Modellierung, Analyse und Simulation von Regeln in der aktiven Schicht ALFRED

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät

der Universität Bern

vorgelegt von

Georg Lörincze

1996

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

Zusammenfassung

In vielen Unternehmen werden Datenbanksysteme eingesetzt, um wichtige Unternehmensdaten zu speichern und zu verwalten. Das Bedürfnis, Abläufe automatisieren zu können, hat dazu geführt, dass Datenbanksysteme, die nur zur Speicherung von Daten eingesetzt werden, nicht mehr den Anforderungen genügen. Aus diesem Grund werden seit einigen Jahren auf dem Gebiet der Datenbanken Lösungen gesucht, um den neuen Anforderungen gerecht zu werden. Eine mögliche Lösung bieten *aktive* Datenbanksysteme. Diese besitzen die Eigenschaft, dass sie auf bestimmte Situationen automatisch reagieren können. Ein solches Verhalten wird als *aktiv* bezeichnet und kann mit Hilfe von Regeln beschrieben werden. In diesen werden die Situationen spezifiziert, die erkannt werden müssen, und Aktionen festgelegt, die beim Eintreten dieser Situationen ausgeführt werden. In diesem Zusammenhang ergeben sich neue Aufgaben, die eine Lösung erfordern. So muss festgelegt werden, wie *aktives* Verhalten mit Hilfe von Regeln realisiert werden kann. Dazu muss spezifiziert werden, wie Regeln in einem ADBS *modelliert* und *verarbeitet* werden können.

Diese Diplomarbeit befasst sich mit der Modellierung, der Analyse sowie der Simulation von Regeln in der aktiven Schicht ALFRED (**A**ctive **L**ayer **F**or **R**ule **E**xecution in **D**atabase Systems). Diese Schicht kann auf prinzipiell beliebige konventionelle (passive) Datenbanksysteme aufgesetzt werden und erweitert diese um Funktionalitäten, mit denen *aktives* Verhalten realisierbar ist. In dieser Schicht werden Regeln mit all ihren Komponenten gesamthaft in einem einzigen Modell dargestellt.

Ein Dankeschön ...

Diese Diplomarbeit wurde betreut durch:

Institut für Informatik und angewandte Mathematik (IAM)
der Universität Bern
Prof. Dr. Oscar Nierstrasz

Institut für Wirtschaftsinformatik (IWI)
der Universität Bern
Prof. Dr. Gerhard Knolmayer
und
Dipl. Inf. Markus Schlesinger

Ein besonderer Dank geht an *Markus Schlesinger*. Seine Betreuung und Unterstützung war stets hervorragend.

Danken möchte ich auch all jenen Personen, die in irgendeiner Weise zum Gelingen dieser Arbeit beigetragen haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Konventionelle Datenbanksysteme sind heutzutage weit verbreitet. Durch den Einzug der EDV in unterschiedlichste Anwendungsbereiche sind die Anforderungen an diese Datenbanksysteme (DBS) jedoch deutlich gestiegen. So sollen DBS nicht nur mehr zur reinen Verwaltung und Speicherung von Daten eingesetzt werden. Sie sollen auch die Realisierung von automatischen Abläufen ermöglichen, was im folgenden Szenario verdeutlicht wird:

Wenn z.B. die Lagerverwaltung eines Fahrradherstellers betrachtet wird, so ist es bislang nur möglich, aktuelle Lagerbestände mit Hilfe des DBS zu verwalten, d.h. Lagermengen und Lagerbewegungen zu erfassen. In vielen Fällen ist es jedoch wünschenswert, wenn beim Erreichen einer Mindestlagermenge automatisch neue Teile bestellt würden, damit die Produktion von Fahrrädern nicht aufgrund von Ressourcenknappheit unterbrochen werden muss. Um ein solches *aktives* Verhalten in diesen DBS dennoch realisieren zu können, müssen Applikationsprogramme zur Überwachung der Lagerbestände geschrieben werden.

Eine solche Realisierung ist jedoch mit Problemen verbunden. So muss z.B. das Applikationsprogramm mit der darunterliegenden Datenbank kommunizieren, wodurch gerade zeitkritische Anwendungen nur ungenügend realisiert werden können. Aufgrund solcher Probleme und der Notwendigkeit, aktives Verhalten realisieren zu können, wurden neue Konzepte auf dem Gebiet der DBS entwickelt. Eine mögliche Lösung bieten *aktive Datenbanksysteme* (ADBS), die konventionelle Datenbanksysteme um Funktionalitäten erweitern, mit denen aktives Verhalten definiert, gesteuert und ausgeführt werden kann.

Dieses aktive Verhalten wird dabei durch *Situations-Aktions-Regeln* definiert, die beschreiben, wie in einer bestimmten Situation reagiert werden muss. Der Situationsteil einer Regel kann im DBS durch zwei Teile beschrieben werden, durch ein *Ereignis*, welches eintreten muss (z.B. eine Datenmanipulation), und durch eine *Bedingung*, die den aktuellen Datenbankzustand auf gewisse Eigenschaften hin überprüft. Daraus folgt, dass Regeln im allgemeinen aus den drei Komponenten Ereignis (Event), Bedingung (Condition) und Aktion (Action) bestehen (ECA-Regeln).

In diesem Zusammenhang ergeben sich einige Aufgaben und Probleme, die eine Lösung erfordern. So muss als erstes festgelegt werden, welches Modell für die Regeldarstellung geeignet ist. Daraufhin muss definiert werden, wie eine Regel auf der Grundlage dieses Modells dargestellt werden kann.

Wenn eine Menge von Regeln definiert ist, so muss das ADBS dieses spezifizierte aktive Verhalten realisieren. Dazu ist es notwendig, die definierten Regeln zu verarbeiten. Um dies zu erreichen, muss das ADBS in der Lage sein, Situationen zu überwachen, d.h. Ereignisse zu erkennen und Bedingungen auszuwerten, und es muss die Ausführung von Aktionen ermöglichen.

Die Regelmodellierung und die Regelverarbeitung bergen jedoch Gefahren in sich. So ist es

z.B. möglich, dass sich Regeln gegenseitig auslösen und somit Regelkaskaden entstehen. Wenn diese Regelkaskaden zyklisch sind, so besteht die Gefahr, dass die Verarbeitung dieser Regeln nicht abbricht. Aus diesem und anderen Gründen müssen Möglichkeiten gefunden werden, um die definierten Regeln analysieren zu können. Zusätzlich sollte das ADBS die Simulation von Abläufen unterstützen, mit deren Hilfe geprüft werden kann, ob das realisierte aktive Verhalten mit dem gewünschten aktiven Verhalten übereinstimmt.

Am Institut für Wirtschaftsinformatik, Abteilung Information Engineering, der Universität Bern, wird zur Zeit an Konzepten für die aktive Schicht ALFRED (**A**ctive **L**ayer **F**or **R**ule **E**xecution in **D**atabase Systems) gearbeitet. Diese Schicht ermöglicht es, ein prinzipiell beliebiges konventionelles (passives) Datenbanksystem um Funktionalitäten zu erweitern, mit deren Hilfe aktives Verhalten realisiert werden kann. ALFRED ist datenbank- und branchenunabhängig. Das aktive Verhalten wird durch Regeln definiert, die vollständig mit Hilfe von *gefärbten Petrinetzen* dargestellt werden. Im Rahmen dieser Diplomarbeit werden Konzepte und Lösungen für die Modellierung von Regeln, für einen Teilbereich der Regelanalyse (Erkennung und statische Analyse von Zyklen) und für die Verarbeitung von Regeln, resp. deren Simulation, vorgestellt.

Der Aufbau dieser Arbeit ist wie folgt: In Kapitel 2 folgt eine Einführung in aktive Datenbanksysteme. Kapitel 3 erläutert die Konzepte von ALFRED. In Kapitel 4 werden verschiedene Modelle für die Darstellung von Regeln untersucht und verglichen. Auf der Basis dieses Vergleiches wird ein spezifisches Modell ausgewählt (gefärbte Petrinetze). Kapitel 5 beinhaltet die Grundlagen und die notwendigen Definitionen für gefärbte Petrinetze. In Kapitel 6 wird beschrieben, wie Regeln auf der Grundlage von gefärbten Petrinetzen modelliert werden können. Kapitel 7 befasst sich mit der Analyse von Regeln, genauer mit der Erkennung und der statischen Analyse von Zyklen. In Kapitel 8 wird die Verarbeitung von Regeln (resp. deren Simulation) in ALFRED beschrieben. Die Arbeit endet mit Kapitel 9, in dem eine kurze Zusammenfassung und ein Ausblick gegeben werden.

Kapitel 2

Aktive Datenbanksysteme

Datenbanksysteme werden heutzutage in vielen Unternehmen aus nahezu allen Branchen eingesetzt. Anwendungsbereiche dieser DBS sind unter anderem Datenverwaltung, Computer Aided Design (CAD), Computer Integrated Manufacturing (CIM) und Informationssysteme aus unterschiedlichen Bereichen wie z.B. der Medizin oder der Telekommunikation. Viele dieser Anwendungen verlangen, dass das DBS beim Eintreten von bestimmten Situationen gewisse Reaktionen automatisch ausführen kann.

Beispiel (Inkassogesellschaft) *In einer Inkassogesellschaft werden unter anderem Rechnungen an Schuldner versandt und ausstehende Beträge eingezogen. Damit der Versand von Rechnungen und die Einzahlung von Beträgen kontrolliert werden können, werden die Daten des Rechnungsversandes (Datum, Betrag, Adresse etc.) sowie die Daten der erfolgten Einzahlung in einem DBS abgespeichert. Unter der Annahme, dass nicht alle Schuldner den Betrag umgehend begleichen, ist es notwendig, Mahnungen zu versenden. Diese Mahnungen werden jeweils 50 Tage nach Versand der Rechnungen verschickt, falls bis dahin der Rechnungsbetrag nicht einbezahlt wurde.*

Damit die Schuldner rechtzeitig erkannt und die Mahnungen zugestellt werden können, gibt es zwei Möglichkeiten.

1. Die Bearbeitung erfolgt manuell. Dies bedeutet, dass die entsprechenden Sachbearbeiter jeden Tag überprüfen müssen, ob der geschuldete Betrag beglichen wurde oder ob die Frist von 50 Tagen abgelaufen ist. Wenn letzteres eingetreten ist, so müssen Mahnungen erstellt und versandt werden.
2. Die Bearbeitung erfolgt mit Hilfe von Applikationsprogrammen, die dieses aktive Verhalten realisieren.

Diese zwei Möglichkeiten zeigen, dass *aktives* Verhalten mit konventionellen DBS nur ungenügend realisiert werden kann. Der Grund dafür besteht darin, dass konventionelle DBS *passiv* sind, d.h. keine Mechanismen enthalten, mit denen Situationen erkannt und entsprechende Reaktionen ausgeführt werden können. Um in diesen Systemen dennoch ein solches Verhalten realisieren zu können, müssen Programme auf Applikationsebene geschrieben werden, die mit dem DBS kommunizieren. Diese Kommunikation kann auf zwei verschiedene Arten erfolgen:

1. **Integriert**
   Jedes Applikationsprogramm, das die Einzahlungen der Schuldner registriert, muss um zusätzliche Prozeduren erweitert werden, die bei Eingang der Zahlung sämtliche Schuldner auf die Überschreitung der Zahlungsfrist überprüfen. Wenn dies der Fall ist, so müssen Mahnungen erstellt werden.

2. **Polling**
   Eine spezielle Prozedur kontrolliert in periodischen Zeitabständen die Einhaltung der Zahlungsfristen, indem die notwendigen Informationen aus der Datenbank selektiert werden (Polling). Für Schuldner, welche die Rechnung nicht beglichen und die Zahlungsfrist überschritten haben, wird daraufhin eine Mahnung erstellt.

Beide Ansätze sind mit signifikanten Nachteilen verbunden. Beim integrierten Ansatz wird die Modularität von Programmen verletzt. Da jedes Applikationsprogramm durch entsprechende Codezeilen erweitert werden muss, wird die Überprüfung in verschiedenen Applikationsprogrammen vorgenommen. Wenn diese Überprüfung geändert werden soll, entsteht ein hoher Verwaltungsaufwand, und es besteht die Gefahr von Inkonsistenzen, da eine Änderung an verschiedenen Orten im Applikationsprogramm notwendig wird. Beim zweiten Ansatz muss festgelegt werden, in welchen Zeitabständen die Kontrollen durchgeführt werden sollen (Polling-Frequenz). Wenn die Zeitabstände zu kurz sind, so leidet die Performance darunter, da jede Kontrolle eine gewisse Verarbeitungszeit benötigt. Zudem ist es möglich, dass unnötige Kontrollen durchgeführt werden. Falls die Zeitabstände zu gross gewählt werden (z.B. alle 7 Tage), so können die Schuldner eventuell nicht rechtzeitig erkannt werden.

Die Notwendigkeit, aktives Verhalten realisieren zu können, und die oben aufgeführten Nachteile, haben zu neuen Konzepten auf dem Gebiet der DBS geführt, den sogenannten *aktiven* Datenbanksystemen. Mit Hilfe dieser Datenbanksysteme ist es möglich, aktives Verhalten zu definieren, zu steuern und auszuführen.

Die Definition von *aktivem* Verhalten geschieht mit Hilfe von *Situations-Aktions*-Regeln, die beschreiben, wie beim Eintreten einer gewissen Situation reagiert werden muss. Als Beispiel soll die vorher erwähnte Inkassogesellschaft dienen. Das gewünschte aktive Verhalten, d.h. die automatische Erstellung von Mahnungen, kann durch folgende Situations-Aktions-Regel definiert werden:

- **Situation:** Die Rechnung wurde vor 50 Tagen versandt und der Rechnungsbetrag ist noch ausstehend.
- **Aktion:** Erstellung einer Mahnung für den betreffenden Schuldner.

In den folgenden Abschnitten werden die Merkmale von ADBS sowie Anforderungen an die Spezifikation von Regeln erläutert. Danach folgt ein Überblick über Anwendungsgebiete und eine kurze Übersicht über ADBS-Forschungsprojekte und kommerziell verfügbare (aktive) DBS.

2.1 Merkmale

Aktive Datenbanksysteme stellen eine Erweiterung konventioneller (passiver) DBS um Mechanismen dar, die es ermöglichen, aktives Verhalten zu realisieren. Aus dieser Erweiterung ergeben sich folgende Anforderungen an und Merkmale von ADBS:

- Ein ADBS muss alle notwendigen Funktionalitäten von (passiven) DBS umfassen, so z.B. Transaktionsverarbeitung, Datenrecovery, eine Datendefinitionssprache (Data Definition Language, DDL), eine Datenmanipulationssprache (Data Manipulation Language, DML) und Datenabfragen (Queries).
- Ein ADBS muss es dem Benutzer ermöglichen, aktives Verhalten in Form von Situations-Aktions-Regeln (vgl. Abschnitt 2.2) zu definieren. Die Definition dieser Regeln muss ein persistenter Bestandteil der Datenbank werden, d.h. sie muss im ADBS abgespeichert werden.

- Ein ADBS muss das aktive Verhalten kontrollieren und steuern können. Dazu gehört die Erkennung von entsprechenden Situationen und die automatische Auslösung und Verarbeitung von Aktionen.
- Ein ADBS sollte eine Entwicklungsumgebung zur Verfügung stellen, mit deren Hilfe Regeln betrachtet (Browser) und analysiert, sowie Fehler erkannt werden können (Debugger).

2.2 Regelparadigma

Um auf bestimmte Situationen reagieren zu können, muss das aktive Verhalten zum voraus definiert werden. Dies geschieht mit *Regeln*, die durch Situations-Aktions-Paare umschrieben werden.

Eine Situation setzt sich aus einem *Ereignis* und einer *Bedingung* zusammen. Das Ereignis gibt an, auf was reagiert werden kann. Die Bedingung spezifiziert, welche Eigenschaften die Datenbank nach der Erkennung des Ereignisses erfüllen muss, damit die entsprechende Aktion ausgeführt wird.

Die Aufsplittung der Regel in die Teile Ereignis, Bedingung und Aktion führt zur sogenannten *Event-Condition-Action (ECA) Notation* [MD89]. Eine Regel besteht demnach aus drei Komponenten:

- **Ereignis** (Event)
  Das Eintreten des Ereignisses *löst* die Regel *aus*. Ein Ereignis kann zum Beispiel ein Benutzerbefehl (wie `insert in EMPLOYEE`) oder ein bestimmter Zeitpunkt (wie `1.1.1997 17.00 Uhr`) sein.
- **Bedingung** (Condition)
  Durch die Bedingung wird der aktuelle Datenbankzustand auf bestimmte Eigenschaften hin überprüft (z.B. `name = 'Karl'`). Die Bedingung wird dann ausgewertet, wenn die Regel ausgelöst wurde, d.h. wenn das Ereignis eingetreten ist.
- **Aktion** (Action)
  Durch eine Aktion wird definiert, *wie* auf eine bestimmte Situation reagiert werden soll. Eine Aktion kann z.B. ein Datenmanipulationsbefehl oder ein ganzes Programm sein und wird genau dann ausgeführt, wenn das Ereignis eingetreten und die Bedingung erfüllt ist.

Neben dieser allgemeinen Form (ECA) von Regeln gibt es verschiedene Varianten. EA-Regeln, die nur aus einem Ereignis- und einem Aktionsteil bestehen, ECAA-Regeln [Her95], die zwei mögliche Aktionsteile enthalten (je einen Aktionsteil für beide Möglichkeiten bei der Bedingungsauswertung (TRUE und FALSE)), sowie CA- und CAA-Regeln, die nicht durch ein explizites Ereignis ausgelöst werden.

2.3 Regelspezifikation

Das aktive Verhalten eines ADBS wird durch Regeln beschrieben, die ihrerseits in einer *Regeldefinitionssprache* (Rule Definition Language, RDL) spezifiziert werden. Durch diese RDL wird das Spektrum des aktiven Verhaltens eines ADBS definiert.

Um eine Regel mittels einer RDL definieren zu können, muss abgegrenzt werden, was unter den einzelnen Komponenten, d.h. dem Ereignis, der Bedingung und der Aktion, zu verstehen ist, und welche ausführungsbezogenen Faktoren berücksichtigt werden müssen.

2.3.1 Regelstruktur

In den folgenden Abschnitten werden die Komponenten einer Regel, d.h. Ereignisse, Bedingungen und Aktionen, abgegrenzt.

2.3.1.1 Ereignisse

Eine Regel wird durch das Eintreten eines bestimmten Ereignisses ausgelöst. Dieses Ereignis kann sowohl *primitiv* als auch *komplex* sein. Mögliche primitive Ereignisse sind z.B:

- **Datenmanipulationsereignisse**
  Diese Ereignisse werden durch DML-Befehle ausgelöst (z.B. `update on ...`).
- **Zeitereignisse**
  Diese Ereignisse werden durch das Eintreten eines bestimmten Zeitpunktes ausgelöst (z.B. `12.01.1997 @ 17:15:00`).
- **Transaktionsereignisse**
  Diese werden durch Transaktionsbefehle ausgelöst (z.B. `begin of transaction`).

Ein komplexes Ereignis ist eine Kombination von primitiven und/oder komplexen Ereignissen. Diese Kombination wird mittels einer *Ereignisalgebra* resp. mittels Operatoren für Ereignisse realisiert. Mögliche Ereignisoperatoren sind zum Beispiel:

- **Bool'sche Operatoren**
  Diese Operatoren definieren eine logische Verknüpfung (AND oder OR) von zwei Teilereignissen (z.B. das komplexe Ereignis $E_k := E_1$ *and* E_2 tritt ein, wenn das Ereignis E_1 *und* das Ereignis E_2 eintreten).
- **Sequenzoperatoren**
  Mit diesen Operatoren wird eine Reihenfolge festgelegt, in der die Teilereignisse eintreten müssen (z.B. das komplexe Ereignis $E_k :=$ *sequence* (E_1, E_2, E_3) tritt ein, wenn zuerst E_1, dann E_2 und dann E_3 eintreten).
- **Wiederholungsoperatoren**
  Durch diese Klasse von Operatoren wird ein komplexes Ereignis definiert, das nach einer bestimmten Anzahl Wiederholungen der Teilereignisse eintritt (z.B. das komplexe Ereignis $E_k :=$ *every 5* E_1 tritt ein, jedesmal wenn E_1 5-mal eingetreten ist).
- **Intervalloperatoren**
  Diese Operatoren definieren Intervalle, in welchen Ereignisse eintreten müssen. Zum Beispiel ein komplexes Ereignis $E_k := E_i$ *in* (E_b, E_e) tritt ein, wenn das Ereignis E_i im Intervall eintritt, das durch das Eintreten des Ereignisses E_b beginnt und durch das Eintreten des Ereignisses E_e endet.
- **Zeitoperatoren**
  Diese Operatoren erlauben es, komplexe Ereignisse zu definieren, die z.B. in periodischen Zeitabständen oder zeitverzögert ausgelöst werden sollen (z.B. das komplexe Ereignis $E_k :=$ *4 days after* E_1 tritt ein, nachdem das Ereignis E_1 eingetreten ist, und 4 Tage vergangen sind).

2.3.1.2 Bedingungen

Eine Bedingung beschreibt, welche Eigenschaften die Datenbank zu einem bestimmten Zeitpunkt erfüllen muss, d.h. in welchem Zustand sich die Datenbank zum Zeitpunkt der Bedingungsauswertung befinden muss. Bedingungen sind entweder *primitiv* oder *komplex*. Mögliche primitive Bedingungen sind zum Beispiel:

- **Prädikat**
  Ein Vergleich zweier Operanden (z.B. `Employee.gehalt > 5'000`).
- **Abfrage**
  Eine (möglicherweise leere) Menge von Datensätzen, die mittels eines *retrieves* aus der Datenbank extrahiert werden. Wenn die selektierte Menge leer ist, so ist die Bedingung falsch (FALSE), andernfalls ist sie wahr (TRUE).

Komplexe Bedingungen werden mittels logischer Operatoren (z.B. AND, OR oder NOT) aus primitiven und/oder komplexen Bedingungen zusammengesetzt.

2.3.1.3 Aktionen

In der Aktionskomponente der Regel wird festgelegt, wie in einer erkannten Situation reagiert werden soll. Die Aktionen können sowohl *primitiv* als auch *komplex* sein. Mögliche primitive Aktionen sind zum Beispiel:

- **Datenmanipulationen**
  Mit Hilfe dieser Aktionen können Datensätze im DBS selektiert, eingefügt, geändert oder gelöscht werden (z.B. `retrieve name from Employee`).
- **Ausgabe von Meldungen**
  Diese Aktionen ermöglichen die Ausgabe von Mitteilungen an den Benutzer.
- **Ausführung von Prozeduren**
  Mit diesen Aktionen können Prozeduren aufgerufen werden, die in der Datenbank verarbeitet werden.

Komplexe Aktionen sind Verkettungen von primitiven Aktionen (z.B. Befehlsketten oder ganze Programme).

2.3.2 Regelausführung

Neben der Spezifikation des aktiven Verhaltens ist es notwendig, verarbeitungsbezogene Eigenschaften anzugeben, d.h. Eigenschaften, die bei der Regelverarbeitung berücksichtigt werden. Diese müssen mittels der RDL zum Zeitpunkt der Regeldefinition beschrieben werden. Im folgenden werden einige wichtige Faktoren erläutert.

2.3.2.1 Auslösungszeitpunkte

Bei der Ausführung einer Aktion gilt es, in der Datenbank zwei Zustände resp. Zeitpunkte zu unterscheiden. Der Zustand *vor* der Ausführung der Aktion und der Zustand *danach*. Es ist deshalb sinnvoll, in ADBS auf beide möglichen Zustände reagieren zu können, d.h. Ereignisse für beide Zustände respektive Zeitpunkte definieren zu können. Folgende zwei Auslösungszeitpunkte werden unterstützt:

- **Pre**
  Regeln, deren Ereignisse mit dem Zeitpunkt *pre* definiert sind, werden *vor* der effektiven Befehlsverarbeitung ausgelöst. Regeln dieser Art können z.B. zur Überprüfung von Zugriffsrechten verwendet werden.

- **Post**
  Regeln, deren Ereignisse mit dem Zeitpunkt *post* definiert sind, werden *nach* der effektiven Befehlsverarbeitung ausgelöst. Solche Regeln können z.B. dazu verwendet werden, um komplexe Integritätsbedingungen zu überprüfen.

2.3.2.2 Auslösungsgranularität

Bei der Ausführung von Befehlen wird häufig auf mehrere Datensätze zugegriffen. Es kann deshalb sinnvoll sein, zwei verschiedene Auslösungsgranularitäten zu unterscheiden:

- **Instanzorientiert** (instance-oriented)
  Instanzorientierte Regeln werden für jeden involvierten Datensatz genau einmal ausgeführt.

- **Mengenorientiert** (set-oriented)
  Mengenorientierte Regeln werden je Befehl genau einmal ausgeführt. Die Anzahl der Datensätze, auf die dieser Befehl zugreift, hat keinen Einfluss.

2.3.2.3 Prioritäten

Bei der Verarbeitung einer Regelmenge kann es vorkommen, dass einige Regeln durch dasselbe Ereignis ausgelöst werden. Um in dieser Situation entscheiden zu können, in welcher Reihenfolge die Regeln verarbeitet werden sollen, können bei der Definition der Regeln Prioritäten vergeben werden. Anhand dieser Prioritäten wird in Konfliktsituationen entschieden, in welcher Reihenfolge die Regeln ausgeführt werden müssen.

2.3.2.4 Reihenfolge

Falls eine Konfliktsituation eintritt und keine oder gleiche Prioritäten vergeben wurden, so wird zwischen folgenden zwei Möglichkeiten der Verarbeitungsreihenfolge unterschieden:

- **Sequentielle Ausführung**
  Die Reihenfolge kann zufällig oder dynamisch, z.B. unter Berücksichtigung von Datensperren, festgelegt werden.

- **Parallele Ausführung**
  Die Regeln können parallel verarbeitet werden. Dazu muss das ADBS jedoch entsprechende Funktionalitäten unterstützen.

2.3.2.5 Kopplungsmodi

In ADBS werden Regeln genau dann ausgelöst, wenn bestimmte Ereignisse eintreten. Der Zeitpunkt, zu dem eine Regel verarbeitet wird, muss aber nicht der Zeitpunkt der Auslösung sein. Zum Beispiel kann es sinnvoll sein, die Regelverarbeitung erst am Ende der regelauslösenden Transaktion oder in einer separaten Transaktion durchzuführen. Diese zeitliche Beziehung zwischen *regelauslösender* Transaktion, d.h. derjenigen Transaktion, in der das Ereignis eintritt, und Regelverarbeitung wird *Kopplungsmodus* (KM) genannt. Dieser Kopplungsmodus muss bei der Definition einer Regel festgelegt werden.

Der Kopplungsmodus kann sowohl zwischen Ereignis und Bedingung (EC-Kopplung) als auch zwischen Bedingung und Aktion (CA-Kopplung) definiert werden. Die EC-Kopplung gibt an, wann die Bedingungsauswertung in Bezug zur Regelauslösung zu erfolgen hat, und die CA-Kopplung bestimmt den Zeitpunkt der Aktionsausführung in Bezug zur Bedingungsauswertung. Es wird zwischen sechs verschiedenen Möglichkeiten von Kopplungsmodi [BBKZ92] unterschieden, die sowohl bei der EC- als auch bei der CA-Kopplung verwendet werden. Diese sechs Möglichkeiten werden hier exemplarisch anhand der EC-Kopplung erläutert.

- **Immediate**
  Die Bedingungsauswertung erfolgt in der gleichen Transaktion unmittelbar nach dem Eintritt des Ereignisses.
- **Deferred**
  Die Bedingungsauswertung erfolgt in der gleichen Transaktion, jedoch nach dem letzten Befehl aber bevor die Transaktion auf der Datenbank festgeschrieben wird.
- **Detached**
  Die Bedingung wird in einer separaten Transaktion ausgewertet. Zwischen den beiden Transaktionen besteht *keine* Abhängigkeit, d.h. die Bedingungsauswertung wird in jedem Falle ausgeführt, auch wenn die regelauslösende Transaktion abgebrochen und zurückgesetzt wird.
- **Detached Causally Dependent**
  Bei dieser Klasse von Kopplungsmodi wird die Bedingung ebenfalls in einer separaten Transaktion ausgewertet. Es besteht jedoch eine Abhängigkeit zwischen den beiden Transaktionen. Zwischen den folgenden drei Abhängigkeiten wird unterschieden:
  - **Sequential**
    Die neu erzeugte Transaktion wird erst nach dem *commit* der regelauslösenden Transaktion verarbeitet.
  - **Parallel**
    Die neu erzeugte und die regelauslösende Transaktion können parallel ausgeführt werden. Das *commit* der neuen Transaktion darf jedoch erst nach dem *commit* der regelauslösenden Transaktion erfolgen.
  - **Exclusive**
    Die neu erzeugte und die ursprüngliche Transaktion können parallel ausgeführt werden. Die neue Transaktion wird jedoch nur festgeschrieben, wenn die regelauslösende Transaktion abgebrochen wird.

2.3.2.6 Parameterkontexte

Um bei der Bedingungsauswertung und der Aktionsausführung auf Informationen zum Zeitpunkt der Ereigniserkennung zugreifen zu können, müssen bei der Erkennung von Ereignissen Parameter gebunden werden.

Komplexe Ereignisse setzen sich aus mehreren primitiven und/oder komplexen Ereignissen zusammen, die im Ablauf zu unterschiedlichen Zeitpunkten und verschieden häufig eintreten können (*Ereignisinstanzen* genannt). Somit besteht die Möglichkeit, bei der Ereigniserkennung unterschiedliche Parameterwerte zu binden. Die Festlegung, welche Parameter gebunden werden müssen, wird durch *Parameterkontexte* beschrieben.

Es wird zwischen vier Parameterkontexten unterschieden [CM93], die anhand eines Beispielereignisses erläutert werden:

$$E_K = Sequenz(E_1, E_2)$$

Das komplexe Ereignis E_K wird dann ausgelöst, wenn die primitiven Ereignisse E_1 und E_2 in der angegebenen Reihenfolge auftreten. Zusätzlich wird nun ein Ausschnitt aus einer fiktiven Ereignisgeschichte abgebildet. E_i^n bezeichnet das n-te Auftreten des Ereignisses E_i.

$$E_2^1; E_1^1; E_1^2; E_2^2; E_1^3; E_1^4; E_2^3; E_2^4$$

Mit Hilfe des komplexen Ereignisses E_K und der obenstehenden Ereignisgeschichte, werden nun die vier Parameterkontexte erläutert:

- **Recent**
  In diesem Kontext werden die *jüngsten* Parameterwerte der Teilereignisse gebunden. In der obenstehenden Ereignisgeschichte wird E_K genau 2 mal ausgelöst, dabei werden folgende Parameter berücksichtigt:
  - Auslösung bei E_2^2 mit den Parametern von E_1^2 und E_2^2.
  - Auslösung bei E_2^3 mit den Parametern von E_1^4 und E_2^3.

  Obwohl die Erkennung von E_K schon bei E_1^1 resp. E_1^3 eingeleitet wird, muss in diesem Kontext für die Parameterbindung das jeweils letzte Eintreten von Ereignis E_1 berücksichtigt werden.
- **Chronicle**
  In diesem Kontext wird die chronologische Reihenfolge der Teilereignisse berücksichtigt, d.h. es werden jeweils die *ältesten* Parameter gebunden. Für das Beispielereignis E_K bedeutet dies:
  - Auslösung bei E_2^2 mit den Parametern von E_1^1 und E_2^2.
  - Auslösung bei E_2^3 mit den Parametern von E_1^3 und E_2^3.
- **Cumulative**
  In diesem Kontext werden die Parameter aller Teilereignisse gebunden, die zum Eintreten des komplexen Ereignisses geführt haben. Für das Beispielereignis E_K bedeutet dies:
  - Auslösung bei E_2^2 mit den Parametern von E_1^1, E_1^2 und E_2^2.
  - Auslösung bei E_2^3 mit den Parametern von E_1^3, E_1^4 und E_2^3.
- **Continuous**
  In diesem Kontext werden Kombinationsmöglichkeiten von Parametern berücksichtigt. Alle Teilereignisse, welche die Erkennung des komplexen Ereignisses einleiten, werden als potentielle Parameterkandidaten angesehen und mit den Parametern der restlichen Teilereignisse kombiniert. In der obigen Ereignisgeschichte wird deshalb E_K 4 mal ausgelöst:
  - Auslösung bei E_2^2 mit den Parametern von E_1^1 und E_2^2.
  - Auslösung bei E_2^2 mit den Parametern von E_1^2 und E_2^2.
  - Auslösung bei E_2^3 mit den Parametern von E_1^3 und E_2^3.
  - Auslösung bei E_2^3 mit den Parametern von E_1^4 und E_2^3.

2.4 Systemarchitekturen

Es gibt verschiedene Möglichkeiten, um ein aktives Datenbanksystem zu entwickeln. Die drei wichtigsten Architekturen werden hier vorgestellt.

2.4.1 Schicht-Architektur

Die Komponenten einer aktiven Datenbank werden in einem Modul "über" einem konventionellen DBS zusammengefasst, d.h. über ein konventionelles DBS wird eine aktive "Schicht" gelegt. Sämtliche Befehle eines Benutzers gelangen durch die aktive Schicht an die darunterliegende Datenbank und alle Daten aus der Datenbank werden durch die Schicht hindurch an den Benutzer weitergereicht. Somit sind sämtliche Informationen in der Schicht verfügbar, die für die Regelausführung notwendig sind. Dieser Ansatz wird auch "layered architecture" genannt.

- *Vorteile*
  - Ein konventionelles DBS kann ohne Modifikation des DBS in ein ADBS umgewandelt werden.
  - Verschiedene konventionelle DBS können in ADBS umgewandelt werden und weisen dieselbe Benutzerschnittstelle auf (z.B. für heterogene Datenbanksysteme).
- *Nachteile*
  - Es besteht die Gefahr von geringer Performance, da der Kommunikationsaufwand zwischen der aktiven Schicht und der Datenbank hoch sein kann.
  - Die aktive Schicht kann mit gewissen Subsystemen (z.B. Transaktionsmanager) nicht kommunizieren. Dadurch ist es möglich, dass gewisse ausführungsbezogene Regeleigenschaften nicht erfüllt werden können (z.B. Kopplungsmodi).

2.4.2 Integrierte Architektur

Die Komponenten eines ADBS sind Bestandteil der Datenbank selbst. Dies ist nur möglich, wenn ein bestehendes konventionelles Datenbanksystem abgeändert oder wenn ein DBS von Grund auf neu entwickelt wird. Die Erkennung von Ereignissen, die Auswertung von Bedingungen sowie die Ausführung von Aktionen geschieht bei dieser Architektur im Datenbankmanagementsystem selbst. Dieser Ansatz wird auch "built-in architecture" genannt.

- *Vorteile*
  - Ereigniserkennung, Bedingungsauswertung und Aktionsausführung können sehr effizient erfolgen.
  - Der Zugang zu Subsystemen wie Transaktionsmanager etc. ermöglichen die Realisierung von komplexen ausführungsbezogenen Regeleigenschaften (z.B. Kopplungsmodi, Recovery).
- *Nachteile*
  - Der Aufwand für die Änderung eines bestehenden oder die Entwicklung eines neuen DBS ist sehr hoch.
  - Es wird nur *ein* spezifisches DBS in ein ADBS umgewandelt.

2.4.3 Compilierte Architektur

Jeder Befehl, den der Benutzer ausführen will, wird in eine für die Datenbank verständliche Form gebracht (compiliert) und mit zusätzlichen Programmzeilen versehen, die das aktive Verhalten definieren. Alle Benutzerbefehle müssen also zur Laufzeit erweitert und compiliert werden. Dieser Ansatz wird auch "compiled architecture" genannt.

- *Vorteile*
  - Ereigniserkennung während der Laufzeit und die Regelverarbeitung entfallen.
- *Nachteile*
  - Da alle Ereignisse durch den Compiler erkannt werden müssen, sind komplexe Ereignisse und Zeitereignisse nicht möglich.
  - Jeder Befehl muss zur Laufzeit compiliert werden. Dies kann sich nachteilig auf die Performance auswirken.
  - Zyklische Regelkaskaden können nicht berücksichtigt werden, da die Compilierung rekursiv erfolgen und nicht abbrechen würde.

2.5 Anwendungen

Mit aktiven Datenbanksystemen können Anwendungen, die auf der Basis von "passiven" DBS nur mit Hilfe von zusätzlichen Applikationsprogrammen möglich sind, besser unterstützt werden. Beispiele für solche Anwendungen sind:

- **Integritätsbedingungen** (integrity constraints)
  Integritätsbedingungen spezifizieren Bedingungen über Daten in einem DBS. Oft werden diese Bedingungen verwendet, um Konsistenzen innerhalb der Datenbank sicherzustellen. In konventionellen DBS werden häufig folgende Integritätsbedingungen (IB) unterstützt [Sch95, WCD95]:
  - *Not-Null IB* (not null constraints) garantieren, dass gewisse Attribute einer Relation Werte enthalten, d.h. sie dürfen nicht leer (not null) sein.
  - *Schlüssel IB* (key constraints) garantieren, dass in einer Relation keine zwei Datensätze mit gleichem Schlüsselattribut existieren.
  - *Primärschlüssel IB* (primary key constraints) stellen eine Verschärfung der Schlüssel IB dar, indem sie zusätzlich garantieren, dass Schlüsselattribute keine Null-Werte enthalten.
  - *Referenzielle IB* (referential integrity constraints) werden jeweils auf zwei Relationen definiert, zwischen denen eine Existenzabhängigkeit besteht. Die referenzielle IB garantiert, dass jeder Fremdschlüsselwert der abhängigen Relation einen Primärschlüsselwert der übergeordneten Relation referenziert.

  Diese in konventionellen DBS unterstützten IB reichen für viele Anwendungen jedoch oft nicht aus. So soll z.B. sichergestellt werden, dass alle Datensätze aus einer bestimmten Menge stammen. Zusätzlich kann in einem passiven DBS die Bedingungsprüfung nur nach einer Datenbankoperation oder einer Transaktion erfolgen, während es ein ADBS erlaubt, den Zeitpunkt der Prüfung der IB flexibel zu wählen (z.B. *vor* einer Operation oder zu einem bestimmten Zeitpunkt wie "Am 20.1.1998 um 17.00 Uhr"). Mit dieser Flexibilität lassen sich komplexe IB realisieren.
- **Trigger**
  Ein Datenbanktrigger spezifiziert eine Prozedur, die automatisch ausgeführt wird, wenn eine Datenmanipulation bezüglich einer Relation durchgeführt wird. Mit Hilfe von Triggern können komplexe IB realisiert werden. Regeln in ADBS stellen eine Erweiterung des Triggerkonzeptes dar, in dem sie z.B. komplexe Ereignisse zulassen und verschiedene Auslösungszeitpunkte unterscheiden.

- **Autorisation**
  Konventionelle (passive) DBS beinhalten eine spezielle Komponente, um Zugriffsbeschränkungen und Benutzerprivilegien zu kontrollieren [WC96]. Diese Privilegien müssen jedesmal geprüft werden, wenn ein Benutzer eine Datenbankoperation ausführen möchte. Ein aktives DBS kann die Autorisation direkt realisieren, ohne ein Subsystem dafür zu enthalten.

- **Statistiken**
  Konventionenlle DBS bieten im allgemeinen keine Möglichkeit zur statistischen Auswertung (z.B. Anzahl Datenzugriffe). In ADBS können solche Auswertungen realisiert werden.

- **Netzwerkmanagement**
  Aktive DBS können eingesetzt werden, um Netzwerke und Flüsse in Netzwerken zu steuern und zu überwachen. Beispiele dafür sind Kommunikationsnetzwerke.

- **Überwachung**
  Anwendungen, die Situationen überwachen und entsprechend darauf reagieren, können durch ADBS unterstützt werden. Beispiele dafür sind Flugüberwachung oder Überwachung von Wasserständen.

- **Kontrolle und Steuerung**
  Aktive DBS können für die Kontrolle, Verarbeitung und Steuerung eingesetzt werden, so z.B. für Workflows und zur Inventarkontrolle.

- **Finanzapplikationen**
  Applikationen aus dem Finanzbereich können mittels ADBS realisiert werden, wie z.B. Portfoliomanagement, Geldhandel und Wertpapierhandel.

- **Multimedia Applikationen**
  Multimedia Applikationen können durch ADBS untestützt werden, wenn das ADBS Möglichkeiten bietet, multimedia-spezifische Daten (wie z.B. Grafiken, Musik) abzuspeichern und zu verwalten.

2.6 Überblick

In der Literatur lassen sich unzählige Prototypen von aktiven Datenbanksystemen finden. Diese Systeme unterscheiden sich in vielerlei Hinsicht. Die nachfolgende Auflistung soll einen kleinen Überblick über einige existierende Projekte und Prototypen sowie über kommerziell verfügbare (aktive) DBS geben. Eine solche Aufzählung kann auf keinen Fall vollständig sein, da stets neue Projekte hinzukommen, alte Projekte eingestellt werden und bestehende Projekte abgeändert werden. Die Auflistung wird in drei Gruppen aufgeteilt. Die erste Gruppe von Projekten basiert auf relationalen Datenbanksystemen, die zweite Gruppe auf objektorientierten und die dritte Gruppe umfasst kommerziell verfügbare DBS, mit denen eingeschränktes aktives Verhalten realisierbar ist (vgl. Abbildung 2.1).

2.6.1 Relationale aktive Datenbanksysteme

Folgende Projekte basieren auf relationalen DBS:

- **Ariel**
  Dieses Projekt der Write State University und später der University of Florida legt das Schwergewicht auf eine effiziente Bedingungsauswertung. In Ariel wurde eine integrierte Architektur auf der Basis von EXODUS gewählt ([Han92b, Han92a]).

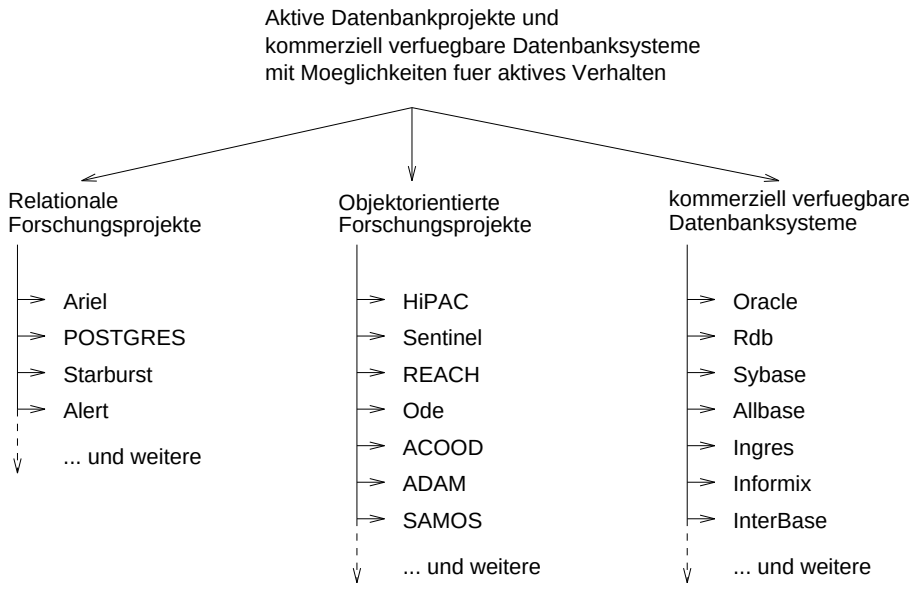

Abbildung 2.1: Überblick über Datenbanksysteme und aktive Datenbankprojekte

- **POSTGRES**
  POSTGRES ist ein Forschungsprojekt der University of California at Berkeley. Die Trigger werden u.a. für die Realisierung von Views und für Integritätsbedingungen verwendet. In POSTGRES wurde eine integrierte Architektur gewählt ([SR86, PS96, SRH90]).

- **Starburst**
  Die Starburst-Regelerweiterung ist ein Projekt am IBM Almaden Research Center, California. Das Regelsystem wurde mit einer integrierten Architektur auf der Basis von Starburst implementiert. Dieses Projekt befasst sich vor allem mit Regelinterferenzen, Terminierung und einer genau definierten Sprachsemantik ([WCL91, Wid92, WF90]).

- **Alert**
  Alert ist ebenfalls ein Projekt am IBM Almaden Research Center und wird auf der Basis der relationalen Datenbank Starburst entwickelt. Alert basiert auf einer Schicht-Architektur. Ziel dieses Projektes ist, eine passive Datenbank mit möglichst geringem Aufwand in ein ADBS umzuwandeln ([SPAM91]). Dies geschieht mittels *aktiver Relationen* und *aktiver Abfragen*.

2.6.2 Objektorientierte aktive Datenbanksysteme

Folgende Projekte basieren auf objektorientierten DBS:

- **HiPAC**
  Eines der ersten objektorientierten Projekte auf dem Gebiet der aktiven Datenbanken. Es wurde durch die DARPA und das Rome Air Development Center gegründet. Entwickelt wurde HiPAC an der Computer Corporation of America und am Xerox Advanced Information Technology Center. Der HiPAC Prototyp stellt eine Erweiterung von PROBE auf der Basis einer integrierten Architektur dar. Das Schwergewicht des Projektes liegt in einer vielfältigen Sprache für die Spezifikation von Ereignissen, einer mächtigen Ausführungssemantik, sowie in der Betrachtung von Zeitgrenzen bei der Regelausführung. Viele Projekte auf dem Gebiet der aktiven Datenbanken bauen auf der Arbeit von HiPAC auf ([DBB+88, MD89, DBC96]).

- **Sentinel**
  Dieses Projekt der University of Florida ist auf der Basis von HiPAC entstanden. Prototypen von Sentinel wurden mit Hilfe einer integrierten Architektur auf der Basis von "Zeitgeist" und dem DARPA Open OODB Toolkit entwickelt. Das Schwergewicht dieses Projektes liegt in einer mächtigen Sprache für die Ereignisspezifikation (Snoop) und in der Möglichkeit, Regeln in eine Datenbank-Programmiersprache zu integrieren ([CM93, CAM93]).

- **REACH**
  REACH ist ein Projekt der Technischen Hochschule Darmstadt und lehnt sich an die Projekte HiPAC und Sentinel an. Das Projekt befasst sich mit der Verwaltung von heterogenen Daten und umfasst u.a. Zeitbeschränkungen bei der Regelausführung (*milestones*). REACH wurde mit Hilfe einer integrierten Architektur auf der Basis von O2 und ObjectStore implementiert ([Deu94, BBKZ92, BBKZ93]).

- **Ode**
  Ode ist ein Projekt an den AT&T Bell Laboratories und erweitert die Sprache O++ um Konstrukte, die es ermöglichen, Regeln zu definieren. Ode ist selbst eine objektorientierte Datenbank, die um aktives Verhalten erweitert wurde, d.h. Ode basiert auf einer integrierten Architektur ([GJ92, GJS93, GJ96]).

- **ACOOD**
  Dieses Projekt wird an der Universität von Exeter und Skövde entwickelt und erweitert das kommerziell verfügbare System ONTOS um Funktionalitäten zur Bearbeitung von Regeln. ACOOD wurde auf der Basis einer integrierten Architektur entwickelt ([Ber91]).

- **ADAM**
  ADAM ist ein aktives objektorientiertes Datenbanksystem auf der Basis einer integrierten Architektur und wird an der University of Aberdeen entwickelt. Ein Prototyp von ADAM wurde in PROLOG implementiert. In ADAM werden die Regeln ebenfalls als Objekte dargestellt und mit in das Datenmodell integriert ([DPG91, DJPaQ94]).

- **SAMOS**
  SAMOS ist ein Projekt an der Universität von Zürich. Es basiert auf einer Schicht-Architektur. Ein Prototyp wurde auf der Basis von ObjectStore entwickelt. Das Projekt umfasst eine umfangreiche Sprache zur Spezifikation von Ereignissen und Mechanismen zur Erkennung dieser Ereignisse mittels Petrinetzen ([GGD94, GD92, Fri93, Gat95]).

2.6.3 Kommerzielle Datenbanksysteme

Auch in kommerziell verfügbaren DBS ist aktives Verhalten integriert. Die unterstützten Möglichkeiten in diesen DBS liegen jedoch weit hinter denen der oben erwähnten Prototypen und Projekte. So werden z.B. ausführungsbezogene Faktoren wie Kopplungsmodi noch nicht unterstützt. In vielen kommerziell verfügbaren DBS wird mit Hilfe von Triggerkonzepten, ein gewisses aktives Verhalten ermöglicht, wie z.B. in *Allbase, Ingres, Informix, InterBase, Oracle, Rdb* und *Sybase* ([WCD95, Sch95]).

Dass die Entwicklung von aktiven Datenbanksystemen in der heutigen Zeit ein Bedürfnis darstellt, zeigt auch der Umstand, dass der SQL3-Standard, der in zukünftigen kommerziellen DBS eingesetzt werden wird, die Möglichkeit bietet, Regeln, Trigger und komplexere Integritätsbedingungen zu definieren. Somit werden wohl konventionelle Datenbanksysteme, die keine Möglichkeit bieten, aktives Verhalten zu definieren und zu steuern, in einigen Jahren der Vergangenheit angehören.

Kapitel 3

Konzepte von ALFRED

Am Institut für Wirtschaftsinformatik der Universität Bern wird zur Zeit ein Entwurf einer aktiven Regelschicht für Datenbanksysteme entwickelt. Diese Schicht trägt den Namen ALFRED, welcher ein Akronym für **A**ctive **L**ayer **F**or **R**ule **E**xecution in **D**atabase Systems ist. Die Entwicklung dieser Schicht wurde vor allem durch zwei Missstände motiviert, die in bisherigen Prototypen und Konzepten von aktiven Datenbanksystemen zu finden sind, und die aus betriebswirtschaftlicher Sicht nicht akzeptierbar sind. Diese zwei Missstände sind:

1. *Datenbankabhängigkeit*
   Die meisten Prototypen und Konzepte von ADBS beruhen auf einer integrierten Architektur und sind demzufolge an *ein* spezifisches Datenbanksystem gebunden. Da in der Praxis jedoch zahlreiche konventionelle DBS existieren und da diese DBS zum Teil grosse Investitionen darstellen, sollte aus betriebswirtschaftlicher Sicht ein ADBS auf der Basis von möglichst beliebigen konventionellen DBS entwickelt werden, damit nicht grundsätzliche Neuinvestitionen notwendig werden.

2. *Branchenabhängigkeit*
   Die Regelsprache in den existierenden Prototypen und Konzepten wurde oft in Hinblick auf ein bestimmtes Anwendungsgebiet festgelegt. Ähnliche Anwendungen oder gar Anwendungen aus ganz anderen Branchen lassen sich somit kaum realisieren. Aus diesem Grund ist es notwendig, dass ein ADBS eine möglichst flexible Regelsprache zur Verfügung stellt, mit der sowohl umfangreiche Ereignisse, Bedingungen und Aktionen definiert werden können als auch notwendige Eigenschaften für die Regelverarbeitung. Mit einer solch flexiblen Regelsprache kann das ADBS universell, d.h. branchenunabhängig, eingesetzt werden.

In den nun folgenden Abschnitten werden die Anforderungen an und die Konzepte von ALFRED erläutert.

3.1 Anforderungen

Das Ziel, die Missstände Datenbankabhängigkeit und Branchenabhängigkeit zu beheben, sowie ausgiebige Auswertungen der Literatur von ADBS haben zu einer Menge von Anforderungen geführt, die in ALFRED erfüllt werden müssen. Diese Anforderungen werden nachfolgend in logische Gruppen aufgeteilt und kurz erläutert.

3.1.1 Unternehmen

Aus der Sicht eines Unternehmens ergeben sich einige wichtige Anforderungen, die bislang in Prototypen kaum oder gar nicht berücksichtigt wurden. Folgende Eigenschaften müssen erfüllt sein:

- **Abbildbarkeit von Geschäftsregeln**
  In Unternehmen gibt es Geschäftsregeln, die sich auf ein Informationssystem beziehen, Regeln, die zwischen einem Informationssystem und dessen Umwelt angesiedelt sind, und Regeln, die nicht in Verbindung mit Informationssystemen gebracht werden können. Ziel von ALFRED ist die Modellierung *aller* unternehmerischer Regeln, also eine *vollständige* Abbildung der Geschäftsregeln, wenngleich gewisse Regeln nur zu Informations- und Analysezwecken im System abgelegt werden und nicht durch das Informationssystem verarbeitet werden können.

- **Branchenunabhängigkeit**
  Unterschiedliche Branchen haben unterschiedliche Bedürfnisse und Geschäftsregeln, mit deren Hilfe sie die Unternehmensziele erfüllen wollen. In ALFRED sollen prinzipiell beliebige Geschäftsregeln aus unterschiedlichen Branchen dargestellt werden können.

- **Sichten**
  In ALFRED soll es möglich sein, die Regeln nach gewissen Gesichtspunkten analysieren zu können. Dabei sollen verschiedene Sichten auf das Regelsystem die Arbeit erleichtern. So könnten z.B. abteilungsspezifische Sichten (z.B. Alle Regeln, die einen Einfluss auf den Bereich F&E haben) als auch Produktsichten (z.B. Alle Regeln, die mit den Aktien xy in Verbindung stehen) für die unternehmerische Arbeit von Nutzen sein.

3.1.2 Benutzer

Aus der Sicht des Benutzers sind folgende Eigenschaften moderner Softwareprodukte wichtig:

- **Selbsterklärbarkeit**
  Das System soll so aufgebaut sein, dass die Funktionalität klar erkenntlich und leicht verständlich ist. Der Benutzer soll soweit unterstützt werden, wie dies nur möglich ist (z.B. bei der Definition von Regeln).

- **Leichte Erlernbarkeit**
  Die Bedienung von ALFRED soll möglichst einfach über eine grafische Benutzeroberfläche erfolgen.

- **Robustheit**
  Fehlmanipulationen durch den Benutzer sollen nicht zu Programmabbrüchen oder sogar zu inkonsistenten Datenbankzuständen führen.

3.1.3 Umgebung

Eine der wichtigsten Anforderungen an ALFRED ist die Datenbankunabhängigkeit. ALFRED muss so konzipiert sein, dass sich ein prinzipiell beliebiges (passives) DBMS in ein aktives DBMS verwandeln lässt. Um diese Forderung zu erfüllen sind folgende Kriterien zu berücksichtigen:

- **Datenmodellunabhängigkeit**
  Ausschnitte aus der realen Welt werden durch *konzeptionelle* Datenmodelle abgebildet. Ein wichtiger Vertreter davon ist das Entity Relationship Model (ERM). Verschiedene Unternehmen mit ebenso verschiedenen Bedürfnissen verwenden unterschiedliche Modelle, um ihre

Sachverhalte darzustellen. Diese modellierten Sachverhalte werden sodann in einem Datenbanksystem auf ein *logisches* Datenmodell abgebildet (relationales Datenmodell, objektorientiertes Datenmodell etc.). Aus diesem Grund wird von ALFRED gefordert, dass sowohl beliebige konzeptionelle als auch logische Datenmodelle unterstützt werden.

- **Datenbankunabhängigkeit**
  Damit mit ALFRED eine gewisse Universalität erreicht werden kann, muss es möglich sein, mit Hilfe von ALFRED heutige kommerziell verfügbare (passive) Datenbanksysteme in aktive Datenbanksysteme zu verwandeln.

3.1.4 Software

Folgende Software-Anforderungen sollen in ALFRED erfüllt werden:

- **Modularität**
  Das System soll in logisch abgeschlossene Module unterteilt werden. Diese Module können nur über vordefinierte Schnittstellen miteinander kommunizieren.

- **Erweiterbarkeit**
  Erfahrungen durch den Betrieb einer Software sowie veränderte Rahmenbedingungen sind nur zwei Gründe, die es unabdingbar machen, dass ein Softwareprodukt erweitert werden kann.

- **Wartbarkeit**
  Ab einer gewissen Grösse beinhaltet jedes Softwareprodukt Fehler. Diese Fehler sollen möglichst einfach und effizient behoben werden können. Zudem soll es möglich sein, die Software einfach an die sich entwickelnden Anforderungen anzupassen.

- **Performance**
  Es soll durch die Verwendung von entsprechenden Datenstrukturen und Algorithmen garantiert werden, dass die Ausführungszeit von ALFRED nicht exponentiell mit der Anzahl der zu verarbeitenden Aufgaben oder Regeln steigt. Nur eine angemessene Performance garantiert, dass ein System auch im täglichen Einsatz brauchbar ist.

3.1.5 Administration

Aus der Sicht des Administrators sind folgende Eigenschaften des Systems wünschenswert:

- **Einfachheit der Erstellung**
  Der Administrator wird im Laufe der Zeit Datenbanken, Objekttypen, Programme und auch Regeln erzeugen. ALFRED soll diese Arbeit weitgehend vereinfachen.

- **Verwaltung**
  Der Administrator soll zu jeder Zeit Überblick über die erstellten Datenbanken, Regeln, Entitätstypen, Benutzer etc. erhalten können. Dazu ist ein entsprechend organisiertes Repository von grosser Bedeutung.

- **Auswertung**
  In ALFRED sollen Funktionalitäten integriert werden, die es erlauben, die Regelmenge auszuwerten. Es soll u.a. möglich sein, Regeln und Beziehungen zwischen Regeln und Objekttypen zu analysieren. Mit Hilfe dieser Analyse wäre es z.B. möglich, alle Regeln zu finden, die eine Änderung in der Relation *Person* bewirken. Somit würde der Administrator einen Überblick über alle Regeln erhalten, die z.B. bei einer Namensänderung der Relation betroffen wären.

- **Automatische Ableitung**
  ALFRED soll so konzipiert sein, dass bei der Definition eines Datenschemas, Integritätsbedingungen, die aus dem konzeptionellen Datenmodell ersichtlich sind (z.B. referentielle IB), automatisch durch entsprechende Regeln garantiert werden.

3.1.6 Regelmodellierung

Bei der Regelmodellierung müssen folgende Anforderungen berücksichtigt und erfüllt werden:

- **Konsistenz**
  Inkonsistenzen können bei der Modifikation des Datenmodells entstehen, wenn z.B. ein Objekttyp gelöscht wird, an welchen noch eine Regel gebunden ist. In ALFRED muss mit entsprechenden Werkzeugen sichergestellt werden, dass diese Regel auch entfernt wird.

