ulm university universität
uulm

Institut für Datenbanken und Informationssysteme
(Leiter: Prof. Dr. P. Dadam)

Management datengetriebener Prozessstrukturen

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

vorgelegt von
Dominic Müller
aus Ulm

2009

Amtierender Dekan:	Prof. Dr.-Ing. M. Weber
Gutachter:	Prof. Dr. M. Reichert Prof. Dr. P. Dadam Dr.ir. H. A. Reijers, Associate Professor, Eindhoven University of Technology
Tag der Promotion:	10. Juli 2009

Kurzfassung

Unternehmen erreichen ihre Geschäftsziele zunehmend durch das systematische Management ihrer (Geschäfts-)Prozesse. Prozesse beschreiben beispielsweise die Buchung einer Reise oder die Entwicklung eines Produkts mit den dafür erforderlichen Arbeitsschritten (z.B. Entwurf, Konstruktion und Produktion). Um komplexe Geschäftsziele zu realisieren, lassen sich Prozesse meist verknüpfen und so Prozessstrukturen aus einer Vielzahl an Prozessen aufbauen.

Ein sehr komplexes Geschäftsziel ist beispielsweise die Entwicklung der Fahrzeugelektronik im Automobilbau. Ihre einwandfreie Funktion kann nur durch ein systematisches Vorgehen bei der Entwicklung und Absicherung des Fahrzeugs sichergestellt werden. Hierbei müssen insbesondere die zahlreichen Abhängigkeiten zwischen elektronischen Systemen erfasst und in entsprechende Abhängigkeiten zwischen Entwicklungsprozessen umgesetzt werden. Das Ergebnis ist eine *datengetriebene Prozessstruktur*, die eine starke Beziehung zwischen der Struktur des Produkts und den auszuführenden Prozessen beschreibt. Sie enthält hunderte bis tausende Prozesse mit entsprechenden Abhängigkeiten. Die Erstellung einer datengetriebenen Prozessstruktur ist sehr aufwändig und kann manuell kaum bewerkstelligt werden. Sie erfordert daher eine geeignete Modellierungsunterstützung.

Die vorliegende Arbeit stellt mit Corepro (*Configuration Based Release Processes*) eine durchgängige IT-Lösung für die Unterstützung datengetriebener Prozessstrukturen vor. Corepro erlaubt ihre formale Beschreibung und Ausführung basierend auf einem intuitiven Basismodell. Dabei nutzen wir die Zusammenhänge zwischen der Struktur eines Produkts und der zugehörigen Prozessstruktur konsequent aus. Wir führen eine Modellierungsunterstützung ein, die die automatische Erzeugung der datengetriebenen Prozessstruktur basierend auf einer gegebenen Produktstruktur erlaubt und damit die Modellierungsaufwände signifikant reduziert.

Darüber hinaus erfordert die Komplexität großer Prozessstrukturen auch eine geeignete IT-Unterstützung für deren Ausführung, also die Koordination und Überwachung der einzelnen Prozesse. Hierbei muss sichergestellt sein, dass es über die gesamte Ausführungsdauer, welche mehrere Monate betragen kann, zu keinen Verzögerungen durch eine ineffiziente Prozesskoordination kommt. Eine große Herausforderung im Umfeld der Fahrzeugentwicklung ist die hohe Dynamik der die Entwicklung elektronischer Systeme unterworfen ist. Die starre Koordination durch klassisch modellierte Prozessstrukturen ist im betrachteten Entwicklungsumfeld nicht ausreichend. Vielmehr verlangt die Komplexität des Umfelds flexible Reaktionen auf sich ändernde Rahmenbedingungen. Sie erfordern einerseits dynamische Adaptionen der Prozessstrukturen, andererseits deren flexible Steuerung zur Abbildung von Ausnahmesituationen.

Corepro liefert Methoden zur Adaption datengetriebener Prozessstrukturen auf einer hohen Abstraktionsebene. Die Methoden transformieren Änderungen einer Produktstruktur direkt auf Adaptionen der zugehörigen Prozessstruktur. Geeignete Konsistenzanalysen stellen sicher, dass bei der Adaption zur Laufzeit mögliche Ausnahmesituationen erkannt werden. Diese lassen sich in Corepro durch verschiedene Mechanismen für die flexible Ausführung behandeln. Sie erlauben dem Nutzer nicht nur Eingriffe in den Ablauf einer Prozessstruktur, sondern zeigen ihm auch die Konsequenzen derartiger Eingriffe an. Darauf basierend liefert Corepro Lösungsvorschläge für die Behandlung von Ausnahmen. Die korrekte, verklemmungsfreie Ausführung der Prozessstruktur wird hierbei durchgehend garantiert.

Vorwort

Diese Dissertation entstand im Rahmen einer Kooperation der *Daimler AG Research & Development* mit der *University of Twente* (Niederlande) und der *Universität Ulm*. Ich war zweieinhalb Jahre als *Ph.D. Student* an der *University of Twente* angestellt, bevor ich 2008 an das *Institut für Datenbanken und Informationssysteme* (DBIS) der *Universität Ulm* wechselte. Die wissenschaftliche Arbeit war durchweg auch durch praktische Tätigkeiten geprägt, die ich als externer Mitarbeiter der Abteilung Prozess- und Datenmanagement der Daimler AG übernahm. Als Ergebnis liegt mit dieser Arbeit ein umfassender Ansatz für das Management datengetriebener Prozessstrukturen vor. Die Herausforderung ist nun, die entwickelten Konzepte auch produktiv einzusetzen. Erfreulicherweise wird sich Daimler dieser Herausforderung stellen und hat mit der Fortführung verschiedener Arbeiten in diesem Umfeld die Nachhaltigkeit des Themas bestätigt.

An dieser Stelle möchte ich einigen Personen danken, deren Unterstützung für die Erstellung dieser Arbeit essentiell war. Mein größter Dank gilt Prof. Dr. Manfred Reichert, der mich für das spannende Thema und das herausfordernde Umfeld mit Auslandsaufenthalt begeistern konnte. Er betreute diese Arbeit ausdauernd und mit einmaligem Engagement. Gleichermaßen bedanke ich mich bei Dr. Joachim Herbst für die Betreuung dieser Arbeit auf Seite der Daimler AG. Er bildete das Bindeglied in die Praxis und hat seine wertvollen Erfahrungen in fruchtbare Diskussionen eingebracht. Seine pragmatische und zielorientierte Vorgehensweise ergänzte sich hervorragend mit der Betreuung von wissenschaftlicher Seite.

Des Weiteren danke ich Prof. Dr. Peter Dadam für die Übernahme des Zweitgutachtens, seinem Interesse an dem Thema und die konstruktive Unterstützung vor und nach meiner „Ankunft“ in Ulm. Ferner danke ich allen Mitarbeitern des Instituts für Datenbanken und Informationssysteme der Universität Ulm. Sie haben mich sehr herzlich aufgenommen und in den letzten Monaten bis zum Abschluss der Dissertation begleitet. Insbesondere danke ich Eva Mader, Rüdiger Pryss, Jens Kolb, Thao Ly, Michael Predeschly und Vera Künzle für die tatkräftige Unterstützung.

Für die Schaffung und Aufrechterhaltung der organisatorischen Rahmenbedingungen und für die intensiven und lehrreichen Diskussionen im Daimler Forschungszentrum danke ich ganz besonders dem Leiter des Team Prozessmanagement, Reiner Siebert, sowie dem höheren Management Ralf Lamberti, Dr. Frank Arbes und Katrin Breitrück. Mein Dank geht des Weiteren an Oliver Anspacher, Dr. Thomas Bauer, Dr. Thomas Beuter, Alena Hallerbach, Bojan Pounarov, Andreas Warkentin, Dr. Erich Müller, Cord Rodefeld und Prof. Dr. Bela Mutschler für das professionelle Arbeitsumfeld und die moralische Unterstützung bei und neben der Arbeit. Meinen Daimler-Bürokollegen Dr. Ralph Bobrik und Stephan Buchwald danke ich insbesondere für das angenehme Büroklima, für die vielen Tipps und die gegenseitige Unterstützung. Ebenso bedanke ich mich bei meinen studentischen Hilfskräften Florian Poppa, Detlef Köntges, Andreas Neubert und Ralf Enderle für ihr Engagement bei der prototypischen Realisierung der Konzepte.

Zu guter Letzt möchte ich mich bei meiner Familie, bei meinen Freunden und insbesondere bei meiner Freundin Alexandra für ihr Verständnis, ihren Rückhalt und das intensive Korrekturlesen bedanken. Ohne sie hätte ich die notwendige Motivation und Disziplin für die Realisierung dieser Arbeit sicher nicht aufbringen können.

Ulm, im April 2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I

Problembeschreibung und Problemanalyse

1

Einleitung

Ein umfangreiches IT-seitiges Prozess-Management unterstützt Unternehmen in dem Vorhaben, ihre Unternehmensziele systematisch umzusetzen. Aspekte des Prozess-Managements betreffen die Modellierung, Ausführung, Überwachung und Änderung von *(Geschäfts-)Prozessen*. Mit der Anwendung von Prozess-Management-Konzepten steigt die Anzahl der in Unternehmen modellierten Prozesse und lässt durch ihre Vernetzung die Beschreibung übergreifender Unternehmensabläufe zu. Die damit verbundene Fragmentierung auf verschiedene IT-Systeme erhöht allerdings auch die Komplexität der Prozesskoordination. Während die Modellierung von Prozessen zunehmend Anwendung findet, ist die automatisierte Ablaufsteuerung komplexer Prozessstrukturen durch Prozess-Management-Technologien nicht durchgängig realisiert. Dadurch ergeben sich Einschränkungen hinsichtlich Überblick und Kontrolle über den Gesamtprozess.

Eine Herausforderung für das Prozess-Management ist die Ausführung großer und lang laufender Prozesse zur Herstellung komplexer Produkte. Beispielsweise soll der Entwicklungsprozess für Fahrzeuge eine schnelle Produktentwicklung bei hoher Produktqualität sicherstellen. Der Prozess verfügt hierbei über verschiedene Detaillierungsebenen. Er aggregiert sich aus den Abläufen, die von verschiedenen Abteilungen für die Entwicklung einzelner Komponenten ausgeführt werden. Der Entwicklungsprozess für ein Fahrzeug resultiert damit in einer großen Prozessstruktur, die aus tausenden von verteilt ausgeführten (Sub-)Prozessen besteht. Der Aufbau der Prozessstruktur ist datengetrieben, d.h. er orientiert sich am Aufbau des Produkts. Um die Ziele der schnellen Produktentwicklung bei hoher Produktqualität sicherzustellen, müssen die einzelnen Prozesse der Prozessstruktur *systematisch* koordiniert werden. Aufgrund der Größe der Prozessstrukturen und der damit verbundenen Komplexität kann dies manuell nicht zuverlässig geschehen. Daher ist eine geeignete Unterstützung durch Prozess-Management-Technologien notwendig.

1.1 Hintergrund

Bevor wir die Problemstellung und den Beitrag dieser Arbeit erörtern, behandeln wir zunächst einige grundlegende Begriffe aus dem Bereich des IT-Prozess-Managements. Sie sind für das weitere Verständnis dieser Arbeit notwendig.

Für Unternehmen ist das Management von Geschäftsprozessen eine wichtige Grundlage für die effiziente und strukturierte Koordination seiner wertschöpfenden Abläufe [SS07]. Prozesse beschreiben den logischen Ablauf von Arbeitsschritten zur Erfüllung eines Geschäftsziels. So dokumentieren Prozesse eines Fahrzeugherstellers beispielsweise diejenigen Aktivitäten, die zur Entwicklung eines Motors notwendig sind (z.B. Spezifikation, Prüfstandtest und Fahrversuch). Die Anordnungsbeziehungen der Arbeitsschritte eines *aktivitätenorientierten Prozesses* werden explizit durch *Kontrollflusskanten* (d.h. gerichtete Verbindungen der Aktivitäten) beschrieben. Durch geeignete Kontrollflusskonstrukte lassen sich *nicht-deterministische Abläufe* mit *bedingten* und *parallelen Verzweigungen* genauso modellieren wie *Rücksprünge* oder *Schleifen* (vgl. Abbildung 1.1). Prozesse lassen sich in der Regel auch hierarchisch schachteln, d.h. eine Aktivität kann selbst wieder einen (Sub-)Prozess repräsentieren. Ein derart dokumentierter Prozess kann beispielsweise als Grundlage für die Optimierung der Abläufe dienen.

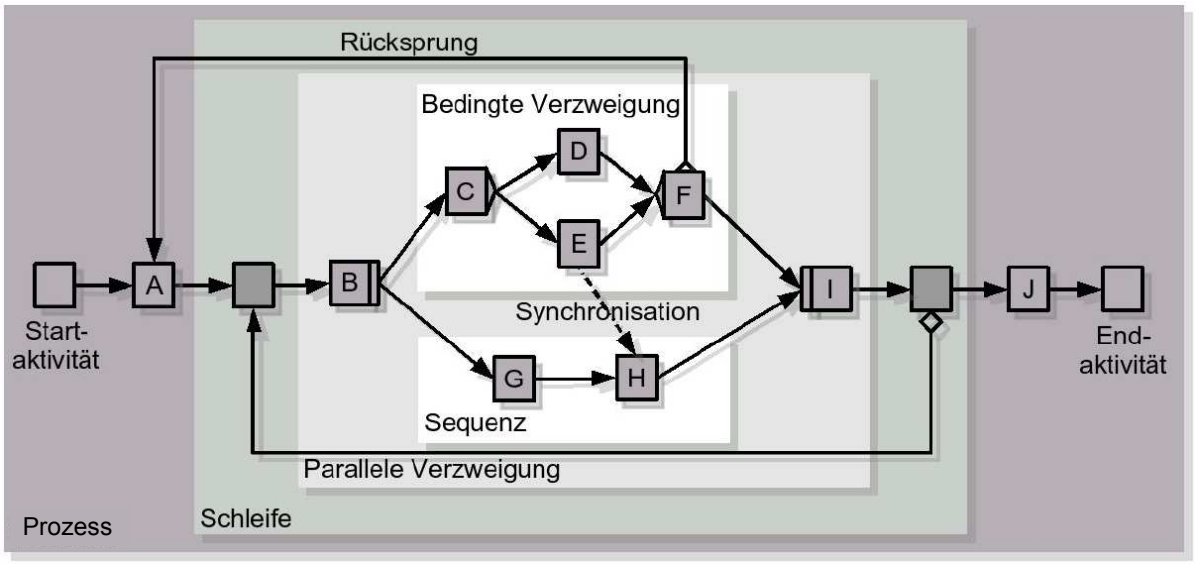


Abbildung 1.1: Aktivitätenorientierte Prozessbeschreibung durch Kontrollflusskonstrukte [Rei00]

Neben der Dokumentation von Geschäftsprozessen besteht vor allem ein zunehmendes Interesse an einer IT-Unterstützung der Ablaufsteuerung [MRB08]. Häufig sind die Aktivitäten des Geschäftsprozesses in Form von Software-Anwendungen (z.B. die Eingabe notwendiger Informationen durch den Benutzer) bereits realisiert. Ziel des Prozess-Managements ist es, die Ablauflogik zur Verschaltung der verschiedenen Aktivitäten nicht fest zu implementieren (*hart codiert*), sondern durch die automatisierte Steuerung der Aktivitäten entsprechend des modellierten Kontrollflusses zu realisieren. Derartige Systeme werden *prozessorientierte Informationssysteme* genannt. Die Steuerung der Prozesse geschieht dabei im Allgemeinen durch den Einsatz von *Prozess-Management-Systemen* bzw. *Workflow-Management-Systemen*. Sie erlauben dem Modellierer,

die Beschreibung eines Geschäftsprozesses um technische Aspekte der enthaltenen Aktivitäten anzureichern (z.B. die konkret auszuführenden Software-Anwendungen) und den Prozess damit ausführbar zu machen [GHS95]. Der ausführbare Prozess wird in der Literatur auch *Workflow* genannt [LR00].[1] Wird der Prozess in einem Prozess-Management-System zur Ausführung gebracht, kann sein *Verlauf* überwacht und zu Dokumentationszwecken protokolliert werden.

Der Vorteil prozessorientierter Informationssysteme liegt in der Trennung der Prozesslogik von der Implementierung der eigentlichen Anwendungen. Diese Entkopplung erhöht nicht nur die generelle Einsetzbarkeit der verschiedenen Anwendungen, sondern erlaubt auch deren Austausch und Wiederverwendung. Des Weiteren kann die Prozesslogik schnell an neue Rahmenbedingungen angepasst werden ohne die Implementierung der Anwendung zu verändern. Die Prozesslogik wird im Allgemeinen im *Prozesssschema* (*Prozessmodell*) definiert. Anschließend wird für jeden konkret auszuführenden Geschäftsprozess eine *Prozessinstanz* erzeugt, die dann vom Prozess-Management-System über ihre komplette Lebensdauer gesteuert und überwacht wird [Wor94, Wes07]. Soll ein in Ausführung befindlicher Prozess verändert werden, indem beispielsweise neue Aktivitäten in den Kontrollfluss eingefügt werden, spricht man von *dynamischer Adaption* der Prozessinstanz. Prozess-Management-Systeme, die eine solche dynamische Adaption unterstützen, werden als *flexibel* bezeichnet [RD98].

Ein Prozesssschema basiert auf einem *Metamodell*, welches die für die Modellierung zur Verfügung stehenden Elemente definiert. Um die Ausführung eines Prozesses zu ermöglichen, muss für dessen Metamodell eine präzise *operationale Semantik*, also klare Regeln für die Abfolge der einzelnen Prozessschritte entsprechend der nutzbaren Kontrollflusskonstrukte, vorliegen. Im Allgemeinen werden in diesem Rahmen auch die *Korrektheitskriterien* für die Modellierung eines Prozesses beschrieben. Sie verhindern beispielsweise *Verklemmungen* (*Deadlocks*) oder *Endlosschleifen* (*Livelocks*) durch unsachgemäße Anordnung von Kontrollflusskonstrukten (vgl. Abbildung 1.2). So fordert das ADEPT-Metamodell die Erstellung eines definierten Start- und Endknotens und die weitgehend *blockstrukturierte Anordnung* der Kontrollflusskanten ohne Verschränkung von Verzweigungsblöcken (vgl. Abbildung 1.1) [RD98].

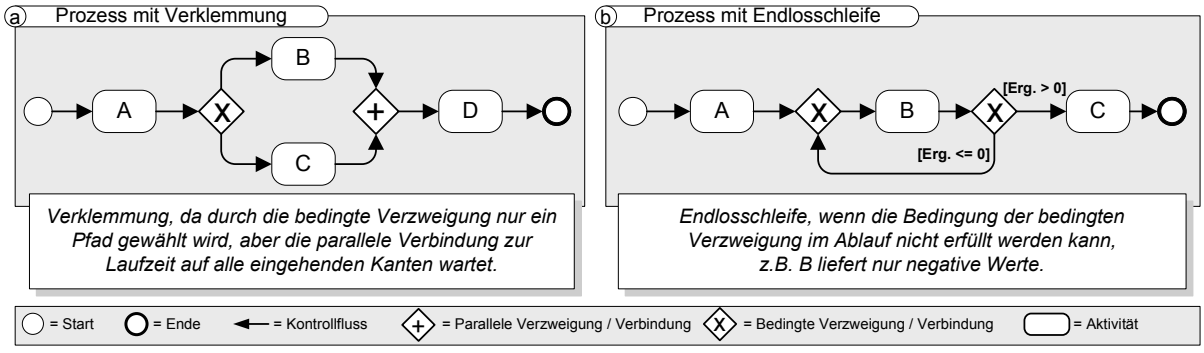


Abbildung 1.2: Inkorrekte Prozessabläufe durch Verschaltung von Kontrollflusskonstrukten

Neben dem Kontrollfluss, bestehend aus Aktivitäten, Kontrollflusskonstrukten und -kanten, bilden Prozessmodelle häufig weitere Prozessaspekte ab. Oft werden Geschäftsprozesse um den *Datenfluss* erweitert. Der Datenfluss definiert mittels *Datenelementen* und *Datenflusskanten* die

[1] Wir nutzen in dieser Arbeit jedoch durchgängig den Begriff *Prozess* und verstehen darunter sowohl ausführbare als auch nicht-ausführbare Prozesse.

Nutzung von Daten als Ein- bzw. Ausgaben der einzelnen Aktivitäten des Prozesses. *Datenorientierte Prozesse* beschreiben i.A. die Abfolge der Aktivitäten exklusiv über den Datenfluss.

Weitere Informationen wie Bearbeiterzuordnungen, die jeder interaktiven Aktivität die ausführende Rolle bzw. Person zuordnen, können ebenfalls Gegenstand der Prozessmodellierung sein. Sie spielen in dieser Arbeit jedoch keine Rolle und werden daher nicht weiter behandelt.

1.2 Problemstellung

In der Praxis sind Geschäftsprozesse häufig stark vernetzt und bilden daher komplexe *Prozessstrukturen*. Insbesondere in technischen Domänen (z.B. im Maschinenbau) bestehen Prozessstrukturen oftmals aus tausenden von (Einzel-)Prozessen, die geeignet modelliert und während der Ausführung koordiniert werden müssen. Die Verarbeitung eines technischen Produkts bzw. seiner Komponenten (z.B. eines Fahrzeugs) geschieht durch unabhängige Prozesse. Ziel ist es, den logischen Gesamtablauf in Form einer Prozessstruktur für das zu bearbeitende Produkt zu beschreiben und dadurch eine übergreifende Kontrolle zur systematischen Erreichung des Geschäftsziels zu erhalten. Typischerweise orientiert sich die Prozessstruktur für die Bearbeitung eines Produkts an dessen Aufbau, also an der *Produktstruktur* (vgl. Abbildung 1.3). Produktstrukturen beschreiben die Zusammensetzung eines Produkts bestehend aus Komponenten, Konfigurationen und deren Beziehungen [SS01]. Sie umfassen alle *möglichen* Varianten eines Produkts (d.h. die vom Kunden auswählbaren Fahrzeugausstattungen; sog. 150% Konfiguration) und werden während der Entwicklungsphase erstellt. Zur Produktion werden aus Produktstrukturen sogenannte *Stücklisten* (auch als *Bill of Material* bezeichnet) abgeleitet. Sie beschreiben eine konkrete Ausprägung des Produkts (sog. 100% Konfiguration) [Sch02].

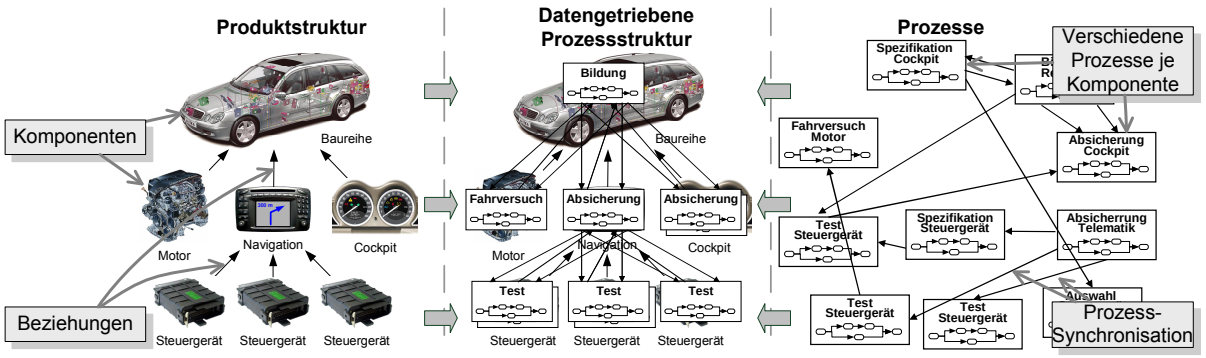


Abbildung 1.3: Abhängigkeiten zwischen Produktstruktur und Prozessen

Die prinzipiell unabhängigen Prozesse einer Prozessstruktur können einerseits bestimmten Komponenten der Produktstruktur zugeordnet (z.B. Testprozesse eines Steuergeräts), andererseits entsprechend der Beziehungen zwischen Komponenten synchronisiert werden (vgl. Abbildung 1.3). So ergibt sich beispielsweise die logische Reihenfolge der Testprozesse aus den Abhängigkeiten der zu testenden Komponenten. Eine solche integrierte Prozessstruktur bezeichnen wir als *datengetriebene Prozessstruktur*.

Die Modellierung datengetriebener Prozessstrukturen ist nicht nur für die durchgängige Dokumentation der Abläufe notwendig. Vielmehr steht deren reibungsloser und verklemmungsfreier

Ablauf im Vordergrund, um teure Unterbrechungen und damit erhöhte Prozesskosten zu vermeiden. Die manuelle Koordination der Prozesse im Sinne der Prozessstruktur führt allerdings aufgrund der Größe mit tausenden von Prozessen und Abhängigkeiten zu hohen Aufwänden. Ferner sind im Kontext der manuellen Koordination der reibungslose und verklemmungsfreie Ablauf genauso wenig sichergestellt, wie die notwendige Überwachung und Dokumentation des Verlaufs der Prozessstruktur.

Ein weiterer wesentlicher Aspekt bei der Ausführung datengetriebener Prozessstrukturen ist deren dynamische Adaption. Durch ihre Abhängigkeit von einer Produktstruktur, muss bei einer Änderung der Produktstruktur im Allgemeinen die Prozessstruktur entsprechend adaptiert werden. Wird beispielsweise während der Ausführung eine Komponente entfernt, muss dafür gesorgt werden, dass auch die zur Komponente gehörenden Prozesse aus der Prozessstruktur entfernt und ggf. terminiert werden. In diesem Rahmen muss jederzeit sichergestellt sein, dass entstehende Ausnahmesituationen rechtzeitig erkannt und dem Benutzer geeignet dargestellt werden, damit er sie mittels geeigneter Methoden kontrollieren kann. Ausnahmesituationen können nicht nur im Rahmen der dynamischen Adaption entstehen, sondern auch in der *realen Welt*. Wird beispielsweise festgestellt, dass ein vorheriger Prozess unsachgemäß ausgeführt wurde (z.B. Einbau einer fehlerhaften Komponente), muss flexibel darauf reagiert werden können (z.B. Wiederholen des Prozesses).

Es ist offensichtlich, dass die Modellierung und Koordination einer großen datengetriebenen Prozessstruktur manuell kaum zu bewerkstelligen ist. Darüber hinaus müssen in Ausnahmesituationen während der Ausführung einzelne Prozesse der Prozessstruktur ausgetauscht, wiederholt oder übersprungen werden. Eine geeignete IT-Unterstützung ist für die konsistente Durchführung derartiger Operationen essentiell. Die Herausforderung für das Prozess-Management ist die Nutzung der zur Verfügung stehenden Informationen über das Produkt und seiner Struktur für eine adäquate Unterstützung datengetriebener Prozessstrukturen. Dafür wird ein durchgängiges Lösungskonzept benötigt, das die beschriebenen Aspekte Modellierung, Ausführung, Adaption und Ausnahmebehandlung vollständig abdeckt.

1.3 Beitrag

Diese Arbeit ist das Ergebnis des Projekts CoRepro (*Configuration Based Release Processes*).[2] Ziel von CoRepro ist die Entwicklung und prototypische Realisierung von generischen IT-Konzepten zur durchgängigen Unterstützung datengetriebener Prozessstrukturen. Derartige Konzepte werden beispielsweise für die Koordination von Prozessen der Fahrzeugentwicklung benötigt. In diesem Bereich geschieht die Koordination datengetriebener Prozessstrukturen aufgrund fehlender technischer Unterstützung derzeit manuell. Die vorliegende Arbeit liefert mit CoRepro einen vollständigen Lösungsansatz, der eine durchgängige Unterstützung für datengetriebene Prozessstrukturen über ihren gesamten Lebenszyklus bereitstellt und folgende Aspekte abdeckt:

[2]Das Projekt ist eine Kooperation der Daimler AG mit der University of Twente und der Universität Ulm.

- **Modellierung**: COREPRO stellt Konzepte für die Modellierung[3] ausführbarer Prozessstrukturen zur Verfügung. Dazu führen wir einen Mechanismus ein, der die automatisierte Erzeugung einer datengetriebenen Prozessstruktur erlaubt und damit den Modellierungsaufwand für große Strukturen signifikant reduziert.
- **Ausführung**: COREPRO definiert eine vollständige operationale Semantik und zugehörige Korrektheitskriterien für Prozessstrukturen. Sie garantieren die korrekte und verklemmungsfreie Koordination der in der modellierten Prozessstruktur enthaltenen Prozesse.
- **Dynamische Adaption**: COREPRO erlaubt die Adaption einer sich in Ausführung befindlichen Prozessstruktur. Dies geschieht auf hoher Abstraktionsebene, d.h. eine Änderung der Produktstruktur führt automatisch zu einer Adaption der entsprechenden Prozessstruktur. Es werden hierfür geeignete Operationen und Konsistenzkriterien, die die Korrektheit der Prozessstruktur weiterhin gewährleisten, definiert.
- **Ausnahmebehandlung**: COREPRO ermöglicht die Behandlung von realen fachlichen Ausnahmen genauso wie von Inkonsistenzen, die durch die dynamische Adaption entstehen können. Wir stellen innovative Mechanismen zur Verfügung, mit deren Hilfe Ausnahmen erkannt, kontrolliert und aufgelöst oder ignoriert werden können. Die Korrektheit der Prozessstruktur wird dabei durchgehend garantiert.

Die Arbeit liefert des Weiteren eine umfangreiche Validation der entwickelten Konzepte. So werden mit der prototypischen Realisierung Fragestellungen zur technischen Umsetzbarkeit der Konzepte bearbeitet. Darüberhinaus zeigt eine umfangreiche Fallstudie die Anwendbarkeit der Konzepte in der Praxis.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in vier Teile.

Teil I besteht aus den Kapiteln 1 bis 3 und beschreibt die Herausforderungen der datengetriebenen Koordination und dynamischen Adaption großer Prozessstrukturen. Kapitel 2 schildert die Problemstellung anhand konkreter Anwendungen datengetriebener Prozessstrukturen und führt eine Anforderungsanalyse durch. Die Abdeckung der definierten Anforderungen durch den Stand der Technik diskutiert Kapitel 3.

In Teil II stellen wir einen technischen Lösungsansatz zur vollständigen Unterstützung der in Kapitel 2 definierten Anforderungen vor. Kapitel 4 entwickelt die benötigten Grundlagen für die formale Beschreibung und Ausführung von Prozessstrukturen. Kapitel 5 definiert die Modellierungskonzepte für die datengetriebene Erzeugung großer Prozessstrukturen. Sie erlauben eine signifikante Reduktion des Modellierungsaufwands bei gleichzeitiger Zusicherung der definierten Korrektheitskriterien. Die Kapitel 6 und 7 beschäftigen sich mit Flexibilitätsaspekten, die bei der Ausführung datengetriebener Prozessstrukturen relevant sind. Kapitel 6 definiert Operationen zur dynamischen Adaption datengetriebener Prozessstrukturen sowie geeignete Kriterien zur Erkennung von Inkonsistenzen, die im Rahmen ihrer Anwendung entstehen können. Kapitel 7

[3] Unter der *Modellierung* einer datengetriebenen Prozessstruktur verstehen wir die Erstellung einer konkreten Prozessstruktur-Instanz für eine gegebene Produktstruktur.

beschreibt schließlich verschiedene Konzepte zur Behandlung von Ausnahmen bzw. Inkonsistenzen und stellt damit sicher, dass entsprechende Prozessstrukturen weiterhin den definierten Korrektheitskriterien genügen.

Teil III befasst sich mit der Validation der in Teil II vorgestellten Lösungskonzepte. Kapitel 8 beschreibt die technischen Aspekte der Implementierung eines mächtigen Modellierungs- und Ausführungswerkzeugs für datengetriebene Prozessstrukturen und zeigt damit die technische Umsetzbarkeit der Lösung. Ferner zeigt die Beschreibung relevanter Szenarien verschiedener Fallstudien in Kapitel 9 die Anwendbarkeit der Lösung datengetriebener Prozessstrukturen in der Praxis.

Teil IV bildet den Abschluss der Arbeit. Hierfür fassen wir in Kapitel 10 die wichtigsten Erkenntnisse dieser Arbeit zusammen und geben einen Ausblick auf zukünftige Arbeiten.

2 Anforderungsanalyse

Die in Kapitel 1 gegebene Einführung in das Thema dieser Arbeit wollen wir in diesem Kapitel durch eine umfangreiche Problem- und Anforderungsanalyse untermauern. Hierfür untersuchen wir relevante Anwendungen für datengetriebene Prozessstrukturen. Typischerweise finden sie sich im technischen Umfeld der Produktionsplanung oder der Materialbedarfsplanung. Dort wird beispielsweise die Reihenfolge der Bestellprozesse für die Komponenten eines Motors durch dessen Produktstruktur (bzw. Stückliste) vorgegeben [KMSOW06]. Datengetriebene Prozessstrukturen sind aber auch in der Produktentwicklung vonnöten [Aal99, ARL01, BHR05, MHHR06]. So erfordert die Entwicklung komplexer technischer Produkte die Koordination vieler Einzelprozesse zur gesamtheitlichen Erfüllung konstruktiver, absichernder und organisatorischer Aufgaben. Im Vergleich zur Produktion sind die Anforderungen der Entwicklung an das Prozess-Management allerdings komplexer. Während in der Produktion die definierten Abläufe üblicherweise strikt eingehalten werden müssen, um die geplante Prozessdauer und die geforderte Produktqualität zu garantieren, müssen langlaufende Prozessstrukturen in der Entwicklung oftmals dynamisch adaptiert werden. So muss flexibel auf den Austausch von Komponenten reagiert und die Prozessstruktur entsprechend angepasst werden. Zudem treten häufig nicht planbare Ausnahmen auf, die beispielsweise eine Wiederholung bereits ausgeführter Prozesse erfordern [BDS98, AB01b, Beu03]. Im Rahmen einer umfassenden Analyse der Prozesse zur Entwicklung elektronischer Komponenten im Automobilbau haben wir die Anforderungen an Konzepte zur Unterstützung datengetriebener Prozessstrukturen detailliert untersucht [BHR05, MHHR06, MRH06]. Die Analyse weiterer Fachdomänen zeigt, dass derartige Anforderungen unter anderem in der Software-Entwicklung und im Finanz- und Versicherungssektor [Rei02, WFJ+03, FWM+04] zu finden sind.

Das vorliegende Kapitel gliedert sich wie folgt: Abschnitt 2.1 beschreibt die analysierten Prozesse der Elektronik-Entwicklung und stellt fachliche Anforderungen an entsprechende Prozess-Management-Konzepte dar. In Abschnitt 2.2 leiten wir aus den Analyseergebnissen generalisierte Anforderungen an die Unterstützung datengetriebener Prozessstrukturen ab. Abschnitt 2.3 beschreibt mit Anwendungsbeispielen aus der Fahrzeugentwicklung, der Software-Entwicklung

und der Finanzbranche weitere Anwendungsdomänen für datengetriebene Prozessstrukturen. Abschnitt 2.4 gibt eine Zusammenfassung.

2.1 Datengetriebene Prozessstrukturen zur Entwicklung der Fahrzeugelektronik

Der Anteil elektronischer Systeme an den wertschöpfenden Fahrzeugfunktionen hat sich im letzten Jahrzehnt enorm gesteigert und wird sich noch weiter erhöhen [Rei07, Tan07]. Durch die zunehmende Vernetzung der elektronischen Systeme ist auch die technische Komplexität des Gesamtsystems stark angestiegen. Aufgrund hoher Qualitäts- und Sicherheitsansprüche ist ein einwandfreies Zusammenspiel der vernetzten Systeme essentiell. Wir beschreiben im Folgenden die Herausforderungen bei der technischen Abbildung entsprechender Entwicklungsprozesse.

Technischer Hintergrund

Das elektrische bzw. elektronische (E/E) *Gesamtsystem* heutiger Fahrzeuge besteht aus bis zu 70 Steuergeräten [KS04]. Diese stellen eine Vielzahl an Funktionen bereit, wie sie zum Beispiel für Multimedia, Navigation, Klimaautomatik oder Fahrwerk (z.B. elektronisches Stabilitätsprogramm; ESP) benötigt werden. Ein *Steuergerät* (*Electrical Control Unit*) ist im Allgemeinen ein eingebettetes System (Subsystem), welches über *Hardware*, darauf laufender *Software* (*Flashware*) und eine *Steuergerät-Konfiguration* (*Codierung*) verfügt (vgl. Abbildung 2.1a). Steuergeräte können mit Sensoren und Aktoren verbunden werden. Sie kommunizieren über Bussysteme miteinander, um verteilte Funktionen zu realisieren [BBK98, KGPW00].

Die Software im Fahrzeug besteht mittlerweile aus mehr als 10 Mio. Zeilen Code [Dai02]. Die *Variantenvielfalt* heutiger Baureihen, also die vom Kunden auswählbaren Fahrzeugausstattungen, führt weiter dazu, dass insgesamt mehr als 270 Steuergerätevarianten (z.B. einfache Klimaanlage und Mehrzonen-Klimaautomatik) parallel entwickelt werden müssen, die sich in Hardware, Software oder Codierung unterscheiden [KS04]. Aus technischer Sicht ist das E/E-Gesamtsystem eines Fahrzeugs ein *äußerst komplexes System*, das aus einer Vielzahl von Elementen mit vielfältigen Beziehungen und Abhängigkeiten besteht [SS01]. Es weist zudem eine hohe Dynamik mit unüberschaubarer Anzahl an Gesamtzuständen auf.

Die steigende Nachfrage nach fortschrittlichen Funktionen im Fahrzeug und neuen Antriebskonzepten führt zu einer weiter zunehmenden Vernetzung von E/E-Steuergeräten und steigender Variantenbildung. Die entstehende kombinatorische Vielfalt der tatsächlich in Fahrzeugen verbauten E/E-Steuergeräte stellt nicht nur hohe Anforderungen an die technische Realisierung, sondern vor allem an den Entwicklungsprozess selbst.

Release-Management in der Elektronik-Entwicklung

Elektronische Komponenten und insbesondere die darauf laufende Software unterscheiden sich erheblich von mechanischen Bauteilen. Klassische Entwicklungsprozesse des Automobilbaus sind in ihrer Komplexität damit nicht anwendbar [BBK98]. Die Software eines Steuergeräts benötigt

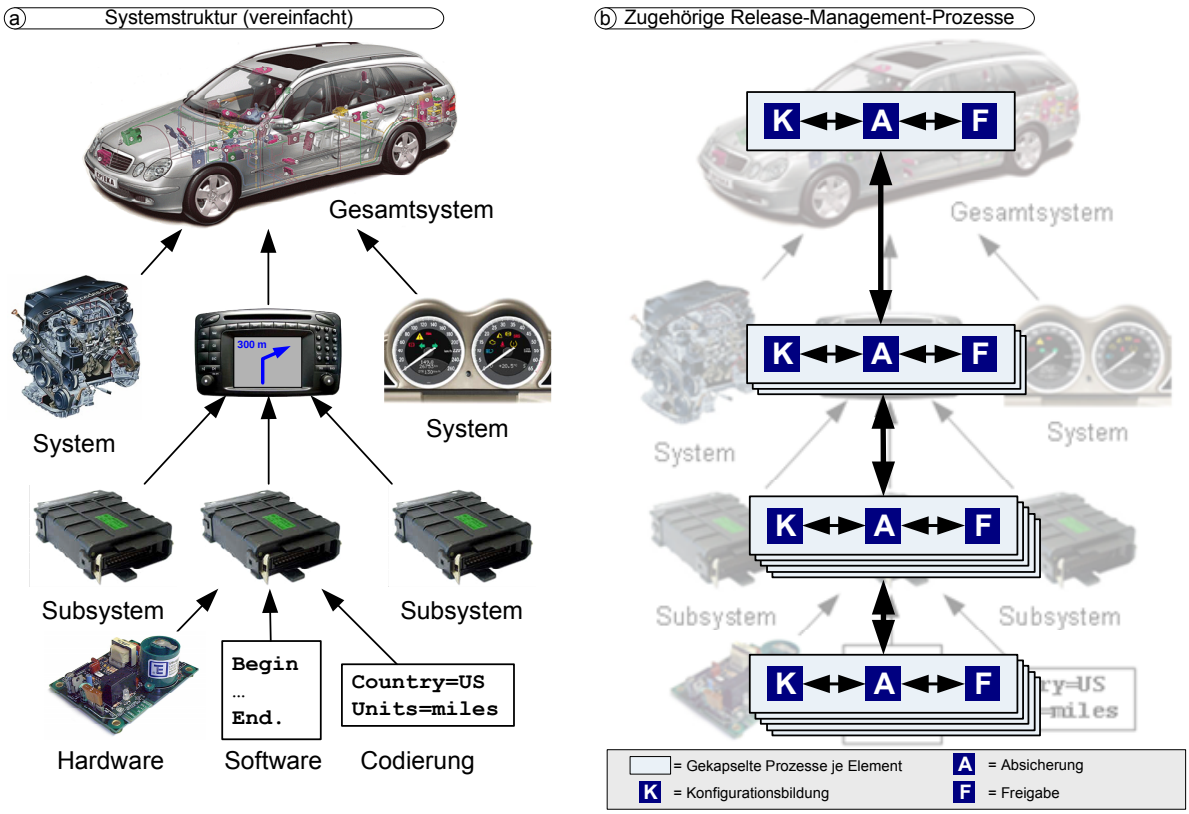


Abbildung 2.1: Bildung, Absicherung und Freigabe einer Systemstruktur

weder eine Vorlaufzeit für die Entwicklung von Produktionsmitteln, noch erzeugt ihre Vervielfältigung Produktionskosten. Dies ermöglicht wesentlich höhere Änderungsraten für elektronische Komponenten als für mechanische Bauteile. Der Vorteil schneller Entwicklungs-, Änderungs- und Einbaumöglichkeiten von Software führt aber zu unterschiedlichen Entwicklungszyklen von Mechanik, Hardware und Software, und darüber hinaus zu häufigem Durchlauf der Absicherungsprozesse und hohem Koordinationsaufwand. Das ist eine große Herausforderung, denn während der Entwicklungsphase kann es pro Tag bis zu 100 Änderungen an den Komponenten des Gesamtsystems geben [Weh00]. Um dennoch das Ziel kurzer Entwicklungszeiten und hoher Produktqualität zu erreichen, ist ein koordiniertes Vorgehen bei der Entwicklung sowie der Integration, Absicherung und Freigabe von E/E-Systemen unerlässlich. Im Automobilbau werden dafür, wie in der Software-Entwicklung, Konzepte des *Release-Managements* angewendet [Weh00, PDAC+01, CAPD03, HFS04, MHHR06]. Zielsetzung ist die *Bildung*, *Absicherung* und *Freigabe* eines E/E-*Release*, also den aktuellen Entwicklungsstand des E/E-Gesamtsystems (vgl. Abbildung 2.1a) als Referenz für weitere Prozesse bereitzustellen [WWA08].

Die Abhängigkeiten zwischen den Komponenten des elektronischen Gesamtsystems reichen von elektrischen über kommunikative, elektromagnetische, geometrische und thermische bis hin zu organisatorischen Abhängigkeiten. Sie gehen über diejenigen klassischer Produktstrukturen hinaus. Wir bezeichnen sie daher als *Systemstrukturen*[1]. Die Abhängigkeiten der Systemstruktur müssen

[1]Wir betrachten im Folgenden die *Systemstruktur* als Spezialisierung des allgemeineren Begriffs der *Produktstruktur* und nutzen ihn, wenn wir auf das elektronische Gesamtsystem im Speziellen verweisen.

sich in der Koordination der Prozesse des Release-Managements widerspiegeln. Ihre Ausführung darf nicht nur auf einzelne E/E-Systeme beschränkt bleiben, sondern muss auch deren Abhängigkeiten in hierarchisch und nicht-hierarchisch aufgebauten Systemstrukturen einbeziehen. Die Prozessstrukturen des Release-Managements sind demzufolge *datengetrieben*.

Die notwendigen Release-Management-Prozesse zur Bildung, Absicherung und Freigabe einzelner Bestandteile der E/E-Systemstruktur müssen nebenläufig (parallel) ausgeführt werden, um im Sinne des *Concurrent Engineering* (d.h. Parallelisierung der Entwicklungsstränge; auch *Simultaneous Engineering* genannt) eine kurze Entwicklungszeit zu garantieren [LK94]. Die Synchronisation der Prozesse ist aber auch wichtig, um ein systematisches Vorgehen im Kontext des Release-Managements zu erlauben (vgl. Abbildung 2.1b). Um etwa ein Release systematisch abzusichern, wird entsprechend anerkannter Entwicklungsstandards (z.B. VDI 2206, DIN EN 61508 oder ISO 26262) vorgegangen. Diese basieren häufig auf dem aus der Software-Entwicklung bekannten *V-Modell* [Drö00, Deu99, Ver04, Rei07, Int08]. Im Rahmen der Absicherung müssen für jede Komponente verschiedene Prozesse durchlaufen werden (z.B. Eingangstest, Prüfung der Kommunikationsschnittstelle, Funktionsprüfung). Um die Komplexität der Absicherung zu reduzieren und Aussagen zum Reifegrad zu präzisieren, werden in der Systemstruktur abhängige *Subsysteme* in *Systemen* integriert und zusätzlich im Verbund abgesichert (vgl. Abbildung 2.1b). Anschließend werden diese Systeme auf Gesamtsystemebene integriert und ebenfalls verschiedenen Absicherungsprozessen unterworfen (z.B. Prüfung der Steuergeräte-Kommunikation, Systemintegrationstest, Echtzeitsimulation und Fahrerprobung), bevor die Freigabe erteilt wird [Krä07]. Die einzelnen Prozesse werden typischerweise bereits in unterschiedlichen prozessorientierten Informationssystemen gesteuert. Die Koordination der gesamten Prozessstruktur erfolgt jedoch noch überwiegend manuell [MHHR06].

Dynamische Aspekte des Release-Managements in der Elektronik-Entwicklung

Entwicklungsprozesse für E/E-Systeme sind, wie bereits beschrieben, sehr dynamisch. Datengetriebene Prozessstrukturen müssen daher vor allem eine schnelle Integration von geänderten Komponenten unterstützen und den Koordinationsaufwand minimieren um eine hohe Qualität des elektronischen Gesamtsystems sicherzustellen [WWA08]. Prozessseitig müssen dafür Änderungen auf ihre Auswirkungen innerhalb der Prozessstruktur untersucht und ggf. entsprechende Maßnahmen für die Behandlung der Auswirkungen durchgeführt werden. So kann es beispielsweise aus fachlichen Gründen vorkommen, dass ein Subsystem während der Absicherungsphase aus dem Release entfernt wird (z.B. weil es Fehler aufweist). In einem solchen Fall müssen alle davon betroffenen Prozessbeteiligten informiert und ggf. laufende Prozesse abgebrochen werden, um Ressourcen zu sparen und damit unnötige Kosten zu vermeiden. Wird ein Subsystem „nur“ ausgetauscht, müssen in der Regel die Absicherungsprozesse auf der System- und Gesamtsystemebene erneut ausgeführt werden. Dabei muss gewährleistet sein, dass nur für die betroffenen Elemente die Absicherungsprozesse erneut ausgeführt werden und nicht für unbeteiligte Subsysteme bzw. Systeme. Ohne eine geeignete IT-Unterstützung sind derartige Auswirkungen in großen datengetriebenen Prozessstrukturen mit komplexen Abhängigkeiten nicht mehr überschaubar.

Im Kontext der dynamischen Änderung der Systemstruktur müssen die Prozess-Verantwortlichen jederzeit in der Lage sein, flexibel auf Ausnahmen zu reagieren. Befinden sich bereits Fahrzeug-Prototypen auf der Fahrerprobung, soll beispielsweise der Austausch eines fehlerhaften Subsys-

tems nicht zu einem erneuten Durchlauf aller Testfälle führen, wenn die Kosten für den Transfer der Fahrzeuge und der Aufwand für den Austausch wirtschaftlich nicht sinnvoll sind. Die Entscheidung dafür muss individuell auf Basis der fachlichen Situation getroffen werden. Die Abschätzung der Konsequenzen eines solchen Austauschs auf andere Prozesse ist zwar äußerst wichtig, aber aufgrund der Vielzahl an Abhängigkeiten manuell kaum möglich.

Aus Gründen der Produkthaftung ist eine lückenlose Dokumentation des Entwicklungsprozesses notwendig (z.B. funktionale Sicherheit in der Elektronik-Entwicklung nach DIN EN 61508 oder IS0 26262). Dazu gehört auch die Protokollierung des Verlaufs einer Prozessstruktur. Dies geschieht im Kontext der manuellen Koordination in der Regel nicht automatisch, sondern muss ebenfalls manuell durchgeführt werden.

Zusammenfassend ist das Ziel einer Prozess-Unterstützung bei der Entwicklung elektronischer Komponenten, die Gesamtheit der Prozesse für alle Elemente (z.B. Subsysteme und Systeme) möglichst gut zu koordinieren. Die Untersuchung der Release-Management-Prozesse hat gezeigt, dass deren Koordination durch eine IT-seitige Steuerung und Überwachung verbessert werden kann [MHHR06]. Die Abbildung der benötigten Prozessstrukturen wird von aktuellen Software-Technologien (z.B. Workflow-Management-Systeme) allerdings nicht adäquat unterstützt (vgl. Kapitel 3). Sie bieten keine ausreichende Unterstützung für die Integration von Produktdaten und Prozessen und für die Erstellung, Koordination und dynamische Adaption datengetriebener Prozessstrukturen [MHHR06, MRH06]. IT-Unterstützung ist für diese Art von Prozessstrukturen jedoch unverzichtbar, denn unvorhergesehene bzw. unzureichend behandelte Ausnahmen können zu Verzögerungen im Entwicklungsprozess, bis hin zu einer nicht ausreichend abgesicherten Produktqualität führen.

2.2 Anforderungen an das Prozess-Management

Aus der beschriebenen Anwendungsdomäne lassen sich Anforderungen an eine IT-Unterstützung datengetriebener Prozessstrukturen ableiten [BHR05, MHHR06, MRH06]. Die Anforderungsanalyse (vgl. Abschnitt 2.1) verdeutlicht, dass die Entwicklung komplexer Produkte die Koordination vieler Einzelprozesse erfordert. Häufig sind die Prozesse bereits in bestehenden Informationssystemen realisiert. Ziel ist es nicht, die bestehenden Prozesse und Informationssysteme zu ersetzen oder zu verändern, sondern deren übergreifende Koordination über den gesamten Lebenszyklus einer datengetriebenen Prozessstruktur zu ermöglichen.

Konzepte zur Unterstützung datengetriebener Prozessstrukturen müssen Funktionalitäten aus dem *Produktdaten-Management (PDM)* und *Prozess-Management* integrieren. PDM unterstützt die Verwaltung von Bauteilen inklusive sämtlicher Entwicklungsdokumente (z.B. technische Spezifikation von Bauteilen, Software-Modelle). Des Weiteren erlaubt PDM durch geeignetes Konfigurationsmanagement die Bildung von Konfigurationen und die Beschreibung von Abhängigkeiten zwischen ihnen. Ferner unterstützt PDM ihre *Versionierung* und *Variantenbildung* [CAPD03]. Die Integration von PDM und Prozess-Management muss sicherstellen, dass bei der Koordination der einzelnen Prozesse die prozessrelevanten Abhängigkeiten zwischen Objekten des PDM-Systems berücksichtigt werden.

Die beiden folgenden Abschnitte behandeln zunächst nicht-funktionale Anforderungen an die Prozess-Management-Konzepte, um anschließend eine integrierte Sicht auf die funktionalen Anforderungen zu geben. Diese werden in Teil II dieser Arbeit weiter detailliert.

2.2.1 Nicht-funktionale Anforderungen

Prozesse zur Verarbeitung von Geschäftsobjekten, unabhängig davon ob es sich um physikalische Komponenten oder um Entwicklungsdokumente handelt, tragen in hohem Maß zur Wertschöpfung eines Unternehmens bei. Um dieser Verantwortung im produktiven Einsatz Rechnung zu tragen, müssen die Konzepte zur Unterstützung großer datengetriebener Prozessstrukturen verschiedene nicht-funktionale Anforderungen erfüllen. Dazu gehören Aspekte der Prozessqualität, Wiederverwendung, Skalierbarkeit und Nachvollziehbarkeit.

Prozessqualität

Die Qualität des Entwicklungsprozesses für ein Produkt wird durch Prozesskosten (z.B. Dauer und Ressourcenaufwand) und die resultierende *Produktqualität* ausgedrückt [Rei02]. Voraussetzung für geringe Prozesskosten für die Ausführung großer Prozessstrukturen ist die Minimierung von Verzögerungen bei der Ausführung, auch in Ausnahmesituationen. Auf technischer Ebene zeichnet sich eine hohe Prozessqualität durch Verklemmungsfreiheit über den gesamten Prozess-Lebenszyklus aus. Voraussetzung hierfür sind die Definition geeigneter Korrektheitskriterien (vgl. Abschnitt 1.1) und die Entwicklung von Mechanismen zur Erkennung inkorrekter Prozessstrukturen (z.B. im Kontext der dynamischen Adaption). Die strukturelle Analysierbarkeit von Prozessstrukturen ist dafür genauso essentiell wie die Bereitstellung aktueller Laufzeitinformationen.

Im Gegensatz zur technischen Prozessqualität wird die Produktqualität maßgeblich durch die fachliche Ausgestaltung der Entwicklungsprozesse beeinflusst. Voraussetzung hierfür ist eine geeignete Ausdrucksmächtigkeit des Metamodells (vgl. Abschnitt 1.1).

Wiederverwendung

Prozesse einer Prozessstruktur sind autonome und kombinierbare Bestandteile, die isoliert benutzt und integriert werden können. Die *Wiederverwendung* von Prozessen bzw. Prozessfragmenten ist eines der Ziele des Prozess-Managements (vgl. Abschnitt 1.1). Dieser Aspekt ist im Rahmen der Unterstützung großer datengetriebener Prozessstrukturen von hoher Bedeutung, da deren Modellierung und Adaption sehr aufwändig sind. Die lose Kopplung und Modularisierung der einzelnen Prozesse einer Prozessstruktur ist für die Wiederverwendung essentiell.

Skalierbarkeit

Die Reduktion der Modellierungsaufwände für datengetriebene Prozessstrukturen muss Ziel bei der Entwicklung geeigneter IT-Konzepte sein. Diese *Skalierbarkeit* muss nicht nur für die Modellierung gelten, sondern auch für die Ausführung. So sollen zum Beispiel Korrektheitsanalysen

während der dynamischen Adaption einer datengetriebenen Prozessstruktur oder während der Ausnahmebehandlung ad-hoc durchgeführt werden können.

Nachvollziehbarkeit

Nachvollziehbarkeit ist eine wichtige Eigenschaft, die Prozessstrukturen über den gesamten Prozesslebenszyklus mitbringen müssen. Zunächst ist die Einfachheit des Modells mit klarer operationaler Semantik die wichtigste Voraussetzung für eine einfache Benutzbarkeit. Sie stellt sicher, dass die Ausführung durch eindeutige Regeln für die Koordination von Prozessstrukturen nachvollziehbar ist. Dazu gehört, dass während der Ausführung jederzeit ein eindeutiger Stand der Prozessstruktur angezeigt wird. Ferner müssen im Rahmen der Ausnahmebehandlung Inkonsistenzen erkannt, Konsequenzen abgeleitet und entsprechende Behandlungsmethoden vorgeschlagen werden.

Während der Ausführung ist es auch unbedingt notwendig, den *Verlauf* der Ausführung aufzuzeichnen, um die Analysierbarkeit der Abläufe zu gewährleisten. Dies geschieht nicht nur mit dem Ziel, die Prozessstruktur an sich zu optimieren, sondern auch, um im Rahmen der Produkthaftung die Ausführung von Prozessen zu dokumentieren.

2.2.2 Funktionale Anforderungen

Basierend auf den nicht-funktionalen Anforderungen definieren wir konkrete funktionale Anforderungen. Deren Ziel ist es, die Elemente einer Produktstruktur mit Prozessen in Beziehung zu setzen und die Abhängigkeiten zwischen den Elementen für die Koordination der Prozesse zu nutzen (vgl. Abbildung 1.3 auf Seite 6). Abbildung 2.2 gibt einen Überblick über die Anforderungen großer Prozessstrukturen, orientiert an deren Phasen Modellierung, Ausführung, Adaption und Ausnahmebehandlung. Im Folgenden beschreiben wir die funktionalen Anforderungen jeder Phase. Sie werden in Teil II dieser Arbeit weiter detailliert, um jeweils davon ausgehend geeignete Lösungskonzepte zu erarbeiten.

Modellierung

Die *Modellierung* einer datengetriebener Prozessstruktur verfolgt zwei Ziele: erstens die Definition der Abfolge der Prozesse für einzelne Komponenten der Produktstruktur und zweitens die übergreifende Synchronisation der Prozesse verschiedener Komponenten (vgl. Abbildung 1.3).

Die Abläufe der Prozesse für die einzelnen Komponenten entsprechen im Allgemeinen nicht nur einfachen Sequenzen, sondern stellen vielmehr komplexe Abfolgen der Prozesse mit bedingten Verzweigungen und Rücksprüngen dar. Die einzelnen Prozesse müssen dabei als *Black Boxes* betrachtet werden, denn sie sind bereits in verschiedenen prozessorientierten Informationssystemen integriert und sollen prinzipiell nicht verändert werden. Daher nehmen wir an, dass Prozesse als Bausteine zur Bearbeitung einzelner Komponenten eingesetzt werden können und jeweils nur ihr Start synchronisiert bzw. auf ihr Ende gewartet werden muss. Die Abfolge der Prozesse soll

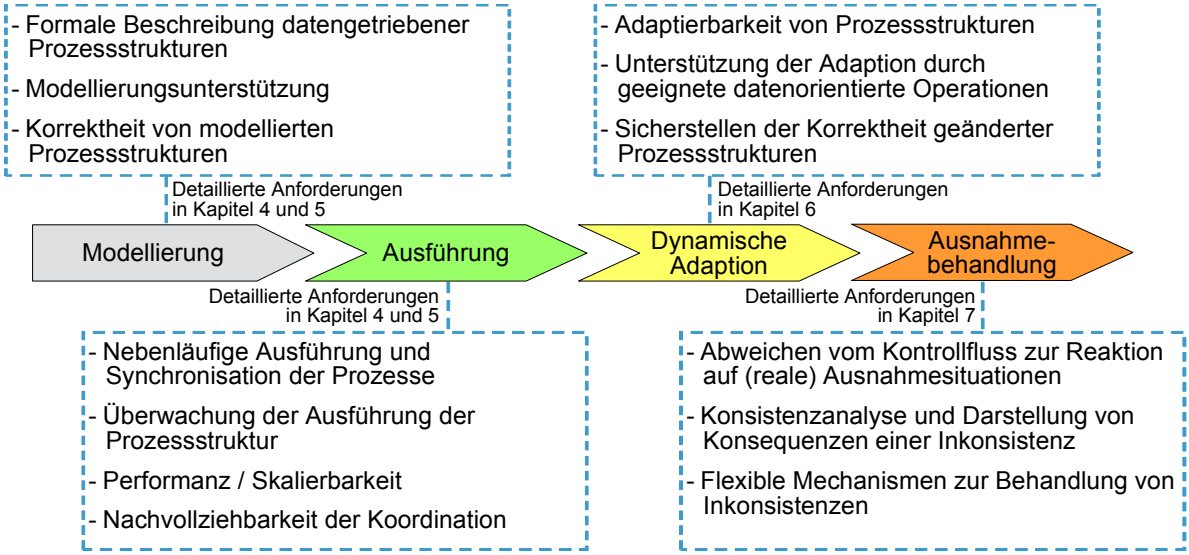


Abbildung 2.2: Anforderungen an die IT-Unterstützung für datengetriebene Prozessstrukturen

zunächst je Komponente modelliert werden. Zum einen können so die Abläufe in einem überschaubaren Rahmen definiert werden, zum anderen erhöht sich dadurch ihre Wiederverwendbarkeit und verbessert die Strukturierbarkeit der entstehenden Prozessstrukturen (vgl. Abbildung 1.3).

Die Synchronisation der Prozesse verschiedener Elemente der Produktstruktur soll nicht in Form einfacher Synchronisationskanten geschehen. Stattdessen sollen im Rahmen der Modellierung die Abhängigkeiten von Objekten der Produktstruktur konsequent ausgenutzt werden. Dementsprechend müssen die Abhängigkeiten von Objekten der Produktstruktur mit Synchronisationen der jeweiligen Prozesse in Beziehung gesetzt werden können. Hinzu kommt, dass für die Synchronisation selbst wiederum Prozesse ausgeführt werden müssen (z.B. *Synchronisiere die Prozesse A von Subsystem X nicht mit Prozess B von System Y direkt, sondern führe zuvor den Prozess Installation aus*).

Produktstrukturen sind grundsätzlich variabel und können sich für jede zu erstellende Prozessstruktur unterscheiden. So können beispielsweise zwei aufeinanderfolgende Releases eines Fahrzeugs nicht nur aufgrund unterschiedlicher Versionen der Komponenten voneinander abweichen, sondern auch durch variierende Ausstattungsvarianten von Fahrzeugen und damit den tatsächlich verbauten Komponenten. Die zugehörigen Prozessstrukturen müssen bei der Modellierung, im Gegensatz zur Instanziierung traditioneller aktivitätenorientierter Prozesse bzw. Prozessstrukturen mit ihrem *festgelegten* Schema (vgl. Abschnitt 1.1), immer für die aktuelle Produktstruktur individuell konfiguriert bzw. erzeugt werden. Sie unterscheiden sich damit von Instanz zu Instanz. Eine manuelle Modellierung oder Anpassung des Schemas der Prozessstruktur an die geänderte Produktstruktur ist äußerst aufwändig. Die entstehende Komplexität kann nur durch eine geeignete *Modellierungsunterstützung* gehandhabt werden, die die Prozessstruktur automatisch konfiguriert bzw. erstellt. Diese Unterstützung soll nicht nur den Modellierungsaufwand reduzieren, sondern auch sicherstellen, dass die modellierten Prozessstrukturen korrekt sind.

Ausführung

Die manuelle Koordination großer Prozessstrukturen ist auf mittlere Sicht zu komplex. Ziel von CoRePro ist es daher, nicht nur die Modellierung datengetriebener Prozessstrukturen zu ermöglichen, sondern auch deren *Ausführung* IT-seitig zu unterstützen. Die Herausforderung bei der Ausführung großer Prozessstrukturen besteht in der nebenläufigen Koordination zahlreicher voneinander mehr oder weniger unabhängiger Einzelprozesse. Hierbei gilt es die Ausführung der Prozesse entsprechend des Modells der Prozessstruktur zu synchronisieren. Eine eindeutige operationale Semantik und geeignete Korrektheitskriterien für datengetriebene Prozessstrukturen sollen eine hohe Prozessqualität sicherstellen (vgl. Abschnitt 2.2.1).

Prozessstrukturen sollen sich wieder hierarchisch als Prozessschritt in einen übergeordneten Prozess integrieren lassen (vgl. Abschnitt 1.1). Hierfür müssen sie während der Ausführung nicht nur einen klaren Status besitzen, sondern auch die Korrektheitskriterien des übergeordneten Prozesses erfüllen (z.B. eindeutiger Start und eindeutiges Ende).

Weitere Aspekte der Ausführung sind eine geeignete Überwachung der Prozesse sowie die Dokumentation ihres Verlaufs. Dies ist nicht nur für die Nachvollziehbarkeit der Ausführung wichtig (vgl. Abschnitt 2.2.1), sondern auch für die Korrektheitsanalyse im Rahmen der dynamischen Adaption.

Adaption

Die untersuchten Prozessstrukturen haben eine sehr lange Ausführungsdauer. Kommt es während der Ausführung zu einer Änderung der Produktstruktur, ist eine dynamische *Adaption* der Prozessstruktur notwendig. Im Gegensatz zur Modellierung, die den Aufbau der gesamten Prozessstruktur betrachtet, soll die Struktur bei der dynamischen Adaption (auf Instanzebene) gezielt angepasst werden können, ohne sie im Gesamten erneut zu erzeugen. So müssen beispielsweise Prozesse, deren bearbeitete Komponenten aus der Produktstruktur entfernt werden, terminiert bzw. ihre weitere Ausführung verhindert werden. Für hinzugefügte Komponenten muss die Prozessstruktur um entsprechende Prozesse und geeignete Synchronisationen erweitert werden. Im Sinne einer hohen Prozessqualität streben wir an, nur diejenigen Prozesse der Prozessstruktur zu manipulieren, deren Ergebnisse durch die Adaption beeinflusst werden.

Nach Durchführung der dynamischen Adaption ist es von höchster Wichtigkeit, dass die resultierende Prozessstruktur weiterhin den definierten Korrektheitsansprüchen genügt. Dies ist nicht trivial, denn unsachgemäß entfernte oder eingefügte Synchronisationen können beispielsweise Verklemmungen in der Prozessstruktur hervorrufen (vgl. Abschnitt 1.1). Hierfür müssen die Korrektheitskriterien der Ausführung geeignet erweitert werden, um die Adaption im Falle einer korrektheitsgefährdenden Veränderung der Prozessstruktur entweder zu verhindern oder mithilfe einer geeigneten Ausnahmebehandlung durchführen zu können.

Ausnahmebehandlung

Die Dynamik der Entwicklungsprozesse sowie deren lange Ausführungsdauer führen dazu, dass *Ausnahmen*, die während der Ausführung datengetriebener Prozessstrukturen auftreten, geeig-

net behandelt werden müssen. Unter Ausnahmen verstehen wir korrektheitsgefährdende Änderungen im Rahmen der dynamischen Adaption sowie fachliche Ausnahmen, die in der „realen Welt“ erkannt werden und eine Abweichung vom modellierten Kontrollfluss erfordern (z.B. die Wiederholung eines Prozesses). Es ist hierbei von höchster Wichtigkeit, Ausnahmen rechtzeitig zu erkennen und sie geeignet zu kontrollieren, um jederzeit ein definiertes und vorhersagbares Verhalten sowie die verklemmungsfreie Ausführung der Prozessstruktur zu garantieren.

Ausnahmen können in einer Prozessstruktur weitreichende Konsequenzen nach sich ziehen. Eine geeignete Prüfung der Abhängigkeiten zwischen Prozessen hilft, die Auswirkungen von Änderungen oder Fehlersituationen zu erkennen. Den Nutzern werden damit Mechanismen zur Verfügung gestellt, die sie in ihren fachlichen Entscheidungen zur Ausnahmebehandlung unterstützen. Geeignete Reaktionsmöglichkeiten sind beispielsweise das Wiederherstellen des letzten korrekten Zustands der Prozessstruktur (*Backward Recovery*), die Wiederholung eines oder mehrerer Prozesse zur Herstellung eines neuen korrekten Zustands der Prozessstruktur (*Forward Recovery*) und das Ignorieren einer Ausnahme. Unabhängig von der gewählten Ausnahmebehandlungsstrategie muss auch in diesem Rahmen die Korrektheit (z.B. Verklemmungsfreiheit) der Prozessstruktur jederzeit sichergestellt werden.

2.3 Datengetriebene Prozessstrukturen in weiteren Domänen

Datengetriebene Prozessstrukturen kommen nicht nur in der Entwicklung elektronischer Systeme vor. Wir diskutieren im Folgenden weitere Anwendungsbeispiele, in denen eine Unterstützung datengetriebener Prozessstrukturen erforderlich ist. Sie unterstreichen ganz oder teilweise die in Abschnitt 2.2 definierten Anforderungen.

2.3.1 Anwendungsbeispiel I: Fahrzeugentwicklung

Der Entwicklungsprozess einer *Baureihe* (z.B. E-Klasse und S-Klasse) ist bei Automobilherstellern durch einen gesamtheitlichen Prozess definiert [Mai04, Büt06]. Es handelt sich dabei um einen sehr lang laufenden Prozess (d.h. mehrere Jahre), der die Entwicklungsprozesse von mechanischen und elektronischen Komponenten über die verschiedenen Entwicklungsphasen koordiniert. Die Prozesse werden auf Basis so genannter *Quality-Gates*, also Meilensteine während der Entwicklung, synchronisiert.

Die hohe Variantenvielfalt der Modelle und Ausstattungen heutiger Fahrzeuge (z.B. Limousine mit vier Türen, Coupé mit zwei Türen) erfordert auch im mechanischen Bereich eine individuelle Anpassung des gesamtheitlichen Entwicklungsprozesses. Stellhebel für die Erzeugung dieses Prozesses sind unter anderem die Fahrzeugvarianten und damit die zugeordneten Komponenten der Produktstruktur. Auf deren Basis werden die Prozesse ausgewählt, die im Gesamtprozess zur Entwicklung der gewünschten Baureihe ausgeführt werden müssen. Es handelt sich damit ebenfalls um datengetriebene Prozessstrukturen.

Der gesamtheitliche Entwicklungsprozess wird für verschiedene Baureihen und Projektformen (z.B. Neuentwicklung oder Modellpflege der Baureihe) manuell angepasst [Mai04, Büt06]. Auch in dieser Anwendung ist die manuelle Anpassung der Prozessstruktur jedoch sehr aufwändig.

Weitere Herausforderungen im Rahmen der Adaption sind die Sicherstellung der Konsistenz der modellierten Prozessstrukturen sowie die dynamische Adaption unter Wahrung gängiger Entwicklungsstandards. Die Simulation und Überwachung der Prozessstruktur sind weitere Anforderungen an die Prozessunterstützung.

2.3.2 Anwendungsbeispiel II: Software-Entwicklung

Die Komplexität der E/E-Entwicklung entsteht durch die Vernetzung der Systeme und deren Zusammenspiel. Die klassische Software-Entwicklung steht vor ähnlichen Herausforderungen. Hier wurde bereits früh erkannt, dass die Komplexität großer Software-Systeme nur durch ein systematisches Vorgehen kontrollierbar ist (z.B. *Wasserfall-*, *Spiral-* und *V-Modell* bzw. *V-Modell-XT*)[Som07]. Während sich die Vorgehensmodelle auf die Durchführung einzelner Projekte beziehen, beschreiben Prozess-Qualitäts-Standards (z.B. *Capability Maturity Model* [Som07]) die notwendige Organisation für die Durchführung von Software-Entwicklungsprojekten (z.B. Einführung von Konfigurations-Management).

Ziel der Vorgehensmodelle ist es, die Komplexität der Entwicklung großer Software-Anwendungen systematisch koordinieren zu können. Dies wird erreicht, indem – wie bei der E/E-Entwicklung – die Anwendung entsprechend ihrer Architektur in Systeme zerlegt und diese unabhängig voneinander entwickelt und getestet werden (vgl. Abbildung 2.3). Anschließend folgt ein Integrationsprozess, der die Systeme integriert und im Verbund testet [BR05]. Dies kann entweder in einem Schritt (*Big Bang*) oder in verschiedenen Integrationsstufen geschehen [Som07]. Die dadurch entstehende Prozessstruktur ist durch die Architektur der Software-Anwendung bzw. deren Zerlegung bestimmt und damit datengetrieben.

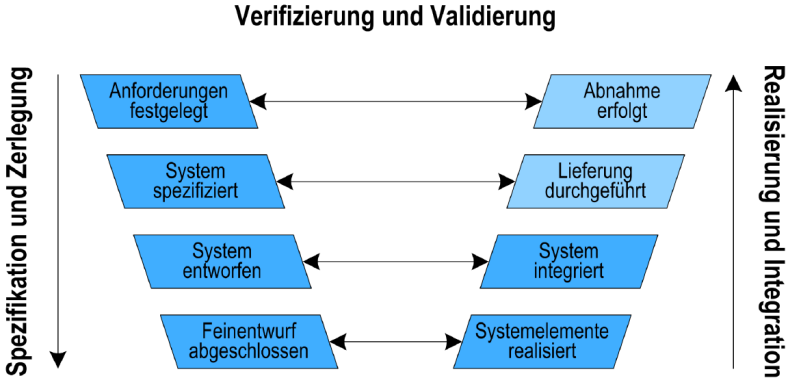


Abbildung 2.3: Software-Entwicklung entsprechend des V-Modell-XT [V-M05]

2.3.3 Anwendungsbeispiel III: Finanzsektor

Datengetriebene Produktstrukturen finden sich nicht nur in technischen Produkten, sondern auch im Dienstleistungssektor. In der Finanzbranche werden beispielsweise Produktstrukturen für die Erstellung von Angeboten genutzt (*Financial Planning*). Hier werden aus standardisierten Produktbausteinen, die zum Teil von verschiedenen Fremdanbietern (z.B. Versicherungsgesellschaften) bereit gestellt werden, individuelle Produkte zusammengestellt [WFJ+03, FWM+04]. Eine

kundenspezifische Produktkonfiguration ist beispielsweise ein Ausbildungssparplan (vgl. Abbildung 2.4). Er stellt ein vorstrukturiertes Angebot seitens einer Bank dar, das auf die finanzielle Vorsorge für die Ausbildung eines Kindes ausgerichtet ist. Gleichzeitig enthält der Ausbildungssparplan eine Absicherung gegen Risiken, deren Eintreten den Eltern eine Fortsetzung der vorgesehenen Ansparleistungen aus bestimmten Gründen nicht mehr erlauben würden. Demzufolge setzt sich die Produktstruktur aus einer Anlage- und einer Versicherungskomponente zusammen. Aufgrund seiner modularen Struktur kann der Ausbildungssparplan flexibel konfiguriert werden, indem verschiedene Ausprägungen seiner Teilkomponenten integriert werden.

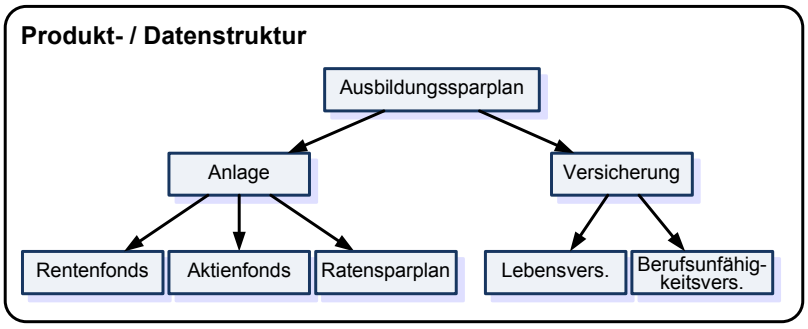


Abbildung 2.4: Ausschnitt einer Produktstruktur aus der Finanzbranche

Es existieren bereits Ansätze, die die Nutzung derartiger Produktstrukturen zur Steuerung von Prozessen beschreiben. So kann aus einer Struktur der entsprechende Erstellungsprozess für den Finanzsparplan abgeleitet werden, also beispielsweise die Reihenfolge der abzuschließenden Verträge [Rei02]. Der Erstellungsprozess umfasst dann die notwendigen Aktivitäten zur Erfassung oder Aufbereitung der für das Produkt erforderlichen Komponenten bzw. Informationen.

Die dynamische Adaption der Prozessstrukturen nimmt im Rahmen des *Financial Planning* eine genauso wichtige Rolle ein, wie die Ausnahmebehandlung. Wird beispielsweise die Produktstruktur angepasst, weil bestimmte Rahmenbedingungen nicht mehr gültig sind (z.B. Korrektur des Kreditrahmens), muss die Prozessstruktur zur Erfassung der Daten ebenfalls angepasst werden. Auch hier ist es wünschenswert, die bereits erstellten Informationen so weit wie möglich zu erhalten, also entsprechende Prozesse nur dann erneut auszuführen, wenn deren Ergebnisse die Bildung der adaptierten Struktur beeinflussen.

2.4 Zusammenfassung

Unsere Problemanalyse hat gezeigt, dass die Entwicklungsprozesse für die Fahrzeugelektronik eine umfassende Unterstützung datengetriebener Prozessstrukturen erfordern. Datengetriebene Prozessstrukturen zeichnen sich dadurch aus, dass sie hunderte bis tausende individuelle Prozesse enthalten, die entsprechend einer Produktstruktur miteinander verknüpft werden müssen, um einen logischen Gesamtablauf zu gewährleisten. Die manuelle Modellierung und Koordination datengetriebener Prozessstrukturen, wie sie derzeit praktiziert wird, ist äußerst aufwändig und fehleranfällig. Die Konzepte müssen jedoch nicht nur eine geeignete Modellierungsunterstützung für datengetriebene Prozessstrukturen bieten, sondern auch ihre Ausführung und dynamische Adaption unterstützen. Im Rahmen der Ausnahmebehandlung sind von den Konzepten angemessene

Reaktionen bei auftretenden Problemen (z.B. Verzögerung der Freigabe eines Subsystems) gefordert. Hierbei müssen die nicht-funktionalen Anforderungen Prozessqualität, Wiederverwendung, Skalierbarkeit und Nachvollziehbarkeit berücksichtigt werden.

Die Untersuchung weiterer Domänen zeigt, dass die Anforderungen an eine systematische Unterstützung von Modellierung, Ausführung, dynamischer Adaption und Ausnahmebehandlung im Kontext datengetriebener Prozessstrukturen nicht spezifisch für die Entwicklung technischer Produkte sind, sondern zum Beispiel auch für Dienstleistungen der Finanzbranche gelten.

Auf Basis der definierten Anforderungen werden im folgenden Kapitel existierende Ansätze untersucht und auf ihre Fähigkeit, datengetriebene Prozessstrukturen zu koordinieren, geprüft.

3

Stand der Technik

Die Anforderungsanalyse aus Kapitel 2 zeigt, dass ein durchgängiger Ansatz für die Unterstützung datengetriebener Prozessstrukturen über alle Phasen ihres Lebenslaufs (d.h. Modellierung, Ausführung, Adaption und Ausnahmebehandlung; vgl. Abbildung 2.2 auf Seite 18) notwendig ist. In diesem Kapitel untersuchen wir, ob und wie gut sich datengetriebene Prozessstrukturen mit bestehenden Ansätzen abbilden lassen. Wir stellen relevante Ansätze sowohl aus der Praxis als auch der Wissenschaft vor und führen eine erste Analyse hinsichtlich deren Erfüllung der Anforderungen aus Abschnitt 2.2 durch. Das Hauptaugenmerk liegt hierbei auf der Durchgängigkeit der Unterstützung datengetriebener Prozessstrukturen (vgl. Abschnitt 1.1). Die Auswahl der Ansätze orientiert sich am Grad der Verfügbarkeit der Ansätze und reicht von Anforderungsmustern über Standards, klassische und flexible Prozess-Management-Systeme bis hin zu wissenschaftlichen Ansätzen (vgl. Abbildung 3.1). Weitere Ansätze und Detailaspekte werden im Rahmen der technischen Lösung in Teil II dieser Arbeit diskutiert.

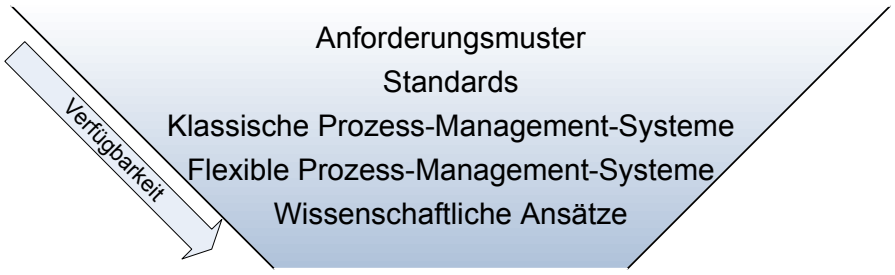


Abbildung 3.1: Verfügbarkeit der untersuchten Konzepte

Das Kapitel gliedert sich wie folgt: Abschnitt 3.1 stellt Ansätze und Produkte aus der Praxis für das Management von Prozessen vor und diskutiert deren Unterstützung datengetriebener

Prozessstrukturen. Abschnitt 3.2 präsentiert wissenschaftliche Konzepte, die über den Funktionsumfang von Ansätzen aus der Praxis hinausgehen und diskutiert deren Abdeckung der Anforderungen für das Management datengetriebener Prozessstrukturen. Abschnitt 3.3 gibt eine Zusammenfassung des Kapitels.

3.1 Ansätze aus der Praxis

Wir beginnen die Evaluierung des Stands der Technik mit existierenden Prozess-Management-Werkzeugen aus der Praxis. Für das Prozess- bzw. Workflow-Management stehen mittlerweile über 100 Produkte zur Verfügung und im wissenschaftlichen Umfeld finden sich noch weitere Ansätze [DAH05]. Grundlage vieler Werkzeuge sind standardisierte Modellierungssprachen, wie die *Unified Modeling Language* (UML) und die *Business Process Modeling Notation* (BPMN).[1] Kommerziell erfolgreiche Systeme, wie die *WebSphere Series* von *IBM* (mehr als 1.500 produktive Installationen), verfügen über ausgeprägte Modellierungsfunktionen mit Unterstützung der genannten Modellierungssprachen [BD05, SW08]. Die *WebSphere Series* enthält mit dem *Process Server* darüber hinaus eine Ausführungskomponente für modellierte Prozesse. Des Weiteren existieren mit der *BPM Suite* von *AristaFlow* und *FLOWer* von *Pallas Athena* mittlerweile Systeme auf dem kommerziellen Markt, die ihren Ursprung und ihre technischen Grundlagen in wissenschaftlichen Arbeiten haben und einen erweiterten Funktionsumfang bieten.

Wir betrachten außerdem Systeme, die eine ausgeprägte Produktdaten-Management-Funktionalität aufweisen. In technischen Domänen, wie der Automobilentwicklung, werden Produktstrukturen in Produktdaten-Management-Systemen (PDM-Systeme) verwaltet. Sie erlauben die Verwaltung von Produktinformationen, die für die Entwicklung (z.B. CAD-Modelle und Entwicklungsdokumente) oder Produktion (z.B. Montagepläne und Schweißpunkte) und darauf folgende Produktphasen notwendig sind. Diese Systeme verfügen mittlerweile über Prozess-Komponenten mit Modellierungs- und Ausführungsfunktionen. Ein wichtiger Vertreter der PDM-Systeme, der auch in der E/E-Entwicklung verschiedener Automobilhersteller Anwendung findet, ist *Teamcenter Engineering* von *UGS* bzw. *Siemens*.

Abbildung 3.2 zeigt eine grobe Einordnung der untersuchten Ansätze aus der Praxis. Wir unterscheiden den Grad der Datenorientierung und der Unterstützung von Prozess-Management-Konzepten, wobei in der Regel die Unterstützung zunimmt (z.B. unterstützen Ansätze mit Funktionen zur Ausnahmebehandlung auch die Prozessmodellierung). Die Abgrenzung ist jedoch nicht immer trennscharf; Tendenzen sind durch Darstellung des Ansatzes im Randbereich des jeweiligen Rasters visualisiert und in den folgenden Abschnitten beschrieben.

3.1.1 Anforderungsmuster

Mehrere Initiativen der Wissenschaft haben auf Basis sogenannter *Patterns* (*Muster*) in den vergangenen Jahren verschiedene Standards und Produkte auf ihre Unterstützung von Prozess-Management-Konzepten untersucht [AHKB03, RHEA04, RHEA05, RHAM06, RAH06, WRR07,

[1] *Ereignisgesteuerte Prozessketten* (EPK) sind in der Prozessmodellierung ebenfalls verbreitet. Sie dienen hauptsächlich der Beschreibung betriebswirtschaftlicher Abläufe und werden in dieser Arbeit daher nicht betrachtet.

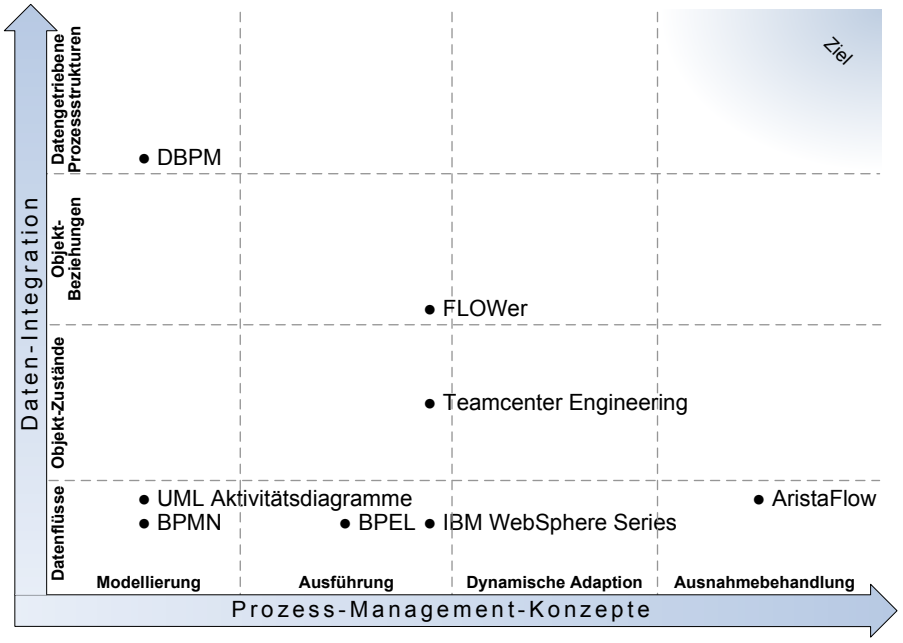


Abbildung 3.2: Einordnung der Ansätze aus der Praxis

WRRM08]. Hierfür wurden verschiedene Kriterienkataloge erstellt, auf deren Basis die Ausdrucksmächtigkeit und Flexibilität von Prozess-Management-Systemen geprüft werden kann. Die Kriterien beschreiben generische Muster (z.B. die *Sequentielle Anordnung von Aktivitäten* und die *exklusive Verzweigung* im Kontrollfluss) unabhängig von einem bestimmten Prozess-Metamodell. Sie erlauben damit eine objektive Evaluierung von Standards und Produkten.

Wir prüfen zunächst, inwiefern sich die Muster zur Abdeckung der in Kapitel 2 definierten Szenarien und Anforderungen eignen. Auf dieser Basis kann eine erste Abschätzung zur Unterstützung von datengetriebenen Prozessstrukturen durch die in [AHKB03, RHEA04, RHEA05, RHAM06, RAH06, WRR07, WRRM08] untersuchten Standards und Prozess-Management-Systeme getroffen werden. Relevante Ansätze aus diesem Umfeld werden wir in den folgenden Abschnitten noch genauer untersuchen.

Workflow (Control-Flow) Patterns

Eine umfangreiche Untersuchung der Ausdrucksmächtigkeit von Metamodellen verschiedener Standards und kommerzieller Prozess-Management-Systeme bieten *Workflow Patterns* bzw. *Workflow Control-Flow Patterns* [AHKB03, RHAM06]. Ziel der Untersuchung ist, die Kontrollflusskonstrukte der verschiedenen Ansätze hinsichtlich der Modellierbarkeit unabhängiger aktivitätenorientierter Prozessfragmente (z.B. *Sequenz* und *bedingte Verzweigung*) zu prüfen. Darauf basierend können die verschiedenen Standards und Systeme (z.B. UML, BPMN, *IBM WebSphere MQ Workflow* und *FLOWer*) evaluiert werden. Hierbei wird nicht nur Wert auf die verfügbaren Modellierungskonstrukte gelegt, sondern auch eine präzise operationale Semantik erwartet.

Interessant für die Evaluierung der Abbildbarkeit datengetriebener Prozessstrukturen sind insbesondere die in den *Workflow Patterns* beschriebenen Muster WCP12 bis WCP15. Sie erlauben die Erweiterung eines Prozesses durch mehrfache Instanziierung einer Aktivität. Hierbei wird anhand einer zur Modellier- bzw. erst zur Laufzeit bekannten Menge von Objekten[2] das entsprechende Prozessfragment mehrfach instanziiert, um es parallel auszuführen und die einzelnen Objekte zu verarbeiten. Während das Muster WCP12 auf die Synchronisierung der Prozessfragmente nach ihrer Ausführung verzichtet (vgl. Abbildung 3.3a), unterstützen dies die Muster WCP13 (vgl. Abbildung 3.3b) bis WCP15 explizit. Sie unterscheiden sich weiter hinsichtlich des Zeitpunkts, zu dem die Objektmenge bekannt ist (zur Modellierzeit, zur Laufzeit *vor* Erreichen der Aktivität oder *während* Ausführung der Aktivität).

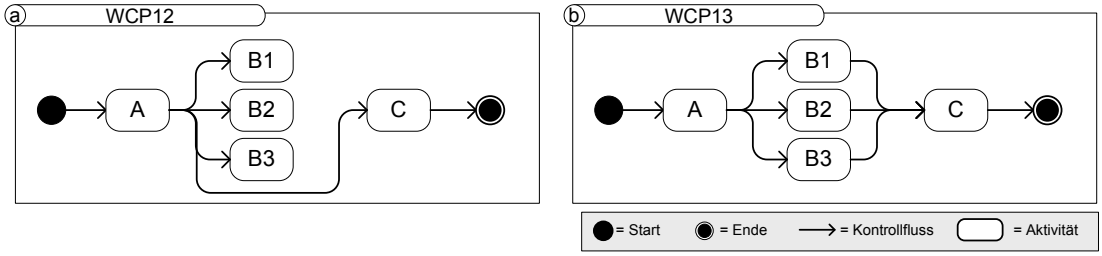


Abbildung 3.3: Muster für die multiple Instanziierung

Die *Workflow (Control-Flow) Patterns* bilden die Grundlage für die Entwicklung von *Yet Another Workflow Language* (YAWL), einer Petri-Netz basierten Sprache (siehe Abschnitt 4.8) mit nahezu vollständiger Unterstützung der Muster [AH05, RHAM06]. Eine prototypische Realisierung existiert [YAW].

Die beschriebenen Muster bzw. Konstrukte reichen, obwohl sie als „fortschrittlich“ zu bezeichnen sind und erst von wenigen Produkten unterstützt werden, bei weitem nicht aus, um datengetriebene Prozessstrukturen geeignet abzubilden. Zwar reicht die Ausdrucksmächtigkeit der Muster aus, um Prozessstrukturen zu beschreiben. Eine Modellierungsunterstützung, wie sie für die Integration von Daten- und Prozessstrukturen benötigt wird, ist jedoch nicht vorhanden. Die Muster zur Verarbeitung einfacher Mengen und die parallele Ausführung der zugeordneten Prozessfragmente eignen sich hierfür aus zwei Gründen nicht: Erstens können die Prozessfragmente untereinander nicht synchronisiert werden. Ist zum Beispiel in Abbildung 3.3a der mehrfach instanziierte Prozessschritt `B` als Sub-Prozess realisiert, können dessen Aktivitäten nicht synchronisiert werden (z.B. Aktivitäten der Instanzen `B1` und `B2`). Zweitens ist der dynamische Aufbau einer Prozessstruktur bzw. deren Erweiterung auf Basis der Beziehungen einer gegebenen Produktstruktur nicht möglich und auch nicht Ziel der *Workflow Patterns*.

Workflow Data Patterns

Eine Untersuchung von Mustern zur prozessualen Verarbeitung von Daten findet sich in den sogenannten *Workflow Data Patterns* [RHEA04, RHEA05]. Für die Koordination datengetriebener Prozessstrukturen sind insbesondere die Muster 33-36 der datenbasierten Steuerung (*Data*

[2]Wir bezeichnen Datenelemente bzw. Datenobjekte im Folgenden als *Objekte*. Wir betrachten keine Datenattribute.

Based Routing) relevant. Sie erlauben die Beschreibung der Beziehung zwischen Daten und Prozessen (bzw. Aktivitäten). Dies geschieht durch Definition von Vor- bzw. Nachbedingungen für die Ausführung von Aktivitäten basierend auf der Existenz oder konkreten Werten von Objekten. Zur Laufzeit wird dann individuell entschieden, ob ein Objekt die Bedingung erfüllt und die Aktivität ausgeführt bzw. abgeschlossen werden kann. Die Muster erlauben damit die Beschreibung eines dynamischen Pfads innerhalb des Prozesses. So kann die Angabe einer Vorbedingung genutzt werden, um Aktivtäten dynamisch zu starten. Wird beispielsweise die Existenz eines Dokuments als Vorbedingung erwartet (vgl. Muster 33 in [RHEA04, RHEA05]), kann die Aktivität erst gestartet werden, sobald eine (beliebige) Aktivität dieses Dokument erstellt. Wird hingegen eine quantifizierbare Qualitätsstufe als Nachbedingung für eine Entwicklungsaktivität angegeben (vgl. Muster 36 in [RHEA04, RHEA05]), kann die Aktivität zur Laufzeit erst dann abgeschlossen werden, wenn die gewünschte Qualitätsstufe tatsächlich erreicht wird.

Die beschriebenen Muster der *Data Patterns* bieten keinen vollständigen und durchgängigen Ansatz für eine Prozessbeschreibung, sondern sind als erweiterte Konstrukte der Prozessmodellierung zu verstehen. Sie betrachten keine Abhängigkeiten zwischen Objekten und bieten damit auch nicht die geforderte Modellierungsunterstützung datengetriebener Prozessstrukturen. Sie werden jedoch durch verschiedene weitergehende Systeme genutzt, die in den folgenden Abschnitten beschrieben werden (z.B. *BPEL*, *Staffware*[3], *FLOWer*, *AristaFlow*). In der Beschreibung der jeweiligen Ansätze diskutieren wir den Beitrag dieser Muster zur Modellierung datengetriebener Prozessstrukturen und welche Einschränkungen sich ergeben.

Change Patterns

Eine Evaluierung typischer Operationen zur Adaption von Prozessen und deren Unterstützung durch kommerzielle und wissenschaftliche Prozess-Management-Systeme ermöglichen *Change Patterns* [WRR07, WRRM08, RMRW08]. Sie unterscheiden Schema- und Instanzänderungen. Insbesondere die Muster mit Unterstützung der Instanzebene sind für die Adaption datengetriebener Prozessstrukturen von Belang. Die Change Patterns unterscheiden weiter zwischen allgemeinen Änderungsoperationen (sog. AP-Patterns) und Operationen für die *späte Modellierung* von Prozessfragmenten (sog. PP-Patterns). Unter einem Prozessfragment kann im Kontext datengetriebener Prozessstrukturen die Kapselung derjenigen Prozesse, die für dieselbe Komponente ausgeführt werden, verstanden werden. Zu den relevanten Mustern im Sinne der Anforderungen aus Abschnitt 2.2 gehören

- das Hinzufügen und Entfernen von Prozessfragmenten (AP1 und AP2),
- das Verschieben von Prozessfragmenten durch Veränderung der Synchronisationsbeziehungen (AP3),
- das Ersetzen von Prozessfragmenten (AP4),
- das Vertauschen von Prozessfragmenten (AP5),
- die Parallelisierung von Prozessfragmenten (AP9) sowie
- das Hinzufügen und Entfernen von Synchronisationskanten (AP11 und AP12).[4]

[3] Mittlerweile *Tibco iProcess Suite*.
[4] Muster AP3 und AP4 lassen sich durch Anwendung der Muster AP1 und AP2 abbilden; Muster AP5 durch AP3.

Interessant ist in diesem Zusammenhang auch das Muster PP3, welches eine „späte“ Komposition unsynchronisierter Prozessregionen erlaubt und eine Idee vermittelt, wie eine datengetriebene Prozessstruktur aufgebaut bzw. adaptiert werden könnte.

Dennoch liefern die *Change Patterns* keinen durchgängigen Ansatz, sondern sind als Anforderungen an die Unterstützung dynamischer Adaption in einem Prozess-Management-System zu verstehen. Sie stellen keinen direkten Bezug zwischen Daten und Prozessen her, liefern jedoch interessante Anhaltspunkte für zu realisierende Operationen im Rahmen der dynamischen Adaption datengetriebener Prozessstrukturen (siehe Abschnitt 6.2.2). Unter den untersuchten wissenschaftlichen und kommerziellen Anwendungen (u.a. *FLOWer*, *Staffware*) bietet einzig das *ADEPT/CBRFlow* System eine umfassende, wenn auch noch nicht vollständige Unterstützung der erwähnten Muster auf Instanzebene. Das Muster PP3 wird von keinem der in [WRR07, WRRM08] untersuchten Ansätze unterstützt. Wir diskutieren die Muster detailliert in Abschnitt 6.4.

Exception Handling Patterns

Die Muster in [RAH06] beschreiben *Exception Handling Patterns* als allgemeine, wiederkehrende Ausnahmesituationen in Prozess-Management-Systemen mit Blick auf die einzelnen Aufgaben einer *Arbeitsliste*. Zwar liegen Arbeitslisten nicht im Fokus dieser Arbeit; drei der beschriebenen Ausnahmen können jedoch – bei großzügiger Interpretation – auch in datengetriebenen Prozessstrukturen auftreten. Dies sind der *Work Item Failure*, also die fehlgeschlagene Ausführung eines Arbeitsschritts bzw. Prozesses, der *External Trigger*, also ein externes Signal, das zum Beispiel von einem anderen unsynchronisierten Prozess kommen kann, und zuletzt die *Constraint Violation*, die in Form von Inkonsistenzen bei der dynamischen Adaption auftreten kann. Wir diskutieren die Bedeutung der Muster im Kontext datengetriebener Prozessstrukturen in Abschnitt 7.6. Es ist jedoch erwähnenswert, dass viele Systeme für die beiden erwähnten Muster keine (z.B. *Staffware*) oder nur eine geringe Unterstützung (z.B. *FLOWer*) bieten.

3.1.2 Business Process Management Standards

Neben den *Workflow Patterns*, die hauptsächlich der Evaluierung von Prozess-Management-Systemen dienen, existieren verschiedene Standards für die Prozessmodellierung. Die Standards bieten den Vorteil, dass – zumindest in der Theorie – ein konform des Standards erzeugtes Prozessmodell in Modellierungs- oder Ausführungsumgebungen verschiedener Hersteller genutzt werden kann (sofern diese den Standard unterstützen). Die wichtigsten Vertreter für Modellierungsstandards sind die *Unified Modeling Language (UML)* und die *Business Process Modeling Notation (BPMN)*.

Neben Modellierungssprachen existieren auch Ausführungssprachen. Sie erlauben die konkrete ausführbare Spezifikation eines Prozesses. Hier untersuchen wir die *Business Process Execution Language (BPEL)* als den wichtigsten Vertreter. BPEL wurde ebenfalls in [RHEA04, RHEA05] auf Unterstützung der Workflow Patterns untersucht.

Wir klären zunächst die Frage, ob sich aktuelle Modellierungsnotationen und Ausführungssprachen für die Abbildung datengetriebener Prozessstrukturen in ausreichender Form eignen. Dem

schließt sich die Frage an, ob datengetriebene Prozessstrukturen automatisiert erzeugbar sind. Auf dieser Basis lassen sich auch Schlüsse im Hinblick auf Prozess-Management-Systeme ziehen, die die untersuchten Standards unterstützen.

Modellierungsnotationen

In der Prozessmodellierung sind die *Unified Modeling Language* (UML) und die *Business Process Modeling Notation* (BPMN) verbreitet. Sie bieten eine reichhaltige Auswahl an Modellierungskonstrukten und damit eine ausreichende Ausdrucksmächtigkeit für die Darstellung vieler Prozesse, also auch nicht-deterministischer und zyklischer Abläufe (vgl. Abschnitt 1.1). Die Ausdrucksmächtigkeit der verfügbaren Kontrollflusskonstrukte genügt, um Prozessstrukturen „als Ganzes“ (d.h. zunächst ohne Bezug zu Datenstrukturen) zu modellieren [Neu09]. Darüberhinaus verfügen diese Standards über die Möglichkeit, Datenflüsse zu beschreiben sowie Objekte und deren Zustände zu unterscheiden.

Die Modellierungsnotationen UML und BPMN erlauben außerdem die dynamische Erzeugung von Prozessfragmenten durch die *multiple Instanziierung* in Form der *Expansion Region* in UML 2.0 Aktivitätsdiagrammen [Obj03b] (vgl. Abbildung 3.4) und *Multiple Instance* in BPMN 1.0 [Bus06]. Zu diesem Zweck wird zur Modellierzeit ein Prozessfragment mit einer Menge oder Liste als Eingabe verknüpft. Zur Laufzeit sollen die Aktivitäten innerhalb des Prozessfragments für jedes Element ausgeführt werden. Die Ausführungssemantik kann in UML und BPMN angegeben werden (z.B. *parallel*, *iterative* und *stream*) und geht damit über die in den Workflow Patterns beschriebenen Muster zur multiplen Instanziierung hinaus. Die instanziierten Prozessfragmente lassen sich allerdings nicht untereinander synchronisieren. Ferner wird die Erzeugung von Prozessstrukturen aus komplexen Produktstrukturen nicht unterstützt, da sich im Kontext der multiplen Instanziierung zwischen Objekten einer Menge oder Liste keine Beziehungen definieren lassen.[5] UML bietet gegenüber BPMN zwar einen erweiterten Funktionsumfang, indem komplexe Datenstrukturen in UML durch Klassendiagramme beschrieben werden können. So lassen sich Klassen bzw. Objekte sowie Beziehungen zwischen ihnen anlegen. Ferner können Klassen bzw. Objekte um eine semantische Beschreibung mittels Aktivitäten- bzw. Zustandsdiagrammen erweitert werden. Deren automatische Verknüpfung auf Basis komplexer Klassen- bzw. Objektstrukturen ist jedoch nicht definiert. Auch das in diesem Kontext evaluierte Werkzeug *IBM Rational Systems Developer*[6], das den UML 2.0 Standard mit erweitertem Funktionsumfang unterstützt, bietet keine Möglichkeit für die automatische Erzeugung einer Prozessstruktur auf Basis eines modellierten Klassen- oder Objektdiagramms [Neu09].

Des Weiteren können die Standards die Anforderungen an eine Ausführungsunterstützung und dynamische Adaption (z.B. Änderung der übergebenen Menge bzw. Liste während der Ausführung) nicht erfüllen. Den Modellierungsstandards fehlt eine definierte formale operationale Semantik, wodurch keine formalen Korrektheitskriterien für die erstellten Modelle spezifiziert sind. Zwar bietet das Werkzeug *IBM Rational Systems Developer* eine einfache strukturelle Prüfung von Aktivitäten- bzw. Zustandsdiagrammen, die Kriterien für Korrektheit sind jedoch nicht ausreichend [Neu09].

[5] Eine detaillierte Diskussion der Modellierungsaspekte von UML und BPMN findet sich in Abschnitt 5.5.
[6] Untersuchte Version: IBM Rational Systems Developer V. 7.05.

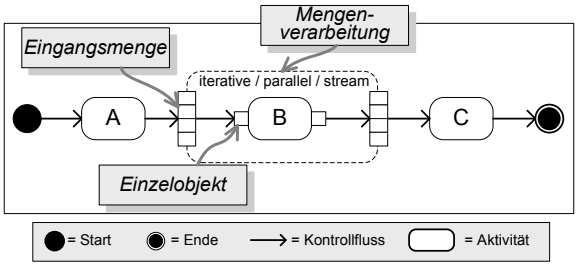


Abbildung 3.4: Abbildung der multiplen Instanziierung in UML-Aktivitätsdiagrammen

Zusammengefasst können die Anforderungen der Modellierung nur zu einem geringen Anteil, die Anforderungen der Ausführung und dynamischen Adaption gar nicht erfüllt werden. Die Ausnahmebehandlung kann, sofern planbar, mittels sogenannter *Exception Handler* in UML ([Obj03b]) bzw. *Error Handler* in BPMN ([Bus06]) modelliert werden. Jedoch sind die Möglichkeiten der Reaktion begrenzt und deren operationale Semantik nicht exakt definiert. Sie sind damit für die *dynamische* Ausnahmebehandlung in datengetriebenen Prozessstrukturen bei weitem nicht ausreichend.

Ausführungssprachen

Ausführungssprachen erlauben die Beschreibung ausführbarer Prozesse. Prozess-Management-Systeme, die diese Standards unterstützen, interpretieren die Prozessbeschreibungen und führen sie mittels einer (individuell) definierten operationalen Semantik aus. Die *Business Process Execution Language* (BPEL) ist der wichtigste Vertreter der Ausführungssprachen. BPEL ist eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, deren einzelne Aktivitäten durch *Web-Services* implementiert sind [Org07]. BPEL beschreibt die Ausführungslogik der *Web-Services* durch die Spezifikation des Kontrollflusses (*Choreographie*). Eine BPEL-Beschreibung wird von einer Ausführungskomponente (z.B. *IBM WebSphere Process Server* [PG05]) interpretiert und selbst wieder als Web-Service zur Ausführung gebracht. Der Ansatz erlaubt damit die hierarchische Schachtelung von Prozessen.

Grundsätzlich bietet der BPEL-Standard keine Unterstützung datengetriebener Prozessstrukturen. Die operationale Semantik ist informell vom Standard vorgegeben, wird aber von den verschiedenen Herstellern von BPEL-Ausführungskomponenten unterschiedlich interpretiert. Während laut [RHAM06] beispielsweise der durch OASIS spezifizierte BPEL-Standard und das Prozess-Management-System IBM WebSphere die multiple Instanziierung nicht unterstützen, realisiert die *Oracle BPEL-Komponente* die entsprechenden Workflow Patterns (WCP-13 und WCP-14).[7] Die Prüfung der Korrektheit eines Prozesses ist vom Standard genauso wenig vorgesehen, wie die dynamische Adaption der Prozessstrukturen. Sie ist damit vollständig abhängig vom ausführenden Prozess-Management-System.

[7] Die multiple Instanziierung wird von IBM WebSphere erst seit Version 6.1 (veröffentlicht 2006) unterstützt.

3.1.3 Klassisches Prozess-Management

Es existieren zahlreiche klassische Prozess-Management-Systeme (bzw. Workflow-Management-Systeme), die eine Unterstützung der in Abschnitt 3.1.1 eingeführten Muster bieten und damit die Realisierung von prozessorientierten Informationssystemen erlauben. Als bekannte Vertreter seien *Tibco iProcess Suite* (ehemals *Staffware*), *IBM WebSphere MQ Series* und *IBM WebSphere Series* genannt. Klassische Prozess-Management-Systeme zeichnen sich durch eine aktivitätenorientierte Prozessmodellierung mit Definition von Kontrollflüssen und deren Anreicherung um Datenaspekte (z.B. Ein- und Ausgabedaten) aus [GHS95, LR00, RHEA05, Bro05].

In der Praxis werden jedoch nicht nur „reine“ Prozess-Management-Systeme genutzt. Im Maschinenbau bezeichnet *Product-Life-Cycle-Management (PLM)* die Strategie, alle innerhalb des Lebenszyklus eines Produkts oder dessen Komponenten, also bei deren Entwicklung, Produktion und Vertrieb, anfallenden Daten durchgehend zu verwalten [ADEK05]. PLM kombiniert damit unter anderem die Verwaltung der Daten, also *Produktdaten-Management (PDM)* und deren Verarbeitung, also *Prozess-Management*. So verfügen heutige PDM-Systeme neben Komponenten zur Datenverwaltung auch über Prozess-Management-Funktionalität [BP07]. Aus der Reihe der PDM-Systeme wurde mit *UGS Teamcenter Engineering* (TcEng)[8] ein relevanter Stellvertreter intensiv untersucht [MHHR06]. *TcEng* wird für das PLM von Entwicklungsdokumenten insbesondere in der Automobilentwicklung von verschiedenen Herstellern genutzt [HRKH04].

Prozess-Management-Systeme

Tibco iProcess Suite, *IBM MQ Series* und *IBM WebSphere Series* verfügen hinsichtlich ihrer Ausdrucksmächtigkeit über einen vergleichbaren Leistungsumfang [RHEA04, RHEA05, RHAM06]. Ihre umfangreiche Untersuchung im Rahmen der Workflow Patterns Initiative zeigt, dass sie die untersuchten Muster nur teilweise unterstützen (vgl. Abschnitt 3.1.1). Die untersuchten Muster der Workflow Patterns erfüllen selbst jedoch bei weitem nicht die Anforderungen aus Abschnitt 2.2. Ferner hat die tiefergehende Diskussion in Abschnitt 3.1.2 gezeigt, dass verbreitete Standards die Abbildung und Adaption datengetriebener Prozessstrukturen ebenfalls nicht direkt unterstützen. Prozess-Management-Systeme bieten jedoch im Allgemeinen einen über die Modellierungsstandards hinausgehenden Leistungsumfang. Die weitergehende Unterstützung der funktionalen Anforderungen diskutieren wir im Folgenden am Beispiel der *IBM WebSphere Series*.

Die *WebSphere Series*[9] ist ein transaktionales Prozess-Management-System, das von IBM für die Realisierung *serviceorientierter Architekturen (SOA)* angeboten wird [PG05]. WebSphere unterstützt BPEL und damit die Choreographie von Web Services. Weiter unterstützt die *WebSphere Series* die Modellierung von *Business State Machines*, also die Abbildung von Objektlebenszyklen als Zustandsautomaten. *Business State Machines* werden intern in BPEL transformiert und dann zur Ausführung gebracht. In diesem Kontext bietet der Ansatz allerdings keine ausreichende Unterstützung für die Synchronisation der *Business State Machines* und damit für den Aufbau großer Prozessstrukturen aus verschiedenen Business State Machines.

[8] Untersuchte Version: UGS Teamcenter Engineering V. 9.12.
[9] Untersuchte Version: WebSphere Process Server V. 6.1.

Ferner fehlen eine geeignete Abbildung von Datenstrukturen und damit auch eine Modellierungsunterstützung für datengetriebene Prozessstrukturen. Die dynamische Adaption wird derzeit nicht ausreichend unterstützt [Neu09]. Zwar existieren Ansätze, die eine dynamische Adaption mittels Workarounds erlauben [WEH08], dies jedoch nur in einem eingeschränkten Umfang. Das Abweichen vom modellierten Ablauf eines BPEL-Prozesses mittels Sprüngen ist möglich.

Prozess-Management in Produktdaten-Management-Systemen

Teamcenter Engineering (TcEng) verfügt über ein flexibles Datenmodell mit integriertem Versions- und Konfigurationsmanagement. Objekten können Zustände (z.B. *freigegeben*) zugeordnet werden. Des Weiteren ist eine Prozess-Management-Komponente in *TcEng* integriert. Sie verfolgt einen aktivitätenorientierten Ansatz. Es können Prozessmodelle angelegt und für die Bearbeitung konkreter Objekte (*Targets*) bzw. Objektversionen der Produktstruktur instanziiert werden. Prozesse lassen sich untereinander synchronisieren. Die Ausdrucksmächtigkeit ist ausreichend, um Prozessstrukturen „als Ganzes“ zu modellieren. Es lässt sich jedoch kein Bezug der Prozesse zu den Beziehungen zwischen Objekten herstellen, um daraus die automatisierte Erzeugung der Prozessstruktur zu realisieren.

Die Korrektheit von Prozessen wird von der Prozess-Management-Komponente zur Modellierzeit nicht geprüft. Es lassen sich somit Verklemmungen und Endlosschleifen modellieren. Zur Laufzeit werden zwar eingetretene Verklemmungen erkannt (vgl. Abbildung 3.5), dem Benutzer stehen aber nur begrenzte Reaktionsmöglichkeiten zur Verfügung (Abbruch der Ausführung mittels Aktion *Anhalten*). Die dynamische Adaption der Prozessstruktur, eine weitere wichtige Anforderung datengetriebener Prozessstrukturen, ist nicht vorgesehen.

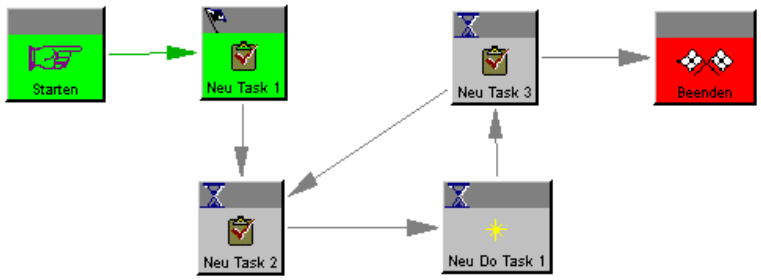


Abbildung 3.5: Prozess in Teamcenter Engineering mit *aktiver* Verklemmung

3.1.4 Flexible Prozess-Management-Systeme

Die Untersuchung klassischer Prozess-Management-Systeme hat verschiedene Einschränkungen bei der Abbildung datengetriebener Prozessstrukturen aufgezeigt. Insbesondere die Möglichkeit zur Integration von Daten und Prozessen, sowie die dynamische Adaption und die zugehörige Ausnahmebehandlung werden nicht ausreichend unterstützt. Es sind jedoch auch Prozess-Management-Systeme verfügbar, die insbesondere mit der Unterstützung von dynamischer Adaption und Ausnahmebehandlung werben. Sie haben ihren Ursprung und ihre technischen Grundlagen in wissenschaftlichen Arbeiten und bieten hinsichtlich der erwähnten Anforderungen einen erweiterten Funktionsumfang.

Wir stellen im Folgenden stellvertretend drei flexible Prozess-Management-Systeme vor, deren zugrundeliegende Paradigmen sich unterscheiden. Mit der *AristaFlow BPM Suite* steht ein aktivitätenorientiertes Prozess-Management-System zur Verfügung, das auf dem ADEPT-Ansatz beruht und die dynamische Adaption von Prozessen erlaubt. Das Werkzeug *Pallas Athena FLOWer* realisiert die Case-Handling Konzepte, welche die datenorientierte Modellierung und Ausführung eines Prozesses unterstützt. Zuletzt stellen wir mit *OptinSolutions DBPM* eine Anwendung vor, die die zielorientierte Modellierung eines Prozesses realisiert. Die vorgestellten Ansätze werden außerdem in den Kapiteln 4 bis 7 detailliert diskutiert.

AristaFlow BPM Suite

Die *AristaFlow BPM Suite* basiert auf den ADEPT-Konzepten (u.a. [RD98, Rei00, RPB03, RRD04b, RRKD05, DRRM$^+$09, DR09]) und erlaubt die aktivitätenorientierte Modellierung von Prozessen. ADEPT unterstützt hierbei die Integration von Datenflüssen und deren durchgängige Korrektheitsprüfung. Eine herausragende Eigenschaft des Ansatzes bzw. des Produkts sind Mechanismen zur ad-hoc Adaption von Prozessinstanzen. Ferner erlauben die Konzepte die Adaption von Prozessmodellen und deren Migration auf ihre laufenden Instanzen. Es stehen vielseitige Operationen zum Hinzufügen, Entfernen und Verschieben von Prozessschritten und Datenelementen zur Verfügung [WRR07, WRRM08]. Die Adaption geschieht unter vollständiger Zusicherung der Korrektheit nicht nur des Kontrollflusses, sondern auch des Datenflusses. Der Ansatz erlaubt zudem die Behandlung von Ausnahmen durch geeignete Sprungoperationen.

ADEPT verfügt zwar mit der Modellierung und Korrektheitsanalyse über umfassende Funktionen zur Abbildung von Datenflüssen, die Datenorientierung mit der Verarbeitung von Produktstrukturen wird mit diesem Ansatz jedoch nicht adressiert. Die Modellierung von großen Prozessstrukturen ist zwar manuell möglich, nicht jedoch automatisiert auf Basis einer gegebenen Produktstruktur. Die Ausführung und dynamische Adaption der modellierten Prozesse bzw. Prozessstrukturen wird durch die Anwendung zwar unterstützt, für die dynamische Adaption von datengetriebenen Prozessstrukturen und entsprechende Ausnahmebehandlung sind diese Konzepte jedoch nur bedingt geeignet (vgl. Kapitel 6 und 7).

Pallas Athena FLOWer

FLOWer ist die Prozess-Management-Komponente der *BPM Suite* von *Pallas Athena* zur Modellierung und Ausführung datenorientierter Prozesse. Datenorientierte Prozesse beschreiben die Abfolge der Prozessschritt durch Modellierung des Datenflusses (vgl. Abschnitt 1.1). *FLOWer* basiert diesbezüglich auf dem Case-Handling Paradigma. Case-Handling (Fallbehandlung) bzw. *Product Driven Case-Handling* beschreibt den Ablauf eines Prozesses nicht durch einen definierten Kontrollfluss (und damit einer festen Reihenfolge der Arbeitsschritte) [AB01b, SW02, ASW03, AWG05]. Stattdessen steht das Ergebnis (d.h. das Produkt) des Prozesses im Mittelpunkt, welches sich aus verschiedenen Objekten (bzw. Daten) zusammensetzt. Objekte sind entweder vorhanden oder nicht vorhanden. Relevant für die Steuerung der Abläufe sind die Abhängigkeiten zwischen Arbeitsschritten und relevanten Daten (vgl. Data Flow Muster 33-36 aus Abschnitt 3.1.1 bzw. [RHEA05]). Durch die Definition der Datenflüsse über Vor- und Nachbedingungen für die Ausführung der einzelnen Aktivitäten, ergibt sich der endgültige Ablauf erst

zur Laufzeit, abhängig von den zur Verfügung stehenden Objekten. Sie repräsentieren somit auch den aktuellen Stand des Prozesses zur Laufzeit. Die Abbildung des Case-Handling-Konzepts erfolgt in *FLOWer* mittels Formularen, über die notwendige Daten manuell eingegeben werden [VRA07, MWR08, Van09].

Der Vorteil dieser Modellierungsmethodik liegt in der flexiblen und effizienten Ausführung. Werden die Objekte zu Beginn der Aktivität erzeugt, kann parallel mit der nächsten Aktivität begonnen werden – auch wenn die erzeugende Aktivität noch gar nicht beendet ist. Das gleiche Verhalten wäre per Kontrollflussmodellierung nur schwer zu erreichen und setzt die Fragmentierung der Aktivitäten (gemäß der Erzeugung von Dokumenten voraus). Ein weiterer Vorteil der datenorientierten Modellierung ist die Weiterverwendung bereits existierender Objekte. Ist beispielsweise ein benötigtes Dokument bereits vorhanden, braucht die erzeugende Aktivität gar nicht instanziiert bzw. ausgeführt zu werden. Dieser Fall kann ebenfalls nur schwer per Kontrollfluss nachgebildet werden. Es müssten bedingte Verzweigungen eingesetzt werden, um die Existenz jedes Eingabeparameters zu prüfen.

Das Case-Handling-Konzept bietet zwar die Möglichkeit, datenorientierte Prozesse flexibel zu beschreiben; die Ausdrucksmächtigkeit ist jedoch eingeschränkt. So kann eine einfache Kontrollflussmodellierung mit vormodellierten Schleifen nur über Workarounds erreicht werden [RHAM06]. Beziehungen zwischen Objekten werden nicht berücksichtigt. Damit liefert Case-Handling keine Unterstützung für die Modellierung datengetriebener Prozessstrukturen. Die Anforderungen an dynamische Adaption mit Ausnahmebehandlung können nur zu geringen Teilen durch Workarounds (Überspringen von Aktivitäten; *Skip*-Rolle) erfüllt werden [AWG05].

OptinSolutions Dynamic Business Process Modeling (DBPM)

Die Anwendung *OptinSolutions Dynamic Business Process Modeling* (DBPM) realisiert einen zielorientierten Ansatz [MWRR06, Mül06]. Hierbei wird die Abfolge der Prozessschritte, im Gegensatz zu aktivitäten- und datenorientierten Ansätzen, nicht manuell modelliert, sondern automatisch erzeugt. Grundlage ist die Verwaltung von Aktivitäten in einem sog. *Repository*. Der Ansatz nimmt an, dass Aktivitäten immer Daten (bzw. Objekte) erzeugen oder deren Zustände manipulieren. Sie werden daher explizit mit ihren Ein- und Ausgabedaten (bzw. deren Zustände) definiert. Die Abhängigkeiten zwischen zwei Aktivitäten A und B lassen sich durch Analyse der gelieferten Ausgabedaten von A und den benötigten Eingabedaten von B (und umgekehrt) ableiten. Die Idee des Ansatzes ist, basierend auf einem gegebenen Objekt (bzw. seines Zustands) und eines zu erreichenden Ziels (Existenz eines Objekts bzw. seines Zustands) den dafür benötigten Prozess automatisch zu erzeugen. Hierfür werden auf Basis der Aktivitäten im Repository mögliche Kontrollflüsse generiert, mit denen sich beginnend mit dem gegebenen Objekt das Ziel erreichen lässt. Sollten mehrere mögliche Kontrollflüsse erzeugbar sein, kann auf Basis bestimmter Kriterien (z.B. Prozessdauer oder -kosten) ein optimaler Kontrollfluss ausgewählt werden.

Die Evaluierung der Anwendung hat gezeigt, dass die Modellierung von Produktstrukturen möglich ist und im Prinzip eine Modellierungsunterstützung für datengetriebene Prozessstrukturen existiert. Die Ausdrucksmächtigkeit ist allerdings eingeschränkt (z.B. keine Modellierung von Schleifen). Ferner ist die Korrektheit der erstellten Prozesse nicht ausreichend gewährleistet [Mül06]. Zwar erlaubt *OptinSolutions DBPM* die Simulation des erstellten Prozesses, es sind jedoch weder die operationale Semantik noch Korrektheitskriterien formal dokumentiert. Des

Weiteren ist keine Ausführungskomponente für die erstellten Prozesse verfügbar. Die dynamische Adaption des Prozessgeflechts bei Änderung der Produktstruktur wird genauso wenig unterstützt, wie die Behandlung von Ausnahmen.

3.2 Wissenschaftliche Ansätze

Die Untersuchung kommerzieller Prozess-Management-Systeme und Standards in Abschnitt 3.1 hat gezeigt, dass Ansätze aus der Praxis nicht ausreichen, um die in Abschnitt 2.2 dargestellten Anforderungen vollständig und durchgängig zu erfüllen. In diesem Abschnitt prüfen wir daher, inwiefern sich wissenschaftliche Ansätze für die Unterstützung datengetriebener Prozessstrukturen eignen. Wir unterscheiden hierbei aktivitätenorientierte, datenorientierte und integrierte Ansätze. Die integrierten Ansätze zeichnen sich dadurch aus, dass Prozesse und Daten für die Prozesskoordination gemeinsam verwendet werden.

Entsprechend Abbildung 3.2 zeigt Abbildung 3.6 eine Einteilung der untersuchten Ansätze aus der Wissenschaft. Auch sie soll lediglich der groben Einordnung der Ansätze dienen.

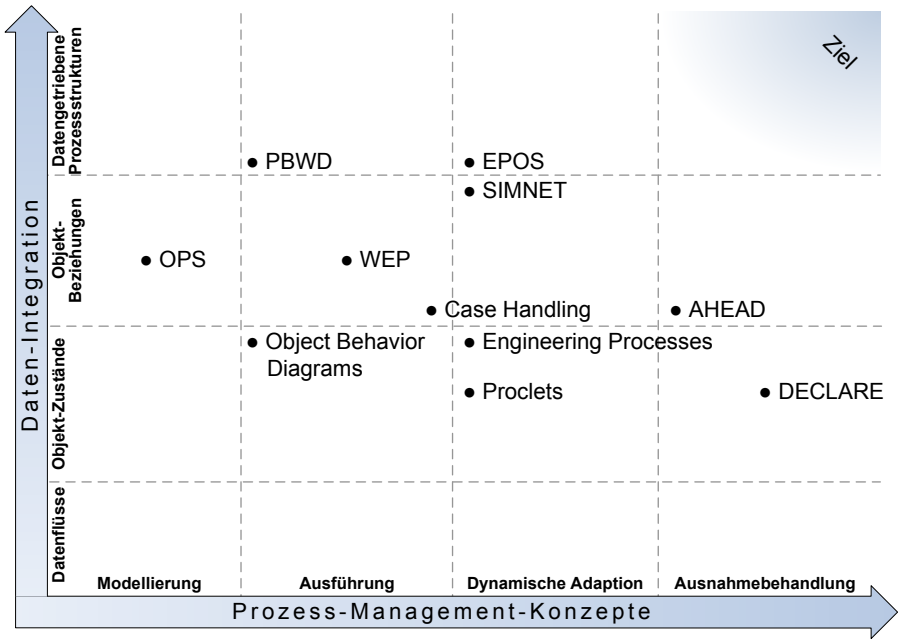


Abbildung 3.6: Einordnung der Ansätze aus der Wissenschaft

3.2.1 Aktivitätenorientierte Ansätze

In Abschnitt 3.1 wurden bereits Konzepte für die aktivitätenorientierte Modellierung und Ausführung von Prozessen vorgestellt (z.B. *WebSphere* und *AristaFlow*). In der wissenschaftlichen Literatur finden sich mit *WEP* und *DECLARE* zwei Ansätze, die Prozesse durch die Abfolge ihrer Prozessschritte beschreiben und zusätzlich einen Bezug zu Datenstrukturen herstellen. Weitere aktivitätenorientierte Ansätze werden in Teil II dieser Arbeit diskutiert.

Workflow Management for Engineering Processes (WEP)

Das Konzept *Workflow Management for Engineering Processes* (WEP) behandelt die Modellierung und Ausführung von Entwicklungsprozessen [BDS98, Beu03]. Basis für die Ausführung ist eine modellierte Prozessstruktur mit grundlegenden Abläufen, wobei einzelne Blöcke als unstrukturierte, zielorientierte Teilprozesse definiert werden können. Sie erlauben die lose Beschreibung kreativer Entwicklungsschritte (d.h. ihre von verschiedenen Rahmenbedingungen abhängige, nicht planbare Abfolge). Diese Blöcke erlauben ein sog. *Late Modeling* (vgl. Muster PP2 in [WRRM08]), bei dem die Reihenfolge der verschiedenen Aktivitäten nicht fest vorgegeben ist, sondern vom Nutzer zur Laufzeit definiert wird. Zur Modellierzeit werden die Eingangsdaten des Blocks, also Objekte mit einer gewissen Datenqualität, definiert. Des Weiteren ist das zu erreichende Ziel, also das zu erstellende Objekt, dessen gewünschte Qualität und der Zeitpunkt des Abschlusses des Blocks (Meilenstein) festgelegt. Die lose gekoppelten Aktivitäten können zur Laufzeit zielorientiert ausgeführt werden. Die Idee der zielorientierten Blöcke in *WEP* wurde im Werkzeug DBPM aus Abschnitt 3.1 aufgegriffen.

