ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung eines generischen Event-Mechanismus für Prozess-Management-Systeme

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Maximilian Lackaw
maximilian.lackaw@uni-ulm.de

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

Betreuer:
Andreas Lanz

2012

Fassung 17. Juni 2012

Kurzzusammenfassung

In den letzten Jahren wurde viel Forschungsarbeit in den Bereich des sogenannten Event Processing investiert. Dies ist eine Technik, mit der schnell auf das Auftreten von Ereignissen (bzw. Events) reagiert und diese anschließend weiterverarbeitet werden können. Dabei dienen Events als Nachrichtenträger. Somit ist es möglich, dass unterschiedliche Systeme über Events kommunizieren und Daten austauschen können. Auf diese Weise können auch Funktionsaufrufe in Events gekapselt werden und die entsprechenden Funktionen, ähnlich wie Web Services in einer SOA, aufgerufen werden.

Dieser Ansatz kann auch für BPM-Systeme genutzt werden. Dort müssen häufig Daten innerhalb einzelner Geschäftsprozesse ausgetauscht werden. Zusätzlich ist es durch die mit Events realisierbare lose Kopplung einfacher möglich, das BPM-System in eine heterogene Systemlandschaft zu integrieren.

Ziel dieser Arbeit ist es daher, eine Event Processing Komponente für BPM-Systeme zu entwickeln. Dazu wird als Grundlage eine sogenannte Event Processing Network (EPN) Architektur genutzt. Dies ist ein Ansatz, bei dem die Eventverarbeitung von einem Netzwerk leichtgewichtiger Komponenten ausgeführt wird. Darauf aufbauend wird ein Eventverteilungsmechanismus entwickelt. Hierbei ist insbesondere zu beachten, dass es möglich sein muss, dass die Produzenten und Konsumenten ohne gegenseitige Kenntnis voneinander Events austauschen können. Dies wird mit einem eigens entwickelten *Publish/Subscribe*-Mechanismus realisiert.

Durch die Kommunikation der Prozesse mittels Events entstehen neue Abhängigkeiten zwischen Prozessen bzw. einzelnen Aktivitäten. Daher musste das Prozess-Modell entsprechend angepasst werden, um auch dies abzubilden. Zusätzlich wurden Algorithmen zur Korrektheitsprüfung der Prozesse entwickelt. Damit können nicht eingehaltene Abhängigkeiten von Events identifiziert werden.

Durch den Einsatz von Event Processing in BPM-Systemen ist es leichter geworden, automatisch auf Events aus verschiedenen Prozessen zu reagieren. Des Weiteren erweitert die Nutzung der Events als Nachrichten die Möglichkeiten der Prozesse, in standardisierter Weise miteinander oder mit externen Systemen zu kommunizieren.

Inhaltsverzeichnis

1 Einleitung

Die Planung und Optimierung von Geschäftsprozessen stellt in der heutigen Zeit die Unternehmen vor eine besondere Herausforderung. Dabei spielen unter anderem BPM[1]-Systeme eine immer wichtigere Rolle. Diese unterstützen die Unternehmen sowohl bei der Planung als auch bei der Ausführung ihrer Geschäftsprozesse. Es ist hierbei besonders wichtig, dass die Ablauf- und Anwendungslogik strikt voneinander getrennt sind [DRRA05, DAG+06], um eine hohe Flexibilität und Wiederverwendbarkeit erreichen zu können. Dabei müssen BPM-Systeme mit einer Vielzahl von unterschiedlichen Systemen, wie z. B. Datenbanken und ERP[2]-Systemen, interagieren. Dies stellt häufig eine besondere Herausforderung dar, da Unternehmen in vielen Fällen eine heterogene Anwendungslandschaft mit vielen Legacy-Systemen[3] einsetzen [Jos08].

Eine weitere immer wichtiger werdende Technik ist das Event Processing. Bisher werden auftretende Ereignisse meist erst nachträglich in Form von Logs ausgewertet [See10]. Dies ist allerdings häufig zu spät. Um zeitnah auf diese Ereignisse reagieren zu können, müssen diese in Form von Events materialisiert und dann in Echtzeit ausgewertet werden. Anschließend kann entweder direkt auf einzelne „einfache“ Events reagiert werden (z. B. „Lagerstand niedrig“) und andererseits das Auftreten eines komplexen Sachverhalts in einem neuen komplexen Event ausgedrückt werden. So können mit Events auch komplexe Sachverhalte dargestellt werden, indem in einer Vielzahl von unterschiedlichen, für sich genommen vielleicht unbedeutenden, Events Muster erkannt und diese dann zu komplexen Events zusammengefasst werden. Diese sogenannten *Complex Events* können anschließend mit anderen Events weiter zu neuen *Complex Events* kombiniert werden. Auf diese Weise kann z. B. direkt auf ein „Lagerstand niedrig“-Event reagiert werden, indem neue Ware nachbestellt wird. Genauso ist es aber auch möglich, beim Eintreffen einer Vielzahl „Bestellung

[1] Business Process Management
[2] Enterprise Resource Planning
[3] Alt-Systeme

für Ware X"-Events bereits auf eine erhöhte Nachfrage zu schließen. So kann eine Nachbestellung schon wesentlich früher eingeleitet und ein Lieferengpass vermieden werden.

Die Möglichkeiten von Event Processing gehen aber noch wesentlich weiter. So können über Events auch Funktionsaufrufe gekapselt werden. Dies bedeutet, dass das Erzeugen eines Events an die Stelle eines direkten Funktionsaufrufs treten kann. Diese Events werden dann beim Empfänger auf konkrete Funktionsaufrufe abgebildet [EN10, YJ10]. Auf diese Weise können, ähnlich wie bei einer Serviceorientierten Architektur (SOA), die Funktionen lose gekoppelt als Dienste aufgerufen werden. Damit stellen Events in einem heterogenen Umfeld ein vielseitiges Medium dar, durch das diverse Systeme über einheitliche Schnittstellen kommunizieren.

Der Einsatz von Event Processing bietet demnach auch neue Möglichkeiten für BPM-Systeme. So lassen sich die Events, welche in Geschäftsprozessen produziert werden, automatisch verarbeiten. Somit kann direkt auf verschiedene Situationen reagiert werden, die zum Zeitpunkt der Prozessmodellierung noch nicht erkennbar waren, z. B. die zu häufige Wiederholung eines bestimmten Prozessschrittes. Zusätzlich können mit Events auch unterschiedliche Prozesse und Aktivitäten standardisiert miteinander kommunizieren. Durch die Kapslung der Nachrichten in Events, statt Punkt-zu-Punkt-Nachrichten, entsteht eine höhere Unabhängigkeit der einzelnen Prozesse. Diese wiederum ist von Vorteil für die Wiederverwendbarkeit. Ein weiterer Vorteil ist die hierdurch gegebene Möglichkeit einer asynchronen Kommunikation verschiedener Prozesse. Schließlich ist noch ein weiterer wichtiger Aspekt die Anbindung von externen Systemen an das BPM-System. Durch die Kapselung mit Events kann eine lose Kopplung auch für BPM-Systeme und Prozesse genutzt werden. Somit bleiben die Prozesse unabhängig von den tatsächlich eingesetzten Systemen und behalten auch dann noch ihre Gültigkeit, wenn einzelne Komponenten ausgetauscht werden.

1.1 Aufgabenstellung und Zielsetzung

Ziel dieser Diplomarbeit ist es, einen Event-Mechanismus für BPM-Systeme zu entwickeln. Dabei sollen zunächst die Möglichkeiten von Event Processing in Geschäftsprozessen näher untersucht werden. Hierfür müssen zunächst die Anforderungen

an ein Event Processing System für BPM-Systeme und die sich daraus ergebenden Vorteile erarbeitet werden. Die lose Kopplung der einzelnen Komponenten und die Kommunikation innerhalb der Prozesse sind hier von entscheidender Bedeutung.

Nachdem die Grundlagen und Vorbedingungen erarbeitet sind, soll eine konkrete Architektur für eine Event Processing Komponente als Erweiterung eines BPM-Systems, entwickelt werden. Dabei müssen vor allem zwei Aspekte beachtet werden: Zum einen muss eine flexible Anbindung von externen Systemen möglich sein und zum anderen braucht man einen Mechanismus zur sicheren Verteilung der Events. Dieser Mechanismus muss es ermöglichen, dass voneinander unabhängige Systeme und Prozesse, ohne gegenseitige Kenntnis, Events austauschen können.

Schließlich muss untersucht werden, inwieweit für die Realisierung eines solchen Systems das bisherige Prozessmodell erweitert werden muss. Da Events auch für die Kommunikation innerhalb eines Prozesses genutzt werden, entstehen beispielsweise auch Abhängigkeiten einzelner Aktivitäten auf Basis der produzierten und konsumierten Events. Daher soll auch untersucht werden, inwieweit die Abhängigkeiten von Events zu Verklemmungen führen und wie diese erkannt werden können.

1.2 Aufbau der Arbeit

In Kapitel 2 stellen wir die Grundlagen für BPM und Event Processing vor. Zudem vertiefen wir diese mit einem komplexeren Beispiel, an dem wir die Anforderungen für einen Event-Mechanismus für BPM-Systeme erarbeiten. Danach stellen wir in Kapitel 3 eine allgemeine Event Processing Network (EPN)-Architektur vor. Dies ist eine mögliche Realisierung von Event Processing, auf welcher der hier nachfolgend vorgestellte Lösungsansatz basiert.

In Kapitel 4 erarbeiten wir eine konkrete Event Processing Architektur für BPM-Systeme. Darauf aufbauend werden im Kapitel 5 nötige Änderungen am Prozessmodell durchgeführt. Kapitel 6 beschäftigt sich mit der Analyse der Korrektheit von Prozessen in Hinblick auf die von ihnen produzierten und konsumierten Events. Dabei wird ermittelt, inwieweit Verklemmungen auftreten können und wie diese automatisch erkannt werden können.

In Rahmen dieser Arbeit ist eine Prototyp-Implementierung, der in den vorherigen Kapiteln erarbeiteten Architektur, entstanden. Diese wird in Kapitel 7 vorgestellt.

Zuletzt widmen wir uns in Kapitel 8 einigen verwandten Themen und Konzepten, bevor in Kapitel 9 eine Zusammenfassung sowie ein Ausblick erfolgt.

2 Grundlagen

Im folgenden Kapitel werden zunächst die Grundlagen für diese Arbeit beschrieben. Diese sollen ein Verständnis für spätere Kapitel erleichtern. Der hier entwickelte Event-Mechanismus ist weitestgehend unabhängig vom eingesetzten BPM-System. Allerdings werden für die Prozessbeispiele die ADEPT-Syntax verwendet und einige Stellen dieser Arbeit sind auf eine Erweiterung von ADEPT ausgelegt. Deswegen wird zunächst in Abschnitt 2.1 das ADEPT-Metamodell näher beschrieben. Zusätzlich soll dieser Abschnitt einheitliche Begriffe für BPM schaffen.

Danach folgt in Abschnitt 2.2 eine Einführung in das Complex Event Processing. Abgeschlossen wird dieses Kapitel in Abschnitt 2.3, mit einem umfangreicheren Beispiel zu Event Processing in einem Geschäftsprozess. Dieses Beispiel soll auch die Anforderungen (Abschnitt 2.4) für den Event-Mechanismus demonstrieren.

2.1 ADEPT-Metamodell

ADEPT ist eine grafische Modellierungssprache für Geschäftsprozesse, die eine symmetrische Blockstrukturierung verwendet [Rei00]. Mithilfe von ADEPT können Kontrollflussgraphen modelliert werden, die als ausführbare Prozesse dienen. Wir unterscheiden hier zwischen Prozess*vorlagen* und Prozess*instanzen*. Die Prozessvorlage wird zur Entwurfszeit von einem Modellierer erstellt und beschreibt den Ablauf eines Prozesses. Diese Vorlage kann zur Laufzeit beliebig oft instanziiert werden. Jede laufende Prozess-Instanz hat ihren eigenen Ausführungsstaus und eigene konkrete Daten. In ADEPT lassen sich Kontroll- und Datenfluss strikt voneinander trennen. Im Rahmen dieser Arbeit gehen wir nicht auf alle Aspekte von ADEPT ein, es werden daher nicht alle Möglichkeiten und Konstrukte behandelt. Für eine ausführliche Beschreibung siehe [Rei00].

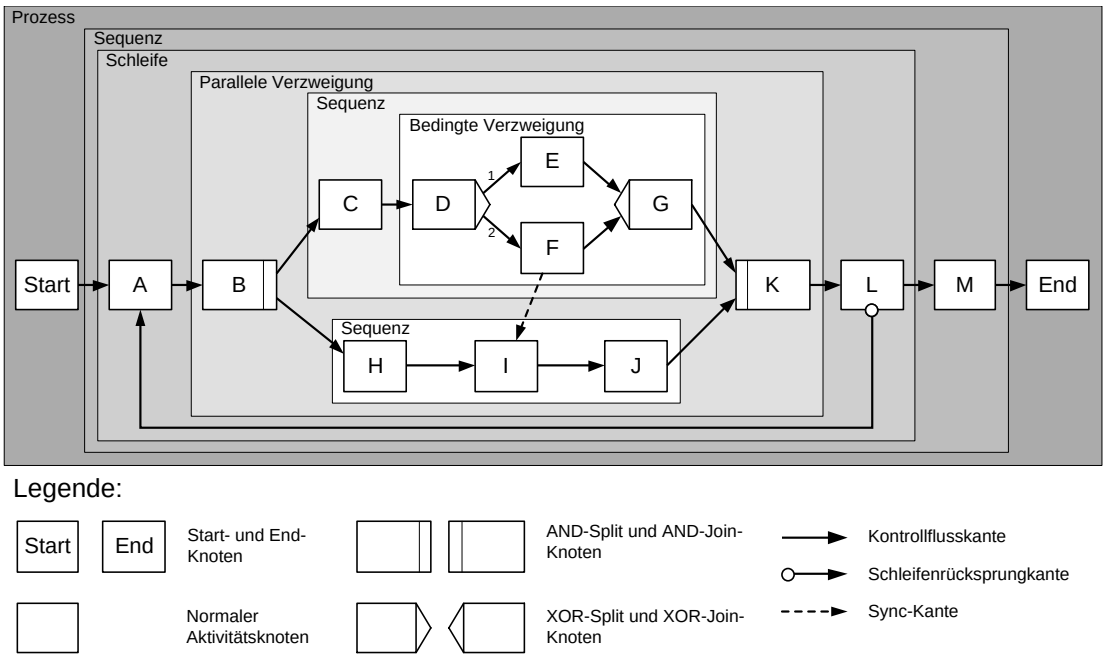

Abbildung 2.1: Schematische Übersicht der ADEPT-Kontrollflusskonstrukte (angelehnt an [Rei00])

2.1.1 Kontrollfluss

Der Kontrollfluss wird in ADEPT über einen gerichteten Graphen dargestellt. Im sogenannten Kontrollflussgraphen bilden die Knoten die einzelnen Prozessschritte (sogenannte Aktivitäten) und die Kanten die Kontrollflussübergänge. Jeder Knoten, bis auf den Start- und Endknoten, hat mindestens eine eingehende und eine ausgehende Kontrollflusskante. ADEPT verwendet eine symmetrische Blockstrukturierung. Dies bedeutet, dass die Modellierung des Geschäftsprozesses in Blöcken erfolgt. Jeder dieser Blöcke besitzt einen eindeutigen Start- und Endknoten. Die Blöcke können sowohl sequentiell, als auch ineinander verschachtelt angeordnet werden. Allerdings ist keine Überlappung der Blöcke erlaubt. Eine Übersicht der für diese Arbeit relevanten Kontrollflusskonstrukte wird in Abbildung 2.1 gegeben.

Als Blockkonstrukte stehen Sequenzen, bedingte und parallele Verzweigung und Schleifen zur Verfügung. Dabei ist der Prozess selbst ein Block mit Start- und Endknoten (siehe `Start` und `End` Knoten in Abbildung 2.1). Wir werden nun die einzelnen Kontrollflusskonstrukte kurz erklären.

Sequenz

Das grundlegendste Blockkonstrukt ist die Sequenz (siehe z. B. die Knoten `H`, `I` und `J`). Die in der Sequenz enthalten Knoten werden zur Laufzeit sequenziell hintereinander ausgeführt. Jeder einzelne Knoten einer Sequenz (mit Ausnahme der Knoten `Start` und `End`) besitzt genau eine Eingangs- und eine Ausgangskontrollflusskante.

Verzweigungen

Eine reine sequentielle Bearbeitung der Arbeitsschritte ist für anspruchsvolle Anwendungen nicht ausreichend [Rei00]. Daher unterstützt ADEPT mehrere Verzweigungsarten. Wir behandeln hier nur die bedingte und parallele Verzweigung. Jede Verzweigungsart besitzt einen eindeutigen Verzweigungsknoten (sog. Split-Knoten) und einen eindeutigen Synchronisationsknoten (sog. Join-Knoten). Hinzu kommt eine endliche, nichtleere Menge von Bearbeitungszweigen. Diese werden wiederum mit Kontrollblöcken versehen.

Parallele Verzweigung Die parallele Verzweigung ermöglicht die gleichzeitige Bearbeitung von unabhängigen Vorgängen. Dabei werden nach Beendigung des *AND-Split-Knoten* (Knoten `B`) zunächst alle Nachfolgerknoten parallel ausgeführt (hier Knoten `C` und `H`). Der *AND-Join-Knoten* (Knoten `K`) kann erst ausgeführt werden, wenn alle eingehende Pfade abgearbeitet sind. In diesem Fall, wenn Knoten `G` und `J` abgeschlossen sind.

Bedingte Verzweigung Bei der bedingten Verzweigung wird immer genau ein Nachfolgezweig des *XOR-Split-Knoten* (Knoten `D`) ausgewählt. Die Knoten der übrigen Pfade werden übersprungen. Sollte z. B. im Prozess von Abbildung 2.1 nach Knoten `D` der Pfad 1 ausgewählt werden, so wird Knoten `E` ausgeführt und Knoten `F` übersprungen. Erst nach Beendigung aller Knoten im gewählten Pfad kann der *XOR-Join-Knoten* (Knoten `K`) ausgeführt werden.

Schleife

Eine Schleife wird durch einen Schleifenstart- (Knoten `A`) und einen Schleifenendknoten (Knoten `L`) begrenzt. Zwischen diesen beiden Knoten gibt es eine Schleifenrück-

sprungkante. Der von der Schleife umschlossene Block wird Schleifenkörper genannt. Zur Laufzeit wird dieser solange durchlaufen, bis die Schleifenabbruchbedingung erfüllt ist. Dadurch wird der Schleifenkörper mindestens einmal durchlaufen (vergleichbar mit einer `do-while`-Schleife in Java oder C#).

Synchronisationskanten

Für eine nebenläufige Bearbeitung kann es notwendig sein, bestimmte Arbeitsschritte zu verzögern, bis andere abgeschlossen sind. Dazu bietet ADEPT Synchronisationskanten (kurz: Sync-Kanten), mit denen eine Ausführung des Zielknotens (Knoten `I`) verzögert wird, bis der Quellknoten (Knoten `F`) abgeschlossen ist. Wir betrachten in dieser Arbeit nur die „einfachen“ Sync-Kanten, bei denen es ausreicht, dass der Quellknoten entweder abgeschlossen oder übersprungen wurde. Im Beispielprozess von Abbildung 2.1 bedeutet dies, wird beim XOR-Split-Knoten `D` der Pfade 1 gewählt, so ist der Knoten `I` sofort ausführbar, sofern Knoten `H` abgeschlossen ist. Sollte Pfad 2 gewählt werden, so muss auch Knoten `F` abgeschlossen sein.

Hierarchische Strukturierung

Damit komplexe Arbeitsabläufe besser strukturiert werden können, bietet ADEPT die Möglichkeit der hierarchischen Strukturierung. Dabei können einzelne Blöcke zu Subprozessen zusammengefasst werden. Diese können schrittweise verfeinert werden, bis in einem Subprozess nur noch elementare Aktivitäten (ein atomarer Prozessschritt, der nicht weiter teilbar ist) vorhanden sind. Ein Knoten im Kontrollflussgraph kann daher eine elementare Aktivität oder ein Subprozess sein. Die Unterteilung in Subprozesse erhöht die Übersichtlichkeit und Wiederverwendbarkeit von Prozessen. Der oberste Prozess in einem hierarchisch strukturierten Prozess wird Top-Level-Prozess genannt. Der übergeordnete Prozess eines Subprozesses wird Vater-Prozess oder Super-Prozess genannt.

2.1.2 Datenfluss

Mithilfe von globalen Datenelementen können Daten zwischen den einzelnen Aktivitäten (unabhängig ob elementarer Schritt oder Subprozess) ausgetauscht werden. Dazu werden Datenelemente und Aktivitäten mit Datenlese- und Schreibkanten

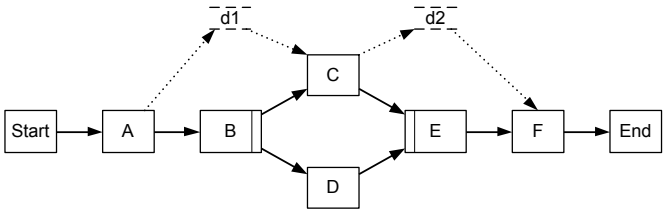

Abbildung 2.2: ADEPT-Prozess mit Datenelementen und Datenflusskanten

verbunden. Ein Beispiel dazu ist in Abbildung 2.2 zu sehen. Dort schreibt Aktivität `A` in das Datenelement `d1`. Im späteren Verlauf des Prozesses liest Aktivität `C` dieses Datenelement. Aktivität `C` schreibt wiederum das Datenelement `d2`, welches von Aktivität `F` gelesen wird.

Beim Lesen und Schreiben von Datenelementen wird unterschieden, ob der Zugriff obligat oder optional erfolgt. Ein obligater Lesezugriff bedeutet, dass zur Laufzeit der Wert des lesenden Datenelementes einen gültigen Wert haben muss, d. h. ungleich `NULL`. Beim optionalen Lesen darf das Datenelement auch den Wert `NULL` haben. Daraus folgt, dass beim obligaten Lesen mindestens ein obligater Schreibzugriff vorhergegangen sein muss. Ist dies der Fall, werden die Datenelemente als „sicher versorgt“ angesehen. Im Beispiel von Abbildung 2.2 sind beide Datenelemente bei ihrem jeweiligen Lesezugriff versorgt, wenn wir davon ausgehen, dass alle Lese- und Schreibzugriffe obligat sind. Würde man allerdings die parallele Verzweigung durch eine bedingte Verzweigung austauschen, ist dies für den Lesezugriff von Datenelement `d2` nicht mehr gewährleistet, da nicht in jeden Fall Aktivität `C` ausgeführt wird.

2.1.3 Dynamisches Verhalten

Zur Laufzeit werden Knoten und Kanten mit Markierungen versehen, um den jeweiligen Bearbeitungszustand darzustellen. Damit ist immer nachvollziehbar, welche Aktivitäten einer Prozess-Instanz bereits ausgeführt sind, übersprungen wurden, sich gerade in Ausführung befinden und welche noch ausgeführt werden. In Abbildung 2.3 sind die Knoten- und Kantenmarkierungen vereinfacht dargestellt. Auf die für diese Arbeit nicht relevanten Zustände wird verzichtet.

Zunächst befinden sich beim Start einer Prozess-Instanz alle Aktivitäten im Zustand `NOT_ACTIVATED`. In diesem Zustand ist die Aktivität noch nicht bearbeitet. Sobald alle Vorbedingungen (siehe Abschnitte 2.1.1) erfüllt sind, kann die Aktivität

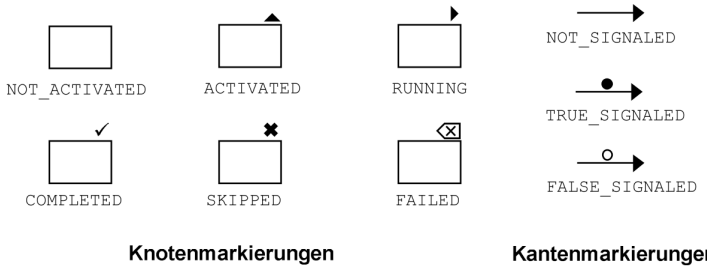

Abbildung 2.3: ADEPT Knoten- und Kantenmarkierungen (aus [Rei00], vereinfacht)

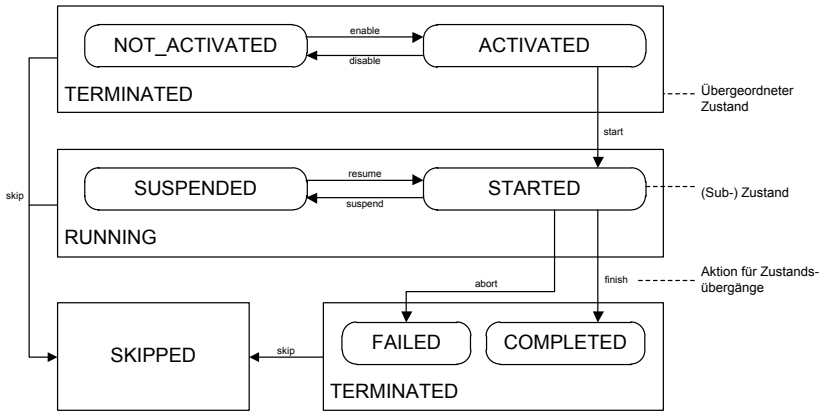

Abbildung 2.4: Zustandsübergangsdiagramm für Aktivitäten in ADEPT (aus [Rei00], vereinfacht)

auf `ACTIVATED` gesetzt werden. Sobald mit der Bearbeitung der Aktivität begonnen wurde, wechselt sie in den Zustand `RUNNING`. Hier wird noch unterschieden zwischen laufend (`STARTED`) und unterbrochen (`SUSPENDED`). Wird eine Aktivität übersprungen (siehe bedingte Verzweigung) wechselt sie in den Zustand `SKIPPED`. Ist eine Aktivität erfolgreich abgeschlossen, erhält sie den Zustand `COMPLETED`. Bei nicht erfolgreichen Abschluss erhält sie den Zustand `FAILED`. Eine Übersicht der Zustandsübergänge ist in Abbildung 2.4 dargestellt.

Die Zustandsbelegung der Kantenmarkierung verhält sich wie folgt: Zu Beginn sind alle Kanten mit `NOT_SIGNALED` versehen. Nach Beendigung einer Aktivität werden alle ausgehenden Kanten auf `TRUE_SIGNALED` gesetzt, es sei dem, der entsprechende Pfad wurde bei einer bedingten Verzweigung nicht gewählt. In diesem Fall wird die

Kante auf `FALSE_SIGNALED` gesetzt. Dies geschieht auch, wenn die Quellaktivität auf `SKIPPED` gesetzt wird.

2.1.4 Besondere Eigenschaften

Eine besondere Eigenschaft von ADEPT ist die Sicherstellung der Korrektheit. Da trotz Blockstrukturierung Verklemmungen möglich sind, sind mehrere Korrektheitskriterien (KF 1 - 8, näheres in [Rei00]) definiert. Damit lässt sich bereits zur Entwurfszeit sicherstellen, dass der Kontrollflussgraph keine Verklemmung besitzt. Zudem sind Regeln definiert, welche die Korrektheit des Datenflusses sicherstellen. Hier sei besonders der `WriterExists` - Algorithmus erwähnt, der sicherstellt, dass obligate Lesezugriffe sicher versorgt sind.

Eine weitere Besonderheit sind die instanzspezifischen Änderungen („Ad-hoc-Änderungen“). Diese bieten die Möglichkeit, Änderungen an laufenden Prozess-Instanzen vorzunehmen. So ist es möglich, neue Prozessschritte hinzuzufügen und zu löschen, ohne das die Korrektheit eingeschränkt wird. Ad-hoc-Änderungen spielen in Rahmen dieser Arbeit keine Rolle. Wir werden daher nicht weiter auf sie eingehen.

2.2 Complex Event Processing (CEP)

Bisher wurden Ereignisse (bzw. Events) meist erst nach Abschluss einer Tätigkeit ausgewertet [EB09]. Typischerweise geschieht dies durch eine Auswertung von Log-Dateien oder von Informationen, die man nach Prozessende aus einem Data-Warehouse-System erhält [See10]. Dies hat allerdings den Nachteil, dass es sehr oft zu spät kommt und deshalb auch unflexibel ist. Deswegen versuchen viele Firmen, dynamische Vorgänge mittels selbst erstellter Skripte zu überwachen. Diese Methode hat allerdings wiederum den Nachteil, dass sie sehr aufwendig und nicht standardisiert ist. Deswegen wird schon seit einigen Jahren immer mehr Forschung im Bereich von Complex Event Processing (CEP) betrieben. Wir behandeln im Folgenden die Grundlagen dafür.

2.2.1 Grundlagen für CEP

Wir betrachten zunächst Abbildung 2.5. Dort ist eine schematische Darstellung einer Event Processing Architektur abgebildet. Auf der linken Seite sind die *Event Producers*

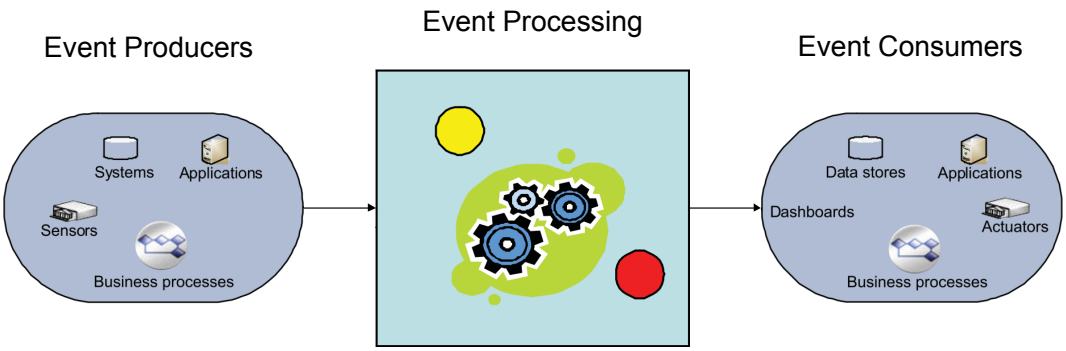

Abbildung 2.5: Überblick auf CEP mit getrennter Event Processing Logic (angelehnt an [EN10])

dargestellt. Diese haben die Aufgabe, die Events zu produzieren. *Event Producers* können z. B. Sensoren, Server, Programme oder Geschäftsprozesse sein. Mit den Events zeigen diese an, dass etwas bestimmtes passiert ist, wie z. B. ein nicht korrekt abgeschlossener Geschäftsprozess. Diese Events können nun zeitnah verarbeitet, und entsprechende Maßnahmen eingeleitet werden. In der Mitte liegt die Event Processing Logic. Hier werden Events entgegengenommen, gefiltert und weiter verteilt. Hier können auch Events weiterverarbeitet und mit zusätzlichen Informationen angereichert werden. Es können auch mehrere (per se unwichtige) Events zu sogenannten *Complex Events* aggregiert werden [Luc02], die komplexere Sachverhalte darstellen. Auf der rechten Seite von Abbildung 2.5 sind die *Event Consumers* dargestellt. Diese übernehmen die Verarbeitung der ihnen zugestellten Events. Dies könnten unter anderem Dash-Boards, Anwendungsprogramme, Datenbanken, Analyse-Tools und Geschäftsprozesse sein. Die Einsatzzwecke sind hierbei sehr vielfältig. Ein Anwendungsfall für ein CEP-System ist ein BPM-System, das nach Beginn und Ende eines Geschäftsprozesses jeweils ein Event produziert. Bei jedem abgeschlossenen Prozess produziert die Event Processing Logic aus dem Start- und End-Event ein *Complex Event*, welches die Bearbeitungszeit des Prozesses angibt. Diese *Complex Events* können nun z. B. von einer Dash-Board-Anwendung verarbeitet und dem Benutzer in Form von Statistiken angezeigt werden.

Dieser Abschnitt soll einen groben Überblick über CEP darstellen. Im nächsten Abschnitt vertiefen wir die Begriffe Event und *Complex Event*.

2.2.2 Events und Complex Events

Im vorherigen Abschnitt wurde bereits ein kurzer Überblick über CEP behandelt. In Folgendem sollen die Begriffe Event und *Complex Event* diskutiert werden. Dazu betrachten wir zunächst eine vereinfachte Definition des (*Complex*)Event -Begriffes.

Der genaue Aufbau eines Events ist nicht einheitlich und kann in unterschiedlichen Event-basierten Architekturen unterschiedlich definiert sein [EN10]. Deswegen wird hier zunächst ein möglichst allgemeingültiger Aufbau eines Events betrachtet. Dies soll dabei helfen eine einheitliche Vorstellung von Event zu erhalten.

Event Ein Event beschreibt ein Ereignis, welches in einem bestimmten System oder einer Domäne aufgetreten ist. Es zeigt, dass etwas aufgetreten ist, oder etwas hätte auftreten sollen [EN10]. Mit dem Wort Event wird dabei meist auch die Entität bezeichnet, die ein Event in einem Computersystem repräsentiert.

Mit dem Wort Event ist häufig sowohl das eigentliche Ereignis als auch die Repräsentation im Computersystem gemeint. Zur besseren Verständlichkeit werden wir daher nun eine Unterscheidung festlegen. Im Folgenden verwenden wir den Begriff Ereignis, wenn wir uns auf das tatsächliche Geschehnis beziehen, z. B. ein Paket verlässt das Versandzentrum. Beziehen wir uns auf die Repräsentation im Computersystem, verwenden wir den Begriff Event-Instanz oder auch kurz Event.

Wir betrachten zunächst die minimalen Informationen, welche zur Beschreibung eines Events nötig sind. Als Erstes muss ein Event eindeutig identifizierbar sein, d. h. es benötigt eine eindeutige Kennung. Damit ist eine Unterscheidung mehrerer Events auch bei gleichem Inhalt gewährleistet. Als Zweites benötigt ein Event einen Zeitstempel. Damit ist es möglich, dass die zeitliche Reihenfolge, in der die entsprechenden Ereignisse aufgetreten sind zu rekonstruieren. Aufgrund des Aufbaus eines CEP-Systems werden Events nicht immer in der gleichen Reihenfolge empfangen, in der sie produziert wurden. Mithilfe eines Zeitstempels ist es somit nachvollziehbar, ob ein Ereignis `A` vor einem Ereignis `B` aufgetreten ist. Des Weiteren ist ein Event einem sogenannten *Event Type* zugeordnet. Dieser beschreibt, um was für eine Art von Ereignis es sich handelt. Beispiele hierfür sind „Versand einer Ware“ oder „Eingang einer Zahlung“.

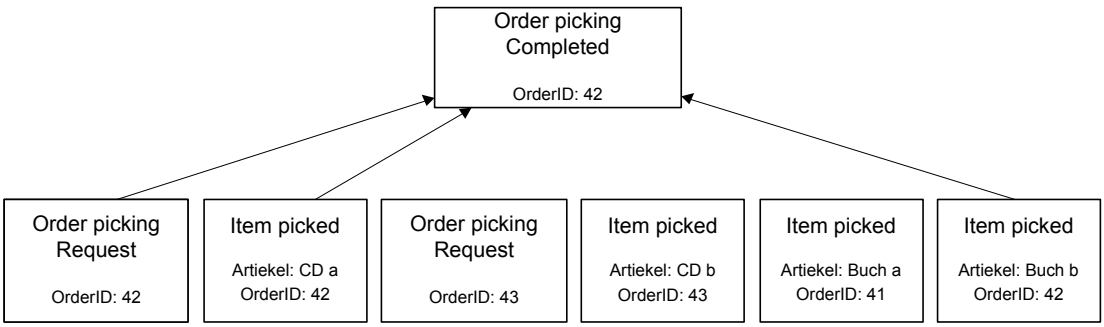

Abbildung 2.6: Events in einer Kommissionierung

Events können auch Nutz-Daten enthalten, den sogenannten Payload [EN10]. Diese sind vom jeweiligen *Event Type* abhängig und müssen jeweils einzeln definiert werden. Betrachten wir dazu ein Beispiel.

Beispiel 2.1 (Event mit Payload)
Der Bordcomputer eines Lkws produziert in regelmäßigen Abständen `Position` (d. h. Events vom *Event Type* `Position`) Events. Diese geben die aktuelle Position des Lkws bekannt. Als Payload könnten diese dabei beispielsweise die folgenden Daten enthalten: eine FahrzeugID, eine FahrerID und GPS Koordinaten. Ob noch weitere Informationen dem Event hinzugefügt werden müssen, hängt vom konkreten Anwendungsfall ab. Mithilfe dieser Events kann in Echtzeit zentral ausgewertet werden, wo sich ein oder mehrere Lkws befinden. Damit kann z. B. abgeleitet werden, in welchem Lager ein Lkw demnächst eintrifft.

Nachdem wir nun den allgemeine Event-Begriff eingeführt haben, betrachten wir nun den *Complex Event*-Begriff.

Complex Event Bei einem *Complex Event* handelt es sich um eine Aggregation von anderen Events [Luc02]. Die Teil-Events werden als die *Member*s des *Complex Event* bezeichnet. Die *Member*s können zu unterschiedlichen Zeitpunkten und in verschiedenen Komponenten von einem oder mehren von Systemen entstanden sein.