- **Konflikte**
  Bei der Definition von Regeln muss verhindert werden, dass Konflikte entstehen. So muss z.B. sichergestellt werden, dass es keine unmittelbare Kombination von Regeln gibt, in der die eine Regel gewisse Datensätze einfügt und die andere Regel diese gleich wieder entfernt. Durch die Vermeidung von Konflikten ist eine Organisation der Regeln viel einfacher, und die Gefahr von Inkonsistenzen wird dadurch verringert.

- **Zyklen**
  Da bei der Definition von Regeln auch kaskadierende Regeln zugelassen sind, besteht die Gefahr, dass Zyklen im Regelwerk entstehen. Diese Zyklen können vom Benutzer beabsichtigt sein und sollten demzufolge terminieren. Nichtterminierende Zyklen müssen jedoch vom System erkannt und verhindert werden.

- **Konfluenz**
  Falls bei der Ausführung von Regeln eine Situation eintritt, in welcher zwei oder mehrere Regeln durch dasselbe Ereignis ausgelöst werden, und falls unter den Regeln keine explizite Reihenfolge besteht, so ist die Ausführungsreihenfolge dieser Regeln dem Zufall überlassen. Wenn diese Reihenfolge keine Auswirkung hat, d.h. wenn unbeachtet der Reihenfolge der gleiche Datenbankzustand erreicht wird, so ist diese Menge von Regeln *confluent*. In ALFRED soll sichergestellt werden, dass die Regelmenge confluent ist.

3.1.7 Regelausführung

Folgende Anforderungen an die Regelausführung werden in ALFRED gestellt:

- **Ereigniserkennung**
  Regeln werden durch Ereignisse ausgelöst, wobei diese Ereignisse primitiv oder komplex sein können. Die Erkennung dieser Ereignisse sowie die Bestimmung der dazugehörigen Regeln muss möglichst effizient geschehen.

- **Bedingungsauswertung**
  In ALFRED sollen Algorithmen und Datenstrukturen verwendet werden, die eine effiziente Bedingungsauswertung zulassen.

- **Simulation**
  Um das Verhalten einer Regel oder einer Regelmenge zu überprüfen und zu verstehen, ist es hilfreich, dieses Verhalten anhand von Simulationen zu beobachten. Mit deren Hilfe ist es möglich, unerwünschtes Verhalten oder Seiteneffekte zu erkennen und zu beseitigen.

- **Parallelität**
  Bei der Verarbeitung von Regeln sollen mögliche parallele Ausführungen von Regeln oder Regelteilen berücksichtigt werden.

- **Fehlerbehebung**
  Falls bei der Verarbeitung von Regeln und Transaktionen Fehler auftreten, so müssen entsprechende Massnahmen getroffen und eventuell Transaktionen und Befehle zurückgesetzt werden. Diese Fehlerbehebung soll möglichst effizient geschehen.

- **Fristen**
  Bei der Definition kann angegeben werden, bis zu welchem Zeitpunkt eine Regel verarbeitet sein muss. Falls dieser Zeitpunkt überschritten wird, so kann eine vordefinierte Aktion stattfinden (contingency plan [DBB+88]). Diese Fristen sollen in ALFRED berücksichtigt werden.

- **Transaktionsabhängigkeit**
  Durch die Kopplungsmodi einer Regel wird der zeitliche Bezug zur regelauslösenden Transaktion festgelegt. In ALFRED sollen die sechs Kopplungsmodi *immediate, deferred, detached* sowie die drei Varianten von *detached causally dependent (sequential, parallel, exclusive)* unterstützt werden (vgl. Abschnitt 2.3.2.5).

- **Auslösungszeitpunkt**
  Die Auslösung einer Regel soll sowohl *vor* (pre) als auch *nach* (post) dem tatsächlichen Effekt einer Aktion möglich sein.

- **Auslösungsgranularität**
  In ALFRED sollen mengenorientierte und instanzorientierte Regelausführungen unterstützt werden.

- **Prioritäten**
  Da in der Regelmenge verschiedene Regeln durch gleiche Ereignisse ausgelöst werden können, soll es möglich sein, Regeln mit Prioritäten zu versehen, damit eine gewünschte Reihenfolge bei der Ausführung eingehalten wird.

3.1.8 Regelsprache

Die Regeln in einem ADBS werden durch eine Regeldefinitionssprache (RDL) spezifiziert. Diese Sprache muss so umfangreich sein, dass sämtliche Regelanforderungen beschrieben werden können. Dazu gehören:

- **Struktur**
  Mit Hilfe der RDL muss es möglich sein, sowohl ECA-Regeln als auch die Varianten ECAA, EA, CA und CAA zu modellieren.

- **Ereignisse**
  Die Regelsprache muss umfangreiche Möglichkeiten zur Definition von primitiven und komplexen Ereignissen bieten.

- **Bedingungen**
  Die RDL muss es ermöglichen, umfangreiche Bedingungen zu modellieren.

- **Aktionen**
  Mit Hilfe der RDL sollen verschiedenartige Aktionen und Programme aus unterschiedlichen Anwendungsbereichen definiert werden können.

- **Ausführung**
  Die im Abschnitt 2.3.2 aufgeführten Eigenschaften der Regelausführung müssen mit Hilfe der RDL definiert werden können.

Da in ALFRED die vollständige Beschreibung von Geschäftsabläufen mittels Regeln möglich sein soll, müssen auch Regeln dargestellt werden können, die nicht oder nur zum Teil im Informationssystem ablaufen (z.B. eine Regel, die besagt, dass nach Ablauf eines Bestellvorgangs die Kopien der Bestellungen in der Kartei abgelegt werden müssen). Aus diesem Grund müssen Ereignisse, Bedingungen und Aktionen unterstützt werden, die nur durch sprachliche Formulierungen umschrieben werden können.

Die Syntax der RDL ist im Anhang A zu finden.

3.2 Architektur

Aufgrund der oben genannten Anforderungen wurde für ALFRED eine Architektur gewählt, die eine Erweiterung der Schichtarchitektur darstellt. ALFRED kommuniziert mit dem zugrundeliegenden DBS nur über wenige, fest vorgegebene Schnittstellenfunktionen. Diese Funktionen stellen die Grundfunktionalität des DBS dar, wie `create table, drop table, insert, update, delete`. Damit in ALFRED aber auch Anforderungen an die Regelverarbeitung erfüllt werden können (wie z.B. Kopplungsmodi), beinhaltet ALFRED einen eigenständigen Transaktionsmanager und ein eigenes Repository für die Speicherung der Regelstrukturen. Nur mit einer solchen Architektur ist es möglich, die Datenbankunabhängigkeit zu erreichen und gleichzeitig eine genügend flexible Regeldefinition und Regelverarbeitung zu gewährleisten, damit Anwendungen aus unterschiedlichen Branchen realisiert werden können.

Um eine einheitliche Modellierung von Regeln gewährleisten zu können, werden in ALFRED *alle* Regelkomponenten als Petrinetze (vgl. Kapitel 5 und 6) dargestellt. Die nachfolgenden Abschnitte geben einen Überblick über die Systeme, die notwendig sind, um diese Petrinetze zu erzeugen, zu verwalten und zu verarbeiten.

3.2.1 Überblick

ALFRED besteht im wesentlichen aus zwei Subsystemen, die miteinander kommunizieren (vgl. Abbildung 3.1). Diese Systeme sind das Benutzersystem (*User System*), mit welchem die Benutzer arbeiten, und das Verarbeitungssystem (*Processing System*), in dem die Regeln modelliert, analysiert, abgelegt und verarbeitet werden. Das Verarbeitungssystem ist mit den zugrundeliegenden DBS verbunden.

Jeder Benutzerbefehl gelangt durch das Benutzersystem in das Verarbeitungssystem, in welchem Regeln, die durch die Benutzerbefehle ausgelöst werden, in die Verarbeitung miteinfliessen. Die notwendigen Regelaktionen sowie die Benutzerbefehle selbst, werden sodann auf der darunterliegenden Datenbank ausgeführt.

Damit die Benutzerbefehle verarbeitet werden können, müssen sie in eine für das System verständliche Form gebracht werden. In ALFRED werden alle Regelkomponenten und alle Benutzerbefehle mit Hilfe von Petrinetzen modelliert. Die Ereignisoperatoren, die Bedingungen und die Aktionen werden dabei als Transitionen dargestellt. In ALFRED gibt es zwei Stufen von Petrinetzen, das sogenannte *Action Flow Petri Net* (AFPN) und das *Action Rule Flow Petri Net* (ARFPN). Ein AFPN besteht im wesentlichen aus einem Anfang, einem oder mehreren Benutzerbefehlen und einem Ende. Das ARFPN ist eine Erweiterung eines AFPN. In einem ARFPN werden zusätzlich zu den Benutzerbefehlen auch Regeln integriert (vgl. Kapitel 6).

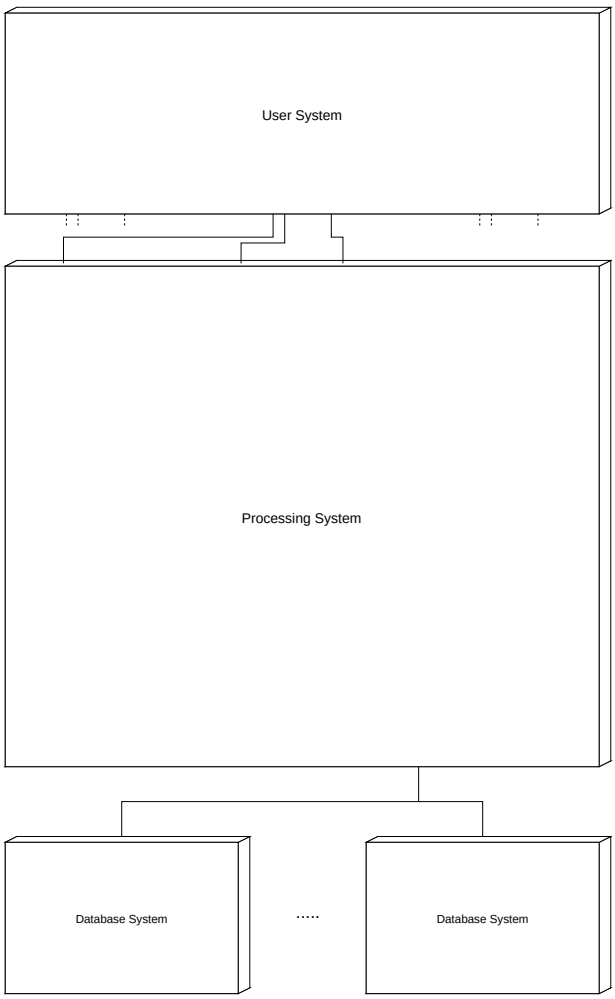

Abbildung 3.1: Das Grobkonzept von ALFRED

Jeder Befehl wird im Benutzersystem in ein AFPN umgewandelt, welches an das Verarbeitungssystem weitergegeben wird. Dieses System fügt in das AFPN Verbindungen für die Ereigniserkennung ein, wodurch das AFPN zu einem ARFPN wird. Die effektive Verarbeitung der Benutzerbefehle und der Regeln geschieht auf der Basis eines ARFPNs.

Im folgenden wird nun auf die beiden Subsysteme von ALFRED eingegangen (vgl. Abbildung 3.2). Dabei wird kurz erläutert, welche Aufgaben die einzelnen Subsysteme haben. Einige Teile aus dem Konzept von ALFRED werden in späteren Kapiteln ausführlicher behandelt. Darunter fallen die Regelmodellierung (vgl. Kapitel 6), die Zyklenerkennung (vgl. Kapitel 7) und die Verarbeitung resp. Simulation von Regeln (vgl. Kapitel 8).

- **Benutzersystem** (User System)
  Das Benutzersystem ist zuständig für die Kommunikation mit dem Benutzer und für die Erstellung eines ersten Ablaufmodelles (AFPN). Aus diesem Grund lässt sich das Benutzersystem wieder in Subsysteme teilen:

  - **Menüsystem** (Menu System)
    Das Menüsystem ist die Verbindung zwischen ALFRED und dem Benutzer. Es stellt eine Oberfläche und entsprechende Funktionalitäten für die Eingabe und Änderung von

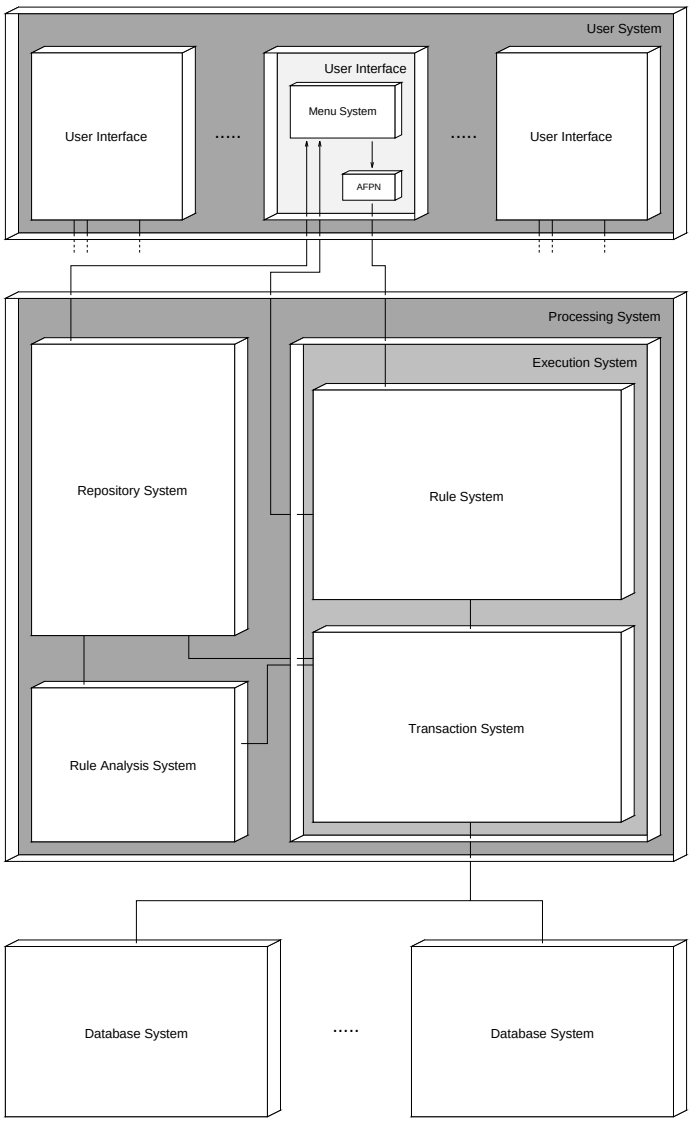

Abbildung 3.2: ALFRED mit aufgebrochenem User System und Processing System

Daten, Datenbanken und Regeln zur Verfügung. Dieses System besteht aus einem Hauptmenü, das die Kontrolle über die aktive Schicht hat und dem Benutzer meldet, ob ein Befehl erfolgreich ausgeführt werden konnte oder nicht. An das Hauptmenü sind verschiedene Untermenüs angegliedert:

* Das Datenbank-Menü, mit welchem eine Datenbank kreiert und eine Verbindung zu ihr hergestellt werden kann.
* Das Datendefinitions-Menü, mit dem Regeln und Objekttypen kreiert und abgeändert werden können.
* Das Datenmanipulations-Menü, in dem Daten selektiert, eingegeben, abgeändert und gelöscht werden können.
* Das Informationsmanagement-Menü, mit welchem Daten im Repository ausgewertet werden können.

  - Ein Simulations-Menü, mit welchem Ablaufsimulationen definiert und ausgeführt werden können.
  - Das Transaktions-Menü, mit welchem Transaktionen definiert und ausgeführt werden können.

- **AFPN Generierung** (AFPN Generation (AFPN))
  Die Befehle aus dem Datenmanipulations-Menü und dem Datendefinitions-Menü müssen in eine für das System verarbeitbare Form gebracht werden. Aus diesem Grund wird jeder Befehl in ein spezielles Petrinetz (AFPN) umgewandelt. Jede Aktion wird in diesem Netz durch eine Transition dargestellt.

- **Verarbeitungssystem** (Processing System)
  Das Verarbeitungssystem ist für folgende Aufgaben zuständig, die durch die jeweiligen Subsysteme des *Processing Systems* erfüllt werden:

  1. **Regelverarbeitung** (Rule System)
     Jedes AFPN aus dem Benutzersystem wird in ein ARFPN umgewandelt. Im Verarbeitungsschritt muss nun sichergestellt werden, dass die Regeln, die durch Aktionen im ARFPN ausgelöst werden, in die Verarbeitung einfliessen. Diese Regeln werden durch eine Ereigniserkennung bestimmt und müssen in das bestehende ARFPN eingehängt werden. Anschliessend werden die Bedingungen ausgewertet und die Aktionskomponenten mit Hilfe des Transaktionsmanagers ausgeführt.
  2. **Regelanalyse** (Rule Analysis System (RAS))
     Regeln, die im Benutzersystem definiert wurden, müssen ebenfalls in ein ARFPN umgewandelt und in die bestehende Regelmenge integriert werden. Es muss jedoch sichergestellt werden, dass keine unerlaubten Zyklen auftreten können, dass das Regelsystem confluent ist und dass die Regelmenge keine Konflikte aufweist.
  3. **Repository**
     Im Repository werden sämtliche Informationen über existierende Objekttypen, Benutzer, Regeln etc. abgespeichert. Damit ist es möglich, entsprechende Analysen und Auswertungen auf der Datenbank durchzuführen.
  4. **Transaktionssystem** (Transaction System)
     Um die Datenbankunabhängigkeit zu erreichen, muss der Transaktionsmanager in ALFRED integriert sein. Dieser stellt sicher, dass die entsprechenden Sperren gesetzt werden, und er ist zuständig für die Fehlerbehebung in Transaktionen, für das Recovery und für die eigentliche Ausführung von Befehlen auf der darunterliegenden Datenbank.

Die vier Subsysteme des Verarbeitungssystems werden im folgenden genauer betrachtet. Dabei werden das Repository und das Transaktionssystem in einem gemeinsamen Abschnitt zusammengefasst. Die Bezeichnungen für die Subsysteme beziehen sich dabei auf Abbildung 3.3.

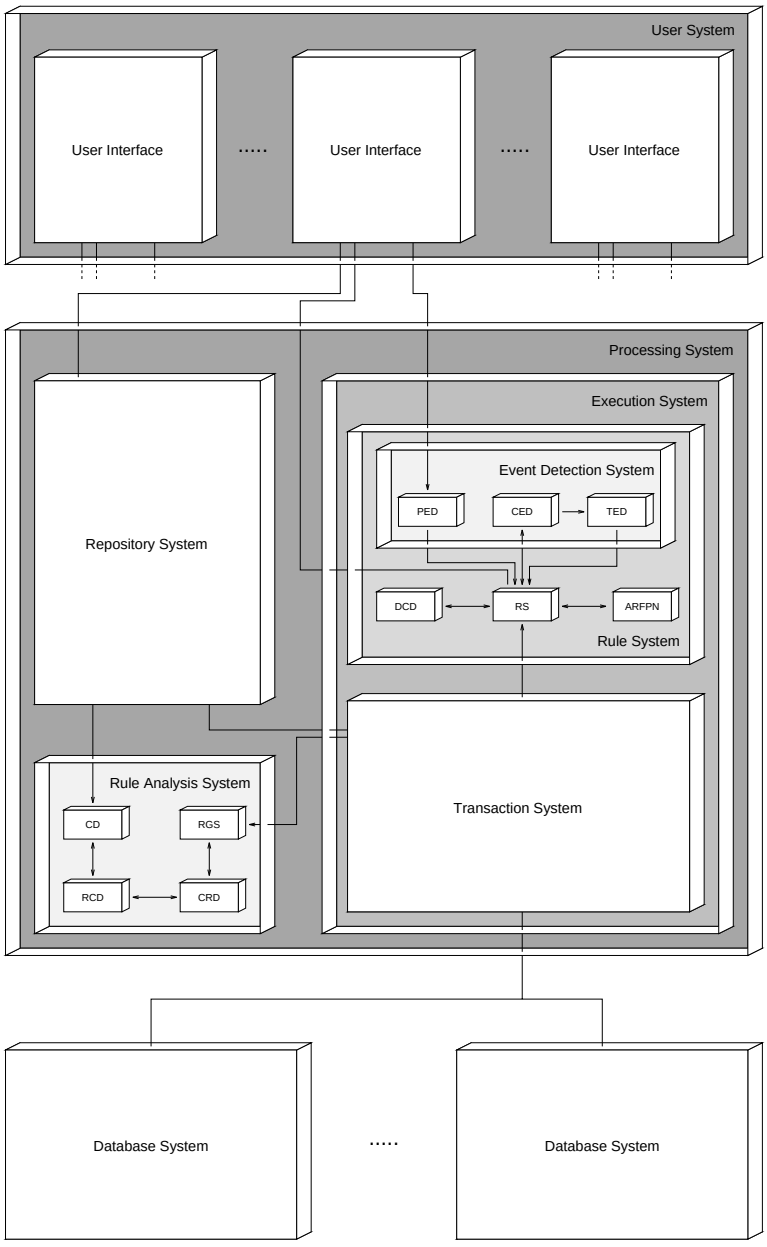

Abbildung 3.3: *Rule Analysis System* und *Rule System* von ALFRED

3.2.2 Regelverarbeitung

Das *Rule System* ist das eigentliche Kernsystem der Regelverarbeitung. Hier werden das ARFPN verarbeitet, Ereignisse erkannt und die Bedingungsauswertungen sowie die Ausführung von Aktionen gesteuert. Damit diese verschiedenen Aufgaben erfüllt werden können, wird das *Rule System*

in folgende Untersysteme aufgeteilt:

- **Ereigniserkennungssystem** (Event Detection System (EDS))
  Dieses System ist für die Erkennung von Ereignissen zuständig, dabei sind primitive Ereignisse, komplexe Ereignisse als auch Zeitereignisse zu berücksichtigen. Aus diesem Grund besteht das EDS aus folgenden Subsystemen:

  - **Primitive Ereigniserkennung** (Primitive Event Detection (PED))
    In der PED wird ein ankommendes AFPN untersucht. Falls in diesem AFPN Aktionen auftreten, die mit Ereignissen aus der Regelmenge übereinstimmen, so werden spezielle Transitionen in das AFPN eingefügt und eine Verbindung zwischen diesen Transitionen und den entsprechenden Ereignissen hergestellt. Durch diese Erweiterung wird das AFPN zu einem ARFPN.
  - **Zeitereigniserkennung** (Time Event Detection (TED))
    Dieses System ist für die Erkennung von Zeitereignissen zuständig. Die TED übernimmt die Funktion eines Weckers. Ereignisse, die auf Zeitpunkten oder Zeitspannen basieren (vgl. Abschnitt 6.2.4), werden der TED übergeben. Dabei wird der "Wecker" auf eine ganz bestimmte Zeit in der Zukunft gestellt. Wird dieser Zeitpunkt erreicht, so sendet die TED ein entsprechendes Signal an das *Rule System*.
  - **Komplexe Ereigniserkennung** (Complex Event Detection (CED))
    Dieses System hat die Aufgabe, komplexe Ereignisse mittels der in Abschnitt 6.2 aufgeführten Operatoren zu erkennen, und der *Rule Simulation* diejenigen Regeln zu signalisieren, die ausgelöst wurden. Mit Ausnahme der Zeitoperatoren können sämtliche Ereignisse durch die CED verarbeitet werden. Falls die CED auf ein Zeitereignis stösst, so wird dieses der *Time Event Detection* übergeben.

- **Regelsimulation** (Rule Simulation (RS))
  In der RS werden das ARFPN und eventuell ausgelöste Regeln verarbeitet. Je nach zu verarbeitender Regelkomponente werden dabei verschiedene Subsysteme angestossen. Ereignisse werden im EDS erkannt und verarbeitet, Bedingungen und Aktionen werden an das *Transaction System* weitergegeben. Wenn ein ARFPN komplett abgearbeitet ist, so erhält das *User System* eine entsprechende Meldung. Somit kann sichergestellt werden, dass die Benutzereingaben sequentiell erfolgen.

- **ARFPN Generierung** (ARFPN Generation (ARFPN))
  In diesem System werden ausgelöste Regeln aus der bestehenden Regelmenge kopiert und in das aktuelle ARFPN eingefügt. Beim Einfügen muss dabei auf die richtigen Verbindungen (u.a. wegen Kopplungsmodi und Prioritäten) geachtet werden.

3.2.3 Regelanalyse

Die Regelanalyse besteht aus den vier Subsystemen Regelgenerierung (*Rule Generation System (RGS)*), Konflikt- und Redundanzerkennung (*Conflict and Redundancy Detection (CRD)*), Zyklenerkennung (*Rule Cycle Detection (RCD)*) und Konfluenzerkennung (*Confluence Detection (CD)*). Eine neu einzuführende oder zu ändernde Regel gelangt vom Benutzersystem durch die Regelsimulation zum Transaktionssystem und sodann zur Regelanalyse. Das RGS generiert daraufhin die Regel als ARFPN. Danach wird die neue Regelmenge in der CRD auf Konflikte und Redundanzen hin untersucht. Anschliessend wird in der RCD untersucht, ob durch diese Regel ein Zyklus im Regelwerk entsteht. Wenn dies der Fall ist, so wird der Zyklus auf seine Terminierung überprüft. Falls die neue Regel keine unzulässigen Zyklen erzeugt, so wird in der CD überprüft, ob das neu entstandene Regelwerk noch confluent ist. Wenn dies der Fall ist, so wird die Regel in das Repository aufgenommen, andernfalls wird die Regel verworfen.

3.2.4 Repository und Transaktionssystem

Im *Repository System* werden alle notwendigen Informationen abgelegt, damit eine umfangreiche Analyse stattfinden kann und damit Informationen über Regeln, Objekttypen, Benutzer etc. in der Schicht gespeichert sind.

Das *Transaction System* nimmt die auszuführenden Befehle vom *Rule System* entgegen und versucht, diese unter Berücksichtigung von existierenden Datensperren auszuführen. Ebenso muss hier garantiert werden, dass die Befehle in den richtigen Transaktionen ablaufen, dass die Regelverarbeitung im richtigen Bezug zur regelauslösenden Transaktion steht (Kopplungsmodi), und dass bei allfälligen Fehlern die Transaktionen abgebrochen werden können und die geänderten Daten zurückgesetzt werden. Das Transaktionssystem stellt sicher, dass sich die Datenbank immer in einem konsistenten Zustand befindet. Das *Rule System* erhält nach der Verarbeitung des Befehls die nötigen Informationen über den Stand der Ausführung zurück.

3.3 Regelmodellierung

In den folgenden Abschnitten wird erklärt, welche Arten von Ereignissen, Bedingungen und Aktionen in ALFRED unterstützt werden.

3.3.1 Ereignisse

Eine Regel wird durch das Eintreten eines bestimmten Ereignisses ausgelöst. Dieses Ereignis kann sowohl *primitiv*, *sprachlich* als auch *komplex* sein (vgl. Abbildung 3.4).

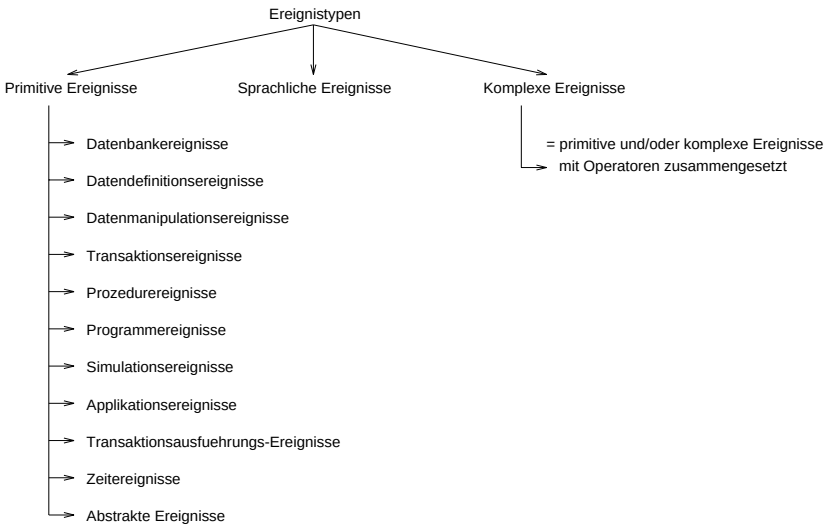

Abbildung 3.4: Arten von Ereignissen

3.3.1.1 Primitive und sprachliche Ereignisse

In der folgenden Liste stehen die primitiven Ereignisse sowie das sprachliche Ereignis von ALFRED, dabei steht `<name>` stellvertretend für einen Identifikator (z.B. Tabellenname, Regelnummer).

- **Datenbankereignisse**
  Datenbankereignisse werden durch Befehle ausgelöst, die zur Erzeugung und zur Löschung von Datenbanken dienen. Folgende zwei Datenbankereignisse können in ALFRED definiert werden (Auslösungszeitpunkte pre und post):
  - `create database`
  - `destroy database <name>`
- **Datendefinitionsereignisse**
  Diese Ereignisse werden durch Befehle ausgelöst, die zur Erzeugung, Löschung und Manipulation von Objekttypen, Benutzern, Prozeduren etc. verwendet werden. In ALFRED können u.a. die folgenden Datendefinitionsereignisse (Auslösungszeitpunkte pre und post) definiert werden (für eine vollständige Auflistung vgl. Anhang A):
  - Erzeugung von Entitäten, Benutzern, Regeln etc.
    `create`
    - `entity`
    - `user`
    - `rule`
  - Änderung und Löschung von Entitäten, Benutzern, Regeln etc.
    `alter / destroy`
    - `entity <name>`
    - `user <name>`
    - `rule <name>`

  Regeln, die durch Datendefinitionsereignisse ausgelöst werden, können z.B. zur Überwachung von Privilegien eingesetzt werden.
- **Datenmanipulationsereignisse**
  Befehle, die Daten erzeugen, manipulieren, selektieren oder löschen, sind die Auslöser dieser Ereignisse. Folgende Datenmanipulationsereignisse, die mit Auslösungszeitpunkt *pre* oder *post* und mit Auslösungsgranularität *set* oder *instance* definiert werden können, werden in ALFRED unterstützt:
  - `retrieve from <name>`
  - `insert in <name>`
  - `update on <name>`
  - `delete from <name>`

  Regeln, die durch Datenmanipulationsereignisse ausgelöst werden, können z.B. zur Sicherung von Integritätsbedingungen eingesetzt werden.
- **Transaktionsereignisse**
  Transaktionsereignisse werden durch Transaktionsbefehle ausgelöst. In ALFRED werden unterstützt (Auslösungszeitpunkte pre und post):
  - `begin of transaction`
  - `end of transaction`
  - `commit of transaction`

  Regeln, die durch Transaktionsereignisse ausgelöst werden, können z.B. zur Überprüfung von komplexen Integritätsbedingungen verwendet werden.

- **Prozedurereignisse**
  Ein solches Ereignis tritt ein, wenn eine Datenbankprozedur oder eine Applikationsprozedur aufgerufen wird (Auslösungszeitpunkte pre und post).
  - `stored procedure <name>`
- **Programmereignisse**
  Ein Programmereignis tritt ein, bevor oder nachdem ein in der Datenbank definiertes Programm aufgerufen wird (pre resp. post). Dieses Programm kann Zugriffe auf die Datenbank enthalten, dann wird es als eingebettetes Programm (embedded program) bezeichnet, oder es enthält keine Datenbankzugriffe und wird Programm genannt.
  - `program <name>`
  - `embedded program <name>`
- **Simulationsereignisse**
  Ein Simulationsereignis tritt z.B. ein, wenn eine vorher definierte Simulation gestartet wird (pre und post).
  - `execute simulation <name>`
- **Applikationserereignisse**
  Ein Applikationsereignis tritt ein, wenn eine Benutzerapplikation gestartet wird (pre und post).
  - `execute application <name>`
- **Transaktionsausführungs-Ereignisse**
  Ein solches Ereignis tritt ein, wenn eine zuvor definierte Transaktion ausgeführt wird (pre und post).
  - `execute transaction <name>`
- **Zeitereignisse**
  Ein Zeitereignis wird durch einen vordefinierten Zeitpunkt bestimmt, zu welchem das Ereignis eintritt (z.B. `12/01/1997 @ 12:00:00`).
- **Abstrakte Ereignisse**
  Mit abstrakten Ereignissen bezeichnet man solche, die nicht vom Datenbanksystem erkannt werden können. Es sind dies z.B. Ereignisse eines externen Programms, oder Ereignisse ausserhalb des Computersystems. Diese Ereignisse müssen dem Datenbanksystem explizit bekannt gemacht werden, indem dem abstrakten Ereignis bei dessen Definition ein Name vergeben wird (z.B. Ereignis MaxTemp = Wenn die Temperatur von 20 Grad Celsius erreicht ist) und das Eintreten des Ereignisses dem System signalisiert wird.
  - `abstract event <name>`
- **Sprachliche Ereignisse**
  Mit Hilfe dieser Ereignisse in Textform, können Ereignisse modelliert werden, die nicht im Zusammenhang mit dem Informationssystem stehen (z.B. wenn der Vertrag unterzeichnet wird).

3.3.1.2 Komplexe Ereignisse

Durch die Kombination von primitiven und/oder komplexen Ereignissen mittels spezieller Operatoren, lassen sich *komplexe* Ereignisse bilden. Die Operatoren, die in ALFRED unterstützt werden, sind nachfolgend aufgelistet (vgl. Abbildung 3.5). Dabei bezeichnen E_1, E_2, E_3, E_i, E_b und E_e beliebige Ereignisse, und E_k bezeichnet das komplexe Ereignis, das durch den Operator definiert wird.

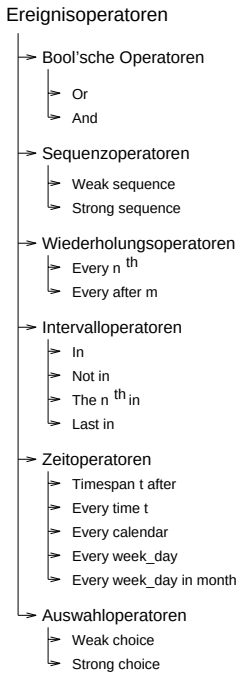

Abbildung 3.5: Die Ereignisoperatoren von ALFRED

- **Bool'sche Operatoren**
  Mit Hilfe dieser Operatoren lassen sich logische Verknüpfungen zwischen Teilereignissen modellieren.
  - **Or**
    Das Ereignis $E_k := E_1$ **or** E_2 tritt ein, wenn E_1 *oder* E_2 eintritt.
  - **And**
    Das Ereignis $E_k := E_1$ **and** E_2 tritt ein, wenn beide Teilereignisse, d.h. E_1 *und* E_2 eintreten. Die Reihenfolge des Eintretens der beiden Teilereignisse spielt dabei keine Rolle, d.h. E_k wird ausgelöst, wenn zuerst E_1 und dann E_2 eintritt als auch, wenn erst E_2 und dann E_1 eintritt.
- **Sequenzoperatoren**
  Die Sequenzoperatoren dienen dazu, ein Ereignis zu definieren, welches nur dann ausgelöst wird, wenn bestimmte Teilereignisse in einer vordefinierten Reihenfolge eintreten.
  - **Weak sequence**
    Das Ereignis $E_k :=$ **weak sequence**(E_1, E_2) tritt ein, wenn das Teilereignis E_1 vor dem Ereignis E_2 eintritt. Falls die Teilereignisse wiederum komplex sind, so dürfen Teilereignisse von E_2 eintreten, bevor E_1 eintritt.

  - **Strong sequence**
    Das Ereignis E_k := **strong sequence**(E_1, E_2) tritt ein, wenn das Teilereignis E_1 vor dem Ereignis E_2 eintritt. Die Semantik des Operators unterscheidet sich von der schwachen Sequenz erst dann, wenn die Teilereignisse E_1 und E_2 komplex sind. Die starke Sequenz verlangt, dass keine Teilereignisse des zweiten Ereignisses E_2 eintreten dürfen, bevor das erste Ereignis E_1 eingetreten ist.

- **Wiederholungsoperatoren**
  Mit Hilfe der Wiederholungsoperatoren lassen sich periodische Ereignisse modellieren.

  - **Every n$^{\text{th}}$**
    Das Ereignis E_k := **every n$^{\text{th}}$** E_1 tritt bei jedem n-maligen Eintreten des Ereignisses E_1 ein, d.h. beim n-ten, 2n-ten, 3n-ten Eintreten des Ereignisses E_1.
  - **Every after m**
    Das Ereignis E_k := **every after m** E_1 wird jedesmal dann signalisiert, nachdem m-Mal das Ereignis E_1 eingetreten ist und weitere E_1 Ereignisse eintreten. Folglich wird E_k beim (m+1)-ten, (m+2)-ten, ... Eintreten von E_1 signalisiert.

- **Intervalloperatoren**
  Diese Operatoren zeichnen sich dadurch aus, dass auf ein Ereignis (nachfolgend E_i) reagiert wird, welches in einem Intervall eintritt. Ein solches Intervall wird durch zwei Ereignisse definiert: einem Beginnereignis (nachfolgend E_b) und einem Endereignis (nachfolgend E_e).

  - **In**
    Das Ereignis E_k := E_i **in** (E_b, E_e) tritt genau dann ein, wenn das Ereignis E_i zwischen den zwei Ereignissen E_b und E_e eintritt. Das Ereignis E_k wird also signalisiert, wenn die Ereignisse in der Reihenfolge $(E_b..E_i..E_e)$ eintreten.
  - **Not in**
    Das Ereignis E_k := E_i **not in** (E_b, E_e) tritt genau dann ein, wenn die Ereignisse E_b und E_e in dieser Reihenfolge eintreten, ohne dass das Ereignis E_i dazwischen eintritt. Zwischen den Ereignissen E_b und E_e dürfen jedoch beliebige andere Ereignisse vorkommen.
  - **The n$^{\text{th}}$ in**
    Das Ereignis E_k := **the n$^{\text{th}}$** E_i **in** (E_b, E_e) tritt genau dann ein, wenn das Ereignis E_i zum n-ten Mal im Intervall eintritt. E_k wird *nicht* mehr ausgelöst, wenn E_i 2n-mal, 3n-mal etc. im Intervall eintritt.
  - **Last in**
    Das Ereignis E_k := **last** E_i **in** (E_b, E_e) wird signalisiert, wenn das Ende des Intervalls (E_e) erreicht wird und das Ereignis E_i mindestens einmal darin eingetreten ist.

- **Zeitoperatoren**
  Mit Hilfe der Zeitoperatoren lassen sich Zeitverzögerungen und periodische Zeitereignisse modellieren.

  - **Timespan t after**
    Das Ereignis E_k := **timespan t after** E_1 wird ausgelöst, nachdem das Ereignis E_1 eingetreten und eine Zeitspanne t vergangen ist. Die Auslösung von E_k erfolgt also mit einer Verzögerung von t Zeiteinheiten.
  - **Every time t**
    Das periodische Ereignis E_k := **every time t** *beginning at X* wird immer dann ausgelöst, wenn die gewünschte Zeit (z.B. 12 Uhr mittags) eintritt. Das Zeitereignis wird aber erst nach dem Zeitpunkt X registriert, wobei X entweder den Zeitpunkt der Definition der Regel (now) oder den Eintrittszeitpunkt eines Ereignis darstellt (z.B. E_1).

  - **Every calendar**
    Das komplexe Ereignis E_k := **every n calendar** *beginnning at X* wird dann ausgelöst, wenn *calendar* n-mal eingetreten ist (Zeitpunkt X analog zu *Every time t*). Der Ausdruck *calendar* bezeichnet dabei (Tage, Wochen, Monate etc.). Mit Hilfe dieses Operators kann also ein komplexes Ereignis definiert werden, welches z.B. jede 4. Woche ausgelöst wird.
  - **Every week_day**
    Ein komplexes Ereignis E_k := **every n week_day** *at time, beginning at X* tritt jeden n-ten Wochentag ($week\_day \in \{Montag, \ldots, Samstag\}$) zu einer bestimmten Zeit *time* ein (Zeitpunkt X analog zu *Every time t*). Somit kann z.B. ein Ereignis definiert werden, welches jeden 2. Montag um 15.00 Uhr eintritt.
  - **Every week_day in month**
    Mit Hilfe dieses Operators kann ein komplexes Ereignis E_k := **every n days** *in month at time, beginning at X* definiert werden. Dieses Ereignis tritt jeden n-ten Wochentag ($days \in \{day, Montag, \ldots, Samstag\}$) *im Monat* zu einer bestimmten Zeit ein (X analog zu *Every time t*). Somit kann z.B. ein Ereignis definiert werden, welches jeden 3. Tag im Monat um 12.15 Uhr eintritt.

- **Auswahloperatoren**
  Die folgenden zwei Auswahloperatoren werden in ALFRED unterstützt, damit eine einfachere Definition von Ereignissen möglich ist. Komplexe Ereignisse, die auf einem Auswahloperator basieren, treten dann ein, wenn eine bestimmte Anzahl Ereignisse n aus einer vordefinierten Ereignismenge M eintreten. Mit M wird eine Menge von Ereignissen, wie z.B. (E_1, E_2, E_3) bezeichnet. Diese Auswahloperatoren können bei der Modellierung mit Hilfe von bool'schen Operatoren und Wiederholungsoperatoren realisiert werden.

  - **Weak choice**
    Das Ereignis E_k := **weak choice** *n of M* tritt genau dann ein, wenn n Ereignisse aus der Ereignismenge M eingetreten sind. Bei der *weak choice* wird auch das Eintreten von mehreren gleichen Ereignissen berücksichtigt. Zum Beipiel wird E_k := **weak choice** *2 of* (E_1, E_2) dann ausgelöst, wenn zweimal E_1, zweimal E_2 oder E_1 und E_2 in beliebiger Reihenfolge eintreten.
  - **Strong choice**
    Das Ereignis E_k := **strong choice** *n of M* tritt genau dann ein, wenn n Ereignisse aus der Ereignismenge M eingetreten sind. Die Teilereignisse müssen bei der *strong choice* jedoch paarweise verschieden sein.

3.3.2 Bedingungen

Eine Bedingung beschreibt, was zu einem bestimmten Zeitpunkt zu überprüfen ist. Bedingungen sind entweder *primitiv*, *sprachlich* oder *komplex*. Letztere werden aus primitiven und/oder komplexen Bedingungen mittels den Operatoren *not, and* und *or* gebildet (vgl. Abbildung 3.6).

Die folgende Liste enthält alle von ALFRED unterstützten primitiven Bedingungen sowie die sprachliche Bedingung und verdeutlicht deren Semantik:

- **Prädikat**
  Ein Prädikat setzt sich aus zwei Operanden und einem Vergleichsoperator zusammen. Operanden können dabei Konstanten, Werte oder Ausdrücke sein. Mögliche Vergleichsoperatoren sind `<`, `<=`, `=`, `>=`, `>` und `!=`.

- **Abfrage**
  Eine Abfrage wird mittels eines `retrieve` Befehls auf einer oder mehreren Datenbankrelationen definiert. Eine Bedingung auf der Basis einer Abfrage ist genau dann erfüllt, wenn die Abfrage nicht leer ist, d.h. mindestens einen Datensatz liefert.

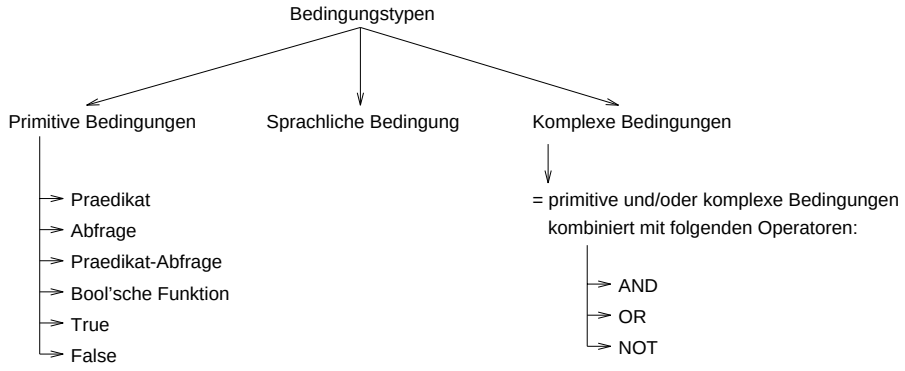

Abbildung 3.6: Typen von Bedingungen

- **Prädikat-Abfrage**
  Dies ist eine Kombination von Prädikaten und Abfragen. Mit einer Prädikat-Abfrage kann überprüft werden, ob sich ein Attribut in einer Menge von Attributen befindet. Diese Menge kann explizit vorgegeben sein, oder sie wird durch eine Abfrage definiert.

- **Bool'sche Funktion**
  Diese Funktion wird in einer speziellen Programmiersprache geschrieben (ALFRED Programming Language) und kann z.B. auch Datenselektionsbefehle enthalten. Eine solche Funktion muss immer einen Wert wahr (TRUE) oder falsch (FALSE) zurückliefern.

- **True und False**

- **Sprachliche Bedingung**
  Um Regeln modellieren zu können, die nur zum Teil oder überhaupt nicht mit dem Informationssystem in Verbindung stehen, ist es nötig, dass Bedingungen in Textform eingegeben werden können. Damit ist es möglich, beliebige Bedingungen anzugeben, die jedoch vom System nicht ausgewertet werden können. Ein Beispiel einer solchen Bedingung ist *"Sind die Dokumente unterzeichnet."*

3.3.3 Aktionen

In der Aktionskomponente der Regel wird festgelegt, was in einer erkannten Situation gemacht werden soll. Aktionen können *primitiv*, *sprachlich* oder *komplex* sein (vgl. Abbildung 3.7).

3.3.3.1 Primitive und sprachliche Aktionen

Die folgende Liste enthält alle primitiven Aktionen und die sprachliche Aktion, die in ALFRED unterstützt werden. Zusätzlich werden die einzelnen Aktionen kurz erläutert.

- **Retrieve**
  Mit dieser Aktion werden Daten aus der Datenbank selektiert.

- **Insert**
  Diese Aktion ermöglicht das Einfügen von Datensätzen.

- **Update**
  Mit dieser Aktion können Datensätze manipuliert werden.

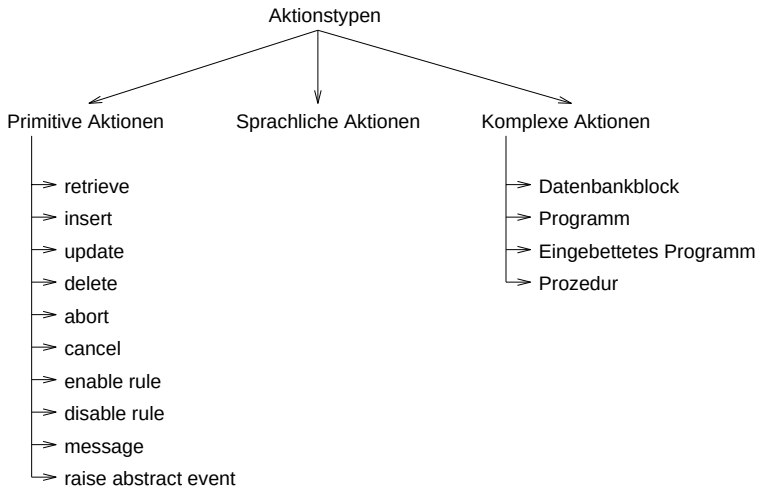

Abbildung 3.7: Typen von Aktionen

- **Delete**
  Diese Aktion ermöglicht das Löschen von Datensätzen.
- **Abort**
  Mit dieser Aktion kann eine laufende Transaktion abgebrochen werden.
- **Cancel**
  Mit dieser Aktion kann die vorherige Aktion abgebrochen und zurückgesetzt werden.
- **Enable rule**
  Diese Aktion gestattet es, eine beliebige Regel in der Regelmenge zu *aktivieren*, d.h. sie zur Verarbeitung frei zu geben.
- **Disable rule**
  Mit dieser Aktion kann eine beliebige Regel in der Regelmenge *deaktiviert* werden, d.h. sie wird nicht verarbeitet.
- **Message**
  Mit dieser Aktion kann über die Menüoberfläche eine Meldung an den Benutzer gegeben werden.
- **Raise abstract event**
  Diese Aktion ermöglicht es, abstrakte Ereignisse innerhalb des Systems auszulösen.
- **Sprachliche Aktion**
  Diese Aktion (in Textform) wird verwendet, um Aktionen darzustellen, die nicht im Informationssystem verarbeitet werden können.

3.3.3.2 Komplexe Aktionen

In ALFRED werden nicht nur primitive Aktionen unterstützt sondern auch komplexe. Von komplexen Aktionen spricht man, wenn es sich um einen der folgenden Punkte handelt:

- **Datenbankblock**
  Innerhalb eines Datenbankblocks können *mehrere* primitive Aktionen auftreten. Zudem ist es möglich, explizite Transaktionen zu definieren, sowie mögliche Parallelisierungen vorzunehmen.

- **Programm**
  Ein Programm wird in einer speziellen Programmiersprache geschrieben (ALFRED Programming Language). Diese Sprache ermöglicht es dem Benutzer, Applikationsprogramme in der aktiven Schicht zu schreiben. Das Programm kann hierbei *nicht* auf die Datenbank zugreifen.

- **Eingebettetes Programm**
  Diese Art von Programm wird in AEPL (ALFRED Embedded Programming Language) geschrieben. Diese Sprache stellt eine Erweiterung der ALFRED Programming Language dar, indem sie es dem Benutzer zusätzlich ermöglicht, auf die Datenbank zuzugreifen.

- **Prozedur**
  Eine Prozedur wird ähnlich wie ein Programm oder eingebettetes Programm in AEPL oder in der ALFRED Programming Language geschrieben, kann aber im Gegensatz zu den Programmen nicht eigenständig ablaufen. Diese Prozedur kann im Aktionsteil einer Regel mit entsprechenden Parametern aufgerufen werden.

3.4 Regelverarbeitung

In diesem Kapitel wird grob erläutert, wie die Regelverarbeitung in ALFRED realisiert ist und welche Systeme (vgl. Abbildung 3.8) für die Verarbeitung zuständig sind. Eine genauere Beschreibung der Regelverarbeitung resp. der Simulation von Abläufen in ALFRED, ist in Kapitel 8 zu finden.

Damit ALFRED das gewünschte aktive Verhalten zeigt, müssen vorab Regeln definiert werden. Sind diese Regeln oder ist diese Regelmenge einmal definiert, so kann der Benutzer über das Benutzermenü Befehle oder Befehlsketten durch die aktive Schicht an die darunterliegende Datenbank senden.

Angenommen, der Benutzer führt einen `insert`-Befehl auf der Datenbank aus. Dieser Befehl wird in der *AFPN Generation* (in Abb. 3.8 als AFPN abgekürzt) in ein AFPN umgesetzt, wobei der `insert`-Befehl durch eine Transition realisiert wird. Dieses AFPN gelangt nun in das *Processing System*. Als erstes kommt das AFPN zum *Event Detection System*, genauer in die *Primitive Event Detection* (PED). Dort wird untersucht, ob in den definierten Regeln ein Ereignis existiert, welches durch ein `insert` ausgelöst wird. Wenn dies der Fall ist, so werden spezielle Verbindungstransitionen *vor* (pre) oder *nach* (post) der Aktionstransition eingefügt und mit dem entsprechenden Ereignis verbunden. Dieses modifizierte AFPN (das sogenannte ARFPN), gelangt nun in die *Rule Simulation* (RS). In diesem System wird das Petrinetz verarbeitet, indem Token durch das Netz transportiert und entsprechende Subsysteme angestossen werden.

- Die Ereigniserkennung wird an die Systeme *Complex Event Detection* (CED) und *Time Event Detection* (TED) delegiert. Alle ausgelösten Regeln werden der *Rule Simulation* (RS) mitgeteilt.

- Die Auswertung von Bedingungen wird durch das *Transaction System* realisiert. Das Resultat der Bedingungsauswertung wird an die *Rule Simulation* (RS) zurückgegeben.

- Die Ausführung von Aktionen wird ebenfalls an das *Transaction System* delegiert. Notwendige Informationen über die Aktionsausführung werden der *Rule Simulation* (RS) zurückgegeben.

- Falls während der Verarbeitung Regeln ausgelöst wurden, so werden die entsprechenden Regeln in der *ARFPN Generation* (in Abb. 3.8 als ARFPN abgekürzt) aus der Regelmenge herauskopiert und in das bestehende ARFPN integriert. Dabei wird auf die korrekten Verbindungen bezüglich Kopplungsmodi, Prioritäten etc. geachtet.

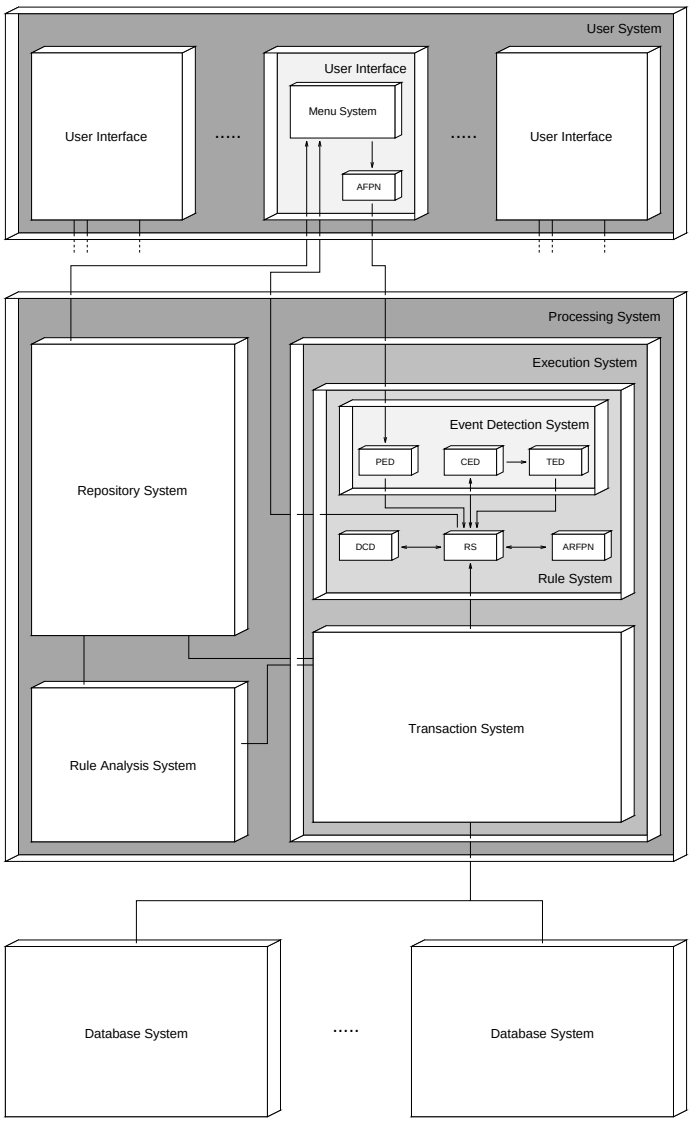

Abbildung 3.8: Die Subsysteme von ALFRED

Wenn der Befehl und alle durch den Befehl ausgelösten Regeln korrekt abgearbeitet wurden, so erhält der Benutzer durch die RS eine entsprechende Meldung.

Kapitel 4

Vergleich von Modellen zur Regelbeschreibung

Bei der Entwicklung eines aktiven Datenbanksystems stellt sich mitunter die Frage, welches Modell für die Abbildung von Regeln und für die Darstellung der gewünschten Funktionalität gewählt werden soll. An dieses Modell werden je nach verfolgtem Ziel des zu entwickelnden aktiven Datenbanksystems verschiedene Anforderungen gestellt. Zum einen sind dies Anforderungen, die durch die Regelmodellierung entstehen, zum anderen aber auch Anforderungen, die durch die Regelverarbeitung entstehen.

In diesem Kapitel werden einige ausgewählte Modelle verglichen und auf ihre Eignung zur Modellierung von Regeln in ALFRED überprüft. Dazu werden zuerst die ALFRED-spezifischen Anforderungen an ein Modell erläutert. Im Anschluss daran, werden die einzelnen Modelle kurz vorgestellt und verglichen.

4.1 Anforderungen

An das Modell für die Darstellung von Regeln werden je nach Zielsetzung unterschiedliche Anforderungen gestellt. In ALFRED sollen alle Anforderungen, die in Abschnitt 3.1 erläutert wurden, erfüllt werden. Davon sind folgende Anforderungen besonders wichtig:

- Darstellbarkeit der Ereignisse
- Darstellbarkeit der Bedingungen
- Darstellbarkeit der Aktionen
- Nahtloser Übergang zwischen Regelkomponenten
- Darstellbarkeit der ausführungsbezogenen Faktoren
- Möglichkeit der Regelverarbeitung

Diese Anforderungen sind in dieser Form sehr grob formuliert. Um die Darstellbarkeit der einzelnen Regelteile und Regelfunktionalitäten tatsächlich garantieren zu können, sind je nach Regelteil und -funktionalität spezifische Anforderungen wichtig. Diese werden in den nun folgenden Unterabschnitten kurz erläutert.

4.1.1 Regelmodellierung

Um sämtliche Regelkomponenten darstellen zu können, müssen Anforderungen an die Darstellung der Ereignisse, Bedingungen und Aktionen gestellt werden.

4.1.1.1 Ereignisse

Um den Ereignisteil darstellen zu können, muss es möglich sein, mit Hilfe des Modells primitive wie auch komplexe Ereignisse zu beschreiben. Aus diesem Grund ist es notwendig, dass sämtliche Ereignisoperatoren dargestellt werden können.

Doch nicht nur die *Darstellung* von Ereignissen soll möglich sein, sondern auch deren effiziente *Erkennung*. Eine einfache Variante, komplexe Ereignisse zu erkennen, ist die folgende: Jedesmal wenn ein Teilereignis eintritt, so wird jedes komplexe Ereignis, welches das Teilereignis enthält, daraufhin untersucht, ob es eintritt oder nicht. Das System müsste sich also eine Ereignisgeschichte anlegen und bei jedem Einreten eines Ereignisses würden die komplexen Ereignisse mit der Ereignisgeschichte verglichen (pattern-matching). Es ist offensichtlich, dass diese einfache Variante der Ereigniserkennung nicht effizient sein kann. Aus diesem Grunde muss das Modell eine Möglichkeit bieten, den aktuellen Stand der Ereigniserkennung zwischenzuspeichern, d.h. eine schrittweise Erkennung von Ereignissen zu ermöglichen.

4.1.1.2 Bedingungen

Um alle Bedingungen darstellen zu können, muss es möglich sein, sowohl alle primitiven Bedingungen als auch alle komplexen Bedingungen zu modellieren (vgl. Abschnitt 3.3.2). Komplexe Bedingungen sollen in "atomare" Bedingungen zerlegt und mittels geeigneter Operatoren (*and, or, not*) und Strukturen (*Sequenz, Auswahl, Wiederholung*) im Modell dargestellt werden können.

