CONTROL_E, SYNC_E, LOOP_E`} zugeordnet. Dabei definiert der Kantentyp `CONTROL_E` „normale" Kontrollkanten bzw. Reihenfolgebeziehungen, der Kantentyp `SYNC_E` eine „Wartet-auf"-Beziehung (zwischen Aktivitäten paralleler Ablaufzweige) und `LOOP_E` Schleifenrücksprungkanten. Auch Knoten n können verschiedene Typen $NT(n)$ besitzen. Sie sind in der Menge `NodeTypes` = {`STARTFLOW, ENDFLOW,`

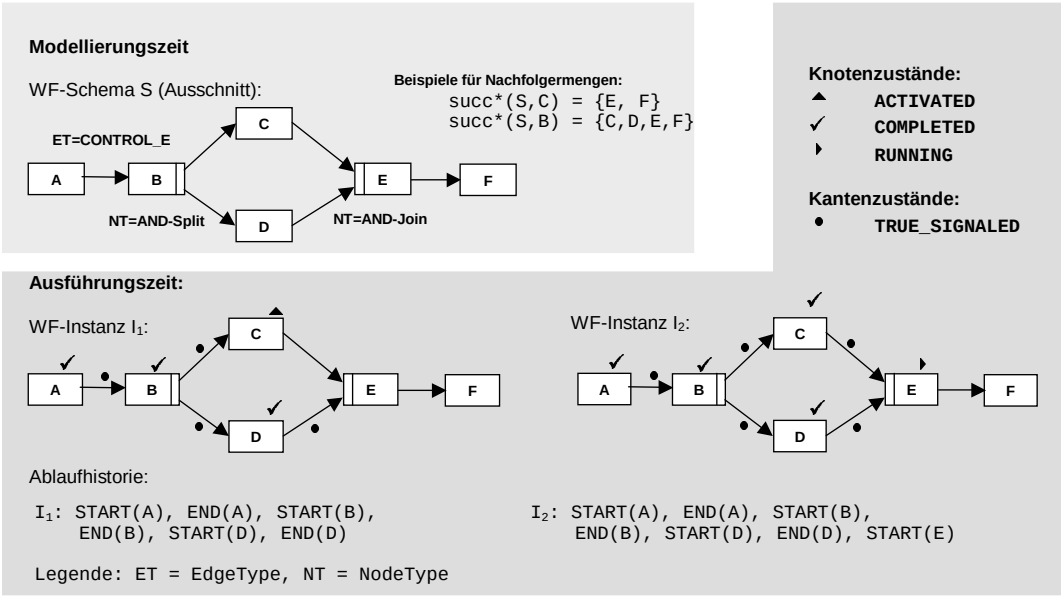


Abbildung 1: Modellierung und Ausführung von Workflows in ADEPT

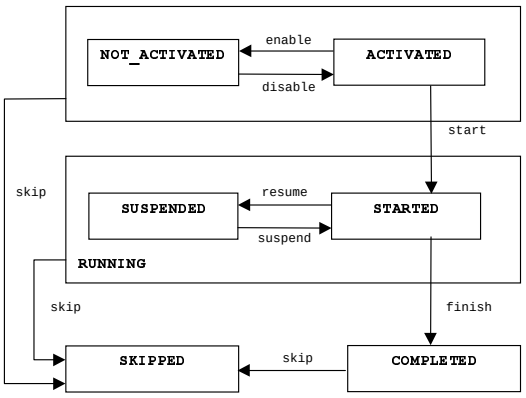
ACTIVITY, STARTLOOP, ENDLOOP, AND-Split, AND-Join, XOR-Split, XOR-Join} (siehe Tabelle 4, Anhang A) zusammengefasst. Wir werden auf die Bedeutung der verschiedenen Knoten- und Kantentypen sowie auf deren Semantik an gegebener Stelle zurückkommen.

Auf Grundlage dieser Modellelemente können Sequenzen, parallele Verzweigungen (AND-Split, AND-Join), alternative Verzweigungen (XOR-Split, XOR-Join) und Schleifen (STARTLOOP, ENDLOOP}) modelliert werden. In unserem Ansatz erfolgt dies über ein („aufgelockertes") Blockkonzept, bei dem Sequenz-, Verzweigungs- und Schleifen-Blöcke ineinander verschachtelt sein können, sich aber nicht überlappen dürfen (für Details siehe [44]). Wir verwenden diese Blockstrukturierung aus Darstellungsgründen auch in diesem Beitrag, sie ist aber für die angestellten Betrachtungen nicht zwingend erforderlich, d. h. die gewonnenen Ergebnisse lassen sich unter gewissen Voraussetzungen auch auf andere WF-Beschreibungsformalismen bzw. WF-Ausführungsmodelle übertragen.

Bei den nachfolgenden Betrachtungen verwenden wir zusätzlich verschiedene Vorgänger- und Nachfolgerfunktionen auf WF-Graphen, die abhängig von den berücksichtigten Kantentypen mehr oder weniger umfangreiche Knotenmengen liefern. Exemplarisch sei an dieser Stelle die Funktion $succ^*(S,n)$ genannt, die für ein WF-Schema S = (N, E, ...) alle direkten und indirekten Nachfolger des Knotens n ∈ N liefert (transitive Hülle, siehe Abb. 1). Alle weiteren, in dieser Arbeit verwendeten Vorgänger- und Nachfolgerfunktionen sind in der Tabelle 4 in Anhang A informell zusammengefasst.

Ausgehend von einem WF-Schema S können zur Ausführungszeit neue WF-Instanzen erzeugt

und gestartet werden. Sie werden dann über ihre komplette Lebenszeit vom WfMS gesteuert. Damit der WF-Server die für eine bestimmte WF-Instanz aktuell zur Ausführung anstehenden Aktivitäten ermitteln kann, benötigt er entsprechende Zustandsinformationen. In ADEPT werden dazu jedem Knoten n und jeder Kante e auf Instanzebene Zustandsmarkierungen $NS(n)$ bzw. $ES(e)$ zugeordnet. Ihre Festlegung bzw. Veränderung erfolgt nach wohldefinierten Regeln, auf die wir später im Zusammenhang mit Zustandsanpassungen bei Migrationen noch im Detail eingehen werden (vgl. Abb. 18).



Der Zustand einer einzelnen Aktivität (siehe Abbildung links) ist initial `NOT_ACTIVATED` und geht bei ihrer Aktivierung in `ACTIVATED` über. Bei manuell auszuführenden Aktivitäten erhalten dann die potenziellen Bearbeiter dieses Schrittes (sie werden über Bearbeiterzuordnungsausdrücke festgelegt) die entsprechenden Arbeitsaufträge in ihren Arbeitslisten angeboten. Wird eine solche Aktivität zur Bearbeitung ausgewählt und gestartet, geht sie in den Zustand `RUNNING` über. Dabei werden auch die entsprechenden Arbeitsaufträge aus der Arbeitsliste entfernt (mit Ausnahme des Eintrags für den aktuellen Bearbeiter).

Bei erfolgreicher Beendigung einer Aktivität erhält sie den Status `COMPLETED`. Steht fest, dass eine Aktivität in der Folge nicht mehr ausführbar ist wird sie durch den Zustand `SKIPPED` gekennzeichnet. Kanten wird initial der Zustand `NOT_SIGNALED` zugeordnet. Im Verlauf der Ausführung können sie dann entweder mit `TRUE_SIGNALED` oder `FALSE_SIGNALED` markiert werden (siehe Abb. 18).

Grundlegend für unsere nachfolgenden Betrachtungen ist außerdem die *Ablaufhistorie* einer WF-Instanz (vgl. Abb. 1), in welcher u. a. beim Starten und Beenden von Aktivitäten entsprechende Einträge protokolliert werden. Präziser ausgedrückt, schreibt jede Aktivität beim Übergang in den Zustand `RUNNING` einen Starteintrag, beim Übergang in den Zustand `COMPLETED` einen Endeintrag in die Ablaufhistorie. Diese Einträge enthalten auch Informationen zum Zeitpunkt der betreffenden Ereignisse und zu den bearbeitenden Akteuren.

2.2 Definition und Änderungen von WF-Schemata

Eine zentrale Forderung bei der Erstellung und Änderung von WF-Schemata ist, dass die Korrektheit und Konsistenz von Modell- und Instanzdaten zu jedem Zeitpunkt gewährleistet sind. Insbesondere müssen sowohl für laufende als auch für zukünftige WF-Instanzen unerwünschte Effekte, wie Daten-Inkonsistenzen, Verklemmungen oder fehlerhafte Aufrufe von Aktivitätenprogrammen (z. B. infolge von unvollständigen Eingabedaten), ausgeschlossen werden. Notwendige Voraussetzung hierfür ist, dass nach Modifikation eines (korrekten) Quell-Schemas S

wieder ein korrektes Ziel-Schema S' resultiert. Diese Forderung betrifft zunächst einmal den statischen Fall, bei dem nur die Transformation des WF-Schemas (ohne Instanzen) betrachtet wird. In unserem Ansatz erfüllen wir diese Forderung, indem der Modellierer ein korrektes WF-Schema S mittels eines syntaxgesteuerten WF-Editors sowie einer vollständigen Menge von Änderungsoperationen modifizieren kann. Für diese Änderungsoperationen gibt es formale Vor- und Nachbedingungen, so dass nach ihrer Anwendung wieder ein korrektes Schema S' resultiert.

Die Korrektheit im statischen Fall steht nicht im Fokus dieser Arbeit, ist aber eine wichtige Voraussetzung im Kontext der Evolution von WF-Schemata. Deshalb werden wir im Folgenden an entsprechenden Stellen darauf verweisen. An dieser Stelle sei auch erwähnt, dass der Modellierer in unserem Ansatz WF-Schemata auf semantisch sehr hoher Ebene abändern kann. Hierzu werden ihm mächtige Operationen an die Hand gegeben, deren korrekte Anwendung durch zahlreiche Überprüfungen unterstützt wird. Auf die Darstellung dieser ,,High-End" Operationen und deren formalen Vor- und Nachbedingungen verzichten wir an dieser Stelle. Ihre Beschreibung und Semantik wurde von uns in anderen Arbeiten detailliert vorgestellt [42, 44]. Intern lassen sich solche komplexen Operationen auf einfache Einfüge- und Löschoperationen von Knoten und Kanten abbilden. In diesem Beitrag beschränken wir uns weitgehend auf diese Elementaroperationen. Dies ist zum einen ausreichend, zum anderen ermöglicht es eine bessere Übertragbarkeit auf andere Ansätze.

3 Effiziente Überprüfung der Verträglichkeit von WF-Instanzen bei Schemaänderungen

In diesem Abschnitt zeigen wir, wie sich effizient überprüfen lässt, ob bzw. wann Änderungen eines WF-Schemas (unter Erhalt von Korrektheits- und Konsistenzeigenschaften) auf laufende WF-Instanzen propagiert werden können. Dazu diskutieren wir in Abschnitt 3.1 zunächst generelle Probleme, die sich bei unkontrollierter Migration von WF-Instanzen ergeben können. Anschließend definieren wir ein formales Kriterium zur Verträglichkeit von WF-Instanzen mit einem geänderten WF-Schema. In den darauffolgenden Abschnitten 3.3 und 3.4 wird dann erstmals systematisch dargelegt, wie sich dieses Kriterium für verschiedene WF-Graphklassen und für verschiedene Änderungsarten effizient überprüfen lässt.

3.1 Probleme bei unkontrollierter Migration von WF-Instanzen

Wie bereits in Abschnitt 1.2 diskutiert, sollte es generell möglich sein, Änderungen des WF-Schemas auf die laufenden WF-Instanzen zu propagieren. Für eine solche Änderungspropagation muss gewährleistet sein, dass eine WF-Instanz des Quell-Schemas S auch als WF-Instanz auf dem modifizierten WF-Schema S' ausgeführt werden kann, ohne dass es dadurch in der Folge zu Inkonsistenzen kommt. Um dem gerecht zu werden, sollte eine Änderungspropagation für eine bestimmte Instanz nur möglich sein, wenn die Änderung mit der bisherigen Ausführung verträglich ist („keine Verfälschung der Vergangenheit"). Die Verletzung dieses Grundsatzes kann zur Laufzeit zu Inkonsistenzen oder gar Fehlern führen, insbesondere im Zusammenhang mit komplexen Änderungen des WF-Schemas. Wir illustrieren dies anhand zweier Beispiele:

Beispiel 3.1 (Einfügen von Aktivitäten) In das WF-Schema S aus Abb. 2a) werden zwei neue Schritte und eine Datenabhängigkeit zwischen ihnen eingefügt, so dass ein korrektes WF-Schema S' resultiert. Abb. 2b zeigt zwei von WF-Schema S abgeleitete WF-Instanzen I_1 und I_2. Bei Propagierung der skizzierten Schemaänderung auf I_1 würde der Schritt `check allergies` zunächst in einen bereits durchlaufenen Bereich eingefügt werden, wodurch das Ausgabedatum `patient data` vor Aktivierung des datenabhängigen Schritts `administer medicine` nicht mehr geschrieben werden könnte. Liest die ebenfalls neu eingefügte Aktivität `administer medicine` das Datum `patient data` obligat, bleiben demzufolge Eingabeparameter des zugeordneten Aktivitätenprogramms unversorgt. Dadurch kommt es zu einem fehlerhaften Aufruf mit möglicherweise schwerwiegenden Konsequenzen (z. B. Gabe des falschen Medikaments, Absturz des Aktivitätenprogramms etc.). Im Fall von WF-Instanz I_2 dagegen könnte die Aktivität `check allergies` das Ausgabedatum `patient data` schreiben, so dass die Aktivität `administer medicine` bzw. das ihr zugeordnete Aktivitätenprogramm korrekt mit diesem Eingabedatum versorgt werden kann.

Beispiel 3.2(Löschen von Aktivitäten) In Abb. 3 wird auf Instanzebene der bereits beendete Schritt `collect patient data` gelöscht, welcher sowohl Daten über ein zugeordnetes Anwendungsprogramm in eine externe Datenbank als auch die Prozessvariable `patient data`

geschrieben hat. Dies führt einerseits zu Inkonsistenzen zwischen WF-Datenbank und externer Datenbank. Andererseits hätte dann der (bereits beendete) Schritt `instruct patient` eine Prozessvariable gelesen, deren Wert von der nunmehr gelöschten Aktivität geschrieben wurde (Dirty-read-Problematik). Abgesehen davon ist das nachträgliche Löschen bereits beendeter Prozess-Schritte in vielen Domänen, etwa der Medizin, allein aus juristischen Gründen nicht zulässig [17].

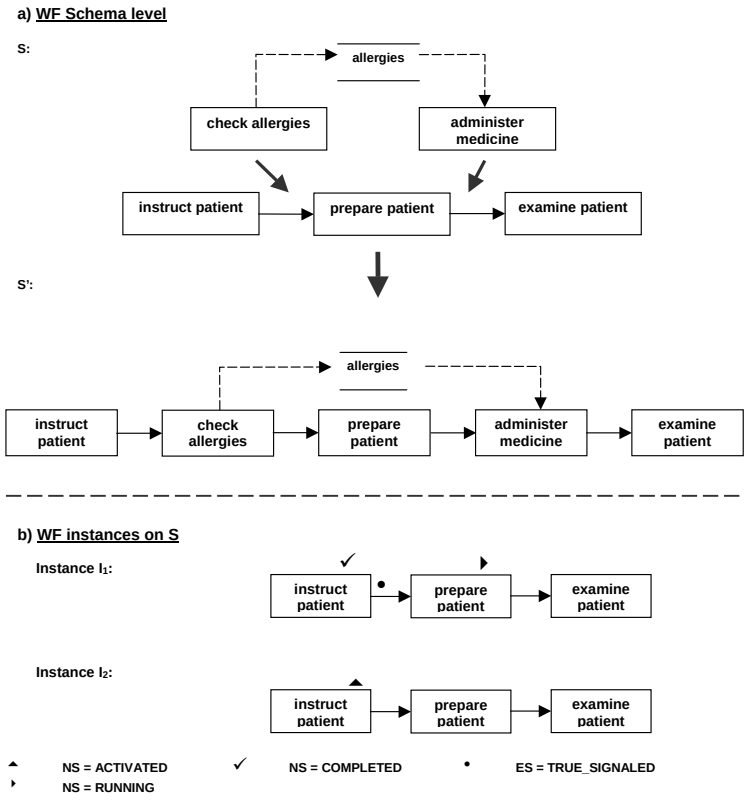


Abbildung 2: Einfügen von Aktivitäten

3.2 Verträglichkeit von WF-Instanzen mit dem geänderten WF-Schema

Damit das WfMS entscheiden kann, welche WF-Instanzen korrekt auf ein geändertes WF-Schema migrieren können und welche nicht, werden geeignete Regeln benötigt. Bezüglich der Propagierung von Schemaänderungen auf laufende WF-Instanzen gibt es in der WF-Literatur bereits ein allgemein anerkanntes Kriterium, das sog. ,,Compliance"-Kriterium (siehe z. B. [12], [29], [33], [46], [48], [52]). Mit seiner Hilfe kann überprüft werden, ob laufende WF-Instanzen mit dem geänderten WF-Schema verträglich (*compliant*) sind oder nicht. Wir wollen uns im Folgenden an diesem Kriterium orientieren, dabei aber die Limitationen der bisherigen, darauf

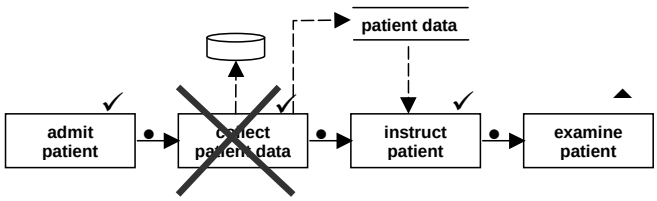


Abbildung 3: Löschen von Aktivitäten

basierenden Ansätze aufzeigen.

Ein Compliance-Kriterium für die Migration von WF-Instanzen bei Schemaänderungen wurde erstmals in [12] vorgestellt. Hierbei wird für alle bereits beendeten Aktivitäten einer WF-Instanz von WF-Schema S gefordert, dass diese auf dem geänderten WF-Schema S' ebenfalls beendet worden wären und sich die selben Effekte auf die Variablen (Daten) des Workflow ergeben hätten. Die Überprüfung dieser Bedingung erfordert, dass Informationen zum Start und Ende von Aktivitäten vorliegen. Im ADEPT-WfMS werden diese Informationen, wie in einigen anderen Ansätzen auch, über eine Ablaufhistorie verwaltet. Diese wird in ADEPT unter anderem für das korrekte Zurücksetzen von WF-Instanz im Fehlerfall verwendet. Die Definition von Verträglichkeit auf Grundlage der Ablaufhistorie lautet dann: Eine von S abgeleitete WF-Instanz I ist genau dann verträglich mit dem geänderten WF-Schema S', wenn die Ablaufhistorie von I auch bei der Ausführung auf S' erzeugbar gewesen wäre. Formal:

Definition 3.1 (Compliance-Kriterium) *Sei S = (N, E, ...) ein korrektes WF-Schema und I eine davon abgeleitete WF-Instanz mit Ablaufhistorie* $\mathcal{A} =< e_0, \ldots, e_t >$ *und Markierung* M_t*;* $e_0, \ldots, e_t$ *entsprechen dabei den Start-/Endereignisse zu laufenden bzw. beendeten Aktivitäten, deren sequentielle Anwendung auf I die WF-Instanz von ihrer Anfangsmarkierung* M_0*, über Zwischenmarkierungen* $M_1, \ldots, M_{t-1}$*, in ihre aktuelle Markierung* M_t *überführt hat (kurz:* $M_0[e_0 > M_1[e_1 > \ldots M_{t-1}[e_t > M_t)$[2]*.*

S werde nun durch eine Änderung Δ *auf das (korrekte) WF-Schema S' transformiert. Sei* M'_t *die Markierung des Ausführungsgraphen von I, der sich ergibt, wenn* Δ *auf I propagiert wird und anschließend eine Neubewertung des Zustands des Ausführungsgraphen (nach wohldefinierten Regeln) durchgeführt wird. Dann ist I genau dann verträglich mit S', wenn die selbe Folge von Ereignissen* $e_0, \ldots, e_t$*, ausgehend von einer Anfangsmarkierung* M'_0*, auf dem geänderten WF-Schema S' anwendbar ist und im Anschluss wieder die konsistente Markierung* M'_t *resultiert. Formal:*

$$I \text{ ist verträglich mit } S' :\Leftrightarrow M'_0[e_0 >, \ldots, [e_t > M'_t$$

Diese Definition ist allgemeingültig, d. h. sie ist unabhängig vom verwendeten WF-Ausführungs-

[2] $M_i[e_i > M_{i+1}$ bedeutet hierbei, dass die Markierung M_i durch Anwendung des Ereignisses e_i in die Markierung M_{i+1} überführt wird

modell. Eine Instanz, die im Sinne dieser Definition verträglich mit dem neuen WF-Schema ist, kann problemlos migriert werden. Insbesondere ist auch in der Folge ihre korrekte Ausführbarkeit gewährleistet. Darüber hinaus bleiben andere, hier nicht weiter betrachtete orthogonale Systemfunktionen (z. B. semantisches Rollback) unverändert bestehen. Mit einigen wenigen Ausnahmen (z. B. [33]) wird in der Literatur jedoch nicht diskutiert, ob bzw. wie sich Compliance einer WF-Instanz mit einem neuen Schema effizient überprüfen lässt. Ebensowenig gibt es systematische Betrachtungen dazu, wie sich verschiedene Arten von Änderungen (additiv, subtraktiv, ordnungsverändernd) für verschiedene WF-Graphklassen auf die Verträglichkeit von WF-Instanzen auswirken und welche Informationen für entsprechende Verträglichkeitsprüfungen benötigt werden. Wie wir noch zeigen werden, können entsprechende Betrachtungen sinnvoll sein, um entscheiden zu können, wann für Verträglichkeitsprüfungen Zugriffe auf die (komplette) Ablaufhistorie erforderlich sind und wann dies ausgeschlossen werden kann. Bevor wir darauf eingehen, wollen wir das obige Kriterium anhand eines einfachen Beispiels illustrieren.

Beispiel 3.3 (Compliance-Kriterium) In Abb. 4a ist der Schritt `instruct patient` beendet und der Schritt `examine patient` gestartet. Dementsprechend enthält die Ablaufhistorie Informationen zum Start bzw. Ende dieser Aktivitäten. Wird nun auf Schemaebene die Aktivität `check allergies` vor `calculate dose` eingefügt, könnte die bisherige Ablaufhistorie auch auf dem geänderten WF-Schema erzeugt werden. Damit ist die gegebene WF-Instanz verträglich mit dem geänderten WF-Schema im Sinne der obigen Definition. Sie kann deshalb ohne Probleme auf das neue WF-Schema migriert werden und ihre Ausführung im Folgenden auf Grundlage dieses WF-Schemas S' erfolgen. Dies wäre z. B. nicht möglich, wenn der Schritt `check allergies` vor der bereits gestarteten Aktivität `examine patient` eingefügt wird. In diesem Fall müsste `check allergies` seine Start- und Endeinträge vor denen von `examine patient` in die Ablaufhistorie schreiben. Dies ist jedoch nicht mehr möglich, da die Ausführungshistorie bereits einen Eintrag `Start(examine patient)` enthält.

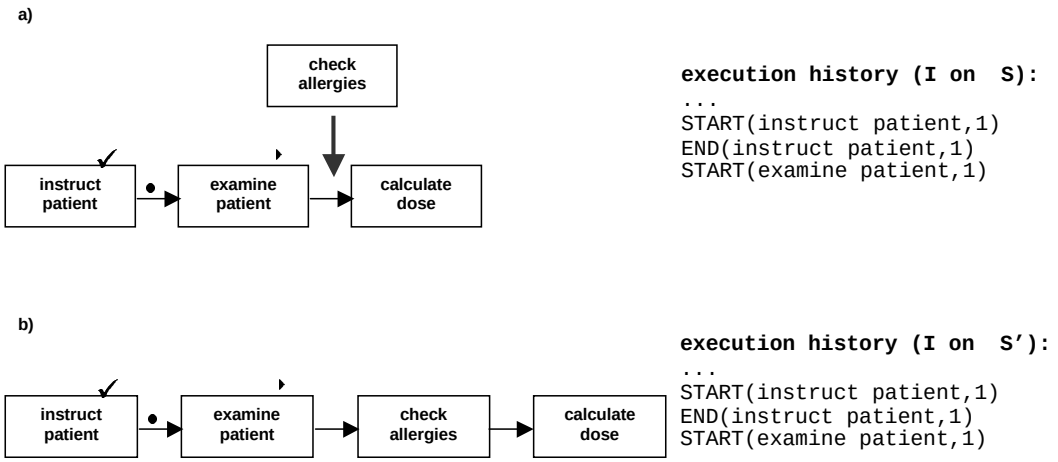


Abbildung 4: Korrektes Einfügen mit Ablaufhistorie (vereinfacht)

Wie soll nun die Erzeugbarkeit der Ablaufhistorie einer gegebenen WF-Instanz I eines Quell-Schemas S auf dem aus einer Änderung von S hervorgehenden Ziel-Schema S' überprüft werden? Eine naive Lösung wäre es, hierfür stets die komplette Ablaufhistorie heranzuziehen, da diese sehr umfangreich sein kann (siehe Abb. 6), was aufwendige Überprüfungen zur Folge hat. So müssen bei der Ausführung einer einzelnen WF-Instanz alle relevanten Ereignisse (z. B. zum Start, zur Unterbrechung und zur Beendigung von Aktivitäten) in einer Logdatei protokolliert werden. Neben Angaben zum Typ des jeweiligen Ereignisses umfasst ein einzelner Historieneintrag relevante Ausführungsinformationen zu Bearbeitern oder Zeitpunkten der mit den Ereignissen verknüpften Aktionen. Die Informationen werden z. B. benötigt, um im Fehlerfall ein korrektes Rücksetzen der WF-Instanz in einen früheren Bearbeitungs-Zustand durchführen zu können oder um gewisse Abhängigkeiten zwischen Aktivitäten (z. B. Zeitabstände) auch nach Systemabsturz aufrecht erhalten zu können.

Die Größe der Ablaufhistorie einer WF-Instanz kann in Verbindung mit Schleifen beträchtlich sein, da für jede Iteration die Informationen zur Ausführung der Schleifen-Aktivitäten protokolliert werden. Wie umfangreich Ablaufhistoriendaten werden können, soll folgendes Rechenbeispiel illustrieren.

Beispiel 3.4 (Größe einer Ablaufhistorie) Ein einzelner Eintrag der Ablaufhistorie einer WF-Instanz bezieht sich auf ein konkretes Ausführungsereignis dieser WF-Instanz und umfasst in jedem Fall folgende Informationen (in Klammern der für die Speicherung dieser Daten jeweils benötigte Datentyp bzw. Speicherplatz):

- *Ereignistyp*, z. B. Start/Ende einer Aktivitätenbearbeitung (`Short, 2 Bytes`)
- *Bearbeiter*, z. B. Starter einer Aktivitätenbearbeitung (`Long, 8 Bytes`)
- *Zeitstempel*, d. h. Zeitpunkt des Ereignisses (`Long, 8 Bytes`)
- *Iterationszähler*: er gibt für einzelne Aktivitäten an, zum wie vielten Male sie im Verlauf der WF-Ausführung gestartet bzw. beendet wurden (`Short, 2 Bytes`)
- *Verzweigungsentscheidungen*, d. h. Informationen zu gewähltem Ablaufzweig bei XOR-Aufsplittungen und Schleifenendknoten (`Short, 2 Bytes`)

Damit ergibt sich die Größe eines einzelnen Ablaufhistorieneintrags zu insgesamt `22 Byte`. Sei nun ein WF-Schema mit 100 WF-Aktivitäten gegeben, wobei 40 dieser Aktivitäten in eine Schleife eingebunden sind (siehe [49]). Diese Schleife wird durchschnittlich (über alle WF-Instanzen betrachtet) 6-mal durchlaufen. Angenommen es sind 40000 WF-Instanzen auf diesem WF-Schema gleichzeitig aktiv (eine solche Anzahl von WF-Instanzen ist nach [31, 50] durchaus nicht ungewöhnlich), so ergibt sich bereits für diesen einen WF-Typ eine Logfile-Größe von ca. `251,77 MB`.

In den meisten WfMS wird die Ablaufhistorie aufgrund ihrer Größe nicht im Hauptspeicher gehalten, sondern auf Externspeicher ausgelagert. Dies ist in der Regel auch ausreichend, da

Zugriffe auf die komplette Ablaufhistorie nur in Verbindung mit selteneren Operationen (z. B. Rollback, Crash-Recovery) erforderlich sind. Aus diesem Grund sollten auch die bei Schemaevolution notwendigen Verträglichkeitsprüfungen idealerweise ohne Zugriffe auf die Gesamtheit der Historiendaten erfolgen können.

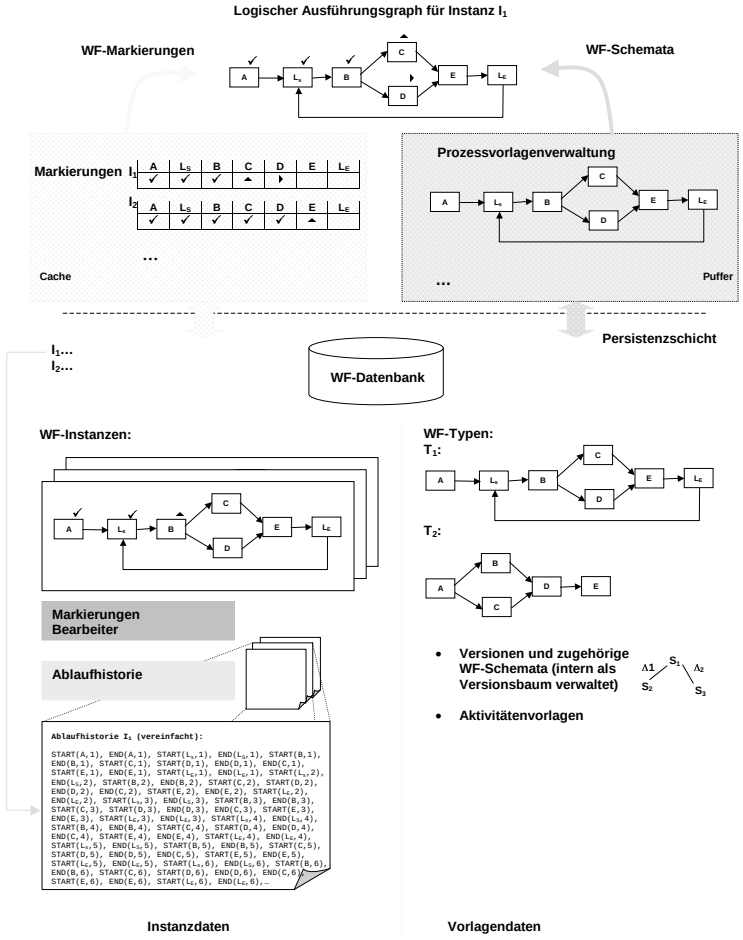


Abbildung 5: Verwaltung von Prozessvorlagen und Prozessinstanzdaten in ADEPT (bezogen auf Kontrollfluss-Aspekte)

Bei genauerer Betrachtung müssen für Verträglichkeitsprüfungen nicht die kompletten Historieninformationen herangezogen werden, sondern man kann Verträglichkeitsaussagen in der Mehrzahl der Fälle auch anhand wesentlich weniger Informationen treffen. Welche Informationen hierfür konkret benötigt werden, werden wir im Folgenden für verschiedene Arten von Änderungen aufzeigen. Des weiteren werden wir sehen, dass im Zusammenhang mit Schleifen keine Informationen zu Ereignissen früherer Schleifeniterationen notwendig sind. Im Kern reichen Informationen zur aktuellen Markierung von Knoten für Verträglichkeitsprüfungen aus. Solche

Informationen werden z. B. in dem von uns implementierten ADEPT-WfMS ohnehin (teilweise) im Hauptspeicher gehalten und dort sowohl für die interpretative Ausführung der Workflows als auch zur Prüfung der Zulässigkeit von Ad-hoc-Änderungen verwendet (siehe Abb. 5). Wird für die Markierung einzelner Knoten `1 Byte` verwendet, so kann bei Zugrundelegung der Daten aus Beispiel 3.4 das für Verträglichkeitsprüfung notwendige Datenvolumen mit `3,81 MB` nach oben abgeschätzt werden. Dies entspricht lediglich 1,6% des in Beispiel 3.4 berechneten Historienumfangs. (Insbesondere wird hier jeder Knoten auf seinen Zustand hin untersucht. In der Realität kommt man jedoch mit deutlich weniger Zustandsprüfungen aus.)

Darüberhinaus ist das in Definition 3.1 genannte Kriterium, das von einigen Ansätzen wie WIDE [12], MOKASSIN [29], [33], BREEZE [48], $WASA_2$ [52] verwendet wird, noch zu restriktiv. Beispielsweise wird dadurch eine Migration von WF-Instanzen bei Änderungen des WF-Schemas innerhalb von Schleifen oftmals ausgeschlossen.

Beispiel 3.5 (Compliance bei Änderungen innerhalb von Schleifen) Betrachten wir hierzu Abb. 6. Angenommen, es ist eine WF-Instanz gegeben, die auf dem Quell-Schema S die dargestellte Ablaufhistorie erzeugt hat. Es ist offensichtlich, dass diese Historie bei der Ausführung des aus S nach Einfügen des Schritts `check allergies` hervorgehenden WF-Schemas S' nicht erzeugbar ist. Nach Instantiierung von S' würde nämlich bereits in der ersten Iteration der Schleifenausführung auch die Information über den Start und das Ende der Aktivität `check allergies` in die Ablaufhistorie geschrieben. Damit ist eine Migration der WF-Instanz nach dem bisher vorgestellten Compliance-Kriterium (Definition 3.1) ausgeschlossen, obwohl strukturell und semantisch keine Probleme auftreten würden. Diese Einschränkung ist zum einen unnötig, zum anderen ist sie für viele WF-Anwendungen schlichtweg nicht akzeptabel. Ein Therapieprozess eines Patienten kann beispielsweise mehrere identische Behandlungszyklen umfassen, zwischen denen größere Zeitabstände liegen können. Schemaänderungen müssen hier aus medizinischen und juristischen Gründen auf aktuelle oder zukünftige Behandlungszyklen anwendbar sein, selbst wenn frühere Behandlungszyklen nach dem alten WF-Schema S ausgeführt wurden [49].

Zusammenfassend lassen sich also folgende Probleme bei Anwendung des in Definition 3.1 vorgestellten Compliance-Kriteriums feststellen:

- Die Überprüfung auf Compliance ist bei Einbeziehung der vollständigen Ablaufhistorie ineffizient.
- Die bislang verfolgten Ansätze zur Überprüfung der Compliance sind teilweise zu restriktiv.