Ferner unterstützt der Ansatz die Ziele des *Concurrent Engineering* durch eine vorzeitige Datenweitergabe (mit Berücksichtigung der Datenqualität). Die Dynamik der Entwicklungsprozesse wird durch die dynamische Anpassung der Prozessstruktur bei Änderung der zu erzeugenden Daten berücksichtigt und durch Prozessfragmente mit multipler Instanziierung ermöglicht.

Der *WEP*-Ansatz erlaubt die Beschreibung von Prozessstrukturen; eine geeignete Modellierungsunterstützung für den Aufbau großer Prozessstrukturen ist jedoch auch hier nicht gegeben. Dennoch liefert der Ansatz mit dem integrierten Datenmodell und der möglichen Ausnahmebehandlung Ideen, die wir in den Abschnitten 5.5 und 7.6 weiter diskutieren.

DECLARE

Entgegen der konventionellen Modellierung eines expliziten Prozessablaufs stellt *DECLARE* einen deklarativen Ansatz für die Beschreibung von Kontrollflüssen vor [PSA07, PSSA07]. Idee ist es, die Ablaufreihenfolge der Prozessschritte auf Basis von Regeln und Einschränkungen implizit zu definieren (*Constraints*). Damit werden nicht nur *erlaubte* Kontrollflüsse (bzw. Prozessfragmente) beschrieben, sondern auch *unerlaubte* Kontrollflüsse vorgegeben. So lassen sich beispielsweise die Sachverhalte *Prozessschritte A und B dürfen nicht gleichzeitig ausgeführt werden* und *wenn Prozessschritt A ausgeführt wird, muss Prozessschritt B nach A ausgeführt werden* ausdrücken. Im Gegensatz zu ähnlichen Ansätzen, wie der Beschreibung von Regeln für die Ausführung von Aktivitäten auf Basis des Datenflusses in Case-Handling, ermöglicht *DECLARE* dies auch auf Basis des Kontrollflusses. Zusätzlich kann ein in *DECLARE* beschriebener Prozess mit einem aktivitätenorientierten, in *YAWL* (vgl. Abschnitt 3.1.1) beschriebenen Prozess kombiniert werden [PSA07, PSSA07].

Die deklarative Beschreibung großer Prozessstrukturen führt zwangsläufig auch zu einer großen und schwer überschaubaren Anzahl von (unstrukturierten) Regeln. Die mehrfache Instanziierung (bzw. Wiederverwendung) von Prozessfragmenten in einer Prozessstruktur ist durch den Ansatz nicht möglich, da die Regeln hier keine Unterscheidung der Fragmente zulassen und damit die unterschiedliche Synchronisation verschiedener Prozessfragmente stark eingeschränkt

sind. Die Modellierung datengetriebener Prozessstrukturen wird nicht unterstützt. Eine geeignete Korrektheitsanalyse und Ausführungsunterstützung sind nach [PSA07, PSSA07] hingegeben genauso möglich, wie die dynamische Adaption.

3.2.2 Datenorientierte Ansätze

Während in aktivitätenorientierten Ansätzen der Kontrollfluss (bzw. geeignete Regeln) den Ablauf der einzelnen Prozessschritte definiert, kontrollieren in datenorientierten Ansätzen die Beziehungen zwischen Daten die Abfolge der Prozessschritte (z.B. durch den Datenfluss; vgl. Abschnitt 1.1). Mit Case-Handling wurde in Abschnitt 3.1.4 bereits ein datenorientierter Ansatz vorgestellt. Die in diesem Abschnitt diskutierten Ansätze verstehen sich teilweise auch als Erweiterung aktivitätenorientierter Ansätze. Sie sind häufig aus Konzepten zur Verarbeitung von Produktstrukturen hervorgegangen (z.B. *PBWD*, *OPS*) oder aus der objektorientierten Software-Entwicklung abgeleitet (z.B. *Object Behavior Diagrams*).

Proclets

Die Partitionierung eines Prozesses in seine Teilaspekte auf Basis der im Prozess verwendeten Ressourcen, also beispielsweise der verwendeten Dokumente, realisiert der *Proclet*-Ansatz [ABEW00]. Ein *Proclet* repräsentiert den Lebenszyklus eines bestimmten Objekts. Proclets sind aktivitätenorientiert modelliert (sog. *Workflow Netze*) und enthalten die Aktivitäten (*Tasks*) zum Durchlaufen des Lebenszyklus (vgl. Abbildung 3.7). Durch Definition von Kommunikationskanälen (*Communication Channels*) können Nachrichten zwischen den Proclets ausgetauscht und damit eine Prozessstruktur aufgebaut werden. Genauso wie Objekte, können auch Kommunikationskanäle dynamisch hinzugefügt oder entfernt werden.

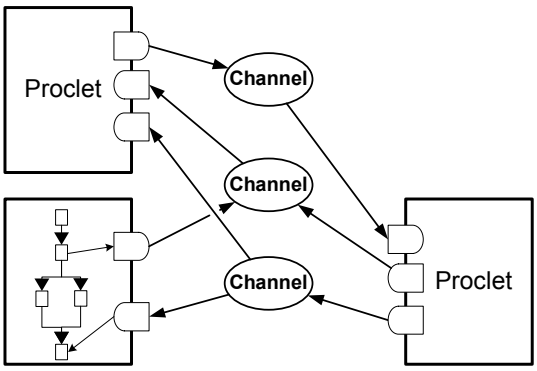


Abbildung 3.7: Schematischer Aufbau einer Prozessstruktur auf Basis von *Proclets* [ABEW00]

Es existiert jedoch kein Mechanismus für die Modellierungsunterstützung datengetriebener Prozessstrukturen. Korrektheitsprüfungen im Rahmen der dynamischen Adaption sind genauso wenig diskutiert, wie Mechanismen zur Ausnahmebehandlung.

Zielorientierte Prozessmodellierung mittels Planungstechniken

In technischen Bereichen, wie beispielsweise der Produktionsplanung und -steuerung, werden Ablaufpläne für die Prozesssteuerung eingesetzt. Derartige Pläne beschreiben den Ablauf von Prozessschritten, durch deren Ausführung ein gegebenes Ziel erreicht werden soll [Rei00]. Viele Methoden für Produktionsplanung nutzen *Stücklisten* (*Bill of Material*) in Kombination mit einem Basisplan für die Erzeugung der Ausführungsreihenfolge der notwendigen Prozessschritte. Planungstechniken erlauben ferner die Optimierung eines Ablaufplans auf Basis bestimmter Parameter. Ein Beispiel für die Nutzung der Konzepte der Planungstechnik, haben wir mit der Anwendung *DBPM* bereits kennen gelernt.

Mit der automatischen Erzeugung einer Prozessstruktur auf Basis der Planungstechnik sind jedoch konzeptionelle Einschränkungen verbunden. Die Ausdrucksmächtigkeit ist meist eingeschränkt, da sich nicht-deterministische Prozessschritte und Zyklen (vgl. Abschnitt 1.1) genauso wenig modellieren lassen, wie die Synchronisation nebenläufiger Prozessschritte. Des Weiteren erlaubt die Planungstechnik im Allgemeinen keine Adaption der Prozessstruktur, sondern erzeugt die Prozessstruktur bei einer Änderung der Parameter von Neuem. Geschieht dies zur Laufzeit, muss sichergestellt werden, dass zum einen die Prozessstruktur nicht erneut gestartet, sondern der aktuelle Laufzeitzustand beibehalten wird. Zum anderen muss die Korrektheit der Prozessstruktur gewährleistet werden. Die hierzu untersuchten Ansätze unterstützen dies nicht (vgl. Ansätze DBPM in Abschnitt 3.1.4 und EPOS in Abschnitt 3.2.3).[10]

Product Based Workflow Design (PBWD)

Product Based Workflow Design (PBWD) adressiert die Modellierung von Produktionsprozessen, die für die Herstellung eines physischen oder abstrakten Produkts notwendig sind [RLA03]. Das Produkt selbst wird durch eine Stückliste, bestehend aus Komponenten und Abhängigkeiten zwischen Komponenten, repräsentiert. PBWD basiert auf dem in [Aal97a, Aal99] vorgestellten Ansatz, mit dessen Hilfe sich Produktionsprozesse für ein Produkt erzeugen lassen. Basis für die Erzeugung der Prozessstruktur ist die Stückliste des Produkts. Das Konzept sieht vor, jeder Komponente den Prozessschritt zuzuordnen, der zur Erstellung der jeweiligen Komponente erforderlich ist. Auf Basis der Abhängigkeiten der Komponenten kann nun automatisch ein geeigneter Kontrollfluss erzeugt werden. Damit bietet der Ansatz aus [Aal97a, Aal99] prinzipiell eine Modellierungsunterstützung datengetriebener Prozessstrukturen. Der in Abschnitt 2.1 beschriebene Anwendungsfall lässt sich dennoch nur eingeschränkt abbilden, da die Ausdrucksmächtigkeit in diesem Ansatz nicht ausreicht, um die Prozessschritte geeignet zu synchronisieren. Wir diskutieren die Einschränkungen der Modellierungsunterstützung detailliert in Abschnitt 5.5. Ferner betrachtet der Ansatz weder die dynamische Adaption der Produktstruktur, noch Mechanismen für die flexible Ausnahmebehandlung.

Ziel von PBWD ist nun, die erzeugte Prozessstruktur nach wirtschaftlichen Kriterien (Qualität, Kosten, Zeit) zu optimieren. Basierend auf diesen Kriterien werden die Produktstruktur und Produktionseigenschaften als Ausgangspunkt herangezogen, um mittels formaler Modelle daraus Regeln für den Produktionsprozess zu formulieren. Unter Berücksichtigung dieser Kriterien

[10] Eine detaillierte Diskussion der Nutzung von deklarativen, planbasierten Ansätzen findet sich in [Rei00].

können optimierte Prozessabläufe erzeugt werden. Die dynamische Adaption datengetriebener Prozessstrukturen und Ausnahmebehandlung ist jedoch nicht das Ziel von PBWD.

Operational Specification (OPS)

Ein weiterer Ansatz, der die Modellierung von Daten und Prozessen betrachtet, ist *Operational Specification* (*OPS*) [NC03, BGH+07, LBW07]. Hier haben Dokumente bzw. Objekte (*Business Artifacts*) eine zentrale Rolle, denn über sie wird die Prozessstruktur definiert. Für jedes Objekt wird ein Lebenszyklus (*Life Cycle*) bestehend aus Geschäftsaktivitäten (*Business Tasks*) angelegt. Lebenszyklen können über ihren Zugriff auf Datenbanken oder über direkte Abhängigkeiten zwischen Aktivitäten synchronisiert werden. Die Gesamtheit der Lebenszyklen aller Objekte und der Interaktionen zwischen ihnen definieren damit die Prozessstruktur.

Der Ansatz erlaubt zwar die Verknüpfung von Daten und Prozessen und die Prüfung auf strukturelle Korrektheit, deren Erzeugung über eine gegebene Produktstruktur wird jedoch nicht unterstützt [BGH+07]. Ausführungs- und Adaptionsaspekte werden in der Literatur nicht behandelt.

Object Behavior Diagrams

Ein Formalismus für die Spezifikation des Verhaltens von Objekten in objektorientierter Software wurde in [KS91, PS98] als *Behavior Diagrams* eingeführt. Der Ansatz stellt eine graphische Notation vor, mit der das Verhalten von Objekten für ihre spätere Implementierung in Form von *Life Cycles* beschrieben werden kann. Sie bestehen aus Zuständen und Aktivitäten, die Zustandsübergänge auslösen. Derartige *Life Cycles* können in *Petri-Netze*[11] transformiert werden. Des Weiteren erlaubt der Ansatz die Synchronisation unterschiedlicher *Life Cycles*.

Der Ansatz erlaubt die manuelle Beschreibung von datengetriebenen Prozessstrukturen. Während die operationale Semantik für einzelne *Life Cycles* durch ihre Umsetzung in Petri-Netze gegeben ist, wird die Synchronisation unterschiedlicher *Life Cycles* nicht formal beschrieben. Die dynamische Adaption der entstehenden Prozessstrukturen sowie Aspekte der Ausnahmebehandlung liegen nicht im Fokus des Ansatzes und werden daher nicht behandelt. Weitere Arbeiten, die auf der Basis dieses Ansatzes Geschäftsprozesse modellieren, werden in den Abschnitten 4.8 und 5.5 detailliert vorgestellt und diskutiert.

3.2.3 Integrierte Ansätze

Für die Unterstützung technischer Entwicklungsprozesse wurden im wissenschaftlichen Umfeld weitere Ansätze entwickelt, die sich durch die gleichwertige Betrachtung und Integration von Daten und Prozessen auszeichnen. Sie sind insbesondere deshalb relevant, da sie den Prozessstruktur-Lebenszyklus aus Abbildung 2.2 durchgängig unterstützen und nicht nur einzelne Phasen davon.

[11] Eine Einführung in die Grundlagen von Petri-Netzen gibt Abschnitt 4.8.2.

Workflow Management for Simultaneous Engineering Networks (SIMNET)

Das Projekt *Workflow Management for Simultaneous Engineering Networks* (*SIMNET*) integriert Produktdaten-Management und Prozess-Management und ist daher als ein Ansatz für das *Product-Life-Cycle-Management* zu verstehen (vgl. Abschnitt 3.1.3) [GS98, RC03b, RC03a]. Ein wesentliches Ziel des Ansatzes ist eine durchgängige Unterstützung technischer Entwicklungsprozesse mithilfe einer datenorientierten Prozesskoordination. Im Mittelpunkt steht dabei die Erzeugung und Änderung von Entwicklungsobjekten (*Konstruktionsparametern*) und der dafür benötigte Informationsfluss. Dies wird durch Verknüpfung von Prozessen mit Objekten erreicht. Der Prozess wird unter anderem durch die Vernetzung der Objekte mit Dokumenten und Aktivitäten beschrieben und erst zur Laufzeit anhand dieser Beziehungen gebildet.

Der Ansatz diskutiert Anforderungen und Lösungen auf einer fachlichen Ebene. Die formale Grundlage des Ansatzes (z.B. das Metamodell und die operationale Semantik) konnte der Literatur nicht entnommen werden. Die Modellierung der Prozessstruktur erfolgt manuell. Ausführungsaspekte werden in [RRA07] diskutiert, die Anforderungen aus Abschnitt 2.2.2 werden jedoch bei weitem nicht erfüllt. So sind keine Korrektheitskriterien für die modellierten Prozessstrukturen definiert. Die dynamische Adaption wird zwar von dem Ansatz ebenfalls adressiert, das Ziel ist jedoch die fachliche Beschreibung eines Änderungsprozesses (*Engineering Change Management*) und nicht die Definition formaler Änderungsoperationen und Konsistenzkriterien [RC03a]. Anforderungen an die Ausnahmebehandlung werden ebenfalls vorgestellt, aber nicht formal beschrieben und damit nicht in der für datengetriebene Prozessstrukturen nötigen Tiefe abgedeckt.

Adaptable and Human-Centered Environment for the Management of Development Processes (AHEAD)

Der Ansatz *Adaptable and Human-Centered Environment for the Management of Development Processes* (*AHEAD*) hat das Ziel, dynamische Entwicklungsprozesse zu koordinieren [Wes01, KKSW98, JSW99]. Idee ist die integrierte Betrachtung der Bereiche Projekt-, Prozess- und Datenmanagement. Prozesse werden in *AHEAD* mit Kontroll- und Datenflüssen modelliert. Dazu werden UML-Klassendiagramme angepasst und erweitert, indem Klassen (bzw. Stereotypen) von Aktivitäten angelegt werden. Sie erlauben durch spezielle Marken die Modellierung von Datenflüssen. Die Prozesse sind intern als Graphen (sog. *PROGRES*-Graphen) formalisiert, aus denen anschließend Programmcode für die Ausführung der Prozesse erzeugt werden kann. Korrektheitskriterien sind damit implizit gegeben, aber nicht formal definiert. Ferner ermöglichen Graphersetzungsregeln die Anpassung der Prozessgraphen zur Laufzeit. Die geplante und ungeplante Ausnahmebehandlung ist durch die Spezifikation von Event-Handlern innerhalb von UML-Kollaborationsdiagrammen realisiert.

Der Ansatz enthält interessante Aspekte hinsichtlich Integration von Daten- und Prozessstruktur, sowie der Ausnahmebehandlung. Die Modellierung von Prozessen mithilfe von Klassendiagrammen bzw. *PROGRES*-Graphen bringen jedoch eine beschränkte Ausdrucksmächtigkeit hinsichtlich bedingten Verzweigungen und Schleifen mit sich. Wir diskutieren diese Einschränkungen in Teil 2 detailliert.

Engineering Processes

Ziel der in [JE00, EJ01] beschriebenen *Engineering Processes* ist die Integration verschiedener Komponenten für das Management von Prozessen und Produktstrukturen. Das hierfür konzipierte Prozess-Management-System besteht aus einer Modellierungs-, Ausführungs-, Kontrollfluss- und Zustandskomponente.[12] Die Modellierung der Prozesse soll in verschiedenen Notationen möglich sein. Sie werden zur Ausführung in ein einheitliches Basismodell überführt. Eine Modellierungsunterstützung im Sinne der Integration von Daten- und Prozessstruktur ist nicht beschrieben.

Der Ansatz unterstützt prinzipiell die Ausführung von Prozessstrukturen sowie deren dynamische Adaption. Die Literatur beschreibt das Metamodell und die weiterführenden Konzepte allerdings nicht ausreichend formal, sodass keine Aussagen zu Korrektheitskriterien und Ausnahmebehandlung getroffen werden können. Eine Diskussion der Konzepte zur Adaption findet sich in Abschnitt 6.4.

Process-centered Software Engineering Environment (EPOS)

Ein weiteres System für das integrierte Produkt- und Prozess-Management resultiert aus dem Projekt *Process-centered Software Engineering Environment* (*EPOS*) [MCL+96, NWC97]. Das Projekt adressiert zwar Entwicklungsprozesse für objektorientierte Software, die abgeleiteten Anforderungen an die Unterstützung datengetriebener Prozessstrukturen sind den in Kapitel 2 gewonnenen Anforderungen sehr ähnlich. So werden ebenfalls die Integrierbarkeit komplexer Datenstrukturen und der zugehörigen Prozesse sowie die dynamische Adaption der Prozessstrukturen als Anforderungen identifiziert.

Der Ansatz unterscheidet unter anderem zwischen dem Produkt- und dem Prozessmodell. Das Produktmodell beschreibt die Struktur der Software auf Basis von *Entity-Relationship-Diagrammen* [Che76]. Das Prozessmodell beschreibt die Prozesse mit Kontroll- und Datenflüssen. Ähnlich dem WEP-Ansatz verfügt auch EPOS über Platzhalter für das sog. *Late Modeling* (vgl. Muster PP2 in [WRRM08]). Für die Erstellung der Prozessstruktur finden zielorientierte Methoden Anwendung, bei dem für jeden Prozess dessen Vor- und Nachbedingungen bezogen auf den Datenfluss definiert werden. EPOS erlaubt damit, ähnlich wie das Tool *DBPM* die automatische Erzeugung einer Prozessstruktur auf Basis einer gegebenen Produktstruktur. Die Ausdrucksmächtigkeit des verwendeten Formalismus ist allerdings beschränkt, denn es können lediglich sequentielle und parallele Aufgabenbearbeitungen, jedoch keine bedingten Verzweigungen und Schleifen, abgebildet werden.

EPOS erlaubt die manuelle Adaption der Prozessstruktur auf Basis einer geänderten Produktstruktur, allerdings werden korrektheitsgefährdende Adaptionen nicht ausgeschlossen. Die in Abschnitt 2.2.2 geforderten Mechanismen zur Ausnahmenbehandlung werden nicht ausreichend unterstützt.

[12] Die Aussagen der Literatur sind hier widersprüchlich. Während [JE00] die Zustandskomponente eigenständig ansiedelt, wird sie in [EJ01] als Teil der Prozess-Management-Systems beschrieben.

3.3 Zusammenfassung

Die Analyse des Standes der Technik in Praxis und Wissenschaft zeigt, dass eine Vielzahl von Ansätzen zur Integration von Daten und Prozessen existieren. Die meisten untersuchten Konzepte bringen eine ausreichende Ausdrucksmächtigkeit für die manuelle Erstellung einer Prozessstruktur mit. Viele der Ansätze beschäftigen sich jedoch lediglich mit Teilaspekten der in Abschnitt 2.2 geforderten durchgängigen Unterstützung datengetriebener Prozessstrukturen. Herausforderungen sind offensichtlich die durchgängige Integration von Daten und Prozessen zur Erzeugung einer datengetriebenen Prozessstruktur, sowie die beiden wichtigen Aspekte der dynamischen Adaption und der Ausnahmebehandlung.

Die kommerziellen Prozess-Management-Werkzeuge unterstützen die Koordination datengetriebener Prozessstrukturen im Allgemeinen nicht ausreichend, da sie keine ausgeprägte Produktdaten-Management-Funktionalität aufweisen und daher auch nicht über Funktionen zur Integration von Produktdaten- und Prozess-Management verfügen. Dies wird zwar durch Produktdaten-Management-Systeme sowie innovative Konzepte aus der Wissenschaft adressiert (z.B. *PBWD*, *Object Behavior Diagrams* und *EPOS*), diese bieten dennoch keine durchgängige Unterstützung für die Modellierung, Ausführung, Adaption und Ausnahmebehandlung. Weitere Ansätze mit umfassender Unterstützung aller Phasen des Prozesslebenszyklus sind zwar verfügbar (z.B. ADEPT), aber die Konzepte erlauben die Integration der Datenstrukturen nicht ausreichend. Dadurch bieten sie zum Beispiel keine ausreichende Modellierungsunterstützung.

In Teil II dieser Arbeit leiten wir aus den Anforderungen aus Abschnitt 2.2.2 (d.h. Modellierung, Ausführung, Adaption und Ausnahmebehandlung) konkrete Anforderungen an eine technische Lösung ab. Basierend darauf entwickeln wir ein vollständiges und durchgängiges Lösungskonzept für die Koordination datengetriebener Prozessstrukturen. Die in diesem Abschnitt vorgestellten Ansätze werden hierbei berücksichtigt und diskutiert.

Teil II

Technische Lösung

4

Grundlagen zur Koordination von Prozessstrukturen

Prozesse zur Entwicklung technischer Produkte (z.B. Fahrzeuge oder Flugzeuge) können sehr umfangreich sein. So erfordert die Entwicklung eines komplexen Systems mit hunderten bis tausenden Komponenten (z.B. das elektrische/elektronische System eines Fahrzeugs) die Koordination mindestens ebenso vieler miteinander vernetzter Prozesse (z.B. Prozesse zur Entwicklung, Absicherung und Freigabe jeder einzelnen Komponente). Die manuelle Koordination dieser Prozesse ist jedoch aufgrund der Vielzahl von Abhängigkeitsbeziehungen zwischen ihnen kaum beherrschbar. Deshalb kommt der angemessenen Beschreibung und Koordination derartiger Prozessstrukturen eine hohe Bedeutung zu.

Die grundlegenden Anforderungen an eine IT-Unterstützung großer Prozessstrukturen wurden bereits in Kapitel 2 vorgestellt. Davon ausgehend hat Kapitel 3 den Stand der Technik diskutiert und bewertet. Es hat sich gezeigt, dass aktuelle Ansätze den identifizierten Anforderungen bei weitem nicht gerecht werden. In Teil II dieser Arbeit werden nun innovative Konzepte zur Erfüllung dieser Anforderungen eingeführt. Um für die Koordination von Prozessen eine IT-Unterstützung bieten zu können, muss die Prozessstruktur als Ganzes geeignet beschrieben werden. Dazu konzipieren wir in diesem Kapitel ein Rahmenwerk (Metamodell; vgl. Abschnitt 1.1) für die formale Beschreibung und Koordination von Prozessstrukturen ohne direkt auf Modellierungsaspekte einzugehen. Darauf basierend werden wir in den anschließenden Kapiteln mächtige Konzepte für die datengetriebene Erzeugung von Prozessstrukturen, deren dynamische Adaption sowie die Behandlung von Ausnahmen, die sich unter anderem in diesem Kontext ergeben können, evaluieren.

Kapitel 4 gliedert sich wie folgt: Abschnitt 4.1 stellt Beispielszenarien für die Abbildung und Ausführung datengetriebener Prozessstrukturen vor und leitet daraus technische Anforderungen ab. Mit der Umsetzung dieser Anforderungen wird in Abschnitt 4.2 begonnen und ein Rahmenwerk zur Verknüpfung einzelner Objekte (z.B. eine Komponente der Produktstruktur), mit

den für ihre Bearbeitung notwendigen Prozessen, eingeführt. Die zugehörige operationale Semantik wird in Abschnitt 4.3 beschrieben. Abschnitt 4.4 diskutiert unverzichtbare dynamische Eigenschaften, die es zu erfüllen gilt, um eine korrekte (bzw. verklemmungsfreie) Ausführung zu gewährleisten. Abschnitt 4.5 beschreibt strukturelle Elemente für die Synchronisation von Prozessen unterschiedlicher Objekte und ermöglicht damit die Bildung von Prozessstrukturen. Die operationale Semantik dieser Prozessstrukturen wird in Abschnitt 4.6 beschrieben und Abschnitt 4.7 behandelt ihre dynamischen Eigenschaften. Abschnitt 4.8 diskutiert verwandte Arbeiten und Abschnitt 4.9 gibt eine Zusammenfassung des Kapitels.

4.1 Einleitung

Aufbauend auf den in Kapitel 2 vorgestellten Anforderungen behandeln wir in diesem Abschnitt folgende fundamentale Fragestellungen:

- Wie kann eine Prozessstruktur *komponentenorientiert* beschrieben werden, also die Verbindung zwischen Produkt und Prozessen ausgedrückt werden?
- Wie kann eine Prozessstruktur modularisiert und deren Elemente wiederverwendbar gemacht werden?
- Welchen Anforderungen muss eine Prozessstruktur genügen, damit sie die Prozesse der einzelnen Komponenten integrieren und korrekt koordinieren kann?
- Nach welchen konkreten Regeln muss eine Prozessstruktur ausgeführt werden, damit es während der Ausführung nicht zu unvorhersehbarem Verhalten kommt?
- Wie können ganze Prozessstrukturen nahtlos und transparent als Sub-Prozess in einen übergeordneten Gesamtprozess integriert werden?

Ausgehend von Beispielszenarien der Entwicklungsprozesse aus der Automobilindustrie leiten wir im Folgenden die spezifischen technischen Anforderungen an die Prozessunterstützung ab.

4.1.1 Motivation

Entwicklungsprozesse werden häufig auf Ebene einzelner Arbeitsschritte oder Prozessfragmente in IT-Systemen (d.h. prozessorientierten Informationssystemen) abgebildet (vgl. Abschnitt 1.1). Durch die Nutzung unterschiedlicher IT-Systeme für die verschiedenen Prozesse einer Prozessstruktur ist die Durchgängigkeit hinsichtlich Modellierung und Synchronisation jedoch nicht gewährleistet. Dadurch ist bislang eine manuelle Koordination, d.h. der manuelle Start und die Überwachung der einzelnen Prozesse im Sinne einer Prozessstruktur, notwendig. Dies hat sich insbesondere bei großen Prozessstrukturen als sehr aufwändig und fehleranfällig erwiesen. Ziel unseres Rahmenwerks ist es, diese manuelle Koordination der Prozesse einer Prozessstruktur durch eine geeignete IT-Unterstützung abzulösen.[1]

[1] Ziel ist nicht das Ersetzen der verschiedenen IT-Systeme, die die Ausführung der einzelnen Prozesse steuern. Vielmehr soll die Abfolge dieser Prozesse im Sinne der Prozessstruktur koordiniert werden.

Unsere Analysen von Prozessstrukturen in der Fahrzeugentwicklung haben gezeigt, dass die zu koordinierenden Prozesse in der Praxis lose gekoppelt sind. Das heißt, sie weisen keine Abhängigkeiten auf Ebene der einzelnen Aktivitäten, sondern *Ende-Start-Beziehungen* mit anderen Prozessen auf [BHR05, MHHR06]. Diese Beziehungen sind in der Regel durch die Relationen zwischen bearbeiteten Objekten bzw. deren Prozessen vorgegeben:

> *Wenn der Prozess Simulationstest für das System x beendet wurde, wird der Prozess Fahrversuch in Gesamtsystem y gestartet.*

Die Modellierung von Prozessstrukturen kann, wie wir in Kapitel 3 gesehen haben, prinzipiell aktivitätenorientiert geschehen (vgl. Abbildung 4.1). Es können Bezüge zu den bearbeiteten Objekten durch Ein- und Ausgabedaten bzw. direkte Prozesszuordnung hergestellt werden. Jedoch sind hier die Aspekte Modularisierung und Wiederverwendung nicht in dem für Prozessstrukturen erforderlichen Maße realisierbar, da aktivitätenorientierte Ansätze eine Prozessstruktur *im Ganzen* abbilden und der Bezug zur Produktstruktur nicht ausreichend hergestellt werden kann. So werden der automatisierte Aufbau und dynamische Änderungen (z.B. Hinzufügen oder Entfernen von Objekten führt zur Adaption der Prozessstruktur) prinzipbedingt nicht unterstützt. Die Herausforderung besteht nun darin, Prozessstrukturen nicht *im Ganzen*, sondern auf Basis der einzelnen Objekte einer gegebenen Produktstruktur zu beschreiben.

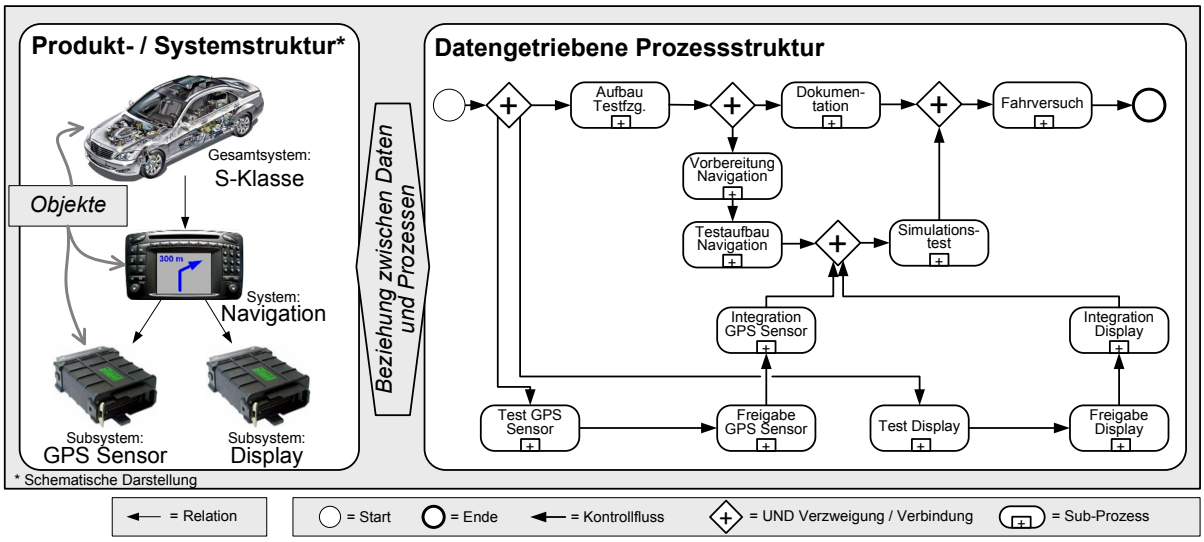


Abbildung 4.1: Datengetriebene Prozessstruktur

Die *zustandsorientierte Abbildung* hat sich als eine intuitive Beschreibung des Lebenszyklus einer Komponente (bzw. allgemein eines Objekts) erwiesen. Sie deckt sich mit bekannten Methoden zur Beschreibung von Abläufen in der Informatik [Boo90, KS91, Dor00, NC03, Bus06, Obj03b], aber auch mit Ansätzen aus dem Maschinenbau [Ver04]. Hierbei werden einzelne Komponenten eines Produkts bzw. Objekte eines Software-Modells in den Mittelpunkt gerückt. Werden diejenigen Zustände zusammengefasst, die ein Objekt während seiner *Lebensdauer* durchläuft, entsteht dessen *Objektlebenszyklus*. Zustandsübergänge innerhalb eines Objektlebenszyklus werden jeweils durch die Ausführung eines Prozesses (z.B. Test oder Freigabe) für dieses Objekt erreicht. Werden mehrere Objekte betrachtet, wie es beispielsweise in der Fahrzeugentwicklung üblicherweise der Fall ist, müssen deren Objektlebenszyklen zu einer *Prozessstruktur* zusammen-

gesetzt werden. Über die Verknüpfung der Zustände verschiedener Objektlebenszyklen können auch Abläufe beschrieben werden, die komplexeres Verhalten ausdrücken:

> *Erst wenn alle Subsysteme ihren Freigabe-Zustand erreicht haben, erreicht System z den Zustand „Testphase“ und der Prozess Simulationstest wird gestartet.*

Ziel des COREPRO-Ansatzes ist es, auf Objektlebenszyklen basierende Prozessstrukturen vollständig zu unterstützen.

In Ansätzen zur objektorientierten Software-Entwicklung werden Objektlebenszyklen als Modell für die Implementierung einer Software-Anwendung gesehen [KS91, EE97, AB97, Dor00, NC03]. Eine hart-codierte Implementierung von Prozessstrukturen ist in der Regel jedoch viel zu starr und unflexibel. Hinzu kommt, dass datengetriebene Prozessstrukturen in der Praxis unterschiedlich ausgeprägt und häufigen Änderungen unterworfen sein können (vgl. Kapitel 2). Die Änderung einer hart-codierten Implementierung wäre angesichts der Dynamik in diesem Umfeld allerdings viel zu aufwändig. Die Herausforderung ist daher, Prozessstrukturen auf Grundlage der verschiedenen Objektlebenszyklen generisch modellier- und leicht änderbar zu gestalten. Ziel ist es, eine Basis für konfigurierbare und lose gekoppelte Prozessstrukturen zu entwerfen. Sie sollen sich direkt ausführen lassen und die enthaltenen Prozesse koordinieren. Damit ergeben sich weitere Anforderungen an das Metamodell: Während viele Modellierungsansätze keine formal definierte operationale Semantik definieren (z.B. UML 2.0), müssen für die Ausführung von Prozessstrukturen sowohl die operationale Semantik, als auch geeignete Korrektheitskriterien präzise spezifiziert werden. Nur so lassen sich Aussagen zur Ausführbarkeit der modellierten Prozessstrukturen treffen. Wir beschreiben in den folgenden Abschnitten einen Basisformalismus, der diesen Anforderungen Rechnung trägt. Mit ihm lassen sich nicht nur das Verhalten einzelner Objekte durch ihren Objektlebenszyklus beschreiben, sondern auch die Abhängigkeiten zwischen Objektlebenszyklen. Der Basisformalismus ermöglicht nicht nur der Aufbau ganzer Prozessstrukturen, sondern stellt auch die Grundlage für weitergehende Mechanismen (z.B. dynamische Adaption).

4.1.2 Szenarien

Im Rahmen der Analyse der Entwicklungsprozesse für Fahrzeugelektronik (vgl. Abschnitt 2.1) wurden zahlreiche Szenarien für die Abbildung datengetriebener Prozessstrukturen und deren Ausführung untersucht. Im Folgenden sind relevante Szenarien aufgeführt, aus denen detaillierte Anforderungen für die IT-Unterstützung datengetriebener Prozessstrukturen abgeleitet werden können.[2]

Die Bildung eines Release in der E/E-Entwicklung (z.B. für den Aufbau eines Testfahrzeugs; vgl. Abschnitt 2.1) erfordert einerseits die Modellierung einer Systemstruktur aus den notwendigen Komponenten, andererseits die Erzeugung einer geeigneten Prozessstruktur für die Absicherung und Freigabe der Komponenten der Systemstruktur. Abhängig von der Systemstruktur des Release müssen die Prozesse für die einzelnen Komponenten geeignet koordiniert werden. Wir betrachten in diesem Kapitel zunächst die Koordination der Prozessstruktur, um auf dieser Grundlage in Kapitel 5 ihre automatische Erzeugung durch eine geeignete Modellierungsunterstützung zu diskutieren.

[2]Eine durchgängige Betrachtung weiterer Szenarien findet sich in Kapitel 9.

Die folgenden Szenarien reflektieren die Anforderungen an eine geeignete Koordination von Prozessstrukturen. Aus ihnen leiten wir Anforderungen an die technische Lösung ab (siehe Abschnitt 4.1.3).

Szenario 4.1 (Ablauf auf Komponentenebene)
Für Absicherung der korrekten Funktion einer Komponente müssen verschiedene Prozesse ausgeführt werden. Angenommen, eine Komponente[3] (z.B. Navigationseinheit) befindet sich im Zustand `entwickelt`, soll der zugehörige Testprozess ausgeführt werden. Ist der Prozess beendet, soll die Komponente nur dann in den Zustand `getestet` überführt werden können, wenn die zugehörigen Testergebnisse den Vorgaben entsprechen. Ansonsten soll die Komponente in den Zustand `fehlerhaft` wechseln. Der Zustandswechsel erfolgt somit in Abhängigkeit des Ergebnisses des Testprozesses. Wird die Komponente in den Zustand `fehlerhaft` überführt, soll zunächst durch einen entsprechenden Analyseprozess geprüft werden, ob der Fehler zügig behoben werden kann. Ist dies nicht der Fall, soll die Komponente in den Zustand `in Entwicklung` zurückgesetzt, ein Prozess zur Überarbeitung der Komponente sowie erneut der Testprozess ausgeführt werden.

Szenario 4.2 (Bildung von Prozessstrukturen)
Eine Systemstruktur umfasst verschiedene Komponenten, zum Beispiel verschiedene Systeme und Subsysteme (vgl. Abschnitt 2.1). In der zu einer Systemstruktur gehörenden Prozessstruktur sind der mit einer Komponente assoziierte Lebenszyklus bzw. dessen Prozesse enthalten. Im Allgemeinen sollen die Prozesse verschiedener Komponenten nebenläufig ausgeführt werden. Häufig hängen Komponentenzustände aber von Zuständen anderer Komponenten ab. Ein System (z.B. Navigationseinheit) im Zustand `getestet` kann beispielsweise erst dann in den Zustand `freigegeben` überführt werden, wenn sich die mit dem System assoziierten Subsysteme (z.B. GPS-Einheit und Soundsystem) bereits im Zustand `freigegeben` befinden. Der Zustand des Systems muss also mit den Zuständen seiner Subsysteme synchronisiert werden.

Synchronisationsbeziehungen, die zu einer Verklemmung führen, müssen ausgeschlossen werden können. Nehmen wir beispielsweise an, dass sich ein System im Zustand `in Entwicklung` befindet und erst dann getestet werden kann, wenn alle seine Subsysteme den Zustand `getestet` erreichen. Dann kann es zu einer Verklemmung kommen, wenn eines der Subsysteme aufgrund einer fehlerhaft modellierten Synchronisationsbeziehung wiederum darauf wartet, dass das übergeordnete System den Zustand `getestet` erreicht.

Szenario 4.3 (Synchronisationsprozesse)
In gewissen Szenarien erfordert die Synchronisation der Abläufe verschiedener Komponenten die Ausführung eines *Synchronisationsprozesses*. Synchronisationsprozesse haben die Ausgangskomponente als Eingabe und die Zielkomponente als Ausgabe (z.B. Integration eines *Subsystems* in einem *System*; vgl. Abbildung 4.1). Sie müssen aber nicht zwangsläufig die Zielkomponente verändern. So können Synchronisationsprozesse auch *nur* eine Information an den Verantwortlichen der Zielkomponente verschicken.

Szenario 4.4 (Globale Synchronisationsmuster)
Die Koordination von Prozessstrukturen kann verschiedenen Synchronisationsmustern folgen. Abbildung 4.2 beschreibt exemplarisch drei Muster, die in der Praxis häufig vorkommen. Die Muster sind durch vereinfachte Systemstrukturen, deren gerichtete Kanten die Synchronisationsrichtung der Bearbeitung andeuten, beschrieben. Abbildung 4.2a zeigt die *Top-down-Synchronisation*,

[3]Streng betrachtet eine *Komponentenversion* (siehe Kapitel 9).

bei der „von oben nach unten“ synchronisiert wird. Diese Vorgehensweise wird beispielsweise bei der Releasefreigabe benötigt. Die Releasefreigabe bestätigt die volle Funktionstüchtigkeit eines Release und dementsprechend der integrierten Systeme und Subsysteme. Der entsprechende Zustand wird „von oben nach unten“ propagiert. Die umgekehrte Reihenfolge, also *Bottom-up*, ist in Abbildung 4.2b dargestellt. Diese Vorgehensweise wird beispielsweise bei der Abbildung von Absicherungsprozessen benötigt, die im Allgemeinfall „von unten nach oben“ durchgeführt werden. Eine Mischung zeigt das Vorgehen gemäß *V-Modell* in Abbildung 4.2c [Drö00, Ver04, Deu99]. Die Konzepte zur Beschreibung datengetriebener Prozessstrukturen sollen die generische Abbildung von Synchronisationsmustern erlauben.

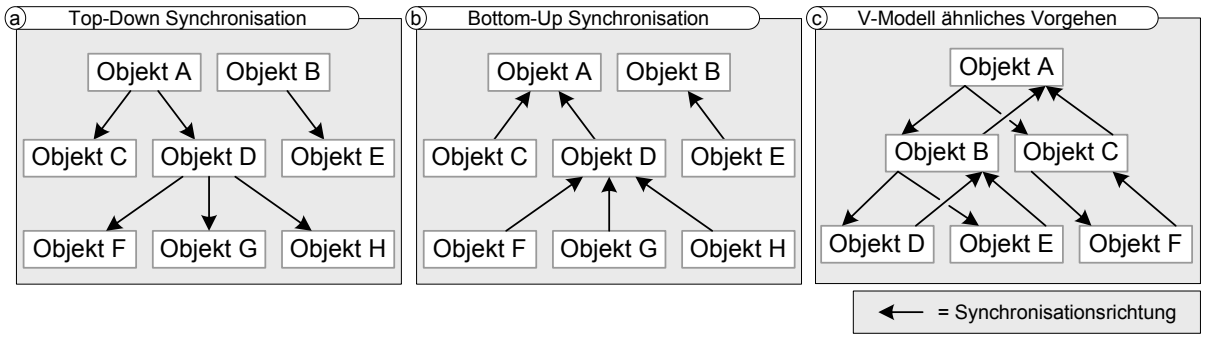


Abbildung 4.2: Globale Synchronisationsmuster

Szenario 4.5 (Integrierbarkeit in übergeordnete Prozesse)
Prozessstrukturen dürfen nicht nur isoliert betrachtet werden. Vielmehr sollen sie sich auch in übergeordnete Prozesse, beispielsweise den gesamtheitlichen Entwicklungsprozess einer Baureihe (vgl. Abschnitt 2.3.1), integrieren lassen. Sie müssen also Eigenschaften eines *Sub-Prozesses* aufweisen, damit sie in den umschließenden Prozess eingefügt und durch ihn ausgeführt werden können. Zu diesen Eigenschaften gehören ein definierter Beginn und ein definiertes Ende der Prozessstruktur, bei dem keiner ihrer Prozesse mehr ausgeführt wird.

Im Zusammenhang mit der Ausführung einer Prozessstruktur sind noch eine Reihe weiterer Szenarien relevant. Während der Durchführung der Release-Management-Prozesse etwa soll die Prozessstruktur bei Bedarf angepasst werden können. Ein Bearbeiter muss in der Lage sein, eine Komponente aus der Produktstruktur zu entfernen und ggf. eine andere Komponente einzufügen. Die dadurch notwendige Adaption der Prozessstruktur soll durch geeignete Operationen möglich sein. Dies kann jedoch zu Ausnahmen in der Prozessstruktur führen. Der Zustand `freigegeben` eines Systems kann zum Beispiel ungültig werden, wenn eines seiner Subsysteme entfernt wird. Derartige Szenarien werden ausführlich in den Kapiteln 6 und 7 behandelt.

4.1.3 Anforderungen

Eine zentrale Rolle bei der Realisierung der vorgestellten Szenarien stellen die Beschreibung von Objektlebenszyklen und aus ihnen gebildeten Prozessstrukturen dar. Bei reinen Modellierungsansätzen für Objektlebenszyklen oder Prozessstrukturen (z.B. [Boo90, KS91, Dor00, NC03,

Bus06, Obj03b]) steht die Dokumentation von Verhalten für die Implementierung einer Software-Anwendung im Vordergrund. In dieser Arbeit sind jedoch weitere Aspekte, wie operationale Semantik, Laufzeit-Korrektheit und dynamische Änderungen, fundamental. Um diese Aspekte angemessen unterstützen zu können, müssen das dynamische Verhalten der Objektlebenszyklen und der daraus erstellten Prozessstrukturen vollständig beschrieben, sowie deren Eigenschaften umfassend geprüft werden. Dies erfordert ein möglichst intuitives *Metamodell* auf dessen Basis Objektlebenszyklen und die sie verknüpfenden Prozessstrukturen beschrieben werden können.

Entwurfsziel für die formale Beschreibung einer Prozessstruktur ist ein praxisgerechter Kompromiss zwischen Ausdrucksmächtigkeit in der Modellierung und Flexibilität bei der Ausführung. Je komplexer das Modell zur Darstellung von Prozessstrukturen ist, desto unverständlicher wird es für Entwickler und Anwender und desto aufwändiger gestalten sich die Verifikation und Änderung der Modelle [Rei00]. Ein Beispiel für einen Kompromiss zwischen Ausdrucksmächtigkeit und Flexibilität ist der Umgang mit Nebenläufigkeit innerhalb einer Prozessstruktur. Nebenläufigkeit kann hier auf drei Ebenen auftreten:

1. *Prozessebene*: Innerhalb eines einzelnen Prozesses (z.B. Testprozess für eine Komponente), sollen Aktivitäten bzw. ganze Pfade parallel ausgeführt werden. Diese Art der Nebenläufigkeit ist in dem individuellen Prozess gekapselt und daher für die Beschreibung und Koordination von Objektlebenszyklen nicht relevant.

2. *Objektebene*: Prozesse für ein einzelnes Objekt sollen parallel ausgeführt werden; beispielsweise verschiedene Testverfahren für eine Komponente. Nebenläufigkeit innerhalb eines Objektlebenszyklus würde das Modell jedoch sehr komplex hinsichtlich Verifikation und Änderung machen.

3. *Prozessstrukturebene*: Innerhalb der Prozessstruktur sollen Prozesse für unterschiedliche Objekte parallel ausgeführt werden. Diese Art der Nebenläufigkeit ist zwingend notwendig, um parallele Entwicklungsmethoden aus der Praxis (*Concurrent Engineering*) adäquat abbilden zu können [LK94].

Die Untersuchung der Abläufe in der Praxis hat gezeigt, dass Nebenläufigkeit auf Objektebene im Allgemeinen durch die in Punkt 1 beschriebenen Mechanismen erfüllt werden kann. Alternativ kann das Objekt (bzw. die Komponente) in Teilobjekte (bzw. Teilkomponenten) gegliedert und damit sein Objektlebenszyklus aufgesplittet werden. Das erlaubt die Abbildung der Nebenläufigkeit durch parallele Ausführung der entstandenen Objektlebenszyklen (vgl. Punkt 3). Zur Reduktion der Komplexität sowie der Unterstützung dynamischer Adaption und flexibler Ausführung nehmen wir Nebenläufigkeit auf Objektebene (d.h. innerhalb eines Objektlebenszyklus) nicht in die Anforderungen auf.

Im Folgenden stellen wir die wichtigsten Anforderungen an ein technisches Rahmenwerk zur Beschreibung und Koordination von Prozessstrukturen vor. Die Anforderungen basieren auf der Anforderungsanalyse aus Kapitel 2 und den Szenarien aus Abschnitt 4.1.2.

Anforderung 4.1 (Ausdrucksmächtigkeit)
Das Metamodell soll über eine ausreichende Ausdrucksmächtigkeit zur Beschreibung von Prozessstrukturen verfügen. Es müssen Elemente bereitgestellt werden, mit denen sich Objektlebenszyklen, d.h. die Abfolge von Prozessen in Abhängigkeit von Objektzuständen, modellieren lassen.

Dazu gehört auch die Abbildung von nicht-deterministischen, zyklischen und nicht blockstrukturierten Objektlebenszyklen. Nicht-deterministische Objektlebenszyklen enthalten *bedingte Verzweigungen*, wobei der tatsächlich durchlaufene Pfad erst zur Laufzeit gewählt wird.

Verzweigungen können des Weiteren für die Modellierung von *Rücksprungpfaden* (d.h. im Sinne der Ablauflogik rückwärts gerichtete Kontrollflusskanten) und damit auch für die Modellierung von *Schleifen* genutzt werden. Das Verhalten kommt einer *Do-While-Schleife* (bzw. *Repeat-Until-Schleife*) gleich, deren Abbruchbedingung als Transitionsbedingung modelliert ist und damit von einem Prozessergebnis abhängt.

Anforderung 4.2 (Strukturierbarkeit und Wiederverwendung)
Die auf Grundlage des Metamodells erstellten Beschreibungen (*Modelle*) für Objektlebenszyklen sollen in Prozessstrukturen integriert und dort nebenläufig ausgeführt werden können. In diesem Kontext ist es notwendig, die Objektlebenszyklen innerhalb einer Prozessstruktur zu synchronisieren, um die Abbildung globaler Synchronisationsmuster zu realisieren. Die Synchronisationskriterien sollen nicht implizit und hart codiert in den Objektlebenszyklen modelliert werden, sondern in Form einer losen Kopplung angegeben werden. Dies erlaubt die Wiederverwendung der Objektlebenszyklen in unterschiedlichen Kontexten und gewährleistet eine geeignete Modellierungsunterstützung für datengetriebene Prozessstrukturen (siehe Kapitel 5) sowie deren einfache Adaptierbarkeit (siehe Kapitel 6).

Anforderung 4.3 (Ausführbarkeit)
Die erstellten Beschreibungen für Objektlebenszyklen und Prozessstrukturen sollen ausführbar sein. Dies setzt eine präzise operationale Semantik des Metamodells voraus. Sie bestimmt, unter welchen Bedingungen einzelne Prozesse gestartet werden und welche Zustandsänderungen sich bei deren Beendigung ergeben. Dies ist unabdingbar, um präzise Aussagen über das dynamische Verhalten von Prozessstrukturen treffen zu können. Zur Laufzeit soll die Ausführungsumgebung die nebenläufige Ausführung der Objektlebenszyklen innerhalb einer Prozessstruktur erlauben. Des Weiteren soll es nicht nur möglich sein, einfache Synchronisationsbeziehungen zwischen Objektlebenszyklen zu definieren, sondern diese mit der Ausführung (Koordination) von Synchronisationsprozessen zu verbinden (vgl. Szenario 4.3).

Die Darstellung der Laufzeitinformationen, zum Beispiel die Kennzeichnung des aktuell aktivierten Zustands, die Anzeige der zukünftig auszuführenden Prozesse oder die Sicherung des bisherigen Ablaufs der Prozessstruktur (*Historie*), ist aus mehreren Gründen wichtig. Erstens ist sie notwendig, um geeignete Regeln für die Ausführung einer Prozessstruktur zu definieren. Zweitens kann der Benutzer dadurch die bisherigen Abläufe nachvollziehen (dazu gehört z.B. auch, dass nicht gewählte Pfade entsprechend markiert werden). Drittens ist die modellinhärente Abbildung der Historie unabdingbar, um im Kontext der dynamischen Adaption die Korrektheit einer geänderten Prozessstruktur prüfen zu können [RD98, RRD04a, RRD04b]. Viertens muss eine Prozessstruktur in einen übergeordneten Prozess als Sub-Prozess einsetzbar sein. Voraussetzung dafür ist die Darstellung der Phase, in der sich eine Prozessstruktur derzeit befindet (z.B. „in Ausführung“ oder „beendet“).

Anforderung 4.4 (Korrektheit)
Während der Ausführung einer Prozessstruktur darf es nicht zu undefinierten Situationen oder Verklemmungen kommen. Aus diesem Grund müssen präzise Aussagen zur Korrektheit der erstellten Modelle getroffen werden. Idealerweise werden korrektheitsgefährdende Situationen durch

die strukturellen Eigenschaften der erstellbaren Modelle ausgeschlossen, indem zum Beispiel gar keine Verklemmungen in einem Objektlebenszyklus modellierbar sind. Auf Basis der strukturellen und dynamischen Eigenschaften müssen auch Aussagen zur Terminierung der Prozessstruktur getroffen werden, um die Integration in übergeordnete Prozesse adäquat zu realisieren.

Anforderung 4.5 (Änderbarkeit und flexible Ausführung)
Die Änderbarkeit und flexible Ausführung von Prozessstrukturen muss gewährleistet werden (siehe Kapitel 6 und 7). Zwar wird in diesem Kapitel auf technische Lösungen noch nicht explizit eingegangen, dennoch ist durch den Entwurf eines geeigneten Metamodells sicherzustellen, dass sie grundsätzlich realisierbar sind. So müssen das Einfügen und Entfernen von Objektlebenszyklen in bzw. aus einer Prozessstruktur sowie das Hinzufügen oder Entfernen von Synchronisationsbeziehungen zwischen Objektlebenszyklen (vgl. Anforderung 4.3) angemessen unterstützt werden. Dasselbe gilt für die Durchführung von Vorwärts- oder Rückwärtssprüngen, die im Rahmen der Ausnahmebehandlung notwendig werden (siehe Kapitel 7). Ein kompaktes Modell mit Historie und klaren Strukturen ist dafür Voraussetzung.

Um die oben genannten Anforderungen vollständig erfüllen zu können, müssen wir einige Eigenschaften für die zu koordinierenden Prozesse fordern. Wir nehmen an, dass die Prozesse einer Prozessstruktur atomar und gekapselt sind. Das heißt, sie haben einen definierten Beginn sowie ein definiertes Ende und terminieren immer. Des Weiteren gehen wir davon aus, dass Prozesse ein bestimmtes Prozessergebnis als Ausgabe liefern (z.B. Testprozess mit den Ergebnissen „Komponente in Ordnung“ oder „Komponente fehlerhaft“). Das Prozessergebnis wird in nichtdeterministischen OLCs für die Auswahl des Pfads von bedingten Verzweigungen benötigt.

4.2 Struktur eines Object Life Cycle (OLC)

Basierend auf den in Abschnitt 4.1 beschriebenen Szenarien und Anforderungen stellen wir in diesem Abschnitt eine technische Lösung für die Abbildung von Objektlebenszyklen vor. Dafür definieren wir ein formales Metamodell und diskutieren seine strukturellen Eigenschaften. Objektlebenszyklen, die auf Basis des Corepro-Rahmenwerks beschrieben werden, nennen wir *Object Life Cycles (OLCs)* [MRH07, MRH08a].[4]

4.2.1 Aufbau

Ein OLC beschreibt die Zustände, die ein Objekt zur Laufzeit annehmen kann, sowie die zugehörigen Zustandsübergänge. Er wird in Corepro durch ein *Transitionssystem* [CKLY98] bzw. einen endlichen Automaten repräsentiert. Dessen *Zustände* stellen die möglichen Objektzustände dar, während seine (internen) *Transitionen* die Zustandsübergänge des Objekts abbilden. Zustände können nur mittels Transitionen verknüpft werden, Transitionen wiederum nur über Zustände. Das Transitionssystem eines OLC verfügt über je einen definierten Start- und Endzustand, was exakte Aussagen zum Beginn und zur Terminierung des OLC erlaubt. Abbildung 4.3 zeigt zwei OLCs sowie die zu ihrer Darstellung verwendete graphische Notation (d.h. Symbole

[4] Wir definieren für *Object Life Cycles* im Corepro-Ansatz eine eigene formale Grundlage. Sie sind nicht zu verwechseln mit *Life Cycles* aus anderen Ansätzen (z.B. *Object/Behavior Diagrams* aus [KS91]).

für Start- und Endzustände, Objektzustände und Transitionen). Im Folgenden nehmen wir an, dass innerhalb eines OLC die verschiedenen Zustände immer eindeutig benannt sind.

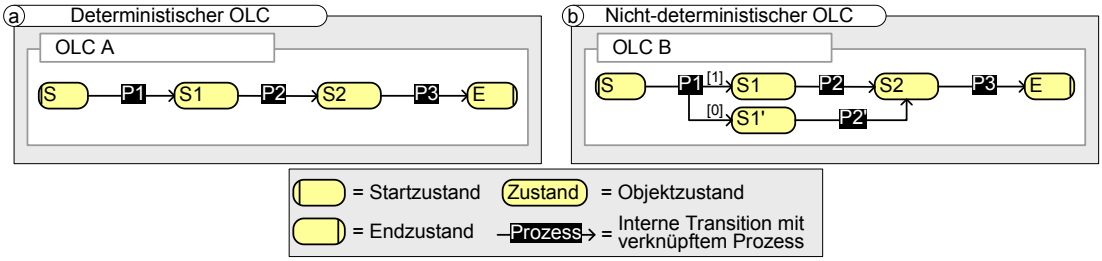


Abbildung 4.3: Deterministischer und nicht-deterministischer OLC

Eine Transition verbindet einen *Quell-* mit einem *Zielzustand*. Generell kann jede Transition mit einem Prozess (bzw. Prozessschema) verknüpft werden. Während der OLC-Ausführung wird für jede Transition eine neue Instanz des Prozesses erstellt und gestartet (siehe Abschnitt 4.3.2). Wird die Prozessinstanz beendet, führt die mit ihr verknüpfte Transition zu dem beschriebenen Zustandsübergang: Der Zielzustand wird aktiviert, wohingegen der Quellzustand als „durchlaufen“ markiert wird (vgl. Anforderung 4.1).[5] Der OLC aus Abbildung 4.3a durchläuft zum Beispiel in jedem Fall die vier Zustände `S`, `S1`, `S2` und `E`, die über Transitionen verbunden sind. Mit diesen Transitionen wiederum sind die Prozesse `P1`, `P2` und `P3` verknüpft. Wird der Zustand `S` aktiviert, so startet Prozess `P1`. Sobald `P1` endet, wird `S1` aktiviert und anschließend mit der Ausführung von `P2` begonnen.[6] Dies wird fortgesetzt, bis der Endzustand des OLC erreicht ist.

Für die Erfüllung von Anforderung 4.1 (Ausdrucksmächtigkeit) müssen auch nicht-deterministische Objektlebenszyklen (d.h. Objektlebenszyklen mit bedingten Verzweigungen) abbildbar sein. Dies kann durch die Definition mehrerer aus einem Quellzustand ausgehender Transitionen realisiert werden. Da Nebenläufigkeit auf der Ebene einzelner OLCs ausgeschlossen wird (vgl. hierzu die Diskussion in Abschnitt 4.1.3), müssen Transitionen mit demselben Quellzustand immer mit demselben Prozess (d.h. mit derselben Prozessinstanz) verknüpft werden.[7]

Ein Prozess liefert nach seiner Beendigung ein *Prozessergebnis*. Die Angabe einer *Transitionsbedingung* für jede Transition einer bedingten Verzweigung erlaubt zur Laufzeit die Wahl des entsprechenden Pfads anhand des tatsächlich gelieferten Ergebnisses. Um auszuschließen, dass zur Laufzeit mehrere Transitionen gewählt werden, müssen die mit den Transitionen verknüpften Ergebnisse disjunkt sein. Des Weiteren müssen alle durch einen Prozess gelieferten Ergebnisse durch Transitionsbedingungen abgefangen werden, da ansonsten nicht immer eine Transition gewählt werden und dies zum Blockieren der OLC-Ausführung führen könnte.[8] In Abbildung 4.3b ist die Ergebnismenge des Prozesses `P1` gegeben durch $\sigma(\texttt{P1}) = \{0, 1\}$, wobei die Transition mit Zielzustand `S1` mit dem Ergebnis $v = 1$ und die Transition mit Zielzustand `S1'` mit dem

[5] Ein Zustandsübergang kann auch in Abhängigkeit der Zustände anderer OLCs stattfinden (siehe Abschnitt 4.5).
[6] Da Prozessschemas mit unterschiedlichen Transitionen verknüpft sein können, erfolgt die Aktivierung des Nachfolgezustands nicht durch den Prozess selbst. Stattdessen erfolgt der Zustandswechsel auf Basis formaler Regeln (siehe Abschnitt 4.3.7).
[7] Um diese Eigenschaft in Abbildungen auszudrücken, sind interne Transitionen mit demselben Quellzustand „bis“ zum Prozess als einzelne Kante dargestellt. Sie verzweigen erst dann zu den verschiedenen Zielzuständen (vgl. Abbildung 4.3b, Transition mit Prozess `P1`).
[8] Eine Diskussion zu nicht abgefangenen Ergebnissen findet sich in Abschnitt 4.9.

Ergebnis $v = 0$ verknüpft ist. Endet `P1` mit Ergebnis 1, wird als Nachfolgezustand entsprechend `S1` aktiviert, während `S1'` abgewählt wird (zur exakten operationalen Semantik dieser Abwahl siehe Abschnitt 4.3). Hat ein Quellzustand genau eine ausgehende Transition mit Prozess p (d.h. es handelt sich nicht um eine bedingte Verzweigung), wird auf die explizite Angabe einer Transitionsbedingung v verzichtet, d.h. hier gilt standardmäßig: $v = \sigma(p)$.[9] Definition 4.1 beschreibt den formalen Aufbau eines OLC.

Definition 4.1 (Object Life Cycle)
Seien $\mathcal{O}$ die Menge aller Objekte und $o \in \mathcal{O}$ ein Objekt. Dann ist der Object Life Cycle (OLC) von o beschrieben durch ein Tupel $olc = (P, V, TS)$ mit

- *P ist eine Menge von Prozessen, die auf o angewendet werden können*
- *V beschreibt die Menge aller möglichen Prozessergebnisse ($\sigma : P \mapsto \mathcal{P}(V)$ mit $\sigma(p) \subseteq V$ ordnet jedem Prozess $p \in P$ die Menge seiner möglichen Prozessergebnisse zu)*
- *$TS = (S, T, s_{start}, s_{end})$ ist ein Transitionssystem mit:*
  - *S ist die Menge der Zustände, die o während seiner Bearbeitung annehmen kann.*
  - *$T \subseteq S \times (P \times V) \times S$ ist eine Menge von internen Transitionen des OLC mit*
    - *$t = (src, (p, v), targ) \in T, \Rightarrow v \in \sigma(p)$; d.h. eine interne Transition vom Quellzustand src zum Zielzustand $targ$ wird durch die Beendigung von Prozess p mit Prozessergebnis v ausgelöst.*
    - *Für $t_i = (src_i, (p_i, v_i), targ_i) \in T, i = 1, 2$ mit $t_1 \neq t_2$ gilt: $src_1 = src_2, \Rightarrow p_1 = p_2 \wedge v_1 \neq v_2$; d.h. gibt es mehrere interne Transitionen mit gemeinsamem Quellzustand, so werden diese Transitionen mit demselben Prozess verknüpft. Der Zielzustand wird zur Laufzeit auf Basis des gelieferten Prozessergebnisses von p_1 ausgewählt (Transitionsbedingung).*
  - *$s_{start} \in S$ ist der* Start- *und $s_{end} \in S$ der* Endzustand *des Transitionssystems TS; s_{end} ist der einzige Zustand des OLC ohne ausgehende interne Transitionen.*

$\mathcal{OLC}$ beschreibe die Menge aller Object Life Cycles. Für $olc \in \mathcal{OLC}$ bezeichnet $s_{start}(olc)$ den Start- und $s_{end}(olc)$ den Endzustand des jeweiligen Transitionssystems.

4.2.2 Funktionen zur strukturellen Analyse

Für nachfolgende Betrachtungen (z.B. Spezifikation der operationalen Semantik) ist es notwendig, OLCs strukturell analysieren zu können. Zu diesem Zweck definieren wir verschiedene Hilfsfunktionen (vgl. Definition 4.2). Die Funktionen $outTrans_{int}(s)$ und $inTrans_{int}(s)$ liefern für einen gegebenen Zustand s dessen ein- und ausgehende interne *Transitionen*. Die Funktion $path_{int}(s_0, s_1)$ liefert die Menge aller (internen) *Transitionen*, die auf einem *zyklenfreien Pfad* zwischen Zustand s_0 und Zustand s_1 liegen, d.h. diejenigen Transitionen, über die, beginnend bei s_0, der Zustand s_1 direkt erreicht werden kann. Transitionen, die zu einem Rücksprung auf dem Pfad zwischen s_0 und s_1 führen, sind in $path_{int}(s_0, s_1)$ nicht enthalten.

[9] Eine Diskussion zur Verwendung von Ergebnismengen als Transitionsbedingungen findet sich in Abschnitt 4.9.

Definition 4.2 (Funktionen zur strukturellen Analyse eines OLC)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$. Dann gilt:

(a) $outTrans_{int}(s)$ liefert für einen Zustand $s \in S$ die Menge aller aus s ausgehenden internen Transitionen: $outTrans_{int}(s) := \{t \in T | t = (src, (p, v), targ) \wedge src = s\}$.

(b) $inTrans_{int}(s)$ liefert für einen Zustand $s \in S$ die Menge aller in s eingehenden internen Transitionen: $inTrans_{int}(s) := \{t \in T | t = (src, (p, v), targ) \wedge targ = s\}$.

(c) $path_{int}(s_0, s_1)$ liefert die Menge aller internen Transitionen, die auf dem zyklenfreien Pfad zwischen s_0 und s_1 liegen (vgl. Algorithmus 4.1). Rücksprungpfade *(d.h. Transitionen, die innerhalb des Pfads von s_0 zu s_1 zu einem Zyklus führen) sind nicht enthalten.*[a]

(d) $states_{int}(s_0, s_1)$ liefert die Menge aller Zustände, die auf dem zyklenfreien Pfad zwischen s_0 und s_1 liegen: $states_{int}(s_0, s_1) := \{s \in S | s = s_0 \vee \exists\, t = (src, (p, v), s) \in path_{int}(s_0, s_1)\}$.

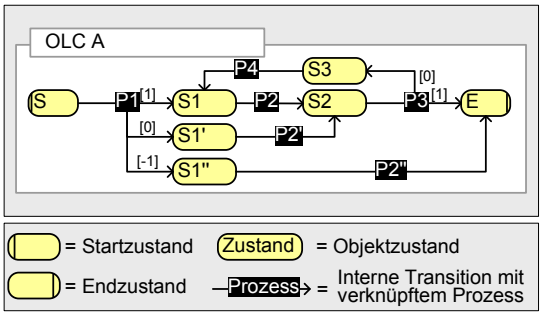
[a] Wir verzichten auf eine formale Definition der Funktion für $path_{int}$ und beschreiben sie stattdessen über Algorithmus 4.1. Aus Symmetriegründen wurde $path_{int}$ trotzdem in die Definition der Hilfsfunktionen aufgenommen.

Die in Definition 4.2 eingeführten Funktionen werden im Folgenden verwendet, um die operationale Semantik eines OLC zu beschreiben. Hierbei sind insbesondere die *Hauptpfade* eines OLC von Bedeutung, also diejenigen Transitionen, die sich auf einem zyklenfreien Pfad zwischen seinem Start- und Endzustand (d.h. s_{start} bzw. s_{end}) befinden. Für einen Object Life Cycle $olc = (P, V, TS) \in \mathcal{OLC}$ mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ gilt:

- $path_{int}(s_{start}, s_{end})$ beschreibt die *Hauptpfade* von *olc*. Darin sind alle Transitionen enthalten, die sich auf einem Pfad zwischen Start- und Endzustand des OLC befinden und die nicht Teil eines Rücksprungpfads sind.

- $states_{int}(s_{start}, s_{end})$ umfasst die Zustände von *olc*, die sich auf den Hauptpfaden befinden.

Abbildung 4.4 zeigt einen OLC mit den Zuständen `S`, `S1`, `S1'`, `S1''`, `S2`, `S3` und `E` sowie beispielhaft die Anwendung der oben definierten Funktionen.



Beispiele für die Anwendung der Funktionen zur strukturellen Analyse

a) `outTrans`$_{int}$`(S1)` liefert die Transition `t=(S1,(P2,σ(P2)),S2)` von S1 nach S2 (mit Prozess P2).

b) `intTrans`$_{int}$`(S1)` liefert die Transition `t1=(S,(P1,{1}),S1)` von S nach S1 (mit Prozess P1) und die Transition `t2=(S3,(P4,σ(P4)),S1)` von S3 nach S1 (mit Prozess P4).

c) `path`$_{int}$`(S,S2)` liefert die Transitionen `t1=(S,(P1,{1}),S1)`, `t2=(S,(P1,{0}),S1')`, `t3=(S1,(P2,σ(P2)),S2)` und `t4=(S1',(P2,σ(P2)),S2)`.

d) `path`$_{int}$`(S,E)` liefert alle Transitionen des OLC außer dem Rücksprungpfad mit seinen Transitionen `t1=(S2,(P3,{0}),S3)` und `t2=(S3,(P4,σ(P4)),S1)`.

e) `states`$_{int}$`(Sstart,Send)` liefert die Menge aller Zustände, die auf den Hauptpfaden des OLC liegen, also S, S1, S1', S1'', S2 und E.

Abbildung 4.4: Nicht-deterministischer OLC mit Rücksprung

```
1  // Zunächst Definition einer rekursiven Hilfsfunktion für die Sammlung der
       aus einem Zustand ausgehenden Pfade
2  Function pathSearch(State st, State en, State current, Set T)
3  Input : State st (Beginn der Suche), State en (Ziel der Suche), State current (aktueller
           Zustand), Set T (Menge der im Teilpfad durchlaufenen Transitionen)
4  Output : Set (Menge der im Teilpfad durchlaufenen Transitionen inklusive der
           Folgepfade von current)
5  begin
6      // Abbruch, wenn Ziel erreicht
7      if current = en then return T;
8      // Abbruch bei Zyklus oder Endzustand des OLC
9      if (∃ t = (src, (p, v), targ) ∈ T : src = current) ∨ (outTrans_int(current) = ∅) then
       return ∅;
10     if |outTrans_int(current)| = 1 then
11         // Wenn Sequenz, dann rekursiver Aufruf mit aktueller Liste
12         t = (src, (p, v), targ) ∈ outTrans_int(current);
13         T.add(t);
14         return pathSearch(st, en, targ, T);
15     else
16         // Wenn Verzweigung, dann Kopie der Liste erstellen, rekursiver
               Aufruf mit der Kopie und Ergebnis anschließend hinzufügen
17         forall t = (src, (p, v), targ) ∈ outTrans_int(current) do
18             TP = T.copy();
19             TP.add(t);
20             T.add(pathSearch(st, en, targ, TP));
21         end
22     end
23     return T;
24 end
25 // Function path_int liefert zyklenfreien Pfad zwischen zwei Zuständen
26 Function path_int(State s0, State s1)
27 Input : State s0 (Beginn der Suche), State s1 (Ende der Suche)
28 Output : Set (Transitionen, die auf einem zyklenfreien Pfad zwischen s0 und s1 liegen)
29 begin
30     // Aufruf der rekursiven Funktion pathSearch mit aktuellem Zustand s0
           (Beginn der Suche) und initial leerer Menge transFound
31     Set TransFound = new Set();
32     return pathSearch(s0, s1, s0, TransFound);
33 end
```

Algorithmus 4.1 : Funktion $\texttt{path}_{int}$

Die nachfolgenden Betrachtungen zur operationalen Semantik von OLCs erfordern eine Unterscheidung der verschiedenen *Abschnitte* eines OLC. Ein Abschnitt beschreibt die Elemente, die auf einen Zustand folgen. Abbildung 4.5 zeigt die drei unterschiedenen OLC-Abschnitte: *Sequenzabschnitt*, *Verzweigungsabschnitt* und *Verzweigungsabschnitt mit Rücksprung*. Die Unterscheidung von Abschnitten (vgl. Definition 4.3) erfolgt auf Basis der in Definition 4.2 eingeführten Hilfsfunktionen.

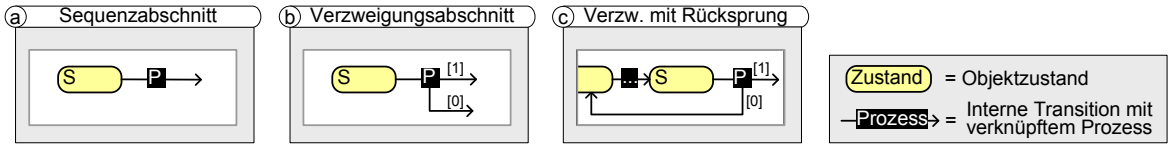


Abbildung 4.5: Klassifikation von Abschnitten

Definition 4.3 (Abschnitte eines OLC)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Dann unterscheiden wir folgende Abschnitte des OLC:*

(a) Die Menge **seq** *der* **Sequenzabschnitte** *von olc umfasst die Zustände, die höchstens eine ausgehende interne Transition und damit höchstens einen Folgezustand besitzen (vgl. Abbildung 4.5a):*

$$seq(olc) := \{s \in S \mid |outTrans_{int}(s)| \leq 1\}$$

(b) Die Menge **splits** *der* **Verzweigungsabschnitte** *von olc umfasst die Zustände, die mindestens zwei ausgehende Transitionen und damit mehrere Nachfolgezustände besitzen (vgl. Abbildung 4.5b):*

$$splits(olc) := \{s \in S \mid |outTrans_{int}(s)| > 1\}$$

(c) Die Menge **splitLoops** *der* **Verzweigungsabschnitte mit Rücksprung** *von olc umfasst die Zustände, die mindestens zwei ausgehende Transitionen und damit mehrere Nachfolgezustände besitzen und bei denen sich mindestens eine ausgehende Transition in einem Rücksprungpfad befindet (vgl. Abbildung 4.5c):*

$$splitLoops(olc) := \{s \in splits(olc) \mid \exists\, t \in outTrans(s) : t \notin path_{int}(s_{start}, s_{end})\}^a$$

[a] es gilt: $splitLoops(olc) \subseteq splits(olc)$.

Für den OLC `A` aus Abbildung 4.4 ergeben die folgenden Mengen (vgl. Definition 4.3):

- $seq(A) = \{\texttt{S1}, \texttt{S1'}, \texttt{S1''}, \texttt{S3}, \texttt{E}\}$
- $splits(A) = \{\texttt{S}, \texttt{S2}\}$
- $splitLoops(A) = \{\texttt{S2}\}$

4.2.3 Statische Eigenschaften

Die Unterscheidung von OLC-Abschnitten (vgl. Definition 4.3) kann genutzt werden, um OLCs hinsichtlich ihrer strukturellen Eigenschaften zu klassifizieren. Entsprechend Definition 4.4 unterscheiden wir drei Klassen von OLCs: *deterministische OLCs*, *nicht-deterministische azyklische OLCs* sowie *nicht-deterministische zyklische OLCs* (vgl. Abbildung 4.6). Basierend auf dieser Klassifikation definieren wir in Abschnitt 4.3 sukzessive die operationale Semantik von OLCs.

Definition 4.4 (Klassifikation von OLCs)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Dann heißt olc*

(a) **deterministisch**, *wenn TS keine Verzweigungen besitzt; d.h. es gilt:* $splits(olc) = \emptyset$. *Jeder Zustand von olc hat eine oder keine ausgehende interne Transition (vgl. Abbildung 4.6a).*

(b) **nicht-deterministisch azyklisch**, *wenn TS Verzweigungsabschnitte besitzt, diese aber nicht zu einem Rücksprung führen; d.h. es gilt:* $splits(olc) \neq \emptyset \wedge splitLoops(olc) = \emptyset$ *(vgl. Abbildung 4.6b). Entsprechend werden in einem nicht-deterministischen OLC zur Laufzeit nicht immer alle Zustände erreicht (vgl. Abbildung 4.6b).*

(c) **nicht-deterministisch zyklisch**, *wenn TS Verzweigungsabschnitte besitzt, von denen mindestens einer zu einem Rücksprung führt (vgl. Abbildung 4.6c); d.h. es gilt:* $splitLoops(olc) \neq \emptyset$. *Zustände innerhalb des Zyklus können mehrfach erreicht werden.*

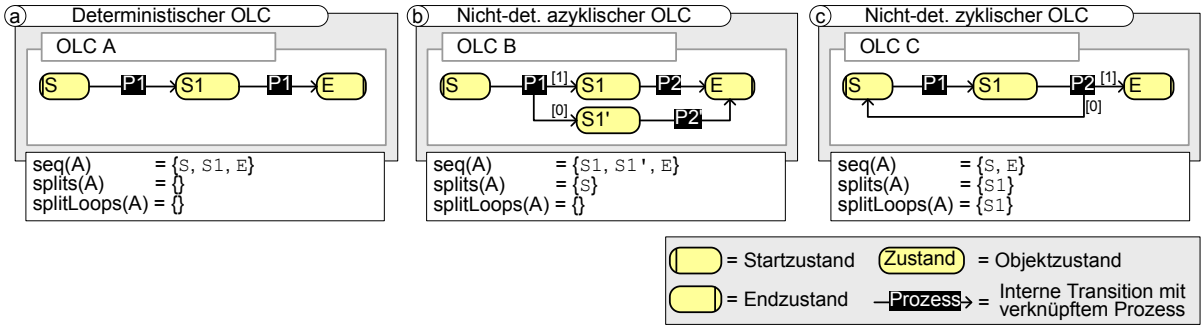


Abbildung 4.6: Klassifikation von OLCs

Unabhängig von ihrer Einordnung sind OLCs immer *determiniert*. Das heißt, bei mehrmaliger Ausführung wird zur Laufzeit bei gleicher Eingabe immer der gleiche Pfad innerhalb des OLC gewählt. Voraussetzung ist, dass die verknüpften Prozesse ebenfalls *determiniert* sind und die gleichen Prozessergebnisse liefern (vgl. Abschnitt 4.1.3).

Bevor wir auf die operationale Semantik eines OLC eingehen, diskutieren wir strukturelle Korrektheitseigenschaften von OLCs. Beispielsweise wollen wir OLCs ausschließen, deren Zustände nicht korrekt durch Transitionen verbunden sind und deren Ausführung daher nicht möglich ist. In [Aal97b, Wes07] werden verschiedene Kriterien für die Prüfung der Korrektheit eines Prozesses vorgestellt. Hierbei wird zwischen *struktureller Korrektheit* (*Structural Soundness*) und *dynamischer Korrektheit* (*Soundness*) unterschieden. Strukturelle Korrektheit beschäftigt sich mit der

korrekten Verknüpfung von Knoten und Transitionen. Die Literatur beschreibt die Kriterien für Korrektheit meist auf Basis von *Workflow-Netzen* (Workflow Nets), einer auf *Petri-Netzen* basierenden Notation.[10] Strukturelle Fehler werden bereits mit der Definition von Workflow-Netzen ausgeschlossen, indem folgende Kriterien erfüllt sind [Aal97b, Aal98, Wes07]:

(a) Es gibt in einem Workflow-Netz genau einen Startknoten, der keine eingehenden Transitionen besitzt.

(b) Es gibt in einem Workflow-Netz genau einen Endknoten, der keine ausgehenden Transitionen besitzt.

(c) Jede Transition bzw. jeder Knoten muss ausgehend vom Startknoten erreichbar sein. Genauso muss, ausgehend von jeder beliebigen Transition bzw. jedem Knoten der Endknoten des Workflow Netzes erreichbar sein. Damit sind „nicht verbundene“ Transitionen und Knoten ausgeschlossen.

Die genannten Kriterien lassen sich auf OLCs übertragen. Die Bedingungen (a) und (b) der strukturellen Korrektheit ergeben sich bereits aus der formalen Definition von OLCs und durch den Ausschluss von Nebenläufigkeit (vgl. Definition 4.1). Bedingung (c) ist für Transitionen ebenfalls durch die formale Definition von OLCs erfüllt, für Zustände kann sie mithilfe der Funktion $path_{int}$ aus Definition 4.2 sichergestellt werden (vgl. Definition 4.5).

Definition 4.5 (Strukturelle Korrektheit eines OLC)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$. Dann ist olc strukturell korrekt, wenn gilt:

- *Es gibt eine Folge von (internen) Transitionen, über die vom Startzustand s_{start} jeder Zustand von olc erreicht werden kann, d.h.*

$$\forall\, s \in S, s \neq s_{start} : path_{int}(s_{start}, s) \neq \emptyset$$

- *Es gibt eine Folge von (internen) Transitionen, über die von jedem Zustand von olc der Endzustand s_{end} erreicht werden kann, d.h.*

$$\forall\, s \in S, s \neq s_{end} : path_{int}(s, s_{end}) \neq \emptyset$$

Wir schränken die Betrachtungen in dieser Arbeit auf strukturell korrekte OLCs ein. Die strukturelle Korrektheit eines OLC ist auch Voraussetzung für die Analyse der dynamischen Korrektheit (siehe Abschnitt 4.4).

4.3 Operationale Semantik von Object Life Cycles

Ein wichtiges Ziel des CoreFro-Ansatzes ist die Ausführung von OLCs bzw. Prozessstrukturen. Basierend auf der strukturellen Spezifikation von Object Life Cycles (OLCs), führen wir nun

[10] In Workflow-Netzen bzw. Petri-Netzen haben Transitionen eine leicht andere Bedeutung als in unseren OLCs. Dies ist jedoch für die Korrektheitsuntersuchung zunächst nicht von Belang.

eine zugehörige *operationale Semantik* ein. Sie ist unabdingbar für die korrekte Ausführung von OLCs. Konkret beschreibt sie das Verhalten eines einzelnen OLC (also einer OLC-Instanz) zur Laufzeit und legt damit fest,

- wann Zustände aktiviert werden und in welchen Situationen ein Zustandswechsel durchgeführt werden muss,
- wann die mit Transitionen verknüpften Prozesse gestartet werden können und was nach deren Beendigung geschieht und
- wie ein OLC gestartet wird und wann er als beendet gilt.

Ziel dieses Abschnitts ist die Definition einer eindeutigen und konsistenten operationalen Semantik zur Erfüllung der in Abschnitt 4.1.3 diskutierten Anforderungen an die Ausführung (vgl. Anforderung 4.3) und Verifikation von OLCs (vgl. Anforderung 4.4).

In Prozess-Management-Systemen bildet die *Laufzeitmarkierung* (kurz: *Markierung*) einzelner Prozesselemente häufig die Basis für die Beschreibung der operationalen Semantik [RD98, LR00]. Aktivitäten werden beispielsweise mit den Markierungen *Activated* oder *Running* versehen, um ihren aktuellen Ausführungszustand anzuzeigen. Die im Folgenden dargestellte operationale Semantik nutzt derartige Markierungen für die Ausführung von OLCs. Sie bildet die Grundlage für die Ausführung von Prozessstrukturen, d.h. die Koordination miteinander verknüpfter OLCs (siehe Abschnitt 4.5.2).

4.3.1 Laufzeitmarkierungen

Wir definieren die operationale Semantik von OLCs auf Basis der Laufzeitmarkierungen seiner Elemente sowie konkreten Regeln für den Wechsel dieser Markierungen. Die Markierung der Elemente hat beispielsweise in nicht-deterministischen OLCs den Vorteil, dass der gewählte Pfad im Rahmen der Ausführung jederzeit schnell ersichtlich ist. Darüber hinaus sind die Markierungen nicht nur für die Definition geeigneter Ausführungsregeln hilfreich, sondern werden auch für die Definition von Konsistenzkriterien im Kontext der dynamischen Adaption von Prozessstrukturen benötigt (siehe Kapitel 6). Hierzu ist es wichtig, die Markierung nicht nur für das aktuell aktive Element des Prozesses anzuzeigen, sondern diese auch zu erhalten. So kann später festgestellt werden, ob das Element bereits durchlaufen oder etwa abgewählt wurde. Zunächst müssen wir uns allerdings mit der Frage befassen, welche Elemente eines OLC für dessen Ausführung mit Markierungen versehen werden müssen und welche Markierungen benötigt werden:

- *Zustände* erfüllen für die Ausführung eines OLC zwei Funktionen. Erstens repräsentieren sie durch ihre Aktivierung die Beendigung bzw. das Ergebnis des vorangegangenen Prozesses. Zweitens ist ihre Aktivierung die Voraussetzung für die Ausführung des nachfolgenden Prozesses (d.h. des mit den ausgehenden Transitionen verknüpften Prozesses). Die Markierung von Zuständen (z.B. als *aktiviert*) ist damit für die Koordination von Prozessen bzw. die Ausführung von OLCs essentiell.
- *Transitionen* sind mit Prozessen sowie einer Transitionsbedingung verknüpft. Auf Basis des Prozessergebnisses steuern sie den Fluss durch den OLC. In nicht-deterministischen OLCs führt nach Ausführung des Prozesses eines Verzweigungsabschnitts die Transitionsbedingung zur Auswahl eines Zielzustands. Die Markierung von Transitionen erlaubt hier die

Anzeige nicht gewählter Pfade. Dies erhöht nicht nur die Übersichtlichkeit und Nachvollziehbarkeit im Modell, sondern vereinfacht auch die Definition von Regeln für die OLC-Ausführung.

Für das COREPRO-Rahmenwerk definieren wir eine explizite operationale Semantik mit Laufzeitmarkierungen für Zustände und Transitionen. Zu diesem Zweck ergänzen wir das in Abschnitt 4.2 vorgestellte Metamodell für OLCs um *Markierungsfunktionen* für Zustände (vgl. Definition 4.6) und Transitionen (vgl. Definition 4.7). Aus diesen beiden Markierungen resultiert dann die Gesamtmarkierung eines OLC (vgl. Definition 4.8).

Definition 4.6 (Markierungsfunktion für Zustände)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Wir bezeichnen eine Abbildung* $SM_{olc} : S \mapsto StateMarkings$, *die jedem Zustand* $s \in S$ *eine Markierung* $SM_{olc}(s) \in StateMarkings$ *zuordnet, als Zustandsmarkierung von olc. Die relevanten Zustandsmarkierungen sind definiert durch*

$$StateMarkings := \{\texttt{NOTACTIVATED}, \texttt{ACTIVATED}, \texttt{DONE}, \texttt{SKIPPED}\}$$

Jeder Zustand besitzt genau eine dieser vier Markierungen (vgl. Tabelle 4.1). Im Folgenden wird auf die Indizierung von SM *verzichtet, wenn der referenzierte OLC aus dem aktuellen Kontext hervorgeht.*

Zustandsmark.	Beschreibung
NOTACTIVATED	Der Zustand wartet auf seine Aktivierung (Initialmarkierung).
ACTIVATED	Der Zustand ist aktiviert. Es gibt stets nur einen aktivierten Zustand innerhalb eines OLC.[11]
DONE	Der Zustand wurde durchlaufen und ein Folgezustand wurde aktiviert.
SKIPPED	Der Zustand befindet sich in einem abgewählten Pfad.

Tabelle 4.1: Markierungen von Zuständen und ihre Bedeutung

Definition 4.7 (Markierungsfunktion für (interne) Transitionen)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Wir bezeichnen eine Abbildung* $TM_{olc} : T \mapsto TransitionMarkings$, *die jeder Transition* $t \in T$ *eine Markierung* $TM_{olc}(t) \in TransitionMarkings$ *zuordnet, als* Transitionsmarkierung *von olc. Die relevanten Transitionsmarkierungen sind definiert durch*

$$TransitionMarkings := \{\texttt{WAITING}, \texttt{PROCESSING}, \texttt{FIRED}, \texttt{DISABLED}\}$$

Jede Transition besitzt somit genau eine dieser vier Markierungen (vgl. Tabelle 4.2). Im Folgenden wird auf die Indizierung von TM *verzichtet, wenn der referenzierte OLC aus dem aktuellen Kontext hervorgeht.*

[11] Siehe hierzu die in Abschnitt 4.1.3 geführte Diskussion. Anzumerken ist an dieser Stelle nochmals, dass in einer Prozessstruktur, die in der Regel mehrere OLCs integriert, sehr wohl mehrere Zustände parallel aktiviert sein können (in verschiedenen OLCs).

Transitionsmark.	Beschreibung
WAITING	Die Transition wartet auf die Aktivierung ihres Quellzustands (Initialmarkierung).
PROCESSING	Der Quellzustand der Transition ist aktiviert; der mit ihr verknüpfte Prozess ist instanziiert und gestartet.
FIRED	Der mit der Transition verknüpfte Prozess wurde beendet und das Prozessergebnis entspricht der Transitionsbedingung (vgl. Definition 4.1).
DISABLED	Der Prozess wurde beendet und das Ergebnis entspricht nicht der Transitionsbedingung (vgl. Definition 4.1); oder die Transition befindet sich in einem abgewählten Pfad.

Tabelle 4.2: Markierungen interner Transitionen und ihre Bedeutung

Die Tabellen 4.1 und 4.2 beschreiben die Markierungen für Zustände und Transitionen. Für sie gibt es jeweils eine *Initialmarkierung*: NOTACTIVATED bzw. WAITING. Mit diesen Markierungen werden die Zustände und Transitionen vor der Ausführung des OLC initialisiert (siehe Abschnitt 4.3.3). Das Beispiel aus Abbildung 4.7 zeigt die von COREPRO verwendeten Symbole für Zustands- und Transitionsmarkierungen. Zur besseren Übersichtlichkeit wird in nachfolgenden Abbildungen auf die Darstellung der Initialmarkierungen für Zustände und Transitionen verzichtet.

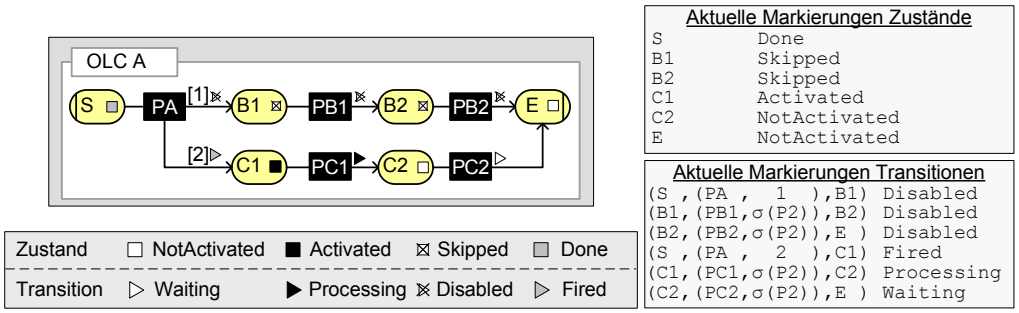


Abbildung 4.7: OLC mit Darstellung der Zustands- und Transitionsmarkierungen

Die *Markierung eines OLC* setzt sich zusammen aus den Markierungsfunktionen für Zustände und Transitionen (vgl. Definitionen 4.6 und 4.7). Die aktuellen Zustände der einzelnen Prozesse werden hier nicht mit abgebildet, da sie bereits implizit über die Markierungen der Transitionen vorliegen. Diese Beziehung wird in Abschnitt 4.3.2 erläutert.

Definition 4.8 (Markierung eines OLC)
Sei $olc \in \mathcal{OLC}$ ein Object Life Cycle. Die Markierung M von olc setzt sich aus den Zustandsmarkierungen SM_{olc} und den Transitionsmarkierungen TM_{olc} zusammen:

$$M_{olc} := (SM_{olc}, TM_{olc}).$$

$\mathcal{M}$ bezeichne die Menge der Markierungen aller Object Life Cycles aus $\mathcal{OLC}$. Im Folgenden wird die Indizierung von M, SM und TM verzichtet, wenn der referenzierte OLC aus dem aktuellen Kontext hervorgeht.

Abbildung 4.8 zeigt einen deterministischen OLC vor und nach Beginn seiner Ausführung (vgl. Abbildung 4.8a bzw. 4.8b). Ist ein Zustand oder eine Transition nicht mit einem Symbol annotiert, befindet sich das betreffende Element in seiner *Initialmarkierung* (NOTACTIVATED bzw. WAITING). Die Änderung der Markierungen von Zuständen und Transitionen erfolgt auf Grundlage wohldefinierter Regeln, die in den Abschnitten 4.3.4, 4.3.5 und 4.3.6 erörtert werden.

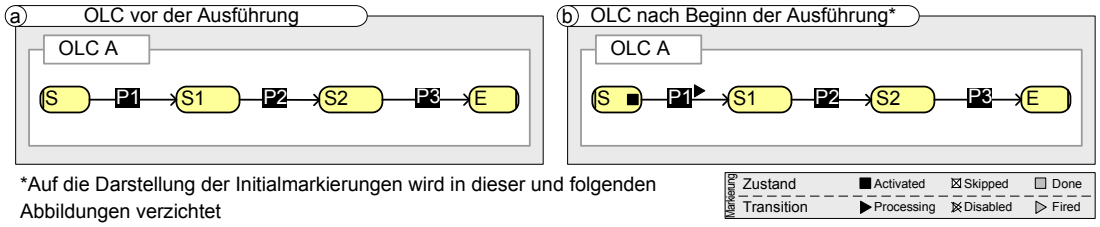


Abbildung 4.8: OLC vor und während der Ausführung

4.3.2 Prozesszustände

Zur Laufzeit koordinieren OLCs den Start der mit den Transitionen verknüpften Prozesse. Den aktuellen Status eines Prozesses während der Ausführung nennen wir *Prozesszustand*[12]. Er zeigt beispielsweise an, ob sich ein Prozess aktuell in Ausführung befindet oder bereits beendet wurde. Wir definieren im Folgenden diejenigen Prozesszustände, die für die Koordination der Prozesse innerhalb OLCs (bzw. Prozessstrukturen) relevant sind (vgl. Definition 4.9 und Tabelle 4.3).[13]

Definition 4.9 (Prozesszustände)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Wir bezeichnen eine Abbildung* $PS : P \mapsto ProcessStates$, *die jedem Prozess* $p \in P$ *einen Zustand* $PS(p) \in ProcessStates$ *zuordnet, als Prozesszustand. Die Menge der Prozesszustände ist wie folgt definiert:*

$$ProcessStates := \{\text{INITIAL}, \text{RUNNING}, \text{COMPLETED}, \text{SKIPPED}\}$$

Jeder Prozess befindet sich somit stets genau in einem dieser Zustände (vgl. Tabelle 4.3).

Für die Realisierung einer geeigneten Koordination müssen Prozesse des Weiteren einerseits durch eine entsprechende Aktion gestartet, andererseits in bestimmten Situationen aber auch abgebrochen werden können:

- START: Instanziierung und Start des Prozesses (d.h. Überführung des Prozesszustands von INITIAL nach RUNNING). In unserem Ansatz wird ein Prozess in einem Schritt instanziiert und gestartet. Dies erfolgt, sobald die Transition, mit welcher der Prozess verknüpft ist, als PROCESSING markiert wird. Durch entsprechende Parameterübergabe kann dem Prozess in der Realisierung die zur Ausführung nötige Kontextinformation mitgegeben werden (z.B. OLC-Name und aktivierter Zustand).

[12] Nicht zu verwechseln mit OLC-Zuständen.
[13] Die Ausführung von Prozessen, deren Metamodelle (z.B. [RD98, LR00]) die Prozesszustände feingranularer definieren, wird in Kapitel 8 gezeigt.

Prozesszustand	Beschreibung
INITIAL	Die Transition, mit welcher der Prozess verknüpft ist, ist als WAITING markiert. Der Prozess ist noch nicht instanziiert (initiale Phase).
RUNNING	Die Transition, mit welcher der Prozess verknüpft ist, ist als PROCESSING markiert. Der Prozess wird mittels der Aktion START instanziiert und gestartet; er befindet sich damit in Ausführung.
COMPLETED	Der Prozess ist beendet und liefert ein Ergebnis $pv \in \sigma(p)$ (vgl. Definition 4.1). Die Transition, mit welcher der Prozess verknüpft ist, wird als FIRED oder DISABLED markiert (die zugehörigen Regeln werden in den folgenden Abschnitten vorgestellt).
SKIPPED	Die Transition, mit welcher der Prozess verknüpft ist, wurde als DISABLED markiert.

Tabelle 4.3: Prozesszustände und ihre Bedeutung

- CANCEL: Abbruch eines sich in Ausführung befindlichen Prozesses (Überführung des Prozesszustands von RUNNING nach SKIPPED). In COREPRO wird ein laufender Prozess mit Zustand RUNNING abgebrochen, sobald die Transition, mit welcher er verknüpft ist, als DISABLED markiert wird. Dieses Verhalten ist für die Synchronisation verschiedener OLCs (siehe Abschnitt 4.6), die dynamische Adaption von Prozessstrukturen (siehe Kapitel 6) sowie die Ausnahmebehandlung (siehe Kapitel 7) relevant.

4.3.3 Initialisierung eines OLC

Die Regeln der operationalen Semantik nutzen die im vorherigen Abschnitt vorgestellten Markierungen. Um eine definierte Anfangsmarkierung für einen OLC festzulegen, muss allen Zuständen und Transitionen die jeweilige *Initialmarkierung* zugewiesen werden. Zu diesem Zweck markiert die Initialisierungsregel IR1 alle Zustände als NOTACTIVATED und alle Transitionen als WAITING. Ebenso werden alle mit Transitionen verknüpften Prozesse in den Zustand INITIAL versetzt. Prozesszustände werden über die Markierung der Transitionen abgebildet und fließen daher im Folgenden nicht direkt in die Markierung des OLC ein (vgl. Definition 4.8). Wir gehen im Folgenden davon aus, dass OLCs immer durch Regel IR1 initialisiert werden.

Initialisierungsregel IR1 (Initialisierung eines OLC)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Die initiale Markierung* $M = (SM, TM)$ *von olc ist festgelegt durch:*

(a) $\forall\, s \in S : SM(s) :=$ NOTACTIVATED

(b) $\forall\, t \in T : TM(t) :=$ WAITING

Ferner sind die mit Transitionen verknüpften Prozesse $p \in P$ *als* $PS(p) :=$ INITIAL *initialisiert. Besitzen alle Elemente ihre Initialmarkierung, wird der OLC ausführbar. Er wird gestartet, indem die Markierung seines Startzustands* s_{start} *von* NOTACTIVATED *auf* ACTIVATED *gesetzt wird (vgl. Abbildung 4.8b). Die Markierung der aus* s_{start} *ausgehenden Transitionen wird automatisch durch die im folgenden Abschnitt vorgestellten Ausführungsregeln angepasst.*

4.3.4 Laufzeitmarkierungen deterministischer OLCs

In den folgenden Abschnitten definieren wir *Ausführungs-* und *Markierungsregeln* und damit die operationale Semantik von OLCs (vgl. Abbildung 4.9). Ausführungsregeln beschreiben die Änderung der Markierung von Transitionen, die dann zur Ausführung (bzw. Abwahl) von Prozessen führen. Markierungsregeln wiederum beschreiben die Neubewertung von Markierungen nach Beendigung (oder Abwahl) eines Prozesses. Ausführungs- und Markierungsregeln werden mit einem Kontext definiert, innerhalb dessen sie angewendet werden. Als Kontext einer Ausführungsregel kann beispielsweise der Markierungswechsel eines Zustands von NOTACTIVATED in ACTIVATED dienen. Durch Anwendung einer Regel und Anpassung der Markierungen von Zuständen bzw. Transitionen kann wiederum der Kontext einer (anderen) Regel erfüllt sein. Ausführungsregel AR1 und Markierungsregel MR1 beschreiben die operationale Semantik von Sequenzabschnitten. Das Beispiel 4.1 zeigt deren Anwendung für einen deterministischen OLC.

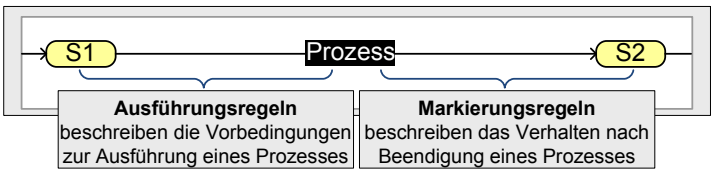


Abbildung 4.9: Anwendungsmechanismus für Ausführungs- und Markierungsregeln

Beispiel 4.1 (Ausführungsszenario in einem deterministischen OLC)
Abbildung 4.10 zeigt den Ausschnitt eines OLC. In Schritt 1 ist die Ausgangssituation dargestellt, in der Zustand A von NOTACTIVATED auf ACTIVATED markiert wird. Sowohl Zustand B als auch die Transition von A nach B befinden sich in ihrer Initialmarkierung. Durch Änderung des Kontextes, d.h. durch die Aktivierung von A, soll die Ausführung von Prozess P beginnen (vgl. Abschnitt 4.2). Dieses Verhalten beschreibt Ausführungsregel AR1: Durch Wechsel der Markierung des Zustands A von NOTACTIVATED auf ACTIVATED wird die aus A ausgehende Transition als PROCESSING markiert (Abbildung 4.10, Schritt 2). Damit wird der Prozess P ausführbar und entsprechend in den Prozesszustand RUNNING überführt (vgl. Abschnitt 4.3.2). Sobald P beendet ist, also ein neuer Kontext vorliegt, soll die verknüpfte Transition von Zustand A nach B feuern und ihr Zielzustand B aktiviert werden. Markierungsregel MR1 definiert genau dieses Verhalten: Sobald P den Prozesszustand von RUNNING auf COMPLETED ändert, werden die Transition (A, (P, σ(P)), B) als FIRED und ihr Zielzustand B als ACTIVATED markiert. Der Quellzustand A wird als DONE markiert. B ist damit der einzige Zustand des OLC mit Markierung ACTIVATED (Abbildung 4.10, Schritt 3).

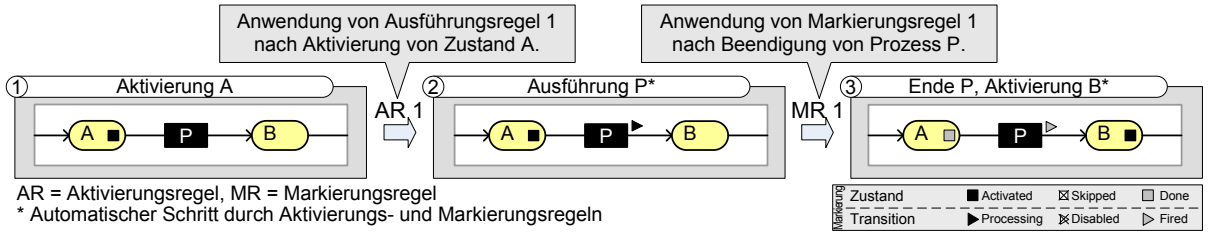


Abbildung 4.10: Anwendung der Regeln für deterministische OLCs

Ausführungsregel AR1 (Sequenzabschnitt)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren seien $s \in seq(olc)$ ein Sequenzzustand (vgl. Definition 4.3) und $t^* = (s, (p, v), targ) \in T$ die aus s ausgehende interne Transition. Wenn sich die Zustandsmarkierung $SM(s)$ von NotActivated *in* Activated *ändert, wird der Prozess p ausführbar. Es ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM'(t) = \begin{cases} \textsc{Processing} & \text{falls } t = t^* \\ TM(t) & \text{sonst} \end{cases}$$

$$SM' \equiv SM$$

Durch Markierung der ausgehenden Transition als Processing *wird automatisch der mit ihr verknüpfte Prozess in den Prozesszustand* Running *überführt (vgl. Abschnitt 4.3.2).*

Markierungsregel MR1 (Sequenzabschnitt)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren seien $t^* = (src^*, (p, v), targ^*) \in T$ eine interne Transition und $src^* \in seq(olc)$ ein Sequenzzustand. Wenn sich für den Prozess p der Prozesszustand $PS(p)$ von Running *in* Completed *ändert, ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM'(t) = \begin{cases} \textsc{Fired} & \text{falls } t = t^* \\ TM(t) & \text{sonst} \end{cases}$$

$$SM'(s) = \begin{cases} \textsc{Activated} & \text{falls } s = targ^* \\ \textsc{Done} & \text{falls } s = src^* \\ SM(s) & \text{sonst} \end{cases}$$

Generell gilt, dass bei Aktivierung des Endzustands eines deterministischen OLC alle Zustände, mit Ausnahme des Endzustands, als Done und alle Transitionen als Fired markiert sind (vgl. Satz 4.1). Dadurch ist sichergestellt, dass sich kein Prozess mehr im Prozesszustand Running befindet, wenn der Endzustand des OLC erreicht wird. Diese Eigenschaft ist grundlegend für die korrekte Terminierung eines OLC (vgl. Anforderung 4.4). Die Markierungen Skipped für Zustände und Disabled für Transitionen treten in deterministischen OLCs nicht auf.

Satz 4.1 (Terminierung deterministischer OLCs)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein deterministischer Object Life Cycle (vgl. Definition 4.4) mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Wird der Endzustand des OLC entsprechend der definierten operationalen Semantik als $SM(s_{end}) =$ Activated *markiert, gilt:*

$$\forall\, s \in S, s \neq s_{end} : SM(s) = \textsc{Done} \text{ und } \forall\, t \in T : TM(t) = \textsc{Fired}.$$

Ein Beweis von Satz 4.1 findet sich in Anhang A.

4.3.5 Laufzeitmarkierungen nicht-deterministischer azyklischer OLCs

Nicht-deterministische OLCs enthalten neben Sequenzabschnitten auch Verzweigungsabschnitte, d.h., es gibt Zustände mit mehreren ausgehenden bzw. eingehenden (internen) Transitionen. Daher müssen die Regeln für Sequenzabschnitte aus Abschnitt 4.3.4 für die Markierung von Verzweigungsabschnitten geeignet erweitert werden. Gehen aus einem Zustand mehrere Transitionen aus, müssen wir zum einen sicherstellen, dass der mit ihnen verknüpfte Prozess nur einmal instanziiert und gestartet wird (vgl. Definition 4.1 bzw. Ausführungsregel AR2). Zum anderen müssen nach Beendigung des Prozesses diejenigen Transitionen, deren Transitionsbedingung nicht erfüllt ist, abgewählt werden (*Deadpath-Eliminierung*; vgl. Markierungsregel MR2). Deren Zielzustände werden ebenfalls abgewählt, wenn sie im weiteren Verlauf durch den gewählten Pfad nicht mehr erreicht werden können. Um diese Erreichbarkeit festzustellen, aber auch um die Übersichtlichkeit im Modell zu erhöhen und insbesondere Konsistenzprüfungen im Rahmen der dynamischen Adaption zu ermöglichen (siehe Kapitel 6), führen wir eine vollständige Deadpath-Eliminierung durch. Das heißt, Transitionen und Zustände, die auf abgewählte Zielzustände einer Verzweigungsabschnitts folgen, werden als `DISABLED` bzw. `SKIPPED` markiert (vgl. Anforderung 4.1). Dieses Verhalten wird mit Ausführungsregel AR3 und Markierungsregel MR3 realisiert. Die Markierungsregel MR3 stellen sicher, dass Zustände nur dann abgewählt werden, wenn sie nicht mehr durch Transitionen des Hauptpfads erreicht werden können (d.h. wenn alle eingehenden Transitionen als `DISABLED` markiert sind).[14] Die Anwendung der Regeln wird anhand des Beispiels 4.2 verdeutlicht.

Beispiel 4.2 (Ausführungsszenario in einem nicht-det. azyklischen OLC)
Abbildung 4.11 zeigt einen Ausschnitt eines nicht-deterministischen OLC. In Schritt 1 wird (der zuvor nicht aktivierte) Zustand `A` als `ACTIVATED` markiert. Die ausgehenden Transitionen sind mit demselben Prozess `PA` (bzw. derselben Prozessinstanz) verknüpft. Ausführungsregel AR2 stellt sicher, dass im Zuge der Aktivierung von `A` beide ausgehenden Transitionen als `PROCESSING` markiert werden, `PA` aber nur einmal instanziiert wird (Schritt 2). Die aus `A` ausgehenden Transitionen unterscheiden sich durch ihre Transitionsbedingung (0 bzw. 1), also in dem von `PA` erwarteten Ergebnis.

In Schritt 3 wechselt `PA` den Prozesszustand von `RUNNING` nach `COMPLETED` und liefert als Ergebnis $pv = 0$. Markierungsregel MR2 stellt nun sicher, dass nur diejenige Transition feuert, deren Transitionsbedingung durch das von `PA` gelieferte Ergebnis pv erfüllt ist. Dementsprechend markiert sie die Transition mit der Transitionsbedingung $v = 0$ als `FIRED`, woraufhin der Zustand `C1` als `ACTIVATED` markiert wird. Des Weiteren werden durch Regel MR2 alle anderen aus `A` ausgehenden Transitionen (deren Transitionsbedingung zwangsläufig nicht erfüllt ist) als `DISABLED` markiert (vgl. Abbildung 4.11, Schritt 3). Zustand `A` wiederum wird als `DONE` markiert. Ausführungsregel AR1 startet nun infolge der Aktivierung von `C1` den mit den von `C1` ausgehenden Transitionen verknüpften Prozess `PC1`. Ferner ist durch die Abwahl der Transition (`A`, (`PA`, `1`), `B1`) der Kontext von Markierungsregel MR3 erfüllt. Sie markiert `B1` als `SKIPPED`. Um auch den auf `B1` folgende Pfad abzuwählen, kommen Ausführungsregel AR3 und Markierungsregel MR3 erneut zur Anwendung. Ausführungsregel AR3 markiert die von `B1` ausgehende Transition (`B1`, (`PB1`, σ(PB1)), `B2`) als `DISABLED` (d.h. ihr Prozess wird nicht gestartet). Daraufhin wird Markierungsregel MR3 angewendet und der Zielzustand `B2` der Transition als `SKIPPED` markiert. Durch Veränderung des

[14] Abschnitt 4.3.6 diskutiert die Unterschiede zwischen der Deadpath-Analyse azyklischer und zyklischer OLCs.

Kontextes wird Ausführungsregel AR3 erneut angewendet und die aus B2 ausgehende Transition (verknüpft mit Prozess PB2) als DISABLED markiert. Zustand D hat eine eingehende Transition, die nicht als DISABLED markiert ist. Deshalb kann D während der Ausführung des OLC noch erreicht werden. Die Markierung des abgewählten Pfads wird demzufolge beendet und Markierungsregel MR3 verändert die Markierung von D nicht.

Sobald der Prozess PC1 den Prozesszustand COMPLETED erreicht (Schritt 4 in Abbildung 4.11), feuert die mit PC1 verknüpfte Transition. Daraufhin wird Zustand D durch Markierungsregel MR1 als ACTIVATED markiert, wohingegen C1 die Markierung DONE erhält.

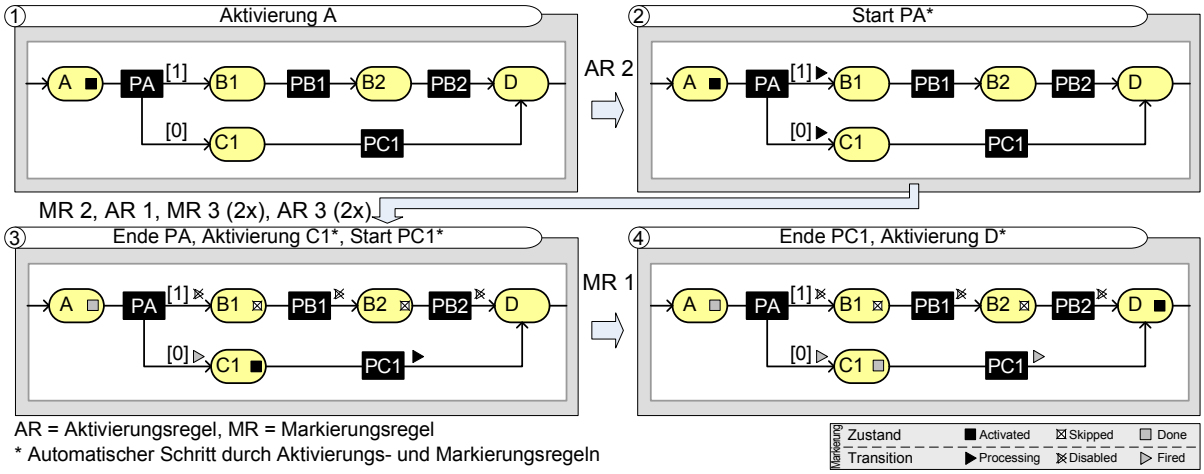


Abbildung 4.11: Anwendung der Regeln für nicht-deterministische azyklische OLCs

Ausführungsregel AR2 (Verzweigungsabschnitt)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$*. Des Weiteren sei* $s^* \in splits(olc)$ *der Zustand eines Verzweigungsabschnitts (vgl. Definition 4.3). Wenn sich die Markierung* $SM(s^*)$ *von* NOTACTIVATED *in* ACTIVATED *ändert, wird der mit den aus* s^* *ausgehenden Transitionen verknüpfte Prozess ausführbar.*[a] *Es ergibt sich für olc als Folgemarkierung* $M' = (SM', TM')$ *mit:*

$$TM'(t) = \begin{cases} \text{PROCESSING} & \forall\, t \in outTrans_{int}(s^*) \\ TM(t) & sonst \end{cases}$$

$$SM' \equiv SM$$

Durch Markierung der ausgehenden Transitionen als PROCESSING *wird automatisch der mit ihnen verknüpfte Prozess in den Prozesszustand* RUNNING *überführt (vgl. Abschnitt 4.3.2).*

[a] Alle aus einem Zustand ausgehenden (internen) Transitionen sind mit demselben Prozess (bzw. Prozessinstanz) verknüpft (vgl. Definition 4.1).

Markierungsregel MR2 (Verzweigungsabschnitt)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren seien $s^* \in splits(olc)$ der Zustand eines Verzweigungsabschnitts und $T^* = outTrans_{int}(s^*)$ die Menge aller aus s^* ausgehenden Transitionen. Dabei seien alle Transitionen $t = (s^*, (p, v), t) \in T^*$ mit demselben Prozess $p \in P$ verknüpft (vgl. Definition 4.1). Wenn der Prozesszustand $PS(p)$ von RUNNING *nach* COMPLETED *wechselt und p das Ergebnis pv liefert, gilt $\exists\, \hat{t} = (s^*, (p, v), \hat{targ}) \in T^* : v = pv$. Es ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM'(t) = \begin{cases} \text{FIRED} & \text{falls } t = \hat{t} \\ \text{DISABLED} & \text{falls } t \in T^* \setminus \{\hat{t}\} \\ TM(t) & \text{sonst} \end{cases}$$

$$SM'(s) = \begin{cases} \text{ACTIVATED} & \text{falls } s = \hat{targ} \\ \text{DONE} & \text{falls } s = s^* \\ SM(s) & \text{sonst}^a \end{cases}$$

[a] Die Markierung der Zielzustände abgewählter Transitionen der Verzweigung geschieht durch Markierungsregel MR3 bzw. MR5.

Ausführungsregel AR3 (Abwahl toter Pfade; Deadpath-Eliminierung)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren sei $s^* \in S$ ein Zustand mit einer oder mehreren ausgehenden Transitionen der Menge $T^* = outTrans_{int}(s^*)$. Wenn sich die Markierung des Zustands s^* von NOTACTIVATED *in* SKIPPED *ändert, ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM'(t) = \begin{cases} \text{DISABLED} & \text{falls } t \in T^* \wedge TM(t) = \text{WAITING} \\ TM(t) & \text{sonst} \end{cases}$$

$$SM' \equiv SM$$

Markierungsregel MR3 (Abwahl toter Pfade; Deadpath-Eliminierung)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren sei $s^* \in S$ ein Zustand mit einer oder mehreren eingehenden Transitionen der Menge $T^* = inTrans_{int}(s^*)$ und es gilt: $\forall t \in T^* : t \in path_{int}(s_{start}, s_{end})$ (d.h. die eingehenden Transitionen liegen auf dem Hauptpfad und sind damit nicht Teil von Rücksprungpfaden; vgl. Abschnitt 4.2.2 bzw. Abbildung 4.4). Wenn für $t^* = (src, (p, v), s^*) \in T^*$ die Markierung $TM(t^*)$ von WAITING *in* DISABLED *wechselt, ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM' \equiv TM$$

$$SM'(s) = \begin{cases} \text{SKIPPED} & \text{falls } s = s^* \wedge\ \forall\, t \in T^* : TM(t) = \text{DISABLED} \\ SM(s) & \text{sonst} \end{cases}$$

Nach Aktivierung des Endzustands eines nicht-deterministischen OLC und Abarbeitung der anwendbaren Ausführungs- und Markierungsregeln sind alle Zustände mit Ausnahme des Endzustands als DONE oder SKIPPED sowie alle Transitionen als FIRED oder DISABLED markiert (vgl. Satz 4.2). Damit befindet sich kein Prozess mehr im Prozesszustand RUNNING, wenn der Endzustand des OLC erreicht wird. Hinsichtlich der Laufzeitmarkierungen müssen wir beachten, dass die Regeln für die Deadpath-Eliminierung dynamisch angewendet werden. Gehen mehr als eine Transition in den Endzustand ein, besteht die Möglichkeit, dass im Moment der Aktivierung des Endzustands die Deadpath-Eliminierung der abgewählten Pfade noch nicht abgeschlossen ist (vgl. Abbildung 4.12a und 4.12b). Wir stellen durch die Regeln jedoch sicher, dass zu diesem Zeitpunkt kein Prozess mehr ausgeführt wird und die Abwahl in jedem Fall stattfindet. Damit hat dieser Umstand keine praktische Relevanz und beeinflusst die dynamischen Eigenschaften eines OLC nicht.

Satz 4.2 (Terminierung nicht-deterministischer OLCs)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$*. Wird nun der Endzustand des OLC entsprechend der definierten operationalen Semantik als* $SM(s_{end}) =$ ACTIVATED *markiert, gilt* $\forall\, s \in S, s \neq s_{end} : SM(s) \in \{\text{DONE}, \text{SKIPPED}\}$ *und* $\forall\, t \in T : TM(t) \in \{\text{FIRED}, \text{DISABLED}\}$.

Ein Beweis von Satz 4.2 findet sich in Anhang A.

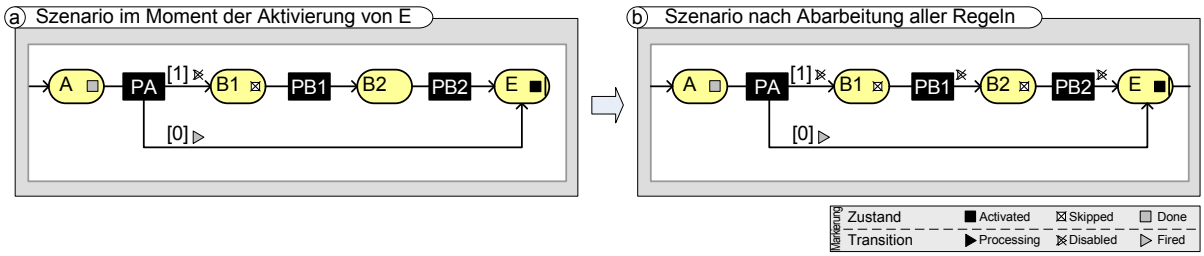


Abbildung 4.12: Mögliche Markierung eines OLC im Moment der Aktivierung des Endzustands

4.3.6 Laufzeitmarkierungen nicht-deterministischer zyklischer OLCs

Innerhalb eines OLC können Verzweigungen auch zur Modellierung von Schleifen bzw. Rücksprüngen genutzt werden (vgl. Anforderung 4.1). Zur Laufzeit führt ein Rücksprung zur erneuten Markierung bereits durchlaufener Zustände und Transitionen. Unser Ziel bei der Definition der operationalen Semantik für zyklische OLCs ist es, die Schleifenrücksprünge mit dem in den Abschnitten 4.3.4 und 4.3.5 bereits eingeführten Markierungsmechanismus abzubilden. Dazu ist es notwendig, nach einem erfolgten Rücksprung die Markierung des OLC wieder so herzustellen, dass die bekannten Regeln für das erneute Durchlaufen der Schleife anwendbar sind. Die Markierungen der Zustände und Transitionen, die erneut zu durchlaufen sind, müssen entsprechend zurückgesetzt werden. Damit ist es möglich, den Ablauf für Rücksprünge ohne Einführung zusätzlicher Mechanismen (z.B. Schleifenzähler) zu realisieren.

Während *Verzweigungsabschnitte mit Rücksprung* (vgl. Definition 4.3) bei der Modellierung im Prinzip wie „einfache“ Verzweigungen ohne Rücksprung behandelt werden können, bedürfen sie

während der Ausführung einer angepassten operationalen Semantik. Wie in Abschnitt 4.2.2 beschrieben, können die zu Rücksprüngen führenden Verzweigungen bereits zur Modellierzeit erkannt werden. Das heißt, sie lassen sich mit den Hilfsfunktionen aus Definition 4.2 bzw. den OLC-Abschnitten aus Definition 4.3 identifizieren. Nach Beendigung des Prozesses eines Verzweigungsabschnitts ohne Rücksprung werden diejenigen Transitionen, deren Transitionsbedingung nicht erfüllt ist, zusammen mit nachfolgenden Pfaden abgewählt (vgl. Abschnitt 4.3.5). Dieses Verhalten würde allerdings bei einem Verzweigungsabschnitt mit Rücksprung dazu führen, dass bei Auswahl des Rücksprungpfads eine Abwahl aller nachfolgenden Pfade bis zum Endzustand des OLC durchgeführt wird. In Abbildung 4.4 (siehe Seite 58) würde beispielsweise die Beendigung von Prozess `P3` mit Ergebnis 0 bei diesem naiven Ansatz dazu führen, dass der Endzustand `E` als `SKIPPED` markiert wird. Dies entspricht nicht der gewünschten Semantik, denn die Markierung von `E` zeigt dem Benutzer den Fortschritt im OLC nicht *korrekt* an.

Um eine intuitive Markierung zyklischer OLCs zu erhalten, wählt Markierungsregel MR2 (siehe Seite 72) bei einem Schleifenrücksprung zwar die nicht gewählten Transitionen des Verzweigungsabschnitts ab, nicht jedoch deren Zielzustände. Die Anwendung von Markierungsregel MR3 verhindert in diesem Kontext, dass eine „falsche“ Markierung des abgewählten Pfades bis zum Endzustand geschieht. Des Weiteren stellen Markierungsregel MR4 und Ausführungsregel AR4 sicher, dass im Fall eines Schleifenrücksprungs die Markierungen der Zustände und Transitionen innerhalb der Schleife wieder in ihre Initialmarkierung zurückgesetzt werden. Die Abbruchbedingung der Schleifen-Reinitialisierung ergibt sich aus den Markierungen der Zustände. So werden durch Markierungsregel MR4 nur Zustände zurückgesetzt, die derzeit nicht als `ACTIVATED` oder `NOTACTIVATED` markiert sind. Der Kontext für die Anwendung der Regeln MR4 und AR4 ergibt sich jeweils aus der Reinitialisierung der Markierung eines Zustands bzw. einer Transition. Damit befinden sich diejenigen Zustände und Transitionen wieder in ihrer Initialmarkierung, die im aktuellen Schleifendurchlauf erreicht werden können. Ferner können in der dann folgenden Schleifeniteration, also dem erneuten Durchlauf der zuvor aktivierten Zustände, wieder die bekannten Regeln aus den Abschnitten 4.3.4 und 4.3.5 angewendet werden. Die Laufzeitmarkierung von Zuständen und Transitionen innerhalb einer Schleife stellt letztendlich immer nur den aktuellen (bzw. letzten) Schleifendurchlauf dar.[15]

Die Modellierung von Schleifen beeinflusst auch die Deadpath-Eliminierung, d.h. die operationale Semantik von Verzweigungsabschnitten bzw. die Regeln zur Markierung der abgewählten Pfade müssen Schleifen berücksichtigen. Für nicht-deterministische azyklische OLCs endet die Deadpath-Eliminierung, sobald ein Zustand erreicht wird, der mindestens eine eingehende Transition besitzt, die nicht als `DISABLED` markiert ist (vgl. Abschnitt 4.3.5). Während dies in Abbildung 4.13a innerhalb einer Schleife funktioniert, zeigt Abbildung 4.13b einen zyklischen OLC, in dem diese Regel zu einem Abbruch der Deadpath-Eliminierung bei Zustand `S2` führen würde. Markierungsregel MR3 wurde deshalb mit einer weiteren Bedingung versehen, die bei der Deadpath-Eliminierung eingehende Transitionen nicht berücksichtigt, wenn sie Teil eines Rücksprungpfads sind (formal: $t^* \notin path_{int}(s_{start}, s_{end})$).

Rückwärts gerichtete Transitionen können auch genutzt werden, um *Schlingen* zu modellieren. Eine Schlinge entspricht einer Transition mit identischem Quell- und Zielzustand. Sie führt daher zu einem Zyklus (vgl. Transition mit Prozess `P3` in Abbildung 4.13c). Feuert diese Transition bei

[15] Das Protokoll der Prozessausführung (siehe Abschnitt 4.3.7) bleibt erhalten. Der Verlauf der Ausführung eines OLC ist damit rekonstruierbar.

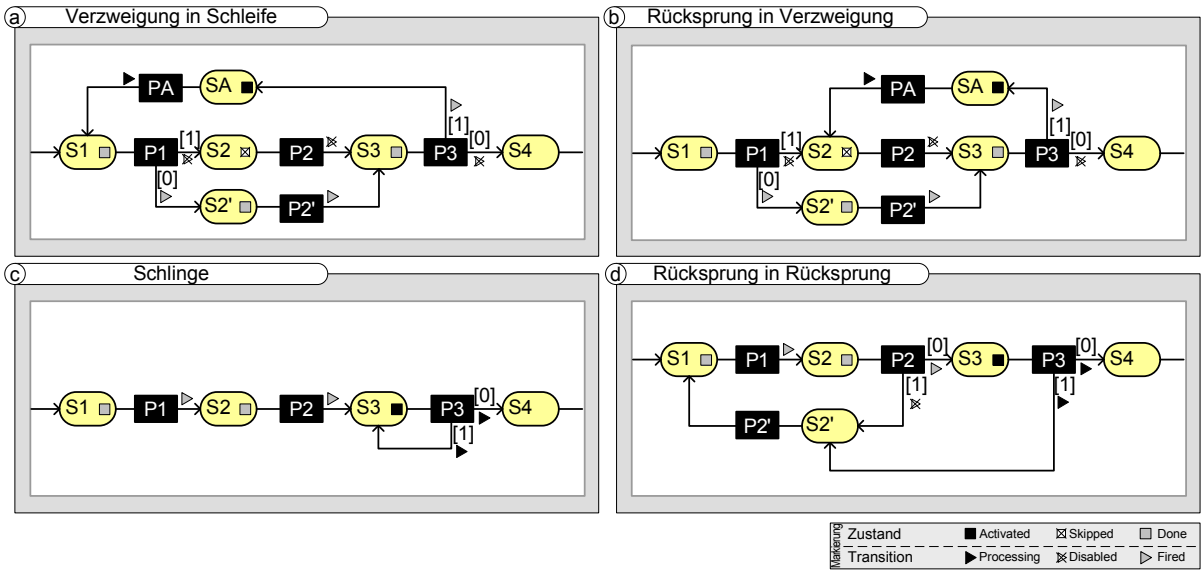


Abbildung 4.13: Rücksprung-Szenarien innerhalb eines OLC

Erfüllung ihrer Verzweigungsbedingung, wird ihr Zielzustand – und damit ihr bereits aktivierter Quellzustand – als `ACTIVATED` markiert. Ausführungsregel AR4 kommt dann wie bei einer „normalen“ Schleife zur Anwendung.