Da bei komplexen Bedingungen die Möglichkeit besteht, die Auswertung durch geeignete Massnahmen (z.B. parallele Auswertung von Teilbedingungen) zu optimieren, soll das Modell die Möglichkeit bieten, diese parallele Ausführung darzustellen.

4.1.1.3 Aktionen

Mit dem zu wählenden Modell sollen sämtliche Aktionen darstellbar sein. So soll es möglich sein, Einzelaktionen sowie die sequentielle und die parallele Verkettung von mehreren Aktionen darzustellen. Damit auch Programme modelliert werden können, müssen diese in Einzelaktionen aufgesplittet und mit den nötigen Strukturelementen (*Sequenz, Auswahl, Wiederholung*) verbunden werden. Aus diesem Grund muss das Modell Möglichkeiten zur Darstellung dieser Strukturen bieten.

Da durch die Ausführung von Aktionen Ereignisse im Regelwerk ausgelöst werden können, muss es möglich sein, diese Abhängigkeit zwischen Aktionen und Ereignissen mit Hilfe dieses Modells zu beschreiben.

4.1.2 Regelausführung

Das Modell soll nicht nur die Darstellung aller Regelkomponenten ermöglichen, sondern auch deren Ausführung. Dazu werden die folgenden Anforderungen aufgestellt:

- **Semantik**
  Das Modell soll eine wohldefinierte Semantik aufweisen.

- **Nebenläufigkeit**
  Das Modell soll die Darstellung von Nebenläufigkeit unterstützen.

- **Darstellung von Instanzen**
  In einem ADBS können Ereignisse beliebig oft eintreten. Dies bedeutet, dass während der Verarbeitung verschiedene "Instanzen" von Ereignissen existieren. Mit Hilfe des Modells soll es möglich sein, Instanzen von Ereignissen zu repräsentieren.

- **Verarbeitung/Simulation**
  Das Modell soll eine Verarbeitung von Regeln resp. eine Simulation der Regelverarbeitung erlauben.

- **Parameter**
  Damit bei der Regelverarbeitung auf gewisse Informationen zugegriffen werden kann (z.B. gelöschte Datensätze), müssen bei der Ereigniserkennung Parameter gebunden und zur Bedingungsauswertung resp. Aktionsverarbeitung weitergereicht werden. Eine solche Parameterübergabe muss das Modell ermöglichen.

- **Parameterkontexte**
  Alle Parameterkontexte, die in Abschnitt 2.3.2.6 definiert wurden, müssen dargestellt werden können.

- **Instanz- und mengenorientierte Regeln**
  Das Modell soll es ermöglichen, instanzorientierte und mengenorientierte Auslösungen von Regeln darzustellen.

- **Pre- und post-Ausführung**
  Die Zeitpunkte der Ereignisauslösung sollen aus dem Modell ersichtlich sein.

- **Kopplungsmodi**
  Mit Hilfe des Modells sollen alle sechs Kopplungsmodi (vgl. Abschnitt 2.3.2.5) sowohl dargestellt als auch während der Verarbeitung berücksichtigt werden können.

4.1.3 Einheitlichkeit und Analysierbarkeit

Neben den Anforderungen an die Darstellbarkeit der einzelnen Regelkomponenten und an die Ausführung müssen folgende Eigenschaften erfüllt sein:

- **Einheitliches Modell**
  Sämtliche Regelkomponenten müssen mit dem gleichen Modell dargestellt werden können. Damit eine effiziente Verarbeitung der Regeln erfolgen kann, muss ein nahtloser Übergang zwischen den einzelnen Komponenten gewährleistet sein.

- **Analysemöglichkeit**
  Das Modell soll die Analyse von Regeln ermöglichen. Dazu gehören u.a. die Zyklenerkennung und die Überprüfung der Confluence.

4.2 Modelle und Methoden

In diesem Abschnitt werden ausgewählte Modelle und Methoden kurz vorgestellt. Es sind dies Modelle, die in Prototypen von aktiven Datenbanken bereits eingesetzt, jedoch in den meisten Fällen nur zur Darstellung und Erkennung von Ereignissen verwendet werden.

In allen hier vorgestellten Modellen ist die Erkennung von Ereignissen möglich. Aus diesem Grund werden die einzelnen Modelle anhand eines komplexen Beispielereignisses erläutert. Dieses komplexe Ereignis $E_k := E_1$ **and** E_2 besteht aus zwei nicht näher spezifizierten primitiven Teilereignissen E_1 und E_2 und ist mit Hilfe des Konjunktionsoperators (and) gebildet.

4.2.1 Integer Array

Eine Möglickeit zur effizienten Ereigniserkennung bietet der in Composite Event DEtector (CEDE) [Eri93] entwickelte *Integer Array Algorithm*. Diese Methode bedient sich einer zweidimensionalen Matrix von Integerzahlen. Die Anzahl der Felder der Matrix wird durch die Arität des komplexen Ereignisses bestimmt. Im komplexen Beispielereignis E_k beträgt die Arität zwei (das komplexe Ereignis setzt sich aus zwei Teilereignissen zusammen), d.h. die Matrix besteht aus zwei Feldern. Das erste Feld repräsentiert das primitive Ereignis E_1, das zweite Feld E_2. Jedesmal wenn ein primitives Ereignis erkannt wird, so muss der entsprechende Wert in der Matrix um eins erhöht werden und die entsprechenden Parameter werden gespeichert. Das komplexe Ereignis wird genau dann ausgelöst, wenn alle Matrixfelder einen Wert > 0 enthalten. In diesem Fall wird für jedes am komplexen Ereignis beteiligte primitive Ereignis der entsprechende Zähler um eins verringert.

Die Auswertung des Arrays, sowie die Weitergabe von Parametern, die bei jedem Eintreten eines primitiven Ereignisses gebunden werden, geschieht durch zusätzliche Programme.

Dieser Ansatz eignet sich gut für die Ereigniserkennung. Er eignet sich jedoch nicht für die *Darstellung* von Ereignissen sowie die Darstellung und Ausführung von Bedingungen und Aktionen.

4.2.2 Endlicher Automat

Eine weitere Möglickeit zur Darstellung und Erkennung von komplexen Ereignisses bietet der endliche Automat. Ein endlicher Automat ist auf die Erkennung von regulären Ausdrücken "spezialisiert". Da bei der Definition von komplexen Ereignissen eine Ereignisalgebra definiert werden muss, ist es naheliegend, eine Äquivalenz zwischen eben dieser Algebra und den regulären Ausdrücken zu suchen.

Sei $\mathcal{A}$ ein Alphabet, d.h. eine endliche Menge von Symbolen. Ein endlicher Automat EA über diesem Alphabet $\mathcal{A}$ ist definiert als:

Definition 1
Ein **endlicher Automat** *EA über dem Alphabet $\mathcal{A}$ ist ein 4-Tupel (Z,S,E,K) mit:*

1. *Z ist eine endliche Menge von Zuständen*
2. *S ist eine Untermenge von Z (Anfangszustände)*
3. *E ist eine Untermenge von Z (Endzustände)*
4. *K ist eine endliche Menge von Kanten, die Zustände miteinander verbinden. Es gilt:* $K \subset (Z \times \mathcal{A} \times Z)$

Abbildung 4.1 zeigt einen endlichen Automaten, der das komplexe Beispielereignis darstellt und erkennen kann.

Zu Beginn der Ereigniserkennung wird der endliche Automat in den Anfangszustand S versetzt. Falls ein Ereignis E_1 eintritt, so gelangt der Automat in den neuen Zustand 1. Analoges gilt für das Ereignis E_2 und Zustand 2. Weitere E_1 Ereignisse im Zustand 1 oder E_2 Ereignisse im Zustand 2 bewirken keine Zustandsänderung des Automaten. Wenn in Zustand 1 ein Ereignis E_2 oder im Zustand 2 ein Ereignis E_1 eintritt, so wechselt der Automat in den Endzustand (3). Das komplexe

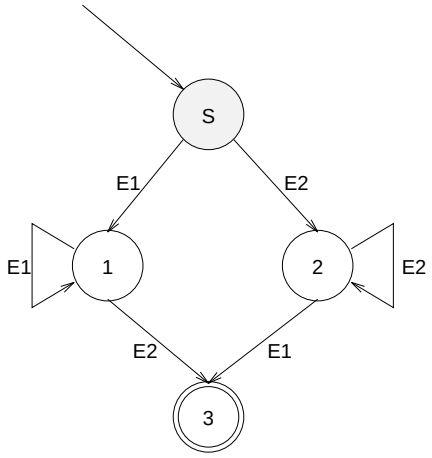

Abbildung 4.1: Ein endlicher Automat für E_1 **and** E_2

Ereignis ist somit eingetreten.

Endliche Automaten werden in Ode ([GJS93]) zur Erkennung von komplexen Ereignissen verwendet. Es wurden bislang jedoch keine Bedingungen und Aktionen und somit auch keine ausführungsbezogenen Faktoren mit Hilfe dieses Modells dargestellt.

4.2.3 Bäume

Die Darstellung und Erkennung von komplexen Ereignissen kann mit Hilfe von Bäumen realisiert werden. Auch hier wird, wie beim endlichen Automaten, die Äquivalenz zwischen Ereignisalgebra und regulären Ausdrücken verwendet.

Definition 2 (rekursiv)
Ein **Baum** *ist eine endliche Menge* $\mathcal{B} := \{W\} \cup T_1 \cup T_2 \cup \ldots \cup T_r$ $(r \geq 0)$ *mit:*

- *W ist ein ausgezeichnetes Element (***Wurzel***)*
- T_i *ist ein Baum (Unterbaum)* $\quad (\forall i \in [1 \ldots r])$
- *Die Vereinigung von W und allen* T_i *ist disjunkt.*

Definition 3
Falls $x \in \mathcal{B}$ *keine Nachfolger hat* $(\forall i : T_i = \emptyset)$*, so heisst x* **Blattknoten** *andernfalls heisst x* **Nichtblattknoten** *oder innerer Knoten.*

Ein Baum $\mathcal{B}$ besteht folglich aus einer Wurzel, einer endlichen Anzahl von Nichtblattknoten und einer endlichen Anzahl von Blattknoten. Jeder Baum kann grafisch repräsentiert werden, indem jeder Blatt- und Nichtblattknoten sowie die Wurzel als Kreise dargestellt werden. Von der Wurzel des Baumes werden Kanten zu den Wurzeln der Unterbäume gezogen.

Ein komplexer Ereignisausdruck wird wie folgt in einen Baum resp. Ereignisbaum übersetzt. Der äusserste Ereignisoperator bildet die Wurzel. Alle inneren Knoten stellen die im Ausdruck enthaltenen Suboperatoren dar. Primitive Ereignisse stehen in den Blattknoten. Jeder Teilbaum des Ereignisbaums ist ein primitives oder komplexes Teilereignis des ganzen.

Die einzelnen Ereignisbäume können sodann in einem Ereignisgraphen zusammengefasst wer-

den, um eine Menge von komplexen Ereignissen erkennen zu können. Falls ein primitives Ereignis eintritt, so wird der dazugehörige Blattknoten aktiviert. Dieser wiederum aktiviert alle mit ihm verbundenen Nichtblattknoten. Wenn ein solcher Nichtblattknoten stimuliert wird, so muss eine Prozedur ausgeführt werden, die den gewünschten Operator realisiert und eventuell weitere Nichtblattknoten aktiviert. Falls die Wurzel W aktiviert wird, so ist das komplexe Ereignis eingetreten.

Abbildung 4.2 zeigt einen Baum, der das komplexe Beispielereignis darstellt und erkennen kann.

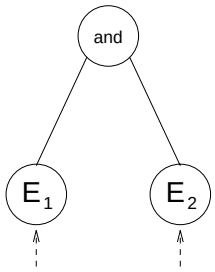

Abbildung 4.2: Der Ereignisbaum für E_1 **and** E_2

Falls die Ereignisse E_1 oder E_2 eintreten, so werden die entsprechenden Knoten aktiviert. Diese Aktivierung wird sodann an den übergeordneten Nichtblattknoten (and) weitergegeben. In diesem Knoten wird eine Prozedur ausgeführt, die in diesem Fall überprüft, ob beide Teilereignisse eingetreten sind. Wenn dies zutrifft, so wird das komplexe Ereignis ausgelöst.

Dieser Baumansatz wird in Snoop ([CM93]) verwendet, um komplexe Ereignisse darzustellen und zu erkennen. Dieses Modell wird jedoch nicht verwendet, um Bedingungen, Aktionen oder ausführungsbezogene Faktoren zu modellieren.

4.2.4 Extended Syntactic Tree

Dieses Modell stellt eine Erweiterung von Bäumen, genauer syntaktische Bäume, in Richtung Petrinetze dar. Hier wird wiederum die Äquivalenz von Ereignisausdrücken und regulären Ausdrücken ausgeschöpft. Mit Hilfe dieses Modells können komplexe Ereignisse dargestellt und erkannt werden.

Ein *Extended Syntactic Tree* ist ein Baum wie bereits in Abschnitt 4.2.3 vorgestellt. Mit Hilfe eines normalen Baumes ist es jedoch nur schwer möglich, Parameter von Ereignissen zu binden und weiterzugeben. Aus diesem Grund wurde der Baumansatz erweitert. Hinzu kommt ein *Token* (Marke), das von den Blättern des Baumes bis an die Wurzel weitergegeben wird. Wenn ein primitives Ereignis eintritt, so werden dessen Parameter in einem Token gespeichert und in den entsprechenden Knoten gelegt. Daraufhin wird der Knoten das Token zum übergeordneten Knoten weiterreichen, der dann entscheidet, was mit diesem Token geschehen soll. Hat ein Token die Wurzel erreicht, so wird das komplexe Ereignis ausgelöst.

Abbildung 4.3 zeigt einen *Extended Syntactic Tree*, der das komplexe Beispielereignis darstellt und erkennen kann.

Wenn das Ereignis E_1 eintritt, so werden die entsprechenden Parameter gebunden und in einem Token gespeichert. Dieses Token wird sodann in den Knoten E_1 gelegt (analoges gilt für Ereignis E_2). Das Token wird nun an den übergeordneten Knoten (die Wurzel des Baumes,*and*) weitergereicht. Die Wurzel entscheidet sodann anhand der Anzahl Token und der darin gespeicherten Parameter, ob das komplexe Ereignis eintritt oder nicht.

Dieses Modell wird in REACH [Deu94] zur effizienten Erkennung von komplexen Ereignissen verwendet. Bislang wurde jedoch nicht versucht, Bedingungen oder Aktionen mit diesem Modell darzustellen.

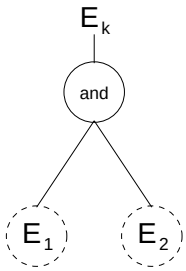

Abbildung 4.3: Extended Syntactic Tree für E_1 **and** E_2

4.2.5 Petrinetze

Eine weitere Möglichkeit, nicht nur zur Darstellung und Erkennung von Ereignissen, sondern auch zur Darstellung von Aktionen und Bedingungen, bieten die Petrinetze. In ihrer einfachsten Form (sog. Platz-/Transitions-Netze) lassen sie sich wie folgt definieren:

Definition 4
Die Struktur eines Platz-/Transitions-Petrinetzes ist ein 3-Tupel (P,T,A) mit:

1. *P ist eine endliche Menge von* **Plätzen**
2. *T ist eine endliche Menge von* **Transitionen**
3. $A \subset ((P \times T) \cup (T \times P))$ *ist eine endliche Menge von gerichteten* **Kanten**, *die Plätze und Transitionen miteinander verbinden.*

Meist werden Petrinetze graphisch dargestellt. Dabei werden Plätze durch Kreise, Transitionen durch Balken und Kanten durch Pfeile repräsentiert. Um nicht nur eine Struktur darzustellen, sondern auch z.B. Ereignisse zu erkennen, muss ein Petrinetz um eine Dynamik erweitert werden. Dies geschieht mit Hilfe sogenannter *Token* (Marken), die in die Plätze gelegt werden. Diese Belegung der Plätze mit Marken wird als *Markierung* des Petrinetzes bezeichnet. Wenn das Netz markiert ist, so können die Transitionen *feuern* oder *schalten*, d.h. die Token werden von Platz zu Platz transportiert (über die entsprechenden Transitionen). Dieser Transport geschieht nach fest vorgegebenen Schaltregeln.

Abbildung 4.4 zeigt ein einfaches Platz-/Transitions-Petrinetz, welches das komplexe Beispielereignis modelliert und auch erkennen kann.

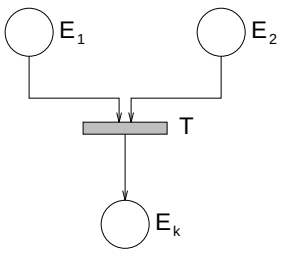

Abbildung 4.4: Ein Platz-/Transitions-Petrinetz für E_1 **and** E_2

Wenn ein Ereignis E_1 eintritt, so wird ein Token in den entsprechenden Platz gelegt (analog für

E_2). Die Transition T kann aufgrund von vorgegebenen Schaltregeln nur feuern, wenn in beiden Plätzen vor der Transition (E_1 und E_2) ein Token liegt. Wenn dies der Fall ist, so wird das Token aus den Plätzen entfernt und ein neues Token wird in den Platz E_k gelegt. Wenn der Platz E_k markiert wird, so ist das komplexe Ereignis eingetreten.

Mit diesem Modell lassen sich Ereignisse darstellen und erkennen. Bislang wurden im Kontext von ADBS jedoch keine Bedingungen und Aktionen mit Hilfe von einfachen Petrinetzen modelliert. Die Darstellung von Aktionen mit Petrinetzen, insbesondere konkurrente Aktionen, ist jedoch weit verbreitet.

4.2.6 Gefärbte Petrinetze

Eine erweiterte Form der Petrinetze stellen die gefärbten Petrinetze (coloured petri nets, vgl. [Jen92]) dar. Mit Hilfe dieser Netze ist es ebenfalls möglich, Komponenten von Regeln, d.h. Ereignisse, Bedingungen und Aktionen, darzustellen und ebenso zu erkennen resp. auszuführen. Gefärbte Petrinetze werden im Abschnitt 5.2 definiert (Definition 8).

Der Hauptunterschied zwischen einfachen Petrinetzen und gefärbten Petrinetzen liegt in der Möglickeit der gefärbten Petrinetze, komplexe Informationen, wie z.B. einen ganzen Datenbankzustand, in einem Token zu speichern. Dadurch ist es möglich, während der Erkennung von Ereignissen Parameter zu binden.

Abbildung 4.5 zeigt ein gefärbtes Petrinetz, welches das komplexe Beispielereignis modelliert und ebenso erkennen kann. Im Unterschied zum einfachen Petrinetz sind hier zusätzliche Angaben für die Parameterbindung zu finden.

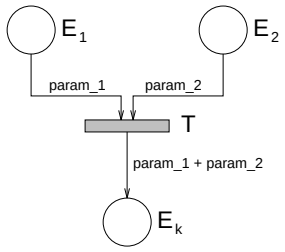

Abbildung 4.5: Gefärbtes Petrinetz für E_1 **and** E_2

Die Funktionsweise des Netzes ist wie folgt: Falls das Ereignis E_1 eintritt, so wird in den entsprechenden Platz ein Token gelegt. Die Transition T muss solange warten, bis in beiden Eingangsplätzen mindestens ein Token liegt (Schaltregel). Tritt nun das Ereignis E_2 ein, so werden die entsprechenden Parameter (hier exemplarisch *param_1* und *param_2*) gebunden. Die Transition kann feuern, d.h. die Token werden aus den Eingangsplätzen entfernt und in den Ausgangsplatz E_k wird ein neues Token gelegt. Dabei wird dem Token ein entsprechender Parameterwert mitgegeben (hier exemplarisch *param_1 + param_2*, als Symbol für die Kombination der beiden Parameterwerte). Durch die Belegung von Platz E_k mit diesem Token wird signalisiert, dass das komplexe Ereignis E_k eingetreten ist.

Eine modifizierte Form von gefärbten Petrinetzen wird im Projekt SAMOS ([Gat95]) zur Darstellung und Erkennung von komplexen Ereignissen verwendet. Im Zusammenhang mit ADBS wurde mit diesem Modell bislang nicht versucht, Bedingungen, Aktionen und ausführungsbezogene Faktoren darzustellen.

4.3 Modellvergleich und Modellentscheid

In den nun folgenden Abschnitten werden zuerst Kriterien aufgeführt, mit deren Hilfe die hier vorgestellten Modelle verglichen werden können. Danach folgt ein tabellarischer Vergleich der Modelle. Zum Schluss wird angegeben, welches Modell die Basis für die Konzepte von ALFRED darstellt.

4.3.1 Kriterien

Nachstehend werden ausgewählte Kriterien aufgelistet und kurz erläutert, die für den tabellarischen Vergleich der einzelnen Modelle verwendet werden. Diese Kriterien basieren auf den Anforderungen, die in Abschnitt 4.1 aufgestellt wurden. Die Angabe in eckigen Klammern ist die Abkürzung für das entsprechende Kriterium.

- **Darstellung aller Ereignisoperatoren** [Operatoren]
  Sind sämtliche Ereignisoperatoren aus Abschnitt 3.3.1.2 darstellbar?
- **Darstellung von Zeitereignissen** [Zeitereignisse]
  Können Zeitereignisse, d.h. Zeitpunkte modelliert werden?
- **Erkennung von Ereignissen** [Erkennung]
  Kann mit dem Modell eine Ereigniserkennung durchgeführt werden?
- **Parameterbindung** [Parameter]
  Können bei der Ereigniserkennung Parameter gebunden und weitergegeben werden, so dass diese für die Bedingungsauswertung und Aktionsausführung zur Verfügung stehen?
- **Parameterkontexte** [P-Kontext]
  Können bei der Erkennung komplexer Ereignisse resp. bei der Parameterbindung alle vier Paramterkontexte (vgl. Abschnitt 2.3.2.6) berücksichtigt werden?
- **Darstellung von Bedingungen** [Bedingungen]
  Kann eine komplexe Bedingung in ihren Einzelkomponenten dargestellt werden?
- **Darstellung von Aktionen** [Aktionen]
  Kann eine komplexe Aktion in ihren Einzelkomponenten dargestellt werden?
- **Darstellung von Kontrollstrukturen** [Kontrollstruktur]
  Können die Strukturelemente *Sequenz*, *Auswahl* und *Wiederholung* für komplexe Bedingungen und Aktionen im Modell dargestellt werden?
- **Interne Verbindungen** [Verbindung]
  Können Ereignisse, die durch Bedingungs- oder Aktionsteile ausgelöst werden, modell-intern erkannt werden (z.B. durch entsprechende Verbindungen)?
- **Semantik** [Semantik]
  Ist die Semantik aller Strukturen, insbesondere Ereignisstruktur, aus dem Modell eindeutig ersichtlich?
- **Nebenläufigkeit** [Nebenl.]
  Können nebenläufige resp. parallele Ausführungen modelliert werden?
- **Instanzen** [Instanz]
  Können mehrere Instanzen von Ereignissen im Modell verwaltet werden?
- **Simulation** [Simulation]
  Unterstützt das Modell eine Simulation von Abläufen?

- **Mengen-/Instanzorientiert** [set/inst]
  Können mengen- und instanzorientierte Regeln dargestellt und verarbeitet werden?
- **Pre/Post** [pre/post]
  Können pre- und post-Auslösungen im Modell dargestellt werden?
- **Kopplungsmodi** [KM]
  Sind mit diesem Modell alle sechs Kopplungsmodi (vgl. Abschnitt 2.3.2.5) darstellbar und können diese bei der Ausführung berücksichtigt werden?
- **Regelanalyse** [Analyse]
  Können auf der Grundlage der Regeldarstellung Analysen durchgeführt werden (z.B. Zyklenerkennung, Überprüfung der Confluence)?

4.3.2 Vergleich

In Tabelle 4.1 werden die Modelle miteinander verglichen. Die einzelnen Modelle werden folgendermassen abgekürzt: Integer Array [IA], Endlicher Automat [EA], Baumansatz [Baum], Extended Syntactic Tree [EST], Petrinetz [PN], Gefärbtes (coloured) Petrinetz [CPN].

Damit ein Vergleich möglich ist, werden die einzelnen Modelle bezüglich der Kriterien bewertet. Die Einträge in den Zeilen haben folgende Bedeutung:

- \- Das Modell unterstützt dieses Kriterium nicht, resp. der zusätzliche Aufwand dafür ist (sehr) hoch.
- \+ Das Kriterium wird vom Modell unterstützt, jedoch ist zusätzlicher Programmieraufwand nötig (Algorithmen und Datenstrukturen).
- ++ Das Modell unterstützt diese Forderung gut. Es müssen eventuell geringfügige Änderungen hinzugefügt werden.

4.3.3 Modellentscheid

Die Anforderungen an das Modell, welches ALFRED zugrunde liegen soll, sind weit höher als in anderen Prototypen. Es soll in erster Linie nicht nur möglich sein, Ereignisse darzustellen und erkennen zu können, sondern das Modell soll auch eine Darstellung und Verarbeitung von Bedingungen resp. Aktionen ermöglichen.

Unter diesem Gesichtspunkt sind einige der in diesem Kapitel vorgestellten Modelle nicht mächtig genug. Dies sind die Ansätze auf der Basis von Bäumen (*Baum* und *Extended Syntactic Trees*), die *Integer Array* Methode und der *endliche Automat*. Alle diese Methoden und Modelle sind dazu ausgelegt, Ausdrücke, oder in diesem Falle Ereignisse, zu erkennen.

Folglich bleiben nur die beiden Petrinetz-Modelle, d.h. *Platz-/Transitionsnetze* und *gefärbte Petrinetze* übrig. Da es in gewöhnlichen *Platz-/Transitionsnetzen* nicht möglich ist, Parameter in den Token zu speichern, genügt auch dieses Modell nicht den Anforderungen für die Regeldarstellung in ALFRED.

Die erweiterte Form der Petrinetze, die sogenannten *gefärbten Petrinetze*, werden den meisten der aufgestellten Anforderungen gerecht. Sie können verwendet werden, um Ereignisse darzustellen und zu erkennen sowie um Bedingungen und Aktionen darzustellen und auszuwerten resp. auszuführen. Ebenso können Anforderungen an die Regelverarbeitung erfüllt werden.

Die Darstellung und Verarbeitung von Regeln basiert in ALFRED auf einer etwas vereinfachten Form von *gefärbten Petrinetzen*, dem sogenannten *Action Rule Flow Petri Net* (ARFPN). In einem

Kriterien	IA	EA	Baum	EST	PN	CPN
Operatoren	++	+	+	++	++	++
Zeitereignisse	-/+	-/+	-/+	-/+	+	+
Erkennung	++	++	++	++	++	++
Parameter	+	-/+	-/+	++	-	++
P-Kontext	+	-	+	+	-	+
Bedingungen	-	-	+	+	+	+
Aktionen	-	-	-	-	+	++
Kontrollstruktur	-	-	-	-	+	+
Verbindung	-	-	-	-	++	++
Semantik	-/+	+/++	+/++	+/++	+/++	+/++
Nebenl.	-	-	-	-	++	++
Instanz	-/+	-/+	+	+	+/++	++
Simulation	+	+	+	+	++	++
set/inst	-/+	-/+	-/+	-/+	-/+	-/+
pre/post	+	+	+	+	++	++
KM	-	-	-	-	++	++
Analyse	-	-	-	-	+/++	+/++

Tabelle 4.1: Tabellarischer Vergleich der Modelle zur Regelbeschreibung

ARFPN werden nicht sämtliche Möglichkeiten von gefärbten Petrinetzen ausgeschöpft. So wird zum Beispiel die Bedingungsauswertung nicht über Kantenfunktionen realisiert, sondern sie wird in den Schaltvorgang des Petrinetzes mit einbezogen (durch den Simulationsalgorithmus). Eine genauere Beschreibung von *gefärbten Petrinetzen* ist in Abschnitt 5.2 zu finden.

Kapitel 5

Petrinetze

Petrinetze, benannt nach deren Erfinder C.A. Petri, sind Systemmodelle für Vorgänge, bei welchen insbesondere Informationsflüsse eine Rolle spielen. Sie zeichnen sich durch ihre Einfachheit und Flexibilität bei der Modellierung von Vorgängen sowie durch Anschaulichkeit — Petrinetze werden graphisch repräsentiert — aus. Mit ihrer Hilfe können z.B. Informationsflüsse in Unternehmen oder Flüsse von Steuerinformationen in maschinellen Anlagen modelliert werden. Petrinetze werden heutzutage vor allem in der Informatik verwendet und bieten u.a. folgende Vorzüge:

- Mit Petrinetzen ist es möglich, den Aufbau, die Arbeitsweise und die Eigenschaften von Systemen zu beschreiben. Dabei können auch parallele Abläufe repräsentiert werden.
- Mit Hilfe von Petrinetzen können Systeme auf unterschiedlichen Abstraktionsebenen dargestellt werden, ohne dabei die Beschreibungssprache ändern zu müssen.
- Netzdarstellungen von Systemen ermöglichen den Nachweis von Systemeigenschaften und eignen sich zur Durchführung von Korrektheitsbeweisen.

Durch ihre Flexibilität und Anschaulichkeit eignen sich Petrinetze ebenso für die Darstellung von Vorgängen in aktiven Datenbanksystemen. In der jüngeren Forschung zu ADBS wurden bereits Petrinetze zur Beschreibung von Abläufen verwendet (vgl. [Gat95]).

In diesem Kapitel werden Grundlagen für Petrinetze dargestellt. Dazu werden zuerst einfache Petrinetze (vgl. auch [RW82, Rei82, Rei86, Thi86]) und anschliessend gefärbte Petrinetze (vgl. [Jen92]) erläutert.

5.1 Einfache Petrinetze

Die statische Struktur der einfachsten Form eines Petrinetzes, der sogenannten Stellen-/Transitions- resp. Platz-/Transitions-Netze (P/T-Netze), kann durch einen gerichteten bipartiten Graphen dargestellt werden. Er besteht aus gerichteten *Kanten* und zwei wohlunterschiedenen Klassen von Knoten, die *Plätze* und *Transitionen* genannt werden. Ein Platz wird über eine Kante mit einer Transition verbunden oder umgekehrt, nie aber wird ein Platz mit einem Platz oder eine Transition mit einer Transition verknüpft. Formal kann die statische Struktur eines P/T-Petrinetzes wie folgt definiert werden:

Definition 5
Die statische Struktur eines Petrinetzes ist ein Tupel (P,T,A) mit:

1. *P ist eine endliche Menge von* **Plätzen** *(Stellen)*
2. *T ist eine endliche Menge von* **Transitionen**
3. $A \subset ((P \times T) \cup (T \times P))$ *ist eine endliche Menge von gerichteten* **Kanten**, *die Plätze und Transitionen miteinander verbinden.*

Für jede Transition werden die dazugehörigen *Eingangsplätze* und *Ausgangsplätze* wie folgt definiert:

Definition 6
Die Menge der **Eingangsplätze** (EP_t) *resp. der* **Ausgangsplätze** (AP_t) *einer Transition t ist wie folgt definiert* $(p \in P, t \in T, (p,t) \in A, (t,p) \in A)$*:*

$$EP_t = \{p \mid (p,t) \in A\}$$

$$AP_t = \{p \mid (t,p) \in A\}$$

Um in einem Petrinetz einen Vorgang modellieren zu können, muss das Netz um eine Dynamik erweitert werden. Zu diesem Zweck werden *Marken* (Token) verwendet. Eine Marke in einem P/T-Netz stellt einen bool'schen Wert dar. Eine Verteilung von Marken auf die Plätze eines Petrinetzes wird als *Markierung* bezeichnet. Sobald ein Petrinetz markiert ist, kann das Netz schalten, d.h. das Netz transportiert nach gewissen Schaltregeln Marken von Eingangsplätzen über die gerichteten Kanten zu den Transitionen. Diese entfernen die eingehenden Marken und generieren neue Marken, die wiederum über die gerichteten Kanten an die Ausgangsplätze weitergereicht werden. Dieser Vorgang wird auch als *feuern von Transitionen* bezeichnet.

P/T-Petrinetze lassen sich in zwei Klassen aufteilen. Die erste Klasse, sogenannte *Ein-Marken-Netze*, haben die Eigenschaft, dass sich in jedem Platz höchstens eine Marke befinden darf. In der zweiten Klasse, der *Mehr-Marken-Netze*, dürfen die Plätze mehrere Marken enthalten. Die *Schaltregel* für die erste Klasse von P/T-Netzen lautet wie folgt:

- Eine Transition kann **feuern**, d.h. Marken können von den Eingangsplätzen entfernt und neue Marken können in den Ausgangsplätzen hinzugefügt werden, wenn *alle* Eingangsplätze dieser Transition mit einer Marke belegt sind und sich in *keinem* der Ausgangsplätze eine Marke befindet.

Graphisch wird ein Petrinetz wie folgt repräsentiert: Plätze werden als Kreise, Kanten als Pfeile, Transitionen als Balken und Marken als gefüllte Kreise dargestellt. Abbildung 5.1 zeigt ein Beispiel eines einfachen P/T-Petrinetzes vor und nach dem Schalten der Transition T. Die Beschriftung neben den Plätzen und der Transition dient nur zur einfacheren Identifikation.

Das Petrinetz aus Abbildung 5.1 besteht aus einer Transition (T) zwei Eingangsplätzen (P1 und P2) von T und einem Ausgangsplatz (P3) von T. In den Plätzen P1 und P2 befindet sich jeweils eine Marke. Wenn nun die Schaltregel auf dieses Petrinetz angewandt wird, so bedeutet dies folgendes:

Die Transition T kann nur dann feuern, wenn in P1 und in P2 je ein Token vorhanden ist, und wenn sich im Platz P3 kein Token befindet. Dies ist im Petrinetz auf der linken Seite der Abbildung 5.1 der Fall. Die Transition T entfernt nun aus jedem seiner Eingangsplätze (P1 und P2) das Token

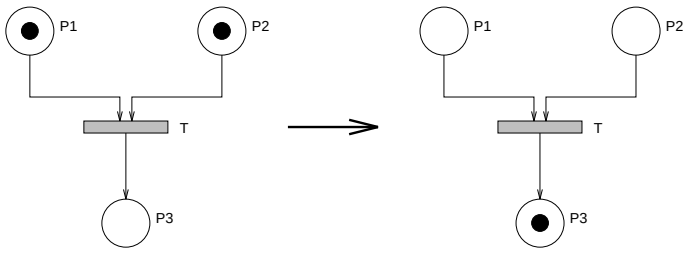

Abbildung 5.1: P/T-Petrinetz vor und nach dem Schaltvorgang

und fügt eine neue Marke in den Platz P3 ein. Die resultierende Markierung ist auf der rechten Seite von Abbildung 5.1 zu sehen.

In der zweiten Klasse der P/T-Netze, den sogenannten Mehr-Marken-Netzen, muss das Netz um ein weiteres Element ergänzt werden — die *Kapazität* eines Platzes — welche angibt, wieviele Marken sich maximal auf einem Platz befinden dürfen. Diese Kapazität kann als unendlich gross definiert werden, womit zum Ausdruck gebracht wird, dass sich in diesem Platz beliebig viele Token befinden dürfen. Die einfache Schaltregel von Ein-Marken-Netzen muss ebenfalls angepasst werden und lautet im Mehr-Marken Fall wie folgt:

- Eine Transition kann **feuern**, wenn in jedem ihrer Eingangsplätze mindestens ein Token liegt und in jedem ihrer Ausgangsplätze genügend freie Kapazität vorhanden ist, d.h. die Anzahl der Token in jedem ihrer Ausgangsplätze darf höchstens *Kapazität - 1* betragen.

Die oben vorgestellten Klassen von Petrinetzen (Ein- und Mehr-Marken P/T-Netze) gehören zu den *"low-level"* Petrinetzen. Damit reale Situationen besser dargestellt werden können, wurden die bestehenden Petrinetze zu sogenannten *"high-level"* Petrinetzen erweitert. Diese Netze unterscheiden sich von den "low-level" Netzen dadurch, dass sie als Marken (Token) nicht nur einen bool'schen Wert tragen können, sondern beliebige komplexe Informationen (z.B. Text, Tabellen oder ganze Datenbankzustände). Zu dieser erweiterten Klasse gehören auch die gefärbten Petrinetze, die im folgenden Abschnitt erläutert werden.

5.2 Gefärbte Petrinetze

Gefärbte Petrinetze (Coloured Petri Nets (CPN), vgl. [Jen92]) stellen eine Erweiterung von einfachen Petrinetzen dar. Sie ermöglichen es, komplexe Informationen in den Marken zu speichern und diese Informationen durch das Netz zu transportieren. Damit auf die Tokeninformationen zurückgegriffen werden kann, beinhaltet das CPN Funktionen und Ausdrücke, die auf den Kanten und an den Transitionen des Petrinetzes stehen können.

Um ein CPN definieren zu können, sind sogenannte Multimengen notwendig. Eine *Multimenge* (MM), auch *multi-set* genannt, ist eine Erweiterung einer Menge im üblichen mathematischen Sinn, indem in einer Multimenge *mehrere gleiche Elemente* vorkommen dürfen. Wenn zu einer Menge $\{a, b, c\}$ ein Element c hinzugefügt wird, so ist die resultierende Menge immer noch $\{a, b, c\}$. Wird dieselbe Operation jedoch in Multimengen durchgeführt, so ist die resultierende Multimenge $\{a, b, c, c\}$.

Eine Multimenge MM wird stets über einer Grundmenge S definiert. Um die Multimenge formal darstellen zu können, genügt es zu wissen, welche Elemente von S wie häufig in der Multimenge MM vorkommen. Diese Definition kann mittels einer *formalen Summe* erfolgen.

Die Definitionsmenge S sei die Menge {a,b,c,d}. Eine Multimenge MM={a,b,c,b,c} über der Definitionsmenge S wird wie folgt als formale Summe dargestellt:

$$MM = 1`a + 2`b + 2`c$$

Die Multimenge MM besteht demnach aus einem Element a und aus je zwei Elementen b und c. Elemente von S, die nicht in der Multimenge vorkommen, also Koeffizienten mit dem Wert Null, werden per Definition weggelassen. Koeffizienten mit Wert 1 werden explizit geschrieben, um die Übersichtlichkeit zu fördern.

Auf dieser informellen Grundlage können die Multimengen formal definiert werden. Im folgenden wird mit $\mathcal{N}$ die Menge aller natürlichen Zahlen bezeichnet.

Definition 7
Eine **Multimenge** *MM über einer nichtleeren Menge S kann durch eine formale Summe wie folgt dargestellt werden:*

$$MM = \sum_{s \in S} m(s)`s \qquad \text{wobei gilt: } m(s) \in \mathcal{N}$$

Zusätzlich gilt für die Elemente von S:

$$\forall s \in S: \quad s \in MM \Longleftrightarrow m(s) \neq 0$$

Mit S_{MM} wird die Menge aller Multimengen über S bezeichnet.

5.2.1 Struktur

Auf der Grundlage der zuvor definierten Multimengen kann nun die Struktur von gefärbten Petrinetzen als 9-Tupel formal beschrieben werden. Im folgenden wird davon ausgegangen, dass alle Ausdrücke, die in einem gefärbten Petrinetz existieren, eine wohldefinierte Syntax und Semantik aufweisen.

Definition 8
Ein **Gefärbtes Petrinetz** *(Coloured Petri Net),* $CPN = (\Sigma, P, T, A, N, C, G, E, I)$*, ist ein 9-Tupel mit folgenden Eigenschaften:*

1. Σ *ist eine endliche Menge von Typen, die als* **Farbenmengen** *bezeichnet werden.*
2. P *ist eine endliche Menge von* **Plätzen**.
3. T *ist eine endliche Menge von* **Transitionen**, *dabei gilt:* $P \cap T = \emptyset$.
4. A *ist eine endliche Menge von gerichteten* **Kanten**, *wobei gilt:* $P \cap A = T \cap A = \emptyset$.
5. N *ist eine* **Knotenfunktion**, *die als* $N : A \rightarrow ((P \times T) \cup (T \times P))$ *definiert ist.*
6. C *ist eine* **Farbenfunktion**, *die als* $C : P \rightarrow \Sigma$ *definiert ist.*
7. G *ist eine* **"Guard"-Funktion**, *die als* $G : T \rightarrow E_B$ *definiert ist. Mit* E_B *wird die Menge aller Ausdrücke bezeichnet, deren Auswertung ein bool'sches Resultat liefert.*
8. E *ist eine* **Kantenausdruckfunktion**, *die als* $E : A \rightarrow E_K$ *definiert ist. Mit* E_K *wird die Menge der Ausdrücke bezeichnet, deren Auswertung eine Multimenge über dem zur Kante gehörenden Platz ist.*
9. I *ist die* **Initialisierungsfunktion**, *die als* $I : P \rightarrow E_{clo}$ *definiert ist, wobei* E_{clo} *die Menge aller Ausdrücke darstellt, die keine Variablen enthalten.*

Um die Bedeutung der Definition 8 zu verdeutlichen, werden nachfolgend einige Punkte näher erläutert. Die folgenden Zahlen korrespondieren dabei mit den Zahlen in der Definition.

(1) Σ, die *Menge der Farbenmengen*, bestimmt die Typen von Variablen, Operationen und Funktionen (z.B. Kantenfunktionen, Guardfunktionen).

(5) Die *Knotenfunktion* bildet jede Kante auf ein Knotenpaar ab. Das erste Element ist der Quellknoten, das zweite der Zielknoten. Die beiden Knoten müssen vom Typ her verschieden sein, d.h. ein Knoten muss ein Platz der andere eine Transition sein. Die Knotenfunktion stellt also die Verbindung von Plätzen zu Transitionen und umgekehrt sicher. Im Gegensatz zu einfachen Petrinetzen können in CPN mehrere Kanten zwischen Quell- und Zielknoten existieren.

(6) Die *Farbenfunktion* C bildet jeden Platz p auf eine Farbenmenge C(p) ab. Jedes Token in p muss eine Tokenfarbe haben, die zu C(p) gehört (zulässige Farbe).

(7) Die *Guard-Funktion* bildet jede Transition auf einen bool'schen Ausdruck ab, d.h. ein Ausdruck, dessen Auswertung einen bool'schen Wert liefert. Die Menge der Variablentypen von G(T) muss eine Teilmenge von Σ sein. Der bool'sche Ausdruck spezifiziert eine zusätzliche Bedingung. Wenn die Bedingung erfüllt ist, so kann die Transition feuern.

(8) Da in "high-level" Petrinetzen die Marken nicht nur bool'sche Werte repräsentieren, sondern unter Umständen komplexere Informationen (z.B. Datenbankzustände), wird das "Spiel mit den Token" dementsprechend komplizierter. Es genügt nun nicht zu sagen, wieviele Token aus den Plätzen entfernt und wieviele erzeugt werden. Die Ausführung der Transition sowie der Typ des Ausgangstokens können nun vom Typ des Eingangstokens abhängen. Diese Abhängigkeit wird durch die *Kantenausdruckfunktion* bestimmt.

Die *Kantenausdruckfunktion* bildet jede Kante auf einen Ausdruck ab. Die Auswertung dieses Ausdrucks ergibt eine Multimenge über der Farbenmenge des zur Kante gehörenden Platzes, d.h. der Typ des Resultats muss gleich sein wie der Typ des zur Kanten gehörenden Platzes.

Die Ausdrücke auf den Eingangskanten bestimmen, welche Token und zugehörige Informationen von den Eingangsplätzen zu den Transitionen geführt werden.

Die Ausdrücke auf den Ausgangskanten bestimmen, welche Token und welche Informationen neu erzeugt und an die Ausgangsplätze weitergegeben werden.

(9) Die *Initialisierungsfunktion* bildet jeden Platz auf einen Ausdruck ab, der keine Variablen enthält (geschlossener Ausdruck). Die Auswertung eines solchen Ausdrucks ergibt immer denselben Wert und weist jedem Platz eine (evtl. leere) Anfangsmarkierung zu. Dabei muss der Typ des Ausdrucks mit dem Typ des jeweiligen Platzes übereinstimmen.

5.2.2 Schaltvorgang

Der Schaltvorgang in einem CPN ist aufgrund der komplexen Informationen, die durch das Netz transportiert werden können, etwas aufwendiger als in einfachen Petrinetzen. Anhand eines Beispiels soll dieser Schaltvorgang erläutert werden.

Plätze, Transitionen und Kanten werden gleich dargestellt wie bei einfachen Petrinetzen. Bei den CPN müssen zusätzlich die Typen der Plätze, die Kantenausdrücke, die Guardfunktionen und die Inhalte der Token beschrieben werden (vgl. Abbildung 5.2).

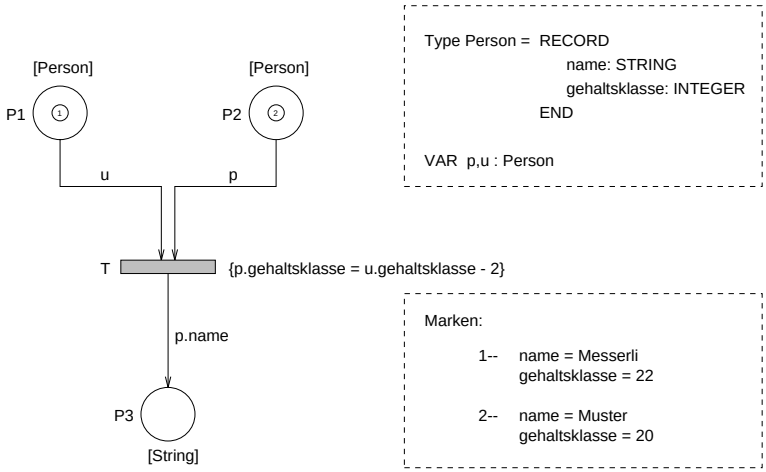

Abbildung 5.2: Graphische Darstellung eines einfachen CPN

Das Beispielpetrinetz aus Abbildung 5.2 besteht aus einer Transition T, zwei Eingangsplätzen (P1 und P2) von T und einem Ausgangsplatz (P3) von T. Bei jedem Platz ist in eckigen Klammern '[ ]' der dem Platz zugeordnete Markentyp angegeben. Alle Typen und Variablen des Petrinetzes werden im oberen Rechteck deklariert und definiert. Die Plätze P1 und P2 können Marken vom Typ [Person] beinhalten, Platz P3 jedoch nur Marken vom Typ [String]. In P1 und P2 liegen zwei Marken, deren Inhalte im unteren Rechteck aufgeführt sind. Auf der Kante von P1 nach T steht ein Kantenausdruck (u). Mit diesem wird erreicht, dass ein Token in P1 vom Typ [Person] zu T geführt wird (da Typ(u) = [Person]). Analoges gilt für den Ausdruck auf der Kante von P2 nach T. Die Guardfunktion bei T ($\{p.gehaltsklasse = u.gehaltsklasse - 2\}$) prüft die Information in den Token. Der Ausdruck auf der Kante von T nach P3 ($p.name$) garantiert, dass nur gerade der Name von p an das Token gebunden wird, welches in den Platz P3 zu liegen kommt.

Die Transition T kann nun die Marken von den Eingangsplätzen entfernen und neue Marken in den Ausgangsplatz legen. Dazu gilt die Schaltregel aus den Mehrmarkennetzen, d.h es müssen sich genügend Marken in den Eingangsplätzen befinden und die Kapazität des Ausgangsplatzes muss gross genug sein. Da nun festgelegt werden muss, welche Token oder welche Informationen der Token benötigt werden, muss der Schaltvorgang näher spezifiziert werden. Dies geschieht in CPN durch die Kantenausdrücke.

Da sich in beiden Plätzen P1 und P2 eine Marke befindet, werden die Variablen in den Kantenausdrücken von P1 nach T resp. von P2 nach T *gebunden*. Durch diese Bindung werden auch sämtliche Variablen, die sich in der Guardfunktion der Transition und im Ausdruck auf der Ausgangskante der Transition befinden, gebunden. Dies bedeutet, dass sämtliche Variablen in den Kantenausdrücken durch explizite Werte, welche in den Token stehen, substituiert werden. In CPN aus Abbildung 5.2 ist folgende Bindung möglich:

$$\text{Bindung b: } \langle u = (Messerli, 22);\ p = (Muster, 20)\rangle$$

Der Kantenausdruck kann nun mit der Bindung zu einem Wert aufgelöst werden. Dieser Wert muss aus dem Wertebereich des Markentyps im Eingangsplatz entstammen ([Person]). Durch die Auswertung des Kantenausdrucks ist die Transition *aktiviert*. Da sämtliche Variablen bereits gebunden sind, kann nun die Guardfunktion ausgewertet werden, in Abbildung 5.2 ist dies {*p.gehaltsklasse = u.gehaltsklasse - 2*}. Die Auswertung ergibt den Wert TRUE und somit kann die Transition feuern. Die beiden Marken in den Eingangsplätzen P1 und P2 werden entfernt und eine neue Marke wird in den Platz P3 gelegt (vgl. Abbildung 5.3). Dazu muss der Ausdruck auf der Ausgangskante (Kante von T nach P3) auswertet werden. Der Typ des Resultats dieses Ausdrucks muss mit dem Typ des Platzes übereinstimmen (Typ von P3 = [String]). Die Auswertung ergibt den Namen "Muster" (weil mit der oben genannten Bindung p.name = Muster). In P3 wird also eine neue Marke mit dem Wert "Muster" erzeugt.

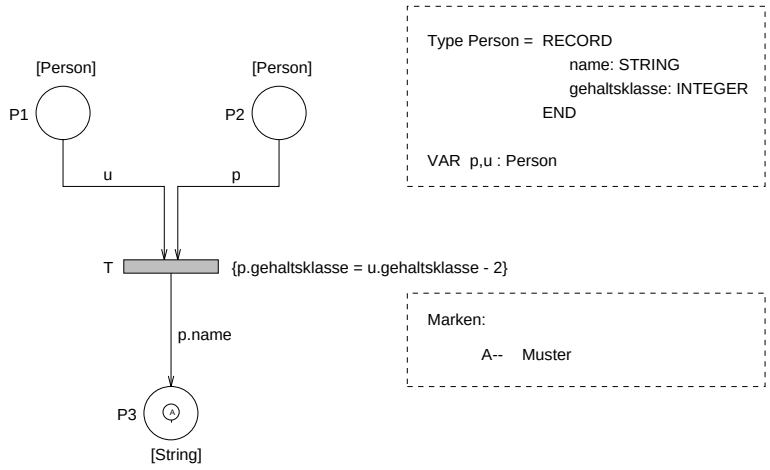

Abbildung 5.3: CPN nach dem Feuern von T

Kapitel 6

Modellierung von Regeln

In einem ADBS werden Regeln verwendet, um aktives Verhalten realisieren zu können. Diese Regeln müssen mit Hilfe eines Modells dargestellt werden.

Die Modellierung von Regeln wird in ALFRED mit einem Action Rule Flow Petri Net (ARFPN) realisiert. Ein ARFPN ist ein gefärbtes Petrinetz, das um eine etwas spezifische Verarbeitungslogik ergänzt wird. Mit Hilfe des ARFPN werden *sämtliche* Regelkomponenten modelliert. Zudem ermöglicht das Petrinetz auch die Darstellung der ausführungsbezogenen Faktoren und die Verarbeitung von Regeln (vgl. Kapitel 8).

In diesem Kapitel werden sämtliche Petrinetz-Strukturen definiert und erläutert, die für die Modellierung und Verarbeitung von Regeln benötigt werden. Dazu gehören die Modellierung von primitiven und komplexen Ereignissen, Bedingungen und Aktionen sowie die Integration aller Komponenten in ein und dasselbe Petrinetz.

Alle Beispiele in diesem Kapitel beinhalten nur die *mengenorientierte* Auslösung. Auf die Problematik der *instanzorientierten* Regelauslösung wird in Kapitel 8 eingegangen, da bei der *instanzorientierten* Auslösung Strukturen hinzugefügt und der Petrinetz-Algorithmus erweitert werden müssen.

6.1 Einführung

In ALFRED werden sowohl die Ereigniskomponenten, die Bedingungskomponenten als auch die Aktionskomponenten der Regeln mit Hilfe von Petrinetzen modelliert. Mit diesem Ansatz ist es möglich, einen nahtlosen Übergang zwischen den einzelnen Komponenten einer Regel zu gewährleisten. Dieser Übergang ist vor allem bei der Verarbeitung (vgl. Kapitel 8) wichtig, wo Schritt für Schritt das ganze Regelnetz verarbeitet wird. Abbildung 6.1 zeigt die Komponenten einer ECAA-Regel in ALFRED als Petrinetz dargestellt.

Eine ECAA-Regel in ALFRED besteht demnach aus sechs Teilen:

- *Ereigniskomponente*
  In diesem Teil wird das primitive oder komplexe Ereignis der Regel dargestellt.
- *Ereignisende*
  Dieser Platz nach der Ereigniskomponente wird benötigt, um den Ereignis- und Bedingungsteil zu verbinden.
- *Bedingungskomponente*
  Dieser Teil des Petrinetzes stellt die Bedingung der Regel dar. Am Ende dieser Komponente

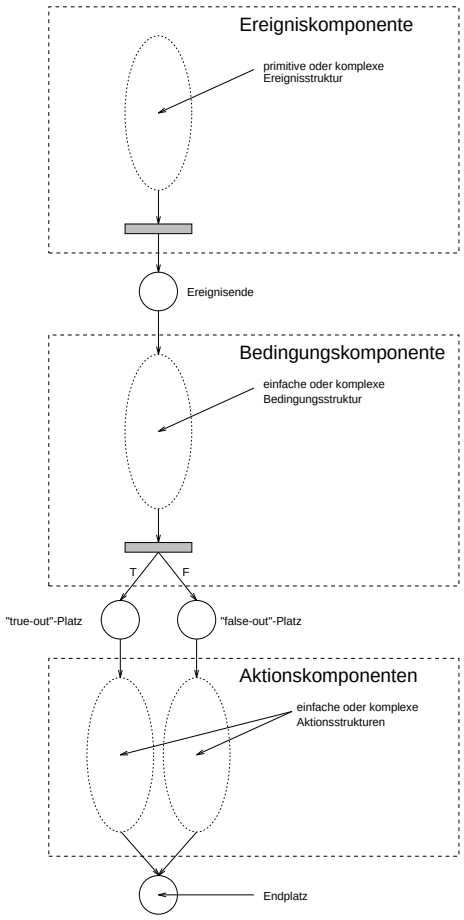

Abbildung 6.1: Eine Regel in ALFRED mit ihren Komponenten

wird mittels eines Algorithmus entschieden, ob der Wahr-Zweig oder der Falsch-Zweig der Regel verarbeitet werden soll.

- *"true_out" und "false_out" Plätze*
  Diese zwei Plätze werden benötigt, um die Bedingungskomponente mit der jeweiligen Aktionskomponente zu verbinden.
- *Aktionskomponenten*
  In diesen Komponenten werden die Aktionen der Regel modelliert.
- *Endplatz*
  Dieser Platz wird an beide Aktionskomponenten angehängt und bildet den Abschluss der Regel.

In den folgenden Abschnitten werden sämtliche Teil-Petrinetze definiert, die nötig sind, um die in ALFRED unterstützten Ereignisse, Bedingungen und Aktionen darzustellen.

6.2 Ereigniskomponente

In ALFRED wird zwischen primitiven und komplexen Ereignissen unterschieden, die bereits in Abschnitt 3.3.1 definiert wurden. In den folgenden Abschnitten werden die Petrinetze modelliert,

die für die Darstellung und Erkennung dieser komplexen Ereignisse notwendig sind. Die primitiven Ereignisse werden dabei als Plätze im Petrinetz dargestellt (vgl. auch Abschnitt 6.5).

Für die Darstellung und Erkennung von komplexen Ereignissen sind *Parameterkontexte* (vgl. Abschnitt 2.3.2.6) notwendig, die festlegen, welche Parameter bei der Erkennung von komplexen Ereignissen gebunden werden müssen. Aus Gründen der Übersichtlichkeit wird bei den folgenden Operatoren auf die Parameterkontexte verzichtet. Stattdessen wird in Abschnitt 6.2.8 anhand eines Beispiels erläutert, wie Parameterkontexte in Petrinetze aufgenommen werden können und wie sich diese Kontexte auf die Semantik der Operatoren auswirken.

6.2.1 Bool'sche Operatoren

In ALFRED werden zwei bool'sche Operatoren, der Disjunktionsoperator *or* und der Konjunktionsoperator *and*, unterstützt, die durch folgende zwei Petrinetze modelliert werden (E_1 und E_2 bezeichnen primitive Ereignisse):

6.2.1.1 Or

Abbildung 6.2 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E := E_1$ **or** E_2. Falls einer der Eingangsplätze (E_1 oder E_2) markiert wird, so kann die entsprechende Transition (T1 oder T2) feuern und der Platz E wird markiert. Das komplexe Ereignis ist somit eingetreten.

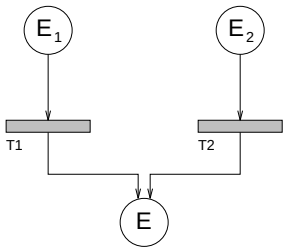

Abbildung 6.2: Petrinetz für $E := E_1$ **or** E_2

6.2.1.2 And

Abbildung 6.3 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E := E_1$ **and** E_2. Die Transition T1 kann nur genau dann feuern, wenn *beide* Eingangsplätze (E_1 und E_2) markiert sind. Die primitiven Ereignisse E_1 und E_2 können in beliebiger Reihenfolge eintreten. Die Transition markiert den Ausgangsplatz E und signalisiert somit das Eintreten des komplexen Ereignisses.

6.2.2 Intervalloperatoren

In ALFRED werden vier mögliche Intervalltypen unterstützt. Intervalle dienen dazu, die Erkennung eines Ereignisses auf einen bestimmten zeitlichen Rahmen zu beschränken. Das Eintreten des Ereignisses wird nicht mehr in einer globalen Zeitgeschichte betrachtet, sondern in einer eigenen, vom Intervall bestimmten. Die Intervallgrenzen werden durch ein Anfangsereignis (Beginnereignis E_b) und ein Endereignis (E_e) definiert, wobei die Art dieser Ereignisse beliebig ist. Falls Endereignis und Intervallereignis wiederum komplexe Ereignisse sind, so müssen deren Teilereignisse alle in der Zeitspanne des Intervalls eintreten. In ALFRED ist die Überlappung und Verschachtelung von identischen Intervallen nicht möglich. Folgende vier Intervalle werden in ALFRED unterstützt:

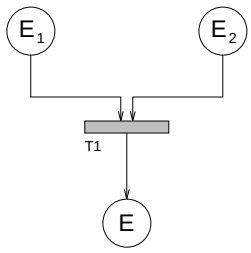

Abbildung 6.3: Petrinetz für $E := E_1$ **and** E_2

1. (**In**): Das komplexe Ereignis tritt jedesmal dann ein, wenn das zu betrachtende Ereignis im Intervall eintritt.
2. (**Not in**): Das komplexe Ereignis tritt ein, wenn das zu betrachtende Ereignis nicht im Intervall eintritt.
3. (**The n$^{\text{th}}$ in**): Das komplexe Ereignis tritt genau dann ein, wenn das zu betrachtende Ereignis n-mal im Intervall eingetreten ist.
4. (**Last in**): Das komplexe Ereignis tritt ein, wenn das zu betrachtende Ereignis zum letzten Mal im Intervall eintritt.