Im Folgenden wird gezeigt, wie diese Probleme durch ein geeignetes WF-Ausführungsmodell sowie durch die Definition einfach überprüfbarer Vorbedingungen für Änderungsoperationen und einer ,,Aufweichung" des Compliance-Kriteriums effizient zu lösen sind. Im Anschluss daran werden wir in Kapitel 4 dann zeigen, wie für verträgliche Instanzen die notwendigen Zustandsanpassungen bei ihrer Migration automatisch vorgenommen werden können. Wir beziehen uns dabei zwar auf das in Kapitel 2 vorgestellte WF-Metamodell, die angestellten Überlegungen

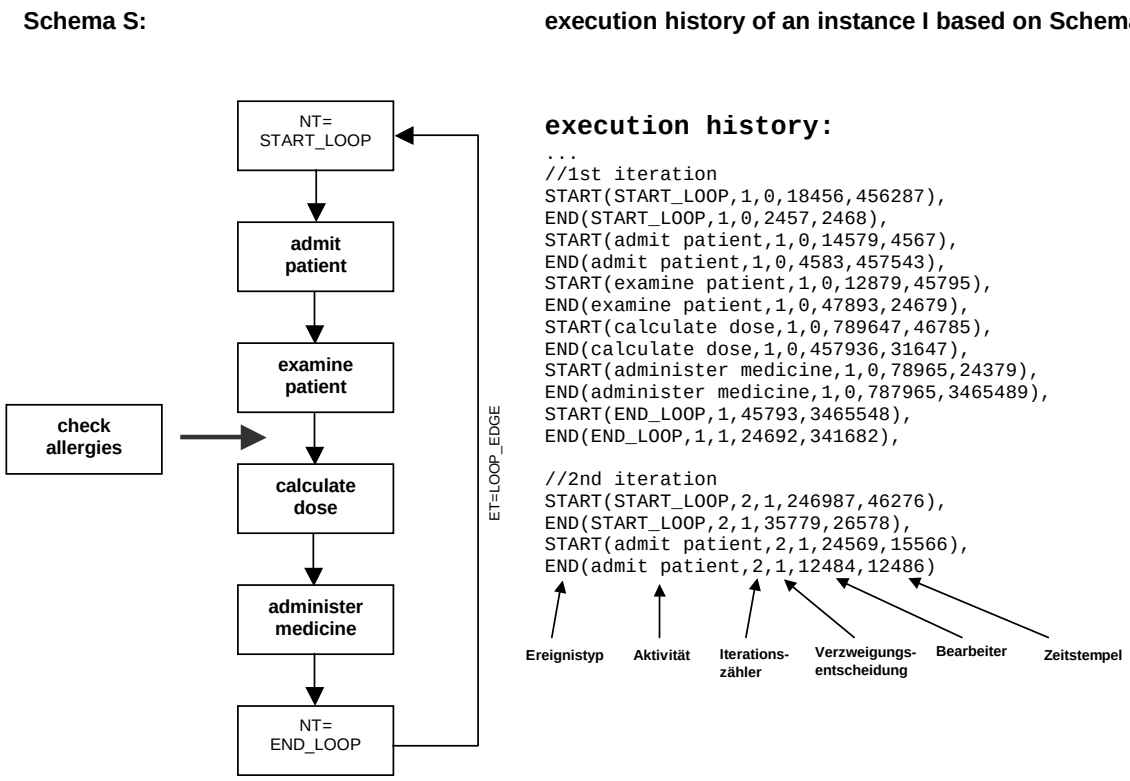


Abbildung 6: Ausschnitt aus einem Chemotherapie-Prozess (siehe [49]) mit Ablaufhistorie (vereinfacht)

lassen sich aber auch auf anderen WF-Ausführungsmodelle übertragen, wenn dieses gewisse Eigenschaften (z. B. Zustandsmarkierungen für Aktivitätenknoten) aufweisen.

3.3 Effiziente Prüfung auf Compliance (Kontrollflusssicht)

Wir nutzen im Folgenden aus, dass für jede einzelne WF-Instanz neben ihrer Ablaufhistorie auch die aktuellen Zustandsmarkierungen ihrer Aktivitätenknoten gespeichert werden. Im Gegensatz zu Ausführungsmodellen mit reinem Tokenfluss (z. B. Petrinetze) bleiben (mit Ausnahme von Schleifen) auch die Markierungen durchlaufener Bereiche erhalten. Zusätzlich werden für Ablaufpfade, für die feststeht, dass sie nicht mehr durchlaufen werden (vgl. Abb. 11), die zugehörigen Knoten als `SKIPPED` und die zugehörigen Kanten als `FALSE_SIGNALED` markiert. Ein ähnlicher Ansatz wird z. B. von MQSeries Workflow [34, 38] und WASA [52] verfolgt, wo nicht mehr auszuführende Ablaufpfade über eine sogenannte Dead-Path-Elimination abgewählt werden [34]. Im Folgenden zeigen wir, dass man Compliance (im Sinne von Definition 3.1) in den meisten Fällen über einfache und effizient überprüfbare Statusbedingungen zusichern kann. Des weiteren nehmen wir eine Erweiterung des bisherigen Compliance-Kriteriums vor, um die im Zusammenhang

mit Schleifen beschriebenen Restriktionen aufzuheben.

Dazu sei ein korrektes WF-Schema S mit davon abgeleiteten, laufenden WF-Instanzen $I_1, \ldots, I_n$ gegeben. S werde nun durch eine Änderung Δ auf ein (wiederum korrektes) WF-Schema S' transformiert. Wir wollen zeigen, dass die Verträglichkeit einer WF-Instanz I_k mit S' in der Mehrzahl der Fälle genau dann zugesichert werden kann, wenn I_k vor der Migration bestimmte Zustandsmarkierungen aufweist. Dadurch garantieren wir die Konsistenz der betroffenen WF-Instanz auch nach Propagation der Schemaänderungen. Wir werden anhand von Beispielen sehen, dass solche Vorbedingungen im Gegensatz zur Erzeugbarkeit einer gegebenen Ablaufhistorie auf dem neuen Schema einfach und effizient überprüfbar sind. Deswegen wird die benötigte Information bei dem von uns implementierten WF-Prototyp im Hauptspeicher gehalten.

Um die Probleme, die bei der Migration von WF-Instanzen auf ein geändertes WF-Schema auftreten, besser vermitteln zu können, teilen wir WF-Graphen in verschiedene Klassen ein (siehe Abb. 7). Dies trägt nicht nur zu einem besseren Verständnis des Verträglichkeitsbegriffs, sondern auch zu einer besseren Abgrenzung gegenüber existierenden Ansätzen bei. Dabei beschränken wir uns zunächst auf Kontrollflussaspekte und klammern Datenflüsse zwischen Aktivitäten noch aus. Wir erweitern die betrachteten Graphklassen sukzessive, d. h. Klassen einer höheren Stufe subsumieren die untergeordneten Klassen. Zusätzlich nehmen wir eine Einteilung der Änderungsoperationen in additive, subtraktive und ordnungsverändernde Operationen vor, um präzise angeben zu können, welche Auswirkungen verschiedene Arten von Änderungen auf die Verträglichkeit von WF-Instanzen haben.

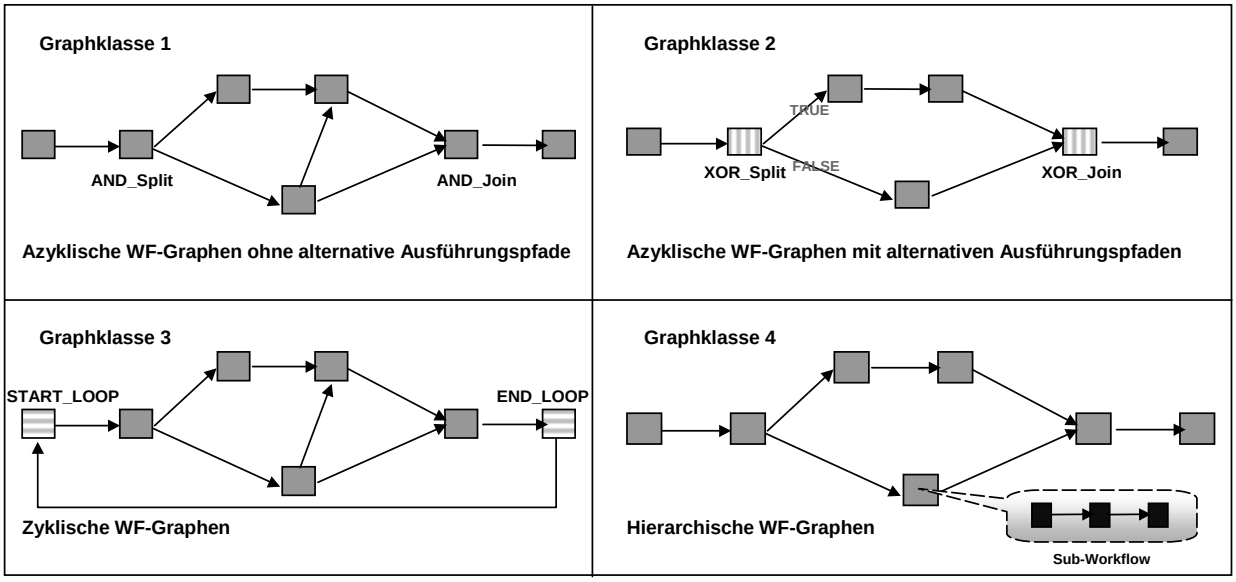


Abbildung 7: WF-Graphklassen

3.3.1 Azyklische WF-Graphen ohne alternative Ausführungspfade (Graphklasse 1)

Die einfachste, hier betrachtete Graphklasse umfasst azyklische, gerichtete WF-Graphen ohne alternative Ausführungspfade (siehe Abb. 8). Das sind WF-Graphen, deren Aktivitäten sequentiell und/oder parallel ausführbar sind. Alternative Verzweigungen im Ablauf werden hier also noch ausgeklammert, ebenso wie Schleifen. Beispiele für solche Graphen sind einfache Netzpläne, die in MQSeries Workflow [38] verwendeten Aktivitätennetze (ohne Transitionsbedingungen für Kontrollkonnektoren) und ADEPT WF-Graphen (ohne XOR-Verzweigungen und Schleifen) [44]. Für eine entsprechende WF-Instanz wird eine bestimmte Aktivität trivialerweise genau dann ausführbar, wenn alle ihre Vorgängerknoten beendet worden sind. Verklemmungen können auf Instanzebene aufgrund der azyklischen Graphstruktur nicht auftreten. Ein Beispiel zeigt Abb. 8.

Für die verschiedenen Arten von Änderungen (additiv, subtraktiv, ordnungsverändernd) wird nun gezeigt, dass die Verträglichkeit einer WF-Instanz I_k mit dem geänderten WF-Schema S' in der Mehrzahl der Fälle bei Einhaltung bestimmter Vorbedingungen hinsichtlich der aktuellen Markierung von I_k zugesichert werden kann. Dabei wird vorausgesetzt, dass das aus der Änderung hervorgegangene WF-Schema S' korrekt ist, insbesondere muss S' wieder ein Vertreter der Graphklasse 1 sein.

3.3.1.1 Additive Änderungsoperationen für Graphen der Graphklasse 1

Wir untersuchen nun, welche Zustandsinformationen bei WF-Instanzen der Graphklasse 1 beim Hinzufügen von Aktivitätenknoten und Kontrollkanten (additive Änderung) notwendig sind, um Aussagen zur Verträglichkeit mit dem neuen WF-Schema machen zu können.

1. Hinzufügen von Aktivitätenknoten

Eine Knoteneinfügeoperation erweitert die Knotenmenge N eines Schemas S = (N, E, ...) um eine Aktivität. Zusätzlich werden Kontrollabhängigkeiten eingefügt, um diese Aktivität in den WF-Kontext einzubetten. Die Migration einer WF-Instanz I auf ein modifiziertes WF-Schema S' ist genau dann möglich, wenn für die Einfügeoperation gewisse Vorbedingungen hinsichtlich des Status von I zugesichert werden können (vgl. Satz 3.1):

Satz 3.1 (Einfügen von Aktivitätenknoten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde nun durch Einfügen eines Aktivitätenknotens n_{insert} (inklusive normaler Kontrollkanten zwecks Einbindung in den WF-Kontext) in ein wiederum korrektes WF-Schema S' der Graphklasse 1 transformiert. Dann gilt:*

I ist verträglich mit S' $\Leftrightarrow$

$\forall n \in succ(S', n_{insert}) : NS(n) \in \{\texttt{NOT\_ACTIVATED}, \texttt{ACTIVATED}\}$

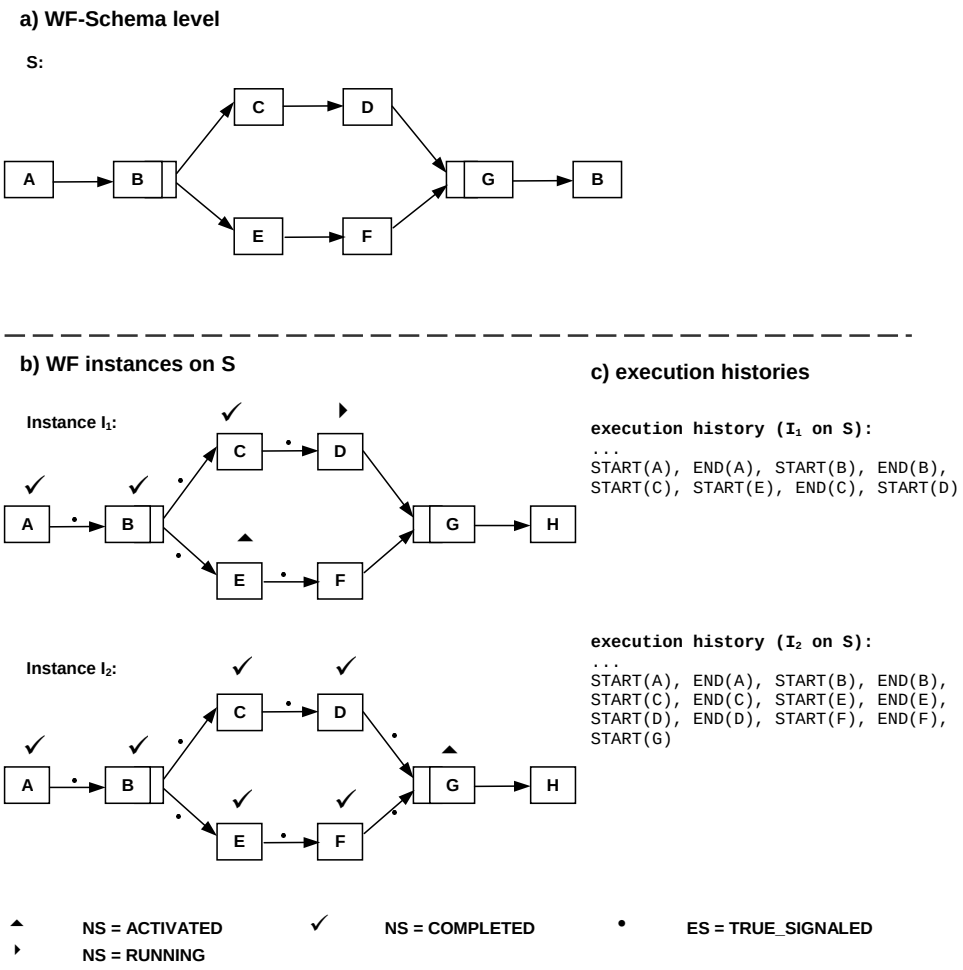


Abbildung 8: WF-Graph der Graphklasse 1

Der Beweis zu Satz 3.1 befindet sich, ebenso wie die Beweise nachfolgender Sätze, in Anhang B.

Satz 3.1 besagt, dass die gewünschte Compliance für Einfügeoperationen genau dann zugesichert werden kann, wenn sich die (direkten) Nachfolger des neu eingefügten Schritts aktuell in einem der beiden Zustände `ACTIVATED` oder `NOT_ACTIVATED` befinden. Knoten im Zustand `NOT_ACTIVATED` (vgl. Kapitel 2) sind generell unproblematisch. Bereits aktivierte Knoten werden zwar schon in den Benutzer-Arbeitslisten zur Auswahl angeboten, sind aber noch nicht gestartet worden. Insbesondere haben sie noch keine Einträge in die Ablaufhistorie der WF-Instanz geschrieben, so dass sie bei Bedarf problemlos wieder aus den Arbeitslisten entfernt werden können. Wird beispielsweise ein neuer Aktivitätenknoten X direkt vor E in den Ausführungsgraph von Instanz I_1 in Abb. 8b eingefügt, so werden begleitend zu dieser Änderung alle Arbeitslisteneinträge von E entfernt. Anschließend werden die entsprechenden Arbeitslisteneinträge zu Aktivität

X erzeugt, da der betreffende Aktivitätenknoten nach seiner Einfügung sofort ausführbar ist. Insgesamt kann also durch einfache Statusüberprüfungen von Knoten festgestellt werden, ob eine WF-Instanz I im Anschluss an eine additive Änderung der Knotenmenge N mit dem neuen WF-Schema verträglich ist oder nicht.

Für verträgliche WF-Instanzen, die auf das neue WF-Schema migriert werden, kann für die vorliegende Graphklasse zudem zugesichert werden, dass ein neu eingefügter Knoten in der Folge auch zur Ausführung kommt. Aufgrund der azyklischen Graphstruktur von S' kann es zu keinen Verklemmeungen kommen, so dass der eingefügte Knoten aktiviert wird, sobald alle seine Vorgänger beendet worden sind.

2. Hinzufügen von Kontrollabhängigkeiten

In bestimmten Fällen möchte der Modellierer eine Serialisierung bisher parallel angeordneter Aktivitäten vornehmen. Dazu ist das Einfügen weiterer Kontroll-Kanten notwendig.[3] In der Mehrzahl der Fälle lässt sich hier die Verträglichkeit einer WF-Instanz mit dem neuen WF-Schema wiederum über einfache Statusüberprüfungen feststellen. Allerdings kann es in Einzelfällen auch vorkommen, dass hierfür Informationen aus der Ausführungshistorie benötigt werden, etwa wenn für die neu einzufügende Kante sowohl der Quell- als auch der Zielknoten bereits beendet worden sind. Auch für solche Fälle lässt sich für eine gegebene WF-Instanz unter gewissen Voraussetzungen Compliance mit dem neuen Schema zusichern.

Satz 3.2 (Einfügen von Kontrollabhängigkeiten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch Hinzufügen einer Kontrollabhängigkeit $n_{src} \rightarrow n_{dest}$ ($n_{src}, n_{dest} \in N$) auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert . Dann gilt:*

I ist verträglich mit S' $\Leftrightarrow$

$NS(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED}\}$

$\vee$ $(NS(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge NS(n_{src}) = \texttt{COMPLETED}$

$\wedge$ $((\exists\ e_i = END(n_{src}),\ e_j = START(n_{dest})) \in \mathcal{A}) \wedge i < j))$

In jedem Fall kann man bei Einfügung einer Kontrollkante Compliance zusichern, wenn der Zielknoten der hinzukommenden Kontrollkante noch nicht gestartet worden ist. Offensichtlich können Aussagen zur Verträglichkeit einer WF-Instanz mit dem neuen Schema aber nicht immer nur auf Grundlage solcher Zustandsprüfungen gemacht werden, sondern es muss hierzu unter Umständen doch die Ablaufhistorie (bzw. Ausschnitte davon) herangezogen werden. Eine Kontrollkante kann beispielsweise auch dann noch in den Ausführungsgraphen einer WF-Instanz eingefügt werden, wenn ihr Quell- und Zielknoten bereits beendet sind und diese ihre Ablaufhistorieneinträge in der Reihenfolge der neuen Kontrollabhängigkeit geschrieben haben. In Abb.

[3] In ADEPT liegt durch Verwendung der Blockstruktur eindeutig fest, welche Zweige parallel zueinander angeordnet sind. Für die Beschreibung von ,,wartet-auf"-Beziehungen zwischen parallelen Knoten gibt es Sync-Kanten [42] (vgl. Abschnitt 3.3.2).

8b beispielsweise kann die Kontrollabhängigkeit $E \rightarrow D$ nicht in die Instanz I_1 eingefügt werden, da sich der Zielknoten D bereits im Zustand `RUNNING` befindet, der Knoten E jedoch noch nicht beendet wurde. Dies ist auch aus der Ablaufhistorie der Instanz I_1 (vgl. Abb. 8c) ersichtlich: D hat seinen Starteintrag in die Ablaufhistorie geschrieben, bevor irgendein Eintrag zu E existiert. Folglich lässt sich die entsprechenden Ablaufhistorie auf dem geänderten Schema nicht mehr erzeugen. Eine Einfügung der Kontrollabhängigkeit $E \rightarrow D$ in Instanz 2 ist wiederum möglich. Zwar sind hier sowohl Quell- als auch Zielknoten der Kante bereits im Status `COMPLETED`, jedoch haben D und E ihre Historieneinträge in der ,,richtigen" Reihenfolge in die Ablaufhistorie geschrieben (siehe Abb. 8c). Bei der Implementierung von Verträglichkeitsprüfungen im Zusammenhang mit dem Einfügen von Kontrollkanten sollte deshalb eine mehrstufige Vorgehensweise gewählt werden, bei der in der ersten Stufe noch keine Zugriffe auf die Ablaufhistorie erfolgen.

3.3.1.2 Subtraktive Änderungsoperationen für Graphen der Graphklasse 1

Wir betrachten nun subtraktive Änderungen, bei denen Aktivitätenknoten bzw. Kontrollabhängigkeiten aus dem WF-Schema gelöscht werden. Im Gegensatz zu additiven Änderungen können hier die Verträglichkeitsprüfungen vollständig auf Grundlage von Zustandsmarkierungen durchgeführt werden.

1. Löschen von Aktivitätenknoten

Zunächst betrachten wir das Löschen von Aktivitäten aus dem WF-Schema. Dabei werden auch zugehörige Kontextkanten gelöscht, die Reihenfolgebeziehungen zwischen den im Schema S' verbleibenden Knoten bleibt aber erhalten. Die Vorbedingungen für die Zusicherung der Verträglichkeit einer WF-Instanz I auf S mit dem geänderten WF-Schema S' sind in Satz 3-3 zusammengefasst:

Satz 3.3 (Löschen von Aktivitätenknoten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch Löschen Aktivitätenknoten n_{delete} ($n_{delete} \in N$) inklusive Kontextkanten unter Erhalt der Anordnungsbeziehung der anderen Aktivitäten auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert . Dann gilt:*

I ist verträglich mit S' $\Leftrightarrow$

$NS(n_{delete}) \in$ {`NOT_ACTIVATED, ACTIVATED`}

Intuitiv ist klar, dass man nur solche Knoten löschen darf, die in einem der beiden Zustände `ACTIVATED` oder `NOT_ACTIVATED` sind. Bei nicht aktivierten Knoten sind keine weiteren Anpassungen notwendig. Ansonsten müssen die entsprechenden Arbeitslisten-Einträge aus Benutzer-Arbeitslisten gelöscht werden, bevor mit der WF-Kontrolle fortgefahren werden kann. Löscht man beispielsweise Knoten E (mit Status `ACTIVATED`) aus Instanz I_1 in Abb. 8b, so kann die Ablaufhistorie aus Abb. 8c auch auf dem geänderten Schema erzeugt werden. Allerdings müssen

bei der Migration von I_1 auf das neue Schema die bisherigen Einträge von E zunächst einmal wieder aus den entsprechenden Arbeitslisten entfernt werden.

2. Löschen von Kontrollabhängigkeiten

In bestimmten Situationen kann auch das Löschen von Kontrollkanten erforderlich sein, etwa wenn eine Serialisierung zwischen zwei Aktivitäten wieder aufgehoben werden soll. Interessanterweise kann hier Verträglichkeit ohne weitere Überprüfung zugesichert werden (vgl. Satz 3.4). Der Grund dafür ist, dass die nun parallel ausführbaren Aktivitäten auf dem neuen Schema S' prinzipiell auch in ihrer bisherigen Reihenfolge ausführbar sind.

Satz 3.4 (Löschen von Kontrollabhängigkeiten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch Löschen einer Kontrollabhängigkeit $n_{src} \rightarrow n_{dest}$ ($n_{src}, n_{dest} \in N$) auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert . Dann gilt:*

I ist verträglich mit S'

3.3.1.3 Ordnungsverändernde Operationen für Graphen der Graphklasse 1

Wir betrachten nun ordnungsverändernde Operationen als eine spezielle, für die Praxis sehr wichtige Art von WF-Graph-Transformationen. Sie ändern die Anordnungsbeziehungen zwischen Aktivitätenknoten, ohne dass die Knotenmenge selbst modifiziert wird. Entsprechende Änderungen werden etwa notwendig, wenn die Bearbeitungsreihenfolge von Knoten vertauscht werden soll oder - allgemeiner - wenn ein Knoten von seiner aktuellen Position im WF-Graphen an eine andere Stelle verschoben werden soll. Prinzipiell kann man ordnungsverändernden Operationen auf eine Menge elementarer Kanteneinfüge- und Kantenlöschoperationen zurückführen. Bevor wir hierauf eingehen, betrachten wir zwei einfache Beispiele. Sie illustrieren, welche Anforderungen und Probleme im Zusammenhang mit ordnungsverändernden Operationen bestehen.

Beispiel 3.6 (Verschieben von Aktivitäten) Betrachten wir zunächst Abb. 9a. Das Ausgangsschema S umfasst sequentiell angeordnete Aktivitäten A, B, C, D und E, insbesondere folgt der Aktivitätenknoten D transitiv auf B, d. h. es gilt B $\in$ pred*(S, D). Im Folgenden wird eine parallele Anordnung dieser Knoten gewünscht, wie in Abb. 9b dargestellt. Dazu muss der Knoten D zunächst aus seinem Kontext im WF-Graphen herausgelöst werden (Entfernen von C $\rightarrow$ D und D $\rightarrow$ E). Anschließend wird er durch Einfügen der Kontrollkanten A $\rightarrow$ D, D $\rightarrow$ C und C $\rightarrow$ E parallel zum Knoten B angeordnet. Wir fassen die neu hinzugefügten Kontrollkanten in einer Menge E_{add} = {A $\rightarrow$ D, D $\rightarrow$ C, C $\rightarrow$ E} und die gelöschten Kontrollkanten in einer Menge E_{delete} = {C $\rightarrow$ D, D $\rightarrow$ E} zusammen.

Beispiel 3.7 (Vertauschen der Reihenfolge zweier Aktivitäten) Die Reihenfolge der Aktivitätenknoten B und C aus der Sequenz A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D in Abb. 10a soll vertauscht werden, wie in Abb. 10b dargestellt. Dazu müssen zunächst beide Knoten B und C aus ihrem bisherigen Kontext im Graphen herausgelöst werden, was durch Löschen der Kontrollkanten A $\rightarrow$ B, B $\rightarrow$ C und C $\rightarrow$ D bewerkstelligt wird (E_{delete} = {A $\rightarrow$ B, B $\rightarrow$, C $\rightarrow$ D}). Anschließend

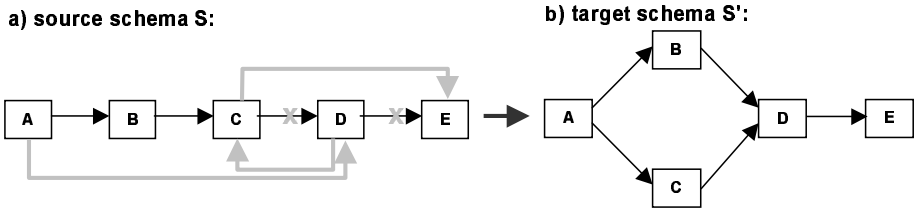


Abbildung 9: Verschieben von Aktivitäten

werden die Kontrollkanten A → C, B → D und C → B eingefügt (es ergibt sich E_{add} = {A → C, B → D, C → B}).

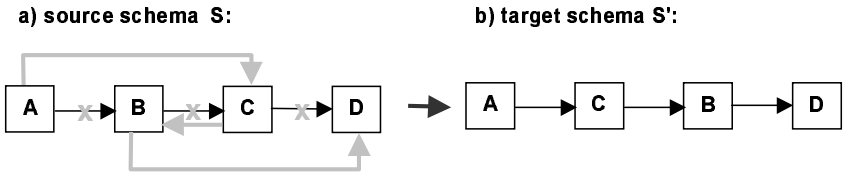


Abbildung 10: Vertauschen der Reihenfolge zweier Aktivitäten

Bei genauerer Betrachtung der Beispiele 3.6 und 3.7 lässt sich eine Kategorisierung der Kantenänderungsoperationen bzgl. ihrer Auswirkungen auf die Verträglichkeit einer WF-Instanz mit dem neuen Schema ableiten. Kantenlöschoperationen sind, wie bereits in Satz 3.4 gezeigt, unkritisch, da bestehende Reihenfolgebeziehungen zwischen Aktivitätenknoten aufgelöst werden. Beispielsweise besteht nach dem Entfernen der Kontrollkanten C → D und D → E in Abb. 9 keine Reihenfolgebeziehung mehr zwischen diesen Knoten.

Kanteneinfügeoperationen haben keine Auswirkung auf die Verträglichkeit von WF-Instanzen mit dem neuen WF-Schema, wenn sie Reihenfolgebeziehungen zwischen Aktivitätenknoten festlegen, die schon im Quellschema S' bestanden oder (transitiv) auch im Zielschema S' weiterhin gelten. Ein Beispiel hierfür stellt das Einfügen der Kante C → E in Abb. 10a dar, da der Knoten C bereits für WF-Instanzen des Schemas S immer vor E ausgeführt wurde. Kritisch sind nur solche Kanteneinfügeoperationen, die neue Reihenfolgebeziehungen zwischen Aktivitätenknoten definieren (die also im Quellschema S noch nicht enthalten waren, jedoch für das Zielschema S' gelten). In Abb. 9a wird z. B. durch Einfügen der Kontrollkante D → C eine völlig neue Reihenfolgebeziehung festgelegt, was die Beachtung des Status von C erfordert.

Wir treffen nun eine allgemeingültige Aussage in Bezug auf die Verträglichkeit von WF-Instanzen bei ordnungsverändernden Operationen.

Satz 3.5 (Änderung der Anordnung von Aktivitätenknoten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S.*

S werde durch eine ordnungsverändernde Operation auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert. D. h. es wird - ausgehend von der für S definierten Kantenmenge E - eine Menge E_{add} von Kontrollkanten hinzugefügt und eine Menge E_{delete} von Kontrollkanten entfernt. Die Knotenmenge N selbst bleibt unverändert. Dann gilt:

I ist verträglich mit S' $\Leftrightarrow$

$\forall\ e = n_{src} \rightarrow n_{dest} \in E_{add}$*:*

$NS(n_{dest}) \in \{$`NOT_ACTIVATED, ACTIVATED`$\}$

$\vee\ (NS(n_{dest}) \in \{$`RUNNING, COMPLETED`$\} \wedge NS(n_{src}) =$ `COMPLETED`

$\wedge\ ((\exists\ e_i = END(n_{src}),\ e_j = START(n_{dest})) \in \mathcal{A}) \wedge i < j))$

Wie man leicht erkennen kann, ergeben sich die Voraussetzungen für die Verträglichkeit von I mit S' durch Aggregation der entsprechenden Bedingungen für die Anwendung einzelner Kanteneinfüge- und Kantenlöschoperationen. Die Begründung hierfür liefert der Beweis zu Satz 3.5 im Anhang B.1. Er nutzt aus, dass man für eine beliebige Menge von Kantenoperationen $op_1, \ldots, op_n$ stets eine Serialisierung $op_{i_1}, \ldots, op_{i_n}$ findet, so dass bei ihrer Anwendung auf ein korrektes WF-Schema S ein wiederum korrektes WF-Schema S' resultiert (S =: S_0 [$op_{i_1} > S_1 \ldots S_{n-1}$ [$op_{i_n} > S_n$:= S') und jedes ,,Zwischen-"Schema S_μ ebenfalls korrekt ist. Wir werden dies später auch zur Definition der Vorbedingungen für beliebige komplexe Änderungen nutzen.

Gegebenenfalls lässt sich die Menge der Knoten, für welche Zustandsprüfungen notwendig werden, noch reduzieren. Dazu betrachten wir die Menge $N_{dep} = \{n_{dest} \mid \exists\ n_{src} \rightarrow n_{dest} \in E_{add}\}$. Solange es je zwei Knoten $n_1, n_2 \in N_{dep}$ gibt mit $n_1 \in$ pred*(S, n_2), kann n_2 aus N_{dep} entfernt werden. (N_{dep} ist die Menge aller Knoten, für welche Zustandsüberprüfungen im Zuge ein ordnungsverändernden Operation notwendig sind.) Die Zustandsüberprüfungen müssen dann nur noch für die verbleibenden Knoten durchgeführt werden.

3.3.1.4 Komplexe Operationen für Graphen der Graphklasse 1

Um ein gegebenes WF-Schema in der gewünschten Form abändern zu können, muss der Modellierer gegebenfalls mehrere (elementare) Änderungen vornehmen, d. h. es müssen ausgehend von dem aktuellen WF-Schema gegebenfalls mehrere Aktivitäten und Kontrollkanten hinzugefügt bzw. entfernt werden. Für eine solche zusammenhängende Anwendung von Elementaränderungen ist zu fordern, dass bei ihrer Propagation auf laufende WF-Instanzen entweder alle Einzeloperationen Anwendung finden oder aber die Propagation nicht zulässig ist.

Für eine solche komplexe Änderung ist die Verträglichkeit einer WF-Instanz mit dem neuen WF-Schema genau dann gegeben, wenn für jede zur Anwendung kommende Elementaroperation die Statusvoraussetzungen gemäss der Sätze 3.1 bis 3.5 erfüllt sind. Die Voraussetzungen für die Verträglichkeit einer WF-Instanz I mit dem aus der Anwendung einer komplexen Änderung hervorgehenden WF-Schemas S' ergeben sich somit – ähnlich wie bei Satz 3.5 – durch Aggregation der Voraussetzungen für die einzelnen der zur Anwendung kommenden Einfüge- und

Löschoperationen. Formal:

Satz 3.6 (Komplexe Änderungen (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch die (serielle) Anwendung von Elementaroperationen $op_1, \ldots, op_n$ (Einfügen/Löschen von Aktivitätenknoten und Kontrollabhängigkeiten) in ein wiederum korrektes WF-Schema S' transformiert. Dann gilt:*

I verträglich mit S' ⇔

Für jede Elementaroperation op_i sind ihre Voraussetzungen für die (isolierte) Anwendung auf S (gemäss der Sätze 3.1 bis 3.5) erfüllt.