Rücksprünge können auch als Pfade, d.h. mit mehreren Zuständen und Transitionen, modelliert werden (vgl. Abbildung 4.13d). Dann muss für die Markierung abgewählter Rücksprungpfade eine andere Abbruchbedingung als für „normale“ Verzeigungen gelten. Abbildung 4.13d zeigt einen zyklischen OLC mit zwei Rücksprungpfaden, die sich „vereinen“. Kommt hier Markierungsregel MR3 zur Anwendung, würde der gesamte Rücksprungpfad als abgewählt markiert, obwohl Zustand `S2'` in Abbildung 4.13d über die eingehende Transition (`S3`, (`P3`, `1`), `S2'`) noch regulär erreicht werden kann. Daher haben wir die Anwendung von Markierungsregel MR3 auf Verzweigungen innerhalb der OLC-Hauptpfade ($t \in path_{int}(s_{start}, s_{end})$) beschränkt und führen in Markierungsregel MR5 zusätzliche Kriterien zur Abwahl von Rücksprungpfaden ein: befindet sich ein Zustand in einem Rücksprungpfad, müssen alle in ihn eingehenden Transitionen (auch Transitionen, die Teil eines Rücksprungpfads sind) als `DISABLED` markiert sein, damit er als `SKIPPED` markiert wird. Beispiel 4.3 zeigt die Anwendung der vorgestellten Regeln für Verzweigungsabschnitte mit Rücksprung.

Beispiel 4.3 (Ausführungsszenario in einem nicht-det. zyklischen OLC)
Abbildung 4.14 zeigt als Beispiel den Ausschnitt eines nicht-deterministischen zyklischen OLC. In Schritt 1 wird Zustand `A` als `ACTIVATED` markiert. Die Markierungsänderungen in den Schritten 2 und 3 entsprechen dem definierten Verhalten einer Sequenz und werden daher nicht weiter erläutert (vgl. Abschnitt 4.3.4). In Schritt 4 wird durch Ausführungsregel AR2 der Prozess `PC` gestartet und beide aus Zustand `C` ausgehenden Transitionen als `PROCESSING` markiert. Sobald der Prozess `PC` endet, wird Markierungsregel MR2 angewendet. In Schritt 4 in Abbildung 4.14 liefert `PC` das Ergebnis $pv = 0$. MR2 markiert die Transition, die zu Zustand `B` führt, zunächst als `FIRED` und den Zustand `B` selbst als `ACTIVATED`, während `C` als `DONE` markiert wird. Die andere aus

C ausgehende Transition wird als DISABLED markiert. Ihr Zielzustand D verbleibt in seinem Initialzustand. Der Kontext von Markierungsregel MR5 ist zwar erfüllt, die andere aus C ausgehende Transition ist allerdings eine Rücksprungtransition, die gefeuert hat (d.h., es wird derzeit ein Schleifenrücksprung durchgeführt). Damit ist gewährleistet, dass keine Deadpath-Eliminierung bis zum Ende des OLC erfolgt. Ansonsten müsste im Anschluss an den Schleifenrücksprung die Markierung des abgewählten Pfads wieder rückgängig gemacht werden.

In Schritt 5 unseres Beispiels aus Abbildung 4.14 wird durch die erneute Aktivierung von Zustand B die Ausführungsregel AR4 angewendet. Sie markiert die aus B ausgehende Transition erneut als PROCESSING, wodurch der mit dieser Transition verknüpfte Prozess PB abermals instanziiert und gestartet wird. Des Weiteren wird Zustand C als NOTACTIVATED gekennzeichnet. Durch Änderung der Markierung von C ist der Kontext von Ausführungsregel AR4 erneut erfüllt und die aus C ausgehenden Transitionen werden als WAITING markiert (sofern nicht bereits in Initialmarkierung). Die Markierungen von Zustand B (bereits markiert als ACTIVATED) und Zustand D (bereits markiert als NOTACTIVATED) werden durch Markierungsregel MR4 nicht verändert.

Sobald der Prozess PB den Prozesszustand COMPLETED erreicht, werden in Schritt 6 durch Markierungsregel MR1 der Zustand C als ACTIVATED sowie Zustand B als DONE markiert und durch Ausführungsregel AR2 der Prozess PC in den Prozesszustand RUNNING überführt. Ferner werden beide aus Zustand C ausgehenden Transitionen als PROCESSING markiert. Endet PC mit dem Ergebnis 1, wird durch Markierungsregel MR2 die in Zustand B eingehende Transition als DISABLED markiert, während die in Zustand D eingehende Transition die Markierung FIRED erhält. Zustand D wird als ACTIVATED und Zustand C als DONE markiert.

Ausführungsregel AR4 (Schleifen-Reinitialisierung)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$*. Des Weiteren sei* $s^* \in S$ *ein Zustand, dessen Markierung* $SM(s^*)$ *eine der in Tabelle 4.4 beschriebenen Markierungsänderungen erfahren hat. Die Menge* $T^* := outTrans_{int}(s^*)$ *enthalte alle aus* s^* *ausgehenden Transitionen. Dann ergibt sich für olc als Folgemarkierung* $M' = (SM', TM')$ *mit:*

$$TM'(t) = \begin{cases} \texttt{PROCESSING} & \text{falls } t \in T^* \wedge SM(s^*) = \texttt{ACTIVATED} \\ \texttt{WAITING} & \text{falls } t \in T^* \wedge SM(s^*) = \texttt{NOTACTIVATED} \\ TM(t) & \text{sonst} \end{cases}$$

$$SM' \equiv SM$$

Markierungswechsel	Angewendete Markierungsregel	Beispiel
DONE auf ACTIVATED	MR1 oder MR2	Abbildung 4.13a, Zustand S1
DONE auf NOTACTIVATED	MR4	Abbildung 4.13a, Zurücksetzen von Zustand S2' nach Schleifen-Reinitialisierung
SKIPPED auf ACTIVATED	MR1 oder MR2	Abbildung 4.13a, Zustand S2
ACTIVATED auf ACTIVATED	MR2	Abbildung 4.13c, Zustand S3

Tabelle 4.4: Mögliche Markierungswechsel von Zuständen bei Schleifendurchlauf

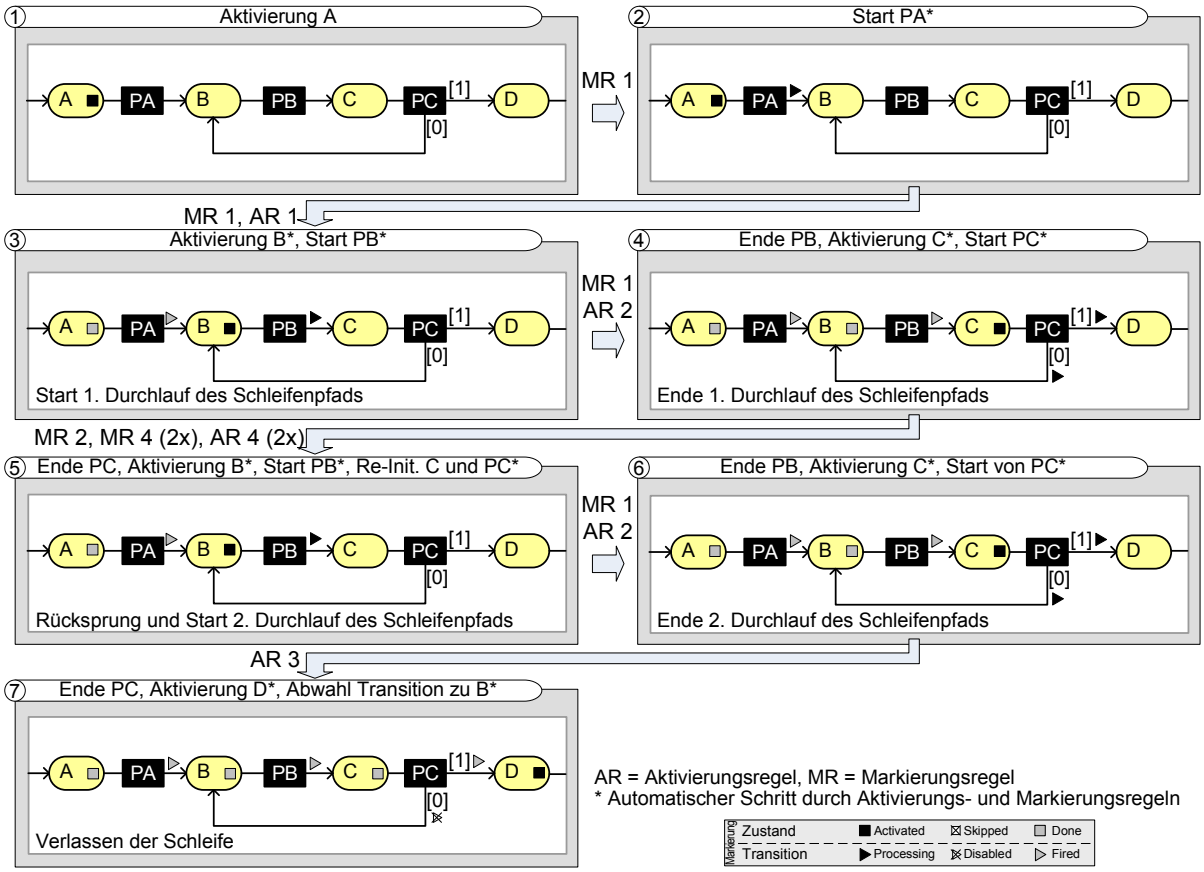


Abbildung 4.14: Anwendung der Markierungsregeln für nicht-deterministische zyklische OLCs

Markierungsregel MR4 (Schleifen-Reinitialisierung)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$*. Des Weiteren seien* $s^* \in S$ *ein Zustand und* $t^* = (src, (p, v), s^*) \in inTrans_{int}(s^*)$ *eine in* s^* *eingehende interne Transition, deren Markierung* $TM(t^*)$ *eine der in Tabelle 4.5 beschriebenen Markierungsänderungen erfahren hat. Dann ergibt sich für olc als Folgemarkierung* $M' = (SM', TM')$ *mit:*

$$TM' \equiv TM$$

$$SM'(s) = \begin{cases} \texttt{NotActivated} & \text{falls } s = s^* \wedge SM(s) \notin \{\texttt{Activated}, \texttt{NotActivated}\} \\ & \wedge\ (t^* \in path_{int}(s_{start}, s_{end}) \vee s \notin states_{int}(s_{start}, s_{end})) \\ SM(s) & \text{sonst} \end{cases}$$

Markierungswechsel	Angewendete Ausführungsregel	Beispiel
FIRED auf PROCESSING	AR4	Abbildung 4.13a, Zurücksetzen der Transition von S1 nach S2 nach Schleifen-Reinitialisierung
FIRED auf WAITING	AR4	Abbildung 4.13a, Zurücksetzen der Transition von S2' nach S3 nach Schleifen-Reinitialisierung
DISABLED auf PROCESSING	AR4	Abbildung 4.13a, Zurücksetzen der Transition von S1 nach S2 nach Schleifen-Reinitialisierung
DISABLED auf WAITING	AR4	Abbildung 4.13a, Zurücksetzen der Transition von S2 nach S3 nach Schleifen-Reinitialisierung

Tabelle 4.5: Mögliche Markierungswechsel von Transitionen bei Schleifendurchlauf

Markierungsregel MR5 (Abwahl toter Schleifenpfade; Deadpath-Eliminierung)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren sei $s^* \in S$ ein Zustand mit einer oder mehreren eingehenden Transitionen der Menge $T^* = inTrans_{int}(s^*)$. Ferner habe s^* mindestens eine eingehende Transition $t^* \in T^*$ mit $t^* \notin path_{int}(s_{start}, s_{end})$. Wenn für eine eingehende Transition $\tilde{t} = (\tilde{src}, (\tilde{p}, \tilde{v}), \tilde{targ}) \in T^*$ die Markierung $TM(\tilde{t})$ von WAITING *in* DISABLED *wechselt, ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM' \equiv TM$$

$$SM'(s) = \begin{cases} \text{SKIPPED} & \text{falls } s = s^* \wedge s \neq s_{start} \wedge \big((s \in states_{int}(s_{start}, s_{end}) \\ & \quad \wedge \forall t \in T^* \cap path_{int}(s_{start}, s_{end}) : TM(t) = \text{DISABLED} \\ & \quad \wedge \exists \hat{t} \in outTrans_{int}(\tilde{src}) : \hat{t} \in path_{int}(s_{start}, s_{end}) \wedge TM(\hat{t}) = \text{FIRED}) \\ & \quad \vee (s \notin states_{int}(s_{start}, s_{end}) \wedge \forall t \in T^* : TM(t) = \text{DISABLED})\big) \\ SM(s) & \text{sonst} \end{cases}$$

Hinsichtlich der Terminierung gilt für einen nicht-deterministischen zyklischen OLC dieselbe Aussage wie für einen nicht-deterministischen azyklischen OLC (vgl. Satz 4.2): Nach Erreichen des Endzustands des OLC sind alle Zustände als DONE oder SKIPPED sowie alle Transitionen als FIRED oder DISABLED markiert.

4.3.7 Ausführung

Im Abschnitt 4.5 gehen wir auf die Koordination unterschiedlicher OLCs innerhalb einer Prozessstruktur ein. Hierbei ist es notwendig, von den Markierungen der Elemente eines OLC zu abstrahieren und grobgranulare Aussagen über die aktuelle Ausführung eines OLC treffen zu können. Beispielsweise wird relevant sein, ob sich ein OLC derzeit in Ausführung befindet oder ob er seinen Endzustand bereits erreicht hat. Bei näherer Betrachtung ergeben sich vier *Phasen*, die ein OLC durchläuft: INITIAL, RUNNING, TERMINATED und ARCHIVED (vgl. Definition 4.10). Die Bedeutung dieser Phasen beschreibt Beispiel 4.4.

Beispiel 4.4 (Phasenübergänge eines OLC)
Abbildung 4.15 zeigt einen OLC in vier verschiedenen Situationen seiner Ausführung. Nach Anwendung der Initialisierungsregel IR1 (siehe Seite 67) befindet sich der OLC in der Phase INITIAL (vgl. Definition 4.10). Mit der Aktivierung seines Startzustands beginnt die Ausführung des OLC und die aus den Abschnitten 4.3.4, 4.3.5 und 4.3.6 bekannten Regeln zur Anpassung der Laufzeitmarkierungen werden angewendet. Er befindet sich nun in der Phase RUNNING. Der Kreislauf mit Kontextänderung sowie nachfolgender Anwendung weiterer Regeln wiederholt sich, bis der OLC schließlich seinen Endzustand s_{end} erreicht. Durch Aktivierung des Endzustands wird der OLC in die Phase TERMINATED überführt. Die Phase ARCHIVED tritt schließlich in Kraft, sobald der Endzustand des OLC als DONE markiert ist. Dies erfolgt mit Beendigung der Prozessstruktur, in die der OLC eingebettet ist (siehe Abschnitt 4.6.4).

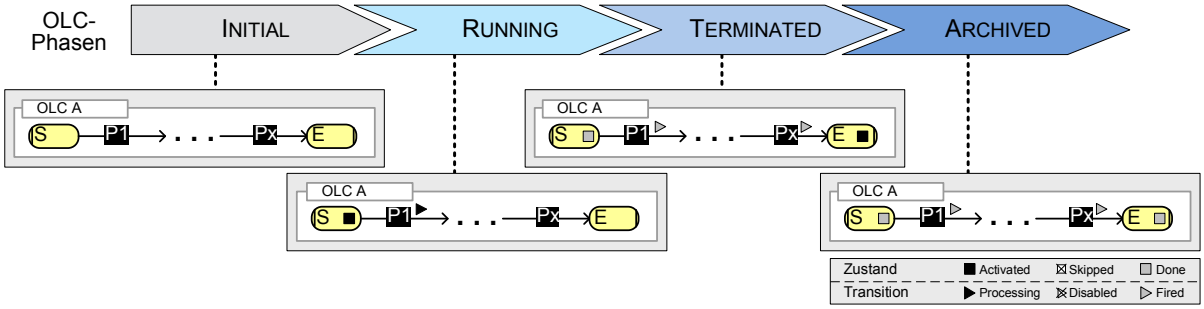


Abbildung 4.15: Phasen eines OLC während der Ausführung

Definition 4.10 (OLC Phasen)
Wir bezeichnen eine Abbildung $OP : \mathcal{OLC} \mapsto OLCPhases$*, die jedem* $olc = (P, V, TS) \in \mathcal{OLC}$ *eine Phase* $OP(olc) \in OLCPhases$ *zuordnet, als Phase von olc. OLCPhases ist wie folgt definiert:*

$$OLCPhases := \{\text{INITIAL}, \text{RUNNING}, \text{TERMINATED}, \text{ARCHIVED}\}.$$

Seien $TS = (S, T, s_{start}, s_{end})$ *das Transitionssystem von olc und* $OP(olc)$ *die aktuelle Phase von olc. Ändert sich die Markierung des Startzustands* s_{start} *oder des Endzustands* s_{end} *von olc, ergibt sich folgende Phasenänderung* $OP'(olc)$*:*

$$OP'(olc) = \begin{cases} \text{INITIAL} & \text{falls } SM(s_{start}) = \text{NOTACTIVATED} \\ \text{RUNNING} & \text{falls } SM(s_{start}) \in \{\text{ACTIVATED}, \text{DONE}\} \wedge\ SM(s_{end}) = \text{NOTACTIVATED} \\ \text{TERMINATED} & \text{falls } SM(s_{end}) = \text{ACTIVATED} \\ \text{ARCHIVED} & \text{falls } SM(s_{end}) = \text{DONE} \end{cases}$$

Wir gehen weiter davon aus, dass in einem OLC (bzw. einer OLC-Instanz) der *Verlauf* (*Trace*) protokolliert wird. Das Protokoll beinhaltet die folgenden Felder: *Prozess*, *Prozessstart*, *Prozessende* und *Prozessergebnis* (vgl. Tabelle 4.6). Die Protokollierung des OLC wird spätestens dann relevant, wenn mehrere OLCs in eine Prozessstruktur integriert und während der dynamischen Adaption Aussagen zur Konsistenz von Änderungen notwendig werden. Abbildung 4.16 zeigt den Verlauf für das Szenario aus Abbildung 4.14. Der Verlauf erlaubt es, nach Ausführung eines

OLC den exakten Ablauf zu rekonstruieren bzw. dessen Ausführung zu analysieren (z.B. mittels *Process Mining* Techniken [ADH+03]).

Attribute	Beschreibung
Prozess	Name bzw. Instanz-ID des Prozesses.
Prozessstart	Zeitstempel der Instanziierung und des Starts des Prozesses (Wechsel auf Prozesszustand RUNNING; vgl. Abschnitt 4.3.2).
Prozessende	Zeitstempel der Beendigung des Prozesses (Wechsel auf Prozesszustand COMPLETED; vgl. Abschnitt 4.3.2).
Prozessergebnis	Das vom Prozess nach Beendigung gelieferte Ergebnis.

Tabelle 4.6: Relevante Felder für das Verlaufsprotokoll eines OLC

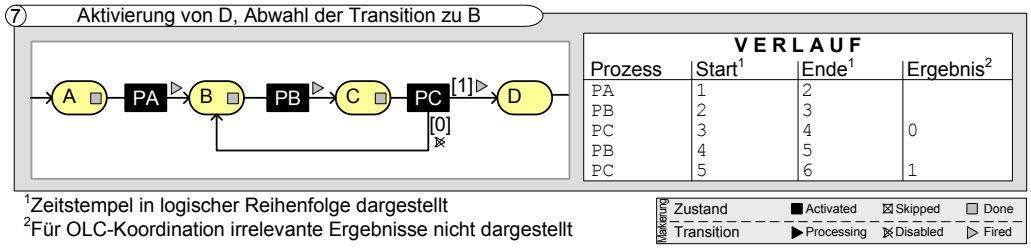


Abbildung 4.16: Verlauf für das Szenario aus Abbildung 4.14

4.4 Dynamische Eigenschaften von Object Life Cycles

Die Zusicherung gewisser dynamischer Eigenschaften für OLCs ist ein wichtiger Grundpfeiler unseres Ansatzes, um den Forderungen nach korrekter Ausführbarkeit und dynamischer Adaptierbarkeit gerecht zu werden (vgl. Anforderung 4.4). Strukturelle Korrektheit definiert zwar grundlegende statische Eigenschaften (vgl. Abschnitt 4.2.3), allerdings kann es zur Laufzeit immer noch zu unerwünschtem Verhalten kommen. Die Literatur nennt verschiedene Verhaltensmuster, die in strukturell korrekten Prozessen auftreten können [Aal97b, Wes07]:

- *Verklemmungen* (Deadlocks): es kann keine Aktivität mehr ausgeführt werden, aber der Endknoten ist noch nicht erreicht,
- *Endlosschleifen* (Livelocks): es können Aktivitäten ausgeführt werden, aber der Endknoten wird nicht erreicht und
- *Inkorrekte Terminierung*: es befinden sich noch Aktivitäten in der Ausführung wenn der Endknoten bereits erreicht wurde.

In der Literatur für Workflow-Netze werden Kriterien definiert, auf deren Basis die genannten Verhaltensmuster ausgeschlossen werden können [Aal97b, Wes07]. Bilden wir diese Kriterien auf OLCs ab, müssen wir zeigen, dass ein OLC keine Verklemmungen aufweist, korrekt terminiert und alle Transitionen und Zustände vom Startzustand aus erreicht werden können (vgl. Definition 4.11).

Definition 4.11 (Dynamische Korrektheit eines OLC)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$.

(a) Verklemmungsfreiheit*: Wird der Startzustand von olc als $SM(s_{start})$ =* ACTIVATED *markiert, so wird nach endlicher Zeit der Endzustand von olc als $SM(s_{end})$ =* ACTIVATED *markiert.*

(b) Terminierung*: Wird der Endzustand von olc als $SM(s_{end})$ =* ACTIVATED *markiert, müssen alle Zustände außer dem Endzustand als* DONE *oder* SKIPPED *sowie alle Transitionen als* FIRED *oder* DISABLED *markiert sein.*

(c) Aktivierbarkeit*: Zum Zeitpunkt des Starts von olc, müssen alle Zustände als* NOTACTIVATED *und alle Transitionen als* WAITING *markiert und olc muss strukturell korrekt sein.*

Durch den Ausschluss von Nebenläufigkeit innerhalb eines einzelnen OLC haben wir es mit einem sequentiellen Ablauf zu tun. Dadurch kann es auf Ebene einzelner OLCs nicht zu Verklemmungen (Deadlocks) kommen. Endlosschleifen (Livelocks) haben wir behandelt, indem eine Transitionsbedingung nur einen Wert aufweisen darf, den der mit der Transition verknüpfte Prozess auch liefert.[16]

Die Terminierung eines OLC ist durch die Sätze 4.1 und 4.2 gewährleistet. So wird nach Aktivierung des Endzustands kein Prozess mehr ausgeführt (es sind also alle internen Transitionen als FIRED oder DISABLED markiert). Hierfür setzen wir voraus, dass Prozesse in endlicher Zeit terminieren, d.h. der Prozesszustand COMPLETED wird in endlicher Zeit nach Ausführung der Aktion START erreicht.

Die dynamische Korrektheit kann somit für alle erstellbaren OLCs zugesichert werden.

4.5 Bildung von Prozessstrukturen

Auf Grundlage von Object Life Cycles kann die Abfolge der Prozesse zur Verarbeitung einzelner Objekte beschrieben werden. In unserem Anwendungsbeispiel, dem Release-Management in der Fahrzeugentwicklung (vgl. Abschnitt 2.1), müssen typischerweise komplexe *Prozessstrukturen* gebildet werden. Dafür sollen die Prozesse unterschiedlicher Objekte nebenläufig ausgeführt und miteinander synchronisiert werden (vgl. Anforderung 4.2). So darf zum Beispiel der Testprozess für das Navigationssystem erst dann gestartet werden, wenn alle seine Subsysteme freigegeben und integriert sind (vgl. Abbildung 4.1). Um die Prozesse der verschiedenen Objekte koordinieren zu können, müssen wir in der Lage sein, entsprechende Abhängigkeiten zwischen den OLCs abzubilden und damit eine Prozessstruktur beschreiben. In den folgenden Abschnitten wird das eingeführte Metamodell dahingehend erweitert.

[16] Es ist die Aufgabe des Modellierers zu garantieren, dass diejenige Transitionsbedingung, die zum Abbruch der Schleife führt, zur Laufzeit auch erfüllt werden kann (vgl. Abschnitt 4.2).

4.5.1 Konzept zur Bildung von Prozessstrukturen

Unsere Ziele für die Bildung von Prozessstrukturen sind die Erstellung von Prozessstrukturen durch Integration verschiedener OLCs, ihre nebenläufige Ausführung und die Synchronisation ihrer Prozesse. Die Ausführung verschiedener OLCs kann durch die in Abschnitt 4.3 vorgestellte operationale Semantik nebenläufig erfolgen. Die Synchronisation der OLCs durch geeignete Synchronisationsmechanismen wurde jedoch bislang nicht betrachtet.

Ein im Prozess-Management häufig genutzter Mechanismus ist die *ereignisorientierte Synchronisation* [Wes07]. Soll beispielsweise ein Prozess A mit einem Prozess B synchronisiert werden, wartet A an der Position, an der synchronisiert werden soll. Empfängt A das erwartete Ereignis, welches B zu einem bestimmten Zeitpunkt sendet, fährt A mit der Ausführung ab dieser Position fort. Der Ansatz hat jedoch für die Anwendung in datengetriebenen Prozessstrukturen zwei Nachteile. Zum einen müssen die Ereignisse im Prozess fest verankert und der jeweilige Sender bzw. Empfänger der Ereignisse bekannt sein. In datengetriebenen Prozessstrukturen soll jedoch der flexible Aufbau einer Prozessstruktur, ohne Anpassung der einzelnen Prozesse bzw. OLCs, möglich sein. Zum anderen lassen sich mit ereignisbasierter Synchronisation die Synchronisationsprozesse (vgl. Anforderung 4.3) nur kompliziert durch „Workarounds“ realisieren.

Ein weiterer Ansatz ist die Beschreibung der Synchronisation unterschiedlicher Prozesse (bzw. der Zustände der OLCs) durch eine explizite Nachrichtenbeziehung (z.B. *Message Flow* in BPMN [Bus06]). Die Synchronisationsinformation wird dabei nicht im OLC (bzw. Prozess) selbst verankert, sondern als Element der Prozessstruktur angeboten. Die OLCs können damit unabhängig vom konkreten Kontext modelliert und müssen bei Einfügen einer Synchronisation nicht angepasst werden. Ferner erlaubt diese Synchronisation von OLCs eine intuitive und zugleich robuste Modellierung von Abhängigkeiten. Bei der Synchronisation über Zustände in OLCs können Prozesse in verschiedenen OLCs wiederverwendet werden, ohne die Synchronisationsbeziehungen anpassen zu müssen. Durch eine solche *lose Kopplung* der OLCs stellen wir sicher, dass eine Adaption der Prozessstruktur durch Hinzufügen oder Entfernen von OLCs oder Synchronisationsbeziehungen gewährleistet bleibt (vgl. Anforderung 4.5). Wir nennen derartige Synchronisationsbeziehungen *externe Transitionen*. Wird eine externe Transition – wie eine interne Transition – mit einem Prozess verknüpft, können im Kontext der Synchronisation auch *Synchronisationsprozesse* ausgeführt werden (vgl. Anforderung 4.3). Beispiel 4.5 zeigt ihre Anwendung zur Synchronisation von OLCs in einer Prozessstruktur.

Beispiel 4.5 (Synchronisation von OLCs innerhalb einer Prozessstruktur)
Abbildung 4.17a zeigt zunächst unsynchronisierte OLCs innerhalb einer Prozessstruktur. Kommt die Prozessstruktur zur Ausführung, werden die OLCs gleichzeitig gestartet und nebenläufig ausgeführt (vgl. Anforderung 4.3). In Abbildung 4.17b sind die OLCs aus Abbildung 4.17a mit zusätzlichen Kanten, sogenannten externen Transitionen, abgebildet. Eine externe Transition wird in Abbildung 4.17b zum Beispiel genutzt, um den Zustand `S1` in OLC `C` mit Zustand `S1` in OLC `A` zu verbinden. Durch Erweiterung der operationalen Semantik (siehe Abschnitt 4.6) bewirkt die externe Transition, dass `S1` in OLC `A` nur dann als `ACTIVATED` markiert werden kann, wenn `S1` in OLC `C` aktiviert ist oder bereits durchlaufen wurde.

In Corepro lassen sich Abhängigkeiten zwischen OLCs mit der Ausführung von Synchronisationsprozessen verbinden, indem wir externe Transitionen mit Prozessen verknüpfen. Die von

Zustand `S1` in OLC `D` ausgehende und in Zustand `S2` in OLC `A` eingehende externe Transition ist mit dem (Synchronisations-)Prozess `PE1` verknüpft. Zustand `S2` in OLC `A` kann erst aktiviert werden, wenn der Prozess `PE1` beendet wird. `PE1` wird wiederum erst gestartet, wenn Zustand `S1` in OLC `D` als `ACTIVATED` markiert ist.

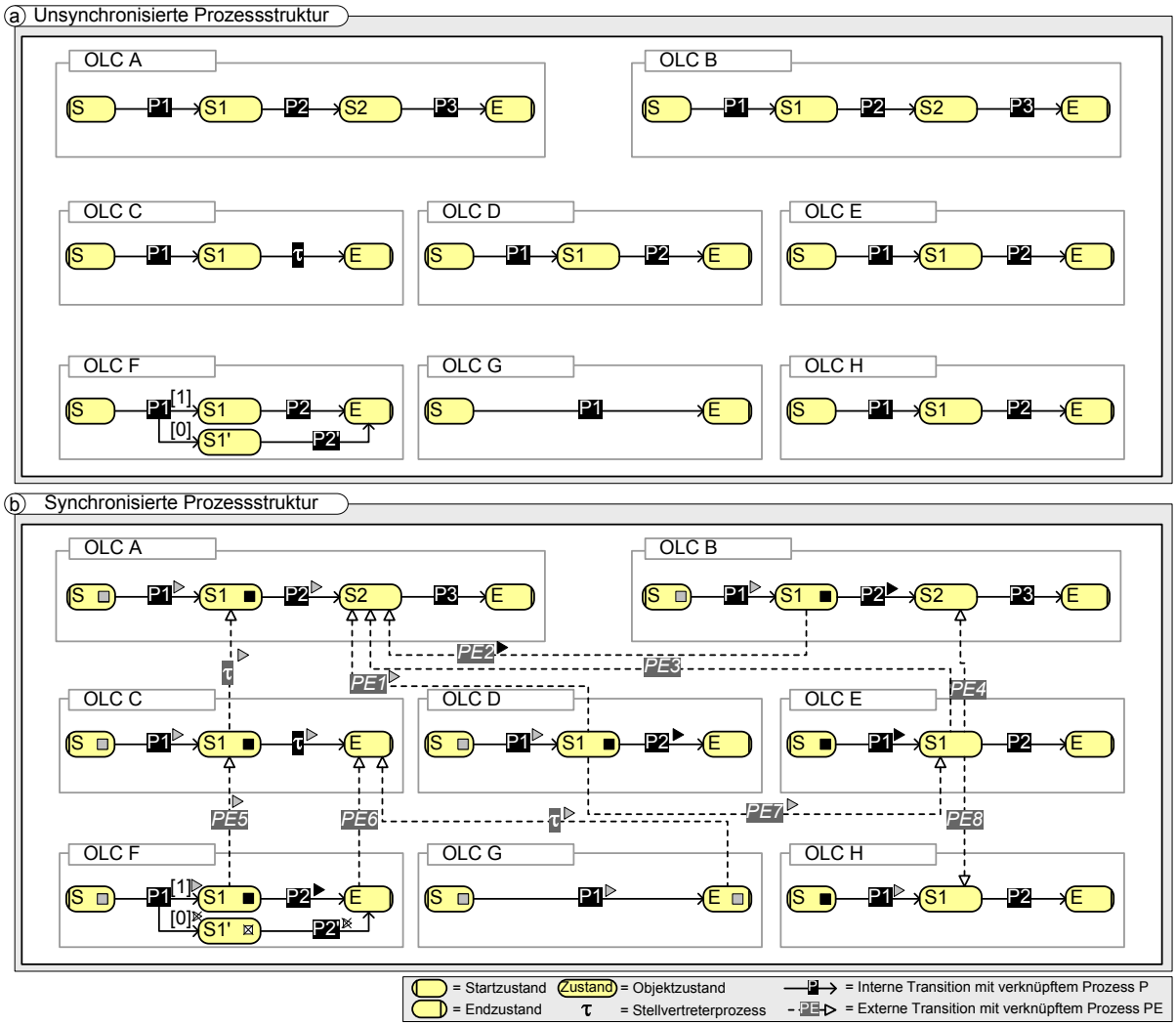


Abbildung 4.17: Prozessstruktur mit synchronisierten und unsynchronisierten OLCs

Die Anwendung von Synchronisationsprozessen ist optional (vgl. Szenario 4.3). Um nicht formal zwischen externen Transitionen mit und ohne Synchronisationsprozess unterscheiden zu müssen, definieren wir einen *aktionsneutralen Stellvertreterprozess* τ (*Silent Process*). Er kann sowohl mit externen als auch internen Transitionen verknüpft werden (vgl. externe Transition von Zustand `S1` in OLC `C` zu Zustand `S1` in OLC `A` in Abbildung 4.17b). Der Stellvertreterprozess τ kann ebenfalls die in Abschnitt 4.3.2 beschriebenen Prozesszustände annehmen, wobei er mit Aufruf der Aktion START sofort in den Prozesszustand `COMPLETED` überführt wird. Seine Verwendung kann auch für interne Transitionen in Sequenzabschnitten sinnvoll sein, wenn die Aktivierung des

Folgezustands nur von eingehenden externen Transitionen abhängen soll (vgl. interne Transition (`C.S1`, (τ, $\sigma(\tau)$), `C.E`) in Abbildung 4.17b).

Zwar erscheint die Synchronisation von OLCs bzw. ihrer Zustände mittels externer Transitionen auf den ersten Blick trivial. Es kommt jedoch insbesondere bei der Synchronisation nicht-deterministischer OLCs mit bedingten Verzweigungen und Rücksprüngen zu Situationen, die einer genauen Untersuchung bedürfen. Als Beispiel zeigt Abbildung 4.17b mit OLC `F` einen nicht-deterministischen OLC, der eine ausgehende externe Transition von Zustand `S1` in OLC `F` zum Zustand `S1` in OLC `C` aufweist. Der Zustand `S1` in OLC `F` befindet sich in einem Pfad, der zur Laufzeit abgewählt werden könnte. Die folgenden Abschnitte diskutieren geeignete Mechanismen für den Umgang mit derartigen Szenarien. Zuvor wollen wir den formalen Aufbau einer Prozessstruktur bestehend aus OLCs und externen Transitionen festlegen.

4.5.2 Aufbau einer Prozessstruktur

Für die Synchronisation von Zuständen unterschiedlicher OLCs führen wir mit *externen Transitionen* einen neuen Transitionstyp ein. Externe Transitionen haben, wie interne Transitionen auch, einen *Quell-* und einen *Zielzustand*. Diese befinden sich bei externen Transitionen jedoch in unterschiedlichen OLCs (vgl. Definition 4.12). Ferner können externe Transitionen mit einem Prozess, genauer gesagt einem *Synchronisationsprozess*, verknüpft werden (vgl. Anforderung 4.3).

Definition 4.12 (Externe Transition)
Seien $olc_i = (P_i, V_i, TS_i) \in \mathcal{OLC}, i = 1, 2$ *zwei unterschiedliche Object Life Cycles mit Transitionssystemen* $TS_i = (S_i, T_i, s_{start_i}, s_{end_i})$ *(vgl. Definition 4.1). Eine externe Transition von* olc_1 *nach* olc_2 *ist ein Tupel* $est = (s_1, p, s_2)$ *mit:*

- $s_1 \in S_1$ *ist Quellzustand der externen Transition aus* olc_1
- p *ist der Synchronisationsprozess*
- $s_2 \in S_2$ *ist Zielzustand der externen Transition in* olc_2*;* s_2 *darf nicht der Startzustand von* olc_2 *sein (d.h.* $s_2 \neq s_{start_2}$*).*

$\mathcal{EST}$ *beschreibe die Menge aller externen Transitionen.*

Das Verhalten von externen Transitionen unterscheidet sich zur Laufzeit von dem interner. Während interne Transitionen die Zustandswechsel innerhalb von OLCs steuern, übernehmen externe Transitionen die Synchronisation von Zuständen verschiedener OLCs. Der OLC, in dem sich der Quellzustand einer externen Transition befindet, wird unabhängig durchlaufen (vgl. Abschnitt 4.6). Der OLC, in dem sich ihr Zielzustand befindet, blockiert dagegen die Aktivierung des Zielzustands solange, bis die externe Transitionen feuert. Das Feuern einer externen Transition ist somit eine notwendige, aber nicht hinreichende Bedingung für die Aktivierung ihres Zielzustands (d.h. sie kann keinen expliziten Zustandsübergang im Ziel-OLC auslösen). Um die korrekte Ausführung synchronisierter OLCs zu gewährleisten, dürfen keine externen Transitionen in einen OLC-Startzustand eingehen (vgl. Definition 4.12). Ansonsten könnten der Startzustand bei Ausführung der Prozessstruktur ggf. nicht aktiviert und damit der OLC nicht gestartet werden (vgl. Initialisierungsregel IR1).

Um die Synchronisation eines Zustands mit mehreren innerhalb einer Prozessstruktur nebenläufigen OLCs zu erlauben, können aus einem Zustand mehrere externe Transitionen ausgehen bzw. in ihn eingehen. Die Verknüpfung mehrerer aus einem Zustand ausgehender oder in einen Zustand eingehender externer Transitionen entspricht einer *UND-Semantik*. Das bedeutet, dass sie parallel ausgeführt werden und, im Gegensatz zu internen Transitionen, weder mit demselben Prozess (bzw. dessen Instanzen) noch mit einem Prozessergebnis als Bedingung verknüpft werden (vgl. Definition 4.12). Zur Aktivierung eines Zustands müssen **alle** in ihn eingehenden externen Transitionen gefeuert haben (siehe Abschnitt 4.6).

Mit Hilfe der formalen Beschreibung externer Transitionen (vgl. Definition 4.12) lassen sich nun Prozessstrukturen formal definieren. Eine Prozessstruktur besteht aus einer endlichen Menge von OLCs sowie einer endlichen Menge externer Transitionen, die Zustände verschiedener dieser OLCs verbinden (vgl. Definition 4.13). Eine Prozessstruktur ist abgeschlossen: OLCs innerhalb einer Prozessstruktur haben keine zusätzlichen externen Transitionen „nach oder von außerhalb“ der Prozessstruktur. Derartige Abhängigkeiten können jedoch durch flexible Eingriffe in den Ablauf abgebildet werden, wie wir sie in Kapitel 7 noch zeigen werden.

Definition 4.13 (Prozessstruktur)
Eine Prozessstruktur ist ein Tupel $ps = (OLC, EST)$ *mit*

- OLC *ist die Menge der in* ps *enthaltenen Object Life Cycles*
- EST *ist die Menge externer Transitionen zwischen jeweils zwei Zuständen unterschiedlicher Object Life Cycles aus* OLC.

$\mathcal{PS}$ *beschreibe die Menge aller Prozessstrukturen.*

Die verschiedenen OLCs innerhalb einer Prozessstruktur seien immer eindeutig benannt. OLC-Zustände dagegen sind zwar innerhalb eines OLC eindeutig benannt, innerhalb einer Prozessstruktur werden sie aber zusätzlich mit dem Namen ihres OLC in der Form

```
<OLC-Name>.<Zustandsname>
```

qualifiziert (sofern nicht Eindeutigkeit des referenzierten Zustands vorliegt).

4.5.3 Funktionen zur strukturellen Analyse

In Abschnitt 4.2.2 haben wir Funktionen für die strukturelle Analyse von OLCs eingeführt. Die strukturelle Analyse muss nun auf ganze Prozessstrukturen ausgedehnt werden, um deren operationale Semantik beschreiben zu können (siehe Abschnitt 4.6). Die Funktionen $inTrans_{ext}(s)$ und $outTrans_{ext}(s)$ liefern für einen gegebenen Zustand s dessen ein- und ausgehende externe *Transitionen*.

Definition 4.14 (Funktionen zur strukturellen Analyse einer Prozessstruktur)
Sei $ps = (OLC, EST)$ eine Prozessstruktur und $olc = (P, V, TS) \in OLC$ ein zugehöriger Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$. Dann gilt:

(a) $inTrans_{ext}$ liefert für einen Zustand $s^* \in S$ die Menge aller in s^* eingehenden externen Transitionen:

$$inTrans_{ext}(s^*) := \{e = (s_1, p, s_2) \in EST \mid s_2 = s^*\}$$

(b) $outTrans_{ext}$ liefert für einen Zustand $s^* \in S$ die Menge aller aus s^* ausgehenden externen Transitionen:

$$outTrans_{ext}(s^*) := \{e = (s_1, p, s_2) \in EST \mid s_1 = s^*\}$$

4.6 Operationale Semantik von Prozessstrukturen

Zustände und interne Transitionen in OLCs verfügen über Laufzeitmarkierungen zur Anzeige des aktuellen Stands ihrer Ausführung. Wir erweitern externe Transitionen ebenfalls um derartige Markierungen, um auch für sie die Vorteile dieser Anzeige nutzen zu können und darauf basierend die operationale Semantik von Prozessstrukturen zu definieren. Hierfür können wir die operationale Semantik von OLCs als Grundlage heranziehen (vgl. Abschnitt 4.3), müssen diese aber um geeignete Regeln für externe Transitionen erweitern. In den vorangehenden Abschnitten haben wir bereits verdeutlicht, dass externe Transitionen das Verhalten ihres Zielzustands beeinflussen. Deshalb muss auch die bestehende operationale Semantik für OLCs ergänzt werden.

4.6.1 Laufzeitmarkierungen externer Transitionen

Die Nutzung von Laufzeitmarkierungen für externe Transitionen erfüllt verschiedene Zwecke. Einerseits soll ihre Markierung, ebenso wie bei internen Transitionen, die Koordination der mit ihnen verknüpften Prozesse erlauben. Andererseits können sich externe Transitionen bzw. ihre Quellzustände in abwählbaren Pfaden befinden. Sie weisen daher zur Laufzeit ein ähnliches Verhalten auf, wie wir es von internen Transitionen kennen, und wir können die Laufzeitmarkierungen `WAITING`, `PROCESSING`, `FIRED` und `DISABLED` für sie übernehmen (vgl. Definition 4.15 bzw. Tabelle 4.7). In den folgenden Abbildungen entspricht die Notation der Markierungen externer Transitionen der für Markierungen interner Transitionen.

Die *Markierung einer Prozessstruktur* setzt sich zusammen aus den Markierungen aller enthaltenen externen Transitionen und OLCs (vgl. Definition 4.8).

Definition 4.15 (Markierungsfunktion für externe Transitionen)
Sei $ps = (OLC, EST)$ eine Prozessstruktur. Wir bezeichnen eine Abbildung $EM : EST \mapsto TransitionMarkings$, die jeder externen Transition $e \in EST$ eine Markierung $EM(e) \in TransitionMarkings$ zuordnet, als Transitionsmarkierung *von ps. Es gilt:*

$$TransitionMarkings := \{\texttt{WAITING}, \texttt{PROCESSING}, \texttt{FIRED}, \texttt{DISABLED}\}$$

Jede externe Transition besitzt somit genau eine dieser Markierungen (vgl. Tabelle 4.7).

Transitionsmark.	Beschreibung
WAITING	Die externe Transition wartet auf die Aktivierung ihres Quellzustands (Initialmarkierung).
PROCESSING	Der Quellzustand der externen Transition ist aktiviert worden; der mit ihr verknüpfte Prozess ist instanziiert und gestartet.
FIRED	Der mit der externen Transition verknüpfte Prozess wurde beendet.
DISABLED	Der Quell- oder Zielzustand der externen Transition befindet sich in einem abgewählten Pfad.

Tabelle 4.7: Markierungen externer Transitionen und ihre Bedeutung

Definition 4.16 (Markierung einer Prozessstruktur)
$\mathcal{M}$ bezeichne die Menge der Markierungen aller OLCs der Menge $\mathcal{OLC}$. Die Markierung PSM_{ps} einer Prozessstruktur $ps = (OLC, EST)$ setzt sich aus den Markierungen der einzelnen OLCs (d.h. $OM_{ps} = \{M_{olc} \in \mathcal{M} | olc \in OLC\}$) und den Markierungen der externen Transitionen (d.h. $EM_{ps} : EST \mapsto TransitionMarkings$) zusammen:

$$PSM_{ps} := (OM_{ps}, EM_{ps}).$$

Im Folgenden wird auf die Indizierung von PSM verzichtet, wenn Eindeutigkeit besteht.

4.6.2 Initialisierung einer Prozessstruktur

Bevor eine Prozessstruktur ausgeführt werden kann, muss sie über eine definierte Ausgangsmarkierung verfügen (vgl. Initialisierungsregel IR2). Hierfür können wir die Initialisierungsregel IR1 nutzen (siehe Seite 67), um alle OLCs in ihre Initialmarkierung zu versetzen. Initialisierungsregel IR2 versetzt weiter alle externen Transitionen in ihre Initialmarkierung WAITING.

Initialisierungsregel IR2 (Initialisierungsregel für Prozessstrukturen)
Sei $ps = (OLC, EST)$ eine Prozessstruktur. Sie wird folgendermaßen initialisiert:

- *$\forall$ $olc \in OLC$ wenden wir Initialisierungsregel IR1 an*
- *$\forall$ $e \in EST : EM(e) :=$ WAITING*

Erst wenn alle Elemente initialisiert bzw. markiert sind, wird die Prozessstruktur ausführbar. Sie wird gestartet, indem die Startzustände aller $olc \in OLC$ von Markierung NOTACTIVATED in Markierung ACTIVATED überführt werden (siehe Abschnitt 4.6.4).

4.6.3 Operationale Semantik externer Transitionen

Ergänzend zu den Regeln für die isolierte Ausführung einzelner OLCs aus Abschnitt 4.3, beschreiben wir nun die operationale Semantik ganzer Prozessstrukturen. Dies erfordert die Definition geeigneter Ausführungs- und Markierungsregeln für externe Transitionen zur Synchronisation der in der Prozessstruktur enthaltenen OLCs. Wir betrachten zunächst die Synchronisation deterministischer OLCs und gehen anschließend auf die Synchronisation nicht-deterministischer OLCs

ein. Der in Abschnitt 4.3.4 eingeführte Mechanismus für die Anwendung von Ausführungs- und Markierungsregeln wird beibehalten.

Synchronisation deterministischer OLCs

Die Synchronisation deterministischer OLCs erfordert einerseits den korrekten und nebenläufigen Start der mit externen Transitionen verknüpften Prozesse. Andererseits müssen nach deren Beendigung ggf. die jeweiligen Zielzustände aktiviert werden. Das Vorgehen bei der Markierung externer Transitionen entspricht im Grunde dem interner Transitionen: sobald ihr Quellzustand als ACTIVATED markiert ist, werden die externe Transition in die Markierung PROCESSING überführt und dadurch automatisch der mit ihr verknüpfte Prozess gestartet (vgl. Ausführungsregel AR5). Wird der Prozess beendet, kann die externe Transition als FIRED markiert werden (vgl. Markierungsregel MR6). Im Gegensatz zu Markierungsregeln für interne Transitionen darf Markierungsregel MR6 den Zielzustand einer externen Transition nur dann aktivieren, wenn sowohl alle in ihn eingehenden externen Transition, als auch eine eingehende interne Transition gefeuert haben. Ferner müssen wir diejenigen Fälle abdecken, in denen eine eingehende **interne** Transition feuert. Hierbei gilt es zu beachten, dass der Zustand nur dann aktiviert werden darf, wenn zu dem Zeitpunkt auch bereits alle eingehenden externen Transitionen (sofern vorhanden) als FIRED (bzw. DISABLED) markiert worden sind. Um genau diese Bedingung erweitert Markierungsregel MR7 die in Abschnitt 4.3 eingeführten Markierungsregeln MR1 und MR2. Beispiel 4.6 erläutert die Funktionsweise der genannten Regeln.

Beispiel 4.6 (Ausführungsszenario in der Prozessstruktur)
In Abbildung 4.18 ist die Prozessstruktur aus Abbildung 4.17b während ihrer Ausführung dargestellt. Zustand `D.S1` ist mit Zustand `A.S2` durch eine externe Transition verbunden. Diese ist mit Prozess `PE1` verknüpft. Entsprechend der nachfolgend vorgestellten Ausführungsregel AR5 wird `PE1` gestartet, sobald der Quellzustand, also `D.S1`, von NOTACTIVATED nach ACTIVATED markiert wird. Mit dem Start von `PE1` wird auch die externe Transition als PROCESSING markiert. Endet der Prozess `PE1`, kommt Markierungsregel MR6 zur Anwendung. Zunächst wird die externe Transition, mit der `PE1` verknüpft ist, als FIRED markiert. Dies entspricht auch dem Verhalten interner Transitionen. Da externe Transitionen die Semantik von Synchronisationsbeziehungen inne haben, darf der Zielzustand jedoch nur dann als ACTIVATED markiert werden, wenn bereits **eine** eingehende interne Transition als FIRED markiert und **alle** eingehenden externen Transitionen als FIRED (oder DISABLED; vgl. folgender Abschnitt) markiert sind. In Abbildung 4.18 hat Zustand `A.S2` eine eingehende interne Transition und drei eingehende externe Transitionen mit den Prozessen `PE1`, `PE2` und `PE3`. Während `PE1` bereits beendet ist und die Transition gefeuert hat, befindet sich `PE2` noch in der Ausführung. `PE3` wurde noch nicht gestartet. Der Zustand `A.S2` kann erst dann aktiviert werden, wenn sowohl `PE2` als auch `PE3` beendet sind und deren externe Transitionen als FIRED markiert wurden.

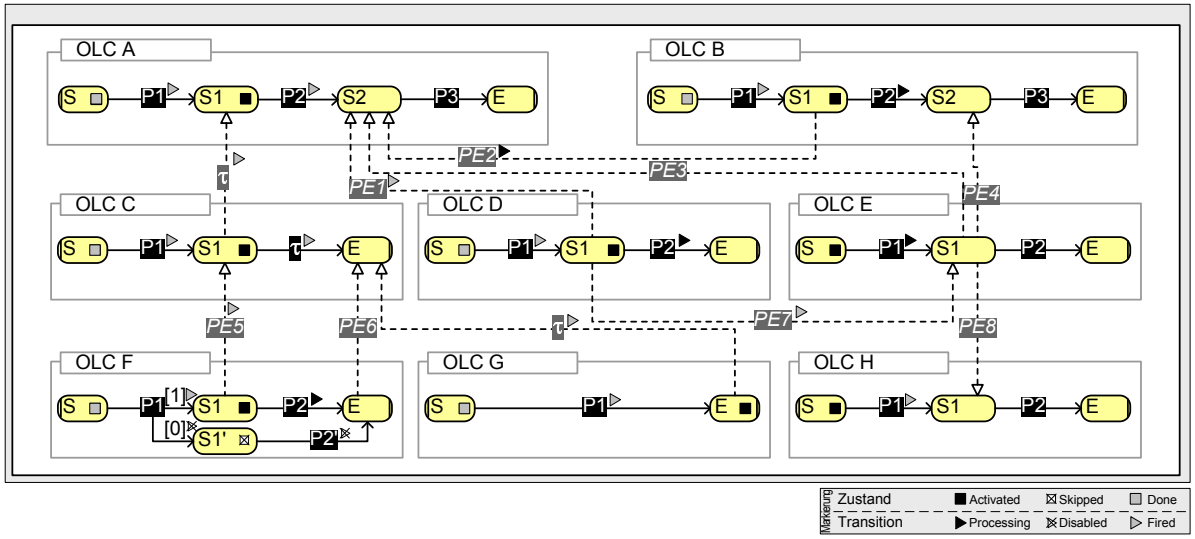


Abbildung 4.18: Prozessstruktur mit synchronisierten OLCs zur Laufzeit

Ausführungsregel AR5 (Externe Transition)
Sei $ps = (OLC, EST)$ eine Prozessstruktur und $olc = (P, V, TS) \in OLC$ ein in ihr enthaltener Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$. Die aktuelle Markierung von ps sei $PSM = (OM, EM)$. Des Weiteren sei $e^* = (s_1, p, s_2) \in EST$ eine externe Transition mit $s_1 \in S$. Wenn der Zustand s_1 von NotActivated *auf* Activated *markiert wird, dann ergibt sich die Folgemarkierung $PSM' = (OM', EM')$ für ps:*

$$EM'(e) = \begin{cases} \texttt{Processing} & \text{falls } e = e^* \wedge EM(e) = \texttt{Waiting}^a \\ EM(e) & \text{sonst} \end{cases}$$

$$OM' \equiv OM$$

Durch Markierung der ausgehenden externen Transition als Processing *wird automatisch der mit ihr verknüpfte Prozess in den Prozesszustand* Running *überführt (vgl. Abschnitt 4.3.2).*

[a] Die Prüfung der Markierung ist vonnöten, um die mehrfache Markierung externer Transitionen zu vermeiden (siehe Abschnitt 4.7).

Markierungsregel MR6 (Externe Transition)
Sei $ps = (OLC, EST)$ eine Prozessstruktur und $olc = (P, V, TS) \in OLC$ ein in ihr enthaltener Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$. Die aktuelle Markierung von ps sei $PSM = (OM, EM)$ und die aktuelle Markierung von olc sei $M_{olc} = (SM_{olc}, TM_{olc}) \in OM$. Des Weiteren sei $e^* = (s_1, p, s^*) \in EST$ eine externe Transition mit $s^* \in S$. Wenn der Prozesszustand von p von `RUNNING` *in* `COMPLETED` *wechselt, ergibt sich folgende Markierung $PSM' = (OM', EM')$ für ps:*

$$EM'(e) = \begin{cases} \texttt{FIRED} & \text{falls } e = e^* \\ EM(e) & \text{sonst} \end{cases}$$

Ist e^* die letzte in s^* eingehende externe Transition, die gefeuert hat oder abgewählt worden ist (d.h. $\forall\ e \in inTrans_{ext}(s^*) : EM(e) \in \{\texttt{FIRED}, \texttt{DISABLED}^a\}$) und hat eine in s^* eingehende interne Transition ebenfalls bereits gefeuert (d.h. $\exists\ t = (src, (p, v), s^*) \in inTrans_{int}(s^*) : TM(t) = \texttt{FIRED}$), dann gilt für $M'_{olc} = (SM'_{olc}, TM'_{olc})$:

$$SM'_{olc}(s) = \begin{cases} \texttt{ACTIVATED} & \text{falls } s = s^* \\ \texttt{DONE} & \text{falls } s = src \\ SM(s) & \text{sonst} \end{cases}$$

$$TM'_{olc} \equiv TM_{olc}$$

[a] Die Markierung `DISABLED` einer externen Transition wird im folgenden Abschnitt beschrieben.

Markierungsregel MR7 (Erweiterung MR1 und MR2)
Sei $ps = (OLC, EST)$ eine Prozessstruktur und $olc = (P, V, TS) \in OLC$ ein in ihr enthaltener Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und Markierung $M = (SM, TM)$. Des Weiteren sei $s^* \in S$ ein Zustand von olc. Wenn s^* eingehende externe Transitionen hat ($inTrans_{ext}(s^*) \neq \emptyset$), kann $SM(s^*)$ nur dann durch die Markierungsregeln MR1 und MR2 als `ACTIVATED` *und der zuvor in olc aktivierte Zustand von* `ACTIVATED` *auf* `DONE` *markiert werden, wenn zusätzlich gilt:*

- $\forall\ e \in inTrans_{ext}(s^*) : EM(e) \in \{\texttt{FIRED}, \texttt{DISABLED}^a\}$

[a] Die Markierung `DISABLED` wird im folgenden Abschnitt motiviert.

Synchronisation nicht-deterministischer OLCs

Zu Beginn dieses Abschnitts haben wir erwähnt, dass externe Transitionen auch aus Zuständen ausgehen können, die sich in einem abwählbaren Pfad befinden (vgl. OLC `F` in Abbildung 4.18). Würde der Prozess `P1` in OLC `F` aus Abbildung 4.18 mit dem Ergebnis 0 enden und den Zustand `F.S1` als `SKIPPED` markieren, wird der Kontext von Ausführungsregel AR5 nicht erfüllt. Die externe Transition (`F.S1`, `PE5`, `C.S1`) würde nie als `PROCESSING` markiert werden und damit auch nie feuern. Sie erzeugt dadurch eine Verklemmung im Ziel-OLC, denn der Kontext für die Aktivierung von Zustand `C.S1` kann nicht erfüllt werden (vgl. Markierungsregel MR7). Letztendlich entsteht dann auch eine Verklemmung in der ganzen Prozessstruktur. Um dies zu verhindern, erweitern wir die *Deadpath-Eliminierung* auf externe Transitionen und markieren sie ebenfalls

als DISABLED (vgl. Ausführungsregel AR6). Die Deadpath-Eliminierung soll allerdings mit der externen Transition enden, damit ihr Ziel-OLC wie erwartet fortfahren kann. Markierungsregel MR6 haben wir bereits dementsprechend vorbereitet, indem externe Transitionen entweder als FIRED oder DISABLED markiert sein können, um die Aktivierung eines Zustands zu veranlassen. Ein Ausführungsszenario wird in Beispiel 4.7 erläutert.

Beispiel 4.7 (Deadpath-Eliminierung in der Prozessstruktur)
Abbildung 4.19a stellt eine Prozessstruktur mit zwei OLCs dar. OLC `A` enthält eine bedingte Verzweigung, für die bereits eine Deadpath-Eliminierung durchgeführt wurde (d.h. Zustand `S1'` wurde als SKIPPED markiert). Die externe Transition (`A.S1'`, `PE1`, `B.S1`) ist als DISABLED markiert, da sich ihr Quellzustand `S1'` in OLC `A` in einem abgewählten Pfad befindet. Damit kann `B.S1` in OLC `B` aktiviert werden, d.h., es kommt nicht zu einer Verklemmung.

Abbildung 4.19b zeigt ein weiteres Szenario, in dem die externe Transition (`B.S1`, `PE1`, `A.S1'`) in einen abgewählten Zustand eingeht. Auch in diesem Fall wird die externe Transition als DISABLED markiert, sofern sie nicht bereits gefeuert hat. Dies hat zwei Gründe: erstens wird in OLC `A` die Prozessausführung nicht mehr „überwacht". Der Prozess `PE1` könnte sich also noch in der Ausführung befinden, wenn die OLCs `A` und `B` jeweils ihre Endzustände erreichen. Zweitens wird die Transition ihren Zielzustand `S1'` und damit OLC `A` nicht mehr beeinflussen. Daher braucht der Synchronisationsprozess `PE1` gar nicht ausgeführt zu werden (bzw. `PE1` kann abgebrochen werden), was in der Praxis aufgrund hoher Prozesskosten eine hohe Relevanz besitzt.

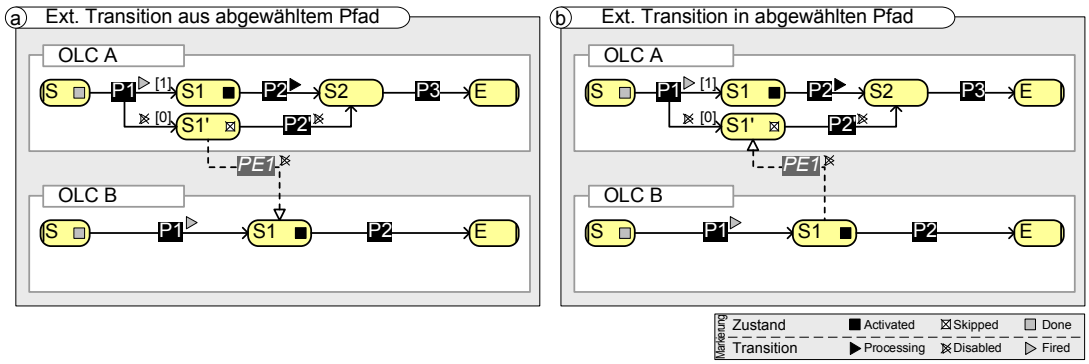


Abbildung 4.19: Deadpath-Eliminierung in der Prozessstruktur

Ausführungsregel AR6 (Abwahl externer Transitionen; Deadpath-Eliminierung)
Sei $ps = (OLC, EST)$ eine Prozessstruktur mit der aktuellen Markierung $PSM = (OM, EM)$. Des Weiteren sei $e' = (s_1, p, s_2) \in EST$ eine externe Transition. Wenn sich die Markierung von Zustand s_1 oder s_2 von NOTACTIVATED *auf* SKIPPED *ändert, ergibt sich die Folgemarkierung $PSM' = (OM', EM')$:*

$$EM'(e) = \begin{cases} \texttt{DISABLED} & \text{falls } e = e' \wedge EM(e) \in \{\texttt{WAITING}, \texttt{PROCESSING}\} \\ EM(e) & \text{sonst} \end{cases}$$

$$OM' \equiv OM$$

Durch Markierung der externen Transition als DISABLED *wird automatisch der mit ihr verknüpfte Prozess in den Prozesszustand* SKIPPED *überführt (vgl. Abschnitt 4.3.2).*

4.6.4 Ausführung einer Prozessstruktur

Die Ausführung einer Prozessstruktur folgt ähnlichen Regeln wie die Ausführung einzelner OLCs. Eine Prozessstruktur soll allerdings nicht nur isoliert ausgeführt, sondern auch als Sub-Prozess in einen übergeordneten Prozesses integriert werden können (vgl. Anforderung 4.3). Dazu muss sie nicht nur einen definierten Beginn und ein definiertes Ende aufweisen, sondern auch

- eine Methode START bereitstellen, mit der sie gestartet werden kann, und
- einen Bearbeitungszustand anzeigen, damit das Prozess-Management-System ihres übergeordneten Prozesses, die zur Ausführung notwendige Überwachung betreiben kann (etwa zum Erkennen der Terminierung der Prozessstruktur).

Der Bearbeitungszustand einer Prozessstruktur, ihrer *Phase*, leitet sich aus den Phasen ihrer OLCs ab (vgl. Abbildung 4.20). Die Regeln für diese Ableitung sind in Definition 4.17 beschrieben.

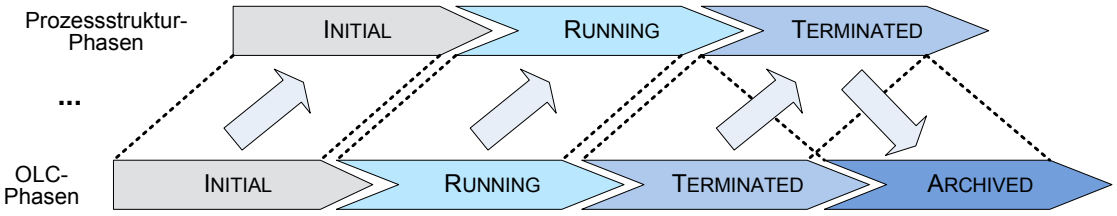


Abbildung 4.20: Phasen einer Prozessstruktur abgeleitet aus den Phasen ihrer OLCs

Definition 4.17 (Prozessstruktur-Phasen)
In COREPRO *bezeichnen wir eine Abbildung* $PP : \mathcal{PS} \mapsto PSPhases$*, die jeder Prozessstruktur* $ps \in \mathcal{PS}$ *eine Phase* $PP(ps) \in PSPhases$ *zuordnet als Phase von* ps*. Die Menge der* $PSPhases$ *ist wie folgt definiert:*

$$PSPhases := \{\texttt{INITIAL}, \texttt{RUNNING}, \texttt{TERMINATED}\}$$

Sei $ps = (OLC, EST)$ *eine Prozessstruktur und beschreibe* $PP(ps)$ *die aktuelle Phase von* ps*. Ändert sich der Status eines Object Life Cycle* $olc^* \in OLC$*, ergibt sich folgende Phasenänderung* $PP'(ps)$*:*

$$PP'(ps) = \begin{cases} \texttt{INITIAL} & \text{falls } \forall\, olc \in OLC : OP(olc) = \texttt{INITIAL} \\ \texttt{RUNNING} & \text{falls } \exists\, olc \in OLC : OP(olc) = \texttt{RUNNING} \\ \texttt{TERMINATED} & \text{falls } \forall\, olc \in OLC : OP(olc) = \texttt{TERMINATED} \end{cases}$$

Durch Aktivierung einer Prozessstruktur ps (d.h. Ausführen der Methode START) wird mit ihrer Ausführung begonnen. Folgende Schritte werden durchgeführt, um den korrekten Durchlauf der Prozessstruktur zu gewährleisten:

1. Initialisiere Prozessstruktur mittels Initialisierungsregel IR2
2. Starte jeden OLC: Markiere die Startzustände aller in ps enthaltenen OLCs als `ACTIVATED`

3. Warte bis alle in *ps* enthaltenen OLCs die Phase TERMINATED erreicht haben

4. Setze die Endzustände aller in *ps* enthaltenen OLCs auf DONE (und damit die OLCs in Phase ARCHIVED)

Die Prozessstruktur muss für die Dokumentation ihrer Ausführung einen Verlauf bereitstellen. Der Verlauf einer Prozessstruktur ergibt sich aus den *Verläufen* der einzelnen OLCs sowie der externen Transitionen innerhalb der Prozessstruktur. Der Verlauf eines OLC protokolliert bereits Start und Ende der enthaltenen Prozesse (vgl. Abschnitt 4.3.7). Sie können in den Verlauf der Prozessstruktur integriert werden. Die Protokollierung der Synchronisationsprozesse ist damit allerdings noch nicht abgedeckt. Sie kann auf unterschiedliche Arten geschehen. Eine Möglichkeit ist, die Protokollierung aller Prozesse der Prozessstruktur (d.h. die Prozesse aller OLCs und externen Transitionen) in einer „großen“ Verlaufstabelle zu integrieren. Weiter könnten auch die Verläufe der einzelnen OLCs beibehalten und nur die Ausführung der Prozesse externer Transitionen in einer Verlaufstabelle dokumentiert werden. Die dritte und von uns gewählte Möglichkeit ist, die Synchronisationsprozesse jeweils in den durch ihre externen Transitionen verbundenen OLCs mit zu protokollieren. Dabei wird entsprechend der Ausführungs- und Markierungsregeln für externe Transitionen im Quell-OLC der externen Transition der Start und in deren Ziel-OLC das Ende ihres verknüpften Prozesses protokolliert (vgl. Abbildung 4.21). Dies hat gegenüber den erstgenannten Möglichkeiten den Vorteil, dass im Rahmen der dynamischen Adaption die Konsistenzanalyse effizient für die adaptierte Prozessstruktur erfolgen kann (siehe Abschnitt 6.3).

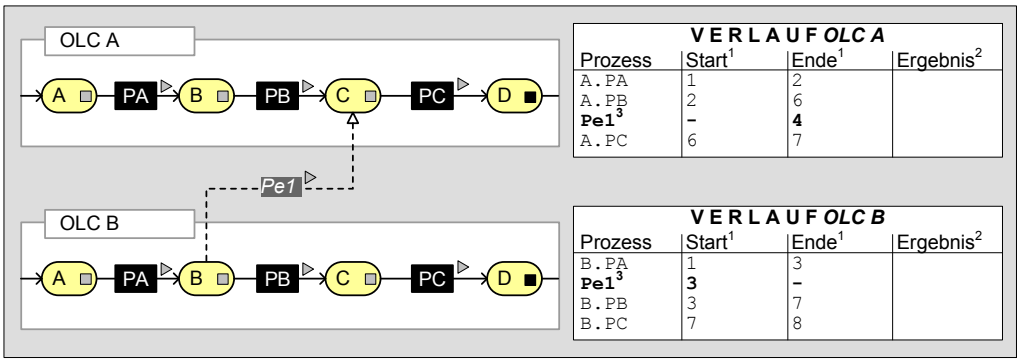


Abbildung 4.21: Verlauf der Prozessausführung innerhalb einer Prozessstruktur

4.7 Dynamische Eigenschaften von Prozessstrukturen

In den Abschnitten 4.2.3 und 4.4 wurden statische und dynamische Korrektheitskriterien aus der Literatur eingeführt und im Kontext von OLCs diskutiert. Die Korrektheitskriterien lassen sich auch auf Prozessstrukturen anwenden; es muss jedoch beachtet werden, dass in Prozessstrukturen, im Gegensatz zu OLCs, auch Nebenläufigkeit sowie Synchronisationsbeziehungen vorliegen.

Die Nutzung externer Transitionen führt zu keiner strukturellen Änderung der einzelnen OLCs. Die *strukturelle Korrektheit* (vgl. Definition 4.5) kann somit für OLCs weiterhin garantiert werden. Darüber hinaus kann sie auf Prozessstrukturen ausgedehnt werden, denn externe Transitionen erfüllen durch ihre formale Definition ebenfalls die Anforderungen an die Erreichbarkeit von Zuständen verschiedener OLCs. Damit bleibt auch in einer Prozessstruktur die strukturelle Korrektheit erhalten.

Die dynamische Korrektheit, also die Verhinderung von Verklemmungen, Endlosschleifen und inkorrekter Terminierung, bedarf jedoch aufgrund der neu eingeführten bzw. geänderten Ausführungs- und Markierungsregeln einer Überprüfung und einer intensiven Diskussion bezüglich ihrer Erweiterung auf Prozessstrukturen. In diesem Kontext unterscheiden wir zunächst unsynchronisierte und synchronisierte Prozessstrukturen.

Besteht eine Prozessstruktur nur aus unsynchronisierten OLCs (d.h. die Prozessstruktur enthält keine externen Transitionen), terminiert sie immer, wenn die einzelnen OLCs korrekt sind (vgl. Abschnitt 4.4). Dies folgt unmittelbar aus der Tatsache, dass die Terminierung der Prozessstruktur nur davon abhängt, ob die Endzustände aller OLCs erreicht werden (vgl. Definition 4.17). Da die Einbettung von OLCs in die Prozessstruktur nicht zu Änderungen an den OLCs selbst führt, ist die Korrektheit entsprechend der dynamischen Eigenschaften von OLCs in Abschnitt 4.4 weiterhin gewährleistet.

Besteht eine Prozessstruktur aus synchronisierten OLCs, enthält sie also auch externe Transitionen, muss ihre *dynamische Korrektheit* (d.h. Verklemmungsfreiheit, Terminierung und Aktivierbarkeit) explizit geprüft werden. Grund für die zusätzliche Prüfung ist, dass die OLCs in der Prozessstruktur nebenläufig ausgeführt werden und die Definition der operationalen Semantik von externen Transitionen Einfluss auf die operationale Semantik der enthaltenen OLCs ausübt. Für die korrekte Terminierung der Prozessstruktur muss deshalb geprüft werden, ob externe Transitionen zu Verklemmungen oder Endlosschleifen führen können. Letzteres kann ausgeschlossen werden, da externe Transitionen „lediglich“ Synchronisationsbeziehungen repräsentieren. Verklemmungen lassen sich mit ihnen allerdings erzeugen. Eine Verklemmung kann (spätestens) dann erkannt werden, wenn die Prozessstruktur einen oder mehrere OLCs enthält, die ihren Endzustand noch nicht erreicht haben, in der Prozessstruktur aber auch kein Prozess mehr ausgeführt wird. Dann ist für keine Ausführungs- oder Markierungsregel der entsprechende Kontext erfüllbar.

Eine typische Ursache für Verklemmungen sind „unerwünschte“ *Zyklen* in der Prozessstruktur. Ein Zyklus führt zu einer Verklemmung, wenn mehrere OLCs synchronisiert werden und eine zyklische Abhängigkeit über externe Transitionen entsteht. Zyklen können in Prozessstrukturen in verschiedenen Kontexten auftreten. Sie führen allerdings nicht zwangsläufig zu einer Verklemmung. Beispiel 4.8 beschreibt hierfür verschiedene Szenarien.

Beispiel 4.8 (Zyklen in einer Prozessstruktur)
In Abbildung 4.22a ist eine Prozessstruktur mit den beiden externen Transitionen (`A.S1`, `PE1`, `B.S1`) und (`B.S2`, `PE2`, `A.S2`) dargestellt. Sie synchronisieren wechselseitig die OLCs `A` und `B`, ohne zu einer Verklemmung zu führen.

In Abbildung 4.22b sind ebenfalls zwei externe Transitionen ((`A.S2`, `PE1`, `B.S1`) und (`B.S2`, `PE2`, `A.S1`)) dargestellt. In OLC `A` bildet die interne Transition (`S2`, (`P4`, `1`), `S1`) eine Schleife. Entsprechend der in Abschnitt 4.6 vorgestellten operationalen Semantik von externen Transitionen, wird

in Abbildung 4.22b die aus Zustand `A.S1` ausgehende externe Transition nur einmal als `FIRED` (bzw. `DISABLED`) markiert, d.h. beim ersten Schleifendurchlauf.[17] Die in Zustand `A.S2` eingehende externe Transition behindert einen erneuten Durchlauf der Schleife in OLC `A` ebenfalls nicht, denn ihre Markierung bleibt nach erstmaligem Setzen als `FIRED` ebenfalls unverändert. Damit kommt es auch im Szenario aus Abbildung 4.22b zu keiner Verklemmung.

In Abbildung 4.22c ist eine Prozessstruktur mit den beiden externen Transitionen (`A.S1`, `PE1`, `B.S1`) und (`B.S2`, `PE2`, `A.S2`) dargestellt. Die Prozessstruktur enthält, wie in Abbildung 4.22b, einen Zyklus. Er wird jedoch nicht durch eine interne Transition, sondern durch die Anordnung der externen Transitionen erzeugt. Zur Laufzeit warten jeweils die Zustände `S1` der OLCs `A` und `B` wechselseitig auf das Feuern der eingehenden externen Transitionen. Sie können jedoch nicht als `PROCESSING` und damit auch nicht als `FIRED` markiert werden, da jeweils die Zustände `S2` der OLCs `A` und `B` noch nicht aktiviert worden sind. Obwohl die OLCs an sich korrekt sind, kommt es durch die Anordnung der externen Transitionen zu einer Verklemmung und die Prozessstruktur kann nicht terminieren.

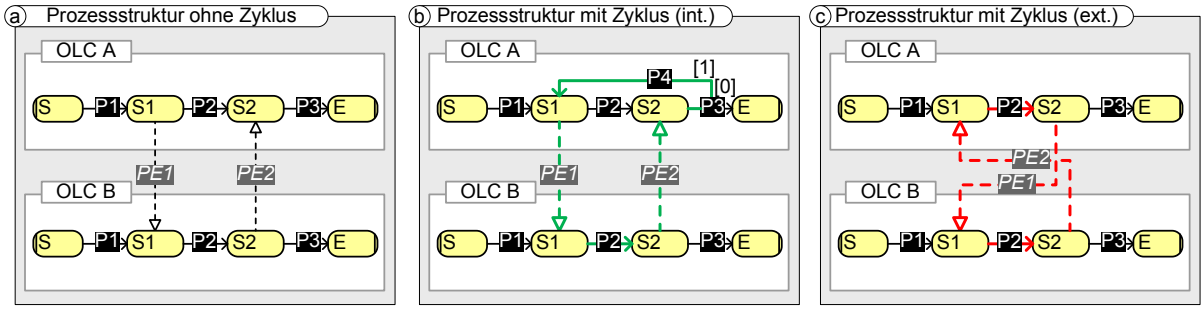


Abbildung 4.22: Prozessstrukturen mit verschiedenen Arten von Zyklen

Beispiel 4.8 illustriert triviale Szenarien, deren Zyklen manuell erkennbar sind. Erstrecken sich Zyklen in einer komplexen Prozessstruktur zum Beispiel über mehrere OLCs, ist die manuelle Suche von Zyklen in der Regel nicht mehr praktikabel. Korrektheitsgefährdende Zyklen müssen daher automatisch gefunden werden können. Theoretisch lassen sich Zyklen in einer Prozessstruktur leicht identifizieren: Ein Zyklus innerhalb der Prozessstruktur entsteht, wenn vom Zielzustand einer externen Transition ein Pfad zu ihrem Quellzustand gefunden werden kann. Wie Beispiel 4.8 zeigt, führt jedoch nicht jeder Zyklus zu einer Verklemmung. Wir müssen uns daher zunächst die Frage stellen, wodurch sich verklemmungsfreie von verklemmenden Zyklen unterscheiden, bevor wir eine geeignete *Zyklensuche* definieren können.

Die Prozessstruktur aus Abbildung 4.22b enthält zwar einen (OLC-internen) Zyklus, er führt aber nicht zu einer Verklemmung und ist daher erlaubt. Die Zyklensuche darf also keine Pfade zurückliefern, die als Rücksprünge interner Transitionen realisiert sind. Rücksprungpfade überhaupt nicht zu durchlaufen, reicht allerdings nicht aus. In Abbildung 4.23a zum Beispiel befindet sich der Zustand `S2'` innerhalb eines Rücksprungpfads, der über die aus `S3` ausgehende interne Transition erreicht werden kann. Wird der OLC in einer Prozessstruktur mit einem anderen OLC durch eine von `S2'` ausgehende externe Transitionen verknüpft (vgl. Abbildung 4.23a), muss der Rücksprungpfad bis Zustand `S2'` mit berücksichtigt werden, ansonsten könnte der Zyklus

[17] Eine Diskussion zur Semantik dieses Szenarios findet sich am Ende dieses Abschnitts.

(von `B.S1` über `B.S2` und von `A.S3` nach `A.S2'`) nicht gefunden werden. Er löst eine Verklemmung aus, da zur Laufzeit der Zustand `S2'` erst dann als `SKIPPED` markiert wird, wenn der Prozess `P3` mit dem Ergebnis 0 endet (vgl. operationale Semantik zur Abwahl von Zuständen in Abschnitt 4.3). Rücksprünge müssen daher bei der Zyklensuche durchlaufen werden. Um das mehrfache Durchlaufen von Schleifen bei der Zyklensuche zu verhindern, muss der Durchlauf bei der letzten Transition des Rücksprungpfads (d.h. bei der in einen Hauptpfad des OLC einmündenden internen Transition) abgebrochen werden (vgl. interne Transition (`S2'`, (`P2'`, σ(P2')), `S1`) in OLC `A` aus Abbildung 4.23b). Durch diesen Mechanismus können verklemmungsfreie von nicht-verklemmungsfreien Zyklen unterschieden werden. So wird auch für die Prozessstruktur aus Abbildung 4.22b kein Pfad von Zustand `B.S1` zu Zustand `A.S1` gefunden, wenn die interne Schleife nicht (vollständig) durchlaufen wird.

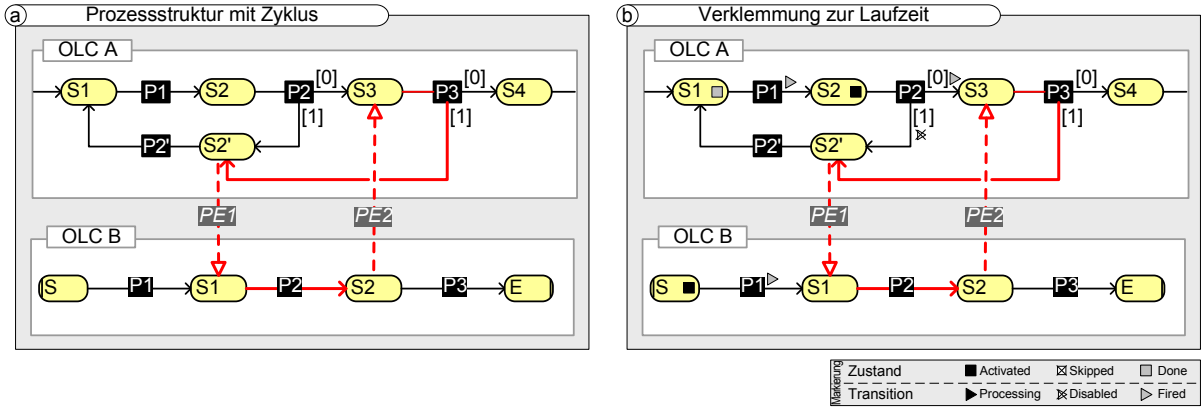


Abbildung 4.23: Zyklensuche in einer Prozessstruktur

Für die Zyklensuche in Prozessstrukturen definieren wir die Funktion $path_{ext}(s_0, s_1)$ (vgl. Definition 4.18). Sie liefert als Ergebnismenge die internen und externen *Transitionen* über die, beginnend bei s_0, der Zustand s_1 erreicht werden kann. Die Vorgehensweise entspricht dabei der von $path_{int}$ aus Algorithmus 4.1, wobei zusätzlich auch externe Transitionen durchlaufen werden. Die Anwendung der Funktion $path_{ext}(A.S1, B.S2)$ für die Prozessstruktur aus Abbildung 4.22a liefert zum Beispiel die Ergebnismenge { (`A.S1`, `PE1`, `B.S1`), (`B.S1`, (`P2`, σ(P2)), `B.S2`) }.

Definition 4.18 (Funktion $path_{ext}$)
Seien $ps = (OLC, EST)$ eine Prozessstruktur und $olc_i = (P_i, V_i, TS_i) \in OLC, i = 1, 2$ zwei Object Life Cycles mit Transitionssystemen $TS_i = (S_i, T_i, s_{start_i}, s_{end_i})$. Ferner seien $s_1 \in S_1$ ein Zustand in olc_1 und $s_2 \in S_2$ ein Zustand in olc_2. Die Funktion $path_{ext}(s_1, s_2)$ liefert eine Menge mit denjenigen internen und externen Transitionen, über die beginnend bei Zustand s_1 der Zustand s_2 erreicht werden kann. Die Vorgehensweise entspricht der von Algorithmus 4.1 (siehe Seite 59) mit dem Unterschied, dass externe Transitionen ebenfalls durchlaufen werden. Auf eine formale Erweiterung von Algorithmus 4.1 wird verzichtet.[a]

[a] Im Prototyp (siehe Kapitel 8) wurde der Algorithmus vollständig implementiert.

Die Zyklensuche für die Prozessstruktur aus Abbildung 4.22c kann durch Prüfung der externen Transition (`A.S2`, `Pe1`, `B.S1`) durch Anwendung des Algorithmus $path_{ext}(B.S1, A.S2)$ erfolgen.

Der Algorithmus liefert in diesem Fall die Transitionen { (B.S1, (P2, σ(P2)), B.S2), (B.S2, PE2, A.S1), (A.S1, (P2, σ(P2)), A.S2) } und damit einen Zyklus, der im Zusammenspiel mit der anderen externen Transition (B.S2, PE2, A.S1) erzeugt wird.

Basierend auf der Funktion $path_{ext}$ kann nun die automatische Zyklensuche durchgeführt werden und die dynamische Korrektheit einer Prozessstruktur sichergestellt werden. Wie bereits erwähnt, entsteht innerhalb der Prozessstruktur ein Zyklus, wenn vom Zielzustand einer externen Transition ein Pfad zu ihrem Quellzustand gefunden werden kann. Liefert die Funktion $path_{ext}$ für diese Suche kein Ergebnis, liegt auch kein verklemmender Zyklus vor. Alle Zyklen, die gefunden werden, sind hingegen korrektheitsgefährdend (vgl. Satz 4.3).

Satz 4.3 (Dynamische Korrektheit einer Prozessstruktur)
Sei $ps = (OLC, EST)$ eine Prozessstruktur. Sie ist dynamisch korrekt *(ihre Terminierung kann garantiert werden), wenn jeder Object Life Cycle $olc \in OLC$ korrekt ist (vgl. Definition 4.5) und ps keinen Zyklus enthält, der durch externe Transitionen verursacht wird (d.h. $\forall\, est = (s_1, p, s_2) \in EST : path_{ext}(s_2, s_1) = \emptyset$).*

Ein Beweis von Satz 4.3 findet sich in Anhang A.

Wie diskutiert, führt in Abbildung 4.22b die aus A.S1 ausgehende externe Transition nicht zu einer Verklemmung. Es entsteht allerdings beim Rücksprung in OLC A prinzipiell eine Situation, die eine mehrfache Aktivierung der externen Transition (A.S1, PE1, B.S1) hervorrufen kann: Die externe Transition ist als PROCESSING oder FIRED markiert, während ihr Quellzustand S1 durch die Reinitialisierung als NOTACTIVATED markiert wird. Derartige Situationen können auch in weiteren Szenarien auftreten, wie Abbildung 4.24 zeigt. Im Szenario aus Abbildung 4.24a hat ein Rücksprung keine Auswirkung auf die externe Transition. In den Szenarien aus den Abbildungen 4.24b und 4.24c kann die Anordnung des Rücksprungs jedoch zu einer (nachträglichen) Kontextänderung der externen Transition führen. Ebenso können Fälle auftreten, in denen sich der Quellzustand der externen Transition in einer Verzweigung befindet. So kann in Abbildung 4.24d der Quellzustand A.S2' der externen Transition zur Laufzeit zunächst abgewählt, in einem nachfolgenden Schleifendurchlauf jedoch wieder aktiviert werden.

Grundsätzlich führt keines der dargestellten Szenarien zu einer Korrektheitsgefährdung der Prozessstruktur. Die Ausführungsregel AR5 für externe Transitionen verhindert die Reinitialisierung der Markierung einer externen Transition bei einer (nachträglichen) Kontextänderung. Wird der Quellzustand einer externen Transition als ACTIVATED markiert und die externe Transition ist bereits in Ausführung oder gefeuert, wird sie **nicht** erneut als PROCESSING markiert und der Prozess auch nicht erneut gestartet (vgl. Ausführungsregel AR5). Stattdessen bleibt ihre Markierung erhalten. Die dynamischen Eigenschaften der Prozessstruktur sind damit weiterhin garantiert, es findet jedoch keine „tatsächliche“ Synchronisation mit dem Quellzustand mehr statt.[18]

[18] Zur Laufzeit muss in manchen Fällen die externe Transition nach einem Rücksprung und der Reinitialisierung ihres Quellzustands ebenfalls erneut ausgeführt werden. In Kapitel 7 stellen wir Konzepte vor, die die Behandlung derartiger Szenarien zur Laufzeit erlauben. Wichtig ist zu erwähnen, dass auch solche Eingriffe in den Ablauf unsere Aussagen zu dynamischer Korrektheit nicht gefährden, da sie durch geeignete Mechanismen behandelt werden (siehe Kapitel 7).

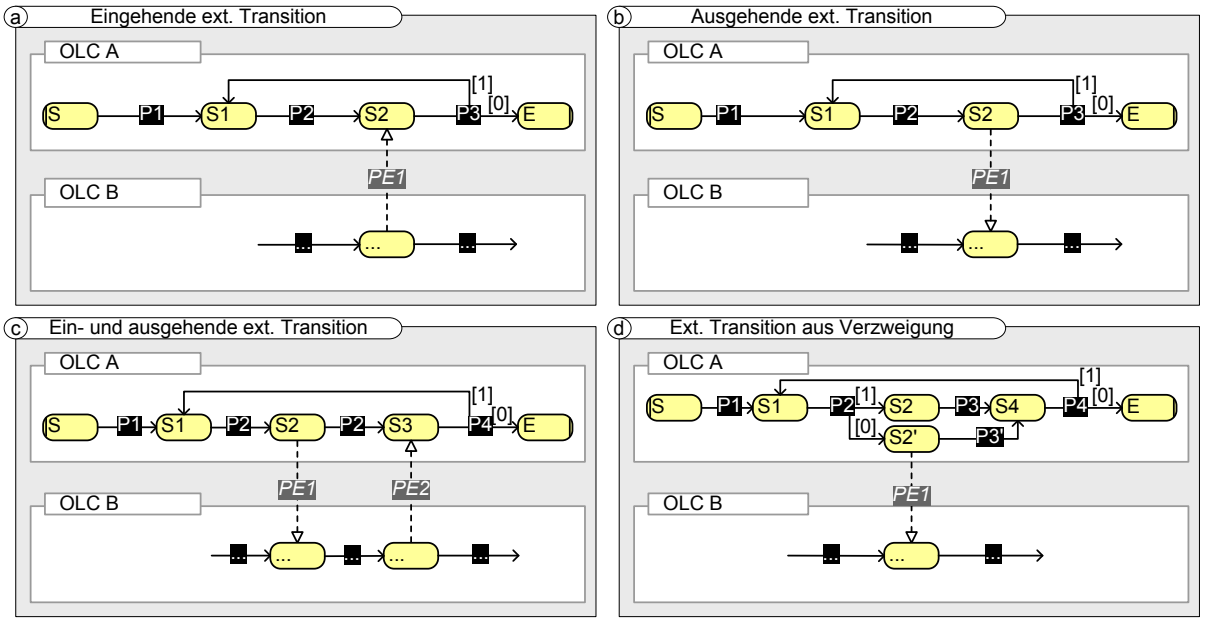


Abbildung 4.24: Anordnung von externen Transitionen in Verbindung mit Rücksprüngen

Damit der Modellierer die Möglichkeit hat, derartige Situationen zu erkennen und – falls notwendig – zu vermeiden (indem er die Prozessstruktur entsprechend anpasst), ist deren automatisierte Erkennung notwendig. Durch eine strukturelle Analyse der Prozessstruktur ist eine Aussage möglich, ob im „normalen Ablauf“ der Prozessstruktur eine solche Kontextänderung auftreten kann (vgl. Definition 4.19). Sie erlaubt, diejenigen externen Transitionen einer Prozessstruktur zu identifizieren, deren Quellzustand während der Ausführung mehrfach unterschiedlich markiert werden kann.

Definition 4.19 (Prüfung von Kontextänderungen in der Prozessstruktur)
Seien $ps = (OLC, EST)$ eine Prozessstruktur und $olc \in OLC$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$. Ferner sei $e^* = (s_1^*, p^*, s_2^*) \in EST$ eine externe Transition mit $s_1^* \in S$. Die Menge $ST := path_{int}(s_{start}, s_1^*)$ beschreibe diejenigen Zustände, die zwischen dem Startzustand von olc und dem Quellzustand von e^* liegen. Für die externe Transition e^* kann eine Kontextänderung auftreten, wenn gilt:

(a) Ihr Quellzustand ist direkt Teil einer Schleife (vgl. Abbildung 4.24c):

$$\exists\, t = (s, (p, v), targ) \in outTrans_{int}(s_1^*) : path_{int}(targ, s) \neq \emptyset$$

(b) Ihr Quellzustand ist Teil eines abwählbaren Pfades, dessen Beginn innerhalb einer Schleife liegt (vgl. Abbildung 4.24d):

$$\exists\, \hat{s} \in ST \;:\; \exists t = (s, (p, v), targ) \in outTrans_{int}(\hat{s}) : \\ path_{int}(targ, s) \neq \emptyset \wedge s_1^* \notin path_{int}(s, s_{end})^a$$

[a] Die Pfadsuche muss hierbei auf dem Hauptpfad des OLCs erfolgen. Hierfür ist eine geringfügige Anpassung von Algorithmus 4.1 notwendig. Wir verzichten an dieser Stelle auf die formale Beschreibung dieser Anpassungen und verweisen auf die konkrete Realisierung des Algorithmus in Anhang B.

4.8 Diskussion

In diesem Kapitel haben wir die Grundlagen geschaffen, um Objekte mit Prozessen zu verknüpfen und ihr Verhalten innerhalb einer Prozessstruktur zu beschreiben. Die Literatur dokumentiert verschiedene Ansätze für die Beschreibung von gekapselten Prozessen, die einem Objekt (oder einem System) zugeordnet sind. In diesem Abschnitt diskutieren wir Formalismen, die sich theoretisch für die Beschreibung von Objektlebenszyklen und die Beschreibung von Prozessstrukturen eignen.[19] Für die folgende Betrachtung unterscheiden wir Ansätze, die auf Petri-Netzen und Zustandsautomaten aufbauen, sowie Ansätze, die sich mit der Synchronisation von Prozessen im Allgemeinen beschäftigen.

4.8.1 Modellierung und Synchronisation von OLCs mittels Zustandsautomaten

In der Literatur finden sich verschiedene Konzepte, die sich mit der Beschreibung von gekapselten Abläufen und deren Synchronisation für ein Objekt oder einen Geschäftsvorfall befassen. Viele dieser Ansätze unterstützen unsere Auffassung, dass die zustandsbezogene Sicht auf Objektlebensläufe für Benutzer leicht und intuitiv verständlich ist (z.B. [Har87, Boo90, KS91, Obj03b]). Grundlage für die Modellierung von Objektlebenszyklen sind in der Regel die Konzepte endlicher Automaten. Sie beschreiben Zustände sowie Transitionen, die mit Bedingungen für Zustandsübergänge verknüpft sind.

Diese Methodik wurde durch die Definition von *State-* und *Acitivitycharts* erweitert, mit deren Hilfe sich das Verhalten eines Systems beschreiben lässt [Har87, HLN+90, HN96, HG97]. Über *Activitycharts* lassen sich Aktivitäten und Datenflüsse zwischen Aktivitäten beschreiben, während *Statecharts* die Kriterien für die Ausführung der Aktivitäten abbilden. Im Gegensatz zu einfachen Zustandsübergangsdiagrammen erlauben *Statecharts* auch die hierarchische Strukturierung und die Komposition von Zuständen (und damit Nebenläufigkeit). In [HG97] werden *Statecharts* genutzt, um direkt ausführbaren Code zu erzeugen.

Die Idee, das Verhalten eines Systems auf Basis seiner Zustände zu beschreiben, wird auch in der Modellierung von objektorientierten Software-Anwendungen genutzt. Hier beschreiben *Statecharts* das Verhalten von Objekten und werden unter anderem von *Object Oriented Design* [Boo90] und *Unified Modeling Language* (UML) [Obj03b] als geeigneter Modellierungsformalismus vorgeschlagen.

Zustands- und Aktivitätendiagramme werden auch im Projekt *MENTOR* (*Middleware for Enterprise-wide Workflow Management*) als Formalismus für die Spezifikation und Verifikation von partitionierten Prozessen verwendet [WKDM+95, WWWKD96]. Aktivitäten des Prozesses werden dabei als Aktivitätendiagramme modelliert, während Zustandsdiagramme die Ausführung der Aktivitäten koordinieren.

Eine formale Methode für die Synchronisation von Zustandsautomaten wird als *Team-Automata* in [Ell97, Kle03] beschrieben. Hier ist die Idee, die Beziehungen zwischen abhängigen und nebenläufigen Systemen durch Synchronisation *ihrer* Zustandsautomaten auszudrücken. Dafür werden

[19] Weitere Ansätze, die eine integrierte Betrachtung von Daten- und Prozessstrukturen vornehmen, werden in Kapitel 5 ausführlich diskutiert.

interne Transitionen mit *shared Actions* verknüpft. Sie erlauben die ereignisbasierte Synchronisation (vgl. Abschnitt 4.5.1). Die Ausführung einer *shared Action* führt dann zum Zustandsübergang in allen Automaten, die einen Übergang mit diesem Ereignis verknüpft haben. Die Modellierung von Synchronisationsprozessen wird damit jedoch nicht unterstützt.

Eine weitere Methode für die Integration von Statecharts wurde in [FE98] vorgestellt. Hier ist das Ziel, verschiedene Sichten auf die Zustände *eines* Objekts zu erhalten bzw. zu integrieren.

Das CoRepro-Metamodell orientiert sich an den Ansätzen zur zustandsbasierten Beschreibung von Systemverhalten, da diese für Nutzer leicht verständlich ist und das Vorgehen in der Produktentwicklung intuitiv widerspiegeln (vgl. Abschnitt 2.1). Des Weiteren erlauben Zustandsautomaten die gekapselte Beschreibung von Systemverhalten. Verglichen mit den genannten Ansätzen liegen die Stärken von CoRepro in der klaren Ausführungssemantik mit definierten Korrektheitskriterien sowie in der Kapselung der Abläufe in OLCs. Um die Anforderungen zur Ausführbarkeit erfüllen zu können, führt CoRepro Laufzeitmarkierungen ein. Über sie können klare Ausführungsregeln definiert und während der Ausführung eine Historie bereitgestellt werden, die für die später folgende Konsistenzanalyse im Rahmen dynamischer Änderungen wichtig ist. Zugunsten eines schlanken Metamodells und einer handhabbaren Komplexität der operationalen Semantik verzichten wir dafür auf einen Teil der Ausdrucksmächtigkeit von Statecharts, die beispielsweise in UML-Zustandsdiagrammen zum Einsatz kommen (z.B. History-Konnektor, Condition-Konnektor, temporale Logik sowie Entry-, Exit-, Throughout-Actions von Zuständen). Dafür bieten wir jedoch die Möglichkeit, erstellte Prozessstrukturen zu verifizieren und garantieren die Ausführbarkeit der erstellten Modelle. Des Weiteren nutzen wir für die Synchronisation von OLCs keine Events, sondern explizite Abhängigkeiten in Form externer Transitionen. Dies erlaubt das explizite Hinzufügen und Entfernen von Synchronisationsbeziehungen und deren Verknüpfung mit Synchronisationsprozessen. Damit können OLCs und deren Synchronisationen unabhängig modelliert und benutzerfreundlich dargestellt werden. Ferner bietet die explizite Modellierung externer Transitionen bei der Behandlung von abgewählten Pfaden Vorteile gegenüber ereignisbasierter Synchronisation, denn es kann eine explizite Deadpath-Eliminierung vorgenommen werden. Während die genannten Ansätze teilweise auch eine operationale Semantik beschreiben, bleiben dynamische Änderungen der erstellten Prozessstrukturen, wie sie von uns angestrebt werden (siehe Kapitel 6), außerhalb der Zielsetzung dieser Ansätze.

4.8.2 Modellierung und Synchronisation von OLCs mittels Petri-Netzen

Petri-Netze sind ein graphischer Formalismus zur Beschreibung von Automaten [Pet62, CKLY98]. Ein (statisches) Petri-Netz ist ein gerichteter, bipartiter Graph mit zwei Elementen: *Stellen* und *Transitionen*. Stellen beschreiben Bedingungen, wohingegen Transitionen Aktionen repräsentieren. Sie werden wechselseitig durch gerichtete Kanten miteinander verbunden. Marken repräsentieren den Fluss durch das Netz. Sie werden durch Transitionen von Stelle zu Stelle befördert. In einem Petri-Netz können sich mehrere Marken nebenläufig voneinander bewegen. Petri-Netze zeichnen sich durch ihren einfachen Aufbau, die zahlreichen Untersuchungen hinsichtlich Ausführungsverhalten und Korrektheit sowie verschiedene Erweiterungen (z.B. [Jen86]) aus. Die Anwendung von Petri-Netzen zur Modellierung von Prozessen wird unter anderem in [Des05] beschrieben und beispielsweise in *Workflow-Netzen* als Basisformalismus genutzt [Aal98, Wes07].

Die Ausdrucksmächtigkeit von Petri-Netzen ist sehr hoch. So können beispielsweise Zustandsautomaten durch Petri-Netze beschrieben werden [CKLY98, Des05]. Die Transitionen eines entsprechenden Petri-Netzes haben jeweils genau eine Vorbedingung und genau eine Nachbedingung. In ihm befindet sich nur eine Marke, die von Stelle zu Stelle transportiert wird. Es erlaubt damit, wie gefordert, die Abbildung von Sequenzen, Verzweigungen und Rücksprüngen. Abbildung 4.25 zeigt die Abbildung eines COREPRO-OLC als Petri-Netz.

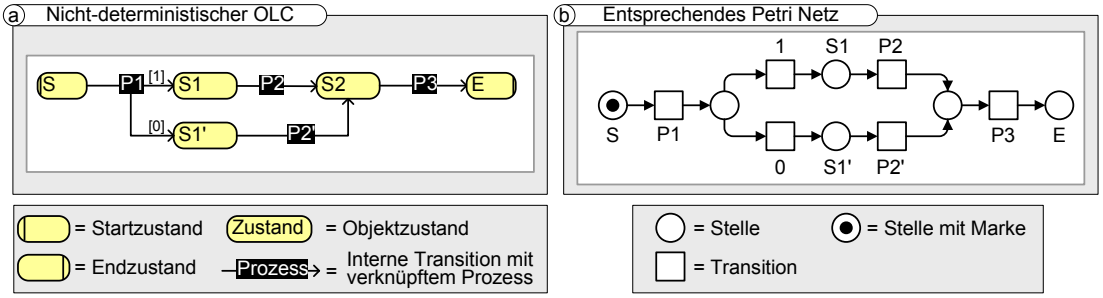


Abbildung 4.25: Abbildung eines OLC als Petri-Netz

Ein Formalismus für die Spezifikation des Verhaltens von Objekten auf Basis von Petri-Netzen wurde mit *Object Behavior Diagrams* eingeführt [KS91, PS98]. Ziel ist die Modellierung einer objektorientierten Software-Anwendung und deren spätere (manuelle) Implementierung. Dieser Ansatz stellt eine graphische Notation vor, mit der sich das Verhalten von Objekttypen in Form von *Life Cycles*[20] bestehend aus Zuständen und Aktivitäten beschreiben lässt. Die erstellten Modelle werden anschließend in Petri-Netze überführt, indem Zustände auf Stellen und Aktivitäten auf Transitionen abgebildet werden. Im Gegensatz zu Petri-Netzen, in denen Transitionen sofort schalten, sobald entsprechende Marken vorliegen, wird in Object Behavior Diagrams die mit der Transition verbundene Aktivität gestartet und auf ihre Beendigung gewartet, bevor die Marken befördert werden. Die Modellierung ermöglicht sowohl die Modellierung nicht-deterministischer zyklischer OLCs als auch die Abbildung von Nebenläufigkeit. Darüber hinaus erlaubt der Ansatz die Synchronisation unterschiedlicher OLCs. Die konkrete operationale Semantik der Synchronisationsbeziehungen wird jedoch nicht beschrieben. Damit bleibt beispielsweise unklar, wie eine Implementierung zur Laufzeit mit der Synchronisation von Zuständen in abgewählten Pfaden oder Rücksprüngen umgeht.

Ein weiterer Ansatz für die Beschreibung von Objektlebenszyklen mittels Petri-Netzen wird in [AB97] vorgestellt. Hier liegt der Fokus auf der Frage, wie sich Objektlebenszyklen und insbesondere Varianten von ihnen beschreiben und analysieren lassen. Der Ansatz unterstützt die Modellierung von Nebenläufigkeit in Objektlebenszyklen. Die Synchronisation zwischen Objekten wird hier jedoch in die auszuführenden Aktivitäten „ausgelagert“ und nicht direkt in Objektlebenszyklen realisiert. Die Prüfung auf dynamische Korrektheit durch eine Graphanalyse ist damit nicht realisierbar.

Die Vorteile von Petri-Netzen sind ihre formale Grundlage und ihre sehr gut entwickelte Theorie. Dadurch sind die Voraussetzungen für die Prüfung wichtiger Dynamikeigenschaften gegeben

[20] Nicht zu verwechseln mit COREPRO-OLCs.

[Aal97b]. Trotz dieser Vorteile sind Petri-Netze für die Beschreibung von OLCs im Kontext datengetriebener Prozessstrukturen nur bedingt geeignet. Insbesondere die in den betrachteten Ansätzen impliziten Synchronisationsbedingungen sowie die fehlende Historie und Kennzeichnung abgewählter Pfade erfüllen nicht die geforderten Kriterien. Theoretisch könnten wir auf Laufzeitmarkierungen bzw. die Historie verzichten und die operationale Semantik von OLCs basierend auf dem Verlaufsprotokoll beschreiben. Diese Vorgehensweise ist jedoch aufwändig, wenig intuitiv und erfordert zudem eine komplizierte Beschreibung von Ausführungsregeln. Die Markierung von Zuständen erlaubt hingegen schnelle Aussagen zur Historie eines OLC.[21] Die Markierung von Transitionen hat weiter den Vorteil, dass die Markierung von abgewählten Pfaden durch einfache Regeln beschrieben werden kann.

4.8.3 Weitere Ansätze

Das Prozess-Metamodell von *Teamcenter Engineering* (vgl. Abschnitt 3.1.3) erlaubt die Modellierung nicht-deterministischer Prozesse mit nebenläufiger Anordnung von Aktivitäten (Tasks). Aktivitäten lassen sich hierarchisch schachteln. Die Prozess-Komponente von *Teamcenter Engineering* unterstützt die Ausführung modellierter Prozesse und nutzt hierfür Laufzeitmarkierungen. Mit Aktivitäten und internen Übergängen der Laufzeitmarkierungen lassen sich sogenannte Action- und Rule-Handler verknüpfen. Verschiedene Prozesse lassen sich über Rule-Handler synchronisieren (*check-process-completion*) [TcE06]. Bedingte Verzweigungen sind ebenfalls über Rule-Handler realisierbar. Die Modellierung von Datenflüssen wird nicht unterstützt. Stattdessen werden Prozesse bei deren Instanziierung mit Objekten (bzw. Objektversionen) der Produktstruktur (sog. *Target*) verknüpft und die Aktivitäten des Prozesses dann auf diesem Objekt ausgeführt. Die Nutzung verschiedener Objekte in einem Workflow kann nur über Workarounds erfolgen. *Teamcenter Engineering* erlaubt zwar auch die Ausführung der modellierten Prozesse, verfügt jedoch nicht über ausreichende Korrektheitsprüfungen (vgl. Abschnitt 3.1.3).

ADEPT unterstützt die aktivitätenorientierte Modellierung und Ausführung von Prozessen (vgl. Abschnitt 3.1). Das Metamodell erlaubt die Definition von nicht-deterministischen und hierarchisch geschachtelten Prozessen. Zur Laufzeit erfolgt die Markierung der Elemente entsprechend des Fortschritts des Prozesses. In ADEPT [RD98, Rei00, RPB03, RRD04b, RRKD05, DRRM$^+$09, DR09] kann zum Beispiel ein Sub-Prozess die Bearbeitungszustände *Not_ Activated*, *Activated*, *Started*, *Suspended*, *Completed* und *Failed* annehmen, wobei insbesondere die Bearbeitungszustände *Not_ Activated*, *Activated* und *Completed* für die Prozesssteuerung relevant sind. Der Ansatz bietet eine umfangreiche Prüfung der Korrektheit von Kontroll- und Datenflüssen. Das Metamodell beschränkt allerdings die für OLCs relevanten Abläufe auf Blockstrukturen (mit Ausnahme von Synchronisationskanten). Die Kapselung der Abläufe, wie sie zur Abbildung von OLCs notwendig ist, wird nicht unterstützt.

AHEAD [Wes01, KKSW98, JSW99] bzw. dessen Aufgabenmodell DYNAMITE (*Dynamic Task Nets*) erlaubt die Beschreibung von Prozessen mit Kontroll- und Datenflüssen. Aufgaben können hierarchisch geschachtelt werden. Zusätzlich zum normalen Kontrollfluss, der grundsätzlich keine Rücksprungkanten aufweist, enthält das Metamodell noch informelle Beziehungen (Feedback-Kanten), mit denen sich Rückgriffe und Zyklen im Aufgabennetz beschreiben lassen. DYNAMITE

[21] Diese Eigenschaft ist insbesondere für die Konsistenzprüfung im Rahmen der dynamischen Adaption und der Ausnahmebehandlung relevant.

bzw. AHEAD lassen jedoch in der Kontrollflussmodellierung nur Sequenzen und Parallelitäten zu. Bedingte Verzweigungen sind gar nicht möglich; Schleifen können zwar über Rückgriffbeziehungen realisiert werden, sie erlauben jedoch einzig einen Informationsfluss und haben keine operationale Funktion. Aufgaben können hierarchisch geschachtelt werden, wobei jede Aufgabe einen Lebenszyklus besitzt. Das Verhalten der Aufgaben, also deren interne Abläufe, wird durch die verschiedenen möglichen Zustände einer Aufgabe in Zustandsübergangsdiagrammen modelliert. Dabei wird zunächst ein allgemeiner Ablauf bei der Bearbeitung einer Aufgabe spezifiziert, der später in einem Zustandsdiagramm noch durch Änderung der Bedingungen für Zustandsübergänge, Reaktionen auf die Ausführung von Operationen und Aktionen beim Betreten und Verlassen des Zustandes angepasst werden kann. Die Korrektheit für die erstellen Prozessstrukturen ist nicht formal zugesichert. Der Ansatz erfüllt damit die Anforderungen an die Ausdrucksmächtigkeit und Korrektheit abgebildeter Prozessstrukturen nicht.

COO-Flow ist ein Ansatz für die optimierte Koordination von Prozessen [GCG04]. Unabhängig von einem konkreten Metamodell wird mit *Anticipation* ein Konzept vorgeschlagen, das die traditionelle kontroll- bzw. datenflussorientierte Ausführung von Prozessen flexibler macht. *Anticipation* erlaubt den vorzeitigen Start einer Aktivität, auch wenn noch nicht alle Eingangsvoraussetzungen erfüllt sind. Es ermöglicht damit die Parallelisierung von Aktivitäten, die im Kontrollfluss eigentlich sequentiell modelliert sind, unter der Annahme, dass kreative Aktivitäten häufig nicht bereits zu Beginn alle Eingangsdaten benötigen. Hierfür werden spezielle interne Laufzeitmarkierungen eingeführt (*ready_to_anticipate* und *anticipating*), die der eigentlichen Ausführung „vorgeschaltet“ werden. Es ist damit möglich, eine Aktivität als *ready_to_anticipate* und auch bereits mit Markierung *anticipating* zu starten, bevor die eigentliche *active*-Markierung erfolgt (nämlich dann, wenn die Eingangsvoraussetzungen vollständig erfüllt sind). Die logische Reihenfolge des Ablaufs bleibt im Sinne der Beendigung der Aktivität (Ende-Ende Beziehung zwischen aufeinanderfolgenden Aktivitäten) erhalten. Ein Vorteil von COO-Flow ist die Möglichkeit, Aktivitäten vorzeitig zu starten. Eine ähnliche Eigenschaft bietet auch der Case-Handling Ansatz [AB01b, AWG05]. Hier werden im Prinzip Datenflüsse modelliert (vgl. Abschnitt 3.1.4). Liefert eine Aktivität A „vorzeitig“ ein Datenobjekt, dann können nachfolgende Aktivitäten bereits gestartet werden, auch wenn A noch nicht beendet wurde.

Im Bereich der Prozesssynchronisation existieren noch weitere Ansätze, die sich mit der Spezifikation von Prozess-Abhängigkeiten, unabhängig vom zugrundeliegenden Metamodell, beschäftigen. Aspekte der unabhängigen Synchronisation von Prozessen mittels sogenannter Interaktionsausdrücke und -graphen werden in [Hei01] beschrieben. Das Konzept stellt sicher, dass die beschriebenen Synchronisations- oder Integritätsbedingungen zur Laufzeit von den nebenläufig ausgeführten Prozessen auch eingehalten werden. Der Ansatz ist sehr mächtig, erfüllt aber nicht die praktischen Anforderungen hinsichtlich der Unterstützung von Synchronisationsprozessen und Flexibilität. *Prozessalgebren* erlauben ebenfalls die Beschreibung von Prozessstrukturen und die Synchronisation der enthaltenen Prozesse. Algebraische Gesetze ermöglichen die Analyse und Umformung von Prozessbeschreibungen. Ein Vertreter der Prozessalgebren ist die Programmiersprache für kommunizierende Prozesse *Communicating Sequential Processes* (CSP) [Hoa78]. Sie erlaubt die Beschreibung nicht-deterministischer Prozesse und deren Interaktionen über sogenannte *Kanäle* (*Channels*). Mithilfe von CSP lassen sich Automaten beschreiben, indem Abhängigkeiten zwischen Ereignissen und Zuständen formuliert werden. Das π-Kalkül ist ein weiterer Vertreter der Prozessalgebren [Mil99]. Es erlaubt ebenfalls die nebenläufige und nichtdeterministische Koordination von Prozessen. Im Mittelpunkt stehen hier Prozesse, die durch

einfache Namen repräsentiert werden sowie Kanäle, über die Nachrichten ausgetauscht werden. Damit lassen sich die Prozessstrukturen unseres Ansatzes im Kern repräsentieren. Die entstehenden Ausdrücke werden jedoch bereits in kleinen Prozessstrukturen sehr komplex [AHD05]. Prozessalgebren bieten daher zwar einen theoretisch fundierten und mächtigen Formalismus, der in der Praxis aber aufgrund der Komplexität kaum Anwendung findet.

Weitere Ansätze aus der Praxis beschäftigen sich mit der Beschreibung von Web-Service-Choreographien (z.B. [Wor05]). Sie beschreiben die Synchronisation von Web-Services durch Modellierung der Abfolge von Nachrichten zwischen ihren öffentlichen Schnittstellen. Für die Abbildung datengetriebener Prozessstrukturen essentielle Anforderungen wie Synchronisationsprozesse, Verifikation und dynamische Änderungen haben bei diesen Ansätzen bisher jedoch noch keine Beachtung gefunden.

Die vorgestellten weiteren Ansätze erfüllen die Anforderungen an die Unterstützung datengetriebener Prozessstrukturen nur unzureichend. Sie weisen insbesondere Schwächen hinsichtlich Strukturierbarkeit, Wiederverwendung, Änderbarkeit und Nachvollziehbarkeit auf. Die Unterstützung von Synchronisationsprozessen und dynamischer Adaptierbarkeit werden häufig nicht oder in anderen Kontexten betrachtet. So wird beispielsweise der vorgezogene Start eines Prozesses, wie er von *COO-Flow* unterstützt wird, erst dann notwendig, wenn die Prozesse nicht feingranular genug geschnitten sind und eine Reduktion auf Ende-Start-Beziehungen nicht möglich ist. Diese Anforderung ist für datengetriebene Prozessstrukturen nicht gegeben. Da OLCs in CoRepro nicht blockstrukturiert sein müssen, lässt sich diese Eigenschaft durch geeignete Anwendung externer Transitionen abbilden [MRH07].

4.9 Zusammenfassung

Die Koordination datengetriebener Prozessstrukturen stellt hohe Anforderungen an die Ausdrucksmächtigkeit, Strukturierbarkeit, Wiederverwendung, Ausführbarkeit und Korrektheit des zugrundeliegenden Metamodells bzw. modellierter Prozessstrukturen. In diesem Kapitel haben wir die Grundlagen für die Abbildung großer Prozessstrukturen entwickelt und damit die in Abschnitt 4.1.3 gestellten Anforderungen erfüllt. Unser Basisformalismus unterstützt die Beschreibung von *Object Life Cycles* und deren Integration in Prozessstrukturen. Der eingeführte Ansatz unterscheidet sich von bestehenden, indem er die zustandsbasierte Synchronisation von OLCs bzw. der mit ihnen verknüpften Prozesse ermöglicht und damit eine einfache und zugleich robuste Modellierung von Abhängigkeiten erlaubt. Außerdem definiert CoRepro nicht nur eine vollständige operationale Semantik zur Ausführung solcher Prozessstrukturen, sondern erlaubt auch präzise Aussagen bezüglich ihrer Korrektheit. Der eingeführte Basisformalismus bildet die Grundlage für die in den folgenden Kapiteln entwickelten Lösungskonzepte.

Im Rahmen der Ausarbeitung des Basismodells ergaben sich weitere Fragestellungen, die in diesem Kapitel zugunsten der Verständlichkeit zunächst ausgeklammert wurden:

- In Abschnitt 4.2 haben wir internen Transitionen eine Verzweigungsbedingung in Form von Einzelwerten zugewiesen. Unser Formalismus lässt sich jedoch problemlos auf die Verzweigung basierend auf (disjunkten) Ergebnismengen erweitern. Der besseren Lesbarkeit halber haben wir auf die Darstellung von Ergebnismengen verzichtet.

- In der Praxis kann es vorkommen, dass nicht alle möglichen Ergebnisse eines Prozesses bekannt sind. Hier kann die Definition einer *Standardtransition* je Verzweigung Sinn ergeben, die immer dann feuert, wenn das Prozessergebnis des verknüpften Prozesses durch keine andere Transitionsbedingung abgedeckt ist. Dies würde in der Praxis bei der Definition der Transitionsbedingungen die vollständige Abdeckung der möglichen Prozessergebnisse eines Prozesses vereinfachen. Die Angabe einer Standardtransition bei bedingten Verzweigungen könnte auch für die Berücksichtigung unvorhergesehener Prozessergebnisse (beispielsweise hervorgerufen durch einen fehlerhaft implementierten Prozess) genutzt werden. Ein derartiger Mechanismus kann leicht in die Konzepte integriert werden. Die statischen und dynamischen Eigenschaften von OLCs bzw. Prozessstrukturen sind davon nicht betroffen.
- Die operationale Semantik für zyklische OLCs führt beim erneuten Durchlaufen einer Schleife zu einer erneuten Instanziierung der mit ihren internen Transitionen verknüpften Prozesse. Die bereits ausgeführten Prozesse werden ab diesem Zeitpunkt nicht mehr berücksichtigt. Abhängig vom jeweiligen ausführenden Prozess-Management-System der einzelnen Prozesse, kann mit der Reinitialisierung noch die Durchführung weiterer Aktionen (z.B. zur Archivierung) gewünscht sein. Dieser Aspekt ist durch eine Erweiterung der Regeln für Prozesszustände (vgl. Abschnitt 4.3.2) auf Basis von Ausführungsregel AR4 und Markierungsregel MR4 leicht zu integrieren.
- In der Praxis können Rücksprungkanten in verschiedenen Kontexten genutzt werden. Zum einen können damit Schleifen modelliert, zum anderen können *geplante Rücksprunge* als Reaktion auf fachliche Ausnahmen angelegt werden. Ein geplanter Rücksprung kann im Gegensatz zur Schleife die Anwendung von *Rollback-Aktionen* oder *Kompensationsprozessen* erfordern [LR00]. Dies wird durch COREPRO ermöglicht, indem Rücksprungkanten nicht direkt in den Zielzustand eingehen müssen, sondern auch hier (nicht-blockstrukturierte) Rücksprungpfade mit der Ausführung weiterer Prozesse modelliert werden können.

5

Modellierung, Erzeugung und Ausführung datengetriebener Prozessstrukturen

Kapitel 4 beschreibt die Grundlagen des COREPRO-Ansatzes zur Abbildung und Koordination datengetriebener Prozessstrukturen. Die Komplexität der Modellierung wurde dabei noch ausgeklammert. Sie resultiert aus der Größe der Prozessstrukturen und dem damit verbundenen Modellierungsaufwand. Unser Ziel ist es, in diesem Kapitel eine geeignete Modellierungsunterstützung zu entwickeln. Sie reduziert den Modellierungsaufwand signifikant, wenn die Abhängigkeiten zwischen Datenstrukturen und zugehörigen Prozessstrukturen konsequent ausgenutzt werden. Zu diesem Zweck führen wir ein generisches Datenmodell ein, dessen Komponenten sich mit Bausteinen der Prozessstruktur (d.h. mit OLCs und externen Transitionen) in Beziehung setzen lassen. Ferner stellen wir einen Mechanismus für die automatische Erzeugung einer datengetriebenen Prozessstruktur vor.

Das Kapitel gliedert sich wie folgt. Abschnitt 5.1 stellt den fachlichen Hintergrund eines Anwendungsbeispiels vor und diskutiert entlang dieses Beispiels charakteristische Anforderungen an die Modellierungsunterstützung datengetriebener Prozessstrukturen. Insbesondere gehen wir auf den engen Zusammenhang ein, der zwischen einer Datenstruktur auf der einen Seite und der damit assoziierten Prozessstruktur auf der anderen Seite besteht. Abschnitt 5.2 beschreibt den formalen Aufbau einer *Datenstruktur* in COREPRO. Sie bildet die Grundlage für die Erzeugung datengetriebener Prozessstrukturen. Abschnitt 5.3 stellt die Konzepte zur Modellbildung einer datengetriebenen Prozessstruktur vor und geht auf die automatische Erzeugung konkreter Instanzen einer Prozessstruktur ein. Abschnitt 5.4 diskutiert Ausführungsaspekte und die geforderten Korrektheitseigenschaften erzeugter Prozessstrukturen. Abschnitt 5.5 vergleicht den vorgestellten Ansatz mit verwandten Arbeiten und Abschnitt 5.6 gibt schließlich eine Zusammenfassung des Kapitels.

5.1 Einleitung

Die Modellierung großer Prozessstrukturen zur Koordination hunderter bis tausender Einzelprozesse erfordert eine geeignete Modellierungsunterstützung (vgl. Abschnitt 2.2.2). Unser Anwendungsbeispiel aus Kapitel 2, welches das Release-Management in der Fahrzeugentwicklung adressiert, hat darüber hinaus gezeigt, dass in dieser Domäne häufig Prozessstrukturen für unterschiedliche Systemstrukturen erstellt werden müssen. So können sich die zugehörigen Komponenten und die Beziehungen zwischen ihnen von Release zu Release sowie, orthogonal dazu, von Baureihe zu Baureihe unterscheiden. Dies hat Einfluss auf die Prozessstruktur, denn sie soll nur diejenigen Prozesse und Abhängigkeiten umfassen, die für die jeweiligen Komponenten der Systemstruktur notwendig sind.

Das CoRePro-Rahmenwerk hat zum Ziel, die Modellierung *großer* Prozessstrukturen zu unterstützen. Der in Kapitel 4 vorgestellte Basisformalismus hilft, die Prozesse für einzelne Komponenten in Object Life Cycles (OLCs) zu kapseln. Damit können entsprechend der Komponenten einer Produktstruktur vollständige OLCs anstatt einzelner Prozesse in die Prozessstruktur eingefügt werden. Die Modellierung hunderter OLCs und deren Synchronisation mittels externer Transitionen bleiben dennoch mit hohem Aufwand verbunden und erfordern eine geeignete Modellierungsunterstützung. Wir stellen im Folgenden zunächst den fachlichen Hintergrund des Anwendungsbeispiels vor und motivieren damit die Anforderungen an die Modellierung datengetriebener Prozessstrukturen. Anschließend skizzieren wir das CoRePro-Modellierungskonzept.

5.1.1 Fachlicher Hintergrund des Anwendungsbeispiels

Die Analyse der Release-Management-Prozesse der Fahrzeugentwicklung für elektronische Systeme hat gezeigt, dass die Objektlebenszyklen verschiedener Komponenten häufig identisch sind. Die in dem Umfeld zunehmende Standardisierung von Prozessen (z.B. durch Anwendung des V-Modells, der DIN EN 61508 oder der ISO 26262 [Deu99, Int08]) unterstreicht diese Beobachtung.[1] Von dieser Erkenntnis können wir im Kontext der Modellierung profitieren. So kann ein Object Life Cycle für verschiedene Komponenten des gleichen Typs wiederverwendet werden. Er muss dann mehrfach instanziiert werden, um individuelle Laufzeitmarkierungen für jede Komponente zu erhalten. Bei der technischen Ausgestaltung der Wiederverwendung können wir uns an Ansätzen des Prozess-Managements orientieren, die zwischen Prozessschemas (oder Prozessmodellen) und Prozessinstanzen unterscheiden (vgl. Abschnitt 1.1). Prozessmodelle können zur Laufzeit beliebig oft instanziiert und ausgeführt werden [Wor94, RD98, Rei00, Wes07].

Ein weiterer wichtiger Aspekt ist die Synchronisation von Zuständen (bzw. Prozessen) unterschiedlicher Objektlebenszyklen. Die Analyse der Release-Management-Prozesse hat gezeigt, dass Abhängigkeiten zwischen Objektlebenszyklen häufig durch Beziehungen zwischen entsprechenden Objekten in der Systemstruktur impliziert werden [BHR05, MHHR06]. Sind etwa einem System (z.B. `Navigation` in Abbildung 5.1) mehrere Subsysteme über eine `hatSubsystem`-Beziehung in der Systemstruktur zugeordnet (vgl. `GPS Sensor` und `Display`), müssen auch deren Prozesse entsprechend synchronisiert werden (z.B. Eingangstest der Subsysteme *vor* dem Labortest des

[1] Diese Beobachtung kann auch in anderen technischen Bereichen gemacht werden, z.B. in der Luftfahrt mit den Prozessstandards DO-178b und DO-254 zur Software- bzw. Hardware-Entwicklung.

Systems in Abbildung 5.1). Ist also bekannt, welche Beziehungen zwischen zwei Objekten bestehen, können damit Rückschlüsse auf die notwendige Synchronisation ihrer Objektlebenszyklen geschlossen werden. Die Systemstruktur, bestehend aus Objekten und Objektbeziehungen, liefert somit genügend Informationen, um eine passende Prozessstruktur, bestehend aus OLCs und externen Transitionen, aufzubauen. Voraussetzung hierfür ist ein ausreichend mächtiges Modell der datengetriebenen Prozessstruktur.