Somit können *Complex Events* verschiedene Sachverhalte darstellen. Dabei können die *Member*s Events von unterschiedlichen Systemen und über einen längeren Zeitraum gesammelt sein. Betrachten wir dazu ein Beispiel.

Beispiel 2.2 (Complex Event in einer Kommissionierung)
Wir betrachten im Folgenden die Events in einem Lagersystem. Für jede Bestellung muss

die Ware kommissioniert werden. Dies wird durch einen `Order picking Request` Event eingeleitet. Darin ist ein eindeutige `OrderID` und eine Liste mit Artikelnummern und gewünschter Menge enthalten. Zwei dieser Events sind in Abbildung 2.6 zu sehen (erstes und drittes Event in der unteren Reihe). Nach der Kommissionierung jedes Artikels wird vom Lagersystem automatisch ein `Item picked` Event produziert. In Abbildung 2.6 sind mehrere dieser Events von unterschiedlichen Bestellungen abgebildet (`OrderID` 41 - 43). Sobald alle Artikel kommissioniert sind, wird das komplexe Event `Order picking Completed` produziert. Dieses Event ist eine Aggregation aus dem `Order picking Request` Event und den zugehörigen `Item picked` Events. Diese drei Events sind damit die *Member*s des `Order picking Completed` Events. In Abbildung 2.6 ist nur ein `Order picking Completed` Event zu sehen, da die übrigen Kommissionierungen noch nicht abgeschlossen sind.

Dieses Beispiel demonstriert die Möglichkeiten von CEP. Die für sich genommen weniger wichtigen Events können zu *Complex Events* aggregiert werden, die eine wesentlich höhere Aussagekraft besitzen. Das `Order picking Completed` Event kann auch in andere *Complex Events* einfließen.

2.3 Beispiel eines Geschäftsprozesses mit Eventunterstützung

Nachdem in den vorherigen Abschnitten die Grundlagen für das ADEPT-BPM-System und CEP behandelt wurden, werden in diesem Abschnitt die Möglichkeiten von Events im Prozess-Managements-System anhand eines Beispiels erläutert. Dieses Beispiel schildert einen vereinfachten Warenbestell-Prozess. Das Beispiel orientiert sich nur grob an real existierende Bestellvorgänge.

Beispiel 2.3 (Bestell-Prozess)
Abbildung 2.7 a zeigt den Top-Level-Prozess des Bestell-Prozesses. Dieser besteht aus drei Aktivitäten. Als Erstes wird die `Bestellung Aufnehmen`-Aktivität ausgeführt. Hier werden Details zur Bestellung aufgenommen, darunter fallen Artikelnummern, Anzahl und Lieferadresse. Zusätzlich wird eine eindeutige `OrderID` vergeben, damit die Bestellung im weiteren Verlauf eindeutig identifizierbar ist. Nach dieser Aktivität erfolgt ein AND-Split, wodurch die `Bezahlen`- und `Versand`-Aktivität parallel ausgeführt werden. Bei beiden Aktivitäten handelt es sich um Subprozesse. Der Bezahlen-Prozess behandelt die Überwachung der ordnungsgemäßen Bezahlung und das Erstellen der Rechnung. Der Versand-Prozess regelt die Bereitstellung und den Versand der Ware. Mit Abschluss der beiden Subprozesse ist der Bestell-Prozess abgeschlossen.

Abbildung 2.7 b zeigt die oben genannten Subprozesse detaillierter. Zusätzlich sind noch die beiden externen Systeme Finanzverwaltung und ERP-Lagerverwaltung abgebildet, mit

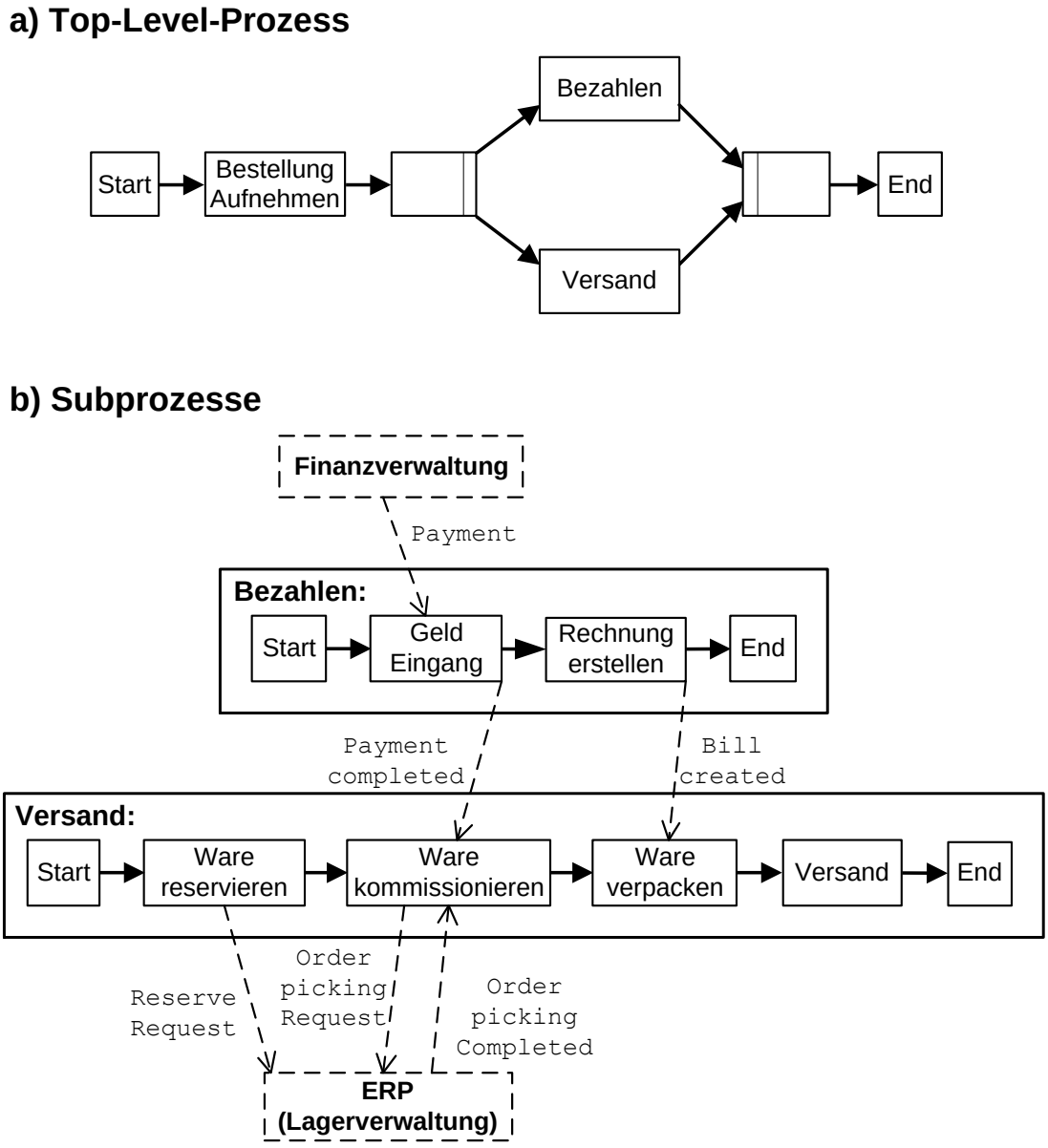

Abbildung 2.7: Bestell-Prozess

denen die beiden Subprozesse per Events kommunizieren. Als Erstes wird beim Bezahlen-Prozess die `Geld-Eingang`-Aktivität gestartet. Hier wird überprüft, ob der Rechnungsbetrag vollständig eingegangen ist. Hierfür werden die `Payment` Events des Finanzverwaltungssystems genutzt. Bei jeder eingehenden Zahlung wird automatisch ein `Payment` Event erzeugt, in dem die zugehörigen Daten, wie Betrag, Kontonummer und Bankleitzahl des Quellkontos usw. zu finden sind. In der `Geld Eingang`-Aktivität werden die eingehenden `Payment` Events analysiert. Sobald ein `Payment` Event eintrifft, welches eine vollständige Bezahlung des Rechnungsbetrages anzeigt, ist die `Geld Eingang`-Aktivität beendet. Zum Abschluss wird ein `Payment Completed` Event produziert. Dieses Event teilt allen Interessenten mit, dass der Rechnungsbetrag für diesen Bestellvorgang eingegangen ist. Die zweite und letzte Aktivität des Bezahlen-Prozesses ist die `Rechnung erstellen`-Aktivität. Zur Vereinfachung wird die Rechnung erst nach dem Zahlungseingang erstellt, damit diese ausgedruckt der Lieferung beigelegt werden kann. Dazu wird eine Rechnungsnummer vergeben und ein `Bill created` Event produziert. In diesem Event sind alle Informationen enthalten, damit die Rechnung später korrekt zusammen mit der Ware verpackt werden kann.

Im Versand-Prozess von Abbildung 2.7 b werden in der ersten Aktivität, `Ware reservieren`, die entsprechenden Artikel in der Datenbank als reserviert markiert. Dies geschieht hier jedoch nicht durch direktes Aufrufen im ERP-System, sondern mittels eines `Reserve Request` Events, welches vom ERP-System verarbeitet wird. Dieses Event enthält alle notwendigen Daten, wie Artikelnummer und -anzahl, damit das ERP-System die Waren als reserviert markieren kann. Die nächste Aktivität kann erst ausgeführt werden, nachdem der Zahlungsbetrag vollständig eingegangen ist. Deswegen wird zunächst auf das `Payment Completed` Event aus dem Bezahlen-Prozess gewartet. Sobald dieses eingetroffen ist, kann die `Ware kommissionieren`-Aktivität ausgeführt werden. In dieser Aktivität werden die vorher reservierten Artikel für den Versand zusammengestellt. Dazu wird ein `Order picking Request` Event produziert. Das ERP-System verarbeitet dieses Event und leitet die entsprechenden Maßnahmen ein. Im Gegensatz zur letzten Aktivität ist nach dem Versand des Events die `Ware kommissionieren`-Aktivität noch nicht abgeschlossen. Stattdessen muss für den nächsten Schritt sichergestellt sein, dass die Kommissionierung abgeschlossen ist. Dafür produziert das ERP-System ein `Order picking Completed` Event, welches allen Interessenten mitteilt, dass die Kommissionierung beendet ist. Events von diesem *Event Type* werden von der `Ware kommissionieren`-Aktivität abonniert. Empfängt die Aktivität nun ein entsprechendes Event, so kann die Aktivität abgeschlossen und die nächste gestartet werden.

Als vorletzter Schritt wird die `Ware verpacken`-Aktivität ausgeführt. Hier wird die kommissionierte Ware verpackt und zum Versand vorbereitet. Da der Lieferung ein Ausdruck der Rechnung beigelegt werden soll, muss auf das `Bill created` Event, vom Bezahlen-Prozess gewartet werden. In diesem Event stehen die entsprechenden Daten, damit ein Rech-

nungsausdruck korrekt zugewiesen werden kann. Als Letztes wird die `Ware versenden`-Aktivität ausgeführt. Hier wird die fertig verpackte Ware einen Transportdienstleister, wie z. B. DHL, übergeben. Damit ist der Versand-Prozess abgeschlossen.

Ein wesentlicher Aspekt, des hier behandelten Beispielprozesses, ist der Einsatz von Events zur Kommunikation. Hierbei unterscheiden wir zum einem die Kommunikation mit externen Systemen und zum anderen die prozessinterne Kommunikation. In diesem Beispiel haben wir als externe Komponenten das Finanzverwaltungssystem und das ERP-System für die Lagerverwaltung. Events können dabei ähnlich wie Web-Services[4] genutzt werden. Die Dienste sind lose gekoppelt und werden per Events aufgerufen. Als Beispiel dient hier das `Reserve Request` Event vom Versand-Prozess. Hierbei kann das ERP-System entweder selbst das Event verarbeiten oder die entsprechenden Funktionen werden mittels eines Adapters aufgerufen. Somit lassen sich auch Legacy-Anwendungen in das BPM-System integrieren.

Ebenso ist mit Events auch eine Inter-Prozess-Kommunikation möglich [HA99]. Dies bedeutet, dass Aktivitäten aus unterschiedlichen Prozessen miteinander kommunizieren können. Ein Beispiel dafür ist das `Payment Completed` Event des Bezahlen-Prozesses. Mittels Events ist eine Synchronisation über Prozessgrenzen hinweg möglich. Wenn statt Events Synchronisationskanten verwendet werden, muss der Bezahlen-Prozess abgeschlossen sein, damit der Versand-Prozess gestartet werden kann. Die Alternative dazu ist, auf die Nutzung von Subprozessen zu verzichten. In diesem Fall kann eine Synchronisationskante direkt von der `Gelde Eingang` Aktivität zur `Ware kommissionieren` Aktivität gezogen werden. In diesem Fall ist auch ein Datenaustausch per Datenelemente möglich. Allerdings verlieren wir so die Möglichkeit, die einzelnen Funktionen in eigene wiederverwendbare Prozesse auszulagern.

Ein weiterer Vorteil bei der Verwendung von Events, ist die Verbesserung der Wiederverwendbarkeit von Aktivitäten, Prozessen und Komponenten. Durch die Kapselung mit Events ist es z. B. möglich, den Bezahlen- und Versand-Prozess auch in andere Prozesse zu integrieren. Hier muss dann lediglich auf die Verfügbarkeit der Events geachtet werden. Zusätzlich können die externen Komponenten, wie z. B. das ERP-System, jederzeit ausgetauscht werden, solange das neue System die gleichen Events verwendet. Dies kann im Bedarfsfall auch mittels Adapter realisiert werden.

[4] vergleiche hierzu Serviceorientierte Architektur (SOA) [Jos08]

Da die Funktionen der externen Systeme nur lose gekoppelt sind, können die entsprechenden Aktivitäten diese unabhängig von den tatsächlichen Systemen verwenden. Durch diese Unabhängigkeit erhöht sich wiederum die Wiederverwendbarkeit der Aktivitäten und Prozesse.

Ein weiterer wichtiger Aspekt von Events, die in diesem Beispiel behandelt werden, ist, dass die Events auch von anderen Prozessen oder Systemen verwendet werden können. Die im Beispiel verwendeten Events können vielseitig genutzt werden. So kann z. B. das `Payment` Event nicht nur von der `Geld Eingang`-Aktivität genutzt werden, sondern auch von allen möglichen anderen Aktivitäten und Systemen. Die hier genutzten Events können auch mit anderen Events verknüpft werden, um *Complex Events* zu produzieren. Das `Order picking Completed` ist in diesem Fall schon ein *Complex Event*, welches unter anderem aus dem `Order picking Request` Event entstanden ist (siehe hierzu Beispiel 2.2).

2.4 Anforderungen

Wir haben in diesem Kapitel die Grundlagen für BPM und Event Processing behandelt und diese in einem Prozessbeispiel miteinander verknüpft. Daraus leiten wir nun mehrere Anforderungen ab, die ein Event-Mechanismus im Prozess-Management leisten muss.

Anforderung 1: Lose Kopplung bedeutet eine (hohe) Unabhängigkeit zwischen mehreren Komponenten. So kann an einer Komponente oder einem System Änderungen durchgeführt werden, ohne das andere Systeme oder Prozesse geändert werden müssen. Im Beispiel 2.3 kann z. B. das ERP-System ausgetauscht werden, ohne dass der Versand-Prozess davon beeinträchtigt wird. Dazu ist es nötig, dass Sender und Empfänger von Events keinerlei Kenntnis voneinander benötigen. Damit trotzdem die Events ihr Ziel finden, wird ein flexibles Eventverteilungssystem benötigt. Dabei müssen auch neue Quellen und Ziele hinzugefügt und entfernt werden können.

Anforderung 2: Inter-Prozess-Kommunikation bedeutet, dass Aktivitäten aus verschiedenen Prozessen per Events miteinander kommunizieren können. Damit können auch während der Ausführung der Prozesse Nachrichten und Daten

ausgetauscht werden. Im Beispiel 2.3 wird z. B. das `Payment Completed` Event zwischen den Bezahlen- und Versand-Prozess versandt. Auch hierzu ist ein Eventverteilungssystem von entscheidender Bedeutung.

Anforderung 3: Asynchrone Kommunikation bedeutet, dass Sender und Empfänger zeitlich versetzt agieren [Jos08]. Der Sender soll nicht blockieren, bis eine Antwort eingeht oder bis der Vorgang abgeschlossen ist, sondern soll zunächst weitere Aufgaben bearbeiten können. Ein Beispiel hierfür ist das `Payment Completed` Event aus Beispiel 2.3. Dieses Event kann produziert werden, bevor die empfangende Aktivität (`Ware kommissionieren`) gestartet ist. Bei der Eventverteilung muss dementsprechend darauf geachtet werden, dass Events auch für später gestartete Aktivitäten zur Verfügung stehen. Dementsprechend müssen sie für solche Aktivitäten gespeichert werden.

Anforderung 4: Anbindung externer Systeme bedeutet die freie Integration von externen Systemen mit dem BPM-System. Darunter fallen z. B. ERP- und CRM-Systeme. Diese müssen per Events mit den Prozessen des BPM-Systems kommunizieren können. Dadurch sind die Systeme mithilfe der Events mit dem BPM-System verbunden. Auch hier wird ein Eventverteilungssystem benötigt. Allerdings müssen diese Systeme auch flexibel eingebunden und wieder entfernt werden können.

Anforderung 5: Anbindung von Legacy-Systemen verhält sich ähnlich wie die Anbindung von externen Systemen. Allerdings haben diese Systeme keine Event Processing Funktionalitäten. Ihr weiterer Einsatz ist aber häufig aus wirtschaftlichen Gründen gewünscht [Jos08]. Um diese mit dem Event Processing System zu verbinden, ist eine Anbindung von Adaptern notwendig.

Anforderung 6: Erhöhung Wiederverwendbarkeit bedeutet, dass Aktivitäten, Prozesse oder Komponenten besser in anderen Prozessen oder Systemen wiederverwendet werden können. Damit müssen diese nicht für jeden Einsatzzweck neu entwickelt werden. Im Beispiel 2.3 bedeutet dies z. B., dass der Bezahlen-Prozess auch in anderen Prozessen verwendet werden kann. Dazu kann eine Kapselung in Events verwendet werden.

3 Event Processing Network

Zunächst soll eine abstrakte Beschreibung einer allgemeinen Event Processing Network (EPN)-Architektur behandelt werden. Ein EPN ist eine spezielle Variante von Event Processing. Diese bildet die Grundlage für die, in dieser Diplomarbeit, entwickelte Event Processing Architektur für BPM-Systeme.

Betrachten wir zum Einstieg Abbildung 3.1. Hier ist eine vereinfachte Darstellung einer allgemeinen Event Processing Architektur abgebildet. Links sind die *Event Producers* dargestellt. Hierbei handelt es sich um unterschiedliche Hard- und Softwaresysteme, die Events produzieren und in das Event Processing System einbringen. Im Event Processing System werden die Events verarbeitet und anschließend von den rechts dargestellten *Event Consumers* empfangen. Diese stehen stellvertretend für die Hard- und Softwaresysteme, welche die Events konsumieren. Das Event Processing ist ein kontinuierlicher Prozess. Die verarbeiteten Events können selbst wieder neue *Complex Events* auslösen. Diese neuen Events können dann von den *Event Consumers* verarbeitet werden. Allerdings können sie auch vom Event Processing System mit anderen Events kombiniert werden und neue *Complex Events* bilden. Dieser Vorgang ist theoretisch beliebig häufig wiederholbar.

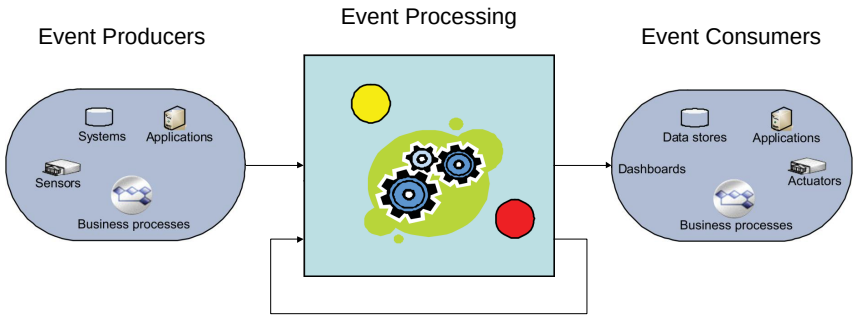

Abbildung 3.1: Abstrakte Darstellung einer Event Processing Architektur (angelehnt an [EN10])

In diesem Kapitel diskutieren wir eine Architektur, bei der das Event Processing mithilfe eines Netzwerks aus *Event Processing Agents* erfolgt. Dazu behandeln wir zunächst die Elemente eines EPNs. Anschließend wird eine allgemeine EPN-Architektur vorgestellt.

3.1 Elemente eines Event Processing Network

Das zentrale Element eines EPNs, das Event, haben wir bereit in Abschnitt 2.2.2 behandelt. Nun beschreiben die weiteren Bausteine eines EPNs: *Event Producer*, *Event Consumer* und *Event Processing Agent*.

3.1.1 Event Producer und Consumer

Wir beginnen nun mit den in Abbildung 3.1 angedeuteten *Event Producers* und *Event Consumers*. Allerdings diskutieren wir sie hier als Elemente in einem EPN.

Event Producer Ein *Event Producer* (EP) ist eine Entität am Rande eines Event Processing Systems, welches Events ins System einbringt [EN10].

Betrachten wir hierzu noch einmal Abbildung 3.1. Dort sind die *Event Producers* auf der linken Seite dargestellt. Diese können ausschließlich Events genieren, diese aber nicht empfangen oder weiterverarbeiten. Deswegen befinden sie sich „am Rande des Event Processing Systems“ und stellen die Quelle aller Events dar. *Event Producer* können sowohl Hard- als auch Softwarekomponenten sein. Ein Beispiel für einen *Event Producer* ist der bereits zuvor, in Beispiel 2.1 diskutierte, GPS-Sensor in einem Lkw. Dieser kann z. B. aus den empfangenen GPS-Daten regelmäßig `Position` Events produzieren und in das Event Processing System einspeisen. Dadurch ist dem System immer die aktuelle Position des Lkws bekannt. Alle anderen Komponenten, die Kenntnis über die Position der Lkws benötigen, können diese nun vom Event Processing System erhalten. Alternativ kann es sich bei einem *Event Producer* beispielsweise auch um eine Softwarekomponente handeln, die beim Eintreten von bestimmten Fehlersituationen Events produziert. Als Payload können diese Fehler-Events beispielsweise einen Fehlercode oder eine komplette Fehlerbeschreibung enthalten.

Schließlich können *Event Producers* auch Adapter für Hard- oder Softwaresystem sein, die selber keine Event-Funktionalitäten besitzen. Auf diesem Weg können unter

anderem Legacy-Anwendungen mit dem System verbunden werden. Dazu muss ein entsprechender Adapter-EP entwickelt werden. Dieser hat die Aufgabe, das entsprechende System zu überwachen und aus den Daten, die von diesem erzeugt werden, Events zu produzieren. Nehmen wir als Beispiel wieder den GPS-Sensor. Wir gehen davon aus, dass dieser selber nicht Events produzieren kann. Stattdessen hat der GPS-Sensor eine Schnittstelle. Mit dieser kann ein System die derzeitigen GPS-Koordinaten auslesen. Ein Adapter-EP hat nun die Aufgabe, in vorher festgelegten Zeitintervallen die Positionsangabe des GPS-Sensors abzurufen. Daraufhin produziert der Adapter-EP `Postition` Events. Diese Events enthalten unter anderem die GPS-Koordinaten als Payload.

Event Consumer Ein *Event Consumer* (EC) ist eine Entität am Rande eines Event Processing Systems, welche Events vom System empfängt [EN10].

Die *Event Consumers* empfangen Events vom Event Processing System, können aber keine Events produzieren. Deswegen befinden sie sich genau wie die *Event Producers* „am Rande des Event Processing Systems“. Ihre Aufgabe liegt in der finalen Verarbeitung der Events. Die Verwendungsmöglichkeiten sind vielfältig. Events können für eine spätere Analyse gespeichert oder direkt in einem Programm zur Anzeige gebracht werden. Beispielsweise kann ein Programm die, zuvor diskutierten, `Position` Events von mehreren Lkws verwalten. Durch die Verarbeitung dieser Events kann das Programm ermitteln, wo sich die Lkws gerade befinden. Damit lässt sich z. B. eine Karte erstellen, auf der alle Lkws mit ihren jeweiligen Positionen aufgeführt sind. Eine alternative Verwendungsmöglichkeit ist ein Lagersystem, bei dem mithilfe der `Position` Events ermittelt wird, wann ein Lkw voraussichtlich eintrifft.

Genau wie bei EPs können auch ECs als Adapter für Legacy-Anwendungen genutzt werden. Um eine Legacy-Anwendung an ein Event Processing System anzubinden, muss ebenfalls ein Adapter-EC erstellt werden. Dieser hat die Aufgabe, stellvertretend für die Legacy-Anwendung, die Events zu empfangen. Daraufhin verarbeitet der Adapter-EC die Events. Mithilfe der in den Events enthaltenen Daten können dann entsprechende Funktionen in der Legacy-Anwendung aufgerufen werden. Betrachten wir als Beispiel hierfür den Bestell-Prozess von Beispiel 2.3. Das dort verwendete ERP-System für die Lagerverwaltung erhält hier `Reserve` Events. Darin befindet sich eine Auflistung, welche Artikel reserviert werden sollen. Der EC kann nun diese

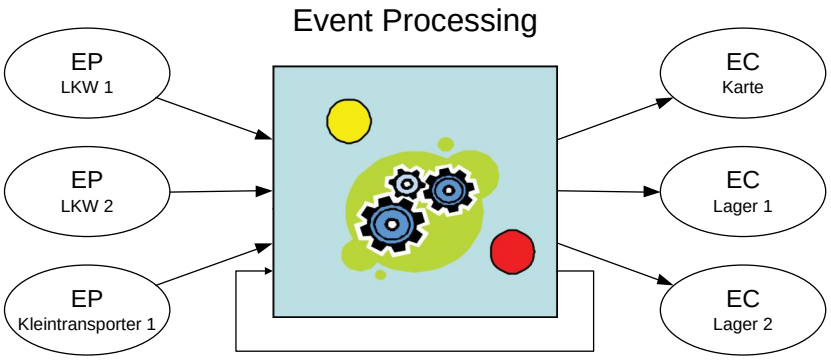

Abbildung 3.2: Beispiel mit mehreren Produzenten und Konsumenten

Informationen auswerten und damit die entsprechende Funktion im ERP-System aufrufen.

Ein wichtiger Aspekt von EPs und ECs ist, dass sie keine Kenntnis voneinander haben müssen und somit vollständig entkoppelt sein können. Dadurch lassen sich die entsprechenden Komponenten sehr leicht austauschen und erweitern. Das Event Processing System hat dabei die Aufgabe, die Events vom EP zum EC zu leiten. Zudem kann das Event Processing System die Events filtern oder anderweitig bearbeiten.

Beispiel 3.1 (Event Producers und Consumers)
Abbildung 3.2 zeigt ein Beispiel für das Zusammenspiel von *Event Producers* und *Consumers*. Links sind drei EPs für Lkws und Kleintransporter dargestellt. Diese drei EPs stehen stellvertretend für einen ganzen Fuhrpark aus Lkws, Kleintransportern und sonstigen Transportfahrzeugen. Die EPs produzieren `Position` Events. Ein Event Processing System verarbeitet diese Events und verteilt sie an die, auf der rechten Seite dargestellten, ECs. Der in der Abbildung ganz oben dargestellte EC hat die Aufgabe, die Positionen der einzelnen Fahrzeuge auf einer Karte darzustellen. Die anderen ECs bekommen vom Event Processing System nur `Position` Events, die von Lkws stammen, welche als Ziel das zugehörige Lager haben. Mit diesen Events haben die Lagersysteme Kenntnis darüber, wo sich die für sie relevanten Lkws befinden. Damit können sie besser beurteilen, wann welcher Lkw bei ihnen eintrifft. Entscheidend ist hierbei immer, dass keine direkte Kommunikation zwischen den *Event Producers* (Transportfahrzeugen) und den *Event Consumers* (Lager-Systeme) besteht. Beide sind voneinander entkoppelt. Damit ist es leicht möglich, neue EPs und ECs, die `Position` Events produzieren bzw. konsumieren, ins System aufzunehmen.

3.1.2 Event Processing Agents

Nachdem wir in den vorherigen Abschnitten Events und deren Produzenten und Konsumenten behandelt haben, diskutieren wir nun die Verarbeitung der Events zwischen EPs und ECs. Abbildung 3.1 stellt das Event Processing zwischen EPs und ECs noch als monolithische Komponente dar. Im Folgenden behandeln wir einen Ansatz, bei dem die Verarbeitung der Events, zwischen EPs und ECs, von einem Netzwerk aus *Event Processing Agents* (EPA) übernommen wird.

Im Gegensatz zu EPs und ECs können EPAs Events sowohl senden als auch empfangen. Des Weiteren können EPAs sowohl Produzenten als auch Konsumenten von Events sein. Die Funktionen von EPAs sind sehr vielfältig und reichen von einfachen Routing-Aufgaben über Filterung und Transformation von Events, bis hin zu Erkennung von komplexen Mustern. Im einfachsten Fall nehmen EPAs Events entgegen und leiten diese an andere EPAs bzw. ECs weiter.

EPAs können grob nach folgenden Schemata eingeordnet werden:

Filter Agents nehmen Events entgegen und haben die Aufgabe nicht relevante Events zu entfernen bzw. nicht weiterzuleiten. Beispielsweise könnte nach `Position` Events gefiltert werden, die von einem bestimmten Lkw produziert wurden.

Transformation Agents modifizieren den Inhalt von eingehenden Events. Hier sind verschiedene Arten möglich, unter anderem entfernen bzw. hinzufügen von Payload Elementen, Aufteilung in mehrere Events oder Aggregation von mehreren Events zu einem.

Pattern Detect Agents nehmen einen oder mehrere Ströme von Events entgegen und versuchen darin ein vorher definiertes (komplexes) Mustern zu finden. Ein Muster wäre z. B. drei aufeinanderfolgende `Logon Failed` Events ohne mindestens ein dazwischenliegendes `Logon` Event. Der *Pattern Detect Agent* produziert als Reaktion auf das gefundene Muster ein neues *Complex Event*.

Ein EPA hat die Möglichkeit mehrere dieser Funktionen gleichzeitig zu erfüllen. Er kann aber auch nur eine Funktion, z. B. Filter-EPA, haben. In einigen Fällen übernimmt ein EPA auch nur die Funktion, Events an andere EPAs und ECs zu verteilen. Abbildung 3.3 stellt ein Beispiel mit unterschiedlichen EPAs dar. Drei EPs sind dabei an jeweils einen Filter EPA angeschlossen. Die Filter EPAs nehmen die Events von

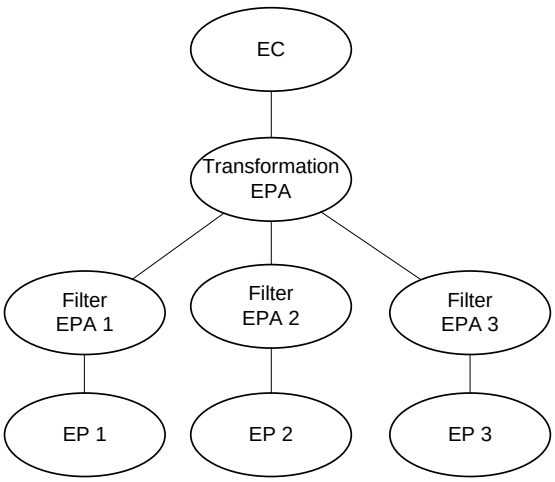

Abbildung 3.3: Kombination von Filter-EPAs mit einem Transformation-EPA

den EPs entgegen und leiten alle Events weiter, die der vorher festgelegten Filterregel entsprechen. Ziel der gefilterten Events ist der Transformation Agent, welcher mit den Filter EPAs verbunden ist. Dieser hat die Aufgabe, die eingehenden Events zu aggregieren und die daraus resultierenden Events an den ganz oben abgebildeten EC weiterzuleiten.

In anderen EPN-Architekturen [EN10, Luc02] wird bei der Verbindung zwischen zwei EPAs häufig unterschieden, in welche Richtung die Events gesendet werden können. Auf diese Unterscheidung verzichten wir in dieser Arbeit. Daraus folgt, dass zwei miteinander Verbundene EPAs die Möglichkeit haben, sich gegenseitig Events zu schicken. Da die Verbindung zwischen zwei EPA in beide Richtungen verläuft, ist sie bidirektional. Die Verbindung zwischen einem EP bzw. EC und einem EPA ist dagegen unidirektional, d. h. sie geht nur in eine Richtung. Ein EP kann lediglich Events produzieren und keine empfangen. Umgekehrt kann ein EC nur Events empfangen aber keine versenden.

3.2 Event Processing Networks

Wir haben bisher die einzelnen Elemente eines EPNs behandelt. Im Folgenden diskutieren wir nun, wie diese zu einem Netzwerk zusammengeschlossen werden können.

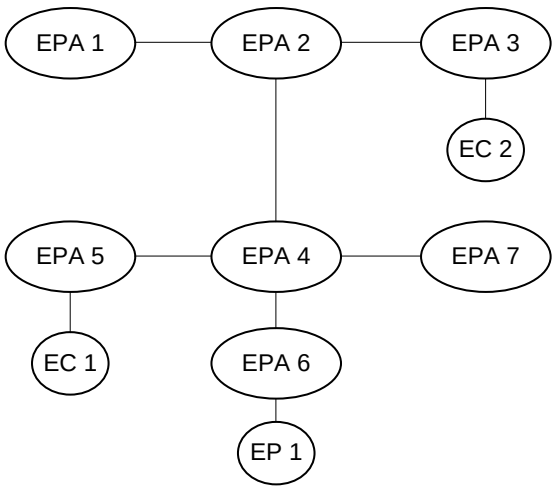

Abbildung 3.4: Beispiel für ein Event Processing Network

3.2.1 Aufbau

Event Processing Network Ein Event Processing Network (EPN) ist eine Sammlung von *Event Processing Agents*, *Event Producers* und *Event Consumers*, die miteinander verbunden sind. Diese Komponenten bilden einen zusammenhängenden, ungerichteten Graphen.

Ein Beispiel für ein EPN zeigt Abbildung 3.4. Hier sind mehrere EPAs miteinander verbunden. Hinzu kommt, dass an EPA 3 und 5 jeweils ein EC angeschlossen ist und EPA 6 mit einem EP verbunden ist.

Eine Besonderheit von EPNs ist, dass sie dynamisch zur Laufzeit verändert werden können. Es können neue EPAs hinzugefügt und wieder entfernt werden. Gleiches gilt für EPs und ECs. Dies erhöht die Flexibilität des Event Processing Systems. Dadurch können angeschlossene Systeme auch während des Betriebs ausgetauscht werden. Betrachten wir dazu als Beispiel noch einmal die GPS-Sensoren (vergleiche Beispiel 2.1). Jedes Mal wenn ein Lkw losfährt, wird ein neuer EP instanziiert und mit einem EPA verbunden. Daraufhin produziert der EP `Position` Events. Wird der Lkw wieder im Fuhrpark abgestellt, wird auch der EP beendet und vom EPN getrennt.

3.2.2 Routing von Events

Bei der Einführung von EPAs wurde schon angemerkt, dass eine der Aufgaben für einen EPA das Routing[5] von Events ist. Das bedeutet, dass der EPA eingehende Events an die mit ihm verbundenen EPAs und ECs weiterleitet. Die Weiterleitung der Events geschieht allerdings nur, wenn Interesse am entsprechenden *Event Type* angemeldet wurde.

Beispiel 3.2 (Event Routing in einem EPN)
Angenommen EP 1, aus dem in Abbildung 3.4 dargestellten EPN, ist die einzige Quelle von Events von Typ X. Die beiden *Event Consumer* EC 1 und EC 2 wollen Events dieses Typs konsumieren. Es existieren sonst keine weiteren Konsumenten für X Events. Da keine direkte Verbindung zwischen EP 1 und EC 1 bzw. EC 2 existiert, müssen die Events von EP 1 über mehrere EPAs geleitet werden. In diesem Fall schickt EP 1 seine Events an EPA 6. Dieser wiederum leitet die Events an EPA 4 weiter. An dieser Stelle werden nun die Events an mehrere EPAs verteilt. Zum einem müssen die Events an EPA 5 und zum anderen an EPA 2 weitergeleitet werden. EPA 5 ist direkt mit EC 1 verbunden. Dadurch kann das Event direkt an das erste Ziel (EC 1) weitergeleitet werden. Um EC 2 zu erreichen, muss zunächst EPA 2 das Event an EPA 3 weiterleiten. Von dort aus kann direkt das zweite Ziel (EC 2) erreicht werden.

