Entwicklung eines Prototypen für die aktive Schicht ALFRED

Roger Blum
Universität Bern
Institut für Informatik und angewandte Mathematik
Neubrückstr. 10
CH - 3012 Bern
E-Mail: roger.blum@app.ch

29. Mai 2000

Zusammenfassung

Die zunehmende Automatisierung von Geschäftsprozessen und -regeln hat dazu geführt, dass herkömmliche Datenbankmanagementsysteme, mit denen in praktisch allen modernen Unternehmungen die Daten verwaltet werden, den Anforderungen nicht mehr genügen. Als ein möglicher Ausweg haben sich die *aktiven* DBMS erwiesen. Aktive Datenbanksysteme erweitern herkömmliche Datenbanksysteme um die Fähigkeit, selbständig auf gewisse Situationen zu reagieren.

Am Institut für Wirtschaftsinformatik, Abteilung Information Engineering, der Universität Bern wird die aktive Schicht ALFRED (**A**ctive **L**ayer **F**or **R**ule **E**xecution in **D**atabase Systems) entwickelt. Mit dieser kann prinzipiell jedes beliebige (passive) Datenbanksystem in ein aktives verwandelt werden.

In dieser Arbeit wird die Entwicklung eines Prototypen beschrieben, in dem einige der entwickelten Konzepte realisiert sind. Ein erster Teil beschreibt den Entwurf und die Implementierung. Im zweiten Teil wird die Leistungsfähigkeit des realisierten Prototypen und damit die prinzipielle Realisierbarkeit der erarbeiteten Konzepte gezeigt.

Vielen Dank

An dieser Stelle möchte ich mich bedanken bei allen, die in irgendeiner Form zum Gelingen dieser Arbeit beigetragen haben.

Vor allem aber

Dipl. Inf. Markus Schlesinger und Prof. Oscar Nierstrasz für ihre Betreuung und Unterstützung und auch für ihre Geduld.

Meiner Frau für ihre moralische Unterstützung und meiner Tochter für die Freude, die sie mir jeden Tag bereitet.

Und last but not least meinem Arbeitgeber Andreas Aebi von der Firma APP Unternehmensberatung AG, der es mir ermöglicht, mein studentisches und berufliches Leben so zu gestalten, wie ich mir das wünsche.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Aktive Datenbankmanagementsysteme erweitern traditionelle Datenbankmanagementsysteme (DBMS) um Konzepte und Mechanismen, mit denen auf das Vorkommen bestimmter Situationen automatisch durch die Ausführung gewisser Aktionen reagiert werden kann. Aktives Verhalten wird durch Regeln definiert, die bestimmte Situationen beschreiben und Aktionen beinhalten, welche angeben, wie auf diese Situationen reagiert werden soll (vgl. Kapitel 2.2). Viele aktive Datenbankmanagementsysteme (aDBMS) unterstützen die Administration einer Regelmenge sowie die Möglichkeit das aktive Verhalten mit dem Gewünschten zu vergleichen nur schlecht oder aber gar nicht. Auch fehlen meist Werkzeuge und eine Entwicklungsumgebung (mit Debugger, Regel-Browser, usw.). Untersuchungen haben gezeigt, dass bestehende kommerzielle Systeme nur beschränkte aktive Funktionalitäten unterstützen. Es können zum Beispiel keine Zeitereignisse erkannt werden [Sch95]. Aktive Datenbankmanagementsysteme sind also wegen ihrer begrenzten aktiven Funktionalität zur Zeit nur beschränkt einsetzbar.

1.2 Aufgabenstellung

An der Abteilung "Information Engineering" des Instituts für Wirtschaftsinformatik der Universität Bern wird ein Konzept für eine aktive Schicht für Datenbanksysteme entwickelt und implementiert. In dieser Schicht können Regeln definiert und verarbeitet werden. Sie trägt den Namen ALFRED (**A**ctive **L**ayer **F**or **R**ule **E**xecution in **D**atabase Systems). ALFRED soll möglichst alle der oben genannten Nachteile aktiver DBMS beseitigen und berücksichtigt deshalb die meisten in der Literatur [DGG95] angegebenen Merkmale aktiver DBMS. Dazu gehören verschiedene Regelstrukturen, eine umfangreiche Regelsemantik und vor allem diverse Werkzeuge zur Regeladministration und -analyse. Durch die gewählte Architektur, eine erweiterte Schichtenarchitektur, kann mit ALFRED prinzipiell jedes (passive) DBMS in ein aktives überführt werden.

Im Rahmen dieser Diplomarbeit wird in einem ersten Teil ein Prototyp mit den minimal benötigten Funktionalitäten entworfen und implementiert. Die bereits in einer früheren Arbeit realisierte Benutzerschnittstelle [Blu97] wird dabei in die zu erstellende Applikation integriert. In einem zweiten

Teil werden gewisse Messungen durchgeführt, die Aufschluss über das Laufzeitverhalten des Systems sowie Ansatzpunkte für eine Optimierung liefern sollen.

1.3 Zielsetzung

Das Ziel dieser Diplomarbeit ist es, die für ALFRED entwickelten Konzepte auf ihre *Realisierbarkeit*, ihr *Laufzeitverhalten* und ihre *Effizienz* zu überprüfen. Der zu realisierende Prototyp soll Anhaltspunkte dafür liefern, welche Konzepte auf welche Art angepasst werden müssen, in welchen Bereichen oder Subsystemen Leistungsprobleme vorhanden sind und auf welche Art diese allenfalls behoben werden können.