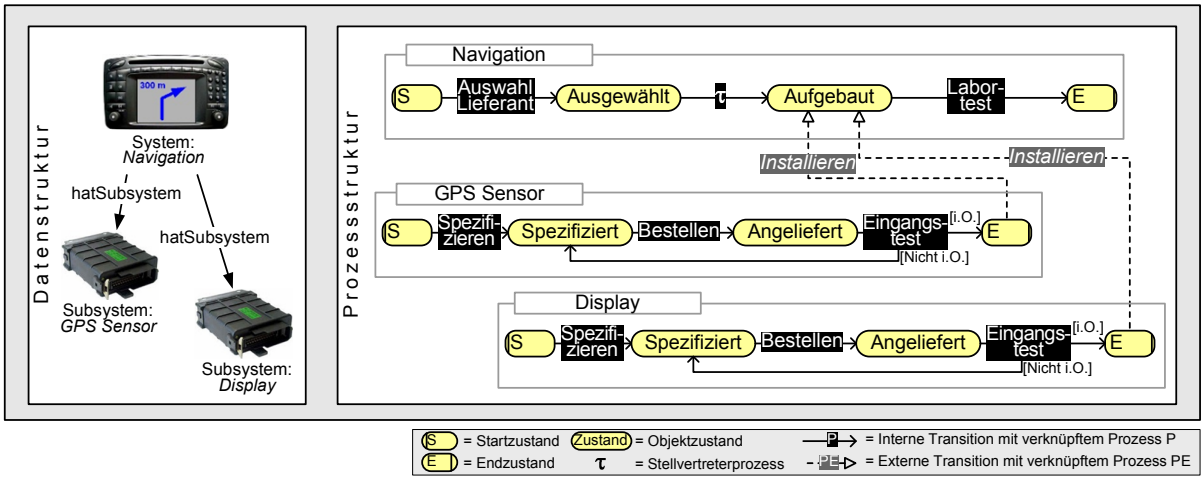


Abbildung 5.1: Systemstruktur und zugehörige Prozessstruktur

In vielen technischen Domänen werden Produktdaten-Management-Systeme für die Verwaltung von Produkt- bzw. Systemstrukturen eingesetzt [CAPD03, KS04]. So werden im Rahmen des Release-Managements bereits Systemstrukturen mit ihren Komponenten (z.B. Systeme und Subsysteme) angelegt und verwaltet. Ähnlich wie bei der objektorientierten Software-Entwicklung wird in Produktdaten-Management-Systemen zwischen Klassen- und Instanzebene unterschieden, indem sich Komponentenklassen beschreiben und anschließend konkret anlegen lassen (vgl. Abschnitt 3.1.3). Damit ist die erforderliche Information zum Aufbau einer Prozessstruktur vorhanden. Sie kann genutzt werden, indem Komponentenklassen mit entsprechenden Objektlebenszyklen, sowie Komponentenbeziehungen mit Synchronisationsbeziehungen zwischen den Objektlebenszyklen verknüpft und anschließend instanziiert werden. Die Zielsetzung der Modellierung liegt daher nicht in der Beschreibung der internen Semantik von Produktstrukturen (z.B. unterschiedliche Formen der Beziehung), sondern in der integrierten Betrachtung von Daten- und Prozessstrukturen.

5.1.2 Anforderungen

Die Analyse der fachlichen Beziehungen zwischen Produkt- und Prozessstruktur (vgl. Abschnitt 5.1.1) hat gezeigt, dass eine Prozessstruktur auf Basis einer gegebenen Systemstruktur aufgebaut werden kann. Diese Erkenntnis hilft, den Modellierungsaufwand bei der Erzeugung von Prozessstrukturen signifikant zu reduzieren. Unter diesen Rahmenbedingungen lassen sich die folgenden Anforderungen an die konzeptionelle Unterstützung der Modellierung datengetriebener Prozessstrukturen ableiten.

Anforderung 5.1 (Integration von Produkt- und Prozessstrukturen)
Die fachliche Betrachtung datengetriebener Prozessstrukturen hat gezeigt, dass durch eine integrierte Betrachtung von Produkt und Prozessen der Modellierungsaufwand stark verringert werden kann. Der Modellierer muss die Möglichkeit haben, einen individuellen *Bauplan* für eine produkt- bzw. datengetriebene Prozessstruktur zu modellieren, um diesen als Basis für die Erzeugung datengetriebener Prozessstrukturen zu nutzen. Hieraus ergibt sich die Anforderung, ihm ein generisches Metamodell für die Beschreibung der Beziehung zwischen Produkt- und Prozessstruktur zur Verfügung zu stellen. Das Modell soll die Verknüpfung von Elementen der Produktstruktur mit Elementen der Prozessstruktur erlauben.

In Produktdaten-Management-Systemen sind Modelle von Produkt- bzw. Systemstrukturen im Allgemeinen sehr komplex, da sie eine Vielzahl an Informationen der verwalteten Objekte bereitstellen (z.B. physikalische Eigenschaften einer E/E-Komponente). Ferner erlauben sie die Spezifikation einer internen Aufbaulogik der Systemstrukturen (z.B. komplexe Datenbedingungen mit Versionierung und Variantenmanagement) [CAPD03]. Eine Prozessstruktur bearbeitet allerdings häufig nur einen Teil dieser Systemstruktur. Die zum Aufbau von Prozessstrukturen relevanten Produktstrukturen sind dementsprechend *Sichten* auf komplexe Systemstrukturen in einem Produktdaten-Management-System. Wir bezeichnen die für den Aufbau einer Prozessstruktur relevante Sicht im Folgenden als *Datenstruktur*.

Ziel von COREPRO ist es nicht, die Funktionalität eines Produktdaten-Management-Systems nachzubilden. Es ist jedoch notwendig, die für datengetriebene Prozessstrukturen relevanten Teile einer Systemstruktur in entsprechenden Datenstrukturen geeignet abbilden zu können. Damit dies unabhängig vom Metamodell des Produktdaten-Management-Systems geschehen kann, sollen Datenstrukturen einen generischen Aufbau mit gerichteten, binären und benannten Relationen zwischen Objekten aufweisen. Dies soll die Nutzung der Beziehungskonzepte (z.B. Versionierung, Assoziation, Generalisierung, Vererbung) innerhalb einer Systemstruktur nicht einschränken, sondern deren einfache Abbildung ermöglichen. Ein ähnliches Konzept ist auch mit UML2-Objektdiagrammen realisiert, die die Modellierung eines *Schnappschusses* einer instanziierten Klassenstruktur zur Laufzeit erlauben [Obj03b, JRH+04].

Anforderung 5.2 (Generierung von Prozessstrukturen)
Durch Eingabe einer gegebenen Datenstruktur soll mit dem generischen Bauplan aus Anforderung 5.1 eine datengetriebene Prozessstruktur automatisch erzeugt werden können. Dabei sollen auf Basis der in der Datenstruktur vorhandenen Objekte die verknüpften OLCs der Prozessstruktur hinzugefügt und aufgrund der Beziehungen zwischen diesen Objekten externe Transitionen zwischen den entsprechenden OLCs eingefügt werden.

Anforderung 5.3 (Korrektheit generierter Prozessstrukturen)
Um eine korrekte Ausführung großer Prozessstrukturen zu gewährleisten, müssen die in Kapitel 4 eingeführten Korrektheitseigenschaften zugesichert werden. Dabei soll der Benutzer die erzeugten Prozessstrukturen nicht manuell überprüfen müssen, sondern eine geeignete IT-Unterstützung zur Analyse der Prozessstruktur erhalten. Die Analyse einer großen Prozessstruktur ist jedoch äußerst komplex. Ziel muss es daher sein, den erforderlichen Analyseaufwand auf ein Minimum zu reduzieren. Die Korrektheit datengetriebener Prozessstrukturen soll daher weitestgehend *per Konstruktion*, also durch das Modellierungskonzept, sichergestellt werden.

5.1.3 Modellierungskonzept

Um die Erkenntnisse aus Abschnitt 5.1.1 auch tatsächlich für die Modellierungsunterstützung nutzen zu können, muss die Prozessstruktur geeignet beschrieben und anschließend auf Basis einer gegebenen Datenstruktur aufgebaut werden. Dieser Abschnitt führt das CoPro -Modellierungskonzept auf informeller Ebene ein.

Um eine hohe Flexibilität in der Modellierung unter Beibehaltung der Trennung von Daten und Prozessen zu erreichen, orientieren wir uns an dem Ansatz zur Entwicklung modellgetriebener Architekturen (*Model Driven Architecture*, MDA). MDA ist ein Ansatz aus der Software-Entwicklung, der die Spezifikation von Systemen zum Ziel hat [Obj03a]. Dabei werden drei Ebenen unterschieden. Das unabhängige Metamodell *Computation Independent Model* (CIM) definiert diejenigen Komponenten, mit denen ein domänenspezifisches, aber plattformunabhängiges Modell (*Platform Independent Model*, PIM) erstellt werden kann. Basierend auf dem PIM können dann konkrete, d.h. plattformabhängige ausführbare Modelle abgeleitet werden (*Platform Specific Models*, PSMs).

Die Unterscheidung zwischen Metamodell-, Modell- und Instanzebene lässt sich auch für datengetriebene Prozessstrukturen anwenden. Zur Beschreibung von Datenstrukturen stellen wir ein Metamodell zur Verfügung, das sich für die Erstellung konkreter *Datenmodelle* bestehend aus *Objekttypen* und *Relationstypen* eignet. In Abbildung 5.2a liegt ein Datenmodell mit den beiden Objekttypen `System` und `Subsystem` vor. Sie sind durch die Relationstypen `hatSubsys` und `nutztSubsys` verbunden.[2]

Objekt- und Relationstypen können anschließend mit den Elementen einer Prozessstruktur (d.h. OLCs und externe Transitionen) in Beziehung gesetzt werden (vgl. Anforderung 5.1). Dies geschieht im *Life Cycle Coordination Model (LCM)*. Dazu wird für jeden Objekttyp der entsprechende OLC definiert. Im Beispiel aus Abbildung 5.2b sind das die OLCs für die Objekttypen `System` und `Subsystem`. Sie werden weiter durch externe Transitionen synchronisiert. Um die Verbindung zu den Beziehungen innerhalb der Systemstruktur herzustellen (vgl. Abschnitt 5.1.1), werden die externen Transitionen mit den Relationstypen `hatSubsys` bzw. `nutztSubsys` verknüpft. Wir betrachten die Beziehungsbezeichnung und deren Richtung im Datenmodell immer als eindeutig. Die Richtung der Relation ist bei der Verknüpfung mit externen Transitionen ebenfalls implizit gegeben, da externe Transitionen über eindeutige Quell- und Zielzustände definiert sind. Zusammengefasst beschreibt das LCM die Semantik des Datenmodells und den Bauplan für die automatisch erzeugbare Prozessstruktur (vgl. Abbildung 5.2b).

Die angelegten Modelle können nun instanziiert werden. Zur Instanziierung des Datenmodells lassen sich, basierend auf seinen Objekt- und Relationstypen, konkrete Objekte und Relationen[3] anlegen. Instanzen eines Datenmodells werden in CoPro als *Datenstrukturen* bezeichnet. Während Datenstrukturen manuell angelegt werden (bzw. aus einem vorhandenen Produktdaten-Management-System exportiert werden; vgl. Anforderung 5.1), sollen Instanzen des LCM (d.h. konkrete Prozessstrukturen) auf Basis einer gegebenen Datenstruktur automatisch erzeugt werden können (vgl. Anforderung 5.2).

[2] Das Datenmodell ist generisch. Abhängig vom Anwendungsfall sind beliebige Objekt- und Relationstypen abbildbar.

[3] Ein Relationstyp kann in CoPro höchstens einmal zwischen zwei Objekten instanziiert werden.

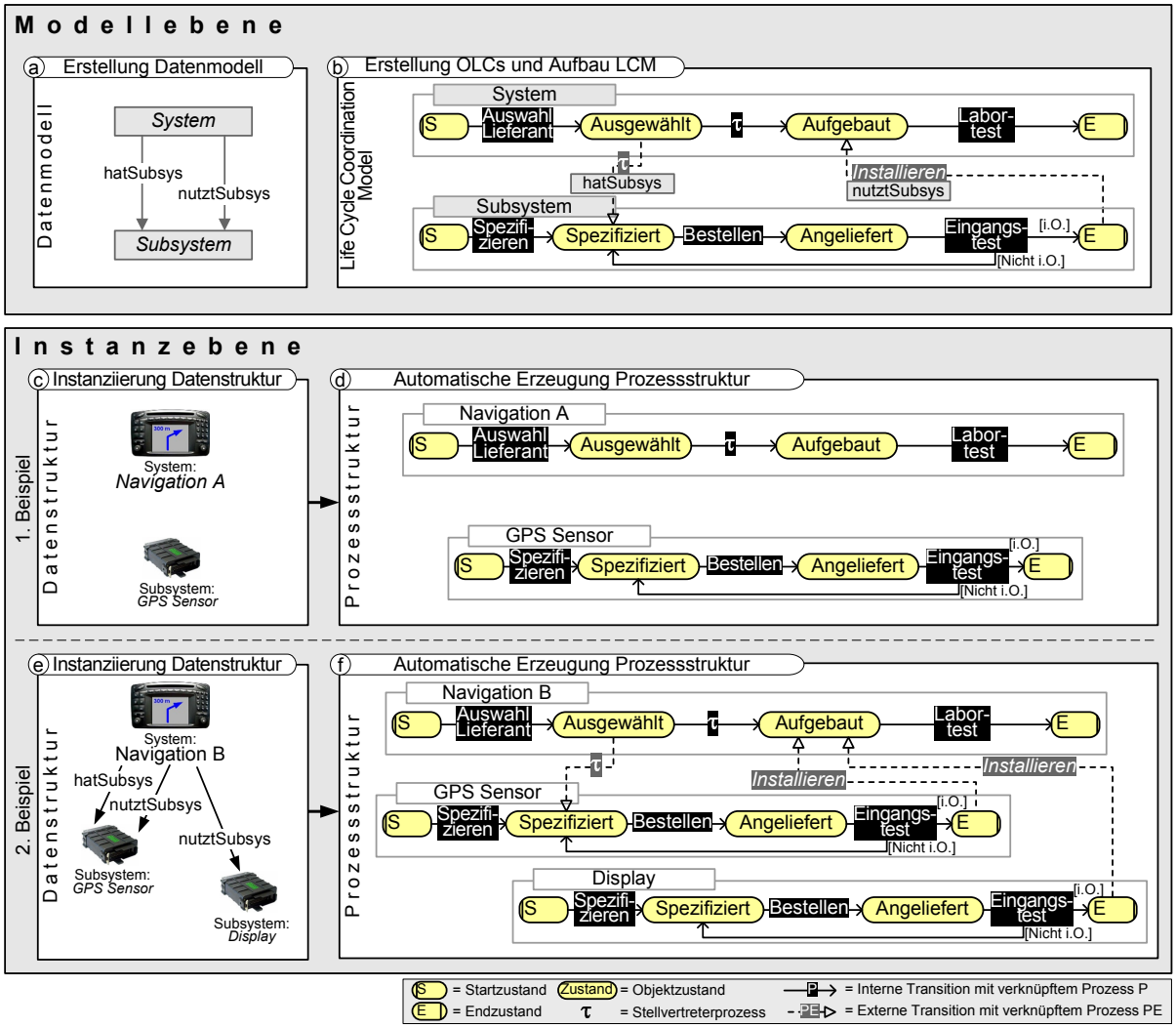


Abbildung 5.2: Vorgehen zur Modellierung datengetriebener Prozessstrukturen

Abbildung 5.2c zeigt ein einfaches Beispiel für eine gültige Datenstruktur, die aus den Objekten `Navigation A` und `GPS Sensor` besteht. Es wurden keine Relationen zwischen den beiden Objekten instanziiert. Entsprechend den in der Datenstruktur verwendeten Objekttypen und den im LCM hinterlegten Bausteinen der Prozessstruktur, also den OLCs, kann nun die Prozessstruktur aufgebaut werden. Im Beispiel aus Abbildung 5.2d wird für das Objekt `Navigation A` der entsprechende OLC für den Objekttyp `System` aus dem LCM instanziiert. Dasselbe geschieht für das Objekt `GPS Sensor`. Das Ergebnis ist eine ausführbare Prozessstruktur mit zwei OLCs für die beiden Objekte der gegebenen Datenstruktur.

Abbildung 5.2e zeigt eine etwas komplexere Datenstruktur mit dem Objekt `Navigation B` vom Typ `System` und den beiden Objekten `GPS Sensor` und `Display` vom Typ `Subsystem`. In diesem Beispiel sind zudem verschiedene Relationen zwischen den Objekten angelegt (z.B. Relationen `hatSubsystem` und `nutztSubsystem` zwischen den Objekten `Navigation B` und `GPS Sensor`). Diese Relationen werden in der zur Datenstruktur gehörenden Prozessstruktur entsprechend ih-

rer Verknüpfung im LCM als externe Transitionen abgebildet (vgl. Abbildung 5.2f). Das Objekt `Display` ist dagegen nur über die Relation `nutztSubsystem` mit dem Objekt `Navigation B` verbunden. Daher wird nur die mit dem Relationstyp `nutztSubsystem` verknüpfte externe Transition zwischen den entsprechenden OLCs eingefügt (vgl. Abbildung 5.2f).

Die dargestellten Beispiele illustrieren den Mechanismus für die Modellierung datengetriebener Prozessstrukturen. Sie deuten den Vorteil einer automatischen Erzeugung der Prozessstruktur zugegebenermaßen nur an. Der Modellierungsaufwand realistischer Anwendungsszenarien lässt sich durch das CoRePro-Modellierungskonzept hingegen drastisch reduzieren. Die Systemstrukturen für das E/E-Release-Management sind mit hunderten von Objekten und entsprechend vielen Relationen sehr groß. Ferner unterscheiden sie sich von Release zu Release und von Baureihe zu Baureihe und müssen daher häufig neu gebildet werden. Eine manuelle Modellierung oder Adaption der Prozessstruktur ist bislang mit hohem Aufwand verbunden. Unser Modellierungsansatz erlaubt dagegen die automatische Erstellung unterschiedlicher Prozessstrukturen basierend auf gegebenen Datenstrukturen und damit auf hoher Abstraktionsebene [MRH07]. Im Abschnitt 5.5 zeigen wir exemplarisch die Reduktion des Modellierungsaufwands anhand einer Beispielrechnung.

5.2 Modellierung und Instanziierung von Datenstrukturen

Das Konzept der Modellierung datengetriebener Prozessstrukturen beruht auf der engen Integration von Daten- und Prozessstrukturen sowie der Trennung von Modell- und Instanzebene. Wie in Abschnitt 5.1.3 beschrieben, werden auf Modellebene die Daten und Prozesse integriert. Ziel ist es, durch Wiederverwendung von OLCs und externen Transitionen den Modellierungsaufwand zu reduzieren (vgl. Anforderung 5.2).

Der Vorteil der in Abschnitt 5.1.3 beschriebenen Lösung liegt darin, dass auf Modellebene bereits Elemente des Datenmodells mit Elementen der zukünftigen Prozessstruktur in Beziehung gesetzt und anschließend in der instanziierten Datenstruktur *verwendet* werden. Ferner lassen sich mit unserem Ansatz entsprechend Anforderung 5.1 sehr einfache, aber mächtige Datenmodelle abbilden ohne deren Semantik einzuschränken (z.B. Datenattribute, Vererbungsbeziehungen). Wir verknüpfen sie stattdessen mit Prozessbausteinen, um diese anschließend instanziieren zu können (siehe Abschnitt 5.3.1).

5.2.1 Erstellung des Datenmodells

Das *Datenmodell* bildet den Bauplan einer konkreten Datenstruktur. Für die Modellierung komplexer Datenmodelle stehen in der Theorie verschiedene ausgereifte Ansätze zur Verfügung (z.B. *Entity-Relationship*-Diagramme [Che76] oder UML2-Klassendiagramme [Obj03b]). In dieser Arbeit stehen jedoch weniger die Eigenschaften der Datenstruktur im Vordergrund, sondern vielmehr deren Beziehungen zur Prozessstruktur. Es wird daher ein einfaches, generisches Datenmodell angenommen, das die für die Integration mit der Prozessstruktur erforderlichen Eigenschaften aufweist. Das Datenmodell beschreibt damit eine bestimmte Sicht auf eine komplexe Datenstruktur, wie sie beispielsweise in einem Produktdaten-Management-System verwaltet wird (vgl. Anforderung 5.1).

In CoreFRO bestehen Datenmodelle aus zwei generischen Elementen: *Objekttypen* und *Relationstypen* (vgl. Definition 5.1). Objekttypen stellen die *Klasse* für Objekte einer Datenstruktur dar. Relationstypen verbinden jeweils zwei Objekttypen. Objekt- und Relationstypen können im Datenmodell beliebig oft verwendet werden. Das heißt, es können beliebig viele Objekttypen sowie Relationstypen zwischen ihnen angelegt werden, sofern sie eindeutig benannt sind. In Abbildung 5.3a ist ein Datenmodell mit drei Objekttypen (`A`, `B` und `C`) sowie zwei Relationstypen (`hatB` und `hatC`) beschrieben.

Definition 5.1 (Datenmodell)
Sei $\mathcal{T}$ die Menge aller Objekttypen und $\mathcal{R}$ die Menge aller Relationstypen. Dann ist ein Datenmodell ein Tupel $dm = (OT, RT)$ mit

- *$OT \subseteq \mathcal{T}$ umfasst die Menge der Objekttypen des Datenmodells dm*
- *$RT \subseteq OT \times \mathcal{R} \times OT$ umfasst die Menge der binären Relationstypen zur Verbindung von Objekttypen mit $rt = (ot_1, id, ot_2) \in RT$; d.h. zwei Objekttypen $ot_1, ot_2 \in OT$ werden durch eine mit id benannte Relation verbunden.*

$\mathcal{DM}$ bezeichne die Menge aller Datenmodelle.

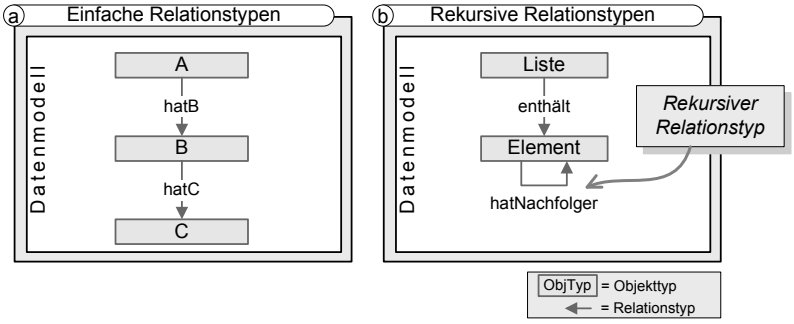


Abbildung 5.3: Datenmodelle bestehend aus generischen Objekt- und Relationstypen

Des Weiteren ist es möglich, *rekursive Relationstypen* (*zirkuläre Assoziationen* [JRH+04]) anzulegen. Sie werden benötigt, um in der Datenstruktur zwei Objekte desselben Objekttyps miteinander verbinden zu können. In Abbildung 5.3b ist als Beispiel das Datenmodell einer Liste dargestellt, deren Elemente aufeinander verweisen können.

5.2.2 Instanziierung konkreter Datenstrukturen

Mittels eines gegebenen Datenmodells können konkrete *Datenstrukturen* angelegt (d.h. instanziiert) werden. Eine Datenstruktur besteht aus *Objekten* und *Relationen*, die auf Objekt- bzw. Relationstypen basieren (vgl. Definition 5.2). Letztgenannte können beliebig oft instanziiert werden, wobei zu beachten ist, dass eine Relation vom Relationstyp `relType` nur Objekte miteinander verknüpft, deren Objekttypen den in `relType` definierten Objekttypen entsprechen. Um redundante Beziehungen auszuschließen, müssen die Relationen zwischen zwei Objekten immer von unterschiedlichen Relationstypen sein.

Definition 5.2 (Datenstruktur)
Sei $dm = (OT, RT) \in \mathcal{DM}$ ein Datenmodell. Dann ist eine aus dm erstellte Datenstruktur ein Tupel $ds = (O, R)$ mit

- *O ist die Menge der Objekte der Datenstruktur. Jedes Objekt $o \in O$ ist mit einem Objekttyp $ot \in OT$ verknüpft; ferner liefere die Funktion $objtype(o) = ot$ den Objekttyp für ein Objekt o.*
- *$R \subseteq O \times RT \times O$ ist die Menge der Relationen. Jede Relation $r = (o_1, rt, o_2) \in R$ ist mit einem Relationstyp $rt \in RT$ verknüpft; ferner liefere die Funktion $reltype(r) = rt$ den Relationstyp für eine Relation r.*

$\mathcal{DS}$ bezeichne die Menge aller Datenstrukturen.

Abbildung 5.4a zeigt zwei verschiedene Instanzen des Datenmodells aus Abbildung 5.3 mit einfachen Relationstypen. Abbildung 5.4b zeigt zwei Datenstrukturen mit Relationen basierend auf dem rekursiven Relationstyp `hatNachfolger` aus dem Datenmodell aus Abbildung 5.3b. Die rekursiven Beziehungstypen, die im Datenmodell einen Objekttyp mit sich selbst verbinden, müssen in instanziierter Form unterschiedliche Objekte desselben Typs referenzieren. Wie in anderen modellbasierten Ansätzen, können ausgehend von einem Datenmodell beliebig viele verschiedene Datenstrukturen instanziiert werden.

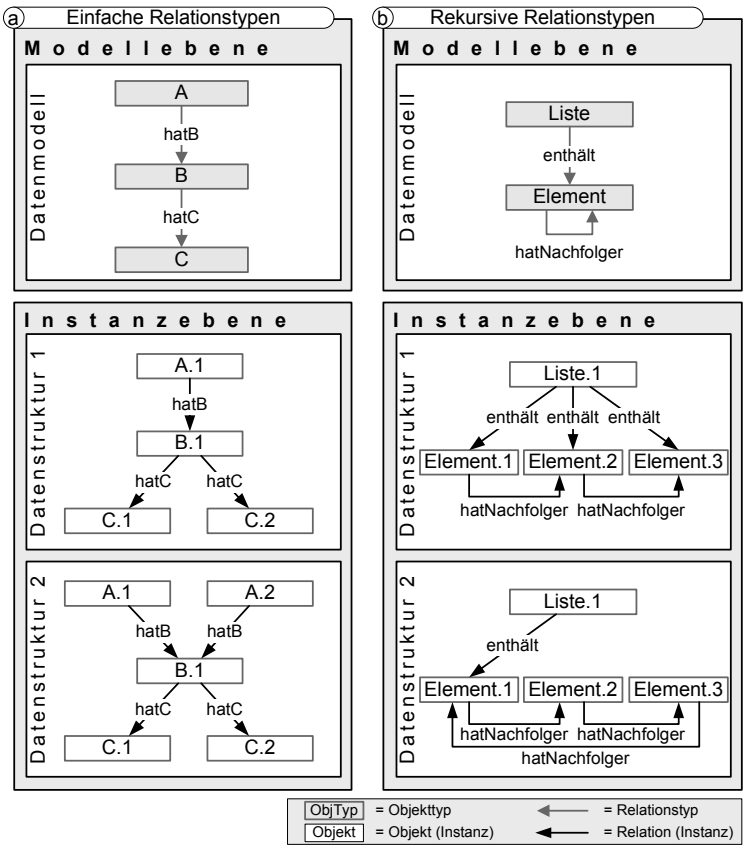


Abbildung 5.4: Beispiel-Datenstrukturen basierend auf den Datenmodellen aus Abbildung 5.3

5.3 Modellierung und Erzeugung datengetriebener Prozessstrukturen

Die manuelle Erstellung großer Prozessstrukturen, wie sie beispielsweise im Release-Management der E/E-Entwicklung vorkommen, ist mit hohem Aufwand verbunden. Laut Anforderung 5.2 soll die Prozessstruktur nicht manuell erzeugt, sondern möglichst automatisch generiert werden. Hierfür ist die Trennung von Modell- und Instanzebene ein wichtiger Baustein (vgl. Abschnitt 5.1.3). Ziel ist die Integration von Daten und Prozessen auf Modellebene im *Life Cycle Coordination Model (LCM)* und darauf basierend die Wiederverwendung dieser Informationen in verschiedenen Prozessstrukturen durch geeignete Instanziierungskonzepte. Damit kann die automatische Erzeugung von Prozessstrukturen auf Basis gegebener Datenstrukturen, unter Nutzung der im LCM hinterlegten Beziehungen zwischen Datenmodell und Bausteinen der Prozessstruktur, erfolgen.

5.3.1 Erstellung des Life Cycle Coordination Model

Um eine automatische Erzeugung der Prozessstruktur zu realisieren, müssen die Bausteine der Prozessstruktur so definiert werden, dass sie entsprechend einer gegebenen Datenstruktur instanziiert werden können. Hierfür beschreiben wir die Prozessstruktur auf Modellebene im *Life Cycle Coordination Model (LCM)*. Die Abbildung ihrer Beziehung zu Objekttypen aus dem Datenmodell geschieht über die in Abschnitt 4.2 eingeführten Object Life Cycles (OLCs), indem für jeden Objekttyp der entsprechende OLC definiert und in das LCM eingefügt wird. So wird in Abbildung 5.5 für die Objekttypen `A`, `B` und `C` des Datenmodells jeweils ein OLC definiert. Auf Instanzebene kann damit für jede Instanz eines Objekttyps der entsprechende OLC erzeugt werden (vgl. Abschnitt 5.1.3).

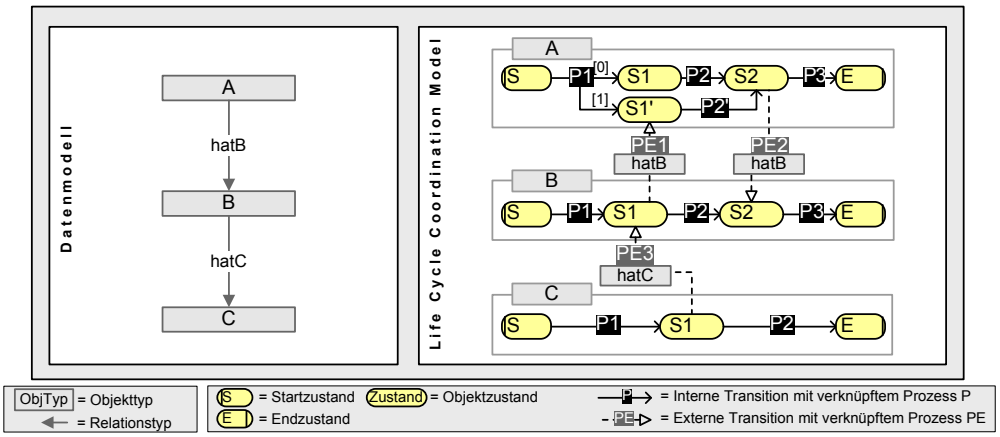


Abbildung 5.5: Life Cycle Coordination Model für das Datenmodell aus Abbildung 5.3a

Um eine vollständig automatisch erzeugbare Prozessstruktur zu erhalten, müssen auch externe Transitionen geeignet modelliert werden. Hier profitieren wir von der Eigenschaft, dass in datengetriebenen Prozessstrukturen die Relationen zwischen Objekten direkt auf die Synchronisationsbeziehungen ihrer OLCs abbildbar sind (vgl. Abschnitt 5.1.1 bzw. globale Synchronisationsmuster in Abschnitt 4.1.2). Für die Definition des LCM bedeutet das, dass externe Transitionen

mit Relationstypen verknüpft und entsprechend der Relationen in der Datenstruktur instanziiert werden. Die direkte Verknüpfung ist jedoch nicht ausreichend, da eine Relation zu verschiedenen Synchronisationen der Prozesse zweier Objekte führen kann (vgl. Relationstyp `hatB` in Abbildung 5.5). Aus diesem Grund bündeln wir externe Transitionen formal als *OLC-Dependencies* (vgl. Abbildung 5.6 bzw. Definition 5.3). Sie werden im LCM mit Relationstypen verknüpft (vgl. Definition 5.4). Eine externe Transition kann hierbei in höchstens einer OLC-Dependency enthalten sein, da ansonsten die Eindeutigkeit beim Hinzufügen verschiedener Relationen mit derselben externen Transition (bzw. bei der späteren dynamischen Adaption) nicht gegeben ist. Dies ist jedoch nur eine kleine Einschränkung, denn entsprechend Definition 4.12 können mehrere externe Transitionen zwischen zwei Zuständen existieren. Sie sind dann durch den verknüpften Prozess unterscheidbar.

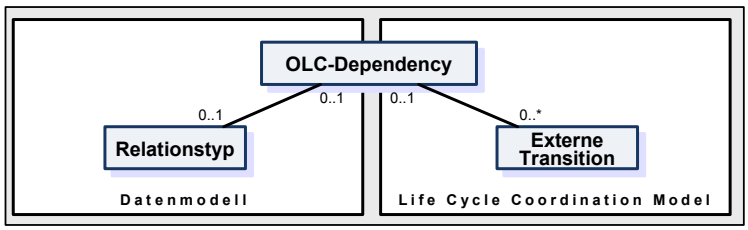


Abbildung 5.6: Verknüpfung von Relationstypen mit OLC-Dependencies

Definition 5.3 (OLC-Dependency)
Seien $olc_i \in \mathcal{OLC}$ mit $olc_i = (P_i, V_i, TS_i), i = 1, 2$ zwei unterschiedliche Object Life Cycles mit Transitionssystemen $TS_i = (S_i, T_i, s_{start_i}, s_{end_i})$ (vgl. Definition 4.1 auf Seite 57). Dann ist eine OLC-Dependency zwischen olc_1 und olc_2 ein Tupel $olc_{Dep} = (P, EST)$ mit

- *P ist die Menge der Synchronisationsprozesse der OLC-Dependency*
- *EST ist die Menge der externen Transitionen der OLC-Dependency mit $est = (s, p, s') \in EST \Leftrightarrow \big((s \in S_1 \wedge s' \in S_2) \vee (s' \in S_1 \wedge s \in S_2)\big) \wedge p \in P \wedge \big(\nexists\, olc'_{Dep} = (P', EST') \in \mathcal{OLC}_{\mathcal{DEP}} : est \in EST'\big)$.*

$\mathcal{OLC}_{\mathcal{DEP}}$ beschreibe die Menge aller OLC-Dependencies. Ferner beschreibe $EST(olc_{Dep})$ die Menge der verknüpften externen Transitionen für eine OLC-Dependency $olc_{Dep} \in \mathcal{OLC}_{\mathcal{DEP}}$.

Definition 5.4 (Life Cycle Coordination Model, LCM)
Sei $dm = (OT, RT) \in \mathcal{DM}$ ein Datenmodell. Dann ist das mit dm verknüpfte Life Cycle Coordination Model (LCM) ein Tupel $lcm = (OLC, OLC_{DEP})$ mit

- *$OLC : OT \mapsto \mathcal{OLC}$ ordnet jedem Objekttyp $ot \in OT$ des Datenmodells einen Object Life Cycle $OLC(ot) \in \mathcal{OLC}$ zu.*
- *$OLC_{DEP} : RT \mapsto \mathcal{OLC}_{DEP}$ ordnet jedem Relationstyp $rt = (ot1, rt, ot2) \in RT$ des Datenmodells eine OLC-Dependency $OLC_{DEP}(rt) \in \mathcal{OLC}_{\mathcal{DEP}}$ bezogen auf die Object Life Cycles $OLC(ot1)$ und $OLC(ot2)$ der Objekttypen $ot1$ bzw. $ot2$ zu.*

Die Richtung einer Relation ist implizit über ihren Relationstyp definiert, der konkrete Objekttypen als Quelle und Ziel angibt (vgl. Definition 5.1). Bei einfachen Relationstypen ist die Richtung

der Relation irrelevant, denn die externen Transitionen, die über eine OLC-Dependency mit einem Relationstyp verknüpft sind, sind an die Richtung der Beziehung nicht gebunden. Das heißt, die externen Transitionen können unabhängig von der Richtung der Relation die Zustände der entsprechenden OLCs synchronisieren. Das ist wichtig, um verschiedene globale Synchronisationsmuster (vgl. Szenario 4.4 auf Seite 51) bedienen zu können. Im Beispiel aus Abbildung 5.5 ist zwar die Relation `hatB` ausgehend von Objekttyp `A` zu Objekttyp `B` modelliert, die Synchronisationsrichtung der mit OLC-Dependencies verknüpften externen Transitionen kann jedoch auch entgegengesetzt verlaufen. Die Richtung der Relation ist für die Definition der OLC-Dependency an dieser Stelle unwichtig, zumal die Richtung der zugehörigen externen Transitionen ohnehin durch die eindeutige Bezeichnung von Quell- und Zielzuständen in den unterschiedlichen OLCs immer klar definiert ist.

Bei rekursiven Relationstypen ist diese Richtung jedoch nicht klar vorgegeben, da Quell- und Zielzustände im selben OLC(-Typ) liegen (vgl. Abbildung 5.7), auf Instanzebene dann aber die Zustände unterschiedlicher OLCs synchronisieren. Dann ist die Synchronisationsrichtung nicht mehr durch die externen Transitionen ersichtlich und wir müssen uns an der Richtung der Relationen orientieren (vgl. Abbildung 5.4). Auf Instanzebene ist dann die Richtung der Relation für die Synchronisation der Zustände ausschlaggebend (siehe Abschnitt 5.3.2).[4]

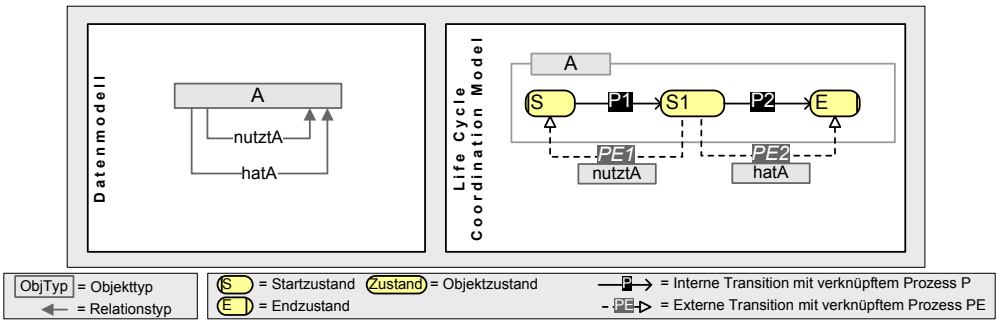


Abbildung 5.7: Abbildung rekursiver Relationstypen im LCM

5.3.2 Datengetriebene Generierung der Prozessstruktur

Auf Basis des erstellten LCM und einer gegebenen Datenstruktur kann nun, wie in Abschnitt 5.1.3 skizziert, die *datengetriebene Prozessstruktur* in zwei Schritten automatisch erzeugt werden. Dafür wird im ersten Schritt für jedes Objekt der gegebenen Datenstruktur der im LCM mit seinem Objekttyp verknüpfte OLC instanziiert (kopiert) und in die Prozessstruktur eingefügt (vgl. Abbildung 5.8a und vgl. Abbildung 5.8b). Das Ergebnis ist eine Prozessstruktur mit unsynchronisierten OLCs, wobei die Anzahl der OLCs in der Prozessstruktur der Anzahl Objekte in der Datenstruktur entspricht. Im zweiten Schritt der Erzeugung werden auf Basis der Relationen

[4] Die Verknüpfung externer Transitionen mit rekursiven Relationstypen kann noch erweitert werden, indem für externe Transitionen, die in einer rekursiven OLC-Dependency gebündelt sind, auch die Synchronisierungsrichtung annotiert wird. Damit lassen sich auch externe Transitionen mit unterschiedlicher Synchronisationsrichtung innerhalb einer rekursiven OLC-Dependency realisieren. Die Korrektheitskriterien bleiben in diesem Fall weiterhin gültig. Dieses Konzept wird aus Gründen der Verständlichkeit in dieser Arbeit nicht weiter beschrieben, wurde jedoch im CoRePro-Demonstrator realisiert.

in der Datenstruktur die OLCs synchronisiert. Hierfür werden für den jeweiligen Relationstyp die in der OLC-Dependency hinterlegten externen Transitionen zwischen den OLCs der verknüpften Objekte eingefügt (d.h. instanziiert). Das Ergebnis ist die vollständige Prozessstruktur mit synchronisierten OLCs (vgl. Abbildung 5.8c).

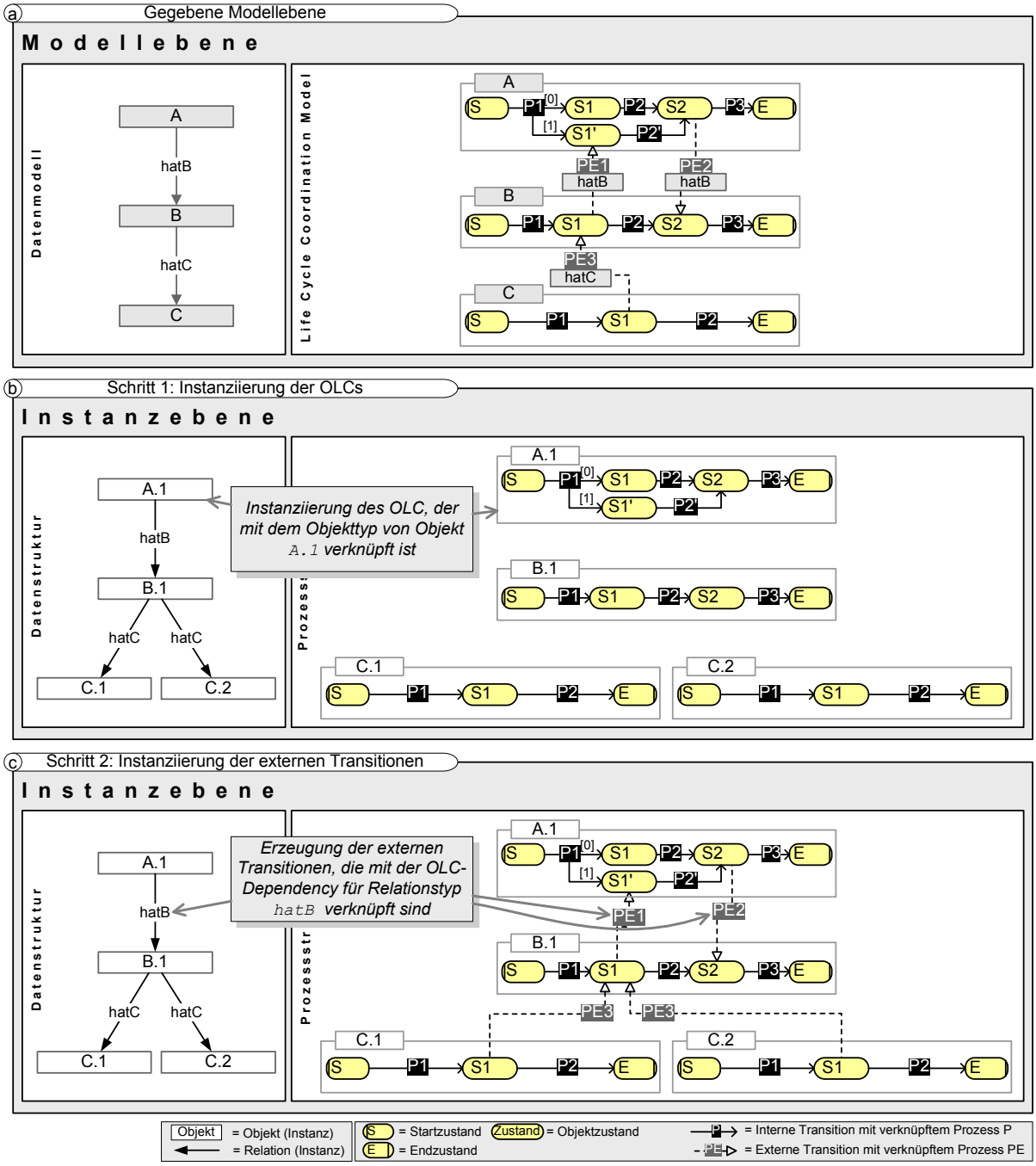


Abbildung 5.8: Erzeugung einer Prozessstruktur für die Modellebene aus Abbildung 5.5 und die Datenstruktur aus Abbildung 5.4

Die *Life Cycle Coordination Structure (LCS)* repräsentiert die vollständige datengetriebene Prozessstruktur bestehend aus Datenmodell, LCM, Datenstruktur und generierter Prozessstruktur (vgl. Definition 5.5).

Definition 5.5 (Life Cycle Coordination Structure, LCS)
Sei $dm = (OT, RT)$ *ein Datenmodell und* $lcm = (OLC, OLC_{DEP})$ *das zugehörige Life Cycle Coordination Model. Des Weiteren sei die Datenstruktur* $ds = (O, R)$ *eine Instanz von* dm. *Dann gilt für die zu* dm, lcm *und* ds *gehörende Prozessstruktur* $ps = (OLC_{inst}, EST_{inst})$:

- $OLC_{inst} : O \mapsto OLC$ *ordnet jedem Objekt* $o \in O$ *eine Instanz* $OLC_{inst}(o) \in OLC$ *des mit seinem Objekttyp* $ot = objtype(o) \in OT$ *verknüpften Object Life Cycle* $OLC(ot)$ *zu.*
- $EST_{inst} : R \mapsto OLC_{DEP}$ *ordnet jeder Relation* $r = (o_1, rt, o_2) \in R$ *die Instanzen* $est_{inst}(r)$ *der mit ihrem Relationstyp* $rt = reltype(r) \in RT$ *verknüpften externen Transitionen* $EST(OLC_{DEP}(rt))$ *zu. Die jeweiligen Quell- und Zielzustände der externen Transitionen in* $est_{inst}(r)$ *seien an die OLC-Instanzen* $OLC_{inst}(o_1)$ *und* $OLC_{inst}(o_2)$ *angepasst.*

Das Tupel $lcs = (dm, lcm, ds, ps)$ *beschreibe die* Life Cycle Coordination Structure *(LCS) (d.h. die vollständige datengetriebene Prozessstruktur von* ds*). Des Weiteren beschreibe* $\mathcal{LCS}$ *die Menge aller Life Cycle Coordination Structures.*

5.4 Ausführung und Korrektheit der erzeugten Prozessstruktur

Eine erzeugte Prozessstruktur repräsentiert im Sinne der Prozess-Management-Konzepte eine *Prozessinstanz* (vgl. Abschnitt 1.1). Sie kann entsprechend der in Kapitel 4 definierten operationalen Semantik direkt ausgeführt werden. Es ist von großer Wichtigkeit, dass erzeugte Prozessstrukturen *korrekt* sind (vgl. Anforderung 5.3). So darf es während der Ausführung zu keinen Verklemmungen kommen. Die Kriterien für die Korrektheit einer Prozessstruktur wurden bereits in Kapitel 4 vorgestellt. Da die erzeugten Prozessstrukturen den in Abschnitt 4.5.2 diskutierten strukturellen Eigenschaften folgen, ist *strukturelle Korrektheit* entsprechend der Argumentation in den Abschnitten 4.5.2 und 5.3 gegeben. Die *dynamische Korrektheit*, also die Zyklenfreiheit einer Prozessstruktur, muss allerdings explizit sichergestellt werden (vgl. Abschnitt 4.7).

Mithilfe der Funktion $path_{ext}$ können Zyklen zwar automatisch gefunden werden (vgl. Satz 4.3), diese Suche ist jedoch in großen Prozessstrukturen sehr aufwändig. Ferner kann dem Benutzer nicht zugemutet werden, die Korrektheit einer zyklischen Prozessstruktur manuell herzustellen, indem er beispielsweise den LCM überarbeitet. Ziel der Modellierung ist vielmehr, die Korrektheit *per Konstruktion* sicherzustellen (vgl. Anforderung 5.3).[5] Dazu müssen wir die Modellebene untersuchen und ihre Korrektheitseigenschaften prüfen.

Die Modellebene bzw. das LCM beschreibt die *Bausteine* (d.h. OLCs und externe Transitionen), aus denen sich eine Prozessstruktur aufbauen lässt, sowie deren mögliche Zusammensetzung. Für die Sicherstellung der Korrektheit per Konstruktion ergibt sich die folgende herausfordernde Fragestellung: Kann Korrektheit bereits im LCM geprüft und für alle erzeugten Prozessstrukturen garantiert werden?

[5] Die Korrektheit im Rahmen dynamischer Adaptionen diskutieren wir in Kapitel 6.

Auf Basis des LCM lassen sich bereits Aussagen über die Eigenschaften und insbesondere der Zyklen einer generierten Prozessstruktur ableiten. Diese Charakteristik können wir nutzen, um die Zyklenfreiheit jeder erzeugbaren Prozessstruktur zu garantieren. Zyklen innerhalb einer Prozessstruktur werden, vereinfacht gesagt, durch sich *überschneidende* externe Transitionen verursacht (vgl. Abschnitt 4.7). Ist ein Zyklus in der Prozessstruktur enthalten, kann dieser entsprechend folgender Annahme *nicht* durch Hinzufügen weiterer Kanten aufgelöst werden:

> Ist eine Prozessstruktur nicht korrekt (entsprechend Definition 4.3), enthält sie also einen durch externe Transitionen verursachten Zyklus, kann die Korrektheit nur durch Wegnahme, nicht jedoch durch Hinzufügen externer Transitionen aufgelöst werden.

Gelingt es, aus dem LCM eine repräsentative Prozessstruktur abzuleiten, die hinsichtlich der Kombination externer Transitionen vollständig ist, dann können wir darauf eine Zyklenprüfung durchführen. Damit sind wir in der Lage auszusagen, ob basierend auf dem Modell grundsätzlich ein Zyklus entstehen kann. Eine solche repräsentative und vollständige Prozessstruktur stellt bereits das LCM selbst dar, denn es enthält alle OLCs und alle externen Transitionen, die in einer Prozessstruktur instanziiert werden können (vgl. Abbildung 5.5). Enthält das LCM also keine Zyklen, wird auch jede auf deren LCM basierende Prozessstruktur keine Zyklen enthalten (vgl. Satz 5.1).

Satz 5.1 (Korrektheit einer erzeugten Prozessstruktur)
Ist ein LCM korrekt (vgl. Definition 4.3), dann ist auch jede auf Basis dieses LCM erzeugte Prozessstruktur korrekt.

Ein Beweis von Satz 5.1 findet sich in Anhang A.

Auf den ersten Blick scheint eine gesonderte Diskussion von rekursiven Beziehungstypen hinsichtlich der Korrektheit entstehender Prozessstrukturen notwendig. Schließlich lassen sich auf Basis eines einzelnen rekursiven Relationstyps Zyklen in der Datenstruktur erzeugen (vgl. Abbildung 5.9a). Zyklen in der Datenstruktur können zwar zu Zyklen in der Prozessstruktur führen (vgl. Abbildung 5.9a), sie müssen aber nicht (vgl. Abbildung 5.9b). Ob eine zyklische Datenstruktur zu einem Zyklus in der Prozessstruktur führt, hängt von den mit dem Relationstyp verknüpften externen Transitionen ab. Bilden diese auf der Modellebene bereits einen Zyklus (vgl. Relationstyp `nutztA` in Abbildung 5.7), können sie auch in der Prozessstruktur zu einem Zyklus führen. Dies ist der Fall, wenn sich im LCM der Zielzustand einer rekursiven externen Transition *logisch vor* deren Quellzustand im selben OLC befindet oder mit ihm *identisch* ist. Ist im LCM kein Zyklus enthalten, ist auch die erzeugte Prozessstruktur zyklenfrei. Die eingeführten Korrektheitskriterien (vgl. Satz 5.1) bleiben daher gültig. Rekursive Beziehungen bzw. die mit ihnen verknüpften externen Transitionen bedürfen damit keiner speziellen Prüfung. Die Kriterien bleiben ebenso gültig, wenn exklusive Pfade innerhalb eines OLC über rekursive externe Transitionen synchronisiert werden, denn es findet hier – wie auch bei „normalen“ externen Transitionen – eine Deadpath-Eliminierung für nichtgewählte Pfade statt.

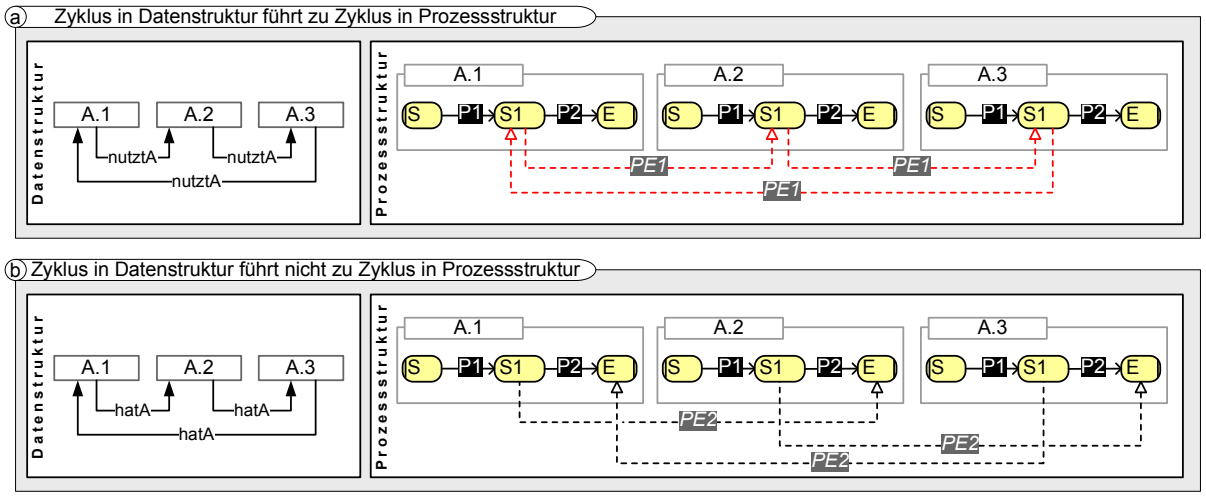


Abbildung 5.9: Abbildung zyklischer Datenstrukturen (vgl. Modellebene aus Abbildung 5.7) in der Prozessstruktur

5.5 Diskussion

In der Literatur finden sich einige Ansätze, die sich mit der Integration von Daten und Prozessen beschäftigen (z.B. [KS91, Aal97a, RD98, AWG05, Obj03b, KRG07]). Die Ursprünge für die integrierte Beschreibung von Daten und Prozessen finden sich in Methoden zur Beschreibung des Systemverhaltens einzelner Objekte (z.B. mittels Zustandsdiagrammen). Eine Ausprägung von Zustandsdiagrammen sind Statecharts, ein Formalismus der ursprünglich für den Entwurf und die Analyse reaktiver Systeme entwickelt wurde (vgl. Abschnitt 4.8) [Har87, Par98]. Der aufkommende Einsatz von objektorientierten Methoden in der Softwaretechnik Anfang der 90er Jahre erforderte unter anderem auch neue Techniken für den Entwurf objektorientierter Software (z.B. Object Oriented Design in [Boo90]). Die *Unified Modeling Language* (UML) vereinigt verschiedene Modellierungsmethoden, die zum Beispiel die Definition einer Objektstruktur über Klassendiagramme und das Verhalten der einzelnen Objekte über Zustandsdiagramme erlauben [Obj97, Obj03b, JRH+04]. Die Beschreibung objektorientierter Systeme ist auch in der Datenbankdomäne aufgegriffen worden, wobei hier ebenfalls Ansätze entwickelt worden sind, die sich mit der Integration von Objektstrukturen und der Beschreibung des Verhaltens von Objekten beschäftigen (z.B. [KS91]).

Im Prozess-Management geschieht die Integration von Daten meist über Datenflüsse, die parallel zu einem Kontrollfluss modelliert werden (vgl. Abschnitt 1.1). Insbesondere Modellierungsansätze bieten mittlerweile eine große Ausdrucksmächtigkeit und erlauben beispielsweise die Beschreibung von Zustandsänderungen der Objekte in Datenflüssen [Obj03b, Bus06]. Inwieweit sich die genannten Ansätze für die Modellierung von Prozessstrukturen basierend auf gegebenen Datenstrukturen eignen, diskutieren wir in den folgenden Abschnitten.

5.5.1 Allgemeine Ansätze zur Prozessmodellierung

Weit verbreitete Ansätze für die aktivitätenorientierte Beschreibung von Prozessen sind BPMN [Bus06] und UML-Aktivitätsdiagramme [Obj03b, DH01, JRH+04]. Sie integrieren Daten durch die Modellierung von Datenflüssen. So erlauben sie die Modellierung von Zustandsänderungen von Objekten bedingt durch die Ausführung von Aktivitäten. Beide Sprachen erlauben mit der *multiplen Instanziierung*, also der automatischen Erzeugung von Prozessschritten für eine gegebene Menge oder Liste an zu verarbeitenden Elementen (vgl. Abschnitt 3.1.2), eine rudimentäre Modellierungsunterstützung. Sie unterstützen jedoch nur die Erzeugung einfacher Prozessstrukturen basierend auf „flachen" Datenstrukturen. Damit reichen diese Konstrukte für den automatischen Aufbau von Prozessstrukturen nicht aus. Zwar würden sich durch die Modellierung von OLCs als Prozessschritte einer multiplen Instanziierung Teile einer hierarchischen Struktur generisch abbilden lassen, eine individuelle Zuordnung zu bestimmten Objekttypen ist jedoch nicht möglich. So muss für jeden Objekttyp der Struktur ein Konstrukt manuell in den Prozess eingefügt werden. Die Struktur ist dann allerdings fest vorgegeben und lässt sich nicht durch die Relationen zwischen Objekten definieren. Des Weiteren lassen sich mit der multiplen Instanziierung synchronisierte Abläufe nicht wie gewünscht darstellen, denn die Synchronisation von Aktivitäten, die in verschiedenen Prozessfragmenten der multiplen Instanziierung enthalten sind, ist nicht möglich.

Der ADEPT-Ansatz erlaubt ebenfalls die Definition von Datenflüssen [Rei00]. Weitergehende Arbeiten betrachten auch die Verwendung zustandsbehafteter Objekte innerhalb der Datenflüsse [Gös05]. Die Modellierung von Abhängigkeiten zwischen Objekten ist mit diesem Ansatz jedoch nicht möglich. Die automatische Erzeugung einer Prozessstruktur auf Basis einer gegebenen Datenstruktur wird daher ebenfalls nicht unterstützt.

5.5.2 Ansätze zur Generierung von Prozessstrukturen

Die Nutzung komplexer Datenstrukturen für die Erzeugung einer geeigneten Verarbeitungssequenz wurde bereits mit der *Jackson Structured Programming Method* (JSP) vorgestellt [Jac75]. Ziel ist die Spezifikation einer plattformunabhängigen Programmstruktur basierend auf der Transformation ihrer Ein- und Ausgangsdaten.

Ein ähnlicher Ansatz wurde von *van der Aalst* in [Aal97a, Aal99] vorgestellt, der auf Basis einer gegebenen Produktstruktur (*Bill of Material*, BOM) den erforderlichen Herstellungsprozess für das Produkt erzeugt (vgl. Abschnitt 3.2.2). Dafür wird die Produktstruktur mittels Komponenten und unbenannten Beziehungen zwischen Komponenten beschrieben. Es wird weiterhin zwischen obligaten, optionalen und alternativen Beziehungen unterschieden. Der Ansatz erlaubt damit zwar die automatische Erzeugung einer Prozessstruktur mit Bottom-up-Synchronisation (vgl. Szenario 4.4), Beziehungen zwischen Komponenten werden jedoch starr *auf* Kontrollflusskanten abgebildet. Eine konfigurierbare Synchronisation der Prozesse verschiedener Objekte, so wie sie in CoreFlow durch externe Transitionen (mit Synchronisationsprozessen) und deren Beziehung zu Relationen möglich ist, wird nicht unterstützt.

Ein Ansatz, der teilweise auf den Konzepten aus [Aal97a, Aal99] basiert, ist *(Product Driven) Case-Handling* (vgl. Abschnitt 3.1.4). Hier wird die Abfolge der Aktivitäten eines Prozesses implizit durch die zur Verfügung stehenden Daten vorgegeben [AB01b, SW02, ASW03, AWG05].

Es wird dabei kein expliziter Kontrollfluss (d.h. keine feste Reihenfolge der Arbeitsschritte) festgelegt, sondern vielmehr der für die Ausführung relevante Datenfluss. Abbildung 5.10 zeigt als Beispiel einen Prozess zur Bearbeitung einer Angebotsanfrage. Die Aktivitäten (z.B. `Anfrage erfassen`) sind nicht durch einen Kontrollfluss verbunden, sondern mittels Datenobjekten als Ein- und Ausgaben (z.B. `Kundendaten` und `Anforderungsprofil`). Damit ergibt sich der endgültige Ablauf erst zur Laufzeit, abhängig von den zur Verfügung stehenden Daten. Der Vorteil dieser Modellierungsmethodik liegt in der flexiblen und effizienten Ausführung. Werden die Ausgabedaten zu Beginn der Aktivität erzeugt, kann parallel mit der nächsten Aktivität begonnen werden, auch wenn die erzeugende Aktivität noch gar nicht beendet ist. Ein weiterer Vorteil der datenorientierten Modellierung ist die Weiterverwendung bereits existierender Daten. Ist beispielsweise ein für die Ausführung benötigtes Datenobjekt bereits vorhanden, braucht die erzeugende Aktivität gar nicht instanziiert bzw. ausgeführt zu werden. Case-Handling erlaubt zwar die flexible Definition eines Prozesses, der Einsatz für datengetriebene Prozessstrukturen, wie wir sie mit unserem Ansatz unterstützen, ist jedoch aus folgenden Gründen nicht möglich: Die Ausführung der Aktivitäten wird zwar durch die zur Verfügung stehenden Daten gesteuert, der Datenfluss ist jedoch fest vorgegeben und muss manuell vollständig modelliert werden. Des Weiteren werden Beziehungen zwischen Objekten nicht direkt berücksichtigt, sondern implizit über die Spezifikation der Eingabedaten einer Aktivität dargestellt (obligat, eingeschränkt oder frei). Die Erzeugung der Prozessstruktur über eine gegebene Datenstruktur ist damit nicht möglich.

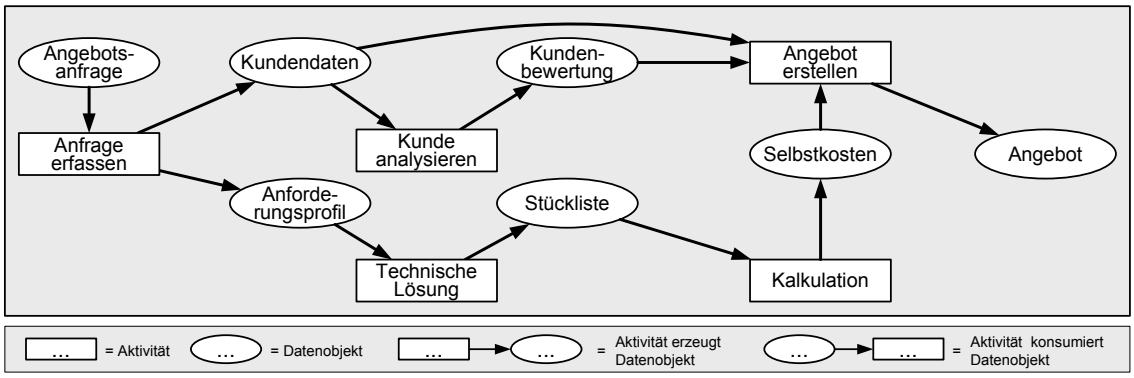


Abbildung 5.10: Datenorientierte Prozessmodellierung mit dem Case-Handling Ansatz [SW02]

Product Based Workflow Design (PBWD) erweitert den oben beschriebenen Ansatz aus [Aal97a, Aal99] um Optimierungsaspekte für die Generierung von Prozessstrukturen. Weitere Arbeiten aus diesem Umfeld erlauben eine direkte Verknüpfung der Elemente einer Produktstruktur mit Prozessen [VRA08, Van09] und damit die *direkte Ausführung der Datenstruktur*. Der Ansatz nutzt damit die Abhängigkeiten innerhalb einer Datenstruktur und interpretiert sie als Datenflüsse um eine flexible Koordination der Prozesse zu erreichen (vgl. Case-Handling-Ansatz). Die Synchronisationsmöglichkeiten sind jedoch beschränkt, denn Abhängigkeiten in der Produktstruktur, wie sie in [RLA03, VRA08] genutzt werden, entsprechen Aggregationsbeziehungen (d.h. sie beschreiben Existenzvoraussetzungen für Objekte der Datenstruktur und entsprechen einem einzelnen Relationstyp in CoRepro). Mehrere Relationstypen, die für die flexible Synchronisation von Prozessen notwendig sind, lassen sich nicht ausdrücken.

Die zielorientierte Prozessmodellierung aus der Planungstechnik, wie sie beispielsweise in *OptinSolutions DBPM* (vgl. Abschnitt 3.1.4) realisiert wird, erlaubt ebenfalls die automatische Erzeugung eines Prozesses auf Basis eines gegebenen Ziels [MWRR06]. Wird beispielsweise als Endergebnis ein Produkt bzw. ein Produktzustand definiert, kann der Prozessablauf durch Anordnung der vordefinierten Aktivitäten generiert werden. Dafür werden automatisch zielorientiert verschiedene mögliche Kontrollflüsse zur Erreichung des Ziels aufgebaut. Basierend auf Anzahl, Anordnung und Eigenschaften der in den Kontrollflüssen angeordneten Aktivitäten, können die effizientesten Kontrollflüsse (z.B. unter zeitlichen Aspekten) ausgewählt werden [MWRR06, Mül06]. Kombiniert mit dem zugrundeliegenden Datenmodell, lassen sich mit *OptinSolutions DBPM* auch zusammengesetzte Objekte in Datenmodellen anlegen und anschließend instanziieren (vgl. Abbildung 5.11a und b). Die Art der Abhängigkeiten ist hierbei auf die Aggregation der Zustände der Unterobjekte beschränkt. Die Abhängigkeiten zwischen Objekten werden bei Erzeugung der Prozessstruktur in Form von Aktivitäten berücksichtigt, die die Zustandsabhängigkeiten zwischen Objekten und ihren Unterobjekten beschreiben (vgl. Abbildung 5.11c). Es handelt sich dabei also um einfache Ende-Start-Synchronisationen. Die Unterscheidung verschiedener Relationstypen zur individuellen Synchronisation ist aber nicht möglich.

5.5.3 Integrierte Modellierung von Daten und Prozessen

Object Behavior Diagrams stellen eine graphische Notation für den Entwurf objektorientierter Datenbanken zur Verfügung. Ziel ist die Beschreibung von Datenbank-Schemas, die nicht nur Objektstrukturen abbilden, sondern auch Objektverhalten. Dafür wird die getrennte Beschreibung von statischen (*Object Diagrams*) und dynamischen Eigenschaften (*Behavior Diagrams*) angestrebt. *Behavior Diagrams* sind dabei in *Life Cycles* zur Beschreibung von Objektlebenszyklen und *Workflow Diagrams* für die Modellierung des globalen Prozesses unterteilt. *Life Cycles* werden auf Basis von Petri-Netzen mit einer Menge von Zuständen, Aktivitäten und Kanten, die diese verbinden, modelliert. Die *Life Cycle*-Diagramme beschreiben damit die Ausführungsreihenfolge von Aktivitäten. Die Vernetzung der einzelnen Life Cycles geschieht in *Object Behavior Diagrams* über Synchronisationen zwischen Zuständen und Aktivitäten verschiedener Life Cycles. Dabei stehen verschiedene Synchronisationsmechanismen zur Verfügung. Da dieser Ansatz nicht von einer Interpretation der Modelle, sondern deren Implementierung ausgeht, wird in einer technischen Ebene die Kommunikation in Diagrammen für *Activity Specification* und *Activity Realization* detailliert. Die Verknüpfung von OLCs erfolgt jedoch starr auf Modellebene. Die Konfigurierbarkeit der Synchronisation durch Relationen, wie sie CoRepro ermöglicht, ist damit nicht gegeben. Des Weiteren betrachtet der Ansatz nur die Modellierung, nicht aber die Ausführung von Prozessstrukturen. So sind weder Korrektheitskriterien, noch die Aspekte der Modellierungsunterstützung näher beschrieben.

[PS98] beschreibt einen Ansatz, der davon ausgeht, dass *ein* Objekt in verschiedenen Prozessen bearbeitet wird, wobei diese Prozesse durch unterschiedliche Objektlebenszyklen für dieses Objekt repräsentiert werden. Derartige Objektlebenszyklen definieren somit unterschiedliche Sichten auf ein Objekt im Hinblick auf seine Verarbeitung in einem Gesamtprozess. Durch Integration der Objektlebenszyklen lässt sich eine Prozessstruktur, also der Gesamtprozess für dieses Objekt erstellen. Dieselbe Idee wird in [FE98] verfolgt (mit dem Unterschied, dass die Prozesse nicht als Objektlebenszyklen, sondern als Zustandsautomaten bezeichnet werden).

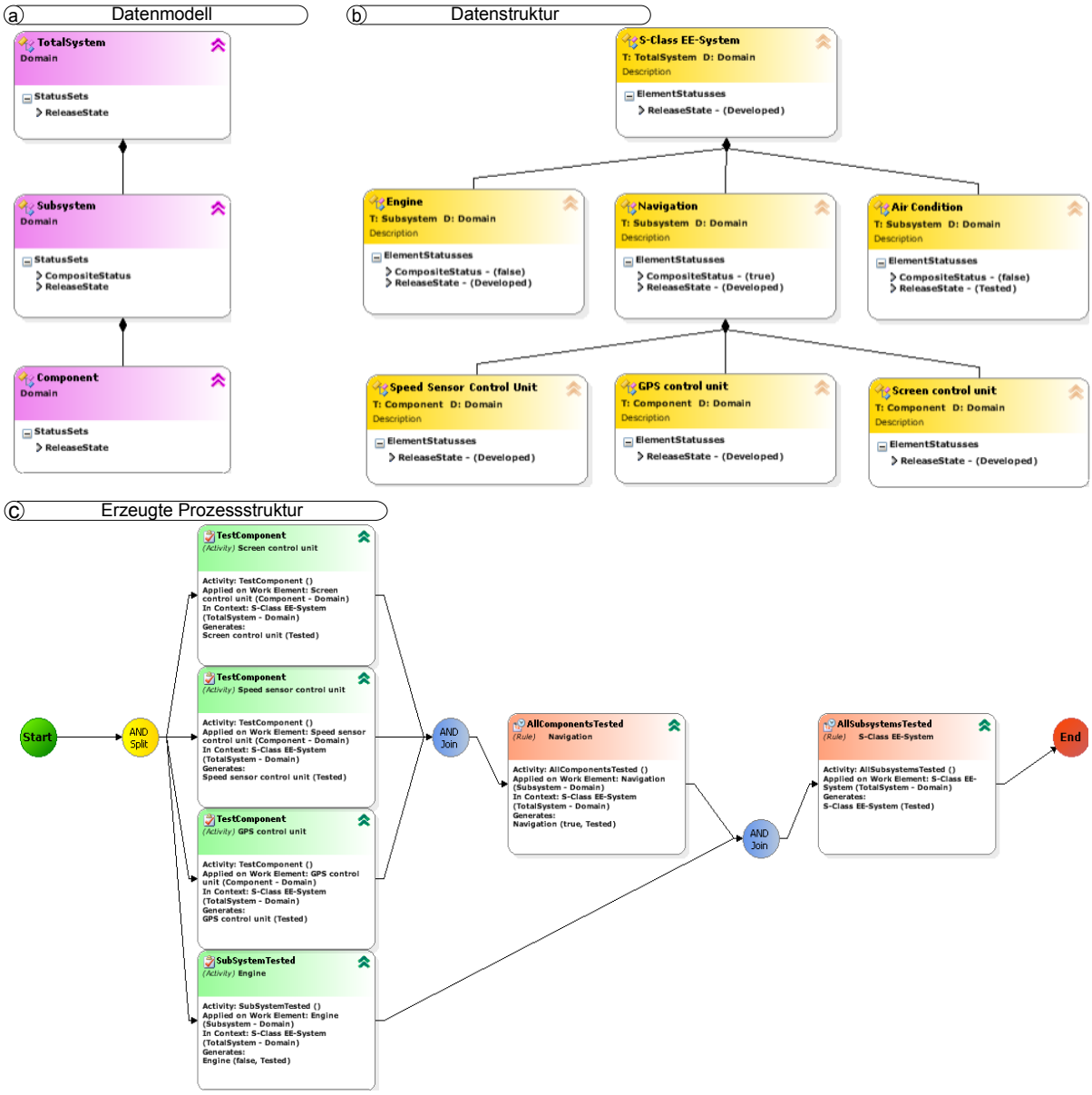


Abbildung 5.11: Modellierung einer datengetriebenen Prozessstruktur in *DBPM*

Mit der Erzeugung einer Prozessstruktur auf Basis gegebener OLCs beschäftigen sich auch [RDHI07, RDHI08] und [RKG06, KRG07, RKG07]. Hierbei werden nicht nur einzelne Objekte, wie in den beiden vorherigen Ansätzen, behandelt. Stattdessen erlauben die Konzepte die Erzeugung eines YAWL-Modells in [RDHI07, RDHI08] bzw. UML-Aktivitätendiagramms mit Datenfluss (inkl. Berücksichtigung von Objektzuständen) in [RKG06, KRG07, RKG07] als Gesamtprozess auf Basis synchronisierter OLCs für verschiedene Objekte. Des Weiteren unterstützt der Ansatz aus [RKG06, KRG07, RKG07] die umgekehrte Richtung, also aus einer gegebenen Prozessstruktur in Form eines UML-Aktivitätendiagramms einzelne OLCs zu extrahieren. Die automatische Erzeugung von Prozessstrukturen auf Basis einer gegebenen Datenstruktur wird jedoch nicht unterstützt. Der Ansatz bietet jedoch durch die Identifikation der OLCs in einer bestehenden Prozessstruktur entsprechende Konzepte für die initiale Modellierung des LCM.

Ein weiterer Ansatz, der sich mit der Beziehung zwischen Daten und Prozessen beschäftigt, ist in [HHH+09] beschrieben. Ziel des Konzepts ist die Erzeugung einer Datenstruktur (basierend auf *Jackson-Datentypen*) aus einer gegebenen Prozessstruktur (basierend auf Petri-Netzen). Die Idee ist hierbei, dass die Abhängigkeiten zwischen im Prozess genutzten Daten automatisch durch die formale Definition einer Abbildungsvorschrift abgeleitet werden können. Dafür werden die Kontrollflusskonstrukte von Prozessen auf Datenstrukturen abgebildet. Der Ansatz geht damit implizit auch von einer Prozessstruktur aus, die auf einer Datenstruktur beruht. Die Zielsetzung ist jedoch entgegengesetzt zu der von Corepro. Das Konzept der direkten Abbildungsvorschrift kann für die Erzeugung einer Prozessstruktur aus einer gegebenen Datenstruktur nicht genutzt werden, da die Datenstruktur „alleine“ nicht genügend Informationen zur Erzeugung des Prozesses (also z.B. die auszuführenden Prozessschritte) enthält. Weiter ist die Synchronisation verschiedener OLCs nicht durchgängig mittels Jackson-Datentypen beschreibbar [HHH+09].

Der WEP-Ansatz vereint die aktivitätenorientierte mit der zielorientierten Modellierung von Prozessen bzw. Prozessstrukturen (vgl. Abschnitt 3.2.1). Dafür stehen Standardkonstrukte (Sequenzen, bedingte Verzweigungen, Schleifen, Nebenläufigkeit), aber auch erweiterte Konstrukte zur Verfügung. Der Ansatz beschreibt weiter ein objektorientiertes Datenmodell. Objekte besitzen Attribute und Relationen (Beziehungen) zu anderen Objekten. Jedes Datenobjekt hat verschiedene vordefinierte Systemzustände (*unset*, *locked*, *preRelease* und *released*). Besonderheit des Datenmodells ist die Definition von individuellen Qualitätsstufen, die für die Prozesssteuerung genutzt werden. Ein Objekt kann eine bestimmte Qualitätsstufe nur dann erreichen, wenn auch seine Sub-Objekte das geforderte Qualitätskriterium erfüllen. Das Konstrukt *variable Parallelität* in WEP ermöglicht die dynamische Verarbeitung eines komplexen Objekts, ähnlich wie die multiple Instanziierung der Workflow Patterns [RHEA04, RHAM06, RR06]. Dafür wird eine Datenstruktur bestehend aus Subobjekten mit einer jeweiligen Qualitätsstufe als Eingang definiert, sowie die geforderte Qualitätsstufe der Ausgabe am Ende des variablen Blocks. Je nach geforderter Ausgabe kann eine Erhöhung der Qualitätsstufe eines Subobjekts oder das Erzeugen oder Entfernen von Subobjekten notwendig sein. Dafür wird für jede Subklasse der notwendige Bearbeitungsschritt modelliert. Zur Laufzeit wird dann das Objekt in die gewünschten Bestandteile zerlegt, indem für alle Subobjekte geprüft wird, ob auf sie die Qualitätsstufe zutrifft. Anschließend wird für jede gefundene Komponente ein paralleler Zweig zur Erreichung des Ziels ausgeführt. Die grundsätzlichen Einschränkungen der zielorientierten Prozessmodellierung mittels Planungstechniken (z.B. Ausdrucksmächtigkeit und explizite Synchronisierung der Prozesse für unterschiedliche (Sub-)Objekte) bleiben jedoch erhalten (vgl. Abschnitt 3.2.2).

Ein weiterer Ansatz, der sich mit der integrierten Modellierung von Daten und Prozessen beschäftigt, ist *Operational Specification* (OPS) [NC03, LBW07]. Hier wird auf Basis von Geschäftsobjekten (*Business Artifacts*; z.B. Bestellung eines Produkts oder ein Versicherungsfall) die Prozessstruktur modelliert. Für jedes Geschäftsobjekt wird ein *Life Cycle*, bestehend aus Aktivitäten, angelegt. Life Cycles können über lesenden bzw. schreibenden Zugriff (*transport* und *request*) auf Daten-Repositories oder direkt über Aktivitätenbeziehungen synchronisiert werden. Die Gesamtheit der Lebenszyklen aller Geschäftsobjekte und der Interaktionen zwischen ihnen definieren damit die Prozessstruktur. Der Ansatz erlaubt damit zwar die Verknüpfung von Daten und Prozessen, die automatische Erstellung einer Prozessstruktur über eine gegebene Datenstruktur wird jedoch nicht unterstützt. Zudem ist der Ansatz auf die Modellierung von Abläufen ausgerichtet, nicht jedoch auf deren Ausführung.

AHEAD erlaubt die integrierte Modellierung von Daten und Prozessen (vgl. Abschnitt 3.2.3). Hierfür stehen ein Konfigurations- bzw. Produktmodell (*Configuation Management*, CoMa) sowie ein Aktivitätenmodell (*Dynamic Task Nets*, DYNAMITE) zur Verfügung. Das Daten- bzw. Produktmodell CoMa dient der Verwaltung von Entwicklungsdokumenten ähnlich wie in einem PDM-System. Mit dem Modell lassen sich Abhängigkeiten zwischen Dokumenten abbilden, genauso wie deren Gruppierung in Konfigurationen. Das Modell unterstützt Versionierung und Variantenbildung für Entwicklungsdokumente und Konfigurationen. Der Dokumentgraph beschreibt Beziehungen zwischen Entwicklungsdokumenten. Sie werden genutzt, um beispielsweise in der Software-Entwicklung die Ablaufreihenfolge für Importbeziehungen festzulegen. Die Struktur des Dokumentgraphen ist nicht im Modell festgelegt. Der Konfigurationsgraph besteht aus Versionen von Dokumenten oder geschachtelten Konfigurationen sowie den vorliegenden Abhängigkeiten zwischen den Objekten. Die Struktur des Konfigurationsgraphen ist im CoMa-Modell festgelegt, kann aber domänenspezifisch angepasst werden. Die Verbindung zwischen Produkt- und Aktivitätenmodell geschieht durch Definition der Datenflüsse im Prozess. Datenflüsse sind durch Ein- und Ausgabeparameter für Aufgaben definiert. Sie verweisen auf versionierte Datenobjekte. Die datengetriebene Erzeugung eines Prozesses wird von diesem Ansatz nicht adressiert. In Abbildung 5.12a ist ein Prozess dargestellt, in dem der Prozessschritt *Redesign Application* als Netzerweiterung deklariert ist. Dieser Teil kann dann zur Laufzeit auf Basis eine gegebenen Datenstruktur aufgebaut werden, allerdings manuell (siehe Abbildung 5.12b und c).

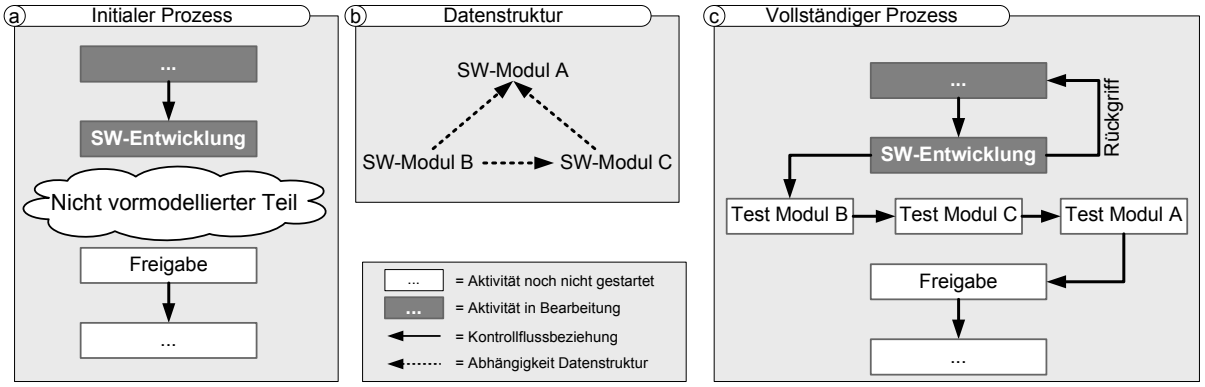


Abbildung 5.12: Modellierung und Ausführung von Aufgabennetzen in DYNAMITE [Wes01]

Die Modellierung von Objektlebenszyklen wird ferner mit *Object-Process Methodology* [Dor00], *Object Coordination Nets* (OCoN) [WG00, WWG01], *Life Cycle Inheritence* [AB97] und *Specia-*

lization of Object Life Cycles [SS02] noch in weiteren Ansätzen adressiert. Sie decken jedoch die Anforderungen nur teilweise ab und werden daher nicht näher vorgestellt.

5.6 Zusammenfassung

In diesem Kapitel haben wir die Konzepte für die datengetriebene Modellierung großer Prozessstrukturen vorgestellt. Im Gegensatz zu Ansätzen aus der Praxis und Wissenschaft erlauben die Konzepte nicht nur eine signifikante Reduktion der Modellierungsaufwände. Sie garantieren auch die dynamische Korrektheit derartiger Prozessstrukturen per Konstruktion und sichern damit dem Ansatz eine hervorragende Skalierbarkeit zu.

Die Reduktion des Modellierungsaufwands durch die in den Abschnitten 5.2 und 5.3 beschriebenen Mechanismen können anhand einer Beispielrechnung aufgezeigt werden [MRH07]. Beispiel 5.1 beschreibt zu diesem Zweck ein plausibles Szenario für die Bildung eines Release in der E/E-Entwicklung (vgl. Abschnitt 2.1). Der Aufwand für die Erstellung der Prozessstruktur mit mehr als 1300 Prozessen und 1500 Prozess-Synchronisationen kann durch den CoRePro-Ansatz um mehr als 90% reduziert werden (vgl. Tabelle 5.1). Angemerkt sei an dieser Stelle, dass diese Reduktion bereits für die Erstellung der ersten Instanz der Prozessstruktur erreicht wird. Die Erstellung weiterer Prozessstrukturen für andere Releases bedarf in CoRePro eines noch weitaus geringeren Modellierungsaufwands, da lediglich eine Datenstruktur angelegt werden muss, während die Modellebene unverändert bleiben kann.

Beispiel 5.1 (Plausibles Zahlenbeispiel aus der E/E-Entwicklung)
Ein Release in der E/E-Entwicklung umfasst (abhängig von der Baureihe und dem Umfang des Release) 200 bis 300 Komponenten. Sie können in Stammkomponenten und Varianten eingeordnet werden (z.B. Fahrerairbag als Stammkomponente und Beifahrerairbag als Variante). Stammkomponenten können weiter in verschiedene Kategorien unterschieden werden. Eine Multimedia-Stammkomponente erfordert beispielsweise andere Absicherungsschritte als eine sicherheitsrelevante Komponente. Darüber hinaus sind Komponenten in Systeme gruppiert (vgl. Abschnitt 2.1), welche wiederum Gesamtsystemen verschiedener Releases zugeordnet werden. In einem plausiblen Szenario führt dies zu 20 Objekttypen und 25 Relationstypen, die diese verbinden. Auf Instanzebene resultiert alleine die Zuordnung von Komponenten zu Systemen und Gesamtsystemen in mehr als 200 instanziierten Relationen. Betrachten wir weitere technische Abhängigkeiten (z.B. Kommunikationsbeziehungen zwischen Komponenten), ergeben sich weitere 400 Relationen. Je Objekt müssen zwischen 5 (bei Komponenten) und 20 Prozesse (bei Systemen) koordiniert und geeignet synchronisiert werden. Die resultierende Prozessstruktur umfasst somit mehr als 1300 Prozesse und 1500 Prozess-Synchronisationen (vgl. Tabelle 5.1).

Die Garantie der Korrektheit erzeugbarer Prozessstrukturen erfolgt prinzipiell pessimistisch, da sie per Konstruktion gewährleistet wird. Wir erlauben damit, beliebige Datenstrukturen zu bilden sowie darauf basierend immer korrekte Prozessstrukturen zu erzeugen und garantieren ihre Korrektheit. In der Praxis kann es allerdings notwendig sein, diese pessimistische Prüfung auf Basis der Zyklenfreiheit des LCM „aufzubrechen“. Dies kann der Fall sein, wenn in der Theorie zwar Zyklen erzeugt werden könnten, in der Praxis durch geeignete Einschränkungen der

Tabelle 5.1: Beispielrechnung für die Reduktion des Modellierungsaufwands durch COREPRO

Modellebene			
Datenmodell		**LCM**	
Objekttypen	Relationstypen	Prozesse	Ext. Transitionen pro Relationstyp
20	25	5-20	2-6
Instanzebene			
Datenstruktur		**LCS**	
Objekte	Relationen	Prozesse	Ext. Transitionen
200-300	>600	>1300	>1500

Datenstruktur (z.B. sich gegenseitig ausschließende Relationen im PDM-System) aber verhindert wird. Theoretisch kann – mit den diskutierten Nachteilen – auch immer direkt die erzeugte Prozessstruktur auf Zyklenfreiheit geprüft werden. Um hier den Aufwand zu reduzieren, könnte zusätzlich im Modell bereits eine Kennzeichnung kritischer externer Transitionen erfolgen, so dass nur diese in der generierten Prozessstruktur dann überprüft werden müssen.

Die Konzepte aus den Kapiteln 4 und 5 decken die Modellierung und Ausführung datengetriebener Prozessstrukturen ab. In der Praxis sind diese Konzepte allerdings nicht ausreichend, denn wir sehen uns mit häufigen Änderungen der zugrundeliegenden Datenstrukturen konfrontiert. Zur Modellierzeit können derartige Änderungen durch eine erneute Erzeugung der Prozessstruktur direkt umgesetzt werden. Zur Laufzeit ist dieses Vorgehen jedoch nicht erwünscht, da es unter anderem den Verlust der Laufzeitmarkierung der Prozessstruktur zur Folge hat. Die folgenden beiden Kapitel beschäftigen sich daher mit der dynamischen Adaption von Prozessstrukturen und geeigneten Methoden zur Behandlung von Ausnahmen, die während der Ausführung oder der Adaption entstehen können.

6

Datengetriebene Adaption von Prozessstrukturen

In den Kapiteln 4 und 5 haben wir die Grundlagen und Konzepte sowohl für die Modellierung als auch die Ausführung datengetriebener Prozessstrukturen vorgestellt. Die untersuchten Fallbeispiele (vgl. Kapitel 2) haben jedoch gezeigt, dass modellierte Prozessstrukturen nicht immer wie geplant ausgeführt werden können. Da datengetriebene Prozessstrukturen typischerweise eine sehr lange Ausführungsdauer aufweisen, können zur Laufzeit Änderungen der Datenstruktur, zum Beispiel das Entfernen eines Objekts, vorkommen. Sie erfordern eine angemessene Adaption der zugehörigen Prozessstruktur. In diesem Kontext ist nicht nur die Definition geeigneter Änderungsoperationen erforderlich, sondern auch weiterhin die Gewährleistung der Korrektheit adaptierter Prozessstrukturen. Hierfür müssen Konsistenzkriterien definiert und ihre adäquate Überprüfung im Rahmen dynamischer Adaptionen untersucht werden.

Kapitel 6 gliedert sich wie folgt: Abschnitt 6.1 präsentiert Szenarien aus dem E/E-Release-Management, die im Rahmen der Adaption datengetriebener Prozessstrukturen eine wichtige Rolle spielen, und leitet aus ihnen Anforderungen an die IT-Unterstützung ab. Abschnitt 6.2 stellt die generell notwendigen Operationen für die Änderung einer Datenstruktur vor und setzt sie mit entsprechenden Operationen zur Adaption der zugehörigen Prozessstruktur in Beziehung. Abschnitt 6.3 betrachtet die dynamischen Aspekte bei der Anwendung der zuvor eingeführten Operationen und führt geeignete Konsistenzkriterien ein. Abschnitt 6.4 diskutiert verwandte Arbeiten zur Adaption von Prozessstrukturen. Abschnitt 6.5 fasst die Inhalte des Kapitels zusammen.

6.1 Einleitung

Die Adaption datengetriebener Prozessstrukturen wurde bereits an verschiedenen Stellen dieser Arbeit motiviert. Während die Änderung einer Datenstruktur ohne große Einschränkungen

durchgeführt werden kann, ist die Adaption der zugehörigen Prozessstruktur weitaus komplexer. Wird beispielsweise ein Subsystem (inkl. seiner Relationen mit anderen Komponenten) aus einem Release entfernt, müssen aus der Prozessstruktur die entsprechenden Prozesse sowie deren Synchronisationsbeziehungen entfernt (und ggf. terminiert) werden. In großen, unübersichtlichen Prozessstrukturen mit hunderten oder tausenden von Prozessen und Synchronisationsbeziehungen ist die manuelle Adaption jedoch sehr aufwändig und fehleranfällig. Ziel muss hier eine geeignete IT-Unterstützung sein, die den Entwickler bei der Adaption der Prozessstruktur unterstützt. Ein naiver Ansatz wäre die erneute Generierung der gesamten Prozessstruktur nach Anpassung der Datenstruktur. Diese Lösung reduziert zwar den Änderungsaufwand erheblich, da eine Änderung der Datenstruktur direkt auf die Prozessstruktur propagiert wird, zur Laufzeit würde die erneute Generierung der Prozessstruktur jedoch zu einem Verlust der Laufzeitinformation (d.h. der Laufzeitmarkierung) führen und damit einen Neustart der Prozessstruktur erfordern. Es ist daher erforderlich, dynamische Änderungen der Datenstruktur durch geeignete Operationen gezielt auf die Prozessstruktur zu übertragen. Damit werden einerseits die Adaption der Prozessstruktur auf hoher Abstraktionsebene erlaubt, andererseits die Laufzeitinformationen so weit wie möglich erhalten. Allerdings ist zu beachten, dass eine dynamische Änderung der Prozessstruktur deren Korrektheit potenziell gefährdet und Inkonsistenzen hinsichtlich der Laufzeitmarkierungen verursachen kann. Die IT-Unterstützung muss hier geeignete Konzepte liefern, um die Korrektheit in jedem Fall zu gewährleisten sowie um Inkonsistenzen zu erkennen und zu verhindern.

6.1.1 Szenarien

Die Analyse der E/E-Entwicklungsprozesse hat gezeigt, dass Änderungen der Datenstruktur in der Praxis sehr häufig auftreten. Bei einer solchen Änderung ist es immer notwendig, auch die zugehörige Prozessstruktur anzupassen, um Aufwände und damit Kosten zu reduzieren. Im Folgenden werden repräsentative Szenarien diskutiert, die die Notwendigkeit zur Unterstützung der datengetriebenen Adaption unterstreichen.

Szenario 6.1 (Hinzufügen oder Entfernen einer Komponente)
Der Durchlauf einer gesamten Release-Management-Prozessstruktur kann mehrere Wochen betragen. Über diesen Zeitraum können Anpassungen an der Datenstruktur notwendig werden. Wird beispielsweise ein gravierender Fehler in einer Komponente festgestellt, kann aufgrund fachlicher Kriterien entschieden werden, die Komponente aus dem Release wieder zu entfernen. Dann sollen für sie keine weiteren Prozesse mehr ausgeführt bzw. derzeit für sie laufende Prozesse beendet werden (vgl. Abbildung 6.1). Ferner soll die Aufnahme neuer Komponenten in ein Release möglich sein, indem sie der Datenstruktur hinzugefügt werden. Die für die Komponenten erforderlichen Prozesse sollen automatisch in die Prozessstruktur eingefügt werden.

Szenario 6.2 (Austausch einer Komponentenversion)
In vielen Fällen ist das vollständige Entfernen einer fehlerhaften Komponente aus dem Release nicht möglich (z.B. bei zentralen Komponenten) und die Komponente muss durch eine andere, funktionierende Komponente ersetzt werden (vgl. Abbildung 6.1). Die Prozesse (z.B. Tests) müssen in der Regel für diese Komponente erneut ausgeführt werden.[1]

[1] In Kapitel 7 werden weitere Konzepte für die flexible Ausführung vorgestellt.

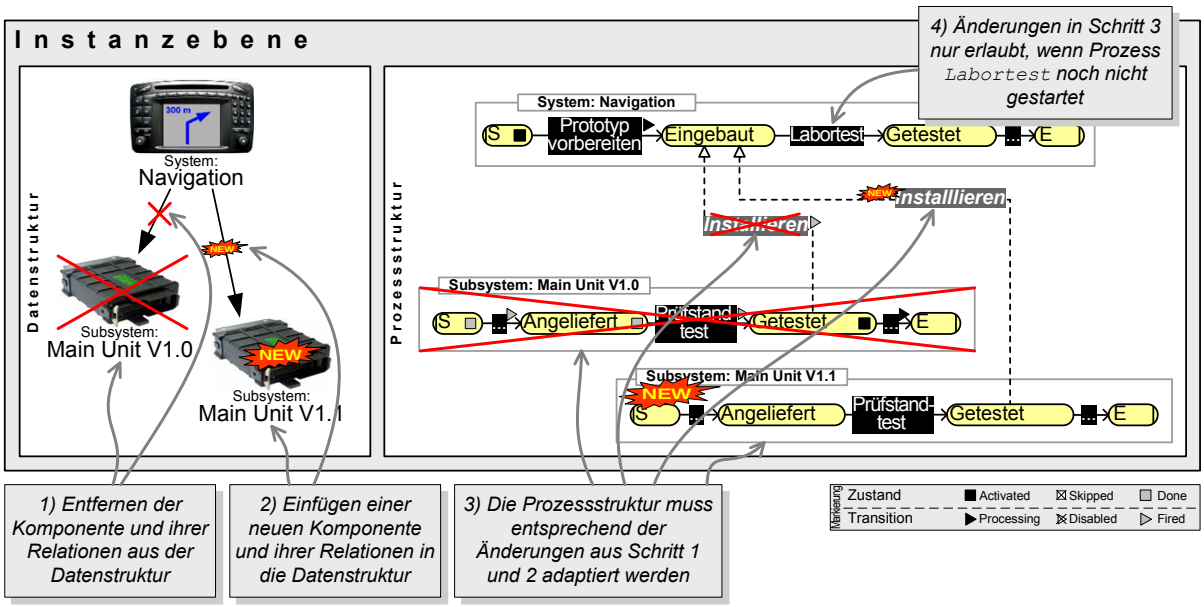


Abbildung 6.1: Beispielszenario für dynamische Adaptionen einer Prozessstruktur

Szenario 6.3 (Hinzufügen oder Entfernen einer Abhängigkeit)
Während der Ausführung der Release-Management-Prozessstruktur kann es nicht nur zum Entfernen oder Hinzufügen von OLCs kommen, sondern auch zur Anpassung der Synchronisationsbeziehungen (d.h. der externen Transitionen zwischen OLCs). Wird eine Relation zwischen zwei Subsystemen dynamisch eingefügt (z.B. aufgrund eines zusätzlichen Nachrichtenaustauschs), müssen der Prozessstruktur entsprechende Abhängigkeiten zwischen ihren Objektlebenszyklen hinzugefügt werden (z.B. um ihre Absicherungsprozesse zu synchronisieren).

Wird eine Abhängigkeit zwischen zwei Komponenten entfernt, etwa wenn eine Komponente `k1` nicht mehr auf die Kommunikation mit einer anderen Komponente `k2` angewiesen ist oder Komponente `k1` entfernt wird (vgl. Szenario 6.1 bzw. Abbildung 6.1), müssen entsprechend auch die externen Transitionen zwischen ihren OLCs entfernt werden.

Szenario 6.4 (Synchronisation nach unabhängigem Start)
Im Rahmen des Release-Managements ist es notwendig, globale Synchronisationsmuster zu realisieren (vgl. Abschnitt 4.1.2). In bestimmten Fällen ist allerdings auch der unsynchronisierte Start eines Objektlebenszyklus erwünscht. Ist beispielsweise noch keine Systemstruktur gebildet, sollen die Objektlebenszyklen für Subsysteme trotzdem (unsynchronisiert) gestartet werden können. Innerhalb einer datengetriebenen Prozessstruktur könnte dies realisiert werden, indem OLCs zunächst ohne Synchronisationsbeziehungen gestartet und durch *spätes* Einfügen der Synchronisationsbeziehungen zur Laufzeit synchronisiert werden. Wird die Abhängigkeit *zu spät* eingefügt, zum Beispiel wenn das Release aufgrund der fehlenden Abhängigkeit bereits freigegeben wurde, muss dies entsprechend erkannt und angezeigt werden.

6.1.2 Anforderungen

Aus den im vorherigen Abschnitt definierten Szenarien lassen sich die folgenden IT-Anforderungen ableiten. Um eine sinnvolle Aufteilung der Anforderungen zu erreichen, stellen wir zunächst die *statische* Adaption von Daten- und Prozessstrukturen vor (vgl. Anforderungen 6.1 und 6.2), um dann auf deren *dynamische* Eigenschaften einzugehen (vgl. Anforderung 6.3).

Anforderung 6.1 (Änderung von Daten- und Prozessstrukturen)
Die Adaption von Prozessstrukturen erfordert geeignete Änderungsoperationen sowohl für Daten- als auch Prozessstrukturen. Die bereitgestellte Menge an Operationen muss vollständig sein, also die Überführung einer gegebenen Daten- oder Prozessstruktur in eine beliebige andere Daten- oder Prozessstruktur erlauben, die auf dem gleichen Modell beruht.

Anforderung 6.2 (Datengetriebene Adaption von Prozessstrukturen)
In Kapitel 5 wurde die starke Beziehung zwischen Daten und Prozessen bereits dazu verwendet, eine Prozessstruktur auf Basis einer gegebenen Datenstruktur automatisch erzeugen zu können. Die gewonnenen Erkenntnisse sollen auch für die Adaption von Prozessstrukturen genutzt werden: Zum einen sollen Änderungen auf hoher Abstraktionsebene, nämlich über die Anpassung der Datenstruktur erlaubt werden, zum anderen soll die *Integrität* von Daten- und Prozessstrukturen aufrecht erhalten werden.[2]

Anforderung 6.3 (Dynamische Durchführung von Adaptionen der Prozessstruktur)
Die Durchführung von Änderungen soll insbesondere zur Laufzeit ermöglicht werden (vgl. Szenarien in Abschnitt 6.1.1). Die Adaption einer Prozessstruktur zur Laufzeit muss deren aktuellen Ausführungszustand berücksichtigen. Es besteht die Gefahr vom operationalen Modell abzuweichen, indem ungültige Ausführungszustände erzeugt werden (z.B. das Hinzufügen einer externen Transition, deren Zielzustand bereits durchlaufen wurde). Dies darf jedoch nicht die Korrektheit der Prozessstruktur gefährden (z.B. Verklemmungsfreiheit; vgl. Kapitel 4). Es müssen daher Kriterien definiert werden, mit denen sich derartige Szenarien erkennen und verhindern lassen, so dass die korrekte Ausführung einer adaptierten Prozessstruktur gewährleistet werden kann.

6.2 Datengetriebene Adaption der Prozessstruktur: Statische Sicht

Die Adaption datengetriebener Prozessstrukturen erfordert sowohl die Änderung der Datenstruktur (Initiator einer datengetriebenen Änderung) als auch die Anpassung der zugehörigen Prozessstruktur. Eines der Ziele der datengetriebenen Adaption von Prozessstrukturen ist die Durchführung von Änderungen auf hoher Abstraktionsebene (vgl. Anforderung 6.2). Dazu führen wir in diesem Abschnitt einen vollständigen Satz an *Basisoperationen* zur Änderung von Datenstrukturen ein, mit denen sich eine Datenstruktur in eine beliebige andere, auf demselben Modell beruhende Datenstruktur überführen lässt. Für die Adaption von Prozessstrukturen definieren wir ebenso einen vollständigen Satz an Basisoperationen (vgl. Anforderung 6.1).

[2]Die von der Datenstruktur unabhängige Adaption einer Prozessstruktur ist keine Anforderung.

Entsprechend der in Kapitel 5 genutzten Beziehung zwischen Daten- und Prozessstruktur lassen sich Änderungen, die auf einer Datenstruktur durchgeführt werden, in Adaptionen der entsprechenden Prozessstruktur *übersetzen*. Hierfür setzen wir die Anwendung von Änderungsoperationen der Datenstruktur mit Operationen zur Adaption der Prozessstruktur in Beziehung und erhalten *öffentliche Änderungsoperationen* zur datengetriebenen Adaption einer Prozessstruktur (siehe Abschnitt 6.2.3). Sie stehen dem Benutzer zur Verfügung, um eine Änderung der Datenstruktur vorzunehmen und diese automatisch auf die entsprechende Prozessstruktur zu übertragen. Dieser Mechanismus gewährleistet ferner die Konsistenz zwischen Daten- und Prozessstruktur im Sinne der Modellierung. Die öffentlichen Änderungsoperationen können auch für die Definition *höherwertiger Änderungsoperationen* genutzt werden (siehe Abschnitt 6.2.4).

In diesem Abschnitt erfolgt zunächst die Betrachtung des statischen Falls, d.h. wir vernachlässigen die Laufzeitmarkierungen bei Anwendung der Änderungsoperationen. Ihre Betrachtung folgt in Abschnitt 6.3.

6.2.1 Basisoperationen zur Änderung einer Datenstruktur

Datenstrukturen bestehen aus Objekten und Relationen (vgl. Abschnitt 5.2.2). Um die Vollständigkeit der Änderungsoperationen sicherzustellen, also die Überführung einer Datenstruktur in eine beliebige andere Datenstruktur basierend auf dem selben Datenmodell zu gewährleisten, werden Operationen für das Hinzufügen und Entfernen sowohl von Objekten als auch Relationen benötigt (vgl. Szenario 6.1). Sie sind in den Änderungsoperationen CO1, CO2, CO3 und CO4 inklusive der jeweiligen Vor- und Nachbedingungen zur Erhaltung der strukturellen Korrektheit formal definiert. Abbildung 6.2 illustriert ihre Anwendung.

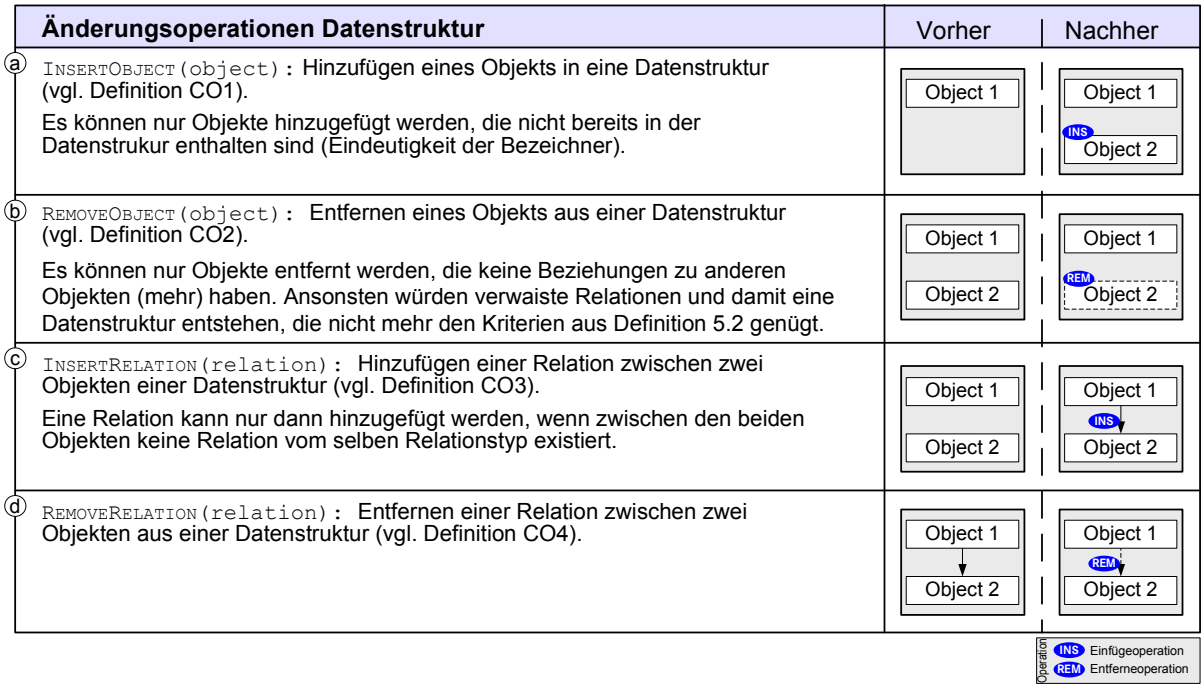


Abbildung 6.2: Basisoperationen zur Änderung einer Datenstruktur

Änderungsoperation CO1 (INSERTOBJECT: Hinzufügen eines Objekts)
Seien $ds = (O, R) \in \mathcal{DS}$ eine Datenstruktur und $o \notin O$ ein Objekt, das noch nicht in ds enthalten ist. Wird die Änderungsoperation INSERTOBJECT*(o) bezogen auf ds ausgeführt, wird o der Menge O hinzugefügt. Wir erhalten eine Datenstruktur $ds' = (O', R')$ mit*

$$O' := O \cup \{o\} \wedge R' \equiv R$$

Änderungsoperation CO2 (REMOVEOBJECT: Entfernen eines Objekts)
Seien $ds = (O, R) \in \mathcal{DS}$ eine Datenstruktur und $o \in O$ ein Objekt, das in ds enthalten ist. Wenn gilt: $\nexists\, r = (o_1, rt, o_2) \in R : o_1 = o \vee o_2 = o$, dann kann die Änderungsoperation REMOVEOBJECT*(o) bezogen auf ds ausgeführt werden und o wird aus der Menge O entfernt. Wir erhalten eine Datenstruktur $ds' = (O', R')$ mit*

$$O' := O \backslash \{o\} \wedge R' \equiv R$$

Änderungsoperation CO3 (INSERTRELATION: Hinzufügen einer Relation)
Seien $ds = (O, R) \in \mathcal{DS}$ eine Datenstruktur und $r \notin R, r = (o_1, rt, o_2)$ eine Relation, die noch nicht in ds enthalten ist. Wird die Änderungsoperation INSERTRELATION*(r) bezogen auf ds ausgeführt, wird r der Menge R hinzugefügt. Wir erhalten eine Datenstruktur $ds' = (O', R')$ mit*

$$O' \equiv O \wedge R' := R \cup \{r\}$$

Änderungsoperation CO4 (REMOVERELATION: Entfernen einer Relation)
Seien $ds = (O, R) \in \mathcal{DS}$ eine Datenstruktur und $r \in R$ eine Relation, die in ds enthalten ist. Wird die Änderungsoperation REMOVERELATION*(r) bezogen auf ds ausgeführt, dann wird r aus der Menge R entfernt. Wir erhalten eine Datenstruktur $ds' = (O', R')$ mit*

$$O' \equiv O \wedge R' := R \backslash \{r\}$$

6.2.2 Basisoperationen zur Adaption einer Prozessstruktur

Wie für Datenstrukturen, wollen wir auch für Prozessstrukturen einen vollständigen Satz an Operationen definieren. Hierbei ist nicht das Ziel, dynamische Änderungen innerhalb einzelner Prozesse oder OLCs durchzuführen (dies wird u.a. in [Tee96, Tee98, RD98, RRD04a, RRD04b, WRR07, WRRM08, WSR09] behandelt), sondern den Operationen zur Änderung der Datenstruktur die entsprechenden Operationen zur Adaption der zugehörigen Prozessstruktur „gegenüber“ zu stellen. Dazu werden Operationen zum Hinzufügen und Entfernen sowohl von OLCs, als auch externer Transitionen benötigt.

Wir betrachten in diesem Abschnitt zunächst nur die statische Adaption einer Prozessstruktur. Bei der Definition entsprechender Änderungsoperationen ist aber bereits zu beachten, dass Änderungen zur Laufzeit nur dann möglich sein sollen, wenn die Prozessstruktur noch nicht beendet ist. Damit verhindern wir, dass Prozessstrukturen, die in übergeordneten Prozessen eingesetzt sind, im Nachhinein noch verändert werden (vgl. Szenario 4.5 auf Seite 52). Des Weiteren

müssen bei Anwendung der Einfügeoperationen die OLCs bzw. externen Transitionen in ihre Initialmarkierung versetzt werden, um sie anschließend entsprechend der in Kapitel 4 definierten operationalen Semantik ausführen zu können.

Die Änderungsoperationen CO5, CO6, CO7 und CO8 beschreiben die Möglichkeiten zur Adaption von Prozessstrukturen formal und berücksichtigen die genannten Vor- und Nachbedingungen. Abbildung 6.3 illustriert ihre Anwendung.

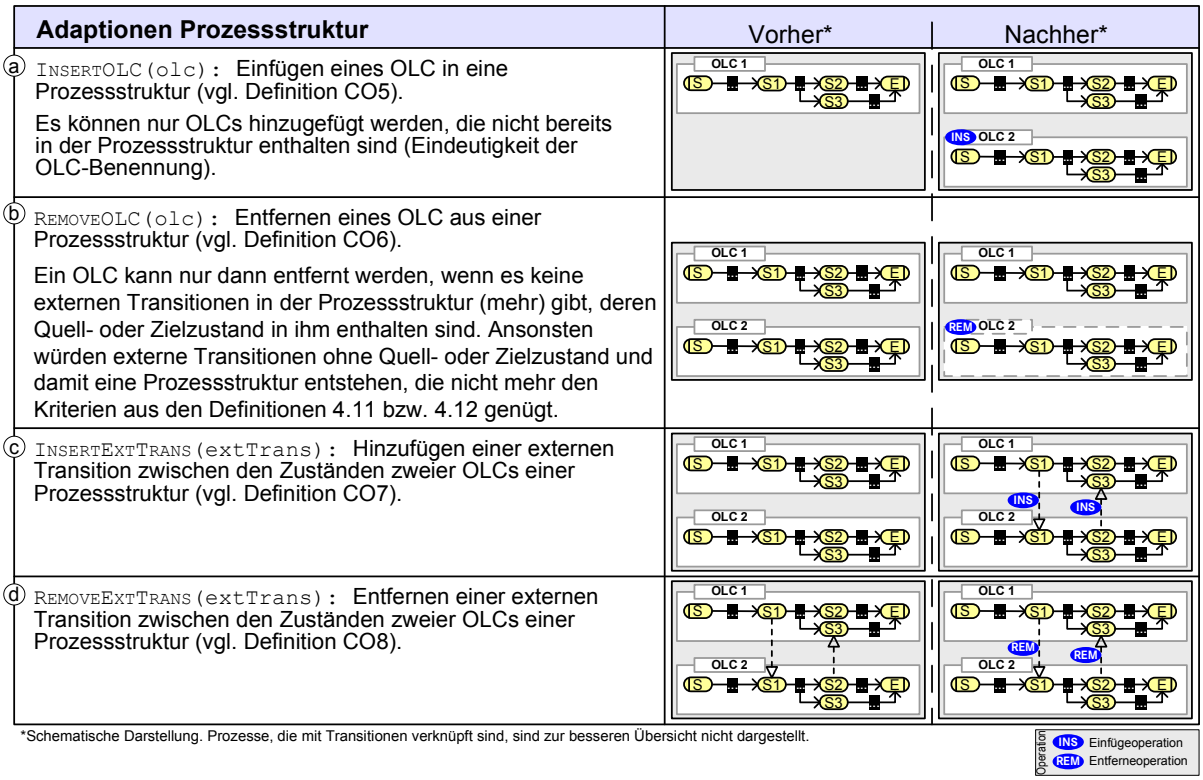


Abbildung 6.3: Basisoperationen zur Adaption einer Prozessstruktur

Änderungsoperation CO5 (InsertOLC: Hinzufügen eines OLC)
Seien $ps = (OLC, EST) \in \mathcal{PS}$ *eine Prozessstruktur und* $olc \notin OLC$ *ein Object Life Cycle, der noch nicht in ps enthalten ist. Ferner gelte:* $PP(ps) \neq$ Terminated. *Wird die Änderungsoperation* InsertOLC(olc) *auf ps angewendet, dann wird olc der Menge OLC hinzugefügt. Wir erhalten eine Prozessstruktur* $ps' = (OLC', EST')$ *mit*

$$OLC' := OLC \cup \{olc\} \wedge EST' \equiv EST$$

Des Weiteren wird olc durch Initialisierungsregel IR1 (siehe Seite 67) in die initiale Markierung versetzt.

Änderungsoperation CO6 (REMOVEOLC: Entfernen eines OLC)
Seien $ps = (OLC, EST) \in \mathcal{PS}$ *eine Prozessstruktur und* $olc = (P, V, TS) \in OLC$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$. *Ferner gelte:* $\forall\ e = (s_1, p, s_2) \in EST : s_1 \notin S \wedge s_2 \notin S$ *und* $PP(ps) \neq$ TERMINATED. *Wird die Änderungsoperation* REMOVEOLC*(olc) auf ps angewendet, dann wird olc aus der Menge OLC entfernt.*[a] *Wir erhalten eine Prozessstruktur* $ps' = (OLC', EST')$ *mit*

$$OLC' := OLC \setminus \{olc\} \wedge EST' \equiv EST$$

[a]Der OLC muss hier nicht in eine bestimmte Phase versetzt werden, da er vollständig aus der Prozessstruktur entfernt wird. Wir verzichten auf eine formale Beschreibung des Abbruchs derzeit ausgeführter Prozesse (Prozesszustand RUNNING). Der Verlauf des OLC (vgl. Abschnitt 4.6.4) bleibt erhalten.

Änderungsoperation CO7 (INSERTEXTTRANS: Hinzufügen einer externen Transition)
Seien $ps = (OLC, EST) \in \mathcal{PS}$ *eine Prozessstruktur und* $olc_i = (P_i, V_i, TS_i) \in OLC, i = 1, 2$ *zwei unterschiedliche Object Life Cycles mit den Transitionssystemen* $TS_i = (S_i, T_i, s_{start_i}, s_{end_i}), i = 1, 2$. *Ferner gelte:* $PP(ps) \neq$ TERMINATED. *Wird die Änderungsoperation* INSERTEXTTRANS*(e) mit* $e = (s_1, p, s_2) \notin EST$ *angewendet, dann wird eine neue externe Transition zwischen* $s_1 \in S_1$ *und* $s_2 \in S_2$ *mit Prozess p der Prozessstruktur hinzugefügt. Wir erhalten eine Prozessstruktur* $ps' = (OLC', EST')$ *mit*

$$OLC' \equiv OLC \wedge EST' := EST \cup \{e\}$$

Des Weiteren wird die Markierung von e auf $EM(e) =$ WAITING *gesetzt.*

Änderungsoperation CO8 (REMOVEEXTTRANS: Entfernen einer externen Transition)
Seien $ps = (OLC, EST) \in \mathcal{PS}$ *eine Prozessstruktur und* $olc_i = (P_i, V_i, TS_i) \in OLC, i = 1, 2$ *zwei unterschiedliche Object Life Cycles mit den Transitionssystemen* $TS_i = (S_i, T_i, s_{start_i}, s_{end_i}), i = 1, 2$. *Ferner gelte:* $PP(ps) \neq$ TERMINATED. *Wird die Änderungsoperation* REMOVEEXTTRANS*(e) mit* $e = (s_1, p, s_2) \in EST$ *angewendet, dann wird e aus der Menge EST entfernt.*[a] *Wir erhalten eine Prozessstruktur* $ps' = (OLC', EST')$ *mit*

$$OLC' \equiv OLC \wedge EST' := EST \setminus \{e\}$$

[a]Wir verzichten auf eine formale Beschreibung des Abbruchs derzeit ausgeführter Prozesse (Prozesszustand RUNNING).

6.2.3 Öffentliche Änderungsoperationen

Durch die in den Abschnitten 6.2.1 und 6.2.2 vorgestellten Basisoperationen ist eine beliebige Anpassung von Daten- und Prozessstrukturen möglich. Die manuelle Adaption der Prozessstruktur ist jedoch aus mehreren Gründen nicht sinnvoll. Zum einen kann durch unkontrollierte Adaption die Korrektheit der Prozessstruktur nicht gewährleistet werden (beispielsweise Erzeugung von Zyklen durch Einfügen einzelner externer Transitionen). Zum anderen erfordert die manuelle Adaption großer Prozessstrukturen viel Aufwand und ein ausreichendes Verständnis der Konzepte. Durch unsachgemäße Anwendung der Basisoperationen kann außerdem die Integrität zwischen

der Daten- und der Prozessstruktur gefährdet werden. Wird beispielsweise ein Objekt aus der Datenstruktur entfernt, ohne den zugehörigen OLC ebenfalls zu entfernen, werden die Prozesse für das entfernte Objekt in der Prozessstruktur weiter ausgeführt und das Konzept der datengetriebenen Prozessstruktur wird ad absurdum geführt. Die Anwendung der Basisoperationen durch den Benutzer ist daher kein gangbarer Weg. Stattdessen muss eine öffentliche Schnittstelle für die Adaption datengetriebener Prozessstrukturen bereitgestellt werden, die derartige Konsequenzen ausschließt.

Ähnlich wie bei der automatischen Erzeugung von Prozessstrukturen können wir auch bei der Beschreibung datengetriebener Änderungen von der Beziehung zwischen Daten- und Prozessstruktur profitieren (vgl. Abschnitt 5.3.2). So kann die Adaption der Prozessstruktur (z.B. Hinzufügen des entsprechenden OLCs) auf Basis der Änderung der Datenstruktur (z.B. Einfügen eines Objekts) durch eine Kopplung der Operationen abgebildet werden.[3]

Abbildung 6.4a zeigt eine Life Cycle Coordination Structure (LCS) mit Datenmodell, LCM, Datenstruktur und erzeugter Prozessstruktur. Auf Grundlage des LCS kann der Bezug zwischen Operationen zur Änderung der Datenstruktur und Operationen zur Adaption der Prozessstruktur hergestellt werden. Hier wird im Rahmen der Modellierung bereits die Abbildung von Datenstrukturen auf Prozessstrukturen vorgenommen, in dem Objekte auf OLCs und Relationen auf OLC-Dependencies (also die Menge der externen Transitionen, die mit einer Relation im LCM verknüpft sind) abgebildet werden (vgl. Abschnitt 5.3.2). Um die Konformität zu diesem Konzept aufrecht zu erhalten, definieren wir für die datengetriebene Adaption von Prozessstrukturen die *öffentlichen Änderungsoperationen* CO9, CO10, CO11 und CO12, die auf Grundlage von Datenstrukturänderungen die Prozessstruktur automatisch anpassen (vgl. Abbildung 6.5). So muss für die `RemoveObject`-Operation auf der Datenstruktur die entsprechende `RemoveOLC`-Operation auf der Prozessstruktur ausgeführt werden, um die Integrität beider Strukturen zu garantieren (vgl. Abbildung 6.4b). Wir „überschreiben“ daher die bekannten Änderungsoperationen für Datenstrukturen CO1, CO2, CO3 und CO4 und erweitern sie um entsprechende Aufrufe von Änderungsoperationen für die Prozessstruktur. Nachfolgend wird jede Änderung der Datenstruktur über die in CO9, CO10, CO11 und CO12 definierten öffentlichen Operationen getätigt. Die Vorbedingungen für die Ausführung der Änderungsoperationen CO5, CO6, CO7 und CO8 werden im Kontext der öffentlichen Änderungsoperationen jeweils als erfüllt angenommen.

Durch Adaption der Prozessstruktur im Rahmen von Datenstrukturänderungen ist sichergestellt, dass wie im Kontext der Generierung einer Prozessstruktur (vgl. Abschnitt 5.3.2) die adaptierte Prozessstruktur ebenfalls keine Zyklen enthält. Sie erfüllt damit die Kriterien für die dynamische Korrektheit weiterhin.

[3] Die umgekehrte Richtung, also die Abbildung von Änderungen der Prozessstruktur auf die Datenstruktur wurde in den betrachteten Szenarien nicht als Anforderung identifiziert und daher nicht betrachtet.

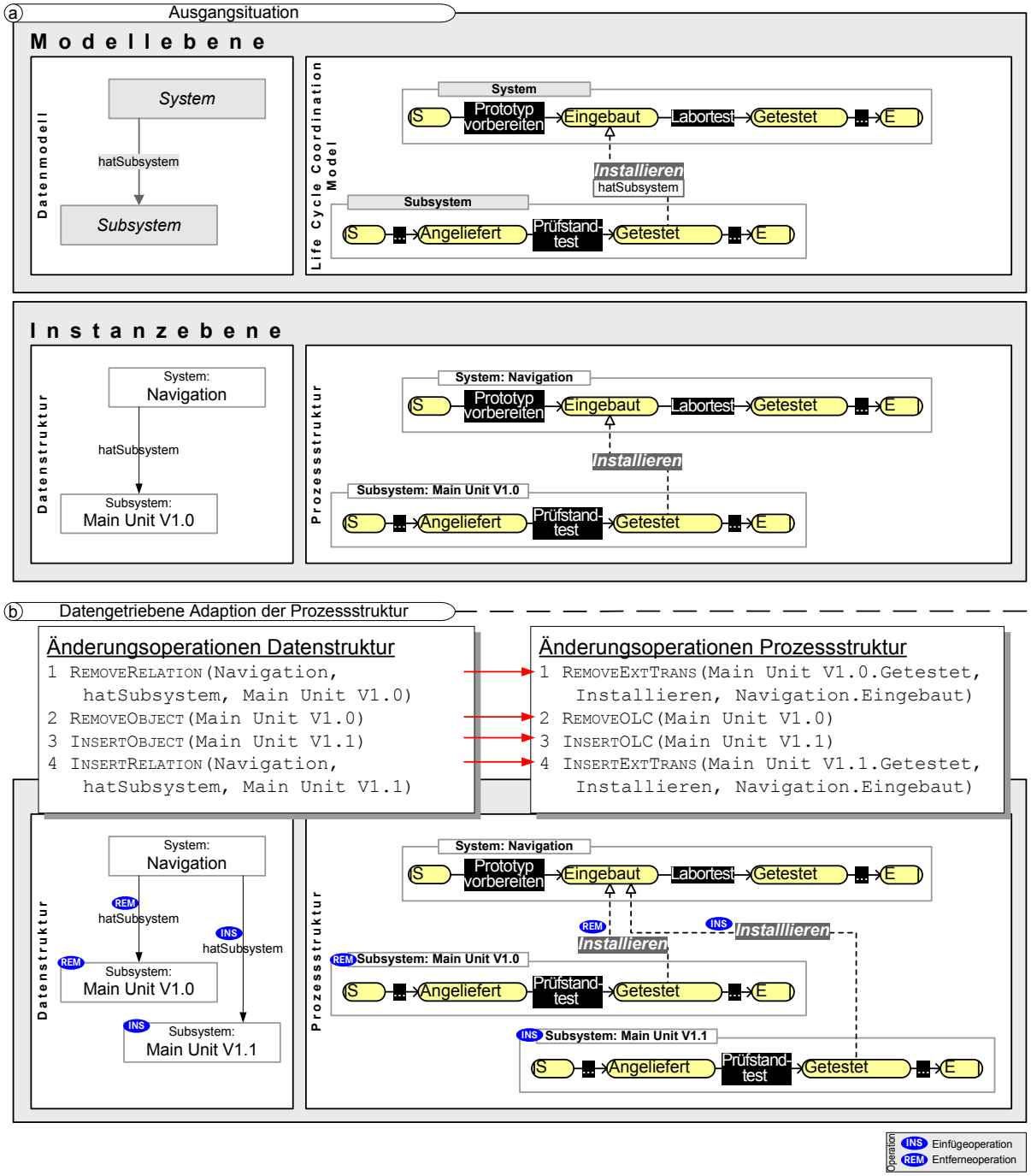


Abbildung 6.4: Beispiel für datengetriebene Adaptionen der Prozessstruktur

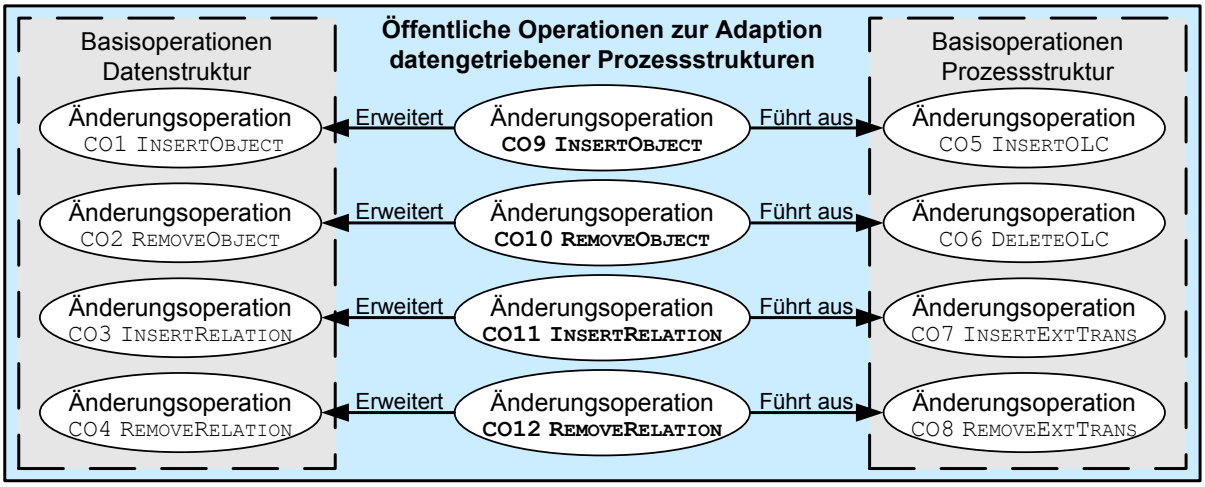


Abbildung 6.5: Öffentliche Operationen zur datengetriebenen Adaption von Prozessstrukturen

Änderungsoperation CO9 (InsertObject: Hinzufügen eines Objekts (erweitert))
Seien $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ eine Life Cycle Coordination Structure und $ds = (O, R)$ die zugehörige Datenstruktur. Wird die Änderungsoperation InsertObject*(o) für ein Objekt $o \notin O$ ausgeführt, wird automatisch eine Instanz des mit seinem Objekttyp verknüpften Object Life Cycle mittels Operation* InsertOLC*$(OLC_{inst}(o))$ in ps eingefügt (vgl. Definition 5.5).*

Änderungsoperation CO10 (RemoveObject: Entfernen eines Objekts (erweitert))
Seien $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ eine Life Cycle Coordination Structure und $ds = (O, R)$ die zugehörige Datenstruktur. Wird die Änderungsoperation RemoveObject*(o) für ein Objekt $o \in O$ ausgeführt, wird automatisch der mit ihm verknüpfte Object Life Cycle mittels Operation* RemoveOLC*$(OLC_{inst}(o))$ aus ps entfernt (vgl. Definition 5.5).*

Änderungsoperation CO11 (InsertRelation: Hinzufügen einer Relation (erweitert))
Seien $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ eine Life Cycle Coordination Structure und $ds = (O, R)$ die zugehörige Datenstruktur. Wird die Änderungsoperation InsertRelation*(r) für eine Relation $r \notin R$ ausgeführt, werden automatisch alle mit ihrem Relationstyp $reltype(r)$ verknüpften externen Transitionen mittels Operation $\forall e \in est_{inst}(r)$:* InsertExtTrans*(e) in ps eingefügt (vgl. Definition 5.5).*

Änderungsoperation CO12 (RemoveRelation: Entfernen einer Relation (erweitert))
Seien $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ eine Life Cycle Coordination Structure und $ds = (O, R)$ die zugehörige Datenstruktur. Wird die Änderungsoperation RemoveRelation*(r) für eine Relation $r \in R$ ausgeführt, werden automatisch alle mit ihrem Relationstyp $reltype(r)$ verknüpften externen Transitionen mittels Operation $\forall e \in est_{inst}(r)$:* RemoveExtTrans*(e) aus ps entfernt (vgl. Definition 5.5).*

Um die Rückverfolgbarkeit und Analysierbarkeit von Änderungen zu gewährleisten (vgl. Abschnitt 2.2.1), kann die Anwendung der Änderungsoperationen in einem *Änderungsprotokoll* (*Change Log*) protokolliert werden. Letzteres kann zu Optimierungszwecken beispielsweise mittels *Change Mining* Techniken ausgewertet werden [GRRA06, GRMR+08, LRW08b, LRW08a].