Dieses einfache Beispiel zeigt bereits einige der Herausforderungen beim Routing von Events in einem EPN. So muss auch bei einem sich verändernden EPN sichergestellt sein, dass alle Events ihre Ziele finden. Dabei gilt, dass ein Anbieter von Events keine Informationen darüber hat, von welchen Konsumenten seine Events konsumiert werden. Analog dazu hat ein Konsument keine Kenntnis darüber, welche Anbieter für die von ihm konsumierten Events existieren. Um das Event Routing von der Quelle zu den Zielen dennoch zu ermöglichen, kann beispielsweise ein Typ basiertes *Publish/Subscribe*-System verwendet werden [EFGK03]. Das *Publish/Subscribe* funktioniert grob nach folgendem Schema. Ein Produzent (*Publisher*) von Events kündigt an, dass er Events von bestimmten *Event Types* produziert, d. h. er *published* den *Event Type*. Diese *publish* Ankündigung wird nun an alle mit dem Produzenten verbundenen Knoten weitergeleitet. Diese leiten darauf die Ankündigung wiederum weiter. Auf diese Weise wird die *publish* Ankündigung sukzessive durch das gesamte Netzwerk geleitet.

Auf der anderen Seite reagiert der Konsument (*Subscriber*) auf einen *publish*, indem er sein Interesse an den *Event Type* ankündigt, d. h. er abonniert (*subscribes*) den

[5] weiterleiten

Event Type. Diese *subscribe* Anfrage kann der Konsument nun an die möglichen Produzenten richten. Auch hier verteilt sich die *subscribe* Anfrage sukzessive im gesamten Netzwerk. Dadurch ist es möglich, dass ein Event von seiner Quelle zu seinen Zielen geroutet werden kann.

Beispiel 3.3 (Publish/Subscribe in einem EPN)
Gegeben sei das EPN aus Abbildung 3.4. EP 1 produziert dort Events vom Type `x`. Deswegen published dieser den *Event Type* an EPA 6. Von dort aus wird die *publish* Ankündigung durch das gesamte EPN geleitet. EC 1 konsumiert Events vom Typ `x`. Daher abonniert EC 1 diesen *Event Type* bei EPA 5. Von dort aus wird die *subscribe* Anfrage über die EPAs 4 und 6 an EP 1 weitergereicht. Die *subscribe* Anfrage wird nicht von EPA 4 an EPA 2 weitergereicht, da EPA 2 nur über EPA 4 Events von Typ `x` erhalten kann.

Wir haben bei diesem *Publish/Subscribe*-Schema ausgeblendet, dass im EPN auch Zyklen enthalten sein können. Dadurch ist die Verteilung der *Publish/Subscribe* Anfragen nicht eindeutig. Dies werden wir an dieser Stelle nicht weiter verfolgen. Bei der Behandlung einer konkreten EPN-Architektur für BPM-System wird dieser Aspekt näher beleuchtet werden.

3.3 Zusammenfassung

In diesem Kapitel haben wir eine allgemeine EPN-Architektur behandelt. Ein EPN besteht hier aus den drei Elementen EP, EC und EPA. Es wurde diskutiert, welche Aufgaben diese haben und wie sie miteinander, innerhalb des EPNs, in Verbindung stehen. Dabei haben wir auch in vereinfachter Form ein *Publish/Subscribe*-Mechanismus für dieses EPN diskutiert. Im folgenden Kapitel soll nun eine EPN Architektur ausgearbeitet werden, die konkret für die Bedürfnisse von BPM-Systeme ausgerichtet ist.

4 EPN-Architektur für BPM-Systeme

In den vorherigen Kapiteln haben wir die Grundlagen von BPM, CEP und EPN diskutiert. In diesem Kapitel sollen nun diese Techniken kombiniert werden. Ziel dieses Kapitel ist es dabei, eine Architektur zu entwickeln, welche auf der Grundlage von EPNs ein Event Processing in Geschäftsprozessen ermöglicht.

Dazu werden wir zunächst die Grundidee dieser Architektur in abstrakter Weise behandeln (Abschnitt 4.1). Als nächstes werden die EPEs aus dem vorherigen Kapitel genauer betrachtet. Dabei werden ihre Funktionen und Aufbau im Rahmen der hier entwickelten Architektur diskutiert (Abschnitt 4.2). Danach wird der Aufbau von Events detaillierter betrachtet (Abschnitt 4.3); dabei werden die einzelnen Attribute und der Aufbau der Payload näher behandelt. In Abschnitt 4.4 diskutieren wir dann die dynamische Anpassung des EPNs zur Laufzeit. Dabei gehen wir darauf ein, welche Veränderungen am EPN vorgenommen werden müssen, wenn ein Prozess gestartet oder beendet wird und wenn externe Systeme mit dem EPN verbunden werden sollen. Daraufhin passen wir den *Publish/Subscribe*-Mechanismus an die Gegebenheiten dieser EPN-Architektur an (Abschnitt 4.5). In Abschnitt 4.6 behandeln wir den Event-Manager, eine Erweiterung für ein BPM-System, welche die zuvor beschriebenen Aufgaben koordiniert. Abgeschlossen wird dieses Kapitel durch eine Zusammenfassung (Abschnitt 4.7).

4.1 Motivation

In diesem Abschnitt stellen wir das Grundkonzept für eine spezialisierte Event Processing Architektur vor. Diese Architektur hat die Aufgabe, ein umfangreiches Event Processing in BPM-Systemen zu ermöglichen.

Ein sehr wichtiger Aspekt des Event-Mechanismus ist, dass Prozess-Instanzen dynamisch zur Laufzeit gestartet und beendet werden. Des Weiteren ist zu beachten, dass auch externe Systeme (vergleiche Abschnitt 2.4) dynamisch hinzufügbar und

austauschbar sein müssen. Daraus folgt, dass ein dynamisches System benötigt wird, bei dem sowohl Prozess-Instanzen als auch externe Systeme neu hinzukommen und wieder entfernt werden können.

Wie in Abschnitt 2.1.1 diskutiert, haben ADEPT und viele andere BPM-Systeme die Möglichkeit, Prozesse hierarchisch zu strukturieren [FG08], dies bedeutet, einen Prozess in mehrere Subprozesse zu unterteilen. Die einzelnen Subprozesse können wiederum in weitere Subprozesse unterteilt werden. Dadurch ist es unter anderem auch möglich, einzelne Subprozesse wiederzuverwenden. Zusätzlich erhöht sich die Übersichtlichkeit, da nicht immer der gesamte Prozess betrachtet werden muss; stattdessen kann auch nur ein Teil des Gesamtprozesses betrachtet werden. Ein Beispiel für einen Prozess mit Subprozessen ist der Bestell-Prozess aus Beispiel 2.3.

Dieses hierarchische Konzept übertragen wir nun auf eine EPN-Architektur. Dabei soll ein baumförmiges EPN entstehen. Beim Starten einer Prozess-Instanz wird dieser ein eigener EPA zugewiesen, dies gilt sowohl für die Instanz eines Top-Level-Prozesses als auch für Subprozesse. Der EPA ist damit ein Knoten im *EPN-Baum*. Jeder EPA übernimmt die Event Processing Aufgaben für seine jeweilige Prozess-Instanz. Zusätzlich wird der EPA eines Subprozesses mit dem EPA des jeweiligen Vaterprozesses verbunden. Der jeweilige Vaterknoten eines EPAs wird dabei *SuperEPA* genannt. Durch diesen Baum aus EPAs ist es dann möglich, dass die Prozess-Instanzen mit anderen Prozess-Instanzen mit Events kommunizieren können.

Betrachten wir hierzu als Beispiel die EPAs für eine Instanz des Bestell-Prozesses aus Beispiel 2.3. In Abbildung 4.1 sind die drei zugehörigen EPAs abgebildet, jeweils einer für den Bestell-, Bezahlen- und Versand-Prozess. Die beiden EPAs für den Bezahlen- und Versand-Prozess haben einen gemeinsamen SuperEPA, nämlich den EPA des Bestell-Prozesses. Zusammen bilden diese drei EPAs einen EPN-Baum, mit dem EPA des Bestell-Prozesses als Wurzelelement. Allgemein hängt dabei die Tiefe des EPN-Baumes von der Tiefe der Subprozesse ab.

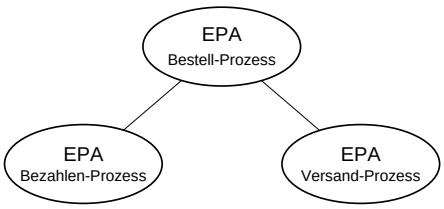

Abbildung 4.1: EPAs für eine Bestell-Prozess-Instanz

Die Erstellung der einzelnen EPAs erfolgt hierbei dynamisch zur Laufzeit. Wenn das BPM-System einen Prozess startet, wird auch ein EPA erstellt. Dies gilt auch beim Starten eines Subprozesses. Ein einzelner Prozess kann sich gleichzeitig mehrfach in Ausführung befinden. Für jede einzelne Instanz muss ein EPA erstellt werden. Wird ein Prozess bzw. Subprozess beendet, wird der entsprechende EPA aus dem System entfernt. Die einzelnen EPN-Bäume werden mit einem zentralen EPA des Gesamtsystems verbunden. Diesen EPA bezeichnen wir als *MainEPA*. An den MainEPA können nicht nur EPAs von Prozess-Instanzen angeschlossen werden, sondern auch andere EPs, ECs und EPAs. Dadurch ist es möglich, auch externe Komponenten mit dem EPN des BPM-Systems zu verbinden. Eine Möglichkeit ist z. B. ein Monitor-EC, der eine Übersicht über alle abgeschlossenen Zahlungen aus den Bestellvorgängen liefert. Dazu abonniert dieser den `Payment Completed` *Event Type*. Dadurch erhält der Monitor-EC jedes Mal, wenn die Aktivität `Bezahlen` abgeschlossen wurde, ein `Payment Completed` Event.

Abbildung 4.2 zeigt ein Beispiel für solch einen EPN-Baum. Der MainEPA ist mit insgesamt vier EPAs. und einem EC verbunden: jeweils ein EPA für Prozess 1 und 2, ein EPA für das ERP-System und ein Filter-EPA, der wiederum mit einem Monitor-EC verbunden ist. Die EPAs für die beiden Prozesse 1 und 2 sind jeweils mit weiteren EPAs für Subprozesse verbunden. Diese repräsentieren beliebige Prozess-Instanzen. Prozess 1 kann z. B. der Bestell-Prozess sein, aber auch andere Prozesse sind möglich. Der Monitor-EC kann z. B. die Aufgabe haben, `Payment Completed` Events zu sammeln und daraus eine statistische Auswertung zu erstellen. Der vorgeschaltete Filter-EPA hat die Aufgabe, nicht relevante Events zu entfernen. Der EPA für das ERP-System bietet die Möglichkeit, dass z. B. dieses System mit Instanzen des Versand-Prozesses mithilfe von Events kommunizieren kann.

Einer der Vorteile von Bäumen gegenüber allgemeineren Graphen ist, dass sie azyklisch sind. Dadurch sind die Knoten, die ein Event von der Quelle bis zum Ziel durchlaufen muss, immer eindeutig. Nehmen wir als Beispiel die Situation, der EPA von Prozess 2 produziert ein Event, welches an den Monitor-EC geroutet werden soll. Wenn wir davon ausgehen, dass ein EPA ein eingehendes Event nicht an den Knoten zurückschickt, von dem er das Event erhalten hat, ist der Weg immer eindeutig. In diesem Fall verläuft der Weg des Events von EPA von Prozess 2 an den MainEPA zu den Filter EPA und zuletzt an den Monitor-EC.

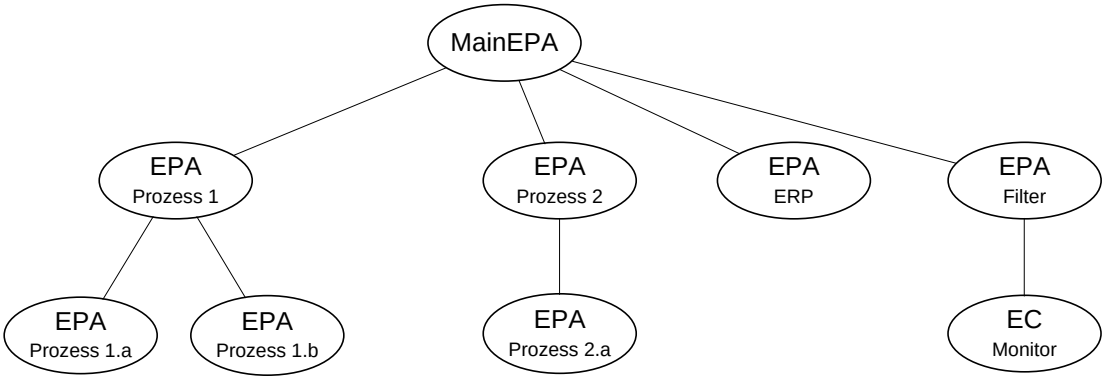

Abbildung 4.2: Beispiel für einen EPN-Baum

Die hier abstrakt dargestellte EPN-Architektur bildet das Grundkonzept für die in dieser Arbeit entwickelte Event Processing Architektur. Im Folgenden werden wir dieses Konzept weiter vertiefen. Dabei diskutieren wir unter anderem die Aufgaben der EPAs, die dynamische Anpassung des EPNs und passen das *Publish/Subscribe*-System aus Abschnitt 3.2.2 an die Anforderungen dieser Architektur an.

4.2 Event Processing Elements

Wir diskutieren nun die einzelnen Elemente der in dieser Arbeit entwickelten EPN-Architektur. Ein EPN besteht, in dieser Architektur, aus drei verschiedenen Elementen: *Event Producer* (EP), *Event Consumer* (EC) und *Event Processing Agent* (EPA). Diese drei Elemente werden auch als *Event Processing Elements* (EPE) bezeichnet. Im Folgenden werden die genauen Aufgaben und Eigenschaften dieser Elemente genauer erörtert. Zusätzlich behandeln wir noch spezialisierte Formen für Prozess-Instanzen.

4.2.1 Event Producer

Wie bereits im vorherigen Kapitel erwähnt, ist die Aufgabe der *Event Producers*, beim Eintreten von Ereignissen, Events zu produzieren und in das EPN einzubringen. Um die Baumeigenschaft sicherzustellen, ist ein EP immer nur mit seinem Vaterknoten (SuperEPA) verbunden und hat selbst keine Kindknoten. Wie bereits schon in

Abschnitt 3.1.1 angedeutet, können über EPs verschiedene Systeme mit dem Event Processing System bzw. mit dem BPM-System kommunizieren.

Beispiel 4.1 (EP für das Finanzverwaltungssystem)
Betrachten wir das Finanzverwaltungssystem aus dem Bestell-Prozess von Beispiel 2.3. Wir nehmen an, dass zu diesem System ein EP existiert, der unter anderem `Payment` Events produziert. Diese Events können auf unterschiedlich Weise verwendet werden. Im Bezahlen-Prozess werden sie genutzt, um automatisch darauf zu reagieren, dass die Bezahlung der Ware getätigt wurde (`Geld Eingang` Aktivität). Die `Payment` Events können auch in anderen Prozessen verarbeitet werden. Die Events können anschließend von allen Prozessen und weiteren angeschlossenen Systemen verwendet werden. Die betroffenen Systeme müssen nicht selbst beim Finanzverwaltungssystem die Daten abrufen. Stattdessen werden ihnen die Daten sobald sie anfallen, in Form von Events zugesandt.

Spezifizierung eines EPs

Jeder EP muss eigens für seine Aufgabe implementiert werden. Wir unterscheiden zwischen der Schnittstellendefinition des EPs und seiner konkreten Implementierung. In der Schnittstellendefinition eines EPs werden folgende Attribute definiert:

Identifier(ID) Ein eindeutiger Name zur Unterscheidung von mehreren EPs, z. B. Finanzverwaltung EP.

Annotation Ein optionales Freitextfeld für eine menschenlesbare Beschreibung des EPs, z. B. „Finanzverwaltung EP für Finanzsoftware X“.

EventTypes Eine Liste von *Event Types*, von denen die produzierten Events sein können.

Die konkrete Implementierung ist vom jeweiligen Anwendungsfall abhängig und muss für jeden EP speziell erstellt werden.

4.2.2 Event Consumer

Der *Event Consumer* ist das Gegenstück zum EP. Dieser hat die Aufgabe Events zu konsumieren bzw. final zu verarbeiten. Genau wie EPs ist ein EC nur mit dem Vaterknoten, dem SuperEPA, verbunden und hat keine Kindknoten. Damit ist er ein Blattknoten im EPN. ECs erhalten Events aus dem EPN, können aber keine produzieren. Dadurch

bieten sie die Möglichkeit, Events von unterschiedlichen Prozess-Instanzen oder angeschlossenen Systemen zu empfangen und zu verarbeiten. Ein Beispiel hierfür sind Adapter für (Legacy-) Systeme. Die eingehenden Events können dann beispielsweise auf Funktionsaufrufe in diesen Systemen übertragen werden. Alternativ können auch, sofern vorhanden, entsprechende Web-Services aufgerufen werden. Auf diese Weise kann das BPM-System mit anderen Systemen kommunizieren.

Beispiel 4.2 (EC für ein ERP-Lagerverwaltungssystem)
Betrachten wir hier als Beispiel wieder dem Bestell-Prozess aus Beispiel 2.3. Hier verwaltet das ERP-Lagerverwaltungssytem unter anderem die Kommissionierung von Waren. Ein EC für dieses System hätte nun die Aufgabe, die `Order picking Request` Events zu sammeln. Mit diesen Events kann nun die Kommissionierung im Lager veranlasst werden. Hierzu ruft der EC bei Eingang eines Events die entsprechenden Funktionen im System auf. Der Vorteil hierbei ist, dass die `Ware kommissionieren` Aktivität keine direkte Verbindung zum ERP-System benötigt. Führt das Unternehmen ein neues ERP-System ein, muss lediglich ein neuer Adapter-EC erstellt werden. Der Bestell-Prozess muss in diesem Fall nicht angepasst werden, sofern sich an den verwendeten Events nichts ändert. Zusätzlich können auch mehrere Systeme parallel genutzt werden, z. B. bei mehr als einem Lager. Im Prozess selbst muss nicht entschieden werden, welches System letztendlich die Aktion ausführen soll.

Die weiteren Verwendungsmöglichkeiten von ECs sind, wie bei den EPs, sehr vielfältig. Statt mit Events Funktionsaufrufe auszuführen, können die Events auch zur Analyse verwendet werden. So kann z. B. mitgezählt werden, wie viele Waren kommissioniert wurden. Aus diesen Daten kann bei Bedarf eine Nachbestellung einzelner Artikel veranlasst werden.

Spezifizierung eines ECs

Genau wie die EPs müssen auch die ECs für jeden Einsatzzweck eigens entwickelt werden. Auch hier unterscheiden wir daher zwischen einer Schnittstellendefinition und der konkreten Implementierung. Die konkrete Implementierung ist auch hier wieder vom Anwendungsfall abhängig. In der Schnittstellendefinition sind folgende Attribute definiert:

Identifier(ID) Ein eindeutiger Name zur Unterscheidung von ECs, z. B. Monitor EC.

Annotation Ein optionales Freitextfeld für eine menschenlesbare Beschreibung des EPs.

EventTypes Eine Liste von *Event Types*, welche die konsumierte Events haben können. Diese *Event Types* werden automatisch vom EC abonniert.

Filter Spezifiziert einen oder mehrere Filterausdrücke. Damit können ungewünschte Events herausgefiltert werden, um so die Anzahl der zu verarbeitenden Events zu reduzieren.

4.2.3 Event Processing Agent

Das dritte und letzte EPE ist der *Event Processing Agent*. Diese übernehmen die eigentliche Eventverarbeitung im EPN. Darunter fallen, wie in Abschnitt 3.1.2 bereits diskutiert, die Funktionalitäten Filtering, Transformation und Pattern Detection. Im Gegensatz zu den EPs und ECs können die EPAs Events sowohl empfangen, als auch versenden. Sie können jedoch nicht eigenständig neue Events produzieren. Stattdessen produzieren sie Events nur als Reaktion auf eingehende Events.

EPAs können beliebig viele EPEs als Kindknoten haben. Ein EPA kann somit als Wurzelknoten eines *EPN-Teilbaumes* des gesamten EPNs betrachtet werden. Damit lassen sich mit EPAs eigenständige komplexe Strukturen erstellen. Diese komplexen Strukturen können entweder im laufenden Prozess mit dem MainEPA verbunden werden oder der Wurzelknoten wird mit einem anderen EPA verbunden, um eine noch komplexere Struktur zu erreichen. Damit können mit einem EPA verschieden Funktionen über seine Kindknoten zusammengefasst werden.

Beispiel 4.3 (EPN-Teilbaum zu einem ERP-Lagersystem)
Betrachten wir das ERP-Lagersystem aus dem Bestell-Prozess (siehe Beispiel 2.3). Da diese Komponente Events sowohl konsumiert als auch produziert, werden auch mindestens ein EP und EC benötigt[6]. Damit diese Komponente als Ganzes in das EPN eingefügt werden kann, bekommen EP und EC einen gemeinsamen SuperEPA. Dies ist in Abbildung 4.3 dargestellt. Der EPA hat jeweils einen EP und EC mit dem die Produktion und Konsummierung stellvertretend für das ERP-Lagersystem durchgeführt wird. Dabei wird z. B. ein `Order picking Request` Event vom EC empfangen und die entsprechenden Funktionen im ERP System aufgerufen. Nach Beendigung der Kommissionierung wird vom EP ein `Order picking Completed` Event produziert. Damit gelangt das Event und damit auch die Informationen über die abgeschlossene Kommissionierung, in das EPN. Zusätzlich kann der EPA noch mit weiteren Event Processing Funktionen ausgestattet werden. Ein Beispiel hierfür ist ein Filter, bei dem

[6] Wenn Funktionalitäten getrennt werden sollen, dann können auch durchaus mehrere EPs und ECs eingesetzt werden.

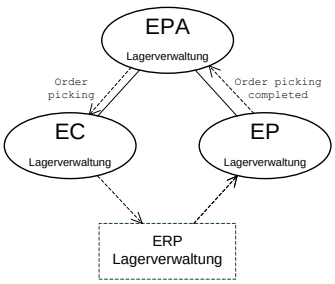

Abbildung 4.3: EPA mit jeweils einem EP und EC zur Kommunikation mit dem ERP-Lagerverwaltungssytem

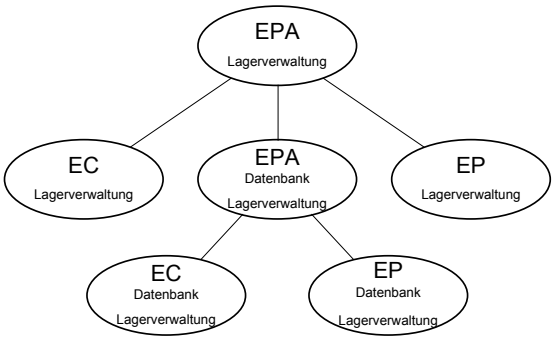

Abbildung 4.4: EPN zur Kommunikation mit einem ERP-Lagerverwaltungssytem mit zusätzlichen EPEs für die Datenbank

`Reserve Request` Events nur durchgelassen werden, wenn sie bestimmte Artikelnummern beinhalten.

Das in Abbildung 4.3 dargestellte EPN kann beliebig um weitere Komponenten erweitert werden. Es können EPEs, die mittels Events eine Verbindung zu einer Datenbank kapseln, hinzugefügt werden, die wiederum mit einem EPA logisch gruppiert werden (vergleiche hierzu Abbildung 4.4).

CEP-Module

Die Grundfunktionalität eines EPAs ist, in der bisher vorgestellten Architektur, das Routing von Events. Allerdings können EPAs auch die eigentlichen Event Processing Aufgaben übernehmen. Um die Event Processing Funktionen möglichst wiederverwendbar zu gestalten, definieren wir sie in Form von CEP-Modulen. Diese enthalten fertige Bausteine für Complex Event Processing (CEP). Wir unterscheiden

dabei zwei verschiedene Arten von CEP-Modulen: `Tranformation`- und `Pattern Detection`-Module.

`Tranformation`-Module orientieren sich an den Transformation Agents (siehe Abschnitt 3.1.2). Sie nehmen Events entgegen, manipulieren deren Inhalt und produzieren daraus neue Events. Möglich sind hier z. B. die Anreicherungen mit Informationen aus der Datenbank oder entfernen von optionalen Payload Attributen. Nähere Informationen über die unterschiedlichen Arten von Transformation sind in [EN10] aufgelistet.

`Pattern Detection`-Module erkennen Muster in einer Reihe von Events. Dies bedeutet, dass aus einer Menge von scheinbar unbedeutenden Events ein *Complex Event* erkannt wird (siehe Beispiel 2.2). Die Konsequenz beim Erkennen eines Musters ist entweder die Produktion eines neuen Events oder die Ausführung einer Aktivität bzw. eines Prozesses. Auf diese Weise kann ein Event-Handler erstellt werden, der ähnlich funktioniert wie in BPEL.

Spezifizierung eines EPAs

Ähnlich wie bei ECs und EPs müssen auch EPAs eigens für ihre Aufgabe spezifiziert werden. Dabei haben sie folgende Attribute:

Identifier(ID) Ein eindeutiger Name zur Unterscheidung von *Event Processing Agents*, z. B. ERP Lagerverwaltung EPA

Annotation Ein optionales Freitextfeld für eine menschenlesbare Beschreibung des EPAs.

Input EventTypes Eine Liste von *Event Types*, welche die konsumierten Events haben können. Diese *Event Types* werden vom EPA abonniert (siehe *Publish/Subscribe*).

Output EventTypes Eine Liste von *Event Types*, welche die vom EPA produzierten Events haben können.

Filter Spezifiziert einen oder mehrere Filterausdrücke. Damit wird die Anzahl der eingehenden Events reduziert. Es kann zu jedem verbunden EP oder EPA einzeln ein Filterausdruck spezifiziert werden.

SubEPE Eine Liste der mit dem EPA verbundenen SubEPEs. Siehe hierzu z. B. den EP und EC von Abbildung 4.3.

CEP Module Eine Liste von CEP-Modulen, mit denen der EPA ausgestattet ist.

4.2.4 EPEs für Prozess-Instanzen

In Abschnitt 4.1 sind wir davon ausgegangen, dass zu jeder Prozess-Instanz ein EPA zugewiesen ist. Dieser soll die Event Processing Aufgaben für die jeweilige Prozess-Instanz übernehmen. Allerdings können innerhalb eines Prozesses sowohl Events produziert, als auch konsumiert werden. Daher ist es besser, diese Aufgaben voneinander zu trennen und von separaten EPs und ECs durchführen zu lassen.

Zu diesem Zweck definieren wir drei spezialisierte EPEs für Prozess-Instanz: *Instance Event Producer* (IEP), *Instance Event Consumer* (IEC) und *Instance Event Processing Agent* (IEPA). Diese drei EPEs werden beim Start einer Prozess-Instanz jeweils instanziiert. IEP und IEC bekommen den IEPA als gemeinsamen SuperEPA. Dadurch entsteht ein EPN, wie es in Abbildung 4.5 dargestellt ist. Diese drei EPE sind an die Laufzeit ihrer Prozess-Instanz gebunden. Daher werden nach Beendigung ihrer Prozess-Instanz heruntergefahren und aus dem Gesamt-EPN entfernt. Wir betrachten nun die einzelnen Aufgaben der spezialisierten EPEs näher.

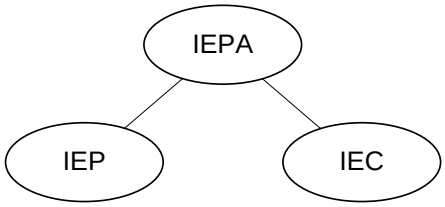

Abbildung 4.5: EPN zu einer einzelnen Prozess-Instanz

Instance Event Producer

Ein spezieller EP ist der *Instance Event Producer* (IEP). Der IEP hat die Aufgabe, die Ereignisse, welche von elementaren Aktivitäten ausgelöst werden, in Events zu

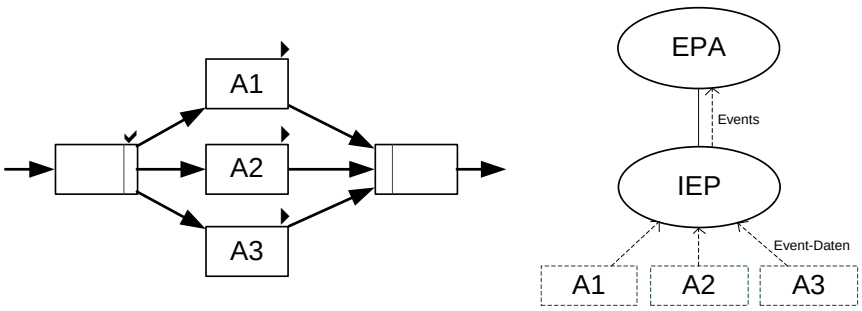

Abbildung 4.6: Beispiel für einen IEP während ein Prozess ausgeführt wird.

verpacken und in das EPN einzubringen. Hiermit ist der IEP der Stellvertreter der elementaren Aktivitäten im EPN.

Beispiel 4.4 (Instance Event Producer)
Betrachten wir Abbildung 4.6. Links ist ein Ausschnitt aus einem Prozess abgebildet. In diesem befinden sich aktuell drei Aktivitäten parallel in Ausführung. Rechts ist ein Ausschnitt aus dem EPN dargestellt. Die einzelnen Aktivitäten (`A1` - `A3`) liefern Daten an den IEP. Daraus produziert der IEP Events und leitet sie an seinen SuperEPA weiter. Von dort aus gelangen sie in das gesamte EPN.

Beim Ablauf eines Geschäftsprozesses entstehen diverse Statusänderungen. Darunter fallen z. B. Start/Ende einer Prozess-Instanz oder Aktivität oder die Änderung eines Datenelementes. Auf Grundlage dieser Ereignisse können entsprechende Events produziert werden. Diese Events können an anderen Stellen weiterverarbeitet und genutzt werden, z. B. in einer Monitoring-Anwendung (vergleiche hierzu Abschnitt 2.2.1). Allerdings müssen die entsprechenden Events an einer definierten Stelle produziert und in das EPN eingebracht werden. Der beste Ausgangspunkt für diese Events ist der IEP der entsprechenden Prozess-Instanz. Damit der IEP diese Events produzieren kann, informiert das BPM-System den IEP über die jeweilige Statusänderung und übergibt die entsprechenden Daten.

Instance Event Consumer

Eine Spezialform des ECs ist der *Instance Event Consumer* (IEC). Elementare Aktivitäten in Prozessen können mittels des IECs Events empfangen. Der IEC verteilt eingehende Events an laufende Aktivitäten. Beim Start einer Aktivität meldet diese sich beim IEC

an. Dabei teilt sie mit, welche *Event Types* sie benötigt. Empfängt nun der IEC ein entsprechendes Event, leitet er dieses an die entsprechenden Aktivitäten weiter.

Beispiel 4.5 (Instance Event Consumer)
Betrachten wir hierzu als Beispiel die `Geld Eingang` Aktivität aus dem Bezahlen-Prozess (siehe Beispiel 2.3). Beim Starten meldet sich die Aktivität beim IEC an und abonniert so die `Payment` Events. Erhält nun der IEC ein `Payment` Event, leitet dieser das Event an die Aktivitätsinstanz weiter. Die Aktivitätsinstanz kann nun entsprechend auf das empfangene Event reagieren.

Instance Event Processing Agent

Genau wie bei EP und EC gibt es auch beim EPA eine spezielle Form, den *Instance Event Processing Agent* (IEPA). Dieser hat die Aufgabe das Event Processing einer Prozess-Instanz zu koordinieren. Dazu hat er als SubEPEs jeweils einem IEP und IEC und eine beliebige Anzahl von weiteren IEPAs, abhängig von der Anzahl der gerade aktiven Subprozesse (siehe hierzu Abschnitt 4.1). Zusätzlich kann ein IEPA auch mit mehreren CEP-Modulen und Filter ausgestattet sein, mit denen eine Aktivität übergreifende Eventverarbeitung möglich ist. Ein Beispiel hierfür ist ein Filter, der nur für eine bestimmten Bestellvorgang Events durchlässt.

4.3 Events

Events sind der zentrale Gegenstand einer Event Processing Architektur. In diesem Abschnitt diskutieren wir nun einen konkreten Aufbau eines Events.

Ein Event kann auf unterschiedliche Weise definiert werden [EN10]. Welche genauen Elemente ein Event bzw. eine Event-Instanz beinhaltet, kann sich durch das verwendete System und Anforderungen sehr stark unterscheiden. Wir definieren im Folgenden einen konkreten Aufbau für ein Event, um eine einheitliche Grundlage zu schaffen. Dieser Aufbau ist an die Event-Spezifikation in [EN10] angelehnt. Wir nutzen als Grundlage, den in Abschnitt 2.2.2, beschriebenen Event-Aufbau und erweitern diesen. Ein Event besteht aus zwei Teilen, dem Header und den Payload. Der Header enthält Metainformationen über ein Event. Er beschreibt unter anderem, von welchem Typ das Event ist und wann es aufgetreten ist. Der Payload enthält *Event Type* spezifische Daten über den eigentlichen Vorfall.

4.3.1 Header Attribute

In der hier vorgestellten Architektur enthält ein Event sechs Header Attribute. Diese enthalten Metainformationen und geben allgemeine Auskunft über ein Event. Hier diskutieren wir nun die einzelnen Attribute. Wir fordern im Folgenden, dass jede Event-Instanz alle diese Attribute enthält. Damit ist ein einheitlicher Aufbau für alle Events gewährleistet.

Event Identity Das *Event Identity* Attribut (auch kurz *EventID* genannt) enthält eine systemgenerierte und eindeutige ID zu einem Event. Mit diesem Attribut lassen sich zwei oder mehrere Event-Instanzen eindeutig unterscheiden, selbst wenn diese den gleichen Inhalt haben.

Event Type Identifier Der *Event Type Identifier* beschreibt eindeutig den *Event Type* des Events. Dadurch lässt sich das Event in eine Kategorie einordnen. Zusätzlich ist darüber auch definiert, welche Payload Attribute das Event haben muss.

Occurence Time Mit dem *Occurence Time* Attribute wird angegeben, zu welchem Zeitpunkt das Ereignis aufgetreten ist.

Detection Time Das *Detection Time* Attribut gibt Auskunft darüber, wann das dem Event zugrunde liegende Ereignis im Event Processing System bemerkt wurde. Damit ist die genaue Zeit der Generierung des Events gemeint.

Zu beachten ist, dass sich die *Occurence* und *Detection Time* unterscheiden können. Wenn das Event in einem externen System aufgetreten ist, wird für die *Occurence Time* die Systemzeit dieses Systems gemessen. Da die Informationen noch an das Event Processing System übertragen werden müssen, kann eine Zeitdifferenz zur *Detection Time* entstehen. Zudem ist nicht immer gesichert, dass eintreffende Events in der richtigen Reihenfolge verarbeitet werden. Dies kann insbesondere aufgrund von Multithreading geschehen. Häufig ist es aber wichtig zu wissen, in welcher Reihenfolge die Ereignisse tatsächlich aufgetreten sind. Ein weiterer Grund für die Unterscheidung der beiden Zeiten ist, dass die Systemzeit des Event Processing Systems und einer externen Komponente, möglicherweise nicht gleich sind.

Event Annotation Das *Event Annotation* Attribut gibt die Möglichkeit eines freien Textes über das Event. Mit diesem Attribut können Angaben in menschenlesbarer Form gemacht werden. Diese Angabe kann für Dokumentationszwecke genutzt werden. Falls keine Angabe gewünscht ist, kann dieses Feld auch einen leeren Text enthalten.

Event Source Mit dem *Event Source* Attribut wird angegeben, welcher EPE dieses Event erzeugt hat. Es kann sich hier sowohl um einen Event Producer, als auch um einen Event Processing Agent handeln. Events können über mehrere EPAs zu ihren Zielen geleitet werden. Mit dem *Event Source* Attribut kann später ermittelt werden, welcher EP bzw. EPA ein Event produziert hat.

4.3.2 Payload Attribute

Die Header Attribute enthalten Metainformationen über ein Event. Diese haben eine feste Struktur und sind unabhängig vom *Event Type*. Demgegenüber enthalten die Payload Attribute die eigentlichen Daten eines Events und beschreiben das zugrunde liegende Ereignis.

Die Payload Attribute werden bei jedem *Event Type* individuell zusammengestellt. Dazu braucht jedes Attribut einen, für diesen *Event Type*, eindeutigen Namen und einen Datentyp. Dabei reichen primitive Datentypen meistens nicht aus. Für eine bessere Wiederverwendbarkeit ist es daher notwendig, aus verschiedenen Datentypen zusammengesetzte, komplexe Datentypen verwenden zu können. In vielen Fällen werden eine unbekannte Anzahl von Elementen eines Typs benötigt, z. B. enthält eine Bestellung eine unbegrenzte Anzahl von Artikeln. Daher kann es sich auch um eine Liste eines Datentyps handeln. Da manche Events sich auf andere Events beziehen, werden auch Referenzen auf Events benötigt.