1.4 Gliederung

Der Aufbau dieser Arbeit ist wie folgt: Im zweiten Kapitel werden die Grundlagen aktiver Datenbankmanagementsysteme erläutert. Dazu gehören die Merkmale solcher aDBMS, das Regelparadigma und die Regelspezifikation. Es werden auch mögliche Architekturen und Anwendungsgebiete beschrieben. Ein Überlick der bestehenden Prototypen und der kommerziellen Systeme schliesst dieses Kapitel ab.
Das Kapitel 3 beschreibt die Konzepte von ALFRED. Dabei werden zuerst die wichtigsten an das System gestellten Anforderungen erläutert. Darauf folgt die Beschreibung der entwickelten Regeldefinitionssprache, welche das Spektrum des aktiven Verhaltens von ALFRED festlegt. Dazu gehört auch die Art der Modellierung von Befehlen und Regeln. Die Erluterung der für ALFRED gewählten Architektur macht den Schluss des dritten Kapitels.
Mit der Beschreibung der gewählten Designmethode beginnt das Kapitel 4. Darauf folgt eine Abgrenzung der Funktionalitäten des Prototypen. Das Kapitel endet mit einer Beschreibung aller realisierten Objekttypen.
Das fünfte Kapitel beinhaltet das Konzept sowie die Ergebnisse der Laufzeitanalyse. Ebenfalls in diesem Kapitel werden gewisse Optimierungsmöglichkeiten aufgezeigt. Das letzte Kapitel beinhaltet eine Zusammenfassung der geleisteten Arbeiten und einen kurzen Ausblick.

2 Aktive Datenbanken

Datenbanksysteme bilden heute ein unverzichtbares Werkzeug für die meisten Unternehmungen aus praktisch allen Branchen. Insbesondere Anwendungen in der Verwaltung (zB. Datenverarbeitungs- und Personeninformationssysteme) aber auch in der Industrie (zB. Computer Integrated Manufacturing CIM und Computer Aided Design CAD) sind ohne Datenbanksysteme kaum noch denkbar. Viele dieser Anwendungen beinhalten ein reaktives Verhalten, das bei der Implementierung brücksichtigt werden muss. Dieses Verhalten bezieht sich dabei oft auf Daten, die im System gespeichert sind. Die eingesetzten Systeme müssen in bestimmten Situationen automatisch reagieren können. Traditionelle Datenbanksysteme sind aber passiv. Das bedeutet, dass diese Reaktionen explizit durch einen Benutzer oder eine Applikation ausgelöst werden müssen. Dies kann prinzipiell auf zwei Arten erfolgen:

- **Integriert in Applikationen**
  Die Regeln sind fest in den Applikationen verankert. Das bedeutet, dass in jeder Anwendung, in der auf die Datenbank zugegriffen wird, Situationen auf Datenbank-Ebene erkannt werden müssen, damit die notwendigen Aktionen ausgeführt werden können. Regeln werden also oftmals redundant realisiert. Dies ist eine grosse potentielle Fehlerquelle, wenn redundant vorhandene Regeln manipuliert werden müssen. Inkonsistentes Verhalten und eventuell auch inkonsistente Daten können die Folge sein.

- **Polling**
  Spezielle Applikationen fragen in regelmässigen Abständen die Datenbank ab und prüfen, ob bestimmte Situationen eingetreten sind. Die Hauptschwierigkeit bei diesem Ansatz ist es, eine angemessene Frequenz für das Polling festzulegen. Wird die Überprüfung zu häufig durchgeführt, leidet die Performance des ganzen Systems darunter. Ist die Polling-Frequenz zu niedrig, kann auf bestimmte Situationen nicht zeitgerecht reagiert werden.

Diese gravierenden Nachteile haben zu der Erkenntnis geführt, dass neue Konzepte für die Realisierung aktiven Verhaltens in DBMS benötigt werden. Ein Ergebnis der Forschung stellen die *aktiven Datenbankmanagementsysteme* (aDBMS) dar. Dayal definiert aktive DBMS wie folgt [Day95]:

> "An active database system is a database system that monitors situations of interest, and when they occur, triggers an appropriate response in a timely manner."

Aktive DBMS zeichen sich durch einige wesentliche Merkmale aus, die im folgenden erläutert werden.

2.1 Merkmale

Aktive Datenbanksysteme erweitern traditionelle (passive) DBMS um Mechanismen zur Realisierung von aktivem Verhalten. Diese Erweiterung der Funktionalitäten bedeutet, dass aDBMS einige wichtige Merkmale haben müssen [DGG95]:

- Ein wichtiges Merkmal aktiver DBMS ist, dass sie selbst Datenbankmanagementsysteme sind. Dies bedeutet, dass alle Konzepte passiver DBMS auch in aktiven Systemen realisiert sind. Dazu gehören zum Beispiel *Transaktionsverarbeitung*, *Datenrecovery*, sowie eine *Datendefinitionssprache* (DDL) und *Datenmanipulationssprache* (DML).
- Zusätzlich muss ein aDBMS die Definition und Verwaltung von Regeln ermöglichen. Dafür wird eine *Regeldefinitionssprache* (RDL) benötigt, mit der Inhalte und Semantiken von Regeln spezifiziert werden.
- Eine weitere Eigenschaft aktiver Datenbankmanagementsysteme ist ein *Verarbeitungsmodell* mit einer wohldefinierten Semantik, welches ermöglicht, Ereignisse zu erkennen, Bedingungen auszuwerten und Aktionen auszuführen.

Neben diesen zwingenden Eigenschaften können aDBMS auch eine Reihe von optionalen Merkmalen aufweisen. Ein Beispiel dafür ist eine integrierte Entwicklungsumgebung mit Werkzeugen, zB. zum Durchsuchen, Entwerfen und Analysieren der Regelmenge.

2.2 Regelparadigma

Aktives Verhalten wird in aDBMS durch Event-Condition-Action (ECA)-Regeln [MD89] spezifiziert. *Ereignisse* und *Bedingungen* beschreiben eine

Situation. Die *Aktion* legt die Reaktion des Systems auf eine eingetretene Situation fest:

- **E (Event)**
  Das Eintreten des Ereignisses löst die Regel aus. Dies kann zum Beispiel ein spezieller Zeitpunkt wie der 30. September 1998, 12:00 Uhr sein.
- **C (Condition)**
  Durch die Bedingungsauswertung wird überprüft, ob die Datenbank einen bestimmten Zustand hat (zB. ob ein Mieter mit dem Namen "Meier" in der Mieter-Relation eines relationalen DBMS vorhanden ist).
- **A (Action)**
  Die Aktion (zB. im relationalen Kontext das Einfügen eines neuen Datensatzes in eine Relation "Mietvertrag") wird ausgeführt, wenn die Regel ausgelöst und die Bedingung erfüllt ist.