6.2.4 Höherwertige Änderungsoperationen

Auf Basis der im vorangegangenen Abschnitt beschriebenen öffentlichen Operationen zur datengetriebenen Adaption von Prozessstrukturen lassen sich semantisch höherwertige Operationen definieren (vgl. Abbildung 6.6). Sie sollen für spezifische Anwendungsfälle die Adaption datengetriebener Prozessstrukturen durch Kapselung verschiedener öffentlicher Operationen erleichtern. Da die höherwertigen Änderungsoperationen nur die öffentliche Schnittstelle nutzen, bleibt die Integrität zwischen Daten- und Prozessstruktur sowie die Korrektheit der Prozessstruktur erhalten. Nachfolgend stellen wir zwei höherwertige Änderungsoperationen exemplarisch vor.

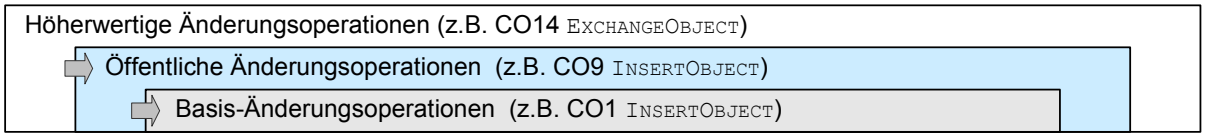


Abbildung 6.6: Schichtenmodell für die Änderungsoperationen in CoreFlow

Beispiel 1: Höherwertige Änderungsoperation RemObjCompletely

Eine einfache höherwertige Änderungsoperation stellt RemObjCompletely dar (vgl. Änderungsoperation CO13). Sie erlaubt das vollständige Entfernen eines Objekts, auch wenn noch Relationen von oder zu diesem Objekt bestehen. Dazu entfernt die Operation zunächst alle Relationen, die mit dem Objekt verbunden sind, und anschließend das Objekt selbst.

Änderungsoperation CO13 (RemObjCompletely: Vollst. Entfernen eines Objekts)
Seien $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ *eine Life Cycle Coordination Structure und* $ds = (O, R)$ *eine zugehörige Datenstruktur. Wird die Änderungsoperation* RemObjCompletely*(o) für ein Objekt* $o \in O$ *ausgeführt, wird folgende Veränderung vorgenommen:*

1. *Entfernen aller Relationen von o oder zu o:*
   $\forall\ r = (o_1, rt, o_2) \in R, (o_1 = o \vee o_2 = o)$: RemoveRelation(r) *(vgl. Änderungsoperation CO12)*
2. *Entfernen des Objekts:* RemoveObject*(o) (vgl. Änderungsoperation CO10)*

Beispiel 2: Höherwertige Änderungsoperation ExchangeObject

Der Austausch eines Objekts (vgl. Abbildung 6.4) kann durch Definition einer höherwertigen Operation geschehen, welche auf die Hinzufügen- und Entfernen-Operationen zurückgreift

[WRRM08]. In diesem Rahmen ist zu beachten, dass nicht nur das auszutauschende Objekt und seine Relationen mit anderen Objekten entfernt sowie das neue Objekt der Datenstruktur hinzugefügt werden müssen. Vielmehr sollen auch alle mit dem ursprünglichen Objekt verknüpften Relationen bzw. Relationstypen erneut hinzugefügt werden. Operation ExchangeObject realisiert diese Anforderung (vgl. Änderungsoperation CO14). Durch die Nutzung der öffentlichen Änderungsoperationen wird die Prozessstruktur entsprechend adaptiert. Abbildung 6.7 zeigt ein Beispiel für die Anwendung der Operation. Die interne Abbildung der höherwertigen Operation auf die Basis-Änderungsoperationen entspricht dem in Abbildung 6.4 gezeigten Szenario.

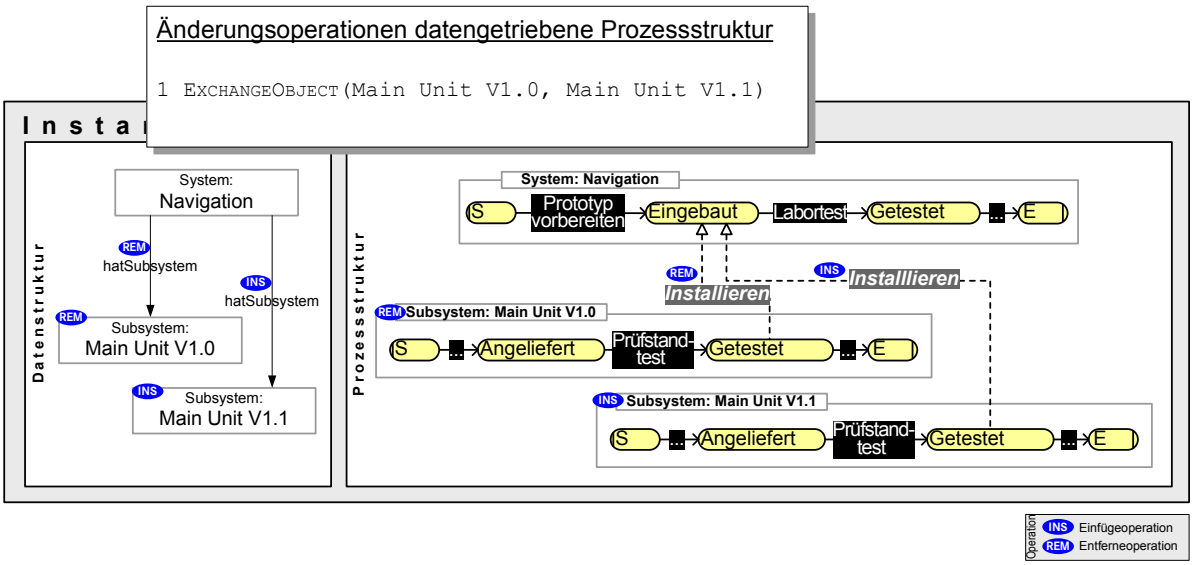


Abbildung 6.7: Austausch einer Komponente durch höherwertige Operation (vgl. Abbildung 6.4)

Änderungsoperation CO14 (ExchangeObject - Austausch eines Objekts)
Seien $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ *eine Life Cycle Coordination Structure und* $ds = (O, R)$ *die zugehörige Datenstruktur. Wird die Änderungsoperation* ExchangeObject(o, o') *bezogen auf zwei Objekte* $o \in O$ *und* $o' \notin O$ *ausgeführt, wird folgende Veränderung vorgenommen:*

1. *Einfügen des neuen Objekts* o'*:* InsertObject(o')
2. *Entfernen der Relationen von* o *zu anderen Objekten und Einfügen der entsprechenden Relationen für* o'*:*
   $\forall\, r = (o_1, rt, o_2) \in R, o_1 = o$: RemoveRelation$(r) \wedge$ InsertRelation$((o', rt, o_2))$
3. *Entfernen der Relationen von anderen Objekten zu* o *und Einfügen der entsprechenden Relationen für* o'*:*
   $\forall\, r = (o_1, rt, o_2) \in R, o_2 = o$: RemoveRelation$(r) \wedge$ InsertRelation$((o_1, rt, o'))$
4. *Entfernen des auszutauschenden Objekts* o*:* RemoveObject(o)

Die beiden eingeführten Änderungsoperationen RemObjCompletely und ExchangeObject zeigen exemplarisch den Nutzen höherwertiger Änderungsoperationen. Abhängig vom Anwendungsfall

können so beispielsweise auch unterschiedliche Vorgehensweisen für das Ersetzen einer Komponente realisiert werden (z.B. mit und ohne Entfernen des zu ersetzenden Objekts). Durch Anwendung der öffentlichen Änderungsoperationen ist zudem sichergestellt, dass keine *unzulässigen* Änderungen durchgeführt werden, die die strukturelle oder dynamische Korrektheit gefährden. In Kapitel 9 zeigen wir weitere Anwendungsfälle, für deren Unterstützung die Definition höherwertiger Änderungsoperationen Sinn macht.

6.3 Datengetriebene Adaption der Prozessstruktur: Dynamische Sicht

Im vorangegangenen Abschnitt haben wir die datengetriebene Adaption von Prozessstrukturen aus statischer Sicht behandelt. Das heißt, wir haben die dynamischen Aspekte, etwa die aktuelle Laufzeitmarkierung der modifizierten Prozessstruktur bisher ausgeblendet. Diese besitzt im Kontext der dynamischen Adaption von Prozessstrukturen allerdings eine große Bedeutung. Werden zum Beispiel bereits durchlaufene Bereiche der Prozessstruktur verändert, können Situationen entstehen, die im *normalen* Ablauf gar nicht erzeugbar wären. Diese sind grundsätzlich unerwünscht, denn sie sind durch die Korrektheitsbetrachtungen in Kapitel 4 nicht abgedeckt. Derartige Situationen können beispielsweise zu Verklemmungen führen und so die (dynamische) Korrektheit gefährden. Situationen, die im *normalen* Ablauf gar nicht erzeugbar wären, werden daher als *Inkonsistenzen* bezeichnet. Im diesem Abschnitt diskutieren wir, wann solche Inkonsistenzen auftreten, welche Konsequenzen sie haben und wie mit ihnen umgegangen werden muss.

6.3.1 Exemplarische Konsistenzprüfung dynamischer Adaptionen

Wir führen zunächst eine exemplarische Betrachtung verschiedener Szenarien, die beim Hinzufügen oder Entfernen sowohl ganzer OLCs als auch externer Transitionen auftreten können, durch. In diesem Kontext untersuchen wir die entstehenden Laufzeitmarkierungen adaptierter Prozessstrukturen und prüfen, ob diese Markierungen im normalen Ablauf hätten entstehen können. Ziel ist es, aus diesen charakteristischen Szenarien allgemeine Konsistenzkriterien abzuleiten.

Hinzufügen und Entfernen von OLCs

Die Beispiele 6.1 und 6.2 beschreiben verschiedene Szenarien für das Einfügen sowie das Entfernen von OLCs und diskutieren mögliche Inkonsistenzen in diesem Kontext. Die Datenstruktur kann für die folgende Anwendung der Änderungsoperationen vernachlässigt werden. Wir beschränken uns daher auf die atomare Anwendung der Basis-Änderungsoperationen für Prozessstrukturen (vgl. Abschnitt 6.2.2).

Beispiel 6.1 (Hinzufügen eines OLC)
Abbildung 6.8a zeigt eine stark vereinfachte Prozessstruktur mit nur einem OLC. In Abbildung 6.8b wird mittels Änderungsoperation `InsertOLC(B)` (vgl. Änderungsoperation CO5 bzw. CO9) der Prozessstruktur ein OLC hinzugefügt. Hierbei ist zu beachten, dass der OLC zunächst die initiale Markierung aufweist (d.h. der Startzustand des eingefügten OLC ist nicht aktiviert; vgl.

Abschnitt 4.3.3). Die entstandene Laufzeitmarkierung ist damit allerdings *syntaktisch inkonsistent*, denn nach Start der Prozessstruktur sind normalerweise die Startzustände aller enthaltenen OLCs als `ACTIVATED` oder `DONE` markiert (vgl. Abschnitt 4.6.4). Dies führt in der Folge zu einer Verklemmung, denn der OLC wird nicht durchlaufen und die Prozessstruktur kann nicht terminieren. Das Szenario ist vergleichbar mit dem *Dynamic Change Bug* bei strukturellen Änderungen von Petri-Netzen [Aal01]. Zur Lösung des Problems kann die Markierung des Startzustands des eingefügten OLC „nachgezogen“ werden (siehe Abschnitt 6.3.4).

Beispiel 6.2 (Entfernen eines OLC)
Abbildung 6.8c zeigt eine weitere Prozessstruktur mit den beiden OLCs `A` und `B`. Wird mithilfe der Änderungsoperation `REMOVEOLC(B)` (vgl. Änderungsoperation CO6 bzw. CO10) ein OLC aus der Prozessstruktur entfernt (vgl. Abbildung 6.8d), entsteht in der (verbleibenden) Prozessstruktur eine Markierung, die im normalen Ablauf erzeugbar ist. Das Szenario führt zu keiner Inkonsistenz.

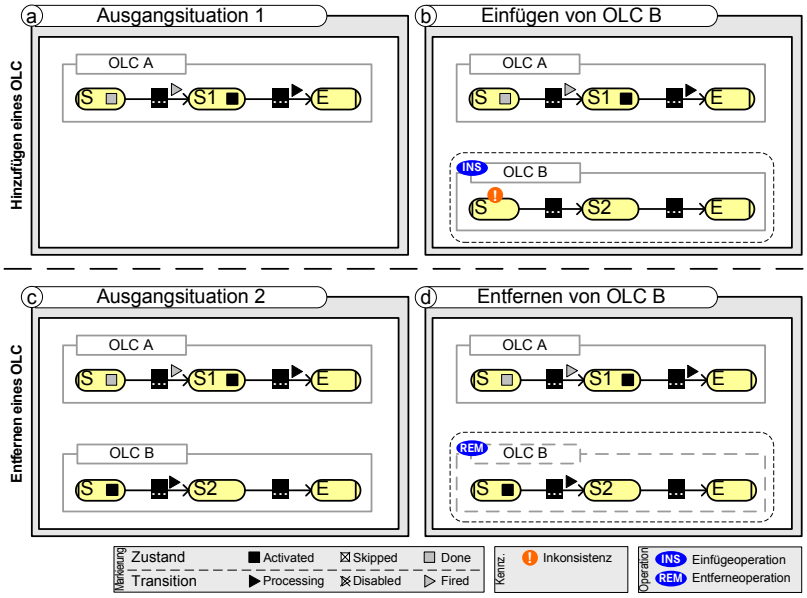


Abbildung 6.8: Konsistenz beim Hinzufügen oder Entfernen unsynchronisierter OLCs

Hinzufügen und Entfernen externer Transitionen

Die Beispiele 6.3 und 6.4 zeigen verschiedene Szenarien für das Einfügen sowie das Entfernen von externen Transitionen und diskutieren das Auftreten von Inkonsistenzen in diesem Kontext.

Beispiel 6.3 (Hinzufügen einer externen Transition)
Abbildung 6.9a zeigt den Ausschnitt einer Prozessstruktur ohne Laufzeitmarkierungen als Ausgangsszenario. Die Szenarien aus den Abbildungen 6.9b-d zeigen diesen Ausschnitt mit unterschiedlichen Markierungen der Zustände `A.S1` und `B.S2`. Wir untersuchen im Folgenden jeweils die Konsistenz beim Hinzufügen einer externen Transition.

In Abbildung 6.9b wird mittels Operation INSERTEXTTRANS die externe Transition $e = (\texttt{A.S1}, \texttt{P}, \texttt{B.S2})$ zwischen den Zuständen A.S1 und B.S2 eingefügt. Sie befindet sich in der Initialmarkierung (vgl. Änderungsoperation CO7). Die Anwendung der Operation führt hier zu keiner Inkonsistenz, da sich die erzeugte Markierung in der adaptierten Prozessstruktur herstellen lässt. Im Szenario aus Abbildung 6.9c ist hingegen der Quellzustand A.S1 von e bereits als ACTIVATED markiert. Wird e in diesem Szenario eingefügt, entsteht eine Markierung der Prozessstruktur, die im normalen Ablauf der modifizierten Prozessstruktur nicht erzeugbar ist. Schließlich wird e durch INSERTEXTTRANS als WAITING markiert. Wäre die externe Transition von Beginn an in der Prozessstruktur enthalten gewesen, hätte jedoch Ausführungsregel AR5 e sofort nach der Aktivierung von A.S1 als PROCESSING markiert. Somit entsteht durch das Einfügen der externen Transition e in das Szenario aus Abbildung 6.9c eine syntaktische Inkonsistenz. Diese Inkonsistenz führt in der Folge zu einer Verklemmung der Prozessstruktur. Da A.S1 bereits als ACTIVATED markiert ist, wird weder der Kontext von Ausführungsregel AR5 noch der Kontext einer anderen Ausführungsregel für e erfüllt und e wird nie als PROCESSING markiert.

Abbildung 6.9d zeigt das bekannte Szenario; Zustand A.S1 ist jedoch bereits als DONE und Zustand B.S2 als ACTIVATED markiert. Auch hier entsteht nach Einfügen der externen Transition e die aus Abbildung 6.9c bekannte syntaktische Inkonsistenz. Die Konstellation der Markierungen von e und Zustand B.S2 führt allerdings zu einer weiteren syntaktischen Inkonsistenz, denn entsprechend den Regeln aus Kapitel 4 müssen alle in B.S2 eingehenden externen Transitionen die Markierung FIRED erreichen, bevor B.S2 als ACTIVATED markiert werden kann. Diese Inkonsistenz führt in der Folge zu einer erneuten Aktivierung von B.S2 durch Markierungsregel MR6, sofern e feuert (die „erste“ Inkonsistenz sei an dieser Stelle ausgeklammert). Ist der OLC B zu diesem Zeitpunkt schon fortgeschritten, ist B.S2 also bereits als DONE markiert, würde seine erneute Aktivierung dazu führen, dass es in OLC B zwei aktivierte Zustände gibt. Eine solche Situation widerspricht der operationalen Semantik aus Kapitel 4 und ist unbedingt zu vermeiden.

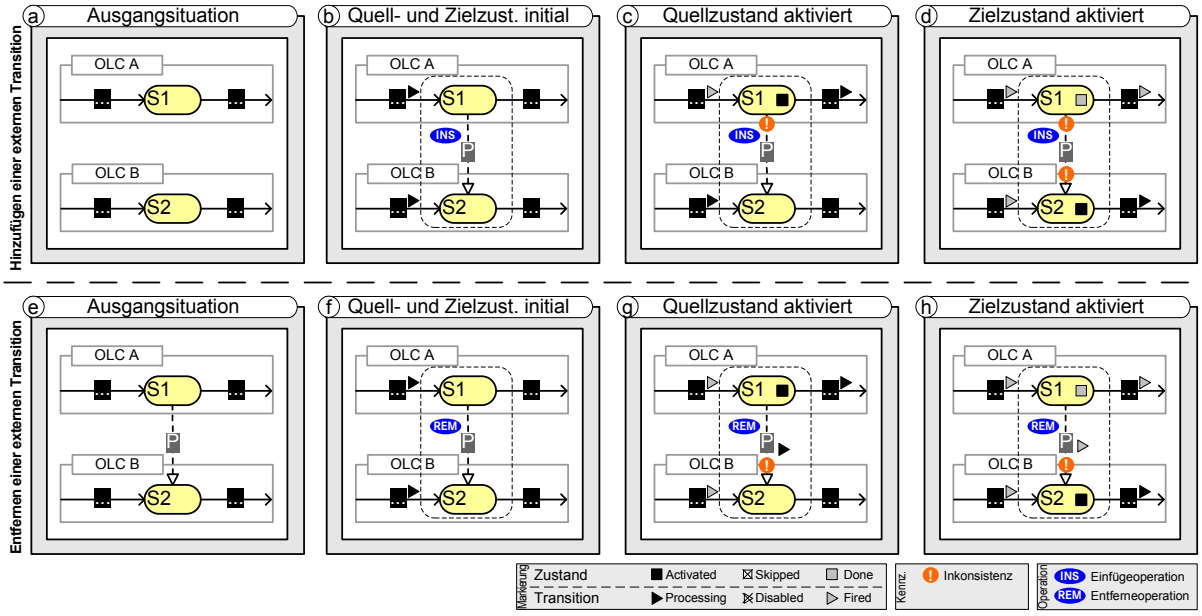


Abbildung 6.9: Konsistenz bei Hinzufügen oder Entfernen externer Transitionen

Beispiel 6.4 (Entfernen einer externen Transition)
Die Abbildungen 6.9e-h zeigen Szenarien für das Entfernen externer Transitionen. Abbildung 6.9e zeigt das Ausgangsszenario, aus dem in Abbildung 6.9f die externe Transition $e = (\texttt{A.S1}, \texttt{P}, \texttt{B.S2})$ entfernt wird. Die Markierungen ihres Quell- und Zielzustands sind initial. Die Durchführung der Operation führt nicht zu einer syntaktischen Inkonsistenz, da sich die erzeugte Markierung auf der adaptierten Prozessstruktur erneut herstellen lässt. Hat die in `B.S2` eingehende interne Transition bereits gefeuert, wie es Abbildung 6.9g zeigt, entsteht mit dem Entfernen von e eine syntaktische Inkonsistenz, da `B.S2` als `NotActivated` markiert ist. Diese Markierung ist in der modifizierten Prozessstruktur nicht erzeugbar, denn das Feuern der eingehenden internen Transition würde (ohne die eingehende externe Transition) zu einer Aktivierung von `B.S2` führen. Es entsteht somit eine Verklemmung, da es keine Markierungsregel gibt, die im weiteren Ablauf `B.S2` aktiviert.

Ist, wie Abbildung 6.9h zeigt, der Zielzustand `B.S2` bereits als `Done` markiert, hat die externe Transition e also bereits einen Zustandswechsel im Zielzustand veranlasst, entsteht durch Entfernen von e keine syntaktische Inkonsistenz. Allerdings hat das Feuern von e die Aktivierung von Zustand `B.S2` durch Markierungsregel MR6 bzw. MR7 sowie die Ausführung der auf `B.S2` folgenden Prozesse erst erlaubt. Das Entfernen von e ändert somit den Kontext für die Anwendung der Regeln im Nachhinein. In der Praxis tritt eine derartige Situation auf, wenn ein Absicherungsprozess für ein elektronisches System mit positivem Ergebnis durchlaufen wurde, im Nachhinein allerdings eines seiner Subsysteme entfernt wird. Damit sind die Ergebnisse des durchgeführten Absicherungsprozesses in Bezug auf die modifizierte Datenstruktur nicht mehr gültig. Wir sprechen hier von einer *semantischen Inkonsistenz*.

6.3.2 Konsistenzkriterien für die dynamische Adaption

In Kapitel 4 haben wir Korrektheitskriterien für OLCs und Prozessstrukturen vorgestellt und argumentiert, dass nur korrekte Prozessstrukturen modelliert werden bzw. zur Ausführung kommen dürfen. Wir müssen somit die konsistenzgefährdenden Szenarien zuverlässig identifizieren, um ein vorhersagbares und korrektes Verhalten von Prozessstrukturen nach einer dynamischen Adaption sicherzustellen. Allerdings dürfen Szenarien nur dann als kritisch eingestuft werden, wenn sie die Vorhersagbarkeit des Verhaltens der Prozessstruktur tatsächlich beeinflussen. So sollen keine „Pauschalaussagen“ getroffen werden, die zum Beispiel die Änderung eines bereits durchlaufenen Bereichs grundsätzlich verhindern [RRD04a]. In datengetrieben Prozessstrukturen würde dies zum Beispiel das Entfernen eines OLC aus der Prozessstruktur nicht erlauben. Konkret ergeben sich folgende Anforderungen an die Konsistenzanalyse:

- sie muss die korrekte Ausführung einer Prozessstruktur auch nach Durchführung dynamischer Änderungen garantieren,
- sie sollte die Anzahl möglicher Änderungen nicht unnötig einschränken und
- sie muss effizient implementierbar sein.

Die Untersuchung der Szenarien in Abschnitt 6.3.1 zeigt, dass in verschiedenen Situationen inkonsistente Laufzeitmarkierungen auftreten können und dass diese Inkonsistenzen unterschiedliche Ursachen haben. Wir wollen nun generische Konsistenzkriterien definieren, die nicht nur die in Abschnitt 6.3.1 betrachteten Szenarien abdecken, sondern mit denen sich Inkonsistenzen generell

erkennen lassen. Dazu muss zunächst die Frage beantwortet werden, wie Konsistenz im Rahmen dynamischer Änderungen formal definiert ist.

In Kapitel 4 haben wir eine operationale Semantik basierend auf Laufzeitmarkierungen von Zuständen und Transitionen eingeführt. Alle im normalen Ablauf infolge der Anwendung von Markierungs- und Ausführungsregeln (siehe Kapitel 4) erzeugbaren Laufzeitmarkierungen einer Prozessstruktur sind dynamisch korrekt. Entsteht durch die dynamische Adaption der Prozessstruktur eine Laufzeitmarkierung die im normalen Ablauf der adaptierten Prozessstruktur nicht hätte erzeugt werden können, sprechen wir von einer *syntaktischen Inkonsistenz* (vgl. Definition 6.1). Eine syntaktische Inkonsistenz entspricht der Verletzung einer Ausführungs- oder Markierungsregel, die im normalen Ablauf für die korrekte Markierung gesorgt hätte.

Definition 6.1 (Syntaktische Inkonsistenz)
Sei $ps = (OLC, EST)$ *eine (adaptierte) Prozessstruktur mit aktueller Markierung* $PSM = (OM, EM)$. *ps wird als syntaktisch inkonsistent bezeichnet, wenn nach Anwendung der Initialisierungsregel IR2 (siehe Seite 87) auf ps mittels der in Kapitel 4 definierten operationalen Semantik keine neue Markierung* PSM' *erzeugt werden kann, für die gilt:*

$$PSM' = PSM$$

Ferner zeigt die Diskussion des Szenarios für das Entfernen externer Transitionen aus Abbildung 6.9h, dass bei der Adaption von Prozessstrukturen neben syntaktischen auch semantische Inkonsistenzen auftreten können. Grob gesagt tritt eine semantische Inkonsistenz genau dann auf, wenn sich der Kontext einer angewendeten Ausführungs- oder Markierungsregel im Nachhinein ändert. Im Szenario aus Abbildung 6.4a (siehe Seite 140) etwa entsteht eine semantische Inkonsistenz, wenn die externe Transition mit Prozess `Installieren` aus der Prozessstruktur entfernt wird, nachdem der `Installieren` Prozess beendet und der Zielzustand aktiviert wurde.

Im Gegensatz zur syntaktischen Inkonsistenz bezieht sich die semantische Inkonsistenz damit nicht direkt auf die Markierung der adaptierten Prozessstruktur, sondern auf deren Verlauf. Der Verlauf einer Prozessstruktur ergibt sich aus den Verläufen ihrer OLCs sowie den Start- und Ende-Zeitstempeln der Synchronisationsprozesse (vgl. Abschnitt 4.6.4). Abbildung 6.10 zeigt eine Prozessstruktur mit den beiden OLCs `A` und `B` sowie zwei externen Transitionen. Das Entfernen der externen Transition (`B.B`, `Pe2`, `A.C`) verursacht, wie im Szenario aus Abbildung 6.9h, in ihrem Zielzustand eine semantische Inkonsistenz. Schließlich wurde dieser bereits aktiviert und darauf folgende Prozesse in OLC `A` durchlaufen (z.B. Prozess `PC`). Anhand des Verlaufs für OLC `A` lassen sich die Auswirkungen der externen Transition bzw. ihres Synchronisationsprozesses erkennen. Findet sich ein Eintrag über dessen Ende im Verlauf eines OLC und wurde danach ein weiterer Prozess im Verlauf des betroffenen OLC gestartet, führt das Entfernen der externen Transition zu einer semantischen Inkonsistenz (vgl. Abbildung 6.10). Eine solche Inkonsistenz entsteht somit genau dann, wenn der bisherige Verlauf der in der geänderten Prozessstruktur enthaltenen OLCs, bereinigt um die Start-Ereignisse externer Transitionen, auf der geänderten Prozessstruktur nicht erzeugt werden kann (vgl. Definition 6.2).

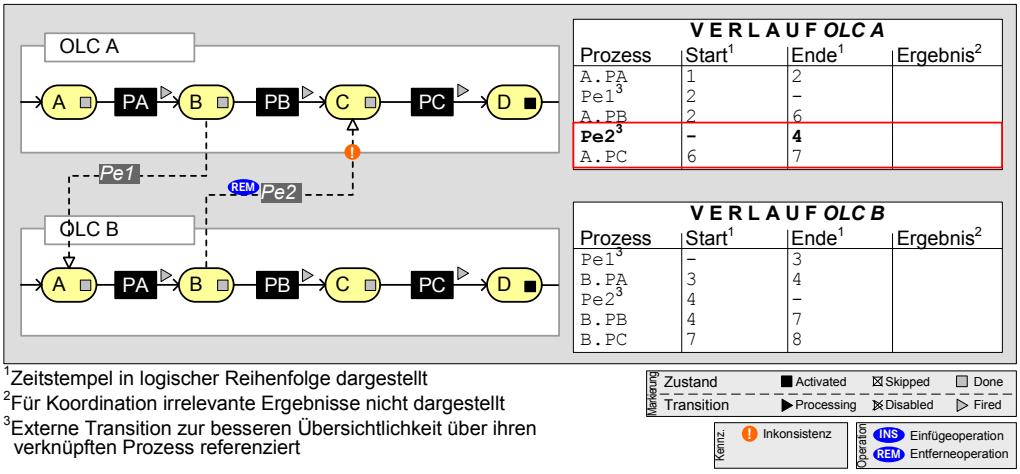


Abbildung 6.10: Erkennen einer semantischen Inkonsistenz durch den Verlauf

Definition 6.2 (Semantische Inkonsistenz)
Seien $ps = (OLC, EST)$ *eine Prozessstruktur mit aktueller Markierung* PSM *und* $olc = (P, V, TS) \in OLC$ *ein Object Life Cycle. Sei* L *der Verlauf von* ps *und enthalte* $L(olc)$ *die Zeitstempel für den Start und das Ende der für* olc *relevanten Prozesse (wir verzichten auf eine Formalisierung des Verlaufs; vgl. Abschnitt 4.6.4).* ps *wird als semantisch inkonsistent bezeichnet, wenn nach Anwendung der Initialisierungsregel IR2 (vgl. Definition auf Seite 87) mittels der in Kapitel 4 definierten operationalen Semantik für* ps keine *neue Markierung* PSM' *erzeugt werden kann, in der für ihren Verlauf* L' *gilt: es befindet sich in* $L(olc)$ *ein Eintrag für das Ende eines Synchronisationsprozesses und dieser Eintrag kann für* $L'(olc)$ *nicht erzeugt werden.*

Die Untersuchung relevanter Szenarien und die Definition der semantischen Inkonsistenz (vgl. Definition 6.2) zeigen, dass eine semantische Inkonsistenz zwar beim Entfernen externer Transitionen auftreten kann, nicht aber beim Entfernen unsynchronisierter OLCs. Grund ist, dass vom Entfernen eines unsynchronisierten OLC kein Verlauf eines anderen OLC der Prozessstruktur betroffen ist.

6.3.3 Identifizierung von Inkonsistenzen

Zu Beginn des Abschnitts 6.3.2 haben wir gefordert, dass die Konsistenzanalyse effizient implementierbar sein soll. Die Erzeugung möglicher Laufzeitmarkierungen adaptierter Prozessstrukturen und die Prüfung ihrer Verläufe zur Identifikation von syntaktischen oder semantischen Inkonsistenzen sind jedoch mit hohem Aufwand verbunden. Wir stellen daher in diesem Abschnitt Mechanismen vor, mit denen sich derartige Inkonsistenzen einfacher erkennen lassen und die keine vollständige Traversierung durch (große) Prozessstrukturen erfordern.[4] Zuvor diskutieren wir die generelle Vorgehensweise für die Identifikation und Behandlung von Inkonsistenzen.

In der Literatur finden sich zwei grundsätzliche Vorgehensweisen für den Umgang mit Inkonsistenzen. Der ADEPT-Ansatz beispielsweise schließt die Entstehung von Inkonsistenzen durch die

[4]Eine Diskussion bzgl. Effizienz von Korrektheitskriterien findet sich in [RRD04a].

Definition formaler Vor- und Nachbedingungen für Änderungsoperationen aus [RD98]. Bei der Adaption von Prozessstrukturen können Inkonsistenzen jedoch nicht immer vermieden werden. So kann das Entfernen einer Relation bzw. entsprechender externer Transitionen zu (temporären) Inkonsistenzen in OLCs führen. Durch Entfernen derjenigen Objekte, die mit inkonsistenten OLCs verbunden sind, kann die Konsistenz der Prozessstruktur wieder hergestellt werden. Die Gesamtheit der Änderungen kann damit wieder zu einer konsistenten Prozessstruktur führen. Im Gegensatz zu ADEPT unterstützt das Projekt *WIDE* derartige Szenarien mit temporären Inkonsistenzen. Der Ansatz erlaubt die Durchführung mehrerer Änderungsoperationen und fordert anschließend die Prüfung der Korrektheit des geänderten Prozesses [CCPP98]. *WIDE* erlaubt damit die *lose Kopplung* zwischen (1) der Durchführung von Änderungsoperationen, (2) der *Konsistenzanalyse* (auch *Ausnahmedetektion* genannt [Rei00]) und (3) der *Ausnahmebehandlung*. Das COREPRO-Konzept zur dynamischen Adaption und Ausnahmebehandlung orientiert sich an der Idee der losen Kopplung (vgl. Abbildung 6.11) und entwickelt daraus Methoden für die Erkennung und Behandlung von Inkonsistenzen in datengetriebenen Prozessstrukturen.

Abbildung 6.11 zeigt das systematische Vorgehen in COREPRO zur Durchführung einer dynamischen Adaption. Die Konsistenzanalyse, also der auf die Durchführung der Adaption folgende Schritt, identifiziert und kennzeichnet im Kontext der Adaption entstandene Inkonsistenzen. Diese Inkonsistenzen können im Idealfall automatisch aufgelöst werden (*automatische Ausnahmebehandlung*). Ist dies nicht möglich, kann eine manuelle Ausnahmebehandlung durchgeführt oder die Änderungen müssen verworfen werden. Die Möglichkeiten zur manuellen Behandlung von Inkonsistenzen sind vielfältig und reichen vom Entfernen der inkonsistenten Elemente aus der Prozessstruktur über das „Rücksetzen“ der Markierung der Prozessstruktur mittels Initialisierungsegel IR2 und ihre erneute Ausführung bis hin zu komplexen Mechanismen zur Anpassung von Laufzeitmarkierungen (siehe Kapitel 7).

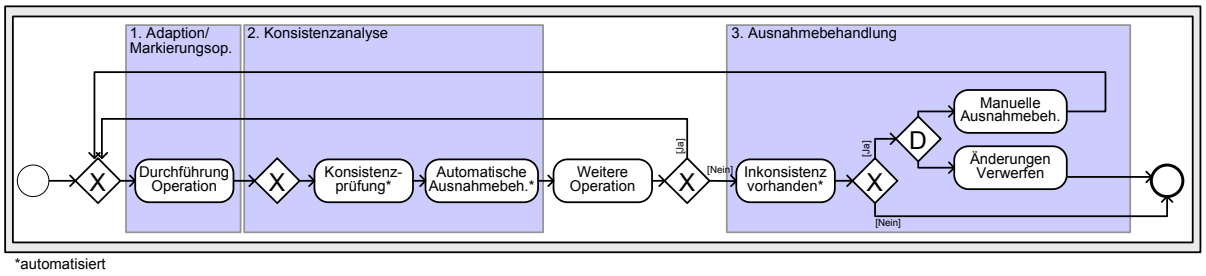


Abbildung 6.11: Durchführung dynamischer Adaptionen und zugehöriger Konsistenzprüfungen

Für die technische Realisierung der Konsistenzanalyse stellt sich zunächst die Frage, wie eine Konsistenzprüfung *operativ* durchgeführt werden kann, ohne die Prozessstruktur auf Basis der formalen Kriterien vollständig durchlaufen und mögliche Verläufe generieren zu müssen. Die exemplarische Betrachtung der Szenarien in Abschnitt 6.3.1 hat verdeutlicht, dass die Betrachtung begrenzter Ausschnitte ausreicht, um Aussagen zur Konsistenz der adaptierten Prozessstruktur treffen zu können. Wir bezeichnen derartige Ausschnitte als *Änderungsregionen*.

Abhängig von der ausgeführten Änderungsoperation unterscheiden wir Änderungsregionen externer Transitionen und ganzer OLCs. Die Wahl des Ausschnitts der Änderungsregion hängt von den benötigten Informationen für die Konsistenzprüfung ab. Für die Prüfung sind jeweils nur diejenigen Markierungen relevant, die von der Adaption direkt betroffen sind. Da eine Inkon-

sistenz die Verletzung einer Ausführungs- oder Markierungsregel repräsentiert (vgl. Abschnitt 6.3.2), können aus ihnen die für die Auswahl der Änderungsregion notwendigen Elemente abgeleitet werden. Bei der Adaption externer Transitionen sind ihre eigene Markierung sowie die Markierungen ihres Quell- und Zielzustands notwendig. So ist die Markierung einer externen Transition e abhängig von den Markierungen ihres Quell- bzw. Zielzustands (vgl. Ausführungsregeln AR5 und AR6). Die Markierung des Zielzustands wiederum wird durch die Markierung von e beeinflusst (vgl. Markierungsregeln MR6 und MR7). Weitere Elemente sind entsprechend der Regeln nicht direkt betroffen. Zur Änderungsregion einer externen Transition gehören deshalb die externe Transition, sowie ihr Quell- und Zielzustand (in Abbildungen 6.9b-d sowie f-h sind die Änderungsregionen durch gestrichelte Rahmen angedeutet).

Beim Hinzufügen eines OLC wird die syntaktische Inkonsistenz nicht auf Basis der Ausführungs- und Markierungsregeln geprüft. Entscheidend ist die Markierung des Startzustands des OLC, denn sie ist syntaktisch inkonsistent, wenn die OLC-Phase (vgl. Definition 4.10) nicht mit der aktuellen Prozessstruktur-Phase übereinstimmt (vgl. Definition 4.17). Befindet sich beispielsweise die Prozessstruktur in der Phase RUNNING und wird ein OLC eingefügt, dann befindet sich dieser in der Phase INITIAL (vgl. Änderungsoperation CO5). Wir betrachten daher bei Hinzufügen oder Entfernen eines OLC diesen als Änderungsregion (in Abbildungen 6.8b und d sind die Änderungsregionen durch gestrichelte Rahmen angedeutet).[5]

Um eine geeignete Kennzeichnung von inkonsistenten Elementen einer Änderungsregion zu erlauben, führen wir ferner die Kennzeichnungsfunktionen SF und EF für Zustände bzw. externe Transitionen ein (vgl. Definition 6.3). Sie erlauben die direkte Kennzeichnung als INCONSISTENT. So kann eine syntaktische Inkonsistenz innerhalb einer Änderungsregion sowohl für die Markierung einer externen Transition, als auch für ihren Zielzustand auftreten (vgl. Abbildung 6.9c, d und g). Die Stelle hängt davon ab, ob eine Ausführungs- oder Markierungsregel durch die Laufzeitmarkierungen verletzt wurde. Diese feingranulare Kennzeichnung von Inkonsistenzen einzelner Elemente, und nicht etwa ganzer Änderungsregionen, ist notwendig, um eine gezielte Ausnahmebehandlung zu erlauben.

Definition 6.3 (Kennzeichnungsfunktionen SF und EF für Inkonsistenzen)
*Seien $ps = (OLC, EST)$ eine Prozessstruktur und S_{Targ} die Menge der Zustände aller OLCs mit $S_{Targ} := \{s^* | \exists\ olc = (P, V, TS) \in OLC \wedge TS = (S, T, s_{start}, s_{end}) \wedge s^* \in S\}$. Wir bezeichnen die Abbildungen $EF : EST \mapsto Flags$ und $SF : S_{Targ} \mapsto Flags$, die jeder externen Transition $e \in EST$ und jedem Zustand in $s \in S_{Targ}$ eine Kennzeichnung $EF(e) \in Flags$ bzw. $SM(s) \in Flags$ zuordnet als Kennzeichnungsfunktionen von ps. Die Menge der Kennzeichnungen (*Flags*) ist wie folgt definiert:*

$$Flags := \{\text{NONE}, \text{INCONSISTENT}\}^a$$

Die Kennzeichnung einer externen Transition e bzw. eines Zustands s wird mit $EF(e) =$ NONE bzw. $SF(s) =$ NONE initialisiert.

[a] Die Menge der Kennzeichnungen wird in Kapitel 7 erweitert.

[5] Das Entfernen von OLCs führt nicht zu einer Inkonsistenz (vgl. Abschnitte 6.3.1 und 6.3.2).

Die Betrachtung von Änderungsregionen erlaubt, die Prozessstruktur für die Konsistenzprüfung auf ihre Änderungsregionen zu reduzieren und damit unabhängig von ihrer Größe vorzunehmen. Anhand der Kriterien aus den Definitionen 6.1 und 6.2 ist es nun möglich, jede erzeugbare Änderungsregion basierend auf den Laufzeitmarkierungen ihrer Elemente hinsichtlich der syntaktischen und semantischen Konsistenz zu bewerten (vgl. Abbildung C.1 in Anhang C). Kennzeichnungsregel KR1 beschreibt eine derartige Konsistenzprüfung von Änderungsregionen sowie die Kennzeichnung entsprechender Elemente als INCONSISTENT. Hierfür wird anhand der Laufzeitmarkierungen der Elemente einer Änderungsregion die Verletzung von Ausführungs- und Markierungsregeln geprüft. Dabei wird auf die Unterscheidung semantischer und syntaktischer Inkonsistenzen verzichtet, denn sie ist für die Erarbeitung der Konzepte zur Ausnahmebehandlung nicht notwendig. Vielmehr erlaubt ihre gemeinsame Betrachtung die Definition generischer Methoden zur Ausnahmebehandlung (siehe Kapitel 7).

Kennzeichnungsregel KR1 (Inkonsistenz Änderungsregion)
Seien $ps = (OLC, EST)$ eine Prozessstruktur mit aktueller Markierung $PSM = (OM, EM)$ und $ps' = (OLC', EST')$ eine Prozessstruktur, die durch Adaption von ps entstanden ist.

Erfährt ps durch die externe Transition $e^* = (s_1, p, s_2) \in \mathcal{EST}$ eine Änderung, gilt:

(a) Wenn e^* durch Operation INSERTEXTTRANS(e^*) in ps eingefügt wurde (bzw. ihre Markierung auf $EM(e^*)$ = WAITING gewechselt hat[6]) und gilt $SM(s_1) \in \{\text{ACTIVATED}, \text{DONE}, \text{SKIPPED}\} \vee SM(s_2) = \text{SKIPPED}$, dann folgt:
$EF(e^*) :=$ INCONSISTENT (Verletzung von Ausführungsregel AR5 bzw. AR6).

(b) Wenn e^* durch Operation INSERTEXTTRANS(e^*) in ps eingefügt wurde (bzw. ihre Markierung auf $EM(e^*) =$ WAITING gewechselt hat[6]) und gilt $SM(s_2) \in \{\text{ACTIVATED}, \text{DONE}\}$, dann folgt:
$SF(s_2) :=$ INCONSISTENT (Verletzung von Markierungsregel MR6 bzw. MR7).

(c) Wenn e^* durch Operation REMOVEEXTTRANS(e^*) aus ps entfernt wurde und gilt $SM(s_2) \in \{\text{ACTIVATED}, \text{DONE}\}$, dann folgt:
$SF(s_2) :=$ INCONSISTENT (Veränderung des Verlaufs bzw. des Kontextes für Markierungsregel MR6 bzw. MR7).

(d) Wurde e^* durch Operation REMOVEEXTTRANS(e^*) aus ps entfernt und sei $SM(s_2) =$ NOTACTIVATED. Ferner habe eine in s_2 eingehende interne Transition bereits gefeuert (d.h. $\exists t' = (src, (p, v), s_2) \in inTrans_{int}(s_2) : TM(t') =$ FIRED) und für alle in s_2 eingehenden externen Transitionen $\hat{e} \in inTrans_{ext}(s_2), \hat{e} \neq e^*$ gilt $EM(\hat{e}) \in \{\text{FIRED}, \text{DISABLED}\}$, dann folgt:
$SF(s_2) :=$ INCONSISTENT (Verletzung von Markierungsregel MR6 bzw. MR7).

Erfährt ps durch den Object Life Cycle $olc \in \mathcal{OLC}$ eine Änderung, gilt:

(a) wenn olc durch Operation INSERTOLC(olc) in ps eingefügt wurde und die Phase von olc entspricht nicht der Phase von ps (d.h. $OP(olc) \neq PP(ps)$; vgl. Definition 4.10 und 4.17), dann folgt:
$SF(s_{start}(olc)) :=$ INCONSISTENT.

[6]In Kapitel 7 werden sog. Markierungsoperationen eingeführt. Wir wollen Konsistenzregel KR1 in diesem Fall ebenfalls für die Konsistenzanalyse nutzen.

6.3.4 Automatische Behandlung von Inkonsistenzen

Entstehen im Kontext dynamischer Adaptionen Inkonsistenzen, muss dies nicht zwangsläufig zu einer Ablehnung der Änderungen oder einer manuellen Ausnahmebehandlung zum Auflösen der Inkonsistenzen führen (vgl. Abbildung 6.11). Syntaktische Inkonsistenzen können teilweise auch durch Neubewertung der Markierung inkonsistenter Elemente vermieden werden.

Um beispielsweise eine Verklemmung der Prozessstrukturen aus den Abbildungen 6.8b, 6.9c und 6.9d zu verhindern, kann die Markierung der betroffenen Zustände und externen Transitionen angepasst werden. Das gewünschte Verhalten wird durch die Markierungsregeln MR8 und MR9 realisiert und damit die syntaktische Inkonsistenz aufgehoben. Markierungsregel MR8 sorgt für die automatische Ausnahmebehandlung inkonsistenter Zustände. So wird einerseits der Startzustand derjenigen OLCs aktiviert, die einer Prozessstruktur hinzugefügt wurden. Andererseits wird beim Entfernen einer externen Transition die Markierung ihres Zielzustands auf ACTIVATED geändert, wenn er noch nicht aktiviert ist, aber bereits alle eingehenden Transitionen gefeuert haben oder abgewählt wurden (vgl. Szenario in Abbildung 6.8g).

Markierungsregel MR8 (Neubewertung der Markierung von Zuständen)
Seien $ps = (OLC, EST)$ eine Prozessstruktur mit aktueller Markierung $PSM = (OM, EM)$ und $olc \in OLC$ ein Object Life Cycle mit der Markierung $M = (SM, TM) \in OM$ und der Phase $OP(olc) =$ INITIAL.

(a) Wird der Startzustand von olc als $SF(s_{start}(olc)) :=$ INCONSISTENT gekennzeichnet und befindet sich die Prozessstruktur in der Phase $PP(ps) =$ RUNNING, ergibt sich als Folgemarkierung für $PSM' = (OM', EM')$ bzw. $M' = (SM', TM') \in OM'$:

$$TM' \equiv TM$$

$$SM'(s) = \begin{cases} \text{ACTIVATED} & \text{falls } s = s_{start}(olc) \\ SM(s) & \text{sonst} \end{cases}$$

(b) Wird ein Zustand $s^* \in S$ als $SF(s^*) :=$ INCONSISTENT gekennzeichnet und hat eine in s^* eingehende interne Transition bereits gefeuert (d.h. $\exists t = (src, (p, v), s^*) \in inTrans_{int}(s^*) : TM(t) =$ FIRED) und gilt für alle in s^* eingehenden externen Transitionen $e \in inTrans_{ext}(s^*) : EM(e) \in \{$FIRED, DISABLED$\}$, ergibt sich als Folgemarkierung für $PSM' = (OM', EM')$ bzw. $M' = (SM', TM') \in OM'$:

$$TM' \equiv TM$$

$$SM'(s) = \begin{cases} \text{ACTIVATED} & \text{falls } s = s^* \\ \text{DONE} & \text{falls } s = src \\ SM(s) & \text{sonst} \end{cases}$$

Wird eine Neubewertung der Markierung eines Zustands s durchgeführt, dann wird seine Kennzeichnung von $SF(s) =$ INCONSISTENT auf $SF(s) =$ NONE geändert.

Die Markierungsregel MR9 beschreibt die automatische Ausnahmebehandlung inkonsistenter externer Transitionen, d.h. die mögliche Neubewertung ihrer Markierung. Abbildung 6.12 illustriert

die Anwendung der Regel. Wird in die Prozessstruktur aus Abbildung 6.12a eine externe Transition eingefügt, entsteht aufgrund ihrer Markierung zunächst eine Inkonsistenz (vgl. Abbildung 6.12b). Diese wird im Anschluss automatisch aufgelöst: MR9 markiert die externe Transition als PROCESSING und verhindert damit eine Verklemmung (vgl. Abbildung 6.12c). Der einfache Eingriff der Markierungsregeln MR8 und MR9 erlaubt die Herstellung einer Markierung, wie sie entstanden wäre, wenn die Ausführungs- und Markierungsregeln aus Kapitel 4 im „normalen" Ablauf die jeweilige Markierung gesetzt hätten. Die Neubewertung von Markierungen im Kontext der automatischen Ausnahmebehandlung ist allerdings nur möglich, wenn dadurch nicht weitere Inkonsistenzen entstehen (vgl. Szenarien in Abbildung 6.9d und 6.9h).

Markierungsregel MR9 (Neubewertung der Markierung externer Transitionen)
Sei $ps = (OLC, EST)$ eine Prozessstruktur mit aktueller Markierung $PSM = (OM, EM)$. Wird eine externe Transition $e^* = (s_1, p, s_2) \in EST$ als $EF(e^*) = \text{INCONSISTENT}$ gekennzeichnet, dann ergibt sich die Folgemarkierung für $PSM' = (OM', EM')$:

$$EM'(e) = \begin{cases} \text{PROCESSING} & \text{falls } e = e^* \wedge EM(e) = \text{WAITING} \wedge SM(s_1) \in \{\text{ACTIVATED}, \text{DONE}\} \\ & \wedge\ SM(s_2) \neq \text{SKIPPED} \\ \text{DISABLED} & \text{falls } e = e^* \wedge EM(e) = \text{WAITING} \\ & \wedge\ (SM(s_1) = \text{SKIPPED} \vee SM(s_2) = \text{SKIPPED}) \\ EM(e) & \text{sonst} \end{cases}$$

$$OM' \equiv OM$$

Wird eine Neubewertung der Markierung durchgeführt, dann wird die Kennzeichnung von $EF(e^*) = \text{INCONSISTENT}$ auf $EF(e^*) = \text{NONE}$ geändert.

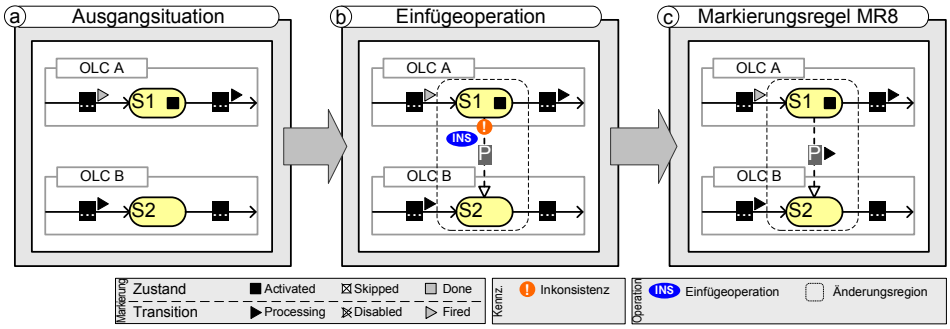


Abbildung 6.12: Neubewertung der Laufzeitmarkierung einer externen Transition

6.3.5 Änderungstransaktionen

Die Garantie der geforderten Korrektheits- und Konsistenzeigenschaften innerhalb der Prozessstruktur ist essentiell (vgl. Anforderung 6.3). Es können jedoch nicht alle Inkonsistenzen durch die automatische Ausnahmebehandlung aufgelöst werden. So bleiben die syntaktische Inkonsistenz der externen Transition im Zielzustand `B.S2` in Abbildung 6.9d, wie auch die semantische

Inkonsistenz in Abbildung 6.9h bestehen. Problem ist in beiden Fällen, dass in OLC `B` nicht nur der jeweils auf `B.S2` folgende Prozess gestartet, sondern dieser und ihm nachfolgende Prozesse bereits beendet sein können. Die Inkonsistenz kann in diesen Situationen nicht über eine einfache Markierungsanpassung des Zielzustands aufgelöst werden, sondern es sind weitergehende Mechanismen zu ihrer Behandlung nötig.

Bevor wir in Kapitel 7 derartige Mechanismen einführen, stellen wir zunächst ein Konzept vor, das dem Nutzer die Möglichkeit gibt, seine Änderungen rückgängig zu machen und mit der Ausführung der ursprünglichen Prozessstruktur fortzufahren. In Prozess-Management-Systemen findet hierzu, wie in traditionellen Datenbank-Systemen, das Konzept der *Transaktion* seine Anwendung [Elm92]. Ziel ist die Kapselung verschiedener Änderungen, um sie entweder vollständig durchzuführen – sofern sie in ihrer Gesamtheit zu keiner Inkonsistenz führen – oder zu verwerfen und damit zum letzten konsistenten Zustand zurückzukehren (*Backward Recovery*). Während es in klassischen Prozess-Management-Systemen das Ziel ist, konkurrierende Zugriffe zu verwalten, Systemausfälle ohne Inkonsistenzen zu überstehen oder mehrere Arbeitsschritte als Einheit auszuführen [SR93], verfolgen wir in CoRepro mit dem Transaktionsmanagement etwas andere Ziele.[7] In CoRepro wollen wir durch ein geeignetes Transaktionsmanagement sicherstellen, dass eine Änderung nur dann in der Prozessstruktur dauerhaft persistent gemacht werden kann, wenn sie nicht zu einer Inkonsistenz führt. Der Transaktionsmechanismus ist daher sehr einfach und lehnt sich an den aus Datenbanken bekannten *flachen Transaktionen* an.[8]

Wir kapseln die konkreten Änderungsoperationen einer Prozessstruktur in einer sogenannten *Änderungstransaktion*, damit diese atomar angewendet werden (vgl. Definition 6.4). Dadurch werden logisch zusammenhängende Operationen (z.B. die Entfernung mehrerer Relationen und Objekte) gruppiert und deren Einfluss auf die Konsistenz der adaptierten Prozessstruktur ganzheitlich betrachtet. Das ist notwendig, da bei der Durchführung logisch zusammenhängender Operationen zwischenzeitlich Inkonsistenzen auftreten können (z.B. Entfernen einer Relation führt zu einer Inkonsistenz in einem OLC, das entsprechende Objekt wird aber anschließend selbst entfernt). Eine Änderungstransaktion kann nur dann abgeschlossen werden, wenn es keinen Zustand s mit $SF(s) = \texttt{INCONSISTENT}$ und keine externe Transition e mit $EF(e) = \texttt{INCONSISTENT}$ in der adaptierten Prozessstruktur gibt (vgl. Definition 6.5).

[7] Zwar sind die genannten Ziele für die praktische Realisierung genauso relevant, sie befinden sich jedoch auf einer technischen Ebene und werden im Rahmen dieser Arbeit nicht näher betrachtet.

[8] CoRepro unterstützt einen einstufigen, flachen Transaktionsmechanismus. Für die Realisierung von Änderungstransaktionen existieren im Umfeld der Prozessadaption verschiedene Konzepte, wie z.B. geschachtelte Transaktionen, mehrstufige Transaktionen, Kompensationssphären [Rei00, LR00].

Definition 6.4 (Beginn Änderungstransaktion)
Sei lcs eine Life Cycle Coordination Structure. Dann ist eine Änderungstransaktion (Change Transaction) ct beschrieben durch ein Tupel $ct = (lcs, lcs')$ *mit*

- *lcs ist die ursprüngliche Life Cycle Coordination Structure*
- lcs' *ist die durch die Änderungsoperationen angepasste Life Cycle Coordination Structure.*

Jede Änderungs- oder Markierungsoperation (siehe Abschnitt 7.2) muss im Rahmen einer Änderungstransaktion, also auf lcs' *durchgeführt werden. Wird eine Änderungstransaktion für lcs angelegt, so wird eine Kopie von lcs erstellt und als* lcs' *der Änderungstransaktion zugeordnet. Alle Änderungsoperationen, die im Rahmen von ct ausgeführt werden, operieren auf* lcs'*.*[a]

[a] Aus Effizienzgründen kann anstatt einer vollständigen Kopie auch nur die relevante Differenz (z.B. die Sequenz der Operationsaufrufe) in der Änderungstransaktion gehalten werden. Um die Verständlichkeit zu erhalten, wird in dieser Arbeit darauf verzichtet.

Definition 6.5 (Abschluss Änderungstransaktion)
Sei $ct = (lcs, lcs')$ *eine Änderungstransaktion mit ursprünglicher Life Cycle Coordination Structure* $lcs = (dm, lcm, ds, ps)$ *und der durch Änderungsoperationen angepassten Life Cycle Coordination Structure* $lcs' = (dm, lcm, ds', ps')$ *mit* $ps' = (OLC, EST)$*. Dann kann ct mittels Operation*

- `Commit` *abgeschlossen und dauerhaft gemacht werden, wenn es keine Inkonsistenzen innerhalb der Prozessstruktur gibt:*

  $\nexists\, e \in EST' : EF(e) = \textsc{Inconsistent} \;\wedge$
  $\nexists\, olc = (P, V, TS) \in OLC' : TS = (S, T, s_{start}, s_{end}) \wedge \exists\, s \in S : SF(s) = \textsc{Inconsistent}$

  Durch Ausführung der Operation `Commit` *wird die ursprüngliche Life Cycle Coordination Structure lcs durch die geänderte Version* lcs' *ersetzt.*

- `Abort` *abgebrochen werden. Die ursprüngliche Life Cycle Coordination Structure lcs bleibt unverändert erhalten, während* lcs' *verworfen wird.*

In traditionellen Datenbank- und Prozess-Management-Systemen folgen Änderungstransaktionen dem ACID-Paradigma (d.h. *Atomicity*, *Consistency*, *Isolation*, *Durability* [Elm92, LR00, Rei00]). CoreFlow unterstützt das ACID-Paradigma bereits weitgehend. So werden Änderungsoperationen einer Änderungstransaktion entweder vollständig oder gar nicht in die Prozessstruktur eingebracht (*Atomicity*). Weiter ist durch die Änderungstransaktion sichergestellt (vgl. Definition 6.5), dass die Anwendung der CoreFlow-Änderungsoperationen die Prozessstruktur von einem konsistenten in einen neuen konsistenten Zustand überführen (*Consistency*). Wurde die Änderungstransaktion erfolgreich in die Prozessstruktur durch die `Commit`-Operation übernommen, so ist sie dauerhaft gespeichert (vgl. Definition 6.5). Erweiterte Mechanismen zur *Isolation* von Änderungstransaktionen existieren in verschiedenen Ansätzen (z.B. [LR00, Rei00]). Wir verzichten auf eine technische Diskussion dieses Aspekts, d.h. wir gehen davon aus, dass ps zwischen Beginn und Abschluss der Änderungstransaktion nicht verändert wird (z.B. durch die Ausführung von Prozessen oder weitere Änderungstransaktionen). Die ACID-Kriterien werden im Folgenden als *erfüllt* angenommen.

6.4 Diskussion

Die automatische Übersetzung von Änderungen einer Datenstruktur in entsprechende Adaptionen der zugehörigen Prozessstruktur ist ein essentieller Mechanismus für die Unterstützung datengetriebener Prozessstrukturen. Wichtige Kriterien sind hierbei einerseits die Erhaltung der Integrität von Daten- und Prozessstruktur. Andererseits muss die Adaption von Prozessstrukturen zur Laufzeit unterstützt werden. Dies erfordert die Definition geeigneter Kriterien, um die Korrektheit der Prozessstrukturen nicht zu gefährden.

6.4.1 Datengetriebene Adaption von Prozessen

Betrachten wir gängige Modellierungs- und Ausführungssprachen (z.B. BPMN, UML2, BPEL) und Prozess-Management-Systeme (z.B. IBM Websphere), ist auf den ersten Blick die multiple Instanziierung eine Möglichkeit, dynamische Aspekte abzubilden (vgl. Workflow Pattern WCP-14 in [AHKB03, RHEA05]). Dieses Konzept reicht jedoch für die dynamische Adaption datengetriebener Prozessstrukturen bei weitem nicht aus, denn es müssen zum einen komplexere Datenstrukturen als Mengen oder Listen abgebildet werden können, zum anderen müssen auch Adaptionsmöglichkeiten für das Hinzufügen bzw. Entfernen von Prozessen und deren Synchronisationen zu *jedem* Zeitpunkt der Ausführung zur Verfügung stehen. Das Workflow Pattern WCP-15 aus [AHKB03, RHEA05] bietet zwar noch eine erweiterte Flexibilität hinsichtlich des Zeitpunkts der multiplen Instanziierung (d.h. während der Ausführung des entsprechenden Blocks), doch auch dies reicht bei weitem nicht an die benötigte Flexibilität heran. So ist es beispielsweise nicht möglich, einen einmal instanziierten Pfad wieder zu entfernen.

Die dynamische Adaption wird von den meisten in Kapitel 3 vorgestellten wissenschaftlichen Ansätzen ebenfalls nicht unterstützt, da ihr Fokus auf der reinen Modellierung liegt [RLA03, KRG07, BGH+07, LBW07]. Weitergehende Ansätze definieren zwar eine operationale Semantik (z.B. basierend auf Petri-Netzen), die dynamische Adaption wird aber selbst bei diesen Ansätzen nicht oder unzureichend unterstützt [KS91, Aal99, AB01a]. Der Case-Handling Ansatz, der mit verschiedenen Ausnahmesituationen umgehen kann, unterstützt ebenfalls die dynamische Adaption des Prozesses nur unzureichend [AB01b, AWG05]. So kann das Entfernen eines Prozessschritts nur durch das Überspringen dieses Schritts (*Skip*-Rolle) zur Laufzeit „simuliert“ werden. Das dynamische Hinzufügen von Prozessschritten ist nicht möglich.

Die integrierten Ansätze AHEAD und WEP (vgl. Abschnitt 3.2) erlauben die Erweiterung der Prozessstruktur durch spätes Modellieren gekennzeichneter Regionen des Prozesses [BDS98, Beu03, JSW99, Wes01]. In AHEAD wird dies durch eine sogenannte Netzerweiterung realisiert, während WEP einen zielorientierten Ansatz verfolgt. Diese Erweiterung erlaubt zwar das Einfügen weiterer Prozessschritte in einen Prozess zur Laufzeit, Konsistenzprüfungen werden aber nicht betrachtet, da Änderungen nur in „noch nicht durchlaufenen“ Regionen möglich sind. Die dynamische Adaption ist damit in einem sehr begrenzten Rahmen möglich. Für die dynamische Adaption datengetriebener Prozessstrukturen sind diese Mechanismen jedoch nicht ausreichend. Zum einen erfordert sie die Veränderung der Prozessstruktur im Ganzen, zum anderen werden wesentlich flexiblere Maßnahmen, wie zum Beispiel das Entfernen von bereits gestarteten oder beendeten Prozessen, benötigt.

In [RR06] wird ein weitergehender Ansatz für die datengetriebene Prozesskoordination und Ausnahmebehandlung vorgestellt. Hier werden (flache) Datenstrukturen (d.h. Mengen und Listen) mit Prozessen verknüpft und entsprechend der Änderung der Datenstruktur der zugehörige Prozessablauf angepasst. Die Beschränkung auf einfache Listen zur (seriellen) Koordination der Prozesse ist jedoch nicht ausreichend, um die Komplexität datengetriebener Prozessstrukturen darzustellen.

6.4.2 Adaption von Prozessen in Prozess-Management-Systemen

Betrachten wir die Prozess-Management-Systeme, stellen wir fest, dass – im Gegensatz zur automatisierten datengetriebenen Adaption – bereits verschiedene Ansätze die manuelle Adaption unterstützen. Hier muss zwischen Ansätzen, die die Adaption von Prozess-Schemata adressieren (z.B. *WIDE* [CCPP98]) und Ansätzen, die Adaption auch auf Instanzebene zulassen (z.B. ADEPT [RD98, Rei00, RRD04b, RRKD05]) unterschieden werden. Dem Modellierer steht in beiden Fällen ein vollständiger Satz an Änderungsoperationen zur Verfügung. Die Herausforderung bei der Adaption bzw. Migration von Prozess-Instanzen ist die Konsistenzanalyse. In *WIDE* wird hierfür das sog. *Compliance* Kriterium eingeführt, mit dessen Hilfe die Konsistenz von Änderungen sichergestellt werden kann [CCPP98, RRD04a, RRD04b]. Die Propagierung der Änderungen auf die Prozess-Instanz ist demnach genau dann erlaubt, wenn der bisherige Verlauf (d.h. die Reihenfolge von Start und Beendigung der Aktivitäten) auf dem geänderten Prozess(-Schema) exakt erzeugt werden kann.

ADEPT erweitert das aus WIDE bekannte *Compliance* Kriterium um weitere Eigenschaften, die beispielsweise die Veränderung von Schleifen tolerieren (*Loop Tolerance*) [RRD04b]. Um die Skalierbarkeit der Prüfung eines adaptierten Prozesses sicherzustellen, sind in ADEPT Vor- und Nachbedingungen für Änderungsoperationen auf Basis der Laufzeitmarkierungen definiert. Damit kann *Compliance* jederzeit garantiert werden, ohne den Verlauf tatsächlich erneut erzeugen zu müssen. Das *Compliance* Kriterium bzw. seine Erweiterung sind für zur Konsistenzprüfung in datengetriebenen Prozessstrukturen nur bedingt geeignet. Es hilft, die Veränderung bereits durchlaufener Regionen eines Prozesses zu erkennen und zu verhindern. Dieses Szenario hat in datengetriebenen Prozessstrukturen jedoch eine besondere Bedeutung. Gerade hier soll die Vergangenheit insofern geändert werden können, als dass laufende oder bereits beendete Prozesse und Synchronisationsbeziehungen (d.h. OLCs und externe Transitionen) entfernt werden können (vgl. Szenario in Abbildung 6.8d). Das Verändern von Schleifen, die derzeit durchlaufen werden (vgl. *Loop Tolerance* Szenario in [RRD04a]), spielt hingegen in CoRepro eine untergeordnete Rolle. Zwar können die Quell- oder Zielzustände externer Transitionen Teil einer Schleife innerhalb eines OLC sein. Wir können im Kontext der Prozessstrukturen diesen Aspekt aber ignorieren, da wir nicht die Schleifen selbst adaptieren. Die Skalierbarkeit der Konsistenzprüfung im Rahmen der dynamischen Adaption ist in CoRepro durch die Prüfung der Laufzeitmarkierungen innerhalb der Änderungsregionen ebenfalls gegeben.

In [RRD04a] werden weitere verallgemeinerte und Metamodell-unabhängige Problemszenarien beschrieben, die im Rahmen dynamischer Adaptionen auftreten können. Das *Dangling-States*-Problem tritt auf, wenn Prozess-Management-Systeme nicht zwischen aktivierten und gestarteten **Aktivitäten** unterscheiden. Die Konsistenzkriterien sind dann häufig zu restriktiv, denn eine aktivierte, aber noch nicht gestartete Aktivität kann im Normalfall problemlos entfernt

werden. Dieses Problem kann jedoch nicht direkt auf datengetriebene Prozessstrukturen übertragen werden. Während auf Aktivitätenebene die Unterscheidung zwischen den Markierungen ACTIVATED und STARTED sinnvoll ist [RD98], ist dies nicht ohne weiteres auf die Koordination von Black-Box-Prozessen übertragbar, denn die internen Prozesszustände sind nicht transparent. Wir sind daher auf möglichst generische Prozesszustände, die von den Prozess-Management-Systemen unterstützt werden, angewiesen. Reihenfolgeänderungen (*Order Changing*) bzw. paralleles Einfügen (*Parallel Insertion*) wird in COREPRO im weitesten Sinne durch die Änderungsoperationen für externe Transitionen unterstützt. Während Reihenfolgeänderungen in COREPRO durch höherwertige Änderungsoperationen realisiert und damit nicht direkt adressiert werden, wird das Problem der Verklemmung bei Einfügen eines parallelen Zweigs (d.h. in COREPRO das Einfügen eines OLC in die Prozessstruktur) durch die automatische Ausnahmebehandlung abgefangen.

Neben der Erkennung von Inkonsistenzen spielt auch deren Behandlung eine große Rolle. Zwar erlaubt ADEPT grundsätzlich nur Änderungen, die nicht zu einer Inkonsistenz führen. Inkonsistenzen, die durch die Neubewertung von Markierungen automatisch aufgelöst werden können, sind jedoch erlaubt. Dafür kapselt ADEPT Änderungsoperationen in einer Änderungstransaktion und führt die Neubewertung der Markierungen automatisch durch. Inkonsistenzen, die nicht automatisch aufgelöst werden können (bzw. ihre auslösenden Änderungsoperationen) sind jedoch nicht erlaubt.[9] In datengetriebenen Prozessstrukturen muss jedoch toleranter mit Inkonsistenzen umgegangen werden. Einerseits muss es in diesem Kontext erlaubt sein, temporäre Inkonsistenzen zu erzeugen. Andererseits sollen die Auswirkungen von Inkonsistenzen fachlich beurteilt und unabhängig von der verursachenden Operation durch verschiedene Mechanismen behandelt werden können (siehe Kapitel 7). Hierfür ist die explizite Kennzeichnung von Inkonsistenzen notwendig.

Der Ansatz *Engineering Processes* erlaubt ebenfalls die Adaption von Prozessstrukturen [EJ01]. Prozesse haben in diesem Ansatz Laufzeitmarkierungen, die mittels Zustandsautomaten abgebildet werden. Sie sind generisch und beispielsweise mit verschiedenen Endzuständen modellierbar. Die Zustandsautomaten verschachtelter Prozesse werden automatisch in einem „vollständigen" Zustandsautomaten integriert. Er dient der Steuerung der Abläufe der Prozessstruktur. Der Ansatz erlaubt die dynamische Adaption von Prozessstrukturen. In diesem Rahmen wird die strukturelle und dynamische Korrektheit von Prozessen geprüft, um syntaktische und semantische Inkonsistenzen zu erkennen. Der Ansatz ermöglicht die Verhinderung von Inkonsistenzen durch die strukturelle Veränderung der Prozessstrukturen (d.h. auch der Object Life Cycles). So werden automatisch Prozesse verschoben (vgl. Prozess *P2* Abbildung 6.13), um Inkonsistenzen durch die Laufzeitmarkierungen auszuschließen. Dies führt allerdings dazu, dass die Prozessstruktur strukturell verändert wird. Das Konzept schließt damit jedoch Rücksprünge aus, ansonsten würde sich der Ablauf vom ursprünglich modellierten Ablauf unterscheiden.

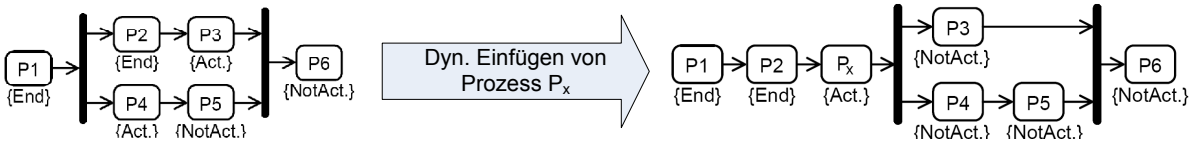


Abbildung 6.13: Dynamische Adaption in *Engineering Processes* [EJ01]

[9] Im Prinzip muss die manuelle Behandlung von Inkonsistenzen *vor* der konsistenzgefährdenden Adaption durchgeführt werden.

Im Sinne der Vergleichbarkeit evaluieren wir abschließend die in COREPRO definierten öffentlichen und die exemplarisch eingeführten höherwertigen Änderungsoperationen (vgl. Abschnitt 6.2) auf Basis der in [WRR07, WRRM08] eingeführten *Change Patterns* (vgl. Abschnitt 3.1.1). Dabei betrachten wir OLCs als *Prozessfragmente* im Sinne der *Change Patterns*. Somit unterstützt COREPRO die Muster

- AP1 - *Fragment einfügen* durch die Operation `InsertObject`.
- AP2 - *Fragment entfernen* durch die Operation `RemoveObject` bzw. die höherwertige Operation `RemObjCompletely`.
- AP3 - *Fragment verschieben* durch das Hinzufügen bzw. Entfernen von Relationen bzw. externer Transitionen. Hierbei ist jedoch kein „freies“ Verschieben möglich, da externe Transitionen auf Modellebene (d.h. im LCM) vordefiniert sind.
- AP4 - *Fragment ersetzen* durch die höherwertige Operation `ExchangeObject`.
- AP9 - *Aktivitäten parallelisieren* durch das Entfernen von Relationen bzw. externer Transitionen.
- AP10 - *Kontrollflussabhängigkeit hinzufügen* durch das Hinzufügen von Relationen bzw. externer Transitionen. Hierbei ist jedoch kein „freies“ Synchronisieren möglich, da externe Transitionen auf Modellebene (d.h. im LCM) vordefiniert sind.
- AP11 - *Kontrollflussabhängigkeit entfernen* durch das Entfernen von Relationen bzw. externer Transitionen.
- AP14 - *Prozessfragment kopieren* durch das Einfügen neuer Objekte bzw. OLCs (und ggf. manuelles Anpassen der Markierung; siehe Ausnahmebehandlung in Kapitel 7). Hierbei ist jedoch die Auswahl der Prozessfragmente auf die im Life Cycle Coordination Model modellierten OLCs beschränkt.
- PP3 - *Später Aufbau der Prozessstruktur* durch dynamisches Hinzufügen von Relationen bzw. externer Transitionen. Hierbei ist jedoch kein „freies“ Synchronisieren möglich, da externe Transitionen auf Modellebene (d.h. im LCM) vordefiniert sind.
- PP4 - *Multi Instance Activity* durch das Einfügen neuer Objekte in die Datenstruktur und die automatische Adaption der Prozessstruktur.

Die dynamische Adaption von OLCs selbst, wie sie beispielsweise im Projekt *HieraStates* [Tee96, Tee98] vorgeschlagen wird, ist keine Anforderung an die COREPRO-Konzepte. COREPRO bietet jedoch durch die Trennung von Modell- und Instanzebene sowie der präzisen operationalen Semantik und zugehöriger Korrektheitskriterien beste Voraussetzungen für die Umsetzung derartiger Anforderungen.

6.5 Zusammenfassung

Die Konzepte zur Adaption datengetriebener Prozessstrukturen bilden einen wichtigen Grundpfeiler des COREPRO-Ansatzes. Das vorliegende Kapitel führt die dafür notwendigen Änderungsoperationen sowohl für Daten- als auch Prozessstrukturen ein. Die direkte Verknüpfung der Operationen und ihre Bereitstellung in Form einer öffentlichen Schnittstelle unterstützt nicht nur die Adaption datengetriebener Prozessstrukturen auf hoher Abstraktionsebene, sondern auch die Definition höherwertiger Änderungsoperationen. Ferner erlauben wir die dynamische Anwendung

der Operationen und definieren hierfür geeignete Konsistenzkriterien. Sie stellen in Verbindung mit dem eingeführten Transaktionsmechanismus sicher, dass nur Änderungen der Prozessstruktur angewendet werden, die ihre Korrektheit nicht gefährden. Tritt eine Inkonsistenz auf, wird zunächst versucht, diese durch die automatische Ausnahmebehandlung aufzulösen. Gelingt dies nicht, kann der Nutzer die Änderungstransaktion verwerfen oder aber die Prozessstruktur neu initialisieren und deren Ausführung erneut beginnen.

Die Konzepte liefern damit wichtige Grundlagen für die Anwendung des CoRePro-Ansatzes in der Praxis. In der untersuchten Domäne sind diese Möglichkeiten jedoch noch nicht ausreichend. Hier sind weitergehende Mechanismen gefragt, die eine fallspezifische Behandlung der Inkonsistenz zulassen. Diese werden in Kapitel 7 vorgestellt.

7

Ausnahmebehandlung in datengetriebenen Prozessstrukturen

Datengetriebene Prozessstrukturen sind typischerweise lang laufend und werden in einem dynamischen Umfeld ausgeführt. So können fachliche Umstände eine dynamische Adaption der Prozessstruktur erfordern. Tritt bei der dynamischen Adaption eine Ausnahme (d.h. eine Inkonsistenz) auf, wird diese gekennzeichnet und – wenn möglich – im Rahmen der automatischen Ausnahmebehandlung aufgelöst. Ist diese Auflösung nicht möglich, kann der Benutzer mithilfe der in Kapitel 6 vorgestellten Mechanismen weitere dynamische Adaptionen durchführen oder die Änderungstransaktion verwerfen, um die Prozessstruktur wieder in einen konsistenten Zustand zu überführen und mit ihrer Ausführung fortfahren zu können.

In der Fahrzeugentwicklung kann eine Ausnahmesituation nicht nur durch die dynamische Adaption einer Prozessstruktur, sondern beispielsweise auch durch außerplanmäßiges Auftreten eines technischen Fehlers in einem elektronischen System entstehen. Dies kann zur Folge haben, dass bestimmte Prozesse der Prozessstruktur wiederholt werden müssen. In diesem Fall sind die bislang vorgestellten Mechanismen zur Ausnahmebehandlung noch nicht ausreichend. Es müssen weitere Methoden für die flexible Ausführung der Prozessstruktur sowie für das Auflösen von Inkonsistenzen – auch im Rahmen der dynamischen Adaption – zur Verfügung gestellt werden. Mit ihrer Hilfe soll sich eine neue, konsistente Markierung der Prozessstruktur (d.h. *Forward Recovery*) herstellen lassen. Darüber hinaus sind robuste Techniken für das Ignorieren von Inkonsistenzen vonnöten, um den sich in der Praxis stellenden Anforderungen in vollem Umgang gerecht zu werden. Die Erhaltung der Korrektheitseigenschaften der Prozessstruktur muss weiterhin durchgehend gewährleistet sein.

Das vorliegende Kapitel gliedert sich wie folgt: Abschnitt 7.1 beschreibt einerseits verschiedene Abläufe der E/E-Entwicklung, für die eine adäquate Ausnahmebehandlung essentiell ist, und leitet andererseits daraus relevante Anforderungen ab. Abschnitt 7.2 führt diejenigen Operationen ein, die für eine entsprechende Ausnahmebehandlung (bzw. *Forward Recovery*) nötig sind.

Abschnitt 7.3 diskutiert die Anwendung dieser Operationen zur Behandlung von Inkonsistenzen in einer Prozessstruktur. Abschnitt 7.4 beschreibt erweiterte Operationen zum Ignorieren von Inkonsistenzen. In Abschnitt 7.5 geben wir einen Ausblick auf weitere Szenarien aus der Praxis, in denen eine flexible Reaktion auf Ausnahmen notwendig wird, sowie auf deren mögliche Umsetzung. Abschnitt 7.6 diskutiert verwandte Arbeiten. Abschnitt 7.7 gibt eine Zusammenfassung und schließt damit Teil II dieser Arbeit ab.

7.1 Einleitung

Während der Ausführung einer Prozessstruktur können verschiedene Ausnahmesituationen auftreten. Ausnahmen lassen sich unterscheiden in *planbare* und *nicht-planbare Ausnahmen* (vgl. Abbildung 7.1). Während planbare Ausnahmen (z.B. Aufdecken eines Fehlers in einem elektronischen System während einer Absicherungsfahrt) bereits zur Modellierzeit bekannt sind und mittels modellierter Verzweigungen auf Basis des Prozessergebnisses behandelt werden können, erfordern nicht-planbare Ausnahmen eine flexible manuelle Behandlung zur Laufzeit. Sie können noch weiter unterschieden werden in technische Ausnahmen (z.B. Inkonsistenz während dynamischer Adaption oder technischer Fehler bei Ausführung eines Prozesses) und fachliche Ausnahmen (z.B. fehlerhafter Versuchsaufbau oder Abbruch einer Probefahrt wegen Fahrzeugdefekts). Im Gegensatz zu technischen Ausnahmen können fachliche Ausnahmen vom System nicht automatisch erkannt werden. Die Herausforderung bei der Behandlung fachlicher Ausnahmen ist es, dem Nutzer geeignete Behandlungs- bzw. Eingriffsmöglichkeiten zur Verfügung zu stellen. Diese müssen die realen Möglichkeiten so gut wie möglich abdecken. So müssen geeignete Methoden bereitgestellt werden, die beispielsweise die notwendig gewordene Wiederholung eines Absicherungsprozesses in der Prozessstruktur erlauben. Unser Ziel ist es, generische Operationen zur Verfügung zu stellen, die es erlauben auf eine Ausnahme angemessen und praxisnah zu reagieren ohne die Korrektheit der Prozessstruktur zu gefährden oder vom fachlich geforderten Vorgehen abweichen zu müssen. Hierfür wollen wir folgende Fragestellungen beantworten:

- Welche fachlichen Ausnahmen können auftreten und wie wirken sich diese auf die Prozessstruktur aus?
- Welche Operationen werden benötigt, um Ausnahmesituationen zu kontrollieren bzw. Inkonsistenzen aufzulösen?
- Wie wirkt sich die Auflösung einer Inkonsistenz auf die Prozessstruktur aus?
- In welcher Reihenfolge müssen Operationen angewandt werden, um ein sinnvolles Ergebnis zu erhalten und eine möglichst effiziente Ausnahmebehandlung zu realisieren?
- Wie lassen sich Inkonsistenzen ignorieren und dennoch die Korrektheit der Prozessstruktur sicherstellen?

7.1.1 Szenarien

Unsere Anforderungsanalyse aus Abschnitt 2.1 hat gezeigt, dass datengetriebene Prozessstrukturen in der E/E-Entwicklung nur sinnvoll eingesetzt werden können, wenn sie durch geeignete

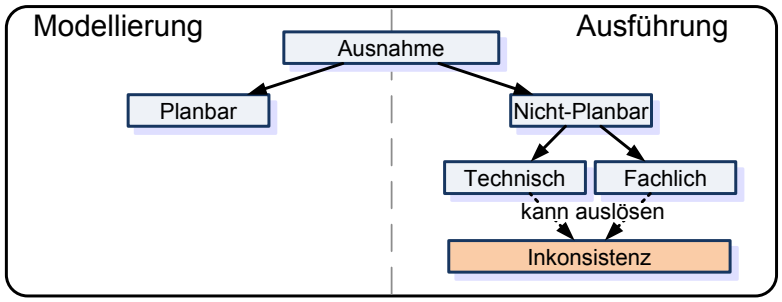


Abbildung 7.1: Einordnung von Ausnahmesituationen

Mechanismen zur Ausnahmebehandlung unterstützt werden. Die Abschnitte 4.2 und 6.3.3 haben bereits Möglichkeiten zur Behandlung planbarer Ausnahmen und Inkonsistenzen beschrieben. Wir stellen nun weitere Beispielszenarien vor, die geeignet unterstützt werden müssen.

Szenario 7.1 (Manueller Rücksprung)
Während der Absicherung eines Fahrzeugs können Fehler in elektronischen Komponenten auftreten, deren Auswirkungen auf die Prozessstruktur nur schwer vorhersagbar und damit nicht modellierbar sind. Abhängig von der Fehlerschwere und weiterer fachlicher Kriterien, müssen ggf. die Komponente überarbeitet und Absicherungsprozesse wiederholt werden (*Regressionstests*). In der Prozessstruktur soll eine derartige Wiederholung durch einen *Rücksprung* (d.h. Rückwärtssprung) innerhalb des betroffenen OLC auf einen vormals aktivierten Zustand geschehen.

Der Rücksprung kann zu einer „Fortpflanzung“ des Fehlers an das übergeordnete Gesamtsystem führen. Wurden hier bereits Prozesse nach Installation des fehlerhaften Systems ausgeführt, müssen diese ggf. wiederholt werden. Ist beispielsweise eine Testfahrt mit einem fehlerhaften Steuergerät der Klimaautomatik durchgeführt worden, könnten die Testergebnisse hinsichtlich des Temperaturverhaltens nicht aussagekräftig sein.

Szenario 7.2 (Manueller Vorsprung)
In Szenario 6.2 wird ein Fall vorgestellt, das den Austausch einer Komponente beschreibt. Im Rahmen der dynamischen Adaption wird dies durch das Entfernen des betroffenen Objekts und seines OLC sowie durch Hinzufügen eines neuen Objekts und dessen OLC realisiert. Damit werden alle Prozesse des OLC erneut ausgeführt. In bestimmten Szenarien ist es jedoch nicht notwendig, alle Prozesse erneut auszuführen (z.B. wenn nur geringe Änderungen vorgenommen wurden). In diesem Fall macht es Sinn, auf Basis einer fachlichen Entscheidung einen bestimmten Zustand „manuell“ zu aktivieren, also einen *Vorsprung* (d.h. Vorwärtssprung) vom derzeit aktivierten Zustand (d.h. dem Startzustand im beschriebenen Szenario) zu einem bislang noch nicht erreichten Zustand innerhalb des OLC vorzunehmen.

Szenario 7.3 (Manuelles Rücksetzen einer externen Transition)
Ausnahmen können nicht nur OLCs betreffen, sondern auch Synchronisationsprozesse. Wird beispielsweise festgestellt, dass ein Synchronisationsprozess zur Installation einer Komponente fehlerhaft durchgeführt wurde (z.B. Synchronisationsprozess `Installation` in Szenario 6.2), muss dieser erneut ausgeführt werden können. Es wird also ein Pendant zum Rücksprung auf OLC-Ebene benötigt (vgl. Beispiel 7.1).

Szenario 7.4 (Dynamische Adaption)
In Abschnitt 6.3 werden Konsistenzkriterien für die dynamische Adaption von Prozessstrukturen beschrieben. Kommt es zu einer Inkonsistenz, die nicht durch die automatische Ausnahmebehandlung aufgelöst werden kann, ist die Durchführung der Änderungsoperation zunächst ausgeschlossen worden. In der Praxis kann die Durchführung einer Änderung jedoch zwingend notwendig sein, selbst wenn es dadurch zu einer Inkonsistenz kommt. Der Austausch eines Steuergeräts etwa muss aus Gründen der Sicherheit, Qualität oder Kosten auch dann durchgeführt werden, wenn es bereits in einem Testfahrzeug verbaut ist (vgl. Abbildung 7.2a). Das Entfernen des Objekts `Main Unit V1.0` (bzw. des entsprechenden OLC) führt in Abbildung 7.2b zunächst zu einer semantischen Inkonsistenz in Zustand `Navigation.Eingebaut`. Das anschließende Hinzufügen des Objekts `Main Unit V1.1` führt im Prinzip zu einer syntaktischen Inkonsistenz. Somit wird der Zustand durch Kennzeichnungsregel KR1 als `INCONSISTENT` gekennzeichnet (die Kennzeichnungsfunktion aus Definition 6.3 unterscheidet nicht zwischen syntaktischen und semantischen Inkonsistenzen; vgl. Abschnitt 6.3). Der Ansatz muss das gezielte Auflösen von syntaktischen und semantischen Inkonsistenzen erlauben, um die Ausführung mit einer konsistenten Prozessstruktur fortsetzen zu können. Im erwähnten Beispiel müssen dazu alle bereits durchgeführten Absicherungsprozesse erneut durchgeführt werden (vgl. Abbildung 7.2c).

In bestimmten Situationen kann es (aus fachlicher Sicht) sinnvoll sein, eine Inkonsistenz zu ignorieren. In unserem Beispiel aus Abbildung 7.2c könnte etwa auf den Rücksprung im OLC `Navigation` verzichtet werden, wenn dies aus zeitlichen Gründen nicht mehr rentabel ist (z.B. wenn sich das Fahrzeug bereits auf einer Testfahrt befindet und die ausgetauschte Komponente keine sicherheitskritischen Einflüsse hat). In diesem Fall wird die Inkonsistenz im weiteren Ablauf ignoriert, wobei Sie weiterhin bestehen bleibt und entsprechend gekennzeichnet werden muss.

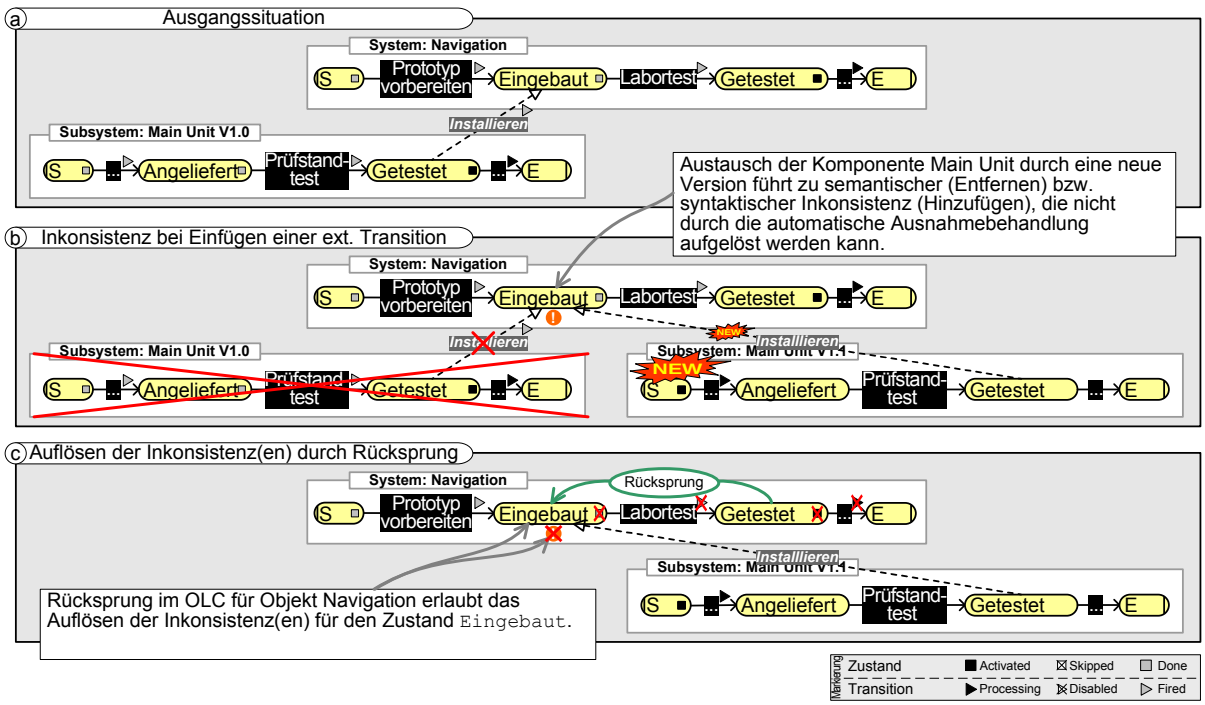


Abbildung 7.2: Behandlung einer entstandenen Inkonsistenz nach dynamischer Adaption

Szenario 7.5 (Übergehen von Prozess-Synchronisationen)
Die Anforderungsanalyse aus Abschnitt 2.1 hat gezeigt, dass die nebenläufige Ausführung der Prozesse innerhalb einer Prozessstruktur und ihre Synchronisation für ein systematisches Vorgehen zwingend erforderlich sind (*Concurrent Engineering*). Es kann jedoch aufgrund von zeitlichen Anforderungen notwendig sein, Synchronisationsbeziehungen gezielt zu übergehen, um mit dem Ablauf unsynchronisiert fortfahren zu können. Das Übergehen einer Synchronisationsbeziehung soll nicht zum Abbruch des mit ihr verknüpften Synchronisationsprozesses führen. Stattdessen soll der Synchronisationsprozess weiter ausgeführt, allerdings nicht auf dessen Beendigung gewartet werden. Dies muss im Protokoll der Prozessausführung entsprechend vermerkt werden, um den Eingriff nachvollziehbar zu dokumentieren (vgl. Abschnitt 2.2.1).

7.1.2 Anforderungen

Ausgehend von den vorgestellten Szenarien können wir allgemeine Anforderungen an die Ausnahmebehandlung in datengetriebenen Prozessstrukturen ableiten. Die Szenarien haben zum Ausdruck gebracht, dass unabhängig von der Art einer nicht-planbaren Ausnahmesituation (vgl. Abbildung 7.1) die notwendigen Behandlungsmethoden auf generische Operationen und Mechanismen zurückzuführen sind (vgl. Szenario aus Abbildung 7.2). Eine entscheidende Erkenntnis aus der Analyse der Szenarien ist, dass sowohl syntaktische als auch semantische Inkonsistenzen durch eine Anpassung der Laufzeitmarkierungen (z.B. durch einen Sprung) aufgelöst werden können und es dabei keiner Fallunterscheidung bedarf (siehe Abschnitt 7.3).

Die beschriebenen Szenarien lassen sich im Grunde auf vier Aspekte reduzieren: Erstens muss es möglich sein, von der Ausführung der Prozessstruktur flexibel abzuweichen, indem zum Beispiel Sprünge durchgeführt werden. Zweitens müssen Inkonsistenzen, die in diesem Kontext auftreten, erkannt werden. Drittens sollen ihre Auswirkungen auf die Prozessstruktur geprüft und die Inkonsistenzen durch geeignete Mechanismen (z.B. Sprünge) behandelt werden können. Viertens muss es möglich sein, die Ausführung der Prozessstruktur ohne Behandlung der Inkonsistenzen fortzusetzen (d.h. Inkonsistenzen werden ignoriert), ohne dabei auf die bislang zugesicherten dynamischen Eigenschaften verzichten zu müssen.

Anforderung 7.1 (Flexible Ausführung)
Eine fachliche Ausnahme wird im Regelfall durch den Benutzer erkannt und von ihm durch geeignete Operationen behandelt (z.B. Rücksprung innerhalb eines OLC zur Wiederholung von Prozessen). Die Szenarien 7.1, 7.2 und 7.3 beschreiben derartige Situationen. Als Anforderung ergibt sich hieraus die Definition einer geeigneten Sprungoperation für OLCs. Ein Sprung ist – formal gesehen – eine Abweichung vom vorgesehenen Ablauf, indem eine andere Laufzeitmarkierung der Prozessstruktur hergestellt wird. Dementsprechend müssen die Markierungen ausgewählter Zustände und Transitionen durch geeignete Vor- und Rücksprungoperationen für OLCs sowie eine Rücksetzoperation für externe Transitionen verändert werden können. Diese Operationen bilden auch die Grundlage für die Kontrolle von Ausnahmesituationen.

Anforderung 7.2 (Erkennen von Inkonsistenzen)
Werden die Markierungen von Zuständen oder Transitionen angepasst, können, wie schon bei der dynamischen Adaption einer Prozessstruktur, wieder Inkonsistenzen entstehen. Diese müssen zuverlässig erkannt und entsprechend gekennzeichnet werden.

Anforderung 7.3 (Kontrolle von Inkonsistenzen)
Ist während der Durchführung einer Markierungsänderung oder der dynamischen Adaption eine Inkonsistenz aufgetreten, muss diese geeignet behandelt werden. Hierbei reicht es häufig nicht aus, die Änderungen im Rahmen der Änderungstransaktion zu verwerfen. Stattdessen müssen geeignete Mechanismen entwickelt werden, welche das Auflösen einer Inkonsistenz erlauben. Auch die Anwendung derartiger Mechanismen muss kontrolliert erfolgen, um die Korrektheit der Prozessstruktur weiterhin sicherzustellen. Ferner müssen in diesem Kontext Inkonsistenzen nicht nur identifiziert, sondern auch ihre Konsequenzen auf die gesamte Prozessstruktur geprüft und dargestellt werden.

Anforderung 7.4 (Ignorieren von Inkonsistenzen)
Aus fachlichen Gründen kann es Sinn machen, eine Inkonsistenz oder eine externe Transition zu ignorieren (vgl. Szenarien 7.4 und 7.5). Dazu muss die Markierung eines Zustands oder einer externen Transition gezielt vernachlässigt werden können, um in der Ausführung der Prozessstruktur fortzufahren. Während die Vernachlässigung semantischer Inkonsistenzen keine korrektheitsgefährdenden Konsequenzen nach sich zieht (vgl. Abschnitt 6.3.2), muss mit syntaktischen Inkonsistenzen geeignet umgegangen werden, um weiterhin die Korrektheit der Prozessstruktur und damit ein vorhersagbares Verhalten gewährleisten zu können (vgl. Kapitel 6). Das Ignorieren von Inkonsistenzen bzw. externen Transitionen muss dokumentiert werden, denn die Prozessstruktur befindet sich dann fachlich in einem „potentiell inkonsistenten Zustand“.

7.2 Operationen zur Anpassung von Laufzeitmarkierungen

Wir definieren im Folgenden die zur Ausnahmebehandlung notwendigen Konzepte. Hierfür sind insbesondere *Markierungsoperationen* vonnöten, die eine kontrollierte Anpassung der Laufzeitmarkierung erlauben (vgl. Anforderung 7.1), zum Beispiel in Form von *Rück-* und *Vorsprungoperationen*. Die Fehlfunktion einer Komponente kann es beispielsweise erfordern, innerhalb eines OLC bestimmte Prozesse zu wiederholen, also einen Ad-hoc-Rücksprung durchzuführen.

Herausforderung ist die Anpassung der Markierungen unter Beibehaltung der dynamischen Korrektheit der OLCs bzw. der gesamten Prozessstruktur (vgl. Abschnitte 4.4 und 4.7). Hierfür müssen im Kontext der Anwendung der Markierungsoperationen auftretende Inkonsistenzen korrekt erkannt werden (vgl. Abbildung 7.1). Wir definieren daher Konsistenzkriterien, mit deren Hilfe derartige Inkonsistenzen identifiziert und dann entsprechend des aus Abbildung 6.11 (siehe Seite 150) bekannten Ablaufs behandelt werden können.

7.2.1 Sprungoperation für OLCs

Die Szenarien 7.1 und 7.2 zeigen die Notwendigkeit für die Einführung einer Sprungoperation für OLCs. Sie soll die Markierungen der Zustände und Transitionen eines OLC automatisch anpassen. Herausforderung bei der technischen Realisierung ist es, die dynamische Korrektheit des OLC auch nach Durchführung des Sprungs sicherzustellen. Dies wollen wir gewährleisten, indem wir eine Laufzeitmarkierung für den OLC erzeugen, die im „normalen Ablauf“ ebenfalls hergestellt werden kann. Schließlich ist im normalen Ablauf die dynamische Korrektheit immer garantiert (vgl. Abschnitt 4.4). Wird beispielsweise ein Rücksprung innerhalb des OLC aus Abbildung 7.3a

vom derzeit aktiven Zustand `A4` auf Zustand `A1` durchgeführt, soll die Laufzeitmarkierung aus Abbildung 7.3b hergestellt werden. Die Realisierung der Sprungoperation kann in OLCs jedoch grundsätzlich auf verschiedene Arten geschehen. Wir betrachten zunächst verschiedene Lösungsalternativen, stellen dann die technische Umsetzung der Sprungoperation vor und diskutieren abschließend deren Einfluss auf die dynamischen Eigenschaften des OLC.

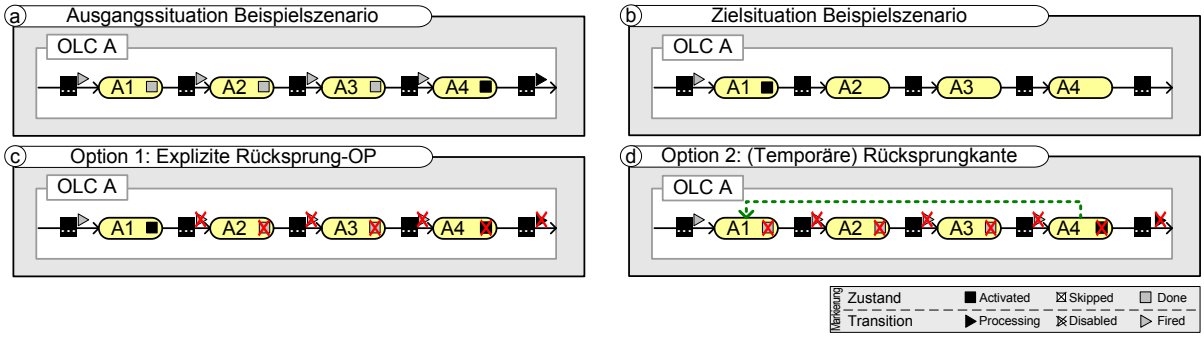


Abbildung 7.3: Optionen für die Umsetzung des Rücksprungs

Lösungsalternativen

Eine mögliche Vorgehensweise für die Realisierung des *Rücksprungs* ist es, den Zielzustand von `DONE` auf `ACTIVATED` zu markieren. Um eine konsistente Markierung innerhalb des OLC herzustellen, müssen anschließend die Markierungen von Transitionen und Zuständen zwischen dem vormals aktivierten Zustand und dem Zielzustand zurückgesetzt werden (vgl. Abbildung 7.3c). Ein Vorsprung kann auf analoge Art und Weise durchgeführt werden, indem der Zielzustand von `NOTACTIVATED` auf `ACTIVATED` markiert und alle zwischen dem vormals aktivierten Zustand und dem Zielzustand liegenden Transitionen und Zustände abgewählt werden.[1]

Diese Vorgehensweise erinnert an die Mechanismen der operationalen Semantik aus Kapitel 4. Auch hier werden bei einem Rücksprung Teile des OLC reinitialisiert bzw. bei einer bedingten Verzweigung abgewählt. Um etwa einen Rücksprung zu realisieren, könnte eine interne (Rücksprung-)Transition (mit Markierung `FIRED`) zwischen dem derzeit aktivierten Zustand und dem Zielzustand eingefügt werden (vgl. Abbildung 7.3d). Da die Struktur eines OLC nicht auf Blockstrukturen beschränkt ist, lässt sich solch ein (Ad-hoc-) Sprung von jedem Quell- in jeden Zielzustand (d.h. auch in abgewählte Pfade) realisieren. Die operationale Semantik mit den Markierungsregeln MR2 und MR4 sowie der Ausführungsregel AR4 sorgt dann automatisch für die korrekte Reinitialisierung der zwischen dem Zielzustand des Sprungs und dessen Quellzustand liegenden internen Transitionen und Zustände. Derselbe Mechanismus kann außerdem für die Realisierung eines Vorsprungs genutzt werden.

Obwohl wir die Sprungoperation zunächst für isolierte OLCs betrachten, dürfen bei der Diskussion der Alternativen zur Realisierung der Sprungoperationen externe Transitionen nicht außer Acht gelassen werden. Abbildung 7.4a entspricht der Ausgangssituation aus Abbildung 7.3a. Abbildung 7.4b zeigt denselben Ausschnitt, aber mit eingehender externer Transition in Zustand `A1`.

[1] Die Abwahl der Transitionen und Zustände stellt nur eine mögliche Realisierung dar. Es könnte ebenso eine neue Markierung eingeführt oder ein möglicher Pfad zum aktivierten Zustand als „durchlaufen“ markiert werden.

Wurde die eingehende externe Transition bereits gefeuert oder abgewählt, kann `A1` bei dem oben diskutierten Rücksprung entsprechend der operationalen Semantik aus Abschnitt 4.6 direkt aktiviert werden (vgl. Markierungsregel MR7). Wird die externe Transition jedoch durch eine dynamische Adaption hinzugefügt[2] (vgl. Abschnitt 6.3), befindet sie sich in ihrer Initialmarkierung (vgl. Abbildung 7.4b). Dann muss ein klares Verhalten für die Markierung des Zielzustands des Sprungs definiert werden.[3] Eine Möglichkeit wäre es, den Zielzustand des Rücksprungs nicht als `ACTIVATED`, sondern als `NOTACTIVATED` zu markieren. Allerdings muss dann entschieden werden, welcher Zustand stattdessen aktiviert wird. Schließlich muss es während der Ausführung einer Prozessstruktur in jedem OLC einen Zustand mit der Markierung `ACTIVATED` geben, der einen definierten Zustand für das mit ihm verknüpfte Objekt repräsentiert. Prinzipiell könnte einfach der Vorgängerzustand des Sprungziels aktiviert werden. Dies ist allerdings problematisch, wenn der Zielzustand des Rücksprungs mehrere eingehende interne Transitionen (z.B. aus vorausgehenden Verzweigungen oder aus modellierten Rücksprüngen) besitzt und damit keine nachvollziehbare Anzeige des derzeit aktivierten Zustands mehr möglich ist. Diese Aspekte gelten auch für den Vorsprung.

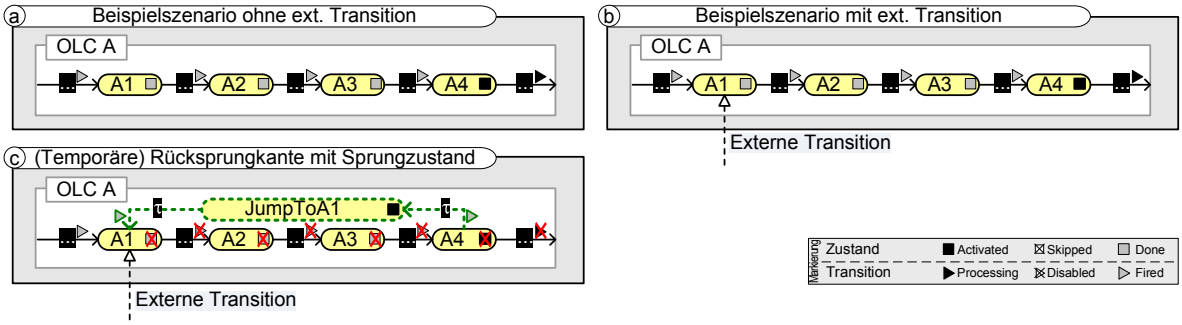


Abbildung 7.4: Betrachtung der Rücksprungoperation ohne und mit externen Transitionen

Eine gute Lösung lässt sich in Verbindung mit Option 2 aus Abbildung 7.3d erreichen, wenn zusätzlich ein temporärer Zwischenzustand eingefügt wird. So wird in Abbildung 7.4c der Zwischenzustand `JumpToA1` mit jeweils einer eingehenden und einer ausgehenden temporären Transition eingefügt. Die in `JumpToA1` eingehende Transition feuert sofort bei Durchführung der Sprungoperation und aktiviert damit den Zwischenzustand. Entsprechend der Ausführungsregel AR1 kommt die aus `JumpToA1` ausgehende interne Transition zur Ausführung. Da sie mit dem Stellvertreterprozess τ verknüpft ist, wird sie sofort als `FIRED` markiert. Im Allgemeinen sind die in den Zielzustand des Sprungs eingehenden externen Transitionen bereits als `FIRED` oder `DISABLED` markiert und der Zielzustand des Sprungs kann direkt aktiviert werden. Ist jedoch eine der in den Zielzustand eingehenden externen Transitionen noch nicht als `FIRED` oder `DISABLED` markiert, bleibt der Zwischenzustand `JumpToA1` aktiviert und der Zielzustand in seiner ursprünglichen Markierung. Der aktivierte Zwischenzustand zeigt „nach außen“ an, dass derzeit ein Rücksprung durchgeführt wird. Vorteil dieser Vorgehensweise ist die intuitive Nutzung der vorhandenen Konstrukte. Für die Durchführung der Rücksprungoperation wird also die operationale Semantik vormodellierter Rücksprünge (vgl. Abschnitt 4.3.6) angewendet.[4]

[2] Diese kann zum Beispiel in derselben Änderungstransaktion wie der Rücksprung selbst erfolgen.
[3] Abschnitt 7.4.4 beschreibt weitere Mechanismen, um das „Warten“ auf eine eingehende externe Transition zu umgehen.
[4] Der Sprung aus einem temporären Sprungzustand wird in Abschnitt 7.5 beschrieben.

Das Problemszenario aus Abbildung 7.4a kann auch in Verbindung mit einem Vorsprung eintreten. Die vorgestellten Mechanismen der Option 2 aus Abbildung 7.3d lassen sich jedoch grundsätzlich auch für den Vorsprung einsetzen. Ein vormodellierter Vorsprung wird in der Regel durch eine bedingte Verzweigung realisiert. Während der Ausführung werden die Markierungen der übersprungenen Elemente nicht zurückgesetzt, sondern entsprechend der operationalen Semantik nicht-deterministischer OLCs abgewählt. Auch für den Vorsprung sind damit die aus Abschnitt 4.3 bekannten Regeln anwendbar.

Technische Umsetzung

In COREPRO werden interne Transitionen formal nicht in vorwärts- und rückwärtsgerichtete Transitionen unterschieden (vgl. Abschnitt 4.2). Wir können uns dies bei der Definition geeigneter Sprungoperationen zunutze machen und so auf eine (aufwändige) Fallunterscheidung zwischen Vor- und Rücksprung verzichten. Um dem Benutzer die Durchführung von Sprüngen auf hoher Abstraktionsebene zu erlauben, definieren wir die Sprungoperation `JUMP` (vgl. Operation MO1). Sie fügt, unabhängig von der Sprungrichtung, automatisch die erforderlichen Sprungtransitionen mit einem Zwischenzustand ein (vgl. Abbildung 7.5a).[5]

Die Sprungtransitionen und der eingefügte Zustand haben die aus Kapitel 4 bekannte operationale Semantik, sie verfügen jedoch zusätzlich über die Kennzeichnung `TEMPORARY`.[6] Die Sprungtransitionen sind mit dem Stellvertreterprozess τ verknüpft (vgl. Abbildung 7.5a). Wird die Sprungtransition t_{jump1} als `PROCESSING` markiert, feuert sie sofort, da sie mit dem Stellvertreterprozess τ verknüpft ist (vgl. Abbildung 7.5b).[7] Markierungsregel MR2 wird automatisch für die Transition t_{jump1} angewendet. Dadurch werden die anderen aus dem derzeit aktivierten Zustand s_{src} ausgehenden Transitionen abgewählt und die CANCEL-Operation auf dem mit diesen Transitionen verknüpften Prozess angewendet. Markierungsregel MR1 aktiviert daraufhin Zustand s_{jump1} als `ACTIVATED`. Damit ist der Kontext von Ausführungsregel AR1 erfüllt, die daraufhin t_{jump2} als `PROCESSING` markiert. Da t_{jump2} ebenfalls mit dem Stellvertreterprozess τ verknüpft ist, ist sofort der Kontext von Markierungsregel MR1 erfüllt. Die Anwendung von MR1 markiert dann t_{jump2} als `FIRED` und den Zielzustand s_{targ} als `ACTIVATED` (vgl. Abbildung 7.5b).

[5] Die Sprungrichtung ist auf Basis der aktuellen Laufzeitmarkierungen jederzeit nachvollziehbar.

[6] Wir verzichten an dieser Stelle auf eine formale Definition der Operationen für das Einfügen und Entfernen von Transitionen und Zuständen sowie deren Kennzeichnung als `TEMPORARY`.

[7] Wir verstoßen damit zugunsten der Übersichtlichkeit der Lösung gegen die zugesicherte strukturelle Eigenschaft von OLCs, dass mehrere aus einem Zustand ausgehende interne Transitionen mit demselben Prozess verknüpft sind. Es ist sichergestellt, dass dies nicht zu einer Gefährdung der Korrektheit führt (siehe folgende Abschnitte).

Markierungsoperation MO1 (Sprungoperation JUMP)
Seien $ps = (OLC, EST) \in \mathcal{PS}$ *eine Prozessstruktur und* $olc = (P, V, TS) \in OLC$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$. *Wird die Sprungoperation* JUMP *auf den Zustand* $s_{src} \in S, SM(s_{src}) = \text{ACTIVATED} \wedge SF(s_{src}) \neq \text{TEMPORARY}$ *und den Zielzustand* $s_{targ} \in S$ *angewendet, werden folgende Schritte durchgeführt:*

1. *Es wird ein temporärer Zustand* s_{jump1} *mit Initialmarkierung* NOTACTIVATED *zu* S *hinzugefügt.*
2. *Es wird eine temporäre Transition* $t_{jump1} = (s_{src}, (\tau, \sigma(\tau)), s_{jump1})$ *mit Initialmarkierung* WAITING *zu* T *hinzugefügt.*
3. *Es wird eine temporäre Transition* $t_{jump2} = (s_{jump1}, (\tau, \sigma(\tau)), s_{targ})$ *mit Initialmarkierung* WAITING *zu* T *hinzugefügt.*
4. *Transition* t_{jump1} *wird als* $t_{jump1} =$ PROCESSING *markiert, d.h. der Sprung wird eingeleitet und die entsprechenden Markierungs- und Ausführungsregeln werden angewendet.*
5. *Zielzustand* s_{targ} *wird als* $SM(s_{targ}) :=$ NOTACTIVATED *markiert, falls* $\exists\, e \in inTrans_{ext} : EM(e) \notin \{\text{FIRED}, \text{DISABLED}\}$.
6. *Wird der Sprungzustand* s_{jump1} *als* DONE *markiert, ist also der Sprung vollständig durchgeführt worden, werden der Sprungzustand* s_{jump1} *sowie seine ein- und ausgehenden temporären Transitionen* t_{jump1} *und* t_{jump2} *wieder entfernt.*

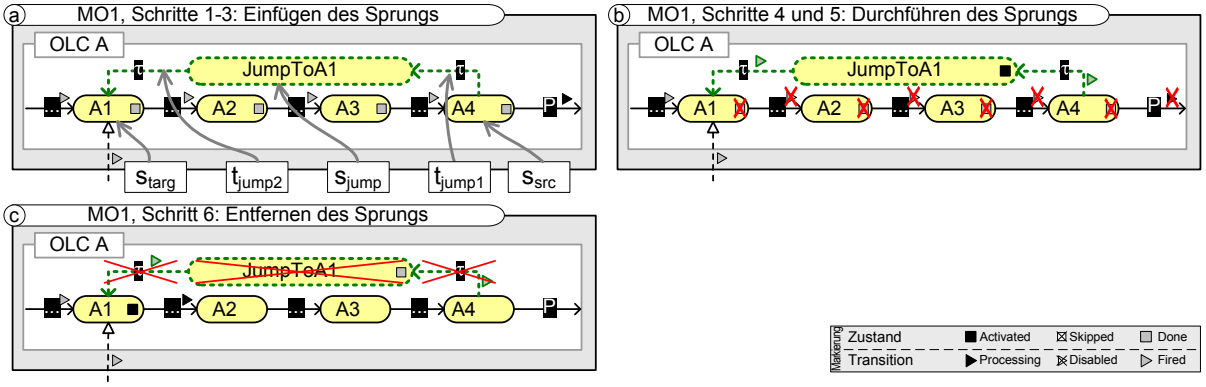


Abbildung 7.5: Rücksprung innerhalb eines OLC durch Markierungsoperation JUMP

Sind nicht alle in den Zielzustand des Sprungs eingehenden externen Transition als FIRED oder DISABLED markiert, kann dessen Aktivierung entsprechend Markierungsregel MR7 nicht erlaubt werden. Bei einem Rücksprung im normalen Ablauf tritt eine solche Situation nicht auf, denn zum Zeitpunkt eines Schleifenrücksprungs wurde der Zielzustand bereits mindestens einmal durchlaufen und eingehende externe Transitionen sind daher immer als FIRED oder DISABLED markiert. Ansonsten hätte die Rücksprungtransition gar nicht erreicht werden können. Der Zielzustand kann damit immer sofort als ACTIVATED markiert und die Reinitialisierung durchgeführt werden.

Handelt es sich hingegen um einen Vorsprung oder wurde die externe Transition nachträglich eingefügt, wird der Zielzustand durch Markierungsregel MR1 (siehe Seite 69) bzw. ihre Erweiterung durch Markierungsregel MR7 (siehe Seite 90) nicht als ACTIVATED markiert. Stattdessen wird mit der Aktivierung entsprechend der operationalen Semantik aus Kapitel 4 gewartet, bis alle eingehenden externen Transitionen gefeuert haben oder abgewählt wurden. Die Reinitialisierung kann allerdings auch erst zu diesem Zeitpunkt geschehen. Dies ist für den Benutzer nicht nachvollziehbar, zumal die Durchführung der Operation Inkonsistenzen verursachen kann, welche sofort behandelt werden müssen. Daher wird durch Markierungsoperation JUMP die Markierung des Zielzustands reinitialisiert und der Zielzustand des Sprungs als NOTACTIVATED markiert.[8] Sie erlaubt damit nicht nur die Reinitialisierung der Markierungen darauffolgender Transitionen und Zustände, sondern auch die sofortige Kennzeichnung von Inkonsistenzen (vgl. Abschnitt 7.2.3).

Nach erfolgtem Sprung müssen temporär eingefügte Sprünge wieder entfernt werden, um den modellierten Ablauf nicht dauerhaft zu verändern. Hier stellt sich die Frage, zu welchem Zeitpunkt dies geschehen soll. Im Sinne der operationalen Semantik ist hierfür der frühest mögliche Zeitpunkt die *Aktivierung des Zielzustands*, der spätest mögliche Zeitpunkt die *(erneute) Aktivierung des Quellzustands*. Aus Sicht des Benutzers ist ein sofortiges Entfernen nach Aktivierung des Zielzustands eines Rücksprungs sinnvoll, denn es soll nicht suggeriert werden, dass der außerplanmäßige Rücksprung noch einmal durchlaufen wird. Beim Vorsprung hingegen ist aus den aktuellen Markierungen schwer ersichtlich, dass ein Sprung vorgenommen wurde. Ein „spätes Entfernen“ der Sprungtransitionen ist hier wünschenswert. In der Praxis ist durchaus vorstellbar, temporäre Transitionen und Zustände nach Durchführung des Sprungs nur logisch von der operationalen Semantik abzukoppeln und in der Oberfläche dauerhaft anzuzeigen. Aus Gründen der Übersichtlichkeit gehen wir in dieser Arbeit exemplarisch vom sofortigen Entfernen eines temporären Sprungs zum frühestmöglichen Zeitpunkt, also nach Aktivierung des Zielzustands, aus (vgl. Markierungsoperation MO1). Dies kann im Kontext der Anwendung von Markierungsregeln MR1 bzw. MR7 geschehen (vgl. Abbildung 7.5c). Bei der technischen Realisierung kann dann individuell auf die Wünsche der Benutzer eingegangen werden.

Dynamische Eigenschaften der Sprungoperation innerhalb eines OLC

Der Zeitpunkt der Entfernung temporärer Transitionen muss nicht nur aus Sicht des Benutzers diskutiert werden, sondern auch hinsichtlich der dynamischen Eigenschaften. Wird die temporäre Sprungtransition nicht rechtzeitig vor Aktivierung ihres Quellzustands entfernt, würde sie aufgrund des Stellvertreterprozesses automatisch erneut feuern – wir würden uns in einer Endlosschleife (*Livelock*) befinden. Die rechtzeitige Entfernung muss also sichergestellt werden. Wir führen daher, wie oben erwähnt, die frühe Entfernung temporärer Sprünge nach Aktivierung ihres Zielzustands durch. Damit können die dynamischen Eigenschaften auf OLC-Ebene weiter zugesichert werden (vgl. Abschnitt 4.4). Die strukturellen Eigenschaften auf Ebene der Prozessstrukturen bleiben ebenfalls erhalten, da interne Zyklen, die beim Rücksprung entstehen, keine korrektheitsgefährdenden Zyklen auf Ebene der Prozessstruktur verursachen können (vgl. Abschnitt 4.7). Darüber hinaus können per Definition keine externen Transitionen in temporäre Sprungpfade hinein oder aus ihnen heraus modelliert werden. Externe Transitionen werden

[8] Alternativ könnte dies auch durch eine Erweiterung von Markierungsregel MR7 geschehen.

schließlich auf Modellebene definiert und können damit nicht mit dem (dynamisch eingefügten) Sprungzustand verknüpft werden. Die Bildung korrektheitsgefährdender Zyklen kann damit ausgeschlossen werden.

Betrachten wir weiter strukturelle und dynamische Korrektheit im Rahmen der JUMP Operation, verstoßen wir mit dem Einfügen einer temporären Transition bewusst gegen eine strukturelle Eigenschaft von OLCs. Um Nebenläufigkeit in OLCs zu verhindern, müssen theoretisch alle aus einem Zustand ausgehenden Transitionen mit demselben Prozess P verknüpft sein (vgl. Abschnitt 4.2.3). Bei Durchführung der Sprungoperation JUMP verknüpfen wir hingegen die aus dem Quellzustand ausgehende Sprungtransition mit dem Stellvertreterprozess τ. Wir stellen allerdings in jedem Fall sicher, dass keine Nebenläufigkeit entsteht, indem wir die temporäre Transition sofort als PROCESSING markieren. Da sie mit τ verknüpft ist, feuert sie sofort, während alle anderen aus dem Quellzustand ausgehenden Transitionen abgewählt werden. Damit kann es nicht zu Nebenläufigkeit in einem OLC kommen.

7.2.2 Rücksetzoperation für externe Transitionen

Die fachliche Ausnahmebehandlung kann auch externe Transitionen bzw. die mit ihnen verknüpften Synchronisationsprozesse betreffen. So wird zum Beispiel die erneute Ausführung eines Synchronisationsprozesses notwendig, wenn dessen vorherige Ausführung nicht korrekt durchgeführt wurde (vgl. Szenario 7.3). Ferner kann die Anpassung der Markierung externer Transitionen gewünscht sein, wenn sich deren Quellzustand in einer Schleife befindet. Entsprechend der Ausführungsregel AR5 wird die Markierung einer externen Transition nicht verändert, wenn ihr Quellzustand eine neue Markierung enthält, die externe Transition aber bereits als PROCESSING, FIRED oder DISABLED markiert wurde (vgl. Abschnitt 4.7). Soll nach einem Rücksprung und erfolgter Reinitialisierung der Markierung des Quellzustands einer externen Transition diese ebenfalls erneut ausgeführt werden, ist das Rücksetzen ihrer Markierung erforderlich.[9]

Technische Umsetzung

Die technische Realisierung der entsprechenden Operation ist weniger komplex als die Durchführung eines Sprungs, denn es muss schließlich nur die Markierung einer einzelnen externen Transition angepasst werden. In COREPRO kann dies technisch umgesetzt werden, indem die externe Transition in ihre Initialmarkierung zurückversetzt und anschließend erneut bewertet wird. Dadurch kann der mit ihr verknüpfte Prozess erneut instanziiert und ausgeführt werden (vgl. Abschnitt 4.3.2).

Wir definieren mit Markierungsoperation MO2 die *Rücksetzoperation* RESET. Sie gestattet es, die Markierung externer Transitionen in WAITING zu verändern. Abbildung 7.6a zeigt eine Prozessstruktur mit zwei OLCs. OLC A weist einen vormodellierten Rücksprung auf, der bereits durchlaufen wurde (d.h. die externe Transition (A.A2, AB, B.B2) wurde bereits als PROCESSING

[9] In diesem Abschnitt betrachten wir die Anpassung der Laufzeitmarkierungen für externe Transitionen bzw. Synchronisationsprozesse isoliert von der Sprungoperation für OLCs. Das Zusammenwirken der Markierungsoperationen wird in Abschnitt 7.3 diskutiert.

bzw. FIRED markiert). Wird die Rücksetzoperation RESET auf eine als FIRED markierte externe Transition angewendet, wird diese als WAITING markiert. Damit einhergehend wird der mit ihr verknüpfte Prozess AB automatisch abgebrochen (vgl. Abschnitt 4.3.2). Abbildung 7.6b zeigt das Szenario nach Durchführung der Operation. Im Abschnitt 7.2.3 diskutieren wir die Korrektheit der neuen Markierung der externen Transition und ihre Auswirkungen auf die dynamische Korrektheit der Prozessstruktur.