Neben Name und Datentyp muss bei jedem Payload Attribut auch angegeben sein, ob dieses Attribut obligat oder optional ist. Dadurch lässt sich zur Laufzeit sicherstellen, dass alle Events korrekt produziert werden. Dies ist insbesondere wichtig, da bei der Verarbeitung von Events häufig bestimmte Daten benötigt werden.

4.3.3 Payload Beispiele

Da wir nun den genauen Aufbau und Inhalt eines Events betrachtet haben, soll dies nun durch zwei Beispiele veranschaulicht werden. Dazu werden im Folgenden die Payload-Attribute von zwei *Event Types* behandelt. Zu beachten ist, dass diese Beispiele nur eine von mehreren Möglichkeiten darstellen. Je nach Bedarf können auch andere Attribute oder andere Datentypen genutzt werden.

Beispiel 4.6 (Position)
Den `Position` *Event Type* haben wir zuvor bereits in Beispiel 2.1 betrachtet. Hier soll dieser noch einmal, mit mehr Details, als Beispiel dienen. Mit einem `Position` Event kann ein Lkw oder anderes Fahrzeug eine neue Positionsangabe mitteilen. Daraus lässt zum einen die gefahrene Route des Fahrzeugs ermitteln, zum anderen kann mithilfe der letzten Positionsangabe bestimmt werden, wo das Fahrzeug gerade ist. Die Payload Attribute sind in Tabelle 4.1 zu sehen.

Attribut Name	Datentyp	Obligat
DriverID	Integer	obligat
VehicleID	Integer	obligat
GPS coordinates	String	obligat
VehicleType	String	optional

Tabelle 4.1: Payload Attribute für `Position` Events

Bei diesem *Event Type* ist `VehicleType` das einzige Attribut, welches optional ist. Da sich der Typ des Fahrzeuges normalerweise auch über die `VehicleID` ermitteln lässt, kennzeichnen wir dieses Attribut als optional.

Beispiel 4.7 (Reserve Request)
Events vom Typ `Reserve Request` kennen wir bereits aus dem Prozess von Beispiel 2.3. Diese Events werden von einem ERP-System empfangen und verarbeitet, um die entsprechenden Artikel zu reservieren. Der Inhalt des Payloads ist in Tabelle 4.2 zu sehen.

Attribut Name	Datentyp	Obligat
OrderID	Integer	obligat
Items	List of Item	obligat

Tabelle 4.2: Payload Attribute für `Reserve Request` Events

Als Erstes sehen wir hier das Attribut `OrderID`, mit dem sich ein `Reserve` Event eindeutig zu einer Bestellung zuordnen lässt. Eine Besonderheit bei diesem Payload ist, dass dieser eine

Liste von einem zusammengesetzten Datentypen enthält. In diesem Fall handelt es sich um eine Liste vom Typ *Item*. Der Aufbau dieses Datentyps ist in Tabelle 4.3 zu sehen. Jedes Element vom Typ *Item* enthält die Ware (`ArticleID`) und die jeweilige Anzahl (`Quantity`).

Attribut Name	Datentyp	Benötigt
ArticleID	Integer	benötigt
Quantity	Integer	benötigt

Tabelle 4.3: Attribute für den zusammengesetzten Datentyp `Item`

4.3.4 Prozessbezogene Events

Da die hier behandelte Event Processing Architektur hauptsächlich für BPM-System ausgelegt ist, werden wir hier noch einige Ergänzungen machen.

Ein Event hat im Header die Information über den Event Producer gespeichert. Ist die Quelle von einem Event ein Prozess, so reicht die Kenntnis über den *Event Producer* häufig nicht aus. Es kann lediglich ermittelt werden, in welcher Prozess-Instanz das Event produziert wurde. Die Information über die genaue Aktivität fehlt jedoch. Dies kann in einigen Fällen wichtig werden, da mehrere Aktivitäten in einem Prozess Events vom gleichem Typ produzieren können.

Um bei einem Event, dass innerhalb eines Prozesses produziert wurde, den genauen Ursprung zu ermitteln, benötigen wir zusätzliche Informationen. Diese müssen aber unabhängig vom *Event Type* sein, da Prozesse die gleichen *Event Types* verwenden, wie das restliche Event Processing System. Daher fügen wir Events, die innerhalb eines Prozesses produziert wurden, zusätzliche Payload Attribute hinzu. Diese bezieht sich in diesem Fall auf die AristaFlow BPM Suite [Ari, DRRM+09, RDJ+08]. Bei einem anderen BPM-System müssen die Felder je nach Bedarf angepasst werden.

ProzessID enthält eine eindeutige Identifikationsnummer der jeweiligen Prozessvorlage. Dies kann auch der Prozessname sein, sofern dieser eindeutig ist.

InstanceID enthält die eindeutige Identifikationsnummer der jeweiligen Prozess-Instanz.

NodeID enthält die eindeutige Identifikationsnummer des Knoten der Prozessvorlage, der das Event ausgelöst hat.

NodeIteration gibt an, in der wievielten Ausführung des Knoten bzw. Aktivität, das Event produziert wurde. Durch Schleifen kann ein Knoten mehrfach innerhalb einer Prozess-Instanz ausgeführt werden.

4.4 Dynamisches Anpassung im Event Processing Network

In diesem Abschnitt behandeln wir das dynamische Verhalten eines EPNs. Der Aufbau des EPNs ist immer abhängig, von den gerade im BPM-System laufenden Prozess-Instanzen. Hinzu kommen die EPEs von angeschlossenen externen Systemen. Wir diskutieren nun, wie sich das EPN zur Laufzeit verändert. Dabei beginnen wir mit der Auswirkung beim Starten und Beenden von Prozess-Instanzen und betrachten darauf folgend die Verbindung von externen Systemen mit dem EPN.

4.4.1 Verhalten beim Start eines Prozesses

Wir behandeln nun die Aufgaben des Event Processing Systems beim Starten eines Prozesses. Dabei wird zwischen Top-Level-Prozessen und Subprozessen unterschieden. Der Einfachheit halber betrachten wir im Folgenden nur den EPN-Teilbaum einer Prozess-Instanz, samt der Instanzen ihrer Subprozesse.

Verhalten bei Start eines Top-Level-Prozesses

Beim Starten eines Top-Level-Prozesses erstellt das Event Processing System jeweils ein IEP, IEC und IEPA und verbindet diese miteinander (siehe hierzu Abschnitt 4.2.4). Der IEPA eines Top-Level-Prozesses wird mit dem zentralen MainEPA verbunden. Nach Starten eines beliebigen Top-Level-Prozesses entsteht hiermit zunächst ein EPN, wie es in Abbildung 4.7 dargestellt ist.

Verhalten beim Start eines Subprozesses

Beim Starten eines Subprozesses wird ein vergleichbarer EPN-Teilbaum wie in Abbildung 4.7 erstellt. Allerdings wird dieser nicht direkt mit dem MainEPA verbunden. Stattdessen wird der IEPA eines Subprozesses mit dem IEPA des Vater-Prozesses verbunden. Dadurch entsteht ein neuer EPN-Teilbaum. Abbildung 4.8 zeigt den EPN-Teilbaum eines Prozesses mit einem gestarteten Subprozess.

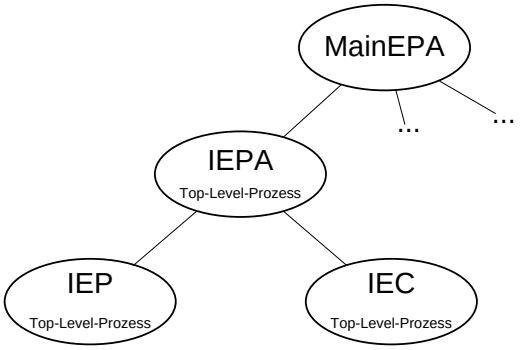

Abbildung 4.7: EPN mit einer laufenden Prozess-Instanz ohne Subprozesse

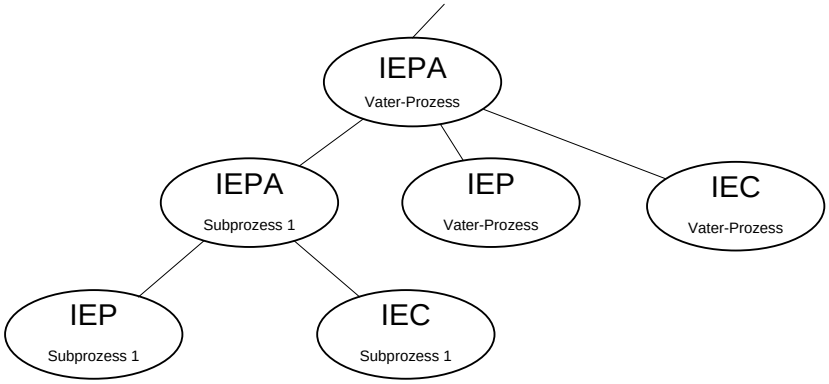

Abbildung 4.8: EPN-Teilbaum zu einem Prozess mit einem Subprozess

Ein Top-Level-Prozess kann mehrere Subprozesse starten. Diese können auch parallel ausgeführt werden. Zudem können auch Subprozesse weitere Subprozesse starten. Für jeden einzelnen Subprozess wird jeweils ein IEPA, IEP und IEC erstellt. IEP und IEC werden jeweils mit dem IEPA verbunden. Letzterer wird mit dem IEPA des jeweiligen Vater-Prozesses verbunden. Dadurch entsteht ein EPN-Baum, dessen Tiefe von der Hierarchietiefe des Top-Level-Prozesses abhängig ist.

4.4.2 Verhalten beim Beenden eines Prozesses

Jedes Mal, wenn ein Prozess beendet wird, muss auch der jeweilige IEPA beendet und aus dem EPN entfernt werden. Dies gilt auch für die mit den IEPA verbundenen IEP und IEC. Durch das ständige Starten und Beenden von Prozess-Instanzen variiert das

EPN zur Laufzeit des BPM-Systemes. Dies ist abhängig von den aktuell laufenden Prozessen.

4.4.3 Beispiel zum Verhalten eines EPNs während der Laufzeit einer Prozess-Instanz

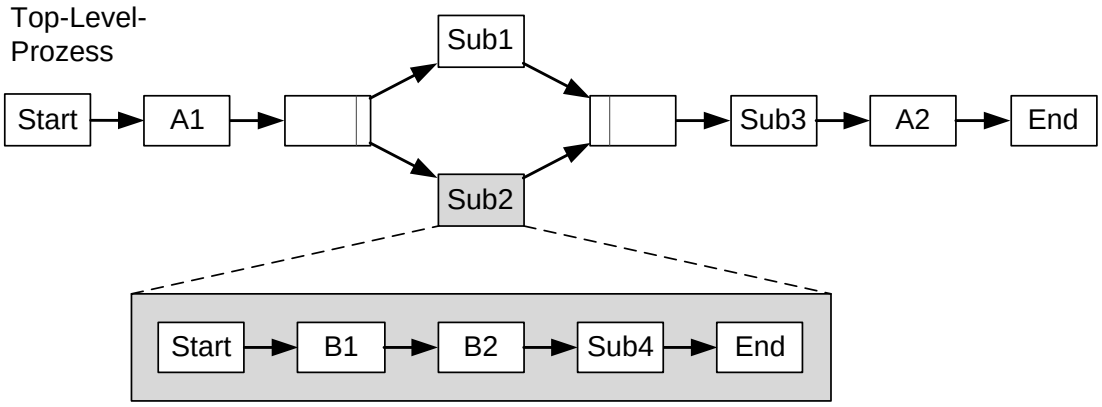

Abbildung 4.9: Prozess-Beispiel für dynamische EPN-Veränderung

Wir wollen das Verhalten eines EPNs während der Laufzeit eines Prozesses mit einem Beispiel vertiefen. Betrachten wir dazu Abbildung 4.9. Dort ist ein Top-Level-Prozess mit insgesamt drei Subprozessen dargestellt. Der Top-Level-Prozess beginnt zunächst mit Aktivität `A1`. Daraufhin erfolgt ein AND-Split und die Subprozesse 1 und 2 werden parallel ausgeführt. Sind Subprozess 1 und 2 abgeschlossen kann Subprozess 3 gestartet werden. Anschließend wird als Letztes die Aktivität `A2` ausgeführt, bevor der Prozess abgeschlossen ist. Subprozess 1 und 3 werden wir nicht im Detail behandeln. Wir gehen lediglich davon aus, dass diese keine weiteren Subprozesse besitzen. Subprozess 2 ist im unteren Teil von Abbildung 4.9 dargestellt. Nach dem Start von Subprozess 2 werden die Aktivitäten `B1` und `B2` ausgeführt. Danach wird Subprozess 4 ausgeführt. Wie bei Subprozess 1 und 3 gehen wir davon aus, dass Subprozess 4 keine weiteren Subprozesse hat.

Wir betrachten nun den Zustand des EPN-Teilbaumes, während der Laufzeit des in Abbildung 4.9 dargestellten Prozesses. Die Wurzel des EPN-Teilbaumes ist immer ein direkter Kindknoten vom MainEPA, der in den folgenden Abbildungen ausgeblendet ist. Direkt nach dem Start des Top-Level-Prozesses und während der Laufzeit von

Aktivität `A1`, sieht der EPN-Teilbaum aus, wie in Abbildung 4.10 dargestellt. Der EPN-Teilbaum hat zunächst einen IEPA mit jeweils einem angeschlossenen IEP und IEC.

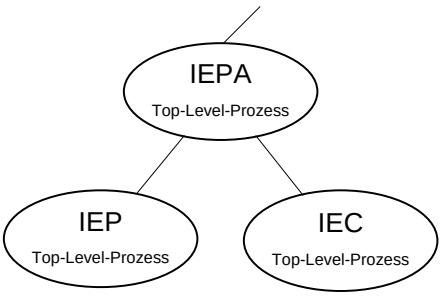

Abbildung 4.10: EPN-Teilbaum direkt nach dem Start des Top-Level-Prozesses

Sobald der AND-Split erreicht ist, werden sowohl Subprozess 1 als auch Subprozess 2 gestartet. Betrachten wir zunächst den Zustand des EPN-Teilbaumes, während in Subprozess 2 die Aktivität `B1` oder `B2` ausgeführt wird. In diesem Fall sieht der EPN-Teilbaum aus, wie in Abbildung 4.11 dargestellt. Dort ist ein IEPA für den Top-Level-Prozess zu finden. Dieser ist mit den IEPAs für Subprozess 1 und 2 verbunden. Zusätzlich hat jeder IEPA einen IEP und IEC.

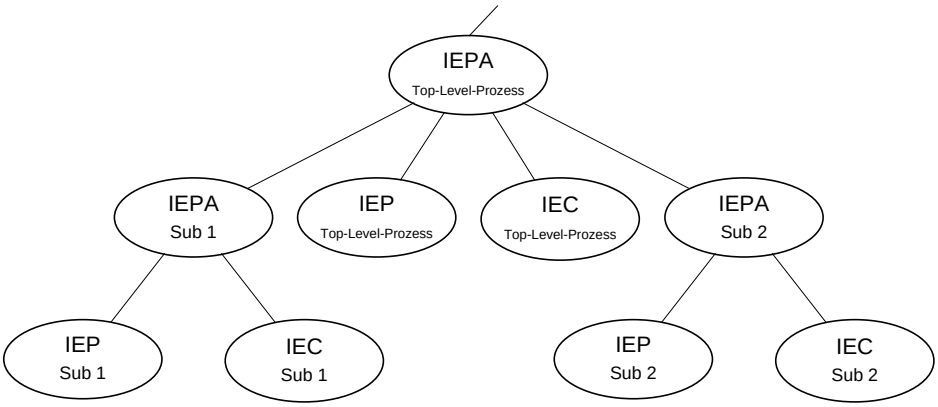

Abbildung 4.11: EPN-Teilbaum während Sub 1 und Sub 2 parallel ausgeführt werden

Gehen wir davon aus, dass Subprozess 1 immer noch nicht beendet wurde. Nach dem Start von Subprozess 4, wird ein weiterer IEPA (und mit ihm ein IEP und IEC) erzeugt. Dieser erhält den IEPA von Subprozess 2 als Vaterknoten bzw. SuperEPA. Dadurch ergibt sich der EPN-Teilbaum, welches Abbildung 4.12 zeigt.

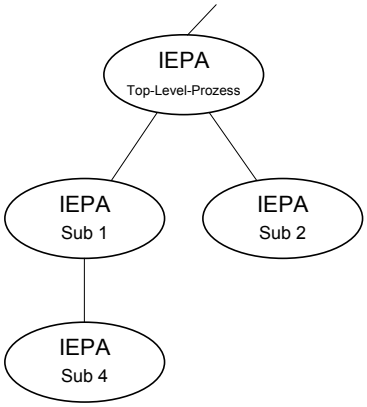

Abbildung 4.12: EPN-Teilbaum nach Start von Sub 4 (IEPs und IECs sind ausgeblendet)

Nach Abschluss von Subprozess 1 und 2 wird Subprozess 3 gestartet. Die IEPAs für Subprozess 1 und 2 wurden in der Zwischenzeit entfernt. Daraus folgt ein EPN-Teilbaum mit 2 IEPAs wie in Abbildung 4.13.

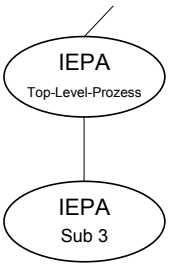

Abbildung 4.13: EPN-Teilbaum nach Start von Sub 3 (IEPs und IECs sind ausgeblendet)

Nach Beendigung von Subprozess 3 wird auch dessen IEPA aus dem EPN-Teilbaum entfernt. Dadurch entsteht der gleiche EPN-Teilbaum wie zu Beginn der Prozess-Instanz (siehe Abbildung 4.10). Der EPN-Teilbaum behält diesen Aufbau, während Aktivität `A2` ausgeführt wird. Nach Beendigung von Aktivität `A2` ist der Prozess abgeschlossen und der EPN-Teilbaum, für diese Prozess-Instanz, kann entfernt werden.

Dieses Beispiel demonstriert, wie sich ein einzelner EPN-Teilbaum während der Laufzeit einer Prozess-Instanz verändert. Üblicherweise verwaltet ein BPM-System

mehrere Prozess-Instanzen von unterschiedlichen Prozessvorlagen. Jede einzelne Prozess-Instanz besitzt dabei im Event Processing System einen eigenen EPN-Teilbaum.

4.4.4 Event Processing Network während der Laufzeit

Wir haben im vorangegangenen Abschnitt das Verhalten eines einzelnen EPN-Teilbaumes während der Laufzeit einer Prozess-Instanz diskutiert. Im Folgenden behandeln wir die Auswirkungen auf den gesamten EPN-Baum. Die hier vorgestellte EPN-Architektur hat als Wurzelknoten einen EPA, der MainEPA genannt wird. An den MainEPA werden alle EPN-Teilbäume der laufenden Prozess-Instanzen angehängt. Dabei wird das jeweilige Wurzelelement des EPN-Teilbaumes mit dem MainEPA verbunden. Auf diese Weise kann der MainEPA mit den EPAs der Prozess-Instanzen Events austauschen. Allerdings werden nicht nur die EPN-Teilbäume von Prozess-Instanzen mit dem MainEPA verbunden. Es können auch andere Systeme mithilfe von EPAs, ECs, und EPs mit dem MainEPA verbunden werden. Diese Komponenten können dynamisch zu Laufzeit hinzugefügt und entfernt werden. Auf diesem Weg können unter anderem externe Systeme und sonstige Komponenten mit dem EPN und damit auch mit dem BPM-System, kommunizieren.

Betrachten wir hierzu wieder den Bestell-Prozess aus Beispiel 2.3. Dort existiert ein System für die Finanzverwaltung, welches per `Payment` Events mit dem Bestell-Prozess kommuniziert. In diesem Fall wird ein EP für die Finanzverwaltung erstellt, welcher an den MainEPA angeschlossen wird. Dieser EP hat nun die Aufgabe, aus den Daten der eingehenden Zahlungen `Payment` Events zu produzieren. Daraufhin gelangen die Events von diesem EP aus in das EPN und werden an alle Abonnenten von `Payment` Events verteilt. Mögliche Abonnenten sind z. B. alle IECs der laufenden Versand-Prozess-Instanzen. Allerdings können auch andere angeschlossene Systeme den `Payment` *Event Type* abonniert haben.

Abbildung 4.14 zeigt einen möglichen Zustand eines solchen EPNs. Hier sind zwei Prozess-Instanzen des Bestell-Prozesses gestartet. Diese befinden sich in unterschiedlichen Ausführungszuständen. Beim links dargestellten EPN-Teilbaum befinden sich Versand- und Bezahlen-Prozess parallel in der Ausführung. Beim weiter rechts dargestellten EPN-Teilbaum ist der Bezahlen-Prozess entweder noch nicht gestartet oder bereits abgeschlossen. Deswegen existiert hier kein IEPA für den Bezahlen-Prozess. Zusätzlich ist am MainEPA ein EP für die Finanzverwaltung angeschlossen. Dieser

produziert `Payment` Events und versorgt so die IEPAs für die gerade laufenden Bezahlen-Prozesse. Als Letztes hängt am MainEPA ein EPA für die ERP-Lagerverwaltung. Diese verteilt die Aufgaben für ein- und ausgehende Events auf jeweils einen EP und EC.

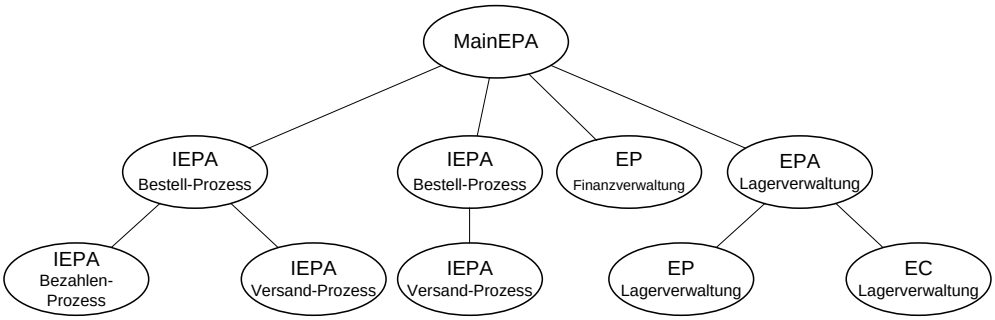

Abbildung 4.14: EPN mit zwei Laufenden Prozess-Instanzen des Bestell-Prozesses und angeschlossener Finanz- und Lagerverwaltung (IEPs und IECs sind ausgeblendet)

4.4.5 Zusammenfassung

Wir haben in diesem Abschnitt das Laufzeitverhalten eines EPN-Baumes für ein BPM-System diskutiert. Der EPN-Baum hängt einerseits von den aktuell laufenden Prozess-Instanzen des BPM-Systems ab. Anderseits kommen noch weitere EPAs, EPs und ECs von anderen Systemen hinzu, die mit dem BPM-System verbunden sind. Diese Systeme können jederzeit mit dem EPN verbunden und wieder abgekoppelt werden. Dadurch erreichen wir eine hohe Flexibilität beim Anbinden externer Systeme.

4.5 Publish/Subscribe und Routing

Wir haben bereits in Abschnitt 2.4 diskutiert, dass die Kommunikation per Events eine entscheidende Anforderung an den hier entwickelten Event-Mechanismus darstellt. Dabei ist sowohl die Kommunikation zwischen den Prozessen als auch mit externen Systemen von Bedeutung. Dazu wird ein Eventverteilungsmechanismus benötigt.

Bisher haben wir in diesem Kapitel nur betrachtet, wie der EPN-Baum aufgebaut ist und wie er sich zur Laufzeit verändert. Dabei haben wir zunächst ausgeblendet,

wie die Events von einem produzierenden Knoten, an alle interessierten Knoten verteilt wird. Dazu entwickeln wir in diesem Abschnitt einen Mechanismus, der dieses ermöglicht.

Wir haben bereits in Abschnitt 3.2.2 das Routing in einem EPN diskutiert. Das dort beschriebene Verfahren passen wir nun an die hier verwendete EPN-Architektur an. Ziel ist es, ein automatisches Verfahren zu beschreiben, damit ein Produzent (*Publisher*) allen interessierten Konsumenten mitteilen kann, welche Events von ihm produziert werden. Auf der anderen Seite muss ein Konsument (*Subcriber*) den Produzenten mitteilen können, welche Events er abonnieren möchten. Dabei ist auch entscheidend, dass ein später hinzukommender Konsument allen bisherigen Produzenten sein Interesse mitteilen kann.

Wir gehen zunächst auf das Grundprinzip des hier verwendeten *Publish/Subscribe*-Mechanismus ein (Abschnitt 4.5.1). Daraufhin gehen wir auf die eigentlichen Algorithmen ein (Abschnitt 4.5.2). Die *Publish/Subscribe*-Algorithmen werden dann mit einem ausführlicheren Beispiel veranschaulicht (Abschnitt 4.5.3). Als Letztes diskutieren wir, wie das eigentliche Routing im EPN-Baum durchgeführt wird (Abschnitt 4.5.4).

4.5.1 Grundprinzip

Der hier vorgestellte *Publish/Subscribe*-Mechanismus nutzt die Baumeigenschaft der vorgestellten EPN-Architektur aus. Wir betrachten dazu jeden EPA als die Wurzel eines EPN-Teilbaumes. Da EPs und ECs immer nur Blattknoten sind, blenden wir diese zunächst aus. Jeder EPA benötigt die Informationen darüber, welche Typen die Events haben, die sein gesamter EPN-Teilbaum produziert und konsumiert. Dazu informiert jeder EPE im EPN seinen jeweiligen SuperEPA darüber, was für Events er produziert und konsumiert. Diese Informationen werden sukzessive von jedem EPA an seinen jeweiligen SuperEPA weitergegeben, bis der MainEPA erreicht ist. Dadurch erhält ein EPA nicht nur die Informationen über seine direkten Kindknoten, sondern auch über seinen gesamten EPN-Teilbaum. Ein EPA kann dadurch die *Event Types*, welcher er benötigt, gezielt bei seinen Kindknoten abonnieren. Damit hat ein EPA auch die Information darüber, welche *Event Types* sein SuperEPA bzw. das restliche EPN konsumiert. Durch die Informationen, welche *Event Types* die Kindknoten bzw. der SuperEPA abonniert haben, kann der EPA eintreffende Events gezielt an die Interessenten weiterleiten.

Ein wichtiger Aspekt beim *Publish/Subscribe* ist, dass der Vorgang dynamisch zur Laufzeit erfolgt. EPEs können dem EPN hinzugefügt und wieder entfernt werden. Dadurch entstehen neue Produzenten und Konsumenten, die ihre möglichen *Event Types* dem EPN mitteilen müssen. Des Weiteren kann sich die Liste, der produzierten bzw. konsumierten Events, zur Laufzeit eines EPEs ändern. Dies passiert insbesondere bei IEPs und IECs sehr häufig, da die möglichen *Event Types* von den jeweils laufenden Aktivitäten abhängig sind. Daher muss es auch möglich sein, dass ein EPE auch mitteilen kann, dass er bestimmte *Event Types* nicht mehr produziert bzw. konsumiert. Wir betrachten nun ein Beispiel, welches das Grundprinzip des *Publish/Subscribe*-Mechanismus darstellt.

Beispiel 4.8 (Vereinfachtes Publish/Subscribe)
Betrachten wir als Beispiel das EPN aus Abbildung 4.15. EPA 2, 3 und 4 produzieren Events von jeweils einem anderen Typ (`x`, `y` und `z`). EPA 3 konsumiert Events vom Typ `z`. EPA 3 und 4 teilen ihrem gemeinsamen SuperEPA (EPA 1) jeweils mit, welche Events sie produzieren und konsumieren. EPA 1 gibt diese Informationen gesammelt an EPA 0 weiter. Obwohl EPA 1 selber keine Events produziert oder konsumiert, erscheint EPA 1 als Produzent von Events der Typen `x` und `y` und als Konsument von Events vom Typ `z`. Analog dazu ist EPA 0, durch die EPAs 1 und 2, Produzent von `x`, `y` und `z` Events und Konsument von `z` Events. Nun hat EPA 0 die Aufgabe die `z` Events bei EPA 2 zu abonnieren, damit diese an EPA 1 weitergeleitet werden können. Bei eingehenden `z` Events leitet EPA 1 diese dann an EPA 3 weiter, da dieser der einzige Konsument aus Sicht von EPA 1 ist.

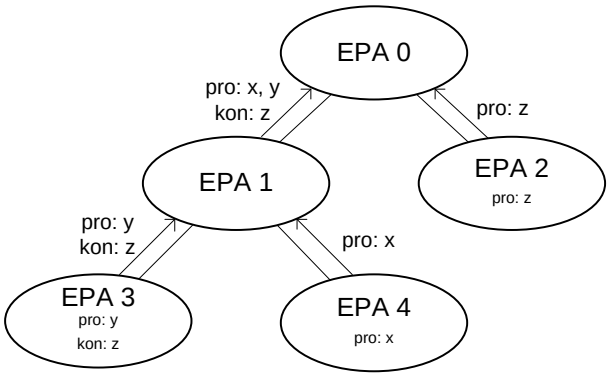

Abbildung 4.15: Beispiel EPN für Publish/Subscribe

4.5.2 Publish/Subscribe Algorithmen

Wir stellen nun die verschiedenen Teile eines *Publish/Subscribe*-Algorithmus vor, welcher auf die hier vorgestellte EPN-Architektur ausgelegt ist. Dieser besteht aus vier Operationen: `Publish`, `Unpublish`, `Subscribe` und `Unsubscribe`. Mit `(Un)Publish` kündigt ein EPA oder EP an, dass dieser Events eines bestimmten Typs liefern kann, bzw. nicht mehr liefert. Mit `(Un)Subscribe` kündigt ein EPA oder EC an, dass dieser Events eines bestimmten Typs konsumiert, bzw. nicht mehr konsumiert. Jeder EPA muss daher alle vier Operationen unterstützen. Ein EP muss nur die `(Un)Subscribe` Operationen unterstützen. Dieser hält lediglich eine Liste, der abonnierten *Event Types* seines SuperEPAs. Dagegen unterstützt ein EC keine der vier Operationen. ECs können lediglich die `(Un)Subscribe` Operationen bei ihrem SuperEPA aufrufen.

Wir stellen nun die einzelnen Operationen bei einem EPA vor. Die Algorithmen sind vereinfacht und ignorieren Konflikte, die durch nebenläufige Bearbeitung ausgelöst werden können. Wir setzten bei den Algorithmen voraus, dass jeder EPA folgende Variablen besitzt:

- `ETpublish`: Eine Tabelle, in der verzeichnet ist, welcher Kindknoten (EP oder EPA) welche *Event Types* produziert.
- `ETsubscribe`: Eine Tabelle, in der verzeichnet ist, welcher Kindknoten (EC oder EPA) welche *Event Types* konsumiert.
- `ETsuperEPA`: Eine Liste von *Event Types*, die der SuperEPA abonniert hat.
- `superEPA`: Verweis auf den jeweiligen SuperEPA.
- `producingET`: Die Liste der *Event Types*, welche der EPA selber produziert.
- `consumingET`: Die Liste der *Event Types*, welche der EPA selber konsumiert.

Die `Publish` Operation (vergleiche Algorithmus 1) ermöglicht es einen Produzenten anzukündigen, dass dieser einen neuen *Event Type* anbieten kann. Diese Operation darf nur an den SuperEPA gerichtet sein. Sollte es für den aufgerufenen EPA ein neuer *Event Type* sein, so wird der Subscribe an den SuperEPA weitergereicht. Ist der aufgerufene EPA Konsument von dem *Event Type* (selber oder ein verbundener

Knoten) so wird die `Publish` Operation mit der `Subscribe` Operation beantwortet. Ist dies nicht der Fall, so erfolgt keine Rückantwort an den Aufrufer.

Algorithmus 1 Publish

```
    // Ankündigung von Produzenten (EP), dass Event Type (ET) ab sofort produziert wird.
    procedure Publish(ET, EP)
        if ET ∉ ETpublish AND ET ∉ producingET then  // Falls ET bisher nicht produziert
            superEPA.Publish(ET, self); // SuperEPA über neuen Event Type informieren
 5:     end if
          // Zur Liste der produzierten ETs hinzufügen, falls noch nicht vorhanden
        ETpublish := ETpublish ∪ (ET,EP);
         // ET subscriben, wenn mindestens ein anderer EC bzw. EPA den ET konsumiert
        if ET ∈ (ETsuperEPA ∪ producingET) OR (ETsubscribe(ET) - EP) ≠ ∅ then
            EP.Subscribe(ET, self);
10:     end if
    end procedure
```

Die `Subscribe` Operation (vergleiche Algorithmus 2) bietet einem Konsumenten die Möglichkeit, bei einem EPA einen bestimmten *Event Type* zu abonnieren. Diese Operation kann sowohl vom SuperEPA als auch von den Kindknoten des betroffenen EPAs aufgerufen werden.

Algorithmus 2 Subscribe

```
    // Ankündigung von Konsumenten (EC), dass Event Type (ET) ab sofort abonniert wird.
    procedure Subscribe(ET, EC)
        if EC = superEPA then // Fall 1, subscribe kommt vom SuperEPA
            if ET ∉ ETsuperEPA then
 5:             ETsuperEPA := ETsuperEPA ∪ ET;
                // Sollte es bisher noch keinen Abonnenten gegeben haben, so muss an alle
                // Produzenten subscribe gesendet werden
                if ET ∉ ETsubscribe AND ET ∉ consumingET then
                    for all EP ∈ ETpublish(ET) do
10:                     EP.Subscribe(ET, self);
                    end for
                end if
            end if
```

```
    else // Fall 2, subscribe kommt vom eigenen EPN-Teilbaum (Kindknoten)
15:      // Prüfen, ob es bisher einen Konsumenten für ET gab
        if ET ∉ ETsubscribe AND ET ∉ consumingET AND ET ∉ ETsuperEPA then
            superEPA.Subscribe(ET, self); // Subscribe an SuperEPA weiterleiten
            // Subscribe an entsprechende Produzenten weiterleiten
            for all EP ∈ ETpublish(ET) do
20:             EP.Subscribe(ET, self);
            end for
        end if
        ETsubscribe := ETsubscribe ∪ (ET, EC);
    end if
25: end procedure
```

Die `Unpublish` Operation (vergleiche Algorithmus 3) ist das Gegenstück zur `Publish` Operation. Mit ihr kann ein Produzent ankündigen, dass dieser einen bestimmten *Event Type* nicht mehr produziert. Genau wie `Publish` Operation kann die `Unpublish` Operation nur auf den entsprechenden SuperEPA ausgeführt werden. Sollte ein EPA, durch Wegfallen eines Produzenten, selbst kein Anbieter für den entsprechenden *Event Type* sein, so wird die `Unpublish` Operation an den jeweiligen SuperEPA weitergereicht.

Algorithmus 3 Unpublish

```
// Ankündigung von Produzenten (EP), dass Event Type (ET) ab sofort nicht
// mehr produziert wird.
procedure Unpublish(ET, EP)
    ETpublish := ETpublish - (ET, EP); // aus Liste der Produzenten entfernen
5:   // Falls kein weiterer Produzent vorhanden ist, unpublish an den SuperEPA
    if ET ∉ ETpublish AND ET ∉ producingET then
        superEPA.Unpublish(ET, self); // Unpublish an SuperEPA weiterleiten
        ETsuperEPA := ETsuperEPA - ET; // sofern dieser vorher Konsument war
    end if
10: end procedure
```

Die `Unsubscribe` Operation (vergleiche Algorithmus 4) dient dazu, dass ein Konsument einem EPA mitteilen kann, dass er einen bestimmten *Event Type* nicht mehr benötigt. Dieser Aufruf kann sowohl vom Kindknoten als auch vom SuperEPA

ausgelöst werden. In beiden Fällen muss, sofern kein Konsument mehr vorhanden ist, die `Unsubscribe` bei allen Produzenten ausgeführt werden.

Algorithmus 4 Unsubscribe

```
    // Ankündigung von Konsumenten (EC), dass Event Type (ET) ab sofort nicht
    // mehr abonniert wird.
    procedure Unsubscribe(ET, EC)
        if EC = superEPA then // Fall 1, unsubscribe kommt vom SuperEPA
 5:         ETsuperEPA := ETsuperEPA - ET;  // nicht mehr an SuperEPA weiterleiten
            // Wenn kein Abonnenten mehr vorhanden, dann unpublish an alle Produzenten
            if ET ∉ ETsubscribe AND ET ∉ consumingET then
                for all EP ∈ ETpublish(ET) do
                    EP.Unsubscribe(ET, self);
10:             end for
            end if
        else // Fall 2, subscribe kommt von eigenen EPN-Teilbaum
            ETsubscribe := ETsubscribe - (ET, EC)  // Aus Liste der Abonnenten entfernen
            // Wenn kein Abonnenten mehr vorhanden, dann unpublish an alle Produzenten
15:         // (hier auch an den SuperEPA)
            if ET ∉ ETsubscribe AND ET ∉ ETsuperEPA AND ET ∉ consumingET then
                for all EP ∈ ETpublish(ET) do
                    EP.Unsubscribe(ET, self);
                end for
20:             superEPA.Unsubscribe(ET, self);
            end if
        end if
    end procedure
```

4.5.3 Publish/Subscribe Beispiel

Wir wollen nun die eben vorgestellten Algorithmen mit einem größeren Beispiel vertiefen. Dieses Beispiel wird in vier Schritten ausgeführt werden. Wir beginnen dazu mit dem EPN aus Abbildung 4.16 a. In den folgenden Abbildungen ist bei jedem EPA die Information annotiert, welche Events produziert (P), konsumiert (K) und vom SuperEPA abonniert (S) werden. Die Information, von welchem Kindknoten ein Event produziert bzw. konsumiert wird ist dabei, der besseren Verständlichkeit wegen, ausgeblendet. Ist ein *Event Type* unterstrichen, bedeutet dies, dass der EPA

selber Events von diesem Typ produziert bzw. konsumiert. Ein Fett hinterlegter *Event Type* bedeutet, dass dieser *Event Type* in der jeweiligen Liste neu hinzugekommen ist.