6.2.2.1 In

Abbildung 6.4 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E := E_1$ **in** (E_b, E_e). Das Intervall muss durch das Eintreten des Beginnereignisses E_b zuerst geöffnet werden. Ist das Intervall offen, so wird durch *jedes* Eintreten des Ereignisses E_1 das komplexe Ereignis E ausgelöst. Tritt bei offenem Zustand jedoch das Endereignis E_e auf, so wird das Intervall geschlossen und wieder zurückgesetzt. Die Transition TB garantiert, dass bei offenem Intervall eintretende Beginnereignisse (E_b) geschluckt werden. Die Transitionen T1 und T3 entfernen bei geschlossenem Intervall eintretende Ereignisse E_1 und E_e. Der Platz P1 in Abbildung 6.4 wird als *Speicherplatz* bezeichnet.

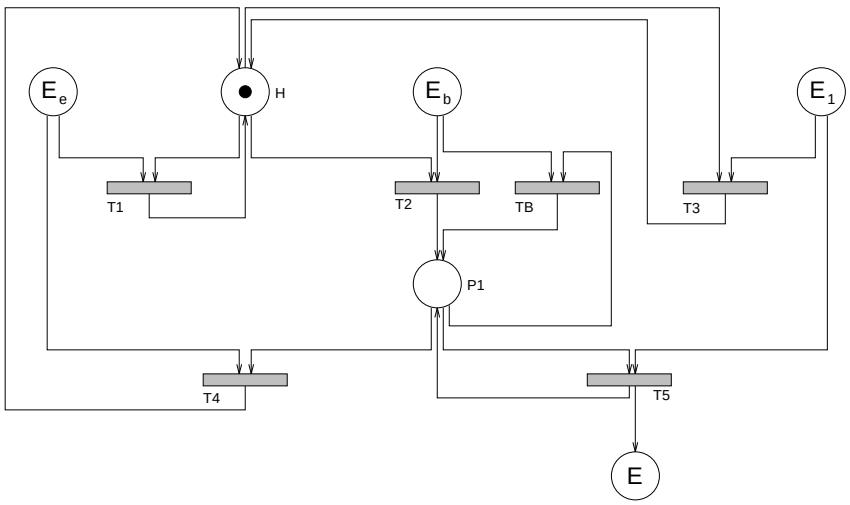

Abbildung 6.4: Petrinetz für $E := E_1$ **in** (E_b, E_e)

6.2.2.2 Not in

Die Petrinetz-Struktur für das komplexe Ereignis $E := E_1$ **not in** (E_b, E_e) wird in Abbildung 6.5 gezeigt (Platz P1 wird als *Speicherplatz* bezeichnet). Das komplexe Ereignis E tritt nur dann ein, wenn das Ereignis E_1 *nicht* im Intervall liegt, d.h. Anfangs- und Endereignis müssen in der richtigen Reihenfolge eintreten, ohne dass das Ereignis E_1 dazwischen eintritt. Der genaue Ablauf ist wie folgt:

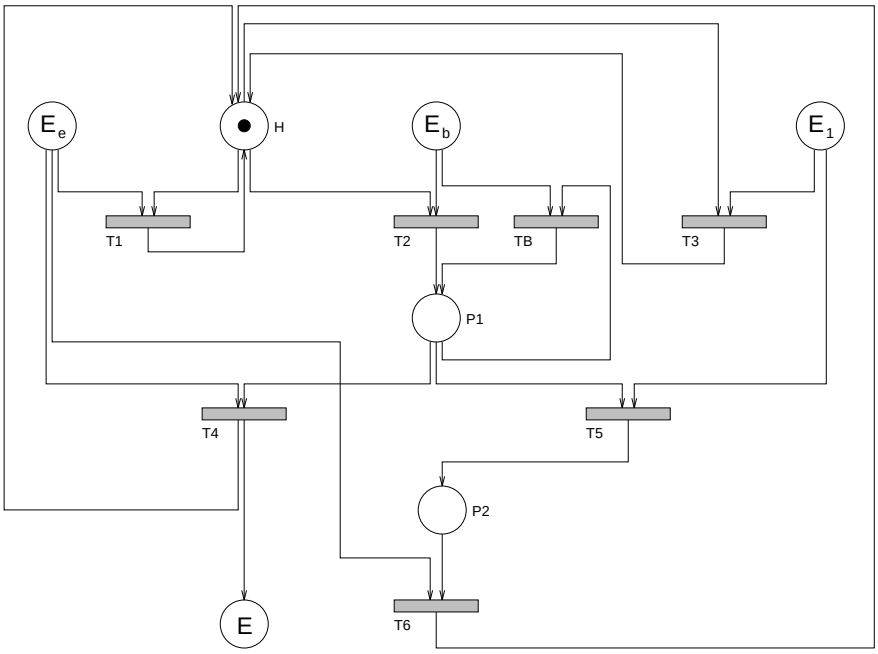

Abbildung 6.5: Petrinetz für $E := E_1$ **not in** (E_b, E_e)

- Falls als erstes das Anfangsereignis E_b eintritt, d.h. in E_b ein Token zu liegen kommt, so wird die Transition T2 feuern. Die Token in H und in E_b werden gelöscht und ein neues Token wird in P1 plaziert. Das Intervall ist nun *offen*.
  - Falls als erstes ein Endereignis E_e eintritt, so wird dieses mit Hilfe des Tokens in Hilfsplatz H und der Transition T1 aus dem Platz E_e entfernt.
  - Falls als erstes das Ereignis E_1 eintritt, so wird dieses mit Hilfe des Tokens in Hilfsplatz H und der Transition T3 aus dem Platz E_1 entfernt.
- Falls bei offenem Intervall das Endereignis E_e eintritt, so kann die Transition T4 feuern. Die Token werden aus E_e und aus P1 entfernt und ein neu erzeugtes Token wird in E und in H plaziert. Das komplexe Ereignis ist somit eingetreten, und der Operator zurückgesetzt.
  - Falls bei offenem Intervall das Ereignis E_1 eintritt, so kann die Transition T5 feuern. Die Token werden aus E_1 und P1 entfernt und ein neu erzeugtes Token wird in P2 gelegt. Die Transition T6 muss nun solange warten, bis das Endereignis E_e eintritt, welches das Intervall schliesst. Wenn dies der Fall ist, so wird die Transition T6 feuern, ein neues Token in H legen und somit den Operator in seinen Ursprungszustand versetzen.
  - Falls bei offenem Intervall das Ereignis E_b eintritt, so wird dieses mit Hilfe des Tokens in P1 und der Transition TB geschluckt.

6.2.2.3 The nth in

Abbildung 6.6 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **the nth** E_1 **in** (E_b, E_e). Das komplexe Ereignis E tritt genau dann ein, wenn E_1 bei offenem Intervall n-mal eingetreten ist. Damit die Transition T5 erst bei n-maligem Eintreten des Ereignisses feuern kann, ist eine einfache Form eines Kantenausdruckes notwendig. Dieser Kantenausdruck (in Abb. 6.6 als n bezeichnet, auf der Kante von E_1 nach T5) ist eine ganze Zahl, die angibt, wieviele Token sich im Platz befinden müssen, damit die Transition feuern kann. Der Platz P1 in Abbildung 6.6 wird als *Speicherplatz* bezeichnet. Der genauere Ablauf dieses Operators ist wie folgt:

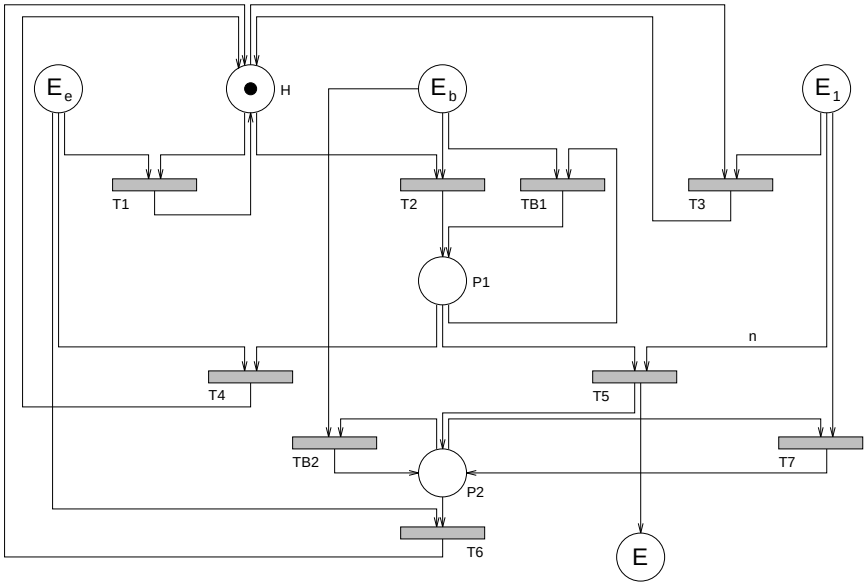

Abbildung 6.6: Petrinetz für $E :=$ **the nth** E_1 **in** (E_b, E_e)

- Tritt bei geschlossenem Intervall das Beginnereignis E_b ein, so wird durch Feuern der Transition T2 das Token aus H und E_b entfernt und ein neues Token in P1 gelegt. Das Intervall ist somit *offen*.
  - Tritt n-mal das Ereignis E_1 ein, so kann die Transition T5 feuern. Sämtliche n Token werden aus dem Platz E_1 entfernt, ebenso das Token aus P1. Das komplexe Ereignis E wird als eingetreten signalisiert. Zusätzlich wird in P2 ein Token plaziert. Das Netz wartet nun auf das Endereignis E_e des Intervalls.
    - Tritt das Endereignis E_e ein, so kann die Transition T6 feuern, d.h. die Token werden aus E_e und P2 entfernt und das Netz in seinen Originalzustand zurückversetzt.
    - Tritt ein Ereignis E_1 ein, so wird dieses mit Hilfe von T7 und dem Token in Platz P2 geschluckt.
    - Tritt ein Ereignis E_b ein, so wird dieses mit Hilfe von TB2 und dem Token in Platz P2 geschluckt.
  - Tritt das Endereignis E_e ein, so wird das Token aus P1 und E_e beim Feuern von T4 entfernt, ein neues Token in H plaziert und das Netz somit in seinen Originalzustand zurückgesetzt.
  - Tritt das Beginnereignis ein, so wird das Token in E_b mittels der Transition TB1 und dem Token in P1 geschluckt.

- Tritt bei geschlossenem Intervall das Endereignis E_e ein, so wird es mit Hilfe des Tokens in H und der Transition T1 geschluckt.
- Tritt bei geschlossenem Intervall das Ereignis E_1 ein, so wird dieses mit Hilfe des Tokens in H und der Transition T3 geschluckt.

6.2.2.4 Last in

Abbildung 6.7 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **last** E_1 **in** (E_b, E_e) (Die Plätze P1 und P2 werden als *Speicherplätze* bezeichnet). Das komplexe Ereignis wird genau dann ausgelöst, wenn das Ereignis E_1 zum letzten Mal im Intervall eintritt. Damit festgestellt werden kann, ob es sich tatsächlich um das letzte E_1 Ereignis im Intervall handelt, kann das komplexe Ereignis erst ausgelöst werden, wenn das Intervall durch Eintreten des Endereignisses E_e geschlossen wird. Mit den Transitionen TB1 und TB2 wird wiederum bewirkt, dass bei offenem Intervall — ein Token in P1 oder P2 — eintretende Beginnereignisse (E_b) geschluckt werden.

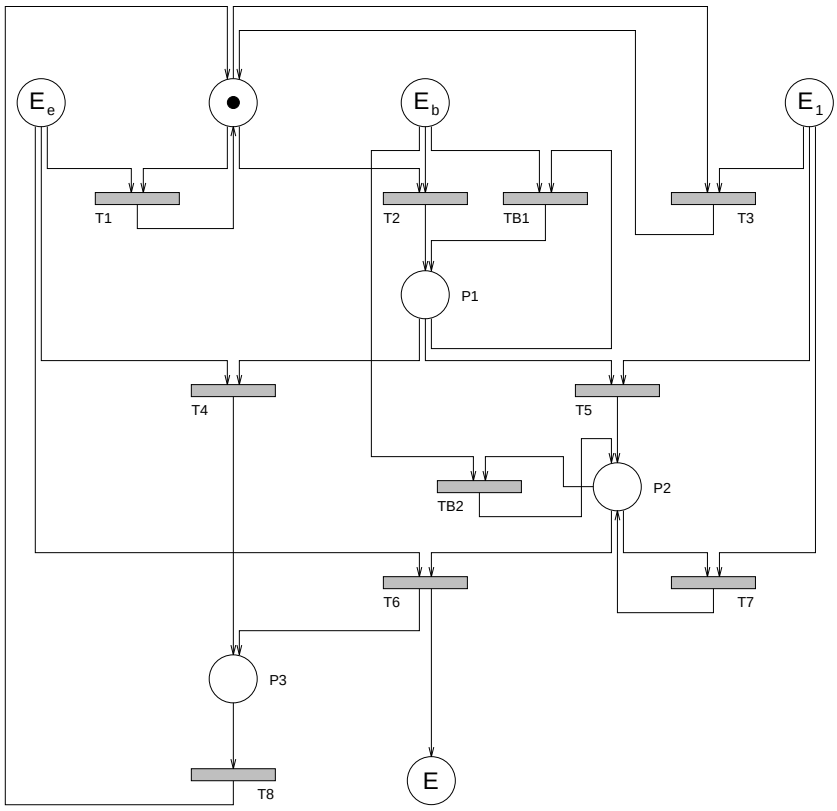

Abbildung 6.7: Petrinetz für $E :=$ **last** E_1 **in** (E_b, E_e)

6.2.3 Wiederholungsoperatoren

ALFRED bietet zwei Operatoren, um ein sich wiederholendes Ereignis zu definieren. Diese Operatoren werden im folgenden näher erläutert.

6.2.3.1 Every $\mathbf{n^{th}}$

Abbildung 6.8 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **every** $\mathbf{n^{th}}$ E_1. Das komplexe Ereignis wird jedesmal ausgelöst, wenn n-mal das Ereignis E_1 eingetreten ist. Dies wird garantiert, indem vom Platz P1 eine Kante nach T1 führt, die eine Konstante (in diesem Fall n) trägt. Diese Konstante gibt an, wieviele Marken sich im Platz P1 befinden müssen, damit die Transition T1 feuern kann. Man beachte, dass der *Every* n^{th}-Operator nur sinnvoll ist für $n > 1$ (Für n=1 entspricht er dem Ereignis E_1 selbst).

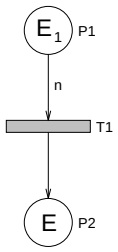

Abbildung 6.8: Petrinetz für $E :=$ **every** $\mathbf{n^{th}}$ E_1

6.2.3.2 Every after n

Abbildung 6.9 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **every after n** E_1. Das Ereignis E wird beim (n+1)-ten, (n+2)-ten etc. Eintreten von Ereignis E_1 ausgelöst. Dieser Operator beinhaltet ebenfalls einen Kantenausdruck (Konstante n). Wenn n E_1 Ereignisse eingetreten sind, so kann die Transition T1 feuern und ein neues Token in P1 legen. Bei jedem weiteren Eintreten von E_1 wird nun die Transition T2 feuern und somit das komplexe Ereignis auslösen. Zusätzlich wird in P1 wieder ein Token plaziert. Der Hilfsplatz H wird benötigt, falls die Konstante n den Wert 1 hat, damit im Petrinetz kein Konflikt entsteht. Ebenso wird er benötigt, falls dieser Operator mit weiteren Operatoren verbunden werden soll (vgl. Abschnitt 6.2.7).

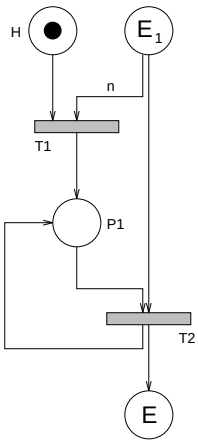

Abbildung 6.9: Petrinetz für $E :=$ **every after n** E_1

6.2.4 Zeitoperatoren

ALFRED bietet fünf mögliche Operatoren um Zeitaspekte zu modellieren. Es sind dies:

1. (**Timespan t after**): Das komplexe Ereignis tritt mit einer Verzögerung von t Zeiteinheiten ein.
2. (**Every time t**): Das komplexe Ereignis tritt jedesmal zur Zeit t ein.
3. (**Every calendar**): Das komplexe Ereignis tritt periodisch ein (täglich, wöchentlich etc.).
4. (**Every x weekday at t**): Das komplexe Ereignis tritt an gewissen Tagen zu einer Zeit t ein (z.B jeden 3. Montag um 15 Uhr).
5. (**Every x weekday in month at t**): Das komplexe Ereignis tritt an gewissen Tagen im Monat zu einer Zeit t ein (z.B. jeden 2. Tag im Monat um 12 Uhr).

Bei den letzten vier Zeitoperatoren kann zur Definitionszeit angegeben werden, ab welchem Zeitpunkt der Operator aktiv sein soll. Dies kann der Definitionszeitpunkt oder der Eintrittszeitpunkt eines Ereignisses sein.

Die Behandlung von einfachen Zeitereignissen, d.h. das Eintreten eines bestimmten Zeitpunktes, erfolgt in ALFRED durch die *Time Event Detection (TED)* (vgl. Kapitel 8). Damit eine zeitliche Verzögerung modelliert werden kann, wird der TED der zu überwachende Zeitpunkt übergeben. Bei gegebener Zeit signalisiert die TED durch Markierung eines entsprechenden Platzes im Netz, das Eintreten des Ereignisses.

Nachfolgend werden die zwei ersten Zeitoperatoren ausführlicher dargestellt und darauf hingewiesen, wie die restlichen Zeitoperatoren modelliert werden können.

6.2.4.1 Timespan t after

Abbildung 6.10 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **timespan t after** E_1. Das komplexe Ereignis wird nur dann ausgelöst, wenn, nachdem das Ereignis E_1 eingetreten ist, t Zeiteinheiten vergangen sind. Dies wird wie folgt gelöst:

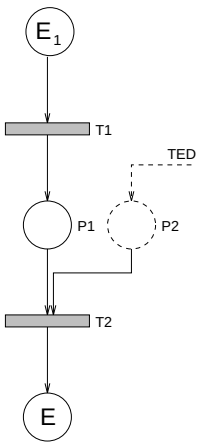

Abbildung 6.10: Petrinetz für $E :=$ **timespan t after** E_1

- Falls das Ereignis E_1 eingetreten ist, so kann die Transition T1 feuern und somit das Token aus E_1 entfernen und ein neues Token in P1 legen. Zusätzlich übergibt nun T1 der TED den zu überwachenden Zeitpunkt ((Zeitpunkt des Feuerns von T1) + (t Zeiteinheiten für die Verzögerung)) und den Platz, den die TED zu gegebener Zeit markieren muss (P2). Das neue Token, das in P1 gelegt wurde, erhält als Identifikation ebenfalls den zu betrachtenden Zeitpunkt.

- Die Transition T2 kann nicht feuern, solange in P2 kein Token liegt. Dieses Token wird von der TED erst nach einer Verzögerung um t Zeiteinheiten in P2 gelegt.

- Ist die gewünschte Zeitspanne abgelaufen, so kommt in P2 ein Token zu liegen. Da sich in P1 schon ein Token mit demselben Zeitstempel befindet (Identifikation), kann die Transition T2 nun feuern. Ein Token wird aus P1 und P2 entfernt, und das Eintreten des komplexen Ereignisses E signalisiert.

Der Operator *Timespan t after* kann problemlos überlappend verwendet werden. Wenn der Platz P1 schon markiert ist, d.h. die Transition bereits auf das Ende der Zeitverzögerung wartet, kann ein Ereignis E_1 eintreten. Dieses Eintreten wird genau gleich behandelt, d.h. die Transition T1 übergibt der TED wiederum die entsprechende Zielzeit und feuert das Token nach P1. Die Transition T2 wird jeweils dann feuern, wenn von der TED ein Token in P2 zu liegen kommt, und signalisiert somit das Eintreten des komplexen Ereignisses E.

6.2.4.2 Every time t

Abbildung 6.11 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **every time t** *beginning at X*, wobei X sowohl *now* oder E_1 sein kann. Der Unterschied der beiden Operatoren liegt in der Startzeit. Der erste Operator wird von dem Moment an aktiviert, an dem er definiert wurde (now). Der andere Operator wird erst aktiv, wenn ein Ereignis E_1 eingetreten ist.

Das komplexe Ereignis kann jedesmal nur dann eintreten, wenn die Marke t Zeiteinheiten in P1 oder PA verweilt ist. Dies wird auf die gleiche Art sichergestellt wie schon beim Operator *timespan t after*. Die entsprechende Zeitverzögerung t und der zu markierende Platz (P2 resp. PB) werden der TED beim Feuern der Transition T1 resp. TA mitgeteilt. Somit kann die Transition T2 resp. TB erst feuern, wenn von der TED eine Marke in P2 resp. PB gelegt wird. Da mit diesem Operator eine zeitliche Wiederholung modelliert wird, muss die Transition T2 resp. TB dieselbe Funktionalität wie T1 resp. TA aufweisen, d.h. der TED müssen beim Feuern dieser Transitionen wiederum die Zeitverzögerung und der zu markierende Platz mitgeteilt werden.

Der Operator {**every time t** *beginning at* E_1} weist gegenüber dem Operator {**every time t** *beginning at now*} zusätzliche Elemente auf. Diese Elemente sind:

- **Hilfsplatz H**: Mit diesem Hilfsplatz wird garantiert, dass nicht gleichzeitig zwei oder mehr Token in den Wiederholungskreislauf eingespeist werden. Der Operator ist von dem Moment an aktiv, wenn das erste Ereignis E_1 eintritt. Alle weiteren E_1 Ereignisse werden nicht mehr berücksichtigt, da der Operator "ewig" läuft.

- **Transition TX**: Die Verbindung von E_1 und PA mit der Transition TX stellt sicher, dass bei "gestartetem" Operator die nachfolgenden E_1 Ereignisse geschluckt werden, da die Token sonst im Platz E_1 gespeichert würden. Diese Speicherung hat für den Operator, falls er für sich alleine betrachtet wird, keine Auswirkung. Die Speicherung wird aber beim Zusammensetzen von verschiedenen Operatoren (vgl. Abschnitt 6.2.7) problematisch.

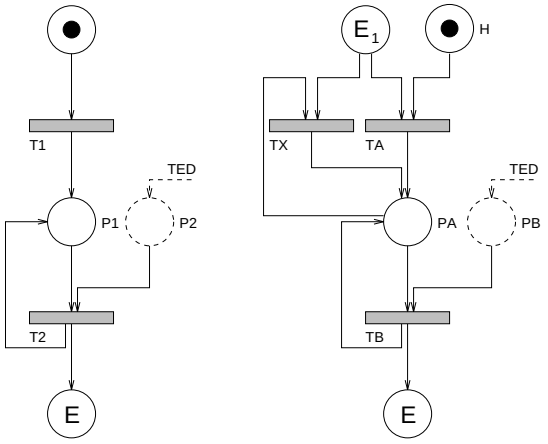

Abbildung 6.11: Petrinetz für $E :=$ **every time t** *beginning at now* oder E_1

6.2.4.3 Weitere Zeitoperatoren

Für die Modellierung der restlichen Zeitoperatoren **Every calendar**, **Every x weekday at t** und **Every x weekday in month at t** bedarf es keiner neuen Strukturen mehr. Mit Hilfe der Petrinetze für die ersten beiden Zeitoperatoren können die übrigen Zeitoperatoren ebenfalls dargestellt werden. Dazu muss jedoch die Transition, welche der TED den Auslösungszeitpunkt übergibt, die entsprechende Zielzeit ausrechnen können. Die unterschiedlichen Zeitoperatoren wurden in ALFRED definiert, um dem Benutzer eine einfachere Eingabe von Zeitereignissen zu ermöglichen.

6.2.5 Sequenzoperatoren

In ALFRED werden zwei Sequenzoperatoren unterstützt, um die Eintrittsreihenfolge von Ereignissen definieren und erkennen zu können. Diese zwei Operatoren unterscheiden sich semantisch erst, wenn die Teilereignisse komplex sind. Dann werden die Eintrittsreihenfolgen der Ereignisse unterschiedlich streng gehandhabt.

6.2.5.1 Weak sequence

Abbildung 6.12 zeigt das Petrinetz für ein komplexes Ereignis $E :=$ **weak sequence**(E_1, E_2). Das komplexe Ereignis E wird nur dann ausgelöst, wenn das Ereignis E_2 *nach* dem Ereignis E_1 eintritt. Falls E_2 vorher eintritt, so wird es mittels dem Token in H und der Transition T2 geschluckt. Falls E_1 und E_2 komplexe Ereignisse sind, so dürfen Teile des Ereignisses E_2 vor Teilen des Ereignisses E_1 auftreten.

6.2.5.2 Strong sequence

Die *strong sequence* stellt eine Verschärfung des zuvor betrachteten Sequenzoperators dar. Abbildung 6.13 zeigt die Petrinetz-Struktur für ein komplexes Ereignis $E :=$ **strong sequence**(E_1, E_2) (Platz P1 wird als *Speicherplatz* bezeichnet). Wenn die Teilereignisse komplex sind, so verlangt *strong sequence*, dass zuerst das ganze Ereignis E_1 eingetreten sein muss, bevor auch nur ein Teil des zweiten Ereignisses E_2 eintreten darf. Dies wird erreicht, indem für jedes primitive Teilereignis von E_2 eine zusätzliche Transition generiert wird, die mit dem Hilfsplatz H und dem primitiven Ereignis in derselben Weise verbunden werden muss, wie dies zwischen H, T2 und E_2 in Abbildung

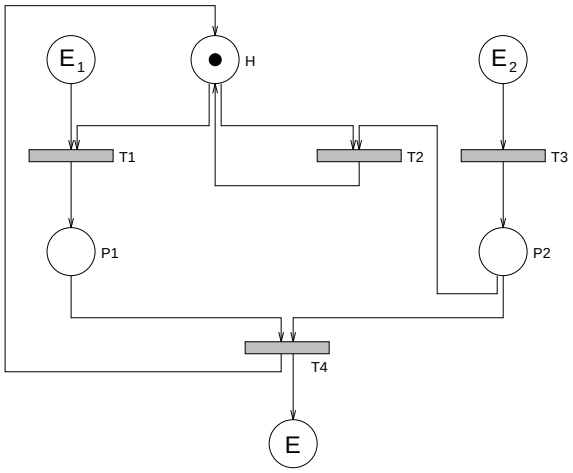

Abbildung 6.12: Petrinetz für $E :=$ **weak sequence**(E_1, E_2)

6.13 geschehen ist. Mit Hilfe dieser Verbindungen werden sämtliche Teilereignisse von E_2 gelöscht, die vor dem Ereignis E_1 eintreten (vgl. Abschnitt 6.2.7).

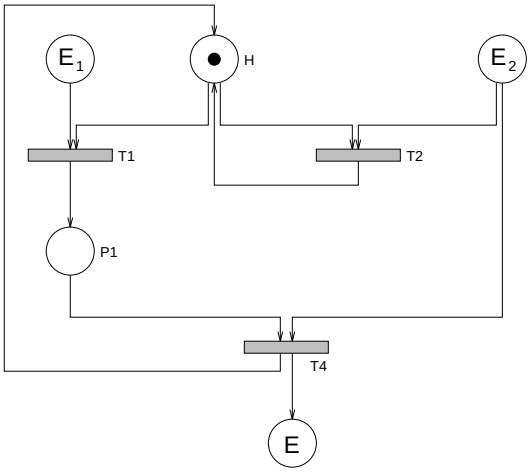

Abbildung 6.13: Petrinetz für $E :=$ **strong sequence**(E_1, E_2)

6.2.6 Auswahloperatoren

Die beiden Auswahloperatoren von ALFRED werden nicht durch eigene Petrinetzstrukturen definiert, sondern als Kombination von bool'schen Operatoren und dem Wiederholungsoperator *every* n^{th}.

6.2.6.1 Weak choice

Ein komplexes Ereignis $E :=$ **weak choice** *n of M* besagt, dass n Ereignisse aus der Ereignismenge M eintreten müssen. Dabei kann auch mehrmals dasselbe Ereignis eintreten. Folglich kann dieser

Operator mit Hilfe von *or* und *every* n^{th} dargestellt werden. Ein Ereignis $E :=$ *weak choice 2 of* (E_1, E_2) lässt sich als $E :=$ *every* $2^{nd}(E_1$ *or* $E_2)$ umschreiben (vgl. Abbildung 6.14).

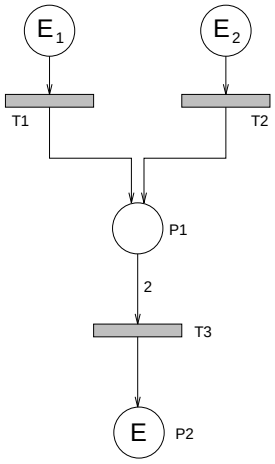

Abbildung 6.14: Weak choice umgeschrieben mit *or* und *every* n^{th}

6.2.6.2 Strong choice

Im Gegensatz zur *weak choice* müssen bei einem komplexen Ereignis $E :=$ **strong choice** n *of* M die Teilereignisse, die eintreten, paarweise verschieden sein. Ein Ereignis $E :=$ *strong choice 2 of* (E_1, E_2, E_3) wird genau dann ausgelöst, wenn 2 Ereignisse aus der Ereignismenge eintreten. E tritt also ein, wenn irgend eine Kombination von 2 Ereignissen aus der Ereignismenge eingetreten ist. Diese Kombinationsmöglichkeit kann mit Hilfe von *and* und *or* dargestellt werden. Das komplexe Ereignis E lässt sich also wie folgt darstellen (vgl. Abbildung 6.15): $E :=$ *(*E_1 *and* E_2*) or (*E_1 *and* E_3*) or (*E_2 *and* E_3*)*. Die Transitionen T1, T2 und T3, sowie deren Vorplätze, stellen die Verknüpfung mit *or* dar. In diesem Fall könnten sie wegfallen (Optimierung), wodurch die Semantik des Netzes nicht verändert würde.

6.2.7 Mehrstufige komplexe Ereignisse

Mit den vorgestellten Ereignisoperatoren ist es möglich, komplexe Ereignisse zu modellieren, die nur aus primitiven Teilereignissen bestehen. Damit aber mit Hilfe dieser Operatoren beliebig komplexe Ereignisstrukturen aufgebaut werden können, muss eine entsprechende Bildungsvorschrift existieren.

Die Zusammensetzung der Ereignisoperatoren ist mit Ausnahme der *strong sequence* und den Intervalloperatoren trivial. Die einzelnen Operatoren werden im Baukastenprinzip zusammengesetzt, in dem Sinne, dass der Ausgangsplatz eines übergeordneten Ereignisses wiederum Eingangsplatz des untergeordneten Ereignisses wird. Nicht ganz so trivial ist jedoch die Zusammensetzung für die verschiedenen Intervalloperatoren und den Operator *strong sequence*.

- **Intervalloperatoren**
  Durch die Integration eines Intervalloperators wird ein neuer Zeitkontext definiert. Ein Ereignis, das auf einen Intervalloperator aufgesetzt wird, steht nicht mehr im Kontext der *globalen* Zeitgeschichte, sondern in Bezug zu den Intervallgrenzen. Das aufgesetzte Ereignis darf für

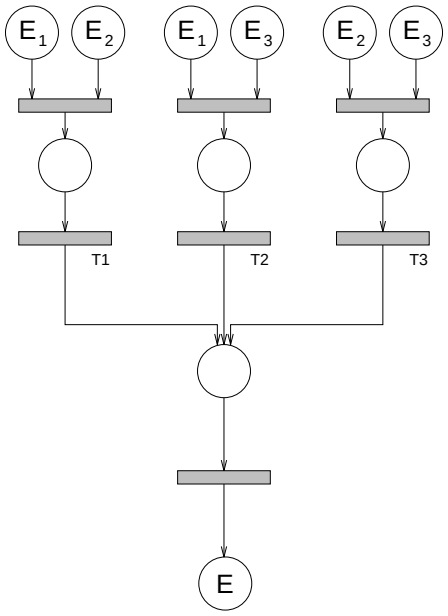

Abbildung 6.15: Strong choice umgeschrieben mit *and* und *or*

die Erkennung des darunterliegenden Ereignisses erst berücksichtigt werden, wenn das darunterliegende Intervall geöffnet ist, d.h. das Anfangsereignis des Intervalls eingetreten ist. Somit dürfen also auch keine Teile des aufgesetzten Operators vor dem Beginn des Intervalls eintreten.

- **Strong sequence**
  Die Semantik der *Strong sequence* besagt, dass keine Teile des späteren Sequenzereignisses vor dem ersten Sequenzereignis eintreten dürfen. Dieser Sachverhalt muss beim Zusammensetzen der Ereignisoperatoren berücksichtigt werden.

Um die Bildungsvorschrift verstehen zu können, werden vorab einige Begriffe und Strukturen erläutert.

6.2.7.1 Begriffserläuterungen

Die nun folgenden Begriffe werden benötigt, damit die für die Bildung wichtigen Plätze und Transitionen eindeutig referenziert werden können.

- **Hilfsplatz**
  Hilfsplätze sind diejenigen Plätze im Petrinetz, die standardmässig mit einem Token belegt sind. Sie garantieren beim jeweiligen Operator, dass der Ablauf semantisch korrekt erfolgt (Platz H bei allen Operatoren, die einen Hilfsplatz benötigen, vgl. Abbildung 6.2 bis 6.13).

- **Speicherplatz**
  In allen Intervalloperatoren sowie in den Sequenzoperatoren existieren Plätze, in welche im Verlaufe der Ereigniserkennung Token zu liegen kommen, die nicht sofort weitergefeuert werden können. Diese Plätze "speichern" also sozusagen das Token für eine gewisse Zeit. Die folgenden Plätze werden als *Speicherplätze* definiert. Sie wurden in den jeweiligen Operatoren bereits definiert.

- *in*: Platz P1 in Abbildung 6.4.
- *not in*: Platz P1 in Abbildung 6.5.
- *the n^{th} in*: Platz P1 in Abbildung 6.6.
- *last in*: Plätze P1 **und** P2 in Abbildung 6.7.
- *strong sequence*: Platz P1 in Abbildung 6.13.

Alle anderen Plätze in den Ereignisoperatoren, die auch Token zwischenspeichern können, werden in diesem Kontext *nicht* als Speicherplätze bezeichnet.

- **Startereignisse**
  Als *Startereignisse* werden diejenigen Ereignisse bezeichnet, mit deren Eintreten ein neuer Zeithorizont beginnt. Dies sind bei Intervalloperatoren die Beginnereignisse, beim Operator *strong sequence* das erste Ereignis und bei allen anderen Operatoren die primitiven Teilereignisse. Dabei wird der Begriff *Startereignis* nur bei zusammengesetzten Operatoren verwendet.
- **Starttransitionen**
  Starttransitionen werden diejenigen Transitionen genannt, die unmittelbar nach einem *Startereignis* stehen und mit einem *Hilfsplatz* verbunden sind.

6.2.7.2 Zusätzliche Strukturen

Damit die Petrinetze korrekt zusammengefügt werden können, müssen zusätzliche Verbindungsstrukturen definiert werden. Diese Strukturen erlauben die zeitlich und semantisch korrekte Verarbeitung des komplexen Petrinetzes. Die notwendigen Strukturen sind *coordinator* und *resolver*.

Coordinator: Der *coordinator* stellt sicher, dass Ereignisse, die eintreten, aber im Moment für den Operator nicht von Bedeutung sind, aus den Plätzen entfernt werden. Der *coordinator* (vgl. Abbildung 6.16) verbindet jedes *Startereignis* der aufgesetzten Struktur über eine zusätzliche Transition mit dem *Hilfsplatz* der Basisstruktur.

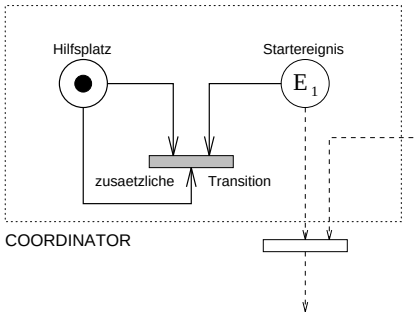

Abbildung 6.16: Der Coordinator

Der *coordinator* wird auch in einigen Operatoren intern verwendet, um den korrekten Ablauf zu garantieren (z.B. Verbindung von H, E_e und T1 in Abbildung 6.4).

Resolver: Der *resolver* (vgl. Abbildung 6.17) verbindet die *Starttransitionen* des Operators mit dem *Speicherplatz* des Operators. Durch das Einfügen eines *coordinators* können im Petrinetz Konflikte entstehen, die durch den *resolver* beseitigt werden.

Da der *resolver* Speicherplätze verbindet, gilt es folgenden Spezialfall zu beachten:

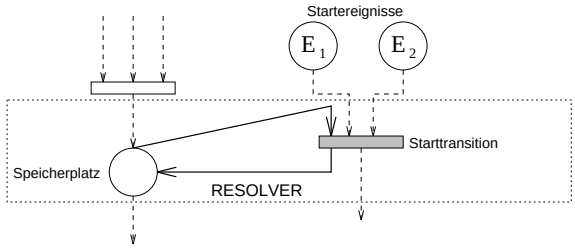

Abbildung 6.17: Der Resolver

Spezialfall: Da der Intervalloperator *last in* zwei Speicherplätze hat, müssen für diesen Operator *zwei resolver* hinzugefügt werden. Der erste *resolver* wird wie oben beschrieben in das Netz eingefügt. Damit das Petrinetz die gewünschte Semantik aufweist, muss beim zweiten *resolver* eine zusätzliche Transition generiert und die Plätze entsprechend verbunden werden. In Abbildung 6.18 ist ein solcher doppelter *resolver* für ein Ereignis $\{last\ (E_1\ or\ E_2)\ in\ (E_b, E_e)\}$ zu sehen.

Bei dieser Kombination von Operatoren sind zwei zusätzliche Koordinatoren notwendig, die garantieren, dass die Ereignisse der aufgesetzten Sequenz nicht vor Beginn des Intervalls in den Operator gelangen. Diese zwei Koordinatoren verbinden jeweils den Hilfsplatz des Intervalls über eine zusätzliche Transition mit den *Startereignissen* E_1 resp. E_2. Durch diese Koordinatoren kann nun ein Konflikt im Netz entstehen. So kann beim Eintreten von z.B. E_1 sowohl die Transition T1 als auch die Transition des entsprechenden Koordinators feuern. Aus diesem Grund muss ein Resolver eingefügt werden (dünne durchgezogene Linien). Wenn sich bei offenem Intervall das Token nun nicht mehr im Speicherplatz 1 sondern im Speicherplatz 2 befindet, d.h. das komplexe Ereignis ist mindestens einmal im Intervall eingetreten, so würde der erste Resolver verhindern, dass weitere Ereignisse E_1 und E_2 berücksichtigt werden, da die Transitionen T1 und T2 nicht mehr feuern können. Deshalb muss bei diesem Intervalloperator ein zweiter Resolver eingefügt werden (dicke durchgezogene Linien). Die folgende Beschreibung zeigt, wie dieser Resolver eingefügt werden muss. Die Numerierung der Elemente in Abbildung 6.18 entspricht dabei den Nummern in folgender Aufzählung.

1. Verdopple die Starttransitionen (T1 und T2)
2. Verbinde alle primitiven Ereignisse vor den Starttransitionen (Startereignisse) mit den Kopien der Starttransitionen
3. Verbinde jede kopierte Transition mit dem Platz, der auf die ursprüngliche Transition folgt
4. Führe eine Kante von den kopierten Transitionen zum zweiten *Speicherplatz*
5. Führe eine Kante vom zweiten *Speicherplatz* zu den kopierten Transitionen.

6.2.7.3 Der Verbindungsalgorithmus

In Anhang B ist ein Algorithmus in Pseudocode angegeben, der die Vorschrift für das Zusammensetzen von Ereignisoperatoren, d.h. das Verbinden von Operatoren mittels den Strukturen *coordinator* und *resolver*, definiert.

Ein Ereignis, das aus mehreren komplexen Ereignissen zusammengesetzt ist, kann als *Ereignisbaum* angesehen werden, in dem Sinne, dass der äusserste Operator die Wurzel des Baumes bildet und alle darauf aufgesetzten Operatoren als innere Knoten oder als Blattknoten des Baumes angesehen werden. Unter dieser Betrachtungsweise zeigt der rekursive Algorithmus folgendes Verhalten, wobei angenommen wird, dass sich die Blattknoten unten und die Wurzel oben im Baum befinden.

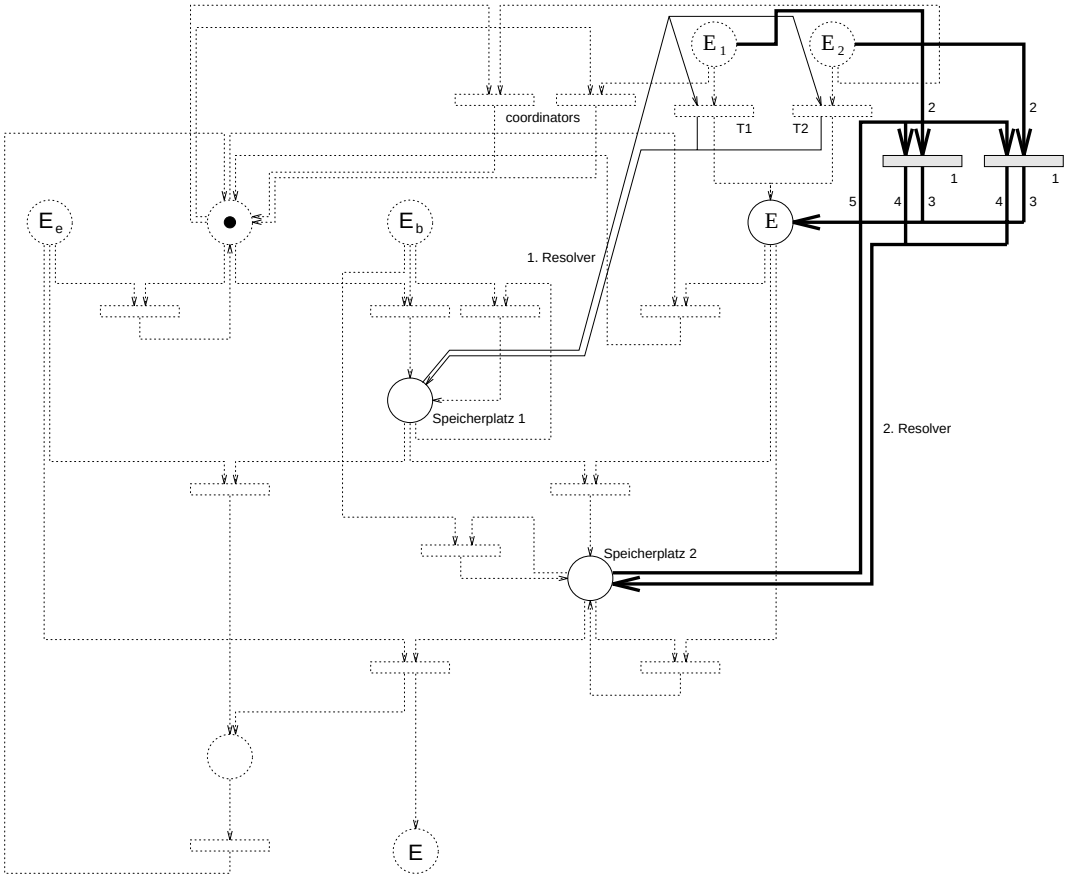

Abbildung 6.18: Spezialfall des Resolvers für *last* (E_1 *or* E_2) *in* (E_b, E_e)

1. Gehe bis zu den primitiven Ereignissen hinunter (Blattknoten)
2. Gehe schrittweise den Baum hinauf und
   - Nimm alle für ein möglicherweise übergeordnetes Intervall oder eine möglicherweise übergeordnete *strong sequence* notwendigen *Startereignisse* und *Starttransitionen* mit
   - Verbinde, falls bei einem Intervall oder bei einer *strong sequence* angelangt, *Startereignisse* (Plätze) und *Starttransitionen* mit *coordinator* und *resolver*
   - Gehe weiter den Baum hinauf. Falls von einem Intervall oder einer *strong sequence* ausgehend, so merke die neuen *Startereignisse* und *Starttransitionen*.

Die *Startereignisse* und *Starttransitionen*, ausgehend von einem Intervall oder einer *strong sequence*, werden also im Ereignisbaum solange gesucht, bis eine der folgenden Situationen eintritt. Dazu müssen die Situationen genau in der untenstehenden Reihenfolge erkannt werden!

1. Suche bis ein Intervall gefunden wurde. Hier wird nur das Beginnereignis aufgeschlüsselt (falls komplex). Endereignis und Intervallereignis stehen unter der Kontrolle des gefundenen Intervalls.
2. Suche bis eine *strong sequence* gefunden wurde. Hier wird das erste Sequenzereignis aufgeschlüsselt (falls komplex). Das zweite Sequenzereignis steht unter der Kontrolle der gefundenen Sequenz.

3. Suche bis ein primitives Ereignis gefunden wurde. Dieses wird als *Startereignis* betrachtet. Die *Starttransition* zu diesem Ereignis ist diejenige, die auf dieses primitive Ereignis folgt, und falls die Nachfolgetransition nicht eindeutig ist, so ist es jene, die mit dem Hilfsplatz der entsprechenden Struktur verbunden ist.

6.2.7.4 Reset

Bei einstufigen komplexen Ereignissen wird die Petrinetz-Struktur so gewählt, dass sich die Operatoren nach erfolgtem Eintreten des komplexen Ereignisses selbst zurücksetzen, falls dies notwendig ist. Wenn aber mehrstufige komplexe Ereignisse betrachtet werden, so ist es möglich, dass ein "äusserer" Operator beendet wird und die "inneren" Operatoren daraufhin zurückgesetzt werden müssen. Um diese Rücksetzung realisieren zu können, gibt es folgende zwei Möglichkeiten:

1. Zusätzliche Strukturen einfügen, mit deren Hilfe die Token solange durch das Petrinetz transportiert werden, bis alle Operatoren inklusive Teiloperatoren in den Anfangszustand zurückgesetzt sind.
2. Ein zusätzlicher Algorithmus, der beim Eintreten des komplexen Ereignisses sämtliche Teiloperatoren zurücksetzt.

Der Vorteil der ersten Variante ist der, dass nicht ein zusätzliches Programm benötigt wird, um diese Rücksetzung zu realisieren. Die Strukturen, die für diesen *reset* notwendig wären, würden jedoch bei der Verarbeitung einen zu hohen Zeitaufwand erfordern. In ALFRED wird aus diesem Grund die zweite Variante gewählt.

Ein *reset* ist nur dann notwendig, wenn im komplexen Netz Token zwischengespeichert sind, welche die Semantik des Netzes beeinflussen könnten. Dies ist zum Beispiel der Fall, wenn als Basisoperator ein Intervall verwendet wird, und darauf ein weiteres komplexes Ereignis aufgesetzt wird. Da das Intervall durch Eintreten des Endereignisses beendet werden kann, während der aufgesetzte Operator mitten in der Erkennung steckt, kann es vorkommen, dass im aufgesetzten Operator Token liegen bleiben. Diese Token müssen entfernt werden, da beim nächsten Beginnereignis des Intervalls der zeitliche Kontext neu gesetzt wird.

In ALFRED sorgt während der Ereigniserkennung ein Algorithmus für die korrekte Rücksetzung der Ereignisstrukturen. Wenn ein Intervall durch das Eintreten des Endereignisses geschlossen wird, so werden durch den Reset-Algorithmus alle auf dem Intervall aufgesetzten komplexen Ereignisse ebenfalls zurückgesetzt, d.h. der Algorithmus geht durch den Baum, dessen Wurzel durch das Intervallereignis dargestellt wird, entfernt sämtliche Token aus den Plätzen und fügt in den Hilfsplätzen jeweils ein Token ein. Somit sind sämtliche Ereignisoperatoren zurückgesetzt.

6.2.7.5 Beispiel eines zusammengesetzten Ereignisses

In Abbildung 6.19 ist ein vollständiges komplexes Ereignis dargestellt. Es soll die oben beschriebenen Plätze, Transitionen und Strukturen aufzeigen. Als Beispiel wird das komplexe Ereignis $\{$*(strong sequence*(E_1, E_2)*) in* $(E_{beg}, E_{end})\}$ verwendet. Es wurde hier absichtlich ein abstraktes Beispiel gewählt, da die Verbindungsstrukturen im Vordergrund stehen. Die Basisstruktur dieses komplexen Ereignisses ist also ein *every in*-Intervall auf welches eine *strong sequence* aufgesetzt wird. Anfangs- und Endereignis des Intervalls sind primitiv.

Der zusätzliche *coordinator* sorgt dafür, dass das Ereignis E_1 geschluckt wird, wenn das Intervall noch nicht geöffnet ist. Da durch diesen *coordinator* ein Konflikt im Netz entsteht — ein Token in E_1 könnte sowohl durch Transition T1 als auch durch Transition T2 gefeuert werden — muss ein *resolver* eingefügt werden. Dieser stellt sicher, dass die Transition T2 erst feuern kann, wenn das Intervall geöffnet ist, d.h. ein Token in P1 liegt.

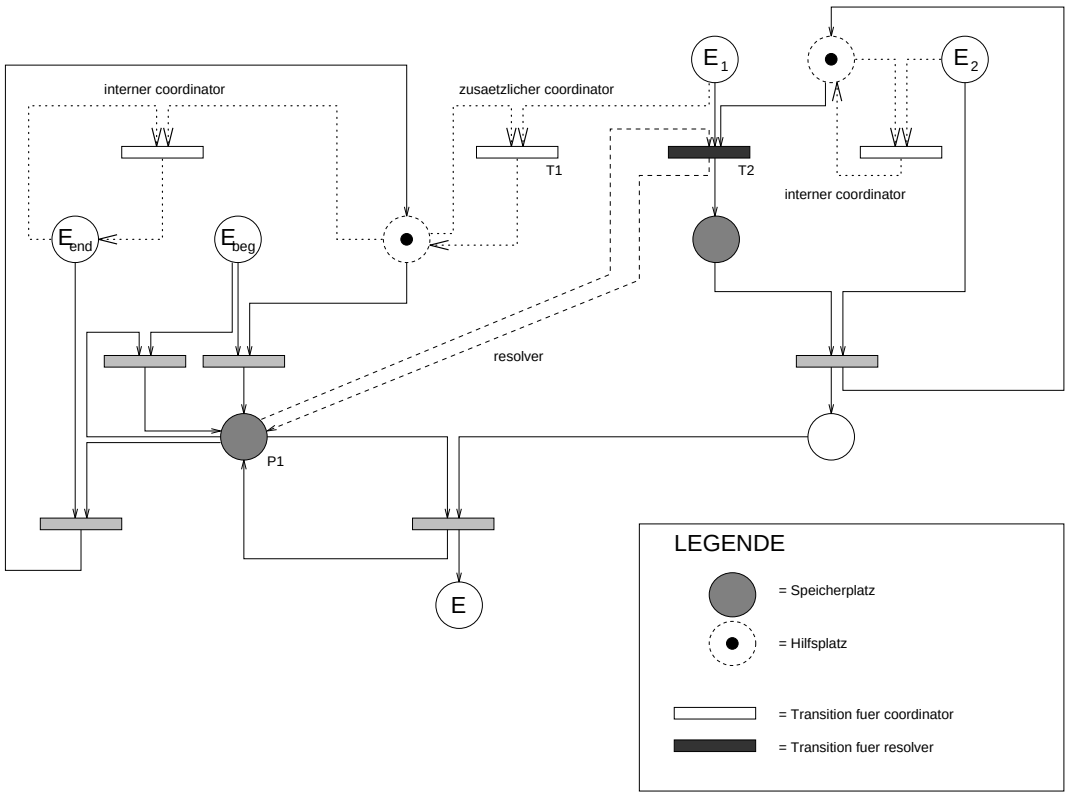

Abbildung 6.19: Komplexes Ereignis *(strong sequence(E_1, E_2)) in (E_{beg}, E_{end})*

6.2.8 Parameterbindung

In den Bedingungs- und Aktionskomponenten einer Regel muss auf die Informationen bei der Ereigniserkennung zurückgegriffen werden können. Aus diesem Grund müssen Parameter definiert werden, die beim Erkennen eines primitiven Ereignisses gebunden werden (Parameter von komplexen Ereignissen werden aus denjenigen der primitiven Ereignisse zusammengesetzt). Wenn ein primitives Ereignis eintritt, so wird ein Token in den für das primitive Ereignis zuständigen Platz (Primärplatz) gelegt und die erforderlichen Parameterwerte werden im Token gespeichert. Somit werden die Parameterwerte beim Durchlaufen des Petrinetzes zusammen mit den Token transportiert und können beim Feuern der Transitionen berücksichtigt werden.

Nun muss definiert werden, welche Parameter existieren, welche Parameterwerte gebunden werden müssen und wie diese verschiedene Parameter beim Feuern einer Transition kombiniert werden sollen.

6.2.8.1 Parameter

In ALFRED werden mehrere Parameter für Ereignisse unterstützt. Beispiele dafür sind:

- *Occ_prim_Event*
  Der Zeitpunkt, an dem das primitive Ereignis eingetreten ist.
- *Occ_comp_Event*
  Der Zeitpunkt, an dem das komplexe Ereignis eingetreten ist, d.h. der Zeitpunkt, an dem das letzte Teilereignis des komplexen Ereignisses eingetreten ist.

- *Transaction_id*
  Die Nummer der Transaktion, in welcher das primitive Ereignis eingetreten ist.
- *User_id*
  Die Identifikation des Benutzers, dessen Befehl ein primitives Ereignis ausgelöst hat.
- *New_values*
  Die Datensätze, die durch die Aktion, welche das primitive Ereignis auslöste, eingefügt, geändert, gelöscht oder selektiert wurden.
- *Actual_values*
  Die aktuellen Datensätze, die durch ein instanzorientiertes Ereignis gebunden werden. Diese Datensätze sind Teil der *neuen* Datensätze (New_values).
- *Old_values*
  Die Datensätze, welche sich vor der Änderung durch die entsprechende Aktion in der Datenbank befanden.
- *Database_id*
  Ein Identifikator, welcher angibt, in welcher Datenbank das Ereignis eingetreten ist.

6.2.8.2 Parameteroperatoren

Die Parameter von komplexen Ereignissen werden durch die Parameter der zugehörigen primitiven Ereignisse mittels *Parameteroperatoren* gebildet. Sie werden als Kantenausdrücke in den Ereignisoperatoren dargestellt. Die Parameteroperatoren, die in ALFRED unterstützt werden, sind:

- *Projektion* ($@_{left}$ oder $@_{right}$)
  Mit diesem Operator werden nur die Parameter der angegebenen Seite verwendet
  - $@_{left}$ nimmt nur denjenigen Parameterwert, der bei einer Transition an dasjenige Token gebunden ist, welches sich im linken Platz befindet. Dabei muss die Transition zwei Eingänge haben, d.h. Arität 2 besitzen.
  - $@_{right}$ analog für rechts.
- *2er-Kombination* (&(a,b))
  Durch diesen Operator werden zwei Parameter kombiniert, d.h. sie werden in einer neuen Parameterstruktur gesammelt und gemeinsam gespeichert. Es kann jedoch auf die einzelnen Parameterwerte zugegriffen werden.
- *Verschmelzung* ($\sim$())
  Die Verschmelzung ist eigentlich eine 1er-Kombination mit n gleichen Ereignissen. Dieser Operator wird verwendet, wenn mehrere gleiche Ereignisse, resp. deren Parameter, kombiniert werden sollen.
- *first()*
  Dieser Operator wird verwendet, um jene Parameter zu übernehmen, welche den ältesten Zeitstempel tragen, also jene Parameter, die zuerst gebunden wurden.
- *last()*
  Mit diesem Operator werden jene Parameter referenziert, die zuletzt gebunden wurden.

6.2.8.3 Parameterkontexte

Parameterkontexte werden in komplexen Ereignissen verwendet, um festzulegen, welche Parameter bei der Ereigniserkennung gebunden werden müssen. Die vier möglichen Kontexte wurden bereits in Abschnitt 2.3.2.6 definiert, werden hier aber noch einmal kurz erläutert:

- Parameterkontext *recent* besagt, dass die *jüngsten* Parameterwerte gebunden werden müssen.
- Parameterkontext *chronicle* definiert, dass die *ältesten* Parameterwerte gebunden werden müssen.
- Durch den Parameterkontext *cumulative* wird festgelegt, dass die Parameter aller Teilereignisse gebunden werden müssen, sie also kombiniert werden.
- Im Parameterkontext *continuous* werden Kombinationsmöglichkeiten von Parametern betrachtet. Dabei werden sämtliche Teilereignisse, welche die Erkennung des komplexen Ereignisses einleiten, als potentielle Parameterkandidaten angesehen.