3.3.2 Azyklische WF-Graphen mit alternativen Ausführungspfaden (Graphklasse 2)

Wir erweitern nun die Graphklasse 1 um die Möglichkeit zur Definition alternativer Ausführungspfade (sogenannte XOR-Verzweigungen). Für WF-Graphen der hieraus resultierenden Graphklasse 2 lassen sich sequentielle, parallele und alternative Ausführungspfade modellieren (siehe WF-Graph in Abb. 11a). Im Unterschied zur Graphklasse 1 kann es also auch Pfade geben, für die sich im Verlauf der WF-Ausführung (z. B. abhängig von WF-relevanten Daten) ergibt, dass sie nicht mehr ausgeführt werden sollen. In ADEPT werden solche nicht mehr wählbaren Pfade auf Instanzebene, ähnlich wie in einigen anderen Ansätzen (z. B. MQSeries Workflow, $WASA_2$ [52]) als `SKIPPED` markiert. Im Gegensatz zu dem meisten Petri-Netz-basierten Ansätzen (Tokenfluss mit TRUE-Semantik) sprechen wir in diesem Zusammenhang auch von einem Ausführungsmodell mit TRUE/FALSE-Semantik. Beispielsweise wurde bei der Ausführung von WF-Instanz 1 in Abb. 11b der Pfad mit Selektions-Code `sc1` gewählt[4] (siehe auch entsprechender Eintrag in der Ablaufhistorie in Abb. 11c). Damit wurden alle anderen Pfade der XOR-Verzweigung abgewählt, d. h. die Aktivitätenknoten D, E, F und G als `SKIPPED` markiert.

Im ADEPT-Ansatz ist jedem XOR-Split-Knoten ein eindeutiger XOR-Join-Knoten zugeordnet (und umgekehrt), so dass jede Aufsplittung des Kontrollflusses wieder eindeutig zusammengeführt wird. Demzufolge kann ein normaler Aktivitätenknoten wie bisher genau dann ausgeführt werden, wenn alle Vorgänger beendet worden sind. Zur Ausführung eines XOR-Join-Knotens kommt es, wenn einer der Vorgängerknoten beendet ist und die anderen Vorgängerknoten als `SKIPPED` markiert sind. Ein Beispiel hierzu liefert Instanz 2 aus Abb. 11b. Zu erwähnen bleibt, dass Aktivitätenknoten, die als `SKIPPED` markiert werden, keine Einträge in die Ablaufhistorie schreiben, wie durch die beiden Ablaufhistorien in Abb. 11c illustriert wird. Wir konzentrieren uns im Folgenden auf das oben beschriebene Modell. Die Überlegungen sind aber auch auf

[4] In ADEPT kann ein solcher Selektions-Code von einem Aktivitätenprogramm geliefert werden, er kann aber auch das Ergebnis einer Prädikatevaluation sein (entsprechende Prädikate sind dann dem XOR-Splitknoten zugeordnet).

andere Ansätze mit TRUE/FALSE-Markierungen übertragbar.

Im Zusammenhang mit der Unterstützung von XOR-Verzweigungen ermöglicht ADEPT dem Modellierer noch verschiedene andere Arten von Änderungen. Die angebotenen Änderungsoperationen sind vollständig, erlauben also die Transformation eines gegebenen WF-Graphen der Klasse 2 in einen beliebigen anderen WF-Graphen derselben Klasse. Dies beinhaltet zunächst alle Modifikationen, die auch auf Graphklasse 1 anwendbar sind. Zusätzlich werden spezielle Änderungsoperationen für das Hinzufügen und Löschen von XOR-Verzweigungen (inklusive einzelner XOR-Zweige) zur Verfügung gestellt. Im Folgenden diskutieren wir, wie sich die Unterstützung von XOR-Verzweigungen auf die Verträglichkeit von WF-Instanzen bei Schemaänderungen auswirkt.

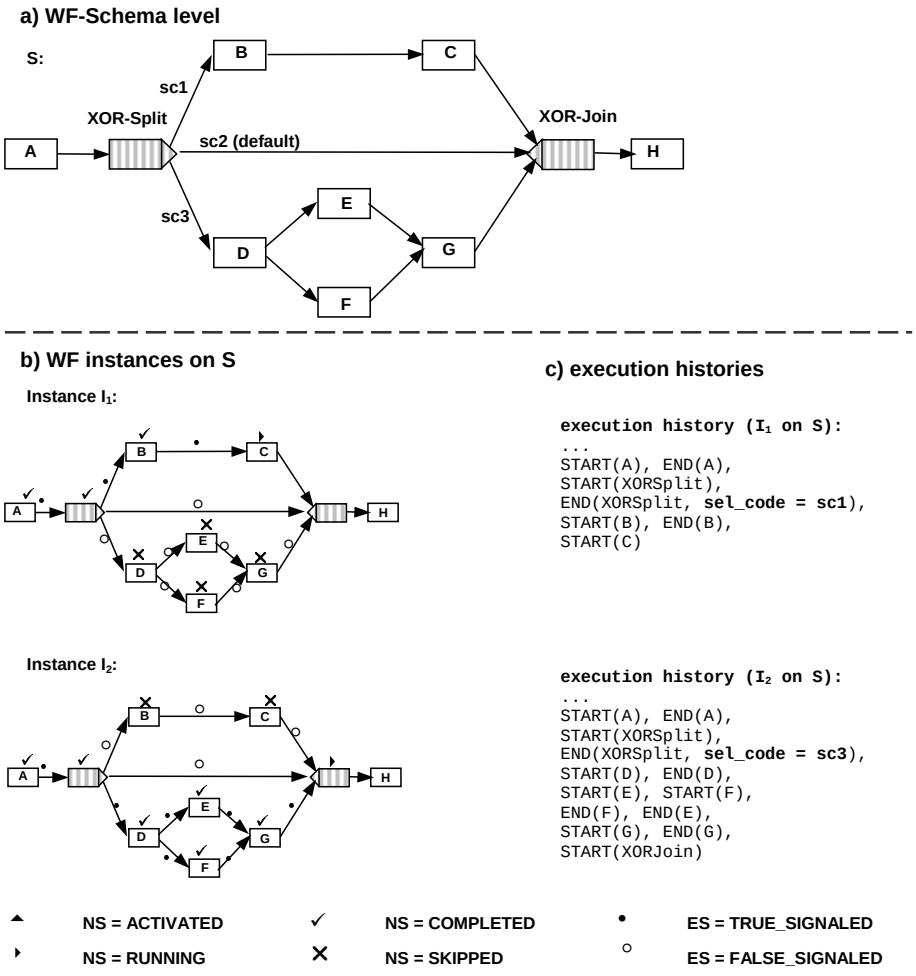


Abbildung 11: WF-Graph der Graphklasse 2

3.3.2.1 Additive Änderungsoperationen für Graphen der Graphklasse 2

Gegeben sei ein WF-Schema der Graphklasse 2 und eine WF-Instanz I auf S. S werde entweder durch Einfügen einer Aktivität oder durch Hinzunahme einer Kontrollkante bzw. Sync-Kante (zwischen parallelen Aktivitäten) in das WF-Schema S' (der Graphklasse 2) verändert. Wir untersuchen nun, wie sich effizient prüfen lässt, ob I verträglich mit S' ist. Im Unterschied zu Abschnitt 3.3.1.3 können hier nun auch Aktivitäten im Zustand `SKIPPED` von den Änderungen betroffen sein.

Satz 3.7 (Einfügen von Aktivitätenknoten/Sync-Kanten (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine additive Änderungsoperation op in ein korrektes WF-Schema S' der Graphklasse 2 transformiert.*

(a) op fügt einen neuen Knoten n_{insert} (inklusive normaler Kontrollkanten zwecks Einbindung in den WF-Kontext) in S ein. Dann gilt:

I ist verträglich mit S' ⇔

[∀ n ∈ c_succ(S', n_{insert})[5] : NS(n) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}] ∨

[n_{insert} wird in einen abgewählten Teilzweig einer XOR-Verzweigung eingefügt]

(b) op fügt eine Kontrollkante $n_{src} \rightarrow n_{dest}$ (zwischen zwei nicht parallelen Knoten) in S ein. Dann gilt:

I verträglich mit S' ⇔

NS(n_{dest}) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} ∨

[NS(n_{src}) = `COMPLETED` ∧ NS(n_{dest}) ∈ {`RUNNING, COMPLETED`}) mit

(∃e_i = `END`(n_{src}), e_j = `START`(n_{dest}) ∈ $\mathcal{A}$ ∧ $i < j$)]

(c) op fügt eine Sync-Kante $n_{src} \rightarrow n_{dest}$ ($n_{src}, n_{dest} \in N$) zwischen zwei parallel zueinander liegenden Knoten in S ein. Dann gilt:

I ist verträglich mit S' ⇔

NS(n_{dest}) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} ∨

(NS(n_{src}) = `COMPLETED` ∧ NS(n_{dest}) ∈ {`RUNNING, COMPLETED`}) mit

(∃e_i = `END`(n_{src}), e_j = `START`(n_{dest}) ∈ $\mathcal{A}$ ∧ $i < j$)) ∨

(NS(n_{src}) = `SKIPPED` ∧ NS(n_{dest}) ∈ {`RUNNING, COMPLETED`}) mit

∀n ∈ $N_{critical}$ mit NS(n) ≠ `SKIPPED`: ∃e_i = `START`(n_{dest}), e_j = `END(n)` ∈ $\mathcal{A}$ mit $j < i$),

[5] Hier werden nur Nachfolger einbezogen, die über normale Kontrollkanten erreichbar sind (siehe Beschreibung Tabelle 4 in Anhang A

wobei $N_{critical} = (c\_pred^*(n_{src}) \neg c\_pred^*(n_{dest}))$

Die Feststellung, ob ein neu eingefügter Aktivitätenknoten n_{insert} in einen abgewählten Teilzweig fällt oder nicht, kann über einfache Statusüberprüfungen erfolgen. Dazu genügt es für die betreffende Instanz festzustellen, ob ein direkter Vorgänger oder Nachfolger (oder beide) von n_{insert} im Quell-Schema S aktuell die Markierung `SKIPPED` besitzt oder ob n_{insert} in einen leeren Teilzweig, dessen Kante als `FALSE_SIGNALED` markiert wurde, eingefügt wird. (Eine formale Darstellung hierzu findet sich im Beweis zu Satz 3.7 in Anhang B.2). Für Aktivitäten, die als `SKIPPED` markiert sind, werden in der Ablaufhistorie keine Einträge protokolliert. Damit hat das Einfügen von Knoten in abgewählte Teilzweige keinen Effekt auf die Ablaufhistorie und die Verträglichkeit einer WF-Instanz mit dem neuen WF-Schema. Beispielsweise kann für die WF-Instanz I_2 aus Abb. 11b ein neuer Aktivitätenknoten X ohne Probleme vor dem Knoten C in den Ausführungsgraphen eingefügt werden. Der Status von X wird dann ebenfalls als `SKIPPED` bewertet, so dass sich die Ausführungshistorie in Abb. 11c auch auf dem geänderten Schema erzeugen lässt.

Das Einfügen einer Sync-Kante $n_{src} \rightarrow n_{dest}$ ist unkritisch, solange der Zielknoten n_{dest} einen der Zustände `NOT_ACTIVATED, ACTIVATED` oder `SKIPPED` besitzt. Grund ist, dass n_{dest} in diesem Fall (noch) keinen Eintrag in $\mathcal{A}$ geschrieben hat. Andernfalls gilt NS(n_{dest}) $\in$ {`RUNNING, COMPLETED`} und die Verträglichkeit von I mit dem neuen WF-Schema ist nicht immer gewährleistet. Dies ist nur dann der Fall, wenn n_{src} bereits beendet ist und seinen Endeintrag in die Ablaufhistorie vor den Starteintrag von n_{dest} geschrieben hat (vgl. Satz 3.2). Falls dagegen NS(n_{src}) = `SKIPPED` gilt, ist I nur dann verträglich mit S', wenn für alle Vorgängerknoten von n_{src} mit Status ungleich `SKIPPED` gilt, dass sie ihre Eindeinträge in $\mathcal{A}$ vor den Starteintrag von n_{dest} geschrieben haben. (Zur Illustration dient Abb. 23 in Anhang B.2).

3.3.2.2 Subtraktive Änderungsoperationen für Graphen der Graphklasse 2

Da eine als `SKIPPED` markierte Aktivität keinen Historieneintrag geschrieben hat, hat das Löschen entsprechender Knoten keine Auswirkungen auf die Verträglichkeit der vorliegenden WF-Instanz mit dem resultierenden WF-Schema. Beispielsweise hat in Abb. 11b der als `SKIPPED` markierte Aktivitätenknoten E der Instanz I_1 keinen Eintrag in die Ablaufhistorie geschrieben (vergleiche Abb. 11c). Damit kann diese Historie auch auf dem WF-Schema erzeugt werden, das sich nach Löschen von E ergibt. Beim Löschen von Kontrollkanten werden Reihenfolgebeziehungen zwischen Aktivitätenknoten aufgehoben, so dass keine Bedingungen an den Status der betroffenen Aktivitätenknoten gestellt werden müssen (vgl. Satz 3.4).

Satz 3.8 (Löschen von Aktivitätenknoten und Kanten (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine subtraktive Änderungsoperation op in ein korrektes WF-Schema S' (der Graphklasse 2) transformiert.*

(a) op löscht einen Aktivitätenknoten n_{delete} *(*$n_{delete} \in N$*) inklusive Kontextkanten unter Erhalt der Anordnungsbeziehung der anderen Aktivitäten aus S. Dann gilt:*

I ist verträglich mit S' ⇔

NS(n_{delete}) ∈ {NOT_ACTIVATED, ACTIVATED, SKIPPED}

(b) op löscht eine Kontroll- oder Sync-Kante $n_{src} \rightarrow n_{dest}$ aus S ($n_{src}, n_{dest} \in N$). Dann gilt:

I ist verträglich mit S'

3.3.2.3 Einfügen, Löschen und Ändern von XOR-Verzweigungsblöcken

Um Vollständigkeit zu gewährleisten, muss es auch möglich sein, XOR-Verzweigungsblöcke in ein WF-Schema neu einzufügen bzw. sie daraus zu entfernen. Die nachfolgenden Aussagen gelten unabhängig davon, ob die jeweiligen Verzweigungsblöcke bereits Teilzweige enthalten oder nicht, da für die Feststellung der Verträglichkeit einer WF-Instanz die Betrachtung des XOR-Splitknotens bzw. XOR-Joinknotens ausreichend ist.

Satz 3.9 (Einfügen/Löschen von XOR-Verzweigungsblöcken (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine Änderungsoperation op in ein korrektes WF-Schema S' (der Graphklasse 2) transformiert.*

(a) op fügt einen Verzweigungsblock mit XOR-Splitknoten n_{split} und XOR-Joinknoten n_{join} (inklusive normaler Kontrollkanten zwecks Einbindung in den WF-Kontext) in S ein. Dann gilt:

I verträglich mit S' ⇔

[∀ n ∈ succ(S', n_{join}): NS(n) ∈ {NOT_ACTIVATED, ACTIVATED, SKIPPED}*] ∨*

[XOR-Verzweigungsblock (n_{split}, n_{join}) wird in abgewählten Teilzweig eingefügt]

(b) op lösche einen Verzweigungsblock mit XOR-Split n_{split} und XOR-Join n_{join} in S. Dann gilt:

I verträglich mit S' ⇔

NS(n_{split}) ∈ {NOT_ACTIVATED, ACTIVATED, SKIPPED} [6]

Ähnlich wie beim Einfügen eines Aktivitätenknotens muss beim Einfügen eines XOR-Verzweigungsblocks (n_{split}, n_{join}) gewährleistet sein, dass alle direkten Nachfolger des Joinknotens n_{join} sich in einem der Zustände {NOT_ACTIVATED, ACTIVATED, SKIPPED} befinden. Dadurch ist sichergestellt, dass keine Historieneinträge für diese Knoten in die Ablaufhistorie geschrieben wurden. Damit ein XOR-Verzweigungsblock (n_{split}, n_{join}) gelöscht werden kann, muss für dessen Aktivitätenknoten – sie sind eindeutig festgelegt durch die Knotenmenge $\{n_{split}, n_{join}\}$

[6] Diese Bedingung kann evtl. dadurch aufgelockert werden, dass bei bereits beendeten Split-Knoten für den ersten Knoten des gewählten Teilzweigs gefordert wird, dass dieser Knoten noch nicht gestartet bzw. beendet wurde.

$\cup$ succ$^*(n_{split}) \cap$ pred$^*(n_{join})$ – gelten, dass sie noch keine Einträge in die Ablaufhistorie geschrieben haben. Dies ist genau dann gegeben, wenn sich der XOR-Splitknoten n_{split} in einem der Zustände {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} befindet.

Satz 3.10 fasst die Voraussetzungen für die Migration von WF-Instanzen auf ein WF-Schema, das sich durch Einfügen oder Löschen eines Teilzweigs in bzw. aus einem XOR-Verzweigungsblock ergeben hat, zusammen.

Satz 3.10 (Einfügen und Löschen von Teilzweigen (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine Änderungsoperation op auf ein korrektes WF-Schema S' der Graphklasse 2 transformiert.*

(a) op fügt einen neuen (gegebenenfalls leeren) Teilzweig in einen XOR-Verzweigungsblock (mit XOR-Splitknoten n_{split} und XOR-Joinknoten n_{join}) in S ein. Dann gilt:

I verträglich mit S'

(b) op löscht einen (gegebenenfalls leeren) Teilzweig aus einer XOR-Verzweigung (mit XOR-Splitknoten n_{split} und XOR-Joinknoten n_{join}) in S. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

[NS(n_{split}) $\neq$ `COMPLETED`*]* $\vee$ *[(NS(n_{split})* $=$ `COMPLETED` $\wedge$ *zu löschender Teilzweig abgewählt)]*[7]

Wenn der Status eines XOR-Splitknotens n_{split} noch nicht `COMPLETED` ist, wurde entweder noch keine Entscheidung bzgl. der Auswahl eines Teilzweigs getroffen oder der betrachtete Verzweigungsblock ist seinerseits ein abgewählter Teilzweig eines XOR-Verzweigungsblocks (n_{split} = `SKIPPED`). Dann kann jeder der Teilzweige gelöscht werden, ohne die Verträglichkeit einer Instanz mit dem geänderten Schema zu verletzen. Ist der XOR-Splitknoten beendet und ein Zweig als `SKIPPED` markiert, kann dieser Zweig ebenfalls ohne Probleme gelöscht werden, da seine Knoten keine Einträge in die Ablaufhistorie geschrieben haben. Ein Beispiel hierfür wäre in Abb. 11b das Löschen des Teilzweigs mit Selektionscode `sc3` für die Instanz I_1. Sollen Knoten eines hinzugefügten Verzweigungsblocks mit Knoten paralleler Zweige synchronisiert werden, müssen allerdings noch Sync-Kanten zugeordnet werden (siehe hierzu Satz 3.6b).

Der Vollständigkeit halber sei erwähnt, dass in ADEPT auf Schemaebene auch Attribute von Knoten verändert werden können, wie Ausführungsprioritäten oder Bearbeiterformeln [6, 41] . Dies ist immer unkritisch, wenn der betreffende Knoten einen der Zustände `NOT_ACTIVATED, ACTIVATED` oder `SKIPPED` besitzt. Auch Selektionscodes, die den ausgehenden Kanten eines XOR-Splitknotens zugeordnet sind, müssen bei Bedarf geändert werden können. Verträglichkeit von Instanzen mit S' ist für diesen Fall gegeben, wenn der XOR-Splitknoten entweder noch nicht beendet worden ist oder von der Änderung ein abgewählter Teilzweig betroffen ist.

[7] d. h. alle Knoten dieses Zweiges besitzen den Zustand `SKIPPED`

3.3.2.4 Ordnungsverändernde und komplexe Operationen für Graphklasse 2

Was komplexe, durch mehrere Elementaroperationen beschriebene Schemaänderungen anbetrifft, gelten die selben Aussagen wie für WF-Graphen der Graphklasse 1. Eine WF-Instanz, deren WF-Schema durch Anwendung von Operationen $op_1, \ldots, op_n$ modifiziert wird, ist genau dann verträglich mit dem neuen WF-Schema, wenn für jede einzelne Operation die entsprechenden Verträglichkeitsvoraussetzungen (gemäß der Sätze 3.7 und 3.8) erfüllt sind. Wir verzichten an dieser Stelle auf eine formale Darstellung. Trivialerweise gilt dies dann auch für ordnungsverändernde Operationen als Spezialfall von komplexen Änderungen. Werden die Anordnungsbeziehungen zwischen den Knoten eines WF-Schemas korrekt abgeändert, so ist eine WF-Instanz genau dann verträglich mit dem geänderten WF-Schema, wenn die Verträglichkeitsbedingung für jede einzelne Kanteneinfüge- bzw. Kantenlöschoperation (vgl. Satz 3.7) erfüllt ist.

3.3.3 Zyklische WF-Graphen mit Schleifen (Graphklasse 3)

Wie eingangs motiviert, ist die Unterstützung von Schleifen für prozessorientierte Anwendungen unerlässlich. ADEPT bietet hierfür ein explizites Schleifenkonstrukt an: Jede Schleife eines WF-Graphen bildet dabei einen Kontrollblock mit eindeutigen Start- und Endknoten, die über eine Schleifenrücksprungkante verknüpft sind. Schleifenblöcke können geschachtelt sein, dürfen sich aber nicht überlappen. Modellintern werden Schleifen durch spezielle Knotentypen (`START_LOOP`, `END_LOOP`) und eine speziellen Kantentyp (`LOOP_EDGE`) repräsentiert (vgl. Abb. 12). Wichtig ist dabei die Unterscheidbarkeit einer Schleifenkante von normalen Kontroll- und Sync-Kanten.

Die Verwendung eines expliziten Konstrukts für die Modellierung von Schleifen und die Unterscheidbarkeit zwischen dem ,,normalen" Fortschritt im Kontrollfluss und Schleifenrücksprüngen bieten zwei wichtige Vorteile. Zum einen ist es möglich, ,,gewünschte" Zyklen von ,,unerwünschten" Zyklen, deren Auftreten zur Laufzeit zu Verklemmungen führen kann, zu unterscheiden. (Dies ist z. B. bei komplexeren Petri-Netzen oftmals nicht aus der Netzstruktur ersichtlich, sondern kann nur durch aufwendige Analyse des dynamischen Netzverhaltens mit exponentieller Komplexität festgestellt werden.) Zum anderen ist zur Laufzeit einfach zu ermitteln, wann es zu einem Schleifenrücksprung kommt. Bei einem solchen Rücksprung werden vom Laufzeitsystem die Knoten- und Kantenmarkierungen des Schleifenkörpers wieder auf `NOT_ACTIVATED` bzw. `NOT_SIGNALED` zurückgesetzt. Gerade bei langlaufenden Prozessen wird dem Modellierer so ein mächtiges Konstrukt zur natürlichen Darstellung sich wiederholender Ablaufpfade an die Hand gegeben.

Im Folgenden erweitere Graphklasse 3 die zuvor vorgestellte Graphklasse 2 um die Möglichkeit zur Definition von Schleifen. Bezogen auf ADEPT lassen sich für WF-Graphen dieser Klasse wieder gewisse Aussagen zum statischen und dynamischen Verhalten treffen. Treten z. B. für ein WF-Schema der Graphklasse 3 bei ,,Ausblendung" von Schleifenrücksprungkanten keine Zyklen auf, kann es zur Laufzeit zu keinen Verklemmungen kommen, vorausgesetzt gewisse ,,Fairnessannahmen" (z. B. Terminierung von Schleifen) treffen zu.

Wir wenden uns nun der Frage zu, wann Änderungen eines WF-Schemas mit Schleifen auf

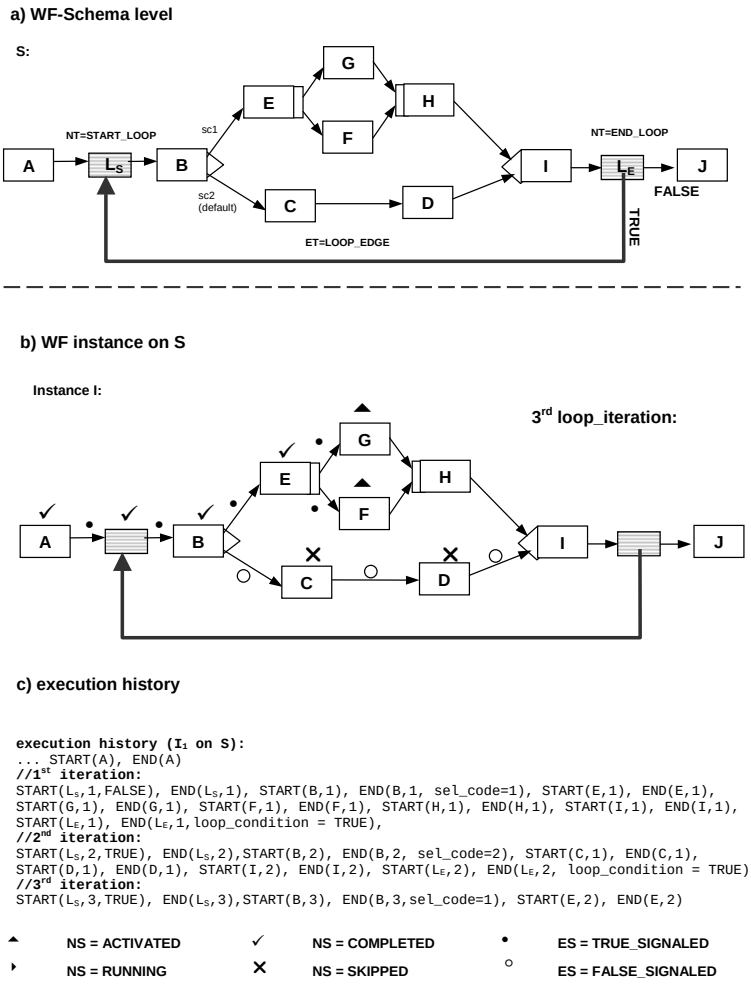


Abbildung 12: WF-Graph der Graphklasse 3

bereits laufende WF-Instanzen propagiert werden können. Das in Definition 3.1 eingeführte Compliance-Kriterium ist hierfür noch zu restriktiv, was folgendes Beispiel verdeutlicht:

Beispiel 3.8 (Änderungen auf zyklischen Graphen) Gegeben sei der in Abb. 12b dargestellte WF-Instanzgraph. Angenommen in der aktuellen, dritten Schleifeniteration soll ein Aktivitätenknoten X zwischen G und H eingefügt werden. Dann könnte die bisherige Ablaufhistorie der Instanz I_1 auf dem geänderten Schema nicht mehr erzeugt werden, weil in der ersten Schleifen-Iteration kein Eintrag zu X geschrieben worden ist. Anschaulich dargestellt würde die Ablaufhistorie von I_1 auf dem geänderten Schema in der ersten Iteration folgendermaßen aussehen:

$\mathcal{A}$ =< ... START(L_S,1,FALSE), END(L_S,1), START(B,1), END(B,1,sel_code=1), START(E), END(E), START(G), END(G), START(F), END(F), **START(X), END(X)**, START(H), END(H),...>

Die betrachtete WF-Instanz ist also bei Zugrundelegung von Definition 3.1 nicht verträglich mit dem geänderten WF-Schema.

Bei strikter Anwendung von Definition 3.1 können also auch Migrationen von Instanzen ausgeschlossen werden, für die bei Propagierung der Änderung keine inkonsistenten Zustände und damit keine unerwünschten Effekte zur Laufzeit resultieren würden. Dies schränkt den Benutzer jedoch unnötig ein, was den Wunsch nach einer ,,Auflockerung" des bisherigen Compliance-Kriteriums für WF-Schemata mit Schleifen aufwirft. Dabei müssen die bisherigen Anforderungen weiter beachtet werden, nämlich die Sicherstellung der Korrektheit und Konsistenz von Modell- und Instanzdaten nach Schemaänderungen sowie die effiziente Überprüfbarkeit der hierfür zugrundegelegten Regeln. Wir benötigen also für Schleifen ein leicht modifiziertes Verträglichkeitskriterium. Aus formaler Sicht bieten sich hier zwei Vorgehensweisen an:

1. Alle bisherigen Schleifeniterationen werden linearisiert, d.h. es erfolgt – logisch gesehen – eine Hintereinanderausführung aller Iterationen des Schleifenkörpers.
2. Für Schleifenausführungen werden nur diejenigen Historieneinträge berücksichtigt, die in der aktuellen bzw. letzten Schleifeniteration erzeugt wurden.

Den folgenden Betrachtungen legen wir Variante 2 zugrunde, da hierbei keine spezielle Behandlung geschachtelter Schleifen notwendig wird.

Definition 3.2 (Iterationsfreie Ablaufhistorie) *Sei I eine Instanz eines WF-Schemas S (der Graphklasse 3) mit Ablaufhistorie $\mathcal{A} =< e_0, \ldots, e_k >$. Die reduzierte Ablaufhistorie $\mathcal{A}_{red}$ von I entstehe durch Streichen von Einträgen aus $\mathcal{A}$ gemäß folgender Regeln:*

Gelte zunächst $\mathcal{A}_{red} := \mathcal{A}$. Für jede durch ihren Anfangs- und Endknoten (L_S, L_E) eindeutig definierte Schleife aus S werden nun alle zugehörigen Historieneinträge gestrichen, die nicht von der aktuellen bzw. letzten Schleifeniteration erzeugt wurden. Formal:

Falls $\exists e_x \in \mathcal{A}_{red}$ mit $e_x =$ `START(`L_S`,i)`, $i > 0 \wedge \nexists e_y \in \mathcal{A}_{red}$ mit $e_y =$ `START(`L_S`,i+1)`,

dann entferne alle Einträge $e =$ `START(n,`μ`)` bzw. $e =$ `END(n,`μ`)` aus $\mathcal{A}_{red}$, für die gilt:

$$n \in (c\_succ^*(S, L_S) \cap c\_pred^*(S, L_E)) \cup \{L_S, L_E\} \wedge \mu < i$$

Wir bezeichnen $\mathcal{A}_{red}$ als iterationsfreie Ablaufhistorie.

Die iterationsfreie Ablaufhistorie einer WF-Instanz entsteht also durch Entfernen aller Historieneinträge, die von Aktivitätenknoten einer Schleife (inklusive des Schleifenknotens selbst) in einer anderen als der letzten (bei beendeten Schleifen) bzw. aktuellen (bei noch laufenden Schleifen) Iteration geschrieben wurden. $\mathcal{A}_{red}$ repräsentiert logisch betrachtet also eine WF-Instanz, bei der alle Schleifen maximal einmal durchlaufen worden sind. Erwähnenswert ist, dass für WF-Instanzen der Graphklasse 1 und 2 trivialerweise $\mathcal{A} = \mathcal{A}_{red}$ gilt.

Beispiel 3.9 (Iterationsfreie Ablaufhistorie) Wir betrachten nochmals die WF-Instanz aus Abb. 12. Hier ergibt sich $\mathcal{A}_{red} =$ <`START(A), END(A), START(`L_S`,3,TRUE), END(`L_S`,3), START(B,3),`

`END(B,3,sel_code=1), START(E,2), END(E,2)`>. Wie man leicht sieht, wäre diese Ablaufhistorie auch auf dem Schema, das sich durch Einfügen des Knotens X zwischen G und H ergibt, erzeugbar.

Wir geben nun ein gegenüber Definition 3.1 modifiziertes Verträglichkeitskriterium für WF-Graphen mit Schleifen an.

Definition 3.3 (Loop-Compliance) *Sei I eine Instanz eines WF-Schemas S (der Graphklasse 3) mit Ablaufhistorie* $\mathcal{A} =< e_0, \ldots, e_k >$ *und iterationsfreier Ablaufhistorie* $\mathcal{A}_{red} =< e_{i_1}, \ldots, e_{i_n} >$.

S werde durch eine Änderung Δ *auf das (korrekte) WF-Schema S' transformiert. Sei* M'_t *die Markierung des Ausführungsgraphen von I, der sich ergibt, wenn* Δ *auf I propagiert wird und anschließend eine Neubewertung des Zustands des Ausführungsgraphen durchgeführt wird. Dann ist I genau dann verträglich mit S', wenn die Folge von Ereignissen* $e_{i_1}, \ldots, e_{i_n}$ *der reduzierten Ablaufhistorie* $\mathcal{A}_{red}$*, ausgehend von einer Anfangsmarkierung* M'_0*, auch auf S' anwendbar ist und im Anschluss wieder eine korrekte Markierung* M'_t *resultiert. Formal:*

I ist verträglich mit S' $:\Leftrightarrow M'_0[e_{i_1} >, \ldots, [e_{i_n} > M'_t$

Mit anderen Worten: Eine WF-Instanz ist genau dann verträglich mit dem geänderten Schema S' gemäß Definition 3.3, wenn sich die reduzierte Ablaufhistorie auch auf dem geänderten WF-Schema S' erzeugen lässt. Die eingangs dargestellten Restriktionen sind dadurch aufgehoben. Dieses Kriterium ist auch bei geschachtelten Schleifen problemlos anwendbar.

Beispiel 3.10 (Loop-Compliance) Die in Beispiel 3.9 dargestellte iterationsfreie Ablaufhistorie ist auch auf dem geänderten WF-Schema, das nach Einfügen von Aktivität X zwischen G und H resultiert, erzeugbar. Damit ist die vorliegende Instanz loop-compliant zum geänderten WF-Schema und ensprechend migriert werden.

Wenn eine WF-Instanz im Sinne von Definition 3.3 verträglich mit dem neuen WF-Schema ist, entstehen keine Probleme bei ,,normalem" Fortschritt des Kontrollflusses. In seltenen Fällen kann es jedoch erforderlich werden, dass eine WF-Instanz beim Auftreten semantischer Fehler (z. B. Scheitern einer Aktivitätenbearbeitung) partiell oder vollständig zurückgesetzt werden muss. Damit verbunden sind sowohl Anpassungen der Markierung der WF-Instanzen und die Rücknahme gewisser Schreiboperationen auf den WF-Datenkontext als auch die Ausführung externer Kompensationsschritte. Soll nun die WF-Instanz in einen Ausführungszustand zurückgesetzt werden, der vor der Propagation einer Schemaänderung gültig war, muss bei Zugrundelegung des Loop-Compliance-Kriteriums die Standardvorgehensweise für das Rollback (siehe [44]) von WF-Instanzen geringfügig modifiziert werden. Vereinfacht ausgedrückt, muss für diesen Fall gegebenenfalls auch die Schemaänderung zunächst zurückgenommen werden (UNDO), um wieder zu konsistenten Graphmarkierungen zu gelangen. Anschließend kann dann versucht werden, die betroffene Änderung wieder auf die zurückgesetzte WF-Instanz zu propagieren (REDO, siehe auch [44]). Auf weitere Details hierzu verzichten wir an dieser Stelle.