Neben der ECA-Struktur existieren auch Varianten wie EA (keine spezifizierte Bedingung, d.h. immer `true`), ECAA (eine Aktion für beide möglichen Ausgänge der Bedingungsauswertung), und CA. Für CA-Regeln muss festgelegt werden, durch welches Ereignis sie ausgelöst werden.

2.3 Regelspezifikation

Die Regeldefinitionssprache definiert das Spektrum aktiven Verhaltens in einem aDBMS. Sie ist unterteilt in einen *deklarativen* und einen *semantischen* Teil [WC96].

2.3.1 Regeldeklaration

Der deklarative Teil einer Regel spezifiziert das reaktive Verhalten. Er legt die Struktur der Regel sowie ihre Kompenenten fest. Es können verschiedene Typen von Ereignissen, Bedingungen und Aktionen unterschieden werden:

- **Ereignisse**
  Ereignisse legen fest, wodurch oder wann Regeln ausgelöst werden. Es

wird zwischen primitiven und zusammengesetzten Ereignissen unterschieden. Primitive Ereignisse sind zum Beispiel:

- Datenmanipulationsereignisse, wie zB. `insert`, `update` und `delete` auf einer Tabelle einer relationalen Datenbank
- Datenselektionsereignisse, zB. ein `select` auf einer Tabelle einer relationalen Datenbank
- Absolute Zeitereignisse, wie zB. `1998/09/30 at 12:00:00`

Zusammengesetzte oder komplexe Ereignisse basieren auf primitiven und/oder anderen komplexen Ereignissen, die durch Ereignisoperatoren kombiniert werden. Beispiele dafür sind:

- Bool'sche Operatoren (zB. `and` und `or`)
- Verzögerungsoperatoren (zB. `5 seconds after E1`)
- Sequenzoperatoren (zB. E4 tritt ein, wenn E1, E2 und E3 in genau dieser Reihenfolge vorkommen).
- Intervalloperatoren (zB. Ereignis E4 tritt ein, wenn E2 zwischen E1 und E3 eingetreten ist)
- Wiederholungsoperatoren (zB. Ereignis E2 tritt bei jedem zehnten Vorkommen von E1 ein).

- **Bedingungen**
  Es wird zwischen primitiven und zusammengesetzten Bedingungen unterschieden. Beispiele für primitive Bedingungen sind:

  - die bool'schen Werte `true` und `false`
  - Prädikate (zB. `Mieter.Name = 'Meier'`)
  - Abfragen (zB. `select * from Mieter where Name = 'Meier'`).

  Komplexe Bedingungen werden aus primitiven und/oder anderen komplexen Bedingungen gebildet, die mit den bool'schen Operatoren `and`, `or` und `not` verknüpft sind.

- **Aktionen**
  Die Aktionskomponente beschreibt die Reaktion des Systems auf eine eingetretene Situation. Die Verarbeitung einer Aktionskomponente kann dazu führen, dass wiederum weitere Regeln ausgelöst werden (zB. durch Datenmanipulationsaktionen), was zu Interdependenzen führt,

die insbesondere bei der Definition und Analyse von Regeln zu berücksichtigen sind. Auch Aktionen können primitiv und zusammengestzt sein. Beispiele für primitive Aktionen sind:

- Datenmanipulationsaktionen (zB. `delete from Mieter where Name = 'Meier'` in einem relationalen DBMS)
- Meldungsaktionen (zB. `message 'Datensatz existiert bereits!'`).

Zusammengesetzte Aktionen sind Verkettungen von primitiven Aktionen.

2.3.2 Regelsemantik

Neben der strukturellen Definition einer Regel weist eine Regeldefinitionssprache auch Elemente auf, welche die Art der Ausführung (d.h. die dynamischen Aspekte) im Detail festlegen:

- **Ausführungszeitpunkte**
  Bei der Regelausführung wird für Ereignisse zwischen den zwei Zeitpunkten `pre` und `post` unterschieden:
  - Pre-Zeitpunkt
    Regeln, die mit `pre` definiert sind, werden *vor* der eigentlichen Verarbeitung des Befehls verarbeitet (zB. für eine Kontrolle der Zugriffsrechte).
  - Post-Zeitpunkt
    Regeln, die mit `post` definiert sind, werden *nach* der eigentlichen Befehlsverarbeitung ausgeführt (zB. für die Prüfungen von (komplexen) Integritätsbedingungen).
- **Granularitäten**
  Bei der Ausführung von Befehlen wird häufig auf mehrere Datensätze zugegriffen. Aus diesem Grund wird zwischen zwei verschiedenen Auslösungsgranularitäten unterschieden:
  - Instanzorientiert (*instance*)
    Instanzorientierte Regeln, werden für jeden involvierten Datensatz einmal ausgeführt.
  - Mengenorientiert (*set*)
    Mengenorientierte Regeln hingegen werden genau einmal für den

Befehl ausgeführt, ohne Berücksichtigung der Anzahl betroffener Datensätze.

- **Prioritäten**
  Ein Ereignis kann mehrere Regeln auslösen. Durch die Festlegung von Prioritäten kann bestimmt werden, in welcher Reihenfolge diese Regeln ausgeführt werden sollen. In aDBMS werden zwei Typen von Prioritäten unterschieden:

  – *Partielle Prioritäten*
    Die Priorität einer Regel wird relativ zu einer anderen angegeben (höher oder niedriger).
  – *Totale Prioritäten*
    Die Priorität einer Regel wird absolut angegeben (zB. durch Zahlenwerte).

- **Kopplungsmodi**
  Kopplungsmodi legen die Regelausführung in Bezug auf Transaktionen fest. Diese können sowohl zwischen Ereignissen und Bedingungen (EC-Kopplung), wie auch zwischen Bedingungen und Aktionen (CA-Kopplung) definiert werden. Mit der EC-Kopplung wird festgelegt, wann die Bedingung in Abhängigkeit von der Regelauslösung ausgewertet werden soll. Analoges gilt für die CA-Kopplung.