Beispiel 4.9 (Publish/Subscribe Schritt 1)
Das EPN aus Abbildung 4.16 a besitzt lediglich zwei EPAs. EPA 0 konsumiert zu beginn Events vom Typ `x`. EPA 1 ist gerade neu hinzugekommen und produziert ebenfalls Events vom Typ `x`. Deswegen ruft EPA 1 nun die `Publish` Operation bei seinem SuperEPA (EPA 0) auf. Daraus ergibt sich das EPN aus Abbildung 4.16 b. Da EPA 0 Events vom Typ `x` konsumiert, ruft dieser, als Reaktion, die `Subscribe` Operation bei EPA 1 auf. Die Folge davon ist in Abbildung 4.16 c zu sehen. Ab sofort routet EPA 1 alle Events von Typ `x` an seinen SuperEPA.

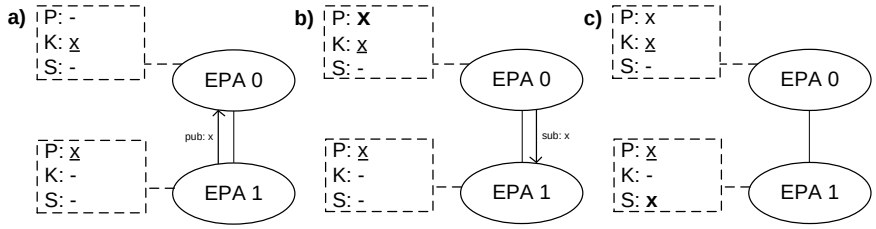

Abbildung 4.16: Publish/Subscribe Beispiel Schritt 1

Beispiel 4.10 (Publish/Subscribe Schritt 2)
In Abbildung 4.17 a ist nun EPA 2 hinzugekommen. Dieser produziert Events von den Typen `x` und `z` und konsumiert Events von Typ `y`. Dies teilt er jeweils mit der `Publish` und `Subscribe` Operation seinem SuperEPA (EPA 1) mit (vergleiche Abbildung 4.17 b). Als Reaktion darauf ruft EPA 1 die `Subscribe` Operation bei EPA 0 (*Event Type* `y`) und EPA 1 (*Event Type* `x`) auf. Zusätzlich ruft EPA 1 die `Publish` Operation bei EPA 0 (`z`) auf. Daraus folgt das EPN aus Abbildung 4.17 c.

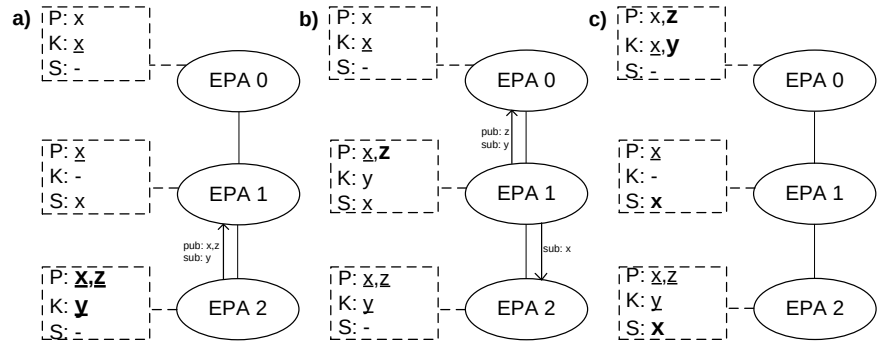

Abbildung 4.17: Publish/Subscribe Beispiel Schritt 2

Beispiel 4.11 (Publish/Subscribe Schritt 3)
In Abbildung 4.18 a kommt EPA 3 neu zu den EPN hinzu. Dieser konsumiert Events vom Typ `z`. Daraus folgt zunächst ein Aufruf der `Subscribe` Operation bei EPA 1. Das Ergebnis ist zunächst das EPN aus Abbildung 4.18 b. EPA 1 hatte bisher noch keinen Konsumenten für Events vom Typ `z`, aber einen Produzenten (EPA 2). Daraus folgt ein `Subscribe` Aufruf bei EPA 2 und dem SuperEPA (EPA 0). Das EPN aus Abbildung 4.18 c ist das Ergebnis dieser beiden `Subscribe` Aufrufe.

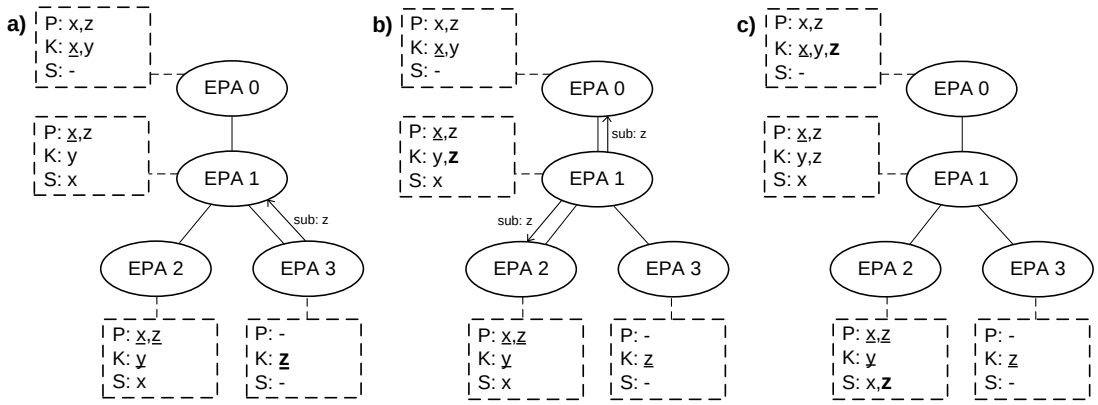

Abbildung 4.18: Publish/Subscribe Beispiel Schritt 3

Beispiel 4.12 (Publish/Subscribe Schritt 4)
In diesem Schritt wird EPA 2 aus dem EPN entfernt. Die Ausgangslage ist in Abbildung 4.19 a dargestellt. EPA 2 muss zunächst seinem SuperEPA mitteilen, welche Typen von Events dieser nicht mehr produzieren bzw. konsumieren kann. Dies geschieht mit der `Unpublish` Operation (*Event Types* `x` und `z`) und der `Unsubscribe` Operation (*Event Type* `y`). Daraus ergibt sich zunächst das EPN aus Abbildung 4.19 b. EPA 1 ist dadurch kein Produzent für Events vom Typ `z` mehr. Ebenso ist EPA 1 auch nicht mehr Konsument für Events vom Typ `y`. Daraus folgen die entsprechenden Operationsaufrufe an den SuperEPA. Die Konsequenz ist das EPN von Abbildung 4.19 c.

4.5.4 Routing

Wir haben im vorherigen Abschnitt diskutiert, wie die EPEs im EPN bekannt geben, welche Events sie produzieren und konsumieren. Nun betrachten wir das eigentliche Routing, d. h. nach welchen Kriterien ein EPA Events weiterleitet.

Durch die vier *Publish/Subscribe*-Operationen für *Event Types* ist jedem EPA bekannt, welche mit ihm verbundenen EPAs, ECs und EPs welche *Event Types* abonniert haben.

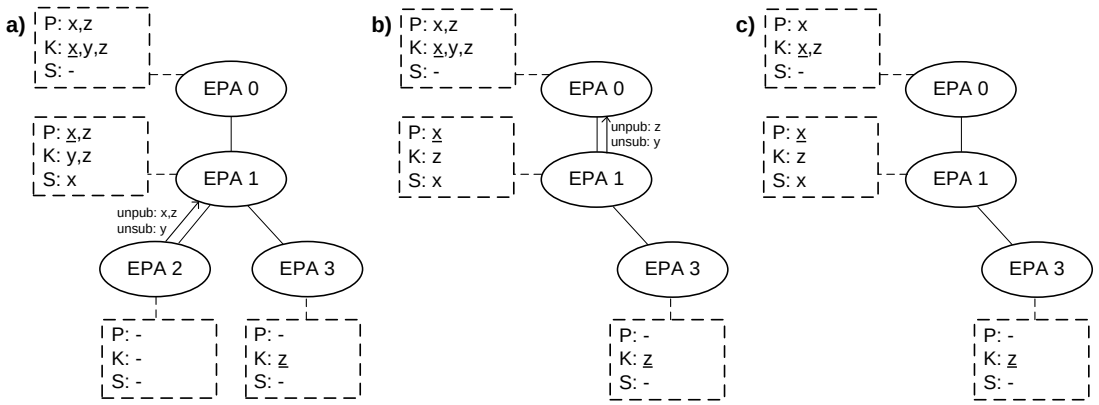

Abbildung 4.19: Publish/Subscribe Beispiel Schritt 4

Beim Eintreffen eines Events in einem EPA wird zunächst geprüft, ob der *Event Type* vom EPA selbst konsumiert wird. Ist dies der Fall, wird zunächst die entsprechende Eventbehandlung ausgeführt. In der Tabelle `ETsubscribe` ist zu einem *Event Type* verzeichnet, welche Knoten diesen abonniert haben. Nachdem ein Event an die entsprechenden Kindknoten verteilt ist, muss in der `ETsuperEPA` Liste geprüft werden, ob der SuperEPA den entsprechenden *Event Type* abonniert hat. Allgemein gilt dabei, dass ein EPA ein Event niemals an den EPE zurückschickt, von dem er das Event erhalten hat. Da der EPN-Baum ein azyklischer Graph ist, ist der Weg eines Events somit immer eindeutig. Wir betrachten zum vertiefen des Routings zwei Beispiele für das Routing in Prozessen.

Beispiel 4.13 (Routing zwischen zwei Aktivitäten im gleichem Prozess)
Abbildung 4.20 zeigt links einen Ausschnitt aus einem Prozess. In diesem wird ein Event `e1` von der Aktivität `A2` zur Aktivität `A1` gesendet. Wir gehen davon aus, dass beide Aktivitäten parallel ausgeführt werden, während das Event produziert wird. Auf der rechten Seite der Abbildung ist ein Ausschnitt aus dem entsprechenden EPN dargestellt. Die Aktivität `A2` sendet die Daten für das Event an den IEP. Dieser produziert daraus ein Event und leitet dieses an seinen SuperEPA (IEPA Prozess 1) weiter. Dieser hat als Abonnenten nur seinen eigenen IEC, an den er das Event nun weiterleitet. Nach Eintreffen des Events `e1` bei IEC (Prozess 1), muss dieser das Event an die Aktivität `A1` weiterleiten.

Beispiel 4.14 (Routing zwischen zwei Prozessen)
Abbildung 4.21 zeigt auf der linken Seite einen Ausschnitt aus einem Prozess. Dieser ist ähnlich aufgebaut wie im Beispiel 4.13. Allerdings besitzt dieser einen Subprozess mit einer

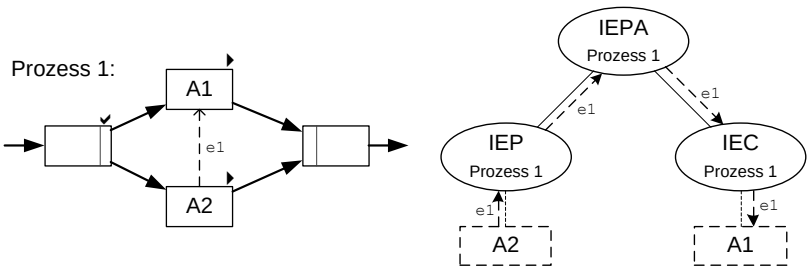

Abbildung 4.20: Beispiel Routing zwischen zwei elementaren Aktivitäten

Sequenz von Aktivitäten (`C1` bis `C3`). Ähnlich wie in Beispiel 4.13 produziert die Aktivität `C2` das Event `e2`. Einziger Empfänger ist die parallel laufende Aktivität `B1` im Vater-Prozess. Der Unterschied zum vorherigen Beispiel ist nun, dass das Event `e2` zunächst vom IEPA des Subprozesses (Sub1) zum IEPA des Vater-Prozesses(Prozess 2) verschickt werden muss. Im Vergleich zum Beispiel 4.13 wird hier also über einen EPA mehr geroutet.

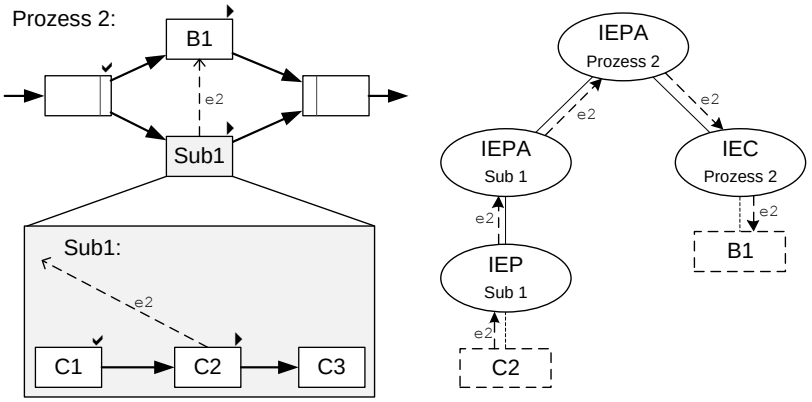

Abbildung 4.21: Beispiel Routing zwischen zwei elementaren Aktivitäten in unterschiedlichen Prozessen

Diese beiden Beispiele demonstrieren, wie das Routing in laufenden Prozess-Instanzen funktioniert. Insbesondere das letzte Beispiel demonstriert die Vorgänge im EPN bei Inter-Prozess-Kommunikation. Diese Beispiele lassen sich auch entsprechend auf größere EPN-Bäume übertragen. In Abbildung 4.22 ist ein solches Beispiel abgebildet. Dort produziert der EP Finanzverwaltung `Payment` Events, die vom Bezahlen-Prozessen konsumiert werden (vergleiche hierzu Beispiel 2.3). Da in diesem Fall zwei Bezahlen-Prozesse aktiv sind. Muss das `Payment` Event an beide Prozess-Instanzen weitergeleitet werden und nimmt dabei den eingezeichneten Weg.

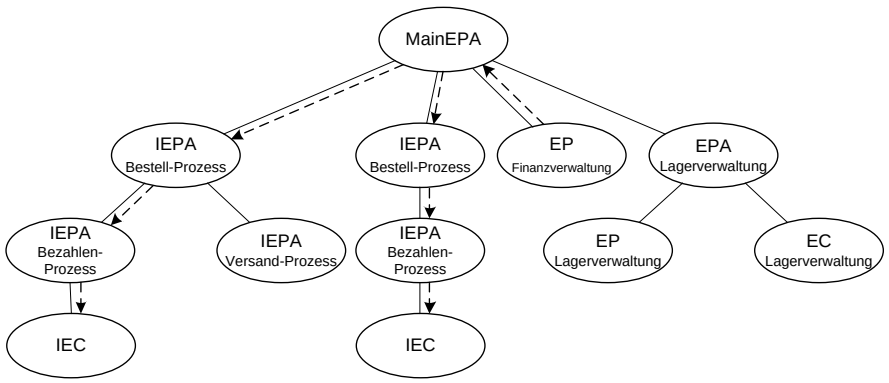

Abbildung 4.22: Routing eines `Payment` Events in einem EPN-Baum (für das Beispiel irrelevante IECs und IEPs sind ausgeblendet)

4.5.5 Zusammenfassung

Wir haben bereits zuvor gesehen, dass das Routing der Events, eine wichtige Eigenschaft des EPNs ist. Dies ist entscheidend, um die Kommunikation zwischen den Prozessen untereinander und mit den externen Systemen zu gewährleisten.

Deswegen wurde zunächst ein *Publish/Subscribe*-Mechanismus vorgestellt, der auf die Gegebenheiten des hier genutzten EPNs aufbaut. Mit diesen können beliebige Produzenten und Konsumenten sich gegenseitig austauschen, welche Events produziert und konsumiert werden. Dabei müssen die entsprechenden Knoten nicht direkt miteinander verbunden sein, sondern nur über das EPN verbunden sein.

Des Weiteren haben wir in diesen Abschnitt das Routing der Events behandelt. Dieses nutzt die Informationen, welche aus dem *Publish/Subscribe* gesammelt wurden, um die Events von Knoten zu Knoten an alle Ziele zu verteilen.

4.6 Event-Manager

Die bisher entwickelte EPN-Architektur ist als Erweiterung für BPM-Systeme ausgelegt. Daher benötigen wir eine zusätzliche Komponente für das BPM-System, welches die Event Processing Aufgaben überwacht.

Dazu wird wird eine Komponente benötigt, die das EPN verwaltet. Dazu gehört das hinzufügen und entfernen von EPEs im EPN. Damit EPEs für Prozess-Instanz und externen Systemen hinzugefügt und entfernt werden können, werden Schnittstellen

nach außen benötigt, damit dieses dem Event Processing System mitgeteilt werden kann. Des Weiteren müssen EPE-Implementierungen und *Event Type* Spezifikationen verwaltet werden. Somit können diese anderen Systemen zur Verfügung gestellt werden.

Für die oben genanten Aufgaben spezifizieren wir nun einen Event-Manager. Wir gehen nun auf die einzelnen Aufgaben des Event-Manager genauer ein.

Verwaltung des EPNs

Der Event-Manager verwaltet das EPN, d. h. er kümmert sich, um das hinzufügen und entfernen von EPEs im EPN-Baum. Dazu stellt er mehrere Schnittstellen, für das BPM-System und für externe Komponenten, bereit. Mit diesen informiert das BPM-System den Event-Manager beispielsweise darüber, dass neue Prozess-Instanzen hinzugekommen oder entfernt wurden. Damit kann der EPN-Baum wie in Kapitel 4.4 beschrieben angepasst werden. Des Weiteren kann über diese Schnittstellen das BPM-System dem Event-Manager mitteilen, dass eine Aktivität gestartet oder beendet wird. Dies ist wichtig für das `Publish/Subscribe`, da sich die möglichen *Event Types*, der produzierten und konsumierten Events, dadurch ändern können. Zusätzlich können sich die Applikationen beim Event-Manager an- und abmelden. Dadurch erhalten sie von jeweiligen IECs ihre Events, bzw. können über die jeweiligen IEPs Events produzieren und ins EPN einbringen.

Zusätzlich bietet der Event-Manager Schnittstellen, für externe Systeme. So können vordefinierte EPE hinzugefügt und wieder entfernt werden. Damit kann z. B. ein ERP-System mit dem EPN verbunden werden, indem die vordefinierten EPEs gestartet werden. Der Event-Manager kümmert sich darum, dass die jeweiligen EPEs mit den MainEPA verbunden werden.

Event Type Repository

Eine weitere Aufgabe des Event-Manager ist die Verwaltung der *Event Type* Spezifikationen. Diese werden im *Event Type* Repository abgelegt. In der *Event Type* Spezifikation werden Eigenschaften definiert, wie z. B. der Name und die Payload Attribute. Mit diesen Spezifikationen können Vorlagen für Event-Instanzen generiert werden. Zusätzlich können Events auf ihre Gültigkeit geprüft werden. Wenn vorher festgelegte Attribute nicht vorhanden sind, kann in den meisten Fällen ein Event

nicht korrekt ausgewertet werden. Dies stellt unter anderem sicher, dass vereinbarte Schnittstellen eingehalten werden.

EPE Repository

Genau wie die *Event Types* müssen EPE spezifiziert werden. Zur Wiederverwendung werden die EPE Vorlagen in einem EPE Repository abgelegt. Dort werden die Spezifikationen und deren Implementierungen gespeichert. Hiermit sind sie für den späteren Einsatz zugreifbar.

4.7 Zusammenfassung

In diesem Kapitel haben wir eine spezialisierte EPN-Architektur für BPM-Systeme behandelt. Diese hat als besondere Eigenschaft die Form eines Baumes. Der EPN-Baum hat in der Praxis mehrere Vorteile. Der Erste wichtige Aspekt ist dabei, dass der Weg eines Events von einem Quellknoten bis zu den Zielknoten immer eindeutig ist. Zudem lässt sich der EPN-Baum leicht und automatisiert zur Laufzeit anpassen. Dies ist wichtig, da permanent Prozess-Instanzen gestartet und beendet werden. Zusätzlich war es wichtig, dass externe Systeme, unter anderem Legacy-Systeme, leicht mit dem EPN verbunden werden können. Da das EPN einen zentralen Wurzelknoten (der MainEPA) hat, ist dieser der geeignetste Verbindungskonten für weitere EPEs. Dabei ist es auch möglich, zu einzelnen externen Systemen eigene hierarchisch strukturierte EPNs zu erstellen. Damit können die einzelnen Funktionen besser auf unterschiedliche EPEs verteilt werden.

Wir haben schon vorher festgestellt, dass die korrekte Verteilung der Events eine entscheidende Anforderung für das Event Processing System ist. Daher wurde ein *Publish/Subscribe*-Mechanismus eingeführt. Dieser ist speziell für die Baumeigenschaft ausgelegt. Damit ist es möglich, dass Knoten zur Laufzeit *Event Types* abonnieren bzw. ankündigen können.

5 Erweiterung des Prozessmodells

Im vorangegangenen Kapitel wurde eine konkrete EPN Architektur für BPM-Systeme vorgestellt. In diesem Kapitel behandeln wir nun die Erweiterung des Prozessmodells für Event Processing. Dabei gehen wir vor allen Dingen auf die Erweiterungen des ADEPT-Metamodells ein. Die Erweiterungen sind aber so ausgelegt, dass sie auch auf andere BPM Modelle übertragen werden können.

Wir erweitern zunächst in Abschnitt 5.1 die Aktivitätenvorlage um Konzepte für Events. Ähnlich, wie bei Datenelementen, muss auch für Events definiert sein, welche Events von einer Aktivität produziert und konsumiert werden. Dabei ist auch entscheidend, ob ein Event bei jeder Ausführung der Aktivität produziert bzw. konsumiert wird oder optional ist. Dies ist für die Erweiterung des Prozessmodells in Abschnitt 5.2 von Bedeutung. Hier wird unter anderem festgelegt, woher die einzelnen Events für die Aktivitäten stammen und welche Event Processing Aufgaben der Prozess ausführt.

In Abschnitt 5.3 werden die Zustände für Aktivitäten um einen neuen Zustand erweitert (`EVENT-LOCKED`). Damit kann in einer laufenden Prozess-Instanz angezeigt werden, dass eine bestimmte Aktivität auf das Eintreffen eines Events wartet. Als Letztes erweitern wir, in Abschnitt 5.4, das ADEPT-Metamodell um eine neue Event-basierte Verzweigung (EB-XOR). Abgeschlossen wird das Kapitel durch eine Zusammenfassung in Abschnitt 5.5.

5.1 Erweiterung der Aktivitätenvorlage

Bisher sind wir davon ausgegangen, dass Aktivitäten Events produzieren und konsumieren. Dabei wurde bisher nicht geklärt, wie dies zur Modellierungszeit festgelegt wird. Dies ist insofern wichtig, da manche Aktivitäten bestimmte Events benötigen. Ein Beispiel hierfür ist die `Ware verpacken` Aktivität aus dem Versand-Prozess von Beispiel 2.3. Diese benötigt ein `Bill created` Event, um bearbeitet werden zu

können. Daher muss vom Benutzer festgelegt werden, welche Events eine Aktivität produziert und konsumiert. Ansonsten können Verklemmungen aufgrund von fehlenden Events zur Laufzeit eintreten.

In vielen Fällen ist aber nicht sicher, ob eine Aktivität alle Events auch sicher produziert und konsumiert. Manche Aktivitäten können z. B. zur Laufzeit auf Events reagieren, aber auch ohne diese abgeschlossen werden. Ebenso ist es auch möglich, dass eine Aktivität nur in speziellen Situationen Events produziert, z. B. wenn eine Ausnahmesituation aufgetreten ist. Daher müssen wir zusätzlich unterscheiden, ob ein *Event Type* obligatorisch oder optional ist.

Dazu passen wir im Folgenden die Aktivitätenvorlage an die oben genannten Vorgaben an. "Eine Aktivitätenvorlage beschreibt eine Aufgabe, die durch eine Schnittstelle und eine Implementierung charakterisiert wird" [Rei00]. Wir werden nun die Schnittstellenbeschreibung für eine Aktivität für den hier vorgestellten Event-Mechanismus erweitern[7]. Jede Aktivitätenvorlage wird dazu um folgende Attribute erweitert:

Consuming Eine Liste mit den *Event Types* der Events, die beim Ablauf der Aktivität konsumiert werden. Zusätzlich wird bei jedem Typ angegeben, ob dieser obligat oder optional ist.

Producing Eine Liste mit den *Event Types* der Events, die von der Aktivität produziert werden. Auch hier wird jeweils angegeben, ob der *Event Type* obligat ist.

Filter Eine Liste mit Filtern für eingehende Events. Ähnlich wie bei den Filtern für ECs und EPAs (siehe Abschnitt 4.2.2 und 4.2.3) kann mit Filtern die Anzahl der zu verarbeitenden Events verringert werden.

Zur Vereinfachung unterscheiden wir nicht, ob eine Aktivität, während ihrer Laufzeit, ein oder mehrere Events eines bestimmten Typs produziert oder konsumiert. Stattdessen geben wir nur an, dass mindestens ein Event eines bestimmten *Event Types* produziert bzw. konsumiert wird.

Mit dem Attribut `Consuming` wir in der Aktivitätenvorlage beschrieben, welche Events eine Aktivität konsumiert. Hier ist insbesondere die Angabe wichtig, ob das Event obligat konsumiert wird. Wenn ja, muss sichergestellt sein, dass eine Aktivität

[7] für eine vollständige Auflistung aller Attribute einer Aktivitätenvorlage siehe [Rei00]

zur Laufzeit mindestens ein Event des jeweiligen Typs bekommen kann. Kann dies nicht gewährleistet werden, gehen wir davon aus, dass auch die Aktivität nicht beendet wird. In diesem Fall liegt eine Verklemmung vor. Im Folgenden verwenden wir den Ausdruck "ein Event ist versorgt", wenn sichergestellt ist, dass zur Laufzeit der Aktivität ein Event des entsprechenden Typs produziert wird. Es muss bei der Zusammenstellung eines Prozesses sichergestellt sein, dass von allen Aktivitäten jeweils alle obligat konsumierten Events versorgt sind.

Mit `Producing` geben wir an, welche verschieden Typen von Events eine Aktivität produzieren kann. Damit erhalten wir potenzielle Quellen für zu versorgende Events innerhalb eines Prozesses. Hier sind insbesondere die obligat produzierten Events wichtig. Diese werden benötigt, um zur Entwicklungszeit prüfen zu können, ob ein Event versorgt ist.

Die Attribute `Consuming` und `Producing` sind auch für das *Publish/Subscribe* wichtig. Beim Starten einer Aktivität werden, durch diese Angaben, die jeweiligen *Event Types* angekündigt, bzw. abonniert.

Die Filter haben die Aufgabe, die Anzahl der eingehenden Events zu begrenzen, um die laufenden Aktivitäten zu entlasten. Mit den Filtern werden Events aussortiert, die keine Relevanz für die jeweilige Aktivität haben. Im Bestell-Prozess (siehe Beispiel 2.3) kann z. B. bei jeder Aktivität ein Filter eingesetzt werden, der nur Events durchlässt, bei denen die zugehörige `OrderID` dem der aktuellen Prozess-Instanz entspricht. So können Events für andere Bestellvorgänge ignoriert werden.

Beispiel 5.1 (Aktivitätenvorlagen im Bestell-Prozess)
Betrachten wir den Bestell-Prozess aus Beispiel 2.3. Die Aktivitäten, aus den beiden Subprozessen des Bestell-Prozesses, konsumieren und produzieren Events von unterschiedlichen Typen. In diesem Beispiel sind alle Events obligat. Tabelle 5.1 gibt eine Übersicht, über die Aktivitäten und ihre obligat produzierten und konsumierten Events.

5.2 Erweiterung der Prozessvorlage

Im vorherigen Abschnitt haben wir bisher nur die Erweiterung für die Aktivitätenvorlagen betrachtet. Dort sind wir bereits auf die Versorgung von Events, in einzelnen Aktivitäten, eingegangen. Diese muss bei der Erstellung eines Prozesses festgelegt sein. Allerdings können nicht in allen Fällen alle benötigten Events im gleichen

Prozess	Aktivität	Consuming	Producing
Bezahlen	Geld Eingang	Payment	Payment Completed
Bezahlen	Rechnung erstellen	-	Bill Created
Versand	Ware reservieren	-	Reserve
Versand	Ware kommissionieren	Order Picking Completed	Order Picking
Versand	Ware verpacken	Bill Created	-

Tabelle 5.1: Die *Event Types* der obligat produzierten und konsumierten Events aus dem Beispielprozess

Prozess produziert werden. Ein Beispiel hierfür findet sich im Versand-Prozess von Beispiel 2.3. Das `Payment Completed` Event wird zwar innerhalb des Versand-Prozesses benötigt, aber nicht dort produziert. Daraus folgt, dass die Versorgung in diesem Fall vom Vater-Prozess oder einem externen System sichergestellt werden muss. Dies muss bei der Modellierung eines Prozesses berücksichtigt werden können.

Zusätzlich muss auch die asynchrone Kommunikation beachtet werden. In einigen Fällen ist konsumierende Aktivität, zum Zeitpunkt der Produktion des entsprechenden Events, noch nicht gestartet[8]. Deswegen muss auch eine spätere Versorgung mit Events möglich sein.

Die oben genanten Fälle lassen sich mit dem bisherigen *Publish/Subscribe*-Mechanismus nicht umsetzten. Daher muss dieser auch entsprechend angepasst werden. Zusätzlich wird auch ein Aktivitätenübergreifendes Event Processing benötigt, wie z. B. Filter oder *Pattern Detection*. Dieses muss auch zur besseren Wiederverwendbarkeit der Prozessvorlagen definiert werden können.

Um die diskutierten Vorgaben umzusetzen, entwickeln wir zunächst die Versorgung von Events innerhalb eines Prozesses und die sich daraus ergebenden Konsequenzen für das *Publish/Subscribe*. Zum Abschluss diskutieren wir die globalen Prozessattribute.

5.2.1 Versorgung von Events

Wir betrachten nun die Versorgung von Aktivitäten mit obligaten Events. Wir definieren in der Prozessvorlage, wie die einzelnen Events versorgt werden. Dabei betrachten wir nicht nur die obligaten, sondern auch die optionalen Events. Dazu werden in der

[8] siehe auch hier `Payment Completed` Event in Beispiel 2.3

Prozessvorlage, bei jedem Knoten, der mit einer Events konsumierenden Aktivität belegt ist, zu jedem *Event Type* folgende Attribute gespeichert (jeweils mit Wert `TRUE` und `FALSE`):

intern Gibt an, ob das jeweilige Event innerhalb dieser Prozessvorlage, d. h. von einer anderen Aktivität im gleichen Prozess, produziert wird. Sollte hier `FALSE` angegeben sein, muss das Event von einer höheren Ebene produziert werden können.

early Gibt an, ob der *Event Type* vorzeitig, d. h. schon vor Beginn der Aktivität, abonniert werden soll. In diesem Fall werden eingehende Events gespeichert, falls die Aktivität noch nicht gestartet ist. Bei `FALSE` wird der entsprechende *Event Type* erst für eine Aktivität abonniert, sobald sie gestartet wurde.

Events, die mit `intern` gekennzeichnet sind, werden nur von dem aktuellen Prozess versorgt. Dadurch erhält die Aktivität zur Laufzeit nur Events innerhalb des Prozesses und nicht vom Vaterprozess, anderen Prozess-Instanzen oder externen Komponenten. Dies bedeutet insbesondere für obligat konsumierte Events, dass mindestens eine Aktivität vorhanden sein muss, die das jeweilige Event obligat produziert. Es hat außerdem zur Konsequenz, dass der entsprechende *Event Type* zur Laufzeit nur innerhalb der Prozess-Instanz abonniert wird.

Ist ein *Event Type* als nicht `intern` gekennzeichnet, so wird dieser normal abonniert. Allerdings muss die Versorgung auf der jeweils höheren Ebene sichergestellt sein. Dies bedeutet, wird der Prozess als Subprozess in einer anderen Prozessvorlage genutzt, muss das Event dort versorgt werden. Auch dort bietet sich die Möglichkeit, die Aufgabe der Versorgung auf die nächsthöhere Ebene weiterzureichen. Dies kann sukzessive geschehen, bis der Top-Level-Prozess erreicht ist. Kann selbst im Top-Level-Prozess die Versorgung nicht prozessintern sichergestellt werden, so wird ein Produzent außerhalb des Prozesses benötigt. Dies kann z. B. ein externes System oder ein anderer Prozess sein. Die Existenz eines entsprechenden Produzenten kann demnach erst beim Starten des Prozesses sichergestellt werden.

Beispiel 5.2 (Versorgung außerhalb eines Prozesses)
Als Beispiel, für eine nicht prozessinterne Versorgung, betrachten wir das `Payment Completed` Event aus dem Versand-Prozess (siehe Beispiel 2.3). Dieses Event wird nicht innerhalb des Versand-Prozesses produziert. Deshalb wird hier `intern` auf `FALSE` gesetzt.

Da der Versand-Prozess als Subprozess im Bestell-Prozess genutzt wird, muss dort das Event versorgt werden. Dies geschieht im Bestell-Prozess prozessintern durch den Bezahlen-Prozess. Würde der Bestell-Prozess als Subprozess in einem anderen Prozess genutzt werden, braucht dieser das Event nicht mehr versorgen.

Ist das Attribut `early` auf `TRUE` gesetzt, wird für die entsprechende Aktivität der *Event Type* schon zu Beginn der Prozess-Instanz abonniert. Zusätzlich werden die eintreffenden Events gespeichert, falls die jeweilige Aktivität noch nicht gestartet wurde. Sobald die Aktivität gestartet wird, erhält sie alle für sie bereitgehaltenen Events. In vielen Fällen ist dies notwendig, da Events innerhalb eines Prozesses produziert werden, bevor die konsumierende Aktivität gestartet wurde. Betrachten wir hierzu erneut als Beispiel den Bestell-Prozess. Das `Payment Completed` Event kann aufgrund der Parallelität der Subprozesse von der `Bezahlen` Aktivität produziert werden, bevor die `Ware kommissionieren` Aktivität gestartet wurde. Wird das entsprechende Event nicht vorgehalten, kann die `Ware kommissionieren` Aktivität nicht abgeschlossen werden. Deswegen muss bereits direkt nach Beginn des Bestell-Prozesses der `Payment Completed` *Event Type* abonniert werden. Sobald der Versand-Prozess gestartet wurde, abonniert dieser dann die `Payment Completed` Events.

Wird die `early` Eigenschaft auf `FALSE` gesetzt, wird der entsprechende *Event Type* erst abonniert, nachdem die Aktivität gestartet wurde. Es ist aber nicht ausgeschlossen, dass eine andere Aktivität schon vorher den entsprechenden *Event Type* abonniert hat.

Beispiel 5.3 (Versorgung der Events aus dem Bestell-Prozess)
Im Bestell-Prozess von Beispiel 2.3 werden mehrere verschiedene Events produziert und konsumiert. In Tabelle 5.2 wird für jede Aktivität und *Event Type* angegeben, ob die Events intern versorgt (Attribut `intern`) werden und ein vorzeitiges abonnieren (Attribute `early`) notwendig ist.

Alle obligaten Events aus den Subprozessen können nicht intern produziert werden. Deswegen geht die Verantwortung für die Versorgung an den Vater-Prozess. Dadurch sind die *Event Types* in Tabelle 5.2 jeweils zweimal aufgeführt, einmal innerhalb des jeweiligen Subprozesses und einmal für den Top-Level-Prozess (Bestell-Prozess). Im Bestell-Prozess können lediglich die `Payment Completed` und `Bill created` Events intern versorgt werden. Die übrigen Events müssen von anderen Quellen bezogen werden, in diesem Fall, von den externen Systemen Finanzverwaltung und ERP-Lagerverwaltung.

Prozess	Aktivität	Event Type	intern	early
Bezahlen	Geld Eingang	Payment	False	True
Versand	Ware kommissionieren	Payment Completed	False	True
	Ware kommissionieren	Order Picking Completed	False	False
Bestell	Bezalen	Payment	False	True
	Versand	Payment Completed	True	True
	Versand	Order Picking Completed	False	False
	Versand	Bill Created	True	True

Tabelle 5.2: Versorgung der obligat konsumierten Events im Bestell-Prozess

Das Attribut `early` ist bei allen Events, außer dem `Order picking Completed` Event, auf `TRUE` gesetzt. Das `Order picking Completed` wird erst nach Beginn der `Ware kommissionieren` Aktivität produziert, da dieses eine Reaktion auf das `Order picking Request` Event ist. Es ist zwar möglich, dass andere Prozesse oder Systeme Events von diesem Typ schon vorher produzieren, diese sind aber nicht für die jeweilige Prozess-Instanz vorgesehen.