6.2.8.4 Beispiel

Die in Abschnitt 6.2 definierten Ereignisoperatoren wurden aus Gründen der Übersichtlichkeit nicht mit Parameterkontexten versehen. Stattdessen wird hier an einem einzelnen Ereignisoperator gezeigt, welche Auswirkungen Parameterkontexte auf die Semantik des Operators haben. Als Beispiel dient der Operator **Every n$^{\mathbf{th}}$** in Abbildung 6.20. Der Kantenausdruck *oper()* auf der Kante von T1 nach P2 steht als Platzhalter für die eigentlichen Parameteroperatoren, die je nach Kontext dort stehen müssen. Die folgende Liste zeigt den notwendigen Operator bei gegebenem Parameterkontext:

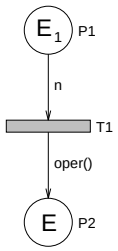

Abbildung 6.20: Petrinetz für $E :=$ **every n$^{\mathbf{th}}$** E_1 mit noch unbestimmtem Parameterkontext

- *chronicle*
  Bei chronicle muss derjenige Parameter (resp. Token) berücksichtigt werden, der als *erster* gebunden wurde, d.h. das älteste Token. Aus diesem Grund muss oper() durch den Operator *first()* ersetzt werden.
- *recent*
  Dieser Kontext stellt den Normalfall des Operators dar. Es wird nur jener Parameter verwendet, der als *letztes* gebunden wurde, d.h. dasjenige Token, das als letztes in E_1 zu liegen kam. In diesem Fall muss für oper() der Operator *last()* eingesetzt werden.
- *cumulative*
  Bei diesem Kontext spielen alle Parameterwerte eine Rolle, die zur Auslösung des komplexen Ereignisses beigetragen haben. Aus diesem Grund müssen alle Parameter aus E_1 kombiniert und weitergereicht werden. Für oper() muss also $\sim()$ eingesetzt werden.

Die obenstehenden Parameterkontexte lassen sich mit geringem Aufwand in die bestehenden Petrinetze integrieren. Anders sieht es beim Kontext *continuous* aus. Da in diesem Kontext verschiedene Kombinationen von Parametern berücksichtigt werden müssen, d.h. Parameterwerte werden mehrfach verwendet, muss die Semantik des Petrinetzes abgeändert werden. Dies kann mittels eines Algorithmus geschehen, wodurch aber die Semantik des Petrinetzes nicht mehr ersichtlich wäre. Eine zweite Möglichkeit ist die Änderung der Petrinetzstruktur, die nachfolgend exemplarisch aufgezeigt wird.

Abbildung 6.21 zeigt das Petrinetz für das Ereignis $\{every\ n^{th}\ E_1\}$ im Parameterkontext *continuous*. In diesem Kontext werden sämtliche Ereignisse, welche die Erkennung des komplexen Ereignisses auslösen, als potentielle Kandidaten betrachtet. Aus diesem Grund müssen die Parameterwerte, die noch für weitere Kombinationen verwendet werden, in den Ausgangsplatz zurücktransportiert werden. Die Semantik des Operators ist in diesem Beispiel also wie folgt: Falls die Transition T1 feuern kann, so wird in E ein Token generiert, das sämtliche Parameterwerte der n Token aus E_1 enthält (Parameteroperator $\sim$ ()). Zusätzlich werden nun die letzten n-1 Token wieder in den Platz E_1 gelegt (Parameteroperator $\sim$ () $- first()$), daher werden das erste Token und dessen Parameter gelöscht. Somit kann bei wiederholtem Eintreten von E_1 eine neue Parameterkombination errechnet werden.

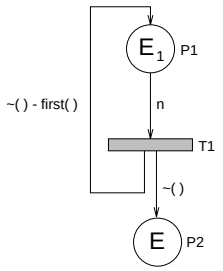

Abbildung 6.21: Petrinetz für $E :=$ **every** $\mathbf{n^{th}}$ E_1 im Kontext continuous

Die Parameterkontexte beziehen sich jeweils nur auf einen Ereignisoperator und haben keinen Einfluss auf weitere aufgesetzte Operatoren. Jeder Operator besitzt also seinen eigenen Parameterkontext, der sich auf das betreffende primitive oder komplexe Ereignis bezieht. Es bleibt jedoch offen, welche Semantik ein Ereignis besitzt, welches aus verschiedenen Ereignisoperatoren mit unterschiedlichen Parameterkontexten zusammengesetzt ist. Aus diesem Grund wurde bei der Zusammensetzung von komplexen Ereignisoperatoren auf die Verwendung von Parameterkontexten verzichtet.

6.3 Bedingungskomponente

In ALFRED wird zwischen primitiven und komplexen Bedingungen unterschieden, die bereits in Abschnitt 3.3.2 erläutert wurden.

Solange man sich auf primitive Bedingungen beschränkt, kann die Bedingung durch eine einzige Transition dargestellt werden. Falls jedoch komplexe Bedingungen betrachtet werden, so muss der Bedingungsteil in seine Einzelteile (d.h. in mehrere primitive Bedingungen) zerlegt werden. Der Grund für diese Zerlegung ist, dass z.B. durch ein im Bedingungsteil ausgeführtes *retrieve* ein Ereignis eintreten kann, welches erkannt werden muss.

6.3.1 Entscheidungskomponente

Bei einer Bedingung, d.h. bei einer Bedingungstransition, muss bei einer ECAA-Regel ein Entscheid gefällt werden, ob der Aktionsteil für TRUE oder der Aktionsteil für FALSE verarbeitet werden muss. Diese Entscheidung, die im Petrinetz als Verzweigung dargestellt wird, kann nicht durch einen gewöhnlichen Petrinetz-Algorithmus realisiert werden. Dieser Algorithmus hätte zur Laufzeit nicht genügend Informationen, um entscheiden zu können, in welchen der beiden Teile er verzweigen muss. Um diese Entscheidung dennoch modellieren zu können, gibt es zwei Lösungsansätze:

1. Der Simulationsalgorithmus muss "intelligenter" werden, d.h. der Algorithmus feuert nicht nur die Token nach den Schaltregeln durch das Netz, sondern muss auch zusätzliche Aufgaben erfüllen (zusätzliche Logik).

2. Zu gegebener Zeit werden Informationen von aussen in das Netz eingefügt, d.h. es werden Marken in bestimmte Plätze gelegt. Mit diesen Marken könnte der Algorithmus sodann entscheiden, in welchen Teil er verzweigen muss.

In den Petrinetzen von ALFRED wird die erste Variante verwendet, womit bei der Verarbeitung des Bedingungsteils von der gewöhnlichen CPN-Verarbeitung abgewichen wird. Diese Verzweigung wird durch eine speziell gekennzeichnete Bedingungstransition dargestellt, welche einen Eingangsplatz und zwei Ausgangsplätze (true_out und false_out) hat (vgl. Abbildung 6.22). Beim Feuern dieser Transition muss der Algorithmus aufgrund des Wahrheitswertes des Tokens entscheiden, ob das Token in den *true_out* oder *false_out* Platz gelegt werden muss.

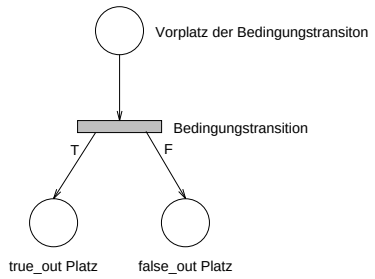

Abbildung 6.22: Struktur der Bedingungstransition

6.3.2 Strukturen für komplexe Bedingungen

Komplexe Bedingungen werden, wie oben schon erwähnt, in primitive Bedingungen aufgespalten. Die so gewonnenen primitiven Bedingungen müssen auf geeignete Weise verkettet werden, damit die komplexe Bedingung modelliert werden kann. Diese Verkettung geschieht mit Hilfe der Strukturelemente *Sequenz, Auswahl* und *Wiederholung*, wie sie aus Programmiersprachen bekannt sind.

Um die Bedingung gegenüber Ereignis und Aktion abzugrenzen, und um die Verarbeitung der Regel zu vereinfachen, wird die ganze Bedingung mit Hilfe von zwei Transitionen "eingeklammert".

6.3.2.1 Klammerung des Bedingungsteils

Die Klammerung des Bedingungsteils geschieht mittels zweier Transitionen und Plätze. Die Anfangstransition (*begin of condition*) wird unmittelbar an das *Ereignisende* angehängt. Die Endtransition (*end of condition*) ist diejenige Transition, bei welcher der Petrinetz-Algorithmus entscheiden

muss, ob in den Wahr- oder Falsch-Teil zu verzweigen ist. Mit dieser letzten Transition wird der Bedingungsteil abgeschlossen.

6.3.2.2 Konjunktionsoperator

Mit Hilfe einer AND-Transition kann der Konjunktionsoperator modelliert werden. Die AND-Transition hat zu diesem Zweck zwei Eingangsplätze und einen Ausgangsplatz (vgl. Abbildung 6.23). Dieser Operator ist von der Struktur her identisch mit dem Ereignisoperator *and*, im Bedingungsteil sind jedoch die Rahmenbedingungen anders. Bei der Auswertung von primitiven Bedingungen wird dem Token ein Wahrheitswert zugewiesen (TRUE oder FALSE), welcher dann zu weiteren Transitionen transportiert wird. Durch die AND-Transition und den dazugehörigen Petrinetz-Algorithmus wird nun sichergestellt, dass das Token, welches in den Ausgangsplatz der Transition gelegt wird, den richtigen Wert erhält, d.h nur den Wert TRUE erhält, wenn beide Token in den Eingangsplätzen den Wert TRUE haben. Dazu muss beim Feuern der Transition nicht nur kontrolliert werden, ob in allen Eingangsplätzen ein Token vorhanden ist, sondern es muss auch der Wahrheitswert der Token berücksichtigt werden.

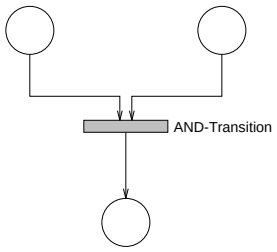

Abbildung 6.23: Petrinetz für Konjunktionsoperator

6.3.2.3 Disjunktionsoperator

Der Disjunktionsoperator kann mit Hilfe einer OR-Transition modelliert werden. Diese Transition wird genau gleich dargestellt wie die AND-Transition (vgl. Abbildung 6.23), nur dass sie entsprechend gekennzeichnet ist (*or* statt *and*). Der Petrinetz-Algorithmus muss beim Feuern der Transition entscheiden, ob in beiden Eingangsplätzen ein Token liegt, und ob mindestens eines der beiden Token den Wert TRUE enthält.

6.3.2.4 Auswahl

Die Auswahl (*if-then-else*) ist im Grunde genommen eine Bedingung. Demzufolge wird diese Kontrollstruktur durch eine Bedingungstransition eingeleitet. Diese Transition hat zwei mögliche Ausgänge, den *true_out* und den *false_out* Platz. An diese Plätze werden die jeweiligen Aktionstransitionen angehängt, d.h. die Transitionen für den Then-Teil und Transitionen für den Else-Teil. Beide Teile werden am Ende wieder zusammengeführt — one-entry, one-exit — (vgl. Abbildung 6.24). Der Petrinetz-Algorithmus muss beim Feuern der Bedingungstransition aufgrund des Wahrheitswertes des Tokens entscheiden, in welchen Teil verzweigt werden soll.

6.3.2.5 Wiederholung

Eine Iteration (*while*) lässt sich folgendermassen in einem Petrinetz abbilden (vgl. Abbildung 6.25). Die Schleifenbedingung wird mittels einer Bedingungstransition realisiert, die wiederum zwei mögli-

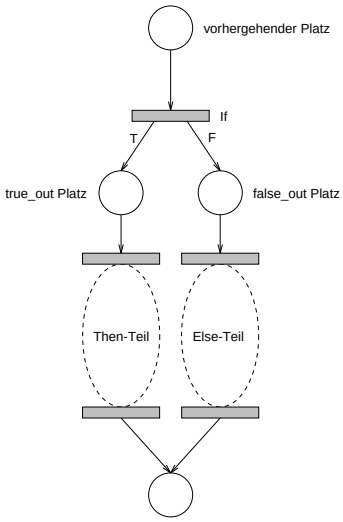

Abbildung 6.24: Petrinetz für Auswahl (if-then-else)

che Ausgänge hat. Alle Aktionen des Bedingungsteils, die in der Schleife ausgeführt werden sollen, werden an den Wahr-Zweig der Bedingungstransition gehängt. Die letzte Transition der Schleife wird mit dem Platz vor der Bedingungstransition (*while*) verbunden.

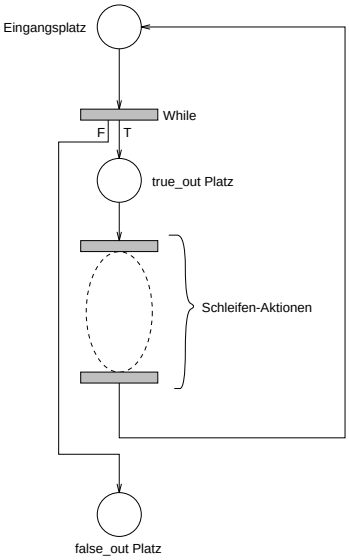

Abbildung 6.25: Petrinetz für Wiederholung (while)

6.3.3 Beispiel einer Bedingungsstruktur

Anhand eines einfachen Beispiels wird gezeigt, wie eine komplexe Bedingung mit Hilfe eines Petrinetzes abgebildet werden kann. Die Bedingung lautet:

- `(Employee.gehalt > 5´000) AND (Employee.gehalt < 10´000)`

Diese Bedingung verlangt, dass der Lohn eines Angestellten (`Employee.gehalt`) zwischen 5000 und 10000 liegt. In Abbildung 6.26 ist das entsprechende Petrinetz abgebildet.

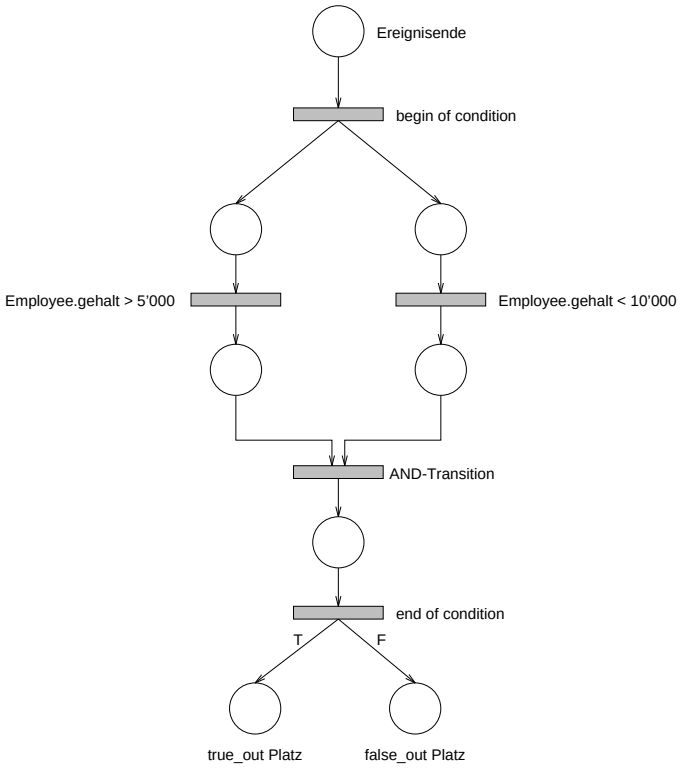

Abbildung 6.26: Petrinetz für eine komplexe Bedingung

Wenn ein Token in das *Ereignisende* gelegt wird, so kann die Transition *begin of condition* feuern. Das Token wird sooft vervielfacht (in diesem Fall zweimal), wie dies notwendig ist. Nun sind die beiden Transitionen, welche die primitiven Bedingungen darstellen, zum Feuern bereit. Je nach dem, ob die parallele Auswertung von Teilbedingungen unterstützt wird, können diese beiden Teile gleichzeitig (parallel) oder sequentiell verarbeitet werden. Das Token, welches über die entsprechende Transition gefeuert wird, erhält als Wahrheitswert das Resultat der Bedingungsauswertung (TRUE oder FALSE). Wenn nun in beiden Plätzen vor der *AND-Transition* ein Token liegt, d.h. beide Teilbedingungen wurden ausgewertet, so ist diese Transition feuerbereit. Der Petrinetz-Algorithmus muss die Wahrheitswerte der Token überprüfen, entfernt die Marken aus den Eingangsplätzen und generiert aufgrund der gefundenen Wahrheitswerte ein neues Token im Ausgangsplatz. Mit diesem Schritt ist die eigentliche Auswertung der Bedingung zu Ende. Die Marke wird nun mit Hilfe von *end of condition* je nach Wahrheitswert in den *false_out* oder *true_out Platz* gelegt.

6.4 Aktionskomponente

In ALFRED werden sowohl primitive als auch komplexe Aktionen unterstützt, die bereits in Abschnitt 3.3.3 erläutert wurden. Die komplexe Aktion einer Regel wird in primitive Aktionen aufgesplittet und im Netz dargestellt. Für jede so erzeugte primitive Aktion wird eine eigene Transition generiert, welche die Aktion im Petrinetz darstellt. Damit alle Aktionen, die ALFRED ermöglicht,

auch im Petrinetz dargestellt werden können, müssen analog zur Bedingungskomponente die Kontrollstrukturen *Sequenz, Auswahl* und *Wiederholung* unterstützt werden.

6.4.1 Strukturen für komplexe Aktionen

Im folgenden werden die notwendigen Strukturen für die Modellierung von komplexen Aktionen erläutert.

6.4.1.1 Klammerung des Aktionsteils

Der Aktionsteil einer Regel folgt bei ECA- und ECAA-Regeln auf die vorhergehenden *true_out* und *false_out* Plätze des Bedingungsteils, bei EA-Regeln auf das Ereignisende. Der Aktionsteil wird, wie auch schon der Bedingungsteil in zwei speziell gekennzeichnete Transitionen eingeschlossen, um eine Abgrenzung zum Bedingungsteil zu erreichen und um die Verarbeitung zu vereinfachen. Diese beiden Transitionen werden *begin_of_action* und *end_of_action* genannt.

6.4.1.2 Sequenz

Um eine Sequenz von primitiven Aktionen im Petrinetz darstellen zu können, wird jede primitive Aktion als Transition dargestellt und in der richtigen Reihenfolge, über zusätzliche Plätze, mit den anderen Transitionen verbunden. Abbildung 6.27 zeigt die Struktur von sequentiellen primitiven Aktionen.

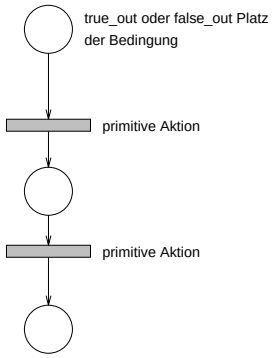

Abbildung 6.27: Primitive Aktionen in Sequenz

Bei der Verarbeitung dieser Struktur werden die Marken sequentiell über die Transitionen transportiert, welche die primitiven Aktionen darstellen. Beim Feuern einer solchen Transition, wird die entsprechende Aktion ausgeführt.

6.4.1.3 Auswahl

Für die Auswahl (*if-then-else*) wird, analog zum Bedingungsteil, eine spezielle Bedingungstransition verwendet, um die Bedingung dieses Aktionsteils darstellen und überprüfen zu können. Diese Transition hat zwei mögliche Ausgänge, der *true_out* und der *false_out Platz*. An diese zwei Ausgangsplätze schliessen sich dann weitere Aktionen an, bis am Ende der Struktur die zwei Teilstücke, d.h. der Then-Teil und der Else-Teil, wieder zusammengefügt werden. Die Auswertung der if-Bedingung und die korrekte Markierung der Ausgangsplätze wird durch den Petrinetz-Algorithmus sichergestellt. Die Auswahl wurde bereits im Abschnitt 6.3.2.4 abgebildet (vgl. Abbildung 6.24).

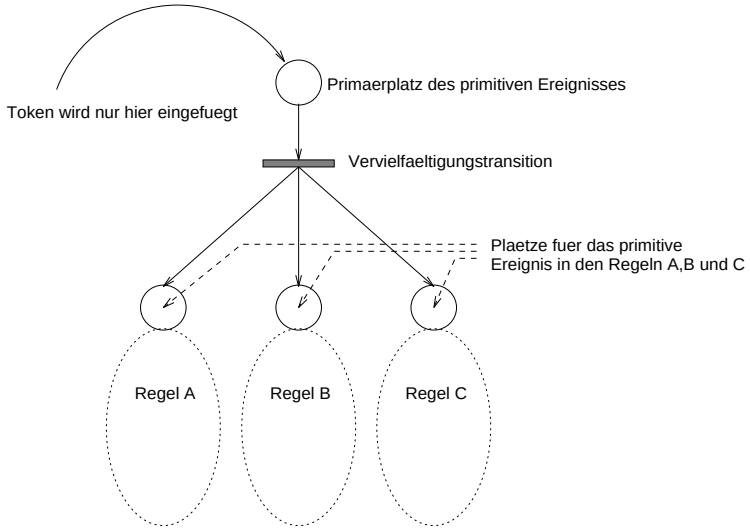

Abbildung 6.28: Vervielfältigung eines Ereignisses

6.4.1.4 Wiederholung

Eine komplexe Aktion einer Regel kann Schleifen beinhalten. In ALFRED wird die vorabprüfende Schleife (*while*) verwendet. Die Struktur für eine solche Schleife wurde bereits in Abschnitt 6.3.2.5 erläutert (vgl. Abbildung 6.25). Die Aktionen innerhalb der Schleife werden solange ausgeführt, bis die Schleifenbedingung falsch ist. Die wiederholte Auswertung der Schleifenbedingung wird erreicht, indem das Token von der letzten Transition innerhalb der Schleife in den Platz vor der *while*-Transition gelegt wird, damit die Bedingung neu ausgewertet werden kann.

6.5 Regelmenge

In den vorhergehenden Abschnitten wurden die Komponenten einer Regel und die internen Verbindungen erläutert. Damit nun eine Regel in die "Umgebung", d.h in die bestehende Regelmenge, integriert werden kann, müssen wohldefinierte Abschlüsse gegen oben (Einstieg) und gegen unten (Ausstieg) existieren.

Da eine Regel andere Regeln auslösen kann, müssen diese Auslösungen mit Hilfe von Verbindungen dargestellt werden, damit diese Regelkaskaden bei der Verarbeitung berücksichtigbar sind.

In den folgenden Abschnitten wird erläutert, wie und wo primitive Ereignisse im Netz dargestellt werden und wie Verbindungen zwischen Regeln realisiert werden. Anhand eines konkreten Beispiels wird sodann erläutert, wie eine mögliche Regelmenge aussehen kann.

6.5.1 Vervielfältigung eines Ereignisses

Ein primitives Ereignis kann in mehreren komplexen Ereignissen von Regeln enthalten sein. Damit diese verschiedenen Plätze nicht verwaltet werden müssen, wird das Petrinetz um ein einfaches Konstrukt erweitert. Jedes primitive Ereignis im Petrinetz wird durch *einen einzigen* Platz im Netz repräsentiert, der als *Primärplatz* des Ereignisses bezeichnet wird. Mit Hilfe einer speziellen Transition (*Vervielfältigungstransition*) wird sichergestellt, dass das Ereignis sooft vervielfältigt wird, wie es in den Ereignisteilen von Regeln auftritt. Somit muss zu jeder Zeit nur der Ort des *Primärplatzes* eines primitiven Ereignisses bekannt sein (vgl. Abbildung 6.28).

Ein komplexes Ereignis, das aus mehreren primitiven Ereignissen besteht, hängt also an so vielen Primärplätzen, wie es unterschiedliche primitive Ereignisse beinhaltet. Abbildung 6.29 zeigt ein komplexes Ereignis $\{E1\ and\ E2\}$, welches mit zwei Primärplätzen verbunden ist.

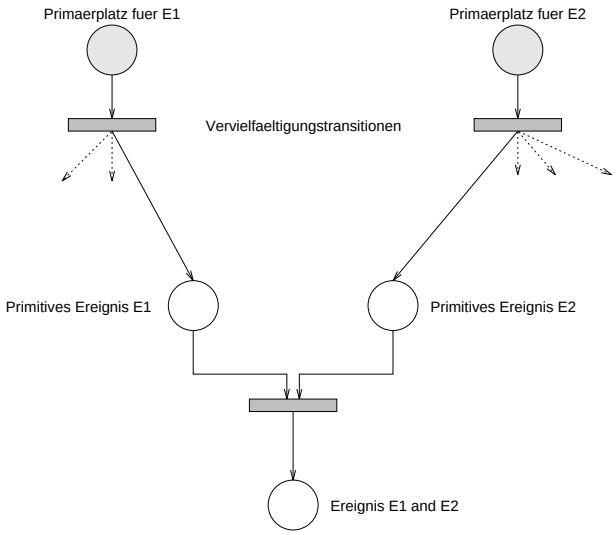

Abbildung 6.29: Vervielfältigung von Ereignissen für *E1 and E2*

6.5.2 Verbindungen zwischen Regeln

Regeln, die in einem ADBS definiert werden, können Aktionen enthalten, deren Ausführung Ereignisse und somit weitere Regeln auslöst. Diese gegenseitige Auslösung von Regeln muss bei der Verarbeitung berücksichtigt werden. Damit dies möglich ist, werden im Petrinetz speziell gekennzeichnete Transitionen eingefügt, die Regeln miteinander verbinden.

Wie bereits in Abschnitt 2.3.2 erwähnt, ist es erforderlich, zwischen den zwei folgenden Möglichkeiten der Regelauslösung zu unterscheiden (Auslösungszeitpunkte):

1. **pre-Auslösung**
   Die Regel wird *vor* der Verarbeitung eines Befehls oder einer Aktion ausgelöst.

2. **post-Auslösung**
   Die Regel wird *nach* der Verarbeitung eines Befehls oder einer Aktion ausgelöst.

Zusätzlich zu diesen beiden Auslösungszeitpunkten, gibt es zwei verschiedene Auslösungsgranularitäten, die *mengenorientierte* und die *instanzorientierte* Auslösung (vgl. Abschnitt 2.3.2.2).

Eine Aktion wird genau zu dem Zeitpunkt verarbeitet, zu dem die Transition, welche die Aktion darstellt, feuern kann. Um nun eine *pre*- und eine *post*-Auslösung realisieren zu können, werden zwei verschiedene Transitionen in das Petrinetz eingefügt. Eine Transition für die *pre*-Auslösung und eine für die *post*-Auslösung. Je nach Auslösungszeitpunkt werden diese Transitionen *vor* oder *nach* der Transition eingefügt, welche die entsprechende Aktion darstellt. Diese Transition wird sodann mit dem *Primärplatz* verbunden, der das zur Aktion gehörende Ereignis repräsentiert. Die Rückverbindung erfolgt erst während der Verarbeitung u.a. anhand der Kopplungsmodi einer Regel.

6.5.3 Beispiel einer Regelmenge

Am Beispiel einer Produktion für Fahrradschläuche wird aufgezeigt, wie ein Petrinetz für eine vorgegebene Regelmenge aussehen kann.

Die Produktion eines Fahrradschlauches erfordert einen Schlauch, bestehend aus einem Gummi und einem Ventil. Die Menge der zur Verfügung stehenden Teile wird in einem DBS abgelegt. Damit bei der Produktion von Schläuchen automatisch die Lagerbestände der Komponenten Gummi und Ventil nachgeführt werden, können folgende zwei (vereinfachte) Regeln definiert werden:

```
Regel 1:  ON  post produziere_Fahrradschlauch
          DO  update  Teile
              set     Bestand = Bestand - 1
              where   Artikel = 'Gummi'

Regel 2:  ON  post update Teile set
          IF  Artikel = 'Gummi'
          DO  update  Teile
              set     Bestand = Bestand - 1
              where   Artikel = 'Ventil'
```

Die erste (EA) Regel basiert auf dem abstrakten Ereignis *post produziere_Fahrradschlauch*. Wenn dieses eintritt, so wird die Lagermenge von *Gummi* in der Relation *Teile* um eins verringert. In der zweiten (ECA) Regel gibt es ein Ereignis, das bei Veränderungen in der Relation *Teile* eintritt (Auslösungszeitpunkt *pre* und Auslösungsgranularität *set*). Diese Regel verringert den Bestand des *Ventils* um eins.

Um die Lagerbestände von *Gummi* und *Ventil* nachzuführen, könnte auch eine Regel mit einer komplexen Aktion definiert werden. Da anhand dieses Beispiels aber die Verbindungen zwischen Regeln aufgezeigt werden sollen, wurde diese Kombination von zwei Regeln gewählt (vgl. Abbildung 6.30).

Regel 1 beinhaltet eine Aktion, durch deren Ausführung das Ereignis *post update Teile set* eintritt. Damit dies dargestellt werden kann, wird eine zusätzliche Transition (gleich benannt wie das entsprechende Ereignis) nach der Aktion von Regel 1 (*update Teile ...*) eingefügt. Von dieser neuen Transition aus wird eine Verbindung zum Primärplatz des entsprechenden Ereignisses gezogen. Mit Hilfe dieser Verbindung wird während der Verarbeitung ein Token von Regel 1 zum Primärplatz von *post update Teile set* transportiert und somit die Ausführung von mit dem Primärplatz verbundenen Regeln berücksichtigt.

Die Aktion in Regel 2 bewirkt ebenfalls eine Änderung in der Relation *Teile*. Da für diese Änderung ein Ereignis in der Regelmenge existiert, muss wiederum eine Transition in der Regel eingefügt werden, mit welcher die Verbindung zwischen der Regel und dem entsprechenden Primärplatz realisiert werden kann. Da das Ereignis *post update Teile set* existiert, wird diese Transition (gleich benannt wie das Ereignis) *nach* der Aktion von Regel 2 eingefügt.

Die Rückverbindungen von den ausgelösten Regeln werden erst während der Verarbeitung, u.a. anhand von Kopplungsmodi, eingefügt (vgl. Kapitel 8).

Wie in Abbildung 6.30 zu sehen ist, besteht bei der Definition von Regelkaskaden die Gefahr, dass ein Zyklus entsteht. Hier in diesem Beispiel löst sich die zweite Regel selber aus. Aus diesem Grund ist es notwendig, dass nach der Definition einer Regel überprüft wird, ob ein Zyklus in der Regelmenge exisitiert, und wenn ja, ob dieser Zyklus terminiert. In diesem Beispiel terminiert der Zyklus, da beim zweiten Eintreten des Ereignisses *post update Teile set* die Bedingung von Regel 2 (`Artikel = 'Gummi'`) nicht mehr erfüllt ist.

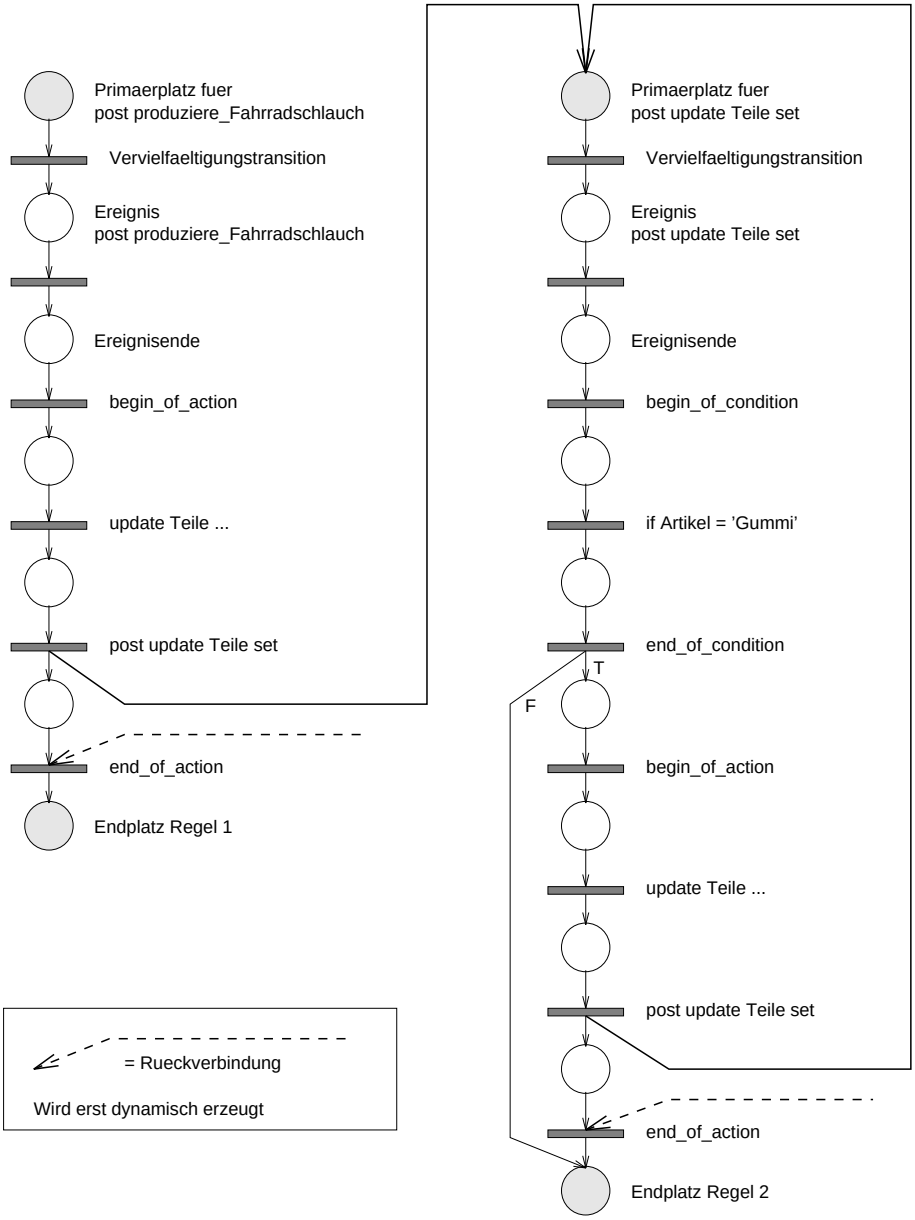

Abbildung 6.30: Eine Beispielregelmenge

Kapitel 7

Terminierung

In einem aktiven Datenbanksystem können Regeln zu unterschiedlichen Zeitpunkten definiert werden. Das Verhalten der so definierten Regelmenge und die Abhängigkeiten zwischen den einzelnen Regeln sind schwer voraussehbar. So können z.B. innerhalb der Regelmenge Kaskaden entstehen, wenn durch die Ausführung einer Regelaktion Ereignisse in der Regelmenge eintreten und somit weitere Regeln ausgelöst werden. Diese Kaskaden können auch zyklisch verlaufen, wenn sich eine Regel selbst direkt oder indirekt über andere Regeln auslöst.

Durch die Existenz einer zyklischen Regelkaskade besteht die Gefahr, dass die Regelverarbeitung nicht abbricht, d.h. dass während der Verarbeitung immer wieder neue Regeln ausgelöst werden. Eine solche endlose Verarbeitung muss verhindert werden. Ein Zyklus kann aber durchaus vom Benutzer erwünscht sein (z.B. für die rekursive Löschung aller Datensätze einer Relation). Dabei sollte der Benutzer aber sicherstellen, dass dieser Zyklus terminiert.

Bei der Analyse einer Regelmenge müssen demzufolge Zyklen erkannt werden, und es muss überprüft werden, ob diese terminieren. Die Analyse kann sowohl *statisch*, d.h. zum Zeitpunkt der Regeldefinition, als auch *dynamisch*, d.h. zur Laufzeit, durchgeführt werden. Bei der statischen Analyse besteht die Einschränkung, dass bestimmte Informationen nicht verfügbar sind (wie z.B. die Eintrittsreihenfolge von Ereignissen oder der Datenbankzustand). Aus diesem Grund lassen sich bei der statischen Analyse oft nur beschränkte Aussagen über die Terminierung einer Regelverarbeitung machen.

Um den Benutzer optimal unterstützen zu können, ist es notwendig, ihn so früh wie möglich auf Zyklen in der Regelmenge hinzuweisen, und ihm wichtige Informationen über den Zyklus zu liefern. Die Analyse der Regelmenge soll also möglichst zum Zeitpunkt der Regeldefinition erfolgen und muss Zyklen erkennen als auch untersuchen können.

Die statische Untersuchung der Zyklen ist von verschiedenen Regeleigenschaften abhängig. Diese Eigenschaften können die statische Analyse sowohl erleichtern als auch erschweren oder sogar unmöglich machen. Beispiele dafür sind:

- *Komplexe Ereignisse*
  Bei Regeln, die durch komplexe Ereignisse ausgelöst werden, müssen sämtliche Teilereignisse und deren Eintrittsreihenfolge berücksichtigt werden. Da diese Reihenfolge nur zur Laufzeit bestimmbar ist, kann die genaue Analyse eines solchen Zyklus nur dynamisch erfolgen.

- *Struktur der Regel*
  Bei ECAA-Regeln müssen beide Aktionszweige für die Analyse der Zyklen betrachtet werden. Die statische Analyse wird dadurch komplexer.

- *Kopplungsmodi*
  Falls eine Regel z.B. mit Kopplungsmodi *immediate/deferred* definiert ist, so ist es möglich, dass sich der Datenbankzustand zwischen dem *immediate*- und dem *deferred*-Teil der Regel ändert, und dass somit ein existierender Zyklus terminieren kann. Die Betrachtung von Kopplungsmodi erschwert die Analyse von Zyklen.

In den folgenden Abschnitten wird die Erkennung und Analyse von Zyklen in ALFRED betrachtet. Dazu werden Zyklen definiert und anhand von Beispielen erläutert. Anschliessend werden Faktoren aufgezeigt, die einen Einfluss auf die Behandlung von Zyklen ausüben. Unter der Betrachtung einiger bestimmter Regeleigenschaften wird danach die Zyklenerkennung und die statische Analyse von Zyklen in ALFRED erläutert.

7.1 Definition der Terminierung

Um die Regelverarbeitung zu starten, muss ein Ereignis von ausserhalb des Systems signalisiert werden. Solche Ereignisse treten als Folge eines Benutzerbefehls, eines Befehls in einem Applikationsprogramm oder eines Zeitpunkts ein. Diese Ereignisse werden als *externe* Ereignisse bezeichnet. Alle übrigen Ereignisse, die durch Aktionsausführungen in der Regelmenge eintreten, bezeichnet man als *interne* Ereignisse.

Alle während der Verarbeitung ausgelösten und noch auszuführenden Regeln werden in eine Menge aufgenommen. In jedem Verarbeitungsschritt muss bestimmt werden, welche Regel aus dieser Menge als nächstes verarbeitet werden soll (z.B. anhand von Prioritäten oder willkürlich). Falls in der Regelmenge eine zyklische Regelkaskade existiert, so ist es möglich, dass diese Menge im Zeitverlauf nie leer wird, und dass die Regelverarbeitung somit nicht abbricht.

Definition 9
*Unter der Annahme, dass zu einem bestimmten Zeitpunkt endlich viele externe Ereignisse eingetreten sind (***Anfangszustand***), und dass im weiteren keine externen Ereignisse mehr eintreten, gilt folgendes:*

- *Eine Regelverarbeitung* **terminiert** *genau dann, wenn gilt: Für jeden beliebigen* Anfangszustand *ist die Menge der ausgelösten Regeln nach endlich vielen Zeiteinheiten leer.*

Gegeben sei eine Menge $\mathcal{R}$ von Regeln. Die Regeln $r_i \in \mathcal{R}$ $(\forall i \in \mathcal{N})$ ($\mathcal{N}$ bezeichnet die Menge der natürlichen Zahlen) bestehen aus den Komponenten Ereignis, Bedingung und Aktion und werden in folgender Form beschrieben: $r_i(e_i, c_i, a_i)$. Mit e_i wird die Ereigniskomponente, mit c_i die Bedingungskomponente und mit a_i die Aktionskomponente der Regel r_i bezeichnet.

Definition 10
Eine Regel $r_i(e_i, c_i, a_i)$ **löst** *eine Regel* $r_j(e_j, c_j, a_j)$ **aus** *(geschrieben als* $r_i \rightarrow r_j$*), wenn durch die Ausführung von* a_i *das Ereignis* e_j *eintritt.*

Auf der Basis von Definition 10 lässt sich nun ein Zyklus innerhalb einer Regelmenge wie folgt definieren:

Definition 11
Ein **Zyklus** *in der Regelmenge existiert, wenn es eine Regel $r_i \in \mathcal{R}$ gibt, für die gilt:*

$$r_i \rightarrow r_{x_1} \rightarrow \ldots \rightarrow r_{x_n} \rightarrow r_i$$

mit:

$$\forall x_j \in \{1, \ldots, n \mid n \in \mathcal{N}, n \geq 0\} : x_j \neq i$$

Definition 12
Eine Regel r_i **löst sich selbst direkt aus** *(geschrieben als $r_i \Rightarrow r_i$) wenn in Definition 11 gilt: $n = 0$*

Eine Regel r_i **löst sich selbst indirekt aus** *(geschrieben als $r_i \stackrel{*}{\Rightarrow} r_i$) wenn in Definition 11 gilt: $n \geq 1$*

Aufgrund von Definition 11 kann die Definition 9 nun präzisiert werden. Dabei gilt: Eine Regelverarbeitung terminiert, falls in der Regelmenge *kein* Zyklus existiert. Falls in der Regelmenge ein Zyklus existiert, so terminiert die Regelverarbeitung genau dann, wenn der Zyklus terminiert.

Um also die Terminierung einer Regelverarbeitung gewährleisten zu können, muss sichergestellt werden, dass die Regelmenge keinen Zyklus enthält, oder dass sämtliche in der Regelmenge existierenden Zyklen terminieren.

7.2 Behandlung von Zyklen

In einem aktiven Datenbanksystem, welches die Analyse der Terminierung unterstützt, muss festgelegt werden, wie Kaskaden und Zyklen in der Regelmenge behandelt werden. Abbildung 7.1 zeigt dafür mögliche Alternativen, die nachstehend erläutert werden:

- **Beschränkte Kaskadentiefe**
  Die Regelmenge wird in diesem Fall nicht analysiert. Stattdessen wird eine feste Zahl vorgegeben, die festlegt, wie tief sich Regeln gegenseitig auslösen dürfen, d.h. wie tief die Regelkaskade sein darf. Beim Erreichen dieser Schranke wird die Verarbeitung abgebrochen. Dabei können zwei Varianten unterschieden werden:

  1. Der letzte Befehl wird zurückgesetzt.
  2. Die ganze Transaktion wird zurückgesetzt.

  Falls eine zyklische Regelkaskade existiert, so wird mit der Beschränkung garantiert, dass die Verarbeitung terminiert.

  Die Festlegung einer Kaskadentiefe hat jedoch einige Nachteile. Es ist z.B. möglich, dass eine gewünschte Regelverarbeitung eine Kaskadentiefe von 10 aufweist, die festgelegte Grenze aber nur bei 5 liegt. Die gewünschte Kaskade könnte in diesem Fall nicht verarbeitet werden.

- **Unbeschränkte Kaskadentiefe**
  Wenn die Kaskadentiefe unbeschränkt ist, so besteht die Gefahr, dass ein Zyklus in der Regelmenge zu einer nicht-terminierenden Verarbeitung führt. In diesem Fall muss festgelegt werden, wie Zyklen zu behandeln sind.

  – **Zyklen nicht zulässig**
    Wenn in der Regelmenge keine Zyklen zugelassen sind, so muss überprüft werden, ob

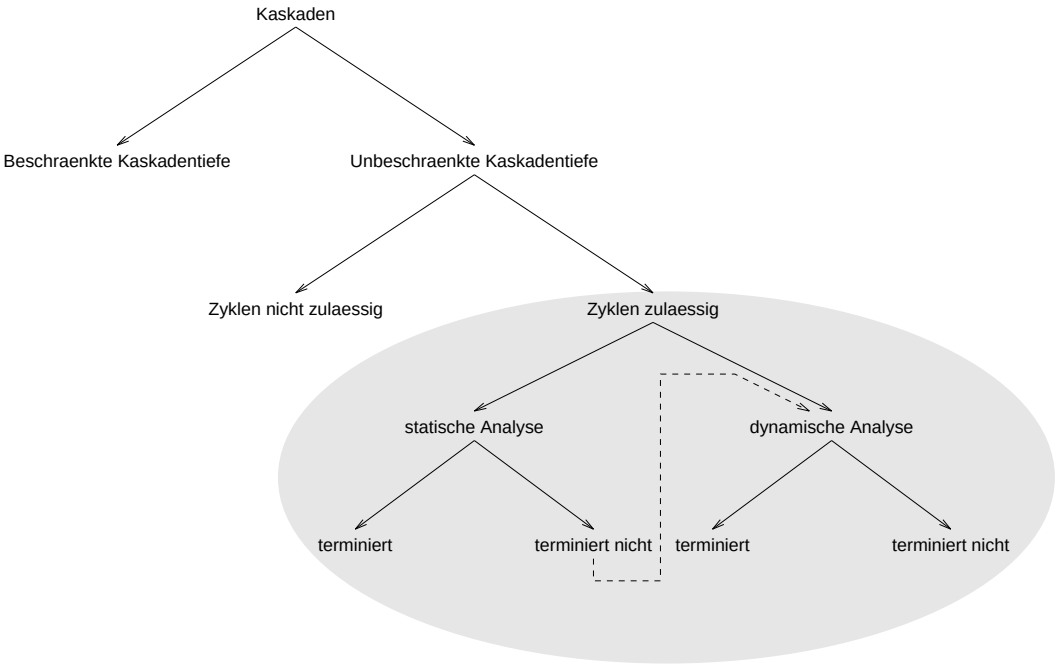

Abbildung 7.1: Behandlung und Analyse von Kaskaden und Zyklen

ein Zyklus existiert. Wenn ja, so muss die den Zyklus erzeugende Regel entfernt werden. Um einen Zyklus zu finden, kann eine statische oder dynamische Analyse durchgeführt werden. Wenn die statische Analyse einen Zyklus findet, so wird die Regel, die den Zyklus verursacht, nicht in die Regelmenge aufgenommen. Wenn keine statische Analyse erfolgt, und der Zyklus erst zur Laufzeit gefunden wird, so muss die dynamische Analyse die den Zyklus verursachende Regel identifizieren. Danach muss die Verarbeitung abgebrochen und die gefundene Regel aus der Regelmenge entfernt werden.

- **Zyklen zulässig**
  Wenn Zyklen erlaubt sind, so müssen diese Zyklen gefunden und weiter analysiert werden. Diese Analyse kann sowohl *statisch* als auch *dynamisch* erfolgen.
  - **Statische Analyse**
    Die statische Analyse erfolgt zum Zeitpunkt der Regeldefinition. Immer wenn eine neue Regel in die Regelmenge eingefügt wird, muss überprüft werden, ob durch die neue Regel ein Zyklus in der Regelmenge entsteht. Wenn dies der Fall ist, so muss überprüft werden, ob dieser Zyklus terminiert.
    Mit Hilfe von bestimmten Kriterien kann in der statischen Analyse bis zu einem gewissen Grad festgestellt werden, ob ein Zyklus terminiert. Wenn dies nicht festgestellt werden kann, so muss zusätzlich eine dynamische Analyse durchgeführt werden.
  - **Dynamische Analyse**
    Die dynamische Analyse erfolgt während der Regelverarbeitung. Die Zyklen, deren Terminierung in der statischen Analyse nicht festgestellt werden konnnte, müssen überwacht werden. Mit Hilfe der Informationen zur Laufzeit (z.B. Datenbankzustand) kann die Terminierung möglicherweise festgestellt werden. Wenn dies nicht der Fall ist, so kann der Zyklus entweder durch eine vorgegebene Kaskadentiefe beschränkt werden, oder er wird vom System nicht akzeptiert.

Der schraffierte Bereich in Abbildung 7.1 zeigt, wie Zyklen in ALFRED behandelt werden. Zyklen sind grundsätzlich zugelassen. Bei der Definition einer Regel wird eine statische Analyse im *Rule Analysis System* (vgl. Abbildung 7.2) durchgeführt. Mit Hilfe der *Rule Cycle Detection (RCD)* werden Zyklen in der Regelmenge erkannt und analysiert. Wenn ein Zyklus gefunden wurde, aber statisch nicht festgestellt werden kann, ob der Zyklus terminiert, so wird dieser zur Laufzeit in der dynamischen Zyklenerkennung (*Dynamic Cycle Detection (DCD)*) untersucht und überwacht. Ein Zyklus, bei dem dynamisch nicht festgestellt werden kann, ob er terminiert, wird nach einer vorgegebenen Anzahl von Durchläufen abgebrochen.

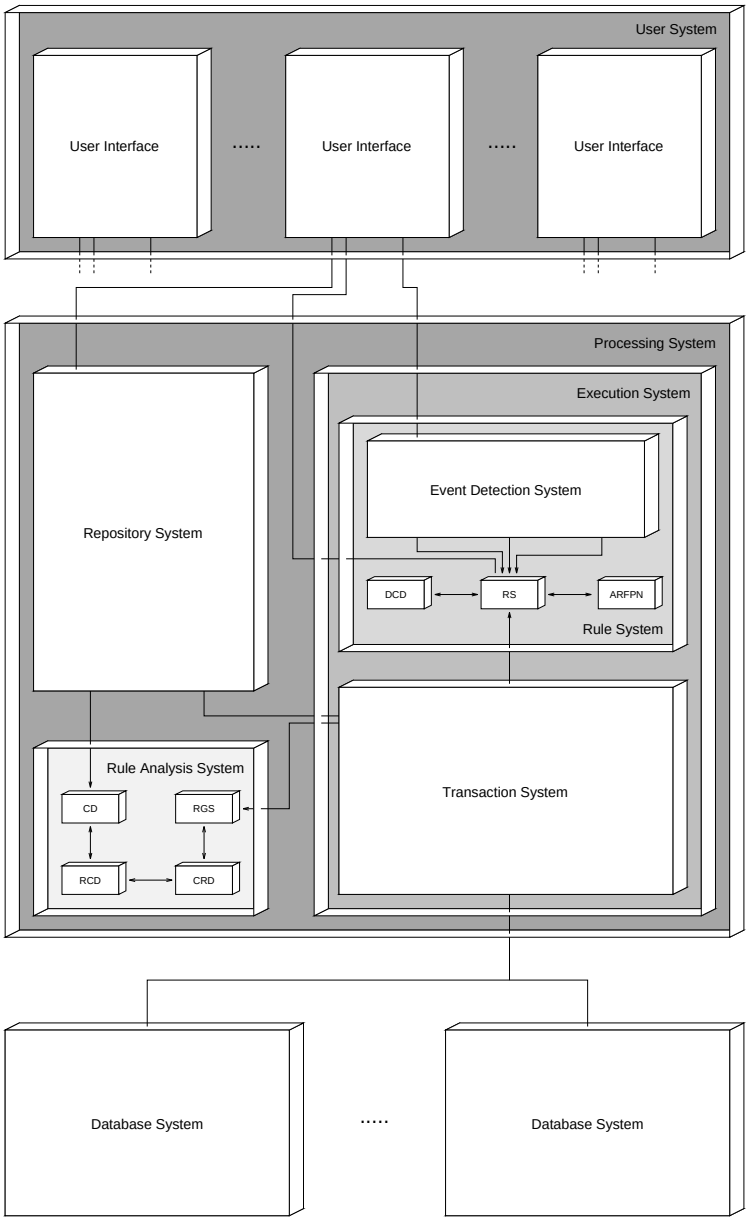

Abbildung 7.2: Regelanalyse und Zyklenerkennung in ALFRED

7.3 Beispiele

Um die Problematik im Zusammenhang mit Zyklen zu veranschaulichen, werden hier drei Regeln definiert. Zur Illustration dient ein Unternehmen, das im Zuge von Sparmassnahmen die Gehälter der Angestellten kürzen muss. Diese Lohnkürzung soll von einem aktiven Datenbanksystem automatisch durchgeführt werden, indem der verantwortliche Sachbearbeiter eine entsprechende Aktion ausführt. Die persönlichen Daten aller Angestellten inklusive deren Löhne seien dazu in einer Relation EMPLOYEE abgelegt. Mit Hilfe von Regeln werden die Löhne aller Angestellten um einen bestimmten Prozentsatz gekürzt. Dazu wird eine Obergrenze vorgegeben, die angibt, wie hoch die Summe der Gehälter aller Angestellten maximal betragen darf. Mit Hilfe von Regeln werden die Löhne solange gekürzt, bis die Obergrenze erreicht oder unterschritten wird.

Um eine Lohnkürzung zu realisieren, muss die Summe aller Gehälter (Obergrenze) angepasst und die Verarbeitung durch Auslösen des abstrakten Ereignisses `verringere_lohn` gestartet werden. Dadurch wird die Regel S ausgelöst, welche die Löhne der Angestellten auf 95 % kürzt.

```
Regel S:   ON abstract event verringere_lohn
           DO update EMPLOYEE
              set gehalt = gehalt*0.95
```

Im folgenden werden zwei Regeln definiert, deren Verarbeitung durch die Regel S ausgelöst wird und zyklisch verläuft. Im ersten Beispiel kann die Terminierung des Zyklus statisch festgestellt werden, im anderen Fall nur dynamisch.

7.3.1 Statisch überprüfbarer Zyklus

Die Regel 1 kürzt die Löhne der Angestellten um jeweils 5 %, bis die vorgegebene Obergrenze unterschritten wird. Dies wird erreicht, indem sich die Regel immer wieder rekursiv auslöst.

```
Regel 1:   ON update EMPLOYEE set
           IF sum(EMPLOYEE.gehalt) > Obergrenze
           DO update EMPLOYEE
              set gehalt = gehalt*0.95
```

Die Terminierung dieses Zyklus kann statisch festgestellt werden, da die Obergrenze fix vorgegeben ist, und da in jedem Durchgang die Gehälter verkleinert werden. Durch diese Verkleinerung wird die Summe aller Gehälter nach endlich vielen Verarbeitungsschritten kleiner sein als die vorgegebene Obergrenze. Die Bedingung von Regel 1 wird somit verletzt und die Verarbeitung terminiert. Falls zu Beginn die Summe der Gehälter bereits kleiner als die Obergrenze ist, so wird die Aktion der Regel nie ausgeführt und der Zyklus terminiert.

7.3.2 Dynamisch überprüfbarer Zyklus

Anstelle von Regel 1 tritt hier Regel 2, mit deren Hilfe die Löhne angepasst werden. Hier wird angenommen, dass die Kürzung spezifisch für jeden Angestellten definiert werden kann. Dazu dient ein Kürzungsansatz (*ansatz*) der für jeden Angestellten angegeben werden muss. Um die effektive Lohnkürzung zu berechnen, wird der gekürzte Lohn mit diesem Kürzungsansatz multipliziert.

```
Regel 2:   ON update EMPLOYEE set
           IF sum(EMPLOYEE.gehalt) > Obergrenze
           DO update EMPLOYEE
              set gehalt = gehalt*ansatz*0.95
```

Bei diesem Zyklus kann die Terminierung nicht statisch festgestellt werden. Der Kürzungsansatz, der als Attributwert in der Relation EMPLOYEE gespeichert ist, muss für die Analyse bekannt sein. Falls dieser Ansatz kleiner als 1 ist, so wird der Zyklus terminieren, andernfalls nicht. In der statischen Analyse kann jedoch der Laufzeitwert von *ansatz* nicht bestimmt werden, da dieses Attribut im Verlaufe der Zeit geändert werden kann. Der Zyklus muss somit dynamisch analysiert werden.

7.4 Einflussfaktoren

Verschiedene Regeleigenschaften haben einen Einfluss auf die Erkennung von Zyklen und somit auf die Überprüfung der Terminierung einer Regelverarbeitung. Durch einige Eigenschaften wird die Analyse der Regelmenge *einfacher* oder auch *komplexer*. Andere Eigenschaften bewirken, dass die Terminierung von Zyklen nicht *statisch* sondern nur *dynamisch* festgestellt werden kann. Unter anderem existieren folgende Einflussfaktoren:

- **Komplexe Ereignisse**
  Zyklenbildende Regeln, die mit komplexen Ereignissen definiert sind, können in der statischen Analyse nur ungenügend überprüft werden. Um die tatsächliche Auslösung einer solchen Regel bestimmen zu können, müssen sämtliche Teilereignisse sowie die Eintrittsreihenfolge dieser Teilereignisse berücksichtigt werden. Diese Informationen sind jedoch nur zur Laufzeit verfügbar und somit sind solche Zyklen nur dynamisch überprüfbar.

- **Komplexe Bedingung**
  In ALFRED ist es möglich, Bedingungen zu spezifizieren, die mittels einer Auswahlstruktur (if-then-else) definiert werden. Durch diese Struktur werden zwei mögliche Auswertungsalternativen erzeugt (If-Teil und Then-Teil). Bei der Analyse müssen beide möglichen Alternativen untersucht werden, womit die Analyse aufwendiger wird.

- **Bedingungsart**
  Wenn im Verlaufe der Verarbeitung die Bedingung einer zyklusbildenden Regel nicht mehr erfüllt ist, so terminiert ein Zyklus. Damit dies festgestellt werden kann, muss die Bedingung untersucht werden. Für jede Bedingungsart (*query, predicate* etc.) gelten dabei unterschiedliche Bewertungskriterien. So muss z.B. untersucht werden, ob die Menge, die durch eine Abfrage spezifiziert wird, im Laufe der Verarbeitung leer wird und somit die Bedingung nicht mehr erfüllt ist.

- **Komplexe Aktion**
  Analog zu komplexe Bedingung.

- **Art der Aktion**
  Durch die Ausführung einer Aktion können potentielle Zyklen terminieren. So z.B. durch die Aktion *disable_rule*, welche eine Regel deaktiviert und somit einen vorhandenen Zyklus beenden kann oder durch die Aktion *abort*, welche die laufende Transaktion abbricht. Die Analyse der Regelmenge muss folglich die Art der Aktion berücksichtigen.

- **EA-Regel**
  Falls in einem Zyklus eine EA-Regel existiert, so kann der Zyklus bei dieser Regel beendet werden, wenn das Ereignis dieser Regel nicht mehr ausgelöst wird. Die Analyse des Zyklus beschränkt sich bei einer solchen Regel auf die Beziehungen, die zwischen Aktions- und Ereignisteilen existieren.

- **ECA-Regel**
  Falls in einem Zyklus eine ECA-Regel existiert, so muss zusätzlich die Bedingungskompo-

nente und die Einflüsse auf diese Komponente untersucht werden. Die Analyse wird dadurch komplexer.

- **ECAA-Regel**
  Bei ECAA-Regeln können zwei mögliche Aktionszweige angegeben werden. Ein Zweig für die wahre Bedingungsauswertung und einer für die falsche. Bei der Analyse der Regelmenge müssen beide Aktionskomponenten berücksichtigt werden.

- **Prioritäten**
  Durch die Definition von Prioritäten wird bei der Verarbeitung eine explizite Reihenfolge vorgegeben. Durch die Berücksichtigung dieser Reihenfolge kann möglicherweise die Terminierung eines Zyklus festgestellt werden.

- **Kopplungsmodi**
  Durch die Angabe von Kopplungsmodi wird der Verarbeitungszeitpunkt von Regelteilen in Bezug zur regelauslösenden Transaktion festgelegt. Durch die Verzögerung zwischen zwei Regelteilen, die mit unterschiedlichen Kopplungsmodi definiert sind (z.B. *immediate/deferred*), können existierende Zyklen terminieren, weil sich zwischen der Ausführung der Regelteile der Datenbankzustand ändern kann. Da die Kopplungsmodi jedoch erst zur Laufzeit berücksichtigt werden können, müssen solche Zyklen in der dynamischen Analyse untersucht werden.