  Es werden verschiedene Kopplungsmodi unterschieden, deren Bedeutung am Beispiel der EC-Kopplung erläutert werden:

  – Immediate
    Die Bedingungsauswertung erfolgt in derselben Transaktion, unmittelbar nach dem Eintritt des Ereignisses.
  – Deferred
    Die Bedingungsauswertung erfolgt in der gleichen Transaktion, jedoch verzögert und zwar nach dem letzten Befehl, jedoch vor dem *commit*.
  – Detached independent
    Die Bedingung wird in einer separaten Transaktion, die von der auslösenden vollständig unabhängig ist, ausgewertet. Dies bedeutet, dass die Bedingungsauswertung auch dann durchgeführt wird, wenn die auslösende Transaktion, zB. aufgrund eines Fehlers, abgebrochen werden muss.

Daneben wird auch der Modus "Detached dependent" mit den drei Varianten *sequential*, *parallel* und *exclusive* vorgeschlagen [BBKZ93]. Bei diesem wird die Bedingung ebenfalls in einer separaten Transaktion ausgewertet. Es bestehen jedoch im Gegensatz zum vorher genannten Kopplungsmodus kausale Abhängigkeiten zwischen den Transaktionen. Beim Kopplungsmodus "Detached dependent sequential" zB. muss die Bedingung in einer separaten Transaktion, aber nach der Ereigniserkennung ausgewertet werden.

- **Zustände**
  Regeln in aDBMS können entweder *aktiv* oder *passiv* sein.
  - aktiv
    Aktive *(enabled)* Regeln werden ausgelöst und verarbeitet.
  - passiv
    Passive *(disabled)* Regeln sind zwar im System gespeichert, werden aber nicht verarbeitet, auch wenn das auslösende Ereignis eintritt.

2.4 Architekturen

In der Literatur werden unterschiedliche Architekturen für aDBMS vorgeschlagen. Zwei der wichtigeren sind:

- **Schichtenarchitektur** (layered architecture)
  Bei diesem Ansatz wird zwischen den Applikationen und dem DBMS eine Schicht eingefügt, die das gesamte aktive Verhalten beinhaltet. Diese Architektur wird auch als *loosely coupled* bezeichnet [WC96].

  Ein wichtiger Vorteil dieser Architektur ist, dass im Prinzip jedes beliebige (passive) DBMS ohne interne Veränderung in ein aDBMS umgewandelt werden kann. Dieser Architekturansatz hat aber auch einige Nachteile. Einer der wichtigsten ist der hohe Kommunikationsaufwand zwischen aktiver Schicht und DBMS, wodurch eine geringe Performance resultieren kann. Auch ist es nicht möglich, mit wichtigen Subsystemen (zB. Transaktionsmanager) des DBMS zu kommunizieren. Dies führt dazu, dass einige Semantiken, wie zum Beispiel Kopplungsmodi, nicht realisiert werden können.

- **Integrierte Architektur** (built-in architecture)
  Bei dieser Architektur sind die aktiven Komponenten Bestandteil des

DBMS. Dies bedeutet, dass bestehende DBMS weiter- oder von Grund auf neu entwickelt worden sind. Diese Architektur wird auch als *tightly coupled* bezeichnet [WC96].

Die Vorteile einer solchen Architektur sind die problemlose Verwendung von Subsystemen des DBMS, sowie die Möglichkeit alle Eigenschaften der Regelausführung, also auch Kopplungsmodi, zu realisieren. Dem stehen die Nachteile von enormen Entwicklungsaufwänden für ein quasi neues DBMS gegenüber sowie die Tatsache, dass nur ein bestimmtes DBMS in ein aktives DBMS transformiert wird.

2.5 Anwendungsgebiete

Regeln können für unterschiedliche Aufgaben eingesetzt werden. Einige Beispiele sind:

- **Integritätsbedingungen**
  In traditionellen DBMS wird für die Überprüfungen der Integritätsbedingungen eine spezielle Komponente benötigt. In aDBMS kann dies durch ECA-Regeln erfolgen.
- **Datenbanktrigger**
  Ein Datenbanktrigger ist eine Prozedur, die automatisch ausgeführt wird, wenn im relationalen Kontext Datenmanipulationen auf einer bestimmten Relation durchgeführt werden. Trigger sind universeller einsetzbar als Integritätsbedingungen. Auch sie können durch Regeln ersetzt werden.
- **Autorisation**
  Mit Regeln können Zugriffsrechte der Benutzer überprüft werden. In traditionellen DBMS wird dafür ein spezielles Subsystem benötigt.
- **Replikation**
  Mit Hilfe von ECA-Regeln können Daten repliziert werden. Die Replikation kann durch Datenmanipulationsbefehle oder aber durch Zeitereignisse ausgelöst werden.
- **Kontrolle und Steuerung**
  Aktive DMBS können für die Kontrolle und Steuerung von Abläufen eingesetzt werden, wie zB. bei Workflow-Systemen und Inventarkontrollen.

2.6 Überblick

In der Literatur gibt es viele Theorien zu aDBMS und Beschreibungen von Prototypen aktiver DBMS. Diese lassen sich in die drei Kategorien *relationale Prototypen*, *objektorientierte Prototypen* und *kommerzielle Systeme* einteilen. Die Abbildung 1 gibt einen Überblick über die wichtigsten Projekte dieser drei Kategorien.