3.3.3.1 Einfügen/Löschen von Aktivitätenknoten und Kontrollabhängigkeiten (Graphklasse 3)

Wir betrachten zunächst die in den Abschnitten 3.1 bzw. 3.2 diskutierten Schemaänderungen. Um bei ihrer Anwendung auf WF-Schemata der Graphklasse 3 für zugehörige WF-Instanzen ,,Loop-Compliance" gewährleisten zu können, genügt es, die selben Voraussetzungen zu fordern, wie im Fall von WF-Graphen der Graphklasse 2. Grund dafür ist, dass durch die Art der Definition der iterationsfreien Ablaufhistorie – sie ist hier für die Verträglichkeitsprüfung maßgebend – Schleifenrücksprungkanten keine Relevanz haben. Bei der Prüfung auf Verträglichkeit bezieht man sich hier also auf einen WF-Graphen, bei dem die Schleifenrücksprungkanten ,,ausgeblendet" sind. Insbesondere erhält man im Anschluss an die Reduzierung – logisch betrachtet – einen azyklischen Graphen. Damit sind alle im vorangegangenen Abschnitt 3.2 getroffenen Aussagen auch im vorliegenden Fall anwendbar. Wie bereits erwähnt, sind WF-Graphen ohne Schleifen genau dann und nur dann loop-compliant, wenn sie im Sinne von Definition 3.1 compliant sind.

Wir müssen uns nun noch mit Schemaänderungen befassen, die durch die bisher vorgestellten Graphtransformationen noch nicht abgedeckt sind. Dies betrifft insbesondere Operationen für das Einfügen bzw. Entfernen von Schleifen.

3.3.3.2 Einfügen, Löschen und Ändern von Schleifenblöcken

Um dem Modellierer eine breite Palette an Änderungsoperationen anzubieten, muss es auch möglich sein, Schleifenblöcke (L_S, L_E) mit Anfangsknoten L_S und Endknoten L_E in ein WF-Schema der (Graphklasse 3) einzufügen bzw. daraus zu löschen. Darüberhinaus sollen unter gewissen Voraussetzungen auch neue Schleifen um existierende Blöcke des WF-Schemas eingefügt oder Schleifenrücksprungkanten entfernt werden können. Unter welchen Voraussetzungen für diese Schemaänderungen auf Instanzebene Verträglichkeit mit dem neuen WF-Schema zugesichert werden kann, ist den folgenden beiden Sätzen zu entnehmen.

Satz 3.11 (Einfügen und Löschen von Schleifenblöcken) *Seien S ein korrektes WF-Schema der Graphklasse 3 und I eine WF-Instanz auf S. S werde durch eine Änderungsoperation op auf ein wiederum korrektes WF-Schema S' der Graphklasse 3 transformiert.*

(a) op fügt einen Schleifenblock mit Schleifenanfangsknoten L_S und Schleifenendknoten L_E in S ein. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

$\forall n \in c\_succ^*(L_E, S')$: *NS(n)* $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

(b) op löscht einen Schleifenblock mit Schleifenanfangsknoten L_S und Schleifenendknoten L_E aus S. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

NS(L_S) $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

Bei Zugrundelegung der iterationsfreien Ablaufhistorie kann der Beweis zu Satz 3.11 analog zur Beweisführung bei Hinzunahme und Entfernen von XOR-Verzweigungsblöcken (vgl. Satz 3.9) erfolgen.

Satz 3.12 (Einfügen und Löschen von Schleifenrücksprungkanten) *Seien S ein korrektes WF-Schema der Graphklasse 3 und I eine WF-Instanz auf S. S werde durch eine Änderungsoperation op auf ein wiederum korrektes WF-Schema S' der Graphklasse 3 transformiert.*

(a) op fügt eine neue Schleife (L_S, L_E) (inklusive Schleifenrücksprungkante $L_S \rightarrow L_E$) um einen existierenden Kontrollblock mit Startknoten A und Endknoten B in S ein. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

$\forall n \in c\_succ^*(B, S)$: *NS(n)* $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

(b) op löscht eine Schleifenrücksprungkante $L_S \rightarrow L_E$ (inklusive des Schleifenanfangsknotens L_S und Schleifenendknotens L_E) aus S. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

$\nexists e \in \mathcal{A}$ *mit* $e =$ `START(`L_S`,2)`

Satz 3.12a besagt, dass um einen Kontrollblock (z. B. eine Sequenz von Aktivitäten) ein Schleifenkonstrukt genau dann eingefügt werden kann, wenn dieser Block noch nicht vollständig durchlaufen wurde oder alle direkten Nachfolger keine Einträge in die Ablaufhistorie geschrieben haben. Eventuell kann es erforderlich werden, dass für den Startknoten der Schleife bei der Änderungspropagationen entsprechende Historieneinträge ,,nachträglich" geschrieben werden. Dies ist aber unkritisch, da es sich hierbei um einen Knoten ohne assoziierte Aktionen handelt (keine wirkliche Verfälschung der Vergangenheit). Umgekehrt ist das Löschen eines Schleifenkonstrukts (vgl. Satz 3.12b) (bei Erhalt des Schleifenkörpers) nur dann auf eine WF-Instanz anwendbar, wenn die Schleife maximal eine Schleifeniteration durchlaufen hat. Hier müssen die entsprechenden Start- und Endeinträge für Start- und Endknoten der Schleife – zumindest logisch betrachtet – aus $\mathcal{A}$ entfernt werden.

Im Zusammenhang mit Schleifenänderungen kann die Situation auftreten, dass eine WF-Instanz aufgrund ihrer aktuellen Markierung nicht verträglich im Sinne von Definition 3.3 mit dem neuen WF-Schema ist (z. B. wenn die aktuellen Schleifeniteration schon zu weit fortgeschritten ist). In einem solchen Fall darf die Änderungspropagation zunächst nicht erfolgen. Wird jedoch eine neue Schleifeniteration betreten, wird die WF-Instanz evtl. verträglich im Sinne von Definition 3.3 mit dem neuen WF-Schema, so dass die Änderungspropagation mit Verzögerung vorgenommen werden könnte. Wir kommen hierauf in Abschnitt 4 nochmals zurück.

3.3.4 Hierarchische Workflows (Graphklasse 4)

Um komplexe Geschäftsprozesse mit vielen Arbeitsschritten adäquat abbilden zu können, sollte ein WF-Metamodell die hierarchische Strukturierung von WF-Modellen gestatten. ADEPT

bietet hierfür zwei Modellierkonzepte an, die sowohl die Modularisierung von WF-Modellen als auch deren Wiederverwendbarkeit unterstützen. Wir wollen in diesem Abschnitt skizzieren, wie sich bei hierarchisch strukturierten Workflows Verträglichkeitsprüfungen darstellen.

Mit *Blockaktivitäten* wird dem Modellierer ein einfaches Mittel an die Hand gegeben, um seine WF-Modelle übersichtlicher zu gestalten. Eine Blockaktivität fasst eine oder mehrere zusammenhängende Aktivitäten eines WF-Modells in einem Block zusammen, dessen Detaildarstellung – je nach Bedarf – ein- oder ausgeblendet werden kann. Dabei kann die Blockbildung flexibel umgestaltet werden. Zu diesem Zweck bietet ADEPT dem Modellierer einfache Änderungsoperationen zur (nachträglichen) Bildung (`nestBlock`) bzw. Auflösung (`unnestBlock`) von Blockstrukturen bzw. Sub-WF-Modellen an (für Details siehe [44]). Dies bietet ihm die nötige Flexibilität, seine Modelle bei Bedarf neu zu strukturieren. Des weiteren dient das Blockkonzept dazu, die Sichtbarkeit von Prozessvariablen geeignet einzuschränken. Blockaktivitäten dienen ausschließlich Strukturierungszwecken und können deshalb nicht bei der Definition anderer WF-Modelle wiederverwendet werden. Auf Ausführungsebene können die Schritte einer Blockaktiviät – falls gewünscht – von den Schritten des übergeordneten Workflow unterschieden werden. Vereinfacht ausgedrückt basiert dies auf der unterschiedlichen ,,Färbung" von Aktivitäten des Sub- bzw. Super-Workflow. Dem gegenüber stehen *Prozessaktivitäten*. Sie repräsentieren ebenfalls einen Sub-Workflow, ihre Definition erfolgt aber nicht durch logische Klammerung von Prozessschritten eine Modells, sondern durch Einsetzen eines wiederverwendbaren WF-Modells in eine Aktivitätenschablone.

Wir zeigen nun, wie sich Verträglichkeitsprüfungen bei der Verwendung von Block- bzw. Prozessaktivitäten darstellen. Änderungen von WF-Modellen mit Blockaktivitäten unterscheiden sich konzeptuell nicht von Änderungen ,,flacher" WF-Modelle. Der Grund dafür ist, dass hierarchische Modelle mit Blockaktivitäten auf operativer Ebene in ein ausführungsäquivalentes, ,,flaches" Modell abgebildet werden, so dass für zugehörige WF-Instanzen die Verträglichkeitsprüfungen bei Änderungen ihres WF-Schemas nach den bisher definierten Regeln erfolgen können. Zwar erhalten bei dieser Abbildung die Sub-Aktivitäten einer Blockaktivität eine andere ,,Färbung" als die im WF-Modell übergeordneten Aktivitäten (siehe oben), dies dient aber allein Strukturierungs- bzw. Darstellungszwecken, hat also keinen Einfluss auf die Verträglichkeitsprüfungen selbst. Dem entsprechend beeinträchtigt auch die (dynamische) Anwendung von Blockbildungs- bzw. Blockauflösungsoperationen die Verträglichkeit von WF-Instanzen mit dem sich hieraus ergebenden Schema nicht.

Was WF-Modelle mit Prozessaktivitäten anbetrifft, verhält es sich etwas anders. Hier werden Instanzen einer Prozessaktivität auf operativer Ebene als separate Sub-Workflows (im Kontext des jeweils übergeordneten Super-Workflows) ausgeführt, d. h. die hierarchische Strukturierung bleibt im Gegensatz zu Blockaktivitäten physisch erhalten. Wird ein WF-Schema S abgeändert, spielt es im Zusammenhang mit Verträglichkeitsprüfungen jedoch kein Rolle, ob die zugehörigen WF-Instanzen als Toplevel- oder Sub-Workflow ablaufen. Ist eine (Sub-)WF-Instanz verträglich mit einem geänderte Schema, kann sie deshalb entsprechend migriert werden. Zu erwähnen bleibt, dass für noch nicht erzeugte Sub-Workflow-Instanzen keine besonderen Vorkehrungen zu treffen sind, da ADEPT das späte Binden von WF-Modellen unterstützt. Wurde bei der

Ausführung eines Super-Workflows eine zugehörige Prozessaktivität noch nicht aktiviert, können mit dieser Strategie bei Änderungen des zugehörigen Schemas in der Folge ohne weitere Maßnahmen übernommen werden. Wird umgekehrt das Schema eines Super-Workflows abgeändert, hängt die Verträglichkeit einer WF-Instanz auch vom aktuellen Zustand ihrer Prozessaktivitäten ab. Dieser leitet sich aus dem internen Zustand des jeweiligen Sub-WF ab (siehe [44]).

3.4 Effiziente Überprüfung auf Compliance (Datenflusssicht)

Bei unseren bisherigen Betrachtungen haben wir uns auf Kontrollflussaspekte beschränkt. Im Allgemeinen werden im Zusammenhang mit WF-Schemaänderungen jedoch auch Anpassungen des modellierten Datenflusses erforderlich. Inwieweit solche Datenfluss-Schemaänderungen die Verträglichkeit von WF-Instanzen mit dem geänderten WF-Schema beeinträchtigen oder nicht, soll in diesem Abschnitt behandelt werden. Dazu wiederholen wir zuerst in Abschnitt 3.4.1 wichtige Grundlagen der Datenflussmodellierung und -steuerung in ADEPT, sofern sie für das Verständnis der weiteren Ausführungen notwendig sind.

3.4.1 Datenflussmodellierung und -steuerung in ADEPT

In ADEPT erfolgt der *Datenaustausch* zwischen Aktivitäten über globale Prozessvariablen (sog. *Datenelemente*). Der *Datenfluss* wird modelliert, indem man ihre *Eingabe- und Ausgabeparameter* über *Datenkanten* mit diesen *Datenelementen* verknüpft. Dabei wird jeder Aktivitäten-*Eingabeparameter* mittels genau einer *Lesekante* und jeder Aktivitäten-*Ausgabeparameter* über genau eine *Schreibkante* an die *Datenelemente* angebunden. Ein Beispiel zeigt Abb. 13. Hier schreibt Aktivität B das *Datenelement* d_2, das nachfolgend von den Aktivitäten C und D gelesen wird. Die Gesamtheit aller einem WF-Schema zugeordneten *Datenelemente* und *Datenkanten* bezeichnen wir im Folgenden als *Datenfluss-Schema (DF-Schema)*. Neben einer solchen horizontalen DF-Modellierung können auch Daten zwischen Super- und Subworkflow ausgetauscht werden. Dies erfolgt durch Anknüpfung der *Ein- und Ausgabeparameter* der entsprechenden Block- bzw. Prozessaktivität an *Datenelemente* des übergeordneten WF.

Die beschriebene Form der Datenflussmodellierung bedeutet keine Einschränkung im Vergleich zu Ansätzen, bei denen Aktivitätenausgabe- und Aktivitäteneingabeparameter (sog. Datencontainer) direkt mittels Datenkonnektoren verknüpft werden. Grund ist, dass sich solche direkten Verknüpfungen immer auch über Datenelemente und -kanten emulieren lassen. Desweiteren gestattet die Verwendung globaler Datenelemte die einfache Modellierung von For-Schleifen (mit Schleifenzähler) oder den einfachen Datenaustausch zwischen Aktivitätenausführungen verschiedener Schleifeniterationen. Dies trifft für viele der in der WF-Literatur diskutierten Metamodelle nicht zu. Beispielsweise bildet MQSeries Workflow den Datenfluss über das Mapping von Aktivitätenausgabe- auf Aktivitäteneingabecontainern ab, was im Zusammenhang mit der Modellierung von Transitionsbedingungen oder dem Datenaustausch zwischen verschiedenen Blockiterationen zu Problemen bzw. Sonderbehandlungen führen kann.

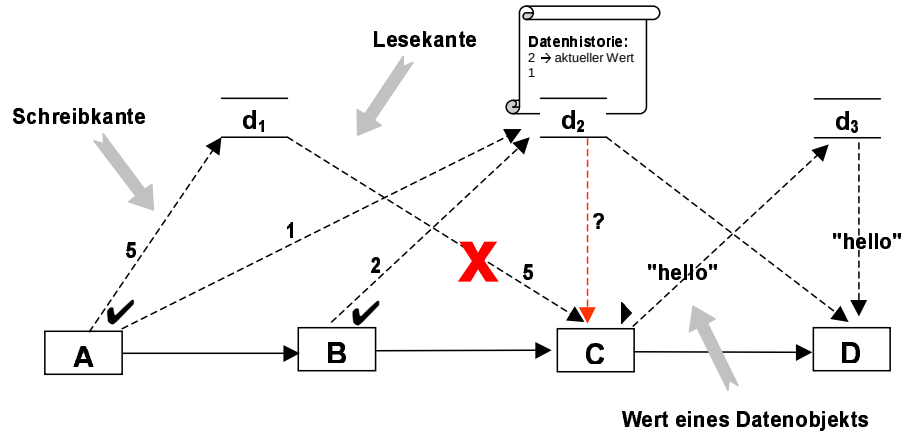


Abbildung 13: Änderung des DF-Schemas

Ebenso wie für die Kontrollflussmodellierung bestehen in ADEPT auch für die Definition von Datenflüssen gewisse Korrektheitsforderungen. Die wichtigste von ihnen ist, dass beim Start eines Aktivitätenprogramms alle obligaten Eingabeparameter versorgt sind, unabhängig von den zuvor durchlaufenen Ablaufpfaden. Die in diesem Zusammenhang bestehenden Beschränkungen und Verfahren zu ihrer effizienten Überprüfung werden in [44] ausführlich diskutiert.

Zur Ausführungszeit einer WF-Instanz verwaltet ADEPT für jedes Datenelement ein oder mehrere Versionen eines Datenobjekts. Bei einem Schreibzugriff auf ein Datenelement wird stets eine neue Version des Datenobjekts angelegt, d. h. Datenobjekte werden niemals physisch überschrieben. Die Verwaltung der verschiedenen Versionen ist wichtig für das kontextabhängige Lesen von Datenelementen und das korrekte Zurücksetzen im Fehlerfall (siehe [44]). Der Datenkontext einer WF-Instanz umfasst also für jedes Datenelement eine *Datenhistorie*, die diesem Datenelement eine geordnete Liste der Versionen des verwalteten Datenobjekts zuordnet.

3.4.2 Änderungen des Datenfluss-Schemas und Verträglichkeit von WF-Instanzen

Änderungen des DF-Schemas eines WF-Graphen können sowohl als begleitende Anpassungen zu Kontrollfluss-Änderungen (z. B. Löschen einer Aktivität mit Entfernen der assoziierten Datenkanten) als auch als eigenständige Operationen durchgeführt werden. Letzteres ist z. B. zur nachträglichen Korrektur von (semantischen) Fehlern bei der Datenflussmodellierung notwendig. Als Operationen für die Abänderung von DF-Schemata unterstützt ADEPT das Einfügen und Löschen von Datenelementen bzw. Datenkanten. Das in Definition 3.1 bzw. 3.3 eingeführte Verträglichkeitskriterium muss nun dahingehend erweitert werden, dass es auch in Verbindung mit Datenfluss-Änderungen sinnvoll anwendbar ist. Wie das nachfolgende Beispiel illustriert, kann es bei alleiniger Zugrundelegung von Definition 3.1 bzw. 3.3 bei der Migration von WF-Instanzen im Zusammenhang mit Datenfluss-Änderungen zu unerwünschten Effekten kommen:

Beispiel 3.11 Wir betrachten die WF-Instanz aus Abb. 13. Aktivität C befindet sich aktuell im Zustand `RUNNING`, hat also das Datenelement d_1 bereits gelesen. Wird nun die Lesekante von C auf d_1 gelöscht und eine neue Lesekante von C auf d_2 eingefügt – diese Änderung ist auf Schemaebene korrekt –, ist nicht mehr unmittelbar nachvollziehbar, von welcher Vorgängeraktivität C den bereits gelesenen Wert konsumiert hat. Dies kann beispielsweise beim späteren Zurücksetzen der WF-Instanz im Fehlerfall zu Problemen führen. Bei ausschließlicher Zugrundelegung des bisher vorgestellten Verträglichkeitskriteriums wäre der skizzierte Fall jedoch zulässig, d. h. die bisherige Ablaufhistorie wäre auch auf dem geänderten Schema erzeugbar. Aus diesem Grund benötigen wir ein angepasstes Verträglichkeitskriterium, das auch Datenflussaspekte einbezieht. Hierzu definieren wir zunächst eine erweiterte Ablaufhistorie für WF-Instanzen. Diese stellt eine um Informationen aus den Datenelementhistorien (vgl. Abschnitt 3.4.1) angereicherte Sicht auf die bisherige Ablaufhistorie dar.[8] Bezogen auf Abb. 13 könnte sich eine solche erweiterte Ablaufhistorie wie in Tabelle 1 darstellen.

Ablaufereignisse	`START(A)`	`END(A)`	`START(B)`	`END(B)`	`START(C)`
geschriebene Datenelemente	-	d_1, 5 d_2, 1	-	d_2, 2	-
gelesene Datenelemente	-	-	d_1, 5	-	d_1, 5

Tabelle 1: Um Schreib- und Lesezugriffe auf Datenelemente erweiterte Ablaufhistorie

Aus dieser erweiterten Ablaufhistorie geht eindeutig hervor, welche Datenelemente eine Aktivität mit welchen Werten gelesen bzw. geschrieben hat. Formal:

Definition 3.4 (Erweiterte Ablaufhistorie) *Seien S ein korrektes WF-Schema (bestehend aus korrektem KF-Schema CFS = (N, E, ...) und korrektem DF-Schema DFS) und I eine WF-Instanz auf S mit Ablaufhistorie $\mathcal{A}$. Dann heißt $\mathcal{A}_{ext}$ die um Datenelementzugriffe erweiterte Ablaufhistorie von $\mathcal{A}$, wenn $\mathcal{A}_{ext}$ für jeden Starteintrag einer Aktivität die Werte der gelesenen Datenelemente und für jeden Endeintrag entsprechend die Werte geschriebener Datenelemente enthält:*

$$\mathcal{A}_{ext} :=< e_1^{(d_1^{(1)},v_1^{(1)}),\ldots,(d_n^{(1)},v_n^{(1)})}, \ldots, e_k^{(d_1^{(k)},v_1^{(k)}),\ldots,(d_m^{(k)},v_m^{(k)})} >,$$

Ein Tupel $(d_i^{(\mu)}, v_i^{(\mu)})$ beschreibt dabei einen Lese- bzw. Schreibzugriff von e_μ auf das Datenelement $d_i^{(\mu)}$ (mit zugehörigem Wert $v_i^{(\mu)}$).

Auf Grundlage dieser Definition geben wir nun ein modifiziertes Verträglichkeitskriterium für WF-Instanzen an, das auch in Verbindung mit Änderungen des DF-Schemas anwendbar ist (hier zunächst ohne Berücksichtigung von Schleifen):

[8] Wir gehen im Folgenden nicht auf die physische Verwaltung der Datenhistorie ein.

Definition 3.5 (Verträglichkeit bei Kontroll- und Datenflussänderungen) *Seien S ein korrektes WF-Schema (bestehend aus korrektem KF-Schema CFS = (N, E, ...) und korrektem DF-Schema DFS) und I eine WF-Instanz mit erweiterte Ablaufhistorie* $\mathcal{A}_{ext} =< e_1^{(d_1^{(1)},v_1^{(1)}),\ldots,(d_n^{(1)},v_n^{(1)})}, \ldots, e_k^{(d_1^{(k)},v_1^{(k)}),\ldots,(d_m^{(k)},v_m^{(k)})} >$.

S werde durch eine Änderung Δ auf das (korrekte) WF-Schema S' transformiert. Dann heißt I verträglich mit S' dann und nur dann, wenn $\mathcal{A}_{ext}$ auch auf dem modifizierten WF-Schema S' erzeugbar ist.

Diese Definition besagt Folgendes: Zum einen müssen die in Definition 3.1 geforderten Bedingungen für Kontrollflussänderungen erfüllt sein, zum anderen muss zusätzlich gelten, dass jede bereits gestartete oder beendete Aktivität bei Ausführung auf dem neuen WF-Schema dieselben Datenelemente (bzw. Datenelementwerte) lesen würde und dass jede beendete Aktivität diesselben Datenelemente geschrieben hätte.

Nun stellt sich die Frage, wie die Verträglichkeit einer WF-Instanz bei Änderungen des DF-Schemas effizient überprüft werden kann. Ziel hierbei ist – ähnlich wie bei Betrachtung des Kontrollflusses – effizient überprüfbare Bedingungen anzugeben, bei deren Einhaltung die Verträglichkeit der betrachteten WF-Instanz mit dem geänderten WF-Schema gewährleistet ist. Der folgende Satz gibt an, welche Statusvoraussetzungen für DF-Änderungsoperationen zu fordern sind, um Verträglichkeit im Sinne von Definition 3.5 zusichern zu können.

Satz 3.13 (Einfügen/Löschen von Datenelementen und Datenkanten) *Seien S ein korrektes WF-Schema (bestehend aus korrektem KF-Schema CFS = (N, E, ...) und korrektem DF-Schema DFS) und I eine Instanz auf S. S werden durch eine Änderungsoperation op auf ein ebenfalls korrektes WF-Schema S' = (CFS', DFS') transformiert.*

(a) op fügt ein Datenelement d in DFS ein. Dann gilt:

I verträglich mit S'.

(b) op löscht ein Datenelement d aus DFS. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

[Es existiert keine Aktivität mit Status `RUNNING` *oder* `COMPLETED`*, die lesend oder schreibend auf dieses Datenelement zugegriffen hat.]*

(c) Zwischen Aktivität n und Datenelement d wird durch op eine Lesekante $d \rightarrow n$ eingefügt bzw. gelöscht. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

NS(n) $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

(d) Zwischen Aktivität n und Datenelement d wird durch op eine Schreibkante $n \rightarrow d$ eingefügt bzw. gelöscht. Dann gilt:

I verträglich mit S' ⇔

NS(n) $\neq$ `COMPLETED`

Das Einfügen eines Datenelements ist immer möglich, da im Verlauf der bisherigen WF-Ausführung noch keine Aktivitäten lesend oder schreibend darauf zugegriffen haben. Das Löschen eines Datenelements ist nur dann unkritisch, wenn noch keine entsprechenden Lese- oder Schreibzugriffe stattgefunden haben. Lesekanten auf Datenelemente können eingefügt oder gelöscht werden, wenn die zugehörige Aktivität noch nicht gestartet worden ist. Schreibkanten können sogar noch gelöscht bzw. eingefügt werden, wenn die zugehörige Aktivität bereits gestartet (aber noch nicht beendet) wurde.

Wie bereits erwähnt, werden auch im Zusammenhang mit dem Einfügen und Löschen von Aktivitäten auf Schemaebene begleitende Anpassungen des Datenflusses vorgenommen. Für diese Fälle sind die Voraussetzungen von Satz 3.13 erfüllt, wenn die Statusbedingungen für die entsprechende Knoteneinfüge- bzw. Knotenlöschoperation gelten (vgl. Sätze 3.7 und 3.8).

Werden bei einer WF-Schemaänderung mehrere Operationen zur Anpassung des modellierten Datenflusses angewendet, müssen entsprechend wieder die Bedingungen für die Anwendung jeder einzelnen Operation erfüllt sein. Hintergrund ist auch hier wieder, wie schon im Zusammenhang mit Kontrollflussänderungen, dass bei Anwendung mehrerer solcher Datenflussänderungsoperationen sich stets eine Serialisierung der angewandten Änderungen finden lässt, bei der die einzelnen Zwischen-Schemata korrekte DF-Schemata sind.

Zu untersuchen bleibt, wie sich Definition 3.3 (Loop-Compliance) und Definition 3.5 im Zusammenspiel ,,vertragen". Als Basis für beide Varianten des Verträglichkeitskriteriums dient eine modifizierte Ablaufhistorie $\mathcal{A}_{ext,red}$. Dazu wir $\mathcal{A}$ zunächst um die Informationen aus der Datenhistorie angereichert ($\rightarrow \mathcal{A}_{ext}$) und dann gemäß Definition 3.2 reduziert ($\rightarrow \mathcal{A}_{red}$). Dann ist eine WF-Instanz I genau dann verträglich mit einen geänderten Schema S', wenn $\mathcal{A}_{ext,red}$ auch auf S' erzeugbar ist. Auf Details verzichten wir an dieser Stelle.

4 Effiziente Migration von WF-Instanzen und Markierungsanpassungen

Im vorangehenden Abschnitt haben wir gezeigt, wie sich bei Schemaänderungen die Verträglichkeit von WF-Instanzen mit dem resultierenden WF-Schema effizient feststellen lässt. Ziel dabei ist es gewesen, Zugriffe auf die (komplette) Ablaufhistorie weitgehend zu vermeiden. In diesem Abschnitt gehen wir darauf ein, wie unter Beibehaltung dieser Zielsetzung verträgliche WF-Instanzen effizient und korrekt auf das neue WF-Schema migriert werden können.

4.1 Problemstellung und Lösungsansätze

Wir gehen im Folgenden davon aus, dass ein gegebenes WF-Schema S (einer beliebigen Graphklasse) in ein wiederum korrektes WF-Schema S' abgeändert wird. Ferner seien durch $I_1, \ldots, I_n$ WF-Instanzen gegeben, die auf Grundlage von S erzeugt wurden und die verträglich mit dem neuen WF-Schema S' sind. Für sie stellt sich nun die Frage, wie ihre Migration auf das neue WF-Schema konkret erfolgen soll. Insbesondere muss für jede WF-Instanz I_k geklärt werden, ob ihre bisherigen Zustandsmarkierungen unverändert übernommen werden können oder ob Anpassungen (von Markierungen und Arbeitslisten) erforderlich werden. Das betrifft auch den Status von Knoten und Kanten, die infolge der Schemaänderung neu erzeugt worden sind.

Abhängig von der Art und dem Umfang einer Schemaänderung sowie dem aktuellen Status einer zu migrierenden WF-Instanz I_k können die erforderlichen Zustandsanpassungen mehr oder weniger umfangreich ausfallen. Betrifft die Schemaänderung einen Bereich des WF-Graphen, den die betrachtete WF-Instanz aktuell noch nicht betreten hat, so sind – abgesehen von der Initialisierung des Status neu hinzukommender Knoten und Kanten (mit `NOT_ACTIVATED` bzw. `NOT_SIGNALED`) – keine Zustandsanpassungen erforderlich. Anders verhält es sich, wenn der Kontext einer Änderung auch Knoten und Kanten umfasst, die bereits aktiviert bzw. signalisiert worden sind. Beispielsweise kann es beim Einfügen eines Aktivitätenknotens (und seiner Kontextkanten) erforderlich werden, dass die Aktivierung bisher aktivierter (und damit in Arbeitslisten gestellter) Schritte wieder aufgehoben wird oder umgekehrt die neu hinzugefügte Aktivität sofort aktiviert wird (vgl. Abb. 14). Damit verbunden sind häufig Anpassungen von Benutzerarbeitslisten, für die entsprechend Einträge entfernt bzw. hinzugenommen werden müssen. Ähnliches gilt auch in Bezug auf das Einfügen oder Entfernen von Kontroll- bzw. Sync-Kanten auf Instanzebene, die ebenfalls die Aktivierung bzw. Deaktivierung von Aktivitäten zur Folge haben können.

Für komplexere Schemaänderungen, die mehrere Elementaroperationen umfassen, können die notwendigen Zustands- bzw. Markierungsanpassungen von Knoten und Kanten sehr viel umfangreicher ausfallen. Hier ist die Beantwortung der Frage, wie die Markierungen effizient und konsistent angepasst werden können, im Allgemeinen nicht trivial. Zudem kann sie von der Semantik des verwendeten WF-Ausführungsmodells abhängen, wie das nachfolgende Beispiel aus dem Petri-Netz-Bereich illustriert:

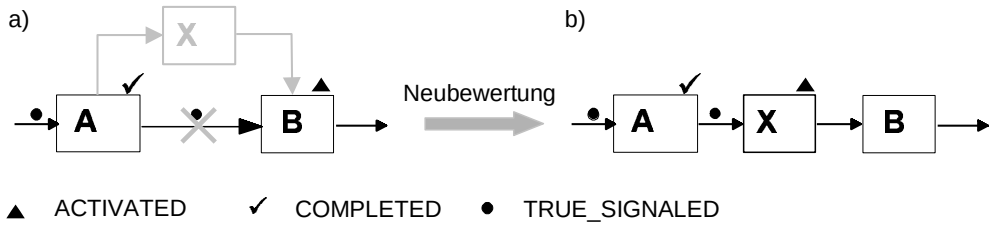


Abbildung 14: Neubewertung von Zustandsmarkierungen bei Einfügen von Knoten (hier: Einfügen von X zwischen der beendeten Aktivität A und der bereits aktivierten Aktivität B)

Beispiel 3.12 Abb. 15a zeigt ein Petri-Netz-basiertes WF-Schema, bei dem z. B. die Aktivitäten A, B, C, D und E sequentiell und die Aktivitäten F und G parallel ausführbar sind (entspricht in unserem Ansatz Graphen der Klasse 1). Die Vor- und Nachbedingungen für die Ausführung von Aktivitäten werden dabei über Stellen abgebildet, die hier durch Kreise dargestellt sind. Solche Stellen können bei der WF-Ausführung Marken (engl.: Token) tragen, die angeben, ob die assoziierte Bedingung erfüllt ist oder nicht. Eine Netzaktivität ist dem entsprechend ausführbar, wenn alle Stellen ihres Vorbereichs eine Marke tragen. Dies trifft aktuell nur für die Aktivität C zu. Beim Start der Aktivität werden diese Marken abgezogen, bei ihrer Beendigung wird umgekehrt den Stellen des Nachbereichs eine Marke zugeführt. Der Zustand einer Instanz I wird durch die aktuellen Token-Belegungen des zugehörigen Netzes beschrieben, wobei auch mehrere Stellen gleichzeitig markiert sein können.

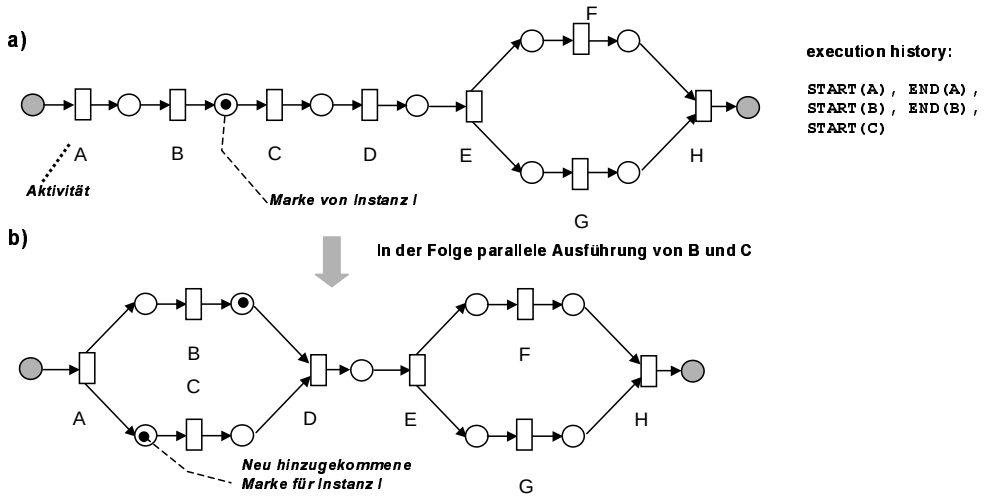


Abbildung 15: Schema-Änderungen bei Petri-Netzen

Das in Abb. 15a dargestellte Schema soll nun so modifiziert werden, dass die bisher sequentiell ausführbaren Aktivitäten B und D nun parallel angeordnet sind. Dazu wird S wie in Abb. 15b

dargestellt abgeändert. Wie man der Ablaufhistorie von I aus Abb. 15a entnehmen kann, ist diese Instanz verträglich mit dem resultierenden Netz. Allerdings werden bei der Migration Markierungsanpassungen notwendig, um mit der Ausführung in einem konsistenten Zustand fortfahren zu können. Abb. 15b kann entnommen werden, wie diese Markenanpassungen für I aussehen müssen. Wie man leicht sieht, besitzt die betrachtete Instanz nach der Migration mehr Marken als zuvor. Dies wird notwendig, damit es in der Folge zu keinen Verklemmungen kommt. Im Allgemeinen sind solche Markierungsanpassungen jedoch nicht trivial, insbesondere wenn komplexere Netze vorliegen oder wenn Tokens nicht nur Bedingungen repräsentieren, sondern auch Informationen tragen (siehe z. B. [16, 39]).