- **Parameterkontexte**
  Parameterkontexte geben vor, welche Parameter bei der Ereigniserkennung gebunden werden müssen. Diese Parameter werden bei der Auswertung von Regelbedingungen verwendet und haben demzufolge einen Einfluss auf den Wahrheitswert einer Bedingung. Die Parameter müssen aus diesem Grund in die Analyse miteinfliessen. Da die Parameterbindung jedoch erst zur Laufzeit realisiert werden kann, ist eine dynamische Analyse unumgänglich.

- **Fristaktion**
  Regeln, die mit Fristen definiert werden, können nur dynamisch untersucht werden, da für die Analyse der Auslösezeitpunkt der Fristaktion wichtig ist.

- **Parallele Ausführung**
  Wenn in einem ADBS die parallele Ausführung von Aktionen möglich ist, so wird die Zyklenerkennung komplexer. Dadurch, dass die Aktionen nicht mehr in einer vorgegebenen Reihenfolge ausgeführt werden, müssen sämtliche möglichen Ausführungsreihenfolgen von Aktionen berücksichtigt werden. Je nach Reihenfolge können neue Zyklen entstehen, oder die Terminierung von existierenden Zyklen kann festgestellt werden. Da die tatsächliche Reihenfolge jedoch erst zur Laufzeit bekannt ist, können solche Zyklen nur dynamisch untersucht werden.

7.5 Statische Analyse

Kommerzielle Datenbanksysteme beinhalten heute keine Möglichkeiten für die Analyse von Zyklen. Oft wird nur eine systemspezifische Kaskadentiefe vorgegeben, bei deren Erreichung die Verarbeitung abgebrochen wird (z.B. in Oracle, Ingres oder Sybase; vgl. [Sch95]).

Bei ADBS ist die Erkennung und Analyse von Zyklen erst in wenigen Prototypen integriert (vgl. [VGD96, BW96, AWH92, BCP95b, Rys95]). In diesen Prototypen geschieht die Analyse mittels eines sogenannten Auslösegraphen (*triggering graph*). Ein Auslösegraph besteht aus Knoten und Kanten. Knoten repräsentieren Regeln, während Kanten die Beziehung zwischen Regeln darstellen. Zwei Knoten werden mit einer Kante verbunden, wenn durch die Ausführung einer Regelaktion das Ereignis einer anderen Regel eintritt, d.h. wenn durch die Verarbeitung der einen Regel die andere ausgelöst wird. Die Erkennung von Zyklen wird sodann mit Hilfe eines Auslösegraphen durchgeführt. Falls der Graph keine Zyklen enthält, so terminiert die Regelverarbeitung. Falls im

Graph Zyklen existieren, so müssen diese Zyklen genauer untersucht werden, indem Beziehungen zwischen den Regeln im Zyklus analysiert werden. Um diese Untersuchung durchzuführen, sind verschiedene Informationen notwendig, wie z.B. die Art der Bedingungen und die Art der Aktionen. Diese Informationen sind jedoch in einem Auslösegraph nicht verfügbar.

In ALFRED wird die Erkennung von Zyklen direkt auf dem ARFPN durchgeführt. Ein ARFPN ist ebenfalls ein Graph, der auf Zyklen hin untersucht werden kann. Ein Vorteil liegt jedoch darin, dass im ARFPN alle Informationen enthalten sind, die für die Analyse von Zyklen notwendig sind, wie z.B. die Art der Bedingung, die Komplexität der Bedingung, Kopplungsmodi und Parameter.

Ein Zyklus kann in der Regelmenge erst entstehen, wenn Abhängigkeiten zwischen Regelaktionen und Regelereignissen existieren. Nur wenn durch die Ausführung einer Aktion Ereignisse eintreten, kann ein Zyklus entstehen. Um diesen beenden zu können bestehen folgende zwei Möglichkeiten:

- Es gibt eine Regelaktion, die eine Regel im Zyklus deaktiviert.
- Es gibt eine Regelaktion, deren Ausführung Werte verändert, die in der Bedingungsauswertung einer zyklusbeteiligten Regel verwendet werden. Falls durch diese Änderung die Bedingungsauswertung FALSE ergibt, so wird der Zyklus terminieren.

7.5.1 Untersuchungsgegenstand

In Rahmen dieser Arbeit werden nicht alle Regeleigenschaften für die Erkennung und Analyse von Zyklen berücksichtigt, da eine solche Betrachtung zu komplex ist. Damit auf einer wohldefinierten Grundlage aufgebaut werden kann, muss als erstes festgelegt werden, welche Regeleigenschaften und welche Einschränkungen betrachtet werden.

7.5.1.1 Betrachtete Regeleigenschaften

In der Analyse werden sowohl EA- als auch ECA-Regeln unterstützt (vgl. Tabelle 7.1). Diese Regeln können aus primitiven Ereignissen, primitiven Bedingungen und primitiven sowie komplexen Aktionen bestehen. Von den möglichen primitiven Ereignissen werden nur *retrieve, insert, update, delete* und *abstract event* unterstützt. Bei den primitiven Bedingungen werden *true, false, predicate* und *query* betrachtet, jedoch muss bei *query* die *where-clause* ein (primitives) Prädikat sein. Von den möglichen primitiven Aktionen werden *retrieve, update, delete, abort, enable_rule, disable_rule, message* und *raise* unterstützt. Bei *retrieve* und *delete* wird ebenfalls die *where-clause* betrachtet, die jedoch auch nur ein Prädikat sein darf. Zusätzlich werden auch Sequenzen von Aktionen und Aktionen innerhalb von Transaktionen unterstützt. Ferner werden Regeln mit Auslösungszeitpunkt *post* und Auslösungsgranularität *set* betrachtet. Von den möglichen Kopplungsmodi wird in der Analyse nur der Kopplungsmodus *immediate* unterstützt (sowohl EC-, EA- als auch CA-Kopplung). Regeleigenschaften, die nicht in der Tabelle stehen, werden im Rahmen dieser Arbeit nicht berücksichtigt (z.B. komplexe Ereignisse, Parameterkontexte).

7.5.1.2 Zyklenbezogene Regelbeziehungen

Wenn in der Regelmenge ein Zyklus existiert, so gibt es möglicherweise Regeln, die diesen Zyklus beeinflussen. Es gilt zwischen drei möglichen Beeinflussungen zu unterscheiden, die nachfolgend als *Regelbeziehungen* bezeichnet werden:

- Durch die Ausführung einer Regelaktion tritt ein Ereignis (oder mehrere Ereignisse) ein.
- Die Aktion einer Regel verändert Datenwerte, die in Regelbedingungen verwendet werden.

In der Analyse betrachtete Regeleigenschaften	
ECA-Struktur	ja
EA-Struktur	ja
Primitive Ereignisse	retrieve, insert, update, delete, abstract event
Primitive Bedingungen	true, false, predicate, query
Primitive Aktionen	retrieve, insert, update, delete, abort, enable_rule, disable_rule, message, raise
Komplexe Aktionen	Sequenz von Aktionen, Aktionen innerhalb von Transaktionen
Auslösungszeitpunkt	post
Auslösungsgranularität	set
Kopplungsmodus	immediate

Tabelle 7.1: Regeln in der Terminierungsanalyse

- Die Aktion einer Regel aktiviert oder deaktiviert eine Regel.

Regeln, die zyklusbeteiligte Regeln beeinflussen, können sich innerhalb oder ausserhalb des Zyklus befinden. Aus diesem Grund müssen vier Arten von zyklenbezogenen Regelbeziehungen betrachtet werden (vgl. Abbildung 7.3).

Angenommen eine Regelmenge bestehe aus fünf Regeln $\{R_1, R_2, R_3, R_4, R_5\}$. In dieser Regelmenge bestehe ein Zyklus, der sich aus den ersten drei Regeln bildet ($Z := \{R_1 \Rightarrow R_2 \Rightarrow R_3 \Rightarrow R_1\}$). Unter dieser Annahme gilt folgendes:

- *Interne Beziehung*
  Eine Beziehung wird als *intern* bezeichnet, wenn sie zwischen den Regeln besteht, die einen Zyklus bilden (z.B. zwischen R_3 und R_2 in Abbildung 7.3).

- *Externe Beziehung*
  Eine Beziehung wird als *extern* bezeichnet, wenn sie zwischen Regeln besteht, die ausserhalb eines Zyklus liegen (zwischen R_5 und R_4 in Abbildung 7.3).

- *Int_ext Beziehung*
  Eine *int_ext* Beziehung besteht, wenn eine Regel innerhalb eines Zyklus eine Beziehung zu einer Regel ausserhalb eines Zyklus hat (R_1 und R_4 in Abbildung 7.3).

- *Ext_int Beziehung*
  Eine *ext_int* Beziehung besteht, wenn eine Regel ausserhalb eines Zyklus eine zyklusbeteiligte Regel beeinflusst (R_5 und R_2 in Abbildung 7.3).

Die statische Analyse in ALFRED beschränkt sich nur auf *interne* Beziehungen, d.h. Beziehungen die innerhalb eines Zyklus auftreten können. Alle anderen Beziehungsarten werden hier nicht betrachtet.

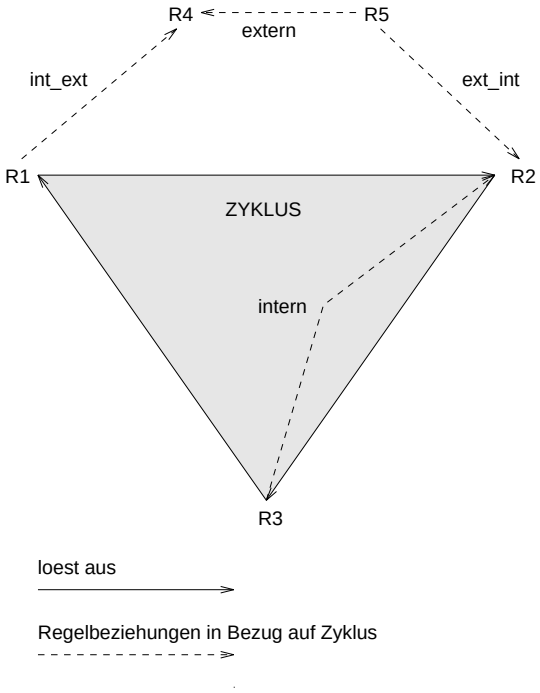

Abbildung 7.3: Zyklenbezogene Regelbeziehungen

7.5.1.3 "Net Effect"

Innerhalb eines Zyklus können verschiedene Arten von internen Beziehungen auftreten. Um mit Hilfe dieser Beziehungen statisch bestimmen zu können, ob ein Zyklus terminiert, muss unter anderem die Änderung bestimmt werden, die sich durch die Summe von Änderungen ergibt. Diese kumulierte Änderung wird als *net effect* (vgl. [Han92a]) bezeichnet.

Angenommen ein Zyklus besteht aus drei Regeln (Regel A, Regel B und Regel C) und es gibt eine Beziehung zwischen Regel A und Regel C sowie zwischen Regel B und Regel C. Diese Beziehungen seien wie folgt:

1. Die Aktion von Regel A *deaktiviert* die Regel C (*disable_rule*).
2. Die Aktion von Regel B *aktiviert* die Regel C (*enable_rule*).

Wenn nur die erste Beziehung betrachtet wird, so stellt die Analyse fest, dass der Zyklus terminiert (da nach der Verarbeitung von Regel A die Regel C inaktiv ist). Diese Feststellung ist jedoch falsch, weil durch die Verarbeitung von Regel B die Regel C wiederum aktiviert wird. Der *net effect* dieser beiden Aktionen ist ein *enable_rule*. Um also den *net effect* bestimmen zu können, müssen die Reihenfolgen der Aktionen sowie die Art der Aktionen genauer untersucht werden. Zusätzlich muss festgestellt werden, auf was sich die einzelnen Aktionen beziehen.

Der *net effect* von *enable* und *disable* kann nur ermittelt werden, wenn sich beide Aktionen auf dieselbe Regel beziehen. Bei DML-Aktionen gibt es eine subtilere Unterscheidung. So kann der *net effect* auf Relationsbasis, d.h. alle DML-Aktionen beziehen sich auf dieselbe Relation, oder auf Datensatzbasis, d.h. die DML-Aktionen beziehen sich auf dieselben Datensätze, bestimmt werden. Ein *net effect* auf Datensatzebene lässt sich wie folgt ermitteln:

- *enable_rule und disable_rule*
  Die letzte Aktion ergibt den *net effect*.

- *insert und delete*
  Wenn ein *delete* auf ein *insert* folgt, so gibt es keine Auswirkung, da die Datensätze nicht mehr existieren.
- *insert und update*
  Wenn ein *update* auf ein *insert* folgt, so ist der *net effect* das *update*.
- *update und update*
  In diesem Fall ist das letzte *update* der *net effect*.
- *update und delete*
  Analog zu *insert und delete*.

7.5.2 Regelabhängigkeiten

Die Analyse eines Zyklus muss die Abhängigkeiten innerhalb der zyklusbeteiligten Regeln untersuchen. Dabei gilt es zwischen Abhängigkeiten innerhalb einer Regel (Regelintradependenzen) und Abhängigkeiten zwischen Regeln (Regelinterdependenzen) zu unterscheiden.

7.5.2.1 Regelintradependenzen

Die Struktur einer Regel gibt die Abhängigkeiten vor, die innerhalb der Regel existieren (vgl. Abbildung 7.4).

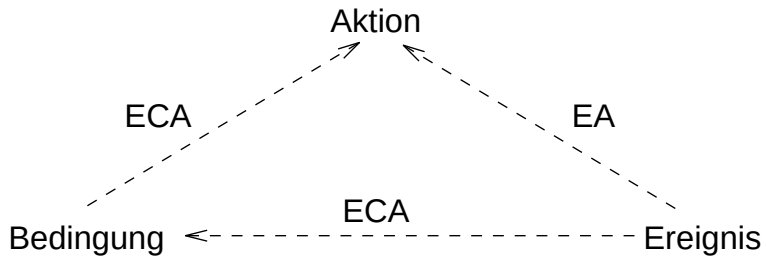

Abbildung 7.4: Regelintradependenzen

- **ECA-Regeln**
  - **Ereignis-Bedingung:** Es besteht eine Abhängigkeit zwischen dem Ereignis- und dem Bedingungsteil der Regel. Die Bedingung wird nur dann ausgewertet, wenn das Ereignis eingetreten ist.
  - **Bedingung-Aktion:** Es besteht ferner eine Abhängigkeit zwischen dem Bedingungs- und dem Aktionsteil der Regel. Die Aktion wird nur dann ausgeführt, wenn die Bedingung erfüllt ist.
- **EA-Regeln**
  - **Ereignis-Aktion:** Es besteht eine Abhängigkeit zwischen dem Ereignis- und dem Aktionsteil einer Regel. Die Aktion wird nur dann ausgeführt, wenn das Ereignis eingetreten ist.

7.5.2.2 Regelinterdependenzen

Durch die Ausführung von Regelaktionen können sich Regeln gegenseitig beeinflussen. Es gilt folgende Abhängigkeiten zwischen Regeln zu unterscheiden, die in der Analyse näher untersucht werden müssen (vgl. Abbildung 7.5):

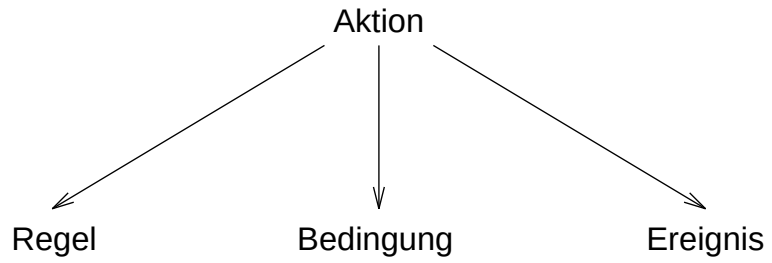

Abbildung 7.5: Regelinterdependenzen

- **Aktion-Regel:** Es gibt zwei mögliche Aktionen, die einen Einfluss auf eine ganze Regel haben. Mit *enable_rule* wird eine Regel aktiviert, mit *disable_rule* wird sie deaktiviert.
- **Aktion-Bedingung:** Die Aktionskomponente kann Daten modifizieren, die für die Auswertung einer Regelbedingung verwendet werden. Somit kann die Ausführung der Aktion indirekt das Ergebnis einer Bedingungsauswertung beeinflussen.
- **Aktion-Ereignis:** Durch die Ausführung einer Aktion können Ereignisse eintreten und somit Regeln ausgelöst werden.

7.5.3 Zyklenerkennung

Damit ein Zyklus in der Regelmenge existieren kann, muss mindestens eine Aktion-Ereignis-Beziehung bestehen. Die Tabelle 7.2 zeigt, welche Beziehung zwischen Aktionen und Ereignissen existieren muss, damit durch die Ausführung einer Aktion ein Ereignis eintreten kann. Die Ereignisse und Aktionen sind in der Tabelle aus Gründen der Übersichtlichkeit vereinfacht dargestellt.

Diese Aktion-Ereignis-Beziehung ist im ARFPN von ALFRED bereits integriert. Bei der Generierung einer Regel wird das Regel-ARFPN durch Verbindungstransitionen und durch Kanten von diesen Transitionen zu den entsprechenden Primärplätzen ergänzt. Falls ein Zyklus in der Regelmenge existiert, so besteht auch ein Zyklus im Petrinetz der Regelmenge.

Die Erkennung eines solchen Zyklus kann einfach erfolgen. Das Petrinetz wird mit einem Suchalgorithmus (z.B. Tiefensuche) durchschritten. Jeder Platz, der bei dieser Suche gefunden wird, muss markiert werden. Falls ein Platz mehr als einmal erreicht wird, so exisitiert ein Zyklus in der Regelmenge.

Diese Analyse muss jedesmal dann durchgeführt werden, wenn eine neue Regel durch das *Rule Generation System* erzeugt wird. Die Zyklensuche wird bei dieser Regel gestartet, da ein neuer Zyklus nur dann entstehen kann, wenn die Regel am Zyklus beteiligt ist. Die Erkennung von Zyklen im ARFPN ist in einem Pseudocode-Algorithmus im Anhang C zu finden.

7.5.4 Zyklenanalyse

Wenn in der Regelmenge ein Zyklus existiert, so muss dieser genauer untersucht werden. Dabei muss festgestellt werden, ob dieser Zyklus zur Laufzeit zu einer zyklischen Verarbeitung führt

Aktionen	**Ereignisse**				
	retrieve $< id_e >$	insert $< id_e >$	update $< id_e >$	delete $< id_e >$	abstract event $< id_e >$
retrieve from $< id_a >$	$< id_a >$ = $< id_e >$				
insert in $< id_a >$		$< id_a >$ = $< id_e >$			
update on $< id_a >$			$< id_a >$ = $< id_e >$		
delete from $< id_a >$				$< id_a >$ = $< id_e >$	
raise abstract event $< id_a >$					$< id_a >$ = $< id_e >$

Tabelle 7.2: Aktion-Ereignis-Beziehungen

und ob diese Verarbeitung terminiert. Um dies feststellen zu können, müssen die beiden Einflüsse untersucht werden, welche zur Terminierung führen können. Dies sind Aktion-Regel-Beziehungen und Aktion-Bedingung-Beziehungen.

7.5.4.1 Beziehung Aktion-Regel

Bei der statischen Analyse eines Zyklus wird davon ausgegangen, dass sämtliche Regeln aktiviert sind. Mit diesem Ansatz wird garantiert, dass jeder mögliche Zyklus erkannt werden kann, da eine Obermenge von Regeln betrachtet wird. In dieser Menge von Regeln muss nun der *net effect* von Aktion-Regel-Beziehungen für jede am Zyklus beteiligte Regel berechnet werden. Wenn für irgendeine zyklusbeteiligte Regel der Effekt *disable_rule* besteht, so kann die Terminierung statisch garantiert werden. Andernfalls muss der Zyklus zur Laufzeit näher untersucht werden. Dieser Sachverhalt wird in Tabelle 7.3 aufgezeigt. Die Aktionen innerhalb der Tabelle stehen dabei für den jeweiligen *net effect*.

Aktionen	**Regel** Rule $< id >$
enable rule $< id >$	dynamisch
disable rule $< id >$	**statisch**

Tabelle 7.3: Terminierung bei Aktion-Regel-Beziehungen

7.5.4.2 Beziehung Aktion-Bedingung

Bei der Analyse der Aktion-Bedingung-Beziehungen muss als erstes der *net effect* bestimmt werden. Aufgrund dieses Effekts kann in einigen Fällen statisch bestimmt werden, ob ein Zyklus terminiert.

Die einzigen Aktionen, die einen Einfluss auf die Bedingungsauswertung haben können, sind die Datenmanipulationsaktionen *insert, update* und *delete*. Diese Aktionen müssen im Zyklus ermittelt werden. Es kann nur eine Aktion-Bedingung-Beziehung entstehen, wenn die Aktion (resp. der *net effect*) diejenigen Datensätze ändert, die zur Bedingungsauswertung verwendet werden.

Um einen Zyklus diesbezüglich zu analysieren, müssen die Bedingungsarten (*predicate, query*) und auszuführenden Aktionen bekannt sein. Zwei der drei DML-Aktionen, die einen möglichen Einfluss auf die Bedingungsauswertung haben, können optional durch eine *where-clause* ergänzt werden. Folglich gilt es vier verschiedene Kombinationsmöglichkeiten von Aktion-Bedingung-Beziehungen zu unterscheiden:

1. Datenmanipulationsaktionen *ohne where-clause* kombiniert mit Prädikatbedingungen.
2. Datenmanipulationsaktionen *mit where-clause* kombiniert mit Prädikatbedingungen.
3. Datenmanipulationsaktionen *ohne where-clause* kombiniert mit Abfragen (*queries*).
4. Datenmanipulationsaktionen *mit where-clause* kombiniert mit Abfragen (*queries*).

In der Analyse eines Zyklus muss die Änderung, die der *net effect* von update-Operationen bewirkt, klassifiziert werden. Dabei kann zwischen vier Auswirkungen unterschieden werden:

1. *Increasing*
   Die Auswirkung eines update-Befehls ist *increasing*, wenn der Wert eines Attributs erhöht wird (z.B. `set gehalt = gehalt + 10`).
2. *Decreasing*
   Die Auswirkung eines update-Befehls ist *decreasing*, wenn der Wert eines Attributs verringert wird.
3. *Equal*
   Die Auswirkung eines update-Befehls ist *equal*, wenn der Wert eines Attributs nicht verändert wird.
4. *Unknown*
   Die Auswirkung eines update-Befehls ist *unknown*, wenn die Änderung statisch nicht oder nur aufgrund einer sehr aufwendigen Untersuchung festgestellt werden kann.

DML-Aktionen ohne where-clause kombiniert mit Prädikaten
In den Tabellen 7.4, 7.5 und 7.6 sind diejenigen Fälle zu sehen, die eine statische Erkennung der Terminierung zulassen, wenn die Bedingung ein Prädikat ist. Für diese Tabellen gelten folgende Konventionen:

Aktionen		**Prädikate** **att** $\sim$ *Att*			
		$<$	$=$	$>$	$\neq$
insert in $< id >$		dynamisch	dynamisch	dynamisch	dynamisch
update on $< id >$ set $< atts >$	increasing	**statisch**	dynamisch	dynamisch	dynamisch
	decreasing	dynamisch	dynamisch	**statisch**	dynamisch
	equal	dynamisch	dynamisch	dynamisch	dynamisch
	unknown	dynamisch	dynamisch	dynamisch	dynamisch
delete from $< id >$		dynamisch	dynamisch	dynamisch	dynamisch

Tabelle 7.4: Terminierung bei Aktion-Bedingung-Beziehungen (1)

Aktionen		**Prädikate** **att** $\sim$ **att**			
		$<$	$=$	$>$	$\neq$
insert in $< id >$		dynamisch	dynamisch	dynamisch	dynamisch
update on $< id >$ set $< atts >$	increasing	dynamisch	dynamisch	dynamisch	dynamisch
	decreasing	dynamisch	dynamisch	dynamisch	dynamisch
	equal	dynamisch	dynamisch	dynamisch	dynamisch
	unknown	dynamisch	dynamisch	dynamisch	dynamisch
delete from $< id >$		dynamisch	dynamisch	dynamisch	dynamisch

Tabelle 7.5: Terminierung bei Aktion-Bedingung-Beziehungen (2)

Aktionen		**Prädikate** **att** $\sim$ *Konst*			
		$<$	$=$	$>$	$\neq$
insert in $< id >$		dynamisch	dynamisch	dynamisch	dynamisch
update on $< id >$ set $< atts >$	increasing	**statisch**	dynamisch	dynamisch	dynamisch
	decreasing	dynamisch	dynamisch	**statisch**	dynamisch
	equal	dynamisch	dynamisch	dynamisch	dynamisch
	unknown	dynamisch	dynamisch	dynamisch	dynamisch
delete from $< id >$		dynamisch	dynamisch	dynamisch	dynamisch

Tabelle 7.6: Terminierung bei Aktion-Bedingung-Beziehungen (3)

- Die Aktionen in den Tabellen stehen stellvertretend für den *net effect* und sind in vereinfachter Syntax angegeben.
- Der Operator $\sim$ ist Platzhalter für einen der Vergleichsoperatoren $<, =, >$ und $\neq$.
- Wenn das Attribut *att* in einer Bedingung verwendet wird (in den Tabellen bedeutet dies: $att \in < atts >$) und ein update-Befehl dieses Attribut verändert (increasing, decreasing, equal oder unknown), so wird **att** in Fettdruck geschrieben.
- Mit *Att* wird ein beliebiges Attribut bezeichnet, das durch die Aktion *nicht* verändert wird.
- Mit *Konst* wird eine Konstante bezeichnet.

In Tabelle 7.4 sind diejenigen primitiven Prädikate zu sehen, die sich aus zwei Attributen zusammensetzen. Die DML-Aktion verändert nur einen der beiden Operanden. Aus der Tabelle ist ersichtlich, dass die Terminierung nur in genau zwei Fällen statisch gezeigt werden kann:

1. Wenn die Aktion *increasing* in Bezug auf den ersten Operanden der Bedingung ist, und wenn der Operator in der Prädikatbedingung ein '<' ist. Nach einer endlichen Anzahl Schritten wird **att** grösser sein als der Vergleichsoperand *Att* und die Bedingung ist somit verletzt.
2. Wenn die Aktion *decreasing* in Bezug auf den ersten Operanden der Bedingung ist, und wenn der Operator in der Prädikatbedingung ein '>' ist. In diesem Fall wird **att** in jedem Zyklusdurchlauf erhöht. Nach einer bestimmten Anzahl von Schritten wird **att** grösser sein als *Att*. Die Bedingung ist somit verletzt, und der Zyklus terminiert.

Analoge Aussagen gelten für den Fall ($Att \sim$ **att**). Dies kann als (**att** $\sim Att$) behandelt werden, indem der Operator in der Prädikatbedingung durch sein Pendant ('<' durch '>' und '=' durch '$\neq$') ersetzt wird.

Wenn die DML-Aktion ein *insert* oder ein *delete* ist, so kann die Terminierung nicht statisch bestimmt werden. Die Werte, die durch ein *insert* spezifiziert werden, sind zur Zeit der Regeldefinition nicht bekannt, da sie sich normalerweise erst durch die Bindung von Parametern ergeben. Analoges gilt für das *delete*.

Die Tabelle 7.5 zeigt jene Beziehungen, in denen sich die Prädikatbedingung aus zwei Attributen zusammensetzt, die *beide* durch eine Aktion verändert werden. In diesem Fall kann statisch nicht entschieden werden, ob der Zyklus terminiert, da beide Operanden variabel sind. Eine dynamische Zyklenanalyse ist somit unumgänglich.

In Tabelle 7.6 sind diejenigen Aktion-Bedingung-Beziehungen aufgeführt, in denen die Bedingung ein Prädikat ist, welches sich aus einem Attribut und einer Konstante zusammensetzt. Das Attribut wird durch die DML-Aktion verändert. Dieser Fall ist identisch mit den Angaben in Tabelle 7.4, da immer nur ein Operand geändert wird.

Falls in der Prädikatbedingung zwei Konstanten auftreten oder falls die Bedingung *true* oder *false* ist, so kann das Resultat der Bedingung statisch ausgewertet werden.

DML-Aktionen mit where-clause kombiniert mit Prädikaten

Wenn die Aktionen *update* und *delete* eine *where-clause* enthalten, so können statisch nur sehr beschränkte Terminierungsaussagen formuliert werden. Nur wenn die in der *where-clause* angegebenen Attribute mit den Attributen in der Prädikatbedingung übereinstimmen, kann statisch festgestellt werden, ob der Zyklus terminiert. Diese Untersuchung kann jedoch sehr aufwendig sein. Falls die Attribute nicht übereinstimmen, so muss die Terminierung dynamisch untersucht werden. In Tabelle 7.7 ist dieser Sachverhalt dargestellt. Mit *Value* wird ein Attribut oder eine Konstante bezeichnet.

Aktionen	**Prädikate** $Value \sim Value$			
	$<$	$=$	$>$	$\neq$
update ... where $< search\_cond >$	dynamisch	dynamisch	dynamisch	dynamisch
delete ... where $< search\_cond >$	dynamisch	dynamisch	dynamisch	dynamisch

Tabelle 7.7: Terminierung bei Aktion-Bedingung-Beziehungen mit where-clause

DML-Aktionen ohne/mit where-clause kombiniert mit Abfragen
Wenn die Bedingung in der Aktion-Bedingung-Beziehung eine *query* ist, so kann die Terminierung statisch nur in einem einzigen Fall bestimmt werden. Falls der *net effect* ein *delete* ist, und wenn die *where-clause* der Aktion identisch mit der Bedingung ist, so kann die Terminierung garantiert werden. Mit diesem *delete* wird die Menge, die durch die *query* spezifiziert wird, nach endlich vielen Schritten leer. Dadurch wird auch die Bedingung nach endlich vielen Schritten verletzt und der Zyklus terminiert. Falls die *where-clause* nicht mit der Abfrage übereinstimmt, so kann die Terminierung statisch nicht festgestellt werden (vgl. Tabelle 7.8).

Aktionen		**Abfragen**
insert in $< id >$		dynamisch
update on $< id >$ set $< atts >$	increasing	dynamisch
	decreasing	dynamisch
	equal	dynamisch
	unknown	dynamisch
delete from $< id >$ where $< search\_cond >$		**statisch** wenn $< search\_cond > \equiv Abfrage$ sonst dynamisch

Tabelle 7.8: Terminierung bei Aktion-Bedingung-Beziehungen mit Abfragen

7.5.5 Beispiel

Die in den vorangehenden Abschnitten aufgeführten Kriterien können nun verwendet werden, um die beiden Beispielregeln aus den Abschnitten 7.3.1 und 7.3.2 genauer zu analysieren.

7.5.5.1 Beispiel 1

Mit Hilfe von Regel 1 wird eine Lohnkürzung realisiert. Die Regel ruft sich selbst rekursiv auf und verringert die Löhne in jedem Durchgang um einen gewissen Prozentsatz, bis eine fest vorgegebene Obergrenze unterschritten wird.

Durch die Ausführung der Aktion von Regel 1 tritt das Ereignis der Regel ein. Es existiert also eine Aktion-Ereignis-Beziehung zwischen Regel 1 und Regel 1. Nach der Regelauslösung wird die Bedingung ausgewertet. Diese Bedingung ist ein Prädikat, welches durch die Regelaktion beeinflusst

wird. Somit besteht auch eine Aktion-Bedingung-Beziehung zwischen Regel 1 und Regel 1. Der *net effect* der Regelaktion ist die Regelaktion selbst, da keine weiteren Regeln berücksichtigt werden müssen. Die Regelaktion (*update*) ist *decreasing*, da das Attribut *gehalt* jedesmal verringert wird. Dies ist eine der in Tabelle 7.4 gezeigten Situationen, in welcher die Terminierung statisch gezeigt werden kann. Die Aktion der Regel verringert in jedem Durchgang die Löhne der Angestellten und somit auch die gesamte Lohnsumme. Diese wird nach endlich vielen Schritten kleiner sein als die vorgegebene Obergrenze. Der Zyklus terminiert somit.

7.5.5.2 Beispiel 2

Durch die Verarbeitung von Regel 2 wird ebenfalls eine Lohnkürzung realisiert. Die Regel ruft sich selbst rekursiv auf und verringert die Löhne in jedem Durchgang um einen vorgegebenen Prozentsatz. Dieser gekürzte Lohn wird sodann mit einem Kürzungsansatz multipliziert, der spezifisch für jeden Angestellten definiert werden kann, und der als Attribut in der Relation EMPLOYEE abgespeichert ist.

Dieser Zyklus kann statisch nicht überprüft werden, da die Änderung, welche durch die Aktion von Regel 2 definiert wird, nicht klassifiziert werden kann, d.h sie ist *unknown*. Der Gehalt jedes Angestellten wird um 95 % verringert und zusätzlich mit einem Kürzungsansatz (Attribut *ansatz*) multipliziert. Damit in der Analyse jedoch festgestellt werden kann, welche Änderung die Aktion bewirkt (increasing, decreasing, equal), muss der Wert des Attributes *ansatz* bekannt sein, und dies ist erst zur Laufzeit möglich. Der Zyklus muss demzufolge dynamisch untersucht werden (vgl. auch Tabelle 7.4).

7.6 Dynamische Analyse

In der statischen Analyse kann die Terminierung nur für einen sehr kleinen Teil von Zyklen gezeigt werden. Je mehr Regeleigenschaften berücksichtigt werden, um so komplexer wird die statische Analyse. Auch wenn man nur eine Untermenge von Regeleigenschaften betrachtet, wird deutlich, dass eine statische Terminierungsanalyse unbedingt durch eine dynamische ergänzt werden muss. Gerade wenn sämtliche in ALFRED definierbaren Regeleigenschaften berücksichtigt werden sollen, ist eine dynamische Analyse unumgänglich.

Die dynamische Zyklenanalyse muss die in der statischen Analyse gefundenen Zyklen überwachen und während der Laufzeit genauer untersuchen. Dazu muss der Zyklus mindestens einmal verarbeitet werden, damit u.a. die Parameterwerte und die aktuellen Werte in der Datenbank für die Analyse berücksichtigt werden können. Anhand dieser Informationen lassen sich die Regelabhängigkeiten genauer untersuchen. Der Aufwand für eine solche dynamische Analyse ist jedoch keinesfalls zu unterschätzen. Wenn sämtliche Regeleigenschaften berücksichtigt werden sowie alle Einflüsse von innerhalb und ausserhalb eines Zyklus, so steigt der Analyseaufwand sehr stark an.

Wird eine vollständige Analyse von Zyklen angestrebt, so müsste in einem ADBS eine Expertensystemkomponente existieren, die diese Aufgabe wahrnimmt und dem Benutzer Lösungsvorschläge präsentiert. Durch die Umsetzung dieser Lösungsvorschläge könnte die Terminierung der Regelmenge garantiert werden.

Die Behandlung einer dynamischen Zyklenanalyse liegt ausserhalb des Rahmens dieser Arbeit. Ein möglicher Ansatz ist z.B. in [BCP95b] zu finden. Dort wird zur Laufzeit ein sogenannter *cycle monitor* eingesetzt, der potentiell endlose Zyklen, die zur Definitionszeit gefunden wurden, überwacht und wenn nötig beendet.

Kapitel 8

Simulation

Um in einem aktiven Datenbanksystem aktives Verhalten realisieren zu können, muss als erstes eine Menge von Regeln definiert werden, mit deren Hilfe dieses Verhalten modelliert wird. Bereits in einfachen Anwendungsbereichen kann diese Regelmenge sehr gross und für den Benutzer schwer verständlich werden. Aus diesem Grund muss es ein ADBS dem Benutzer ermöglichen, das definierte aktive Verhalten mit dem realisierten aktiven Verhalten zu vergleichen. In ALFRED wird zu diesem Zweck eine Verarbeitung der Regeln resp. eine Simulation der Regelverarbeitung auf der Basis von Action Rule Flow Petri Net (ARFPN) unterstützt. Mit Hilfe dieser Verarbeitung oder Simulation ist es möglich, das Verhalten des ADBS zu beobachten.

Die Verarbeitung der Regeln geschieht dabei vollständig in der aktiven Schicht. Damit nicht eine Obermenge von Regeln, sondern die tatsächlich ausgelösten Regeln berücksichtigt werden können, sind Informationen zur Laufzeit unumgänglich. Aus diesem Grund geschieht die Simulation von Abläufen in ALFRED schrittweise. Die Regelsimulation verarbeitet dabei ein ARFPN und erhält in jedem Schritt die notwendigen Informationen vom Transaktionssystem von ALFRED (z.B. ob eine Aktion ausgeführt werden konnte).

Die Simulation in ALFRED bietet dem Benutzer unter anderem folgende Möglichkeiten:

1. Bei gegebenem Anfangszustand der Datenbank kann der Benutzer überprüfen, ob die erzeugte Regelmenge so reagiert, wie dies gewünscht ist.
2. Der Benutzer kann unerwünschte Seiteneffekte (frühzeitig) erkennen und beheben.
3. Der Benutzer kann die erzeugte Regelmenge besser verstehen. Insbesondere kann er untersuchen, welche Auswirkungen Regelkaskaden auf das realisierte aktive Verhalten haben.
4. Der Benutzer kann erkennen, ob die in der Regelmenge möglicherweise vorkommenden Zyklen terminieren, falls dies nicht bereits durch die Regelanalyse gezeigt werden konnte.

Zwei weitere Eigenschaften der Simulation in ALFRED sind:

- *Testlauf:*
  Alle Daten, die während der Verarbeitung anfallen, werden nur temporär gehalten und nach Beenden der Simulation gelöscht.
- *Auditing:*
  Die Simulation erstellt eine Auditing Datei, in welcher Informationen über eingefügte Daten, gelöschte Daten etc. abgelegt werden. Diese Datei wird am Ende der Simulation so aufbereitet, dass der Benutzer schnell und einfach erkennen kann, welche Auswirkungen die erfolgten Benutzerbefehle zeigten.

In diesem Kapitel werden die Systeme von ALFRED beschrieben, die für die Verarbeitung von Regeln zuständig sind. Dabei wird zuerst der Ablauf bei der Regelverarbeitung erklärt, anschliessend werden die einzelnen Systeme allgemein und anhand eines Beispiels erläutert.

8.1 Überblick

Die Verarbeitung von Regeln in ALFRED wird mit Hilfe von Subsystemen realisiert, die bestimmte Teilaufgaben erfüllen. Der folgende Abschnitt erläutert die Subsysteme, die für die Verarbeitung zuständig sind, und gibt einen ersten Überblick über deren Funktionalität:

8.1.1 Komponenten

Folgende Komponenten sind in ALFRED für die Modellierung, Analyse und Verarbeitung von Regeln zuständig (vgl. Abbildung 8.1):

- **User System**
  Dieses System stellt die Schnittstelle zum Benutzer dar. Es setzt sich aus zwei Untersystemen zusammen:
  - **Menu System**
    Das Menüsystem ist die eigentliche Verbindung zwischen ALFRED und dem jeweiligen Benutzer. Es stellt Funktionalitäten für die Eingabe und Änderung sowie Verwaltung von Daten, Datenbanken und Regeln zur Verfügung. Dieses System ist in folgende Untermenüs gegliedert:
    * *Datenbankmenü*
      In diesem Menü können Datenbanken kreiert, gelöscht, und die Verbindung zu ihnen hergestellt werden.
    * *Datendefinitionsmenü*
      In diesem Menü können Regeln, Objekttypen und Benutzer kreiert, abgeändert und gelöscht werden.
    * *Datenmanipulationsmenü*
      In diesem Menü können Daten eingegeben, geändert, gelöscht und selektiert werden.
    * *Simulationsmenü*
      In diesem Menü können Simulationen der Regelverarbeitung definiert und ausgeführt werden.
    * *Transaktionsmenü*
      In diesem Menü können Transaktionen definiert und ausgeführt werden.
    * *Informationsmenü*
      In diesem Menü können u.a. das Repository und die Regelbeziehungen ausgewertet werden.
  - **AFPN Generation (AFPN)**
    In diesem System werden die Benutzerbefehle aus dem Menüsystem in eine für das System verarbeitbare Form gebracht. In ALFRED ist dies das Action Flow Petri Net (AFPN). Jeder Befehl wird in einem AFPN als Transition dargestellt, dabei werden notwendige Informationen für die Aktionsausführung in der Transition gespeichert.
- **Processing System**
  In diesem System werden die Regeln analysiert, gespeichert und verarbeitet. Es besteht aus folgenden Subsystemen:

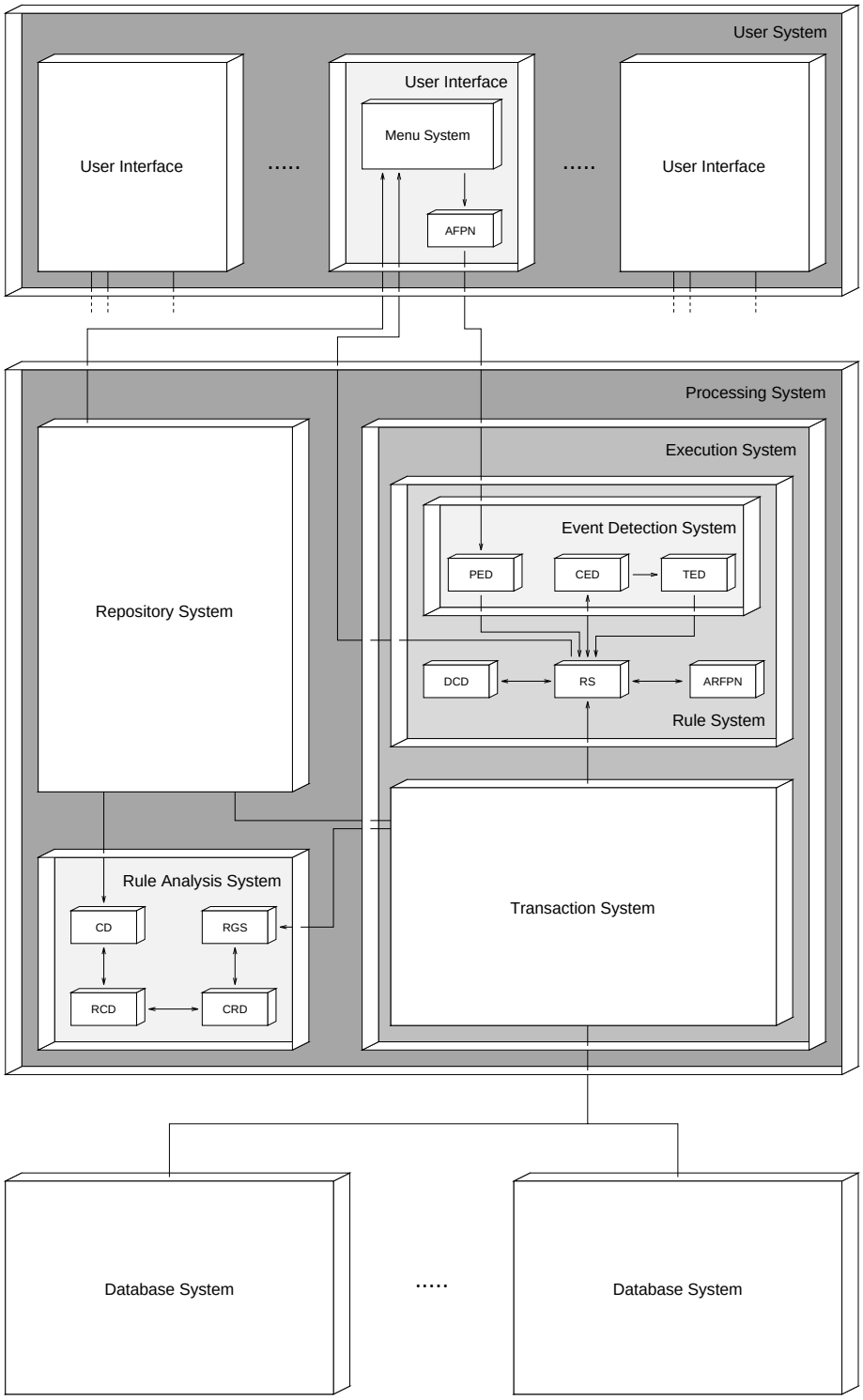

Abbildung 8.1: Die Subsysteme von ALFRED

- **Repository System**
  Hier werden die notwendigen Informationen über die definierten Regeln, Benutzer etc. gespeichert.
- **Rule Analysis System (RAS)**
  In diesem System werden Regeln, die durch den Benutzer definiert wurden, erzeugt und auf gewisse Eigenschaften hin untersucht. Dabei werden sie in folgender Reihenfolge in den Untersystemen des RAS erzeugt und analysiert:
  1. **Rule Generation System (RGS)**
     In diesem System werden die Regeln, die der Benutzer definiert, in ein Action Rule Flow Petri Net (ARFPN) umgewandelt.
  2. **Conflict and Redundancy Detection (CRD)**
     Hier wird die neu erzeugte Regelmenge auf Konflikte und Redundanzen hin untersucht.
  3. **Rule Cycle Detection (RCD)**
     In diesem System wird untersucht, ob durch die Integration einer neuen Regel Zyklen in der Regelmenge entstehen. Wenn dies der Fall ist, so wird statisch überprüft, ob diese Zyklen terminieren.
  4. **Confluence Detection (CD)**
     Hier wird überprüft, ob die neu erzeugte Regelmenge *confluent* ist.
- **Execution System**
  In diesem System erfolgt die eigentliche Regelverarbeitung. Es besteht aus den zwei Subsystemen *Rule System* und *Transaction System*. Das *Rule System* setzt sich aus folgenden Systemen zusammen:
  * **Event Detection System (EDS)**
    In diesem System werden die definierten Ereignisse erkannt. Dazu gehört die Erkennung von primitiven, komplexen und temporalen Ereignissen. Um diese Erkennung zu ermöglichen, setzt sich das EDS aus folgenden Systemen zusammen:
    · **Primitive Event Detection (PED)**
      Hier wird untersucht, ob das von der *AFPN Generation* kommende AFPN Aktionen enthält, durch deren Ausführung primitive Ereignisse in der definierten Regelmenge eintreten. Wenn dies der Fall ist, so wird das AFPN um Transitionen erweitert, die diese Ereignisauslösung modellieren.
    · **Complex Event Detection (CED)**
      Dieses System erkennt die primitiven und komplexen Ereignisse, die während der Regelverarbeitung eintreten und bestimmt somit die Menge der ausgelösten Regeln.
    · **Time Event Detection (TED)**
      Dieses System ist für die Erkennung von Zeitpunkten zuständig.
  * **Rule Simulation (RS)**
    In der Regelsimulation werden das ARFPN und eventuell ausgelöste Regeln verarbeitet. Dieses System verteilt die anfallenden Aufgaben an die entsprechenden Subsysteme. Ereignisse werden im EDS erkannt, Bedingungen und Aktionen werden an das Transaktionssystem weitergegeben.
  * **ARFPN Generation (ARFPN)**
    In diesem System werden diejenigen Regeln in das zu verarbeitende ARFPN integriert, die im Laufe der Verarbeitung ausgelöst werden.
  * **Dynamic Cycle Detection (DCD)**
    In der DCD werden die in der statischen Analyse gefundenen potentiell endlosen Zyklen überwacht und wenn nötig abgebrochen.

* **Transaction System**
  In diesem System werden Bedingungen ausgewertet und Aktionen auf der Datenbank ausgeführt. Dabei werden unter anderem Datensperren berücksichtigt.

8.1.2 Ablauf

Jeder Benutzer gibt seine Befehle über das *User System* ein. Diese Befehle werden sodann an die *AFPN Generation* übergeben, wo sie in ein Petrinetz umgewandelt werden. Dabei wird jeder Befehl als eigene Transition im Netz dargestellt. Das erzeugte AFPN wird daraufhin an das *Event Detection System*, genauer an die *Primitive Event Detection* übergeben. In diesem Subsystem wird überprüft, ob durch die Ausführung der Benutzerbefehle primitive Ereignisse eintreten. Wenn dies der Fall ist, so werden zusätzliche Transitionen in das AFPN eingefügt, mit deren Hilfe diese Ereignisauslösung dargestellt werden kann. Durch diese Erweiterung wird das Petrinetz zu einem ARFPN, das nun der Regelsimulation (*Rule Simulation*) übergeben wird. In der RS wird das ARFPN verarbeitet, indem Token durch das Petrinetz transportiert und entsprechende Aktionen (durch Transitionen dargestellt) zur Verarbeitung weitergegeben werden. Die Erkennung von Ereignissen wird dabei an die *Complex Event Detection* delegiert, die für die Erkennung von Zeitereignissen durch die *Time Event Detection* unterstützt wird. Falls während der Verarbeitung Regeln ausgelöst werden, so wird in der *ARFPN Generation* die entsprechende Regel aus der Regelmenge herauskopiert und in das bestehende Petrinetz (ARFPN) eingefügt. Die Ausführung von Aktionen sowie die Auswertung von Bedingungen wird an das Transaktionssystem weitergegeben. Wenn die vom Benutzer eingegebenen Befehle verarbeitet worden sind, so sendet die *Rule Simulation* eine entsprechende Meldung an das Benutzersystem, welches den Benutzer benachrichtigt. Damit ist gewährleistet, dass Benutzereingaben nur sequentiell erfolgen können.

8.2 Beispiel

In diesem Abschnitt werden zwei Beispielregeln definiert, anhand derer die Funktionsweise der Subsysteme von ALFRED erläutert werden kann. Diese beiden Regeln wurden bereits in Abschnitt 6.5.3 in vereinfachter Form definiert.

Beispiel (Produktion von Fahrradschläuchen): *In einem Unternehmen, welches Fahrradschläuche produziert, werden die Lagerbestände der Produktionsteile mit Hilfe eines DBS verwaltet. Die Herstellung eines Fahrradschlauches benötigt zwei Teile, einen Gummimantel und ein Ventil. Die Menge der zur Verfügung stehenden Teile wird in einem DBS abgespeichert. Diese Mengen sollen bei der Produktion eines Schlauches automatisch nachgeführt werden.*

Um dieses aktive Verhalten realisieren zu können, werden die zwei folgenden (vereinfachten) Regeln definiert:

```
Regel 1:  ON  post abstract event produziere_Fahrradschlauch
          DO  update  Teile
              set     Bestand = Bestand - 1
              where   Artikel = 'Gummi'

Regel 2:  ON  post update Teile set
          IF  Artikel = 'Gummi'
          DO  update  Teile
              set     Bestand = Bestand - 1
              where   Artikel = 'Ventil'
          EC-coupling immediate
          CA-coupling deferred
```

Regel 1, die nur aus einem Ereignis- und einem Aktionsteil besteht, wird durch ein abstraktes Ereignis (`post produziere_Fahrradschlauch`) ausgelöst. Danach wird der Bestand von *Gummi* in der Relation *Teile* um eins verringert. Die Angabe von Kopplungsmodi (KM) ist optional und fehlt in der Definition von Regel 1. Dies bedeutet implizit, dass die Regel mit EA-Kopplung *immediate* verarbeitet werden muss.

Die zweite Regel ist eine mengenorientierte ECA-Regel, die durch eine Änderung in der Relation *Teile* ausgelöst wird. Wenn diese Änderung den Artikel *Gummi* betrifft, d.h. die Bedingung erfüllt ist, so wird die Aktion der Regel ausgeführt. In diesem Fall wird der Bestand von *Ventil* in der Relation *Teile* um eins verringert. Diese Regel ist mit zwei verschiedenen Kopplungsmodi definiert. Die EC-Kopplung *immediate* besagt, dass die Bedingungsauswertung sofort, d.h. unmittelbar nach der Regelauslösung, zu erfolgen hat. Der KM *deferred*, zwischen Bedingungs- und Aktionsteil, legt fest, dass die Ausführung der Aktion erst am Ende der regelauslösenden Transaktion erfolgen muss.

Mit Hilfe dieser beiden Regeln wird das gewünschte aktive Verhalten realisiert, d.h. bei der Produktion eines Fahrradschlauches wird automatisch der Bestand von *Gummi* und *Ventil* nachgeführt. Dazu muss der Benutzer dem ADBS das abstrakte Ereignis (`produziere_Fahrradschlauch`) signalisieren.

Die Nachführung der Bestände von *Gummi* und *Ventil* könnte auch in einer einzigen Regel mittels einer komplexen Aktion erfolgen. Damit aber die für die Verarbeitung in ALFRED notwendigen Systeme besser erläutert werden können, wird diese Kombination von Regeln gewählt.

8.3 Subsysteme von ALFRED

In diesem Abschnitt werden die einzelnen Subsysteme von ALFRED erläutert. Jeder Abschnitt ist hierbei gleich aufgebaut. Zuerst wird allgemein erklärt, welche Aufgaben das System bei der Verarbeitung von Regeln zu erfüllen hat. Danach wird anhand des oben definierten Beispiels und anhand von Schnappschüssen aus der Regelverarbeitung aufgezeigt, was konkret in diesem System geschieht.

8.3.1 AFPN Generation

Die *AFPN Generation*, in Abbildung 8.1 als AFPN bezeichnet, liegt im Benutzersystem von ALFRED.

8.3.1.1 Aufgaben

Dieses System hat die Aufgabe, einen oder mehrere Benutzerbefehle in eine für die Simulation verarbeitbare Form zu bringen. Jeder Befehl oder jede Befehlskette wird in ein spezielles Petrinetz (AFPN) umgewandelt. In einem AFPN wird jeder Benutzerbefehl durch eine Transition dargestellt. Das in diesem System erzeugte AFPN besteht aus folgenden vier Teilen:

1. Der Anfang eines AFPN, der durch eine speziell gekennzeichnete Transition und einen dazugehörigen Vorplatz dargestellt wird. Diese Transition wird als *Begin_AFPN* bezeichnet.
2. Ein oder mehrere Benutzerbefehle. Jeder dieser Befehle wird als eine Transition im AFPN dargestellt, die mit der auszuführenden Aktion bezeichnet (z.B. `retrieve <...>`) wird. Die Informationen für die eigentliche Aktionsausführung wird in der Transition gespeichert.
3. Pseudo-Transaktionsgrenzen. Sofern der Benutzer keine expliziten Transaktionsgrenzen angibt, wird *jeder einzelne* Benutzerbefehl in pseudo-Transaktionsgrenzen eingeklammert, d.h. um die Transition, die den Benutzerbefehl darstellt, werden weitere Transitionen eingefügt.

Diese Transitionen sind *Pseudo_begin_of_transaction (Pseudo_bot), Pseudo_end_of_transaction (Pseudo_eot)* und *Pseudo_commit_of_transaction (Pseudo_cot)*. Mit Hilfe dieser Transaktionsgrenzen kann während der Verarbeitung u.a. die Berücksichtigung von Kopplungsmodi sichergestellt werden.

4. Das Ende des AFPN. Dieses wird durch eine speziell gekennzeichnete Transition (*End_AFPN*) dargestellt.

Nachdem das AFPN in diesem System erzeugt wurde, wird es an das *Rule System*, genauer an die *Primitive Event Detection* weitergereicht.

8.3.1.2 Beispiel

In Abbildung 8.2 ist das AFPN zu sehen, welches in der *AFPN Generation* entstanden ist, nachdem der Benutzer den Befehl zur Auslösung des abstrakten Ereignisses *(Raise abstract event (produziere_Fahrradschlauch))* im Menüsystem eingegeben hat.

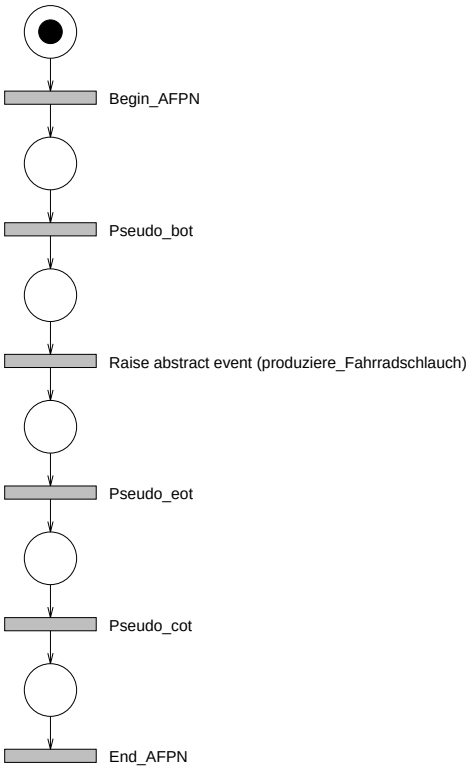

Abbildung 8.2: AFPN für *Raise abstract event(produziere_Fahrradschlauch)*

Im ersten Platz des AFPN liegt ein Token. In dieser Marke werden alle notwendigen Angaben des Benutzers gespeichert, damit während der Verarbeitung auf diese Informationen zurückgegriffen werden kann. Da der Benutzer keine expliziten Transaktionsgrenzen angegeben hat, wird um den eigentlichen Befehl eine pseudo-Transaktion gelegt. Diese Transaktion wird durch die drei Transitionen *Pseudo_bot, Pseudo_eot* und *Pseudo_cot* dargestellt. Der eigentliche Befehl, d.h. die Auslösung des abstrakten Ereignisses, wird als Transition dargestellt und gleich bezeichnet wie die entsprechende Aktion (`Raise abstract event (...)`). Das AFPN endet mit der *End_AFPN* Transition.

8.3.2 Primitive Event Detection

Die *Primtive Event Detection*, in Abbildung 8.1 als PED abgekürzt, liegt im *Rule System* und ist Teil des *Event Detection Systems*.

8.3.2.1 Aufgaben

Die primitive Ereigniserkennung hat die Aufgabe, ein AFPN zu untersuchen und festzustellen, ob in diesem AFPN Benutzerbefehle existieren, durch deren Ausführung primitive Ereignisse in der Regelmenge eintreten (Da die pseudo-Transaktionen keine Benutzerbefehle darstellen, werden sie für die primitive Ereigniserkennung nicht berücksichtigt). Für jeden Benutzerbefehl muss kontrolliert werden, ob ein entsprechendes Ereignis mit eventuell unterschiedlichen Auslösungszeitpunkten und -granularitäten existiert. Wenn dies der Fall ist, so werden speziell gekennzeichnete Transitionen in das AFPN eingefügt. Mit Hilfe dieser Transitionen wird eine Verbindung zwischen AFPN und dem Primärplatz (im Ereignisteil der Regeln) hergestellt, der das Ereignis repräsentiert.