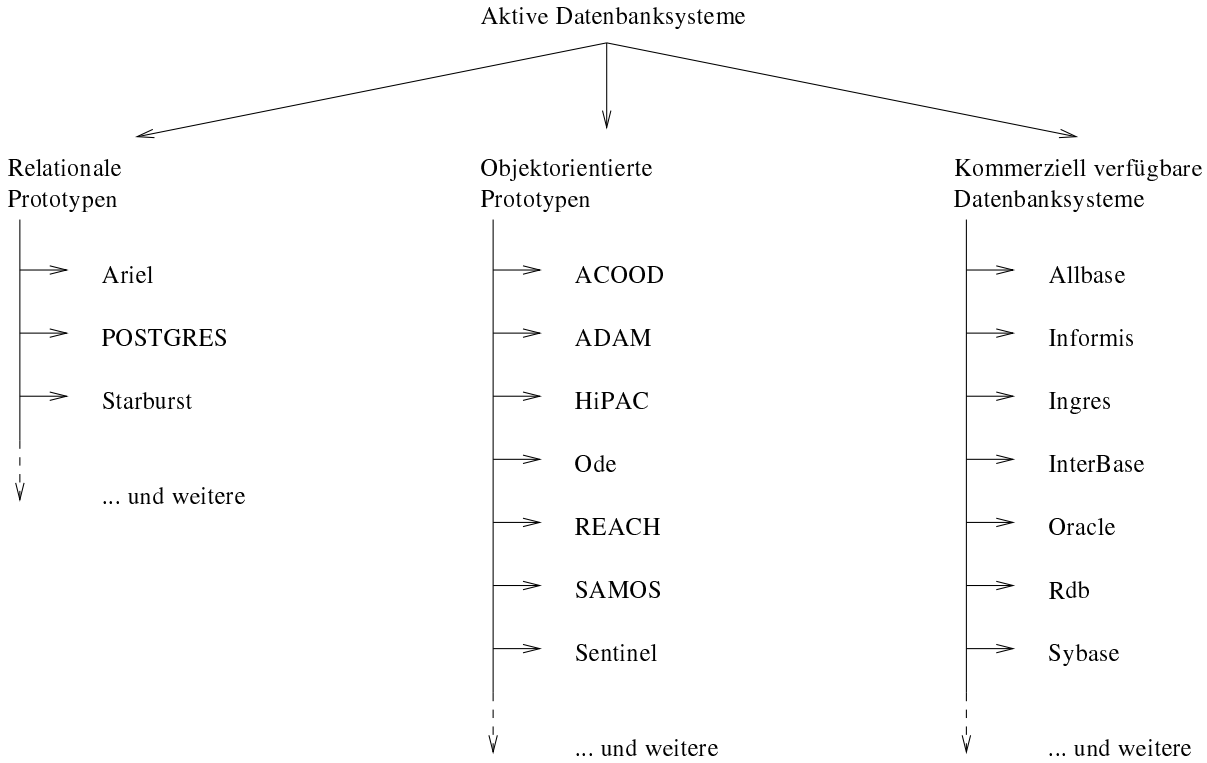


Abbildung 1: Überblick über Prototypen und Systeme aktiver Datenbankmanagementsysteme (aus [Lö96])

2.6.1 relationale Prototypen

Aus bestehenden passiven relationalen DBMS wurden Prototypen relationaler aktiver Datenbanksysteme entwickelt. Beispiele dafür sind:

- **Ariel**
  Ariel ist ein Forschungsprojekt der Write State University und später der Universität von Florida. Das Schwergewicht des Projekts liegt auf einer effizienten Bedingungsauswertung. Für Ariel wurde eine integrierte Architektur gewählt [Han92a, Han92b, Han96].

- **POSTGRES**
  An der University of California at Berkeley wurde dieser Prototyp entwickelt. Unter anderem können in POSTGRES Trigger für die Realisierung von Sichten und Integritätsbedingungen definiert werden. Es handelt sich ebenfalls um eine integrierte Architektur [SR86, SRH90, PS96]

2.6.2 objektorientierte Prototypen

Neuere Ansätze für Prototypen aktiver DBMS basieren auf objektorientierten, passiven DBMS. Beispiele dafür sind:

- **HiPAC**
  Bei HiPAC handelt es sich um einen der ersten objektorientierten Prototypen für aktive Datenbankensysteme. Er wurde an der Computer Corporation of America und am Xerox Advanced Information Technology Center mit einer integrierten Architektur entwickelt. Die Projektschwergewichte lagen auf einer mächtigen Sprache für die Spezifikation von Ereignissen, einer mächtigen Ausführungssemantik sowie der Betrachtung von Fristen bei der Regelausführung. HiPAC ist die Basis für viele weitere Arbeiten im Bereich der aktiven Datenbanksysteme [DBB$^+$88, MD89, DBC96].

- **REACH**
  REACH, ein Projekt der Technischen Hochschule Darmstadt, ist eines dieser Projekte, welches auf HiPAC basiert. Es befasst sich vor allem mit der Verwaltung von heterogenen Daten. REACH wurde als integrierte Architektur auf der Basis von O2 und ObjectStore realisiert [BBKZ92, BBKZ93, Deu94].

- **SAMOS**
  Dieser an der Universität Zürich entworfene Prototyp weist eine Schichtenarchitektur auf und wurde auf der Basis von ObjectStore entwickelt.

Neben einer mächtigen Sprache zur Spezifikation von Ereignissen umfasst SAMOS als Besonderheit Mechanismen zur Ereigniserkennung mit Petri-Netzen [GD92, Fri93, GGD94, Gat95].

2.6.3 kommerzielle Systeme

Bei den kommerziellen aDBMS handelt es sich um bereits verfügbare aktive Datenbankmanagementsysteme, die um Mechanismen und Konzepte für die Realisierung von aktivem Verhalten erweitert wurden. Beispiele sind *Ingres*, *Oracle* und *Sybase*. Aktives Verhalten wird in ihnen mit Datenbanktriggern realisiert. Die aktiven Fähigkeiten dieser Systeme liegen aber weit unter denjenigen der oben beschriebenen Prototypen. So werden zum Beispiel ausführungsbezogene Eigenschaften wie Kopplungsmodi nicht unterstützt. Auch können keine Zeitereignisse erkannt werden [Sch95].

Zudem wird die Spezifikation von aktivem Verhalten, das dem ECA-Paradigma folgt, auch im vorgeschlagenen SQL-3 Standard berücksichtigt.