Für die Anpassung von Zustandsmarkierungen bei der Migration von Instanzen sind verschiedene Alternativen denkbar:

Ansatz 1: Automatische Anpassung durch Einbeziehung der Ablaufhistorie
Die naheliegenste Lösung wäre es, für die Berechnung der Zustandsmarkierungen die (komplette) Ablaufhistorie heranzuziehen und alle bisherigen Ablaufereignisse auf dem neuen WF-Schema ,,nachzuspielen". Dies gestaltet sich jedoch insbesondere in Verbindung mit Schleifen, die möglicherweise viele Iterationen durchlaufen haben, sehr aufwendig. Dieser ineffiziente Ansatz scheidet deshalb bei großer Zahl von WF-Instanzen oder umfangreichen Historiendaten aus den in Abschnitt 3.2 genannten Gründen aus. Insbesondere würde er die in Kapitel 3 erarbeiteten Konzepte für Verträglichkeitsprüfungen konterkarieren.

Ansatz 2: Festlegung der erforderlichen Zustandsanpassungen durch WF-Modellierer
Wie aber lassen sich Zugriffe auf die (komplette) Ablaufhistorie im Zusammenhang mit der Anpassung von Zustandsmarkierungen vermeiden? Ein naiver Ansatz wäre es, diese Anpassungen für jede WF-Instanz ,,manuell" durch den Modellierer vornehmen zu lassen (wie z. B. im System Chauquata [20] vorgeschlagen und implementiert). Dies ist jedoch bei großer Anzahl von WF-Instanzen schlichtweg nicht praktikabel. Der Modellierer müsste für jede mit dem neuen WF-Schema verträgliche WF-Instanz angeben, welche Zustandsmarkierungen bei ihrer Migration beibehalten werden sollen, welche neu hinzukommen und welche ggf. wegfallen. Um dabei sicherzustellen, dass es in der Folge zu keinen Verklemmungen oder unerwünschten Zuständen kommt, müssten bei einer solch manuellen Anpassung von Zustandsmarkierungen auf Instanzebene zudem sehr aufwendige Erreichbarkeitsanalysen (mit exponentieller Komplexität) durchgeführt werden. Dies alles würde zu erheblichen Verzögerungen bei der Ausführung der WF-Instanzen führen, was in der Praxis jedoch nicht akzeptabel ist. Andere Ansätze, etwa aus dem Petri-Netz-Bereich [2, 4], schlagen deshalb vor, dass der Modellierer eine Funktion zur Abbildung der Markierungen zwischen Quell- und Zielnetz angibt, die dann auf jede Instanz anwendbar ist. Auch dieser Ansatz ist eher von theoretischer Natur, da er für den Modellierer mit einem hohen Aufwand und einer hohen Komplexität verbunden ist. Zu Illustrationszwecken wenden wir diese Variante auf das Petri-Netz aus Abb. 15 an. Bereits für dieses sehr einfache Szenario ergeben sich für das (Quell-)Netz aus Abb 15a) 10 mögliche Markierungen. Demenstprechend viele Transformationsvorschriften muss der Modellierer für die Abbildung dieser Markierungen auf Markierungen des neuen Netzes definieren. Bei Petri-Netzen mit hohem Grad an Parallelität würde ein solcher Ansatz allein aus kombinatorischen Gründen zum Scheitern verurteilt sein.

Ansatz 3: Automatische Anpassung durch WfMS gemäß wohldefinierter Regeln
Als einzige vernünftige Lösung verbleibt, dass die Zustandsmarkierungen verträglicher WF-Instanzen bei deren Migration automatisch angepasst werden. Dies muss zum einen effizient, d. h. ohne unnötige Zugriffe auf die Ablaufhistorie, erfolgen, zum anderen müssen die resultierenden Markierungen wieder konsistent sein. Konsistent bedeutet in diesem Zusammenhang, dass sich für eine Instanz I_k nach ihrer Migration auf das neue Schema S' dieselben Zustandsmarkierungen ergeben, die man auch bei vollständiger Ausführung von I auf Grundlage von S' erhalten hätte. Im nachfolgenden Abschnitt skizzieren wir einen Lösungsansatz, der diesen beiden Anforderungen gerecht wird.

4.2 Automatische und effiziente Markierungsanpassungen bei Migrationen

Wir skizzieren nun ein Verfahren, mit dem sich die Knoten- und Kantenmarkierungen von (verträglichen) WF-Instanzen bei ihrer Migration automatisch anpassen lassen. Dabei werden jeweils nur Knoten und Kanten der von der Änderung betroffenen Bereiche des WF-Graphen neu bewertet, wodurch sich der Gesamtaufwand (gegenüber einer Neubewertung aller Knoten und Kanten) erheblich reduzieren lässt. Des weiteren stellt das Verfahren sicher, dass für eine WF-Instanz nach ihrer (automatischen) Migration wieder ein korrekter und konsistener Zustand resultiert. Dadurch wird ein unerwünschtes dynamisches Verhalten (z. B. Blockierungen im Verlauf der weiteren Ausführung) systemseitig ausgeschlossen.

4.2.1 Grundlagen

Grundlegend für das nachfolgend vorgestellte Bewertungsverfahren sind die bereits erwähnten Eigenschaften des ADEPT-Ausführungsmodells: Zum einen bleiben die Markierungen beendeter bzw. nicht mehr ausführbarer Knoten (`COMPLETED` bzw. `SKIPPED`) beim Fortschreiten des Kontrollflusses (in Vorwärtsrichtung) erhalten, wodurch der aktuelle Verlauf der WF-Ausführung direkt aus den aktuellen Markierungen der betreffenden WF-Instanz ersichtlich wird. Zum anderen gibt es für die Markierung und Ausführung von Aktivitätenknoten wohldefinierte, einfach implementierbare Regeln (Markierungs- bzw. Ausführungsregeln), die vergleichbar zu Schaltregeln bei Petri-Netzen sind (siehe [39]). *Markierungsregeln* geben dabei (jeweils abhängig vom Knotentyp) an, welche ausgehenden Kanten bei Beendigung der assoziierten Aktivität mit `TRUE_SIGNALED` und welche mit `FALSE_SIGNALED` markiert werden sollen. Umgekehrt definieren *Ausführungsregeln* die Bedingungen, die für die eingehenden Kanten eines Knotens erfüllt sein müssen, damit er als aktiviert (`ACTIVATED`) bzw. nicht mehr ausführbar (`SKIPPED`) markiert werden darf. Einige einfache Beispiele zeigt Abb. 16. Erwähnt sei an dieser Stelle, dass diese Regeln in ADEPT auch die Grundlage für die ,,normale" WF-Steuerung bilden (siehe [42]).

Wir werden uns diese beiden Ausführungseigenschaften im Zusammenhang mit der Neuberechnung von Zustandsmarkierungen bei Migrationen zunutze machen.

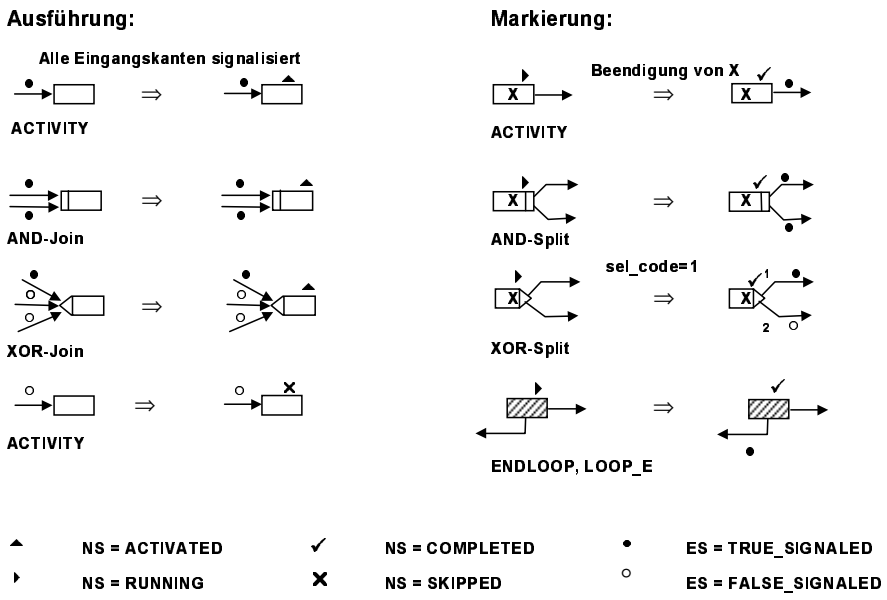


Abbildung 16: Ausführungs- und Markierungsregeln (exemplarische Darstellung)

4.2.2 (Initiale) Bestimmung neu zu bewertender Knoten und Kanten

Im Allgemeinen sollten bei einer Migration nur diejenigen Knoten und Kanten einer Neubewertung unterzogen werden, für die potentiell eine Statusanpassung erforderlich wird. Wir zeigen zuerst, wie sich diese eingeschränkte Knoten- bzw. Kantenmenge bestimmen lässt. Im nachfolgenden Abschnitt gehen wir dann darauf ein, wie hiervon ausgehend die Neubewertung der Zustandsmarkierungen bei der Migration einer verträglichen WF-Instanz erfolgen kann.

Ermittelt werden müssen also die Menge E_{check} der Kanten und die Menge N_{check} der Knoten (aus dem neuen WF-Schema), deren Status bei der Migration von Instanzen (auf das geänderte Schema) neu bewertet werden soll. Tabelle 2 gibt an, wie sich diese Mengen bei der Anwendung elementarer Änderungen (vgl. Kapitel 3) darstellen. Beispielsweise muss beim Einfügen einer neuen Aktivität X (inklusive ihrer Kontextkanten) der Status ihrer direkten Nachfolgerknoten sowie der Status der in X einmündenden Kontrollkanten neu bewertet werden, um inkonsistente Markierungen ausschließen zu können (vgl. Tabelle 2, 1. Zeile). (Eine Neubewertung von X selbst wird nur dann erforderlich, wenn eine dieser Kanten mit `TRUE_SIGNALED` bzw. `FALSE_SIGNALED` markiert werden kann.) Bei Hinzufügen einer Kontroll- oder Sync-Kante $n_{src} \rightarrow n_{dest}$ wiederum müssen sowohl die Kante selbst als auch ihr Zielknoten n_{dest} neu bewertet werden (vgl. Tabelle 2, 2. Zeile). Letzteres ist auch dann erforderlich, wenn die Kante selbst als `NOT_SIGNALED` bewertet

[9] Beim Löschen einer Aktivität wird eine Menge E_{add} von Kontextkanten hinzugefügt und eine Menge E_{delete} von Kontextkanten gelöscht.

[10] e sei die Kontextkante, die Z mit dem XOR-Splitknoten verknüpft

Angewandte Operation ...	... und neu zu bewertende Knoten und Kanten
Hinzufügen eines Aktivitätenknotens X (inkl. Kontextkanten)	$N_{check} := N_{check} \cup \text{succ(S',X)}$ $E_{check} := E_{check} \cup \{n_{src} \rightarrow n_{dest} \in \text{E'} \mid n_{dest} = \text{X}\}$
Hinzufügen einer Kontroll- bzw. Sync-Kante $n_{src} \rightarrow n_{dest}$	$E_{check} := E_{check} \cup \{n_{src} \rightarrow n_{dest}\}$ $N_{check} := N_{check} \cup \{n_{dest}\}$
Löschen einer Kontroll- bzw. Sync-Kante $n_{src} \rightarrow n_{dest}$	$N_{check} := N_{check} \cup \{n_{dest}\}$
Löschen eines Aktivitätenknotens X (inkl. Hinzunahme oder Entfernen von Kontextkanten)	$E_{check} := E_{check} \cup E_{add}$[9] $N_{check} := N_{check} \cup \text{succ(S,X)}$
Einfügen eines neuen Teilzweigs Z in eine XOR-Verzweigung	$E_{check} := E_{check} \cup \{\text{e}\}$[10]
Einfügen eines Kontrollblocks (n_1, n_2) mit Startknoten n_1 und Endknoten n_2 (z. B. Schleifenblock, Sequenz)	$N_{check} := N_{check} \cup \text{succ(S',}n_2\text{)}$ $E_{check} := E_{check} \cup \{n_{src} \rightarrow n_{dest} \mid n_{dest} = n_1\}$

Tabelle 2: Beispiele für Anpassungen betroffener Knoten- und Kantenmengen

wird. Der Grund dafür ist, dass für diesen Fall eine möglicherweise bereits erfolgte Aktivierung von n_{dest} aufgrund des Hinzufügens der Kante wieder aufgehoben werden muss.

Wir wollen noch kurz darauf eingehen, wie sich die Mengen E_{check} und N_{check} bei Anwendung komplexer Änderungen, deren Definition mehrere Elementaroperationen $op_1, \ldots, op_n$ umfasst, bestimmen lassen. Bei oberflächlicher Betrachtung liegt die Vermutung nahe, dass sich E_{check} und N_{check} durch Vereinigung der aus der Anwendung einzelner Änderungsoperationen resultierenden Mengen $E_{check}(op_i)$ bzw. $N_{check}(op_i)$ bestimmen lässt. Dem ist jedoch im Allgemeinen nicht so, da sich einzelne Operationen bei der Definition der Gesamtschemaänderung aufeinander beziehen können. Insbesondere können durch die Anwendung einer Operation op_i die Effekte vorangehend definierter Änderungen $op_{i-1}, \ldots, op_1$ teilweise wieder aufgehoben werden (vgl. Beispiel 3.13). Algorithmus 1 berücksichtigt diesen Aspekt und liefert als Ergebnis die gewünschten Mengen E_{check} und N_{check}.

Algorithmus 1 (Bestimmung von E_{check} und N_{check} bei komplexen Änderungen)

```
input
   op_1, ..., op_n: Zur Anwendung kommende Elementaroperationen
output
   E_check: Menge der neu zu bewertenden Kanten
   N_check: Menge der neu zu bewertenden Knoten
begin
   E_check := ∅; N_check := ∅;
   //Aggregation der relevanten Knoten- und Kantenmengen, die sich bei Anwendung
   //der einzelnen Operationen ergeben
   for i:=1 to n do
      E_check := E_check ∪ E_check(op_i);
      N_check := N_check ∪ N_check(op_i);
   done

   // Entfernen aller Kanten aus E_check, die nicht in der Kantenmenge
   // E' des aus der Änderung hervorgegangenen Zielschemas S' enthalten sind.
   E_check := E_check ∩ E';

   // Entfernen aller Knoten aus N_check, die nicht in der Knotenmenge N
   // des ursprünglichen Quellschemas S enthalten sind.
   N_check := N_check ∩ N;
end
```

Wie man leicht sieht, ergibt sich die Menge E_{check} durch Vereinigung der entsprechenden Kantenmengen der angewandten Änderungsoperationen $op_1, \ldots, op_n$. Aus dieser Menge werden dann noch diejenigen Kanten entfernt, die im neuen WF-Schema nicht enthalten sind. Sie wurden im Verlauf der Änderungsdefinition zwar erzeugt, sind dann aber infolge der Anwendung weiterer Änderungsoperationen wieder weggefallen, so dass ihr Status nicht evaluiert werden muss. Was die Zusammensetzung der Menge N_{check} anbetrifft, müssen nur Knoten betrachtet werden, die bereits im Quell-Schema S enthalten waren. (Das nachfolgende Bewertungsverfahren ist so konstruiert, dass die Bewertung neu hinzugekommener Aktivitätenknoten nur erfolgt, wenn eine in diesen Knoten einmündende Kante bei der Anwendung dieses Verfahrens neu markiert wird.)

Wichtig ist, dass die Bestimmung der beiden Mengen E_{check} und N_{check} nur einmalig bei der Definition der Schemaänderung erfolgen muss. Insbesondere resultiert bei der Durchführung der Migrationen kein zusätzlicher Aufwand. Erwähnenswert ist noch, dass es in ADEPT neben den üblichen Elementaroperationen vordefinierte, komplexe Änderungsoperationen gibt (z. B. für das Einfügen eines Schleifen- oder Verzweigungsblocks), für die die Mengen der neu zu bewertenden Knoten und Kanten ebenfalls möglichst ,,minimal" gehalten werden.

4.2.3 Verfahren zur Anpassung von Markierungen bei Migrationen

Ein gegebenes WF-Schema S werde nun durch Anwendung der Operationen $op_1, \ldots, op_n$ in ein wiederum korrektes WF-Schema S' abgeändert. Ferner seien mit E_{check} und N_{check} die Mengen der bei einer Migration (initial) neu zu bewertenden Kanten bzw. Knoten, wie im vorangehenden

Abschnitt dargestellt, bestimmt worden.

Algorithmus 2 gibt an, wie sich die Markierungen (NS', ES') einer mit S' verträglichen WF-Instanz I von S bei ihrer Migration auf S' effizient bestimmen lassen. (NS ist dabei die Knotenmarkierungs- und ES die Kantenmarkierungsfunktion der WF-Instanz nach Migration auf S'). Im Kern basiert dieser Algorithmus auf den im Abschnitt 4.2.1 beschriebenen Ausführungs- und Markierungsregeln für Knoten und Kanten. Ausgangspunkt für die Anwendung dieser Regeln bilden die bei der Änderungsdefinition ermittelten Mengen N_{check} und E_{check}. Ergeben sich im Verlauf der Bewertung von Knoten und Kanten jedoch gewisse Anpassungen, werden gegebenenfalls auch weitere Knoten und Kanten neu bewertet (z. B. kaskadierende Markierung aller Knoten eines abgewählten Zweiges).

Die Algorithmen 1 und 2 funktionieren bei beliebig komplexen Änderungen. Durch ihre Konstruktion wird die Art und Weise sowie der Umfang der erforderlichen Statusanpassungen gegenüber den beschriebenen Alternativen 1 und 2 erheblich vereinfacht bzw. reduziert. Während Algorithmus 1 nur einmalig bei der Änderungsdefinition anzuwenden ist, muss Algorithmus 2 für jede zu migrierende Instanz ausgeführt werden. Durch die Konstruktion des vorgestellten Bewertungsverfahrens ist in der Mehrzahl der Fälle für die zu migrierenden WF-Instanzen nur eine kleine Teilmenge ihrer Knoten und Kanten neu zu bewerten. Dies gilt selbst dann, wenn die vorgenommenen Änderungen in verschiedenen Bereichen des WF-Graphen vorgenommen wurden (im Gegensatz etwa zu dem in [19] beschriebenen Ansatz). Da in der Praxis WF-Modelle mehrere Dutzend (oder noch mehr) Aktivitäten umfassen können [49], ist eine solche Reduzierung auch zwingend erforderlich.

Man kann formal zeigen, dass bei Anwendung dieses Verfahrens auf verträgliche Instanzen im Anschluss wieder eine konsistente Markierung resultiert. Dieselbe Markierung könnte man auch erhalten, wenn man die komplette Ablaufhistorie auf dem neuen Schema „nachspielt". Dies wäre allerdings mit einem wesentlich höheren Aufwand verbunden, da dann sehr viel mehr Knoten und Kanten, in Verbindung mit Schleifen gegebenenfalls sogar mehrfach, bewertet werden müssen. Hinzu kommt, wie eingangs dieses Beitrags erwähnt, dass die (komplette) Ablaufhistorie einer WF-Instanz oftmals nicht im Hauptspeicher des WF-Servers liegt. Fügt man z. B. in das WF-Schema aus Abb. 12a eine neue Aktivität X zwischen E und G ein, so müssen bei Anwendung der Algorithmen 1 und 2 lediglich die neu hinzukommende Kontrollkante $(E \rightarrow X)$ sowie die Knoten G und X neu bewertet werden, um auch nach Migration der betreffenden WF-Instanz wieder eine konsistente Markierung zu erhalten. Ermittelt man diese Markierung dagegen durch „Nachspielen" der kompletten Ablaufhistorie müssten deutlich mehr Knoten- und Kantenbewertungen erfolgen. In Abb. 12 beispielsweise würden mindestens soviele Markierungsanpassungen wie Einträge in der Ablaufhistorie erfolgen, wobei z. B. im Zusammenhang mit XOR-Verzweigungsknoten und Loop-Endknoten auch mehrere Knoten in ihrer Markierung angepasst werden müssten. Wir illustrieren die prinzipielle Funktionsweise bzw. Anwendung der beiden Algorithmen anhand einfacher Beispiele.

Algorithmus 2 (Neubewertung von Zustandsmarkierungen bei Migrationen)

```
input
   NS, ES: Bisherige Knoten-/Kantenmarkierungen von I (bezogen auf S=(N, E))
   E_check, N_check: Menge der neu zu bewertenden Kanten bzw. Knoten
output
   NS', ES': Neu berechnete Knoten-/Kantenmarkierungen von I (bezogen auf S'=(N', E'))
begin
   // Initialisierung von ES'(e) durch Übernahme der bisherigen Kantenmarkierung
   // bzw. durch Neumarkierung der Kante mit NOT_SIGNALED
   forall e ∈ E' do
      if e ∈ E then // Knoten war bereits in N bzw. S enthalten
         ES'(e):= ES(e)
      else
         ES'(e):= NOT_SIGNALED
   done
   // Initialisierung von NS' durch Übernahme der bisherigen Knotenmarkierung
   // bzw. durch Neumarkierung mit NOT_ACTIVATED
   forall n ∈ N' do
      if n ∈ N then // Knoten war bereits in N bzw. S enthalten
         NS'(n):= NS(n)
      else
         NS'(n):= NOT_ACTIVATED
   done
   // Neubewertung der Kanten aus E_check bzw. der Knoten aus N_check;
   // erforderlichenfalls iterative auch Neubewertung weiterer Kanten und Knoten
   repeat
      //
      while E_check ≠ ∅ do
         Entnehme eine Kante e = n_src → n_dest aus E_check;

         Überprüfe durch Anwendung der ADEPT-Markierungsregeln auf n_src, ob
         e mit ES'(e) = TRUE_SIGNALED bzw. ES'(e) = FALSE_SIGNALED
         markiert werden darf - falls ja, passe Kantenmarkierung entsprechend an
         if Markierung von e geändert then
            N_check := N_check ∪ {n_dest}
         endif
      done
      while N_check ≠ ∅ do
         Entnehme einen Knoten n aus N_check;
         if n ∈ N and NS(n) ∈ {ACTIVATED, NOT_ACTIVATED} then
            Überprüfe durch Anwendung der ADEPT-Ausführungsregeln auf n,
            ob dieser Knoten mit NS'(n) = ACTIVATED bzw. NS'(n) = SKIPPED
            markiert werden darf - falls ja, passe die Markierung entsprechend an
            Falls NS'(n) auf SKIPPED gesetzt wurde:
               E_check:= E_check ∪ succ(S', n)
         endif
      done
   until E_check = ∅ and N_check = ∅;
end
```

Beispiel 3.13 (Anpassung von Markierungen bei Knoteneinfügungen) Wir betrachten zuerst ein Beispiel für eine einfache Änderung, bei der zwei Aktivitäten X und Y, wie in Abb. 17 dargestellt, in einen WF-Ausführungsgraphen eingefügt werden sollen. Die betrachtete WF-Instanz ist wegen ihres aktuellen Zustands (NS(A) = `COMPLETED` und NS(B) = `ACTIVATED`) verträglich mit dem neuen Schema (vgl. Sätze 3.1 und 3.6) und kann entsprechend migriert werden.

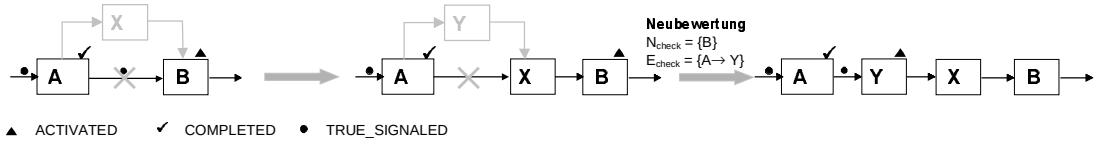


Abbildung 17: Anwendung von Algorithmus 1 und 2 bei additiven Änderungen

Die Anwendung von Algorithmus 1 liefert $N_{check} = \{\text{B}\}$ und $E_{check} = \{A \rightarrow Y\}$. (Nach Durchlaufen des 1. Teils von Algorithmus 1 erhält man zunächst $N_{check} = \{\text{B, X}\}$ und $E_{check} = \{A \rightarrow Y, A \rightarrow X\}$. Im 2. Teil werden dann noch X und $A \rightarrow X$ nach den angegebenen Regeln aus den Mengen N_{check} bzw. E_{check} entfernt.) Hiervon ausgehend bewertet Algorithmus 2 zunächst die Kante $A \rightarrow Y$ mit TRUE_SIGNALED (Knoten A ist bereits beendet), woraufhin Y ebenfalls zur Menge N_{check} der neu zu bewertenden Knoten hinzugefügt wird. Anschließend werden Y und B neu bewertet, was zur Aktivierung von Y (alle eingehenden Kanten sind mit `TRUE` signalisiert) und zur Deaktivierung von B (die eingehende Kante $X \rightarrow B$ wurde noch nicht signalisiert) führt.

Beispiel 3.14 (Anpassung von Markierungen bei ordnungsverändernden Operationen) Wir betrachten eine Schemaänderung bei der die bisherigen Anordnungsbeziehungen zwischen Knoten verändert werden. Im WF-Graphen aus Abb. 18a sollen die sequentiell auszuführenden Aktivitäten B und C nun parallel angeordnet werden (vgl. hierzu Beispiel 4.1 bzw. Abb. 15). Intern realisiert wird diese Änderung durch Hinzunahme der Kontrollkanten $B \rightarrow D$ und $A \rightarrow C$ sowie durch Entfernen der Kante $B \rightarrow C$. Algorithmus 1 liefert dementsprechend $N_{check} = \{C, D\}$ und $E_{check} = \{A \rightarrow C, B \rightarrow D\}$.

Gegeben seien nun die beiden Instanzen I_1 und I_2 von S (vgl. Abb. 18c). Sowohl I_1 als auch I_2 sind augrund ihrer aktuellen Zustandsmarkierungen verträglich mit dem neuen Schema S' (vgl. Satz 3.5) und können deshalb auf S' migriert werden. Bei der Migration von I_1 auf S' werden die Kanten $B \rightarrow D$ und $A \rightarrow C$ gemäß Algorithmus 2 mit `TRUE_SIGNALED` markiert. Sowohl für D als auch C erfolgt jedoch keine Anpassung der Knotenmarkierung, da die Bedingungen für die Ausführung dieser Knoten entweder noch nicht erfüllt sind (Knoten D) oder der betreffende Knoten bereits gestartet wurde (Knoten C). Bei der Migration von I_2 werden zunächst die Markierungen der Kanten $B \rightarrow D$ und $A \rightarrow C$ neu bewertet, wobei lediglich die Kante $A \rightarrow C$ mit `TRUE_SIGNALED` markiert werden kann. Die anschließende Bewertung des Knotens C ergibt dann, dass dieser mit `ACTIVATED` markiert und somit in Arbeitslisten gestartet werden kann. Dieses einfache Beispiel zeigt sehr gut, wie unterschiedlich Markierungsanpassungen für verschiedene WF-Instanzen desselben Typs ausfallen können. Es demonstriert zudem, wie

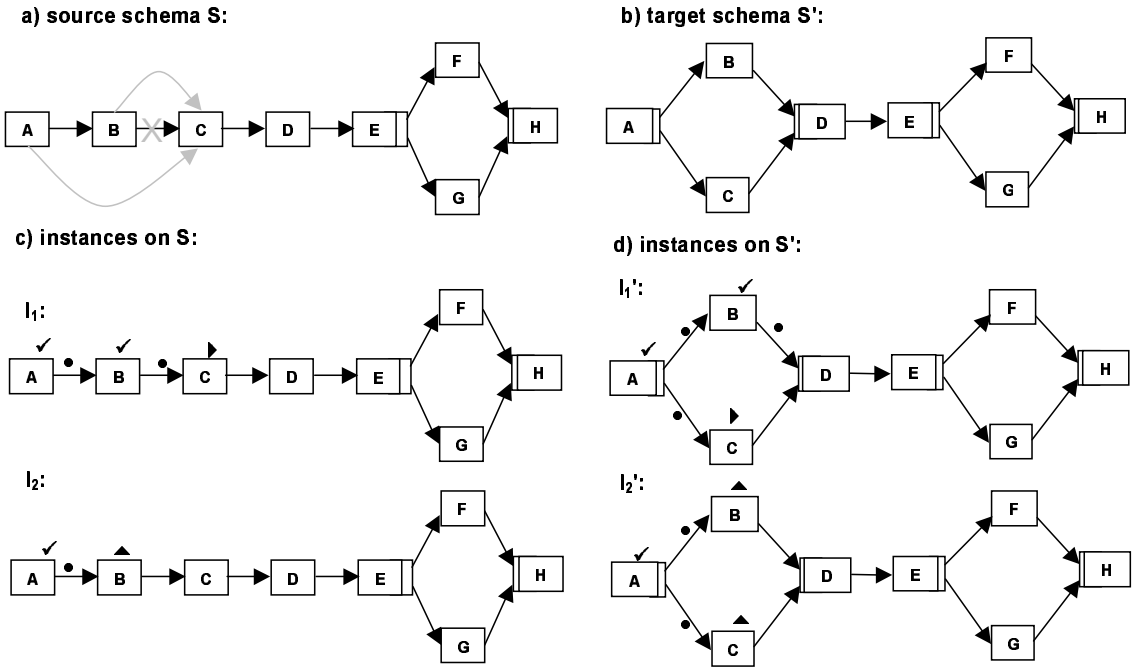


Abbildung 18: Ordnungsverändernde Operation auf WF-Schema in ADEPT („move C from current to position parallel to B")

die notwendigen Anpassungen automatisch und begrenzt auf wenige Knoten und Kanten erfolgen können. Letzeres ist gerade bei WF-Graphen mit vielen Knoten und Kanten von Vorteil. Insgesamt ist der vorgestellte Ansatz wesentlich effizienter und praktikabler als die eingangs skizzierten Lösungsansätze 1 und 2.

Aus Platzgründen verzichten wir auf die Darstellung weiterer Beispiele. Erwähnt sei jedoch, dass die Algorithmen 1 und 2 auch in Verbindung mit beliebig komplexen Änderungen funktionieren. Das trifft auch für die im Zusammenhang mit XOR-Verzweigungen und Schleifen diskutierten Änderungen zu, etwa in Bezug auf das Einfügen neuer Teilzweige in eine bestehende XOR-Verzweigung. Hier kann es auf Instanzebene sogar vorkommen, dass der neu hinzugefügte Zweig nach der Neubewertung der Zustandsmarkierungen vollständig „abgewählt" wird (d. h. Knoten mit `SKIPPED` und Kanten mit `FALSE_SIGNALED` bewertet sind).

Abschließend sei bemerkt, dass Algorithmus 2 auch wichtige Informationen für die Anpassung von Arbeitslisten liefert. So müssen für jede Aktivität des alten Schemas, die vor der Migration aktiviert war und deren Aktivierung bei der Neubewertung der Markierungen aufgehoben wurde, die entsprechenden Arbeitslisteneinträge wieder entfernt werden. Umgekehrt müssen für bisher nicht aktivierte bzw. neu hinzukommende Knoten, die nach der Neubewertung den Status `ACTIVATED` besitzen, entsprechende Arbeitslisteneinträge generiert werden. Wir verzichten an dieser Stelle auf eine Diskussion von Implementierungsaspekten in Bezug auf die Anpassung

von Arbeitslisten.

4.3 Umgang mit nicht verträglichen WF-Instanzen

Abschließend diskutieren wir Strategien für den Umgang mit nicht verträglichen WF-Instanzen. Sie sind in ihrer Ausführung zu weit fortgeschritten, um (aktuell) auf das neue WF-Schema migrieren zu können. Betrachtet man diese unverträglichen WF-Instanzen genauer, lassen sich diese in zwei Gruppen einteilen: Die erste Gruppe enthält diejenigen WF-Instanzen, die im Laufe ihrer regulären Ausführung nie mehr in einen Zustand gelangen können, indem sie verträglich mit dem geänderten WF-Schema sind. Die zweite Gruppe fasst diejenigen WF-Instanzen zusammen, welche zwar aufgrund ihres aktuellen Status nicht verträglich mit dem neuen WF-Schema sind, welche aber bei normalem Fortschreiten der WF-Kontrolle eventuell wieder in einen Zustand gelangen können, in dem Verträglichkeit mit dem neuen WF-Schema gegeben ist. Dieser Fall kann in Verbindung mit Schleifen auftreten. Wir diskutieren kurz Strategien für den adäquaten Umgang mit WF-Instanzen beider Gruppen.

Gruppe 1: WF-Instanzen, die in der Folge keinen Zustand mehr erreichen, in dem sie verträglich mit dem neuen WF-Schema sind
Die naheliegendste Strategie ist es, die betroffenen WF-Instanzen nach dem alten WF-Schema weiterlaufen zu lassen. Um eine solche Koexistenz von WF-Instanzen alter und neuer Form zu ermöglichen, müssen geeignete Versionierungskonzepte bereitgestellt werden (z. B. [29, 33]).

Eine Alternative besteht darin, diese unverträglichen WF-Instanzen durch partielles Rollback in ihrem Zustand so weit zurückzusetzen, dass sie anschließend verträglich mit dem geänderten WF-Schema sind (vgl. z. B. [12, 46]). ADEPT bietet ebenfalls solche Rollback-Mechanismen an. Sie gestatten es, die WF-Bearbeitung vor die Ausführung einer bestimmten Aktivität zurückzusetzen (vorausgesetzt, die zugeordneten Aktivitätenprogramme unterstützen dies durch Bereitstellung von Storno- oder Kompensationsaktionen). Dabei werden auch Rücksetzoperationen innerhalb paralleler Ausführungszweige oder Rücksprünge in frühere Iterationen einer Schleife unterstützt. Durch ihre Anwendung kann natürlich die Verträglichkeit einer WF-Instanz mit dem aus einer Änderung resultierenden WF-Schema immer hergestellt werden. Allerdings ist diese Vorgehensweise in der Praxis meist nicht anwendbar. (Man stelle sich einen weit fortgeschrittenen Behandlungsprozess eines Patienten vor, der zurückgesetzt werden soll, weil die bisherige Abwicklungsform modifiziert wird.)