5.2.2 Konsequenzen für den Publish/Subscribe-Mechanismus

Bisher sind wir zur Vereinfachung davon ausgegangen, dass ein *Event Type* für eine konsumierende Aktivität erst dann abonniert wird, wenn die entsprechende Aktivität gestartet wurde. Der vorherige Abschnitt hat gezeigt, dass diese Betrachtung in der Praxis nicht alle Fälle abdeckt. Es muss zu mindestens ein Mechanismus eingeführt werden, der es erlaubt, dass *Event Types* vorzeitig abonniert und Events zwischengespeichert werden können.

Deshalb wird in diesem Abschnitt der *Publish/Subscribe* Mechanismus erweitert. Zunächst benötigt der IEPA neue Funktionen. Er bekommt dazu eine Tabelle, in der verzeichnet ist, welche Aktivitäten und *Event Types* bzw. `early` und `intern` gekennzeichnet sind. Damit kann der IEPA beim Starten eines Prozesses entscheiden, welche *Event Types* gleich zu Beginn eines Prozesses abonniert werden sollen. Zusätzlich kann bei eintreffenden Events entschieden werden, ob diese nur, an mit dem IEPA verbundene Knoten weitergegeben oder ob die Events auch gespeichert

werden sollen. Des Weiteren kann mithilfe der Informationen aus der Prozessvorlage gewährleistet werden, dass mit `intern` gekennzeichnete *Event Types* nicht vom SuperEPA abonniert werden.

Damit ein IEPA bzw. IEC beim Starten einer Aktivität[9] die bereitgestellten Events abrufen kann, braucht der IEPA eine weitere Operation. Diese führen wir hier unter der Bezeichnung `getSavedEvents(ET)` ein. Damit lassen sich alle Events eines bestimmten Typs (`ET`) abrufen, die bis dahin beim IEPA eingetroffen sind und gespeichert wurden. Damit die Events auch späteren Aktivitäten zur Verfügung stehen, werden die Events nach Abruf nicht aus dem IEPA Speicher entfernt.

5.2.3 Erweiterung der globalen Prozessattribute

Bisher wurde bei der Erweiterung des Prozessmodells nur auf die Erweiterung der einzelnen Aktivitätsknoten eingegangen. Nun diskutieren wir die Auswirkungen auf den globalen Teil der Prozessvorlage. Dazu bekommt die Prozessvorlage vier neue globale Attribute:

CEP-Module Eine Liste mit CEP-Modulen.

Filter Eine Liste mit Filtern für ein- und ausgehende Events.

Consuming global Eine automatisch generierte Liste mit den *Event Types* der Events, die vom Prozess und dessen Aktivitäten konsumiert werden. Zusätzlich wird bei jedem Typ angegeben, ob dieser obligat ist. Hier sind sowohl die nicht innerhalb des Prozess versorgen Events, als auch die Events von CEP-Modulen enthalten.

Producing global Eine automatisch generierte Liste mit den *Event Types* der Events, die vom Prozess produziert werden, jeweils mit der Angabe, ob obligat oder optional. Auch hier sind sowohl die Events von Aktivitäten, als auch von CEP-Modulen enthalten.

Die Listen `Consuming global` und `Producing global` ergeben sich automatisch, aus den Aktivitäten und den CEP-Modulen. Bei `Consuming global` sind alle in den Aktivitäten vorkommenden konsumierten *Event Types* aufgelistet, die nicht nur als prozessintern auftreten. Sollte dabei der *Event Type* mindestens einmal als obligat

[9] unabhängig ob es elementare Aktivität oder ein Subprozess ist

erscheinen, so ist dieser auch in der globalen Liste als obligat aufgeführt. Hierbei ist es unerheblich, ob die entsprechende Aktivität nur in einer bedingten Verzweigung vorkommt, da bei jedem Prozessdurchgang alle Events versorgt werden müssen.

Bei `Producing global` verhält es sich etwas anders. In dieser Liste sind alle *Event Types* aufgeführt, die in mindestens einer Aktivität des Prozesses produziert werden (obligat und optional). Allerdings können nur *Event Types* als obligat produziert gekennzeichnet werden, die auch in jedem Prozessdurchlauf obligat produziert werden. Dies bedeutet z. B., dass wenn ein *Event Type* nur von einer Aktivität in einer bedingten Verzweigung obligat produziert wird, der *Event Type* nicht als obligat produziert für den Prozess eingestuft werden kann. Eine Ausnahme besteht, wenn der gleiche *Event Type* auch in allen anderen Pfaden der bedingten Verzweigung obligat produziert wird.

Die Attribute `Filter` und `CEP-Module` verhalten sich ähnlich wie die Attribute für Aktivitätenvorlagen bzw. EPA Definitionen. Mit Filtern könnten die Anzahl der zu verarbeitenden Events verringert werden. CEP-Modulen können mit globalen *Event Handler*[10] ausgestattet werden. Auf diese Weise kann auf verschiedene Events, die in unterschiedlichen Aktivitäten im Prozess produziert wurden, reagiert werden. So kann entweder mit der Ausführung einer Aktivität oder Produktion eines Events hierauf reagiert werden.

Zur Vertiefung betrachten wir Tabelle 5.3. Dort sind die Attribute `Consuming global` und `Producing global` für den Bestell-Prozess aufgeführt.

5.3 Erweiterung der Aktivitätenzustände

Wir erweitern in diesem Abschnitt die Zustände für laufende Aktivitäten im ADEPT-Metamodell. Eine Aktivität muss gestartet sein, damit sie Events verarbeiten kann. Ist die Aktivität noch nicht gestartet, können die Events gespeichert und der Aktivität nach dem Start übergeben werden (s. o.). Allerdings gibt es Fälle, in denen eine laufende Aktivität auf ein Event wartet und sonst nichts weiter bearbeiten kann. Ein Beispiel hierfür ist der Versand-Prozess (siehe Beispiel 2.3). Die `Ware kommissionieren` Aktivität benötigt das `Payment Completed` Event von der `Geld Eingang` Aktivität (Bezahlen-Prozess). Solange keine Zahlung eingegangen ist, werden auch keine Waren

[10] vergleichbar mit BPEL

Prozessvorlage	Attribut	Event Types
Bezahlen	Consuming global	Payment
	Producing global	Payment Completed, Bill Created
Versand	Consuming global	Payment Completed, Bill Created, Order Picking Completed
	Producing global	Reserve, Order Picking
Bestell (Top-Level-Prozess)	Consuming global	Payment, Order Picking Completed
	Producing global	Reserve, Order Picking, Payment Completed, Bill Created

Tabelle 5.3: Globale Attribute für die Prozesse aus dem Bestell-Prozess und seine Subprozesse

kommissioniert. Die Aktivität `Ware kommissionieren` befindet sich dadurch in einem Wartezustand, bis das `Payment Completed` Event empfangen wird.

Bei Betrachtung der vorhandenen Zustände für Aktivitäten (vergleiche hierzu Abschnitt 2.1.3) fällt auf, dass dort kein Zustand vorhanden ist, um diese Situation adäquat umzusetzen. Am nächsten kommt der Zustand `SUSPENDED`. Allerdings kann eine Aktivität, die sich in diesem Zustand befindet, manuell von Benutzer wieder in den Zustand `STARTED` versetzt werden. Dies soll aber nur automatisch durch Eintreffen eines Events möglich sein.

Um diesem Umstand besser gerecht zu werden, führen wir nun einen neuen Zustand für laufende Aktivitäten ein: `EVENT-LOCKED`. Betrachten wir hierzu Abbildung 5.1. Dies ist die Erweiterung des Zustandsübergangsdiagrammes von Abbildung 2.4. Der neue Zustand hat Ähnlichkeiten mit dem Zustand `SUSPENDED`. Muss eine Aktivität auf ein Event warten und kann nichts weiter bearbeitet werden, liegt eine mit `SUSPENDED` vergleichbare Situation vor. Wenn eine laufende Aktivität nicht mehr weiter bearbeitet werden kann, weil sie auf ein bestimmtes Event warten muss, so wechselt sie in den Zustand `EVENT-LOCKED` (Vorgang `event-lock`). Trifft nun ein Event für diese Aktivität ein, so wird diese aufgeweckt (Vorgang `resume`) und wechselt wieder in den Zustand `STARTED`. In diesem Zustand kann die Bearbeitung fortgesetzt werden.

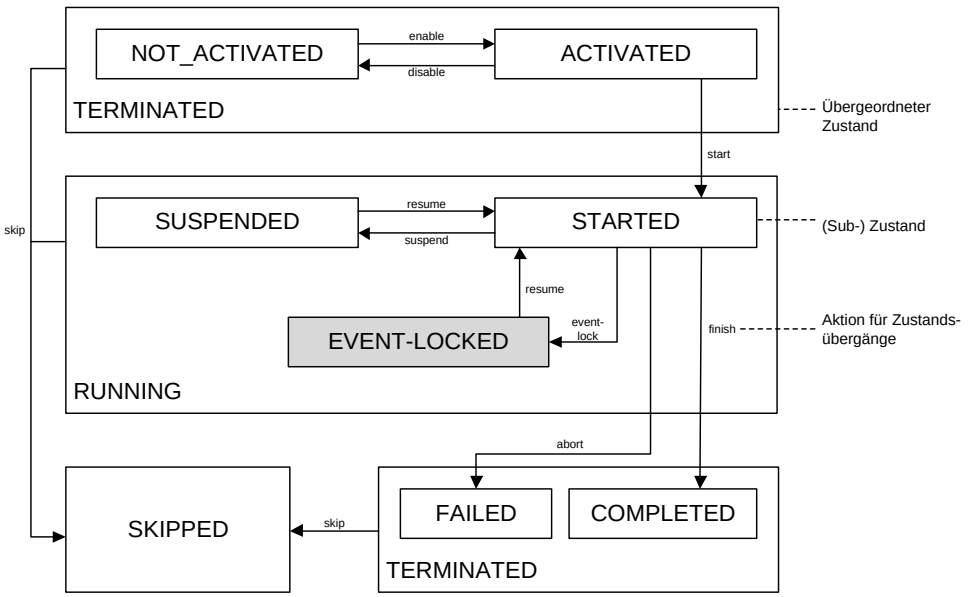

Abbildung 5.1: Erweitertes Zustandsübergangsdiagramm für Aktivitätsinstanzen (angelehnt an [Rei00])

Allgemein können Events unter anderem zur Synchronisation über Prozessgrenzen hinaus verwendet werden (siehe z. B. das `Payment Completed` Event im Bestell-Prozess). Mit dem `EVENT-LOCKED` Zustand wird bei einer Aktivität angezeigt, dass auf mindestens ein Event gewartet wird, ansonsten kann die Bearbeitung nicht fortgesetzt werden. Ein Monitor-Programm kann die Aktivitäten einer Prozess-Instanz anzeigen. Durch den Zustand `EVENT-LOCKED` kann der Benutzer nun erkennen, dass eine Aktivität aufgrund eines Events nicht weiter bearbeitet wird. Diese Verfeinerung dient somit auch der Verbesserung der Übersicht über laufende Programme.

5.4 EB-XOR

In diesem Abschnitt wird das ADEPT-Basismodell um eine neue Form von bedingter Verzweigung erweitert. In ADEPT existiert bereits eine bedingte Verzweigung oder auch ”XOR-Verzweigung” genannt. Dort wird zur Laufzeit einer von mehreren Pfaden ausgewählt (sieh hierzu Abschnitt 2.1.1). Die Auswahl des Nachfolgepfades erfolgt dabei aufgrund eingehender Daten. Events können durch ihren Payload auch als Datenträger genutzt werden. Daher ist es von Vorteil, wenn statt nur aufgrund

von Datenelementen, auch nach eingehenden Events der Nachfolgepfad ausgewählt werden kann. Dies würde bei der Prozessgestaltung mehr Möglichkeiten bieten und es könnten aussagekräftigere Prozesse modelliert werden.

Wir führen deswegen einen neuen Knotentyp für das ADEPT-Metamodell ein: Event-Based-XOR-Split-Knoten (EB-XOR-Split-Knoten). Dieser Knoten orientiert sich am Event-Based-Gateway von BPMN (siehe hierzu z. B. [OMG11]). Ein Symbol für den neuen Knoten ist in Abbildung 5.2 dargestellt.

Abbildung 5.2: Symbol für ein Event-Based XOR-Split-Knoten

Ähnlich wie bei einem datenbasierten XOR muss auch beim EB-XOR eine Menge von Regeln definiert werden, welcher Nachfolgepfad ausgewählt wird. Zur Erfüllung der Korrektheitsbedingungen (sieh [Rei00]) ist es unerlässlich, dass der Nachfolgepfad immer eindeutig ist. Bei eingehenden Daten ist dieses noch vergleichsweise einfach zu erreichen, da hier nur sichergestellt sein muss, dass alle eingehenden Daten vorher geschrieben wurden. Da allerdings auch nicht relevante Events eintreffen können, gestaltet sich das hier umfangreicher. Daher definieren wir die Auswahl der Nachfolgepfade in zwei Phasen.

In Phase eins wird eine Menge von *Event Types* definiert, auf die das EB-XOR reagiert. Zusätzlich wird bei jedem *Event Type* ein Filter definiert, welcher die relevanten Events bestimmt. So kann z. B. nach der entsprechenden `OrderID` gefiltert werden, damit nur Events berücksichtigt werden, die zum gleichen Bestellvorgang gehören.

In Phase zwei werden die Bedingungen zu den Nachfolgepfaden definiert. Dazu wir zu jedem *Event Type* eine Menge von relevanter Payload Attribute ausgewählt. Danach werden mit diesen Attributen Bedingungen definiert, welcher Nachfolgepfad auszuwählen ist (z. B. `x` < 4, `z` > `y`). Dabei ist zu beachten, dass alle möglichen Fälle abgedeckt sein müssen. Es kann bei Bedarf ein *default* Pfad definiert werden, der alle übrigen Fälle abdeckt. Es müssen nicht bei jedem einzelnen *Event Type* alle Nachfolgepfade möglich sein. Allerdings müssen alle Pfade abgedeckt sein.

Beispiel 5.4 (EB-XOR Entscheidung)
In Abbildung 5.3 ist ein Auszug aus einem Prozess dargestellt, in dem ein EB-XOR-Split-Knoten verwendet wird. Das EB-XOR reagiert auf `Order picking Request` Events. Dieses

Event enthält unter anderem eine Auflistung von Waren, welche kommissioniert werden sollen. Die Entscheidung über den Nachfolgepfad ist abhängig davon, wie viele Artikel im Event aufgelistet sind. Sollten nur ein bis drei Artikel enthalten sein, wird Aktivität `A` ausgeführt. Sollen vier bis sechs Artikel kommissioniert werden, wird Aktivität `B` ausgeführt. Sollten mehr als sieben Artikel kommissioniert werden, wird mit Aktivität `C` ausgewählt.

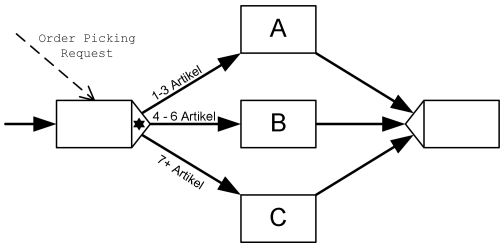

Abbildung 5.3: Beispiel für EB-XOR-Split

5.5 Zusammenfassung

In diesem Kapitel wurden die Aktivitäten- und Prozessvorlagen um Event Processing Funktionen erweitert. Mit der Erweiterung der Aktivitätenvorlage kann nun ausgedrückt werden, welche Events eine Aktivität produziert und konsumiert. Dabei wird auch unterschieden, ob dies obligat geschieht oder nur optional ist.

Mit der Erweiterung der Prozessvorlage kann nun berücksichtigt werden, wie eine Events konsumierende Aktivität versorgt werden soll. Dabei unterscheiden wir, ob eine Aktivität mit einem Event aus dem gleichen Prozess versorgt werden muss oder ob das Event eine beliebige Quelle haben kann. Des Weiteren unterscheiden wir, ob der entsprechende *Event Type* gleich zu Beginn der Prozess-Instanz abonniert wird oder erst nach Beginn der konsumierenden Aktivität. Dies ist insbesondere für die asynchrone Kommunikation wichtig. Hierbei ist es ist möglich, dass eine konsumierende Aktivität zum Zeitpunkt der Produktion des Events noch nicht bereit ist.

Zusätzlich wurde ADEPT um einen neuen Zustand für laufende Aktivitäten und ein neues Kontrollflusselement erweitert. Mit dem neuen Zustand `EVENT-LOCKED` kann nun dem Benutzer dargestellt werden, dass eine Aktivität aufgrund eines fehlenden Events in einem Wartezustand ist. Das neue Kontrollflusselement „Event-Based-XOR“ erweitert die Möglichkeiten bei der Gestaltung von Prozessen.

6 Korrektheitsaspekte

Im vorangegangen Kapitel haben wir bereits die korrekte Versorgung von Aktivitäten mit Events thematisiert. Ziel dieses Kapitels ist es, die sich daraus ergebenen Konsequenzen für die Korrektheit eines Geschäftsprozesses zu diskutieren. Dazu wird zunächst das Problem genauer betrachtet und motiviert (Abschnitt 6.1). Daraufhin wird ein Konzept für Eventkanten und einer Eventkorrektheitsbedingung für Knoten eingeführt (Abschnitt 6.2). Diese bilden die Grundlage eines Lösungsansatzes für die Korrektheitsanalyse (Abschnitt 6.3). Abgeschlossen wird das Kapitel mit einer Zusammenfassung und Fazit in Abschnitt 6.4. Dort wird auch auf einige Probleme, der zuvor entwickelten Analyse, hingewiesen.

6.1 Motivation

Wir haben bereits im vorherigen Kapitel diskutiert, dass ein Event, welches von einer Aktivität obligat konsumiert wird, versorgt sein muss. Dies bedeutet, dass ein Produzent für dieses Event existieren muss, der dieses auch sicher produziert (vergleiche hierzu obligates Lesen und Schreiben von Datenelementen [Rei00]). Es wurde auch unterschieden, ob die Versorgung nur prozessintern geschehen soll oder ob das Event auch von anderen Prozessen und externen Systemen produziert werden kann. In diesem Kapitel betrachten wir nun die prozessinterne Versorgung genauer. Sollte der Produzent außerhalb des jeweiligen Prozesses liegen, so muss die Versorgung an anderer Stelle geprüft werden (siehe dazu Abschnitt 5.2.1).

Bei einer prozessinternen Versorgung von obligaten Events muss sichergestellt sein, dass die Events auch obligat, innerhalb des Prozesses, produziert werden können. Dabei reicht es nicht zu überprüfen, ob mindestens eine Aktivität vorhanden ist, die das entsprechende Event obligat produziert. Es muss auch sichergestellt sein, dass es mindestens eine produzierende Aktivität gibt, die vor oder während der Laufzeit der konsumierenden Aktivität, ausgeführt wird. Dabei ist auch zu beachten, dass bei je-

dem Durchgang des Prozesses, in dem die konsumierende Aktivität ausgeführt, auch die produzierende Aktivität ausgeführt wird. Zur Veranschaulichung des Problems betrachten wir zunächst ein Beispiel:

Beispiel 6.1 (Versorgung von Events bei einer bedingten Verzweigung)
In Abbildung 6.1 ist ein Ausschnitt aus einem Prozess dargestellt. Die Aktivität `G` konsumiert die Events `e1` und `e2`. Da die Aktivitäten `C` und `D` in unterschiedlichen Pfaden einer bedingten Verzweigung liegen, können in einem Durchgang des Prozesses nicht beide Events produziert werden.

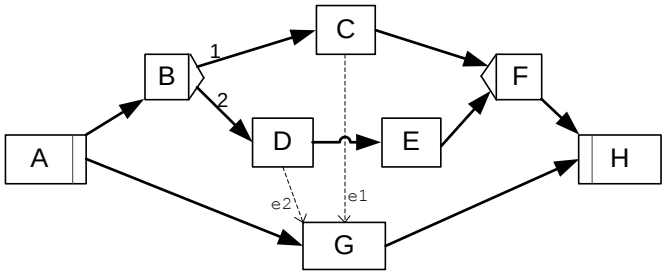

Abbildung 6.1: Beispielprozess für Versorgung mit Events bei einer bedingten Verzweigung

Dieses einfache Beispiel zeigt eine mögliche Ursache für eine Verklemmung durch Events. Benötigt die Aktivität `G` beide Events `e1` und `e2`, so ist der Prozess nicht „korrekt“, da in jedem Fall eine Verklemmung entsteht. Beide Events können nicht in einem Prozessdurchgang produziert werden. Sollte Aktivität `G` allerdings nur eines der beiden Events benötigen, so stellt dies kein Problem dar. Der Prozess kann unabhängig davon, ob Pfad 1 oder 2 gewählt wurde, durchgeführt werden. Wenn wir die bedingte Verzweigung zwischen Knoten `B` und `F` durch eine parallele Verzweigung ersetzen, ist der Prozess, in jedem der beiden oben genannten Fälle, gültig.

Im oben genannten Beispiel ist der Prozess für den Anwender noch gut überschaubar. Allerdings sind die Probleme, die durch zu versorgende Events entstehen können, bei zunehmenden komplexeren Prozessen nur schwer zu erkennen. Dazu stellen wir im Folgenden einen automatisierten Mechanismus vor, mit dem durch Events ausgelöste Verklemmungen erkannt werden können.

6.2 Eventkanten und Eventkorrektheitsbedingung für Knoten

Bevor ein konkreter Mechanismus zur Erkennung von Verklemmung aufgrund von Events vorgestellt werden kann, müssen einige Formalismen erarbeitet werden. Zunächst muss der Eventfluss näher charakterisiert werden. Dazu führen wir die Eventkanten ein. Als Zweites benötigen wir einen Formalismus, der beschreibt, welche konkreten Events ein Knoten benötigt, damit dieser versorgt ist. Dabei benötigen wir auch eine Aussage darüber, ob zur Laufzeit der Knoten noch versorgt werden kann oder nicht. Dazu führen wir die Eventkorrektheitsbedingung für Knoten ein.

6.2.1 Eventkanten

Bisher wurde in dieser Arbeit eine gestrichelte Kante, zwischen zwei Aktivitäten, gezeichnet, um den Eventfluss anzudeuten. Dieser Eventfluss hat Ähnlichkeiten mit dem Datenfluss in Prozessen (vergleiche hierzu Abschnitt 2.1.2). Im Gegensatz zu Datenflusskanten werden die *Eventkanten* allerdings nicht explizit gesetzt. Sie ergeben sich stattdessen implizit, da die Zustellung von Events automatisch vom EPN durchgeführt wird. Wir diskutieren nun, wie diese „impliziten Eventkanten“ gebildet werden und welche Konsequenzen sie für die Korrektheit haben.

Ein Prozess besteht (in ADEPT) aus einer Menge von Knoten, die mit Aktivitäten belegt sind. Einige dieser Knoten sind wiederum mit Aktivitäten belegt, die prozessintern mit Event versorgt werden müssen. Auf der anderen Seite gibt es Knoten, die mit Aktivitäten belegt sind, die Events produzieren, welche die Versorgung sicherstellen können. Daraus ergibt sich, dass es zu jedem prozessintern zu versorgenden Event, eine Menge von produzierenden Knoten gibt. Dabei werden nur produzierende Knoten beachtet, die vor oder während des konsumierenden Knotens ausgeführt werden können. Sollte die Menge der produzierenden Knoten, für ein zu versorgendes Event, leer sein, so ist eine korrekte Ausführung des Prozesses nicht möglich. Dazu definieren wir für jedes zu versorgende Event, eine Menge von Eventkanten, zwischen einem möglichen Produzenten und dem Konsumenten. Eine Eventkante ist ein Tupel (n_p, n_c, ET), wobei n_p der produzierende Knoten, n_c der konsumierende Knoten und ET der *Event Type* ist.

Beispiel 6.2 (Eventkanten)
Als Beispiel für Eventkanten betrachten wir Abbildung 6.1. Wir gehen davon aus, dass

Aktivität G ein Event vom Typ x obligat konsumiert und das die Aktivitäten C und D dieses Event obligat produzieren können. Beide sind keine Nachfolgerknoten von G und kommen daher für Eventkanten infrage. Daraus ergibt sich für e1 das Tupel (C, G, x) und für e2 das Tupel (D, G, x).

Damit eine Aussage über die Ausführbarkeit des Zielknotens getroffen werden kann, benötigen wir eine Kantenmarkierung für die Eventkanten. Diese verhalten sich analog zu denen von Kontrollflusskanten:

- TRUE_SIGNALED: Der Quellknoten wurde auf COMPLETED gesetzt, alle obligaten Events wurden produziert.
- FALSE_SIGNALED: Der Quellknoten wurde auf SKIPPED gesetzt, da die Quellaktivität nicht mehr ausgeführt werden kann, kann auch nicht mehr das entsprechende Event produziert werden.
- NOT_SIGNALED/UNKNOWN: Die Quellaktivität wurde noch nicht abgeschlossen oder gestartet, kann aber noch ausgeführt werden.

6.2.2 Eventkorrektheitsbedingung für Knoten

Wir haben zuvor die Eventkanten und ihre Markierungen behandelt. Nun diskutieren wir die Konsequenzen für konsumierende Aktivitäten. Dabei ist es wichtig zu beachten, dass ein Knoten, bei dem ein oder mehrere eingehende Eventkanten mit FALSE bewertet wurde, nicht automatisch nicht mehr abgeschlossen werden kann. Stattdessen müssen alle eingehenden Kanten betrachtet werden, um eine Aussage darüber treffen zu können, ob eine Aktivität noch abgeschlossen werden kann. Dazu führen wir die *Eventkorrektheitsbedingung* für konsumierende Knoten ein (ECC(x): *Event Correctness Condition* zu einem Knoten x).

Wir gehen dabei zur Vereinfachung aus, dass zwar alle *Event Types* versorgt werden müssen, aber pro *Event Type* nur jeweils ein entsprechendes Event benötigt wird. Daraus folgt: Gibt es zu einem *Event Type* mehrere Eventkanten, mit gleichem Zielknoten, muss mindestens ein Quellknoten ausgeführt werden, damit die Versorgung sichergestellt ist. Dies ist wichtig, da bei einem Prozessdurchgang nicht immer alle Knoten ausgeführt werden. Daraus können für jeden konsumierenden Knoten Boolsche Ausdrücke gebildet werden, die beschreiben, welche Eventkanten benötigt

werden. Diese Ausdrücke nennen wir im Folgenden die Eventkorrektheitsbedingung zu einem Knoten.

Die Eventkorrektheitsbedingung für einen Knoten mit eingehenden Eventkanten bildet sich wie folgt. Alle eingehenden Eventkanten mit gleichem *Event Type* werden für jeden *Event Type* jeweils zu einer Disjunktion zusammengefasst. Die einzelnen Disjunktionen werden danach zu einer Konjunktion zusammengefasst.

Beispiel 6.3 (Eventkorrektheitsbedingung)
In Abbildung 6.2 ist ein Ausschnitt aus einem Prozess dargestellt. Aktivität `H` konsumiert obligat Events vom Typ `x` und `y`. Die Aktivitäten `D` und `E` produzieren Events vom Typ `x` (Eventkanten `e2` und `e3`) und die Aktivität `B` produziert Events vom Typ `y` (Eventkante `e1`). Daraus ergibt sich für Knoten `H` die folgende Eventkorrektheitsbedingung:

$$\text{ECC(H)} = \texttt{e1}\ \textbf{AND}\ (\texttt{e2}\ \textbf{OR}\ \texttt{e3})$$

Daraus folgt, dass der Quellknoten von `e1` immer ausgeführt werden muss. Dagegen reicht es, wenn entweder der Quellknoten von `e2` oder `e3` ausgeführt wird.

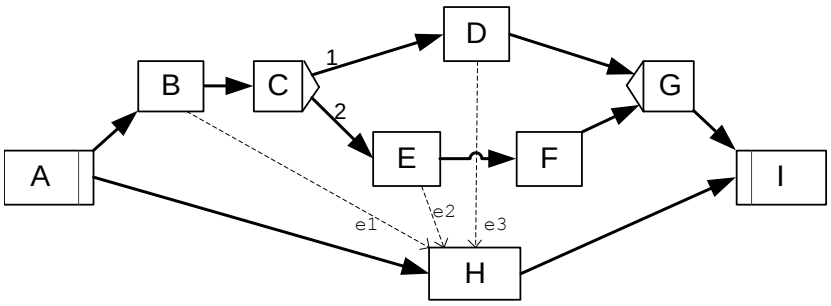

Abbildung 6.2: Beispiel für Eventkanten

In Beispiel 6.3 ist noch leicht zu erkennen, dass der Prozess unabhängig vom Prozessdurchlauf immer funktioniert. In größeren Prozessen wird dies immer schwieriger. Daher diskutieren wir im nächsten Abschnitt, wie eine automatische Erkennung aussehen kann.

Mit der Eventkorrektheitsbedingung für Knoten kann zur Laufzeit geprüft werden, ob ein Knoten noch abgeschlossen werden kann. Da konsumierende Knoten und ihre Nachfolgerknoten auch Events produzieren können, ist es vom Vorteil, wenn diese so früh wie möglich abgeschlossen sind. Für die Eventkorrektheitsbedingung sind nicht in jedem Fall alle eingehenden Events relevant. In Beispiel 6.3 reicht es z. B. aus, wenn

die Eventkante e1 mit FALSE belegt ist, damit die Eventkorrektheitsbedingung von Knoten H nicht mehr erfüllbar ist. Dementsprechend reicht es, wenn e1 und e2 mit TRUE belegt sind, damit die Eventkorrektheitsbedingung erfüllt ist. Dies ist in diesem Fall unabhängig von der Belegung von Eventkante e3.

Damit die Eventkorrektheitsbedingung möglichst früh ausgewertet werden kann, nutzen wir die Dreiwertige Logik (siehe Tabelle 6.1). Ist die Eventkorrektheitsbedingung mit TRUE bewertet, kann der Knoten abgeschlossen werden. Bei FALSE ist eine Beendigung der belegten Aktivität nicht möglich, es liegt eine Verklemmung vor. Sobald der Knoten aktiviert ist, würde dieser unendlich lang warten. Hat der Zielknoten die Markierung SKIPPED ist dies jedoch kein Problem, da dessen Aktivität keine Events mehr konsumiert. Als Letztes kann die Eventkorrektheitsbedingung auch den Wert UNKNOWN besitzen. Damit ist zum gegenwärtigen Zeitpunkt nicht geklärt, ob die Aktivität abgeschlossen werden kann. Dies ist z. B. im Beispiel 6.3 der Fall, wenn Knoten B abgeschlossen ist, aber keiner der beiden Knoten D oder E gestartet wurde. Eventkante e1 hat den Wert TRUE. Die Eventkaten e2 und e3 haben beide den Wert UNKNOWN. Damit ist die Eventkorrektheitsbedingung für Knoten H auch UNKNOWN.

A	B	AND	OR
FALSE	FALSE	FALSE	FALSE
FALSE	UNKNOWN	FALSE	UNKNOWN
FALSE	TRUE	FALSE	TRUE
UNKNOWN	FALSE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE
TRUE	TRUE	TRUE	TRUE

Tabelle 6.1: Dreiwertige Logik für AND und OR

6.3 Korrektheitsanalyse für Events

Im vorherigen Abschnitt haben wir das Prinzip der Eventkanten und die Eventkorrektheitsbedingung für Knoten eingeführt. In diesem Abschnitt stellen wir einen

Algorithmus zu Prüfung der Korrektheit eines Prozesses in Bezug auf Events vor. Dieser Algorithmus basiert auf einer Ermittlung der Erreichbarkeitsmenge[11]. Wir gehen dabei davon aus, dass der Prozess ohne Betrachtung der Events korrekt ist (siehe Abschnitt 2.1.4). Des Weiteren gehen wir davon aus, dass eine konsumierende Aktivität pro *Event Type* nur ein entsprechendes Event benötigt, unabhängig vom Inhalt des Events.

6.3.1 Vorbedingung

Wir haben bereits diskutiert, dass ein Knoten, mit einer konsumierenden Aktivität, eine Eventkorrektheitsbedingung mit `TRUE` benötigt, um auf `COMPLETED` gesetzt werden zu können. Allerdings kann auch eine laufende Aktivität Events produzieren, lange bevor sie abgeschlossen ist. Wenn wir davon ausgehen, dass ein Event nur als produziert gilt, sobald die produzierende Aktivität abgeschlossen ist, wäre dies eine sehr große Einschränkung. Sollten zwei oder mehr parallel laufende Aktivitäten in einem Prozess per Events bidirektional kommunizieren, wäre dies z. B. so nicht zulässig. Aufgrund der Eventkanten würde eine Verklemmung entstehen. Betrachten wir dazu als Beispiel Abbildung 6.3. Auf der linken Seite ist ein Prozessausschnitt dargestellt, bei dem zwei Eventkanten scheinbar eine Verklemmung erzeugen. Auf der rechten Seite sind Ausschnitte aus den beiden Subprozessen dargestellt. Dort ist ersichtlich, dass es niemals zu einer Verklemmung kommen kann.

Um bei der Analyse auch solche Prozesse wie in Abbildung 6.3 zu erlauben, lockern wir die Einschränkung für Eventkanten. Wir erlauben, dass Eventkanten mit `TRUE` bewertet werden können, während die produzierende Aktivität noch läuft, aber nicht im Zustand `COMPLETED` ist. Auf diese Weise können zwei parallel laufende Aktivitäten sich gegenseitig Events schicken und die Verklemmung aufgelöst werden.

Diese Lockerung der Bedingung hat allerdings zur Folge, dass die vorzeitig mit `TRUE` bewerteten Eventkanten im realen Prozessdurchlauf zu Verklemmungen führen können. Dies passiert dann, wenn zwei oder mehrere Events erfordern, dass das jeweils andere Event zuerst produziert wird. Betrachten wir dazu Abbildung 6.4. Auch hier ist links ein Prozess mit zwei parallel laufenden Subprozessen dargestellt. Bei Betrachtung der rechten Seite kann man erkennen, dass Aktivität `A` abgeschlossen sein muss, damit die Aktivität `B` das Event `e2` produziert kann. Allerdings muss vorher die

[11] basierend auf dem Algorithmus aus [Rei00] Seite 109

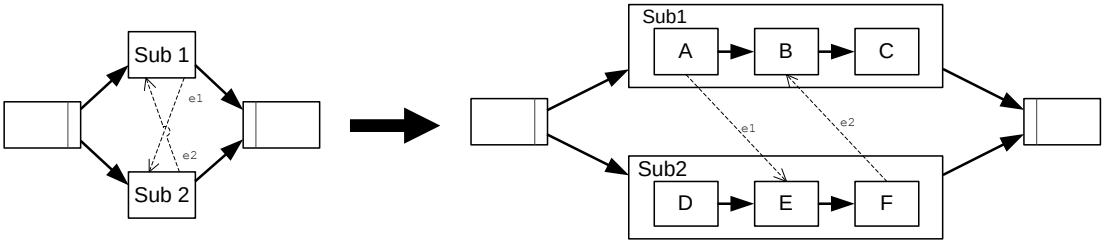

Abbildung 6.3: Beispiel für Eventkanten mit scheinbarer Verklemmung

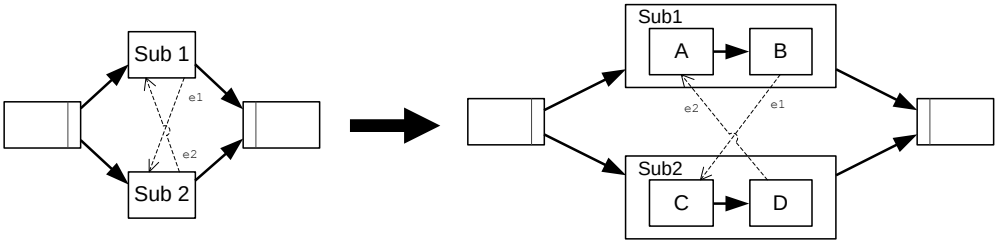

Abbildung 6.4: Beispiel für Eventkanten mit Verklemmung durch Subprozesse

Aktivität `D` das Event `e1` produzieren. Die Aktivität `D` ist aber von der Aktivität `C` und damit von dem Event `e1` abhängig. Daraus folgt, dass niemals eine der Aktivitäten `A` bis `D` beendet werden kann. Nur bei Betrachtung der Subprozesse ist die Verklemmung sichtbar.

Aus der obigen Betrachtung ergibt sich, dass nicht in allen Fällen eine eindeutige Aussage über die Korrektheit eines Prozess gemacht werden kann. Daher ergeben sich folgende drei Aussagen über einen Prozess:

1. Der Prozess hat keine Verklemmungen aufgrund von Events.

2. Der Prozess hat mindestens eine Verklemmung aufgrund von Events. Diese Verklemmung kann sowohl in jedem Prozessdurchlauf auftreten als auch nur in bestimmten Pfaden von bedingten Verzweigungen.