3 Die Konzepte von ALFRED

ALFRED ist ein Forschungsprojekt, das am Institut für Wirtschaftsinformatik der Universität Bern seit 1995 bearbeitet wird. Basis aller durchgeführten Arbeiten ist ein Nationalfonds-Projekt, dessen Ziel eine Untersuchung der Anwendung des Trigger-Konzepts in Bezug auf die Realisierung von Geschäftsregeln ist [KHS94].

Fallstudien haben gezeigt, dass aktives Verhalten in vielen kommerziell eingesetzten Datenbankmanagementsystemen zwar vorhanden ist, aber nur wenig genutzt wird. Ein grosser Teil der gefundenen Regeln wird für die Sicherstellung der Datenintegrität verwendet, welche auf Datenbankebene als Integritätsbedingungen realisiert oder in Applikationen fix codiert sind. Nur sehr wenige Regeln werden für die Steuerung und Kontrolle von Geschäftsprozessen verwendet. Diese sind praktisch ausnahmslos auf Applikationsebene implementiert.
Eine Untersuchung der Triggermechanismen von mehreren kommerziellen DBMS liefert einige Erklärungen dafür, weshalb das vorhandene aktive Potential (insbesondere die Trigger) der DBMS nur schlecht oder gar nicht genutzt wird:

- **Unzureichende Mechanismen**
  Die unterstützten Mechanismen sind zu schwach und unausgereift. Regeln können zum Beispiel nur durch Datenmanipulationsereignisse, wie `insert` oder `update`, ausgelöst werden.
- **Wenige Semantiken**
  Die Art der Verarbeitung lässt sich nur mit wenigen Semantiken, wie einem Auslösezeitpunkt und einer Granularität, beeinflussen.
- **Keine Simulation**
  Kein System unterstützt die Simulation zum Vergleich des implementierten Verhaltens mit dem gewünschten.

Obwohl die aktiven Funktionalitäten in DBMS bis heute nur wenig genutzt werden, ist unbestritten, dass diesem Ansatz eine zentrale Bedeutung in der Weiterentwicklung von Datenbankmanagementsystemen zukommt. Es reicht aber nicht, ein vorhandenes System mit einer Komponente für die Ereigniserkennung zu erweitern. Vielmehr muss das reaktive Verhalten des Systems ins Zentrum gestellt werden. Alle anderen Komponenten, wie das Transaktionsmanagement und das Recovery, müssen dann darauf abgestimmt werden.

Dieser grundlegende Gedanke steht bei der Entwicklung von ALFRED ganz klar im Mittelpunkt.

Im Rahmen mehrerer Arbeiten [Lö96, Sch98, Blu97] wurden diverse Konzepte für ALFRED entwickelt. Die folgenden Abschnitte geben einen Überblick über die erarbeiteten Ergebnisse, soweit sie für diese Arbeit wichtig sind. Zudem wird ein Transaktionskonzept als Basis für die Verarbeitung von Befehlen und Regeln entworfen.

3.1 Anforderungen

An die Entwicklung der Konzepte von ALFRED sind verschiedene Anforderungen gestellt worden. Diese lassen sich in *betriebswirtschaftliche* und in *technische* Anforderungen unterteilen. Zu den betriebswirtschaftlichen gehört unter anderem die Unterstützung von *Geschäftsprozessen* und *Geschäftsregeln*. Geschäftsprozesse beschreiben einen wohldefinierten Ablauf inklusive seiner Schnittstellen und der Verantwortlichkeiten, wie zum Beispiel die "Auftragsbearbeitung". Geschäftsregeln legen statische Eigenschaften fest, wie zum Beispiel "Grosskunden haben 2% Rabatt".
Für diese Arbeit stehen aber die technischen Anforderungen im Vordergrund. Die wichtigsten sind:

- **Modell**
  Für Befehle, Transaktionen und Regeln soll ein einheitliches Modell, das eine ganzheitliche Modellierung der Regeln erlaubt, verwendet werden. Dieses muss sowohl für die Verarbeitung als auch für Analyse und Simulation geeignet sein.

- **Regelsprache**
  ALFRED soll über eine mächtige Regeldefinitionssprache (RDL) verfügen. Diese muss sowohl automatisierbare als auch (Geschäfts-)Regeln, die von Benutzern zu erfüllen sind, unterstützen.

- **Regeladministration**
  Damit der Benutzer eines aDBMS nicht den Überblick verliert, wenn eine grössere Anzahl von Regeln definiert ist, wird eine regelorientierte Entwicklungsumgebung benötigt. Diese sollte neben Werkzeugen zum Verwalten der Regeln auch ein spezielles Simulations-Tool enthalten. Mit diesem kann das implementierte mit dem gewünschten Verhalten einer Regel(-menge) verglichen werden.

- **Regelverarbeitung**
  ALFRED muss für die Verarbeitung von Regeln Ereignisse erkennen, Bedingungen auswerten und Aktionen ausführen können. Dies muss unter Berücksichtigung der bei der Regelspezifikation festgelegten ausführungsbezogenen Eigenschaften erfolgen.
  Auf Fehler, die zum Beispiel bei der Transaktionsverarbeitung auftreten, muss entsprechend reagiert werden. Die nötigen Recovery-Massnahmen müssen dabei ausgelöste und bereits ausgeführte Regeln berücksichtigen.

- **Datenmodelle**
  Um ALFRED auf beliebige Datenbanksysteme aufsetzen zu können, müssen verschiedene Datenmodelle, wie zum Beispiel das relationale und das objektorientierte Modell unterstützt werden.

- **Plattformen**
  ALFRED soll auf allen gängigen Plattformen (zB. *Microsoft Windows*, *Unix*, usw.) eingesetzt und auf beliebige Datenbanksysteme (zB. *Oracle*, *Ingres*, usw.) aufgesetzt werden können.