Wir diskutieren abschließend eine weitere Variante für den Umgang mit WF-Instanzen der Gruppe 1 (siehe Abb. 19). Oftmals umfasst eine Änderungsdefinition Δ mehrere Einzeloperationen $op_1, \ldots, op_n$. Sind die Effekte von Δ nicht als Ganzes auf eine WF-Instanz propagierbar (und sind die Einzeloperationen $op_1, \ldots, op_n$ semantisch unabhängig, müssen also nicht zusammenhängend angewendet werden) kann ggf. nur die Propagierung einer Teilmenge der Änderungen vorgenommen werden. Die dabei entstehenden Abstufungen des WF-Schemas müssen wiederum durch ein geeignetes Versionskonzept (z. B. als Versionsbaum) verwaltet werden. Generell erscheint auch die Einbeziehung von Semantikaspekten in Verbindung mit der

Behandlung nicht migrierbarer WF-Instanzen lohnenswert.

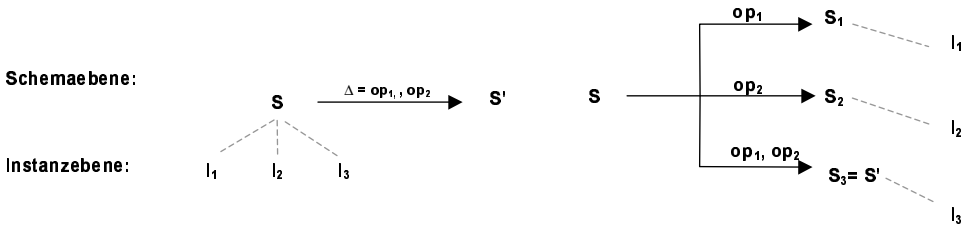


Abbildung 19: Strategie zur Behandlung nicht veträglicher WF-Instanzen

Gruppe 2: WF-Instanzen, die im Verlauf der weiteren Ausführung wieder in einen Zustand gelangen können, in dem sie verträglich mit dem geänderten WF-Schema sind
Die Situation, dass WF-Instanzen zum Zeitpunkt einer Änderung nicht verträglich mit dem geänderten WF-Schema sind, im Verlauf ihrer weiteren Ausführung jedoch wieder in einen verträglichen Zustand gelangen können, entsteht im Zusammenhang mit Änderungen auf Schleifen.

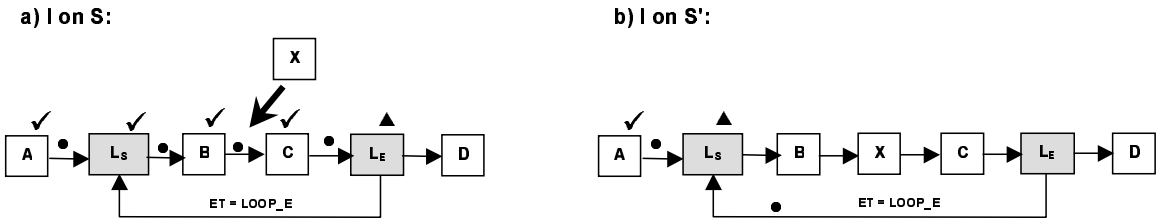


Abbildung 20: Propagierung von Änderungen bei Neubewertung von Schleifen

Die WF-Instanz in Abb. 20a ist zum Zeitpunkt der Änderungsdefinition nicht verträglich mit dem geänderten Schema S' (siehe Kapitel 3). Wird die Schleifenrücksprungkante jedoch bei Beendigung von L_E mit `TRUE_SIGNALED` bewertet, erfolgt eine Neubewertung aller Knoten des Schleifenkörpers mit `NOT_ACTIVATED` (siehe Abb.20b). Damit sind die Voraussetzungen für eine Migration von I auf S' (entsprechend dem Loop-Compliance-Kriterium) ,,verspätet" erfüllt.

Wie soll nun mit solchen WF-Instanzen umgegangen werden? In keinem Fall darf eine unverträgliche WF-Instanz sofort auf das neue WF-Schema migriert werden, da es ansonsten im ungünstigsten Fall zu den eingangs von Kapitel 3 genannten Problemen kommen kann. Prinzipiell könnten solche WF-Instanzen dauerhaft nach dem alten Schema weiterlaufen, unabhängig davon, ob sie in der Folge wieder in einen verträglichen Zustand gelangen oder nicht. Dies widerspricht jedoch dem Prinzip, dem Benutzer eine möglichst große Menge von migrierbaren WF-Instanzen zu bieten. Eine Variante, welche das wiederholte Auftreten von verträglichen Zuständen berücksichtigt, ist die verzögerte Migration (*delayed migration*). Ihr Prinzip soll anhand des folgenden Beispiels illustriert werden.

Beispiel 3.15: (Prinzip der verzögerten Migration) In Abb. 21 laufen zum Zeitpunkt $t = 1$ drei WF-Instanzen I_1, I_2 und I_3 auf WF-Schema S. Zum Zeitpunkt $t = 2$ findet eine Schemaänderung statt, die S auf S' transformiert und infolge derer die bisherigen WF-Instanzen auf S' migrieren sollen (Migration M_1). Die WF-Instanzen I_1, I_2 und I_3 lassen sich dann in drei Gruppen gemäß ihrer Verträglichkeit klassifizieren. I_1 sei zum Zeitpunkt $t = 2$ verträglich mit S' und kann folglich sofort auf S' migriert werden. I_3 sei zum Zeitpunkt $t = 2$ und zu keinem der folgenden Zeitpunkte verträglich mit S' und soll deshalb weiterhin Schema S verwenden. Die WF-Instanz I_2 ist zwar zum Zeitpunkt $t = 2$ nicht verträglich mit S'. Das System erkennt jedoch, dass I_2 sich noch innerhalb einer Schleifenausführung befindet und damit eventuell verzögert auf S' migriert werden könnte. Deshalb wird I_2 als ,,pending to migrate (M_1) to S' " eingestuft. In diesem Zustand wartet I_2 auf einen potentiellen Schleifenrücksprung. Das entsprechende Ereignis wird in ADEPT intern in einer *ECA-Tabelle* (**E**vent **C**ondition **A**ction) verwaltet, die bei der Ausführung berücksichtigt wird. Tritt eine Bewertung der Schleifenrücksprungkante mit `TRUE_SIGNALED` ein, wird die zugehörige Aktion, nämlich eine Migration von I_2 auf S', ausgelöst.

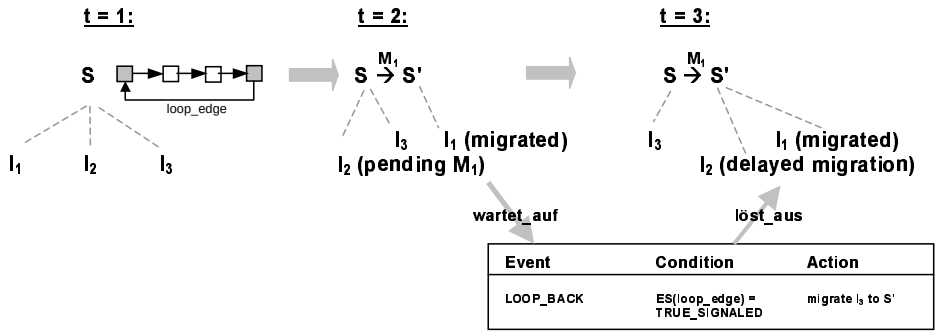


Abbildung 21: Prinzip der delayed migration

Ereignisse einer ECA-Tabelle können dabei UND- bzw. ODER-verknüpft sein. Beispielsweise kann eine verzögerte Migration ggf. bei verschiedenen Schleifenrücksprüngen stattfinden, wenn sich die Ausführung einer WF-Instanz zum Zeitpunkt der Schemaänderung innerhalb einer geschachtelten Schleife befindet. Wird die Schleifenrücksprungkante einer inneren Schleife zunächst nicht mit `TRUE_SIGNALED` bewertet, kann die Migration auch noch bei Signalisierung der Schleifenrücksprungkante der äußeren Schleife stattfinden.

Sehr interessant ist der Fall, dass sich eine WF-Instanz I mit Schema S im Zustand ,,pending" befindet (also auf eine verzögerte Migration auf ein Schema S' wartet) und in dieser Phase eine weitere Schemaänderung S' → S" stattfindet. Zunächst ist I von dieser Änderung noch nicht betroffen, da sie sich auf einer Schemaversion vor der von der Änderung betroffenen Version befindet. Kommt es jedoch später zur verzögerten Migration von I auf S', muss analysiert werden, ob die letzte Schemaänderung auch auf die verzögert migrierte WF-Instanz propagiert werden kann. Auch hier können wieder die diskutierten Fälle – eine Migration ist sofort, verzögert oder überhaupt nicht möglich – eintreten. Auf Details verzichten wir an dieser Stelle aus Platzgründen.

Eine Alternative zur verzögerten Migration von WF-Instanzen in Verbindung mit Schleifen stellt die Transformation des Quellschemas S auf ein WF-Schema S_{mod} dar, das bis zum Zeitpunkt der Änderung ausführungsäquivalent zu S und in der Folge ausführungsäquivalent zu S' ist. S_{mod} kann dabei durch Aufspaltung der von der Änderung betroffenen Schleifen in zwei Schleifenkonstrukte zerlegt werden, wobei der erste Schleifenkörper identisch zum ursprünglichen Schleifenkörper ist (mit fixer Anzahl an Iterationen) und der nachfolgend angefügte Schleifenkörper die durchgeführte Änderung berücksichtigt. Kritische Änderungen in der ,,pending"-Phase von WF-Instanzen werden hier zwar vermieden, jedoch muss noch geklärt werden, wie solche WF-Schemata effizient verwaltet werden sollen.

4.4 Weitere Aspekte

Selbst wenn eine WF-Instanz I verträglich mit einen geänderten Schema S' ist, ist eine Migration von I auf S' nicht immer sinnvoll bzw. wünschenswert. Hierfür kann es juristische, betriebliche oder sonstige Gründe geben. Beispielsweise kann es wünschenswert sein, nur solche WF-Instanzen auf die neue Abwicklung umzustellen (also auf ein entsprechend abgeändertes WF-Schema zu migrieren), die nach einem bestimmten Zeitpunkt gestartet worden sind. Um dem gerecht zu werden, kann der Modellierer in ADEPT angeben, für welche WF-Instanzen eine Migration gewünscht wird und für welche nicht. Dazu gibt es eine bei der Definition der Schemaänderung entsprechend formulierte Migrationsbedingung, wobei auf WF-Attribute (z. B. Startzeitpunkt oder prozessverantwortliche Stellen) und WF-Daten Bezug genommen werden kann. Wir verzichten an dieser Stelle auf Details.

5 Diskussion verwandter Ansätze

In diesem Abschnitt diskutieren wir verwandte Ansätze und grenzen sie gegenüber dem vorliegenden Beitrag ab. Dabei gehen wir zunächst auf Aspekte der Schemaevolution in Datenbanksystemen ein, bevor wir uns den prozessorientieren Informationssystemen (und hier im Speziellen den WfMS) zuwenden. Die Diskussion verwandter Ansätze aus dem WF-Bereich wird hierbei anhand der verwendeten WF-Beschreibungsformalismen untergliedert, da sie maßgeblichen Einfluss auf den jeweiligen Lösungsansatz haben. Wir gehen dabei auf Petrinetz-basierte, graphbasierte, objektorientierte und weitere Ansätze ein.

5.1 Schemaevolution in Datenbanksystemen

Das in diesem Beitrag behandelte Thema weist eine gewisse Analogie zu Schemaänderungen in Datenbanksystemen auf, erfordert letztlich aber einen andersartigen Lösungsansatz. Bei relationalen Datenbanksystemen etwa erfolgen solche Änderungen mittels Operationen wie Hinzufügen, Entfernen und Ändern von Tabellen (`CREATE/DROP/ALTER TABLE`). Anpassungen an Datenbankschemata werden z. B. im Zuge einer Schemaintegration im Umfeld verteilter oder föderierter Datenbanken notwendig [8, 14, 15]. Dieser Aspekt wird hier schon sehr lange untersucht und ist mittlerweile auch recht gut verstanden (siehe z. B. [5, 11, 14, 15, 36]). Der wesentliche Unterschied zwischen der Schemaevolution in Datenbanksystemen und WfMS besteht darin, dass in WfMS sich in Ausführung befindliche WF-Instanzen (in unterschiedlichen Zuständen) auf ein modifiziertes WF-Schema migriert werden sollen (ggf. begleitet durch weitere Anpassungen, etwa von WF-Zuständen und Arbeitslisten). Dagegen bleiben die Tupel eines Datenbanksystems während der Anpassung an ein verändertes Schema statisch. Deshalb ist einsichtig, weshalb sich im Rahmen der Schemaevolution in WfMS ganz neue Herausforderungen ergeben, die durch die Ergebnisse aus dem Datenbank-Bereich nicht abgedeckt sind.

5.2 Schemaevolution in WfMS

Ein viel beachteter Lösungsansatz zur Evolution von WF-Schemata wurde erstmals im Bereich der Petri-Netze präsentiert [19]. Für die WF-Modellierung und -Steuerung werden sog. Flussnetze verwendet, welche zur Laufzeit mehrere WF-Instanzen mit unterscheidbaren Marken kontrollieren. Eine Änderung des Schemas erfolgt formal durch eine Netztransformation, deren Semantik durch eine Graphsubstitution beschrieben wird. Dabei wird ein markiertes Subnetz (old change region) durch ein anderes markiertes Subnetz (new change region) ersetzt. Diese Schemaänderung erfolgt auf einer Kopie des Gesamtnetzes, welches für andere Strukturänderungen währenddessen gesperrt ist. Problematisch gestaltet sich die Migration der sich in Ausführung befindlichen WF-Instanzen. Der Modellierer muss hierbei für jede WF-Instanz die Anpassung der Markierungen manuell vornehmen (wie z. B. im System Chautauqua [20] vorgeschlagen), was bei großer Zahl von WF-Instanzen sehr komplex und schwer überschaubar wird (vgl. hierzu die Diskussion in Abschnitt 4.2). Die Praxistauglichkeit dieses Ansatzes ist

deshalb zumindest fragwürdig. Außerdem werden von den Autoren Fragestellungen wie effiziente Prüfung auf Verträglichkeit einzelner WF-Instanzen sowie Anpassungen von Datenflüssen oder Arbeitslisten vollständig ausgeklammert. Dabei basiert die Überprüfung der Korrektheit eines Modells nach seiner Änderung auf komplexen Erreichbarkeitsanalysen mit exponentiellem Aufwand für jede Instanz, womit der Aufwand besonders bei vielen Instanzen explodiert.

Auch neuere Ansätze aus dem Petri-Netz-Bereich beschäftigen sich mit Schema-Evolution. In [1] wird der von der Änderung betroffene Bereich (change region) innerhalb des Petri-Netzes ermittelt. Eine WF-Instanz kann nicht auf das modifizierte Petri-Netz migriert werden, solange sie sich in ihrer Ausführung in diesem Änderungsbereich befindet. Deshalb wird die Migration verzögert, bis die WF-Instanz den Änderungsbereich wieder verlassen hat. Neben der Migration von WF-Instanzen wird auch die Anpassung von WF-Instanzen nach der Propagation von Schemaänderungen thematisiert (vgl. [2, 4]), was als schwieriges Problem (dynamic change bug) erkannt wird. Als Lösung wird in [2] vorgeschlagen, dass der Modellierer eine Funktion zur Abbildung der Markierungen zwischen Quell- und Zielnetz angibt, die dann auf jede Instanz anwendbar ist. Dieser Ansatz ist für den Modellierer wie in [20] mit einem hohen Aufwand und einer hohen Komplexität verbunden, und ist deshalb in vielen Fällen nicht praktikabel. Das gilt im Besonderen für den Fall, dass durch das Petri-Netz nicht nur der Kontrollfluss, sondern auch die Datenflüsse zwischen Aktivitäten abgebildet werden. Abschließend sei erwähnt, dass zahlreiche Petri-Netz-Formalismen noch weitere Probleme, wie fehlende Verlaufsmarkierung, mangelnde Trennung von Kontroll- und Datenfluss und implizite Modellierung von Schleifen aufweisen.

Weitere Ansätze zur Evolution von WF-Schemata arbeiten auf Grundlage eines graphbasierten WF-Metamodells.

Ein erstes interessantes Beispiel hierfür bietet das Projekt WIDE [12, 13]. Dem Modellierer wird eine vollständige und minimale Menge von Basisoperationen angeboten, mit deren Hilfe ein gegebenes korrektes Schema S in ein beliebiges anderes korrektes Schema S' transformiert werden kann (statischer Fall). Zur Sicherstellung der Korrektheit bei der Migration der von S abgeleiteten, noch laufenden Instanzen auf das neue Schema S' (dynamischer Fall) wird in WIDE erstmals das sogenannte Compliance-Kriterium (vgl. Definition 3.1) formal definiert. Die Autoren machen jedoch keine Angaben, wie bzw. ob sich das Compliance-Kriterium überprüfen lässt und wie die WF-Instanzen nach ihrer Migration auf das geänderte WF-Schema anzupassen sind. Leider werden in diesem Ansatz auch forschrittliche Konstrukte (wie Schleifen) ausgeklammert, wodurch Untersuchungen auf den Zusammenhang zwischen Compliance und solchen Konstrukten fehlen. Nichtsdestotrotz wurde hier ein erster wesentlicher Beitrag zur Migration von WF-Instanzen auf geänderte WF-Schemata geschaffen.

Im Projekt TRAM [33] wird ein Ansatz zur Evolution von WF-Schemata vorgestellt, wobei ein ähnlicher Graph-Formalismus wie in FlowMark zur WF-Modellierung verwendet wird. Die Diskussion eines Versionierungskonzepts bildet den Schwerpunkt in TRAM. Bei Änderung einer Workflowtyp-Version wird eine neue Version abgeleitet und als Sohnknoten im Versionsbaum gespeichert. Dann wird versucht, die nach einer Workflowtyp-Version ω laufenden Instanzen unter Erhaltung des in [12] definierten Compliance-Kriteriums auf die abgeleitete Workflowtyp-

Version ω' zu migrieren. Dabei machen sich die Autoren über eine effiziente Überprüfung der zur neuen Workflowtyp-Version verträglichen WF-Instanzen Gedanken und schlagen die Definition einer sogenannten Migrationsbedingung vor. Diese wird jeder Operation als Bedingung mitgegeben, anhand derer für jede Instanz I überprüft werden kann, ob sie zu einem bestimmten Zeitpunkt auf die neue Workflowtyp-Version ω' migriert werden kann oder nicht. Leider wird diese Migrationsbedingung in [33] nicht formal definiert. Zusätzlich gibt es keine Hinweise zur Anpassung der Datenflüsse und die Behandlung fortschrittlicher Konstrukte.

Die aktuellsten Ergebnisse zur Evolution von WF-Schemata stammen aus dem Projekt BREEZE [46, 48], wobei in [47] auch eine entsprechende Architektur vorgestellt wird. Die vorgeschlagenen Konzepte orientieren sich sowohl bzgl. des Modells als auch bzgl. der vorgeschlagenen Basisoperationen an dem von uns definierten ADEPT-Modell bzw. den ADEPTflex-Änderungsoperationen [42, 44]. Als Korrektheitskriterium wird wiederum das Compliance-Kriterium zugrundegelegt. WF-Instanzen, die nicht verträglich mit dem geänderten WF-Schema S' sind, sollen über ein teilweises Rollback (dargestellt durch ein spezielles Konstrukt, den sog. Compliance-Graphen) in einen verträglichen Zustand zurückversetzt werden. Dies ist theoretisch zwar sehr interessant, kann aber nicht immer durchgeführt werden, da Aktivitäten nicht immer kompensiert werden können. Es existiert der Prototyp FlowMake mit Modellierungs- und Verifikationskomponente, der jedoch keine Laufzeitaspekte addressiert. In BREEZE wird weder diskutiert, wie die Möglichkeit zur Migration von WF-Instanzen (effizient) überprüft werden kann, noch wie fortschrittliche Konstrukte in diesem Kontext, etwa Schleifen, behandelt werden sollen.

Objektorientierte Ansätze finden sich in [29, 28] und [51, 52].

Im Projekt MOKASSIN [29, 28] werden zusätzlich noch Statecharts in das Modell eingebunden und ein Versionierungskonzept angeboten. Änderungen im laufenden Betrieb werden durch Kapselung von Änderungsprimitiven in den laufenden Instanzen realisiert, d. h. die Instanzen selbst sind für die Einhaltung der Konsistenz bei Änderungen „verantwortlich". Das Compliance-Kriterium wird als zu restriktiv eingestuft und eine Lösung über ein feingranulares Versionierungskonzept diskutiert. Die Konzepte sind zwar in MOKASSIN implementiert, Aussagen zur effizienten Überprüfbarkeit der Verträglichkeit von WF-Instanzen und die Diskussion fortschrittlicher Konzepte fehlen aber völlig. Abgesehen davon erschweren die in MOKASSIN angebotenen Konzepte zur Erweiterbarkeit der Workflow-Beschreibungssprache die effiziente und konsistente Propagierung von Schemaänderungen auf laufende WF-Instanzen.

Im Rahmen des Projekts WASA_2 wird ebenfalls ein Versionierungskonzept angeboten [51, 52]. Auch hier diskutiert der Autor Möglichkeiten zur effizienten Überprüfung der Compliance, indem er ein Mapping zwischen dem geänderten Schema und dem Subworkflow, der sich aus dem Status der zu untersuchenden Instanz ergibt, vorschlägt. Hierbei handelt es sich um einen der wenigen Ansätze, zu denen auch Erfahrungen aus der Implementierung der vorgestellten Konzepte vorliegen [53]. Betrachtungen zu Schleifen oder zur Anpassung von Datenflüssen wurden von der betreffenden Arbeitsgruppe unseres Wissens noch nicht veröffentlicht.

Einen interessanten Ansatz bietet [9]. Hier basiert das WF-Metamodell auf High-Level-Petri-

Netzen und Object-Behavior-Charts. Auch hier wird eine Erweiterung der Menge der migrierbaren Instanzen durch den Einsatz von Rollback-Aktivitäten vorgeschlagen. Aussagen zur effizienten Überprüfung, welche Instanzen auf ein geändertes Schema migrieren können fehlen in diesem Ansatz ebenso wie eine Implementierung der vorgestellten Konzepte.

In [18] werden zwar nur Schemaänderungen ohne Propagierung auf laufende WF-Instanzen betrachtet, der Ansatz ist aber im Hinblick auf die Überlegungen zur Erhaltung der Semantik bei Schemaänderungen interessant. Als WF-Metamodell werden hier Statecharts vorgeschlagen, über deren Äquivalenz die Semantik von modellierten Abläufen auch nach Änderungen erhalten bleiben soll (vgl. [21]).

Neben den erwähnten Arbeiten gibt es auch Bestrebungen, WF-Schemata durch Auswertung von WF-Logfiles automatisch abzuleiten (Process Mining). Solche Arbeiten (z. B. [24, 25]) bilden eine sehr wichtige Ergänzung im Hinblick auf ein umfassendes Änderungsmanagement in WfMS.

Tabelle 3 zeigt einige ausgewählte Ansätze und ihren Beitrag zum Thema Schemaevolution in WfMS.

	W. Aalst [1, 2, 3]	BREEZE [46, 47, 48]	Chautauqua [19, 20]	TRAM [33]	$WASA_2$ [51, 52]	WIDE [12, 13]	ADEPT [10, 42, 44]
Korrektheit/Konsistenz							
Kontrollflusssicht	+	+	+	+	+	+	+
Datenflusssicht	−	−	−−	−	−	−	+
Systematik bei Prüfung der Verträglichkeit[11]	+	−	−	−	−	−	+
Effiziente Prüfung auf Verträglichkeit	−	−	−	+	+	−	++
fortschrittl. Konstrukte							
Schleifen	−	−	−	−	−	−	++
hierarch. Workflows	−	−	−	−	∘	−	+
Automatische Anpassung	∘	−	−−	−	∘	−	++

Tabelle 3: Vergleich ausgewählter Ansätze

6 Zusammenfassung und Ausblick

In diesem Beitrag haben wir einen umfassenden und theoretisch fundierten Ansatz für die bei WF- Schemaänderungen notwendig werdenden Verträglichkeitsprüfungen und WF-Instanzmigrationen vorgestellt. Besonderes Augenmerk lag dabei auf Korrektheits- u. Konsistenzaspekten, Benutzerfreundlichkeit und Effizienz.

Den daraus resultierenden Anforderungen werden wir einerseits durch effiziente Prüfung auf Verträglichkeit mit dem neuen WF-Schema und andererseits durch automatische Anpassung von Zuständen bei der Migration verträglicher WF-Instanzen auf dieses neue WF-Schema gerecht. Wir haben teilweise eine formale Darstellung gewählt, um präzise Aussagen darüber treffen zu können, wann eine WF-Instanz auf ein neues Schema migriert werden darf und wann nicht. Aus diesen Aussagen geht auch hervor, welche Informationen hierfür konkret benötigt werden. Wir haben gezeigt, dass man für viele praktische Fälle den Umfang der für die Überprüfung benötigten Informationen auf ein vernünfiges Maß reduzieren kann, was für die Implementation eines solchen Konzeptes, besonders bei großer Zahl von WF-Instanzen, essentiell ist. Auch in Bezug auf die Migration verträglicher WF-Instanzen haben wir einen Ansatz vorgestellt, bei dem sich aufwendige Analysen, Neubewertungen oder reine Benutzerinteraktionen vermeiden lassen.

Wir haben uns in unseren Betrachtungen nicht nur auf Kontrollflussaspekte beschränkt, sondern haben uns auch mit Datenflussaspekten auseinandergesetzt. Dabei haben wir interessante Querbezüge und Zusammenhänge aufgezeigt. Erstmals wurden in dieser Arbeit auch fortschrittliche Konstrukte wie Schleifen und hierarchische Workflows in die Untersuchungen zur Verträglichkeitsprüfung bei WF-Instanzmigrationen einbezogen. Dabei wurden Varianten des in diesem Falle zu restriktiven herkömmlichen Verträglichkeitskriteriums vorgestellt und deren Korrektheit, Konsistenz und effiziente Überprüfbarkeit aufgezeigt.

Die gewonnenen Ergebnisse sind nicht nur spezifisch in ADEPT anwendbar, sondern können (mit kleineren Modifikationen) auch in vergleichbaren Ausführungsmodellen (siehe [38, 46, 47, 48, 51, 52]) für Verträglichkeitsprüfungen und WF-Instanzmigrationen verwendet werden. Insgesamt werden die vorgestellten Konzepte und Verfahren den in der Einleitung genannten Anforderungen gerecht. Wie ebenfalls bereits erwähnt, bestehen aber noch weitergehende Anforderungen. Ihre adäquate Handhabung und die verbundene Entwicklung entsprechender Lösungskonzepte werden Bestandteil unserer zukünftige Forschungsarbeiten darstellen. Zu diesen Anforderungen zählen auch die Einbeziehung von Semantik-Aspekten in Zusammenhang mit Verträglichkeitsprüfungen sowie die Änderung weiterer Bestandteile des WfMS. Insbesondere interessiert uns dabei deren Zusammenspiel und die konkrete Umsetzung in einem lauffähigen System. Nur so lässt sich letztlich überprüfen, ob die betreffende Konzepte in der Praxis und bei Anwendung auf komplexe Ablaufbeispiele adäquat umsetzbar sind. Wir beabsichtigen deshalb, die in dieser Arbeit entwickelten Konzepte prototypisch im ADEPT-WfMS [26] zu implementieren. Hierbei soll auch der Frage nachgegangen werden, ob bzw. inwieweit WF-Schemaänderungen

[11] wird z. B. im vorliegenden Beitrag durch Einteilung in verschiedene Graphklassen und Änderungsoperationen erreicht

auch auf solche WF-Instanzen propagiert werden können, deren Ausführungsgraph in Folge einer Ad-hoc-Änderung strukturell vom ursprünglichen WF-Schema abweicht. Bisherige Ansätze springen hier zu kurz, da sie Ad-hoc-Änderungen entweder gar nicht einbeziehen oder aber für diesen Fall die Propagierung von Schemaänderungen nicht mehr zulassen. Dies ist vor allem bei langlaufenden Prozessen i. A. nicht ausreichend.

Abschließend sei nochmals explizit darauf hingewiesen, dass es im Bereich Schemaevolution in WfMS noch viele offene und hochinteressante Fragestellungen gibt. Diese wurden weder von bisherigen Arbeiten im WF-Bereich noch von Arbeiten aus dem Bereich Schema-Evolution in Datenbanksystemen vollständig oder zufriedenstellend gelöst. Der vorliegende Beitrag dient der Vervollständigung der in der Literatur bisher vorgeschlagenen Konzepte für Schema-Evolution und bildet auch die Grundlage für unsere weiteren Arbeiten in diesem Bereich.

Literatur

[1] van der Aalst, W.M.P.: *Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change.* Information Systems Frontiers, 3(3):297-317, 2001.

[2] van der Aalst, W.M.P., Basten, T.: *Inheritance of Workflows: An Approach to Tackling Problems Related to Change.* Theoretical Computer Science, 270(1-2): 125-203, 2002

[3] van der Aalst, W.M.P., van Hee, K.: *Workflow Management.* MIT Press, 2002

[4] van der Aalst, W.M.P., Jablonski, S.: *Dealing with Workflow Change: Identification of Issues an Solutions.* International Journal of Computer Systems, Science and Engineering, 15(5): 267-276, (2000)

[5] Andany, J.; Leonard, M.; Palisser, C.: *Management of Schema Evolution in Databases.* Proc. Int'l Conf. on Very Large Databases, Barcelona, Sept. 1991, S. 161-170

[6] Bauer, T.; Dadam, P.: *Efficient Distributed Workflow Management Based on Variable Server Assignments.* Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE '00), Sweden, 2000, S. 94-109

[7] Beuter, T.; Dadam, P.; Schneider, P.: *The WEP Model: Adequate Workflow-Management for Engineering Processes.* Proc. European Concurrent Engineering Conference 1998, Erlangen-Nürnberg, April 1998

[8] Batini, C.; Lenzerini, M.; Navathe, S.B.: *A Comparative Analysis of Methodologies for Database Schema Integration.* ACM Computing Surveys, 18(4): 323-364 (1986)

[9] Bichler, P.; Preuner, G.; Schrefl, M.: *Workflow Transparency.* Proc. Advanced Information Systems Engineering (CAiSE '97), Barcelona Juni 1997, S. 423-436

[10] Bauer, T.; Reichert, M.; Dadam, P.: *Adaptives und verteiltes Workflow-Management.* Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, Oldenburg, März 2001, S. 47-66

[11] McBrien, P.; Poulovassilis, A.: *A Formalism of Semantic Schema Integration.* Information Systems, 23(5): 390-334 (1998)

[12] Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G.: *Workflow Evolution.* Data and Knowledge Engineering, 24(3): 211-238 (1998)

[13] Casati, F.; Ceri, S.; Paraboschi, S.; Pozzi, G.: *Specification and Implementation of Exceptions in Workflow Management Systems.* ACM Transactions on Database Systems, 24(3): 405-451 (1999)

[14] Conrad, S.: *Föderierte Datenbanksysteme - Konzepte der Datenintegration.* Springer-Verlag, 1997

[15] Dadam, P.: *Verteilte Datenbanken und Client/Server-Systeme.* Springer-Verlag, 1996

[16] Deiters, W.; Gruhn, V.: *The FUNSOFT Net Approach to software Process Management.* Int'l Journal of Software Engineering and Knowledge Engineering 4(2):229-256 (1994)

[17] Dadam, P.; Reichert, M.; Kuhn, K.: *Clinical Workflows - The Killer Application for Process-oriented Information Systems?.* Proc. 4th Int'l Conf. on Business Information Systems (BIS '00), Posen, 2000, S. 36-59

[18] Eder, J.; Liebhart, W.: *Workflow Transactions.* In: Handbook of the Workflow Management Coalition WfMC, S. 195-202 (1997)

[19] Ellis, C.A.; Keddara, K.; Rozenberg, G.: *Dynamic Change within Workflow Systems.* Proc. Int'l ACM Conf. on Organizational Computing Systems (COOCS '95), Milpitas, Kalifornien, August 1995, S. 10-21

[20] Ellis, C.A., Maltzahn, C.: *The Chautauqua Workflow System.* Proc. 30th Int'l Conf. on System Science, Maui, 1997

[21] Frank, H.; Eder, J.: *Equivalence Transformations on Statecharts.* Proc. 12th Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE '00), Chicago, Juli 2000, S. 150-158

[22] Gartner Group: *Why E-Business Craves Workflow Technology*, RAS Services, T-09-4929, Dez. 1999

[23] Hammer, M.; Champy, J.: *Reengineering the Corporation.* Harper Collins Publ. 1993

[24] Geppert, A., Tombros, D.: *Logging and Post-Mortem Analysis of Workflow Executions Based on Event Histories.* Proc. 3rd Int'l Workshop Rules in Database Systems (RIDS '97), Skövde, Schweden, Juni 1997

[25] Herbst, J., Karagiannis, D.: *Intergrating Machine Learning and Workflow Management to Support Acquisition and Adaption of Workflow Models.* Proc. 9th Int'l Workshop on Database and Expert Systems Applications, Wien, August 1998, S. 745-752

[26] Hensinger, C.; Reichert, M.; Bauer, T.; Strzeletz, T.; Dadam, P.: *ADEPTworkflow - Advanced Workflow Technology for the Efficient Support of Adaptive, Enterprise-wide Processes.* Proc. Software Demonstration Track (EDBT '00), Konstanz, März 2000, S. 29-30

[27] Jablonski, S.; Böhm, M.; Schulze, W.: *Workflow Management Entwicklung von Anwendungen und Systemen.* dpunkt-Verlag, 1999

[28] Joeris, G.: *Defining Flexible Workflow Execution Behaviors.* Enterprise-wide and Cross-enterprise Workflow-Management: Concepts, Systems, Applications, Proc. GI-Workshop (Informatik '99), Okt. 1999, S. 49-55

[29] Joeris, G.; Herzog, O.: *Managing Evolving Workflow Specifications.* Proc. Int'l Conf. on Cooperative Information Systems (CoopIS '98), New York City, August 1998, S. 310-321

[30] Kappel, G.: *Reorganizing Object Behavior by Behavior Composition - Coping with Evolving Requirements in Office Systems.* Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, Kaiserslautern, März 1991, S. 446-453

[31] Kamath, M.; Alonso, G.; Günthör, R.; Mohan, C.: *Providing High Availability in Very Large Workflow Management Systems.* Proc. 5th Int'l Conf. in Extending Database Technology, Avignon, März 1996, S. 427-442

[32] Klarmann, J: *A Comprehensive Support for Changes in Organizational Models of Workflow Management Systems.* Proc. 4th International Conference on Information Systems Modelling (ISM'01), Hradec nad Moravici, Tschechei, Mai 2001

[33] Kradolfer, M.; Geppert, A.: *Dynamic Workflow Schema Evolution Based on Workflow Type Versioning and Workflow Migration.* Proc. Int'l Conf. on Cooperative Information Systems (CoopIS '99), Edinburgh, September 1999, S. 104-114

[34] Leymann, F.; Altenhuber, W.: *Managing Business Processes as an Information Ressource.* IBM Systems Journal, 33(2): 326-348 (1994)

[35] Leymann, F.; Roller, D.: *Production Workflow - Concepts and Techniques.* Prentice Hall PTR, 2000.