Bei der Erweiterung des AFPN um diese Verbindungstransitionen müssen folgende Punkte berücksichtigt werden:

- **Auslösungszeitpunkte**
  Je nach dem, ob das Ereignis *vor* oder *nach* der Aktion ausgelöst werden soll, wird die neue Transition vor oder nach der Transition, welche die Aktion repräsentiert, eingefügt. Der Name dieser Transition beginnt jeweils mit *pre...* oder *post...*.

- **Auslösungsgranularität**
  Die mengen- und instanzorientierten Auslösungen werden durch zusätzliche Transitionen symbolisiert, damit sie bei der Verarbeitung berücksichtigt werden können. Eine mengenorientierte Auslösung wird mittels einer *pre-set...* oder *post-set...* Transition dargestellt, eine instanzorientierte hingegen mit Hilfe einer *pre-inst...* oder *post-inst...* Transition.

  Wenn in der *Primitive Event Detection* instanzorientierte Ereignisse gefunden werden, so müssen für diese Ereignisse *zwei* zusätzliche Transitionen eingefügt werden (vgl. Abbildung 8.3). Die erste Transition realisiert die Verbindung zwischen AFPN und dem entsprechenden Primärplatz (*pre...* oder *post...*). Die zweite Transition, *End-inst* genannt, wird benötigt, um in der *Rule Simulation* die korrekte Verarbeitung von instanzorientierten Auslösungen gewährleisten zu können. Diese Transition wird für jeden Benutzerbefehl, der instanzorientierte Ereignisse auslöst, einmal in das AFPN eingefügt und zwar vor der Transition, welche die Aktion repräsentiert, oder, falls sie existiert, nach der Transition, welche die *post-inst* Verbindung darstellt.

  In Abbildung 8.3 sind die beiden Möglichkeiten für das Einfügen einer *End-inst* Transition zu sehen. Im linken Petrinetz existiert nur eine pre-inst Auslösung, aus diesem Grund wird die *End-inst* Transition umgehend nach der Aktionstransition eingefügt. Das rechte Petrinetz beinhaltet zusätzlich eine post-inst Auslösung, weshalb die *End-inst* Transition erst nach der letzten instanzorientierten Auslösung (hier *post-inst Aktion*) eingefügt wird.

Durch das Einfügen von Verbindungen zu Primärplätzen, wird das AFPN in diesem System zu einem ARFPN. Aus diesem Grund müssen die Transitionen, welche den Anfang und das Ende des AFPN markieren, in *Begin_ARFPN* und *End_ARFPN* umbenannt werden. Das so entstandene ARFPN wird an die *Rule Simulation* weitergegeben.

8.3.2.2 Beispiel

In Abbildung 8.4 ist das ARFPN zu sehen, welches in der *Primitive Event Detection* aus dem AFPN der *AFPN Generation* entstanden ist.

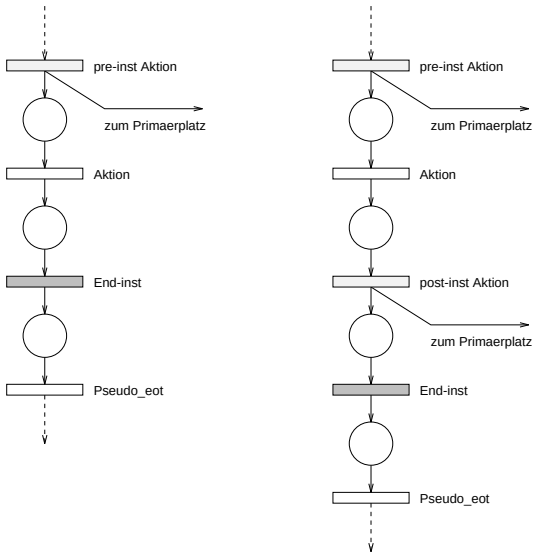

Abbildung 8.3: Einfügen einer *End-inst* Transition

Da in der definierten Regelmenge ein post-Ereignis für den Benutzerbefehl existiert, wird nach der entsprechenden Transition eine zusätzliche Verbindungstransition eingefügt (*post produziere_Fahrradschlauch*). Da es bei abstrakten Ereignissen nicht sinnvoll ist, zwischen mengen- und instanzorientierter Auslösung zu unterscheiden, wird hier die Auslösungsgranularität nicht angegeben. Mit Hilfe der neu eingefügten Transition wird eine Verbindung zwischen dem ARFPN und dem Primärplatz hergestellt, der das primitive Ereignis in der Regelmenge repräsentiert. Die Regel 1 ist nun wiederum mit diesem Primärplatz verbunden, da sie durch das Eintreten dieses Ereignisses ausgelöst wird.

8.3.3 Rule Simulation

In der Regelsimulation (RS) geschieht die eigentliche Verarbeitung des ARFPN. Dieses System liegt im *Rule System* von ALFRED. Die Funktionsweise dieses Systems ist in Anhang F als Pseudocode-Algorithmus dargestellt.

8.3.3.1 Aufgaben

In der *Rule Simulation* wird ein ARFPN verarbeitet. Dies geschieht indem die Marke, welche sich im ersten Platz des ARFPN befindet, nach vorgegebenen Schaltregeln durch das Petrinetz transportiert wird. Wenn während der Verarbeitung Regeln ausgelöst werden, so wird das ARFPN dynamisch um diese Regeln erweitert.

Um die Funktionsweise dieses Systems besser erläutern zu können, wird hier die Struktur der Regelmenge in ALFRED aufgezeigt. Die in ALFRED definierten Regeln werden im *Repository* abgelegt und gleichzeitig als Petrinetze dargestellt. Damit während der Verarbeitung Ereignisse erkannt und ausgelöste Regeln berücksichtigt werden könnnen, ist die ganze Petrinetzmenge wie folgt aufgeteilt (vgl. Abbildung 8.5):

- *ARFPN*
  Dieses ARFPN wird in der *Rule Simulation* verarbeitet. Wenn im ARFPN Befehle existieren, durch deren Ausführung vordefinierte primitive Ereignisse eintreten, so wird diese Abhängig-

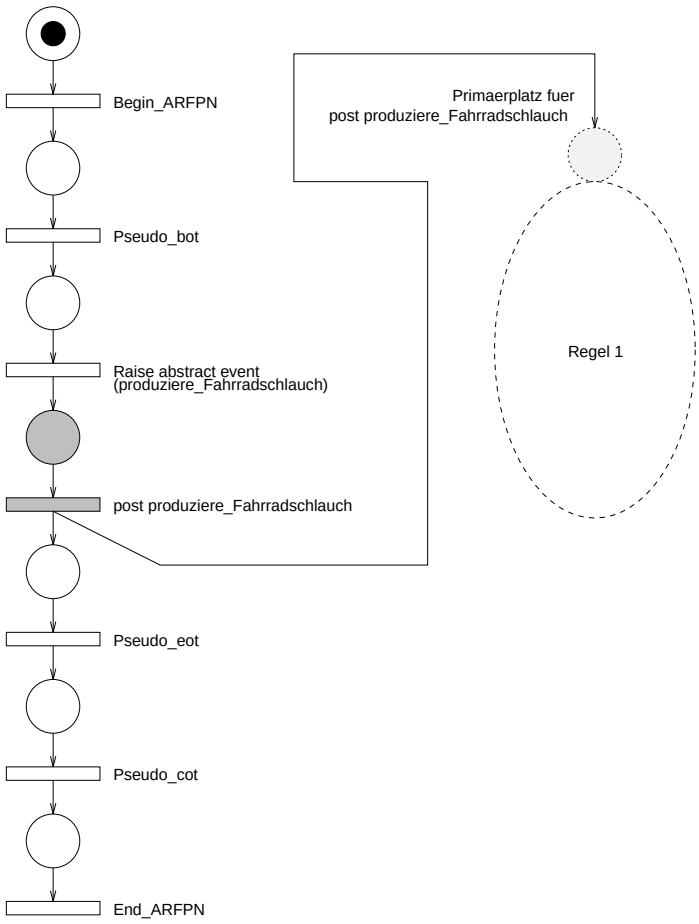

Abbildung 8.4: Beispiel-ARFPN nach der *Primitive Event Detection*

keit mit einer Verbindung zwischen ARFPN und den Ereignissen, resp. Primärplätzen, modelliert. Wenn während der Verarbeitung Regeln ausgelöst werden, so müssen entsprechende Regelinstanzen in das ARFPN integriert werden.

- *Ereignisse der Regelmenge*
  Die Ereignisse aller Regeln werden logisch als eigene Einheit betrachtet. In dieser Menge von Ereignissen werden primitive und komplexe Ereignisse erkannt (*Complex Event Detection*).

- *Regelmenge*
  Alle in ALFRED definierten Regeln sind in der Regelmenge als gefärbte Petrinetze abgespeichert. Diese Regeln können Verbindungen zu Ereignissen aufweisen, falls sie ereignisauslösende Aktionen beinhalten. Die Regelmenge wird *nicht* durchlaufen, sie bleibt unangetastet von der Simulation. Jede Regel, die zur Ausführung kommt, wird durch die *ARFPN Generation* aus der Regelmenge herauskopiert (instanziiert) und in das ARFPN integriert.

Die *Rule Simulation* hat nun die Aufgabe, das Petrinetz zu durchlaufen und je nach Art der zu verarbeitenden Transitionen, geeignete Massnahmen zu treffen, resp. die Verarbeitung von Befehlen an andere Systeme zu delegieren. Die *Rule Simulation* bewirkt selbst *keine* Änderungen auf der zugrundeliegenden Datenbank. Folgende Aufgaben werden an andere Systeme verteilt:

- **Ereigniserkennung**
  Ein primitives Ereignis kann nur eintreten, wenn während der Verarbeitung eines ARFPN

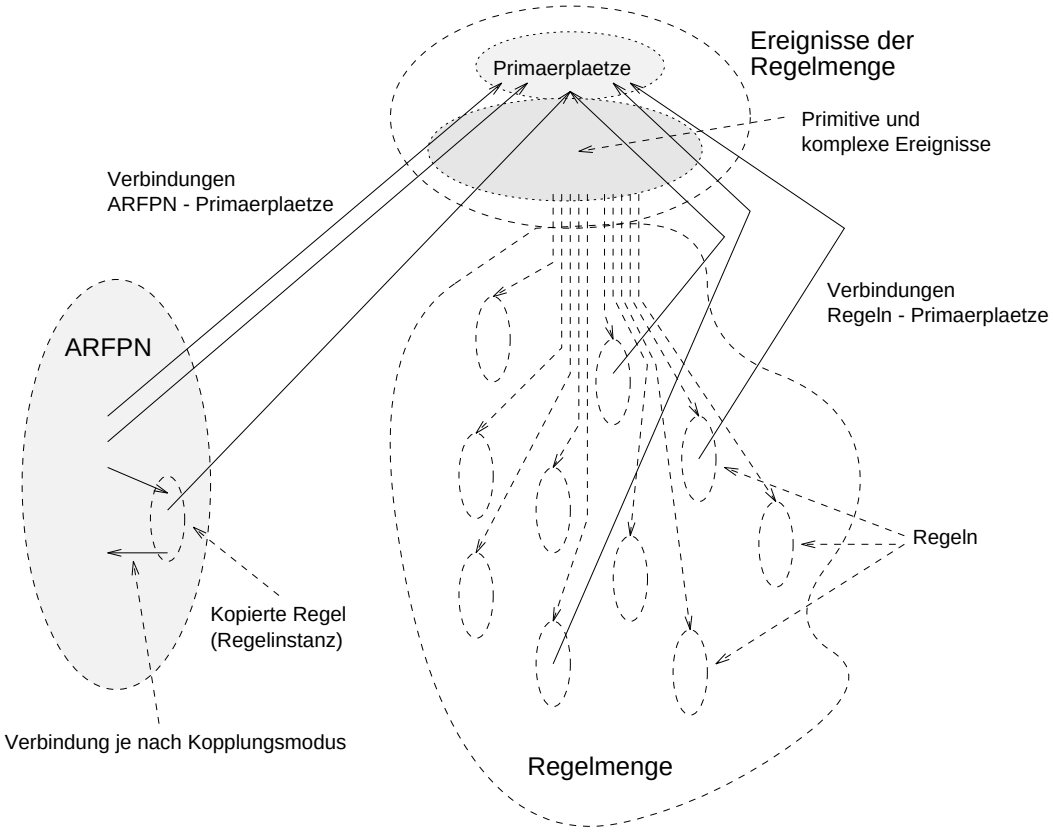

Abbildung 8.5: Aufteilung der Petrinetze in ALFRED

ein Token über eine Transition gefeuert wird, welche das ARFPN mit dem entsprechenden Primärplatz verbindet. Das Token, das auf diese Weise in die Ereignismenge gelangt, wird in der *Complex Event Detection* selbständig durch den Ereignisteil transportiert. Die möglicherweise ausgelösten Regeln werden der *Rule Simulation* gemeldet.

- **Auslösen und integrieren von Regeln**
  Ausgelöste Regeln müssen in die laufende Verarbeitung miteinbezogen werden. Dazu wird der *ARFPN Generation* die ausgelöste Regel mitgeteilt. Dieses System kopiert die Regeln aus der bestehenden Regelmenge heraus und fügt sie unter Berücksichtigung gewisser Kriterien wie z.B. Kopplungsmodi in das ARFPN ein.

- **Aktionsausführung und Bedingungsauswertung**
  Wenn während der Verarbeitung eine Transition gefeuert werden kann, welche eine primitive Aktion oder eine primitive Bedingung darstellt, so wird die Ausführung dieser Aktion resp. die Auswertung der Bedingung an das Transaktionssystem delegiert. Die *Rule Simulation* erhält das Ergebnis dieser Ausführung oder Auswertung zurück.

In der Regelverarbeitung muss also jedesmal, wenn eine Transition feuern kann, entsprechend der Art der Transition agiert werden. Da im Laufe der Simulation nicht nur das ursprüngliche ARFPN verarbeitet wird, sondern auch allfällige Regeln, die durch die Befehle im ARFPN ausgelöst werden, gibt es viele verschiedene Transitionen, welche die Simulation verarbeiten muss. Die möglichen Transitionsarten und eine kurze Beschreibung, was beim Feuern dieser Transitionen gemacht werden muss, ist im Anhang D zu finden.

8.3.3.2 Beispiel

In Abbildung 8.6 ist das Petrinetz für das definierte Beispiel zu sehen. Die *Rule Simulation* erhält von der *Primitive Event Detection* das ARFPN, welches mit dem Primärplatz *post_produziere_-Fahrradschlauch* verbunden ist. Die Primärplätze und die Ereignisse der Regeln, d.h. der Ereignisteil einer Regel bis und mit *Ereignisende*, werden als logische Einheit betrachtet. Auf dieser Ereignismenge kann die *Complex Event Detection* primitive und komplexe Ereignisse erkennen und die ausgelösten Regeln bestimmen. Der Bedingungs- resp. Aktionsteil der beiden Regeln ist mit dem jeweiligen *Ereignisende* verbunden. Man beachte, dass jede Aktion innerhalb einer Regel ebenfalls in pseudo-Transaktionsgrenzen eingeschlossen wird, wenn der Benutzer nicht explizit eine Transaktion definiert hat.

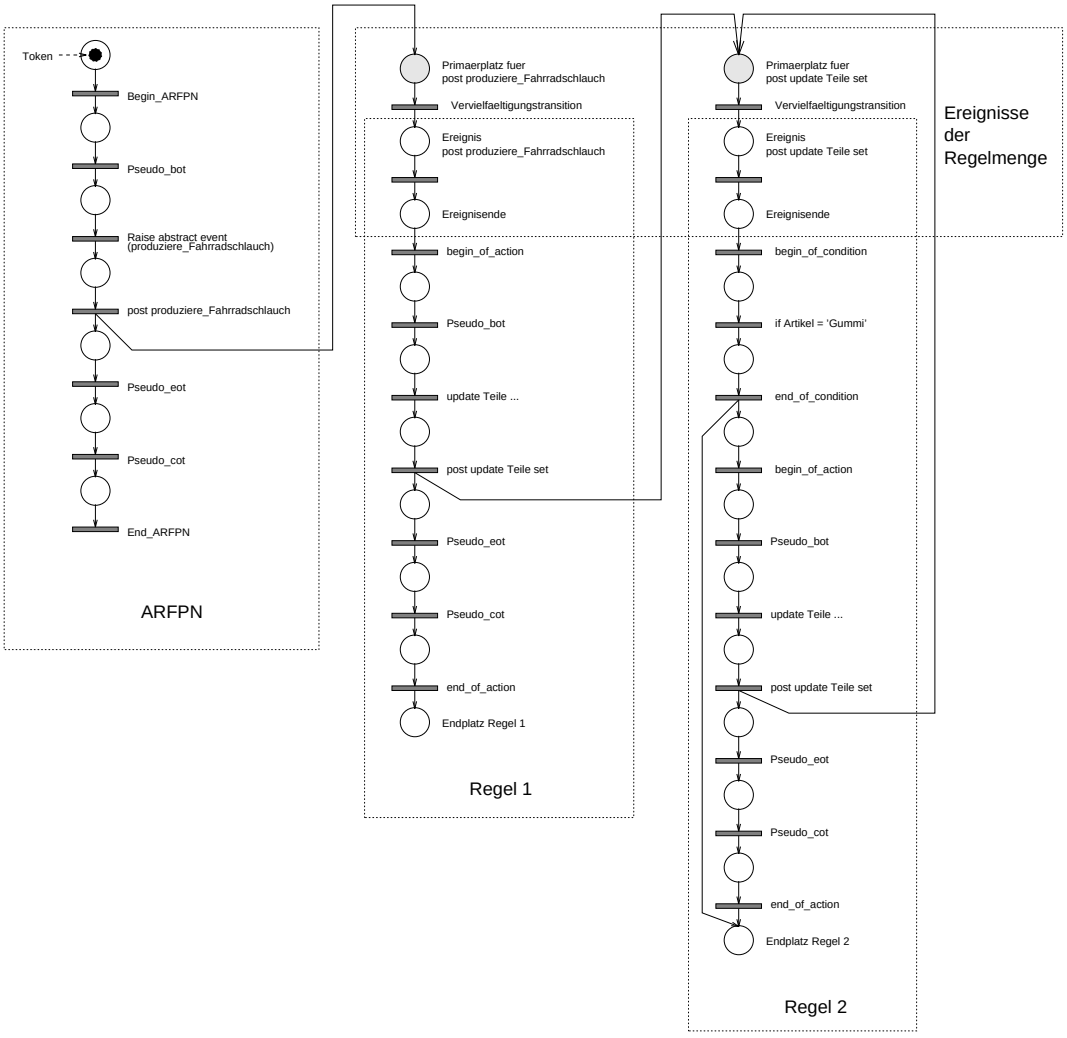

Abbildung 8.6: Petrinetz für das Beispiel

Aus der Sicht der Regelsimulation existiert zu diesem Zeitpunkt nur das ARFPN mit der Verbindung zu einem Primärplatz. Die Simulation wird nun das Token, welches sich im Anfangsplatz des ARFPN befindet, durch das Petrinetz transportieren und beim Feuern der Transitionen entsprechende Aktionen ausführen. Die folgende Auflistung zeigt den Ablauf in der *Rule Simulation* chro-

nologisch bis zum Zeitpunkt der Ereignisauslösung anhand der Transitionen, die gefeuert werden:

1. *Begin ARFPN*
   Das *Transaction System* (TS) wird informiert. Dieses gibt der *Rule Simulation* (RS) die Erlaubnis zur Ausführung, worauf die RS diese Transition feuert.

2. *Pseudo_bot*
   Das TS wird informiert, dass eine pseudo-Transaktion beginnt. Die RS erhält vom TS eine Transaktionsnummer zurück. Daraufhin wird vom jetzigen Zustand des Netzes eine Momentaufnahme gemacht, d.h. die Nummer der pseudo-Transaktion sowie sämtliche Plätze im ARFPN und im Ereignisteil, die mit Token belegt sind, werden gespeichert. Die Speicherung der Token geschieht, damit bei einem allfälligen Abbruch der Aktion oder der Transaktion das Netz auf den Zustand vor Beginn der Transaktion zurückgesetzt werden kann. In diesem Beispiel wird also nur der Platz vor *Pseudo_bot* gespeichert. Danach wird die RS die Transition feuern.

3. *Raise abstract event ...*
   Die Ausführung dieser Aktion wird dem TS übergeben. Da diese Aktion nicht auf der Datenbank ausgeführt werden muss, wird das TS der RS den Befehl geben, die Transition zu feuern.

4. *post produziere_Fahrradschlauch*
   Das TS wird informiert, dass eine Ereignisverbindungstransition gefeuert werden kann. Das TS erteilt der RS die Erlaubnis zur Ausführung, worauf die RS diese Transition feuert. Zu diesem Zeitpunkt existieren nun *zwei* Token im Netz. Das erste Token befindet sich im Vorplatz von *Pseudo_eot*, das zweite Token im Primärplatz von *post produziere_Fahrradschlauch* (vgl. Abbildung 8.7). Da nun ein Token in den Ereignisteil transportiert wurde, wird als nächstes die *Complex Event Detection* angestossen, indem der CED der markierte Primärplatz mitgeteilt wird. Die Verarbeitung des ARFPN wird für die Zeit der Ereigniserkennung unterbrochen.

   Bemerkung: Wenn die Verarbeitung des ARFPN hier weiterginge, so wäre sie nicht mehr korrekt, da an dieser Stelle die Verarbeitung von Regel 1 intergriert werden muss. Dazu ist es notwendig, dass die Regel als ausgelöst erkannt wird (in CED) und dass sie aus der Regelmenge herauskopiert und in das ARFPN integriert wird (in der *ARFPN Generation*).

Um dem chronologischen Ablauf bei der Verarbeitung zu folgen, wird als nächstes die CED erläutert. Einige weitere Ausschnitte aus der Verarbeitung der Beispielregelmenge in der *Rule Simulation* sind im Abschnitt 8.4 zu finden.

8.3.4 Complex Event Detection

Die komplexe Ereigniserkennung, abgekürzt als CED, ist Teil des *Event Detection System* und liegt im *Rule System* von ALFRED. Die Funktionsweise dieses Systems ist in Anhang E als Pseudocode-Algorithmus aufgeführt.

8.3.4.1 Aufgaben

Dieses System ist für die Erkennung von primitiven und komplexen Ereignissen zuständig. Die Erkennung geschieht dabei auf den Petrinetz-Strukturen, die eigens für die Darstellung und Erkennung von komplexen Ereignissen entworfen wurden (vgl. Abschnitte 6.2 und 6.5). Das System hat die Aufgabe, Marken durch den Ereignisteil der Regelmenge zu transportieren. Dabei müssen Parameter gebunden und mittels Parameteroperatoren zusammengefasst werden.

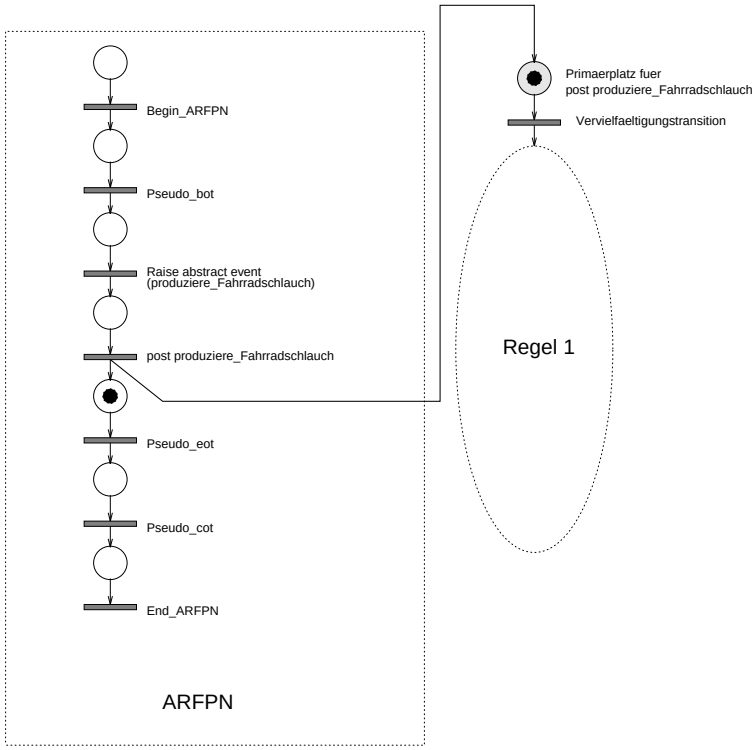

Abbildung 8.7: Situation in RS nach Feuern der Verbindungstransition

Die *Complex Event Detection* wird durch die *Rule Simulation* angestossen. Von ihr erhält das System einen oder mehrere markierte Primärplätze. Ausgehend von diesen markierten Plätzen, können die Marken durch das Petrinetz transportiert werden. Da die CED nur für die Simulation des Ereignisteiles zuständig ist, werden die Marken höchstens bis in das *Ereignisende* gelegt, d.h. in den Vorplatz einer *begin_of_condition* oder *begin_of_action* Transition. Wenn dieser Platz durch die Simulation im Ereignisteil erreicht wird, so ist das Ereignis einer Regel eingetreten. Die CED trägt die Identifikation dieser Regel in einer Liste ein. Das Petrinetz im Ereignisteil wird durch die CED solange durchlaufen, bis keine Transitionen mehr feuern können.

Da auch Ereignisoperatoren für Zeitereignisse existieren (vgl. Abschnitt 6.2.4), müssen die entsprechenden Zeitpunkte berücksichtigt werden. Die Erkennung dieser Zeitpunkte wird dabei an die *Time Event Detection* (TED) delegiert. Die TED erhält von der *Complex Event Detection* einen Zeitpunkt und einen Platz, den die TED beim Erreichen des Zeitpunktes markieren muss. Falls der entsprechende Zeitpunkt eintritt, so wird dies der *Rule Simulation* mitgeteilt, die ihrerseits unverzüglich die CED benachrichtigt, und ihr den entsprechenden markierten Platz im Ereignisteil übergibt. Ausgehend von diesem Platz werden die Marken wiederum solange durch den Ereignisteil transportiert, bis keine Transition im Ereignisteil mehr feuern kann.

Wenn die Ereigniserkennung keine Transitionen mehr feuern kann, so wird der *Rule Simulation* die Liste mit den Regelidentifikationen übergeben. Diese Liste enthält alle in diesem Schritt *ausgelösten* Regeln.

8.3.4.2 Beispiel

In Abbildung 8.8 sind zwei Schnappschüsse aus der Ereigniserkennung zu sehen. Die linke Seite der Abbildung zeigt die Situation *vor* dem Start der CED, die rechte Seite zeigt die Situation danach.

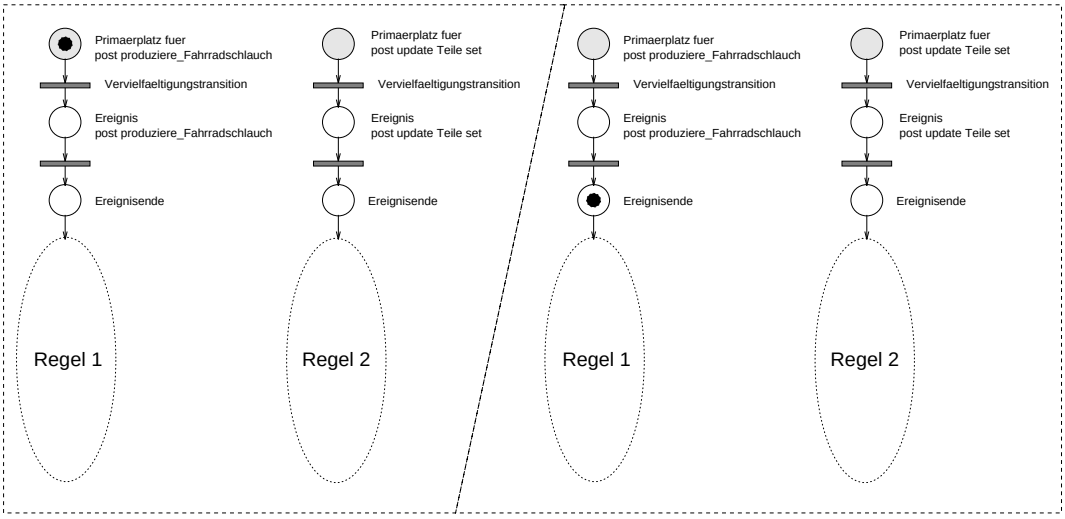

Abbildung 8.8: Ereignisteil *vor* und *nach* der Ereigniserkennung

In diesem Beispiel ist die Ereigniserkennung einfach, da nur eine Regel mit der Vervielfältigungstransition verbunden ist, und da die Regel nur aus einem primitiven Ereignis besteht. Die CED wird nun die Vervielfältigungstransition feuern und das Token somit in den Platz für das primitive Ereignis in Regel 1 legen. Danach wird die CED die nächste Transition feuern, woraufhin das Token in das *Ereignisende* zu liegen kommt. Die Regel, die mit diesem *Ereignisende* verbunden ist (Regel 1), wird nun in eine Liste eingefügt. Da die CED keine Transitionen im Ereignisteil mehr feuern kann, wird die Liste der Regeln an die *Rule Simulation* weitergegeben.

8.3.5 ARFPN Generation

Die ARFPN-Generierung, in Abbildung 8.1 als ARFPN abgekürzt, ist ein Teilsystem von *Rule System*.

8.3.5.1 Aufgaben

Dieses System hat die Aufgabe, Regeln, deren Auslösung durch die Ereigniserkennung registriert wurde, in das bestehende ARFPN zu integrieren. Von *Rule Simulation* erhält das System eine Liste von ausgelösten Regeln. Das *Ereignisende* sowie die Bedingungs- und Aktionsteile dieser Regeln werden nun aus der Regelmenge *herauskopiert*, die Regel wird also *instanziiert*. Damit die Information aus dem Ereignisteil nicht verloren geht, wird das entsprechende Token samt Parametern aus dem originalen *Ereignisende* entfernt und in das kopierte *Ereignisende* gelegt. Nun müssen die Regelkopien in das bestehende ARFPN, unter Berücksichtigung von Kopplungsmodi, Prioritäten und Fristen, integriert werden.

Je nach dem, welche Kopplungsmodi die einzuhängende Regel hat, ergeben sich verschiedene Verbindungsvarianten, sowohl zwischen der Regel und dem bestehenden ARFPN als auch innerhalb der Regel selbst. Da in ALFRED sechs verschiedene KM unterstützt werden, gibt es gesamthaft 36 Kombinationen von Kopplungsmodi (nicht alle Kombinationen sind sinnvoll), wobei an dieser Stelle nur zwei mögliche Kombinationen aufgezeigt werden.

Nachfolgend werden einige Begriffe erläutert, die in den Abbildungen 8.10 und 8.11 verwendet werden:

- *Nachfolgetransition*
  Mit *Nachfolgetransition* wird diejenige Transition im ARFPN oder im Bedingungs- resp. Aktionsteil einer Regel bezeichnet, die nach einer Transition folgt, welche die Verbindung zu einem Primärplatz herstellt (*pre-set...*, *pre-inst...*, *post-set...*, *post-inst...*).

- *EOT-Verteiler*
  Der *EOT-Verteiler* ist eine Struktur, die Marken im Platz nach einer *Pseudo_eot* oder *eot* Transition sooft vervielfacht, wie es Regelteile gibt, die mit KM *deferred* zur regelauslösenden Transaktion stehen (vgl. Abbildung 8.9). Eine Vervielfältigungstransition wird über einen zusätzlichen Platz mit der *Pseudo_eot* oder der *eot* Transition verbunden, und es werden soviele Ausgangsplätze erzeugt, wie nötig sind. Jeder Bedingungsteil und Aktionsteil einer Regel, der mit Kopplungsmodus *deferred* zur regelauslösenden Transaktion steht, benötigt einen solchen Platz. Somit wird garantiert, dass Regeln, die mit KM *deferred* verarbeitet werden müssen, zum richtigen Zeitpunkt (nach dem Ende der regelauslösenden Transaktion aber vor dem commit) berücksichtigt werden.

- *COT*
  Mit *COT* wird entweder die *Pseudo_cot* oder die *cot* Transition bezeichnet.

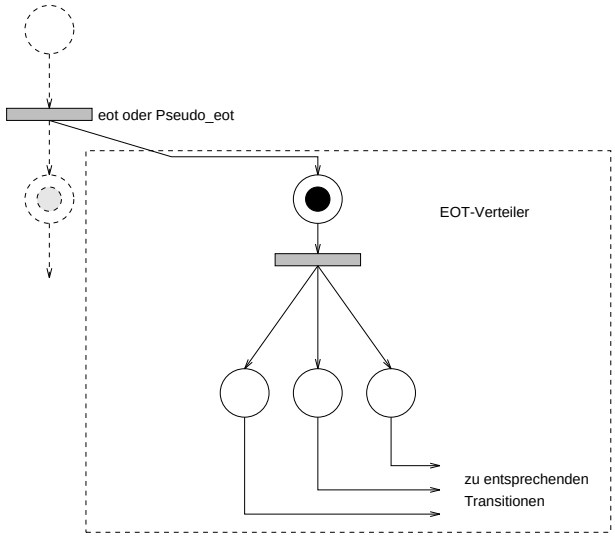

Abbildung 8.9: Verteilung von Marken (EOT-Verteiler)

Abbildung 8.10 zeigt eine ECAA-Regel und die Verbindungen (durchgezogene Linien), die nötig sind, um eine Regel mit Kopplungsmodi (KM) *immediate/deferred* in das ARFPN einzubinden. Bei dieser Kombination von KM wird ein zusätzlicher Platz benötigt, damit nach der Verarbeitung des *immediate*-Teils, ein Token an das ARFPN zurückgegeben werden kann. Dieser Platz, *always_out* genannt (in der Abbildung schraffiert dargestellt), wird an die letzte Transition des *immediate*-Teils (hier *end_of_condition*) gehängt und vom Petrinetz-Algorithmus beim Feuern dieser Transition, unabhängig von der Bedingungsauswertung, *immer* mit einem Token belegt.

Abbildung 8.11 zeigt eine ECAA-Regel und die Verbindungen (durchgezogene Linien), die nötig sind, um eine Regel mit Kopplungsmodi *deferred/immediate* in das ARFPN einzubinden. Im Unterschied zu einer Regel mit *immediate/immediate* KM darf diese Regel erst verarbeitet werden, wenn das Ende der regelauslösenden Transaktion erreicht ist. Aus diesem Grund wird eine Verbindung von EOT-Verteiler (vgl. Abbildung 8.9) zu *begin_of_condition* gezogen. Diese Verbindung garan-

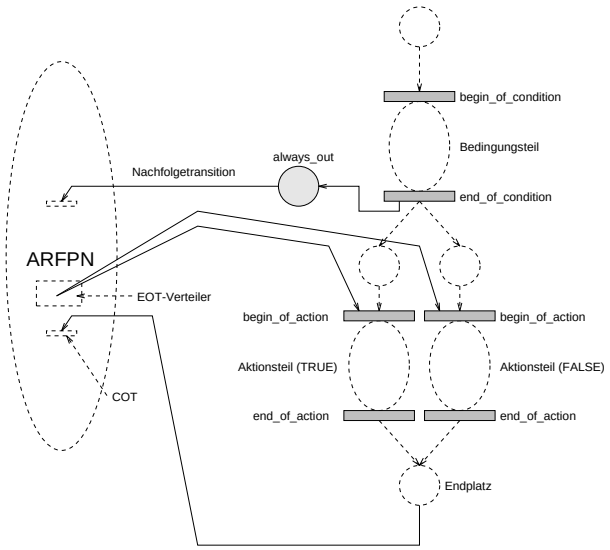

Abbildung 8.10: Verbindungen für Regel mit Kopplungsmodi *immediate/deferred*

tiert, dass die Verarbeitung erst beginnen kann, wenn eine Marke im Platz nach der *Pseudo_eot* oder *eot* Transition liegt.

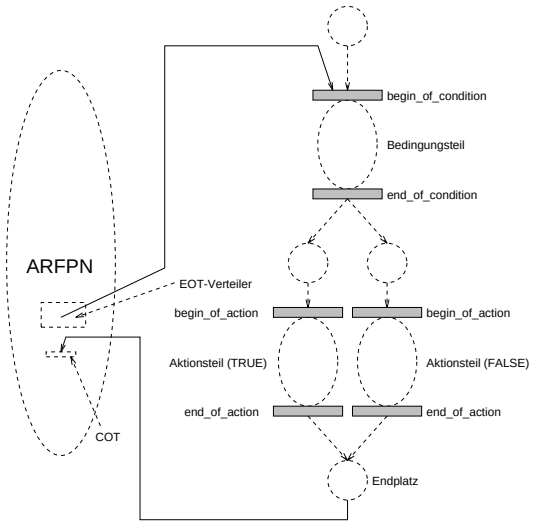

Abbildung 8.11: Verbindungen für Regel mit Kopplungsmodi *deferred/immediate*

Falls mehrere Regeln gleichzeitig ausgelöst werden, so müssen bei der Regelverarbeitung die *Prioritäten* berücksichtigt werden. Wenn alle ausgelösten Regeln dieselbe Priorität besitzen, so müssen keine neuen Verbindungen gezogen werden. Haben die Regeln jedoch unterschiedliche Prioritäten, so müssen durch die *ARFPN Generation* zusätzliche Verbindungen zwischen den Regelinstanzen eingefügt werden. Diese Verbindungen stellen sicher, dass die niedrig priorisierten Regeln erst nach den höher priorisierten Regeln verarbeitet werden.

In Abbildung 8.12 ist die Verbindung zu sehen, die notwendig ist, wenn Regel A eine höhere Priorität als Regel B besitzt. Unter der Annahme, dass beide Regeln KM *immediate/immediate*

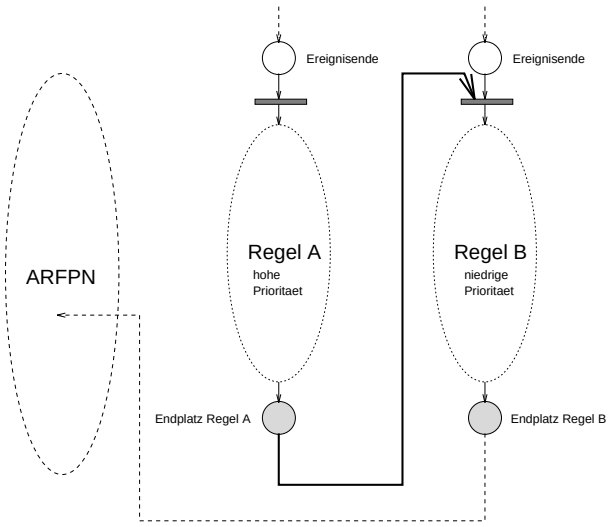

Abbildung 8.12: Verbindungen für die Berücksichtigung von Prioritäten

besitzen, wird vom *Endplatz* der Regel A eine Verbindung zur ersten Transition von Regel B gezogen. Die erste Transition ist diejenige, die unmittelbar auf das *Ereignisende* folgt. Mit dieser Verbindung wird garantiert, dass die Verarbeitung der niedriger priorisierten Regel erst beginnnen kann, wenn die höher priorisierte Regel verabeitet ist, d.h. wenn sich ein Token im *Endplatz* der Regel A befindet. Am Ende der Verarbeitung dieser beiden Regeln, d.h. wenn der *Endplatz* von Regel B erreicht ist, wird das Token mittels einer Verbindung vom *Endplatz* der Regel B an das ARFPN zurückgegeben. Somit kann die Simulation das ARFPN weiterverarbeiten.

8.3.5.2 Beispiel

In Abbildung 8.13 ist das ARFPN und die eingehängte Regel 1 zu sehen. Die *ARFPN Generation* erhält von der *Rule Simulation* eine Liste, die nur gerade die ausgelöste Regel 1 enthält. Das *Ereignisende* sowie der Aktionsteil der Regel werden nun aus der Regelmenge herauskopiert, wobei das Token, das sich im originalen *Ereignisende* befindet, gelöscht und in das kopierte *Ereignisende* gelegt wird. Die Verbindung von *post update Teile set* zum entsprechenden Primärplatz wird ebenfalls mitkopiert. Da die Regel (EA) mit Kopplungsmodus *immediate* definiert ist, muss daraufhin eine Verbindung vom *Endplatz* der Regel zurück auf die der Verbindungstransition (*post produziere...*) folgende Transition (*Pseudo_eot*) gezogen werden. Die Regel ist nun in das bestehende ARFPN integriert, und die *Rule Simulation* kann die Verarbeitung des ARFPN fortsetzen.

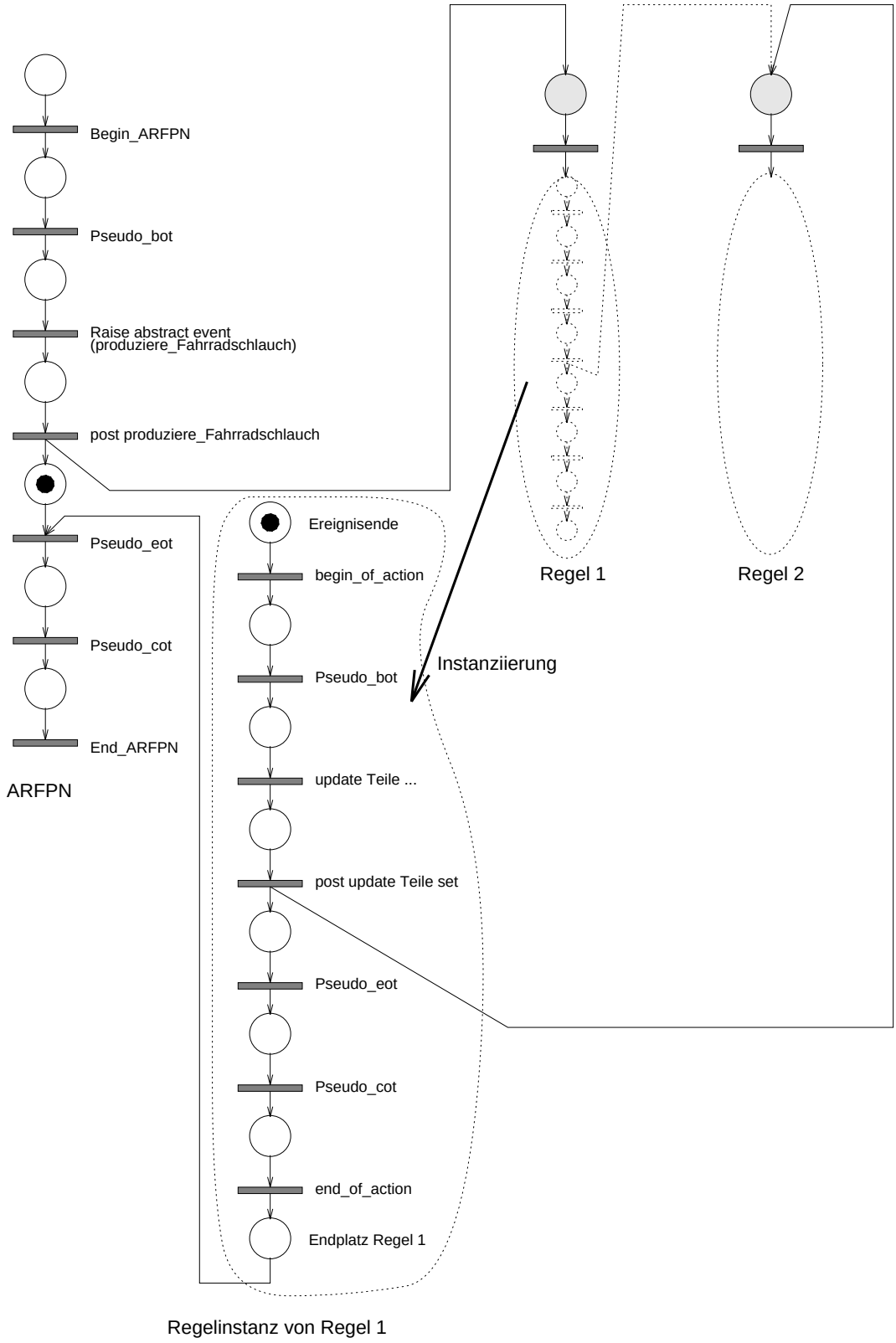

Abbildung 8.13: ARFPN mit eingehängter Regel 1 nach der *ARFPN Generation*

8.4 Momentaufnahmen in der Rule Simulation

Bei der Verarbeitung der Beispielregelmenge wird die Regel 1 die Regel 2 auslösen. Die Situation, wie sie sich in der *Rule Simulation* stellt, ist in Abbildung 8.14 zu sehen.

Nach dem Feuern der *post update Teile set* Transition in der Regelinstanz von Regel 1, übernimmt die *Complex Event Detection* die Ereigniserkennung und meldet die Regel 2 als ausgelöst. Diese wird durch die *ARFPN Generation* in das Petrinetz eingehängt. Da Regel 2 mit KM *immediate/deferred* definiert wurde, muss die Regel entsprechend in das Netz integriert werden. Als erstes muss eine Verbindung vom Ende des *immediate*-Teils, in diesem Fall die *end_of_condition* Transition in der Regelinstanz von Regel 2, zum *always_out*-Platz gezogen werden. Dieser Platz wird daraufhin mit der *Pseudo_eot* Transition von Regelinstanz 1 verbunden (weil die *Pseudo_eot* Transition auf die *post update Teile set* Transition folgt). Als zweites wird ein *EOT-Verteiler* mit der *Pseudo_eot* Transition von Regelinstanz 1 verbunden. Dieser Verteiler (gestricheltes Rechteck) besteht in diesem Fall aus einem Platz vor der Transition und einem Platz danach. Von diesem Platz aus muss eine Verbindung zur ersten Transition gezogen werden, die auf den *immediate*-Teil von Regelinstanz 2 folgt (*begin_of_action*). Als letztes muss die *ARFPN Generation* noch eine Verbindung vom *Endplatz* der Regelinstanz von Regel 2 zur *Pseudo_cot* Transition der Regelinstanz 1 ziehen.

Mit Hilfe dieser Verbindungen ist es nun möglich, dass der Bedingungsteil der Regelinstanz von Regel 2 mit KM *immediate* durchgeführt wird. Beim Feuern der *end_of_condition* Transition werden zwei Token erzeugt. Das eine Token wird in den *always_out*-Platz gelegt, das zweite Token kommt in den Platz vor *begin_of_action* zu liegen (vgl. Abbildung 8.15). Nun wird als nächstes die Verarbeitung von Regelinstanz 1 weitergehen. Die *Rule Simulation* kann die *Pseudo_eot* Transition feuern und legt damit ein Token in jeden der beiden Plätze nach der *Pseudo_eot* Transition. Das eine Token wird mit Hilfe des *EOT-Verteilers* für die weitere Verarbeitung der Regelinstanz 2 zur Verfügung gestellt. Da nun also das Ende der pseudo-Transaktion in Regelinstanz 1 erreicht ist, kann die Verarbeitung des Teils mit KM *deferred* von Regelinstanz 2 in Angriff genommen werden.

Wenn das ARFPN vollständig durchlaufen wurde, d.h. wenn ein Token über die *End ARFPN* Transition gefeuert wird, so erhält der Benutzer die Meldung, dass sein Befehl (zur Erinnerung: Auslösung des abstrakten Ereignisses *produziere_Fahrradschläuche*) verarbeitet wurde.

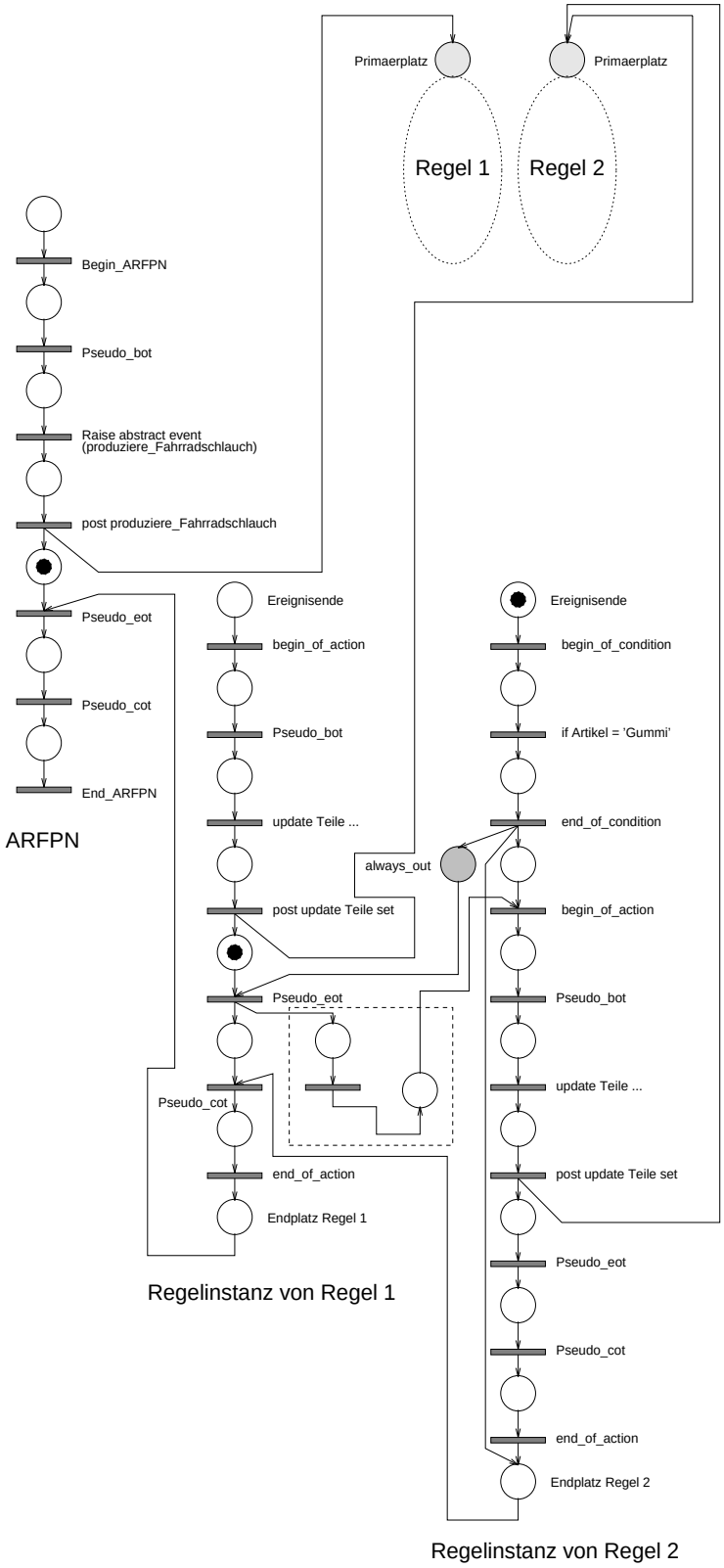

Abbildung 8.14: Auslösung und Integration von Regel 2

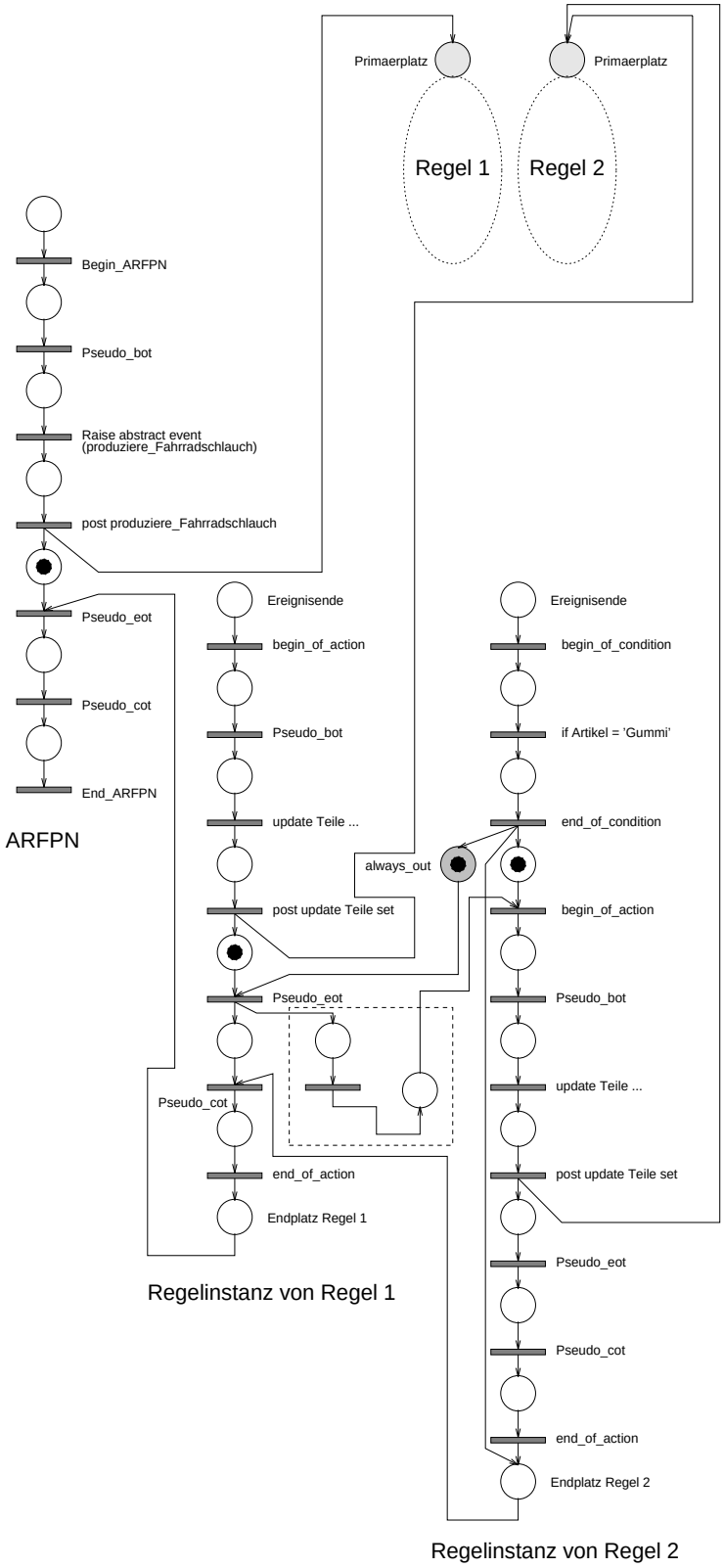

Abbildung 8.15: Zustand nach Feuern von end_of_condition von Regel 1

8.5 Szenarien

Um den Ablauf und einige Spezialfälle etwas genauer zu erläutern, werden im folgenden Szenarien dargestellt, die während der Simulation eintreten können. Einige Szenarien sind bereits bekannt, werden aber zum besseren Verständnis noch einmal angegeben.

- **Ereignisauslösung**
  Falls eine Transition feuern kann, welche eine Verbindung zwischen ARFPN und Ereignisteil darstellt (z.B. *pre-set...*), so wird eine Marke in den entsprechenden Primärplatz gelegt und die *Complex Event Detection* gestartet. Ausgehend vom markierten Primärplatz, spielt diese sämtliche im Ereignisteil vorkommenden Token durch das Netz und bestimmt somit alle Regeln, die ausgelöst werden. Diese Liste von ausgelösten Regeln gelangt zurück an die *Rule Simulation*, welche diese Liste mit zusätzlichen Informationen über Transaktionsgrenzen, Aktionsblockgrenzen etc. an die *ARFPN Generation* weitergibt. Hier werden die ausgelösten Regeln *instanziiert*, mit notwendigen Verbindungen versehen und in das bestehende ARFPN eingehängt. Daraufhin erhält die *Rule Simulation* die Kontrolle zurück, und die Simulation kann weitergehen.

- **Aktionsausführung**
  Falls Transitionen, welche Aktionen darstellen, feuern können, so wird dies dem *Transaction System* mitgeteilt. Dieses System entscheidet, ob es sich um eine Aktion handelt, die auf der Datenbank ausgeführt werden muss, oder um eine Aktion, die datenbankunabhängig (z.B. Erhöhung einer Schleifenvariablen) ausgeführt werden kann.

  Falls es sich um letztere Aktion handelt, so wird diese an das *Component Execution System* weitergeleitet. Dieses System führt die entsprechende Aktion aus und teilt das Ergebnis mit.

  Falls es sich um eine Datenbankaktion handelt, so entscheidet das Transaktionssystem u.a. aufgrund von Datensperren, ob diese Aktion (oder diese Aktionen) ausgeführt werden kann (können). Die *Rule Simulation* erhält vom *Transaction System* eine Liste von ausgeführten Aktionen mit entsprechenden Parametern zurück.

- **Fristen**
  Bei der Definition einer Regel kann eine *Frist* angegeben werden, bis zu welcher die Regel verarbeitet sein muss. Diese Frist kann durch ein beliebiges Ereignis dargestellt werden. Wenn diese Frist abgelaufen ist, so wird eine *Fristaktion* ausgeführt (vgl. *contingency plan* in [DBC96]). Diese Fristaktion ist eine primitive oder komplexe Regelaktion.

  Die Regelausführung kann sich zum Zeitpunkt des Fristablaufs in drei möglichen Zuständen befinden. Je nach Zustand muss entsprechend gehandelt werden:

  1. *Die Regel wurde noch nicht ausgelöst*
     In diesem Fall wird umgehend die Fristaktion ausgeführt.
  2. *Die Regel wurde bereits verarbeitet*
     Da die Regel die Frist in diesem Fall nicht verletzt hat, muss auch keine Fristaktion ausgeführt werden.
  3. *Die Regel wird gerade verarbeitet*
     Wenn die Regel beim Eintreten der Frist gerade verarbeitet wird, so muss diese Regelverarbeitung abgebrochen und entsprechende Transaktionen zurückgesetzt werden. Anschliessend wird anstelle der Regelaktion die Fristaktion ausgeführt.

  Da der festgelegte Zeitpunkt ein Ereignis ist, wird die Auslösung einer Frist durch die *Complex Event Detection* vorgenommen. Das Fristereignis wird jedoch erst dann in die Ereignismenge aufgenommen, wenn die entsprechende Regel ausgelöst wird. Für jede Instanz einer Regel, die eine *Fristaktion* enthält, existiert somit eine Instanz der Ereignisstruktur, welche

das Fristereignis in der Ereignismenge darstellt. Beim Eintreten dieser Ereignisinstanz wird sie gelöscht und die *Rule Simulation* erhält von der *Complex Event Detection* die Identifikation der fristüberschreitenden Regel sowie der auszuführenden Fristaktion. Die Fristaktion, die getrennt von der Regel als Petrinetz verwaltet wird, muss daraufhin durch die *ARFPN Generation* instanziiert und durch die *Rule Simulation* verarbeitet werden.