- **Benutzer**
  Das Benutzersystem soll, den heutigen Standards entsprechend, grafikorientiert sein. Es muss durch möglichst grosse Selbsterklärbarkeit eine leichte Erlernbarkeit gewährleisten. Eine grosse Robustheit gewährleistet auch bei Fehlmanipulationen durch den Benutzer konsistente Datenbankzustände und ein stabiles System. Fehlerhafte Eingaben führen in jedem Fall zu Fehlermeldungen. Das System soll sich jederzeit so verhalten, wie es der Benutzer erwartet. Eine Online-Hilfe gibt dem Benutzer bei allen Arbeiten in ALFRED Unterstützung. Das System muss Werkzeuge zur Verfügung stellen, die den Benutzer bei komplexen Aufgaben auf anschauliche Weise anleiten.

3.2 Regeldefinitionssprache

Das Spektrum reaktiven Verhaltens eines jeden aDBMS wird durch die Regeldefinitionssprache bestimmt. Diese gibt an, welche Arten von Ereignissen, Bedingungen und Aktionen spezifiziert werden können. Zusätzlich wird festgelegt, welche Regelsemantiken wie Prioritäten und Kopplungsmodi unterstützt werden.

Die ALFRED Rule Definition Language (ARDL) besteht aus zwei Teilen. Mit dem Strukturteil werden die Struktur, wie zum Beispiel ECA und ECAA, und die einzelnen Komponenteninhalte definiert. Mit dem Semantikteil wird die Verarbeitung von Regeln festgelegt.

Eine vereinfachte Syntax von ECA-Regeln ist im folgenden Beispiel angegeben:

```
<rule>            ::= 'rule'          <ident>
                                      <rule_structure>
                                      <rule_semantics>

<rule_structure>  ::= 'on'            <rule_event>
                      'if'            <rule_condition>
                      'do'            <rule_action>

<rule_semantics>  ::= 'is'            <rule_state>
                    [ 'priority'      <rule_order>    ]
                    [ 'ec-coupling'   <rule_coupling>
                      'ca-coupling'   <rule_coupling> ]
```

Die vollständige Syntax ist in [Sch98] enthalten. Der Anhang A der vorliegenden Arbeit enthält die Teile daraus, die für das Verständnis verschiedener Abschnitte wichtig sind.

3.2.1 Regelstruktur

Die ARDL unterstützt die Definition von drei ereignisbasierten Regelstrukturen (ECAA, ECA, EA). Jede Regelkomponente kann *sprachlich*, *primitiv* oder *komplex* resp. *zusammengesetzt* sein. Sprachliche Komponenten werden für die Erfassung von Regeln verwendet, die sich nicht vollständig automatisieren lassen. Sie werden mit Hilfe eines Benutzereingriffs verarbeitet.

- **Ereignisse**
  Eine ereignisbasierte Regel wird ausgelöst, wenn das entsprechende einfache oder komplexe Ereignis entdeckt wird. Mit *sprachlichen Ereignissen* werden Situationen ausserhalb von ALFRED spezifiziert. Obwohl diese Ereignisse nicht selbständig vom System erkannt werden können, müssen sie dennoch modelliert werden, um zum Beispiel

Geschäftsprozesse, die durch ECA-Regeln beschrieben sind, in ALFRED abbilden zu können.
Beispiele für unterstützte *primitive Ereignisse* sind:

- Datenbankereignisse (zB. `create database`)
- Datendefinitionsereignisse (zB. `create rule`)
- Datenmanipulationsereignisse (zB. `insert`)
- Absolute Zeitereignisse (zB. `12:00, 16. Oktober, 1998`)
- Abstrakte Ereignisse (zB. `'Bestellung ist eingetroffen'`)

Komplexe Ereignisse bestehen aus primitiven und/oder komplexen Ereignissen, die durch Ereignisoperatoren verknüpft werden. Die ARDL unterstützt mehrere Ereignisoperatoren wie beispielsweise *Auswahl-*, *Boolean-*, *Sequenz-*, *Wiederholungs-*, *Intervall-* sowie *Zeitoperatoren*.

- **Bedingungen**
  Bedingungen legen fest, was überprüft werden soll. Mit *sprachlichen Bedingungen* können beliebige `true/false`-Entscheidungen gefällt werden. Da diese aber nicht vom System ausgewertet werden können, muss dies über einen Dialog vom Benutzer erledigt werden.
  Beispiele für *primitive Bedingungen* sind:

  - die booleschen Konstanten `true` und `false`
  - Prädikate (beispielsweise `Mitarbeiter.Name = 'Meier'`)
  - Abfragen (zB. `select * from Mitarbeiter where Name = 'Meier'`)

  Komplexe Bedingungen bestehen aus primitiven und/oder anderen komplexen Bedingungen, die mit den *booleschen Operatoren* `and`, `or` und `not` kombiniert sind. Zusätzlich unterstützt ALFRED *boolesche Funktionen*, die ein `true` oder `false` als Resultat zurückliefern.

- **Aktionen**
  Die Aktionskomponente einer Regel legt die Reaktion auf eine bestimmte Situation fest. *Sprachliche Aktionen* werden dazu verwendet, Aktionen die ausserhalb von ALFRED auszuführen sind (zB. "Kunden anrufen") darzustellen.
  Als *primitive Aktionen* werden u.a die folgenden unterstützt:

  - Datenmanipulationsaktionen, beispielsweise `insert`, `update` und `delete`.

- Meldungsaktionen, z.B. `message to user Meier: 'Eingegebener Datensatz ist fehlerhaft!'`
- Ereignisauslösungs-Aktionen
  Diese Aktionen werden dazu verwendet, um *abstrakte Ereignisse* zu signalisieren (zB. `raise 'Bestellung ist eingetroffen'`).

Komplexe Aktionen sind Sequenzen von primitiven und anderen komplexen Aktionen, wie Transaktionen, Prozeduren und Applikationen.