[36] Larson, J.A.; Navathe, S.B.; Elmasri, R.: *A Theory of Attribute Equivalence in Databases with Application to Schema Integration.* IEEE Transactions on Software Engineering, 15(4): 449-463 (1989)

[37] Mann, J.E.: *Workflow and EAI.* EAI Journal, September/Oktober 1999, S. 49-53

[38] Martschat, U.: *Comparison and Evaluation of Production Workflow Management Systems.* Diplomarbeit, Universität Ulm, Fakultät für Informatik, 2001 (in German)

[39] Oberweis, A.: *Modellierung und Ausführung von Workflows mit Petri-Netzen.* Teubner Verlag, 1996

[40] Reichert, M.; Bauer, T.; Fries, T.; Dadam, P.: *Realisierung flexibler, unternehmensweiter Workflow-Anwendungen mit ADEPT.* Proc. Arbeitskonferenz ,,Elektronische Geschäftsprozesse", Klagenfurt, September 2001, S. 217-228

[41] Reichert, M.; Bauer, T.; Fries, T.; Dadam, P.: *Modellierung planbarer Abweichungen in Workflow-Management-Systemen.* Proc. Modellierung '02, Tutzing, März 2002, S.183-194

[42] Reichert, M.; Dadam, P.: *ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control.* Journal of Intelligent Information Systems, Kluwer Academic Publ., 10(2): 93-129 (1998)

[43] Reichert, M.; Dadam, P.: *Geschäftsprozessmodellierung und Workflow-Management-Konzepte, Systeme und deren Anwendung.* Industrie Management 16(3):23-27 (2000)

[44] Reichert, M.: *Dynamic Changes in Workflow Management Systems.* Dissertation, Universität Ulm, Fakultät für Informatik, 2000 (in German)

[45] Reichert, M.; Kuhn, K.; Dadam, P.: *Prozessreengineering und -automatisierung in klinischen Anwendungsumgebungen.* Proc. 41. Jahrestagung (GMDS '96), Bonn, Sept. 1996, S. 219-223

[46] Sadiq, S.; Orlowska, M.: *Dynamic Modification of Workflows.* Technical Report 442, Department of Computer Science and Electrical Engineering, University of Queensland, Brisbane, Australia, Oktober 1998

[47] Sadiq, S.; Orlowska, M.: *Architectural Considerations in Systems Supporting Dynamic Workflow Modification.* Proc. Workshop Software Architectures for Business Process Management (im Rahmen der CaiSE '99 Konferenz), Heidelberg, Juni 1999

[48] Sadiq, S.; Orlowska, M.: *On Capturing Exceptions in Workflow Process Models.* Proc. 4th Int'l Conf. on Business Information Systems (BIS '00), Posen, April 2000

[49] Schultheiß, B.; Meyer, J., Mangold, R. Zemmler, T.; Reichert, M.: *Prozessentwurf für den Ablauf einer stationären Chemotherapie - Ergebnisse einer an der Universitätsfrauenklinik Ulm durchgeführten Prozessanalyse.* Abt. DBIS, interner Bericht, November 1995

[50] Sheth, A.; Kochut, K.: *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordniation an Collaboration Systems.* Proc. NATO Advanced Study Institute on Workflow Management Systems and Interoperability, Instanbul, August 1997, S. 12-21

[51] Weske, M.: *Flexible Modeling and Execution of Workflow Activities.* Proc. 31st Int'l Conf. on System Sciences, Hawaii, 1998, S. 713-722

[52] Weske, M.: *Adaptive Workflows based on Flexible Assignment of Workflow Schemes and Workflow Instances.* Enterprise-wide and Cross-enterprise Workflow-Management: Concepts, Systems, Applications, GI-Workshop (Informatik '99), Oktober 1999, S. 42-48

[53] Weske, M.; Hündling, J.; Kuropka, D.; Schuschel, H.: *Objektorientierter Entwurf eines flexiblen Workflow-Management-Systems.* Informatik - Forschung und Entwicklung, 13(4): 179-195 (1998)

A Definitionen

S = (N, E, ...)	WF-Schema S mit Knotenmenge N und Kantenmenge E
$NT : N \rightarrow$ NodeTypes	Funktion, die jedem Knoten einen Typ aus NodeTypes zuordnet.
NodeTypes	Menge der Knotentypen NodeTypes = {STARTFLOW, ACTIVITY, NULL, STARTLOOP, ENDLOOP}
$E \subseteq N \times N \times$ EdgeTypes	E ist (endliche) Menge von gerichteten Kanten.
EdgeTypes	Menge der Kantentypen EdgeTypes = {CONTROL_E, SYNC_E, LOOP_E}
c_succ(S,n) (c_pred(S,n))	Menge der direkten Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten des Typs CONTROL_E in WF-Schema S
c_succ*(S,n) (c_pred*(S,n))	Menge der direkten oder indirekten Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten des Typs CONTROL_E in WF-Schema S
succ(S,n) (pred(S,n))	Menge der direkten Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten und Sync-Kanten in WF-Schema S
succ*(S,n) (pred*(S,n))	Menge der direkten oder indirekten Nachfolgerknoten (Vorgängerknoten) von n bzgl. normaler Kontrollkanten und Sync-Kanten in WF-Schema S
(AND-Split, AND-Join)	parallele Verzweigung
(XOR-Split, XOR-Join)	bedingte Verzweigung
(STARTLOOP, ENDLOOP), LOOP_E	Schleifenblock mit Schleifenrücksprungkante
selcode	Selektionscode zur Auswahl eine Teilzweigs bei bedingten Verzweigungen
$NS : N \rightarrow$ NodeStates	Funtion, die Knoten einen Knotenstatus aus NodeStates zuordnet
NodeStates	Menge der Knotenstati NodeStates = {NOT_ACTIVATED, ACTIVATED, RUNNING, COMPLETED, SKIPPED}
$ES : E \rightarrow$ EdgeStates	Funktion, die Kanten einen Kantenstatur aus EdgeStates zuordnet
EdgeStates	Menge der Kantenstati EdgeStates = {TRUE_SIGNALED, FALSE_SIGNALED}

Tabelle 4: Wichtige Funktionen

B Beweise

B.1 Beweise zu Graphklasse 1

Für die Beweise der nachfolgenden Sätze formulieren wir zunächst folgende Lemmata:

Lemma B.1 *Sei S = (N, E, ...) das Schema eines WF-Graphen der Graphklasse 1 und I eine WF-Instanz auf S mit Ablaufhistorie $\mathcal{A} = e_1, \ldots, e_t$. Sei $w = i_0 \rightarrow i_1 \rightarrow \ldots \rightarrow i_k$ ein beliebiger Weg ($i_0, \ldots, i_k \in N$) mit N(i_k) = {RUNNNING, COMPLETED}. Dann gilt für I, dass alle Knoten auf diesem Weg beendet sein müssen und ihre Endeinträge in der Ablaufhistorie $\mathcal{A}$ vor dem Starteintrag von i_k stehen müssen. Für alle $\mu < k$ gilt also:*

(1) NS(i_μ) = COMPLETED

(2) der Endeintrag von i_μ wurde in $\mathcal{A}$ vor dem Starteintrag von i_k geschrieben.

Beweis zu Lemma B.1:

Wir beweisen die Aussage des Lemmas B.1 durch vollständige Induktion bzgl. der Länge k des Weges w.

k = 1 (Induktionsanfang):

w = $i_0 \rightarrow i_1$, NS(i_1) $\in$ {`RUNNING, COMPLETED`}

Notwendige Voraussetzung für den Start der Aktivität i_1 war, dass alle ihre direkten Vorgänger beendet sind. Demzufolge muss für alle diese Aktivitäten, insbesondere auch für i_0, in der Ablaufhistorie ein entsprechender Endeintrag vorliegen, der vor dem Starteintrag von j geschrieben wurde.

k → k+1 (Induktionsschritt):

w = $i_0 \rightarrow \ldots \rightarrow i_{k+1} \wedge$ NS(i_{k+1}) $\in$ {`RUNNING, COMPLETED`}

Nach Induktionsvoraussetzung gelten (1) und (2) für w' = $i_0 \rightarrow \ldots \rightarrow i_k$. Voraussetzung für den Start von i_{k+1} war, dass alle direkten Vorgänger beendet waren. Insbesondere war i_k, als direkter Vorgänger von i_{k+1}, vor dem Start von i_{k+1} beendet. Dementsprechend muss der Historieneintrag zum Ende von i_k vor dem Starteintrag von i_{k+1} stehen und NS(i_k) = `COMPLETED` gelten. (*)

Noch zu betrachten ist w' = $i_0 \rightarrow \ldots \rightarrow i_k$. Wegen NS($i_k$) = `COMPLETED` gelten gemäß Induktionsvoraussetzung die Aussagen (1) und (2) $\forall \mu < k$. (**)

Mit (*) und (**) folgt unmittelbar die Aussage für k+1. □

Lemma B.2 *Sei S = (N, E, ...) das Schema eines WF-Graphen der Graphklasse 1 und I eine WF-Instanz auf S mit Ablaufhistorie $\mathcal{A} = e_1, \ldots, e_t$. Dann gilt: Die Einträge eines Knotens i werden in der Ablaufhistorie $\mathcal{A}$ stets vor den Einträgen eines Knotens j geschrieben, wenn i Vorgänger von j im Ausführungsgraphen ist. Formal:*

$\forall i, j \in N$ *mit* $j \in succ^*(S, i) \wedge NS(j) \in$ {`RUNNING, COMPLETED`} $\Rightarrow$

$\exists\ e_x, e_y \in \mathcal{A}$ *mit* $e_x =$ `END(i)` $\wedge\ e_y =$ `START(j)` $\wedge\ x < y$

Beweis zu Lemma B.2:

Wegen j $\in$ succ*(S,i) existiert ein Weg w = i = $i_0 \rightarrow \ldots \rightarrow i_k = j$. Mit NS(j) $\in$ {`RUNNING, COMPLETED`} folgt die Aussage von Lemma B.2 unmittelbar durch Anwendung von Lemma B.1.

Satz 3.1 (Einfügen von Aktivitätenknoten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde nun durch Einfügen eines Aktivitätenknotens n_{insert} (inklusive normaler Kontrollkanten zwecks Einbindung in den WF-Kontext) in ein wiederum korrektes WF-Schema S' der Graphklasse 1 transformiert. Dann gilt:*

I ist verträglich mit S' $\Leftrightarrow$

$\forall n \in succ(S', n_{insert}) : NS(n) \in$ {`NOT_ACTIVATED`, `ACTIVATED`}

Beweis zu Satz 3.1:

„⇒" zu zeigen:

I ist verträglich mit S' ⇒

$\forall n \in succ(S', n_{insert}) : NS(n) \in \{\texttt{NOT\_ACTIVATED}, \texttt{ACTIVATED}\}$

Sei $\mathcal{A}$ die Ablaufhistorie von I auf S. Wenn I verträglich mit S' ist, können wir I auf S' migrieren und gemäß S' ausführen. Deswegen besitzt I unmittelbar nach der Migration auf S' mit $\mathcal{A}$ die selbe Ablaufhistorie wie bisher. Insbesondere enthält $\mathcal{A}$ noch keinen Eintrag zu dem neu eingefügten Knoten n_{insert}. Wir beweisen nun obige Aussage durch Widerspruch.

Annahme:

$\exists\ n^* \in$ succ(S',n_{insert})[12] mit NS(n^*) $\notin$ {NOT_ACTIVATED, ACTIVATED}

$\equiv \exists\ n^* \in$ succ(S',n_{insert}) mitNS(n^*) $\in$ {RUNNING, COMPLETED}

$\Rightarrow \exists\ e_i \in \mathcal{A}$ mit $e_i =$ START(n^*)

Da n* ∈ succ(S', n_{insert}) gilt, müsste bei Verträglichkeit von I mit S' gemäß Lemma B.2 gelten: $\exists e_j \in \mathcal{A}$ mit $e_j =$ END(n_{insert}) $\wedge j < i$. Da eine neu eingefügte Kante jedoch eine der beiden Markierungen {NOT_ACTIVATED, ACTIVATED} beitzen muss, kann dies nicht der Fall sein. Folglich ist I nicht verträglich mit S'. □

„⇐" zu zeigen:

$\forall n \in succ(S', n_{insert}) : NS(n)(= NS'(n)) \in \{\texttt{NOT\_ACTIVATED}, \texttt{ACTIVATED}\} \Rightarrow$

I ist verträglich mit S'

Es gilt:

$\forall$ n ∈ $succ(S', n_{insert}) : NS(n) \in \{\texttt{NOT\_ACTIVATED}, \texttt{ACTIVATED}\}$

$\Rightarrow \forall$ n ∈ succ(S',n_{insert}): $\nexists\ e_i \in \mathcal{A}$ mit $e_i \in$ {START(n), END(n)}

$\Rightarrow \forall$ n ∈ succ*(S',n_{insert}): $\nexists\ e_i \in \mathcal{A}$ mit $e_i \in$ {START(n), END(n)} (*),

d. h. für keinen direkten oder indirekten Nachfolger von n_{insert} existiert ein Eintrag in der Ablaufhistorie $\mathcal{A}$. Ferner gilt für den neu eingefügten Knoten n_{insert}, dass er nach Neubewertung der Markierungen von i entweder noch nicht aktiviert oder aktiviert ist. Formal:

NS'(n_{insert}) ∈ {NOT_ACTIVATED, ACTIVATED}

$\Rightarrow \nexists\ e_i \in \mathcal{A}$ mit $e_i \in$ {START(n_{insert}), END(n_{insert})} (**)

Hiervon ausgehend zeigen wir nun, dass I verträglich mit S' ist, d. h. dass $e_0, \ldots, e_t$ in der angegebenen Reihenfolge auf S' anwendbar sind und hieraus eine konsistente Markierung M'_t resultiert. Formal: $M'_0[e_0 >, \ldots, [e_t > M'_t$

[12] n^* ist trivialerweise in S enthalten

Sei N_{rel} die Menge aller Knoten aus S', die vor n_{insert} ausgeführt werden können (zur Illustration siehe Abb. 22. Dies sind alle diejenigen Knoten, die entweder vor oder parallel zu n_{insert} liegen. Formal:

N_{rel} := pred*(S', n_{insert}) ∪ {n ∈ N' | n ∉ pred* (S', n_{insert}) ∧ n ∉ succ*(S', n_{insert})}

Wegen (*) und (**) folgt dann für Knoten n ∈ N:

∃ $e_i \in \mathcal{A}$ mit e_i = `START(n)` ∨ e_i = `END(n)` ⇒ n ∈ $N_{rel} \subseteq$ N

Alle bisherigen Ereignisse in der Ablaufhistorie $\mathcal{A}$ beziehen sich also auf Knoten, die bezogen auf S' vor n_{insert} ausgeführt werden können. Da sich der durch N_{rel} induzierte Teilgraph von S (siehe Abb. 22) nicht verändert hat, bleiben folglich $e_1, \ldots, e_t$ in der angegeben Reihenfolge auf diesem Teilgraphen und damit auf S' anwendbar. □

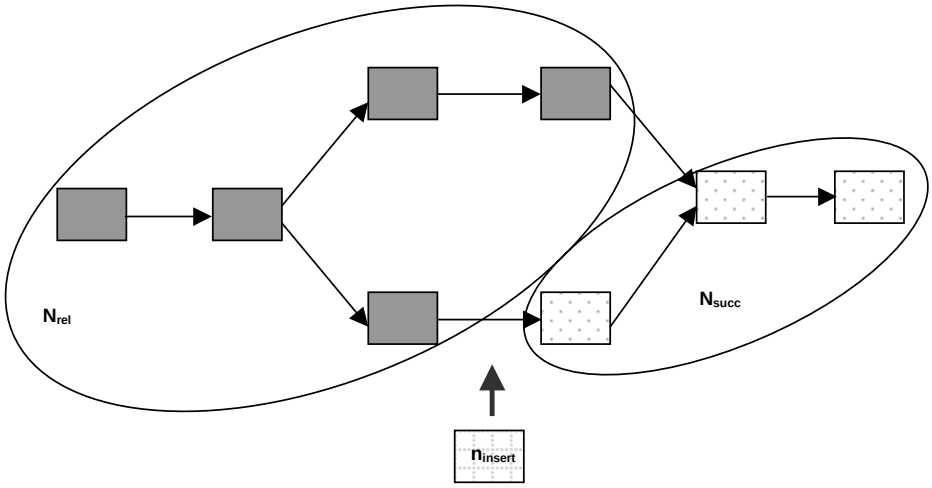


Abbildung 22: Teilmengen eines WF-Graphen bezogen auf n_{insert}

Satz 3.2 (Einfügen von Kontrollabhängigkeiten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch Hinzufügen einer Kontrollkante $n_{src} \rightarrow n_{dest}$ ($n_{src}, n_{dest} \in N$) auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert . Dann gilt:*

I ist verträglich mit S' ⇔

NS(n_{dest}) ∈ {`NOT_ACTIVATED, ACTIVATED`}

∨ *(NS(n_{dest})* ∈ {`RUNNING, COMPLETED`} ∧

NS(n_{src}) = `COMPLETED` ∧ *((∃ e_i = END(n_{src}), e_j = START(n_{dest})* ∈ $\mathcal{A}$*)* ∧ $i < j$*))*

Beweis zu Satz 3.2:

Vorab definieren wir:

$A_1 \equiv \mathrm{NS}(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED}\}$

$A_2 \equiv \mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) = \texttt{COMPLETED} \wedge$

$(\exists\ e_i, e_j \in \mathcal{A} : i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest}))$

„⇒" zu zeigen:

I verträglich mit S' $\Rightarrow A_1 \vee A_2$

Wir beweisen nun diese Aussage durch Widerspruch, d. h. wir zeigen:

$\neg(A_1 \vee A_2) \Rightarrow$ I nicht verträglich mit S'.

Annahme: $\neg\ (A_1 \vee A_2) \equiv \neg\ A_1 \wedge \neg A_2$

$\neg A_1 \equiv \mathrm{NS}(n_{dest}) \notin \{\texttt{NOT\_ACTIVATED, ACTIVATED}\}$

$\equiv \mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\}$ (*)

$\neg A_2 \equiv \mathrm{NS}(n_{dest}) \notin \{\texttt{RUNNING, COMPLETED}\} \vee \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED}$

$\vee\ (\neg\ (\exists\ e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest})))$

$\equiv \mathrm{NS}(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED}\}$

$\vee\ \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED}$

$\vee\ (\nexists\ e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest})))$ (**)

Mit (*) und (**) erhalten wir:

$\neg\ (A_1 \vee A_2) \equiv \neg\ A_1 \wedge \neg A_2$

$\equiv \mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge$

$(\mathrm{NS}(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED}\} \vee \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED} \vee$

$(\nexists\ e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest}))$

$\equiv \mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED} \vee$

$(\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge$

$(\nexists\ e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest})))$

Wir betrachten zunächst den ersten Teilausdruck uznd zeigen , dass I bei Gültigkeit dieses Ausdrucks nicht verträglich mit S' sein kann:

$\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED}$

Dann wird der Endeintrag von n_{src} in der Ablaufhistorie $\mathcal{A}$ in jedem Fall nach dem Starteintrag von n_{dest} geschrieben. Wegen $n_{src} \in \mathrm{pred}(S', n_{dest})$ kann dieser Fall für Instanzen von S' niemals

auftreten (vgl. Lemma B.2), d. h. I kann nicht verträglich mit S' sein.

Wir zeigen dasselbe nun für den zweiten Teilausdruck:

$(\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge$

$(\nexists\ e_i, e_j \in \mathcal{A}\text{: } i < j \ \ \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest})))$

$\Rightarrow \exists\ e_l \in \mathcal{A}\text{: } e_l = \texttt{START}(n_{dest}) \wedge$

$(\nexists e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest}))$

$\Rightarrow\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(n_{src}) \wedge \exists\ e_l \in \mathcal{A}\text{: } e_l = \texttt{START}(n_{dest}) \vee$

$(\exists e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_k = \texttt{START}(n_{dest}))$

In beiden Fällen wird der Starteintrag von n_{dest} vor dem Endeintrag von n_{src} geschrieben. Wegen $n_{src} \in \mathrm{pred}(\mathrm{S'}, n_{dest})$ kann dieser Fall für Instanzen von S' niemals auftreten (vgl. Lemma B.2), d. h. I kann nicht verträglich mit S' sein. □

„⇐" zu zeigen:

$A_1 \vee A_2 \Rightarrow$ I verträglich mit S'

$A_1 \equiv \mathrm{NS}(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED}\}$

Da I Instanz von S ist, folgt mit Lemma B.2:

$\forall e_i \in \mathcal{A}\text{: } e_i \in \{\texttt{START(n), END(n)}\}$ mit $\mathrm{n} \in N_{rel} := \mathrm{N}\neg\ \mathrm{succ}^*(\mathrm{S}, n_{dest}) \cup \{n_{dest}\}$

d. h. alle bisherigen Historeinträge wurden von Knoten erzeugt, die entweder Vorgänger von n_{dest} sind oder die parallel dazu liegen und deshalb prinzipiell auch vor n_{dest} ausgeführt werden können.

Da alle Knoten aus N_{rel} auch im neuen WF-Schema S' prinzipiell vor n_{dest} ausführbar sind, folgt unmittelbar, dass I verträglich mit S' ist.

$A_2 \equiv \mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) = \texttt{COMPLETED}$

$\wedge\ (\exists\ e_i, e_j \in \mathcal{A} : i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest})) \wedge \mathrm{NS}(n_{src}) = \texttt{COMPLETED}$

$\Rightarrow$ Die Ausführungshistorie $\mathcal{A}$ ist auch auf S' erzeugbar, da der Endeintrag zum Knoten n_{src} in der Ablaufhistorie $\mathcal{A}$ vor dem Starteintrag von n_{dest} liegt, d. h. n_{src} wurde vor n_{dest} ausgeführt. Da sich ansonsten an der Anordnungsbeziehung der Knoten n_{src} und n_{dest} nichts geändert hat, folgt daraus unmittelbar Compliance. □

Satz 3.3 (Löschen von Aktivitätenknoten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch Löschen eines Aktivitätenknotens n_{delete} inklusive Kontextkanten unter Erhalt der Anordnungsbeziehung der anderen Aktivitäten auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert . Dann gilt:*

I ist verträglich mit S' ⇔

NS(n_{delete}) ∈ {NOT_ACTIVATED, ACTIVATED}

Beweis zu Satz 3.3:

Sei $\mathcal{A}$ die Ablaufhistorie von I.

„⇒" zu zeigen:

I verträglich mit S' ⇒

NS(n_{delete}) ∈ {NOT_ACTIVATED, ACTIVATED}

Wir beweisen diese Aussage durch Widerspruch, d. h. wir zeigen

NS(n_{delete}) ∉ {NOT_ACTIVATED, ACTIVATED} ⇒ I nicht verträglich mit S'

Annahme:

NS(n_{delete}) ∉ {NOT_ACTIVATED, ACTIVATED}

≡ NS(n_{delete}) ∈ {RUNNING, COMPLETED}

⇒ ∃ $e_i \in \mathcal{A}$ mit e_i = START(n_{delete})

Wegen $n_{delete} \notin$ N' gilt für jede WF-Instanz auf S' mit Ablaufhistorie $\mathcal{B}$:

∄ $e_k \in \mathcal{B}$ mit e_k = START(n_{delete})

⇒ $\mathcal{A}$ ist nicht auf S' erzeugbar

I kann somit nicht verträglich mit S' sein. □

„⇐" zu zeigen:

NS(n_{delete}) ∈ {NOT_ACTIVATED, ACTIVATED} ⇒ I veträglich mit S'

NS(n_{delete}) ∈ {NOT_ACTIVATED, ACTIVATED}

⇒ ∀ n ∈ succ*(S, n_{delete}) ∪ {n_{delete}}: NS(n) ∈ {NOT_ACTIVATED, ACTIVATED}

Da I eine Instanz von S ist, folgt mit Lemma B.2:

∀ n ∈ succ*(S, n_{delete}) ∪ {n_{delete}}: ∄ $e_i \in \mathcal{A}$ mit e_i ∈ {START(n), END(n)}

⇒ ∀$e_i \in \mathcal{A}$ mit e_i ∈ {START(n), END(n)}: n ∉ (succ*(S, n_{delete}) ∪ {n_{delete}})

⇒ ∀$e_i \in \mathcal{A}$ mit e_i ∈ {START(n), END(n)}: n ∈ (pred*(S, n_{delete}) ∪

{n ∈ N | n ∉ pred*(S, n_{delete}) ∧ n ∉ succ*(S, n_{delete})} =: N_{rel}

Alle Ereignisse beziehen sich also auf Knoten, die bezogen auf S vor n_{delete} ausgeführt werden können. Da sich der durch M_{rel} induzierte Teilgraph von S (siehe Abb. 22 nicht verändert hat, sind folglich $e_1, \ldots, e_t$ in der angegeben Reihenfolge auf diesem Teilgraphen und damit auf S'

anwendbar. □

Satz 3.4 (Löschen von Kontrollabhängigkeiten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch Löschen einer Kontrollabhängigkeit zwischen den Aktivitäten n_{src} und n_{dest} ($n_{src}, n_{dest} \in$ N) auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert . Dann gilt:*

I ist verträglich mit S'

Beweis zu Satz 3.4:

Sei $\mathcal{A} = e_0, \ldots, e_t$ die Ablaufhistorie von I. Durch das Löschen der Kontrollkante $n_{src} \rightarrow n_{dest}$ ($n_{src}, n_{dest} \in$ N) sind bisherige Vorgängerknoten von n_{dest} nun parallel ausführbar zu diesem Knoten. Prinzipiell können diese parallelen Aktivitäten bei der Ausführung einer von S' abgeleiteten WF-Instanz auch vor Start von n_{dest} zum Ende kommen. Folglich ist die Ausführungshistorie einer WF-Instanz I von S auch auf dem neuen WF-Schema S' erzeugbar. □

Für den Beweis des nachfolgenden Satzes 3.5 formulieren wir folgendes Lemma:

Lemma B.3 *Sei S ein Schema der Graphklasse 1. S werde nun durch Anwendung einer Menge von Kanteneinfüge- und Kantenlöschoperationen $op_1, \ldots, op_k$ in ein wiederum korrektes Schema S' transformiert. Dann gilt:*

Es gibt eine Serialisierung $op_{i_1}, \ldots op_{i_k}$ von $op_1, \ldots, op_k$, so dass nach der Anwendung einzelner Operationen jeweils ein korrektes ,,Zwischenschema" resultiert. Formal:

$S = S_0\ [op_{i_1} > S_1 \ldots S_{k-1}\ [op_{i_k} > S_k = S'$

mit S_μ ist ein korrektes Schema der Graphklasse 1 ($\mu = 0, \ldots, k$).

Beweis zu Lemma B.3:

Wir beweisen die Aussage des Lemmas B.1 durch vollständige Induktion bzgl. der Anzahl k der verwendeten Elementaroperationen.

k = 1 (Induktionsanfang):

S = $S_0\ [op_{i_1} > S_1 =$ S'

Diese Aussage gilt trivialerweise, da sowohl S als auch S' gemäß der Voraussetzung von Lemma B.3 korrekt sind.

k → k+1 (Induktionsschritt):

S = $S_0\ [op_{i_1} > S_1 \ldots S_k\ [op_{i_{k+1}} > S_{k+1} =$ S'

Gegeben seien die Kantenoperationen $op_1, \ldots, op_{k+1}$, bei deren Anwendung auf S wieder ein korrektes Schema S' resultiert. Für die vorliegende Graphklasse bedeutet das insbesondere, dass die WF-Graphen von S bzw. S' zyklenfrei sind.

zu zeigen: $\exists i_1, \ldots, i_{k+1}$ mit $\{i_1, \ldots, i_{k+1}\} = \{1, \ldots, k+1\}$, so dass gilt:

$S = S_0\ [op_{i_1} > \ldots [op_{i_{k+1}} > S_{k+1} :=$ S' mit korrektem S_μ für alle $\mu = 0, \ldots, k+1$.

Bezüglich der Zusammensetzung der Operationen $op_1, \ldots, op_k$ lassen sich drei Fälle unterscheiden:

Fall 1: Alle Elementaroperationen sind additiv, d. h. bei ihrer Anwendung wird je eine Kontrollkante hinzugefügt.

Für diesen Fall folgt sofort, dass S_k korrekt ist: $S_{k+1} :=$ S' ist nach Voraussetzung des Lemmas korrekt und damit zyklenfrei. Im vorliegenden Fall ergibt sich S' aus S_k durch Hinzufügen einer Kontrollkante. Da S' zyklenfrei ist und S_k genau eine Kontrollkante weniger als S' besitzt, muss S_k trivialerweise auch zyklenfrei bzw. korrekt sein. Nach Induktionsannahme muss dies dann auch für $S_1, \ldots, S_{k-1}$ zutreffen.

Fall 2: Alle Elementaroperationen sind subtraktiv, d. h. bei ihrer Anwendung wird je eine Kontrollkante entfernt.

Hier kann ähnlich argumentiert werden, wie im Fall 1; allerdings bezogen auf S_1: S_1 ist korrekt, da $S_0 :=$ S korrekt ist und S_1 genau eine Kante weniger als S_0 besitzt. Damit müssen nach Induktionsannahme auch $S_2, \ldots, S_k$ korrekt sein.

Fall 3: Es existiert mindestens eine subtraktive und eine additive Elementaroperation.

Sei $op^* \in \{op_1, \ldots, op_{k+1}\}$ eine Kantenlöschoperation. Wähle dann $op_{i_1} = op^*$. Hieraus folgt sofort, dass S_1 korrekt ist. Ferner ergibt die Anwendung von $\{op_1, \ldots, op_{k+1}\} \neg \{op^*\}$ auf S_1 ein korrektes Schema $S_{k+1} :=$S'. Nach Induktionsannahme müssen damit auch die Zwischenschemata $S_2, \ldots, S_k$ korrekt sein. □

Satz 3.5 (Änderung der Anordnung von Aktivitätenknoten (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch eine ordnungsverändernde Operation auf ein korrektes WF-Schema S' der Graphklasse 1 transformiert. D. h. es wird - ausgehend von der für S definierten Kantenmenge E - eine Menge E_{add} von Kontrollkanten hinzugefügt und eine Menge E_{delete} von Kontrollkanten entfernt. Die Knotenmenge N selbst bleibt unverändert. Dann gilt:*

I ist verträglich mit S' $\Leftrightarrow$

$\forall\ e = n_{src} \rightarrow n_{dest} \in E_{add}$:

$NS(n_{dest}) \in \{$`NOT_ACTIVATED, ACTIVATED`$\}$

$\vee\ (NS(n_{dest}) \in \{$`RUNNING, COMPLETED`$\} \wedge NS(n_{src}) =$ `COMPLETED`

$\wedge\ ((\exists\ e_i = END(n_{src}),\ e_j = START(n_{dest})) \in \mathcal{A}) \wedge i < j))$

Beweis zu Satz 3.5:

Seien $op_1, \ldots, op_n$ Kanteneinfüge- bzw. Kantenlöschoperationen, deren Anwendung auf S zum Schema S' führt. Dementsprechend erzeugt jede Einfügeoperation einen Eintrag in E_{add} und jede Löschoperation einen Eintrag in E_{delete}.

Wir beweisen nun die Aussage von Satz 3.5 durch vollständige Induktion bzgl. der Anzahl n der angewandten Kantenoperationen.

n = 1 (Induktionsanfang):

S := $S_0[op_1 > S_1$:= S'

Fall 1: op_1 fügt eine Kante hinzu, d. h. $E_{add} = \{n_{src} \rightarrow n_{dest}\}$ mit $n_{src}, n_{dest} \in$ N, $E_{delete} = \emptyset$.

Hier folgt die Aussage von Satz 3.5 unmittelbar mit Anwendung von Satz 3.2.

Fall 2: op_1 entfernt eine Kante, d. h. $E_{add} = \emptyset$ und $E_{delete} = \{n_{src} \rightarrow n_{dest}\}$ mit $n_{src}, n_{dest} \in$ N.

Hier gilt die Aussage von Satz 3.5 trivialerweise wegen $E_{add} = \emptyset$ und Satz 3.4.

n $\rightarrow$ n+1 (Induktionsschritt):

Wegen Lemma B.3 gibt es $i_1, \ldots, i_{n+1}$, so dass gilt:

S := $S_0[op_{i_1} > S_1 \ldots S_n[op_{i_{n+1}} > S_{n+1}$:= S'

mit S_μ ist korrektes „Zwischen"-Schema der Graphklasse 1 (μ = 0, ..., n+1).

Sei $E_{add}^{(n)}$ die Menge aller durch $op_{i_1}, \ldots, op_{i_n}$ hinzugefügten Kanten. Mit der Induktionsannahme folgt:

I verträglich mit $S_n \Leftrightarrow$

$\forall$ e = $n_{src} \rightarrow n_{dest} \in E_{add}^{(n)}$:

NS(n_{dest}) $\in$ {`NOT_ACTIVATED, ACTIVATED`}

$\vee$ (NS(n_{dest}) $\in$ {`RUNNING, COMPLETED`} $\wedge$ NS(n_{src}) = `COMPLETED`

$\wedge$ (($\exists$ e_i = END(n_{src}), e_j = START(n_{dest})) $\in \mathcal{A}$) $\wedge$ $i < j$)) (*)

Fall 1: $op_{i_{n+1}}$ fügt eine Kante $n_{src} \rightarrow n_{dest}$ ein.

Dann folgt die Aussage von Satz 3.5 (bei Anwendung von $op_{i_1}, \ldots, op_{i_{n+1}}$) unmittelbar aus (*) und Satz 3.2 (mit $E_{add}^{(n+1)} = E_{add}^{(n)} \cup \{n_{src} \rightarrow n_{dest}\}$).

Fall 2: $op_{i_{n+1}}$ ist Kantenlöschoperation

Dann folgt die Aussage von Satz 3.5 (bei Anwendung von $op_{i_1}, \ldots, op_{i_{n+1}}$) unmittelbar aus (*) und Satz 3.4 ($E_{add}^{(n+1)} = E_{add}^{(n)}$). □

Satz 3.6 (Komplexe Änderungen (Graphklasse 1)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 1 und eine WF-Instanz I auf S. S werde durch*

die (serielle) Anwendung von Elementaroperationen $op_1, \ldots, op_n$ (Einfügen/Löschen von Aktivitätenknoten und Kontrollabhängigkeiten) in ein wiederum korrektes WF-Schema S' transformiert. Dann gilt:

I verträglich mit S' ⇔

Für jede Elementaroperation op_i sind ihre Voraussetzungen für die (isolierte) Anwendung auf S (gemäß der Sätze 3.1 bis 3.5) erfüllt.