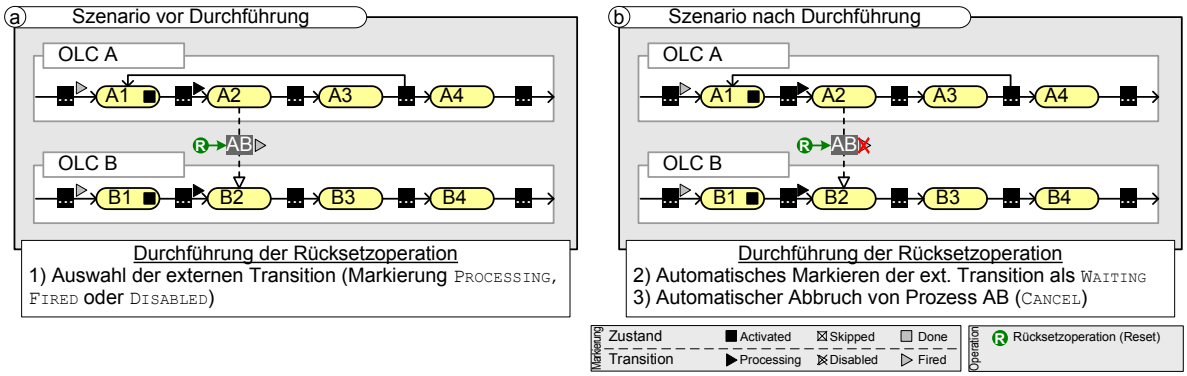


Abbildung 7.6: Rücksetzen einer externen Transition durch Markierungsoperation RESET

Markierungsoperation MO2 (Rücksetzoperation RESET)
Sei $ps = (OLC, EST)$ eine Prozessstruktur und $e^* \in EST$ eine externe Transition mit der Markierung $EM(e^*) \neq$ WAITING. Die aktuelle Markierung von ps sei $PSM = (OM, EM)$. Wird die Rücksetzoperation RESET (e^*) angewendet, ergibt sich folgende Markierung $PSM' = (OM', EM')$ für ps:

$$EM'(e) = \begin{cases} \texttt{WAITING} & \text{falls } e = e^* \\ EM(e) & \text{sonst} \end{cases}$$

$$OM' \equiv OM$$

7.2.3 Konsistenzkriterien für Markierungsoperationen

Die Veränderung der Laufzeitmarkierungen durch die eingeführten Markierungsoperationen JUMP und RESET kann die dynamische Korrektheit der Prozessstruktur beeinflussen. In Abschnitt 7.1.1 haben wir bereits Szenarien vorgestellt, in denen durch einen modellierten Sprung oder eine dynamische Änderung eine (syntaktische oder semantische) Inkonsistenz entstehen kann. Wir betten daher die Anwendung von Markierungsoperationen, ebenso wie die Änderungsoperationen aus Kapitel 6, in Änderungstransaktionen ein. So können wir vor Übernahme der Änderungen in die ausgeführte Prozessstruktur prüfen, ob die veränderte Prozessstruktur korrekt beendet werden kann.

Die im Kontext der dynamischen Adaption (vgl. Abschnitt 6.3.2) eingeführten Mechanismen für die Erkennung von Inkonsistenzen bleiben hier weiter gültig. Insbesondere erfolgen die Anwendung von Markierungsoperationen, die nachfolgende Konsistenzanalyse sowie die Ausnahmebehandlung entsprechend des bekannten Vorgehens aus Abbildung 6.11.

Konsistenzprüfung für Sprungoperation JUMP

Wir betrachten zunächst Inkonsistenzen, die bei Durchführung der Sprungoperation JUMP auftreten können. Hierbei handelt es sich um dieselbe Situation wie bei einem vormodellierten Vor- oder Rücksprung. Der Sprung innerhalb unsynchronisierten OLCs kann grundsätzlich keine Inkonsistenz verursachen. Wird hingegen innerhalb synchronisierter OLCs ein Sprung durchgeführt, können Inkonsistenzen entstehen. Ebenso wie bei der Analyse der Änderungsregionen bei dynamischen Änderungen können, basierend auf den Markierungen der externen Transitionen sowie ihren Quell- und Zielzuständen, genaue und vollständige Aussagen hinsichtlich ihrer Korrektheit getroffen werden.

Wird ein Vorsprung durchgeführt, so werden die externen Transitionen entsprechend der Ausführungsregel AR6 abgewählt. Hierbei können keine Inkonsistenzen entstehen (Abbildung 7.7a), denn es findet wie bei der modellierten Verzweigung eine Abwahl der ein- und ausgehenden externen Transitionen statt.

Bei einem Rücksprung kann es hingegen zu einer wiederholten Markierung des Quell- bzw. Zielzustands externer Transitionen kommen. Definition 4.19 beschreibt, wie derartige externe Transitionen zur Modellierzeit erkannt werden können. Die reine Erkennung reicht beim ungeplanten, fachlich begründeten Sprung allerdings nicht aus. Während der Modellierer geplanter Rücksprünge innerhalb der OLCs eine genaue Kenntnis der Prozessstruktur hat, ist dies beim Endbenutzer, der letztendlich die Sprungoperation anwendet, in der Regel nicht der Fall. Wir kennzeichnen daher vom Rücksprung betroffene externe Transitionen explizit als INCONSISTENT (vgl. Abbildung 7.7b), sodass eine dokumentierte Ausnahmebehandlung erfolgen muss (siehe Abschnitt 7.3). Konsistenzregel KR2 setzt die Kennzeichnung inkonsistenter externer Transitionen technisch um. Wird die Markierung eines Zustands durch Anwendung der Markierungsoperation JUMP von ACTIVATED oder DONE auf NOTACTIVATED oder ACTIVATED zurückgesetzt (durch Markierungsregel MR2 oder MR4) und hat der Zustand ausgehende externe Transitionen, die als PROCESSING oder FIRED markiert sind, entsteht im Prinzip eine Verletzung der Ausführungsregel AR5 und damit eine syntaktische Inkonsistenz (vgl. aus Zustand A3 ausgehende externe Transitionen in Abbildung 7.7b).[10]

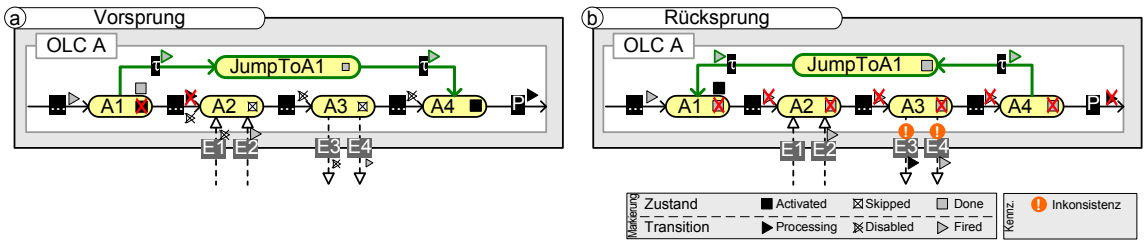


Abbildung 7.7: Kennzeichnung inkonsistenter externer Transitionen nach Sprung

[10] Die Kennzeichnung externer Transitionen im Rahmen des Rücksprungs als INCONSISTENT ist technisch nicht zwingend notwendig. Ist sie fachlich nicht erwünscht, kann auf sie verzichtet werden. Dann entspricht das Verhalten dem in Abschnitt 4.7 beschriebenen Verhalten, d.h. die Markierung externer Transitionen wird auch bei erneuter Aktivierung des jeweiligen Quellzustands nicht erneut bewertet. Die dynamische Korrektheit ist dadurch nicht gefährdet, denn im Prinzip entspricht die operationale Semantik der vormodellierter Rücksprungtransitionen, welche auch im „normalen Ablauf“ nicht korrektheitsgefährdend sind (vgl. Abschnitt 4.7).

In einen Zustand eingehende externe Transitionen mit Markierung ACTIVATED oder DONE sind von einem Rücksprung nicht betroffen, da deren Markierung nicht von der Markierung des Zielzustands abhängt. Die erzeugte Laufzeitmarkierung könnte auch im „normalen Ablauf“ auftreten und verstößt somit gegen keine Regel der operationalen Semantik.

Kennzeichnungsregel KR2 (Inkonsistenz Sprung)
Sei $ps = (OLC, EST)$ eine Prozessstruktur mit aktueller Markierung $PSM = (OM, EM)$. Des Weiteren sei $e^* = (s_1, p, s_2) \in EST$ eine externe Transition. Dann gilt:

(a) Wenn e^* die aktuelle Markierung $EM(e^*) \in \{\text{PROCESSING}, \text{FIRED}\}$ besitzt und die Markierung von Zustand s_1 von $SM(s_1) \in \{\text{ACTIVATED}, \text{DONE}\}$ infolge der Anwendung der Operation JUMP in $SM'(s_1) \in \{\text{NOTACTIVATED}, \text{ACTIVATED}\}$ wechselt (Rücksprung; vgl. Markierungsoperation MO1), dann wird e^* wie folgt gekennzeichnet: $EF(e^*) :=$ INCONSISTENT.

(b) Wenn e^* die aktuelle Markierung $EM(e^*) =$ DISABLED besitzt und die Markierung von Zustand s_1 von $SM(s_1) =$ SKIPPED infolge der Anwendung der Operation JUMP in $SM'(s_1) \in \{\text{NOTACTIVATED}, \text{ACTIVATED}\}$ wechselt (Rücksprung; vgl. Markierungsoperation MO1 und Markierungsregel MR4) und $SM(s_2) \neq$ SKIPPED, wird e^* wie folgt gekennzeichnet: $EF(e^*) :=$ INCONSISTENT.

(c) Wenn e^* die aktuelle Markierung $EM(e^*) =$ DISABLED besitzt und die Markierung von Zustand s_2 von $SM(s_2) =$ SKIPPED infolge der Anwendung der Operation JUMP in $SM'(s_1) \in \{\text{NOTACTIVATED}, \text{ACTIVATED}\}$ wechselt (Rücksprung; vgl. Markierungsoperation MO1 und Markierungsregel MR4) und $SM(s_1) \neq$ SKIPPED, wird e^* wie folgt gekennzeichnet: $EF(e^*) :=$ INCONSISTENT.

In Konsistenzregel KR2 werden abgewählte externe Transitionen gesondert betrachtet. Deren Markierung ist nach Ausführungsregel AR6 auch von der Markierung des Zielzustands abhängig (sie werden als DISABLED markiert, sobald der Quell- oder der Zielzustand abgewählt wurde). Damit muss eine ein- oder ausgehende externe Transition als INCONSISTENT gekennzeichnet werden, wenn ihr Quellzustand zurückgesetzt und ihr Zielzustand nicht als SKIPPED markiert ist bzw. umgekehrt (vgl. Beispiel 7.1).

Beispiel 7.1 (Deadpath Eliminierung in Zusammenhang mit Sprüngen)
Abbildung 7.8a zeigt eine Prozessstruktur mit den beiden aus einem abgewählten Pfad ausgehenden externen Transitionen (`A.A1'`, `PE1`, `B.B1`) und (`A.A1'`, `PE2`, `B.B1'`). Wird nun in OLC `A` ein Rücksprung auf den Startzustand durchgeführt, wird die externe Transition (`A.A1'`, `PE1`, `B.B1`) als INCONSISTENT gekennzeichnet, da weder ihr Quell- noch ihr Zielzustand als SKIPPED markiert ist (vgl. Abbildung 7.8b).

Abbildung 7.8c zeigt eine ähnliche Prozessstruktur wie Abbildung 7.8a, allerdings sind hier die externen Transitionen entgegengesetzt gerichtet. Der Rücksprung auf den Startzustand von OLC `A` führt auch hier zur Kennzeichnung der externen Transition (`B.B1`, `PE1`, `A.A1'`) als INCONSISTENT (vgl. Abbildung 7.8d). Diese Situation hätte im „normalen Ablauf“ mit vormodelliertem Rücksprung nicht zu einer Inkonsistenz geführt, denn – wie oben diskutiert – der Modellierer der Prozessstruktur hat eine genaue Kenntnis der Prozessstruktur. Er wird beim vormodellierten Rücksprung durch die Vorhersage von Kontextänderungen in der Prozessstruktur explizit auf

dieses Szenario hingewiesen (vgl. Definition 4.19). Beim Endanwender, der letztendlich die Sprungoperation durchführt, ist das in der Regel nicht der Fall und die externe Transition wird daher als INCONSISTENT gekennzeichnet.

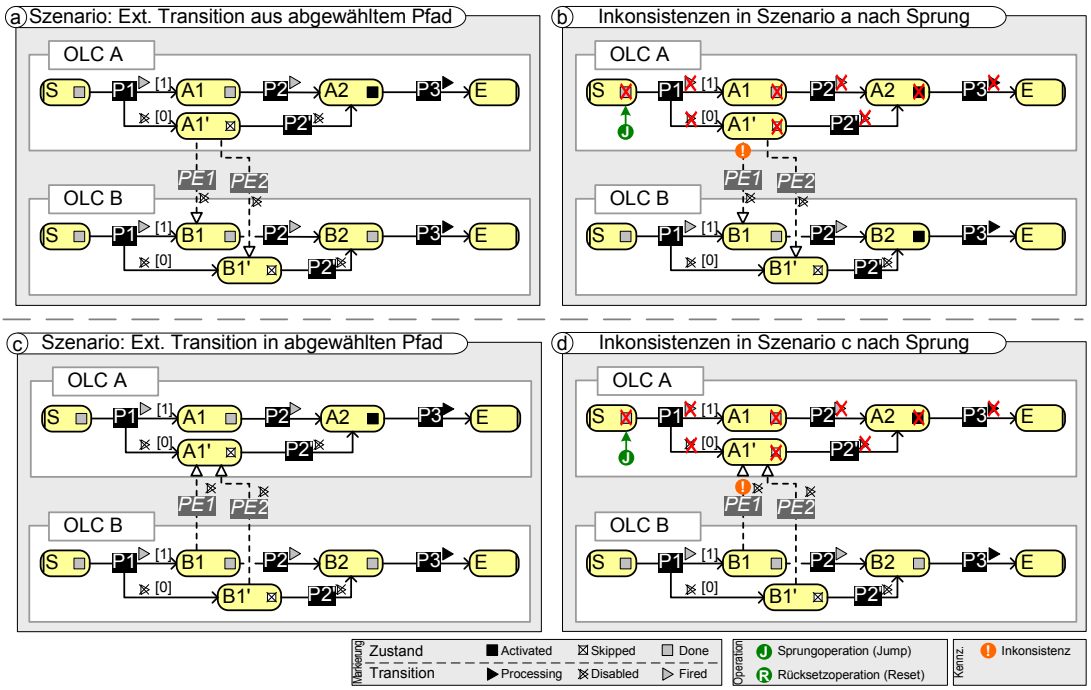


Abbildung 7.8: Kennzeichnung inkonsistenter abgewählter externer Transitionen

Konsistenzprüfung für Rücksetzoperation RESET

Die Anwendung der Rücksetzoperation RESET kann ebenfalls zu Inkonsistenzen führen. Abbildung 7.9a zeigt als Beispiel eine Prozessstruktur mit den beiden OLCs A und B und der gefeuerten externen Transition (A.A2, AB, B.B2). Wird die Rücksetzoperation auf die externe Transition angewendet, entstehen zwei syntaktische Inkonsistenzen (vgl. Abbildung 7.9b). Einerseits ist sie nun als WAITING markiert, was im weiteren Verlauf zu einer Verklemmung führen würde, da sich ihre Markierung nicht mehr ändert. Die Markierung der externen Transition ist damit inkonsistent. Andererseits ist ihr Zielzustand bereits als DONE markiert, was entsprechend der Regeln der operationalen Semantik eigentlich nicht möglich ist und ggf. falsche Aussagen zur Terminierung der Prozessstruktur nach sich zieht (vgl. Abschnitt 4.6.4). Die Markierung von B2 ist damit ebenfalls inkonsistent.

Das in Abbildung 7.9b dargestellte Beispiel ist mit den Szenarien für das dynamische Einfügen einer externen Transition vergleichbar. Die Markierungen der Änderungsregion entsprechen dem Szenario, das beim Einfügen einer externen Transition in Abbildung 6.9d entsteht und führen zu den gleichen Konsequenzen. Sowohl eingefügte als auch zurückgesetzte externe Transitionen werden zunächst als WAITING markiert. Bei der dynamischen Adaption identifiziert die anschließende

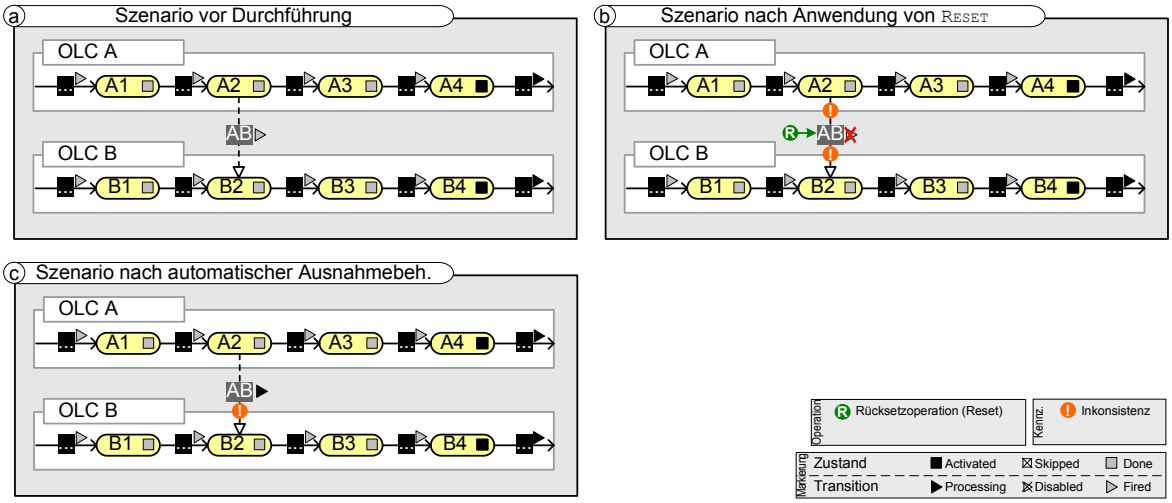


Abbildung 7.9: Inkonsistenz durch Rücksetzoperation im Zielzustand der externen Transition

Korrektheitsanalyse auf Basis Laufzeitmarkierungen der Änderungsregion entstandene Inkonsistenzen. Der Kontext von Kennzeichnungsregel KR1 (siehe Seite 152) wurde bereits so formuliert, dass die Konsistenzanalyse unabhängig von der durchgeführten Operation (vgl. Abschnitt 6.3) geschehen kann. So wird die Regel auch bei Durchführung der RESET-Operation angewendet.

Kommt es durch Anwendung der Markierungsoperation für externe Transitionen zu einer Inkonsistenz, findet wie bei der dynamischen Adaption eine automatische Ausnahmebehandlung statt. Sie führt, sofern möglich, eine Neubewertung der Markierung von externen Transitionen durch (vgl. Markierungsregel MR9). Im Szenario aus Abbildung 7.9b löst sie die Inkonsistenz der externen Transition automatisch auf, indem letztgenannte als PROCESSING markiert wird (vgl. Abbildung 7.9c). Ist auch nach Durchführung der automatischen Ausnahmebehandlung eine Inkonsistenz vorhanden, kann wie bei der dynamischen Adaption und der Anwendung der Sprungoperation die Änderungstransaktion nicht abgeschlossen werden. In den folgenden Abschnitten werden jedoch geeignete Verfahren vorgestellt, um derartige Inkonsistenzen auflösen zu können.

7.3 Kontrolle von Inkonsistenzen in der Prozessstruktur

Entstehen durch die dynamische Adaption oder die Anwendung der Markierungsoperationen Inkonsistenzen, wird der Abschluss der Änderungstransaktion verhindert. Dies ist in der Praxis allerdings häufig nicht akzeptabel. Zwar lassen sich Inkonsistenzen durch eine Adaption der Prozessstruktur immer auflösen; so kann die Inkonsistenz im Szenario aus Abbildung 7.2b durch Entfernen des OLC `Navigation` aufgelöst werden. Allerdings ist dieses Vorgehen im Allgemeinen nicht zielführend. Eine fachlich sinnvolle Reaktion im diskutierten Szenario ist hingegen, den Prozess `Labortest` im OLC `Navigation` mit der korrekten Komponente erneut durchzuführen. COREPRO unterstützt dies, indem durch einen Rücksprung im OLC `Navigation` auf den Zustand `Eingebaut` die Markierungen zurückgesetzt und der Prozess erneut ausgeführt wird.

Die eingeführten Markierungsoperationen (vgl. Abschnitt 7.2) können damit nicht nur Inkonsistenzen auslösen, sondern auch für die Behandlung von Inkonsistenzen genutzt werden. So kann beispielsweise die Inkonsistenz eines Zustands durch einen Rücksprung innerhalb des OLC aufgelöst werden. Die Inkonsistenz einer externen Transition kann durch Rücksetzen ihrer Markierung aufgelöst werden (vgl. Abbildung 7.10). In den folgenden Abschnitten beschreiben wir formale Regeln für das Auflösen von Inkonsistenzen und betrachten die damit verbundenen Konsequenzen in der Prozessstruktur.

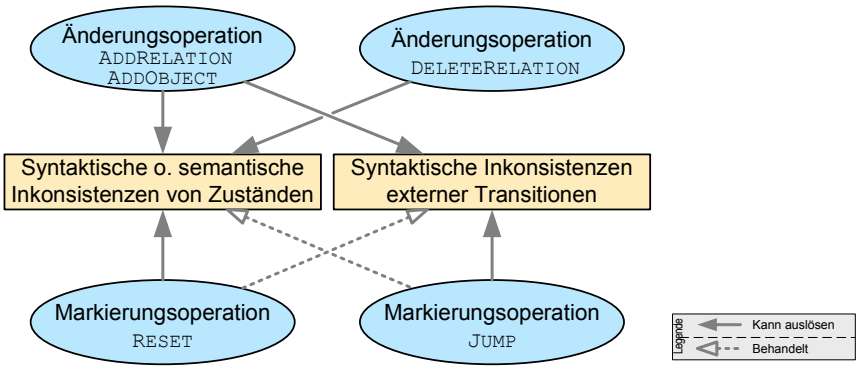


Abbildung 7.10: Zusammenhänge zwischen Inkonsistenzen und Markierungsoperationen

7.3.1 Konsistenzherstellung durch Anwendung von Markierungsoperationen

Beispiel 7.2 gibt einen Einblick in die Mechanismen zum Auflösen von Inkonsistenzen innerhalb der Prozessstruktur. Formal muss das Auflösen von Inkonsistenzen bzw. das Entfernen der INCONSISTENT-Kennzeichnungen mittels definierter Kennzeichnungsregeln geschehen. Die Inkonsistenz einer externen Transition oder eines Zustands zeigt an, dass sie bzw. er eine Markierung besitzt, die im normalen Ablauf nicht hätte entstehen können (vgl. Abschnitt 6.3.2). Durch eine entsprechende Änderung der Markierung können diese Inkonsistenzen aufgelöst werden (vgl. automatische Ausnahmebehandlung in Abschnitt 6.3.4). Durch die Sprung- und Rücksetzoperationen lassen sich die Markierungen inkonsistenter Zustände bzw. externer Transitionen wieder in die „korrekte“ Markierung versetzen. Bei Zuständen kann durch Anwendung der Sprungoperation eine Neubewertung der Markierung vorgenommen werden. Da die Sprungoperation die (inkonsistente) Markierung zurücksetzt und anschließend die bekannte operationale Semantik anwendet, wird durch sie die Inkonsistenz aufgelöst. Im Prinzip ist derselbe Mechanismus auch für externe Transitionen in Verbindung mit der Rücksetzoperation anwendbar. Durch Rücksetzen der Markierung in Kombination mit der automatischen Ausnahmebehandlung ist sichergestellt, dass die externe Transition eine konsistente Markierung erhält.

Die Kennzeichnungsregeln KR3 und KR4 stellen daher sicher, dass die INCONSISTENT-Kennzeichnungen wieder entfernt werden, wenn die Markierung des inkonsistenten Elements in die jeweilige Initialmarkierung zurückgesetzt wird (durch entsprechende Anwendung der Markierungsoperationen JUMP und RESET). Kennzeichnungsregel KR3 beschreibt dies für externe Transitionen. Ihre INCONSISTENT-Kennzeichnung wird wieder entfernt, falls sich ihre Markierung in WAITING ändert und dadurch die Inkonsistenz aufgelöst wird (vgl. Konsistenzegel KR1). Kennzeichnungsregel KR4 beschreibt denselben Sachverhalt für Zustände. Deren Kennzeichnung INCONSISTENT wird

wieder entfernt, wenn sich die Markierung in NOTACTIVATED ändert. Beispiel 7.2 beschreibt die Konsistenzherstellung in einer Prozessstruktur durch Anpassung der Laufzeitmarkierungen.

Beispiel 7.2 (Konsistenzherstellung durch Anpassung der Laufzeitmarkierungen) Abbildung 7.11a zeigt eine Prozessstruktur mit vier OLCs. In OLC A wurde ein Rücksprung durchgeführt, der aufgrund der Markierungen der externen Transitionen (A.A2, E1, B.B1), (A.A3, E2, C.C1) und (A.A3, E3, D.D1) Inkonsistenzen verursacht (vgl. Kennzeichnungsregel KR2).

Durch Anwendung der Operation RESET können jeweils die Inkonsistenzen der externen Transitionen aufgelöst werden (vgl. Abbildung 7.11b). Während die Anwendung der Operation bei der externen Transition (A.A3, E3, D.D1) ohne Folgen ist, entstehen in den Zuständen B.B1 und C.C1 weitere Inkonsistenzen (vgl. Abbildung 7.11b). Sie müssen ebenfalls aufgelöst werden, um die Konsistenz in der Prozessstruktur wieder vollständig herzustellen. Dies kann wiederum jeweils durch einen Rücksprung in OLC B und OLC C auf die inkonsistenten Zustände B.B1 bzw. C.C1 oder einen ihrer Vorgänger geschehen (vgl. Abbildung 7.11c).

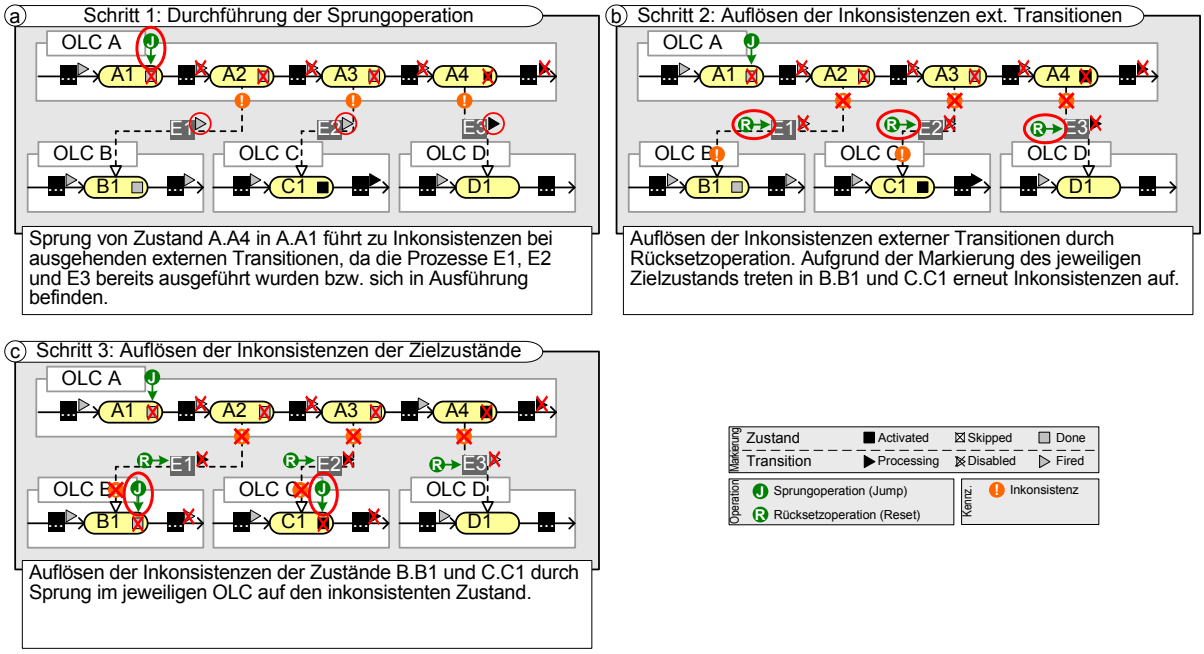


Abbildung 7.11: Auflösen von Inkonsistenzen durch Anwendung der Markierungsoperationen

Kennzeichnungsregel KR3 (Auflösen der Inkonsistenz externer Transitionen) *Sei* $ps = (OLC, EST)$ *eine Prozessstruktur. Des Weiteren sei* $e^* = (s_1, p, s_2) \in EST$ *eine externe Transition mit Kennzeichnung* $EF(e^*) \neq$ NONE. *Wenn sich ihre Markierung von* $EM(e^*) \in \{$PROCESSING, FIRED, SKIPPED$\}$ *in* $EM'(e^*) =$ WAITING *ändert, und es gilt* $SM(s_1) =$ NOTACTIVATED $\wedge\, SM(s_2) \neq$ SKIPPED *(vgl. Konsistenzegel KR1), folgt:*

$$EF(e^*) := \text{NONE}$$

Kennzeichnungsregel KR4 (Auflösen der Inkonsistenz von Zuständen)
Seien $ps = (OLC, EST)$ *eine Prozessstruktur und* $olc = (P, V, TS) \in OLC$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$*. Des Weiteren sei* $s^* \in S$ *ein Zustand mit Kennzeichnung* $SF(s^*) \neq$ `None`*. Wenn sich dessen Markierung von* $SM(s^*) \in \{$`Activated`, `Done`$\}$ *auf* $SM'(s^*) \in \{$`NotActivated`, `Activated`$\}$ *ändert, folgt:*

$$SF(s^*) = \texttt{None}$$

Durch Anwendung der Sprung- bzw. der Rücksetzoperation lassen sich die Markierungen von Zuständen und externen Transitionen zurücksetzen und eine Laufzeitmarkierung in der Prozessstruktur herstellen, die – logisch gesehen – vor Auftreten der Inkonsistenz im „normalen Ablauf“ erreicht werden kann. Damit lässt sich jede Inkonsistenz durch Anwendung geeigneter Markierungsoperationen wieder auflösen (vgl. Satz 7.1). Wir verzichten auf eine technische Beweisführung.

Satz 7.1 (Auflösen einer Inkonsistenz)
Eine Inkonsistenz innerhalb der Prozessstruktur lässt sich durch Anwendung der in Abschnitt 7.2 eingeführten Markierungsoperationen `Jump` *und* `Reset` *immer auflösen (vgl. Konsistenzregeln KR3 und KR4).*

7.3.2 Kennzeichnung erwarteter Inkonsistenzen

Bereits in einfachen Prozessstrukturen kann die Konsistenzherstellung durch Anwendung der Markierungsoperationen zu weiteren Inkonsistenzen führen (vgl. Beispiel 7.2). Die Ausnahmebehandlung kann durch diese „Fortpflanzung“ der Inkonsistenzen für den Benutzer schwer vorhersagbare Auswirkungen auf die ganze Prozessstruktur haben. Idealerweise erhält der Benutzer jedoch sofort bei Auftreten von Inkonsistenzen die Information, welche weiteren Inkonsistenzen im Rahmen der Konsistenzherstellung zu erwarten sind. So muss beispielsweise im E/E-Release-Management analysiert werden, ob der Austausch einer Komponente zu einer Inkonsistenz im OLC des Gesamtsystems und damit zu weitreichenden Konsequenzen führen würde.

CoRePro stellt dem Benutzer ein Werkzeug zur Verfügung, mit dem er auf einen Blick die in der Prozessstruktur *erwarteten Inkonsistenzen* sehen und über die Durchführung der initialen Markierungs- oder Änderungsoperation entscheiden kann, ohne schrittweise die Inkonsistenzen bearbeiten zu müssen. Es bleibt anzumerken, dass die Kennzeichnung erwarteter Inkonsistenzen keine funktionalen Auswirkungen hat, sondern dem Benutzer eine visuelle Entscheidungsgrundlage für die Ausnahmebehandlung dienen soll.

Die Erkennung erwarteter Inkonsistenzen kann auf Basis einer entstandenen Inkonsistenz und den Laufzeitmarkierungen davon betroffener Elemente erfolgen. Wird beispielsweise eine externe Transition als `Inconsistent` gekennzeichnet und wird die Operation `Reset` (vgl. Markierungsoperation MO2) auf die externe Transition angewendet, entsteht im Zielzustand eine Inkonsistenz, sofern dieser aktiviert ist oder bereits durchlaufen wurde (vgl. Kennzeichnungsregel KR1).

Diese Information können wir nutzen, um derartige Inkonsistenzen vorherzusagen ohne die RESET-Operation tatsächlich ausführen zu müssen. Kennzeichnungsregel KR5 kennzeichnet entsprechende Zustände als PREDICTEDINCONSISTENT. Wir erweitern dazu die Kennzeichnungsfunktion aus Definition 6.3 (siehe Seite 151) um die Kennzeichnung PREDICTEDINCONSISTENT.[11]

Die Kennzeichnung externer Transitionen erfolgt durch Kennzeichnungsregel KR6. Hierbei ist zu beachten, dass bei einem Sprung nur ausgehende externe Transition als INCONSISTENT gekennzeichnet werden, deren Quellzustand reinitialisiert wird (d.h. er liegt zwischen dem ursprünglich aktivierten Zustand und dem Zielzustand des Sprungs). Die Regeln KR5 und KR6 werden automatisch wechselseitig angewendet, sobald ein Zustand oder eine externe Transition als INCONSISTENT gekennzeichnet wurde. Beispiel 7.3 verdeutlicht deren Benutzung.

Beispiel 7.3 (Kennzeichnung erwarteter Inkonsistenzen)
Abbildung 7.12a zeigt eine Prozessstruktur mit fünf OLCs. Der Rücksprung in OLC A von Zustand A.A4 in den Zustand A.A1 führt zu inkonsistenten Markierungen der externen Transitionen (A.A2, AB, B.B2) und (A.A4, AC, C.C1). Die direkten Auswirkungen des Sprungs sind damit sichtbar. Unser Ziel ist es nun, alle externen Transitionen und Zustände, die im Rahmen der Konsistenzherstellung von diesen Inkonsistenzen betroffen sind, entsprechend zu kennzeichnen.

Regel KR5 ist durch die Kennzeichnung der erwähnten externen Transitionen als INCONSISTENT erfüllt und Zustand B.B2 wird als PREDICTEDINCONSISTENT gekennzeichnet. Damit ist der Kontext für Regel KR6 erfüllt. Regel KR6 kennzeichnet alle aus diesem OLC ausgehenden externen Transitionen als PREDICTEDINCONSISTENT, die als PROCESSING oder FIRED markiert sind und deren Quellzustände zwischen dem Zielzustand einer (erwartet) inkonsistenten externen Transition und dem aktuell aktivierten Zustand des OLC liegen (d.h. diejenigen externen Transitionen, deren Quellzustände durch die Reinitialisierung der Markierung beim Rücksprung betroffen sind). Im Beispiel aus Abbildung 7.12b wird die aus Zustand B.B3 ausgehende externe Transition (B.B3, BE, E.E1) als PREDICTEDINCONSISTENT gekennzeichnet, was wiederum zur Anwendung von Regel KR5 führt. Die Regeln werden so lange im Wechsel angewendet bis ihr Kontext nicht mehr erfüllt ist. Das Ergebnis ist die Kennzeichnung aller externen Transitionen und Zustände, deren Markierung im Rahmen der vollständigen Herstellung einer konsistenten Prozessstruktur eine Inkonsistenz auslösen wird (vgl. Abbildung 7.12b).

Kennzeichnungsregel KR5 (Erwartete Inkonsistenzen für Zustände)
Sei $ps = (OLC, EST)$ eine Prozessstruktur. Wird eine externe Transition $e = (s_1, p, s_2) \in EST$ als $EF(e) \in \{\text{INCONSISTENT}, \text{PREDICTEDINCONSISTENT}\}$ gekennzeichnet, wird ihr Zielzustand s_2 genau dann als $SF(s_2) :=$ PREDICTEDINCONSISTENT gekennzeichnet, wenn gilt:

$$SM(s_2) \in \{\text{ACTIVATED}, \text{DONE}\} \wedge SF(s_2) = \text{NONE}^a.$$

[a] Die Kennzeichnung als PREDICTEDINCONSISTENT soll keine bestehende Kennzeichnung (z.B. INCONSISTENT) überschreiben.

[11] Auf eine formale Erweiterung der Definition 6.3 wird hier verzichtet.

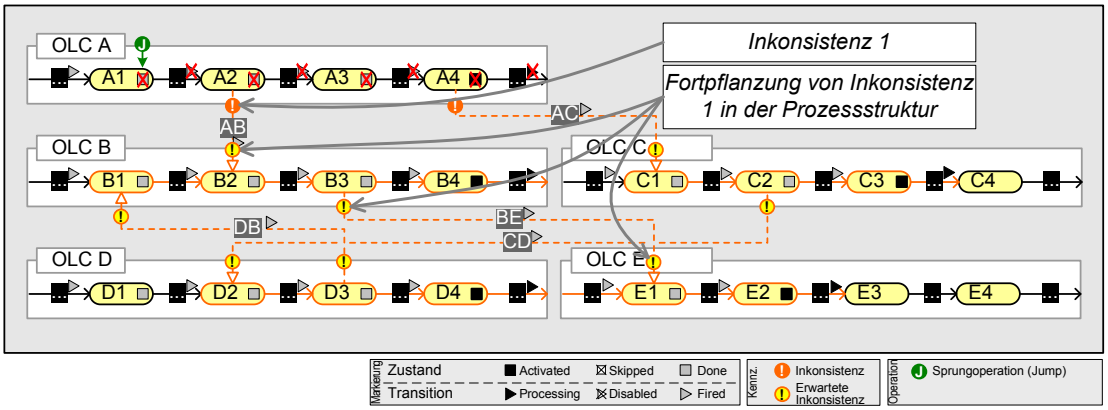


Abbildung 7.12: Kennzeichnung erwarteter Inkonsistenzen in der Prozessstruktur

Kennzeichnungsregel KR6 (Erwartete Inkonsistenzen für externe Transitionen) *Seien* $ps = (OLC, EST)$ *eine Prozessstruktur und* $olc = (P, V, TS) \in OLC$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$*. Der Zustand* $s_{act} \in S$ *sei der derzeit aktivierte Zustand von olc, d.h.* $SM(s_{act}) = \texttt{ACTIVATED}$*. Wird ein Zustand* $s \in S$ *von olc als* $SF(s) \in \{\texttt{INCONSISTENT}, \texttt{PREDICTEDINCONSISTENT}\}$ *gekennzeichnet, werden alle externen Transitionen* $e = (s_1, p, s_2) \in EST$ *mit* $EF(e) = \texttt{PREDICTEDINCONSISTENT}$ *gekennzeichnet, wenn für sie gilt:*

(a) $EM(e) \in \{\texttt{PROCESSING}, \texttt{FIRED}\}$ *und* $EF(e) = \texttt{NONE}$*. Weiter muss gelten: der Quellzustand* s_1 *liegt zwischen Zustand* s *und dem derzeit aktivierten Zustand* s_{act} *(d.h.* $s_1 \in states_{int}(s, s_{act})$*).*

(b) $EM(e) = \texttt{DISABLED}$ *und* $EF(e) = \texttt{NONE}$*. Weiter muss gelten:* s *liegt auf dem Pfad zwischen* s_{start} *und dem Quellzustand der externen Transition* s_1 *(d.h.* $s \in states_{int}(s_{start}, s_1)$*) und* $SM(s_2) \neq \texttt{SKIPPED}$*.*

(c) $EM(e) = \texttt{DISABLED}$ *und* $EF(e) = \texttt{NONE}$*. Weiter muss gelten:* s *liegt auf dem Pfad zwischen* s_{start} *und dem Zielzustand der externen Transition* s_2 *(d.h.* $s \in states_{int}(s_{start}, s_2)$*) und* $SM(s_1) \neq \texttt{SKIPPED}$*.*

Möchte der Benutzer im Rahmen der Konsistenzherstellung den Rücksprung aus fachlichen Gründen nicht in einen als INCONSISTENT gekennzeichneten Zustand, sondern in einen logisch *vor* ihm liegenden Zustand durchführen, können weitere Inkonsistenzen entstehen. Mit den Kennzeichnungsregeln KR5 und KR6 stellen wir sicher, dass die Vorhersage der Inkonsistenzen dynamisch angepasst wird.

Das Entfernen der PREDICTEDINCONSISTENT Kennzeichnungen erfolgt automatisch durch die bereits eingeführten Kennzeichnungsregeln. Im Rahmen der Konsistenzherstellung werden die Inkonsistenzen schrittweise mit den Operationen JUMP und RESET aufgelöst. Hierbei entstehen – der Reihe nach – die durch die PREDICTEDINCONSISTENT-Kennzeichnung vorhergesagten Inkonsistenzen, d.h. die betroffenen Elemente werden automatisch als INCONSISTENT gekennzeichnet (vgl. Kennzeichnungsregeln KR1 und KR2). Wird beispielsweise die Markierungsoperation RESET für die externe Transition mit Prozess AB aus Abbildung 7.12b durchgeführt, werden

die INCONSISTENT Kennzeichnung für die externe Transition entfernt (vgl. Kennzeichnungsregel KR3) und ihr Zielzustand B.B2 als INCONSISTENT gekennzeichnet (vgl. Kennzeichnungsregel KR1 auf Seite 152). Die PREDICTEDINCONSISTENT Kennzeichnung wird „überschrieben“, da die Vorhersage der Inkonsistenz nun eingetroffen ist (vgl. Kennzeichnungsregel KR2). Wird in OLC B ein Rücksprung in Zustand B2 ausgeführt um dessen Inkonsistenz aufzulösen, wird anschließend die aus Zustand B.B3 ausgehende externe Transition als INCONSISTENT gekennzeichnet und deren PREDICTEDINCONSISTENT Kennzeichnung überschrieben. Damit ist sichergestellt, dass nach vollständiger Konsistenzherstellung in der Prozessstruktur (d.h. alle Inkonsistenzen sind aufgelöst) auch kein Element mehr als PREDICTEDINCONSISTENT gekennzeichnet ist.

Durch die Kennzeichnung erwarteter Inkonsistenzen ist das System nun in der Lage, die Tragweite einer Inkonsistenz darzustellen und dem Benutzer die Möglichkeit zu geben, die Konsequenzen der auslösenden Änderung abzuschätzen.

7.3.3 Bestimmung der Operationen zur vollständigen Konsistenzherstellung

Auf Basis der erwarteten Inkonsistenzen lassen sich die notwendigen Operationen zur vollständigen Konsistenzherstellung in einer Prozessstruktur herleiten. Wird die PREDICTEDINCONSISTENT Kennzeichnung durch die Kennzeichnungsregeln KR5 und KR6 gesetzt, kann auch eine entsprechende Markierungsoperation zum Auflösen der Inkonsistenz benannt werden (vgl. Kennzeichnungsregeln KR3 und KR4). Inkonsistenzen externer Transitionen können durch die Rücksetzoperation RESET aufgelöst werden. Entsprechend muss für jede externe Transition mit Kennzeichnung PREDICTEDINCONSISTENT diese Operation vorgemerkt werden. Inkonsistenzen bzw. erwartete Inkonsistenzen der Markierung von Zuständen können mithilfe der Sprungoperation JUMP aufgelöst werden. Abbildung 7.13 zeigt die durchzuführenden Markierungsoperationen zur Herstellung der Konsistenz in der Prozessstruktur aus Abbildung 7.12b ausgehenden von den beiden aus OLC A ausgehenden inkonsistenten externen Transitionen.

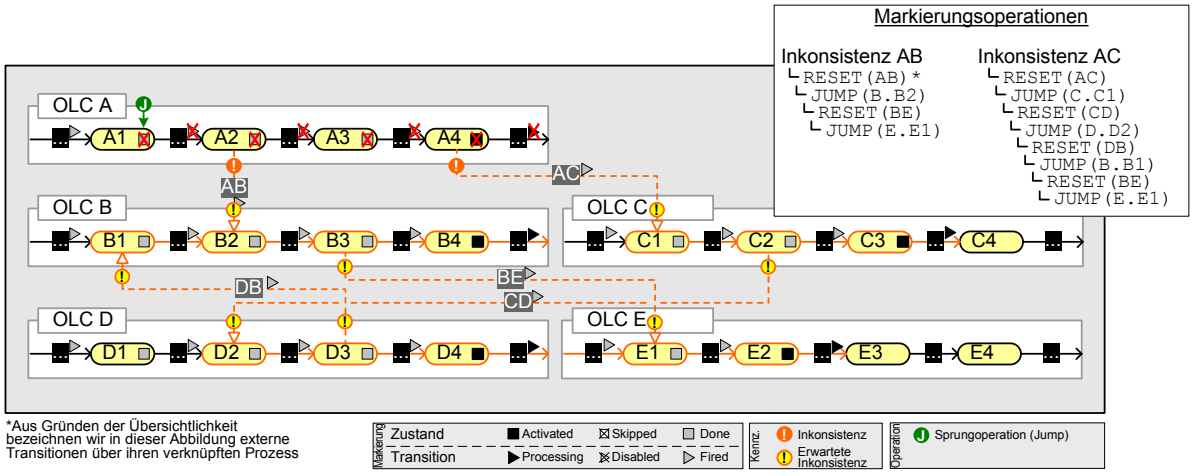


Abbildung 7.13: Auflösen der Inkonsistenzen in der Prozessstruktur aus Abbildung 7.12b

Die Reihenfolge der Durchführung der Operationen zur Konsistenzherstellung ist prinzipiell irrelevant. Es kann jedoch notwendig sein, dass Markierungsoperationen bei Nichtbeachtung der

Reihenfolge mehrfach ausgeführt werden müssen, da beim Rücksprung die Anwendung der Markierungsoperation JUMP den Zielzustand direkt als ACTIVATED markieren kann. Wird eine in diesen Zustand eingehende externe Transition zurückgesetzt, kann dadurch (erneut) in diesem Zustand eine Inkonsistenz entstehen, welche dann wiederum durch einen Sprung aufgelöst werden muss.

Basierend auf den durchzuführenden Operationen kann auch ein formaler *Konsistenzherstellungsprozess* abgeleitet werden (vgl. Abbildung 7.14a). Wir können hier erkennen, dass in OLC B aus Abbildung 7.13 zwei als PREDICTEDINCONSISTENT gekennzeichnete externe Transitionen eingehen und dort in ihren Zielzuständen B.B1 und B.B2 ebenfalls vorhergesagte Inkonsistenzen auslösen. Während die Auflösung der Inkonsistenz von Transition (A.A2, AB, B.B2) einen Rücksprung von Zustand B.B4 in Zustand B.B2 erfordert, ist für die die Transition (D.D3, DB, B.B1) ein Rücksprung in Zustand B.B1 vonnöten. Da der Sprung auf B.B1 auch die Inkonsistenz für B.B2 auflöst (vgl. Kennzeichnungsregel KR4), kann auf Durchführung des Sprungs auf B.B2 verzichtet werden. Dadurch kann eine Verschmelzung der Sprungoperationen und der nachfolgenden Operationen zur Konsistenzherstellung vorgenommen werden (vgl. Abbildung 7.14b).

Da die Prozessstruktur keine Zyklen mit externen Transitionen enthält (vgl. Abschnitt 4.7), besteht auch keine Verklemmungsgefahr für den Konsistenzherstellungsprozess. Die theoretischen Konzepte zur Erzeugung und Optimierung eines geeigneten Konsistenzherstellungsprozesses stehen jedoch nicht im Fokus dieser Arbeit und werden daher nicht weiter untersucht.[12]

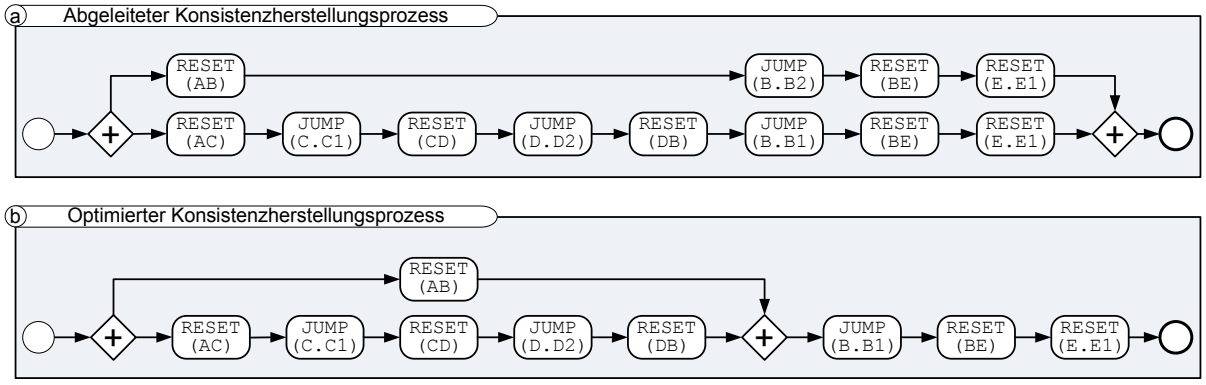


Abbildung 7.14: Aus Abbildung 7.13 abgeleiteter Konsistenzherstellungsprozess (in BPMN)

7.4 Ignorieren von Inkonsistenzen

Im vorangegangenen Abschnitt wurden Methoden für die Ausnahmebehandlung innerhalb einer Prozessstruktur erläutert und die Konsequenzen einer auftretenden Ausnahme näher beleuchtet. Die vollständige Herstellung der Konsistenz ist in der Theorie zwar grundsätzlich erwünscht, kann aber in der Praxis weitreichende Folgen haben, denn Inkonsistenzen können sich in der Prozessstruktur fortpflanzen. Damit ist die vollständige Herstellung der Konsistenz häufig nicht durchführbar (vgl. Szenario 7.4). Stattdessen werden Verfahren benötigt, mit denen die Ausführung einer Prozessstruktur trotz vorhandenen Inkonsistenzen fortgesetzt werden kann – und das

[12] Im Rahmen der prototypischen Implementierung wurden Mechanismen zur Erzeugung und nutzerfreundlichen Darstellung des Konsistenzherstellungsprozesses realisiert.

mit möglichst geringen Einschränkungen. Ziel ist es, nicht alle zur Konsistenzherstellung notwendigen Operationen durchführen zu müssen, sondern den Konsistenzherstellungsprozess (vgl. Abbildung 7.14) an einer beliebigen Stelle abzubrechen. Das bedeutet, dass eine Inkonsistenz gezielt *ignoriert* wird, um sie (und die daraufhin erwarteten Inkonsistenzen) nicht behandeln zu müssen. Fachlich kann das Ignorieren einer Inkonsistenz zwar zu einem „semantisch inkorrekten" Endergebnis, dies ist jedoch stark vom fachlichen Kontext abhängig. Es geht an dieser Stelle vielmehr darum, die „reale" Ausnahmebehandlung abzubilden und dem Benutzer die Freiheit zu geben, diese Inkonsistenz im Modell – wie in der Realität – zu ignorieren. Um dennoch eine spätere Ausnahmebehandlung zu erlauben, soll die verbleibende Inkonsistenz nicht einfach verworfen, sondern entsprechend gekennzeichnet werden.

Im Szenario aus Abbildung 7.13 kann zum Beispiel aus fachlicher Sicht die Anwendung der Markierungsoperation `RESET` zur Auflösung der Inkonsistenz der externen Transition (`A.A4`, `AC`, `C.C1`) nicht in Frage kommen. Systemseitig muss die Inkonsistenz dann ignoriert werden können. Das bedeutet, die (syntaktische) Inkonsistenz soll nicht behandelt und die Ausführung der Prozessstruktur dennoch fortgesetzt werden. Die Markierung der externen Transition soll also beibehalten werden, obwohl sie im normalen Ablauf nicht hätte erreicht werden können. Um ein vorhersagbares Verhalten zu erreichen, soll die Markierungsänderung der externen Transition von nun ab „abgekoppelt" verlaufen und darf von ihrem Quellzustand nicht mehr beeinflusst werden. Zum Beispiel soll die externe Transition nicht erneut als `PROCESSING` markiert werden, wenn ihr Quellzustand aktiviert wird. Wir müssen hierbei in jedem Fall sicherstellen, dass die dynamische Korrektheit auch trotz der bestehenden syntaktischen Inkonsistenz garantiert bleibt.

Dieselben Aspekte gelten auch für Zustände, die als `INCONSISTENT` gekennzeichnet sind. Ein Zustand wird als `INCONSISTENT` gekennzeichnet, wenn eine syntaktische Inkonsistenz zwischen der Markierung einer eingehenden externen Transition und der Markierung des Zustands besteht. Wird im Szenario aus Abbildung 7.13 die Inkonsistenz der externen Transition (`A.A4`, `AC`, `C.C1`) mittels `RESET`-Operation aufgelöst, entsteht eine Inkonsistenz in ihrem Zielzustand `C.C1`. Kommt ein Rücksprung in OLC `C` aus fachlichen Gründen nicht in Frage, soll die Inkonsistenz in `C.C1` ignoriert werden können. Zur Laufzeit sollen dann auch die Abhängigkeiten zwischen ihren Markierungen abgekoppelt werden. Zum Beispiel soll das Feuern der externen Transition (`A.A4`, `AC`, `C.C1`) keine erneute Aktivierung ihres Zielzustands `C.C1` auslösen.

Für den robusten Umgang mit ignorierten Inkonsistenzen müssen folgende Fragestellungen bearbeitet werden:

- Wie kann das betroffene Element gekennzeichnet werden, um die aktuelle Änderungstransaktion abzuschließen?
- Was geschieht mit den `PREDICTEDINCONSISTENT`-Kennzeichnungen von Zuständen und externen Transitionen, die auf das ignorierte Element folgen?
- Welchen Einfluss haben ignorierte Inkonsistenzen auf das dynamische Verhalten der Prozessstruktur (d.h. die weitere Ausführung der Prozessstruktur)?
- Wie gehen wir mit ignorierten Inkonsistenzen um, falls eine erneute Ausnahmesituation auftritt?

7.4.1 Operation zum Ignorieren einer Inkonsistenz

Wird eine Inkonsistenz ignoriert, muss dies formal dokumentiert werden. Dies ist nicht nur aus Gründen der Nachvollziehbarkeit notwendig (vgl. Abschnitt 2.2.1), sondern auch für die Ausführung, da die Inkonsistenz den Ablauf nicht gefährden darf. Die Kennzeichnung eines Elements als *ignoriert* kann mit dem bekannten Kennzeichnungsmechanismus erfolgen, indem die Menge der Kennzeichnungen aus Definition 6.3 um die Kennzeichnung `IGNORE` erweitert wird (d.h. die Menge *Flags* enthält dann die Kennzeichnungen `NONE`, `INCONSISTENT`, `PREDICTEDINCONSISTENT`, `IGNORE` und `TEMPORARY`[13]). Für die Kennzeichnung von inkonsistenten externen Transitionen und Zuständen führen wir die vom Benutzer ausführbare Operation `IGNOREINC` ein (vgl. Kennzeichnungsoperation KO1). Durch Ignorieren einer Inkonsistenz wird die Kennzeichnung des entsprechenden Elements von `INCONSISTENT` auf `IGNORE` geändert. Hieraus resultieren zwei Vorteile: Erstens zeigt die Kennzeichnung `IGNORE` weiterhin an, dass eine Inkonsistenz besteht (d.h. diese Information geht nicht verloren). Zweitens kann so die Änderungstransaktion abgeschlossen bzw. die Suspendierung der Prozessstruktur aufgehoben werden, denn die `INCONSISTENT`-Kennzeichnung wurde aufgehoben (vgl. Definition 6.5).

Kennzeichnungsoperation KO1 (Operation `IGNOREINC`)
Seien $ps = (OLC, EST)$ eine Prozessstruktur und S_{Targ} die Menge der Zustände aller OLCs mit $S_{Targ} := \{s^* | \exists\ olc = (P, V, TS) \in OLC \wedge TS = (S, T, s_{start}, s_{end}) \wedge s^* \in S\}$. Dann ist die Kennzeichnungsoperation `IGNOREINC` *wie folgt definiert:*

- *Wird* `IGNOREINC`*(e^*) auf eine externe Transition $e^* \in EST$ mit aktueller Kennzeichnung $EF(e^*) =$* `INCONSISTENT` *angewendet, wird diese auf $EF'(e^*) :=$* `IGNORE` *gesetzt.*
- *Wird* `IGNOREINC`*(s^*) auf einen Zustand $s^* \in S_{Targ}$ mit aktueller Kennzeichnung $SF(s^*) =$* `INCONSISTENT` *angewendet, wird diese auf $SF'(s^*) :=$* `IGNORE` *gesetzt.*

Im Gegensatz zu den in Abschnitt 7.2 behandelten Markierungsoperationen führt die `IGNOREINC`-Operation in keinem Fall zu weiteren Inkonsistenzen. Die Kennzeichnung der Inkonsistenz eines Elements als `IGNORE` führt vielmehr zum „Abbruch“ der Ausnahmebehandlungskette dieser Inkonsistenz. Allerdings müssen alle weiteren `PREDICTEDINCONSISTENT`-Kennzeichnungen entsprechend entfernt werden. Dies geschieht durch die Kennzeichnungsregeln KR7 und KR8. Sie werden, wie beim Setzen der `PREDICTEDINCONSISTENT`-Kennzeichnung, für deren Entfernung wechselseitig angewendet. Sobald keine Inkonsistenzen mehr existieren bzw. die verbleibenden Inkonsistenzen ignoriert werden, kann die Änderungstransaktion abgeschlossen, die Änderungen in die Prozessstruktur übernommen und die Ausführung fortgesetzt werden. Beispiel 7.4 illustriert die Anwendung der `IGNOREINC` Operation.

Beispiel 7.4 (Ignorieren einer Inkonsistenz)
Abbildung 7.15 zeigt die Prozessstruktur aus Abbildung 7.13. Um die Inkonsistenz der externen Transition $e =$ (`A.A4`, `AC`, `C.C1`) und die folgenden erwarteten Inkonsistenzen nicht behandeln zu müssen, wird die Operation `IGNOREINC` auf e angewendet. Die Ausnahmebehandlungskette bricht daraufhin für die ignorierte Inkonsistenz ab (vgl. Abbildung 7.15). Schließlich wird die erwartete

[13] Die Kennzeichnung `TEMPORARY` wird für die Operation `JUMP` benötigt (vgl. Abschnitt 7.2.1). Auf eine formale Erweiterung der Kennzeichnungsfunktion aus Definition 6.3 wird verzichtet.

Operation zur Behandlung der Inkonsistenz von e nicht durchgeführt, woraufhin die erwarteten Inkonsistenzen auch nicht eintreten. Die erwarteten Inkonsistenzen der Zustände `C.C1`, `D.D2` und `B.B1` sowie der externen Transitionen (`C.C2`, `CD`, `D.D2`) und (`D.D3`, `DB`, `B.B1`) werden daher durch Anwendung der Regeln KR7 und KR8 wechselseitig entfernt.

Die Inkonsistenz der externen Transition (`A.A2`, `AB`, `B.B2`) wird im Szenario aus Abbildung 7.15 nicht ignoriert. Sie muss behandelt werden, um die Änderungstransaktion abschließen zu können. Die aus dieser Inkonsistenz resultierenden erwarteten Inkonsistenzen sowie deren Fehlerbehandlungsprozess bleiben erhalten.

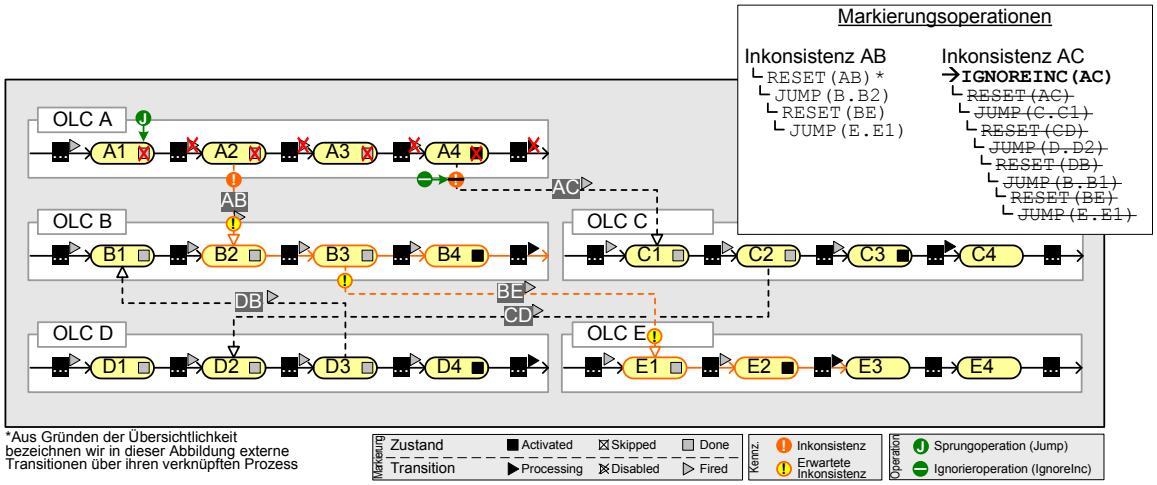


Abbildung 7.15: Ignorieren einer Inkonsistenz

Kennzeichnungsregel KR7 (Entfernen erwarteter Inkonsistenzen von Zuständen)
Seien $ps = (OLC, EST)$ *eine Prozessstruktur und* $S_{Targ} := \{s | \forall\, e = (s_1, p, s_2) \in EST : s = s_2\}$ *die Menge der Zielzustände aller externen Transitionen. Sei* $s^* \in S_{Targ}$ *ein Zustand mit Kennzeichnung* $SF(s^*) = \textsc{PredictedInconsistent}$.
Wechselt die Kennzeichnung einer in s^* *eingehenden externen Transition* $e \in inTrans_{ext}(s^*)$ *von* $EF(e) \in \{\textsc{Inconsistent}, \textsc{PredictedInconsistent}\}$ *in* $EF'(e) \in \{\textsc{Ignore}, \textsc{None}\}$ *und alle anderen in* s^* *eingehenden externen Transitionen* $\hat{e} \in inTrans_{ext}(s^*), \hat{e} \neq e$ *sind gekennzeichnet mit* $EF(\hat{e}) \in \{\textsc{Ignore}, \textsc{None}\}$, *dann wird die Kennzeichnung von* s^* *aufgehoben:* $SF'(s^*) := \textsc{None}$.

Kennzeichnungsregel KR8 (Entfernen erwarteter Inkonsistenzen ext. Transitionen)
Seien $ps = (OLC, EST)$ *eine Prozessstruktur und* $e = (s_1, p, s_2) \in EST$ *eine externe Transition mit Kennzeichnung* $EF(e) = \textsc{PredictedInconsistent}$. *Ferner seien* $olc = (P, V, TS) \in OLC$ *der Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$, *in dem sich der Quellzustand von* e *befindet (d.h.* $s_1 \in S$*), und* $S_P := path_{int}(s_{start}, s_1)$ *die Pfade vom Startzustand von* olc *zum Quellzustand von* e.
Wechselt die Kennzeichnung eines Zustands $s^* \in S_P$ *von* $SF(s^*) \in \{\textsc{Inconsistent}, \textsc{PredictedInconsistent}\}$ *in* $SF'(s^*) \in \{\textsc{Ignore}, \textsc{None}\}$ *und* $\nexists\, s' \in S_P : SF(s') \in \{\textsc{Inconsistent}, \textsc{PredictedInconsistent}\}$, *dann wird die Kennzeichnung von* e *aufgehoben:* $EF'(e) = \textsc{None}$.

7.4.2 Einfluss ignorierter Inkonsistenzen auf die dynamischen Eigenschaften einer Prozessstruktur

Wurden Inkonsistenzen ignoriert, um die Ausführung der Prozessstruktur fortzusetzen, muss dafür gesorgt werden, dass die Ausführung der Elemente mit IGNORE-Kennzeichnung abgekoppelt erfolgt. Das ist notwendig, damit ihre (inkonsistente) Markierung beibehalten werden kann und nicht durch das Element, mit dem die Inkonsistenz der Markierungen besteht, verändert wird. Die korrekte Ausführung der Prozessstruktur soll aber weiterhin gewährleistet bleiben. Wir betrachten im Folgenden den Einfluss ignorierter Inkonsistenzen externer Transitionen auf die dynamischen Eigenschaften der Prozessstruktur und anschließend den Einfluss ignorierter inkonsistenter Zustände.

Dynamische Eigenschaften ignorierter externer Transitionen

Die Inkonsistenz einer externen Transition zeigt an, dass ihre Markierung nicht den Regeln entspricht. Das Ignorieren der Inkonsistenz soll dafür sorgen, dass die Markierung trotzdem beibehalten werden kann und nicht zurückgesetzt wird (z.B. durch die automatische Anwendung einer Ausführungsregel). Ausführungsregel AR5, die eine externe Transition als PROCESSING markiert, sobald ihr Quellzustand aktiviert wird, darf dann nicht zur Anwendung kommen.

Für die technische Umsetzung einer solchen Abkopplung kommen prinzipiell zwei Realisierungsmöglichkeiten in Betracht. Eine Möglichkeit ist die *strukturelle Umsetzung*, bei der externe Transitionen „physisch“ von ihrem Quellzustand abgekoppelt werden (z.B. mittels Ersetzen des Quellzustands durch einen Stellvertreterzustand). Damit ist auch die operationale Abkopplung gewährleistet (vgl. Ausführungsregel AR5). Eine alternative Vorgehensweise ist die *operationale Umsetzung*, bei der ignorierte Inkonsistenzen durch Anpassung der operationalen Semantik abgekoppelt werden (z.B. wird für externe Transitionen mit Kennzeichnung IGNORE der Kontext der bekannten Ausführungs- und Markierungsregeln nicht mehr erfüllt). Grundsätzlich lassen sich beide Lösungsmöglichkeiten realisieren.

Die operationale Umsetzung zum Ignorieren von Inkonsistenzen bietet in COREPRO allerdings verschiedene Vorteile gegenüber der strukturellen Umsetzung. Die Ausführungs- und Markierungsregeln aus Kapitel 4 sind bereits sehr robust formuliert. Inkonsistenzen, wie sie durch Anwendung der Änderungs- und Markierungsoperationen auftreten, können durch minimale Änderungen an der bestehenden operationalen Semantik behandelt werden. Die Ausführung des Prozesses einer externen Transition durch Ausführungsregel AR5 etwa ändert die Markierung einer externen Transition nur dann, wenn die externe Transition die Markierung WAITING besitzt. Da eine externe Transition jedoch im Endeffekt nur dann als INCONSISTENT bzw. IGNORE gekennzeichnet wird, wenn sie nicht als WAITING markiert ist (vgl. Kennzeichnungsregel KR2; in anderen Fällen sorgt die automatische Ausnahmebehandlung für das Auflösen der Inkonsistenz), ist dieses Verhalten bereits implizit realisiert. Beispiel 7.5 verdeutlicht den Umgang mit ignorierten externen Transitionen zur Laufzeit.

Beispiel 7.5 (Operationale Semantik ignorierter externer Transitionen)
Im Beispiel aus Abbildung 7.16a wird in OLC A ein Rücksprung in Zustand A.A1 durchgeführt. Die externe Transition (A.A2, P, B.B2) wird als INCONSISTENT gekennzeichnet, da sie als FIRED

markiert ist und die Markierung ihres Quellzustands reinitialisiert wurde. Durch Anwendung der Operation IgnoreInc wird die externe Transition als Ignore gekennzeichnet und die Änderungstransaktion wird in Abbildung 7.16b ohne Behandlung der Inkonsistenz abgeschlossen.

Wird während der weiteren Ausführung die Markierung des Zustands A.A2 verändert (z.B. wenn er als Activated markiert wird), soll dies keinen Einfluss auf die Markierung der externen Transition haben (vgl. Abbildung 7.16c). Die entkoppelte bzw. nicht-synchronisierte Ausführung der externen Transition mit der ignorierten Inkonsistenz muss daher gewährleistet werden.

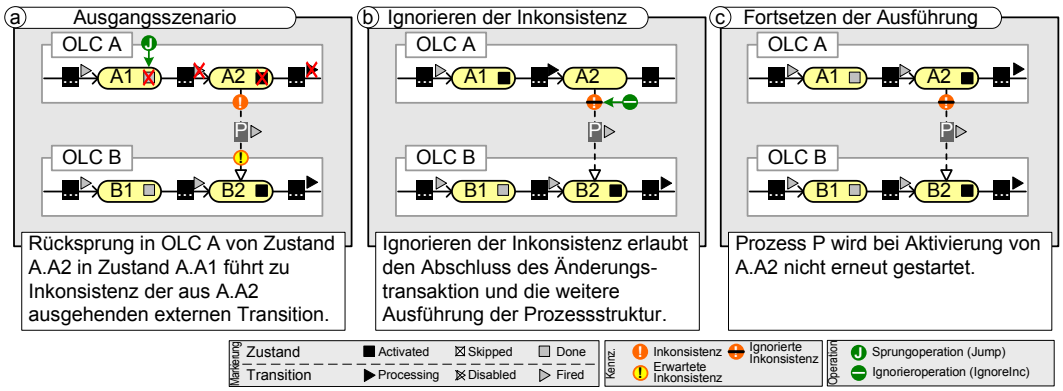


Abbildung 7.16: Ignorieren der Inkonsistenz einer externen Transition

Im Kontext der operationalen Umsetzung müssen wir auch die strukturelle und dynamische Korrektheit der Prozessstruktur nach Kennzeichnung einer externen Transition als Ignore sicherstellen. Die operationale Umsetzung beeinflusst die strukturelle Korrektheit nicht, da keine strukturelle Veränderung der Prozessstruktur durchgeführt wird. Da die externe Transition selbst nicht verklemmen kann (die automatische Ausnahmebehandlung verhindert ihre Verklemmung in Markierung Waiting), entsteht vielmehr durch die ignorierte Inkonsistenz eine Markierungskonstellation, die im normalen Ablauf auch im Rahmen eines modellierten Rücksprungs bereits behandelt wurde (vgl. Diskussion in Abschnitt 4.7). Das gewünschte Verhalten entspricht genau dem bereits realisierten Verhalten. Die operationale Semantik (z.B. Ausführungsregel AR5) muss somit nicht verändert werden. Da sich die Regeln für die Aktivierung des Zielzustands einer ignorierten externen Transition ebenfalls nicht ändern, bleiben auch die Aussagen zu dynamischen Eigenschaften, also Verklemmungsfreiheit, Terminierung und Aktivierbarkeit der Prozessstruktur im Kontext ignorierter externer Transitionen gültig.

Dynamische Eigenschaften ignorierter Zustände

Die Inkonsistenz eines Zustands zeigt an, dass seine Markierung im Zusammenhang mit der Markierung einer eingehenden externen Transition nicht entsprechend der Markierungsregeln MR6 und MR7 gesetzt wurde. Das Ignorieren der Inkonsistenz soll dafür sorgen, dass die Markierung des Zustands trotzdem beibehalten werden kann und nicht durch das Feuern einer eingehenden externen Transition verändert wird.

Die technische Realisierung kann, wie bei ignorierten externen Transitionen, entweder strukturell, d.h. durch „physische“ Abkopplung der eingehenden externen Transitionen, oder operational umgesetzt werden. Die operationale Umsetzung bietet auch im Zusammenhang mit Zuständen Vorteile, denn sie kann durch eine einfache Erweiterung der Markierungsregel MR6 geschehen. Hier kann direkt die Kennzeichnung des Zustands geprüft und seine Markierung infolge des Feuerns einer externen Transition nur dann auf `ACTIVATED` verändert werden, wenn er nicht als `IGNORE` gekennzeichnet ist.[14] Wir verzichten auf eine formale Erweiterung der Regel. Beispiel 7.6 illustriert die Auswirkungen der `IGNORE`-Kennzeichnung auf die Markierung von Zuständen während der Ausführung.

Beispiel 7.6 (Operationale Semantik ignorierter Zustände)
Abbildung 7.17a zeigt das Szenario aus Abbildung 7.16, wobei jedoch die Inkonsistenz der externen Transition nicht ignoriert, sondern durch Anwendung der Rücksetzoperation aufgelöst wird. Die Rücksetzoperation erzeugt, wie durch die erwartete Inkonsistenz in Abbildung 7.16a angezeigt, eine Inkonsistenz im Zielzustand `B.B2` der externen Transition.

Zum Auflösen der Inkonsistenz müsste eigentlich ein Rücksprung in OLC `B` durchgeführt werden. Im Beispiel aus Abbildung 7.17b wird die Inkonsistenz des Zustands `B.B2` jedoch ignoriert, um die Änderungstransaktion abschließen und mit der Ausführung der Prozessstruktur fortfahren zu können. Diese ignorierte Inkonsistenz führt zu einer entkoppelten Ausführung der eingehenden externen Transition. Feuert sie, wird die Markierung des Zielzustands nicht verändert (vgl. Abbildung 7.17c).

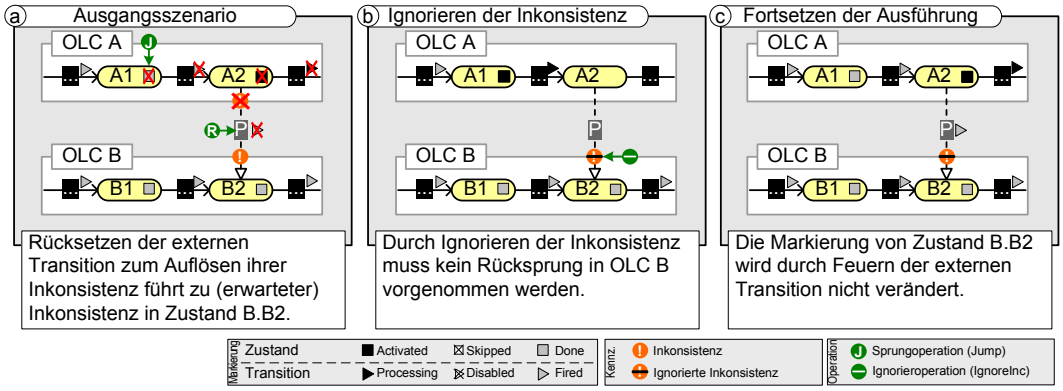


Abbildung 7.17: Ignorieren der Inkonsistenz eines Zustands

Während das Ignorieren der Inkonsistenz einer externen Transition keinen Einfluss auf die dynamische Korrektheit der Prozessstruktur hat (die automatische Neubewertung der Markierung zur Verhinderung von Verklemmungen vorausgesetzt; vgl. Kennzeichnungsregel MR9), verletzt das Ignorieren der Inkonsistenz eines Zustands das *Terminierungskriterium* (vgl. Abschnitt 4.7). Beim normalen Durchlauf „wartet“ der Zielzustand einer externen Transition, bis diese gefeuert hat. Dadurch ist bei Erreichen des Endzustands des OLC gewährleistet, dass alle mit internen

[14] Ein Zustand kann nur dann als `INCONSISTENT` bzw. als `IGNORE` gekennzeichnet werden kann, wenn er bereits aktiviert wurde (vgl. Kennzeichnungsregel KR1). Da diese Markierungskonstellation im „normalen Ablauf“ nicht entstehen kann, kann als Alternative zur direkten Abfrage der `IGNORE`-Kennzeichnung des Zielzustands in Markierungsregel MR6 auch geprüft werden, ob dieser derzeit als `ACTIVATED` oder `DONE` markiert ist.

oder externen Transitionen verknüpften Prozesse terminiert sind. Beim Ignorieren der Inkonsistenz eines Zustands wird diese Synchronisation bewusst entfernt. Damit kann es zu Situationen kommen, in denen der Ziel-OLC oder sogar alle OLCs der Prozessstruktur ihr Ende erreichen, wohingegen der Prozess der externen Transition noch ausgeführt wird. Dieser Ausnahmefall kann zum Beispiel durch eine Erweiterung von Definition 4.17, welche die Phasen der Prozessstruktur beschreibt, abgefangen werden (vgl. Definition 7.1). In der Praxis kann es allerdings auch Sinn machen, bei Erreichen der Endzustände aller OLCs nicht mehr auf externe Transitionen „zu warten“, sondern die mit ihnen verknüpften Prozesse abzubrechen. Schließlich haben diese externen Transitionen keinen Einfluss mehr auf die Markierung der Prozessstruktur. Die Realisierung dieser Anforderung kann ebenfalls durch eine Erweiterung von Definition 4.17 erfolgen: sobald alle OLCs ihren Endzustand erreicht haben, werden diejenigen externen Transitionen, die noch nicht als `DISABLED` oder `FIRED` markiert sind, als `DISABLED` markiert und damit ihr derzeit ausgeführter Prozess abgebrochen.

Definition 7.1 (Prozessstruktur-Phasen (erweitert))
Sei $ps = (OLC, EST)$ eine Prozessstruktur in der Phase PP(ps). Ändert sich der Status eines Object Life Cycle $olc^* \in OLC$ oder die Markierung einer externen Transition $e^* \in EST$, ergibt sich folgende Phasenregel für $PP'(ps)$:

$$PP'(ps) = \begin{cases} \texttt{INITIAL} & \textit{falls } \forall\, olc \in OLC : OP(olc) = \texttt{INITIAL} \\ \texttt{RUNNING} & \textit{falls } \exists\, olc \in OLC : OP(olc) = \texttt{RUNNING} \\ \texttt{TERMINATED} & \textit{falls } \forall\, olc \in OLC : OP(olc) = \texttt{ARCHIVED} \\ & \wedge\, \forall e \in EST : EM(e) \in \{\texttt{FIRED}, \texttt{DISABLED}\} \end{cases}$$

7.4.3 Auflösen ignorierter Inkonsistenzen

Das Ignorieren einer Inkonsistenz erlaubt die Fortsetzung der Ausführung einer Prozessstruktur und die Behandlung der Inkonsistenz zu einem späteren Zeitpunkt. So kann es beispielsweise fachlich Sinn machen, eine Inkonsistenz zunächst zu ignorieren und erst im Rahmen einer weiteren Ausnahmebehandlung aufzulösen. Grundsätzlich können ignorierte Inkonsistenzen mit denselben Mechanismen aufgelöst werden, wie „normale“ Inkonsistenzen. Hierfür kann jederzeit eine entsprechende Markierungsoperation angewendet werden (vgl. Abschnitt 7.3.1). Dann wird die `IGNORE`-Kennzeichnung, die stellvertretend für die `INCONSISTENT`-Kennzeichnung steht, entfernt (vgl. Kennzeichnungsregel KR3). So kann beispielsweise bei Ausführung der Rücksetzoperation `RESET` für eine externe Transition mit ignorierter Inkonsistenz, diese Inkonsistenz „nachträglich“ aufgelöst werden.

Nach Ignorieren einer Inkonsistenz können auch Situationen entstehen, in denen die ignorierte Inkonsistenz „automatisch“ im normalen Ablauf aufgelöst wird. Dieser Fall kann nur im Zusammenhang mit Zuständen auftreten und nur dann, wenn die Markierung des Zustands durch einen modellierten Rücksprung zurückgesetzt wird (vgl. Abschnitt 7.3.1). Abbildung 7.18 zeigt eine Prozessstruktur mit den OLCs `A` und `B`, wobei OLC `B` einen modellierten Rücksprung aufweist. Die in Zustand `B2` eingehende externe Transition verursachte eine Inkonsistenz in `B2`, die zunächst ignoriert wird. Sobald der modellierte Rücksprung in Zustand `B1` feuert und die Markierungen entsprechend Ausführungsregel AR4 und Markierungsregel MR4 reinitialisiert werden

(vgl. Abschnitt 4.3.6), wird die (ignorierte) Inkonsistenz durch Kennzeichnungsregel KR4 automatisch aufgelöst.

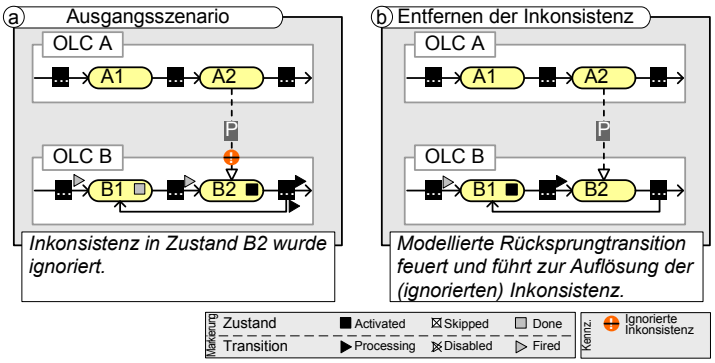


Abbildung 7.18: Herstellung der Konsistenz durch modellierten Rücksprung

Im Rahmen der normalen Ausnahmebehandlung müssen wir noch die Wechselwirkungen mit anderen Kennzeichnungen bzw. Kennzeichnungsregeln diskutieren. Es könnte beispielsweise geschehen, dass während der Ausnahmebehandlung eine bereits als `IGNORE` gekennzeichnete externe Transition (1) als `INCONSISTENT` oder (2) als `PREDICTEDINCONSISTENT` gekennzeichnet werden soll. Es bestehen jeweils zwei Möglichkeiten für den Umgang mit derartigen Situationen: entweder bleibt die `IGNORE`-Kennzeichnung erhalten oder sie wird ersetzt.

Die eingeführten Regeln zur Kennzeichnung von Inkonsistenzen bzw. erwarteten Inkonsistenzen beschreiben bereits für Fall 1 die erste Möglichkeit (d.h. Überschreiben der `IGNORE` Kennzeichnung). Tritt zum Beispiel im Szenario aus Abbildung 7.16b eine erneute Inkonsistenz der Markierung der als `IGNORE` gekennzeichneten externen Transition auf (z.B. durch einen Rücksprung in OLC `A`), wird sie entsprechend der in Abschnitt 6.3.3 vorgestellten Regel KR1 erneut als `INCONSISTENT` markiert. Die `IGNORE` Kennzeichnung wird dann entfernt und eine erneute Ausnahmebehandlung wird notwendig (ggf. mit erneuter Anwendung der `IGNOREINC` Operation).

Für Fall 2 realisieren die Kennzeichnungsregeln die zweite Möglichkeit (d.h. kein Überschreiben der `IGNORE` Kennzeichnung). Damit werden `IGNORE`-Kennzeichnungen durch `INCONSISTENT` überschrieben, während sie bei Anwendung der Regeln für die Kennzeichnung erwarteter Inkonsistenzen erhalten bleiben. Die Darstellung der erwarteten Inkonsistenzen endet so immer bei ignorierten Inkonsistenzen.

7.4.4 Ignorieren externer Transitionen

Das Szenario 7.5 (*Übergehen externer Transitionen*) wurde mit den bislang in Kapitel 7 eingeführten Methoden noch nicht behandelt. Es beschreibt die Anforderung, einen Zustand zu aktivieren, obwohl eine oder mehrere eingehende externe Transitionen noch nicht als `FIRED` oder `DISABLED` markiert sind. Der mit den externen Transitionen verknüpfte Synchronisationsprozess soll dabei nicht abgebrochen werden. Dies ist mit den bislang vorgestellten Operationen nicht erfüllbar.

Wir können jedoch vom eingeführten Mechanismus für das Ignorieren inkonsistenter Markierungen externer Transitionen profitieren. Die Diskussion der dynamischen Eigenschaften hat gezeigt,

dass sowohl dieser Mechanismus als auch die operationale Semantik robust genug sind, um mit diesen ignorierten Inkonsistenzen umzugehen und trotzdem die dynamische Korrektheit sicherzustellen. Wir können demzufolge das Ignorieren von externen Transitionen realisieren, indem wir folgende Anpassungen vornehmen:[15]

- Einführen der Kennzeichnung `IgnoreExtTrans` für externe Transitionen. Sie kann gesetzt werden, wenn die externe Transition als `Waiting` oder `Processing` markiert ist.
- Erweiterung der Markierungsregeln MR6, MR7 und MR8 zur Aktivierung eines Zustands. Dessen Aktivierung darf geschehen, sobald alle in ihn eingehenden externen Transitionen als `Fired` bzw. `Disabled` markiert oder als `IgnoreExtTrans` gekennzeichnet sind.
- Entfernen der Kennzeichnung `IgnoreExtTrans`, wenn der Zielzustand der externen Transition von `Activated` oder `Done` auf `NotActivated` markiert wird (z.B. im Rahmen eines Rücksprungs).

Beispiel 7.7 illustriert das Ignorieren einer externen Transition.

Beispiel 7.7 (Ignorieren einer externen Transition)
Abbildung 7.19a zeigt ein Anwendungsszenario, in dem der Zustand `A.A2` noch nicht aktiviert werden kann, da die externe Transition (`D.D1`, `E3`, `A.A2`) noch nicht gefeuert hat. Durch Ignorieren der externen Transition kann der Zustand `A.A2` aktiviert und die Ausführung in OLC `A` fortgesetzt werden, während der Synchronisationsprozess weiter ausgeführt wird (vgl. Abbildung 7.19b).

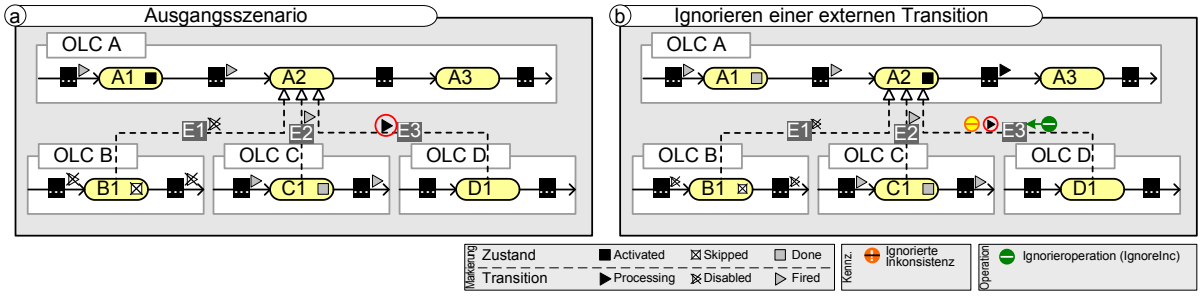


Abbildung 7.19: Übergehen einer externen Transition (exemplarisch)

7.5 Weitergehende Aspekte und Sonderfälle – ein Ausblick

In den vorangegangenen Abschnitten dieses Kapitels liegt der Schwerpunkt der Betrachtungen auf der Konzeption von Ausnahmebehandlungsmechanismen für Prozessstrukturen. Für eine praktische Umsetzung der Konzepte können noch weitere Aspekte relevant werden. Sie werden im Folgenden erörtert, und es wird jeweils kurz diskutiert, wie ihre Realisierung auf Grundlage der in diesem Kapitel beschriebenen Konzepte erfolgen könnte.

[15] Der Mechanismus wurde prototypisch im CoPro-Demonstrator (siehe Kapitel 8) realisiert. Auf die formale Beschreibung der Anpassungen wird an dieser Stelle verzichtet.

Kompensationsprozesse

In der Praxis kann es notwendig sein, bei Durchführung eines Rücksprungs mittels Markierungsoperation JUMP auch Prozesse zur fachlichen *Kompensation* der Effekte bereits ausgeführter Prozesse durchzuführen [LR00]. Der tatsächliche Nutzen hängt aber stark davon ab, ob die ausführenden Prozess-Management-Systeme derartige Kompensationsprozesse unterstützen und natürlich auch, inwieweit eine Kompensation aus fachlicher Sicht überhaupt erforderlich bzw. möglich ist.

In COREPRO kann diese Anforderung durch die Erweiterung der Markierungsoperation JUMP jedoch direkt realisiert werden. Dazu muss die aus dem temporären Zustand ausgehende Transition t_{jump2} statt mit dem Stellvertreterprozess τ mit einem Kompensationsprozess verknüpft werden (vgl. Abbildung 7.20a). Die Sprungoperation setzt die Markierung des Zielzustands des Sprungs bereits sofort zurück. Dadurch können Inkonsistenzen, die durch die Anwendung der Markierungsoperation hervorgerufen werden, trotz laufendem Kompensationsprozess festgestellt werden. Sollen Vor- und Rücksprünge im Rahmen der Kompensation unterschiedlich behandelt werden, kann diese Fallunterscheidung basierend auf einer simplen Prüfung der Markierung des Zielzustands des Sprungs getroffen werden.

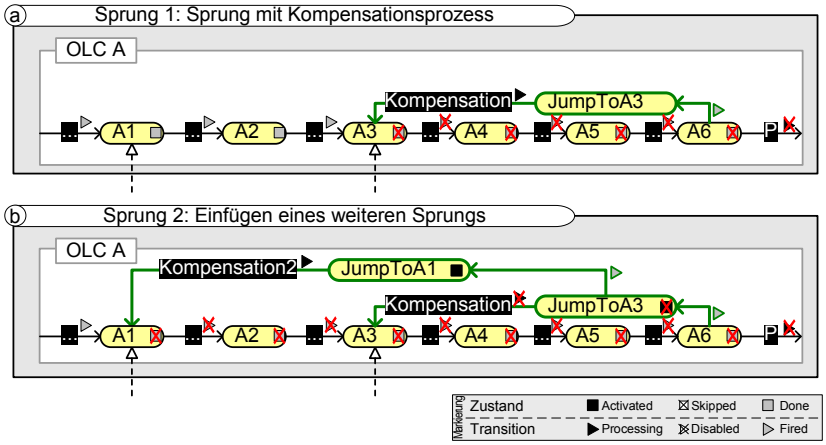


Abbildung 7.20: Durchführung eines Kompensationsprozesses

Mehrfacher Sprung

Wird der Zielzustand eines Sprungs aufgrund eingehender externer Transitionen oder durch Anwendung eines Kompensationsprozesses nicht sofort aktiviert, kann eine weitere Ausnahmebehandlung erforderlich werden. Dann kann es notwendig sein, den Sprung bzw. den Kompensationsprozess seinerseits abzubrechen und an eine andere Stelle im Prozess zu springen. Dies kann ebenfalls über die bekannte Sprungoperation JUMP realisiert werden (vgl. Abbildung 7.20b). Dazu muss in Markierungsoperation MO1 die Anwendung des Sprungs aus Zuständen erlaubt werden, die als TEMPORARY gekennzeichnet sind, und nach Durchführung des „letzten“ Sprungs muss darauf geachtet werden, dass die Sprünge vollständig entfernt werden.

Weitere Markierungsoperationen

Wir haben mit den Markierungsoperationen JUMP und RESET zwei mächtige Mechanismen zur flexiblen Ausführung von Prozessstrukturen vorgestellt. Es lassen sich allerdings weitere Szenarien konstruieren, die zusätzliche Markierungsoperationen erfordern. So kann der Bedarf entstehen, die Ausführung des Installationsprozesses im Szenario aus Abbildung 7.2a abbrechen zu können, wenn sich nur die Software, nicht aber die Hardware des zu installierenden Systems geändert hat. Eine entsprechende Operation kann äquivalent zur Operation RESET definiert werden, die eine externe Transition als DISABLED markiert. So wird der verknüpfte Synchronisationsprozess nicht ausgeführt und der Ziel-OLC kann dennoch fortfahren.

Eingeschränkter Sprung

In der Praxis kann es erforderlich sein, die „Distanz“ eines Sprungs zu beschränken. So gibt es in der Fahrzeugentwicklung auch Zustände, auf die aus physikalischen oder anderen Gründen nicht mehr zurückgesprungen werden kann (z.B. Rücksprung „vor“ einen Crashtest). Auf diese Eigenschaft könnte bereits in der Modellierung Bezug genommen werden, indem entweder, *physikalisch nicht wieder herstellbare Zustände* entsprechend gekennzeichnet werden oder aber geprüft wird, ob für einzelne Prozesse ein fachliches *Rollback* bzw. eine Kompensation möglich ist. Diese Zustände dürfen dann bei einem Rücksprung nicht *übersprungen* werden, da sie physikalisch nicht mehr herstellbar sind. Diese Eigenschaft hat jedoch keine direkten Auswirkungen auf unsere Konzepte. Sie schränkt lediglich den *Radius* der Markierungsänderungen ein und ist daher in die bestehenden Konzepte voll integrierbar.

Aufteilung eines Sprungs

In praktischen Szenarien könnte es notwendig sein, bei einem Vorsprung ein- oder ausgehende externe Transitionen, die zwischen Quell- und Zielzustand des Sprungs liegen, nicht abzuwählen, sondern trotzdem auszuführen. Dieses Verhalten kann mit der vorhandenen Sprungoperation abgebildet werden, indem der Sprung in einzelne Schritten aufgeteilt wird: Zunächst wird auf den Quellzustand derjenigen externen Transition, die nicht abgewählt, sondern ausgeführt werden soll, gesprungen. Der Zustand wird damit aktiviert und die externe Transition als PROCESSING markiert. Anschließend kann der Sprung auf den eigentlichen Zielzustand erfolgen. Für die Abbildung derartiger Szenarien kann auch eine höherwertige Sprungoperation definiert werden (vgl. [RPB03]).

Ein solches Vorgehen kann im Prinzip auch für interne Transitionen sinnvoll sein. Wird ein Vorsprung durchgeführt, kann es Sinn machen, bestimmte Prozesse, die mit internen Transitionen verknüpft sind, trotzdem auszuführen. Auch dieses Szenario kann durch eine Aufteilung des Sprungs bzw. die Definition einer höherwertigen Sprungoperation unterstützt werden.

Vergleichbare Anforderungen können sich auch beim Rücksprung ergeben, wenn Prozesse beim wiederholten Durchlauf nicht erneut ausgeführt werden sollen. Dazu kann entweder – den Ausführungsregeln für interne Transitionen vorgelagert – die Anwendung der Sprungoperation auf

den Zielzustand der mit dem Prozess verknüpften Transition erfolgen, oder der Prozess wird durch einen zu definierenden Mechanismus temporär durch den Stellvertreterprozess τ ersetzt.

Technischer Abbruch eines Prozesses

Die Ausführung eines einzelnen Prozesses kann aus verschiedenen Gründen fehlschlagen. So kann beispielsweise die Prozessausführung aus technischen Gründen scheitern, etwa wenn der Prozess gar nicht gestartet werden kann (z.B. keine Verbindung zum entsprechenden Prozess-Management-System) oder wenn er im ausführenden Prozess-Management-System vom Benutzer aufgrund eines Prozessfehlers abgebrochen wird (d.h. Beendigung mit Prozesszustand FAILED). Wir verzichten an dieser Stelle auf eine formale Beschreibung dieser Szenarien und skizzieren stattdessen die Behandlung dieser Ausnahme:[16] Da der Prozess nicht den Prozesszustand von RUNNING in COMPLETED, sondern in FAILED wechselt, gibt es keine Markierungsregel, deren Kontext erfüllbar ist, und die Ausführung des OLC blockiert zwangsläufig. Dies ist auch sinnvoll, denn in dieser Situation ist ein manueller Eingriff notwendig. Wird der Prozesszustand FAILED erkannt, kann auch die verknüpfte Transition entsprechend gekennzeichnet und – wie bei einer aufgetretenen Inkonsistenz – eine manuelle Ausnahmebehandlung eingefordert werden. Dem Nutzer stehen dann alle Möglichkeiten zur Verfügung, die er auch im Rahmen der „normalen“ Ausnahmebehandlung hat. So kann entweder die Ausführung des Prozesses wiederholt werden oder ein Vorwärtssprung auf den nachfolgenden Zustand gemacht werden.

Unterstützung bei der Bearbeitung vieler Inkonsistenzen

In großen Datenstrukturen kann ein einzelnes Objekt (z.B. System `Navigation`) sehr viele Relationen zu anderen Objekten (z.B. Subsystem `GPS-Sensor` und `TV Tuner`) haben. Für die entsprechende Prozessstruktur bedeutet dies in der Regel, dass der entsprechende OLC viele ein- oder ausgehende externe Transitionen besitzt. Kommt es zu einer Ausnahme, etwa infolge eines Rücksprungs innerhalb des OLC, ist ggf. für viele dieser externen Transitionen eine Ausnahmebehandlung vonnöten. Der hohe Aufwand für die Ausnahmebehandlung ist nicht immer in vollem Umfang gerechtfertigt, insbesondere wenn für alle externen Transitionen dieselbe Ausnahmebehandlung stattfinden soll (z.B. Rücksetzen der Markierung). In solchen Fällen kann der Aufwand für die Ausnahmebehandlung stark reduziert werden, indem für alle inkonsistenten externen Transitionen dieselbe Operation (z.B. IGNOREINC) durchgeführt wird. Dieser Aspekt hat jedoch keine direkten Auswirkungen auf unsere Konzepte, sondern lediglich auf die Durchführung der Änderungen und ist damit auf Grundlage der bestehenden Konzepte realisierbar.

7.6 Diskussion

Die Behandlung von Ausnahmen wird in der Literatur unter verschiedenen Gesichtspunkten betrachtet. Modellierungsansätze unterstützen häufig nur planbare Ausnahmen (vgl. Abbildung 7.1) und fordern hierfür die explizite Modellierung eines Ausnahmebehandlungsprozesses (z.B.

[16] Der Mechanismus zur Behandlung von Prozessen im Zustand FAILED wurde prototypisch realisiert (siehe hierzu Kapitel 8).

UML [Obj03b]). Die generische Ausnahmebehandlung ist hingegen eine große Herausforderung hinsichtlich der Konsistenz von Laufzeitmarkierungen. [WKDM+95] schreibt hierzu:

> „Da in solchen Fällen jedoch Ablaufeigenschaften des Workflows vom System nicht mehr garantiert werden können, sollten Ausnahmebehandlungen und dynamische Änderungen selten bleiben."

In datengetriebenen Prozessstrukturen kommen Änderungen jedoch häufig vor und es ist von größter Wichtigkeit, die Ablaufeigenschaften bzw. die Korrektheitskriterien **jederzeit** zu garantieren. Dieser Herausforderung stellen sich jedoch nur wenige Ansätze.

Grundsätzlich gibt es in der Literatur die Unterscheidung zwischen planbaren Ausnahmen (und damit modellierbarer Ausnahmebehandlung) und nicht-planbaren Ausnahmen, d.h. Ad-hoc-Ausnahmebehandlung [RPB03]. Planbare Ausnahmen werden im Allgemeinen direkt im Prozessschema vormodelliert und können damit auch durch viele Modellierungsansätze [RKG06, RKG07, KRG07, LBW07, NC03] behandelt werden. COREPRO deckt durch modellierbare Verzweigungen und Rücksprünge die Anforderungen datengetriebener Prozessstrukturen an die Unterstützung planbarer Ausnahmen ausreichend ab.

Die Behandlung nicht-planbarer Ausnahmen, die Ad-hoc-Abweichungen vom modellierten Ablauf in Form von Sprüngen benötigen, ist in der Literatur durch verschiedene Mechanismen gelöst. Obwohl der Case-Handling Ansatz bzw. das Tool FLOWer die dynamische Adaption von Prozessen nicht unterstützt (vgl. Abschnitt 6.4), erlaubt er mit den Operationen *Skip* und *Redo* einfache Abweichungen vom modellierten Prozess. Der Prozess muss hierbei immer in einen konsistenten Zustand gebracht werden; d.h. bei Wiederholung einer Aktivität als Reaktion auf die Anwendung einer *Undo* Operation müssen alle nachfolgenden Aktivitäten ebenfalls wiederholt werden [AWG05]. Das Ignorieren von Inkonsistenzen (vgl. Abschnitt 7.4.1) ist hier nicht gestattet.

Der ADEPT-Ansatz erlaubt durch Einführung komplexer Operationen für Rück- und Vorwärtssprünge die Ad-hoc-Abweichung vom geplanten Workflow [Rei00, RPB03]. Durch Definition von Vor- und Nachbedingungen für die entsprechende Operation ist „per Konstruktion" sichergestellt, dass eine Abweichung vom Kontrollfluss nur dann erlaubt ist, wenn keine Inkonsistenz entsteht. Damit ist jedoch auch in diesem Ansatz die flexible Reaktion auf Inkonsistenzen, wie beispielsweise das Ignorieren, nicht möglich.

Die *Exception Handling Patterns* aus [RAH06] beschreiben Ausnahmesituationen in Prozess-Management-Systemen mit Blick auf die einzelnen Aufgaben einer Arbeitsliste (vgl. Abschnitt 3.1.1). Ausnahmen sind in diesem Kontext beispielsweise eine fachliche Ausnahme (*External Trigger*), die in einem anderen unsynchronisierten Prozess entdeckt wird oder Inkonsistenzen (*Constraint Violation*), wie sie in COREPRO beispielsweise bei der dynamischen Adaption auftreten können. Der Ansatz beschreibt hierfür unterschiedliche Behandlungsmethoden. So sind bei einer fachlichen Ausnahme die Fortführung der Aufgabe (Ignorieren der Inkonsistenz) und der Abbruch der Aufgabe mit verschiedenen Optionen, die im Kontext einer Arbeitsliste relevant sind, möglich. Die Behandlung von Ausnahmen wird teilweise von Prozess-Management Systemen (z.B. *IBM WebSphere MQ*) und Standards unterstützt (z.B. BPMN und BPEL), indem über sog. *Exception-Handler* für die verschiedenen Ausnahmen jeweils ein spezieller Ausnahmebehandlungsprozess vormodelliert werden kann. Die flexible ad-hoc-Ausnahmebehandlung mit dynamischer Erkennung von Inkonsistenzen und ihre Behandlung durch Markierungsoperationen werden in dieser Form in [RAH06] allerdings nicht untersucht.

Der Ansatz *AHEAD* unterstützt zu einem gewissen Grad die Behandlung von Ausnahmen. So erlaubt der Ansatz, vom ursprünglich modellierten Prozess abzuweichen [JSW99]. Der Begriff *Inkonsistenz* hat jedoch eine andere Bedeutung als in COREPRO: eine Inkonsistenz bedeutet in *AHEAD* das Abweichen vom Prozessmodell (z.B. durch Veränderung des Kontrollflusses). Die Idee ist hier jedoch Instanzänderungen des Kontrollflusses auf das entsprechende Prozessschema zu propagieren und umgekehrt.

Der WEP Ansatz beschreibt ebenfalls Mechanismen zur Behandlung von Ausnahmen. Ausnahmen sind in diesem Ansatz durch das Vorliegen *neuerer* Eingabedaten definiert (d.h. die Ergebnisse früherer Prozessausführungen sind nicht mehr gültig; vgl. semantische Inkonsistenz in Abschnitt 6.3.2) [Beu03]. Hinsichtlich Konsistenzherstellung im Falle von Ausnahmen werden im WEP-Ansatz zwei verschiedene Mechanismen verfolgt: *Undo-Redo-* sowie *Merge-Konsistenzsicherung*. Beim *Undo-Redo-Konsistenzsicherungsverfahren* werden (im Sinne einer Kompensation) die Prozesse, die auf den betroffenen Daten arbeiten, abgebrochen und Datenobjekte mit „Undo“ markiert. Anschließend wird der Prozess erneut ausgeführt. Dies entspricht in etwa dem Rücksprung in COREPRO. Das *Merge-Konsistenzsicherungsverfahren* in WEP veranlasst nur die Wiederholung derjenigen Schritte, die für die Erhaltung der Datenkonsistenz notwendig sind. Diese Verfahren bzw. die Auswirkungen von Inkonsistenzen werden jedoch nicht näher beschrieben.

Im *DYCHOR*-Ansatz wird die Adaption einer Prozess-Choreographie (d.h. Prozessstrukturen), bestehend aus öffentlichen Schnittstellen und privaten Prozessen, untersucht [RWR06b, RWR06a]. Hierbei wird insbesondere die Frage adressiert, welchen Einfluss Änderungen des Prozess-internen Kontrollflusses auf die Prozessstruktur haben. In datengetriebenen Prozessstrukturen würde dies der Adaption von OLCs und den Auswirkungen auf externe Transitionen entsprechen, was COREPRO nicht adressiert (vgl. Abschnitt 6.4). *DYCHOR* betrachtet damit aber im Prinzip den Aspekt der Fortpflanzung von Inkonsistenzen innerhalb einer Prozessstruktur, aber auf struktureller Ebene [RWR06a, RWR06b]. So erfordert die Änderung einer öffentlichen Schnittstelle der Prozess-Choreographie ggf. die Anpassung weiterer Schnittstellen. Allerdings betrachtet *DYCHOR* nicht die dynamische Adaption. Diese Betrachtung ist zwar erklärtes Ziel des Ansatzes, es wurden aber bislang nur Arbeiten zur statischen Adaption veröffentlicht.

Das Fortpflanzen von Inkonsistenzen findet sich auch im Bereich der Datenbanken in Form des *Cascading Rollback* Problems [MFS82]. In Datenbanken können Änderungstransaktionen ineinander geschachtelt werden. Hierbei ist der Abschluss einzelner Transaktionen wünschenswert, um anderen Transaktionen bereits das Lesen der geänderten Daten zu gestatten. Wird in diesem Kontext eine „äußere“ Transaktion aber nicht erfolgreich abgeschlossen, sondern abgebrochen, müssen alle davon betroffenen Transaktionen ebenfalls abgebrochen werden. Dies ist im Allgemeinen jedoch nicht tolerabel. Daher wird der Zugriff auf geänderte Daten in der Regel erst mit Abschluss der Transaktion erlaubt.

Die Fortpflanzung von Inkonsistenzen wurde in der untersuchten Literatur zum Thema Prozess-Management ansonsten kaum behandelt. Dies liegt an der Tatsache, dass viele Ansätze konsistenzgefährdende Operationen durch Definition von Vorbedingungen für Operationen erst gar nicht zulassen. Der Benutzer wird damit auch bei der Herstellung einer geeigneten Laufzeitmarkierung, die den Vorbedingungen für die Durchführung der Operation genügt, nicht geeignet unterstützt. Sind Inkonsistenzen durch Vorbedingungen für Markierungsoperationen ausgeschlossen, muss die Ausnahmebehandlung mit der letzten erwarteten Inkonsistenz starten und der

Konsistenzherstellungsprozess praktisch in umgekehrter Reihenfolge durchlaufen werden (vgl. Abbildungen 7.13 und 7.14). Hier liegt eine der Stärken des COREPRO-Ansatzes, der durch Kennzeichnung erwarteter Inkonsistenzen und die automatische Erstellung eines Konsistenzherstellungsprozesses erst eine systematische Ausnahmebehandlung erlaubt und auch in diesem Fall die Korrektheit der Prozessstruktur garantiert.

7.7 Zusammenfassung

Dieses Kapitel beschreibt detailliert verschiedene Konzepte zur Behandlung von Ausnahmen. Hierfür abstrahieren wir von der Herkunft und Art einer Inkonsistenz und stellen stattdessen generische Operationen für die Behandlung von fachlichen Ausnahmen und Inkonsistenzen vor. Das Kapitel rundet mit den Konzepten zur manuellen Ausnahmebehandlung den in Abschnitt 6.3.3 beschriebenen Fehlerbehandlungsprozess (vgl. Abbildung 6.11) ab. Sie erlauben es, die inkonsistente Laufzeitmarkierung einer Prozessstruktur in eine konsistente Laufzeitmarkierung zu überführen (*Forward-Recovery*).

Der Vergleich mit verwandten Arbeiten zeigt, dass COREPRO einen wichtigen Beitrag für die Behandlung von Ausnahmen liefert. Praktische Relevanz hat hier insbesondere die Möglichkeit, temporär Inkonsistenzen zu erzeugen und die Konsequenzen derartiger Ausnahmen geeignet zu analysieren. Dazu gehört auch die Erzeugung eines optimierten Konsistenzherstellungsprozesses, von dessen Ablauf selbstverständlich abgewichen werden kann.

Das Ignorieren von Inkonsistenzen vervollständigt die Ausnahmebehandlung in datengetriebenen Prozessstrukturen und hilft, praxisnahe Abläufe besser abzubilden. Der Grund für das Ignorieren von Inkonsistenzen liegt häufig in fachlichen Entscheidungen. Auch der Ausblick auf weitergehende Aspekte und Sonderfälle zeigt, dass der COREPRO-Ansatz eine gute Grundlage für die Abbildung erweiterter Anforderungen darstellt.

Kapitel 7 schließt damit Teil II zur technischen Lösung der in Kapitel 2 definierten Anforderungen ab. Teil III beschreibt die prototypische Realisierung und die Validation der vorgestellten Lösungskonzepte.

Teil III

Validation der Konzepte

8 Prototypische Realisierung

In Teil II dieser Arbeit haben wir innovative Konzepte für die IT-Unterstützung datengetriebener Prozessstrukturen vorgestellt und damit die in Kapitel 2 aufgestellten Anforderungen erfüllt. Die technische Realisierbarkeit und praktische Anwendbarkeit dieser Konzepte wollen wir nun weiter untermauern und gehen in Teil III dieser Arbeit auf die Aspekte der Realisier- und Anwendbarkeit der Konzepte ein. Kapitel 8 präsentiert den im Rahmen dieser Arbeit implementierten (Software-)Demonstrator, der sämtliche entwickelten Konzepte umsetzt und damit ihre Realisierbarkeit unterstreicht. In Kapitel 9 diskutieren wir Anwendungsszenarien aus einer durchgeführten Fallstudie und diskutieren kritisch deren Umsetzung mit den CoRePro-Konzepten.

Kapitel 8 gliedert sich wie folgt: In Abschnitt 8.1 werden die Zielsetzung und die Anforderungen an die technische Realisierung der Konzepte zusammengefasst. Weiter beschreiben wir die gewählte Architektur für den Demonstrator. Die Abschnitte 8.2 bis 8.6 behandeln ausgewählte Aspekte der Realisierung des Ansatzes. Hierbei unterscheiden wir jeweils zwischen technischer Umsetzung der Konzepte und ihrer Anwendung. Abschnitt 8.2 beschreibt die Realisierung der Modellierungskonzepte aus den Kapiteln 4 und 5 mit Erstellung der Modellschicht, der automatischen Generierung einer datengetriebenen Prozessstruktur, der persistenten Speicherung der Modelle sowie der Korrektheitsprüfung von *Object Life Cycles* (OLCs) und *Life Cycle Coordination Model* (LCM). Abschnitt 8.3 diskutiert die Realisierung der operationalen Semantik durch Implementierung der in Kapitel 4 vorgestellten Ausführungs- und Markierungsregeln. Die Realisierung erlaubt die Simulation der Ausführung einer Prozessstruktur durch eine geeignete Interaktion mit dem Benutzer. Abschnitt 8.4 beschreibt die Implementierung der in Kapitel 6 behandelten Operationen zur dynamischen Adaption der Prozessstruktur und die Implementierung der zugehörigen Konsistenzanalyse. Abschnitt 8.5 präsentiert die Umsetzung der in Kapitel 7 diskutierten Operationen zur Ausnahmebehandlung sowie die technische Umsetzung der Kennzeichnung erwarteter Inkonsistenzen. Abschnitt 8.6 diskutiert Aspekte der Realisierung einer Ausführungskomponente zur Koordination „realer“ Prozesse und Abschnitt 8.7 gibt eine Zusammenfassung.

8.1 Einleitung

Die Realisierung des (Software-)Demonstrators stellt einen wichtigen Grundpfeiler hinsichtlich der Überprüfung und Umsetzbarkeit der COREPRO-Konzepte dar. Mit der prototypischen Implementierung verfolgen wir des Weiteren das Ziel, die erarbeiteten Konzepte anschaulich zu präsentieren. Aus diesen beiden Zielsetzungen ergeben sich verschiedene Fragestellungen für die Umsetzung des Demonstrators:

- Wie können die Lösungskonzepte aus Teil II technisch abgebildet werden?
- Wie kann die technische Abbildung der Konzepte „nahe“ an den formalen Teilen der Arbeit geschehen, mit dem Ziel, diese geeignet zu validieren?
- Wie muss die technische Architektur gewählt werden, um die Abbildung der formalen Modelle und der Logik zu unterstützen?
- Wie können die Konzepte geeignet visualisiert werden?

Zunächst definieren wir wichtige funktionale und technische Anforderungen an die prototypische Implementierung und stellen anschließend das gewählte Architekturkonzept vor.

8.1.1 Funktionale Anforderungen

Der Demonstrator soll den kompletten Lebenszyklus datengetriebener Prozessstrukturen aus Abbildung 2.2 (siehe Seite 18) mit Modellierung, Ausführung, Adaption sowie Ausnahmebehandlung abdecken. Im Folgenden fassen wir die Konzepte in Form von funktionalen Anforderungen an die Implementierung kurz zusammen und erweitern diese um spezifische Anforderungen an den Demonstrator.

Anforderung 8.1 (Erstellung der Modellebene)
Der Demonstrator soll die grafische Erstellung der Modellschicht bestehend aus Datenmodell, OLCs und LCM ermöglichen. Die Korrektheit der Modelle soll in der grafischen Oberfläche, soweit möglich, „per Konstruktion“ garantiert werden (z.B. Transitionen verbinden immer zwei Zustände im OLC). Des Weiteren sollen Korrektheitsprüfungen der erstellten Modelle (OLCs und LCM) erfolgen. Dazu gehört auch die geeignete Visualisierung etwaiger Probleme (z.B. Zyklus im LCM; vgl. Abschnitt 5.4). Die Modelle sollen gespeichert und wieder geladen werden können.

Anforderung 8.2 (Erzeugung datengetriebener Prozessstrukturen)
Auf Basis der erstellten Modelle soll die grafische Erstellung von Datenstrukturen ermöglicht werden. Dazu sollen dem Benutzer die im Datenmodell enthaltenen Objekt- und Relationstypen zur Verfügung stehen. Ferner soll die Prozessstruktur, wie in Abschnitt 5.3.2 beschrieben, auf Basis der modellierten Datenstruktur automatisch erzeugt werden.

Anforderung 8.3 (Simulation und Ausführung datengetriebener Prozessstrukturen)
Die vollständige Implementierung der operationalen Semantik für Prozessstrukturen ist wichtig für die Validation der in Kapitel 4 definierten Ausführungs- und Markierungsregeln. Dazu soll der Demonstrator über einen Simulations-Modus verfügen, mit dem zunächst keine realen Prozesse ausgeführt werden, sondern die Ausführung von Prozessen (bzw. deren Beendigung) durch

Benutzerinterkation simuliert werden kann. Der aktuelle Ausführungszustand, also die Laufzeitmarkierung der Prozessstruktur, soll geeignet visualisiert sowie gespeichert und anschließend wieder geladen werden können. Des Weiteren soll es zu Demonstrationszwecken möglich sein, die Prozessstruktur „zurückzusetzen“, also die Markierung zu reinitialisieren, um die Prozessstruktur erneut zu durchlaufen (vgl. Initialisierungsregel IR2). Ferner soll die Simulation über einen „Rückgängig / Wiederherstellen“ (*Undo/Redo*) Mechanismus verfügen, um Markierungsänderungen geeignet präsentieren zu können.

Die Koordination realer Prozesse erfordert des Weiteren, dass konkrete Prozesse mit Transitionen verknüpfbar sind. Diese Prozesse können verschiedene Implementierungen aufweisen (z.B. Java-Programm oder Web-Service [ACKM04]). Der Demonstrator soll daher über eine generische Schnittstelle verfügen, mit der die Anbindung verschiedener Prozesse möglich wird.

Anforderung 8.4 (Dynamische Adaption datengetriebener Prozessstrukturen)
Die dynamische Adaption von Prozessstrukturen soll, wie in Kapitel 6 beschrieben, im Demonstrator realisiert werden. Eine Änderung der Datenstruktur soll automatisch in Adaptionen der Prozessstruktur transformiert werden. Die damit einhergehende Konsistenzanalyse sowie die Kennzeichnung und Anzeige auftretender Inkonsistenzen ist dabei genauso von Bedeutung, wie die Realisierung des Transaktionsmechanismus aus Abschnitt 6.3.5.

Anforderung 8.5 (Ausnahmebehandlung)
Der Demonstrator soll die vorgestellten Konzepte zur Ausnahmebehandlung (vgl. Kapitel 7) realisieren. Dazu gehören die Bereitstellung von Markierungsoperationen für Sprünge sowie das Zurücksetzen externer Transitionen. Die Konsistenzanalyse der dynamischen Adaption soll auf die Markierungsoperationen ausgedehnt werden und zusätzlich die erwarteten Inkonsistenzen visualisieren. Die Mechanismen zum „Ignorieren“ von Inkonsistenzen und externen Transitionen (vgl. Abschnitt 7.4) sollen ebenfalls vollständig implementiert werden.

8.1.2 Technische Anforderungen

Aus den zu Beginn dieses Abschnitts genannten Zielen ergeben sich neben den funktionalen auch technische Anforderungen. Aspekte sind hierbei die grafische Darstellung der Konzepte, die Persistenz der Modelle, die Mächtigkeit und Erweiterbarkeit der technischen Plattform sowie, im Kontext dieser Arbeit, organisatorische Rahmenbedingungen.

Anforderung 8.6 (Grafische Darstellung)
Für die Demonstration der Konzepte ist eine grafische Oberfläche unerlässlich. Sie soll nicht nur für die Modellierung und Darstellung datengetriebener Prozessstrukturen genutzt werden, sondern auch während der Simulation die Interaktion mit dem Benutzer und während der Ausführung eine geeignete Überwachung der Prozesse erlauben. Ziel für die Realisierung der Benutzersicht des Demonstrators ist weniger ein Sichtenkonzept (z.B. eine Aufbereitung komplexer Modelle durch Ausblenden von Teilstrukturen für Endbenutzer) zu implementieren, als vielmehr eine geeignete Darstellung zur Demonstration der Konzepte zu erarbeiten.

Die betrachteten Prozessstrukturen können sehr groß werden (vgl. Beispielrechnung in Abschnitt 5.6). Der Demonstrator soll daher über eine grafische Darstellung verfügen, die auch große Prozessstrukturen übersichtlich darstellen kann (z.B. durch Zoom-Funktionalität).

Anforderung 8.7 (Mächtigkeit)
Die technische Plattform des Demonstrators benötigt eine ausreichende Mächtigkeit, um die Logik der COREPRO-Modelle abzubilden. Dazu gehört die Analysierbarkeit der erstellten Modelle, ebenso wie Umsetzbarkeit des Mechanismus zur Erzeugung der Prozessstruktur auf Basis einer gegebenen Datenstruktur. Des Weiteren muss die technische Plattform die Realisierung der Laufzeitaspekte unterstützen. Das beinhaltet sowohl die Abbildung vom Laufzeitmarkierungen und Kennzeichnungen als auch die Umsetzung der operationalen Semantik. Sie sind schließlich Voraussetzung für die Realisierung der Operationen zur Ausnahmebehandlung (z.B. Sprung mit temporärem Einfügen von Transitionen) sowie der dazugehörenden Konsistenzanalyse.

Anforderung 8.8 (Erweiterbarkeit)
Das mit der Entwicklung des Demonstrators verbundene dynamische Umfeld erfordert eine möglichst flexible und erweiterbare Architektur.[1] Um auch Folgekonzepte abbilden zu können, ist eine offene Systemarchitektur nötig, die „in alle Richtungen“ erweiterbar ist. Dazu gehört eine ausreichende Dokumentation der Programmierschnittstellen der technischen Plattform.

Es ist ebenfalls eine gewisse Flexibilität in der Handhabung der persistenten Informationen gefordert, um auf neue Konzepte reagieren zu können und um beispielsweise alte Dateien an neue Dateiformate anzupassen. Dazu soll ein menschenlesbares Speicherformat gewählt werden, das die Veränderung der gespeicherten Information sowie die Kopie und den Austausch der Daten auf einfache Art und Weise ermöglicht. Des Weiteren muss der „Offline-Betrieb“ für Demonstrationszwecke möglich sein.

Eine weitere Anforderung ist die einfache Handhabung der Demonstrator-Software und die Minimierung des Installationsbedarfs für die notwendige Umgebung. Performanz spielt in der Demonstration der Konzepte eine untergeordnete Rolle.

Anforderung 8.9 (Organistorische Rahmenbedingungen)
Ein wichtiger Aspekt bei einer Software-Implementierung ist der Ressourceneinsatz. Im Rahmen dieser Arbeit standen nur begrenzte Entwicklungsressourcen zur Verfügung. Um den Implementierungsaufwand so gering wie möglich zu halten, war es bei der Implementierung das Ziel, nicht „auf der grünen Wiese“ zu beginnen. Stattdessen sollte ein bestehendes Rahmenwerk als technische Plattform verwendet werden, welches bereits ein Grundgerüst an Funktionalitäten besitzt (z.B. grafische Benutzeroberfläche mit grafischen Modellierungskomponenten). Dazu musste ein Kompromiss zwischen Mächtigkeit, Erweiterbarkeit und Einarbeitungsaufwand in die technische Plattform gefunden werden.

8.1.3 Architekturentscheidung

Die Planung eines Software-Demonstrators wirft zunächst die Frage nach der technischen Plattform auf. Entsprechend der technischen Anforderungen ist hier das Ziel, auf einem bestehenden Rahmenwerk aufzusetzen, das die notwendigen Voraussetzungen für die Erfüllung der Anforderungen an Darstellung, Persistenz, Mächtigkeit und Erweiterbarkeit mit sich bringt. Für diese Realisierung kommen grundsätzlich verschiedene Plattformen in Frage.

[1] Zu Beginn der Implementierung waren die vorgestellten Konzepte noch nicht vollständig erarbeitet. Die Entwicklung des Demonstrators erfolgte daher parallel zur Erarbeitung der Konzepte.

Als Vertreter der Produktdaten-Management-Systeme wurde *UGS Teamcenter Engineerung*[2] auf die Umsetzbarkeit der Konzepte untersucht. Es bietet zum einen eine mächtige Datenverwaltung, zum anderen eine (einfache) Prozess-Management-Komponente. Diese unterstützt zwar flexible Abläufe (z.B. durch Sprünge), bietet jedoch keine Korrektheitsanalysen für modellierte Prozesse [MHHR06]. Die geschlossene Architektur und die ungenügende Dokumentation der Programmierschnittstelle (*Application Programming Interface*) erlauben jedoch kaum Erweiterungsmöglichkeiten, so dass die Umsetzung der Konzepte in diesem System unter den gegebenen Rahmenbedingungen (vgl. hierzu die Diskussion in Anforderung 8.9) nicht möglich ist.

Weitere kommerzielle Systeme aus dem Bereich Prozess-Management, wie *IDS Scheer ARIS Toolset* oder *MID Innovator* bieten einen großen Leistungsumfang hinsichtlich der Prozessmodellierung mittels *Ereignis Prozess Ketten (EPKs)* oder UML-Aktivitätendiagrammen. Ihre Nutzung für die Abbildung der CoRepro-Konzepte ist allerdings aufgrund kaum dokumentierter Schnittstellen bzw. deren schlechter Erweiterbarkeit nicht möglich [Inn07]. Dazu kommt die geringe Datenorientierung, die eine umfangreiche Erweiterung oder Transformation der vorhandenen Modelle erfordern würde. Ferner ist in diesen Anwendungen keine ausreichende Unterstützung zur flexiblen Steuerung und dynamischen Adaption von Prozessen gegeben.

Ein *Open-Source* Editor für die grafische Modellierung von Ontologien ist *Protégé* [Pro]. Damit können, basierend auf dem *Ressource Description Framework* (RDF), Beziehungen zwischen Daten semantisch beschrieben werden. Datenmodelle und Datenstrukturen, wie sie in CoRepro genutzt werden, lassen sich theoretisch abbilden, denn die in CoRepro genutzten Modelle sind generisch (vgl. Abschnitt 5.2.1). Protégé stellt außerdem verschiedene Erweiterungsmöglichkeiten, z.B. für die Visualisierung von Ontologien, bereit. Die Erzeugung und insbesondere die Transformation der Prozessstrukturen werden jedoch aufgrund der unterschiedlichen Paradigmen nicht unterstützt. Die Repräsentation von Laufzeitinformationen für dynamische Abläufe, wie sie für die Simulation von Prozessstrukturen benötigt werden, ist nicht Ziel einer semantischen Beschreibung in Protégé. Des Weiteren werden hier weder die Abbildung einer operationalen Semantik noch die Durchführung von Konsistenzanalysen direkt unterstützt. Eine Beschränkung auf dieses Rahmenwerk birgt vielmehr die Gefahr, dass sich die Anforderungen hinsichtlich Mächtigkeit und Erweiterbarkeit nicht vollständig umsetzen lassen.

Die *Eclipse Development Platform* ist ein *Open-Source* Rahmenwerk für die Entwicklung von Software [Ecl]. Die offene Architektur von Eclipse erlaubt die Erweiterung der Plattform. Eine solche Erweiterung ist das *Graphical Editing Framework* (GEF), welches die grafische Erstellung generischer Modelle erlaubt [GEF]. GEF stellt zusammen mit Eclipse ein Rahmenwerk zur Verfügung, das beispielsweise die Dateiverwaltung, hilfreiche Benutzerfunktionen (z.B. Rückgängig-Funktion) sowie die grafische Modellierung entsprechend eines gegebenen Metamodells erlaubt. Eclipse und GEF sind Grundlage zahlreicher Demonstratoren, die in den letzten Jahren im Prozessumfeld entstanden sind (z.B. *Event Driven Process Chains* [EPC] und *Eclipse Process Framework* [EPF]). Eclipse und GEF bieten durch die offene Architektur, die verfügbare Dokumentation sowie die Nutzung einer mächtigen Programmiersprache (Java) eine erweiterbare Basis, die den in Abschnitt 8.1.2 eingeführten technischen Anforderungen genügt. Sie wurden daher als technische Plattform für den CoRepro-Demonstrator gewählt. Die Umsetzbarkeit der Konzepte mit Eclipse und GEF wurde in einer ersten Implementierung gezeigt, die die manuelle Erstellung von Prozessstrukturen erlaubt [MRHP07].