3. Der Prozess kann zwar ohne Verklemmung ausgeführt werden. Aber bestimmte Ausführungsfolgen der Eventkanten können Verklemmungen erzeugen. Die kritischen Eventkanten werden ausgegeben.

Ziel des Algorithmus ist es, nicht nur die sicheren Fälle zu bestätigen, sondern auch auf Eventkanten hinzuweisen, die zu Problemen führen können. Danach ist es

dem Benutzer überlassen, mit der entsprechenden Warnung den Prozess zu prüfen. In beiden Prozessen von Abbildung 6.3 und 6.4 werden die Eventkanten `e1` und `e2` als problematisch eingestuft. Allerdings liegt nur im Prozess von Abbildung 6.4 eine tatsächliche Verklemmung vor.

6.3.2 Algorithmus zur Korrektheit eines Prozesses

Im vorherigen Abschnitt haben wir die Vorbedingung und Einschränkungen der Korrektheitsanalyse diskutiert. Nun stellen wir den eigentlichen Algorithmus vor. Dieser basiert auf einer Ermittlung der Erreichbarkeitsmenge. Da allerdings die Bedingung für die Bewertung von Eventkanten eingeschränkt ist, kann es zu *Problemkanten* kommen. Diese können unter bestimmten Umständen Verklemmungen erzeugen (siehe oben). Daher wird zunächst versucht die Erreichbarkeitsanalyse durchzuführen und nur Eventkanten mit `TRUE` markiert, wenn die Quellknoten auf `COMPLETED` gesetzt wurden. Wird ein Knoten auf `SKIPPED` gesetzt werden sofort alle ausgehenden Eventkanten mit `FALSE` bewertet. Kann in einer Situation kein weiterer aktivierter Knoten auf `COMPLETED` gesetzt werden, so wird mit einer zweiten Funktion versucht die Verklemmung aufzulösen. Dabei wird geprüft, ob es Eventkanten gibt, deren Quell- und Zielknoten aktiviert sind. Aus dieser Menge von Eventkanten werden Teilmengen ermittelt, für die gilt: Sind alle Eventkanten mit `TRUE` bewertet, ist die Eventkorrektheitsbedingung eines aktivierten Zielknotens erfüllt. Damit kann der jeweilige Zielknoten beendet werden. Mit diesen geänderten Eventkanten wird die Erreichbarkeitsanalyse neu gestartet. Dies geschieht solange, bis es keine weitere Schaltung mehr gibt.

Bevor wir die beiden Algorithmen behandeln, betrachten wir zunächst die notwendigen Variablen:

- `N`: Menge der Knoten
- `NS(n)`: Status eines Knotens n ∈ N
- `E`: Menge der Kontrollflusskanten
- `ES(e)`: Status einer Kante e ∈ E
- `EE`: Menge der Eventkanten

- `EES(ee)`: Status einer Eventkante ee ∈ EE
- M_0: Startmarkierung, nur der Start-Knoten ist auf `ACTIVATED` gesetzt.

Wir beginnen nun mit der Ermittlung der Erreichbarkeitsmenge (siehe Algorithmus 5). Zur Vereinfachung gibt es für einen Knoten nur vier mögliche Markierungen: `ACTIVATED`, `COMPLETED`, `SKIPPED` und `NOT_ACTIVATED`. Eine gültige Endmarkierung liegt vor, wenn der End-Knoten `COMPLETED` ist und alle übrigen Knoten entweder die Markierung `COMPLETED` oder `SKIPPED` haben.

Algorithmus 5 Ermittlung der Erreichbarkeitsmenge

```
function analyse(M0)
    Erreicht:= {M0}; Erledigt:= ∅;
    Eventabhängig:= Menge der Knoten, die von eingehenden Events abhängig sind
    Problemkanten:= ∅;
5:  while Erreicht ≠ Erledigt do
        wähle ein M = (NS, ES, EES) ∈ Erreicht - Erledigt;
        Aktiviert:= {n ∈ N | NS(n) = ACTIVATED}; // Menge aller unter M aktivierten Knoten
        if Aktiviert = ∅ AND M ist keine gültige Endmarkierung then
            return (FALSE, ∅);
10:     end if
        // Entferne alle Knoten aus Aktiviert, die nicht auf COMPLETED gesetzt werden können
        for all x ∈ Aktiviert ∩ Eventabhängig do
            if ECC(x) = FALSE then
                return (FALSE, ∅);
15:         else
                if ECC(x) = UNKNOWN then // siehe Dreiwertige Logik in Tabelle 6.1
                    Aktiviert:= Aktiviert - {x};
                end if
            end if
20:     end for
        if Aktiviert ≠ ∅ then
            // Bestimmung aller unmittelbaren Folgemarkierungen, die bei Schalten
            // von x resultieren
            for all x ∈ Aktiviert do
25:             Erreicht = Erreicht ∪ {Folgemarkierungen nach Beendigung von Knoten x,
                    siehe [Rei00] S.109}
            end for
```

```
        else
            // Prüfen ob alle aktivierten Aktivitäten einen UNKNOWN Status haben
30:         (solve, NeueProblemkanten):= solveDeadlock(M, Problemkanten);
            if solve = FALSE then
                return (FALSE, ∅);
            else
                Problemkanten:= Problemkanten ∪ NeueProblemkanten;
35:         end if
        end if
        Erledig:= Erledigt ∪ {M}
    end while
    return (TRUE, Problemkanten);
40: end function
```

Erläuterungen zu Algorithmus 5 `analyse`:

- Zeile 8 - 10: Prüfung, ob vorzeitig abgebrochen werden kann, weil keine Schaltung mehr möglich ist und kein gültiger Endzustand erreicht ist. In diesem Fall ist die Menge der kritischen Eventkanten irrelevant.

- Zeile 12 - 20: Prüfen, ob es einen aktivierten Knoten gibt, der Aufgrund der Eventkorrektheitsbedingung nie mehr abgeschlossen werden kann. In diesem Fall wird die Bestimmung der Erreichbarkeitsanalyse abgebrochen (Zeile 14). Sollte der Knoten nur im aktuellen Zustand nicht abgeschlossen werden können, wird dieser für die Bestimmung der Folgemarkierung entfernt (Zeile 17).

- Zeile 24 - 27: Ermittlung aller derzeit möglichen Folgemarkierungen.

- Zeile 28 - 36: Wenn kein Knoten in der aktuellen Markierung mehr in der Menge `Aktiviert` sind, dann wurden zuvor alle entfernt (Zeile 12 - 20) und die aktuelle Markierung ist keine gültige Endmarkierung. Deswegen muss versucht werden, die Verklemmung aufzulösen (Algorithmus 6) und die Problemkanten zu bestimmen.

- Zeile 39: Der Algorithmus wurde erfolgreich abgeschlossen. Falls kritische Eventkanten vorhanden sind, werden

Als Zweites stellen wir die `solveDeadlock` Funktion vor (siehe Algorithmus 6). Diese hat die Aufgabe, wie oben beschrieben, Eventkanten zu finden, welche eine Verklemmung auflösen können.

Algorithmus 6 Auflösen Verklemmung

```
    function solveDeadlock(M, Problemkanten)
        Aktiviert:= {n ∈ N | NS(n) = ACTIVATED};
        // Menge der EventBeziehungen, die unbewertet sind und zwischen zwei aktivierten
        // Aktivitäten liegen
 5:     EventTry:= {e ∈ EE | EES(e) = UNKNOWN AND e.Quelle ∈ Aktiviert
            AND e.Ziel ∈ Aktiviert}
        EventMenge:= {EM ∈ P(EventTry)[12] | für die gilt, es existiert genau ein x ∈ Aktiviert
            welches auf COMPLETED setzbar ist, wenn alle e ∈ EM von UNKNOWN auf TRUE
            gesetzt sind}
10:     while EventMenge ≠ ∅ do
            wähle ein EM ∈ EventMenge;
            M_new:=M;
            setze alle EES ∈ EventMenge von M_new auf TRUE;
            (possible, NeueProblemkanten) := analyse(M_new);
15:         if possible ≠ TRUE then
                EventMenge:= EventMenge - EM;
            else
                Problemkanten:= Problemkanten ∪ EM ∪ NeueProblemkanten;
                return (TRUE, Problemkanten);
20:         end if
        end while
        return FALSE;
    end function
```

Erläuterungen zu Algorithmus 6 `solveDeadlock`:

- Zeile 7: Prüfen, welche Eventkanten die Verklemmung auflösen können.
- Zeile 10 - 21: Erstellt einen neuen Zustand, bei dem die Eventkanten so auf `TRUE` gesetzt wurden, sodass ein neuer aktivierter Knoten auf `COMPLETED` gesetzt werden kann.

[12] Potenzmenge von EventTry

- Zeile 22: Sollte keine weiteren Kombination mehr vorhanden sein, mit der ein Knoten auf `COMPLETED` gesetzt werden kann, so konnte die Verklemmung nicht aufgelöst werden.

Beide Funktionen (`analyse` und `solveDeadlock`) rufen sich gegenseitig auf. Am Ende liefert die ursprünglich aufgerufene `analyse` Funktion eine Antwort. Das Ergebnis gibt an, ob der Prozess korrekt ist oder nicht (`TRUE` oder `FALSE`). Zusätzlich gibt der Algorithmus noch die Menge der Problemkanten an. Diese kann auch leer sein. Zur Verdeutlichung des Algorithmus betrachten wir ein Beispiel:

Beispiel 6.4 (Korrektheitsalgorithmus)
Gegeben sei der Prozessausschnitt von Abbildung 6.5. Aktivität `C` konsumiert Events vom Typ `x`, die Aktivitäten `D` und `E` produzieren Events vom Typ `y` und Aktivität `G` produziert Events vom Typ `x` und konsumiert Events vom Typ `y`. Daraus ergeben sich folgende Eventkanten: `e1` = (`G`, `C`, `x`), `e2` = (`D`, `G`, `y`) und `e3` = (`E`, `G`, `y`). Des Weiteren ergeben sich somit folgende Eventkorrektheitsbedingung: ECC(`G`) = `e2` **OR** `e3` und ECC(`C`) = `e1`. Bei Anwendung von Algorithmus 5 und 6 ergeben sich die Markierungen und Schaltungen von Tabelle 6.2. Bei Markierung M_2 liegt der Fall vor, dass kein Knoten von `ACTIVATED` auf `COMPLETED` geschallten werden kann. Das liegt daran, dass bei allen aktivierten Knoten die Eventkorrektheitsbedingung weder mit `TRUE` noch mit `FALSE` bewertet werden kann, da alle Eventkanten einen `UNKNOWN` Status haben. Deshalb wird `solveDeadlock` aufgerufen. Diese erzeugt die Markierung M_3, bei der die Eventkante `e1` auf `TRUE` gesetzt wurde. Mit dieser Markierung kann die Ermittlung der Erreichbarkeitsmenge fortgeführt werden. Daraus ergeben sich die Folgemarkierungen M_4 bis M_{15}. Als Endergebnis stellt sich heraus, dass der Prozess korrekt ist. Allerdings gibt es eine Warnung bezüglich der Eventkante `e1`, da vorausgesetzt wird, dass dieses Event ausgelöst wird, bevor die produzierende Aktivität `G` das Event von Typ `y` erwartet.

Wenn wir das Beispiel so abwandeln, dass die Aktivitäten `D` und `E` Events von jeweils unterschiedlichen Typen produzieren und Knoten `G` auch beide benötigt, so ergibt sich die Eventkorrektheitsbedingung ECC(G) = `e2` **AND** `e3`. Damit entsteht bei Ausführung der Algorithmen bei Markierung M_4 eine Verklemmung, da die Eventkorrektheitsbedingung von Knoten `G`, bei dieser Markierung, den Wert `FALSE` hat.

Der hier entwickelte Algorithmus kann, wie das kleine Beispiel schon andeutet, einen hohen Aufwand haben. Insbesondere die `solveDeadlock` Funktion kann eine hohe Anzahl von Kombinationen erzeugen, wenn viele Eventkanten vorliegen. Der Algorithmus lässt sich allerdings noch weiter optimieren. In [Rei00] werden dazu Ansätze beschrieben, die auch hier angewandt werden können. Zum einen können nicht

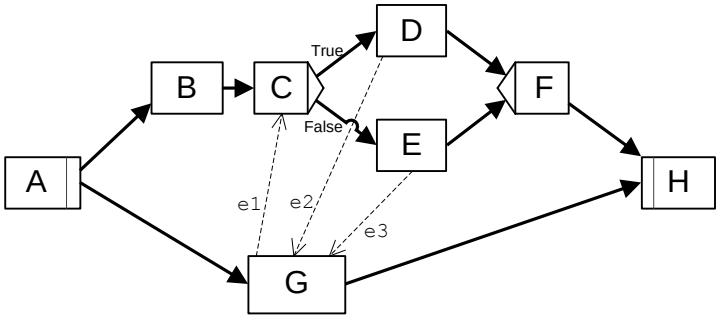

Abbildung 6.5: Beispielprozessausschnitt für Korrektheitsanalyse

relevante Knoten und Blöcke entfernt werden. In Beispiel 6.4 ist dies z. B. der Knoten `B`. Dieser hat keine Relevanz auf die Korrektheitsanalyse. Des Weiteren kann die Analyse auf Blöcke eingeschränkt werden, welche für die Korrektheitsanalyse relevant sind. Dies ist in Beispiel 6.4 in sofern schon geschehen, da nur ein Prozessausschnitt und nicht der gesamte Prozess betrachtet wurde.

Zusätzlich ist es auch in einigen Fällen möglich, mehrere Eventkanten zusammen zu fassen. Wenn mehrere Eventkanten den gleichen Quell- und Zielknoten haben, dann haben sie in der Korrektheitsanlayse immer die gleiche Kantenmarkierung. Durch die Reduktion der Eventkanten, entstehen bei der `solveDeadlock` Funktion, in bestimmten Fällen, weniger zu prüfende Kombinationen.

6.4 Zusammenfassung

Wir haben in diesem Kapitel die Korrektheit von Prozessen in Bezug auf Events behandelt. Dabei haben wir zunächst die Problemfälle charakterisiert und die Eventkanten und Eventkorrektheitsbedingung für Knoten eingeführt. Als Lösungsansatz für das Korrektheitsproblem wurde ein Algorithmus zur Analyse vorgestellt. Mithilfe dieses Algorithmus ist es möglich sichere Veklemmungen in einem Prozess, aufgrund von Events, festzustellen. Zusätzlich können Events bestimmt werden, die zu mindestens unter bestimmten Umständen zu Verklemmungen führen können.

Dieser Algorithmus hat allerdings auch mehrere Nachteile. Hierbei ist insbesondere das Problem, der nicht vollständigen Aussagekraft der Korrektheitsanalyse. Die Ermittlung der Eventkanten, die zu Problemen führen können, bedarf vom Benutzer eine manuelle Prüfung. Diese ist Notwendig um Verklemmungen im laufenden Betrieb,

Markierungs-Nummer	**Knotenmarkierungen**								**Event-kanten**			**Schaltung**
	A	B	C	D	E	F	G	H	e1	e2	e3	
M_0	▲	-	-	-	-	-	-	-	-	-	-	$M_0[A\rangle M_1$
M_1	✓	▲	-	-	-	-	▲	-	-	-	-	$M_1[B\rangle M_2$
M_2	✓	✓	▲	-	-	-	▲	-	-	-	-	**solveDeadlock Ergebnis M_3**
M_3	✓	✓	▲	-	-	-	▲	-	✓	-	-	$M_3[C\rangle_{True} M_4$ $M_3[C\rangle_{False} M_5$
M_4	✓	✓	✓	▲	✖	-	▲	-	✓	-	✖	$M_4[D\rangle M_6$
M_5	✓	✓	✓	✖	▲	-	▲	-	✓	✖	-	$M_5[E\rangle M_7$
M_6	✓	✓	✓	✓	✖	▲	▲	-	✓	✓	✖	$M_6[F\rangle M_8$ $M_6[G\rangle M_9$
M_7	✓	✓	✓	✖	✓	▲	▲	-	✓	✖	✓	$M_7[F\rangle M_{10}$ $M_7[G\rangle M_{11}$
M_8	✓	✓	✓	✓	✖	✓	▲	-	✓	✓	✖	$M_8[G\rangle M_{12}$
M_9	✓	✓	✓	✓	✖	▲	✓	-	✓	✓	✖	$M_9[F\rangle M_{12}$
M_{10}	✓	✓	✓	✖	✓	✓	▲	-	✓	✖	✓	$M_{10}[G\rangle M_{13}$
M_{11}	✓	✓	✓	✖	✓	▲	✓	-	✓	✖	✓	$M_{11}[F\rangle M_{13}$
M_{12}	✓	✓	✓	✓	✖	✓	✓	▲	✓	✓	✖	$M_{12}[H\rangle M_{14}$
M_{13}	✓	✓	✓	✖	✓	✓	✓	▲	✓	✖	✓	$M_{13}[H\rangle M_{15}$
M_{14}	✓	✓	✓	✓	✖	✓	✓	✓	✓	✓	✖	**Endmarkierung**
M_{15}	✓	✓	✓	✖	✓	✓	✓	✓	✓	✖	✓	**Endmarkierung**

✓= `COMPLETED` bzw. `TRUE_SGINALED` ✖ = `SKIPPED` bzw. `FALSE_SGINALED`
- = `NOT_ACTIVATED` bzw. `NOT_SGINALED` ▲= `ACTIVATED`

Tabelle 6.2: Erreichbarkeitsanalyse mit Auflösung einer Verklemmung

bei vorhandenen Problemkanten, zu vermeiden. Gerade bei komplexeren Prozessen ist dies nur mit hohem Aufwand möglich. Des Weiteren werden keine Subprozesse betrachtet. Wird viel mit Kommunikation von Subprozessen via Events gearbeitet, kann dies zu einer sehr großen Anzahl zu prüfender Events führen.

Die eben genannten Probleme bieten Spielraum für weitere Arbeiten in diesem Bereich. Aufgrund des Umfanges sprengen sie jedoch den Rahmen dieser Arbeit.

7 Prototypische Implementierung eines Event-Managers

Im Rahmen dieser Arbeit entstand eine prototypische Implementierung eines Event-Managers für BPM-Systeme. Dieser stellt eine Umsetzung der zuvor diskutierten Architektur dar und soll zeigen, dass diese in der Praxis realisierbar ist. Es handelt sich dabei um eine reine *Proof of Concept* Implementierung, die nicht für den produktiven Einsatz gedacht ist.

7.1 Einordnung in AristaFlow

Die prototypische Implementierung des Event-Managers erfolgt als eine Erweiterung der AristaFlow BPM Suite [Ari]. AristaFlow ist ein BPM-System, welches aus dem ADEPT-Projekt an der Universität Ulm entstanden ist [DRRM+09]. Der Prototyp des Event-Managers ist eine in Java geschriebene Komponente, die sich im AristaFlow Server einbinden lässt.

In Abbildung 7.1 ist die Einordnung des Event-Managers in die AristaFlow-Architektur dargestellt. Dabei wird insbesondere dargestellt, welche Komponenten miteinander interagieren. Wir gehen hier nicht auf alle Bereiche der AristaFlow Architektur ein. Stattdessen behandeln wir nur die für den Event-Manager relevanten Komponenten.

Der Event-Manager ist auf der Ebene des *Execution Layer* eingeordnet. Der *Execution Layer* vereint alle funktionalen Komponenten, welche die korrekte Ausführung von Prozess-Instanzen und den damit verbundenen Aktivitäten ermöglichen [RDJ+08]. Hier liegt auch der Execution-Manager. Dieser koordiniert die Ausführung der Prozess-Instanzen. Dazu gehört unter anderem die Auswahl der Folgemarkierung und die Versorgung der Aktivitäten mit Daten. Der Execution-Manager greift auf den Event-Manager zu, damit er diesen über neue Prozess-Instanzen, gestartete

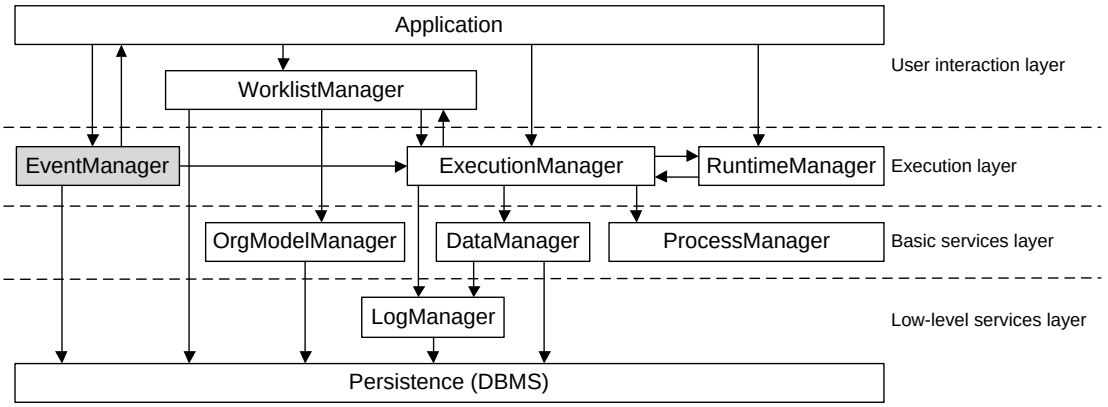

Abbildung 7.1: Einordnung des Event-Managers in die Aristaflow-Architektur (angelehnt an [LKRD10])

Aktivitäten oder veränderte Datenelemente informieren kann. Dies ist notwendig, damit der Event-Manager das EPN aktualisieren bzw. benachrichtigen kann.

In Abbildung 7.1 steht Application stellvertretend für alle Applikationen, die auf den AristaFlow Server zugreifen. Darunter fallen das AristaFlow Control Center und der AristaFlow Client[13]. Im Client werden die einzelnen Aktivitäten ausgeführt. Für Applikationen bietet der Event-Manager verschiedene Schnittstellen an. Damit können Aktivitäten Events an das EPN übertragen. Zusätzlich können Applikationen bestimmte *Event Types* abonnieren. Dazu melden sie sich beim Event-Manager an und werden anschließend mit eintreffenden Events versorgt.

Als Letztes greift der Event-Manager auf die *Persistence* zu, um Daten persistend zu speichern. Normalerweise werden persistierte Daten, wie EPN, *Event Type* Vorlagen und EPE Vorlagen in einer Datenbank abgelegt. In Rahmen der Entwicklung des Prototyps wurde darauf verzichtet. Stattdessen wird alles in Java Objekten gespeichert.

7.2 Ausgewählte Interaktionsszenarien

Um die Funktionsweise des Event-Manager Prototyps zu verdeutlichen, betrachten wir nun ausgewählte Interaktionsszenarien. Hierbei behandeln wir, wie die unterschiedlichen Objekt-Instanzen miteinander interagieren. Die betrachteten Interaktionsszenarien sind: Start und Ende einer Prozess-Instanz, *Publish/Subscribe* für einen

[13] für nähere Informationen siehe [DRRM+09]

Event Type und das Verteilen eines Events. Damit sind, bis auf die Anbindung von externen Systemen, die wichtigsten Anwendungsfälle des Event-Manager abgehandelt.

7.2.1 Starten und Beenden einer Prozess-Instanz

Abbildung 7.2 stellt ein Interaktionsdiagramm für das Starten und Beenden einer Prozess-Instanz dar. Es handelt sich dabei um eine Prozess-Instanz für einen Top-Level-Prozess.

Zu Beginn von Szenario 1 (Starten einer Prozess-Instanz) ruft der Execution-Manager (`ExM`) die `createIEPA` Methode des Event-Managers Event-Manager (`EM`) auf, um diesem mitzuteilen, dass eine neue Prozess-Instanz gestartet wurde. Beim Start einer Prozess-Instanz erzeugt der Event-Manager zunächst jeweils eine IEP, IEC und IEPA Objekt-Instanz. Damit diese Instanzen gestartet werden können, müssen sie sich zunächst mit ihrem jeweiligen SuperEPA verbinden (Methode `connect`). Erst danach können sie vom Event-Manager gestartet werden und Events produzieren bzw. verarbeiten.

Wäre die neue Prozess-Instanz kein Top-Level-Prozess, sondern ein Subprozess, würde sich das Verhalten wie folgt ändern: Zunächst müsste der Event-Manager den IEPA des Vater-Prozesses der neuen Prozess-Instanz ermitteln. Beim Erzeugen der neuen IEPA-Instanz wird dann der entsprechende IEPA als SuperEPA gesetzt. Somit kann der neue IEPA sich mit dem richtigen IEPA verbinden.

In Szenario 2 wird eine Prozess-Instanz eines Top-Level-Prozesses beendet. In diesen Fall muss zuerst der Execution-Manager dem Event-Manager mitteilen, dass die Prozess-Instanz beendet wurde (`stopIEPA` Methode). Danach veranlasst der Event-Manager das Herunterfahren des entsprechenden IEPAs (`shutdown`). Beim Herunterfahren eines IEPAs wird nicht nur der IEPA gestoppt, sondern auch aus dem EPN entfernt. Dazu müssen zunächst alle Kindknoten heruntergefahren sein. Deswegen ruft der IEPA zunächst die `shutdown` Methode seiner Kindknoten auf und trennt abschließend die Verbindung zum MainEPA. Während des Herunterfahrens müssen zunächst für alle entsprechenden *Event Types* die *unpublish* und *unsubscribe* Methoden aufgerufen werden. Dies ist in Abbildung 7.2 zur besseren Übersicht ausgeblendet. Nach dem Herunterfahren endet der Lebenszyklus der Objekt-Instanzen.

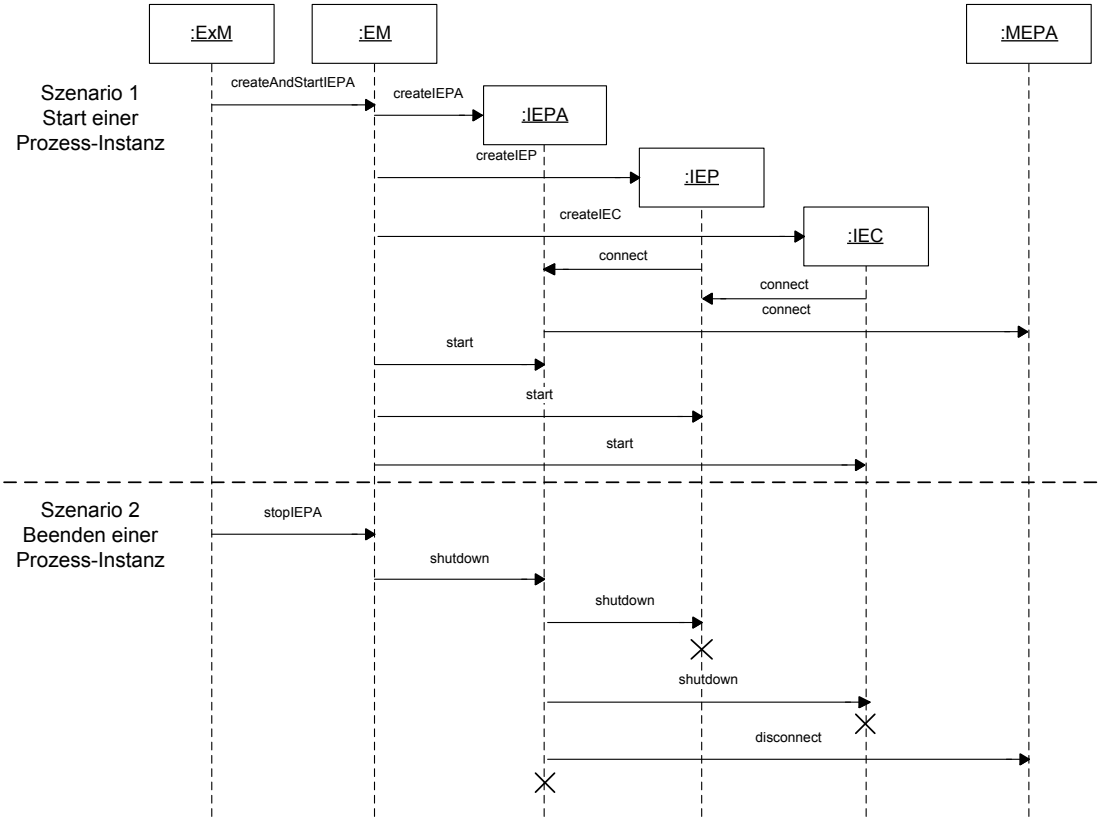

Abbildung 7.2: Interaktionsdiagramm für Start und Ende einer Prozess-Instanz

7.2.2 Publish/Subscribe

Nach dem Start und Ende einer Prozess-Instanz betrachtet wurden, wird nun das *Publish/Subscribe* und Routing vorgestellt. Dazu gehen wir davon aus, dass bereits das EPN aus Abbildung 7.3 vorliegt. Zunächst produziert und konsumiert keiner der hier dargestellten EPEs ein Event.

Abbildung 7.4 zeigt ein Interaktionsdiagramm mit vier Szenarien. Im ersten Szenario kündigt EP 1 den *Event Type* `x` an. Dazu ruft dieser die `publish` Methode bei seinem SuperEPA (EPA 1) auf. Dieser leitet das publish an den MainEPA weiter. Da es bisher keine Konsumenten für diesen *Event Type* gib, ist der Vorgang damit beendet.

Im zweiten Szenario abonniert EC 2 Events vom Typ `x`. Dazu ruft dieser die `subscribe` Methode beim MainEPA auf. Von dort aus wird die Abonnierung über EPA 1 an EP 1 weitergereicht.

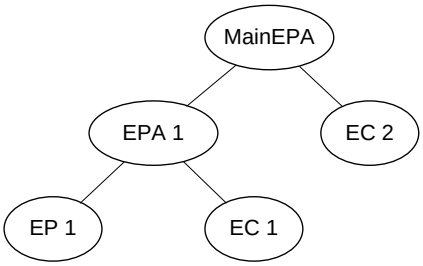

Abbildung 7.3: Beispiel-EPN für Interaktionsdiagramm

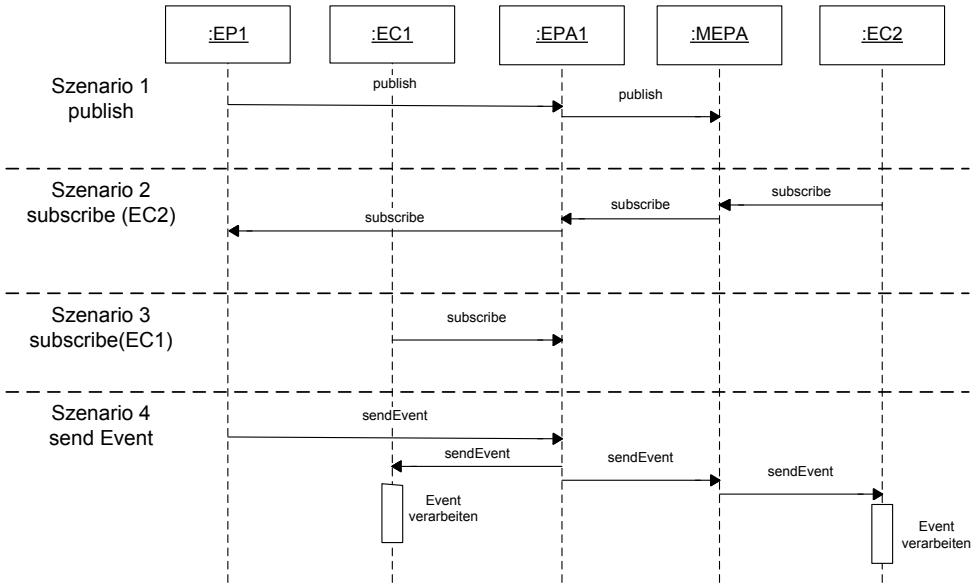

Abbildung 7.4: Interaktionsdiagramm für Publish/Subscribe und Routing

Im dritten Szenario abonniert nun auch EC 1 `x` Events. Dazu wird die `subscribe` Methode beim SuperEPA (EPA 1) aufgerufen. Da EPA 1 bereits Events vom Typ `x` bei EP 1 abonniert hat, ist der Vorgang beendet.

Im letzten Szenario produziert EP 1 ein Event vom Typ `x`. Dieses übermittelt EP 1 an EPA 1. EPA 1 hat zwei Abonnenten für diesen *Event Type*, MainEPA und EC 1, an welche EPA 1 das Event nun verschickt. EC 1 kann nun das empfangene Event direkt verarbeiten, während der MainEPA das Event an EC 2 weiterleitet, damit dieser das Event auch verarbeiten kann.

7.3 Fazit

Die prototypische Implementierung eines Event-Managers hat gezeigt, dass eine Umsetzung der hier vorgestellten EPN-Architektur realisierbar ist. Dabei wurde besonderen Wert auf folgende Eigenschaften gelegt. Zum einen die dynamische Generierung des EPNs, welches von den Starten und Beenden von Prozess-Instanzen und externer Komponenten beeinflusst wird. Zum anderen die automatische Verteilung der Events. Hierbei ist besonders die Umsetzung des *Publish/Subscribe*-Mechanismus hervorzuheben. Zusätzlich ist es möglich den Event-Manager mit weiteren EPEs und CEP-Modulen zu erweitern. Entsprechende Interfaces sind vorhanden.

Allerdings gibt es beim Prototyp auch einige Einschränkungen, die für einen produktiven Einsatz behoben werden müssen. Zunächst gibt es keine Unterstützung bei der Eventverarbeitung, wie z. B. das Erkennen von Mustern. Entsprechende Funktionen müssen bei Bedarf manuell implementiert werden.

Ein weiterer wichtiger Nachteil ist das fehlen einer Datenbankanbindung. Das gesamte EPN und dessen Erweiterungen werden in Java-Objekten verwaltet. Bei einem Neustart des BPM-Systems ist das EPN nicht mehr vorhanden.

Der letzte wichtige Aspekt ist die eingeschränkte Erweiterung des Prozessmodells. Es können lediglich bei Knoten einer Prozessvorlage die produzierten und konsumierten Events angegeben werden. Es kann nicht konfiguriert werden, ob ein *Event Type* direkt nach Start der Prozess-Instanz oder erst nach Start der jeweiligen Aktivität abonniert bzw. angekündigt wird. Stattdessen werden alle *Event Types* direkt nach Start der Prozess-Instanz abonniert bzw. angekündigt. Auch eine explizite prozessinterne Versorgung wird nicht unterstützt.

8 Verwandte Arbeiten

In diesem Kapitel diskutieren wir zwei Themen, die mit den bisherigen Themen verwandt sind. Als Erstes behandeln wir die Ereignisanfragesprachen (Abschnitt 8.1). Diese bieten Techniken, um auf eine Menge von Events zuzugreifen und in diesen Muster zu erkennen, dies ist ein Aspekt, der in der bisherigen Arbeit nur angedeutet wurde. Dann vergleichen wir die beiden Geschäftsprozesssprachen BPEL und BPMN mit dem in dieser Arbeit behandelten Event-Mechanismus (Abschnitt 8.2).

8.1 Ereignisanfragen

Bisher haben wir in dieser Arbeit meist nur betrachtet, dass Aktivitäten oder Systeme Events konsumieren. Dabei haben wir nicht behandelt, wie diese Events konkret weiterverarbeitet werden, um damit Funktionen bzw. Prozesse auszuführen oder neue *Complex Events* zu produzieren. Um einen kurzen Überblick über das Event Processing in der Praxis zu bekommen, betrachten wir nun im Folgenden verschiedene Ereignisanfragesprachen.

Die Anfrage von Events verhält sich ähnlich wie Datenbankanfragen. Allerdings müssen Events kontinuierlich ausgewertet werden, während sie auftreten [EB09]. Datenbanken beinhalten zwar oft auch ereignisbezogene Daten, wie z. B. Aufzeichnungen von Bestellungen. Allerdings beziehen sich Datenbankanfragen immer auf den aktuellen Zustand. Des Weiteren haben Datenbanken nur eine endliche Menge von Daten, während CEP-Systeme es grundsätzlich mit einem prinzipiell unbegrenzten Ereignis*strom* zu tun haben. Deshalb wird eine Anfrage auf Events grundsätzlich ebenfalls einen Antwortstrom ergeben (siehe Abbildung 8.1).

Wir betrachten nun im Folgenden drei der gängigsten Ereignisanfragesprachen und diskutieren zum Abschluss ihre Integration in die bisher entwickelten Event Processing System.

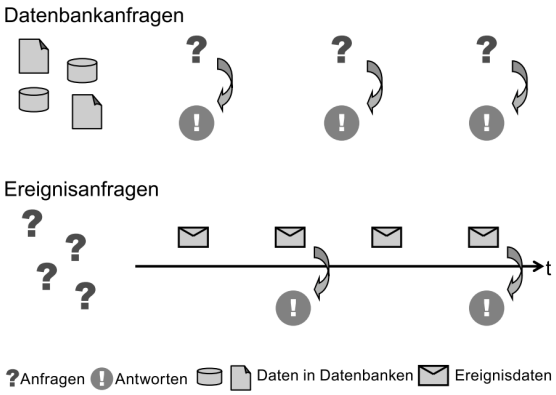

Abbildung 8.1: Unterschied zwischen Datenbank- und Ereignisanfragen [EB09]

8.1.1 Kompositionsoperatoren

Kompositionsoperatoren entstammen den *aktiven* Datenbanken [EB09, AE02]. Ein Beispiel für ein System, das Kompisitionsoperatoren verwendet, ist Amit [AE04]. Bei diesem Ansatz werden Anfragen gegen einzelne Events mithilfe verschiedener Operatoren zu *Complex Events* zusammengesetzt. Deswegen werden diese Sprachen häufig auch „Event Algebra“ genannt [Eck08].