Beweis zu Satz 3.6: Der Beweis kann vom Prinzip her ähnlich wie bei Satz 3.5 – dieser ist ein Spezialfall von Satz 3.6 – erfolgen. Dabei wird ausgenutzt, dass man für eine beliebige Menge von Elementaroperationen (Einfügen/Löschen von Aktivitätenknoten und Kontrollabhängigkeiten) $op_1, \ldots, op_n$ stets eine Serialisierung $op_{i_1}, \ldots, op_{i_n}$ findet, so dass bei Anwendung dieser Operationen aus S wieder ein korrektes Schema S' resultiert (S =: $S_0[op_{i_1} > S_1 \ldots S_{n-1}[op_{i_n} > S_n := S'$) und jedes Zwischenschema S_μ ebenfalls korrekt ist.

B.2 Beweise zu Graphklasse 2

Lemma B.4 *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. Dann gilt:*

NS(n)* ∈ {`RUNNING, COMPLETED`}, *n** ∈ *N*

⇒ ∀*n* ∈ *c_pred*(S, n*): NS(n)* ∈ {`COMPLETED, SKIPPED`}

Dieses Lemma besagt, dass wenn ein Knoten n* gestartet oder beendet wurde, alle seine direkten oder indirekten Vorgänger (bzgl. normaler Kontrollkante) entweder als beendet oder als nicht mehr ausführbar markiert sein müssen.

Beweis zu Lemma B.4:

Sei w = $i_0 \rightarrow \ldots \rightarrow i_k$ ein beliebiger Weg in S (mit normalen Kontrollkanten). Wir zeigen zunächst durch vollständige Induktion bzgl. der Länge k von w, dass folgende Aussage gilt:

w = $i_0 \rightarrow \ldots \rightarrow i_k$ mit NS(i_k) ∈ {`RUNNING, COMPLETED, SKIPPED`}, ⇒

NS(i_μ) ∈ {`COMPLETED, SKIPPED`} ∀$\mu = 0, \ldots, k-1$ (*)]

k = 1 (Induktionsanfang):

$w = i_0 \rightarrow i_1$

Fall 1: i_1 ist kein XOR-Join-Knoten

Notwendige Voraussetzung für den Start von i_1 war, dass alle direkten Vorgänger über normale Kontrollkanten den Status `COMPLETED` besitzen. Folglich muss auch i_0, als direkter Vorgänger von i_1, im Zustand `COMPLETED` sein.

Fall 2: i_1 ist ein XOR-Join-Knoten

Notwendige Voraussetzung für die Ausführung eines XOR-Join-Knotens ist, dass von den direkten Vorgängern (über normale Kontrollkanten) – sie repräsentieren die in diesem Knoten einmündenden Zweige – einer den Status `COMPLETED` und die anderen den Status `SKIPPED` besitzen. Folglich muss auch i_0 als direkter Vorgänger von i_1 (bzgl. normaler Kontrollkanten) im Zustand `COMPLETED` oder `SKIPPED` sein.

k → k+1 (Induktionsschritt):

$$w = i_0 \rightarrow \ldots \rightarrow i_{k+1}$$

Hier kann analog zur Beweisführung von Lemma B.1 vorgegangen werden, unter Anwendung der im obigen Induktionsanfang angestellten Überlegungen.

Damit folgt unmittelbar die Aussage von Lemma B.4. □

Satz 3.7 (Einfügen von Aktivitätenknoten und Sync-Kanten (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine additive Änderungsoperation op in ein korrektes WF-Schema S' der Graphklasse 2 transformiert.*

(a) op fügt einen neuen Knoten n_{insert} (inklusive normaler Kontrollkanten zwecks Einbindung in den WF-Kontext) in S ein. Dann gilt:

I ist verträglich mit S' ⇔

[∀ n ∈ c_succ(S', n_{insert}): NS(n) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}] ∨

[n_{insert} wird in einen abgewählten Teilzweig einer XOR-Verzweigung eingefügt]

(b) op fügt eine Kontrollkante $n_{src} \rightarrow n_{dest}$ (zwischen zwei nicht parallelen Knoten) in S ein. Dann gilt:

I verträglich mit S' ⇔

$NS(n_{dest})$ ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} ∨

[$NS(n_{src})$ = `COMPLETED` ∧ $NS(n_{dest})$ ∈ {`RUNNING, COMPLETED`}) mit

(∃e_i = `END`(n_{src}), e_j = `START`(n_{dest}) ∈ $\mathcal{A}$ ∧ $i < j$)]

(c) op fügt eine Sync-Kante $n_{src} \rightarrow n_{dest}$ ($n_{src}, n_{dest} \in N$) zwischen zwei parallel zueinander liegenden Knoten in S ein. Dann gilt:

I ist verträglich mit S' ⇔

$NS(n_{dest})$ ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} ∨

($NS(n_{src})$ = `COMPLETED` ∧ $NS(n_{dest})$ ∈ {`RUNNING, COMPLETED`}) mit

(∃e_i = `END`(n_{src}), e_j = `START`(n_{dest}) ∈ $\mathcal{A}$ ∧ $i < j$)) ∨

($NS(n_{src})$ = SKIPPED $\wedge$ $NS(n_{dest})$ $\in$ {RUNNING, COMPLETED}) mit

$\forall n \in N_{critical}$ mit $NS(n) \neq$ SKIPPED: $\exists e_i$ = START(n_{dest}), e_j = END(n) $\in \mathcal{A}$ mit $j < i$),

wobei $N_{critical}$ = (c_pred(n_{src}) $\neg$ c_pred*(n_{dest}))*

Beweis zu Satz 3.7:

(a) Einfügen von Knoten

Teil (a) von Satz 3.6 lässt sich formaler wie folgt darstellen:

I ist verträglich mit S' $\Leftrightarrow B_1 \vee B_2 \vee B_3$ mit

$B_1 \equiv [\forall n \in c\_succ(S', n_{insert}) : NS(n) \in \{\texttt{NOT\_ACTIVATED}, \texttt{ACTIVATED}, \texttt{SKIPPED}\}]$

$B_2 \equiv$ [$\forall$ n $\in$ c_pred(S', n_{insert}): NS(n) = SKIPPED]

$B_3 \equiv [n_{insert}$ wird in einen abgewählten, leeren Teilzweig eingefügt]

„⇒" zu zeigen:

I verträglich mit S' $\Rightarrow B_1 \vee B_2 \vee B_3$

Wir beweisen nun diese Aussage durch Widerspruch, d. h. wir zeigen:

$\neg(B_1 \vee B_2 \vee B_3) \Rightarrow$ I nicht verträglich mit S'

Annahme: $\neg(B_1 \vee B_2 \vee B_3) \equiv \neg B_1 \wedge \neg B_2 \wedge \neg B_3$

$\equiv$ [$\exists n^*$ $\in$ c_succ(S', n_{insert}): NS(n^*) $\in$ {RUNNING, COMPLETED}$\wedge$

[$\exists n^{**}$ $\in$ c_pred(S', n_{insert}): NS(n^{**}) $\neq$ SKIPPED]

[n_{insert} wird nicht in einen leeren, abgewähltem Teilzweig eingefügt]

Angenommen I ist verträglich mit S'. Dann folgt aus $\neg B_1$ mit Lemma B.4:
NS'(n_{insert}) $\in$ {COMPLETED, SKIPPED} (*).

Hieraus ergibt sich dann umittelbar, dass NS(n_{insert}) = SKIPPED gelten muss.

Begründung: Ein neu einzufügender Knoten besitzt nach Neubewertung der Graphmarkierung im Allgemeinen entweder den Status SKIPPED (Einfügen in einen abgewählten Teilzweig) oder aber einen der Zustände ACTIVATED bzw. NOT_ACTIVATED. Wegen (*) können die beiden letztgenannten Zustände nicht zutreffen.

Das bedeutet, dass n_{insert} in einen abgewählten Teilzweig einer XOR-Verzweigung (split, join) eingefügt wurde. Wegen $\neg B_3$ kann es sich hierbei jedoch nicht um einen leeren Teilzweig handeln. Dann muss entweder gelten, dass n_{insert} kein direkter Nachfolger von split ist – hieraus folgt unmittelbar $\forall$ n $\in$ c_pred(S', n_{insert}): NS(n) = SKIPPED oder aber dass n_{insert} kein direkter Vorgänger von join ist – hieraus folgt unmittelbar $\forall$ n $\in$ c_succ(S', n_{insert}): NS(n) = SKIPPED. Ersteres kann wegen $\neg B_2$ und letzteres wegen $\neg B_1$ nicht zutreffen. D. h. die Annahme, dass I

verträglich mit S' ist muss falsch sein. □

„⇐" zu zeigen:

Sei

$C_1 \equiv$ [$\forall$ n $\in$ c_succ(S', n_{insert}): NS(n) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}]

$C_2 \equiv$ [n_{insert} wird in einen abgewählten Teilzweig einer XOR-Verzweigung eingefügt]

zu zeigen: $C_1 \vee C_2 \Rightarrow$ I verträglich mit S'

Der Beweis der Aussage [$C_1 \Rightarrow$ I verträglich mit S'] kann analog zum Beweis von Satz 3.1 erfolgen.

Bezüglich des zweiten Teils der Aussage [$C_2 \Rightarrow$ I verträglich mit S'] trifft Folgendes zu: n_{insert} wird in einen abgewählten Teilzweig eingefügt, d. h. es muss gelten NS'(n_{insert}) = SKIPPED. Damit hat n_{insert} noch keinen Eintrag in die Ablaufhistorie $\mathcal{A}$ geschrieben. Daraus folgt unmittelbar, dass die bisherige Ablaufhistorie $\mathcal{A}$ auch auf S' erzeugbar ist. □

(b) Einfügen von Kontrollkanten

Hier kann beweistechnisch ähnlich vorgegangen werden wie im Beweis zu Satz 3.2. Wir verzichten auf die Darstellung und wenden uns dem interessanteren Fall (c) zu.

(c) Einfügen von Sync-Kanten

Vorab definieren wir:

$A_1 \equiv$ NS(n_{dest}) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}

$A_2 \equiv$ (NS(n_{src}) = COMPLETED $\wedge$ NS(n_{dest}) $\in$ {RUNNING, COMPLETED}) mit

($\exists e_i$ = END(n_{src}), e_j = START(n_{dest}) $\in \mathcal{A} \wedge i < j$))

$A_3 \equiv$ (NS(n_{src}) = SKIPPED $\wedge$ NS(n_{dest}) $\in$ {RUNNING, COMPLETED}) mit

$\forall n \in N_{critical}$ mit NS(n) $\neq$ SKIPPED: $\exists e_k$ = START(n_{dest}), e_l = END(n) $\in \mathcal{A}$ mit $l < k$),

wobei $N_{critical}$ = (c_pred*(n_{src}) $\neg$ c_pred*(n_{dest}))

Die Negationen von A_1, A_2, A_3 lauten dann folgendermaßen:

$\neg A_1 \equiv$ NS(n_{dest}) $\in$ {RUNNING, COMPLETED}

$\neg A_2 \equiv$ NS(n_{src}) $\neq$ COMPLETED $\vee$ NS(n_{dest}) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED} $\vee$

$\nexists e_i, e_j \in \mathcal{A}$: $i < j \wedge e_i$ = END(n_{src}), e_j = START(n_{dest})

$\neg A_3 \equiv$ NS(n_{src}) $\neq$ SKIPPED $\vee$ NS(n_{dest}) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED} $\vee$

$\exists n \in N_{critical}$ mit NS(n) $\neq$ SKIPPED: $\nexists e_k, e_l \in \mathcal{A}$: $l < k \wedge e_k$ = START(n_{dest}), e_l = END(n)

„⇒" zu zeigen:

I veträglich mit S' $\Rightarrow A_1 \vee A_2 \vee A_3$

Wir beweisen die obige Aussage durch Widerspruch.

Annahme: $\neg(A_1 \vee A_2 \vee A_3) \equiv \neg A_1 \wedge \neg A_2 \wedge \neg A_3 \equiv (\neg A_1 \wedge \neg A_2) \wedge \neg A_3$

$\equiv [(\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED}) \vee$

$(\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge$

$\nexists e_i, e_j \in \mathcal{A}\text{: } i < j \wedge e_i = \texttt{END}(n_{src}),\ e_j = \texttt{START}(n_{dest}))] \wedge \neg A_3$

$\equiv [(\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED}) \vee$

$((\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest})) \wedge$

$((\nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src})) \vee (\exists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(n_{src}) \wedge i > j)))] \wedge \neg A_3$

$\equiv [(\mathrm{NS}(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{COMPLETED}) \vee$

$((\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src})) \vee$

$(\exists e_i, e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ e_i = \texttt{END}(_{src}) \wedge i > j))] \wedge \neg A_3$

$\equiv [(\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src})) \vee$

$(\exists e_i, e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ e_i = \texttt{END}(_{src}) \wedge i > j)] \wedge \neg A_3$

$\equiv: (B_1 \vee B_2) \wedge \neg A_3$ $(\equiv (B_1 \wedge \neg A_3) \vee (B_2 \wedge \neg A_3))$

Wegen $n_{src} \in \mathrm{pred}(\text{S'}, n_{dest})$ kann für Instanzen von S' der Endeintrag von n_{src} niemals nach dem Starteintrag von n_{dest} in der Historie $\mathcal{A}$ stehen, d. h. B_2 und damit $(B_2 \wedge \neg A_3)$ können nicht zutreffen. Für diesen Fall also folgt, dass I nicht verträglich mit S' ist. Folglich muss $(B_1 \wedge \neg A_3)$ gelten.

$(B_1 \wedge \neg A_3)$

$\equiv [\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src})] \wedge$

$[\mathrm{NS}(n_{src}) \neq \texttt{SKIPPED} \vee \mathrm{NS}(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED, SKIPPED}\} \vee$

$\exists n \in N_{critical}, \mathrm{NS(n)} \neq \texttt{SKIPPED}\text{: } \nexists\ e_k, e_l \in \mathcal{A}\text{: } l < k \wedge e_k = \texttt{START}(n_{dest}),\ e_l = \texttt{END(n)}]$

$\equiv [\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src}) \wedge \mathrm{NS}(n_{src}) \neq \texttt{SKIPPED}] \vee$

$[(\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src})) \wedge$

$\mathrm{NS}(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED, SKIPPED}\}) \vee$

$(\exists e_j \in \mathcal{A}\text{: } e_j = \texttt{START}(n_{dest}),\ \nexists e_i \in \mathcal{A}\text{: } e_i = \texttt{END}(_{src}) \wedge$

$(\exists n \in N_{critical}, \mathrm{NS(n)} \neq \texttt{SKIPPED}\text{: } \nexists\ e_k, e_l \in \mathcal{A}\text{: } l < k \wedge e_k = \texttt{START}(n_{dest}),\ e_l = \texttt{END(n)}))]$

$\equiv: C_1 \vee C_2$

Aus C_1 folgt NS(n_{dest}) $\in$ {RUNNING, COMPLETED} $\wedge$
NS(n_{src}) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}. I kann für diesen Fall also nicht verträglich mit S' sein. Folglich muss C_2 gelten.

C_2

$\equiv$ [$\exists e_j \in \mathcal{A}$: e_j = START(n_{dest}), $\nexists e_i \in \mathcal{A}$: e_i = END($_{src}$)) $\wedge$

NS(n_{dest}) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}] $\vee$

[$\exists e_j \in \mathcal{A}$: e_j = START(n_{dest}), $\nexists e_i \in \mathcal{A}$: e_i = END($_{src}$) $\wedge$

($\exists n \in N_{critical}$, NS(n) $\neq$ SKIPPED: $\nexists\, e_k, e_l \in \mathcal{A}$: $l < k \wedge e_k$ = START(n_{dest}), e_l = END(n))]

$\equiv$ FALSE $\vee$ ($\exists e_j \in \mathcal{A}$: e_j = START(n_{dest}), $\nexists e_i \in \mathcal{A}$: e_i = END($_{src}$)) $\wedge$

($\exists n \in N_{critical}$, NS(n) $\neq$ SKIPPED $\wedge$

($\nexists e_l \in \mathcal{A}$: e_l = END(n) $\vee$ $\exists e_l \in \mathcal{A}$: e_l = END(n) $\wedge$ $j < l$))

$\equiv$ [($\exists e_j \in \mathcal{A}$: e_j = START(n_{dest}), $\nexists e_i \in \mathcal{A}$: e_i = END($_{src}$)) $\wedge$

($\exists n \in N_{critical}$, NS(n) $\neq$ SKIPPED $\wedge$ $\nexists e_l \in \mathcal{A}$: e_l = END(n))] $\vee$

[($\exists e_j \in \mathcal{A}$: e_j = START(n_{dest}), $\nexists e_i \in \mathcal{A}$: e_i = END($_{src}$)) $\wedge$

($\exists n \in N_{critical}$, NS(n) $\neq$ SKIPPED $\wedge$ $\exists e_l \in \mathcal{A}$: e_l = END(n) $\wedge$ $j < l$)]

$\equiv: D_1 \vee D_2$

Aus D_1 folgt, dass es einen Vorgängerknoten n $\in N_{critical}$ von n_{src} gibt, der noch nicht beendet ist, der aber auch nicht den Zustand SKIPPED besitzt (siehe Abb. 23). Bezogen auf S' ist dieser Knoten wegen der hier zusätzlichen Kante $n_{src} \rightarrow n_{dest}$ auch Vorgänger von n_{dest}. Folglich kann I nicht verträglich mit S' sein.

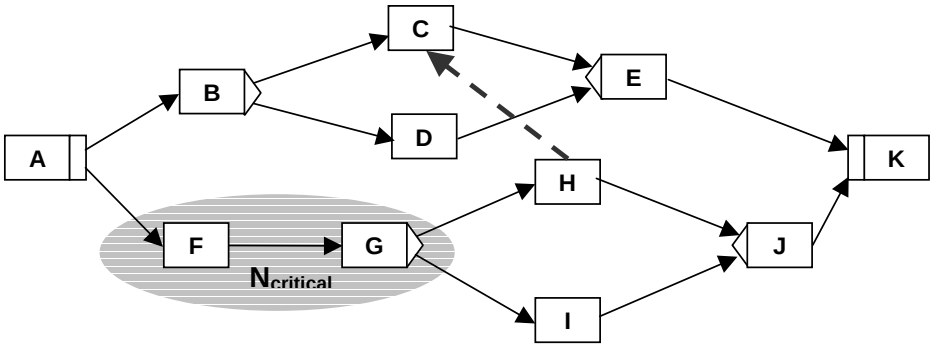


Abbildung 23: Graphbeispiel zu $N_{critical}$

Aus D_2 folgt, dass es einen Vorgängerknoten n $\in N_{critical}$ von n_{src} mit NS(n) = COMPLETED gibt, dessen Endeintrag in der Historie $\mathcal{A}$ nach dem Starteintrag von n_{dest} steht. Da n in S' aber

Vorgänger von n_{dest} ist, folgt, dass I nicht verträglich mit S' ist. □

„⇐" zu zeigen:

$A_1 \vee A_2 \vee A_3 \Rightarrow$ I verträglich mit S'

Aus A_1 folgt, dass $\mathcal{A}$ noch keinen Eintrag zu n_{dest} enthält. Deshalb kann A auch auf Grundlage von S' erzeugt werden, d. h. I ist verträglich mit S'. Dies gilt auch, falls A_2 zutrifft, da hier der Endeintrag von n_{src} in der Historie vor dem Starteintrag von n_{dest} liegt (vgl. Beweis zu Satz 3.2). Durch Hinzunahme von $n_{src} \rightarrow n_{dest}$ wird nicht nur n_{src} vor n_{dest} ausgeführt bzw. als nicht mehr ausführbar markiert, sondern möglicherweise auch weitere (Vorgänger-)knoten von n_{src}, die bisher parallel zu n_{dest} ausführbar waren. Konkret sind dies die Knoten der Menge $N_{critical}$ (vgl. Abb. 23). Nur wenn jeder dieser Knoten entweder mit `SKIPPED` markiert wurde oder aber seinen Endeintrag vor dem Starteintrag von n_{dest} in $\mathcal{A}$ geschrieben hat, ist $\mathcal{A}$ auch auf dem neuen Schema S' erzeugbar. Genau dies wird durch A_3 zugesichert. □

Satz 3.8 (Löschen von Aktivitätenknoten und Kanten (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine subtraktive Änderungsoperation op in ein korrektes WF-Schema S' (der Graphklasse 2) transformiert.*

(a) op löscht einen Aktivitätenknoten n_{delete} ($n_{delete} \in N$) inklusive Kontextkanten unter Erhalt der Anordnungsbeziehung der anderen Aktivitäten aus S. Dann gilt:

I ist verträglich mit S' ⇔

NS(n_{delete}) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

(b) op löscht eine Kontroll- oder Sync-Kante $n_{src} \rightarrow n_{dest}$ aus S ($n_{src}, n_{dest} \in N$). Dann gilt:

I ist verträglich mit S'

Beweis zu Satz 3.8:

(a) Löschen von Knoten

„⇒" zu zeigen:

I ist verträglich mit S' ⇒

NS(n_{delete}) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

Hier kann wie beim Beweis von Satz 3.3 vorgegangen werden, jedoch mit folgender Widerspruchsannahme:

NS(n_{delete}) ∉ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} ≡ NS(n_{delete}) ∈ {`RUNNING, COMPLETED`}

„⇐" zu zeigen:

NS(n_{delete}) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

⇒I ist verträglich mit S'

Fall 1: NS(n_{delete}) ∈ {NOT_ACTIVATED, ACTIVATED}

⇒ ∀n ∈ c_succ*(S, n_{delete}) ∪ {n_{delete}}: NS(n) ∈ {NOT_ACTIVATED, ACTIVATED}

⇒ ∀n ∈ c_succ*(S, n_{delete}) ∪ {n_{delete}}: ∄e_i ∈ $\mathcal{A}$ mit e_i = START(n) ∨ e_i = END(n)

⇒ ∀e_i ∈ $\mathcal{A}$ mit e_i ∈ {START(n), END(n)}: n ∉ c_succ*(S, n_{delete}) ∪ {n_{delete}}

Alle bisherigen Historieneinträge wurden also von Knoten erzeugt, die entweder Vorgänger von n_{delete} waren oder die parallel zu n_{delete} ausführbar waren, d. h. die bisherige Ausführungshistorie ist auch auf S' erzeugbar.

Anmerkung: Letzeres ist in jedem Fall gegeben, wenn ein Knoten echt parallel zu n_{delete} liegt. Eine parallele Ausführung ist bei unserem Ansatz in gewissen Fällen auch dann möglich, wenn ein Knoten in einem Parallelzweig zu n_{delete} liegt und von n_{delete} aus über eine Kantenfolge $n_{delete} \rightarrow n_1 \rightarrow \ldots \rightarrow n^*$ erreichbar ist. Eine solche Konstellation ergibt sich, wenn die Kantenfolge sowohl Kontroll- als auch Sync-Kanten umfasst, n_{delete} und n* in verschiedenen Teilzweigen einer parallelen Verzweigung liegen und mindestens ein Knoten entlang der Kantenfolge den Status SKIPPED aufweist.

Fall 2: NS(n_{delete}) = SKIPPED

Für diesen Fall hat der betroffene Knoten ebenfalls keinen Historieneintrag geschrieben, so dass die bisherige Ausführungshistorie auch auf S' erzeugbar ist. □

(b) Löschen von Kontroll- und Sync-Kanten

op löscht eine Kontrollkante $n_{src} \rightarrow n_{dest}$ aus S (n_{src}, n_{dest} ∈ N). Dann gilt:

I ist verträglich mit S'

Hier kann wie beim Beweis zu Satz 3.4 argumentiert werden. Diese Vorgehensweise lässt sich auch für den Fall, dass der Ziel- bzw. Quellknoten der gelöschten Kante den Status SKIPPED besitzt, anwenden. □

Satz 3.9 (Einfügen und Löschen von XOR-Verzweigungsblöcken (Graphklasse 2)) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine Änderungsoperation op in ein korrektes WF-Schema S' (der Graphklasse 2) transformiert.*

(a) op fügt einen Verzweigungsblock mit XOR-Splitknoten n_{split} und XOR-Joinknoten n_{join} (inklusive normaler Kontrollkanten zwecks Einbindung in den WF-Kontext) in S ein. Dann gilt:

I verträglich mit S' ⇔

[∀ n ∈ succ(S', n_{join}): NS(n) ∈ {NOT_ACTIVATED, ACTIVATED, SKIPPED}] ∨

[XOR-Verzweigungsblock (n_{split}, n_{join}) wird in abgewählten Teilzweig eingefügt]

(b) op lösche einen Verzweigungsblock mit XOR-Split n_{split} und XOR-Join n_{join} in S. Dann gilt:

I verträglich mit S' ⇔

NS(n_{split}) ∈ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} [13]

Beweis zu Satz 3.9:

Aussage (a) folgt unmittelbar mit Satz 3.6a), da die Verträglichkeit einer Instanz mit dem neuen WF-Schema nur vom Zustand der direkten Nachfolger des XOR-Joinknotens n_{join} abhängig ist.

Die Aussage (b) folgt unmittelbar mit Satz 3.7a). Beim Löschen eines XOR-Verzweigungsblocks hängt die Verträglichkeit einer Instanz mit dem neuen WF-Schema nur vom Zustand des XOR-Splitknotens n_{split} ab. □

Satz 3.10 (Einfügen und Löschen von Teilzweigen in XOR-Verzweigungsblöcke) *Gegeben seien ein korrektes WF-Schema S = (N, E, ...) der Graphklasse 2 und eine WF-Instanz I auf S. S werde durch eine Änderungsoperation op auf ein korrektes WF-Schema S' der Graphklasse 2 transformiert.*

(a) op fügt einen neuen (gegebenenfalls leeren) Teilzweig in einen XOR-Verzweigungsblock (mit XOR-Splitknoten n_{split} und XOR-Joinknoten n_{join}) in S ein. Dann gilt:

I verträglich mit S'

(b) op löscht einen (gegebenenfalls leeren) Teilzweig aus einer XOR-Verzweigung (mit XOR-Splitknoten n_{split} und XOR-Joinknoten n_{join}) in S. Dann gilt:

I verträglich mit S' ⇔

[NS(n_{split}) ≠ `COMPLETED`*] ∨ [(NS(n_{split}) =* `COMPLETED` *∧ zu löschender Teilzweig abgewählt)]*

Beweis zu Satz 3.10:

(a) Einfügen eines Teilzweigs in einen XOR-Verzweigungsblock

Hier sind zwei Fälle zu unterscheiden. Falls der XOR-Splitknoten der Verzweigung beendet ist, wurde bereits ein Zweig zur Ausführung selektiert. Der neu eingefügte Teilzweig wird dementsprechend direkt nach seiner Einfügung abgewählt, so dass I auf jeden Fall verträglich mit S' ist. Falls der XOR-Splitknoten noch nicht beendet wurde, d. h. noch keine Teilzweig zur Ausführung gewählt wurde, ist das Einfügen eines neuen Teilzweiges ohnehin unkritisch. □

(b) Löschen eines Teilzweigs aus einem XOR-Verzweigungsblock

[13] Diese Bedingung kann evtl. dadurch aufgelockert werden, indem bei bereits beendeten Split-Knoten für den ersten Knoten des gewählten Teilzweigs gefordert wird, dass dieser Knoten noch nicht gestartet bzw. beendet wurde.

Die Aussage lässt sich durch einfachen Widerspruchsbeweis zeigen, auf dessen Darstellung wir an dieser Stelle verzichten. □

Die Änderung betrifft entweder einen noch nicht durchlaufenen Bereich eines WF-Graphen oder aber einen Bereich, der als nicht mehr ausführbar markiert ist. Hieraus folgt wieder unmittelbar, dass I verträglich mit S' ist. □

B.3 Beweise zu Datenfluss-Betrachtungen

Satz 3.13 (Einfügen/Löschen von Datenelementen und Datenkanten) *Seien S ein korrektes WF-Schema (bestehend aus korrektem KF-Schema CFS = (N, E, ...) und korrektem DF-Schema DFS) und I eine Instanz auf S. S werden durch eine Änderungsoperation op auf ein ebenfalls korrektes WF-Schema S' transformiert.*

(a) op fügt ein Datenelement d in DFS ein. Dann gilt:

I verträglich mit S'.

(b) op löscht ein Datenelement d aus DFS. Dann gilt:

I verträglich mit S' $\Leftrightarrow$

[Es existiert keine Aktivität mit Status `RUNNING` *oder* `COMPLETED`*, die lesend oder schreibend auf dieses Datenelement zugegriffen hat.]*

(c) Zwischen Aktivität n und Datenelement d wird durch op eine Lesekante $d \rightarrow n$ *eingefügt bzw. gelöscht. Dann gilt:*

I verträglich mit S' $\Leftrightarrow$

$NS(n) \in \{$`NOT_ACTIVATED, ACTIVATED, SKIPPED`$\}$

(d) Zwischen Aktivität n und Datenelement d wird durch op eine Schreibkante $n \rightarrow d$ *eingefügt bzw. gelöscht. Dann gilt:*

I verträglich mit S' $\Leftrightarrow$

$NS(n) \neq$ `COMPLETED`

Beweis zu Satz 3.13:

(a) Hinzufügen eines Datenelements d

Das Hinzufügen eines Datenelements d hat keine Auswirkungen auf die Lese- bzw. Schreibzugriffe auf bisherige Datenelemente. Insbesondere ist also $\mathcal{A}_{ext}$ auch auf Grundlage des neuen WF-Schemas S' erzeugbar. □

(b) Löschen eines Datenelements d

„⇒" zu zeigen:

I ist verträglich mit S' $\Rightarrow$

[Es existiert keine Aktivität mit Status `RUNNING` oder `COMPLETED`, die lesend oder schreibend auf dieses Datenelement zugegriffen hat.]

Wir beweisen obige Aussage durch Widerspruch, d. h. wir zeigen:

[$\exists n^* \in N$ mit NS(n^*) $\in$ {`RUNNING, COMPLETED`}

und n^* hat lesend oder schreibend auf d zugegriffen] $\Rightarrow$ I nicht verträglich mit S'

Annahme:

[$\nexists n^* \in N$ mit NS(n^*) $\in$ {`RUNNING, COMPLETED`}

und n^* hat lesend oder schreibend auf d zugegriffen]

$\equiv \exists e_\mu^{((d_1^{(\mu)},v_1^{(\mu)}),\ldots,(d_m^{(\mu)},v_m^{(\mu)}))} \in \mathcal{A}_{ext}$ mit $e_\mu \in$ {`START(`n^*`), END(`n^*`)`}

$\wedge \exists l \in \{1,\ldots,m\}$ mit $d_l^{(\mu)} =$ d, $n^* \in N$

Hieraus folgt, dass I nicht verträglich mit S' sein kann, da das Datenelement d in S' nicht vorhanden ist. □

„⇐" zu zeigen:

A $\equiv$ [Es existiert keine Aktivität mit Status `RUNNING` oder `COMPLETED`, die lesend oder schreibend auf d zugegriffen hat.]

A $\Rightarrow$ I ist verträglich mit S'

I ist verträglich mit S', wenn $\mathcal{A}_{ext}$ ist auch auf S' erzeugbar ist. Insbesondere muss gelten:

$\nexists e_\mu^{((d_1^{(\mu)},v_1^{(\mu)}),\ldots,(d_m^{(\mu)},v_m^{(\mu)}))} \in \mathcal{A}_{ext}$ mit $e_\mu \in$ {`START(`n^*`), END(`n^*`)`}

$\wedge \exists l \in \{1,\ldots,m\}$ mit $d_l^{(\mu)} =$ d, $n^* \in N$

Dies trifft wegen der Gültigkeit von A zu. □

(c) Einfügen und Löschen einer Lesekante

Wir betrachten zunächst den Fall, dass eine Lesekante zwischen Aktivität n und Datenelement d eingefügt wird.

„⇒" zu zeigen:

I ist verträglich mit S' $\Rightarrow$ NS(n) $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`}

Wir beweisen obige Aussage durch Widerspruch, d. h. wir zeigen:

NS(n) $\in$ {`RUNNING, COMPLETED`} $\Rightarrow$ I nicht verträglich mit S'

Annahme:

NS(n) $\in$ {`RUNNING, COMPLETED`}

$\Rightarrow \exists e_{\mu}^{((d_1^{(\mu)},v_1^{(\mu)}),...,(d_m^{(\mu)},v_m^{(\mu)}))} \in \mathcal{A}_{ext}$ mit e_μ = `START(n)`

Da zum Zeitpunkt des Einfügens der Lesekante $d \rightarrow n$ der Knoten n bereits gestartet wurde, hat n lesend auf Datenelemente zugegriffen, jedoch nicht auf d, d. h. $\nexists l \in \{1, ..., m\}$ mit $d_\mu^{(l)} =$ d. Dann kann aber I nicht verträglich mit S' sein, da bei der Ausführung von Instanzen auf S' beim Start des Knotens n (wegen der Kante $d \rightarrow n$) lesend auf d zugegriffen wird. □

„⇐" zu zeigen:

NS(n) $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} $\Rightarrow$ I ist verträglich mit S'

Wenn I verträglich mit S' sein soll, muss $\mathcal{A}_{ext}$ auch auf S' erzeugbar sein. Insbesondere kann es dadurch noch kein $e_{\mu}^{((d_1^{(\mu)},v_1^{(\mu)}),...,(d_m^{(\mu)},v_m^{(\mu)}))} \in \mathcal{A}_{ext}$ geben mit e_μ = `START(n)`. Ansonsten würde $d_\mu^{(l)} \neq$ d $\forall$ l gelten, was im Widerspruch zur Annahme NS(n) $\in$ {`NOT_ACTIVATED, ACTIVATED, SKIPPED`} steht. □

Beim Löschen einer Lesekante $d \rightarrow n$ kann beweistechnisch ähnlich wie beim Einfügen vorgegangen werden. Wird nämlich $d \rightarrow n$ aus S entfernt, darf n noch nicht lesend auf d zugegriffen haben, da dieser Lesezugriff auf S' nicht mehr möglich ist. Lesezugriffe erfolgen jedoch stets beim Start einer Aktivität, woraus folgt, dass n noch nicht gestartet worden sein darf. Auf einen detaillierten Beweis verzichten wir an dieser Stelle. □

(d) Einfügen und Löschen von Schreibkanten

Hier kann ähnlich wie in Fall (c) argumentiert werden. Einziger Unterschied ist, dass die betroffene Aktivität n schon gestartet worden sein darf. Dies ist darin begründet, dass Schreibzugriffe auf Datenelemente erst bei Beendigung der Aktivität erfolgen. Damit können Manipulationen an Schreibkanten solange erfolgen, wie sich die zugehörige Aktivität noch nicht im Zustand `COMPLETED` befindet. Auf weitere Details wird an dieser Stelle verzichtet. □