[2]Untersuchte Version: 9.12

Als Architektur für die Implementierung des Demonstrators wurde ein Mehrschichtenmodell, basierend auf dem *Model-View-Controller (MVC)* Architekturmuster, gewählt [KP88]. Das Muster beschreibt die Kapselung der Software-Funktionen in die drei Schichten *Model*, *View* und *Controller* und ermöglicht damit einen erweiterbaren Architekturentwurf. Abbildung 8.1 zeigt die logische Verteilung der Funktionen unseres Demonstrators. Die Bedeutung der Schichten wird im Folgenden beschrieben.[3]

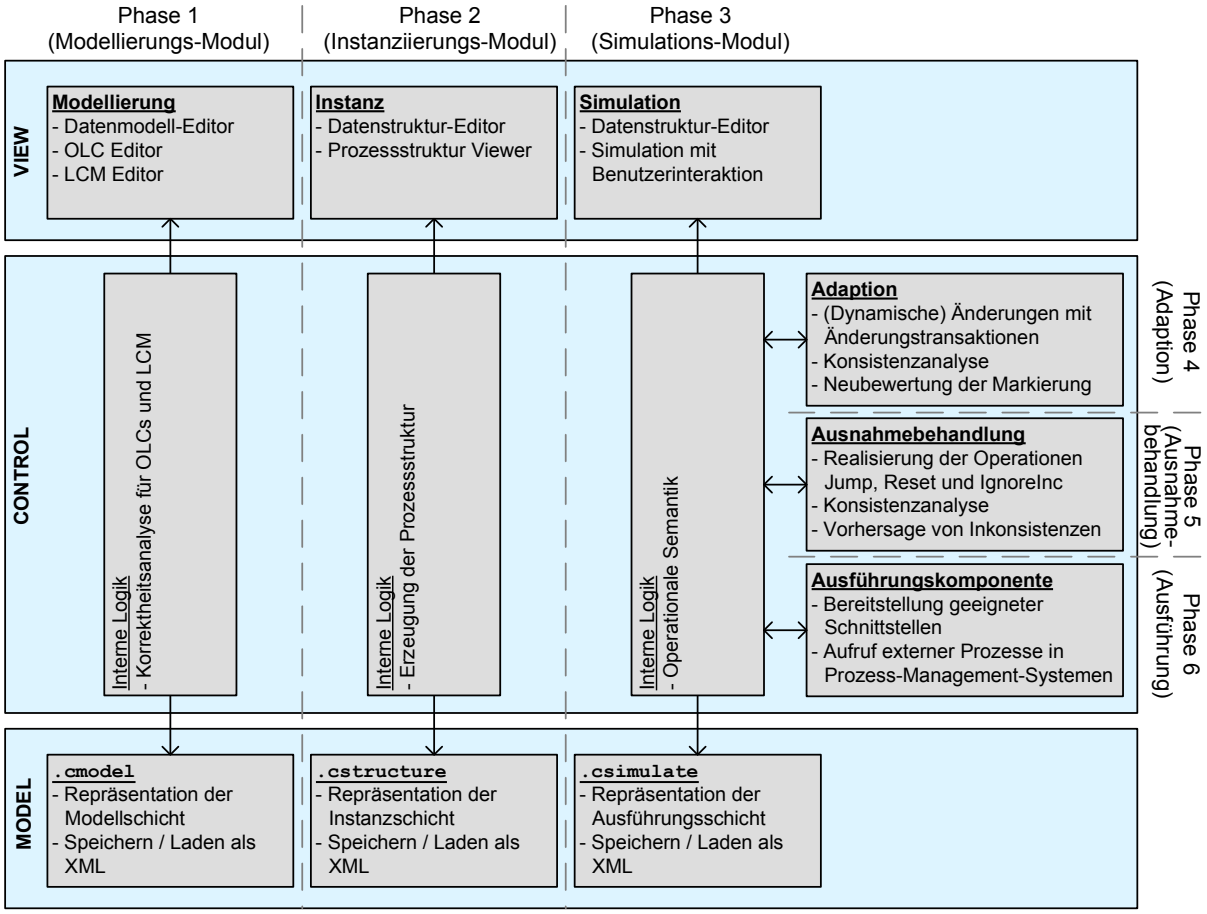


Abbildung 8.1: Architekturkonzept und Realisierungsphasen des Demonstrators

Model (Modellschicht)

Die Modellschicht erledigt die Datenhaltung. Sie repräsentiert die Gesamtheit der COREPRO-Modelle (d.h. Datenmodell, Datenstruktur, LCM und Prozessstruktur) in Form einer Java-Objektstruktur und sorgt für deren persistente Sicherung. Die Speicherung als XML-Struktur erlaubt eine einfache Handhabung der persistenten Daten und erfüllt damit Anforderung 8.8. Es stehen ferner verschiedene Methoden für die Manipulation der Datenmodelle zur Verfügung (z.B. *Hinzufügen eines Objekts*).

[3]Die Unterbringung der Geschäftslogik im MVC-Modell wird in der Literatur unterschiedlich interpretiert. Wir verstehen sie als Teil der Kontrollschicht.

View (Darstellungsschicht)

Die Darstellungsschicht erlaubt die grafische Repräsentation der Modellschicht. GEF unterstützt die Erstellung von grafischen Editoren für Diagramme basierend auf definierbaren Metamodellen. GEF selbst basiert ebenfalls auf einer MVC-Architektur. Die View-Schicht von GEF entspricht der Darstellungsschicht unseres Architekturmodells und visualisiert in der grafischen Oberfläche diejenigen *Java-Objekte*[4], die in CoRePro in der Modellschicht abgebildet sind. Die Control-Schicht von GEF bezieht sich auf die Interaktion mit der Oberfläche und kann daher im CoRePro-Architekturkonzept als Verbindung zwischen Darstellungs- und Kontrollschicht verstanden werden.

Control (Kontrollschicht)

Die Kontrollschicht bildet die interne Logik des Demonstrators ab. Hier sind beispielsweise Funktionen für die Generierung der Prozessstruktur sowie Analysefunktionen für die Korrektheitsprüfung und die Simulationskomponente zur Abbildung der operationalen Semantik logisch angeordnet.

Zur Reduzierung der Komplexität, aber auch bedingt durch die begleitende Entwicklung der Konzepte, wurde das Vorgehen bei der Realisierung des Demonstrators in die Software-Module *Modellierung*, *Instanziierung* und *Simulation* sowie zeitlich in sechs Entwicklungsphasen unterteilt:

- *Phase 1 (Modellierungs-Modul)*: Einarbeitung in die technische Infrastruktur, Definition der internen Datenrepräsentation (inklusive XML-Struktur) und Erstellung des Moduls für die Abbildung der Modellebene (d.h. Editoren für Datenmodell, OLCs und LCM). Erfahrungsgemäß ist in der Software-Entwicklung zu Beginn ein hoher Initialaufwand (z.B. durch Einarbeitung in die technische Plattform) notwendig, weshalb in Phase 1 der Implementierung ein relativ kleiner Funktionsumfang über alle Schichten der Architektur realisiert wurde (*Durchstich* [BMNW05]).
- *Phase 2 (Instanziierungs-Modul)*: Erstellung des Moduls für die Instanzebene (Editor bzw. Viewer für Daten- bzw. Prozessstruktur) sowie Implementierung des Algorithmus zur Generierung datengetriebener Prozessstrukturen.
- *Phase 3 (Simulations-Modul)*: Implementierung der operationalen Semantik und Realisierung des Moduls für die Simulation.
- *Phase 4 (Adaption)*: Realisierung der Operationen zur dynamischen Adaption, der Konsistenzanalyse und der Mechanismen zur Unterstützung von Änderungstransaktionen.
- *Phase 5 (Ausnahmebehandlung)*: Umsetzung der Konzepte zur Ausnahmebehandlung mit Markierungsoperationen `JUMP` und `RESET`, der erweiterten Konsistenzprüfung, der Anzeige erwarteter Inkonsistenzen sowie den flexiblen Konzepten zum Ignorieren von Inkonsistenzen und externen Transitionen.

[4] Um Objekte der Datenstruktur von Objekten der objektorientierten Implementierung im Demonstrator abgrenzen zu können, werden Objekte der Implementierung als *Java-Objekte* bezeichnet.

- *Phase 6 (Ausführung)*: Definition geeigneter Schnittstellen für die Ausführung externer Prozesse und Implementierung geeigneter *Wrapper* bzw. *Adapter* zur Anbindung von Java-Anwendungen, Web-Services und konkreter Prozess-Management-Systeme.

Der Demonstrator (COREPRO$_{SIM}$) umfasst derzeit etwa 400 Klassen und 44.000 Zeilen Quellcode. Im Folgenden kann nicht auf alle Implementierungsaspekte eingegangen werden. Stattdessen werden ausgewählte Konzepte der technischen Umsetzung und ihrer Anwendung aus Benutzersicht vorgestellt.

8.2 Realisierung der Modellierungskonzepte

Die Phasen 1 und 2 der Entwicklung von COREPRO$_{SIM}$ umfassen die Realisierung der Modellierungskonzepte, wie sie in Kapitel 5 beschrieben werden. Wir beschreiben zunächst die technischen Aspekte der Korrektheitsprüfung von OLCs. Aus Benutzersicht entspricht die Modellierung und (automatische) Erzeugung einer Prozessstruktur in COREPRO$_{SIM}$ dem Vorgehen, das bereits in Abbildung 5.2 (siehe Seite 112) eingeführt wurde.

Technische Aspekte

Die Modellschicht der Entwicklungsphase 1 repräsentiert das Datenmodell, die OLCs und den LCM. Diese sind zum einen durch eine geeignete Java-Klassenstruktur implementiert, zum anderen für die persistente Speicherung der Modelle als XML-Dateien (mit der Endung `.cmodel`) durch ein geeignetes XML-Schema definiert (vgl. Abbildung C.2 in Anhang C). Basierend auf diesen Modellen realisiert die Kontrollschicht die Mechanismen für ihre logische Verarbeitung. In Phase 1 der Entwicklung zählt hierzu die strukturelle Korrektheitsanalyse von OLCs. Ihr Ziel ist die Prüfung der Erreichbarkeit aller Elemente eines OLC auf Basis der Hilfsfunktionen aus Abschnitt 4.2.2. Hierfür verlangt Definition 4.5, dass jeder Zustand vom Startzustand aus und ebenso der Endzustand von jedem Zustand aus erreichbar ist. Diese Prüfung kann unter Nutzung des Algorithmus $path_{int}$ sichergestellt werden (vgl. Algorithmus 4.1 auf Seite 59 bzw. dessen Implementierung Anhang B). Um nicht für jeden Zustand einzeln die Erreichbarkeit prüfen zu müssen, erweitert COREPRO$_{SIM}$ den Algorithmus 4.1 dahingehend, dass optional auch Schleifenpfade durchlaufen werden können (Parameter mit Wert `true` in Zeile 5 des Algorithmus 8.1). Damit werden alle Transitionen, die auf einem Pfad zwischen Start- und Endzustand des OLC liegen (inklusive der Schleifenpfade), zurückgegeben. Wird das Delta `invalidStates` zwischen den erreichbaren und allen im OLC enthaltenen Zuständen gebildet (vgl. Zeilen 15-22 in Algorithmus 8.1), lassen sich diejenigen Zustände finden, die entweder vom Startzustand aus nicht erreichbar sind oder die keine Verbindung zum Endzustand haben. Diese Unterscheidung kann durch eine individuelle Prüfung der in `invalidStates` gesammelten Zustände erfolgen (vgl. Zeilen 23-37 in Algorithmus 8.1). Zustände, die nicht korrekt mit dem Start- oder dem Endzustand verbunden sind, werden in der Benutzeroberfläche entsprechend gekennzeichnet. Dafür werden geeignete Attribute der Java-Objekte gesetzt (vgl. Zeilen 17 und 20 in Algorithmus 8.1).

```
1  public boolean calculateOLCStructuralCorrectness(ObjectLifeCycleGraph olcmodel) {
   ...
   // Suche Trans. im OLC, die auf einem Pfad zw. Start- und Endzustand liegen (inkl. Schleifen)
   transitions = getOLCAnalysis(olcmodel).pathSearch(start, end, start, new
5      HashSet<SimpleIntTrans>(), new HashSet<SimpleIntTrans>(), true);
   Set<Integer> validStates = new HashSet<Integer>();
   for (Iterator i = transitions.iterator(); i.hasNext(); ) {
     SimpleIntTrans t = (SimpleIntTrans) i.next();
     validStates.add(t.getSource());
10   validStates.add(t.getTarget());
   }
   ...
   // Suche Zustände, die nicht auf einem Pfad zwischen Start- und Endzustand liegen
   Set<Integer> invalidStates = new HashSet<Integer>();
15 for (Iterator i = olcmodel.getAllStates().iterator(); i.hasNext(); ) {
     SimpleState s = (SimpleState) i.next();
     s.setIsCorrect(true);
     if (!validStates.contains(s.getIntegerID())) {
       invalidStates.add(s.getIntegerID());
20     s.setIsCorrect(false);
     }
   }
   for (Iterator i = invalidStates.iterator(); i.hasNext(); ) {
     SimpleState s = (SimpleState) olcmodel.getStateModelByID(((Integer)
25     i.next()).intValue());
     Set<SimpleIntTrans> sTrans = getOLCAnalysis(olcmodel).pathInt(start, s);
     if (sTrans.isEmpty() && !s.equals(start)) {
       CoreproConsole.out.println("  State " + s.getIntegerID() + " is not reachable from the
              start state.");
30   }
     Set<SimpleIntTrans> eTrans = getOLCAnalysis(olcmodel).pathInt(s, end);
     if (eTrans.isEmpty() && !s.equals(end)) {
       CoreproConsole.out.println("  State " + s.getIntegerID() + " has no connection to the
              end state.");
35   }
   }
}
```

Algorithmus 8.1: Realisierung der strukturellen Analyse von OLCs

Benutzersicht

In der grafischen Oberfläche stehen dem Benutzer verschiedene Editoren für die Erstellung und Änderung von Datenmodell, OLCs und LCM zur Verfügung. Abbildung 8.2 zeigt die Aufteilung der grafischen Oberfläche des Demonstrators in verschiedene Bereiche (z.B. *Navigation*, *Editor* und *Console*). Der Navigationsbereich (vgl. ① in Abbildung 8.2) erlaubt die Verwaltung der XML-Dateien, also zum Beispiel das Laden von `.cmodel`-Dateien (vgl. Abbildung 8.1). Im Editor (vgl. ② in Abbildung 8.2) ist die grafische Erstellung und Bearbeitung des Datenmodells, der OLCs und des LCM möglich. Alle grafischen Modellierungsoberflächen in COREPRO$_{SIM}$ verfügen über eine Übersichtsdarstellung (*Outline*) und eine Zoom-Funktion für die Navigation in großen Modellen bzw. Strukturen (vgl. ③a und ③b in Abbildung 8.2 bzw. Anforderung 8.6). Ferner dient die Console der Ausgabe von Statusinformationen, zum Beispiel für die Korrektheitsanalyse des LCM (vgl. ④ in Abbildung 8.2).

Die Modellierung einer datengetriebenen Prozessstruktur erfolgt in COREPRO$_{SIM}$ in drei Schritten (vgl. Abbildung 5.2a und b auf Seite 112): Im ersten Schritt wird das Datenmodell bestehend aus Objekt- und Relationstypen (zwischen den Objekttypen) angelegt. Abbildung 8.2 zeigt ein Datenmodell mit den Objekttypen `Total System`, `System` und `Subsystem` sowie den Relationstypen `hasSys`, `usesSubsys` und `hasSubsys`.

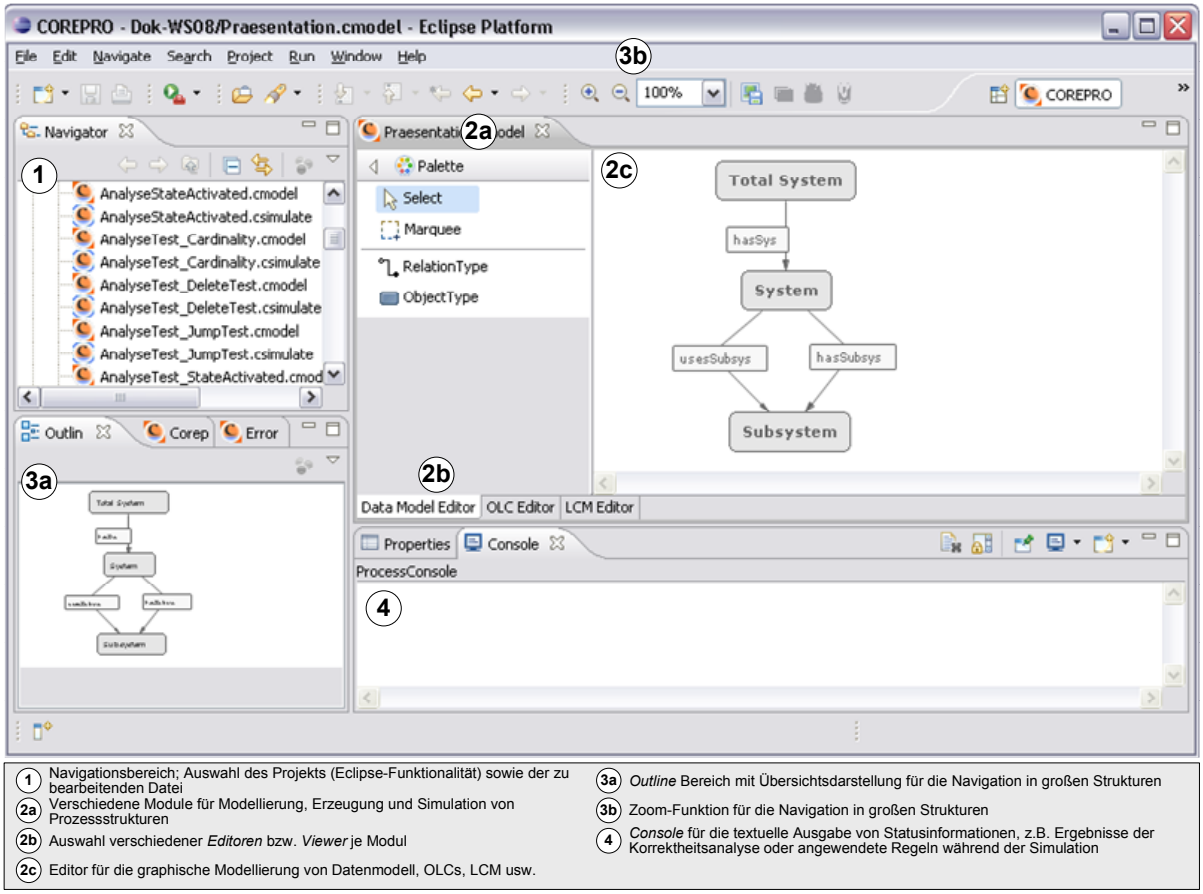


Abbildung 8.2: Benutzeroberfläche mit Aufteilung in unterschiedliche Funktionsbereiche

Im zweiten Schritt der Modellierung werden die OLCs für die modellierten Objekttypen definiert. Sie bestehen aus Zuständen und internen Transitionen (vgl. Abbildung 8.3). Es lassen sich Verzweigungen ebenso wie Rücksprünge modellieren. Für die Simulation der Prozessstrukturen werden interne Transitionen zunächst nicht mit „realen“ Prozessen (d.h. die in Prozess-Management-Systemen zur Ausführung kommen) verknüpft (vgl. Anforderung 8.3). Stattdessen kann zwischen *Default* (d.h. einer Prozesssimulation) und *None* (d.h. dem Stellvertreterprozess τ) gewählt werden. Jeder OLC kann auf strukturelle Korrektheit geprüft werden. Hierfür wird geprüft, ob jeder Zustand bzw. jede Transition vom Startzustand aus erreichbar ist und ob ein Pfad zum Endzustand führt (vgl. Abbildung 8.3).

Im dritten Schritt der Modellierung wird der LCM angelegt, also die externen Transitionen zur Synchronisation von Zuständen verschiedener OLCs definiert. Dafür können externe Transitionen (mit oder ohne Synchronisationsprozess, d.h. mit Stellvertreterprozess τ) eingefügt und mit Relationstypen verknüpft werden (vgl. Abbildung 8.4). Der LCM kann ebenfalls auf Korrektheit geprüft werden. Hier wird die Zyklensuche, entsprechend Definition 5.1, durchgeführt und gefundene Zyklen werden in der Prozessstruktur visualisiert (vgl. Abbildung 8.4).

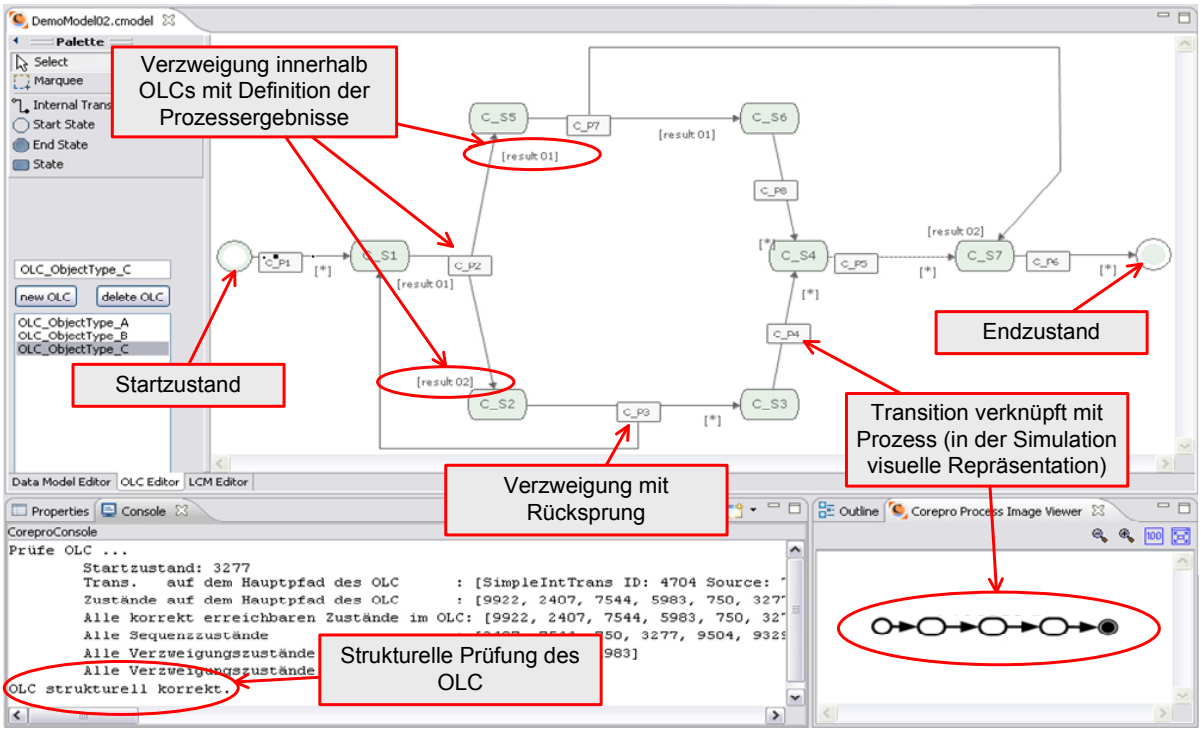


Abbildung 8.3: Grafische Modellierung von OLCs

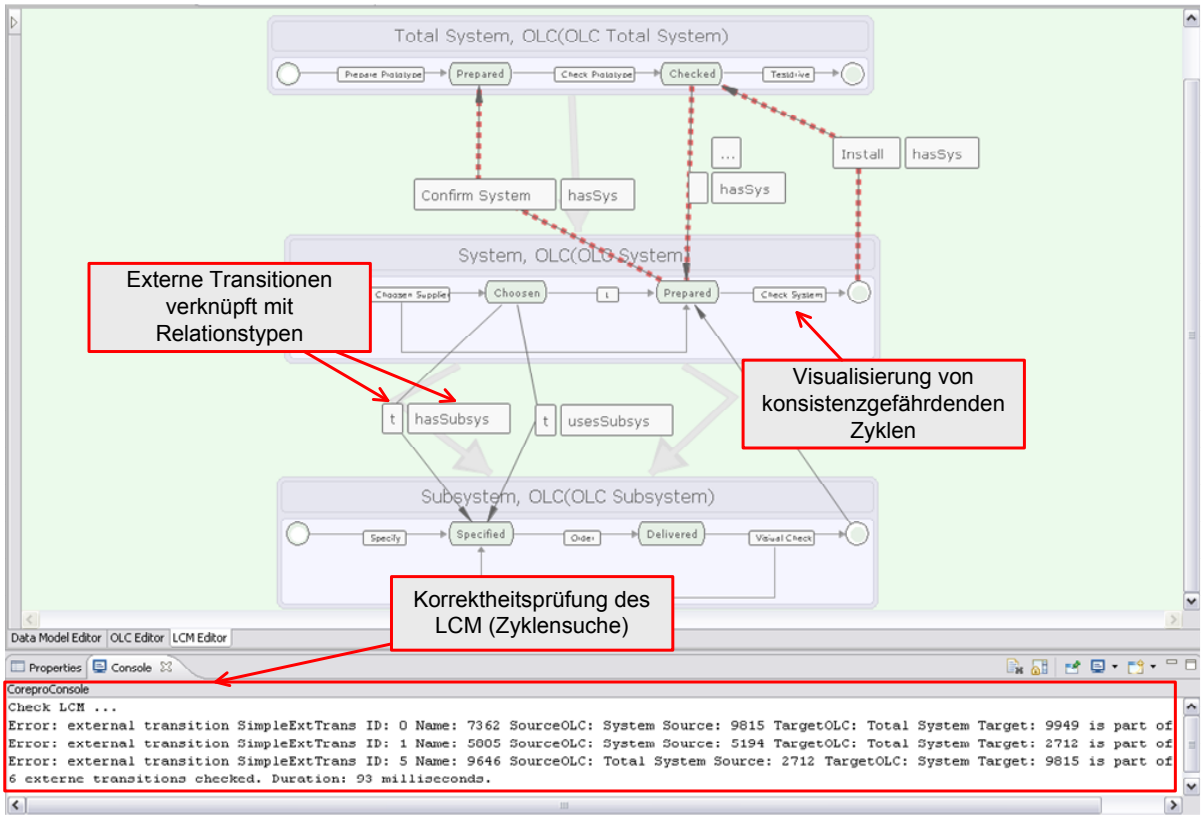


Abbildung 8.4: Synchronisation von OLCs zur Erstellung des LCM

Die Instanzebene ist in COREPRO$_{SIM}$ als eigenständiges Modul implementiert. Im grafischen Editor für die Datenstruktur stehen dem Benutzer die im Datenmodell angelegten Objekt- und Relationstypen zur Verfügung. Aus ihnen können beliebige Datenstrukturen modelliert werden (vgl. Abbildung 8.5a). Ausgehend von der Datenstruktur kann in COREPRO$_{SIM}$ automatisch die zugehörige datengetriebene Prozessstruktur erzeugt werden (vgl. Abbildung 8.5b). Durch Trennung der Modell- und Instanzebene in unterschiedliche XML-Repräsentationen können, basierend auf einem Modell (bzw. der dazugehörenden `.cmodel`-Datei), beliebig viele Datenstrukturen (bzw. `.cstructure`-Dateien) instanziiert und zugehörige Prozessstrukturen erzeugt werden. Damit eine Modelländerung keine Auswirkung auf bereits erstellte Datenstrukturen hat, enthält jede `.cstructure`-Datei eine vollständige Kopie ihrer zugrunde liegenden Modellebene (entsprechend der Life Cycle Coordination Structure aus Abschnitt 5.3.2).

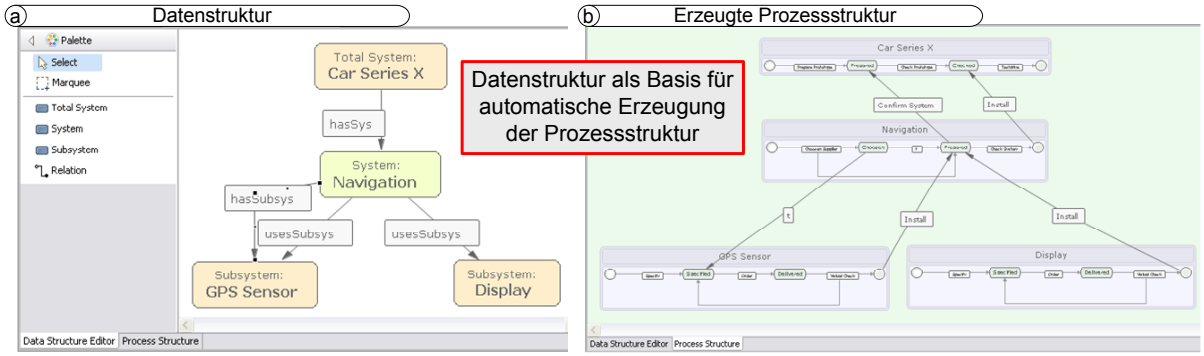


Abbildung 8.5: Automatische Erzeugung einer Prozessstruktur

8.3 Implementierung der operationalen Semantik

Die Realisierung der operationalen Semantik datengetriebener Prozessstrukturen ist eines der zentralen Anliegen, die mit der Implementierung des Demonstrators verfolgt werden. Ziel ist die Simulation erzeugter Prozessstrukturen mittels der in Kapitel 4 definierten Regeln. Dafür muss die vollständige operationale Semantik implementiert werden, ebenso wie die Repräsentation von Laufzeitmarkierungen in der Prozessstruktur. Letztgenannte dienen des Weiteren als Basis für die Implementierung und Prüfung der Konzepte zur Konsistenzanalyse, wie sie im Rahmen der Adaption und Ausnahmebehandlung benötigt werden.

Technische Aspekte

Die technische Umsetzung der operationalen Semantik betrifft alle drei Architekturschichten des Demonstrators (siehe Abbildung 8.1). Während in der Oberfläche (Darstellungsschicht) die Interaktion mit dem Benutzer zur Simulation der Prozessausführung und die Darstellung der Laufzeitmarkierungen realisiert werden, müssen in der Kontrollschicht die Ausführungs- und Markierungsregeln implementiert werden. Die interne Datenrepräsentation (Modellschicht) muss des Weiteren in der Lage sein, die Laufzeitmarkierungen persistent zu speichern. Dafür ist im

Demonstrator das zusätzliche *Interface*[5] `IMiniEngine` (vgl. Algorithmus 8.2) definiert. Es enthält unter anderem Methoden zur Manipulation von Markierungen und Kennzeichnungen. Das Interface wird von denjenigen Java-Objekten implementiert, die Zustände und Transitionen repräsentieren.[6] Kommt es zur Beendigung eines Prozesses, kann die Simulations- bzw. Ausführungskomponente der Kontrollschicht zum einen mittels der Methode `getMarking` die aktuelle Markierung eines Elements auswerten, zum anderen kann sie mittels Methode `setMarking` die Markierung ändern.

```
1  public interface IMiniEngine{
·    public String        getMarking();
·    public void          setMarking(String newMarking);
·
5    public boolean       getFlag(String flagName);
·    public void          setFlag(String flagName, boolean value);
·    public void          addFlag(String flagName);
·    public HashMap<String,Boolean> getFlags();
·
10   public void          propertyChange(PropertyChangeEvent event);
·
·    ...
·  }
```

Algorithmus 8.2: Interface zur Erweiterung der Elemente um Laufzeitinformationen

Die technische Realisierung der Markierungs- und Ausführungsregeln stellt eine große Herausforderung dar. Um den Aufwand für die Realisierung zu minimieren, wurden zunächst für Java verfügbare Ausführungskomponenten zur Interpretation von Regeln, Petri-Netze und Zustandsautomaten auf die Umsetzbarkeit der in Kapitel 4 vorgestellten operationalen Semantik untersucht [NK08]. Eine Anpassung der vorhandenen Ausführungskomponenten zur Realisierung der Markierungs- und Ausführungsregeln von CoRePro wurde jedoch aus verschiedenen Gründen (z.B. Aufwand und schlechte Dokumentation der Ausführungskomponenten) ausgeschlossen. Stattdessen wurden die formal definierten Regeln aus Kapitel 4 in Java realisiert, wobei hier aus Gründen der Überprüfbarkeit auf eine möglichst direkte Umsetzung Wert gelegt wurde. Abbildung 8.6 zeigt als Beispiel die Umsetzung der Ausführungsregel AR1 aus Abschnitt 4.3 in der Implementierung von CoRePro$_{SIM}$.

Benutzersicht

In der grafischen Oberfläche des Simulations-Moduls stehen dem Benutzer Editoren für die Anzeige der Datenstrukturen (zunächst nur lesender Zugriff) und für die Simulation der Prozessstruktur zur Verfügung. Sobald in CoRePro$_{SIM}$ eine Prozessstruktur erzeugt wird (bzw. die entsprechende `.structure`-Datei), kann diese in das Simulations-Modul überführt werden. Dafür wird eine `.csimulation`-Datei angelegt. Sie enthält eine Kopie der Modell- und der Instanzebene und kann damit unabhängig simuliert werden. Beim Start der Simulation initialisiert der Simulator die Markierungen der Elemente entsprechend der Initialisierungsregel IR2 (siehe Seite 87). Anschließend werden die Markierungen der Prozessstruktur durch die implementierten

[5] Um Java-Schnittstellen von allgemeinen Schnittstellen des Demonstrators abgrenzen zu können, wird in dieser Arbeit der Begriff *Interface* als Synonym für *Java-Schnittstelle* verwendet.

[6] In CoRePro$_{SIM}$ wird das Interface auch von denjenigen Java-Objekten, die OLCs und Prozessstrukturen repräsentieren, implementiert. So können OLC- bzw. Prozessstrukturphasen abgebildet werden.

ⓐ Ausführungsregel AR1

Ausführungsregel AR1 (Sequenzabschnitt)

Sei $olc = (P, V, TS) \in \mathcal{OLC}$ ein Object Life Cycle mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Des Weiteren ① sei $s \in seq(olc)$ ein Sequenzzustand (vgl. Definition 4.3) und ② $t^* = (s, (p, v), targ) \in T$ die aus s ausgehende interne Transition. Wenn sich ③ Zustandsmarkierung $SM(s)$ von NotActivated *in* Activated *ändert, wird der Prozess p ausführbar. Es ergibt sich für olc als Folgemarkierung $M' = (SM', TM')$ mit:*

$$TM'(t) = \text{④} \begin{cases} \textsc{Processing} & \text{falls } t = t^* \\ TM(t) & \text{sonst} \end{cases}$$

$$SM' \equiv SM$$

ⓑ Umsetzung von AR1 in der Simulationskomponente

```
public void applyEnactmentRule(EngineStateInstance state, String oldMarking, String newMarking)
{
  ...
③ if (oldMarking.equals(SimulationConstants.MARKING_NotActivated)
    &&   newMarking.equals(SimulationConstants.MARKING_Activated)
    && ① olcA.getSeqs().contains(stateID)) {
    CoreproConsole.out.println("AR1 (Sequenzabschnitt) wird angewendet nach Änderung der
     Markierung von Zustand " + stateID);
④ setIntTransMarkin②(state.getOutgoingElement(), SimulationConstants.MARKING_Processing);
  }
  ...
}
```

Abbildung 8.6: Implementierung der Ausführungs- und Markierungsregeln

Ausführungs- und Markierungsregeln gesetzt. Die Markierungen sind im Demonstrator durch unterschiedlich eingefärbte Elemente dargestellt (vgl. Abbildung 8.7).

Die Interaktion mit dem Benutzer zur Simulation einer Prozessstruktur findet mittels Schaltflächen statt. Sie werden genau dann in die grafische Repräsentation der Prozessstruktur eingeblendet, wenn theoretisch die Methode Start eines Prozesses aufgerufen wird (vgl. Abschnitt 4.3.2). Durch Aktivieren der Schaltfläche kann der Benutzer die Beendigung eines Prozesses „simulieren“. Hat ein Zustand mehrere ausgehende interne Transitionen (mit demselben verknüpften Prozess), kann per Schaltfläche das entsprechende Prozessergebnis und damit der gewählte Pfad festgelegt werden. Ist eine Transition ohne Prozess (bzw. mit dem Stellvertreterprozess τ) modelliert, findet keine Interaktion mit dem Benutzer statt. Stattdessen wird entsprechend der in Abschnitt 4.3.2 vorgestellten Konzepte der Prozesszustand sofort in Completed überführt. Während der Simulation zeigt die Ausgabe (*Console*) die angewendeten Markierungs- und Ausführungsregeln sowie den Verlauf an (vgl. Abbildung 8.7).

Zu Demonstrationszwecken steht dem Benutzer eine *Undo-Funktion* zur Verfügung, mit der schrittweise die jeweils letzte Aktion zurückgenommen bzw. wiederhergestellt werden kann (vgl. Anforderung 8.3). Ihr Mechanismus basiert auf dem Kommando-Muster (*Command Pattern*) aus [GHJV95] und wird als Rahmenwerk von Eclipse zur Verfügung gestellt. Des Weiteren ver-

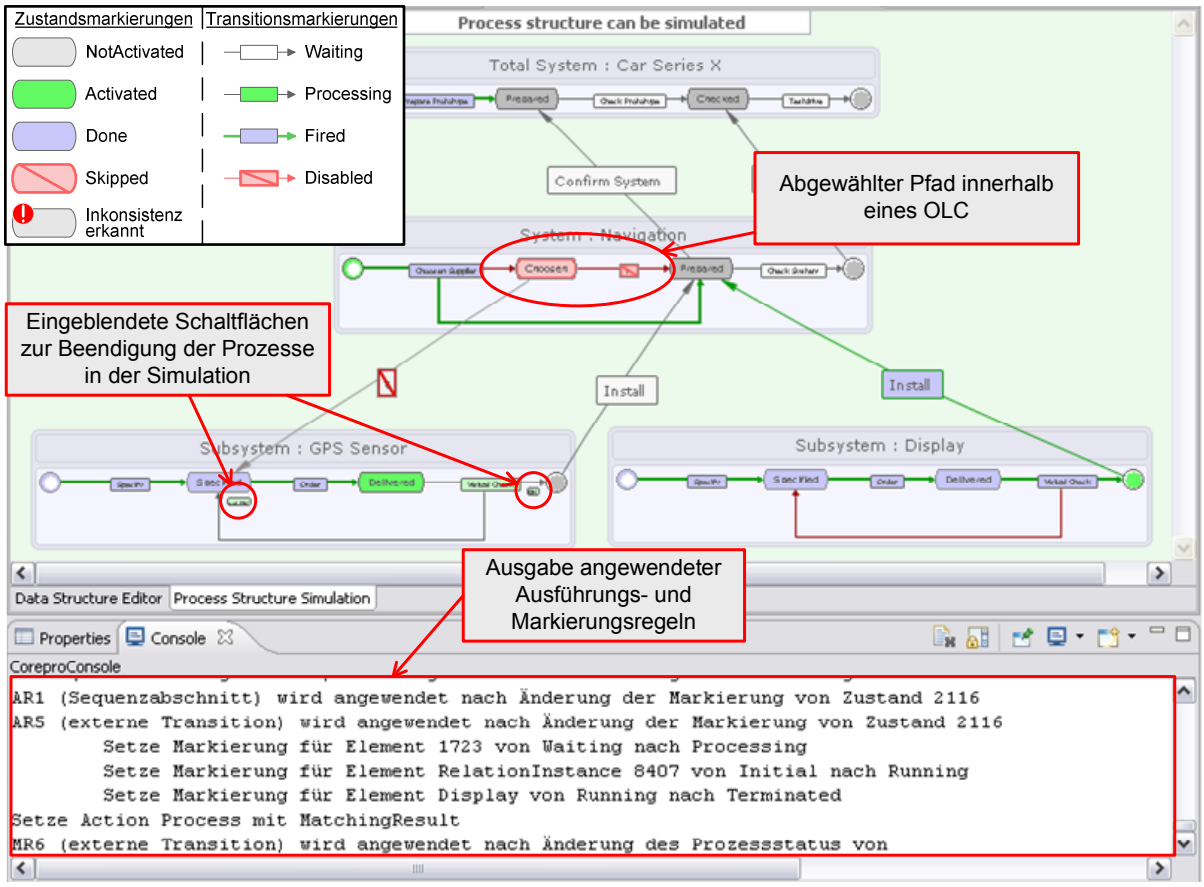


Abbildung 8.7: Simulation der erzeugten Prozessstruktur aus Abbildung 8.5

fügt COREPRO$_{SIM}$ über eine Funktion, mit der sich die Laufzeitmarkierung der Prozessstruktur reinitialisieren und damit die Simulation erneut beginnen lässt.

8.4 Realisierung der Konzepte zur dynamischen Adaption

Die Implementierung der operationalen Semantik in COREPRO$_{SIM}$ ist Grundlage für die Realisierung der Konzepte zur dynamischen Adaption von Prozessstrukturen. Herausforderungen sind zum einen die Transformation von Änderungen der Datenstruktur in Adaptionen der Prozessstruktur und zum anderen die Konsistenzanalyse nach Durchführung von Änderungen. Für letzteres ist insbesondere die Repräsentation der Laufzeitmarkierungen wichtig, um die Konsistenzanalyse im Rahmen dynamischer Änderungen prüfen und demonstrieren zu können.

Technische Aspekte

Die Änderung einer Datenstruktur wird in COREPRO$_{SIM}$ über einen Benachrichtigungsmechanismus (vgl. *Observer* Muster in [GHJV95]) an Elemente der Prozessstruktur übermittelt und

damit in Operationen zur Adaption der Prozessstruktur transformiert. Die Konsistenzprüfung nach Durchführung einer Änderung erfolgt durch die Methode `checkConsistency` (vgl. Algorithmus 8.3). Sie bekommt eine externe Transition übergeben (vgl. Zeile 4 in Algorithmus 8.3) und erzeugt auf Basis ihrer Markierung sowie den Markierungen ihres Quell- und Zielzustands ein Java-Objekt, das die Änderungsregion repräsentiert (vgl. Zeile 10 und 11 in Algorithmus 8.3). Auf Basis dieses Java-Objekts wird geprüft, ob die Markierungen der Änderungsregion konsistent sind (vgl. Zeile 13 in Algorithmus 8.3). Enthält die Änderungsregion eine inkonsistente Laufzeitmarkierung, wird das entsprechende Element (die externe Transition oder ihr Zielzustand) als `INCONSISTENT` gekennzeichnet (vgl. Zeilen 18 und 24 in Algorithmus 8.3).

```
1  protected boolean checkConsistency(IMiniEngine element)
·  {
·    // wenn ext. Transition eingefügt, prüfe deren Change Region
·    if (element.getType().equals( StructureConstants.XML_ELEMENT_NAME_EXT_TRANS_INSTANCE) &&
5          element.getFlag(SimulationConstants.FLAG_CREATED)) {
·      EngineExtTransitionInstance extTrans = (EngineExtTransitionInstance) element;
·      EngineStateInstance srcState = (EngineStateInstance) extTrans.getIncomingElement().get(0);
·      EngineStateInstance targState = (EngineStateInstance) extTrans.getOutgoingElement().get(0);
·
10     ChangeRegion checkCR = new ChangeRegion(srcState.getMarking(), extTrans.getMarking(),
·          targState.getMarking());
·
·      if (invalidChangeRegionsCreated.containsKey(checkCR)) {
·        ChangeRegionInconsistency incCR = invalidChangeRegionsCreated.get(checkCR);
15       if (incCR.isTransInc()) {
·          CoreproConsole.out.println("Setze Inkonsistenz-Kennzeichnung für externe Transition " +
·                 extTrans.getID());
·          extTrans.setFlag(SimulationConstants.FLAG_INCONSISTENT, true);
·          extTrans.setFlag(SimulationConstants.FLAG_IGNORE, false);
20       }
·        if (incCR.isTargInc()) {
·           CoreproConsole.out.println("Setze Inkonsistenz-Kennzeichnung für Zustand " +
·                 targState.getID());
·           targState.setFlag(SimulationConstants.FLAG_INCONSISTENT, true);
25         targState.setFlag(SimulationConstants.FLAG_IGNORE, false);
·          setErrorPathFlag(targState, true);
·        }
·      }
·    ...
```

Algorithmus 8.3: Realisierung der Konsistenzanalyse in COREPRO$_{SIM}$

Tritt während der dynamischen Adaption der Prozessstruktur eine Inkonsistenz auf, kann diese in bestimmten Fällen automatisch behandelt werden (vgl. Abschnitt 6.3.4). Daher wird in COREPRO$_{SIM}$ nach Abschluss einer Änderungsoperation der Mechanismus zur automatischen Ausnahmebehandlung bzw. zur Neubewertung der Markierungen angestoßen (vgl. Abbildung 8.8). Wird die Operation beispielsweise für eine als inkonsistent markierte externe Transition e ausgeführt (vgl. ① in Abbildung 8.8), muss geprüft werden, ob e als `WAITING` markiert und ob der Quellzustand von e bereits als `ACTIVATED` oder `DONE` gekennzeichnet ist (vgl. ② in Abbildung 8.8). Weiter darf der Zielzustand für e nicht als `SKIPPED` markiert sein. Wenn die Bedingung ② aus Abbildung 8.8 zutrifft, wird die externe Transition e als `PROCESSING` gekennzeichnet (vgl. ② in Abbildung 8.8). Trifft Bedingung ③ in Abbildung 8.8 zu, muss e als `DISABLED` markiert werden. Bei Anwendung der automatischen Ausnahmebehandlung auf Zustände wird geprüft, ob der Zustand aktiviert werden muss (z.B. bei Hinzufügen eines neuen OLC oder bei Entfernen einer externen Transition; nicht in Abbildung 8.8 enthalten).

a) Markierungsregel MR9

Markierungsregel MR9 (Neubewertung der Markierung externer Transitionen)
Sei $ps = (OLC, EST)$ eine Prozessstruktur mit aktueller Markierung $PSM = (OM, EM)$. Wird eine externe Transition ① $e^* = (s_1, p, s_2) \in EST$ als $EF(e^*) = \textsc{Inconsistent}$ gekennzeichnet, dann ergibt sich die Folgemarkierung für $PSM' = (OM', EM')$:

$$EM'(e) = \begin{cases} \textsc{Processing} & \text{② falls } e = e^* \wedge EM(e) = \textsc{Waiting} \wedge SM(s_1) \in \{\textsc{Activated}, \textsc{Done}\} \\ & \wedge\ SM(s_2) \neq \textsc{Skipped} \\ \textsc{Disabled} & \text{③ falls } e = e^* \wedge EM(e) = \textsc{Waiting} \\ & \wedge\ (SM(s_1) = \textsc{Skipped} \vee SM(s_2) = \textsc{Skipped}) \\ EM(e) & \text{sonst} \end{cases}$$

$$OM' \equiv OM$$

b) Umsetzung von MR9 in der Simulationskomponente

```
protected String reCalculateMarking(IMiniEngine object) {
① if (object instanceof EngineExtTransitionInstance) {
    EngineStateInstance src = (EngineStateInstance) ((EngineExtTransitionInstance)
        object).getIncomingElement().get(0);
    EngineStateInstance targ = (EngineStateInstance) ((EngineExtTransitionInstance)
        object).getOutgoingElement().get(0);
② if ( (src.getMarking().equals(SimulationConstants.MARKING_Activated)
            || src.getMarking().equals(SimulationConstants.MARKING_Done) )
            && !targ.getMarking().equals(SimulationConstants.MARKING_Skipped)
            && object.getMarking().equals(SimulationConstants.MARKING_Waiting) {
        return SimulationConstants.MARKING_Processing;
    }
③ if ( (src.getMarking().equals(SimulationConstants.MARKING_Skipped)
        || targ.getMarking().equals(SimulationConstants.MARKING_Skipped) )
        && object.getMarking().equals(SimulationConstants.MARKING_Waiting) {
        return SimulationConstants.MARKING_Disabled;
    }
  }
  ...
```

Abbildung 8.8: Implementierung der Regel zur Neubewertung der Markierung

Benutzersicht

Die Realisierung der Konzepte für dynamische Adaptionen datengetriebener Prozessstrukturen wird aus Benutzersicht durch eine Erweiterung der grafischen Oberfläche für die Simulation erreicht. Es stehen dem Benutzer weiterhin der Editor für die Datenstruktur und die Simulation der Prozessstruktur zur Verfügung. Der Editor für die Datenstruktur erlaubt nun schreibenden Zugriff und damit die Änderung der Datenstruktur. So können vor oder während der Simulation einer Prozessstruktur dynamische Adaptionen an ihr vorgenommen werden. Entsprechend der Konzepte aus Kapitel 6 werden Änderungen der Datenstruktur in Adaptionen der Prozessstruktur überführt (vgl. Abbildung 8.9a und 8.9b). Die Änderungsregionen werden sofort auf Inkonsistenzen überprüft und ggf. inkonsistente Elemente (visuell) gekennzeichnet (vgl. Abbildung 8.9b). Die Änderungsoperationen wiederum sind in Änderungstransaktionen eingebettet. Die Änderungstransaktion wird erzeugt, sobald eine Änderungsoperation angewendet wird. In diese Änderungstransaktion werden alle weiteren Änderungsoperationen eingebettet, bis die Transaktion abgeschlossen oder verworfen wird. Der implementierte Mechanismus verhindert das Abschließen einer Transaktion bei existierenden Inkonsistenzen. Dann muss die Transaktion verworfen oder die Inkonsistenz durch Anwendung weiterer Änderungsoperationen beseitigt werden.

Wird die Änderungstransaktion verworfen, werden die ursprüngliche Daten- und Prozessstruktur wieder hergestellt. Während einer Änderungstransaktion ist das Speichern der Prozessstruktur nicht erlaubt.

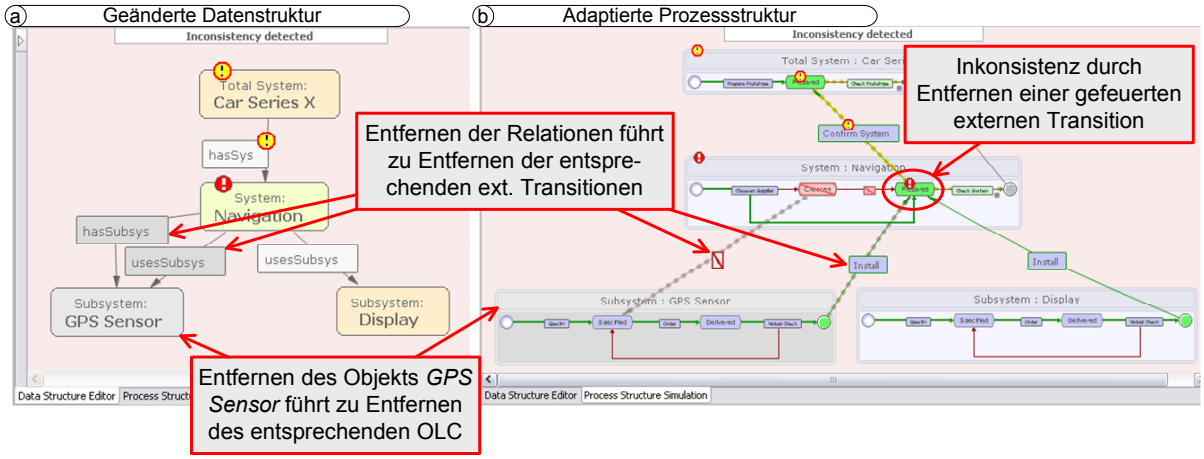


Abbildung 8.9: Dynamische Adaption einer datengetriebenen Prozessstruktur

8.5 Realisierung der Konzepte zur Ausnahmebehandlung

Basierend auf der Realisierung des Simulations-Moduls (vgl. Phase 3 in Abbildung 8.1) und der Integration der Operationen zur Adaption von Prozessstrukturen (insbesondere der Konsistenzanalyse in Phase 4; vgl. Abbildung 8.1) können die Konzepte zur Ausnahmebehandlung realisiert werden. Herausforderungen sind hier unter anderem die Implementierung der Sprungoperation sowie die Analysen zur Vorhersage von Inkonsistenzen.

Technische Aspekte

Im Rahmen der Ausnahmebehandlung ist die Sprungoperation ein wichtiger Mechanismus. Er wurde bereits ausführlich in Abschnitt 7.2.1 behandelt. Für die Realisierung der temporären Sprungelemente wird in CoreProSIM eine Java-Klasse für temporäre Zustände und Transitionen realisiert, die in der Implementierung die Eigenschaften der Standard-Elemente erbt. Wie in Abschnitt 7.2.1 beschrieben, ist auch in CoreProSIM der Sprung innerhalb eines OLC in *einer* Methode ohne Fallunterscheidung der Sprungrichtung realisiert (vgl. Algorithmus 8.4). Diese Methode erhält als Übergabeparameter den Zielzustand des Sprungs (vgl. Zeile 1 in Algorithmus 8.4). Im ersten Schritt wird der aktuell aktivierte Zustand ermittelt und dessen ausgehende interne Transitionen erfasst (vgl. Zeile 3-8 in Algorithmus 8.4).[7] In den Zeilen 10-18 in Algorithmus 8.4 werden die Sprungkanten eingefügt und in den Zeilen 20-23 konfiguriert. Abschließend wird in Zeile 25 die temporäre Transition `src_trans` gefeuert. Sobald der Sprungzustand `jumpState` als

[7] Das Java-Objekt zur Repräsentation interner Transitionen fasst mehrere aus einem Zustand ausgehende interne Transitionen zusammen. Daher wird in Zeile 7 in Algorithmus 8.4 nur ein einzelnes Element und keine Menge als Ergebnis (d.h. ausgehende interne Transition) geliefert.

DONE markiert wird, der Sprung also vollständig durchgeführt worden ist, werden die Transitionen wieder entfernt (nicht in Algorithmus 8.4 enthalten).

```
1  public void performJump(CSStateInstance target)
   {
     ...
     CSStateInstance source = olc.getStateWithMarking(SimulationConstants.MARKING_Activated);
5
     ...
     CSIntTransitionInstance sourceOutTrans = (CSIntTransitionInstance)
         source.getOutgoingElement().get(0);

10   MaskCSStateInstance jumpState = (MaskCSStateInstance)
         olc.addNewElement(SimulationConstants.XML_ELEMENT_NAME_STATE_INSTANCE,
         SimulationConstants.MARKING_NotActivated, null, null);
     MaskCSTargetInstance src_trans = (MaskCSTargetInstance)
         olc.addNewElement(SimulationConstants.XML_ELEMENT_NAME_TARGET_INSTANCE,
15       SimulationConstants.MARKING_Processing, sourceOutTrans, jumpState);
     MaskCSIntTransitionInstance targ_trans = (MaskCSIntTransitionInstance)
         olc.addNewElement(SimulationConstants.XML_ELEMENT_NAME_INT_TRANSITION_INSTANCE,
         SimulationConstants.MARKING_Waiting, jumpState, target);

20   jumpState.setName("JumpTo_" + target.getName());
     src_trans.setName("-");
     targ_trans.setName("Compensation");
     targ_trans.setProcessType(ModelConstants.EMPTY_PROCESS);

25   src_trans.setAction(SimulationConstants.ACTION_Process, new String[]
         {SimulationConstants.ACTION_Process_None});
   }
```

Algorithmus 8.4: Realisierung der Sprungoperation

Die Kennzeichnung erwarteter Inkonsistenzen ist ein weiteres wichtiges Konzept im Rahmen der COREPRO-Ausnahmebehandlung, insbesondere im Umgang mit großen Prozessstrukturen. Die Regeln KR5 und KR6 aus Abschnitt 7.3.2 beschreiben die Kennzeichnung erwarteter Inkonsistenzen. Um eine geeignete Visualisierung der erwarteten Inkonsistenzen im Demonstrator zu erreichen, soll zusätzlich zu den in Abschnitt 7.3.2 vorgestellten Kennzeichnungen von Zuständen und externen Transitionen, eine visuelle Kennzeichnung derjenigen *internen* Transitionen und Zustände vorgenommen werden, die durch erwartete Inkonsistenzen ebenfalls betroffen sind. Anstatt durch die Prozessstruktur zu traversieren, kann der Mechanismus auch unter Nutzung des Beobachter-Musters (*Observer Pattern*) aus [GHJV95] rekursiv realisiert werden. Dafür ist in COREPRO$_{SIM}$ ein so genannter *PropertyChange*-Mechanismus implementiert. Hier registrieren sich diejenigen Java-Objekte, die Zustände und Transitionen der Prozessstruktur repräsentieren, bei ihren jeweiligen Nachbar-Elementen. Bezogen auf einen Zustand s sind das alle in s ein- und aus s ausgehenden internen und externen Transitionen. Wird also ein Element als INCONSISTENT oder PREDICTEDINCONSISTENT gekennzeichnet, werden alle Nachbarn des betroffenen Elements über den Benachrichtigungsmechanismus informiert (vgl. Zeilen 1 und 5 in Algorithmus 8.5). Auf Basis der aktuellen Markierung wird dann individuell entschieden, ob auch das informierte Element als PREDICTEDINCONSISTENT gekennzeichnet werden muss (vgl. Zeile 12 in Algorithmus 8.5).[8] Ist dies der Fall, werden abermals der Benachrichtigungsmechanismus und eine Prüfung auf erwartete Inkonsistenzen bei den direkten Nachbarn ausgelöst.

[8] In Algorithmus 8.5 werden entsprechend der Konzepte aus Abschnitt 7.3.2 erwartete Inkonsistenzen bei Zuständen und externen Transitionen durch die Konstante FLAG_PRED_INCONSISTENT ausgedrückt. Zusätzlich wird in COREPRO$_{SIM}$ für die Visualisierung des „vollständigen“ Pfades erwarteter Inkonsistenzen, also auch von internen Transitionen und Zuständen, die Konstante FLAG_PRED_INCONSISTENT_PATH genutzt.

```
1  public void propertyChange(PropertyChangeEvent event)
·  {
·    ...
·
5    if(strPropertyName.equals(FLAG_CHANGED))
·    {
·      ...
·      if( ((String)event.getNewValue()).equals(SimulationConstants.FLAG_INCONSISTENT)
·       || ((String)event.getNewValue()).equals(SimulationConstants.FLAG_PRED_INCONSISTENT)
10     || ((String)event.getNewValue()).equals(SimulationConstants.FLAG_PRED_INCONSISTENT_PATH))
·      {
·        engine.calculatePredictedInconsistency(this);
·      }
·    ...
15 }
```

Algorithmus 8.5: Mechanismus zur Kennzeichnung erwarteter Inkonsistenzen

Benutzersicht

Die Konzepte zur Ausnahmebehandlung sind ebenfalls in das Simulations-Modul integriert. Es ist damit möglich, während der Simulation die in Kapitel 7 vorgestellten Markierungsoperationen zu nutzen. Sie können zum einen bei Auftreten einer Inkonsistenz angewendet werden, zum anderen aber auch genutzt werden, um *reale* Ausnahmen ad hoc zu bearbeiten (z.B. mittels Operationen JUMP oder RESET; vgl. Abbildung 8.10a). Tritt im Rahmen der Adaption der Prozessstruktur oder Anwendung einer Markierungsoperation eine Inkonsistenz auf, werden entsprechend der Konzepte aus Abschnitt 7.3.2 die erwarteten Inkonsistenzen visuell gekennzeichnet (vgl. Abbildung 8.10b). Durch Anwendung der Markierungsoperationen (z.B. RESET) können Ausnahmen behandelt werden (vgl. Abbildung 8.10c). Ebenfalls können Inkonsistenzen entsprechend der Konzepte aus Abschnitt 7.4.1 gezielt ignoriert werden, um eine Änderungstransaktion abschließen zu können (vgl. Abbildung 8.10d). Die Ausführung bzw. Simulation der Prozessstruktur kann dann fortgesetzt werden.

8.6 Realisierung der Ausführungskomponente

Im Rahmen der technischen Validation der Konzepte wollen wir uns nicht nur auf die Simulation beschränken. Vielmehr ist ein weiteres Ziel die Koordination konkreter Prozesse, die beispielsweise in Prozess-Management-Systemen ausgeführt werden. Hierzu werden generische Schnittstellen und technische Konzepte für die Steuerung der extern ausgeführten Prozesse benötigt. Wir müssen zudem den Anforderungen der Koordination realer Prozesse an die Ausführungskomponente (z.B. unerwarteter Prozessabbruch) Rechnung tragen.

Technische Aspekte

Die Koordination realer Prozesse erfordert die Definition einer geeigneten Schnittstelle, die die benötigten Informationsflüsse abbildet (vgl. Abbildung 8.11a). Dazu gehört die Steuerung und Überwachung der extern ausgeführten Prozesse genauso wie die Möglichkeit auf die öffentlichen Schnittstellen zur Adaption und Ausnahmebehandlung zuzugreifen. Abbildung 8.11b zeigt die logische Schnittstelle zur Ausführung von externen Prozessen. Sie unterscheidet die *Java-Interfaces*

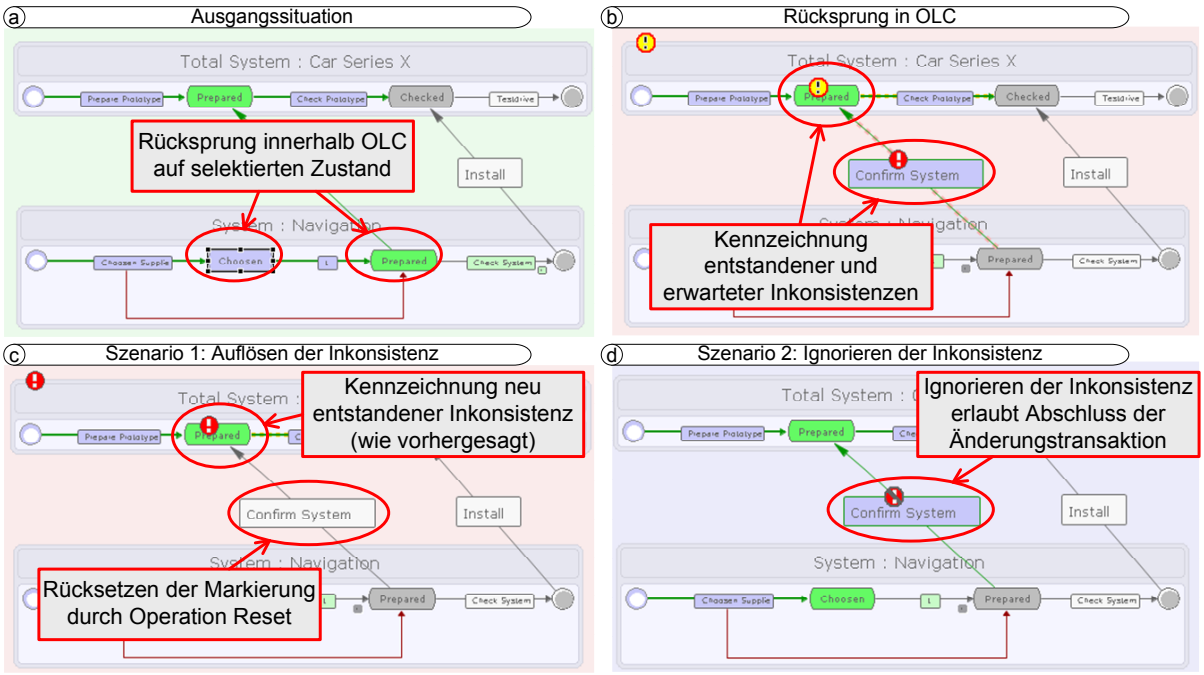


Abbildung 8.10: Darstellung der Konzepte zur Ausnahmebehandlung

`ProcessHolder` und `Process`. Das *Interface* `Process` enthält die zur Steuerung von Prozessen benötigten Methoden (START und CANCEL; vgl. Abschnitt 4.3.2). Das Interface `ProcessHolder` hingegen weist die Methode `submitProcessResult` auf, die von den Prozessen aufgerufen werden kann, um die Beendigung des Prozesses (mit zugehörigem Ergebnis) anzuzeigen (vgl. Abbildung 8.11b).[9] Die Methode `submitProcessResult` kann vom aufrufenden Prozess auch dazu genutzt werden, eine Ausnahme bei der Prozessausführung mitzuteilen. Jeder Prozess (also Java-Objekte vom Typ *Process*) wird durch seinen *ProcessHolder* (d.h. eine interne oder externe Transition) entsprechend der Regeln aus Abschnitt 4.3.2 instanziiert und über den in Java verfügbaren *Thread* Mechanismus gestartet. Die konkrete Implementierung der Methoden des Interface `Process` ist dem Prozess freigestellt, d.h. bei Ausführung der CANCEL Methode muss der Prozess nicht sofort abgebrochen, sondern es können beispielsweise noch Kompensationsaktivitäten durchgeführt werden.[10] Das Interface `Process` kann auch genutzt werden, um beispielsweise *Adapter* (vgl. [GHJV95]) zur Steuerung von Prozessen, die unterschiedliche Schnittstellen aufweisen, zu realisieren. Derartige Adapter können auch die Ausführung von Prozessen steuern, deren Metamodelle feingranulare Prozesszustände definieren (z.B. Unterscheidung von Start und Aktivierung eines Prozesses in ADEPT [RD98]).

Die Schnittstelle der Ausführungskomponente ist bewusst offen gestaltet. So können Prozesse nicht nur das Objekt ihres OLCs bearbeiten, sondern auch weitergehende Informationen zur Analyse der Daten- bzw. Prozessstruktur erhalten (z.B. Relationen zu anderen Objekten). In Kapitel 9 werden Anwendungsfälle für die Nutzung dieser Schnittstellen vorgestellt.

[9] Logische Darstellung der Methode. In der technischen Realisierung sind für den Aufruf weitere Parameter (z.B. Prozess-ID für die Zuordnung des Ergebnisses) notwendig.

[10] Wichtig ist, dass der Prozess im Sinne der Prozessstruktur als abgebrochen gilt und damit die Terminierung der Prozessstruktur weiterhin Gültigkeit besitzt.

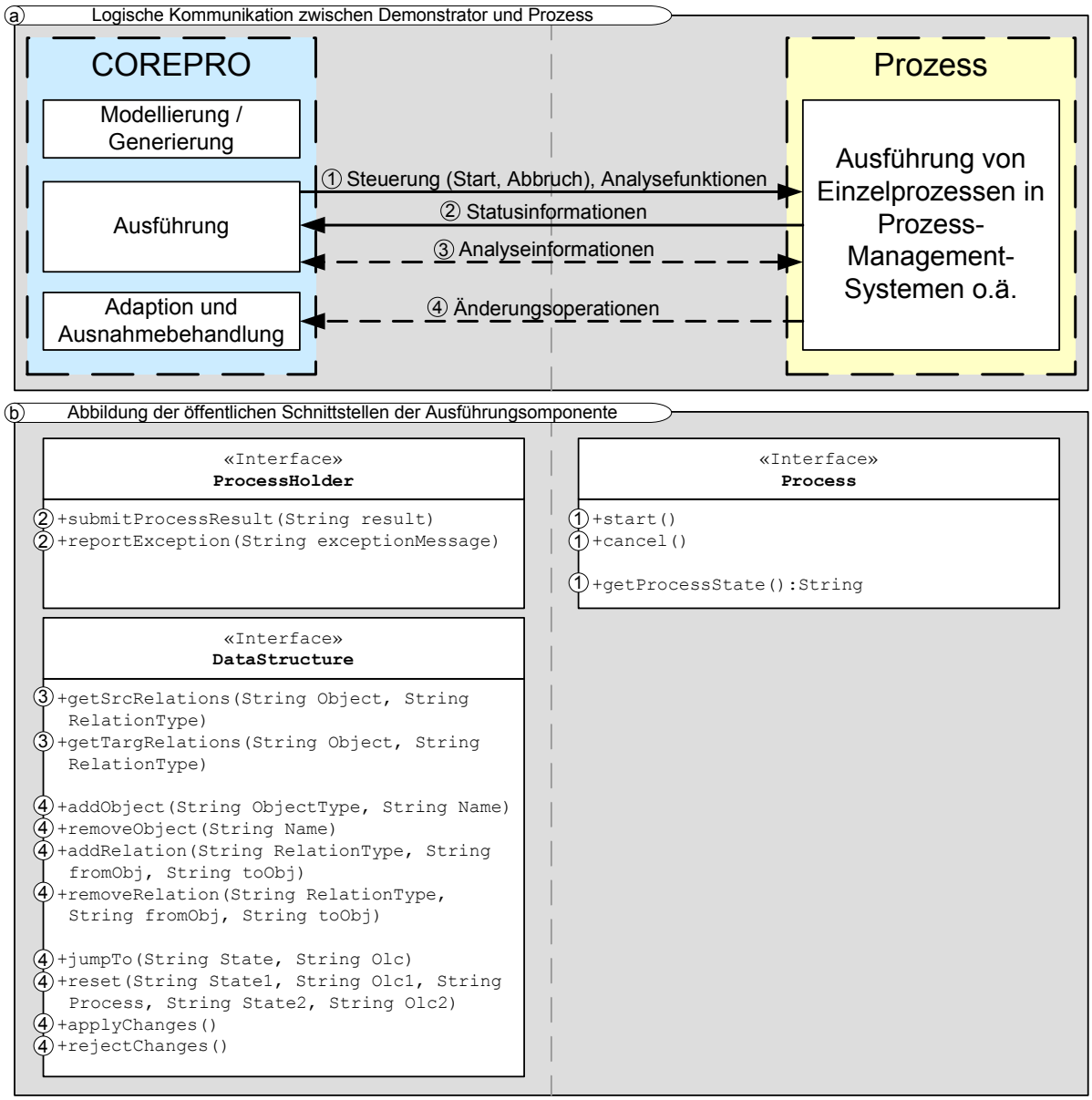


Abbildung 8.11: Architektur der Ausführungskomponente in COREPRO$_{SIM}$

Benutzersicht

Für die Realisierung der Ausführungskomponente haben wir die grafische Oberfläche von COREPRO$_{SIM}$ um die dafür notwendigen Funktionen erweitert. Im Rahmen der Modellierung interner und externer Transitionen können konkrete Prozesse verknüpft werden. Es sind verschiedene Beispielprozesse unter Nutzung des Interface `Process` implementiert (z.B. Java-Anwendung mit grafischer Oberfläche sowie Web-Services). Kommt die Prozessstruktur in COREPRO$_{SIM}$ zur Ausführung und eine Transition mit verknüpftem Prozess wird als `PROCESSING` markiert, wird anstatt der Benutzerinteraktion der Simulation der jeweils mit der Transition, also dem `ProcessHolder`,

verknüpfte Prozess in einem eigenen *Thread* gestartet. Die Prozesse werden damit nebenläufig als „Black-Box“ ausgeführt und teilen nach Beendigung dem `ProcessHolder` das jeweilige Prozessergebnis mit (vgl. Methode `submitProcessResult` in Abbildung 8.11b). Der `ProcessHolder` veranlasst, wie in der Simulation, entsprechend der implementierten operationalen Semantik eine Markierungsänderung. Die aus COREPRO$_{SIM}$ bekannten Simulations-Oberflächen können weiterhin für die Überwachung der Prozessstruktur sowie für die Adaption und Ausnahmebehandlung verwendet werden.

8.7 Zusammenfassung und Diskussion

Mit der Realisierung des Demonstrators COREPRO$_{SIM}$ haben wir gezeigt, dass sich die in Teil II dieser Arbeit vorgestellten Konzepte technisch umsetzen lassen. Die technische Realisierung der Konzepte wurde in zwei wissenschaftlichen Veröffentlichungen dokumentiert und im Rahmen der WETICE'07- und der BPM'08-Konferenzen dem internationalem Fachpublikum vorgestellt [MRHP07, MRH$^+$08b].

Der vorgestellte Demonstrator COREPRO$_{SIM}$ befindet sich weiterhin in der Entwicklung. Er wird genutzt, um auf dieser Arbeit aufbauende Konzepte zu prüfen und deren Transfer in die Praxis vorzubereiten. Dazu gehört beispielsweise die Bereitstellung benutzerfreundlicher grafischer Oberflächen. Die in den Abbildungen gezeigten Oberflächen stellen nur eine mögliche Ausprägung der Benutzeroberflächen dar. Hier gilt es nicht nur große Prozessstrukturen für den Endbenutzer verständlich und intuitiv darzustellen, sondern auch die Interaktion mit dem Benutzer (z.B. im Rahmen der Ausnahmebehandlung) übersichtlich zu gestalten. Dafür werden Konzepte benötigt, die Ausschnitte, Abstraktionen sowie daten- und prozesszentrierte Sichten auf datengetriebene Prozessstrukturen darstellen können und dabei Aspekte wie Benutzerfreundlichkeit und Datensicherheit berücksichtigen. Eine weitere Herausforderung ist die Implementierung von Prozess-Adaptern für Altsysteme (sog. *Legacy Systeme*), um weitere Systeme anzubinden. Hierzu sind jedoch weniger neue Prozess-Management-Konzepte notwendig als die Nutzung vorhandener Middleware-Technologien.

9 Praktische Anwendung

In diesem Kapitel sollen die erarbeiteten Konzepte anhand einer Fallstudie validiert werden. Wir wollen so die Anwendbarkeit des Ansatzes zur Unterstützung datengetriebene Prozessstrukturen in der Praxis (vgl. Abschnitt 2.1) prüfen. Zunächst zeigen wir die Anwendung der CoPro-Konzepte für den standardisierten Entwicklungsprozess einer ISO-Norm, welche die Entwicklung sicherheitskritischer elektronischer Komponenten im Automobilbereich beschreibt. Weiter haben wir eine Fallstudie zur Definition eines zukünftigen E/E-Release-Management-Prozesses durchgeführt, daraus relevante Teilprozesse extrahiert und deren Abbildung mit den CoPro-Konzepten (d.h. mit dem in Kapitel 8 vorgestellten Demonstrator CoPro_{SIM}) untersucht.

Das Kapitel gliedert sich wie folgt: Abschnitt 9.1 führt die beiden zur Validation relevanten Szenarien, den Prozessstandard der ISO 26262 und den E/E-Release-Management-Prozess, ein. Abschnitt 9.2 stellt die Umsetzung des ISO-Prozessstandards mit den CoPro-Konzepten und die Instanziierung konkreter Prozessstrukturen vor. Abschnitt 9.3 diskutiert die Abbildung verschiedener Szenarien des E/E-Release-Management-Prozesses ohne dynamische Konfigurationsbildung (d.h. ohne dynamische Adaption der Prozessstruktur). Abschnitt 9.4 erweitert diese Abbildung um Aspekte der dynamischen Konfigurationsbildung (d.h. mit dynamischer Adaption der Prozessstruktur). Weitergehende Aspekte bei der Modellierung und Ausführung von Prozessstrukturen mit den CoPro-Konzepten beschreibt Abschnitt 9.5. Abschnitt 9.6 gibt eine Zusammenfassung des Kapitels und schließt damit Teil III dieser Arbeit ab.

9.1 Einleitung

Die Basis für die Validation der Konzepte dieser Arbeit bilden zwei unterschiedliche Szenarien. Das erste Szenario ist der zukünftige Prozessstandard *ISO 26262*, der Entwicklungsprozesse sicherheitskritischer E/E-Komponenten definiert. Das zweite Szenario wurde im Rahmen einer

Fallstudie in der E/E-Entwicklung erarbeitet. Dazu wurde der aktuelle E/E-Release-Management-Prozess (vgl. Abschnitt 2.1) umfangreich dokumentiert und ein zukünftiger Prozess abgeleitet [HMP08]. Die beiden Szenarien unterscheiden sich insofern, als dass der Prozessstandard das allgemeine Vorgehen bei der Entwicklung der Elektronik in Fahrzeugen definiert, während der E/E-Release-Management-Prozess sehr detailliert einen Teilaspekt, die Releasebildung und -absicherung, beschreibt.[1]

9.1.1 ISO 26262 – Funktionale Sicherheit von Kraftfahrzeugen

Die *ISO 26262* (*Road Vehicles - Functional Safety*) ist eine zukünftige ISO-Norm für sicherheitskritische E/E-Systeme in Kraftfahrzeugen.[2] Sie definiert ein Prozess-Rahmenwerk das von Automobilherstellern bei der Entwicklung von E/E-Systemen angewendet werden soll, um zu dokumentieren, dass die funktionale Sicherheit der Systeme durch geeignete Entwicklungsprozesse sichergestellt ist [Int08, Jun08].

Ziel des Standards ist die Handhabung der Komplexität sicherheitskritischer Systeme durch deren Zerlegung in Subsysteme. Das erlaubt die feingranulare Definition der Entwicklungsprozesse durch verschiedene *Integrationsstufen* (vgl. *Subsystem B1* in Abbildung 9.1). Die ISO 26262 besteht aus neun Teilen, von denen wir im Folgenden die Anläufe der Produktentwicklung auf System- (Teil 4), Hardware- (Teil 5) und Softwareebene (Teil 6) betrachten. Teil 4 integriert die standardisierten Entwicklungsprozesse auf Hardware- und Softwareebene (vgl. Abbildung 9.1). Das Ergebnis der Entwicklungsphase auf Systemebene ist die Systemstruktur (vgl. Schritte 4-4 bis 4-6 in Abbildung 9.1). Abhängig von der Komplexität des Gesamtsystems, muss die Entwicklungsphase rekursiv für Subsysteme wiederholt werden [Int08]. Die konkreten Schritte innerhalb der einzelnen Integrationsstufen können als isolierte Prozesse betrachtet werden; ihre fachliche Bedeutung ist im Rahmen unserer Validation nicht von Belang. Nach Abschluss der Entwicklung von Hardware und Software werden diese in Subsystemen bis hin zum Gesamtfahrzeug *Bottom-up* integriert und getestet (vgl. Schritte 4-7.4.1, 4-7.4.2 und 4-7.4.3 in Abbildung 9.1).

9.1.2 Der E/E-Release-Management-Prozess

Das Release-Management in der E/E-Entwicklung hat das Ziel, zu definierten Zeitpunkten Entwicklungsstände des E/E-Gesamtsystems *einzufrieren* und systematisch abzusichern (vgl. Abschnitt 2.1). Es umfasst die Phasen *Konfigurationsbildung*, *Absicherung* und *Freigabe* (vgl. Abbildung 2.1 auf Seite 13).

Die Aktivitäten der Konfigurationsbildung erzeugen die Systemstruktur des Release, welches anschließend abgesichert und freigegeben wird. Die Konfigurationsbildung stellt im Hinblick auf die IT-Unterstützung besondere Anforderungen, denn ihr Ziel ist die dynamische Erstellung und Änderung einer Systemstruktur, also die Auswahl der zu einem Release gehörenden Komponenten

[1] Die beiden Szenarien wurden fachlich isoliert betrachtet. Es ist kein Ziel dieser Arbeit, den E/E-Release-Management-Prozess zu verändern und ihn konform zur ISO 26262 Norm zu beschreiben.

[2] Die ISO-Norm befindet sich in der Entwicklung und liegt derzeit im Status *Committee Draft* vor. Sie soll Ende 2010 offiziell verabschiedet werden [Jun08].

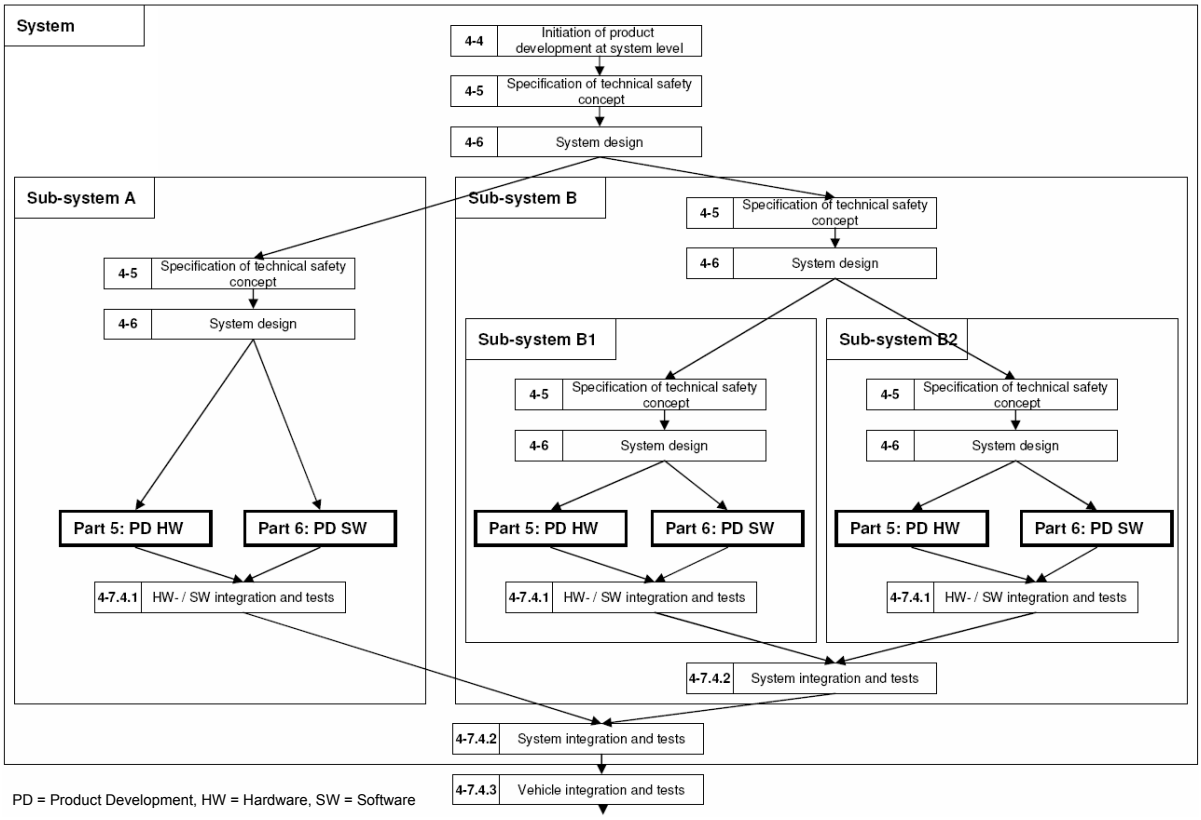


Abbildung 9.1: Teil 4 der ISO26262 (Produktentwicklung auf Systemebene) [Int08]

[MHHR06]. Wird beispielsweise ein Release für die Absicherungsfahrzeuge der Wintererprobung[3] erstellt, muss zunächst entschieden werden, welche Komponenten bzw. welche Komponentenvarianten (z.B. *einfache Klimaanlage* oder *Mehr-Zonen Klimaautomatik*) in den Fahrzeugen verbaut werden und in dem Release enthalten sein müssen. Auf Basis dieser Auswahl werden die Komponentenentwickler aufgefordert, für die jeweilige *Komponente* bzw. *Komponentenvariante* eine geeignete *Komponentenversion* zu *melden* (vgl. Abbildung 9.2). Die gemeldete Komponentenversion muss nicht zwangsläufig die aktuellste Version sein, sondern wird beispielsweise auf Basis des aktuellen Reifegrads und der implementierten Funktionen ausgewählt. Für jede Komponente bzw. Komponentenvariante muss genau eine Version gemeldet werden.

Komponenten können auch in Teilsysteme (Systeme) gekapselt werden (vgl. Abbildung 9.2). Damit lassen sich Komponenten funktional bündeln und absichern, bevor sie in ein Release aufgenommen werden. Auf Teilsystemebene muss dann wiederum eine Konfigurationsbildung durchgeführt und die entsprechenden Komponentenversionen gemeldet werden. Anschließend werden die Teilsysteme (wie Komponenten) in ein Release aufgenommen. Das Vorgehen bei der Integration und Absicherung eines Release entspricht dabei der rechten Seite des V-Modells (vgl. Abschnitt 2.1).

[3]Die Wintererprobung fasst Testfahrten zusammen, die unter winterlichen klimatischen Bedingungen gemacht werden. Aus elektronischer Sicht sind sie u.a. für die Fahrdynamikregelungen (z.B. ESP), die Motorsteuerung und die Klimaautomatik relevant.

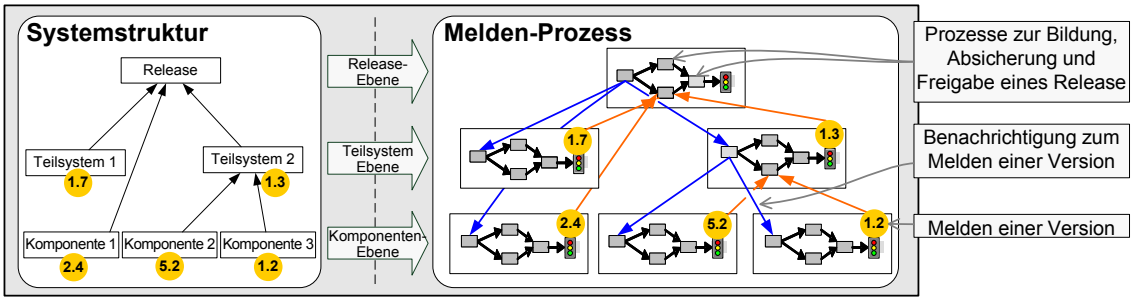


Abbildung 9.2: Illustrative Darstellung des Melden-Prozesses

9.2 Abbildung des Prozessstandards aus Teil 4 der ISO 26262

Die ISO 26262 definiert einen Standard-Entwicklungsprozess für sicherheitskritische E/E-Systeme. Die Norm fordert die Zerlegung des Gesamtsystems in feingranulare Konfigurationen und die Definition der Abläufe für diese Konfigurationen. Wir zeigen im Folgenden die Abbildung des Standards mit dem CoreproAnsatz. Dazu gehören einerseits die Erstellung der Modellebene und andererseits die konkrete Instanziierung einer möglichen Prozessstruktur.

9.2.1 Entwurf der Modellebene

Wie in Abschnitt 5.1.3 skizziert, sind für die Erstellung der Modellebene drei Schritte notwendig. Erstens ist das Datenmodell, bestehend aus Objekt- und Relationstypen, zu definieren. Zweitens ist der *Object Life Cycle* (OLC) für jeden Objekttyp zu modellieren. Drittens müssen die OLCs im *Life Cycle Coordination Model* (LCM) durch externe Transitionen verbunden sowie die externen Transitionen zu OLC-Dependencies gebündelt und Relationstypen zugeordnet werden.

Entwurf des Datenmodells

Entsprechend des Prozessmodells in Abbildung 9.1, definieren wir im Datenmodell vier verschiedene Objekttypen: `System`, `Subsystem`, `Hardware` und `Software` (vgl. Abbildung 9.3).

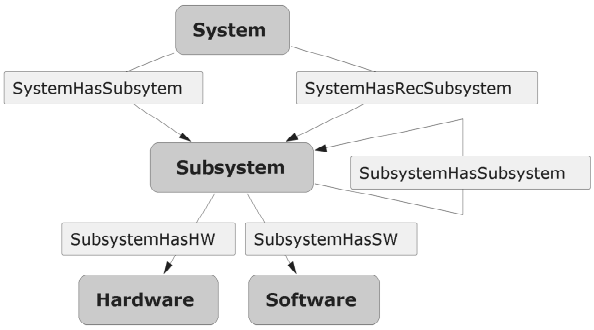


Abbildung 9.3: In Corepro$_{SIM}$ erstelltes Datenmodell der ISO26262

Die Abbildung der Beziehungen zwischen Objekttypen ist implizit durch die Prozessbeschreibung in Abbildung 9.1 gegeben:

- Objekte vom Typ `System` referenzieren Objekte vom Typ `Subsystem`. Die Abbildung kann über einen Relationstyp `SystemHasSubsystem` erfolgen. Enthält das Subsystem selbst wiederum Subsysteme, muss die Beziehung über den Relationstyp `SystemHasRecSubsystem` hergestellt werden. Dies ist notwendig, da sich die Synchronisationsprozesse der verknüpften externen Transitionen unterscheiden (siehe folgende Abschnitte).
- Objekte vom Typ `Subsystem` können mit Objekten vom Typ `Hardware` oder `Software` verknüpft sein. Die Abbildung der Beziehung kann über die Relationstypen `SubsystemHasHW` und `SubsystemHasSW` erfolgen.
- Objekte vom Typ `Subsystem` können ferner Objekte des gleichen Objekttyps referenzieren. Die Abbildung kann über einen rekursiven Relationstyp `SubsystemHasSubsystem` erfolgen (vgl. Abschnitt 5.2.1).

Entwurf der OLCs

Die Abläufe innerhalb der verschiedenen Integrationsstufen des Prozesses aus Abbildung 9.1 lassen sich auf OLCs transferieren. Die Schritte entsprechen den Prozessen innerhalb eines OLC, wobei der Standard lediglich nicht-deterministische Abläufe beschreibt. Die OLCs kapseln nur diejenigen Prozesse, die direkt für dieses Objekt ausgeführt werden müssen. Im Fall von Subsystemen sind das jeweils die Prozesse 4-5, 4-6 und 4-7.4.2 (vgl. Abbildung 9.1). Der OLC für den Objekttyp `Subsystem` enthält die Prozesse 4-5 und 4-6 (vgl. Abbildung 9.4). Der Prozess 4-7.4.2 (`System Integration and Tests`) ist nicht enthalten, denn seine Anwendung hängt vom umgebenden System ab. Er wird daher als Synchronisationsprozess im LCM realisiert.

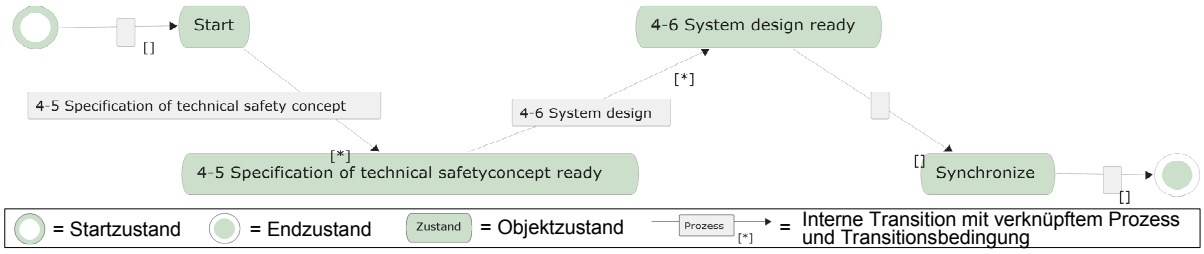


Abbildung 9.4: OLC für den Objekttyp `Subsystem`

Entwurf des LCM

Die Synchronisation der Prozesse verschiedener OLCs erfolgt über externe Transitionen. Da die Objekte bzw. Prozesse in Abbildung 9.1 hierarchisch geschachtelt sind, ergeben sich im LCM hauptsächlich externe Transitionen, die „einfache“ Ende- und Start-Synchronisationen der OLCs vornehmen (vgl. Synchronisation der OLCs `Subsystem`, `Hardware` und `Software` in Abbildung 9.5a). Hierbei ist zu beachten, dass externe Transitionen nicht in den Startzustand eines OLC eingehen dürfen (vgl. Abschnitt 4.5.2). Es muss daher ein Zwischenzustand modelliert und mit diesem synchronisiert werden (vgl. Abbildung 9.4 bzw. OLC `Hardware` in Abbildung 9.5a).

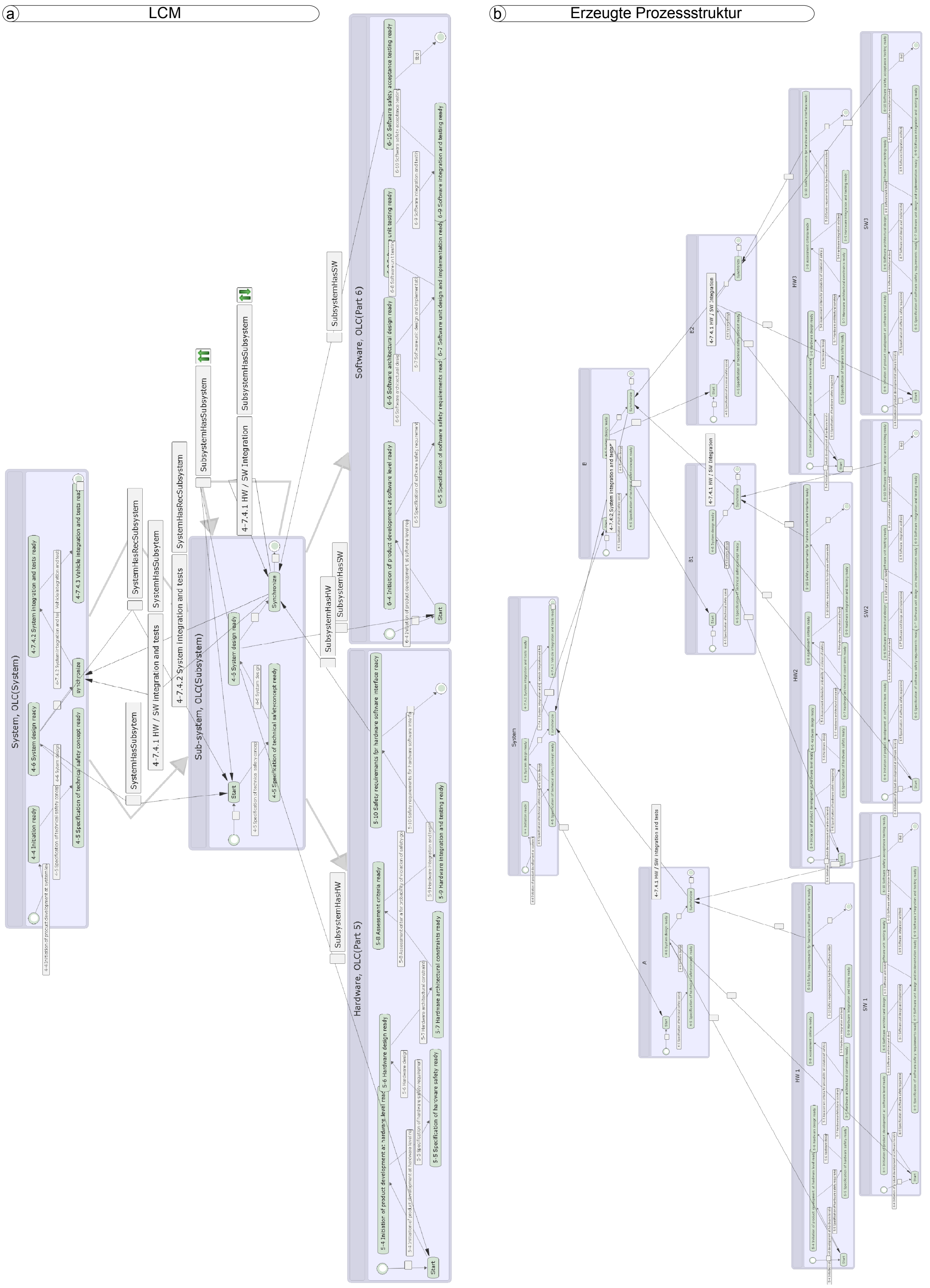


Abbildung 9.5: LCM und erzeugte Prozessstruktur für das Beispiel aus Abbildung 9.1

Die Integrationsschritte von Subsystemen, die rekursiv weitere Subsysteme enthalten, unterscheiden sich von Subsystemen, die Hardware und Software enthalten (vgl. Subsystem A und B in Abbildung 9.1). Subsysteme, die selbst wieder Subsysteme enthalten, führen zur Integration der enthaltenen Subsysteme den Prozess 4-7.4.2 aus. Dies lässt sich in CoreProo mittels Synchronisationsprozessen abbilden. Der Integrationsschritt ist daher als Synchronisationsprozess modelliert und mit einer externen Transition des Relationstyps `SubsystemHasSubsystem` verknüpft (vgl. Abbildung 9.5a). Dies erfordert letztendlich die Unterscheidung der Relationstypen `SystemHasSubsystem` und `SystemHasRecSubsystem` (vgl. Abbildung 9.3).

9.2.2 Erstellung der Instanzebene

Basierend auf der erstellten Modellebene lassen sich nun Datenstrukturen modellieren und damit entsprechende Prozessstrukturen automatisch erzeugen. Die Prozessstruktur aus Abbildung 9.5 basiert zum Beispiel auf der Datenstruktur aus Abbildung 9.1. Aus ihr ergeben sich die Objekte bzw. OLCs `System`, `Subsystem A`, `Subsystem B`, `Subsystem B1` und `Subsystem B2`. Die erzeugte Prozessstruktur entspricht dem Beispiel aus Abbildung 9.1 und damit den Anforderungen des Standards.

Die CoreProo-Prozessstruktur verfügt darüber hinaus über eine vollständige operationale Semantik, d.h. die erzeugte Prozessstruktur kann direkt ausgeführt werden (vgl. Abbildung 9.5). Die CoreProo-Konzepte sorgen zum einen dafür, dass die Prozesse entsprechend des im Standard geforderten Vorgehens koordiniert werden. Zum anderen stellen sie sicher, dass auch im Rahmen einer dynamischen Adaption oder einer Ausnahme der Standard eingehalten wird. So werden Inkonsistenzen in der Prozessstruktur und damit jede Abweichung vom Standard gekennzeichnet. Sofern Ausnahmen nicht ignoriert, sondern entsprechend des Konsistenzherstellungsprozesses (vgl. Abschnitt 7.3.3) aufgelöst werden, ist die standardkonforme Ausführung der Prozessstruktur weiterhin garantiert.

9.3 Abbildung des E/E-Release-Management-Prozesses mit statischer Konfiguration

Der Entwicklungsprozess der ISO 26262 zeichnet sich dadurch aus, dass die Abläufe hierarchisch geschachtelt sind. Im Umfeld des Release-Managements lässt sich eine derartige Schachtelung nicht beobachten. Stattdessen sind hier nicht-blockstrukturierte Abläufe mit vielfältigen Synchronisationsbeziehungen abzubilden.

Eine Systemstruktur kapselt verschiedene Teilsysteme (Systeme) und Komponenten (Subysteme). Auf Instanzebene kann das Anlegen eines Release (-objekts) und dessen Verknüpfung mit Komponenten im Rahmen der Konfigurationsbildung entweder statisch, d.h. zur Modellierzeit bei der Erzeugung der Datenstruktur, oder zur Laufzeit in Form dynamischer Änderungen der Datenstruktur erfolgen (vgl. Abschnitt 9.1.2). Wir unterscheiden daher Anwendungsfälle des Release-Management-Prozesses mit statischer Konfigurationsbildung (d.h. die Konfiguration wird

zur Modellierzeit festgelegt) und Anwendungsfälle des Release-Management-Prozesses mit dynamischer Konfigurationsbildung (d.h. die Konfiguration wird zur Laufzeit festgelegt und angepasst).

In diesem Abschnitt beschreiben wir zunächst Anwendungsfälle des Release-Management-Prozesses mit statischer Konfigurationsbildung und zeigen auf Basis des implementierten Demonstrators die Umsetzung dieser *idealen* Releasebildung mit den CoRePro-Konzepten. Abschnitt 9.4 beschreibt im Anschluss die dynamischen Aspekte des Release-Prozesses.

9.3.1 Anwendungsfälle mit statischer Konfiguration

Der Release-Management-Prozess mit statischer Konfigurationsbildung geht von einer Datenstruktur aus, in der bereits vor dem Start der Prozessstruktur die Konfigurationen für das Release und die Teilsysteme gebildet wurden. Wir beschreiben im Folgenden die Schritte zur Absicherung einer Komponente und eines Release. Die Abläufe für die Absicherung und Freigabe von Teilsystemen sind denen eines Release sehr ähnlich und werden daher nicht betrachtet.

Anwendungsfall 9.1 (Absicherung und Freigabe einer Komponente)
Soll eine Komponente abgesichert und freigegeben werden, erfolgen nach Lieferung des *Musters* (d.h. eines Prototyps) verschiedene Schritte zur Dokumentation und Absicherung der Funktionalität der Hardware und Software. Dazu gehören die Prüfung des Testprotokolls des Lieferanten, Dokumentation des gelieferten Standes und Sicherung der gelieferten Software sowie die Absicherung der Flashbarkeit[4], der Codierdaten[4], der Kommunikation am *Komponenten-Brett (Brett)* und der Funktion am *Hardware-in-the-Loop (HiL)* Prüfstand. Die Absicherung der Funktionalität auf dem HiL-Prüfstand ist abhängig von den technischen Eigenschaften der Komponente und daher optional. Abschließend werden die Ergebnisse der Absicherung dokumentiert und die Komponente kann *in* das Release gemeldet werden.

Wird bei einem der Absicherungsschritte ein gravierender Mangel an der Komponente entdeckt, muss die Komponente als *releaseuntauglich* gekennzeichnet werden.[5]

Anwendungsfall 9.2 (Absicherung und Freigabe eines Release)
Ein Release repräsentiert die gesamte E/E-Systemstruktur des Fahrzeugs. Sie besteht aus Teilsystemen und Komponenten inklusive ihrer Varianten (d.h. eine 150% Konfiguration; vgl. Abschnitt 1.2), sowie Beziehungen zwischen ihnen. Wir betrachten im Folgenden Komponenten und Teilsysteme ohne auf ihre Varianten Bezug zu nehmen.[6]

Einem Release sind, abhängig von der Phase im Entwicklungsprozess, unterschiedliche Teilsysteme und Komponenten zugeordnet. Komponenten können über Teilsysteme gebündelt und in dieser Form dem Release zugeordnet werden. Sie können aber auch direkt einem Release zugeordnet werden, sofern die Bündelung in einem Teilsystem aus fachlicher Sicht nicht sinnvoll ist.

[4] Die Absicherung der Flashbarkeit und Codierdaten prüft, ob das gelieferte Muster mit einer neuen Softwareversion bespielt (geflasht) werden kann und ob die Codierung, also die Konfiguration des Steuergeräts, zum aktuellen Softwarestand kompatibel ist.

[5] Wir betrachten die Modellierung der Abläufe, die *logisch* nach Kennzeichnung einer Komponente als *releaseuntauglich* zur Ausführung kommen, in Abschnitt 9.4.

[6] Die Abbildung von Varianten ist durch das generische Datenmodell möglich und kann in ähnlicher Form wie die Abbildung von Versionen erfolgen (siehe Abschnitt 9.4.2).

Anschließend durchläuft das Release verschiedene Prozesse (u.a. Absicherung Brett, Absicherung HiL, Absicherung im Fahrzeug, Qualitätssicherung). Wird bei einem der Absicherungsschritte an einer im Release enthaltenen Komponente ein gravierender Mangel entdeckt, der nicht durch Austausch der Komponente durch eine andere Version behoben werden kann, muss das Release als *nicht produktionstauglich* gekennzeichnet werden.

9.3.2 Entwurf des Datenmodells

Der Entwurf eines für den Release-Management-Prozess geeigneten Datenmodells wirft zunächst die Frage auf, welche Objekttypen im konkreten Anwendungsfall benötigt werden. Im einfachen Fall, dem Ablauf mit statischer Konfigurationsbildung, sind zunächst *Komponenten*, *Teilsysteme* und *Gesamtsystem* als eigenständige Objekttypen identifizierbar. Sie bedürfen jeweils individueller Abläufe, die in Form von OLCs für die verschiedenen Objekttypen definiert werden. Die Objekttypen sind über Relationstypen, die die Abhängigkeiten innerhalb des Release abbilden, miteinander verknüpft (vgl. Relationstypen `hatTS`, `hatKomp` und `TShatKomp` in Abbildung 9.6a).[7] Das Datenmodell repräsentiert damit die Systemstruktur eines Release.[8] Abbildung 9.6b wird in Abschnitt 9.3.3 erläutert.

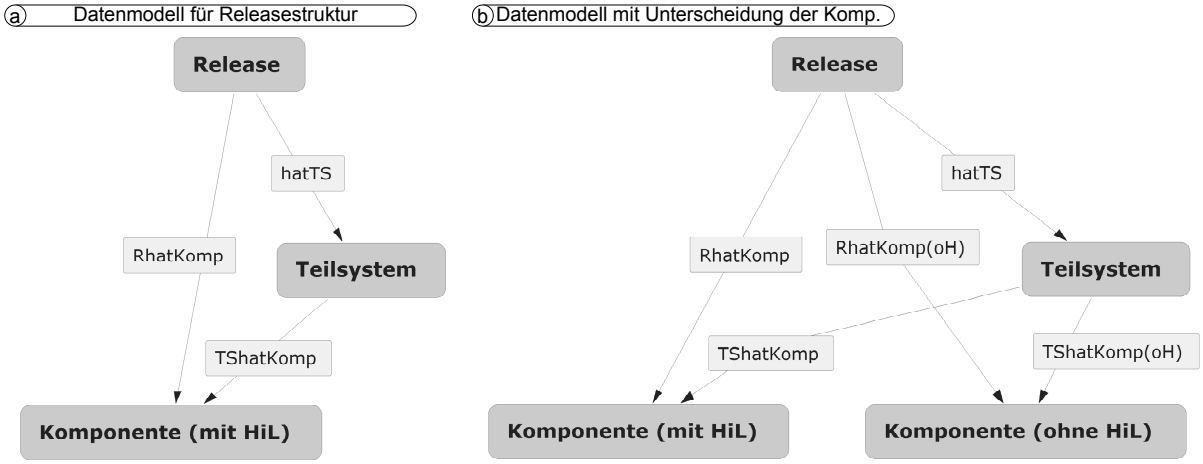


Abbildung 9.6: In COREPRO$_{SIM}$ erstelltes Datenmodell

9.3.3 Entwurf der Object Life Cycles

Die strukturellen Anforderungen an die OLCs der unterschiedlichen Objekttypen aus dem Datenmodell unterscheiden sich nur geringfügig (vgl. Anwendungsfälle 9.1 und 9.2). Die Abläufe für die Absicherung und Freigabe von Komponenten (vgl. Anwendungsfall 9.1) lässt sich durch Modellierung entsprechender Zustände und interner Transitionen abbilden (vgl. Abbildung 9.7). Herausforderungen sind allerdings die Abbildung nebenläufiger Prozesse und die Abbildung der *Releaseuntauglichkeit*. Die Koordination nebenläufiger Prozesse, wie sie beispielsweise für den Komponenten-OLC erforderlich ist (Absicherung Flashbarkeit und Absicherung Codierdaten;

[7] Detaillierte technische Abhängigkeiten wurden im betrachteten Fall nicht abgebildet.
[8] Es wird in Abschnitt 9.4 erweitert, um der Dynamik der Konfigurationsbildung gerecht zu werden.

vgl. Anwendungsbeispiel 9.1), kann auf zwei Arten geschehen. Zum einen können die parallelen Abläufe in *einem* Prozess gekapselt werden. Dies ist möglich, wenn die beiden Prozesse denselben Zustandsübergang verursachen. Zum anderen kann die Granularität im Datenmodell erhöht werden, indem das betroffene Objekt in mehrere „Unterobjekte“ aufgeteilt wird. Durch Instanziieren der Objekte und Einfügen entsprechender Synchronisationsbeziehungen (in Form von externen Transitionen) kann die Nebenläufigkeit der Prozesse abgebildet werden. Für den OLC aus Abbildung 9.7 wurde die erste Möglichkeit gewählt und die Absicherung der Flashbarkeit und der Codierdaten nebenläufig „in einem Prozess“ gekapselt. Dies führt während der Ausnahmebehandlung, beispielsweise im Rahmen von Sprüngen, zu einer Gleichbehandlung der beiden Prozesse (d.h. gleichzeitiger Start, aber auch gleichzeitiger Abbruch).

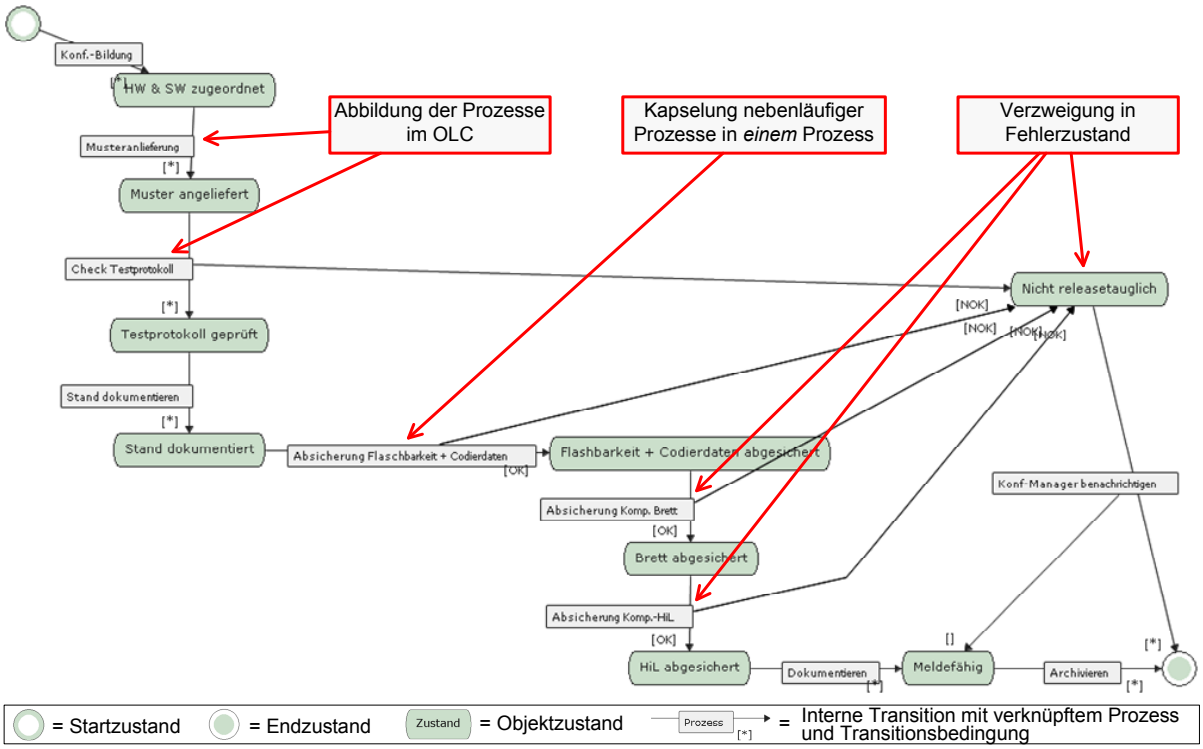


Abbildung 9.7: OLC für den Objekttyp Komponente (vgl. Anwendungsfall 9.2)

Ferner beschreibt Anwendungsfall 9.1 Abläufe für Komponenten mit oder ohne Funktionstest. Dieser optionale Schritt kann beispielsweise durch eine Verzweigung im OLC auf Basis eines Prozessergebnisses erfolgen. Im Allgemeinen ist jedoch bereits zur Modellierzeit bekannt, für welche Komponenten ein Funktionstest durchgeführt werden muss. Die dynamische Verzweigung kann an dieser Stelle durch Definition unterschiedlicher OLCs für Komponenten mit und ohne HiL-Funktionstest vermieden werden (vgl. Abbildung 9.6b).

9.3.4 Erstellen des Life Cycle Coordination Model

Der OLC für das Release-Objekt beinhaltet neben den Abläufen zur Absicherung und Freigabe eines Release auch Schritte zur Synchronisation der Komponenten bzw. ihrer Versionen. Dies ist

notwendig, um die parallelen Abläufe auf Komponenten- und Release-Ebene zu synchronisieren. So muss die Hardwareversion zeitlich vor der Softwareversion in das Release gemeldet werden, denn die Logistikprozesse sind für Hardware deutlich länger als für Software. Die Synchronisation der OLCs für Komponenten und Releases im zugehörigen LCM berücksichtigt diesen Aspekt. So verbinden externe Transitionen die Zustände `Brett abgesichert` und `HW eingefroren` sowie `Meldefähig` und `SW eingefroren` (vgl. Abbildung 9.8). Sie werden mit den entsprechenden Relationstypen `RhatKomp` verknüpft. Erst wenn jeweils alle externen Transitionen gefeuert haben, kann mit dem Ablauf im Release-OLC fortgefahren werden.

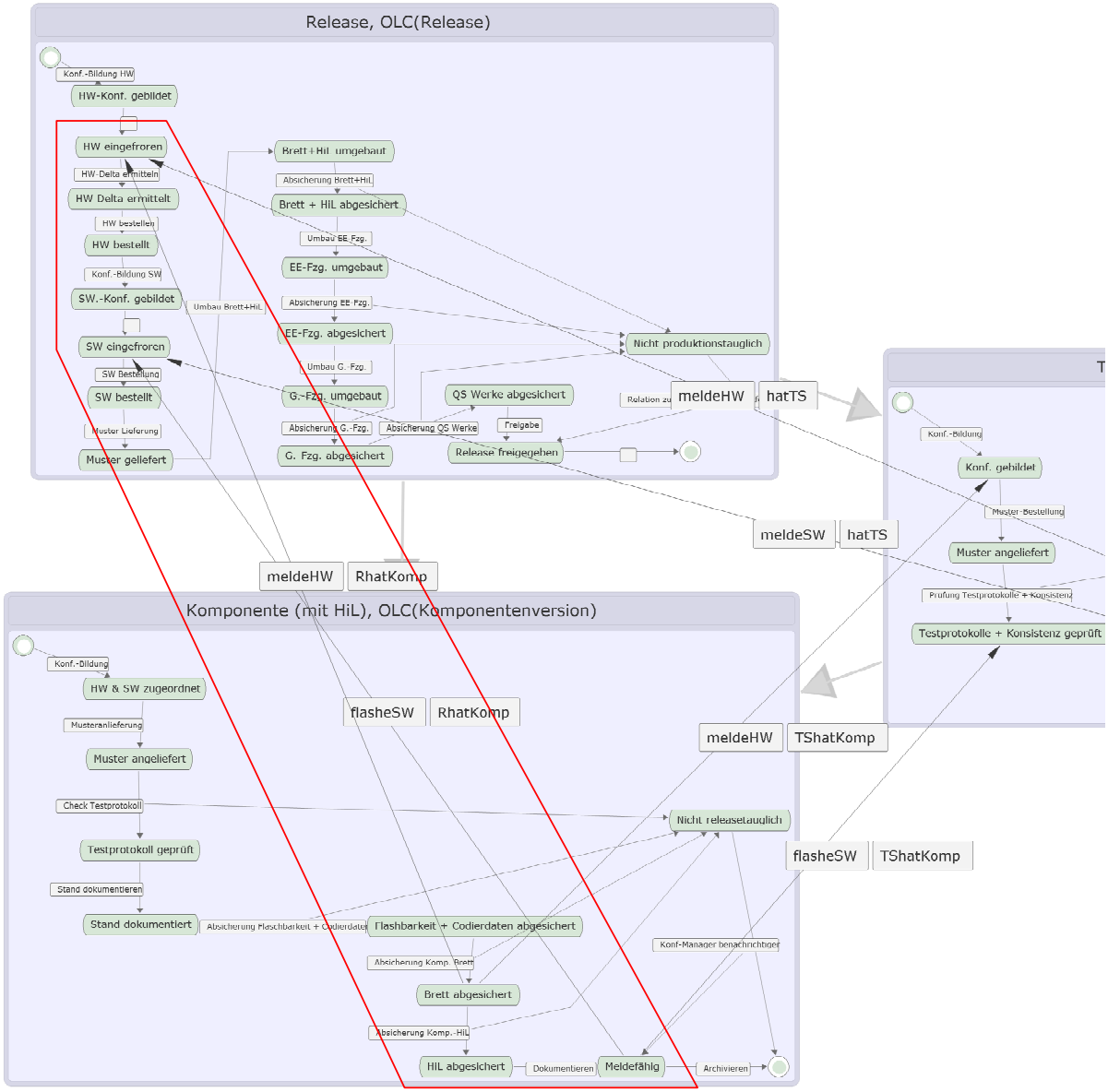


Abbildung 9.8: Verknüpfung der OLCs durch externe Transitionen im LCM

9.3.5 Erzeugung und Simulation der Prozessstruktur

Basierend auf der erstellten Modellebene können konkrete Releases erzeugt werden. Hierfür wird zunächst eine Datenstruktur angelegt, die ein Release-Objekt sowie die Teilsysteme und Komponenten enthält. Abbildung 9.9a zeigt einen Ausschnitt aus einer Datenstruktur mit den Objekten `Release BR X 0808`, `Navigation`, `Xenon Scheinwerfer`, `Display` und `GPS-Sensor`. Eine Komponente wird entsprechend ihrer Abhängigkeiten entweder mit Teilsystem-Objekten durch `TShatKomp`-Relationen oder direkt mit dem Release-Objekt durch eine `hatKomp`-Relation verknüpft (vgl. Anwendungsfall 9.1). Teilsystem-Objekte werden durch eine `hatTS` Relation mit dem Release-Objekt verknüpft. Ist die Datenstruktur für das Release vollständig erstellt, kann die zugehörige Prozessstruktur automatisch erzeugt werden (vgl. Abbildung 9.9b).[9]

Die Simulations- bzw. Ausführungskomponente erlaubt die Ausführung der generierten Prozessstruktur (vgl. Abbildung 9.9b). Wir wollen den Ablauf an dieser Stelle nur skizzieren: Nach Start der Prozessstruktur werden die Startzustände aller OLCs aktiviert. Während der OLC des Objekts `Release BR X 0808` in Zustand `HW-Konf. gebildet` wartet, durchlaufen die OLCs der Komponenten die Prozesse zur Absicherung. Sobald ein OLC den Zustand `Brett abgesichert` erreicht und der `meldeHW`-Prozess der externen Transition ausgeführt wurde, wird die Transition gefeuert. Haben alle externen Transitionen gefeuert, wird der Zustand `HW eingefroren` im OLC des Release-Objekts aktiviert und mit der Ausführung fortgefahren. Erreicht der OLC des Release-Objekts schließlich den Zustand `Release freigegeben` (bzw. seinen Endzustand) und die OLCs der Komponenten (sowie die OLCs der Teilsysteme) ihren jeweiligen Endzustand, ist die Prozessstruktur durchlaufen und wird als `TERMINATED` gekennzeichnet.

9.4 Abbildung des E/E-Release-Management-Prozesses mit dynamischer Konfiguration

Die Releasebildung ist in der Praxis noch komplexer, als es die Anwendungsfälle aus Abschnitt 9.3 erkennen lassen. Auf deren Basis definieren wir weitere Anwendungsfälle, die im Rahmen des Release-Management-Prozesses mit dynamischer Releasebildung auftreten. Diese Anwendungsfälle zeichnen sich insbesondere dadurch aus, dass während des Durchlaufs Komponenten aus der Systemstruktur bzw. aus dem Release dynamisch entfernt oder aber dynamisch hinzugefügt werden. Dies ist zum Beispiel notwendig, wenn aufgrund des Ergebnisses eines Absicherungsschritts eine Komponente als *nicht produktionstauglich* gekennzeichnet wird und durch eine andere Version ersetzt werden muss.

9.4.1 Anwendungsfälle mit dynamischer Konfiguration

Der *Release-Management-Prozess mit dynamischer Konfigurationsbildung* geht von einer dynamisch aufgebauten Datenstruktur aus, in welche die Komponenten für das Release während der Laufzeit gemeldet werden (vgl. Abschnitt 9.1.2). Wir betrachten im Folgenden drei Anwendungsfälle, die die Herausforderungen der dynamischen Konfigurationsbildung beschreiben.

[9] Abbildung C.3 in Anhang C veranschaulicht die Komplexität einer Prozessstruktur mit etwa 100 Komponenten.

Abbildung 9.9: Ausschnitt einer Systemstruktur und erzeugte Prozessstruktur

Anwendungsfall 9.3 (Versionierung von Komponenten)
Die Versionierung von Komponenten ist eine wichtige Voraussetzung für die dynamische Erstellung eines Release. Sie erlaubt die Ableitung verschiedener *Versionen* (Entwicklungsstände) einer Komponente und nur diejenige dem Release zuzuordnen, welche den geforderten Reifegrad besitzt. Soll zum Beispiel eine neue Funktion in die Software eines Steuergeräts integriert werden, wird dazu eine neue Komponentenversion erzeugt und abgesichert (vgl. Anwendungsfall 9.1). Um Seiteneffekte auszuschließen müssen in der Regel alle Prozesse erneut ausgeführt werden. In bestimmten Fällen soll es allerdings möglich sein, Prozesse gezielt zu überspringen.

Anwendungsfall 9.4 (Melden einer Komponentenversion zum Release)
Bei der (dynamischen) Erstellung eines Release stehen zwar die enthaltenen Komponenten von Beginn an fest, die konkret zugeordneten Komponentenversionen jedoch nicht. Damit hat der Entwickler die notwendige Freiheit, verschiedene Komponentenversionen zu erstellen und auf Basis fachlicher Kriterien die am besten geeignete Version dem Release zuzuordnen. Dies erfordert ein *spätes Binden* der Versionen an das Release. Soll eine bestimmte Komponentenversion dem Release zugeordnet werden, wird sie an das Release *gemeldet*. Die Meldung durchläuft hierbei die drei verschiedenen Phasen *geplant*, *HW unverbindlich* und *HW+SW verbindlich*. Über diese Phasen werden das Release und die Komponenten bzw. Teilsysteme synchronisiert.

Ist die Komponente einem Teilsystem zugeordnet, erfolgt zunächst die Meldung der Komponentenversion an die entsprechende Teilsystemversion und – nachdem alle dem Teilsystem zugeordneten Komponenten gemeldet haben – die Meldung der Teilsystemversion an das Release.

Anwendungsfall 9.5 (Änderung der Meldung einer Komponentenversion)
Wird während der Durchführung der Absicherungsprozesse ein schwerwiegender Fehler an einer Komponente festgestellt, muss diese in den Fehlerzustand überführt werden. Dann kann es notwendig sein, die Meldung der fehlerhaften Komponentenversion zurückzuziehen und stattdessen eine andere Komponentenversion in das Release zu melden. Abhängig vom Fortschritt des Release-Management-Prozesses kann dann eine Ausnahmebehandlung erforderlich werden. Befindet sich das Release beispielsweise bereits in der Absicherungsphase und der Austausch der Komponentenversion führt zu einer Inkonsistenz, muss auf Basis einer individuellen fachlichen Abwägung (z.B. Fehlerschwere und -auswirkungen) über die Behandlung der Ausnahme entschieden werden. Reaktion kann das Wiederholen aller oder einzelner Absicherungsschritte für das Release, genauso wie das Ignorieren der Inkonsistenz sein.

9.4.2 Abbildung von Versionen im Datenmodell

In Anwendungsfall 9.3 wird die Versionierung von Komponenten motiviert. CoreProo bietet zwar keinen direkten Versionierungsmechanismus an, durch eine geeignete Definition des Datenmodells mit den Objekttypen *Komponente* und *Komponentenversion* kann das gewünschte Verhalten aber abgebildet werden (vgl. Abbildung 9.10a). Das Datenmodell aus Abbildung 9.6 kann hierzu entsprechend erweitert werden, indem den Objekttypen jeweilige Versionstypen zugeordnet werden. Das Datenmodell erlaubt damit die statische Zuordnung der Komponenten zu einer Baureihe. Die konkreten Ausprägungen der Komponenten (d.h. ihre Versionen) lassen sich dynamisch zur Laufzeit erstellen und dem Release zuordnen (vgl. Abbildung 9.10b).

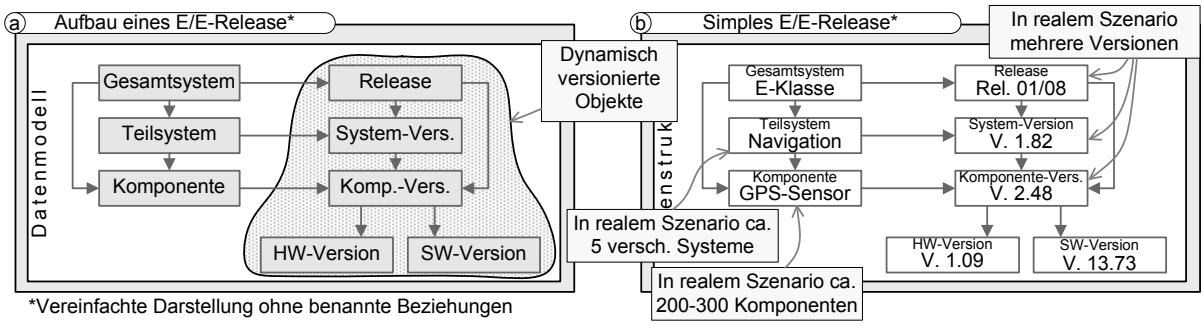


Abbildung 9.10: Erweitertes Datenmodell für die Abbildung von Versionen

Prozess-seitig müssen die jeweiligen Komponentenversionen die OLCs und damit die einzelnen Prozesse von Neuem durchlaufen. OLCs der Komponentenversionen sind daher identisch mit dem in Abschnitt 9.3.3 beschriebenen OLC für Komponenten. Sollen bestimmte Prozesse einer Komponentenversion nicht ausgeführt werden, da aus fachlicher Sicht die Ergebnisse der Vorgängerversion ihre Gültigkeit behalten, können diese mithilfe der Sprungoperation übersprungen werden.

Objekte vom Typ `Komponente` des erweiterten Datenmodells werden keinem OLC zugeordnet, da diese Objekte statisch sind und lediglich der Abbildung von Datenbeziehungen dienen.

9.4.3 Melden einer Komponentenversion zum Release

Das Anwendungsbeispiel 9.4 beschreibt verschiedene Anforderungen an die Unterstützung beim Melden einer Komponente zum Release. Die unabhängige Erstellung von Komponentenversionen und deren *spätes Binden* an ein Release kann mittels dynamischer Adaption der Prozessstruktur erreicht werden. Die Frage ist jedoch, wie vor Start der Prozessstruktur festgelegt werden kann, von welchen Komponenten eine Versionsmeldung erforderlich ist. Zudem stellt sich die Frage, wie der Ablauf dieser Meldung sichergestellt werden kann, so dass genau für jede dem Release zugeordnete Komponente auch genau eine Version gemeldet wird.