Es werden verschiedene Operatoren angeboten, wie z. B. Konjunktion, Disjunktion, Sequenz oder Negation. Diese Operatoren können verschachtelt werden, wodurch komplexere Anfragen möglich sind.

Eine Konjugation von Events sieht z. B. so aus: $A \wedge B$. Dies bedeutet, dass die Bedingung dann erfüllt ist, wenn sowohl ein Event von Typ A als auch vom Typ B aufgetreten sind. Diese Events können zu verschiedenen Zeitpunkten aufgetreten sein, wobei die Reihenfolge irrelevant ist. Eine Sequenz von A und B wird typischerweise als $A;B$ geschrieben. Hier wäre die Bedingung erfüllt, wenn das Event B *nach* A aufgetreten ist. Wenn ausgedrückt werden soll, dass zwischen zwei Events ein bestimmtes Event nicht auftreten darf, kann dies durch eine Negation geschehen. Beispielsweise wird mit $A;\neg B;C$ ausgedrückt, dass die Bedingung erfüllt ist, wenn zwischen dem Auftreten von A und C kein B empfangen wurde.

Es gibt noch weitere Operationen, die sich z. B. mit zeitlichen Aspekten befassen [Eck08], auf die hier nicht weiter eingegangen werden soll. Des Weiteren unterstützen viele Sprachen Zusatzfunktionen, die bei der Erstellung von *Complex Events* hilfreich

sind. Mit Selektion kann z. B. erreicht werden, dass nur das erste Event eines Typs ausgewählt wird.

Mithilfe von Kompositionsoperatoren ist es möglich, kompakt und intuitiv *Complex Events* zu spezifizieren. Sie unterstützen temporale Zusammenhänge und Negation gut [EB09]. Allerdings werden die Daten in den Events oft vernachlässigt.

Es gibt bisher nur wenige Produkte im CEP-Bereich, die auf Kompositionsoperatoren basieren. Hier sei die IBM Active Middleware Technology (Amit) genant.

8.1.2 Produktionsregeln

Bei Produktionsregeln handelt es sich um keine Ereignisanfragesprache im eigentlichen Sinne [EB09]. Hier werden Regeln eng in eine Wirts-Programmiersprache (z. B. Java) eingebettet. Es werden dadurch Aktionen spezifiziert, die ausgeführt werden sollen, wenn bestimmte Zustände (Bedingungen) eingetreten sind. Solche Produktionsregeln werden auch *condition-action rules* genannt, da die Anfragestatements nach dem Prinzip von `WHEN` *`condition`* `DO` *`action`* beschrieben werden [BBB+07]. Diese Systeme sind wegen ihrer inkrementellen Auswertung (z. B. mit Rete [For82]) auch für CEP geeignet. Immer wenn ein Ereignis auftritt, muss ein entsprechendes Faktum erzeugt werden. Dadurch können Ereignisanfragen über diese Fakten ausgedrückt werden [EB09].

Produktionsregeln haben den Vorteil, dass sie sehr flexibel sind. Allerdings muss sehr viel manuell programmiert werden. Der typische Einsatzzweck von Produktionsregeln sind *Business Rule Management Systeme* wie Drools oder ILOG JRules. Diese sind allerdings nicht auf Events fokussiert. Ein CEP-Produkt mit Produktionsregeln ist TIBCO Business Events.

8.1.3 Datenstrom-Anfragesprachen

Bei dem Ansatz der Datenstrom-Anfragesprachen wird von einem kontinuierlichen Strom von Tupeln ausgegangen, die mit Zeitstempeln versehen sind [EB09, ABW04, BW01]. Diese werden in Relationen umgewandelt und auf diesen können nun SQL-Anfragen ausgeführt werden. Die Ergebnisse werden zurück in einen Strom geführt.

Die auf SQL basierenden Datenstrom-Anfragesprachen sind derzeit die im Bereich der Ereignisanfragesprachen erfolgreichsten [EB09]. Der bekannteste Vertreter ist die Continuous Query Langue (CQL), die im Folgenden genauer betrachtet wird.

Aufgrund der Bedeutung von CQL in der Praxis, wird dieses Thema mehr vertieft, als die Kompositionsoperatoren und Produktionsregeln.

Die einzelnen Datenströme bestehen aus einem Strom von Tupeln mit Zeitstempel. Im Kontext von CEP kann also ein Event als ein solches Tupel dargestellt werden. Da die Tupel eines Datenstroms alle den gleichen Aufbau haben müssen, kann ein Datenstrom als ein Strom von Events vom gleichen *Event Type* angesehen werden [EN10].

Beispiel 8.1 (Filter)
Es sollen alle `Order picking Completed` Events ausgegeben werden, welche die `OrderID` mit Wert 42 haben.

```
SELECT *
FROM Order_Picking_Completed
WHERE OrderID = '42'
```

Diese CQL-Query stellt einen einfachen Filter dar. Der Ausdruck ähnelt einem SQL-Ausdruck. Allerdings mit dem Unterschied, dass `Order picking Completed` keine Relation, sondern ein Ereignis-Strom ist. In diesem Fall werden alle eingehenden Events, die vom Typ `Order picking Completed` sind, nach der `OrderID` gefiltert.

Um komplexere Muster zu erkennen, ist es häufig notwendig mehrere Datenströme miteinander zu verknüpfen (*Join*). Eine Verknüpfung zwischen zwei potenziellen Datenströmen ist zwar nicht möglich, dafür lässt sich aber der Umstand ausnutzen, dass häufig nur Events von Interesse sind, die zeitlich nahe beieinander liegen [See10]. Zu diesem Zweck können sogenannte gleitende Zeitfenster genutzt werden.

Beispiel 8.2 (Join)
Es sollen alle `InstanceID`s von Versand-Prozessen ausgegeben werden, die innerhalb von drei Tagen abgeschlossen waren.

```
SELECT Process_Completed.InstanceID
FROM Process_Completed, Process_Started[Range 3 days]
WHERE Process_Completed.InstanceID = Process_Started.InstanceID
      AND Process_Completed.TempleteID = 'Versand'
```

In der hier vorliegenden CQL-Query werden gleitende Zeitfenster genutzt. Somit kann jedes Mal, wenn ein `Process_Completed` Event eintrifft, dieses mit allen `Process_Started` Events der letzten drei Tage verglichen werden, ob die jeweiligen

InstanceIDs identisch sind. Somit wird gleichzeitig geprüft, ob die abgeschlossene Prozess-Instanz auch innerhalb der letzten drei Tage gestartet wurde. Die gleitenden Zeitfenster können auf unterschiedliche Weise genutzt werden. So kann z. B. statt der letzten X Minuten, die letzten X Elemente eines Datenstroms abgefragt werden.

Eine weiterer nützliche Funktion ist der Slide-Operator [ABW06]. Damit ist es möglich, in bestimmten Zeitintervallen eine Anfrage auf einen Strom zu starten.

Beispiel 8.3 (Aggregation mit Slide-Operator)
Es soll jeden Tag die Anzahl der gestarteten Versand-Prozesse ermittelt werden.

```
SELECT Count(*)
FROM Process_Started[Range 1 day Slide 1 day]
WHERE Process_Started.TempleteID = 'Versand'
```

Statt auf jeden eintreffenden Process_Started Event einzeln zu reagieren, wird hier die Anfrage nur einmal am Tag ausgeführt. Ohne den Slide-Operator würde hingegen bei jedem eintreffenden Process_Started Event die Anzahl der gestarteten Versand-Prozesse der letzten 24 Stunden aufs Neue ermittelt.

Datenstrom-Anfragesprachen bieten noch viele weitere Möglichkeiten, wie z. B. die Verschachtlung von CQL-Querys oder den Einsatz von regulären Ausdrücken. Zusätzlich werden noch weitere Operatoren bei dem Überführen von Relationen in einen Datenstrom unterstützt. Diese werden allerdings hier nicht weiter betrachtet.

Der Vorteil von Datenstrom-Anfragesprachen liegt in ihrer guten Unterstützung von Aggregations-Aufgaben und der guten Integration mit Datenbanken. Allerdings sind Negation und temporale Zusammenhänge nur umständlich auszudrücken [EB09].

In diesem Bereich gibt es schon mehrere kommerzielle Produkte wie Oracle CEP, Aleri/Coral8, StreamBase, RTM Realtime Monitoring. Im Open Source Bereich gibt es bisher nur Esper [See10, Esp].

8.1.4 Fazit

In diesem Abschnitt haben wir drei Formen von Ereignisanfragesprachen behandelt. Dabei ist besonders die Sprache CQL, als die verbreitetste im Bereich von CEP gewürdigt worden.

Die beschriebenen Techniken können auch auf die hier vorgestellte EPN-Architektur angewandt werden. So kann ein EPA mit einer Ereignisanfragesprache eingehende

Events filtern oder nach Mustern durchsuchen. Insbesondere mit CQL ergeben sich so interessante Möglichkeiten. In diesem Fall werden die eingehenden Events nach dem *Event Type* sortiert und bilden so die einzelnen Datenströme. Diese werden mit den CQL-Ausdrücken bearbeitet. Das Resultat sind neue Events. Auf ähnliche Weise können auch Kompositionsoperatoren und Produktionsregeln genutzt werden, um die Event Processing Funktionalitäten zu erweitern.

8.2 Vergleich mit BPEL und BPMN

Derzeit befinden sich am Markt einige Geschäftsprozess-Modellierungssprachen, welche bereits mit einem Event-Mechanismus ausgestattet sind. Wir diskutieren nun die Event-Mechanismen von zwei der gängigsten Geschäftsprozess-Modellierungssprachen, die auf dem Markt vorhanden sind: BPEL und BPMN. Dazu betrachten wir in beiden Fällen zunächst die Bedeutung der Events in der jeweiligen Sprache. Danach vergleichen wir diese mit dem in dieser Arbeit entwickelten Event-Mechanismus.

8.2.1 BPEL

Die Web Services Business Process Execution Langue (WS-BPEL oder auch kurz BPEL) ist eine auf XML basierte Sprache [Oas07]. Diese ist als eine Verschmelzung der beiden Sprachen WSFL und XLANG [ACKM04] entstanden und liegt zurzeit in Version 2.0 vor. Mit BPEL können ausführbare Geschäftsprozesse auf Basis von Web Services beschrieben werden. Die ausführbaren Geschäftsprozesse sind selbst wieder Web Services.

In BPEL kann zum einen der Ablauf des Prozesses selbst und zum anderen die Schnittstelle nach außen beschrieben werden. Die Geschäftsprozesse werden dabei in Form von zwei verschiedenen Arten von Aktivitäten beschrieben.

Die Basisaktivitäten stellen elementare Aktivitäten dar. Mit diesen können unter anderem Nachrichten empfangen (`receive`, `pick`) oder verschickt (`invoke`, `reply`) werden. Als Zweites gibt es die Strukturaktivitäten. Damit kann der Ablauf des Prozesses beschrieben werden. Es stehen unter anderem Sequenzen und parallele Abläufe zur Verfügung.

Eine Besonderheit sind Scopes. Mit ihnen kann für die eingeschlossenen Aktivitäten ein Kontext bereitgestellt werden. Dieser kann eigene Variablen, `Compensation Handler`, `Fault Handler` und `Event Handler` beinhalten.

BPEL Event Handler und Message Events

Eine Besonderheit von BPEL ist der *Event Handler*, der in diesem Abschnitt genauer betrachtet wird. Ein *Event Handler* wird innerhalb der `<process>` oder `<scope>` Umgebung definiert. Der *Event Handler* wird ausgelöst, wenn ein definiertes Event eintritt. Dabei werden zwei Arten von Events unterschieden: *Message Events* (`<onEvent>`) und *Alarm Events* (`<onAlarm>`). *Message Events* werden durch eingehende Nachrichten ausgelöst, *Alarm Events* werden nach Ablauf einer definierten Zeitspanne oder zu einem bestimmten Zeitpunkt ausgelöst. Bei jedem ausgelösten Event werden die Aktivitäten eines eingeschlossenen Scope ausgeführt. Betrachten wir dazu als Beispiel folgenden Auszug aus einem *Event Handler* mit einem `<onEvent>` Element.

```
1 <eventHandlers>
2   <onEvent partnerLink="NCName"
3     portType="QName"
4     operation="NCName"
5     messageType="QName"
6     variable="VariableName">
7     <scope ...> ...</scope>
8   </onEvent>
9 </eventHandlers>
```

In Zeilen 2–8 ist ein `<onEvent>` Element definiert. Dieses soll bei einer eingehenden Nachricht, wie sie Zeile 2–5 beschreibt, ausgelöst werden. Dies bedeutet, wenn eine entsprechende Nachricht eingeht, wird das Event ausgelöst und die eingehende Nachricht in die angegebene Variable geschrieben (Zeile 6). Danach wird der eingeschlossene Scope ausgeführt (Zeile 7).

Die besondere Eigenschaft von *Event Handlern* in BPEL ist, dass sie nebenläufig zu den im Scope eingeschlossenen Aktivitäten ablaufen. Dies bedeutet, dass während der Laufzeit eines Scopes auf die definierten Events reagiert wird, ohne die gerade

laufenden Aktivitäten zu unterbrechen. Dabei kann während der Laufzeit des Scopes auch auf mehrere Events reagiert werden.

Dieses Verhalten unterscheidet den *Event Handler* von der `receive` und `pick` Aktivität. Diese Aktivitäten reagieren ebenfalls auf eingehende Nachrichten. Mit `receive` kann jedoch nur auf ein definiertes *Message Event* reagiert werden, anschließend ist Aktivität abgeschlossen. Gleiches gilt für die `pick` Aktivität, nur können hier mehrere Events definiert werden, auf die reagiert werden soll. Die Definition der *Message Events* in `pick` Aktivitäten verläuft ähnlich wie im *Event Handler*.

Fazit

Die Nachrichten, bzw. die dadurch ausgelösten *Message Events* lassen sich mit den in dieser Arbeit behandelten Events vergleichen. Hierbei gibt es allerdings keine konkreten Event-Objekte. Stattdessen werden Nachrichten verschickt und lösen erst auf der Gegenseite Events aus. Mit diesen können unterschiedliche Prozesse und Systeme miteinander kommunizieren. Dies funktioniert auf eine vergleichbare Art, wie sie in dieser Arbeit mit Events vorgestellt wurde. Damit lassen sich durch Verschicken von Nachrichten externe Systeme flexibel mit dem BPM-System verbinden.

Es gibt aber noch weitere Unterschiede. Zum einen fehlt ein komplexes Routing, wie es in dieser Arbeit verwendet wurde. Die Quellen und Ziele von Nachrichten sind durch Angabe von `Partner Links`, `Port Types`, `Operations` und `Correlation Sets` klar definiert. Im Gegensatz zum hier entwickelten Event-Mechanismus fehlt auch das freie Mitlesen von Events durch beliebige Interessenten. Allerdings ist nicht ausgeschlossen, dass die Nachrichten auch an weitere Systeme verteilt werden.

Zudem fehlt die Anbindung an CEP. Es sind keine Möglichkeiten vorgesehen, komplexere Muster in einer Menge von *Message Events* zu erkennen. Es ist zwar z. B. möglich, auf eine bestimmte Sequenz oder einer Auswahl von Events zu reagieren. Allerdings fehlen unter anderem Muster, die auf Daten eingehen oder bestimmte Zeitfenster berücksichtigen.

8.2.2 BPMN

Die Business Process Model and Notation (BPMN) ist eine von der OMG spezifizierte grafische Modellierungssprache für Geschäftsprozesse [OMG11]. Sie bietet Sichten für die Modellierung, mit der sowohl technische als auch fachliche Anwender umgehen

können. Unter den umfangreichen Elementen zur Gestaltung von Prozessen bietet BPMN auch eine große Anzahl verschiedener Events an. Wir behandeln hier das Event-Modell von BPMN 2.0. Mit dieser Version sind einige umfangreiche Änderungen an den Events einhergegangen.

Events in BPMN

Events in BPMN sind Kontrollflusselemente. Sie finden während der Prozessausführung statt und beeinflussen den normalen Kontrollfluss [Bri08]. BPMN besitzt insgesamt 63 verschiedene Events, die in 13 Typen aufgeteilt sind. Darunter fallen *Event Types* wie unter anderem *Message*, *Timer*, *Link*, *Error* oder *Signal*. Dabei wird unterschieden zwischen Start-, Zwischen- und End-Events. Diese Kategorien werden teilweise noch weiter unterschieden. Bei Zwischen-Events wird z. B. zwischen eingetretenen und ausgelösten Events unterschieden. Nicht jeder *Event Type* ist in allen Varianten vorhanden. So gibt es z. B. keine *Timer*-End-Events.

Abgrenzung des Eventbegriffs zur bisherigen Betrachtung

Die Events in BPMN werden zum Teil sehr verschieden von der bisherigen Betrachtung genutzt. Zum einen sind sie unverzichtbare Elemente des Kontrollflusses, bisher haben wir dagegen Events im Kontrollfluss hauptsächlich als Synchronisationsmittel betrachtet. Zudem besitzt BPMN mehrere Events, die sich nicht mit den hier behandelten Events vergleichen lassen. Darunter fallen unter anderem die *Link* und *Conditional Events*. Die *Link Events* sind nur für die Sicht auf den Prozess entscheidend. Mit ihnen können Prozesse an bestimmten Stellen ausgesetzt und anderen Stellen fortgesetzt werden. Damit dienen sie nur der besseren Übersichtlichkeit des Prozesses.

Allerdings gibt es einige *Event Types* die sich durchaus mit den hier verwendeten Events vergleichen lassen. Wir betrachten nun einige davon:

Message Events Die *Message Events* sind in BPMN für den Austausch von Nachrichten zuständig. Diese sind wichtig für die Kommunikation zwischen mehreren *Pools*. *Message Events* können mit einem *Message Flow* verbunden werden, welcher mit Datenelementen assoziiert werden kann. Damit ist mit *Message Events* ein Datenaustausch möglich. Allerdings kann mit *Message Events* nur zwischen Pools und damit zwischen mehreren, wohldefinierten Geschäftspartnern kommuniziert werden. Damit haben

sie nicht die gleichen freien Verwendungsmöglichkeiten wie bei unserer bisherigen Eventbetrachtung.

Signal Events Die *Signal Events* ähneln den *Message Events*. Allerdings ist ein *Signal Event* eine ungerichtete Nachricht, d. h. Quelle und Ziel müssen nicht beide bekannt sein. Damit entsprechen sie am ehesten dem hier behandelten Eventbegriff. Empfangene Signale können eine beliebige Quelle haben und damit auch beliebige Prozess-Instanzen und Systeme. Diese können auch erst zur Laufzeit entschiedenen werden und müssen nicht bei der Modellierung des Prozesses bekannt sein. Auf der anderen Seite kann ein aufgetretenes Signal im Prozess an beliebige Interessenten verschickt werden. Allerdings ist im Gegensatz zu *Message Events* keine Übermittlung von Daten vorgesehen.

Error, Cancel und Compensation Events Mit *Error Events* kann angezeigt werden, dass bei der Bearbeitung eines Prozesses ein Fehler aufgetreten ist. Bei Auftreten von *Error* und *Cancel Events* wird der jeweilige Prozess bzw. Subprozess sofort beendet. *Cancel* und *Compensation Events* können nur in transaktionellen Prozessen verwendet werden. Damit sind diese Events nur für unerwünschte Ereignisse gedacht. Auch bei diesen Events sind Quelle und Ziel klar definiert.

Fazit

Die Events von BPMN lassen sich nur bedingt auf das hier behandelte Event-Modell übertragen. Am ehesten vergleichbar sind die *Signal Events*. Allerdings fehlen hier die Verknüpfungen mit Daten. Die Verwendung des Payloads war ein wesentlicher Bestandteil der in dieser Arbeit behandelten Events.

Ein weiterer Nachteil ist das Fehlen eines Konzepts für ein komplexes Routing der Events. BPMN konzentriert sich dabei lediglich auf Prozesse und beachtet dabei nicht, wie Events in weiteren Komponenten verwendet werden.

Ähnlich wie bei BPEL fehlt auch hier eine Anbindung an CEP Mechanismen. Die Erkennung von komplexen Mustern in eingetretenen Events ist nicht vorgesehen. Einzelne Aspekte, wie z. B. zwei hintereinander auftretende Events, lassen sich durch entsprechende Anordnung noch durchführen (vergleiche hierzu [BDG07]). Allerdings ist dies kein Ersatz für eine Ereignisanfragesprache wie CQL.

8.3 Zusammenfassung

In diesem Kapitel haben wir in Ergänzung dieser Arbeit einen Blick auf zwei verwandte Themen geworfen. Dabei haben wir zunächst die Ereignisanfragesprachen diskutiert. Dies sind Möglichkeiten, um auf einen Strom oder Menge von Events zuzugreifen und neue *Complex Event* zu produzieren. Gerade CQL zeigt einen in der Praxis sehr verbreiteten Ansatz zur Erkennung von Mustern in Events.

Der weitere Teil dieses Kapitels beschäftigte sich mit einem Vergleich der Event-Mechanismen von BPEL und BPMN mit dem in dieser Arbeit entwickelten Event-Mechanismus. Dabei wurden in beiden Fällen Ähnlichkeiten festgestellt. Allerdings gibt es auch auffällige Unterschiede. Insbesondere die Aspekte des freien Verteilens von Events und des Erkennens von Mustern haben in beiden Sprachen keinen hohen Stellenwert.

9 Zusammenfassung und Ausblick

Zum Abschluss der Arbeit werden in diesem Kapitel noch einmal die zentralen Ergebnisse dieser Arbeit in Abschnitt 9.1 zusammengefasst. Anschließend folgt in Abschnitt 9.2 ein Ausblick auf offene Fragestellungen, die bisher noch nicht oder nur unvollständig behandelt wurden.

9.1 Zusammenfassung

BPM-Systeme werden für Unternehmen immer wichtiger. Damit diese Systeme optimal arbeiten, müssen Daten in flexibler Weise innerhalb der Prozesse ausgetauscht und externe Systeme eingebunden werden; dabei stellen vor allem heterogene Systemlandschaften eine große Herausforderung dar. Eine in diesem Zusammenhang immer wichtiger werden Technik ist das Event Processing. Mit dieser Technik können unterschiedliche Systeme und Anwendungen konfigurierbar und flexibel per Events kommunizieren. Daher war es das Ziel dieser Arbeit, eine Event Processing Komponente für BPM-Systeme zu entwickeln.

Die in dieser Arbeit entwickelte Event Processing Architektur für BPM-Systeme nutzt als Grundlage eine EPN-Architektur. Dies ist ein Konzept, bei dem die Aufgaben der Eventverarbeitung nicht von einem monolithischen System, sondern durch ein Netzwerk aus sogenannten *Event Processing Agents* (EPA) ausgeführt wird. Durch den Einsatz dieser leichtgewichtigen Komponenten ist eine flexible Erweiterung jederzeit und leicht möglich.

Da Geschäftsprozesse meistens hierarchisch strukturiert sind, wurde dieses Prinzip auch auf die eingesetzte EPN-Architektur übertragen. Hierbei erhält jede Prozess-Instanz einen EPA, welcher die Eventverarbeitung für diese übernimmt. Die einzelnen EPAs sind mit den jeweiligen EPA des Vater-Prozesses verbunden. Dadurch entsteht ein sich dynamisch veränderndes, baumförmiges EPN. Dieses lässt sich leicht mit

externen Systemen, wie z. B. ERP-Systemen, verbinden, indem weitere EPAs dem Netzwerk hinzugefügt werden.

Eine besondere Herausforderung war dabei die sichere Verteilung der Events im EPN. Die einzelnen Produzenten und Konsumenten sind voneinander unabhängig und können die Events daher nicht direkt miteinander austauschen. Daher wurde ein *Publish/Subscribe*-Mechanismus entwickelt, welcher die Baumeigenschaft des Netzwerkes nutzt, um Produzenten und Konsumenten von Events miteinander in Verbindung zu bringen. Damit ist es dann auch gewährleistet, dass eine Verteilung der Events auch in einem sich verändernden EPN korrekt erfolgt. Dieser Mechanismus wurde anhand der ihm zugrunde liegenden Algorithmen erläutert.

Dadurch, dass die Prozesse nun sowohl als Konsumenten als auch als Produzenten fungieren, entstehen in einigen Fällen Abhängigkeiten der einzelnen Prozessschritte von Events. Daher wurde untersucht, welche Arten von Abhängigkeiten dabei möglich sind und wie das Prozess-Modell angepasst werden muss, um diese abbilden zu können. Hierbei wurde ein besonderer Fokus auf die sichere Versorgung mit Events innerhalb eines Prozesses gelegt.

Ein weiteres wichtiges Thema dieser Arbeit war die Betrachtung der Korrektheit von Prozessen in Bezug auf die konsumierten Events. Dabei wurde zunächst das Problem der Verklemmung aufgrund fehlender Events charakterisiert und anschließend ein Algorithmus zur Erkennung solcher Probleme diskutiert. Dieser basiert auf einer erweiterten Erreichbarkeitsanalyse. Dabei wurde unterschieden, ob ein Prozess in jedem Fall eine Verklemmung aufweist oder nur unter bestimmten Bedingungen. Um die Aussagekraft zu erhöhen, wurde untersucht, ob die Verklemmungsfreiheit sichergestellt ist, wenn die Reihenfolge der produzierten Events angepasst wird.

Um zu zeigen, dass die hier entwickelte EPN-Architektur auch in der Praxis realisierbar ist, wurde diese in einer prototypischen Implementierung umgesetzt. Dabei war vor allem die Umsetzung des sich dynamisch verändernden EPNs und der *Publish/Subscribe*-Mechanismus von Bedeutung.

Abgeschlossen wurde die Arbeit mit einer Betrachtung von verwandten Arbeiten. Dort wurden zunächst verschiedene Ereignisabfragesprachen vorgestellt. Mit diesen können Muster in einer Menge von Events erkannt und bei Bedarf daraus neue Events produziert werden. Anschließend wurden die Event-Modelle von BPEL und BPMN mit dem hier entwickelten Mechanismus verglichen.

In Rahmen dieser Arbeit wurde gezeigt, dass Event Processing viele Vorteile für BPM-Systeme bringen kann. Als Erstes fallen hier die automatische und zeitnahe Verarbeitung der Events auf. Diese können unabhängig vom eigentlichen Prozess genutzt werden. Insbesondere die Möglichkeit Events zur Kommunikation, zwischen unterschiedlichen Teilen des Systems, zu nutzen, bringen neuen Nutzen. Prozesse können per Events Daten austauschen und bleiben trotzdem unabhängig voneinander. Damit wird die Wiederverwendbarkeit von Prozessen und Aktivitäten erhöht. Zusätzlich können die Prozesse mit externen Systemen kommunizieren. Damit wird eine höhere Homogenität erreicht. Dies ist insbesondere von Vorteil, wenn Legacy-Systeme vorerst weiter genutzt und später ausgetauscht werden sollen. Die Prozesse bleiben durch die lose Kopplung unabhängig vom eingesetzten System.

Allerdings hat die Verwendung von Events bzw. das Event Processing nicht nur Vorteile. Die Sicherstellung der Korrektheit ist ein ernst zu nehmendes Problem. Diese kann nach aktuellem Stand noch nicht vollständig automatisiert werden. Dadurch entsteht ein zusätzlicher Aufwand beim Erstellen und Testen von Prozessen.

9.2 Ausblick

Diese Arbeit hat gezeigt, dass in dem Bereich des Event-Managements in BPM-Systemen noch weitere Forschungsarbeit vonnöten ist. Besonders hervorzuheben ist hierbei der Aspekt der Korrektheit. Insbesondere die Betrachtung der Subprozesse und ihre Abhängigkeiten bezogen auf Events müssen weiter untersucht werden.

Ein weiterer Aspekt, der hier nicht betrachtet wurde, ist die Performance. Event Processing Systeme benötigen, bei vielen eintretenden Events, einen hohen Rechenaufwand, um schnell reagieren zu können. Insbesondere bei einer großen Anzahl von auftretenden Events kann der Aufwand schnell ansteigen [Luc02]. Hier ist insbesondere die Betrachtung einer effizienten Skalierbarkeit der hier betrachteten Architektur interessant. Zusätzlich gibt es weitere Optimierungsmöglichkeiten. Darunter fällt eine direkte Verteilung der Events vom Produzenten an den Empfänger.

Des Weiteren wurden in dieser Arbeit keine Sicherheitsaspekte behandelt. Events enthalten Informationen, die in bestimmten Fällen nicht für jeden Benutzter einsehbar sein dürfen. Hier muss weiter geprüft werden, inwieweit Filter oder andere Methoden die Zugriffsbeschränkungen dieser Informationen sicherstellen können.

Literaturverzeichnis

[ABW04] ARASU, Arvind ; BABU, Shivnath ; WIDOM, Jennifer: CQL: A Language for Continuous Queries over Streams and Relations. In: LAUSEN, G. (Hrsg.) ; SUCIU, D. (Hrsg.): *Database Programming Languages* Bd. 2921. Springer Berlin / Heidelberg, 2004. – ISBN 978–3–540–20896–9, S. 123–124

[ABW06] ARASU, Arvind ; BABU, Shivnath ; WIDOM, Jennifer: The CQL continuous query language: semantic foundations and query execution. In: *The VLDB Journal* 15 (2006), June, S. 121–142

[ACKM04] ALONSO, Gustavo ; CASATI, Fabio ; KUNO, Harumi ; MACHIRAJU, Vijay: *Web Services: Concepts, Architectures and Applications*. Springer Berlin / Heidelberg, 2004. – ISBN 978–3–540–44008–6

[AE02] ADI, Asaf ; ETZION, Opher: The Situation Manager Rule Language. In: *International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002, S. 36–57

[AE04] ADI, Asaf ; ETZION, Opher: Amit - the situation manager. In: *The VLDB Journal* 13 (2004), May, S. 177–203

[Ari] *AristaFlow Business Process Management Suite*. – `http://www.aristaflow.com/` zuletzt besucht am 07.05.2012

[BBB+07] BERSTEL, Bruno ; BONNARD, Philippe ; BRY, François ; ECKERT, Michael ; PĂTRÂNJAN, Paula-Lavinia: Reactive Rules on the Web. In: *Reasoning Web* Bd. 4636. Springer Berlin / Heidelberg, 2007, S. 183–239

[BDG07] BARROS, Alistair ; DECKER, Gero ; GROSSKOPF, Alexander: Complex Events in Business Processes. In: ABRAMOWICZ, Witold (Hrsg.): *Business*

Information Systems Bd. 4439. Springer Berlin / Heidelberg, 2007. – ISBN 978–3–540–72034–8, S. 29–40

[Bri08] BRIOL, Patrice: *BPMN the Business Process Modeling Notation Pocket Handbook*. Lulu Press, 2008. – ISBN 978–1–4092–0299–8

[BW01] BABU, Shivnath ; WIDOM, Jennifer: Continuous queries over data streams. In: *SIGMOD Rec.* 30 (2001), September, S. 109–120. – ISSN 0163–5808

[DAG+06] DADAM, Peter ; ACKER, Hilmar ; GÖSER, Kevin ; JURISCH, Martin ; KREHER, Ulrich ; LAUER, Markus ; RINDERLE-MA, Stefanie ; REICHERT, Manfred: ADEPT2 - Ein adaptives Prozess-Management-System der nächsten Generation. In: D. SPATH, O. H. A. Weisbecker W. A. Weisbecker (Hrsg.): *Proceedings Stuttgarter Softwaretechnik Forum: Science meets Business - Aktuelle Trends aus der Softwaretechnikforschung*. Stuttgart : Fraunhofer IRB-Verlag, 2006, S. 1–10

[DRRA05] DADAM, Peter ; REICHERT, Manfred ; RINDERLE, Stefanie ; ATKINSON, Colin: Auf dem Weg zu prozessorientierten Informationssystemen der nächsten Generation : Herausforderungen und Lösungskonzepte. In: SPATH, D. (Hrsg.) ; HAASIS, K. (Hrsg.) ; KLUMPP, D. (Hrsg.): *Aktuelle Trends in der Softwareforschung - Tagungsband zum doIT-Forschungstag, Stuttgart, Germany*, 2005, S. 47–67

[DRRM+09] DADAM, Peter ; REICHERT, Manfred ; RINDERLE-MA, Stefanie ; GOESER, Kevin ; KREHER, Ulrich ; JURISCH, Martin: Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen. In: *EMISA Forum* 29 (2009), January, Nr. 1, S. 9–28

[EB09] ECKERT, Michael ; BRY, Francois: Complex Event Processing (CEP). In: *Informatik-Spektrum* 32 (2009), S. 163–167

[Eck08] ECKERT, Michael: *Complex Event Processing with XChangeEQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events*, Ludwig-Maximilians-Universität München, Diss., Oktober 2008

[EFGK03] EUGSTER, Patrick T. ; FELBER, Pascal A. ; GUERRAOUI, Rachid ; KERMARREC, Anne-Marie: The Many Faces of Publish/Subscribe. In: *ACM Comput. Surv.* 35 (2003), S. 114–131

[EN10] ETZION, Opher ; NIBLETT, Peter: *Event Processing in Action*. Manning Publications Co., 2010. – ISBN 978–1–935182–21–4

[Esp] *Esper Tech Inc. Event Processing with Esper and NEsper*. – `http://esper.codehaus.org/` zuletzt besucht am 29.05.2011

[FG08] FREUND, Jakob ; GÖTZER, Klaus: *Vom Geschäftsprozess zum Workflow: Ein Leitfaden für die Praxis*. Hanser Verlag München, 2008. – ISBN 978–3–446–41482–2

[For82] FORGY, Charles L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. In: *Artificial Intelligence* 19 (1982), Nr. 1, S. 17 – 37

[HA99] HAGEN, Claus ; ALONSO, Gustavo: Beyond the black box: Event-based inter-process communication in process support systems. In: *19th IEEE International Conference on Distributed Computing Systems*. Austin, TX , USA, 1999, S. 450–457

[Jos08] JOSUTTIS, Nicolai: *SOA in der Praxis: System-Design für verteilte Geschäftsprozesse*. Heidelberg : dpunkt, 2008. – ISBN 978–3–89864–476–1

[LKRD10] LANZ, Andreas ; KREHER, Ulrich ; REICHERT, Manfred ; DADAM, Peter: Enabling Process Support for Advanced Applications with the AristaFlow BPM Suite. In: *Proc. of the Business Process Management 2010 Demonstration Track*, 2010 (CEUR Workshop Proceedings 615)

[Luc02] LUCKHAM, David C.: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0–201–72789–7

[Oas07] Organization for the Advancement of Structured Information Standards (OASIS): *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. April 2007. – `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf` Abgefragt am 16.10.2011

[OMG11] OBJECT MANAGEMENT GROUP (OMG) (Hrsg.): *Business Process Model and Notation (BPMN) Version 2.0*. Januar 2011. – `http://www.omg.org/spec/BPMN/2.0/` Abgefragt am 15.10.2011

[RDJ+08] REICHERT, Manfred ; DADAM, Peter ; JURISCH, Martin ; KREHER, Ulrich ; GÖSER, Kevin: Architectural Design of Flexible Process Management Technology. In: *Proc. PRIMIUM Subconference at MKWI* Bd. 328, 2008, S. 415–422

[Rei00] REICHERT, Manfred: *Dynamische Ablaufänderung in Workflow-Management-Systemen*, Universität Ulm, Diss., Mai 2000

[See10] SEEGER, Bernhard: *Complex Event Processing: Auswertung von Datenströmen*. 2010. – `http://heise.de/-905334` zuletzt besucht am 29.05.2011

[YJ10] YE, Chunyang ; JACOBSEN, Hans-Arno: Event Exposure for Web Services: A Grey-Box Approach to Compose and Evolve Web Services. In: CHIGNELL, Mark (Hrsg.) ; CORDY, James (Hrsg.) ; NG, Joanna (Hrsg.) ; YESHA, Yelena (Hrsg.): *The Smart Internet* Bd. 6400. Springer Berlin / Heidelberg, 2010. – ISBN 978–3–642–16598–6, S. 197–215

Abkürzungsverzeichnis

ADEPT Application DEvelopment Based on Pre-Modeled Process Templates

BPEL Business Process Execution Language

BPM Business Process Management

BPMN Business Process Model and Notation

CEP Complex Event Processing

CQL Continuous Query Langue

CRM Customer Relationship Management

EC Event Consumer

EP Event Producer

EPA Event Processing Agent

EPE Event Processing Element

EPN Event Processing Network

ERP Enterprise Resource Planning

IEC Instance Event Consumer

IEP Instance Event Producer

IEPA Instance Event Processing Agent

OMG Object Management Group

SOA Serviceorientierte Architektur

XML Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Maximilian Lackaw Matrikelnummer: 591207

Erklärung

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

Ulm, den .....

Maximilian Lackaw