- **Instanzorientierte Auslösung**
  Die Auslösung von instanzorientierten Regeln stellt einen Spezialfall bei der Regelverarbeitung dar. Die *Rule Simulation* (RS) teilt dem *Transaction System* (TS) mit, dass für eine bestimmte Aktion instanzorientierte Ereignisse (*pre-inst...*, *post-inst...*) existieren. Das TS führt nun ein *retrieve* auf der Datenbank aus, um herauszufinden, welche und wieviele Datensätze von der Aktion betroffen sind. Die Anzahl der Datensätze wird daraufhin der *Rule Simulation* mitgeteilt.

  Die RS muss die Transition für die Aktion sowie alle *pre-inst...* und *post-inst...* Transitionen sooft wiederholt feuern, wie ihr dies vom TS mitgeteilt wurde. Beim Feuern einer *pre-inst...* oder *post-inst...* Transition werden jedesmal neue Regelinstanzen in das ARFPN eingefügt und verarbeitet. Beim Feuern der Transition, welche die Aktion repräsentiert, muss das TS die entsprechende Aktion "instanzorientiert ausführen", d.h einen einzigen Datensatz verarbeiten.

  Falls beim Feuern der *pre-inst...* und *post-inst...* Transitionen Regelteile mit Kopplungsmodus *immediate* ausgelöst werden, so führt nach der *ARFPN Generation* jeweils eine Kante von der Regelinstanz zur Transition nach der Ereignisauslösung. Dies ist nach einer pre-inst-Auslösung die Aktionstransition und nach einer post-inst-Auslösung die *end-inst* Transition (vgl. Abbildung 8.16). Bei jedem wiederholten Feuern dieser Transitionen müssen die Verbindungen, die im vorderen Durchgang entstanden sind, gelöscht werden, da über diese "alten" Verbindungen keine Marke mehr transportiert werden und somit die Transitionen nicht mehr feuern könnten.

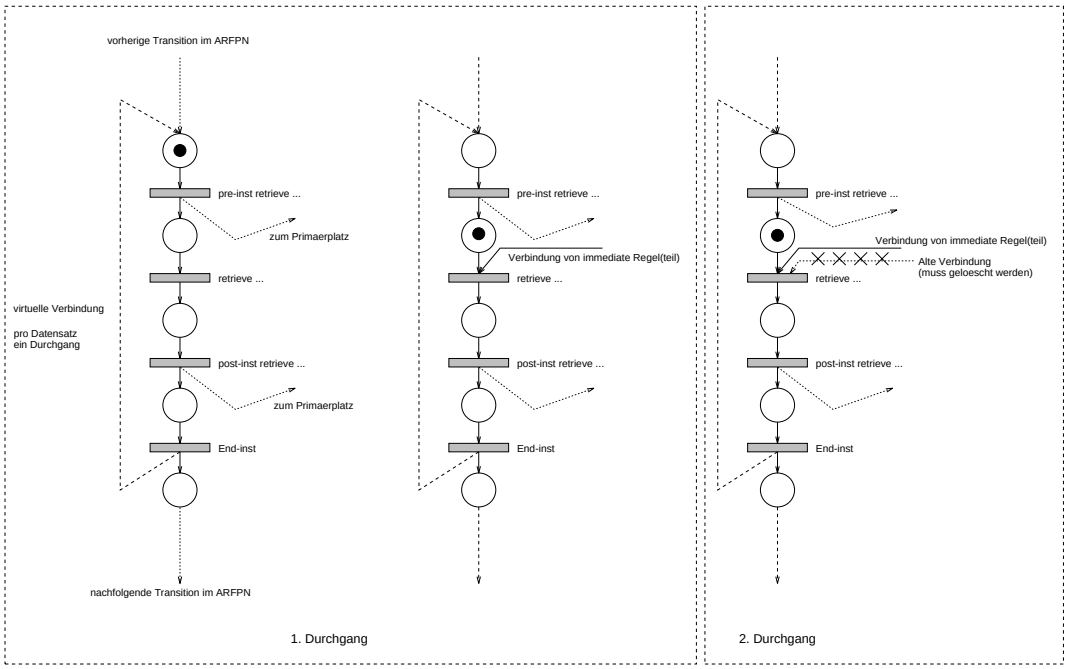

Abbildung 8.16: Instanzorientierte Ausführung und Auslösung

Abbildung 8.16 zeigt den schematischen Ablauf einer instanzorientierten Ausführung eines Teiles des ARFPN (*retrieve*) anhand von drei Schnappschüssen. Es wird davon ausgegangen, dass in der Regelmenge sowohl *pre-inst...* als auch *post-inst...* Ereignisse existieren. Das erste ARFPN zeigt die Verarbeitung im ersten Durchgang *vor* der Ereignisauslösung. Das zweite ARFPN stellt die Situation im ersten Durchgang dar, nachdem die instanzorientierte Regel ausgelöst und von der *ARFPN Generation* eingefügt wurde. Das dritte ARFPN zeigt die Situation im *zweiten* Durchgang, wiederum nachdem die instanzorientierte Regel ausgelöst und von der *ARFPN Generation* eingefügt wurde. Zu diesem Zeitpunkt existieren folglich *zwei* Regelinstanzen, die am ARFPN angehängt sind. Damit nun die Transition *retrieve* feuern kann, muss die Verbindung aus dem vorderen Durchgang (Verbindung ARFPN zur ersten Regelinstanz) gelöscht werden, da über diese Verbindung keine Token mehr transportiert werden.

- **Abort of transaction**
  Falls das *Transaction System* einen Transaktionsabbruch signalisiert, so muss die entsprechende Transaktion und deren Auswirkungen zurückgesetzt werden. Auf der Datenbankebene ist dies Aufgabe des *Transaction Systems*, auf der Stufe der ARFPN muss dies die *Rule Simulation* übernehmen. Das Petrinetz wird in den Zustand vor Eintritt der Transaktion zurückgesetzt. Dazu wird die entsprechende Momentaufnahme des Petrinetzes, die ja bei jedem Feuern einer *bot* oder *Pseudo_bot* Transition angelegt wird, zu Hilfe genommen. In dieser Momentaufnahme stehen sämtliche Plätze, die Token enthalten. Die *Rule Simulation* wird nun das Petrinetz in diesen Zustand zurückversetzen. Daraufhin muss der Benutzer entscheiden, ob die Transaktion noch einmal gestartet oder ob sie übersprungen werden soll. Je nach Antwort des Benutzers, ergeben sich für die *Rule Simulation* unterschiedliche Aufgaben.

  - Wenn die Transaktion wiederholt werden muss, so wird die Verarbeitung an der Stelle weiterfahren, an die sie zurückgesetzt wurde.
  - Falls die Transaktion übersprungen werden soll, so muss die *Rule Simulation* die entsprechende *cot* Transition im Netz finden, die Marke vor der *bot* Transition entfernen und in den Platz nach der *cot* Transition legen. Danach kann die Verarbeitung weitergehen.

Kapitel 9

Zusammenfassung und Ausblick

In dieser Arbeit werden Konzepte und Lösungsvorschläge für Teile einer aktiven Schicht vorgestellt. Diese aktive Schicht wird am Institut für Wirtschaftsinformatik, Abteilung Information Engineering der Universität Bern entwickelt. Kernpunkte der Konzeption von ALFRED (**A**ctive **L**ayer **F**or **R**ule **E**xecution in **D**atabase Systems) sind unter anderem:

- *Datenbankunabhängigkeit*
  Mit Hilfe der gewählten Schichtarchitektur kann ALFRED auf ein prinzipiell beliebiges Datenbanksystem aufgesetzt werden und erweitert dieses um Funktionalitäten, die es ermöglichen, aktives Verhalten zu realisieren. Somit besteht die Möglichkeit, die heute in Unternehmen eingesetzten Datenbanksysteme um aktive Mechanismen zu erweitern.
- *Branchenunabhängigkeit*
  Durch die umfassende Regelsprache von ALFRED ist es möglich, Regeln aus Unternehmen in prinzipiell beliebigen Branchen im System abzubilden.

Um aktives Verhalten zu realisieren, müssen vorab Regeln im ADBS dargestellt werden können. In ALFRED geschieht dies mittels Action Rule Flow Petri Net (ARFPN), die eine etwas modifizierte Form von gefärbten Petrinetzen darstellen. In dieser Arbeit werden Strukturen von ARFPN definiert, mit deren Hilfe sämtliche Regelkomponenten und ausführungsbezogene Regeleigenschaften modelliert werden können. Dazu gehören:

- *Modellierung der Ereignisse*
  Mit Hilfe eines ARFPN können sowohl primitive als auch komplexe Ereignisse modelliert werden. Primitive Ereignisse werden im ARFPN durch einen Platz repräsentiert. Komplexe Ereignisse werden aus primitiven und/oder komplexen Ereignissen mittels Operatoren gebildet. Mögliche Operatoren sind logische Operatoren, Intervalloperatoren, Wiederholungsoperatoren, Sequenzoperatoren und Zeitoperatoren, die ebenfalls mit Hilfe von ARFPN modelliert werden. Zusätzlich können in den Petrinetzen auch Parameterkontexte, Parameter und Parameteroperatoren berücksichtigt werden. Parameterkontexte und -operatoren werden im ARFPN durch Kantenausdrücke realisiert, Parameterwerte werden in den Marken gespeichert.

  Um mehrstufige komplexe Ereignisse darstellen zu können, wird eine Bildungsvorschrift definiert, mit deren Hilfe Ereignisoperatoren so kombiniert werden können, dass die resultierende Semantik korrekt ist.
- *Modellierung der Bedingungen*
  Primitive Bedingungen werden im ARFPN durch Transitionen repräsentiert, in denen die

notwendigen Informationen für die Bedingungsauswertung gespeichert sind. Komplexe Bedingungen werden in primitive Bedingungen zerlegt und mit Hilfe von entsprechenden Strukturen modelliert. Dazu werden Konstrukte wie *Sequenz, Auswahl* und *Wiederholung* im Petrinetz definiert.

- *Modellierung der Aktionen*
  Primitive Aktionen werden im ARFPN als Transitionen repräsentiert. Die notwendigen Informationen für die Aktionsausführung sind in diesen Transitionen gespeichert. Komplexe Aktionen werden, analog zu den Bedingungen, in primitive Aktionen zerlegt und mittels den Strukturen *Sequenz, Auswahl* und *Wiederholung* modelliert.

- *Modellierung der ausführungsbezogenen Regeleigenschaften*
  Die Struktur der ARFPN erlaubt die Darstellung und Berücksichtigung von Kopplungsmodi und Prioritäten. Ebenso ist es möglich, verschiedene Auslösungszeitpunkte (*pre* und *post*) und verschiedene Auslösungsgranularitäten (*set* und *inst*) zu berücksichtigen.

Eine Regelmenge, die durch unterschiedliche Benutzer definiert worden ist, kann schnell unübersichtlich werden. Aus diesem Grund werden in ALFRED Analysen unterstützt. In dieser Arbeit werden Konzepte vorgestellt, mit deren Hilfe Zyklen in der Regelmenge erkannt und untersucht werden können. Anhand von bestimmten Kriterien kann statisch festgestellt werden, ob die gefundenen Zyklen terminieren. Wenn dies der Fall ist, so kann die Terminierung der Regelverarbeitung garantiert werden. Durch die konsequente Verwendung eines einzigen Modells (ARFPN) in ALFRED, wird die Zyklenanalyse einfacher. Im ARFPN sind alle für die Analyse notwendigen Informationen (wie Bedingungen und Aktionen) integriert, und es müssen keine zusätzlichen Strukturen, wie z.B. ein Auslösegraph, eingeführt werden.

Um aktives Verhalten zu realisieren, muss ein ADBS Regeln verarbeiten können. In ALFRED geschieht diese Verarbeitung auf der Basis von ARFPN, wodurch auch parallele Aktionsausführungen ermöglicht werden. Jeder Benutzerbefehl wird durch das Benutzersystem in ein ARFPN umgewandelt. Die Verarbeitung dieses Befehls wird durch einen Algorithmus realisiert, der das Petrinetz durchläuft. Während der Verarbeitung werden diejenigen Regeln integriert, die durch Aktionen ausgelöst wurden. Folgende Aufgaben müssen während der Verarbeitung von Regeln erledigt werden:

- *Erkennung von Ereignissen*
  Die Erkennung von Ereignissen wird durch eine spezielles Subsystem auf einem ARFPN durchgeführt. Die Ereignisstrukturen sowie die Parameterkontexte und die Schaltregeln für das ARFPN geben die Semantik der komplexen Ereignisse vor. Bei der Erkennung werden Parameter gebunden und in den Token des ARFPN gespeichert. Der Algorithmus im Ereigniserkennungssystem hat die Aufgabe, die Petrinetzstruktur im Ereignisteil zu durchlaufen und ausgelöste Regeln zu erkennen.

- *Auswertung von Bedingungen*
  Primitive Bedingungen werden als Transitionen im ARFPN dargestellt. Beim Durchlauf über diese Transitionen wird die Bedingung ausgewertet. Das Resultat der Bedingungsauswertung wird in den Token gespeichert.

- *Ausführung von Aktionen*
  Primitive Aktionen werden als Transitionen im ARFPN dargestellt. Beim Verarbeiten dieser Transitionen werden die Aktionen ausgeführt und notwendige Informationen in den Marken gespeichert.

- *Integration von Regeln*
  Regeln, die durch Benutzerbefehle oder Regelaktionen ausgelöst werden, müssen in die Verarbeitung integriert werden. Diese Aufgabe wird durch das Subsystem *ARFPN Generation*

erfüllt, indem die entsprechenden Regeln instanziiert und in das bestehende Petrinetz integriert werden. Bei dieser Integration sind u.a. Kopplungsmodi und Prioritäten zu berücksichtigen.

- *Schrittweise Verarbeitung und Simulation*
  Durch die Aufspaltung der komplexen Bedingungen und komplexen Aktionen in "atomare" Bestandteile, ist es möglich, die Regeln schrittweise auf einem ARFPN zu verarbeiten. Ebenso kann der Benutzer in ALFRED eine schrittweise Simulation von Abläufen durchführen. Dies wird durch das Transaktionssystem ermöglicht, welches bei der Simulation die Daten nicht dauerhaft auf die Datenbank schreibt. Mit Hilfe der Simulation kann der Benutzer feststellen, ob das realisierte aktive Verhalten mit dem gewünschten aktiven Verhalten übereinstimmt.

Die in dieser Arbeit entwickelten Konzepte werden zur Zeit in einem Prototyp von ALFRED umgesetzt. Mit Hilfe von "intelligenten" Datenstrukturen und Algorithmen wird versucht, die Schichtarchitektur von ALFRED effizient zu realisieren. Einige Probleme wurden in dieser Arbeit erkannt, konnten aber in diesem Rahmen nicht gelöst werden. So müssen u.a. folgende Punkte weiter untersucht werden:

- *Mehrstufige Parameterkontexte*
  Bei der Bildung von mehrstufigen Ereignisoperatoren wurde auf die Verwendung von Parameterkontexten verzichtet. Da jeder Ereignisoperator in ALFRED einen eigenen Parameterkontext besitzt, muss untersucht werden, welche Semantik für das zusammengesetzte komplexe Ereignis resultiert. Dabei muss berücksichtigt werden, welchen Einfluss Parameterkontexte auf andere Parameterkontexte haben und ob die möglichen Kombinationen sinnvoll sind.

- *Dynamische Zyklenanalyse*
  In dieser Arbeit wird eine statische Zyklenanalyse vorgestellt. Diese erlaubt nur sehr begrenzte Aussagen über die Terminierung von Zyklen. Um möglichst alle Zyklen im System zu erkennen und zu analysieren, müsste zusätzlich eine dynamische Zyklenanalyse erfolgen. In dieser würden Zyklen überwacht und wenn nötig abgebrochen werden. Ein möglicher Ansatz für die dynamische Überwachung von Zyklen wird z.B. in [BCP95b] beschrieben.

  Die Überprüfung der Terminierung in grossen Regelmengen ist unabdingbar. Es ist jedoch berechtigt zu fragen, ob die Analyse vollständig durch das Computersystem durchgeführt werden muss. Wenn ja, so müssten für diese Analysezwecke Komponenten der künstlichen Intelligenz eingesetzt werden. Wenn nein, so muss das System dem Benutzer notwendige Informationen liefern können, mit deren Hilfe er entscheiden kann, ob die Verarbeitung terminiert oder ob die Regelverarbeitung abgebrochen werden soll.

- *Simultaner Ereigniseintritt*
  In den hier vorliegenden Konzepten wird davon ausgegangen, dass zu jedem Zeitpunkt nur ein einziges primitives Ereignis eintreten kann. In Mehrprozessorsystemen und bei Client/Server Architekturen können jedoch mehrere Ereignisse gleichzeitig eintreten. Für die korrekte Realisierung dieser Gleichzeitigkeit in ALFRED müssten zusätzliche Konzepte entwickelt werden.

- *Datensperren*
  Datensperren (*locks*) werden in ALFRED im Transaktionssystem verwaltet. Da sich Petrinetze für die Modellierung von locking-Mechanismen gut eignen, könnte die Verwaltung von Datensperren ebenfalls in das ARFPN integriert werden.

Die heutige Weltbevölkerung wandelt sich immer mehr zu einer Informationsgesellschaft. Damit verbunden ist das Bedürfnis, auf verschiedenste Daten überall und schnell zugreifen zu können. Um diesem Bedürfnis gerecht zu werden, bedarf es universell einsetzbarer Datenbanksysteme, welche die Benutzer optimal unterstützen können. Die DBS werden in Zukunft nicht nur mehr als "Datenspeicher" eingesetzt, sondern müssen den Anforderungen der Benuzter gerecht werden. Dazu

gehört unter anderem auch die Realisierung von aktivem Verhalten, mit dessen Hilfe Vorgänge automatisiert werden können. Die Entwicklung im Bereich der kommerziell verfügbaren Datenbanksysteme bestätigt diesen Trend. So wird es z.B. im kommenden SQL-3 Standard möglich sein, Regeln, Trigger und komplexere Integritätsbedingungen zu definieren.

Einen grossen Einfluss wird auch die Globalisierung der Informationen durch die weltweite Vernetzung mit sich bringen. Um den zukünftigen Anforderungen auf diesem Gebiet gerecht zu werden, bedarf es vermutlich noch einiger Forschungsarbeit auf dem Gebiet der Datenbankssyteme. Mit den in dieser Arbeit vorgestellten Konzepten für ALFRED wird ein möglicher Forschungsansatz aufgezeigt.

Anhang A

Regelsyntax

A.1 Text, Strings und Zeichenketten

```
<lingual>     ::= <string>

<text>        ::= <ident> <separator> [ <text> ]

<separator>   ::= ´ ´ | ´,´ | ´.´ | ´!´ | ´?´

<prefix>      ::= ´ERROR´ | ´WARNING´ | ´HINT´

<string>      ::= ´"´ <ident> ´"´

<ident>       ::= <letter> [ <ident_rest> ]

<ident_rest>  ::= <letter_rest> [ <ident_rest> ]

<letter_rest> ::= <letter> | ´-´

<letter>      ::= ´A´ | ´B´ | ... | ´Z´ | ´a´ | ´b´ | ... | ´z´
```

A.2 Integer- und Dezimalzahlen

```
<value>    ::= <int_real> | <string>

<int_real> ::= [ <add_opr> ] ( <integer> | <real> )

<real>     ::= <integer> ´.´ <integer>

<numeral>  ::= <integer>

<integer>  ::= <digit> [ <integer> ]

<digit>    ::= ´0´ | ´1´ | ... | ´9´
```

A.3 Datum, Zeit und Tage

```
<date>     ::= <day> ´/´ <month> ´/´ <year>

<time>     ::= <hour> ´:´ <minute> ´:´ <second>

<day>      ::= ´1´ | ´2´ | ... | ´31´

<month>    ::= ´1´ | ´2´ | ... | ´12´

<year>     ::= ´1996´ | ´1997´ | ... | ´2100´

<hour>     ::= ´0´ | ´1´ | ... | ´23´

<minute>   ::= ´0´ | ´1´ | ... | ´59´

<second>   ::= ´0´ | ´1´ | ... | ´59´

<week_day> ::= ´Mon´ | ´Monday´    |
               ´Tue´ | ´Tuesday´   |
               ´Wed´ | ´Wednesday´ |
               ´Thu´ | ´Thursday´  |
               ´Fri´ | ´Friday´    |
               ´Sat´ | ´Saturday´  |
               ´Sun´ | ´Sunday´

<calendar> ::= ´Days´          |
               ´Weeks´         |
               ´Months´        |
               ´Quarter_years´ |
               ´Half_years´    |
               ´Years´
```

A.4 Operatoren

```
<cmp_opr>  ::= ´<´ | ´<=´ | ´=´ | ´>=´ | ´>´ | ´<>´

<bool_opr> ::= ´and´ | ´or´

<add_opr>  ::= ´+´ | ´-´

<mul_opr>  ::= ´*´ | ´/´
```

A.5 Listen

```
<rule_event_list> ::= <rule_event>          [ ´,´ <rule_event_list> ]

<att_expr_list>   ::= <ident> ´=´ <expr>    [ ´,´ <att_expr_list> ]

<ident_list>      ::= <ident>     [ ´,´ <ident_list> ]

<obj_list>        ::= <ident>     [ [ ´as´ <ident> ] ´,´ <obj_list> ]

<obj_att_list>    ::= <ident> ´.´ <ident> [ ´,´ <obj_att_list> ]
```

```
<val_list>        ::= <value>     [ ´,´ <val_list> ]

<att_list>        ::= <ident_list>

<par_list>        ::= <ident_list>
```

A.6 Bedingungsvarianten

```
<search_cond> ::= <search_cond> <bool_opr> <search_cond> |
                  [ ´not´ ] ´(´ <search_cond> ´)´        |
                  <predicate>                            |
                  <pred_query>

<predicate>   ::= <expr> <cmp_opr> <expr>

<pred_query>  ::= <var> ( ´in´ | ´all´ <cmp_opr> ) ´(´ <val_list> | <query> ´)´

<query>       ::= ´retrieve´ ( <att_list> | ´*´ ) ´from´ <obj_list>
                     ´where´ <search_cond>

<expr>        ::= <factor> <add_opr> <expr> | <factor>

<factor>      ::= <operand> <mul_opr> <factor> | <operand>

<operand>     ::= <value> | <var> | ´(´ <expr> ´)´
```

A.7 Variable

```
<var>          ::= <var> ´.´ <ident> | <var> ´[´ <integer> ´]´ | <ident>
```

A.8 Syntax der Transaktionen

```
<transaction>             ::= ´begin´ ´of´ ´transaction´
                                 <transaction_block>
                              ´commit´ ´of´ ´transaction´ ´;´

<transaction_block>       ::= <transaction_block_cmd> [ <transaction_block> ]

<transaction_block_cmd>   ::= <retrieve_cmd>         |
                              <insert_cmd>           |
                              <update_cmd>           |
                              <delete_cmd>           |
                              <message_cmd>          |
                              <raise_cmd>            |
                              <enable_rule_cmd>      |
                              <disable_rule_cmd>     |
                              <exec_proc_cmd>        |
                              <abort_cmd>

<abort_cmd>               ::= ´abort´ ´;´
```

A.9 Syntax der ALFRED-DML

```
<retrieve_cmd>         ::= ´retrieve´ ´(´ <obj_att_list> ´)´
                           [ ´where´ <search_cond> ] ´;´

<insert_cmd>           ::= ´insert´ ´in´ <ident> [ ´(´ <att_list> ´)´ ]
                           ´values´ ´(´ <val_list> ´)´ ´;´

<update_cmd>           ::= ´update´ ´on´ <ident>
                           ´set´ ´(´ <att_expr_list> ´)´
                           [ ´where´ <search_cond> ] ´;´

<delete_cmd>           ::= ´delete´ ´from´ <ident>
                           [ ´where´ <search_cond> ] ´;´

<message_cmd>          ::= ´message´ ´(´ [ <prefix> ´:´ ] <text> ´)´ ´;´

<raise_cmd>            ::= ´raise´ ´abstract´ ´event´ <ident> ´;´

<enable_rule_cmd>      ::= ´enable´ ´rule´ <ident>
                           ´set´ ´rule´ ´state´
                           ´to´ ´active´ [<rule_tmp_state> ] ´;´

<disable_rule_cmd>     ::= ´disable´ ´rule´ <ident>
                           ´set´ ´rule´ ´state´
                           ´to´ ´deactive´ [ <rule_tmp_state> ] ´;´

<exec_proc_cmd>        ::= ´execute´ ´stored´ ´procedure´ <ident>
                           ´with´ ´parameter´ ´(´ <val_list> ´)´ ´;´
```

A.10 Syntax der Regelstruktur

```
<rule>           ::=  ´rule´ <ident>
                       <rule_general>
                       <rule_structure> ´;´

<rule_general>   ::= [ ´belongs´ ´to´                    <rule_set_list>  ]
                     [ ´has´ ´priority´                  <rule_order>     ]
                     ´is´                                <rule_state>
                     [ ´has´ ´to´ ´be´ ´executed´ ´until´ <rule_timing>
                       ´with´ ´contingency´ ´plan´       <rule_cont_plan> ]

<rule_set_list>  ::= <rule_set> [ ´,´ <rule_set_list> ]

<rule_set>       ::= <ident>

<rule_order>     ::= <integer>

<rule_state>     ::= ( ´active´ | ´deactive´ ) [ <rule_tmp_state> ]

<rule_tmp_state> ::= ´between´ ( ´now´ | <rule_event> ) ´and´ <rule_event>

<rule_timing>    ::= <rule_event>

<rule_cont_plan> ::= <rule_action>
```

```
<rule_structure> ::= ´on´ ´event´                    <event>
                     ´check´ ´condition              <condition>
                     ´execute´ ´true_action´         <action>
                     ´or´      ´false_action´        <action>
                   [ ´with´ ´couplings´ ´:´
                     ´event - condition´ ´:´         <rule_coupling>
                     ´condition - true_action´ ´:´   <rule_coupling>
                     ´condition - false_action´ ´:´ <rule_coupling> ] |

                   ´on´ ´event´                      <event>
                     ´check´ ´condition´             <condition>
                     ´execute´ ´action´              <action>
                   [ ´with´ ´couplings´ ´:´
                     ´event - condition´ ´:´         <rule_coupling>
                     ´condition - action´ ´:´        <rule_coupling> ] |

                   ´on´ ´event´                      <event>
                     ´execute´ ´action´              <action>
                   [ ´with´ ´coupling´ ´:´
                     ´event - action´ ´:´            <rule_coupling> ] |

                   ´check´ ´condition´               <condition>
                     ´execute´ ´true_action´         <action>
                     ´or´      ´false_action´        <action>
                   [ ´with´ ´couplings´ ´:´
                     ´condition - true_action´ ´:´   <rule_coupling>
                     ´condition - false_action´ ´:´ <rule_coupling> ] |

                   ´check´ ´condition´               <condition>
                     ´execute´ ´action´              <action>
                   [ ´with´ ´coupling´ ´:´
                     ´condition - action´ ´:´        <rule_coupling> ]

<rule_coupling>  ::= ´immediate´                        |
                     ´deferred´                         |
                     ´detached´ ´sequential´ ´dependent´ |
                     ´detached´ ´parallel´   ´dependent´ |
                     ´detached´ ´exlusive´   ´dependent´ |
                     ´detached´ ´independent´
```

A.11 Syntax der Ereigniskomponenten

```
<event>            ::= <rule_event> [ <param_context> ] | <lingual>

<param_context>    ::= ´recent´       |
                       ´chronological´ |
                       ´continuous´    |
                       ´cumulative´

<rule_event>       ::= ´(´ <rule_event> ´)´ |
                      <primitive_event>    |
                       <complex_event>

<primitive_event>  ::= <data_base_event>   |
                       <data_def_event>    |
```

```
                        <data_mpl_event>    |
                        <cancel_event>      |
                        <transaction_event> |
                        <procedure_event>   |
                        <program_event>     |
                        <emb_program_event> |
                        <application_event> |
                        <simulation_event>  |
                        <exec_ta_event>     |
                        <time_event>        |
                        <abstract_event>

<time_pnt>          ::= ´pre´ | ´post´

<data_base_event>   ::= <time_pnt> <data_base_cmd>

<data_base_cmd>     ::= ( ´create´  ´database´         ) |
                        ( ´destroy´ ´database´ <ident> )

<data_def_event>    ::= <time_pnt> <data_def_cmd>

<data_def_cmd>      ::= (   ´create´ ´entity´                                ) |
                        (   ´create´ ´view´                                  ) |
                        (   ´create´ ´rule´                                  ) |
                        (   ´create´ ´rule_set´                              ) |
                        (   ´create´ ´user´                                  ) |
                        (   ´create´ ´stored procedure´                      ) |
                        (   ´create´ ´program´                               ) |
                        (   ´create´ ´embedded´ ´program´                    ) |
                        (   ´create´ ´application´                           ) |
                        (   ´create´ ´simulation´                            ) |
                        (   ´create´ ´transaction´                           ) |
                        ( ( ´alter´ | ´destroy´ ) ´entity´ <ident>            ) |
                        (             ´destroy´   ´view´ <ident>             ) |
                        ( ( ´alter´ | ´destroy´ ) ´rule´ <ident>              ) |
                        ( ( ´alter´ | ´destroy´ ) ´rule_set´ <ident>          ) |
                        ( ( ´alter´ | ´destroy´ ) ´user´ <ident>              ) |
                        ( ( ´alter´ | ´destroy´ ) ´stored´ procedure´ <ident>  ) |
                        ( ( ´alter´ | ´destroy´ ) ´program´ <ident>           ) |
                        ( ( ´alter´ | ´destroy´ ) ´embedded´ ´program´ <ident> ) |
                        ( ( ´alter´ | ´destroy´ ) ´application´ <ident>       ) |
                        ( ( ´alter´ | ´destroy´ ) ´simulation´ <ident>        ) |
                        ( ( ´alter´ | ´destroy´ ) ´transaction´ <ident>       ) |
                        ( ( ´grant´ | ´revoke´  ) ´object´ ´access´          ) |
                        ( ( ´grant´ | ´revoke´  ) ´file´ ´access´            ) |
                        ( ( ´grant´ | ´revoke´  ) ´privilege´                )

<transaction_event> ::= <time_pnt> <transaction_cmd>

<transaction_cmd>   ::= ´begin´  ´of´ ´transaction´ |
                        ´end´    ´of´ ´transaction´ |
                        ´commit´ ´of´ ´transaction´

<data_mlp_event>    ::= <time_pnt> <data_mlp_cmd> ( ´set´ | ´instance´ )

<data_mlp_cmd>      ::= ( ´retrieve´ ´from´ <ident> ) |
                        ( ´insert´   ´in´   <ident> ) |
```

```
                    ( ´update´   ´on´   <ident> ) |
                    ( ´delete´   ´from´ <ident> )

<cancel_event>      ::= <time_pnt> ´cancel´ ´operation´

<procedure_event>   ::= <time_pnt> ´execute´ ´stored´ ´procedure´ <ident>

<program_event>     ::= <time_pnt> ´execute´ ´program´ <ident>

<emb_program_event> ::= <time_pnt> ´execute´ ´embedded´ ´program´ <ident>

<application_event> ::= <time_pnt> ´execute´ ´application´ <ident>

<simulation_event>  ::= <time_pnt> ´execute´ ´simulation´ <ident>

<exec_ta_event>     ::= <time_pnt> ´execute´ ´transaction´ <ident>

<time_event>        ::= [ <date> ´@´ ] <time>

<abstract_event>    ::= ´abstract´ ´event´ <ident>


<complex_event>     ::= <boolean_opr>    |
                        <choice_opr>     |
                        <sequence_opr>   |
                        <repetition_opr> |
                        <time_opr>       |
                        <interval_opr>

<boolean_opr>       ::= <rule_event> <bool_opr> <rule_event>

<choice_opr>        ::= ´weak´   ´choice´ <numeral> ´(´ <rule_event_list> ´)´ |
                        ´strong´ ´choice´ <numeral> ´(´ <rule_event_list> ´)´

<sequence_opr>      ::= ´weak´   ´sequence´ ´(´ <rule_event_list> ´)´ |
                        ´strong´ ´sequence´ ´(´ <rule_event_list> ´)´

<repetiton_opr>     ::= ´every´ <numeral>         ´occurence´  ´of´ <rule_event> |
                        ´every´ ´after´ <numeral> ´occurences´ ´of´ <rule_event>

<time_opr>          ::= ´timespan´ ( <time> | <numeral> <calendar> )
                          ´after´ <rule_event>                          |
                        ´every´ <time> <start_pnt>                      |
                        ´every´ [ <numeral> ] <calendar> <start_pnt>    |
                        ´every´ [ <numeral> ] <week_day> ´at time´
                           <time> <start_pnt>                           |
                        ´every´ [ <numeral> ] ( ´day´ | <week_day> )
                           ´in´ ´month´ ´at´ ´time´ <time> <start_pnt>

<start_pnt>         ::= ´beginning´ ´at´ ( ´now´ | <rule_event> )

<interval_opr>      ::= [ ´not´ ]       <rule_event> ´in´ <interval> |
                        ´the´ ´first´   <rule_event> ´in´ <interval> |
                        ´the´ ´last´    <rule_event> ´in´ <interval> |
                        ´the´ <numeral> <rule_event> ´in´ <interval>

<interval>          ::= ´interval´ ´(´ <rule_event> ´;´ <rule_event> ´)´
```

A.12 Syntax der Bedingungskomponenten

```
<condition>           ::= <rule_condition> | <lingual>

<rule_condition>      ::= [ ´not´ ] ´(´ <rule_condition> ´)´ |
                          <primitive_condition>             |
                          <complex_condition>

<primitive_condition> ::= <predicate>     |
                          <query>         |
                          <pred_query>    |
                          <bool_function> |
                          ´true´          |
                          ´false´

<bool_function>       ::= ´function´ <ident> ´(´ [ <par_list> ] ´)´

<complex_condition>   ::= <rule_condition> <bool_opr> <rule_condition>
```

A.13 Syntax der Aktionskomponenten

```
<action>           ::= <rule_action> | <lingual>

<rule_action>      ::= [ ´instead´ ´do´ ]
                         ( <primitive_action> | <complex_action> )

<primitive_action> ::= <database_cmd> | <lingual>

<database_cmd>     ::= <retrieve_cmd>         |
                       <insert_cmd>           |
                       <update_cmd>           |
                       <delete_cmd>           |
                       <abort_cmd>            |
                       <cancel_cmd>           |
                       <enable_rule_cmd>      |
                       <disable_rule_cmd>     |
                       <message_cmd>          |
                       <raise_cmd>

<cancel_cmd>       ::= ´cancel´ ´operation´;


<complex_action>   ::= ´begin´ <database_block> ´end´ ´action´ |
                       <proc_cmd>                               |
                       <program_cmd>                            |
                       <emb_program_cmd>                        |
                       <application_cmd>

<database_block>   ::= <sequential_block> | <parallel_block>

<sequential_block> ::= ( <database_cmd> | <transaction> ) <db_cmd_block>

<db_cmd_block>     ::= ( <database_cmd> | <transaction> ) [ <sequential_block> ]

<parallel_block>   ::= ´begin´ ´parallel´
```

```
                        <database_block>
                      ´end´ ´parallel´ ´;´

<proc_cmd>          ::= <exec_proc_cmd>

<program_cmd>       ::= ´program´ <ident> ´(´ ´)´ ´;´

<emb_program_cmd>   ::= ´embedded´ ´program´ <ident> ´(´ ´)´ ´;´

<application_cmd>   ::= ´application´ <ident> ´;´
```

Anhang B

Verbindungsalgorithmus für komplexe Ereignisse

```
// Recursive algorithm for event connection

////////////////////////////////
// Datastructure for connection
////////////////////////////////
record node
   pl = list of places
   tr = list of transitions
end record


////////////////////////////
// recursive function connect
////////////////////////////
node connect(X)
{
   if(X == primitive)
   {  // recursion base
      // this is a start-event
      node.pl = X
      // now look for start-transition
      if(next transition to X is unique)
         node.tr = next transition to X
      else
         node.tr = next transition to X with H place in front
      return(node)
   }

   else if(X == interval)
   {  // we first make coordinator and resolver for this interval
      // (by recursion), then we give back the start-events (places in
      // begin-part of interval)
      node1 = connect(X.begin)
      node2 = connect(X.end)
```

```
      node3 = connect(X.interval)

      nodeX = merge(node2,node3)

      // making coordinator and resolver

      foreach(elem in nodeX)
      {
         if(coordinator from H to elem.pl does not exist)
         {
            make coordinator from H to elem.pl

            if(X == 'last in')
            {  // resolver for 'last in' is a special case
               make 'last in'-resolver
            }
            else
            {
               make resolver to elem.tr if it does not exist
            }
         }
         mark coordinator from H to elem.pl
      }
      foreach(coordinator from H not marked)
      {  // Here we have the surplus 'built-in' coordinators
         delete coordinator
      }
      return(node1)
   }
   else if((X == and) OR (X == or))
   {
      // look for primitive events (start-events)
      node1 = connect(X.left)
      node2 = connect(X.right)
      nodeX = merge(node1,node2)
      return(nodeX)
   }
   else if(X == strong sequence)
   {  // we first make coordinator and resolver for this
      // sequence (by recursion), then we give back the
      // necessary start-events (places in first part of strong
      // sequence)
      node1 = connect(X.first)
      node2 = connect(X.second)

      foreach(elem in node2)
      {
         if(coordinator to elem.pl does not exist)
         {
            make coordinator from H to elem.pl

            make resolver to elem.tr
```

```
         }
         mark coordinator from H to elem.pl
      }
      foreach(coordinator from H not marked)
      {  // Here we have the surplus ´built-in´ coordinators
         delete coordinator
      }
      return(node1)
   }
   else if(X == weak sequence)
   {
      // look for start-events
      node1 = connect(X.first)
      node2 = connect(X.second)
      nodeX = merge(node1,node2)
      return(nodeX)
   }
   else if(X == timespan t after)
   {
      // look for start-events
      node = connect(Event)
      return(node)
   }
   else if(X == every time t)
   {
      if(X == every time t beginning at Event)
      {
         // look for start-events
         node = connect(Event)
         return(node)
      }
      else if(X == every time t beginning at now)
      {
         // there is no start-event
         return()
      }
   }
   else if((X == every n) || (X == every after n))
   {
      // look for start-events
      node = connect(Event)
      return(node)
   }

}
//////////////
// end connect
//////////////
```

Anhang C

Algorithmus für die Zyklenerkennung

```
////////////////////////////////////////////////
// Go through ARFPN with a depth-first-search
//   algorithm, starting from primary places,
//   where the new inserted rule is attached to
// Mark visited nodes
// If we visit a node a second time -> cycle
// There may be more than one cylce
////////////////////////////////////////////////

//////////////////
// Cycledetection
//////////////////

cycledetection(list_of_primary_places ppl)
{
   unmark all nodes in ARFPN

   for each unmarked element in (ppl)
   {
      dfs(element)
   }
}


//////////////////////
// Depth-First-Search
//////////////////////

dfs(node)
{
   if(node == visited)
   {
      // we have found a cycle

      save cycle-information
      inform user
```

```
      return
   }
   else
   {
      mark node as visited
      foreach successor of node
      {
         dfs(successor)
      }
   }
}
```

Anhang D

Transitionsarten in der Regelsimulation

In diesem Anhang werden sämtliche Transitionsarten aufgelistet, die in der Regelsimulation auftreten können. Dabei wird das *Transaction System* mit TS abgekürzt.

D.1 ARFPN

- **Begin_ARFPN**
  - *Zweck:*
    Bezeichnet den Beginn eines ARFPN.
  - *Aktion:*
    TS informieren und Transition feuern.
- **End_ARFPN**
  - *Zweck:*
    Bezeichnet das Ende des ARFPN.
  - *Aktion:*
    TS informieren, Meldung an Benutzer generieren und ARFPN löschen.

D.2 Ereignisauslösung

- **pre-set**...
  - *Zweck:*
    Bezeichnet eine Ereignisauslösung mit Zeitpunkt *pre* und Granularität *set*.
  - *Aktion:*
    TS informieren. Transition feuern und der *Complex Event Detection (CED)* den markierten Primärplatz übergeben. Warten auf die Rückmeldung der CED. Falls Regeln ausgelöst wurden, diese der *ARFPN Generation* mitteilen. Warten, bis die *ARFPN Generation* die Regeln eingefügt hat. Verarbeitung fortsetzen.

- **pre-inst**...
  - *Zweck:*
    Bezeichnet eine Ereignisauslösung mit Zeitpunkt *pre* und Granularität *instance*.
  - *Aktion:*
    Analog zu *pre-set*.... Falls nötig, alte instance-Rückverbindungen löschen.
- **post-set**...
  - *Zweck:*
    Bezeichnet eine Ereignisauslösung mit Zeitpunkt *post* und Granularität *set*.
  - *Aktion:*
    Analog zu *pre-set*....
- **post-inst**...
  - *Zweck:*
    Bezeichnet eine Ereignisauslösung mit Zeitpunkt *post* und Granularität *instance*.
  - *Aktion:*
    Analog zu *pre-inst*....
- **End-inst**
  - *Zweck:*
    Bezeichnet das Ende einer instance-Auslösung.
  - *Aktion:*
    TS informieren. Überprüfe, ob der Befehl, für den ein instanzorientiertes Ereignis existiert, ein weiteres Mal ausgeführt werden muss. Falls ja, so springe zum Platz vor der *pre-inst*... Transition oder zum Platz vor der Transition, die den Befehl darstellt, zurück. Feure diese Transitionen ein wiederholtes Mal. Falls der Befehl nicht mehr ausgeführt werden muss, so feure die *end-instance* Transition und setze die Verarbeitung fort.

D.3 Bedingungen

- **Begin_of_condition**
  - *Zweck:*
    Bezeichnet den Anfang eines Bedingungsteils.
  - *Aktion:*
    TS informieren und Transition feuern.
- **condition**
  - *Zweck:*
    Bezeichnet eine vom Benutzer definierte primitive Bedingung.
  - *Aktion:*
    TS informieren und auf Erlaubnis zum Feuern warten.
- **AND**
  - *Zweck:*
    Bezeichnet die AND-Verknüpfung einer komplexen Bedingung.

  - *Aktion:*
    TS informieren. Überprüfe, ob sämtliche Marken in den Vorplätzen dieser Transition, die zum Feuern benötigt werden, den Wert TRUE haben. Wenn dies der Fall ist, so generiere eine neue Marke mit Wert TRUE, ansonsten eine mit Wert FALSE.

- **OR**
  - *Zweck:*
    Bezeichnet die OR-Verknüpfung einer komplexen Bedingung.
  - *Aktion:*
    TS informieren. Überprüfe, ob mindestens eine Marke in den Vorplätzen dieser Transition den Wert TRUE hat. Ist dies der Fall, so erzeuge eine Marke mit Wert TRUE, ansonsten eine mit Wert FALSE.

- **End_of_condition**
  - *Zweck:*
    Bezeichnet das Ende eines Bedingungsteils.
  - *Aktion:*
    TS informieren. Entscheide aufgrund des Wahrheitswertes der Marke im Eingangsplatz, in welchen Ausgangsplatz (true_out oder false_out) eine Marke gelegt werden soll. Falls an dieser Transition ein spezieller Platz hängt (*always-out*-Platz im Falle von KM immediate), so markiere diesen Platz unabhängig vom Resultat der Bedingungsauswertung.

D.4 Aktionen

- **Begin_of_action**
  - *Zweck:*
    Bezeichnet den Beginn des Aktionsteils.
  - *Aktion:*
    TS informieren und Transition feuern.

- **action**
  - *Zweck:*
    Bezeichnet die vom Benutzer gewünschte Aktion (z.B. `retrieve (*) from Lager`).
  - *Aktion:*
    TS informieren und auf Erlaubnis zum Feuern warten.

- **while-transition**
  - *Zweck:*
    Bezeichnet die Bedingungsauswertung einer Wiederholung (while-Schleife).
  - *Aktion:*
    TS informieren. Aufgrund des Resultates von TS die Marke in den Wahr-Zweig oder den Falsch-Zweig transportieren.

- **if-transition**
  - *Zweck:*
    Bezeichnet die Bedingungsauswertung einer Auswahl (if).

  - *Aktion:*
    TS informieren. Aufgrund des Resultates von TS die Marke in den Wahr-Zweig oder den Falsch-Zweig transportieren.

- **End_of_action**

  - *Zweck:*
    Bezeichnet das Ende des Aktionsteils.
  - *Aktion:*
    TS informieren und Transition feuern.

D.5 Transaktionen

- **Pseudo_bot**

  - *Zweck:*
    Bezeichnet den Beginn einer pseudo-Transaktion.
  - *Aktion:*
    TS informieren, Momentaufnahme des ARFPN und der Ereignismenge machen und Transition feuern.

- **bot**

  - *Zweck:*
    Bezeichnet den Beginn einer durch den Benutzer definierten Transaktion.
  - *Aktion:*
    TS informieren, Momentaufnahme des ARFPN und der Ereignismenge machen und Transition feuern.

- **Pseudo_eot**

  - *Zweck:*
    Bezeichnet das Ende einer pseudo-Transaktion.
  - *Aktion:*
    TS informieren und Transition feuern.

- **eot**

  - *Zweck:*
    Bezeichnet das Ende einer benutzerdefinierten Transaktion.
  - *Aktion:*
    TS informieren und Transition feuern.

- **Pseudo_cot**

  - *Zweck:*
    Bezeichnet das *commit* der pseudo-Transaktion.
  - *Aktion:*
    TS informieren und Transition feuern.

- **cot**
  - *Zweck:*
    Bezeichnet das *commit* einer benutzerdefinierten Transaktion.
  - *Aktion:*
    TS informieren und Transition feuern.

Anhang E

Algorithmus für Complex Event Detection

```
//////////////////////////////////////
// Fires all the transitions in the
// event part as long as possible
//////////////////////////////////////

complex_event_detection(primary_places)
{
   // merge marked places in event part (snapshot) with
   // marked primary_places

   marked_places = merge(primary_places,snapshot)

   // Try to fire transitions starting from
   // marked places

   while there is a place <> ´Ereignisende´ in (marked_places)
   {
      for each transition next to place
      {
         if transition can be fired

         // A transition can be fired here iff
         //   - enough token in input-places
         //   - It´s a transition from event part

         {
            // Fire transition and change marked_places
            // accordingly

            fire(transition,marked_places)
         }
      }
    }
```

```
    // Search for all marked ´Ereignisende´ places

    for each ´Ereignisende´ in (marked_places)
    {
       find corresponding rule

       // If ´Ereignisende´ is marked, then the corresponding
       // rule is triggered

       add rule to triggered_rule_list

       // Here we don´t use ´Ereignisende´ any more
       delete ´Ereignisende´ from marked_places
    }

    // Make a new snapshot of marked places (for next call)

    snapshot = marked_places

    return(triggered_rule_list)
}
```

Anhang F

Algorithmus für Rule Simulation

```
////////////////////////////////////
// Fire all transitions depending on
// their functionality. The necessary
// distinction is done by the
// function fire( )
////////////////////////////////////

rule_simulation(ARFPN)
{
   add first place of ARFPN to marked_places

   // Fire transitions until ´End_ARFPN´ is reached

   while (! end)
   {
      // look for connections between ARFPN and
      // primary places

      for each place in (marked_places)
      {
         for each transition next to place
         {
            if(transition == connection to primary place)
            {
               add transition to connection_list
            }
         }
      }

      // connections with event part must be fired
      // first. Transitions in connection_list are
      // ordered by time (first transition must
      // be fired first, ...).

      if connection_list not empty
      {
```

```
   for each transition in connection_list
   {
      if transition can be fired
      {
         fire(transition,marked_places)

         extract primary_place from marked_places

         delete transition from connection_list

         // start CED

         triggered_rules = complex_event_detection(primary_place)

         // Instantiate all the triggered rules and
         // include them in ARFPN

         ARFPN_generation(triggered_rules,ARFPN)
      }
   }
}
else
{
   // Now we look for all the other fireable transitions

   for each place in (marked_places)
   {
      for each transition next to place
      {
         if transition can be fired
         {
            add transition to fireable_transitions
         }
      }
    }

    // Ask transaction_management which transitions may
    // be fired (-> firing_list)
    // The entries in firing_list are sorted by time (first
    // entry must be fired first, ...)

    transaction_management(fireable_transitions,firing_list)

    foreach entry in (firing_list)
    {
       if(entry->action == abort)
       {
          set back ARFPN

          ask user what to do

          mark necessary places
```

```
               add those places to marked_places
            }
            else
            {
               // fire the transition and add the marked places
               // to marked_places
               // Some transitions have special meanings and
               // firing is more complex, like End-inst,
               // End_of_condition and so on

               fire(entry->transition,marked_places)

               if(entry->transition == ´End_ARFPN´)
               {
                  inform user
                  end = TRUE
               }
            }
         }
      }  // end else

   }  // end while (!end)

}  // end rule_simulation
```

Abbildungsverzeichnis

Literaturverzeichnis

[AWH92] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59 – 68, San Diego, June 1992.

[BBKZ92] A.P. Buchmann, H. Branding, T. Kudrass, and J. Zimmermann. REACH: A REal-Time, ACtive and Heterogeneous Mediator System. *IEEE Bulletin of the Technical Committee in Data Engineering: Special Issue On Active Databases*, 15(1 - 4):44 – 47, 1992.

[BBKZ93] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems*, pages 111 – 126. Springer, London et al., September 1993.

[BCP95a] E. Baralis, S. Ceri, and S. Paraboschi. Improved Rule Analysisby Means of Triggering and Activation Graphs. In T. Sellis, editor, *Rules on Database Systems, Lecture Notes in Computer Science 985*, pages 165 – 181. Springer, Berlin et al., 1995.

[BCP95b] E. Baralis, S. Ceri, and S. Paraboschi. Run-Time Detection of Non-Terminating Active Rule Systems. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Deductive and Object-Oriented Databases (DOOD), Lecture Notes in Computer Science 1013*, pages 38 – 54. Springer, Berlin et al., 1995.

[Ber91] M. Berndtsson. *ACOOD: An Approach To An Active Object Oriented DBMS*. PhD thesis, Department of Computer Science, University of Skövde, 1991.

[BW96] E. Baralis and J. Widom. Better Static Rule Analysis for Active Database Systems. Technical report, Department of Computer Science, Stanford CA, 1996.

[CAM93] S. Chakravarthy, E. Anwar, and L. Maugis. Design and Implementation of Active Capability for an Object-Oriented Database. Technical Report UF-CIS-TR-93-001, Department of Computer and Information Sciences, University of Florida, 1993.

[CM93] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language For Active Databases. Technical Report UF-CIS-TR-93-023, Department of Computer and Information Sciences, University of Florida, 1993.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 566 – 577, Brisbane, August 1990.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, and S. Sarin. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):51 – 70, March 1988.

[DBC96] U. Dayal, A.P. Buchmann, and S. Chakravarthy. The HiPAC Project. In J. Widom and S. Ceri, editors, *Active Database Systems*, pages 177 – 206. Morgan Kaufmann, San Francisco, 1996.

[Deu94] A. Deutsch. Method and Composite Event in the "REACH" Active Database System. Master's thesis, Technical University Darmstadt, 1994.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204 – 214, Atlantic City, June 1990.

[DJPaQ94] O. Diaz, A. Jaime, N.W. Paton, and G. al Qaimari. Supporting Dynamic Displays Using Active Rules. *ACM SIGMOD Record*, 23(1):21 – 26, 1994.

[DPG91] O. Diaz, N. Patom, and P. Gray. Rule Management in Object-Oriented Databases: A Uniform Approach. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 317 – 326, Barcelona, September 1991.

[Eri93] J. Eriksson. CEDE: Composite Event DEtector In An Active Object-Oriented Database. Master's thesis, Department of Computer Science, Universtity of Skövde, 1993.

[Fri93] H. Fritschi. Entdeckung zusammengesetzter Ereignisse unter Berücksichtigung der Ereignisparameter. Master's thesis, Institut für Informatik, Universität Zürich, 1993.

[Gat95] S. Gatziu. *Events in an Active, Object-Oriented Database System.* Verlag Dr. Kovač, Hamburg, 1995.

[GD92] S. Gatziu and K.R. Dittrich. SAMOS: an Active Object-Oriented Database System. *IEEE Bulletin of the Technical Committee in Data Engineering: Special Issue On Active Databases*, 15(1 - 4):23 – 26, 1992.

[GGD94] S. Gatziu, A. Geppert, and K.R. Dittrich. The SAMOS Active DBMS Prototype. Technical Report 94.16, Institut für Informatik, Universität Zürich, 1994.

[GJ92] N.H. Gehani and H.V. Jagadish. Active Database Facilities in Ode. *IEEE Bulletin of the Technical Committee in Data Engineering: Special Issue On Active Databases*, 15(1 - 4):19 – 22, 1992.

[GJ96] N. Gehani and H.V. Jagadish. Active Database Facilities in Ode. In J. Widom and S. Ceri, editors, *Active Database Systems*, pages 207 – 232. Morgan Kaufmann, San Francisco, 1996.

[GJS93] N. Gehani, H.V. Jagadish, and O. Shmueli. COMPOSE: A System For Composite Specification And Detection. In N.R. Adam and B.K. Bhargava, editors, *Advanced Database Systems, Lecture Notes in Computer Science 759*, pages 3 – 15. Springer, Berlin et al., 1993.

[Han92a] E.N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49 – 58, San Diego, June 1992.

[Han92b] E.N. Hanson. The Design and Implementation of the Ariel Active Database Rule System. Technical Report UF-CIS-018-92, Department of Computer and Information Sciences, University of Florida, September 1992.

[Her95] H. Herbst. A Meta-Model for Specifying Business Rules in Systems Analysis. Proc. of the Seventh Conference on Advanced Information Systems, pages 186–199. Springer Berlin, 1995.

[HM95] H. Herbst and T. Myrach. A Repository System for Business Rules. Technical Report 57, Institute of Information Systems, University of Berne, January 1995.

[HW93] E.N. Hanson and J. Widom. An Overview of Production Rules in Database Systems. *The Knowledge Engineering Review*, 8(2):121 – 143, 1993.

[IS89] Y. E. Ioannidis and T. K. Sellis. Conflict Resolution of Rules Assigning Values to Virtual Attributes. Technical report, University of Maryland, Computer Science Department, 1989.

[Jen92] K. Jensen. *Coloured Petri Nets, Vol 1.* Springer, 1992.

[Joh93] D. Johansson. ERIC: An Analysis and Design of the Connections between Events, Rules, and Items in an Object-Oriented Environment. Master's thesis, Department of Computer Science, Universtity of Skövde, 1993.

[Joo94] S. Joosten. Trigger Modelling for Workflow Analysis. Technical report, University of Twente, Centre for Telematics and Information Technology, 1994.

[Kim95] S. Kim, S-K.and Chakravarthy. A Confluent Rule Execution Model for Active Databases. Technical Report UF-CIS-TR-95-032, Department of Computer and Information Sciences, University of Florida, 1995.

[MD89] D.R. McCarthy and U. Dayal. The Architecture Of An Active Data Base Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215 – 224, Portland, June 1989.

[OW90] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag, 1990.

[PCFW95] N.W. Paton, J. Campin, A.A.A. Fernandes, and M.H. Williams. Formal Specification of Active Database Functionality: A Survey. In T. Sellis, editor, *Rules on Database Systems, Lecture Notes in Computer Science 985*, pages 21 – 35. Springer, Berlin et al., 1995.

[PS96] S. Potamianos and M. Stonebraker. The Postgres Rule System. In J. Widom and S. Ceri, editors, *Active Database Systems*, pages 43 – 61. Morgan Kaufmann, San Francisco, 1996.

[Rei82] W. Reisig. *Petrinetze, Eine Einführung.* Springer, 1982.

[Rei86] W. Reisig. Place/Transition Systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Lecture Notes in Computer Science 254*, pages 117–141. Springer, Berlin et al., 1986.

[RW82] B. Rosenstengel and U. Winand. *Petri Netze, Eine anwendungsorientierte Einführung.* Vieweg-Verlag, 1982.

[Rys95] J. Ryser. Statische Analyse und Visualisierung von Regelbeziehungen in SAMOS. Master's thesis, Institut für Informatik, Universität Zürich, 1995.

[Sch95] M. Schlesinger. Vergleich aktiver Mechanismen in Ingres V6.4, Oracle V7.0 und Sybase V10.0. In *Software-Entwicklung - Methoden, Werkzeuge, Erfahrungen '95*, pages 41–53. Technische Akademie Esslingen, H.-J. Scheibl, September 1995.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 469 – 478, Barcelona, September 1991.

[SR86] M. Stonebraker and L.A. Rowe. The Design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 340 – 355, Washington, D.C., May 1986.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125 – 142, 1990.

[Thi86] R.S. Thiagarajan. Elementary Net Systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Lecture Notes in Computer Science 254*, pages 26–59. Springer, Berlin et al., 1986.

[vdVS93] L. van der Voort and A. Siebes. Enforcing Confluence of Rule Execution. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems*, pages 194 – 207. Springer, London et al., September 1993.

[VGD96] A. Vaduva, S. Gatziu, and K.R. Dittrich. Investigating Rule Termination in Active Database Systems with Expressive Rule Languages. Arbeitsbericht, Institut für Informatik, Universität Zürich, 1996.

[WC96] J. Widom and S. Ceri. Introduction to Active Database Systems. In J. Widom and S. Ceri, editors, *Active Database Systems*, pages 1 – 41. Morgan Kaufmann, San Francisco, 1996.

[WCD95] J. Widom, S. Ceri, and U. Dayal. *Active Database Systems*. Morgan Kaufmann, San Francisco, 1995.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 275 – 285, Barcelona, September 1991.

[WF90] J. Widom and S.J. Finkelstein. Set-Oriented Production Rules in relational Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259 – 270, Atlantic City, May 1990.

[WH95] T. Weik and A. Heuer. An Algorithm for the Analysis of Termination of Large Trigger Sets in an OODBMS. In M. Berndtsson and J. Hansson, editors, *Active and Real-Time database Systems (ARTDB-95)*, pages 170 – 189. Springer, London et al., 1995.

[Wid92] J. Widom. A Denotational Semantics for the Starburst Production Rule Language. *ACM SIGMOD Record*, 21(3):4 – 9, 1992.

[WN87] H. Walther and G. Naegler. *Graphen, Algorithmen, Programme*. Springer, 1987.