In diesem Fall gibt es ebenfalls verschiedene Lösungsvarianten:

- Es wird ein Relationstyp angelegt, über den Komponenten einem Release zugeordnet werden. Des Weiteren wird für jede Meldephase ein Relationstyp angelegt. Der Nutzer verknüpft dann entsprechend der Meldephase über den entsprechenden Relationstyp seine Komponentenversion mit dem Release.[10]

- Es wird ein neuer Objekttyp `KompMeldung` angelegt, welcher zum einen die versionsunabhängige Verknüpfung des Release mit einer Komponente (vom Objekttyp `Komponente (nv)`) erlaubt (vgl. Abbildung 9.11a) und zum anderen die verschiedenen Phasen des

[10] Hierbei können wir nicht sicherstellen, dass jeweils immer dieselbe Komponentenversion gemeldet wird. Die Prüfung von Datenbeziehungen ist allerdings auch keine Anforderung an die CoRepro-Konzepte. Wir gehen davon aus, dass die Datenstruktur in einem Produktdaten-Management-System angelegt und dort die Datenbeziehungen geprüft werden. Alternativ lassen sich derartige Prüfungen auch mittels Workarounds in CoRepro realisieren (vgl. Abschnitt 9.5).

Meldeprozesses abbilden kann. Die Zuordnung der Komponente zu einem Release erfolgt dann durch Anlegen der Objekte vom Typ `KompMeldung` sowie ihrer Verknüpfung mit dem Release-Objekt und der Komponente durch Einfügen der Relationen vom Typ `benötigtKompMeldung` bzw. `MeldungVon` (vgl. Abbildung 9.11b). Soll eine konkrete Komponentenversion zugeordnet werden, kann die entsprechende Relation zum Melden-Objekt dynamisch eingefügt werden. Die zugehörigen externen Transitionen zur Synchronisation der entsprechenden OLCs werden dann der Prozessstruktur automatisch hinzugefügt (vgl. Abbildung 9.11c und 9.11d).

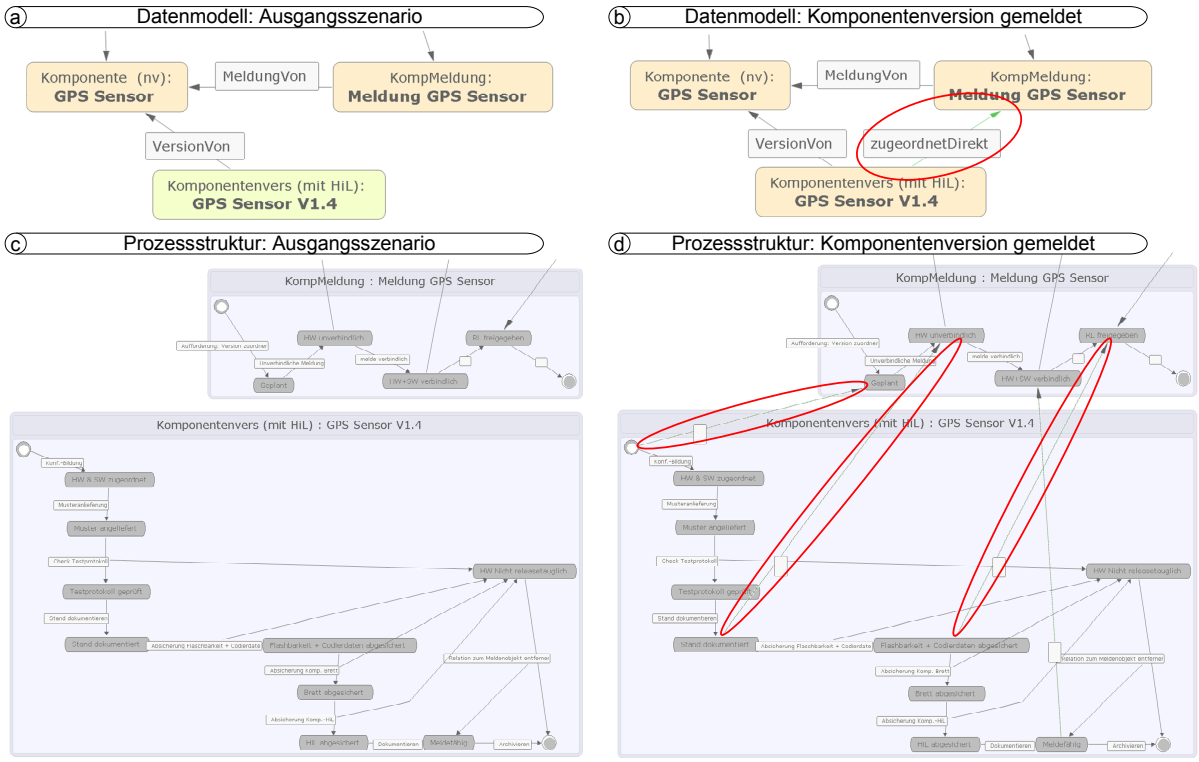


Abbildung 9.11: Melden einer Komponentenversion zum Release

An sich sind beide Lösungsvarianten gleichermaßen für die Abbildung des Melde-Prozesses geeignet. Wir betrachten im Folgenden die zweite Lösung genauer. Die hierfür gezeigten Lösungsaspekte lassen sich in ähnlicher Form auch auf die erste Lösungsvariante anwenden.

Zur Laufzeit muss ab einem bestimmten Zeitpunkt dem *Meldung-Objekt* genau eine Komponentenversion zugeordnet sein. Ohne dynamische Prüfung, ob eine Beziehung zu genau einer Komponentenversion besteht, würde der OLC des Meldung-Objekts ohne „angehängtes“ Versionsobjekt durchlaufen werden und damit zwar eine korrekte, aber fachlich unsinnige Daten- und Prozessstruktur zulassen. Die CoRepro-Konzepte sehen die Definition von Kardinalitäten bzw. dynamischen Einschränkungen für die Datenstruktur nicht vor. Annahme bei der Definition der Anforderungen an die Modellierung war schließlich, dass Datenstrukturen Sichten auf gege-

bene Produkt- bzw. Systemstrukturen sind und damit keiner weiteren Prüfung bedürfen (vgl. Abschnitt 2.2.2).[11]

Der Anwendungsfall selbst kann jedoch trotzdem mit den COREPRO-Konzepten vollständig abgebildet werden. Die eingeführten Schnittstellen der Ausführungskomponente in COREPRO$_{SIM}$ erlauben die Analyse der Datenstruktur (vgl. Abschnitt 8.6). Ein geeigneter Analyseprozess kann auf Basis dieser Schnittstellen die Kardinalität dynamisch prüfen, indem er als Prozess mit einer internen Transition verknüpft wird und Zustandsübergänge erst dann erlaubt, wenn die gewünschte Kardinalität erfüllt ist (vgl. Abschnitt 9.12a und b). Das Konzept ermöglicht nicht nur die gezielte Prüfung der Kardinalität zu einem bestimmten Zeitpunkt der Prozessausführung. Ferner ist mit diesem Mechanismus die dynamische Anpassung der Kardinalität möglich, indem diese als Parameter an den Analyseprozess übergeben wird.

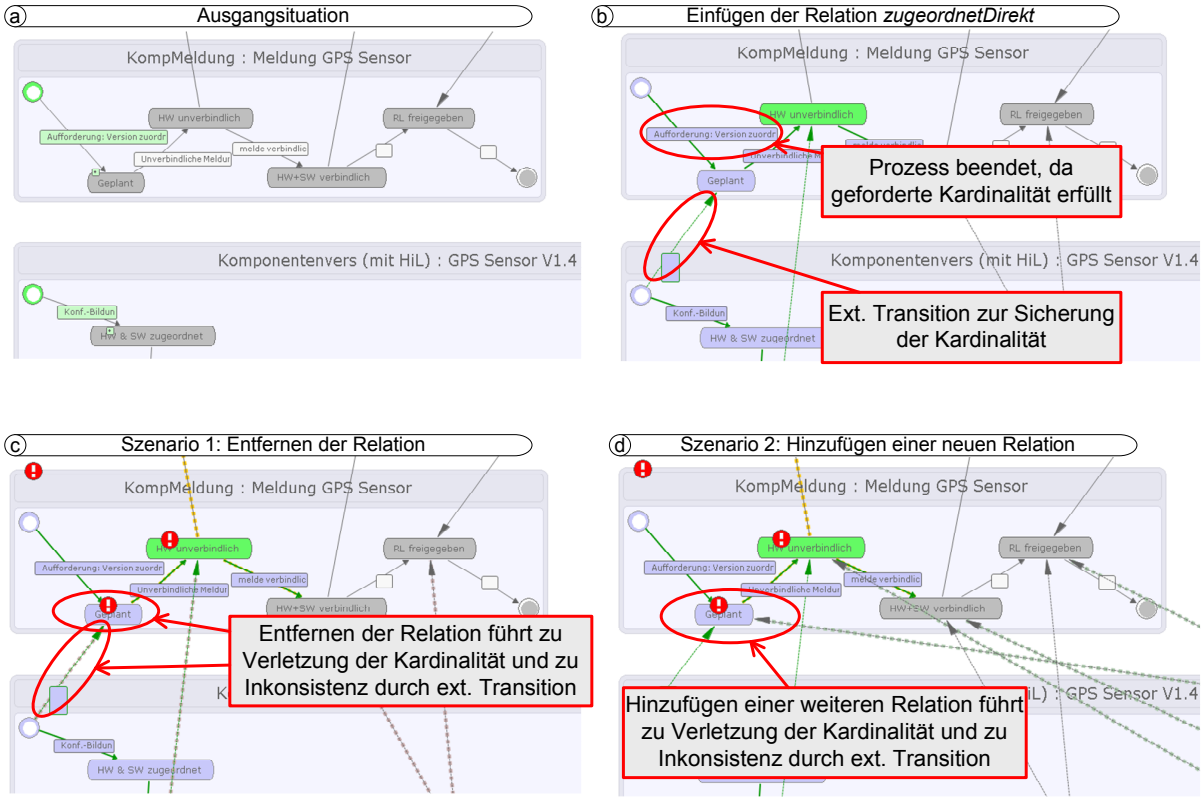


Abbildung 9.12: Prüfung der Kardinalitäten durch Analyseprozess in COREPRO$_{SIM}$

Der beschriebene Mechanismus erlaubt nur die Prüfung der Kardinalität während der Ausführung des entsprechenden Prüfprozesses: Eine nachträgliche dynamische Adaption der Datenstruktur könnte die Kardinalität wieder verletzen. Ihre Gültigkeit kann jedoch durch eine geeignete Modellierung externer Transitionen auch nach Prozessausführung sichergestellt werden. Geht in den Zielzustand des Prüfprozesses (`geplant`) eine externe Transition ein, die sofort mit Anlegen der Relation feuert, etwa wenn ihr Quellzustand der Startzustand des verknüpften OLC ist (vgl. Abbildung 9.12b), entsteht bei einer nachträglichen Änderung der Beziehung (Entfernen einer

[11] Die Prüfung einer Produktstruktur auf Konsistenz geschieht im untersuchten Umfeld beispielsweise durch Produktdaten-Management-Systeme (vgl. Abschnitt 3.1.3).

bestehenden oder Hinzufügen einer weiteren Relation) durch die externe Transition eine Inkonsistenz im Zielzustand des Prüfprozesses (vgl. Abbildung 9.12c und d).[12] Sie zeigt implizit die Verletzung der Kardinalität an.

9.4.4 Änderung der Meldung einer Komponente

Anwendungsfall 9.5 beschreibt die Änderung einer gemeldeten Komponentenversion. In der modellierten Prozessstruktur kann die Meldung einer Komponentenversion `kv` geändert werden, indem die Relation `zugeordnetDirekt` von `kv` zum zugeordneten Melden-Objekt entfernt und dieses stattdessen mit einer andere Komponentenversion verknüpft wird. Das Entfernen der `zugeordnetDirekt` Relation kann unter Ausnutzung der Änderungsoperation `REMOVERELATION` für die Datenstruktur auch automatisch geschehen, wenn beispielsweise der Fehlerzustand in einem Komponenten-Objekt erreicht und dafür ein entsprechender Prozess mit der aus dem Fehlerzustand ausgehenden Transition verknüpft wird (vgl. Abbildung 9.13).

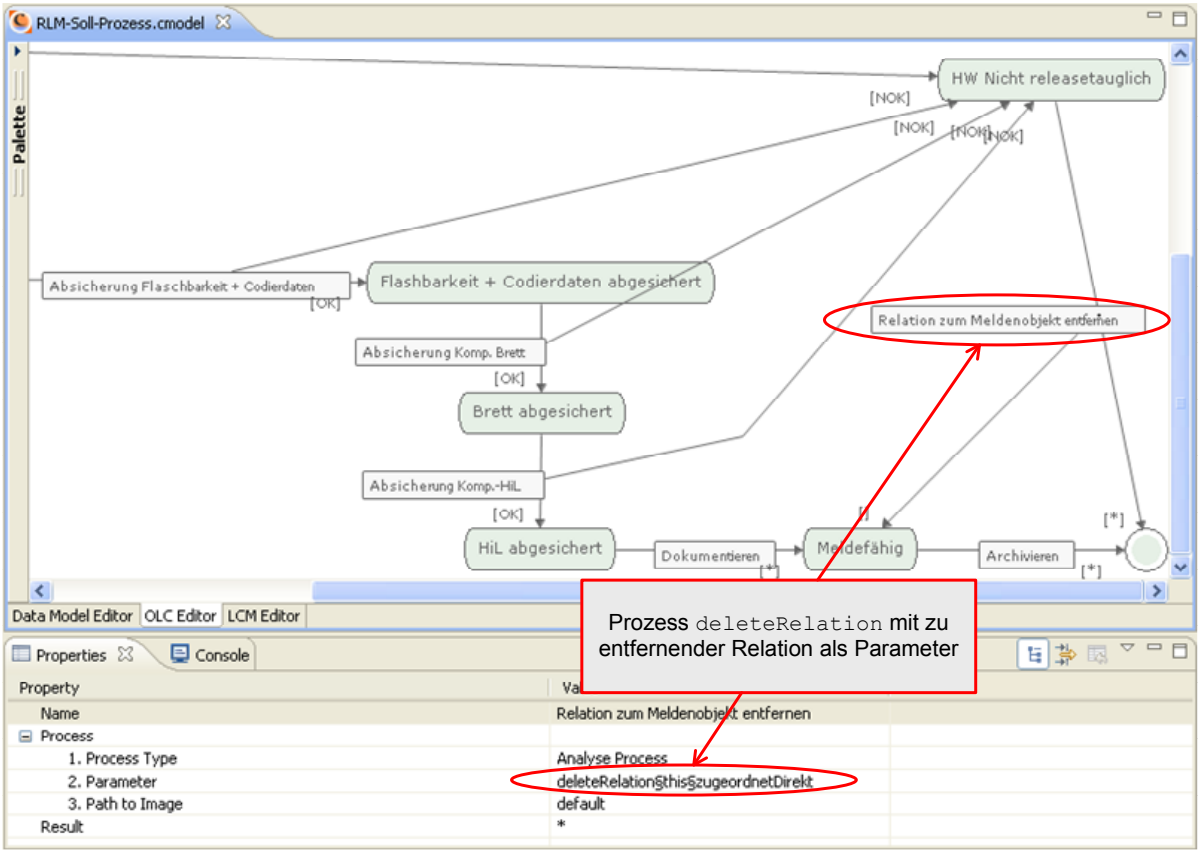


Abbildung 9.13: Automatisches Entfernen der Releasezuordnung bei Erreichen des Fehlerzustands

[12] Im Rahmen der Ausnahmebehandlung muss immer auf einen Vorgängerzustand des Prüfprozesses anstatt auf seinen Nachfolger gesprungen werden, um die Kardinalität nicht zu verletzen.

Nach Entfernen bzw. Hinzufügen der Relation und der entsprechenden externen Transitionen kennzeichnet die Konsistenzanalyse, wie bei der manuellen Anwendung der Operationen, diejenigen Zustände, deren Markierung nach Entfernen der externen Transitionen nicht mehr konsistent ist (vgl. Abschnitt 6.3.2). Die daraufhin erwarteten Inkonsistenzen werden ebenfalls gekennzeichnet (vgl. Abschnitt 7.3.2). Auf Basis dieser Kennzeichnungen kann der Endbenutzer mit den aus Kapitel 7 bekannten Operationen die Ausnahmebehandlung durchführen. Aus fachlicher Sicht sind das zum Beispiel:

- die vollständige Wiederholung aller Absicherungsschritte (z.B. Rücksprung im OLC für das Release-Objekt; vgl. Markierungsoperation JUMP aus Abschnitt 7.2.1),
- die Wiederholung ausgewählter Absicherungsschritte (z.B. Rücksprung im OLC für das Release-Objekt, sowie Überspringen nicht zu wiederholender Prozesse; vgl. Markierungsoperation JUMP aus Abschnitt 7.2.1) oder
- das „normale Fortfahren“ im Prozess (Ignorieren der Inkonsistenzen; vgl. Kennzeichnungsoperation IGNOREINC aus Abschnitt 7.4.1).

9.5 Weitergehende Aspekte

Im Rahmen der Validation wurden nicht nur Standardabläufe modelliert. Ferner untersuchten wir auch spezielle Szenarien, die bei der Nutzung des Ansatzes zu beachten sind. Die Szenarien sind nicht als Teil der Fallstudie zur Prüfung der Abdeckung der fachlichen Anforderungen zu verstehen, sondern dienen vielmehr der Prüfung der praktischen Grenzen des Ansatzes. Wir wollen sie in den folgenden Abschnitten kurz skizzieren.

9.5.1 Modellierung von Fehlerzuständen

Für die Abbildung der *Releaseuntauglichkeit* einer Komponente aus Anwendungsfall 9.1 ist die zustandsorientierte Sicht sehr hilfreich. So kann ein *Fehlerzustand* modelliert werden, der durch eine Verzweigung auf Basis des Ergebnisses eines Absicherungsprozesses erreicht wird (vgl. Abbildung 9.14a). Wird zur Laufzeit durch einen „externen Prozess“ ein Fehler in einer Komponente gefunden, kann ihr Fehlerzustand auch durch Anwendung der Sprungoperation erreicht werden.

Soll nach Erreichen des Fehlerzustands ein Prozess zur (automatischen) Behandlung der Ausnahme ausgeführt werden (vgl. Abschnitt 9.4.4), muss Folgendes beachtet werden: in dem nicht gewählten Pfad des OLC (d.h. im *normalerweise* durchlaufenen Pfad; vgl. Abbildung 9.14b) findet eine Deadpath-Eliminierung statt *bevor* der Fehlerbehandlungsprozess gestartet wird. In ungünstigen Konstellationen kann dies zu einem unerwünschten Zustandswechsel in abhängigen Objekten führen bevor die Behandlung des Fehlers beginnt (vgl. Abbildung 9.14c). Zwar führt dies in den meisten Fällen nicht zu Problemen, die unerwünschte Abwahl der externen Transitionen kann aber umgangen werden, indem aus dem Fehlerzustand eine interne Transition in den Quellzustand der „problematischen“ externen Transition modelliert wird. Dadurch wird entsprechend Markierungsregel MR3 auf Seite 72 seine Markierung als SKIPPED verhindert (vgl. Abbildung 9.14d).

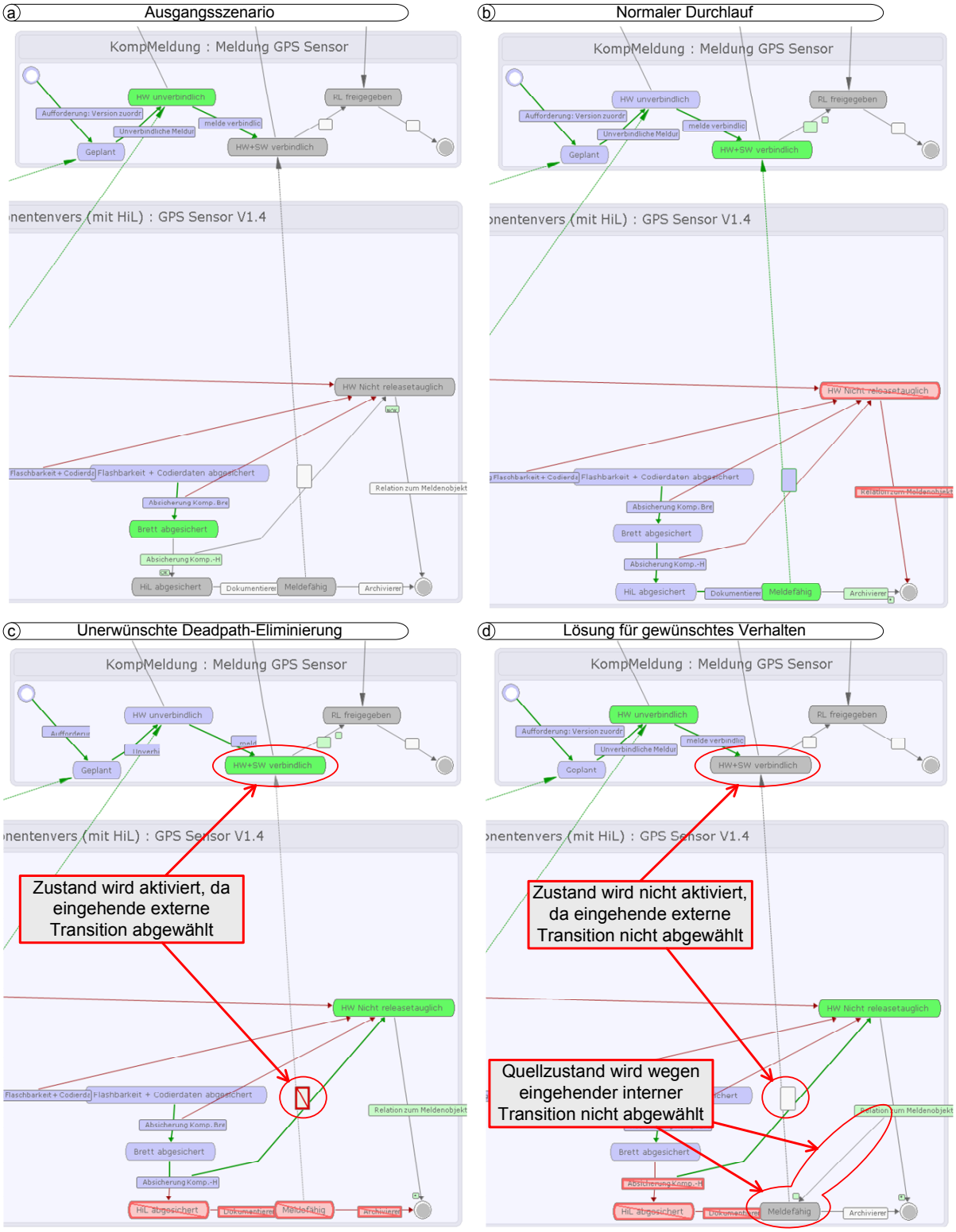


Abbildung 9.14: Abbildung des Fehlerzustands im OLC für Komponentenversionen

9.5.2 Eingeschränktes Melden einer Komponentenversion

In den aktuellen Release-Management-Prozessen ist das Melden einer Komponente in die Status *geplant*, *verbindlich* und *eingefroren* unterteilt. Im Gegensatz zum beschriebenen Anwendungsfall 9.4 für zukünftige Abläufe, wird in aktuellen Abläufen zusätzlich festgelegt, zu welchem Zeitpunkt welcher Status der Komponentenversionen in das Release gemeldet werden kann. Wir nennen diese Festlegung *Statuskonfiguration*. Dabei ist zu beachten, dass aufgrund der Dynamik des Prozesses die Statuskonfiguration keiner Reihenfolge unterworfen ist und dass auch Kombinationen der zu meldenden Status festgelegt werden können. Wird beispielsweise die Kombination *geplant* und *eingefroren* konfiguriert, so können nur diese Status gemeldet werden. Einmal getätigte Meldungen bleiben bestehen und sind von dem Wechsel der Statuskonfiguration nicht betroffen.

Das eingeschränkte Melden einer Komponente kann oberflächlich betrachtet durch ein Objekt zur Konfiguration der möglichen Meldungen und dessen Synchronisation mit den Meldung-Objekten aller Komponenten realisiert werden. Es zeigt sich jedoch, dass dieser Anwendungsfall im Gegensatz zu den „üblichen“ Szenarien folgende besonderen Anforderungen an die Abbildung durch CoRePro stellt:

1. *Beliebige Aktivierung der Zustände* im OLC zur Konfiguration der möglichen Meldungen (d.h. kein sequentieller Ablauf wie üblicherweise in OLCs).
2. *Mehrere aktivierte Zustände* zur Konfiguration der möglichen Meldungen.
3. *Zeitkritische Synchronisation* – der Zustand der Meldung-Objekte der Komponenten darf nur dann erreicht werden, wenn im Objekt zur Konfiguration der entsprechende Zustand derzeit *aktiviert* ist (d.h. er besitzt die Markierung `ACTIVATED`).

Die folgenden Abschnitte zeigen mögliche Umsetzungen der genannten speziellen Anforderungen.

1) Beliebige Aktivierung der Zustände im OLC zur Konfiguration der möglichen Meldungen kann mithilfe der Sprungoperation realisiert werden. Um dennoch einen korrekten OLC (im Sinne der in Kapitel 4 vorgestellten Konzepte) zu modellieren, können die gewünschten Zustände als Verzweigungspfade dargestellt werden (vgl. Abbildung 9.15). Zur Laufzeit ist das Wechseln der Zustände über das Prozessergebnis des Verzweigungspfads oder durch einen Sprung möglich.

2) Mehrere aktivierte Zustände werden durch die CoRePro-Konzepte zwar nicht direkt unterstützt, können aber über verschiedene Mechanismen abgebildet werden. Eine Möglichkeit ist die mehrfache Instanziierung des Meldungskonfigurationsobjekts (und damit seines OLC).[13] Durch geeignete Relationen (z.B. `repräsentiertMeldungsKonfiguration`) kann der Zusammenhang der einzelnen Objekte klar dargestellt werden. Eine weitere Möglichkeit, mehrere aktivierte Zustände abzubilden, ist die „Ausmultiplikation“ der möglichen Zustände und deren Darstellung in einem OLC (vgl. Abbildung 9.15).

[13] Die mehrfache Instanziierung von Objekten zur Darstellung paralleler Zustände entspricht im weiteren Sinne der vorgeschlagenen Aufsplittung von OLCs zur Abbildung nebenläufiger Prozesse (vgl. Abschnitt 9.3.3).

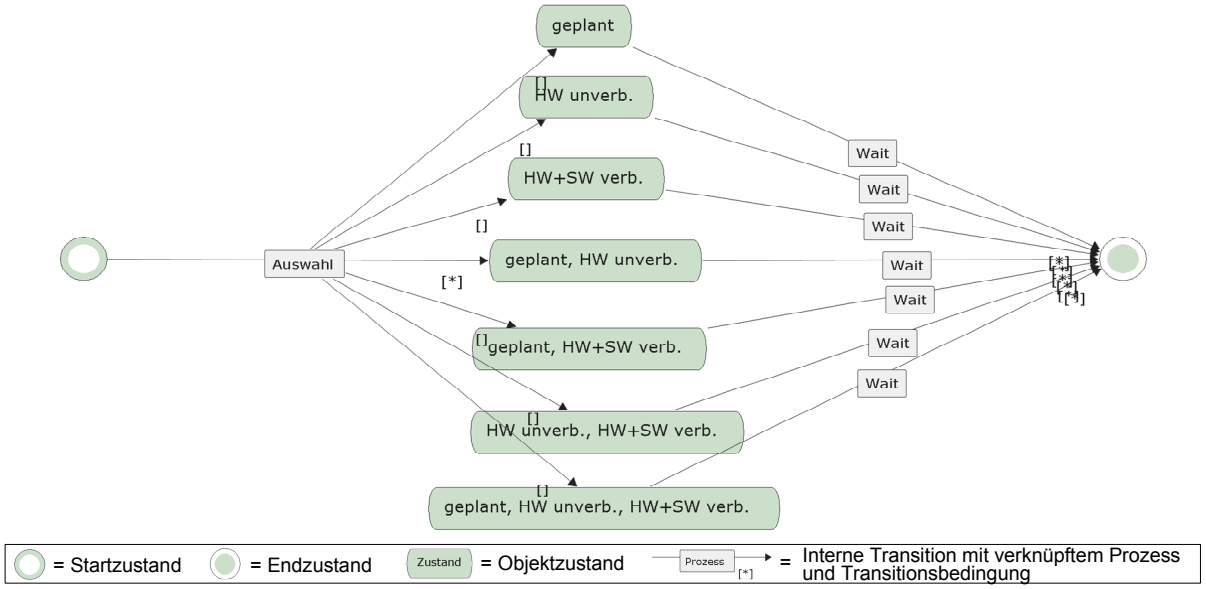


Abbildung 9.15: OLC zur Konfiguration der möglichen Meldungen

3) Zeitkritische Synchronisationen sind in den COREPRO-Konzepten nicht vorgesehen. Zwar können externe Transitionen genutzt werden, um das Erreichen eines Zustands zu verhindern (solange der Quellzustand der externen Transition nicht aktiviert ist), der resultierende Mechanismus zur Darstellung des geforderten Szenarios wäre jedoch sehr komplex. Schließlich muss bei einem Wechsel der möglichen Zustände in der Konfiguration der Meldungen dann auch die derzeit *sperrende* externe Transition gefeuert oder entfernt werden. Ferner müssen weitere externe Transitionen nun das Erreichen anderer Zustände in den OLCs der Melden-Objekte verhindern. Der nicht-sequentielle Ablauf im Meldungskonfigurationsobjekt erfordert dann das dynamische Hinzufügen bzw. Entfernen von externen Transitionen mit zugehöriger Ausnahmebehandlung.

Eine weniger komplexe Möglichkeit ist die Nutzung der Schnittstellen der Ausführungskomponente von COREPRO$_{SIM}$[14] und die Definition eines geeigneten Prozesses für die Prüfung der aktivierten Zustände des OLC zur Konfiguration der Meldungen. Ein derartiger Prozess muss interne Zustandsübergänge der OLCs der Meldung-Objekte mit dem OLC des Meldungskonfigurationsobjekts synchronisieren. Abbildung 9.16 veranschaulicht den Mechanismus. Der Prozess `SyncMit ok.S1` sorgt dafür, dass die interne Transition, mit der er verknüpft ist, erst dann feuert, wenn der Zustand `ok.S1` als `ACTIVATED` markiert ist. Der Prozess `SyncMit ok.S1` wird gestartet, sobald der Zustand `o.S1` aktiviert wird. Erst wenn `ok.S1` als `ACTIVATED` markiert wird, endet `SyncMit ok.S1` und der Folgezustand `E` wird erreicht.

Die konzeptionelle Korrektheitsprüfung auf Verklemmungsfreiheit der Prozessstruktur wird durch Anwendung eines derartigen Prozesses jedoch umgangen.[15] Im vorliegenden Fall kann es theoretisch zu einer Verklemmung im Ablauf kommen, wenn der zu überprüfende Zustand in `ok` (d.h. Zustand `ok.S1`) bereits durchlaufen ist und erst dann `o.S1` aktiviert wird. Dann wird der Pro-

[14] Die implementierte Schnittstelle von COREPRO$_{SIM}$ verfügt über geeignete Methoden zur Abfrage der Laufzeitmarkierungen; nicht in Abbildung 8.11 dargestellt.
[15] Die Prüfung der Semantik einzelner Prozesse ist keine Anforderung an COREPRO.

zess `SyncMit ok.S1` nicht beendet, da Zustand `ok.S1` im regulären Ablauf nicht mehr erreicht wird. Dies spielt jedoch beim Melden-Prozess eine untergeordnete Rolle, da die Zustände des Meldungskonfigurationsobjekts manuell gesetzt werden.

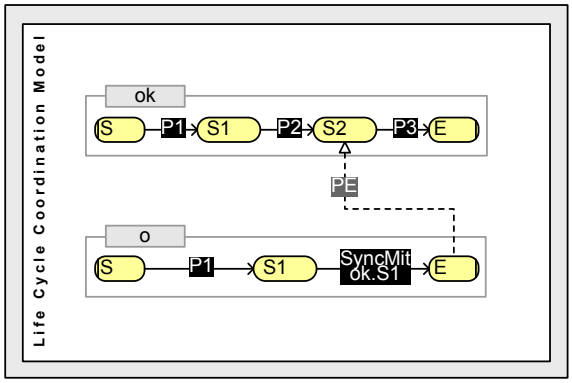


Abbildung 9.16: Schematische Darstellung der zeitkritischen Synchronisation

Das realisierte Teilszenario ist in Abbildung 9.17 abgebildet. Es zeigt zum einen den OLC für das Melden-Objekt, zum anderen die Verknüpfung mit dem Analyseprozess `isStateActivated`. Der Analyseprozess erwartet als Parameter den Relationstyp, mit dem das zu prüfende Objekt verknüpft ist[16], und die zu prüfenden Zustände. Im Beispiel aus Abbildung 9.17 prüft der Prozess `isStateActivated` das per Relation `RT melden` verknüpfte Statuskonfigurations-Objekt auf Aktivierung eines als Parameter übergebenen Zustands: `geplant`; `geplant, HW unverbindlich`; `geplant, HW+SW verbindlich`; `geplant, HW unverbindlich, HW+SW verbindlich` (vgl. Abbildung 9.15).

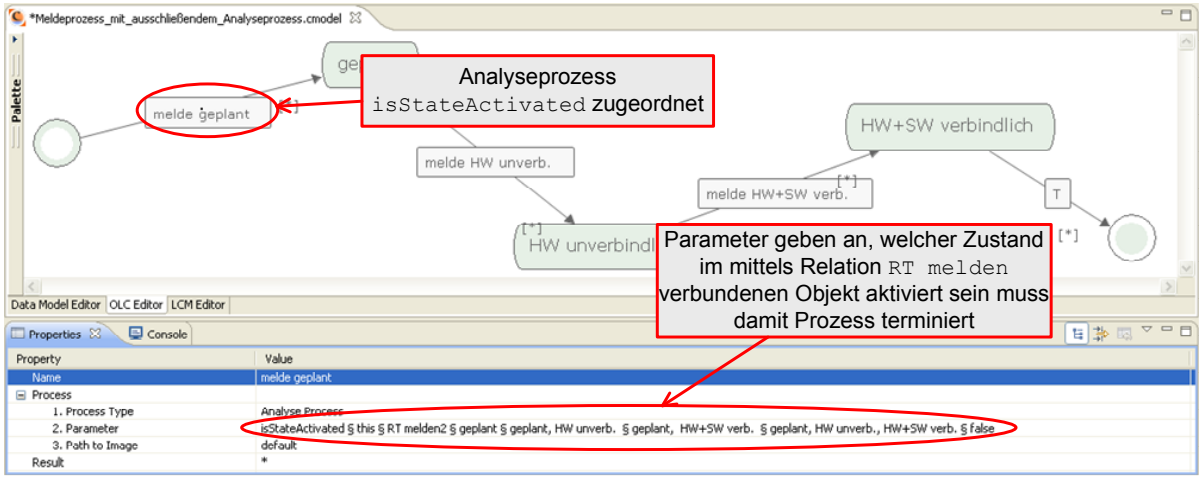


Abbildung 9.17: Realisierung des Melden-Szenarios mit Analyseprozess `isStateActivated`

[16] Zur Modellierzeit ist der Name der OLC-Instanz, mit der synchronisiert werden soll in der Regel noch nicht bekannt. Durch Angabe des Relationstyps kann der OLC zur Laufzeit dann eindeutig identifiziert werden (sofern die Relation nur einmal verwendet wird).

9.6 Zusammenfassung und Diskussion

Die Abbildung des Ablaufs aus dem zukünftigen Standard ISO 26262 hat gezeigt, dass sich die entwickelten Konzepte für die Modellierung von Entwicklungsprozessen komplexer Produkte eignen. Die intuitive Objekt- und Zustands-zentrierte Darstellung von Prozessstrukturen durch den CoRepro-Ansatz erlaubt die annähernd direkte Umsetzung des Prozessstandards. Die erzeugte Prozessstruktur hilft nicht nur, die enthaltenen Prozesse zu koordinieren, sondern garantiert dank der Konsistenzanalyse im Rahmen dynamischer Adaptionen und der Ausnahmebehandlung zugleich die Einhaltung der im Standard geforderten Abläufe.

Im Rahmen der Fallstudie wurde weiter die Abbildung des E/E-Release-Management-Prozesses untersucht.[17] Er stellt weitaus komplexere Anforderungen an die technische Unterstützung als der untersuchte Standard. Bei der Betrachtung des Release-Management-Prozesses mit statischer Konfiguration haben wir insbesondere die Abbildung der OLCs beschrieben, sowie die automatische Erzeugung der Prozessstruktur, die eine signifikante Reduktion des Modellierungsaufwands erlaubt. Bei der Betrachtung des Release-Management-Prozesses mit dynamischer Konfiguration haben wir gezeigt, dass dynamische Abläufe Datenmodelle mit der Abbildung von Versionierungsmechanismen und komplexe Synchronisationsmechanismen erfordern. Auch hierfür konnten schlüssige Lösungen für die Umsetzung mit den CoRepro-Konzepten gefunden werden. Die Konzepte zur Ausnahmebehandlung erlauben dem Endbenutzer fachliche Ausnahmen in der Prozessstruktur umzusetzen und unterstützen ihn bei der Wahl einer geeigneten Ausnahmebehandlung durch Anzeige der erwarteten Auswirkungen. Die Modellierung und Simulation der in diesem Kapitel gezeigten Szenarien im realisierten Demonstrator unterstreicht zudem die Mächtigkeit der praktischen Umsetzung der Konzepte in CoRepro$_{SIM}$.

Um die generelle Anwendbarkeit des Ansatzes zu prüfen, untersuchten wir in diesem Kapitel auch weitere komplexe Szenarien, deren Anforderungen über die in Kapitel 2 definierten Anforderungen hinausgehen. Die Abbildung mit den CoRepro-Konzepten konnte durch geeignete Mechanismen auch tatsächlich umgesetzt werden. Dazu gehören:

- *Kardinalitäten in Datenmodell / Datenstruktur*: Die Abbildung von Kardinalitäten für Beziehungen im Datenmodell bzw. der Datenstruktur wurde in den CoRepro-Konzepten nicht betrachtet, da sie in der Praxis direkt in einem Produktdaten-Management-System abgebildet und dort geprüft werden.[18] Eine geeignete Prüfung im Demonstrator wurde durch die Implementierung eines Prozesses zur Überprüfung der Kardinalität realisiert. Er kann über die zur Verfügung gestellte Schnittstelle der Datenstruktur die Relationen von oder zu einem Objekt prüfen und den Übergang zum Folgezustand erst dann erlauben, wenn die Kardinalität erfüllt ist. Die Implementierung des entsprechenden Prozesses `checkCardinality` befindet sich in Anhang B. Der Prozess garantiert zunächst, dass das Konsistenzkriterium zum Zeitpunkt der Prüfung erfüllt ist. Soll das Kriterium ab diesem

[17] Die umfangreichen Fallstudien haben Defizite in der Prozessdokumentation der E/E-Entwicklung aufgezeigt. Im Rahmen der Definition eines geeigneten SOLL-Prozesses wurde nicht nur der in Kapitel 9 vorgestellte Prozess mit den Konzepten abgebildet, sondern auch weitere Varianten des SOLL-Prozesses [WBS09]. Sie unterscheiden sich in den Anforderungen kaum von dem in Kapitel 9 vorgestellten Prozess und wurden daher nicht näher erläutert.

[18] Wie in Abschnitt 5.2.2 beschrieben, stellt die Datenstruktur eine Sicht auf in Produktdaten-Management-System verwaltete Produktstrukturen dar.

Zeitpunkt dauerhaft gewährleistet sein, ist dies durch geeignete Anwendung externer Transitionen ebenfalls realisierbar (vgl. Abschnitt 9.5.2).

- *Deadpath-Eliminierung bei Erreichen eines Fehlerzustands*: Die Abwahl von Pfaden kann, wie in Abschnitt 9.4.4 beschrieben, zu unerwünschtem Verhalten führen. Um dieses Verhalten zu umgehen, kann direkt im OLC eine interne Transition eingefügt werden, die bei Aktivierung des Fehlerzustands dafür sorgt, dass ausgehende externe Transitionen nicht abgewählt werden und damit die Möglichkeit der Ausnahmebehandlung erhalten bleibt. Die Möglichkeiten einer konzeptionellen Unterscheidung der externen Transitionen, in abwählbare und nicht abwählbare externe Transitionen kann in die CoRepro-Konzepte mit Anpassungen insbesondere der Ausführungsregel AR6 integriert werden.

Die E/E-Release-Management-Prozesse mit dynamischer Konfiguration erfordern in der Praxis teilweise noch weiter gehende Mechanismen. So kann es notwendig sein, eine Komponentenversion in einem Release zu nutzen, deren OLC bereits in einer *anderen* Prozessstruktur einen bestimmten Zustand erreicht hat oder sich dort noch in Ausführung befindet. Für die Abbildung dieses Szenarios kommen in CoRepro drei Alternativen in Frage:

- *Einfügen des OLC mit Initialmarkierung und Sprung auf den Zustand, der in der anderen Prozessstruktur für diesen OLC derzeit aktiviert ist (vollständig unterstützt)*: Der Durchlauf des OLC erfolgt damit transparent, aber unsynchronisiert. Das Objekt kann in unterschiedlichen Prozessstrukturen unterschiedliche Zustände annehmen.
- *Einfügen der OLC-Instanz aus der anderen Prozessstruktur (derzeit nicht unterstützt)*: Die Ausführung eines OLC in verschiedenen Prozessstrukturen ist hierbei zwar durch eine Erweiterung der Konzepte technisch lösbar, wäre jedoch äußerst intransparent. Wird beispielsweise ein OLC über externe Transitionen in unterschiedlichen Prozessstrukturen synchronisiert, ist die Ausführung mit etwaigen „Wartezeiten“ in den unterschiedlichen Prozessstrukturen nicht ersichtlich. Die Behandlung von Ausnahmen würde ebenso wesentlich komplexer, da sich Inkonsistenzen in unterschiedlichen Prozessstrukturen fortpflanzen können.
- *Bearbeitung aller Daten- und Prozessstrukturen innerhalb eines LCS (derzeit teilweise unterstützt)*: Die Synchronisation erfolgt transparent innerhalb *einer* Prozessstruktur, deren zugehörige Datenstruktur alle Releases enthält. Die Prozessstruktur wird jedoch sehr groß und damit unübersichtlich. Des Weiteren müssten hierfür die Terminierungskriterien (vgl. Abschnitt 4.6.4) unter Beachtung der Korrektheitskriterien (vgl. Abschnitt 4.7) angepasst werden.

Die Evaluierung der vorgestellten Möglichkeiten ist Teil zukünftiger Arbeiten.

Die vorgestellten Umsetzungen realer Abläufe zeigen, dass der CoRepro-Ansatz mächtig genug ist selbst Szenarien abzubilden, die neue Anforderungen mit sich bringen. Bei der Definition der Lösungen wurde darauf geachtet, die Elemente und Mechanismen des CoRepro-Modells zu nutzen. Damit bleiben die Vorteile des Konzepts erhalten.

Teil IV

Fazit

10

Zusammenfassung und Ausblick

Prozess-Management unterstützt Unternehmen in dem Vorhaben, ihre Unternehmensziele systematisch umzusetzen. Während die Unterstützung einzelner Prozesse durch Prozess-Management-Systeme bereits relativ ausgereift ist, wird die Koordination der modellierten und auszuführenden Prozesse im Unternehmen zunehmend komplexer. Um eine durchgängige Ablaufsteuerung zu realisieren, müssen die einzelnen Prozesse durch geeignete Abhängigkeiten miteinander verknüpft und bei der Ausführung bzw. Koordination der Prozesse berücksichtigt werden. In dem untersuchten Umfeld orientieren sich die entstehenden Prozessstrukturen an gegebenen Datenstrukturen (z.B. einer Produktstruktur). Die komplexe Beziehung zwischen Daten- und Prozessstruktur erfordert eine systematische Koordination der einzelnen Prozesse. Die integrierte Beschreibung einer derartigen *datengetriebenen Prozessstruktur* ist allerdings äußerst aufwändig und manuell kaum zu bewerkstelligen. Eine wichtige Erkenntnis der Arbeit ist, dass wir von dieser Beziehung zwischen Daten- und Prozessstruktur profitieren können. So liefert die Produktstruktur mit den enthaltenen Datenobjekten und deren Beziehungen bereits Anhaltspunkte für die Form der Prozessstruktur. Eine IT-Unterstützung kann diese Beziehung konsequent ausnutzen und damit nicht nur eine geeignete Modellierungsunterstützung bieten, sondern auch eine umfassende Kontrolle der Prozesse während der Ausführung ermöglichen.

Eine weitere Herausforderung mit der sich Prozess-Management im Umfeld der Entwicklung komplexer technischer Produkte konfrontiert sieht ist Flexibilität. Die starre Koordination der einzelnen Prozesse einer Prozessstruktur ist im betrachteten Umfeld nicht zielführend. Die Dynamik der untersuchten Domäne erfordert vielmehr flexible Reaktionen auf sich ändernde Rahmenbedingungen. Sie resultieren einerseits in Anforderungen an die Unterstützung der dynamischen Adaption großer Prozessstrukturen, andererseits an die flexible Steuerung des Ablaufs. Herausforderung ist es, eine Prozesskoordination zu realisieren, die durchgängig als Unterstützung der fachlichen Arbeit und nicht als deren Einschränkung verstanden wird. Existierende Produkte und wissenschaftliche Ansätze werden dem Anspruch der durchgängigen Unterstützung datengetriebener Prozessstrukturen jedoch nicht gerecht.

Beitrag

Die Modellierung datengetriebener Prozessstrukturen mittels gängiger aktivitätenorientierter Ansätze ist nicht möglich, da ihnen die geforderte Ausdrucksmächtigkeit und Strukturierbarkeit fehlt. Dies spiegelt sich zum einen in der unzureichenden Integration von Daten- und Prozessstrukturen, zum anderen in der nicht vorhandenen Unterstützung der Dynamik bezüglich der Änderungen der Produktstruktur und der damit notwendigen Adaption der Prozessstruktur wider.

Die vorliegende Arbeit liefert mit CoRePro ein durchgängiges Lösungskonzept zur Modellierung und Koordination großer datengetriebener Prozessstrukturen. Die Schwerpunkte lassen sich wie folgt zusammenfassen:

- **Integration von Daten und Prozessen**: Wir haben in dieser Arbeit ein formales Basismodell definiert, das die Beschreibung großer Prozessstrukturen unterstützt. Der Basisformalismus erlaubt die Kapselung der Prozesse jedes Objekts in sogenannten *Object Life Cycles* (OLCs) sowie deren Integration in Prozessstrukturen. Die zustandsbasierte Synchronisation der OLCs ermöglicht eine intuitive und zugleich „robuste“ Behandlung von Abhängigkeiten zwischen Prozessen. Die Modellierung großer Prozessstrukturen wird in CoRePro durch einen geeigneten Instanziierungsmechanismus unterstützt. Er erzeugt automatisch aus einer gegebenen Datenstruktur eine, im Sinne des Modells, zugehörige Prozessstruktur. Die CoRePro-Konzepte erlauben damit nicht nur eine signifikante Reduktion der Modellierungsaufwände. Sie garantieren auch die Korrektheit derartiger Prozessstrukturen *per Konstruktion* und sichern damit dem Ansatz eine sehr gute Skalierbarkeit zu.
- **Arbeit mit großen Prozessstrukturen**: Die Definition einer vollständigen operationalen Semantik erlaubt nicht nur die Ausführung von Prozessstrukturen, sondern auch präzise Aussagen bezüglich ihrer dynamischen Korrektheit. Die Arbeit definiert weiter geeignete Operationen für die dynamische Adaption datengetriebener Prozessstrukturen. Die Operationen stellen sicher, dass Änderungen der Datenstruktur auf Adaptionen der Prozessstruktur transformiert werden. Wir erlauben damit die Anpassung der Prozessstruktur auf hoher Abstraktionsebene und liefern einen wichtigen Beitrag für die Unterstützung datengetriebener Prozessstrukturen in der Praxis. Die Prüfung von Inkonsistenzen garantiert in Verbindung mit dem transaktionsbasierten Fehlerbehandlungsprozess weiterhin die Korrektheit der adaptierten Prozessstruktur.
- **Flexible Steuerung des Ablaufs**: Die Behandlung verschiedener Arten von Ausnahmen rundet die Konzepte zur Unterstützung datengetriebener Prozessstrukturen ab. Hierfür erlauben wir das Abweichen vom Kontrollfluss mittels geeigneter Sprungoperationen zur Behandlung fachlicher Ausnahmen. Dabei auftretende Inkonsistenzen der Laufzeitmarkierungen werden unabhängig von ihrer Entstehung abstrahiert und generische Operationen für ihre Behandlung bereitgestellt. Die Konzepte erlauben nicht nur die Herstellung der Konsistenz in einer Prozessstruktur, sondern unterstützen den Benutzer auch bei der Auswahl und Durchführung entsprechender Operationen. Hierfür kennzeichnen die Konzepte sowohl auftretende Inkonsistenzen, als auch deren Konsequenzen auf die gesamte Prozessstruktur. Ferner erlaubt der Mechanismus die Erzeugung eines Fehlerbehandlungsprozesses für die Herstellung der Konsistenz innerhalb der Prozessstruktur. In diesem Rahmen ist es wichtig, fachliche Entscheidungen nicht einzuschränken. Daher wird nicht nur das Auflösen

von Inkonsistenzen aus Sicht der Prozessstruktur unterstützt, sondern auch das (fachliche) Ignorieren derartiger Inkonsistenzen. Die Korrektheit der Prozessstruktur wird zu jedem Zeitpunkt garantiert.

- **Validation der Konzepte**: Die Konzepte sind aufeinander abgestimmt und zeichnen sich durch die durchgängige Unterstützung datengetriebener Prozessstrukturen über den ganzen Lebenszyklus aus. Die Lösungen dieses Ansatzes wurden in verschiedenen Veröffentlichungen dokumentiert und auf internationalen Konferenzen vorgestellt [MHHR06, MRH06, MRHP07, MRH07, MRH08a, MRH+08b]. Die Umsetzbarkeit wurde mit der Implementierung eines umfangreichen Software-Demonstrators, der die Lösungskonzepte vollständig unterstützt, nachgewiesen. Die praktische Anwendbarkeit der Konzepte konnte mit der Umsetzung verschiedener Prozesse im Rahmen einer aufwändigen Fallstudie untermauert werden. Während der Durchführung der Fallstudie hat sich bestätigt, dass die Modellierung von Prozessstrukturen basierend auf OLCs intuitiv und zügig umsetzbar ist.

Ausblick

Die intensive Auseinandersetzung mit datengetriebenen Prozessstrukturen sowie die im Rahmen der Fallstudie gewonnenen Erkenntnisse haben Fragestellungen aufgeworfen, die Themen für zukünftige Arbeiten liefern. So ergeben sich im operativen Einsatz weitere spannende Herausforderungen hinsichtlich Modellierungsmethodik, Visualisierung und Überwachung. Dazu gehören:

- Die Nutzung von Komponenten in verschiedenen Releases wird bislang nur unsynchronisiert unterstützt (vgl. Abschnitt 9.6). Die Bearbeitung aller Releases innerhalb einer Prozessstruktur kann durch eine Erweiterung der Terminierungskriterien ermöglicht werden.

- Die Abbildung des Release-Management-Prozesses hat gezeigt, dass Varianten in OLCs vorkommen können. Beispiele sind Komponenten mit und ohne HiL-Absicherung (vgl. Abschnitt 9.3.3). Zwar können wir diese Varianten über bedingte Verzweigungen oder die Modellierung verschiedener OLCs abbilden, bei komplexerer Variantenbildung kann es jedoch Sinn machen, hier weitergehende Mechanismen einzusetzen (vgl. [HBR08b, HBR08a]).

- Die Größe der betrachteten Prozessstrukturen führt zu unübersichtlichen Darstellungen. Die geeignete Visualisierung großer Prozessstrukturen für Endbenutzer kann beispielsweise durch eine nutzer- oder rollenspezifische Darstellung der Abläufe erfolgen. Außerdem können generelle Visualisierungskonzepte auf ihre Verwendbarkeit zur Reduktion oder Aggregation datengetriebener Prozessstrukturen untersucht werden (vgl. [BRB07, Bob08]).

- Die Komplexität der Fehlerbehandlung erfordert ebenfalls eine geeignete Visualisierung. Die Herausforderung ist es hier, die Konsequenzen nach Durchführung einer Änderung adäquat darzustellen und den Endbenutzer durch den Konsistenzherstellungsprozess zu führen (vgl. Abschnitt 7.3.3).

- Die Überwachung der Prozessausführung kann für die Erfassung und Berechnung von *Kennzahlen* genutzt werden. Im Umfeld von datengetriebenen Prozessstrukturen sind beispielsweise die Vorhersage der Dauer der Ausführung und die Bestimmung des *kritischen Pfads* relevant (vgl. [Mai04]).

Im Rahmen dieser Arbeit wurden bestimmte Aspekte gezielt ausgeklammert. Dazu gehören sowohl die dynamische Adaption von OLCs, als auch die Adaption der Modellebene (d.h. Schemaänderung) während der Ausführung (vgl. [Rin05]). Mit den Mechanismen zur dynamischen Adaption und der Konsistenzanalyse wurden mit CoreProo bereits wichtige Vorarbeiten für die Nutzung derartiger Konzepte für datengetriebene Prozessstrukturen geleistet.

Die Umsetzung des Standardprozesses aus der ISO 26262 Norm zeigt, dass der Ansatz universell einsetzbar ist. Die Unterstützung von Prozessstrukturen in weiteren technischen Entwicklungsprozessen (z.B. zur Unterstützung der Funktionsorientierung; vgl. [WH07, WJJ09]) und sonstigen Anwendungsdomänen (z.B. im Finanzsektor, Projektmanagement und medizinischen Umfeld; vgl. [HHH+09]) kann ebenfalls Thema weiterer Arbeiten sein.

Mit der vorliegenden Dissertation und den darin beschriebenen CoreProo-Konzepten wurde eine umfassende Unterstützung für das systematische und durchgängige Management datengetriebener Prozessstrukturen geschaffen. Die operative Umsetzung der Konzepte in der Industrie wurde bereits begonnen (vgl. [Neu09]) und bietet viel Potenzial für zukünftige Forschungsarbeiten.

Literaturverzeichnis

[Aal97a] Aalst, W. van der: Designing Workflows Based on Product Structures. In: *9th International Conference on Parallel and Distributed Computing Systems (IASTED)*, IASTED/Acta Press, 1997, S. 337–342

[Aal97b] Aalst, W. van der: Verification of Workflow Nets. In: *18th International Conference on Application and Theory of Petri Nets (ICATPN)* Bd. 1248, Springer, 1997 (LNCS), S. 407–426

[Aal98] Aalst, W. van der: The Application of Petri Nets to Workflow Management. In: *Circuits, Systems and Computers* 8 (1998), Nr. 1, S. 21–66

[Aal99] Aalst, W. van der: On the Automatic Generation of Workflow Processes Based on Product Structures. In: *Computers in Industry* 39 (1999), Nr. 2, S. 97–111

[Aal01] Aalst, W. van der: Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. In: *Information Systems Frontiers* 3 (2001), Nr. 3, S. 297–317

[AB97] Aalst, W. van der ; Basten, T.: Life-Cycle Inheritance: A Petri-Net-Based Approach. In: *18th International Conference on Application and Theory of Petri Nets (ICATPN)* Bd. 1248, Springer, 1997 (LNCS), S. 62–81

[AB01a] Aalst, W. van der ; Basten, T.: Identifying Commonalities and Differences in Object Life Cycles Using Behavioral Inheritance. In: *22nd International Conference on Application and Theory of Petri Nets (ICATPN)* Bd. 2075, Springer, 2001 (LNCS), S. 32–52

[AB01b] Aalst, W. van der ; Berens, P.: Beyond Workflow Management: Product-Driven Case Handling. In: *Conference on Supporting Group Work*, 2001, S. 42–51

[ABEW00] Aalst, W. van der ; Barthelmess, P. ; Ellis, C. ; Wainer, J.: Workflow Modeling using Proclets. In: *7th International Conference on Cooperative Information Systems (CoopIS)* Bd. 1901, Springer, 2000 (LNCS), S. 198–209

[ACKM04] Alonso, G. ; Casati, F. ; Kuno, H. ; Machiraju, V.: *Web Services - Concepts, Architectures and Applications*. Springer, 2004 (Data-Centric Systems and Applications Series). – ISBN 3540440089

[ADEK05] Arnold, V. ; Dettmering, H. ; Engel, T. ; Karcher, A.: *Product Lifecycle Management beherrschen*. Springer, 2005. – ISBN 3540229973

[ADH+03] Aalst, W. van der ; Dongen, B. van ; Herbst, J. ; Maruster, L. ; Schimm, G. ; Weijters, A.: Workflow Mining: A Survey of Issues and Approaches. In: *Data and Knowledge Engineering* 47 (2003), Nr. 2, S. 237–267

[AH05] Aalst, W. van der ; Hofstede, A. ter: YAWL: Yet Another Workflow Language. In: *Information Systems* 30 (2005), Nr. 4, S. 245–275

[AHD05] Aalst, W. van der ; Hofstede, A. ter ; Dumas, M.: Patterns of Process Modeling. In: Dumas, Marlon (Hrsg.) ; Aalst, W. van der (Hrsg.) ; Hofstede, A. ter (Hrsg.): *Process-Aware Information Systems.* John Wiley & Sons, 2005. – ISBN 0471663069, S. 343–362

[AHKB03] Aalst, W. van der ; Hofstede, A. ter ; Kiepuszewski, B. ; Barros, A.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Nr. 1, S. 5–51

[ARL01] Aalst, W. van der ; Reijers, H. ; Limam, S.: Product-driven Workflow Design. In: *6th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, NRC Research Press, 2001, S. 397–402

[ASW03] Aalst, W. van der ; Stoffele, M. ; Wamelink, J.: Case Handling in Construction. In: *Automation in Construction* 12 (2003), Nr. 3, S. 303–320

[AWG05] Aalst, W. van der ; Weske, M. ; Grünbauer, D.: Case Handling: A New Paradigm for Business Process Support. In: *Data and Knowledge Engineering* 53 (2005), Nr. 2, S. 129–162

[BBK98] Broy, M. ; Beeck, M. von d. ; Krüger, I.: SOFTBED: Problemanalyse für das Großverbundprojekt „Systemtechnik Automobil - Software für eingebettete Systeme“ / Technische Universität München. 1998. – Technischer Bericht

[BD05] Born, A. ; Diercks, J.: Definitionsfrage - BPM: Zukunftsfähig mit flexiblen Abläufen. In: *iX* 12 (2005), S. 110–117

[BDS98] Beuter, T. ; Dadam, P. ; Schneider, P.: The WEP Model: Adepquate Workflow-Management for Engineering Processes. In: *5th European Concurrent Engineering Conference (ECEC).* Erlangen, Germany, 1998, S. 94–98

[Beu03] Beuter, T.: *Workflow-Management für Produktentwicklungsprozesse*, Universität Ulm, Dissertation, 2003

[BGH+07] Bhattacharya, K. ; Gerede, C. ; Hull, R. ; Liu, R. ; Su, J.: Towards Formal Analysis of Artifact-Centric Business Process Models. In: *5th International Conference on Business Process Management* Bd. 4714, Springer, 2007 (LNCS), S. 288–304

[BHR05] Bestfleisch, U. ; Herbst, J. ; Reichert, M.: Requirememts For The Workflow-Based Support of Release Management Processes in the Automotive Sector. In: *12th European Concurrent Engineering Conference (ECEC)*, 2005, S. 130–134

[BMNW05] Breu, R. ; Matzner, T. ; Nickl, F. ; Wiegert, O.: *Software-Engineering - Objektorientierte Techniken, Methoden und Prozesse in der Praxis.* Oldenbourg, 2005. – ISBN 3486575740

[Bob08] Bobrik, R.: *Konfigurierbare Visualisierung komplexer Prozessmodelle*, Universität Ulm, Dissertation, 2008

[Boo90] Booch, G.: *Object-Oriented Design with Applications.* Benjamin-Cummings Publishing, 1990. – ISBN 8131702324

[BP07] Binder, K. ; Philipp, S.: *PLM Wissen kompakt 2006/2007.* IT & Production - Zeitschrift für industrielle Informationstechnologie, 2007

[BR05] Broy, M. ; Rausch, A.: Das neue V-Modell XT. In: *Informatik Spektrum* 28 (2005), Nr. 3, S. 220–229

[BRB07] Bobrik, R. ; Reichert, M. ; Bauer, T.: View-Based Process Visualization. In: *5th International Conference on Business Process Management* Bd. 4714, Springer, 2007 (LNCS), S. 88–95

[Bro05] Brown, C.: Workflow Management in Staffware. In: Dumas, Marlon (Hrsg.) ; Aalst, W. van der (Hrsg.) ; Hofstede, A. ter (Hrsg.): *Process-Aware Information Systems.* John Wiley & Sons, 2005. – ISBN 0471663069, S. 343–362

[Bus06] Business Process Management Initiative (BPMI): *Business Process Modeling Notation Specification (BPMN) 1.0: OMG Final Adopted Specification.* 2006

[Büt06] Büttler, A.: *Visualisierung und Modellierung komplexer Produktentwicklungsprozesse in der Automobilindustrie unter dem Aspekt der Modularisierung*, Universität Ulm, Diplomarbeit, 2006

[CAPD03] Crnkovic, I. ; Asklund, U. ; Persson-Dahlqvist, A.: *Implementing and Integrating Product Data Management and Software Configuration Management.* Artech House Publishers, 2003. – ISBN 1580534988

[CCPP98] Casati, F. ; Ceri, S. ; Pernici, B. ; Pozzi, G.: Workflow Evolution. In: *Data and Knowledge Engineering* 24 (1998), Nr. 3, S. 211–238

[Che76] Chen, P.: The Entity-Relationship Model - Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), Nr. 1, S. 9–36

[CKLY98] Cortadella, J. ; Kishinevsky, M. ; Lavagno, L. ; Yakovlev, A.: Deriving Petri Nets for Finite Transition Systems. In: *IEEE Transactions on Computers* 47 (1998), Nr. 8, S. 859–882

[DAH05] Dumas, M. (Hrsg.) ; Aalst, W. van der (Hrsg.) ; Hofstede, A. ter (Hrsg.): *Process-aware Information Systems: Bridging People and Software through Process Technology.* John Wiley & Sons, Inc. New York, NY, USA, 2005. – ISBN 0471663069

[Dai02] DaimlerChrysler AG, Research and Technology: Programme für mehr Intelligenz im Auto. In: *Hightech Report* (2002), Nr. 01, S. 46–49

[Des05] Desel, J.: Process Modeling Using Petri Nets. In: Dumas, Marlon (Hrsg.) ; Aalst, W. van der (Hrsg.) ; Hofstede, A. ter (Hrsg.): *Process-Aware Information Systems*. John Wiley & Sons, 2005. – ISBN 0471663069, S. 147–177

[Deu99] Deutsches Institut für Normung und Verband der Elektrotechnik Elektronik Informationstechnik: *DIN EN 61508-3 - Funktionale Sicherheit sicherheitsbezogener elektrischer / elektronischer / programmierbarer elektronischer Systeme - Teil 3: Anforderungen an Software*. 1999

[DH01] Dumas, M. ; Hofstede, A. ter: UML Activity Diagrams as a Workflow Specification Language. In: *4th International Conference on the Unified Modeling Language: Modeling Languages, Concepts, and Tools* (2001)

[Dor00] Dori, D.: Object-Process Methodology as a Business-Process Modelling Tool. In: *8th European Conference on Information Systems*, 2000

[DR09] Dadam, P. ; Reichert, M.: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements. In: *Computer Science - Research and Development* 23 (2009), Nr. 2

[Drö00] Dröschel, W.: *Das V-Modell 97*. Oldenbourg, 2000

[DRRM+09] Dadam, P. ; Reichert, M. ; Rinderle-Ma, S. ; Goeser, K. ; Kreher, U. ; Jurisch, M.: Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: „Correctness by Construction“ und flexible, robuste Ausführung von Unternehmensprozessen. In: *EMISA Forum* 29 (2009), Nr. 1, S. 9–28

[Ecl] *Eclipse*. `http://www.eclipse.org/`. – Zuletzt besucht am 05.03.2009

[EE97] Ebert, J. ; Engels, G.: Specialization of Object Life Cycle Definitions / Universität Koblenz-Landau. 1997. – Technischer Bericht

[EJ01] Endig, M. ; Jesko, D.: Engineering Processes - An Approach To Realize A Dynamic Process Control. In: *Journal of Integrated Design and Process Science* 5 (2001), Nr. 2, S. 65–82

[Ell97] Ellis, C.: Team Automata for Groupware Systems. In: *International Conference on Supporting Group Work*, ACM, 1997, S. 415–424

[Elm92] Elmargarmid, A.: *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992. – ISBN 1558602143

[EPC] *Event Driven Process Chains*. `http://wwwcs.uni-paderborn.de/cs/kindler/research/EPCTools/`. – Zuletzt besucht am 05.03.2009

[EPF] *Eclipse Process Framework*. `http://www.eclipse.org/epf/`. – Zuletzt besucht am 05.03.2009

[FE98] Frank, H. ; Eder, J.: Integration of Statecharts. In: *3rd International Conference on Cooperative Information Systems (IFCIS)*, IEEE Computer Society, 1998, S. 364–372

[FWM+04] Felfernig, A. ; Wimmer, A. ; Mehlau, J. ; Russ, C. ; Zanker, M.: Konzepte zur flexiblen Konfiguration von Finanzdienstleistungen. In: *Multikonferenz Wirtschaftsinformatik (MKWI)*, 2004

[GCG04] Grigori, D. ; Charoy, F. ; Godart, C.: Coo-Flow: A Process Technology To Support Cooperative Processes. In: *Software Engineering and Knowledge Engineering (IJSEKE)* 14 (2004), Nr. 1, S. 61–78

[GEF] *Graphical Editing Framework (GEF)*. `http://www.eclipse.org/gef/`. – Zuletzt besucht am 05.03.2009

[GHJV95] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns*. Addison-Wesley Professional, 1995. – ISBN 0201633612

[GHS95] Georgakopoulos, D. ; Hornick, M. ; Sheth, A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. In: *Distributed and Parallel Databases* 3 (1995), S. 119–153

[Gös05] Göser, K.: *Plug&Play-Aspekte und Integration zustandsbehafteter Daten in Prozess-Management-Systeme*, Universität Ulm, Diplomarbeit, 2005

[GRMR+08] Günther, C. ; Rinderle-Ma, S. ; Reichert, M. ; Aalst, W. van der ; Recker, J.: Using Process Mining to Learn from Process Changes in Evolutionary Systems. In: *Business Process Integration and Management* 3 (2008), Nr. 1, S. 61–78

[GRRA06] Günther, C. ; Rinderle, S. ; Reichert, M. ; Aalst, W. van der: Change Mining in Adaptive Process Management Systems. In: *14th International Conference on Cooperative Information Systems (CoopIS)* Bd. 4275, Springer, 2006 (LNCS), S. 309–326

[GS98] Goltz, M. ; Schmitt, R.: SIMNET - Workflow Management for Simultaneous Engineering Networks / Technische Universität Clausthal, Institut für Maschinenwesen (IMV). 1998. – Technischer Bericht. – 97–100 S.

[Har87] Harel, D.: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Nr. 3, S. 231–274

[HBR08a] Hallerbach, A. ; Bauer, T. ; Reichert, M.: Anforderungen an die Modellierung und Ausführung von Prozessvarianten. In: *Datenbank Spektrum* 24 (2008), S. 48–58

[HBR08b] Hallerbach, A. ; Bauer, T. ; Reichert, M.: Managing Process Variants in the Process Life Cycle. In: *10th International Conference on Enterprise Information Systems (ICEIS)*, 2008, S. 154–161

[Hei01] Heinlein, C.: Workflow and Process Synchronization with Interaction Expressions and Graphs. In: *16th International Conference on Data Engineering (ICDE)*, 2001, S. 243–252

[HFS04] Heinisch, C. ; Feil, V. ; Simons, M.: Efficient Configuration Management of Automotive Software. In: *2nd European Congress on Embedded Real Time Software*, 2004

[HG97] Harel, D. ; Gery, E.: Executable Object Modeling with Statecharts. In: *IEEE Computer* 30 (1997), Nr. 7, S. 31–42

[HHH+09] Hee, K. van ; Hidders, J. ; Houben, G.-J. ; Paredaens, J. ; Thiran, P.: On the Relationship Between Workflow Models and Document Types. In: *Information Systems* 34 (2009), Nr. 1, S. 178–208

[HLN+90] Harel, D. ; Lachover, H. ; Naamad, A. ; Pnueli, A. ; Politi, M. ; Sherman, R. ; Shtull-Trauring, A. ; Trakhtenbrot, M.: STATEMATE: A Working Environment for the Development of Complex Reactive Systems. In: *IEEE Transactions on Software Engineering* 16 (1990), Nr. 4, S. 403–414

[HMP08] Herbst, J. ; Müller, D. ; Pounarov, B.: *Prozessanalyse Konfigurationsbildung und Absicherung in der E/E-Entwicklung.* Daimler AG, 2008. – Interner Bericht

[HN96] Harel, D. ; Naamad, A.: The STATEMATE Semantics of Statecharts. In: *ACM Transactions on Software Engineering and Methodology* 5 (1996), Nr. 4, S. 293–333

[Hoa78] Hoare, C.: Communicating Sequential Processes. In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677

[HRKH04] Herbst, J. ; Rodefeld, C. ; König, C. ; Handke, N.: *Technologiemonitoring Process Integration Electric/Electronics - Mechanics - Software - Themengebiet: Freigabe und Änderungsmanagement.* DaimlerChrysler AG, 2004. – Interner Bericht

[Inn07] *MID Innovator 2007 Java API.* 2007

[Int08] Internationale Organisation für Normung (ISO): *ISO 26262: Road Vehicles - Functional Safety (Committee Draft).* 2008

[Jac75] Jackson, M.: *Principles of Program Design.* London : Academic Press, 1975. – ISBN 0123790506

[JE00] Jesko, D. ; Endig, M.: Integration von Prozessmodellierungsmethoden im Rahmen einer prozesszentrierten Entwurfsumgebung. In: *2. Workshop zu komponentenorientierte betriebliche Anwendungssysteme (WKBA)*, 2000, S. 57–68

[Jen86] Jensen, K.: Colored Petri Nets. In: *LNCS Advances in Petri Nets* 254 (1986), S. 248–299

[JRH+04] Jeckle, M. ; Rupp, C. ; Hahn, J. ; Zengler, B. ; Queins, S.: *UML 2 glasklar.* Hanser Fachbuchverlag, 2004. – ISBN 3446225757

[JSW99] Jäger, D. ; Schleicher, A. ; Westfechtel, B.: AHEAD: A Graph-Based System for Modeling and Managing Development Processes. In: *International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, 1999 (LNCS 1779), S. 325–339

[Jun08] Jung, C.: ISO WD 26262: Der zukünftige Standard zur Funktionssicherheit in der Automobilindustrie. In: *Elektronik-Systeme im Automobil* Euroforum, 2008

[KGPW00] Krücke, L. ; Gondermann, J. ; Pera, H. ; Wetzel, W.: Automotive Networks, From Idea to Silicon: K-Bus, LIN, CAN, VAN, byteflight. In: *VDI-Berichte*, 2000

[KKSW98] Krapp, C.-A. ; Krüppel, S. ; Schleicher, A. ; Westfechtel, B.: Graph-based Models for Managing Development Processes, Resources, and Products. In: *6th International Workshop on Theory and Application of Graph Transformations (TAGT)*, 1998, S. 455–474

[Kle03] Kleijn, J.: Team Automata for CSCW - A Survey. In: *Petri Net Technology for Communication Based Systems (Advances in Petri Nets)* Bd. 2472, Springer, 2003 (LNCS), S. 295–320

[KMSOW06] Kiener, S. ; Maier-Scheubeck, N. ; Obermaier, R. ; Weiss, M.: *Produktions-Management*. Oldenbourg Wissenschaftsverlag, 2006. – ISBN 3486580590

[KP88] Krasner, G. ; Pope, S.: A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. In: *Journal of Object-Oriented Programming* 1 (1988), Nr. 3, S. 26–49

[Krä07] Krämer, Michael: *Die neue C-Klasse von Mercedes-Benz: Entwicklung und Technik*. Vieweg+Teubner Verlag, 2007. – ISBN 3834803200

[KRG07] Küster, J. ; Ryndina, K. ; Gall, H.: Generation of Business Process Models for Object Life Cycle Compliance. In: *5th International Conference on Business Process Management* Bd. 4714, 2007 (LNCS), S. 165–181

[KS91] Kappel, G. ; Schrefl, M.: Object / Behavior Diagrams. In: *7th International Conference on Data Engineering (ICDE)* (1991), S. 530–539

[KS04] Knippel, E. ; Schulz, A.: Lessons Learned from Implementing Configuration Management within Electrical/Eletronic Development of an Automotive OEM. In: *14th International Council on Systems Engineering (INCOSE)*, 2004

[Las61] Lasser, D.: Topological Ordering of a List of Randomly-numbered Elements of a Network. In: *Communications of the ACM* 4 (1961), Nr. 4, S. 167–168

[LBW07] Liu, R. ; Bhattacharya, K. ; Wu, F.: Modeling Business Contexture and Behavior Using Business Artifacts. In: *19th International Conference on Advanced Information Systems Engineering (CAiSE)* Bd. 4495, Springer, 2007 (LNCS), S. 324–339

[LK94] Lawson, M. ; Karandikar, H.: A Survey of Concurrent Engineering. In: *Concurrent Engineering: Research and Applications* 2 (1994), Nr. 1, S. 1–6

[LR00] Leymann, F. ; Roller, D.: *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, 2000. – ISBN 0130217530

[LRW08a] Li, C. ; Reichert, M. ; Wombacher, A.: Discovering Reference Process Models by Mining Process Variants. In: *6th International Conference on Web Services (ICWS)*, IEEE Computer Society, 2008, S. 45–53

[LRW08b] Li, C. ; Reichert, M. ; Wombacher, A.: Mining Based on Learning from Process Change Logs. In: *4th International Workshop on Business Process Intelligence (BPI)*, 2008

[Mai04] Maier, A.: *Visualisierung dynamischer, komplexer Projektpläne in der Produktentwicklung*, Universität Ulm, Diplomarbeit, 2004

[MCL+96] Munch, B. ; Conradi, R. ; Larsen, J.-O. ; Nguyen, M. ; Westby, P.: Integrated Product and Process Management in EPOS. In: *Integrated Computer-Aided Engineering* 3 (1996), Nr. 1, S. 5–19

[MFS82] Mohan, C. ; Fussell, D. ; Silberschatz, A.: Compatibility and Commutativity in Non-two-phase Locking Protocols. In: *Symposium on Principles of Database Systems (PODS)*, 1982, S. 283–292

[MHHR06] Müller, D. ; Herbst, J. ; Hammori, M. ; Reichert, M.: IT Support for Release Management Processes in the Automotive Industry. In: *4th International Conference on Business Process Management* Bd. 4102, Springer, 2006 (LNCS), S. 368–377

[Mil99] Milner, R.: *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999. – ISBN 0521658691

[MRB08] Mutschler, B. ; Reichert, M. ; Bumiller, J.: Unleashing the Effectiveness of Process-Oriented Information Systems: Problem Analysis, Critical Success Factors, and Implications. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C* 38 (2008), Nr. 3, S. 280–291

[MRH06] Müller, D. ; Reichert, M. ; Herbst, J.: Flexibility of Data-Driven Process Structures. In: *BPM'06 International Workshops, Workshop on Dynamic Process Management* Bd. 4103, Springer, 2006 (LNCS), S. 181–192

[MRH07] Müller, D. ; Reichert, M. ; Herbst, J.: Data-Driven Modeling and Coordination of Large Process Structures. In: *15th International Conference on Cooperative Information Systems (CoopIS)* Bd. 4803, Springer, 2007 (LNCS), S. 131–149

[MRH08a] Müller, D. ; Reichert, M. ; Herbst, J.: A New Paradigm for the Enactment and Dynamic Adaptation of Data-driven Process Structures. In: *20th International Conference on Advanced Information Systems Engineering (CAiSE)* Bd. 5074, Springer, 2008 (LNCS), S. 48–63

[MRH+08b] Müller, D. ; Reichert, M. ; Herbst, J. ; Köntges, D. ; Neubert, A.: COREPRO$_{Sim}$: A Tool for Modeling, Simulating and Adapting Data-driven Process Structures. In: *6th International Conference on Business Process Management (Demonstration Program)* Bd. 5240, Springer, 2008 (LNCS), S. 394–397

[MRHP07] Müller, D. ; Reichert, M. ; Herbst, J. ; Poppa, F.: Data-Driven Design of Engineering Processes with COREPRO$_{Modeler}$. In: *16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE Computer Society, 2007, S. 376–378

[Mül06] Müller, D.: *Prototyping und Fallbeispiel OptinSolutions DBPM.* Daimler AG, 2006. – Interner Bericht

[MWR08] Mutschler, B. ; Weber, B. ; Reichert, M.: Workflow Management Versus Case Handling: Results from a Controlled Software Experiment. In: *ACM Symposium on Applied Computing*, 2008, S. 82–89

[MWRR06] Mentink, R. ; Wijnker, D. ; Reichert, M. ; Reijers, H.: OptIn DBPM: An Information-centric Approach to Business Process Management / OptIn Solutions B.V. 2006. – Technischer Bericht

[NC03] Nigam, A. ; Caswell, N.: Business Artifacts: An Approach to Operational Specification. In: *IBM Systems Journal* 42 (2003), Nr. 3, S. 428–445

[Neu09] Neubert, A.: *Abbildung datengetriebener Prozessstrukturen in Prozess Management Systemen*, Universität Ulm, Diplomarbeit, 2009. – In Bearbeitung

[NK08] Neubert, A. ; Köntges, D.: *Datengetriebene Ausführung von Entwicklungsprozessen im Automobilbereich.* Universität Ulm, 2008. – Praktikumsbericht

[NWC97] Nguyen, M. ; Wang, A. ; Conradi, R.: Total Software Process Model Evolution in EPOS (Experience Report). In: *19th International Conference on Software Engineering (ICSE)*, 1997, S. 390–399

[Obj97] Object Management Group (OMG): *Unified Modeling Language (UML) 1.0.* 1997

[Obj03a] Object Management Group (OMG): *Model Driven Architecture MDA Guide 1.0.1.* 2003

[Obj03b] Object Management Group (OMG): *Unified Modeling Language (UML) 2.0.* 2003

[Org07] Organization for the Advancement of Structured Information Standards (OASIS): *Web Services Business Process Execution Language (WS-BPEL) 2.0.* 2007

[Par98] Partsch, H.: *Requirements-Engineering systematisch: Modellbildung für softwaregestützte Systeme.* Springer, 1998. – ISBN 3540643915

[PDAC+01] Persson-Dahlqvist, A. ; Asklund, U. ; Crnkovic, I. ; H., A. ; Larsson, M. ; Ranby, J. ; Svensson, D.: Product Data Management and Software Configuration Management. In: *The Association of Swedish Engineering Industries*, 2001

[Pet62] Petri, C.: *Kommunikation mit Automaten*, Universität Bonn, Dissertation, 1962

[PG05] Putte, G. van d. ; Gavin, L.: *Technical Overview of WebSphere Process Server and WebSphere Integration Developer*. IBM International Technical Support Organization, 2005

[Pro] *Protege*. `http://protege.stanford.edu/`. – Zuletzt besucht am 05.03.2009

[PS98] Preuner, G. ; Schrefl, M.: Observation Consistent Integration of Views of Object Life-Cycles. In: *16th British National Conference on Advances in Databases* Bd. 1405, Springer, 1998 (LNCS), S. 32–48

[PSA07] Pesic, M. ; Schonenberg, H. ; Aalst, W. van der: DECLARE: Full Support for Loosely-Structured Processes. In: *11th International Conference on Enterprise Distributed Object Computing (EDOC)*, IEEE Computer Society, 2007, S. 287–300

[PSSA07] Pesic, M. ; Schonenberg, M. ; Sidorova, N. ; Aalst, W. van der: Constraint-Based Workflow Models: Change Made Easy. In: *15th International Conference on Cooperative Information Systems (CoopIS)* Bd. 4803, Springer, 2007 (LNCS), S. 77–94

[RAH06] Russell, N. ; Aalst, W. van der ; Hofstede, A. ter: Workflow Exception Patterns. In: Dubois, Eric (Hrsg.) ; Pohl, Klaus (Hrsg.): *18th International Conference on Advanced Information Systems Engineering (CAiSE)* Bd. 4001, Springer, 2006 (LNCS), S. 288–302

[RC03a] Rouibah, K. ; Caskey, K.: Change Management in Concurrent Engineering from a Parameter Perspective. In: *Computers in Industry* 50 (2003), Nr. 1, S. 15–34

[RC03b] Rouibah, K. ; Caskey, K.: A Workflow System for the Management of Intercompany Collaborative Engineering Processes. In: *Engineering Design* 14 (2003), Nr. 3, S. 273–293

[RD98] Reichert, M. ; Dadam, P.: ADEPT$_{flex}$: Supporting Dynamic Changes of Workflow without Losing Control. In: *Journal of Intelligent Information Systems* 10 (1998), Nr. 2, S. 93–129

[RDHI07] Redding, G. ; Dumas, M. ; Hofstede, A. ter ; Iordachescu, A.: Transforming Object-Oriented Models to Process-Oriented Models. In: *Business Process Management Workshops*, 2007, S. 132–143

[RDHI08] Redding, G. ; Dumas, M. ; Hofstede, A. ter ; Iordachescu, A.: Generating Business Process Models from Object Behavior Models. In: *Information Systems Management* 25 (2008), Nr. 4, S. 319–331

[Rei00] Reichert, M.: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*, Universität Ulm, Dissertation, 2000

[Rei02] Reijers, H.: *Design and Control of Workflow Processes - Business Process Management for the Service Industry*, Eindhoven University of Technology, Dissertation, 2002

[Rei07] Reif, K.: *Automobilelektronik*. Bd. 2. Auflage. Vieweg Verlag, 2007. – ISBN 3834804460

[RHAM06] Russell, N. ; Hofstede, A. ter ; Aalst, W. van der ; Mulyar, N.: Workflow Control-Flow Patterns - A Revised View / BPMCenter. 2006 (BPM-06-22). – Technischer Bericht

[RHEA04] Russell, N. ; Hofstede, A. ter ; Edmond, D. ; Aalst, W. van der: Workflow Data Patterns / Queensland University of Technology. 2004. – Technischer Bericht

[RHEA05] Russell, N. ; Hofstede, A. ter ; Edmond, D. ; Aalst, W. van der: Workflow Data Patterns: Identification, Representation and Tool Support. In: *24th International Conference on Conceptual Modeling (ER05)* 3716 (2005), S. 353–368

[Rin05] Rinderle, S.: *Schema Evolution in Process Management Systems*, Universität Ulm, Dissertation, 2005

[RKG06] Ryndina, K. ; Küster, J. ; Gall, H.: Consistency of Business Process Models and Object Life Cycles. In: *International Workshops on Model Driven Engineering Languages and Systems (MoDELS)* Bd. 4364, 2006 (LNCS), S. 80–90

[RKG07] Ryndina, K. ; Küster, J. ; Gall, H.: A Tool for Integrating Object Life Cycle and Business Process Modeling. In: *5th International Conference on Business Processs Management (Demonstration Program)* Bd. 272, CEUR-WS.org, 2007

[RLA03] Reijers, H. ; Limam, S. ; Aalst, W. van der: Product-based Workflow Design. In: *Management Information systems* 20 (2003), Nr. 1, S. 229–262

[RMRW08] Rinderle-Ma, S. ; Reichert, M. ; Weber, B.: On the Formal Semantics of Change Patterns in Process-Aware Information Systems. In: *27th International Conference on Conceptual Modeling (ER)* Bd. 5231, Springer, 2008 (LNCS), S. 279–293

[RPB03] Reichert, M. ; P.Dadam ; Bauer, T.: Dealing with Forward and Backward Jumps in Workflow Management Systems. In: *Software and Systems Modeling* 2 (2003), Nr. 1, S. 37–58

[RR06] Rinderle, S. ; Reichert, M.: Data-Driven Process Control and Exception Handling in Process Management Systems. In: *18th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2006 (LNCS 4001), S. 273–287

[RRA07] Rouibah, K. ; Rouibah, S. ; Aalst, W. van der: Combining Workflow and PDM Based on the Workflow Management Coalition and STEP standards: The Case of Axalant. In: *Computer Integrated Manufacturing* 20 (2007), Nr. 8, S. 811–827

[RRD04a] Rinderle, S. ; Reichert, M. ; Dadam, P.: Correctness Criteria For Dynamic Changes in Workflow Systems: A Survey. In: *Data and Knowledge Engineering* 50 (2004), Nr. 1, S. 9–34

[RRD04b] Rinderle, S. ; Reichert, M. ; Dadam, P.: Flexible Support of Team Processes by Adaptive Workflow Systems. In: *Distributed & Parallel Databases* 16 (2004), Nr. 1, S. 91–116

[RRKD05] Reichert, M. ; Rinderle, S. ; Kreher, U. ; Dadam, P.: Adaptive Process Management with ADEPT2. In: *International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 2005, S. 1113–1114

[RWR06a] Rinderle, S. ; Wombacher, A. ; Reichert, M.: Evolution of Process Choreographies in DYCHOR. In: *14th International Conference on Cooperative Information Systems (CoopIS)* Bd. 4275, Springer, 2006 (LNCS), S. 273–290

[RWR06b] Rinderle, S. ; Wombacher, A. ; Reichert, M.: On the Controlled Evolution of Process Choreographies. In: *22nd International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 2006, S. 124

[Sch02] Schichtel, M.: *Produktdatenmodellierung in der Praxis.* Fachbuchverlag Leipzig, 2002. – ISBN 3446218572

[Som07] Sommerville, I.: *Software Engineering.* 8. Addison Wesley, 2007. – ISBN 3827372577

[SR93] Sheth, A. ; Rusinkiewicz, M.: On Transactional Workflows. In: *IEEE Data Engineering Bulletin* 16 (1993), Nr. 2, S. 37–40

[SS01] Schuh, G. ; Schwenk, U.: *Produktkomplexität managen : Strategien - Methoden - Tools.* Hanser, 2001. – ISBN 3446400435

[SS02] Schrefl, M. ; Stumptner, M.: Behavior-consistent Specialization of Object Life Cycles. In: *ACM Transactions on Software Engineering and Methodology* 11 (2002), Nr. 1, S. 92–148

[SS07] Schmelzer, H. ; Sesselmann, W.: *Geschäftsprozessmanagement in der Praxis.* Hanser Fachbuch, 2007. – ISBN 3446410023

[SW02] Schuschel, H. ; Weske, M.: Fallbehandlung: Ein neuer Ansatz zur Unterstützung Prozessorientierter Informationssysteme. In: *Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen* Bd. 21, GI, 2002 (LNI), S. 52–63

[SW08] Spath, D. ; Weisbecker, A.: Business Process Management Tools 2008 - Eine evaluierende Markstudie zu aktuellen Werkzeugen / Fraunhofer - Institut für Arbeitswirtschaft und Organisation. 2008. – Technischer Bericht

[Tan07] Tanneberger, V.: Herausforderungen an die Fahrzeugelektronik aus Sicht der OEMs. In: *VDI Berichte* 2000 (2007), S. 3 – 15

[TcE06] *UGS Teamcenter Engineering Help Library.* 2006

[Tee96] Teege, G.: HieraStates: Supporting Workflows which Include Schematic and Adhoc Aspects. In: *1st International Conference on Practical Aspects of Knowledge Management (PAKM)* 96 (1996)

[Tee98] Teege, G.: Flexible Workflows: Mitgestaltung durch die Ausführenden. In: *Workshop Flexibilität und Kooperation in Workflow-Management-Systemen* (1998), S. 13–21

[V-M05] *V-Modell XT*. `http://www.v-modell-xt.de/`. Version: 2005. – Zuletzt besucht am 05.03.2009

[Van09] Vanderfeesten, I.: *Product-Based Design and Support of Workflow Processes*, Eindhoven University of Technology, Dissertation, 2009

[Ver04] Verband Deutscher Ingenieure (VDI): *VDI-Richtlinie 2206: Entwicklungsmethodik für mechatronische Systeme*. 2004

[VRA07] Vanderfeesten, I. ; Reijers, H. ; Aalst, W. van der: Case Handling Systems as Product Based Workflow Design Support. In: *9th International Conference on Enterprise Information Systems (ICEIS) - Selected Papers*, Springer, 2007 (Lecture Notes in Business Information Processing), S. 187–198

[VRA08] Vanderfeesten, I. ; Reijers, H. ; Aalst, W. van der: Product Based Workflow Support: Dynamic Workflow Execution. In: *20th International Conference on Advanced Information Systems Engineering (CAiSE)*, Springer, 2008 (LNCS 5074), S. 571–574

[WBS09] Weidlich, U. ; Bailer, M. ; Saborov, M.: Darstellung eines durchgängigen Systemintegrationsprozesses mit Fokus auf die Absicherung und Freigabe des E/E-Releases / ESG Elektroniksystem- und Logistik-GmbH. 2009. – Technischer Bericht. – Interner Bericht

[Weh00] Wehlitz, P.: *Nutzenorientierte Einführung eines Produktdatenmanagement-Systems*, Technische Universität München, Dissertation, 2000

[WEH08] Wörzberger, R. ; Ehses, N. ; Heer, T.: Adding Support for Dynamics Patterns to Static Business Process Management Systems. In: *7th International Symposium on Software Composition*, 2008 (LNCS), S. 84–91

[Wes01] Westfechtel, B.: Ein graphbasiertes Managementsystem für dynamische Entwicklungsprozesse. In: *Informatik, Forschung und Entwicklung* 16 (2001), Nr. 3, S. 125–144

[Wes07] Weske, M.: *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, Berlin, 2007. – ISBN 3540735216

[WFJ+03] Wimmer, A. ; Felfernig, A. ; Jannach, D. ; Mehlau, J. ; Russ, C. ; Zanker, M: Produktmodellierung und Produktkonfiguration im Financial Planning. In: *Banking and Information Technology (BIT)* 1 (2003), S. 43–52

[WG00] Wirtz, G. ; Giese, H.: Skalierbare Beschreibung des Verhaltens verteilter Software-Systeme. In: *Formale Beschreibungstechniken für verteilte Systeme (FBT)*, Shaker, 2000, S. 41–50

[WH07] Warkentin, A. ; Herbst, J.: Funktionsorientierung bei PLM-Systemen: Eine Analyse des Standes der Technik. In: *5. Paderborner Workshop Entwurf mechatronischer Systeme* Bd. 210, 2007 (HNI-Verlagsschriftenreihe, Paderborn)

[WJJ09] Warkentin, A. ; J.Gausemeier ; J.Herbst: Function Orientation beyond Development - Use Cases in the Late Phases of the Product Life Cycle. In: *19th CIRP Design Conference*, Cranfield University Press, 2009, S. 420–427

[WKDM+95] Wodtke, D. ; Kotz-Dittrich, A. ; Muth, P. ; Sinnwell, M. ; Weikum, G.: Mentor: Entwurf einer Workflow-Management-Umgebung basierend auf State- und Activitycharts. In: *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, 1995, S. 71–90

[Wor94] Workflow Management Coalition (WFMC): *Workflow Reference Model.* 1994

[Wor05] World Wide Web Consortium (W3C): *Web Services Choreography Description Language (WS-CDL) 1.0.* 2005

[WRR07] Weber, B. ; Rinderle, S. ; Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: *19th International Confence on Advanced Information Systems Engineering (CAiSE)* Bd. 4495, Springer, 2007 (LNCS), S. 574–588

[WRRM08] Weber, B. ; Reichert, M. ; Rinderle-Ma, S.: Change Patterns and Change Support Features - Enhancing Flexibility in Process-aware Information Systems. In: *Data and Knowledge Engineering* 66 (2008), Nr. 3, S. 438–466

[WSR09] Weber, B. ; Sadiq, S. ; Reichert, M.: Beyond Rigidity - Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-aware Information Systems. In: *Computer Science - Research and Development* 23 (2009), Nr. 2

[WWA08] Weidlich, U. ; Wüchner, C. ; Alyokhin, V.: Optimierungspotentiale der Prozesskette zur Bildung, Absicherung, Freigabe von E/E Releases / ESG Elektroniksystem- und Logistik-GmbH. 2008. – Technischer Bericht

[WWG01] Wirtz, G. ; Weske, M. ; Giese, H.: The OCoN Approach to Workflow Modeling in Object-Oriented Systems. In: *Information Systems Frontiers* 3 (2001), Nr. 3, S. 357–376

[WWWKD96] Wodtke, D. ; Weissenfels, J. ; Weikum, G. ; Kotz-Dittrich, A.: The Mentor Project: Steps Toward Enterprise-Wide Workflow Management. In: *12th International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 1996, S. 556–565

[YAW] *Yet Another Workflow Language (YAWL).* `http://yawl-system.com/`. – Zuletzt besucht am 05.03.2009

Index

Teil V

Anhänge

A Beweise

Nachfolgend werden die Beweise für Sätze aus den Kapiteln des zweiten Teils gegeben.

Satz 4.1 (Terminierung deterministischer OLCs)

Satz 4.1 (Terminierung deterministischer OLCs)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein deterministischer Object Life Cycle (vgl. Definition 4.4) mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$. *Wird der Endzustand des OLC entsprechend der definierten operationalen Semantik als* $SM(s_{end}) = \textsc{Activated}$ *markiert, gilt:*

$$\forall\, s \in S, s \neq s_{end} : SM(s) = \textsc{Done} \text{ und } \forall\, t \in T : TM(t) = \textsc{Fired}.$$

Satz 4.1 stellt einen Spezialfall von Satz 4.2 dar. Wir führen nachfolgend den Beweis für Satz 4.2 (d.h. die Terminierung deterministischer und nicht-deterministischer OLCs).

Satz 4.2 (Terminierung nicht-deterministischer OLCs)

Satz 4.2 (Terminierung nicht-deterministischer OLCs)
Sei $olc = (P, V, TS) \in \mathcal{OLC}$ *ein Object Life Cycle mit Transitionssystem* $TS = (S, T, s_{start}, s_{end})$ *und aktueller Markierung* $M = (SM, TM)$. *Wird der Endzustand des OLC entsprechend der definierten operationalen Semantik als* $SM(s_{end}) = \textsc{Activated}$ *markiert, gilt* $\forall\, s \in S, s \neq s_{end} : SM(s) \in \{\textsc{Done}, \textsc{Skipped}\}$ *und* $\forall\, t \in T : TM(t) \in \{\textsc{Fired}, \textsc{Disabled}\}$.

Beweis 2 (Satz 4.2, Terminierung nicht-deterministischer OLC)
Wir beweisen die Aussage aus Satz 4.2 durch Widerspruch. Wir führen den Beweis für die Aussage zu Zustandsmarkierungen, die Aussage für (interne) Transitionen kann analog bewiesen werden.

Annahme: Wir betrachten einen Object Life Cycle $olc = (P, V, TS) \in \mathcal{OLC}$ mit Transitionssystem $TS = (S, T, s_{start}, s_{end})$ und aktueller Markierung $M = (SM, TM)$. Unter den Voraussetzungen von Satz 4.2 gelte $\exists\, s^* \in S, s^* \neq s_{end} : SM(s^*) \notin \{\textsc{Done}, \textsc{Skipped}\}$, d.h. der Zustand s^* besitzt eine Markierung $SM(s^*) \in \{\textsc{Activated}, \textsc{NotActivated}\}$.

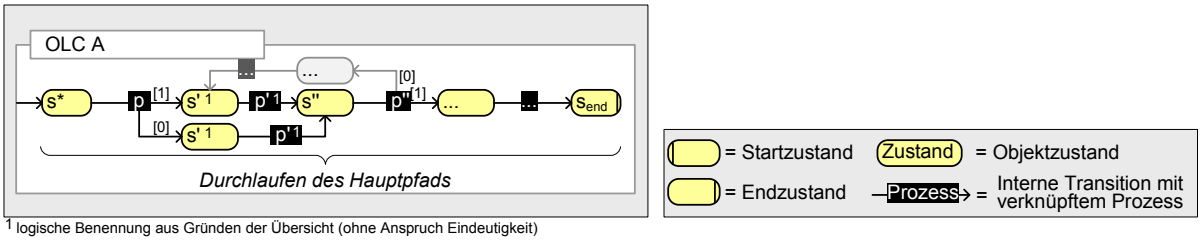
Fall 1: $SM(s^*) = \texttt{ACTIVATED}, s^* \in states_{int}(s_{start}, s_{end})$
Entsprechend Ausführungsregel AR1 bzw. AR2 gilt: $\forall\, t = (s^*, (p, v), s') \in outTrans_{int}(s^*)$: $TM(t) = \texttt{PROCESSING}$ und $SM(s') = \texttt{NOTACTIVATED}$, sofern $s' \in states_{int}(s^*, s_{end})$ (vgl. Abbildung A.1). Dabei liefert letzteres die Menge aller Zustände, die auf dem zyklenfreien Pfad zwischen s^* und s_{end} liegen (vgl. Definition 4.2).

Entsprechend Initialisierungsregel IR1 bzw. Ausführungsregel AR4 gilt für die aus den Zielzuständen s' ausgehenden Transitionen t' weiter: $\forall\, t' = (s', (p', v'), s'') \in outTrans_{int}(s')$: $TM(t') = \texttt{WAITING}$. Für s'' wiederum gilt nach Initialisierungsregel IR1 bzw. Markierungsregel MR4: $SM(s'') = \texttt{NOTACTIVATED}$. Durchlaufen wir auf diese Weise den endlichen Pfad des OLC zum Endzustand, d.h. betrachten wir analog alle Zustände des durch $states_{int}(s^*, s_{end})$ festgelegten Pfads, gilt folglich $SM(s_{end}) = \texttt{NOTACTIVATED}$. Dies bildet einen Widerspruch zu den Voraussetzungen aus Satz 4.2.

Fall 2: $SM(s^*) = \texttt{ACTIVATED}, s^* \notin states_{int}(s_{start}, s_{end})$
Entsprechend Markierungsregel MR2 gibt es einen Verzweigungsabschnitt im Hauptpfad des OLC $s' \in splitLoops(olc)$ mit Markierung $SM(s') = \texttt{DONE}$ der zur Aktivierung des Rücksprungpfads geführt hat, d.h. $\nexists\, \hat{s} \in states_{int}(s', s^*) : \hat{s} \neq s' \wedge \hat{s} \in states_{int}(s_{start}, s_{end})$. Ferner gilt für s' laut Markierungsregel MR5: s' hat einen direkten Nachfolgerzustand $\tilde{s} \in states_{int}(s_{start}, s_{end})$ mit $\exists\, t = (s', (p, v), targ) \in outTrans_{int}(s') \wedge \tilde{s} = targ \wedge SM(\tilde{s}) = \texttt{NOTACTIVATED}$. Es folgt die Argumentation aus Fall 3.

Fall 3: $SM(s^*) = \texttt{NOTACTIVATED}, s^* \in states_{int}(s_{start}, s_{end})$
Hier kann analog zu Fall 1 argumentiert werden.

Fall 4: $SM(s^*) = \texttt{NOTACTIVATED}, s^* \notin states_{int}(s_{start}, s_{end})$
Hier kann analog zu Fall 2 argumentiert werden. □



[1] logische Benennung aus Gründen der Übersicht (ohne Anspruch Eindeutigkeit)

Abbildung A.1: Pfad von Zustand s^* zum Endzustand

Satz 4.3 (Dynamische Korrektheit einer Prozessstruktur)

Satz 4.3 (Dynamische Korrektheit einer Prozessstruktur)
Sei $ps = (OLC, EST)$ eine Prozessstruktur. Sie ist dynamisch korrekt *(ihre Terminierung kann garantiert werden), wenn jeder Object Life Cycle $olc \in OLC$ korrekt ist (vgl. Definition 4.5) und ps keinen Zyklus enthält, der durch externe Transitionen verursacht wird (d.h. $\forall\, est = (s_1, p, s_2) \in EST : path_{ext}(s_2, s_1) = \emptyset$).*

Beweis 2 (Satz 4.3, Dynamische Korrektheit einer Prozessstruktur)
Wir beweisen Satz 4.3 mittels vollständiger Induktion. Dazu betrachten wir eine Prozessstruktur ps mit n externen Transitionen.

Induktionsannahme $I(n)$:
Sei ps eine Prozessstruktur mit n externen Transitionen sowie k korrekten OLCs. Enthält ps keinen Zyklus (mittels externer Transitionen), dann ist ps korrekt (d.h. jeder in ps enthaltene OLC erreicht seinen Endzustand und ps terminiert).

Induktionsanfang ($n = 1$):
Wir betrachten eine Prozessstruktur $ps = (OLC, EST)$ mit k korrekten Object Life Cycles $olc_i = (P_i, V_i, TS_i) \in OLC, i = 1..k$ mit den zugehörigen Transitionssystemen $TS_i = (S_i, T_i, s_{start_i}, s_{end_i})$ (vgl. Abbildung A.2a). Die aktuelle Markierung von ps sei $PSM = (OM, EM)$. Zwischen genau zwei der OLCs (olc_1 und olc_2) existiert die einzige externe Transition $e^* = (s_1, p, s_2) \in EST, |EST| = n = 1$ mit $s_1 \in S_1$ und $s_2 \in S_2$. O.B.d.A. betrachten wir nachfolgend nur die OLCs olc_1 und olc_2, da die restlichen OLCs $olc_j, j = 3..k$ aufgrund der Annahme korrekt terminieren und die Korrektheit von ps nicht beeinflussen.

Da olc_1 korrekt ist, wird er terminieren. Entsprechend Satz 4.2 bzw. der Regeln aus Kapitel 4 wird s_1 in jedem Fall im Ablauf als `ACTIVATED` bzw. `SKIPPED` markiert. Die Ausführungsregel AR5 bzw. AR6 führt dann zu einer Markierung der externen Transition als `PROCESSING` bzw. `DISABLED`. Wird die externe Transition als `DISABLED` markiert, bewirkt sie kein „Warten“ im Zielzustand. Sie kann damit nicht zu einer Verklemmung führen.

Ist die externe Transition als `PROCESSING` markiert, kann der Zielzustand s_2 nicht aktiviert werden, bis der Prozess p der externen Transition beendet und sich die Markierung der externen Transition in `FIRED` ändert (vgl. Markierungsregel MR6 bzw. MR7). Da die externe Transition in jedem Fall als `FIRED` oder `DISABLED` markiert wird, kann der Endzustand von olc_2 ebenfalls immer erreicht werden (olc_2 ist laut Annahme korrekt). Damit ist keines der Korrektheitskriterien (Verklemmungsfreiheit, Terminierung, Aktivierbarkeit) gefährdet und ps ist für $n = 1$ immer korrekt.

Induktionssschluss ($n \rightarrow n + 1$):
Wir betrachten eine Prozessstruktur mit $n + 1$ externen Transitionen (vgl. Abbildung A.2b). O.B.d.A. enthalte ps keine Zyklen (vgl. Definition 4.18). Damit kann im Sinne des Ablaufs der Prozessstruktur eine *topologische Sortierung* für die Quell- und Zielzustände der externen Transitionen hergestellt werden (d.h. jedem Quell- und Zielzustand einer externen Transition wird

ein Index zugeordnet) [Las61]. Die topologische Sortierung drückt eine mögliche Aktivierungsreihenfolge der Zustände aus, wobei der Index des Quellzustands einer externen Transition immer kleiner als der Index ihres Zielzustands ist.

Wir betrachten diejenigen externen Transition deren Quellzustand den höchsten Index $m, m <= 2n$ aller Quellzustände besitzt (vgl. Abbildung A.2b). Aus ihrem Zielzustand oder einem nachfolgenden Zustand kann dann keine externe Transition mehr ausgehen, wodurch sie keine Verklemmung in ihrem Zielzustand auslösen kann. Blenden wir diese externe Transition logisch aus, haben wir eine Prozessstruktur mit n externen Transitionen erzeugt, für die die Korrektheit laut Induktionsannahme gegeben ist, d.h. der Quellzustand der „letzten“ externen Transition wird in jedem Fall aktiviert und damit kann auch *ps* terminieren. □

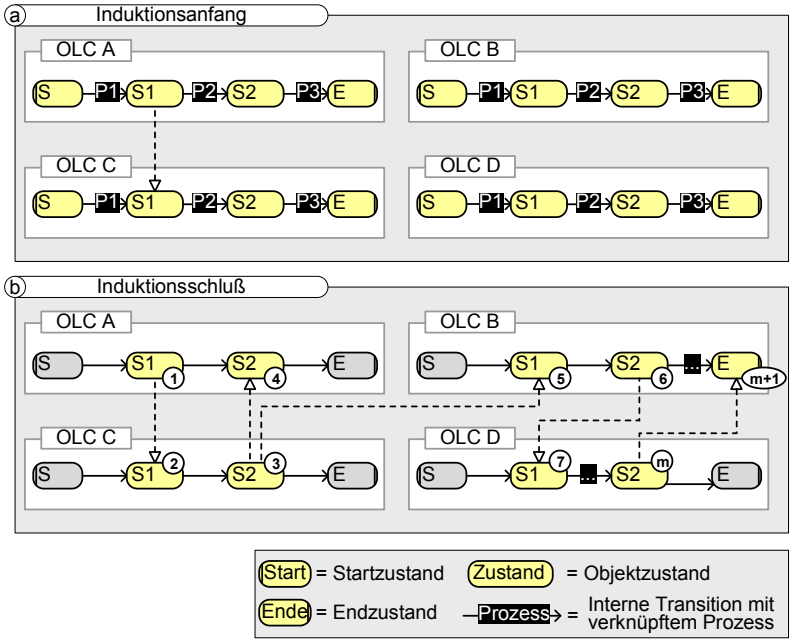


Abbildung A.2: Topologische Sortierung der externen Transitionen einer Prozessstruktur

Satz 5.1 (Korrektheit einer erzeugten Prozessstruktur)

Satz 5.1 (Korrektheit einer erzeugten Prozessstruktur)
Ist ein LCM korrekt (vgl. Definition 4.3), dann ist auch jede auf Basis dieses LCM erzeugte Prozessstruktur korrekt.

Beweis 2 (Satz 5.1, Dynamische Korrektheit einer Prozessstruktur)
Wir skizzieren den Beweis für Satz 5.1. Unter den gegebenen Voraussetzungen gelte: $lcs = (dm, lcm, ds, ps) \in \mathcal{LCS}$ sei eine *Life Cycle Coordination Structure* mit korrektem Life Cycle Coordination Model *lcm*.

Wird jeder Objekttyp und jeder Relationstyp des Datenmodells dm in der Datenstruktur ds genau einmal instanziiert, entspricht die resultierende Prozessstruktur ps im Grunde dem lcm. Die Prozessstruktur weist in diesem Fall dieselben strukturellen Eigenschaften wie lcm auf. Da lcm zyklenfrei ist, ist auch ps zyklenfrei. Werden eines oder mehrere der instanziierten Objekte sowie alle mit ihnen verbundenen Relationen entfernt, entspricht die resultierende Prozessstruktur ps' einer Teilstruktur des lcm. Auch ps' ist trivialerweise wieder zyklenfrei.

Externe Transitionen sind sowohl beim Quell- als auch beim Zielzustand mit einer UND-Semantik verknüpft, d.h. alle aus einem Zustand ausgehenden externen Transitionen werden parallel durchlaufen und alle eingehenden externen Transitionen müssen gefeuert haben, um den Zielzustand aktivieren zu können. Seien die Objekttypen sowie alle mit ihnen verbundenen Relationstypen beliebig oft instanziiert. Suchen wir in der dann vorliegenden Prozessstruktur ps von einem beliebigen Zustand s_1 einen Weg zu einem beliebigen anderen Zustand s_2, kann sich im Gegensatz zu der entsprechenden Suche im lcm (d.h. auf Basis der entsprechenden OLCs; vgl. Abbildung A.2a) nur die **Anzahl** der möglichen Wege erhöhen, nicht die strukturelle Eigenschaft des Pfads selbst (d.h. Anzahl und Reihenfolge der Zustände; vgl. Abbildung A.2b). Prüfen wir die Prozessstruktur $ps = (OLC, EST)$ dann auf Zyklenfreiheit, wird die Zyklensuche mittels $\forall\, est = (s_1, p, s_2) \in EST : path_{ext}(s_2, s_1) = \emptyset$ weiterhin gültig sein, da sich die Pfade strukturell nicht unterscheiden, d.h. es werden die gleichen Zustände in den Instanzen der OLCs durchlaufen. Daher kann kein Zyklus in der Prozessstruktur entstehen.

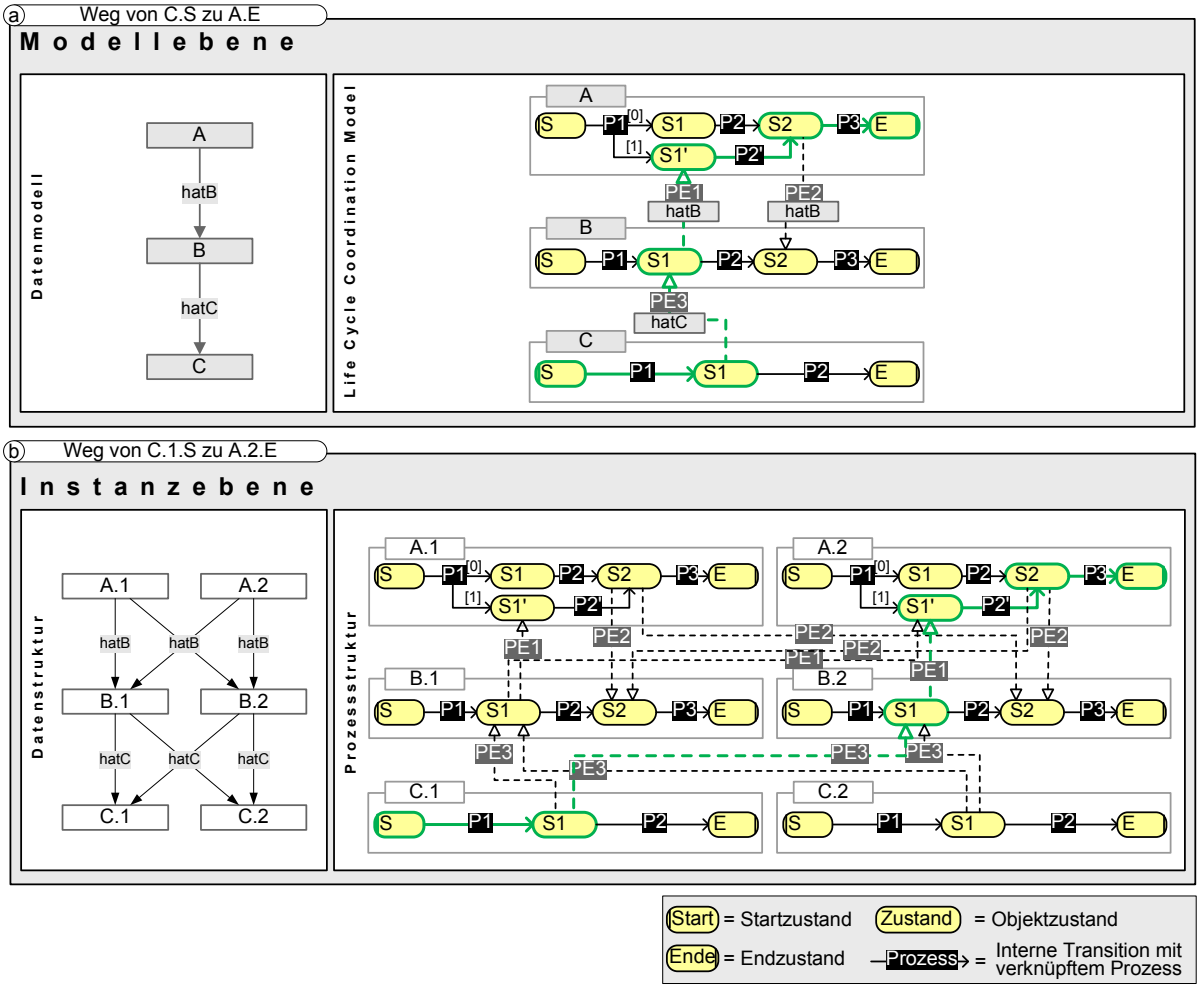


Abbildung A.3: Suche zyklenfreier Teilstrukturen

B

Weitere Algorithmen

```
1  public Set<SimpleIntTrans> pathSearch(SimpleState s0, SimpleState s1, SimpleState current,
   Set<SimpleIntTrans> m, boolean withLoopTrans, boolean mainPathOnly) {
     if (current.getIntegerID() == s1.getIntegerID()) return m;
     // Prüfe, ob aktueller Zustand bereits durchlaufen wurde; falls ja: Schleife
5    for(Iterator i = m.iterator(); i.hasNext();) {
       SimpleIntTrans trans = (SimpleIntTrans) i.next();
       if (trans.getSource() == current.getIntegerID()) {
         if (withLoopTrans) return m;
         else return null;
10     }
     }
     // Zusätzlicher Korrektheits-Check für Zustände, die "in der Luft" hängen
     if (current.getOutgoingIntTransitions().size() == 0) {
       return null;
15   }
     else
       // An Sequenzzustand angelangt?
       if (current.getOutgoingIntTransitions().size() == 1) {
         SimpleIntTrans trans = (SimpleIntTrans) current.getOutgoingIntTransitions().iterator().next();
20       // Wenn Pfad mit "exklusiver" Transition gefunden wird, ebenfalls "NULL" zurück
         m.add(trans);
         return retPath(s0, s1, (SimpleState) olc.getStateModelByID(trans.getTarget()), m, withLoopTrans,
   mainPathOnly);
       }
25     else
         // An Verzweigungszustand angelangt?
         if (current.getOutgoingIntTransitions().size() > 1) {
           Set<SimpleIntTrans> res = new HashSet<SimpleIntTrans>();
           // Für alle ausgehenden int. Transitionen eine "neue" Pfadliste erzeugen und durchlaufen
30         for(Iterator i = current.getOutgoingIntTransitions().iterator(); i.hasNext();) {
             Set<SimpleIntTrans> mt = new HashSet<SimpleIntTrans>(m);
             SimpleIntTrans trans = (SimpleIntTrans) i.next();

             // Prüfe, ob Transition auf Hauptpfad liegt (falls notwendig)
35           if ((mainPathOnly && !mainPath.contains(trans))) continue;

             mt.add(trans);

             Set<SimpleIntTrans> p = retPath(s0, s1, (SimpleState)
40 olc.getStateModelByID(trans.getTarget()), mt, withLoopTrans, mainPathOnly);
             if (p != null) res.addAll(p);
           }
       if (res.isEmpty()) return null;
       else m.addAll(res);
45   }
     return m;
   }

   public Set<SimpleIntTrans> pathInt(SimpleState start, SimpleState end) {
50   Set<SimpleIntTrans> res = pathSearch(start, end, start, new HashSet<SimpleIntTrans>(), false,
   false);
     return res;
   }
```

Algorithmus B.1: Realisierung von `pathInt` bzw. `pathSearch` im Demonstrator

```
1  /**
   * Prüft die Kardinalität der Relation in Verbindung mit dem Objekt obj. Um Zu beenden,
   * muss der Wert zwischen countMin und countMax liegen.
   */
5
   private void checkCardinality(String obj, String relationType, int countMin, int countMax) {
     boolean cardinalityFits = false;
     ISimpleDataStructure simpleDataStructure = (ISimpleDataStructure)getHolder();
     String object = obj;
10   if(obj.equals(SELF_REFERENCE)) object = simpleDataStructure.getCurrentParentID()
     while(!cardinalityFits) {
       // Anzahl der Relationen vom Typ: relationType
       int quantity = 0;
       // hole zuerst die Src und füge danach die Targ-Relationen an.
15     List<ISimpleRelation> relations_to_check = simpleDataStructure.getRelationSrc(object);
       relations_to_check.addAll(simpleDataStructure.getRelationTarg(object));
       for(ISimpleRelation simpleRelation : relations_to_check ) {
         if(simpleRelation.getRelationType().equals(relationType)) quantity++;
       }
20     if(quantity >= countMin && quantity <= countMax) cardinalityFits = true;
       if(!cardinalityFits)
       try {
         Thread.sleep(SimulationConstants.ANALYSE_CHECK_RANGE);
       } catch (InterruptedException e) {
25       getHolder().reportException("Thread-Fehler in der Analysemethode.");
         e.printStackTrace();
       }
     }
     setEndResult("Cardinality Checked");
30 }
```

Algorithmus B.2: Realisierung des Analyseprozesses für Kardinalitäten

C

Weitere Abbildungen

Änderungsregion			Szenario	Einfluss auf Korrektheit
Quellzustand	**Ext. Transition**	**Zielzustand**		
NotActivated	*Waiting*	*Activated*	1,2	Terminierungskriterium nicht erfüllt
NotActivated	*Waiting*	*Done*	1,2	Terminierungskriterium nicht erfüllt
NotActivated	*Waiting*	Skipped	1,2	Terminierungskriterium nicht erfüllt
Activated	*Waiting*	NotActivated	1,2	Verklemmung
Activated	*Waiting*	Activated	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Activated	*Waiting*	Done	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Activated	*Waiting*	Skipped	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Done	*Waiting*	NotActivated	1,2	Verklemmung
Done	*Waiting*	Activated	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Done	*Waiting*	Done	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Done	*Waiting*	Skipped	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Skipped	*Waiting*	NotActivated	1,2	Verklemmung
Skipped	*Waiting*	Activated	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Skipped	*Waiting*	Done	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
Skipped	*Waiting*	Skipped	1,2	Verklemmung, Terminierungskriterium nicht erfüllt
NotActivated	*Processing*	NotActivated	2	-
Activated	*Processing*	*Activated*	2	Terminierungskriterium nicht erfüllt
Activated	*Processing*	*Done*	2	Terminierungskriterium nicht erfüllt
Done	*Processing*	*Activated*	2	Terminierungskriterium nicht erfüllt
Done	*Processing*	*Done*	2	Terminierungskriterium nicht erfüllt
Done	*Processing*	*Skipped*	2	Terminierungskriterium nicht erfüllt
NotActivated	*Fired*	NotActivated	2	-
NotActivated	*Fired*	Activated	2	-
NotActivated	*Fired*	Done	2	-
NotActivated	*Fired*	Skipped	2	-
NotActivated	*Disabled*	NotActivated	2	-
NotActivated	*Disabled*	Activated	2	-
NotActivated	*Disabled*	Done	2	-
Activated	*Disabled*	NotActivated	2	-
Activated	*Disabled*	Activated	2	-
Activated	*Disabled*	Done	2	-
NotActivated	*(Removed)*	*Activated*	1	-
NotActivated	*(Removed)*	*Done*	1	-
Activated	*(Removed)*	*Activated*	1	-
Activated	*(Removed)*	*Done*	1	-
Done	*(Removed)*	*Activated*	1	-
Done	*(Removed)*	*Done*	1	-
1) Durch Einfügen oder Entfernen einer ext. Trans erzeugbar; 2) Durch Sprung- oder Resetoperation erzeugbar				

Abbildung C.1: Änderungsregionen mit inkonsistenter Laufzeitmarkierung

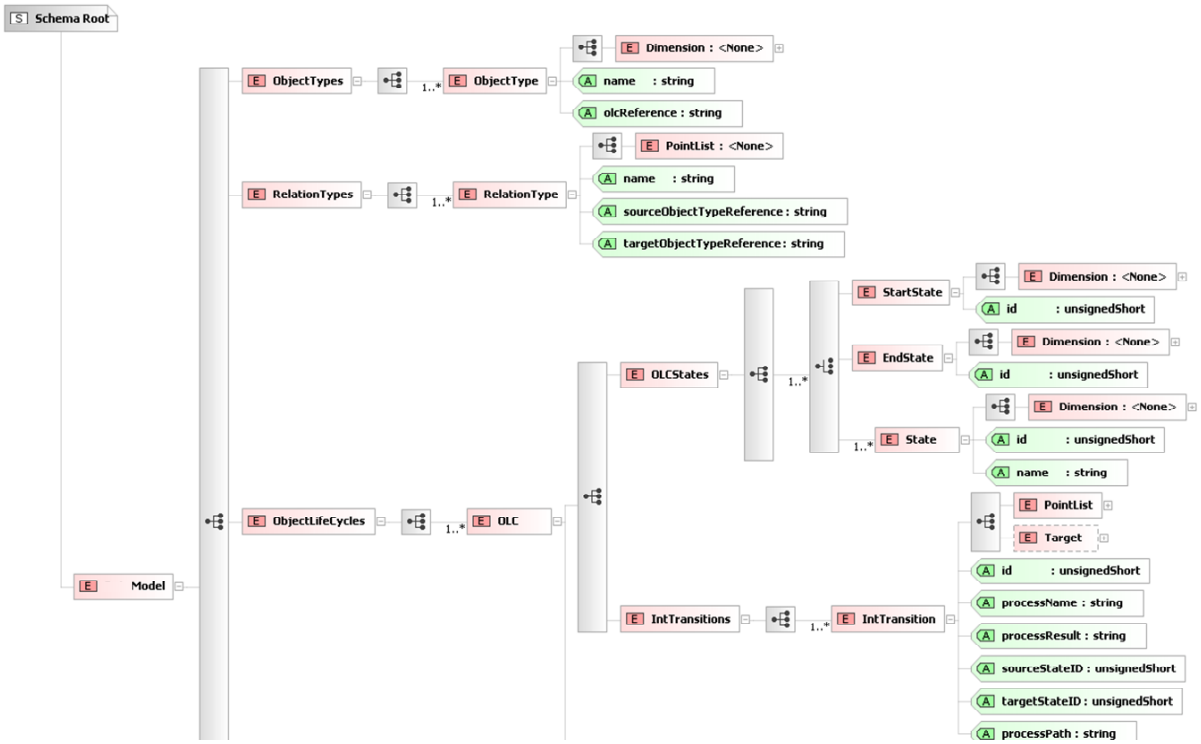


Abbildung C.2: Ausschnitt des XML-Schemas für die Abbildung der Modellebene in CORE-PRO$_{SIM}$

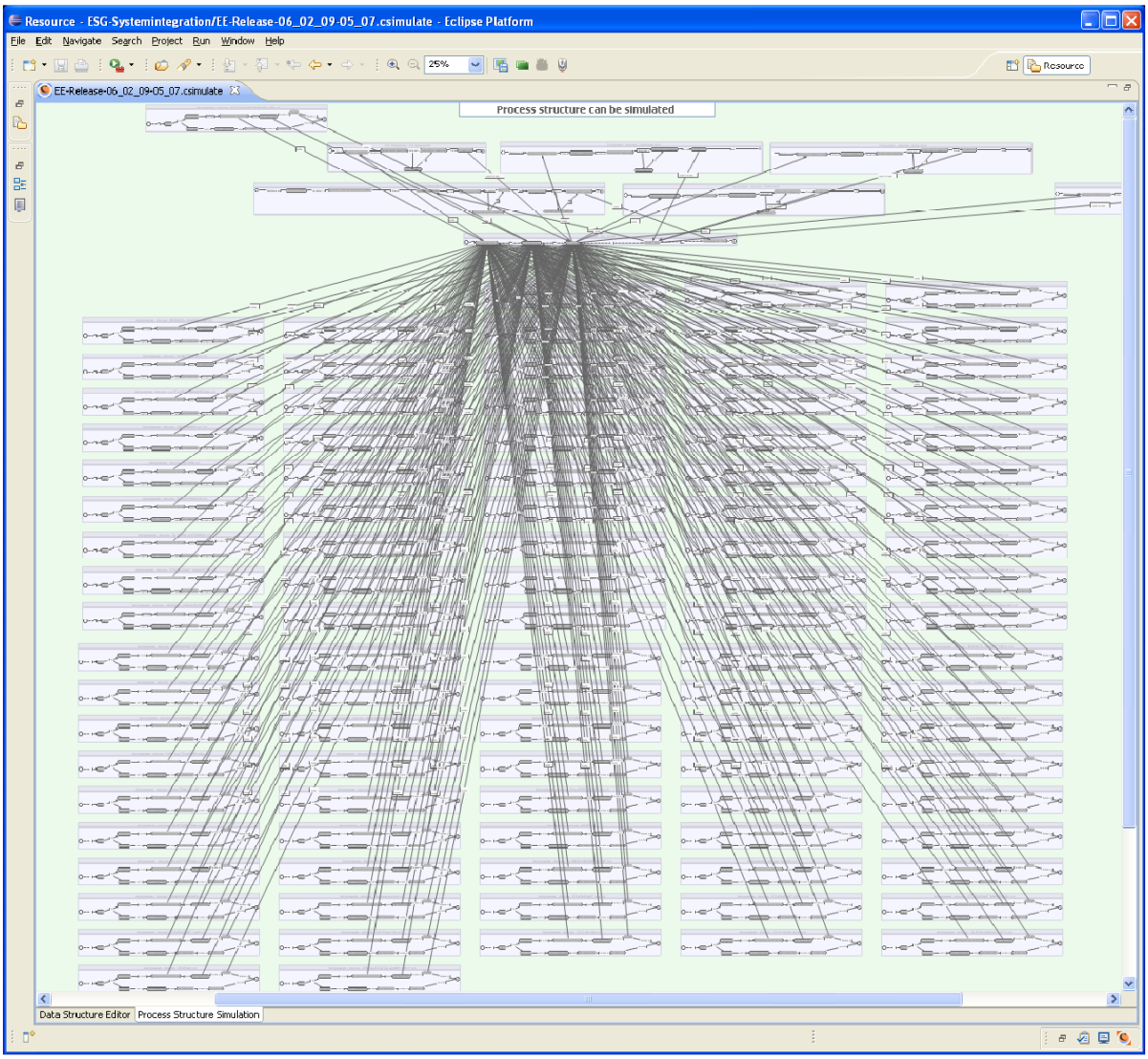


Abbildung C.3: Prozessstruktur mit etwa 100 Komponenten in CoreproSIM