3.2.2 Regelsemantik

Der Semantikteil der ARDL besteht aus einer Anzahl von Anweisungen, welche bei der Regeldefinition spezifiziert werden und die Regelverarbeitung bestimmen. Dazu gehören:

- **Verarbeitungszeitpunkte**
  ALFRED unterstützt sowohl `pre` als auch `post` als Verarbeitungszeitpunkte für Regeln. Als `pre` spezifizierte Regeln werden vor der eingentlichen Befehlsverarbeitung ausgelöst, `post`-Regeln danach.
- **Prioritäten**
  In ALFRED werden globale Prioritäten unterstützt, die als ganzzahlige Werte angegeben werden. Regeln, welche gleichzeitig ausgelöst werden und dieselbe Priorität haben, werden gleichzeitig ausgeführt, wenn es keine Konflikte zwischen ihnen gibt. Sonst bestimmt das System selbst die Ausführungsreihenfolge.
- **Kopplungsmodi**
  Die ARDL unterstützt die sechs Kopplungsmodi *immediate*, *deferred*, *detached*, *detached dependent sequential*, *detached dependent parallel* und *detached dependent exclusive* (vgl. Kapitel 2.3.2).
- **Granularitäten**
  Die Verarbeitung von Regeln kann sowohl instanz-, wie auch mengenorientiert erfolgen.

3.3 Modellierung mit erweiterten gefärbten Petri-Netzen

Die Modellierung von Befehlen und Regeln erfolgt in ALFRED vollständig auf der Basis von erweiterten *gefärbten Petri-Netzen*, die *Action Rule Flow*

Petri Nets (ARFPN) genannt werden [Lö96, Sch98]. Bei ihrer Konzeption wurde insbesondere darauf geachtet, dass alle Komponenten dargestellt und auch nahtlos miteinander verbunden werden können. Diese ARFPN sind ebenfalls für die Verarbeitung und für die Simulation verwendbar.

3.3.1 Verarbeitungsmodell

Benutzereingaben und Befehle sowie Regeln werden in ALFRED als ARFPN modelliert und transaktionsbasiert verarbeitet. Zu diesem Zweck werden vier Arten von Transaktionen unterschieden:

1. `ROOT`-**Transaktionen**
   Diese Transaktionsart bildet die Basis jeder Verarbeitung und erfüllt die für Transaktionen wichtigen *ACID*-Eigenschaften [AEA95]:

   - **Atomicity (A)**
     Eine Transaktion ist *atomar*. Sie wird als Einheit betrachtet. Dies bedeutet insbesondere, dass entweder alle oder keine Operation festgeschrieben ('committed') wird.
   - **Consistency (C)**
     Die Verarbeitung der Transaktion gewährleistet einen konsistenten Zustand der Datenbank.
   - **Isolation (I)**
     Transaktionen können nicht auf Zwischenergebnisse anderer Transaktionen zugreifen. Datenbankänderungen sind für andere erst nach dem `commit` sichtbar.
   - **Durability (D)**
     Nachdem eine Transaktion festgeschrieben wurde, können Änderungen nicht mehr verlorengehen. Die manipulierten Daten sind persistent gespeichert.

   In ALFRED werden zwei Typen von ROOT-Transaktionen unterschieden:

   - **Pseudo-Transaktionen**
     ROOT-Transaktionen dieses Typs werden für die Verarbeitung von Benutzerbefehlen, wie zum Beispiel Datenmanipulationsbefehlen, verwendet. Diese Transaktionen bestehen aus vier Befehlen:

  - P_BOT (Pseudo-Begin-Of-Transaction)
    Durch die Verarbeitung dieses Befehls wird eine neue ROOT-Transaktion erzeugt.
  - P_EOT (Pseudo-End-Of-Transaction)
    Dieser Befehl schliesst den Anweisungsblock der Pseudo-Transaktion ab.
  - P_COT (Pseudo-Commit-Of-Transaction)
    Mit der Verarbeitung dieses Befehls werden die manipulierten Datensätze festgeschrieben und die Transaktion terminiert.
  - P_AOT (Pseudo-Abort-Of-Transaction)
    Mit der Verarbeitung dieses Befehls werden alle durchgeführten Datenmanipulationen verworfen ('rollback') und die Transaktion beendet.

- **Benutzerdefinierte Transaktionen**
  ROOT-Transaktionen dieses Typs werden für benutzerdefinierte Transaktionen sowie für Regelkomponenten, die losgelöst von der eigentlichen Befehlsverarbeitung (Kopplungsmodus detached) verarbeitet werden, verwendet. Analog zu den Pseudo-Transaktionen gibt es auch hier vier spezielle Befehle für die Modellierung. Dies sind BOT, EOT, COT und AOT. Sie haben im ARFPN die selbe Funktion wie ihre entsprechenden Befehle in Pseudo-Transaktionen.

2. INSTRUCTION-**Transaktionen**
   Jeder einzelne Befehl einer benutzerdefinierten Transaktion oder Aktionskomponente wird als INSTRUCTION-Transaktion verarbeitet. Diese erfüllt nur die 'Atomicity' der ACID-Eigenschaften. Für die Modellierung gibt es zwei spezielle Befehle:

   - BOI (Begin-Of-Instruction)
     Bei der Verarbeitung dieses Befehls wird eine neue INSTRUCTION-Transaktion als Subtransaktion einer ROOT- oder ACTION-Transaktion erzeugt. Bei Datenmanipulationsaktionen werden die zu bearbeitenden Datensätze bestimmt und in einer Übergangsrelation, die später für Zugriffe auf alte und neue Werte der Datensätze verwendet wird, gespeichert.
   - EOI (End-Of-Instruction)
     Bei der Verarbeitung dieses Befehls werden manipulierte Datensätze mit denjenigen der übergeordneten Transaktion zusammengefasst und die INSTRUCTION-Transaktion beendet.

3. `CONDITION`-**Transaktionen**
   Diese Transaktionsart wird für die Auswertungen von Regelbedingungen verwendet. Die Modellierung erfolgt mit folgenden zwei Befehlen:

   - `BOC` (Begin-Of-Condition)
     Mit diesem Befehl wird eine neue `CONDITION`-Transaktion als Subtransaktion erzeugt. Diese erhält von der übergeordneten Transaktion alle die Datensätze, die sie für die Auswertung der Bedingung benötigt.
   - `EOC` (End-Of-Condition)
     Der `EOC`-Befehl schliesst die Bedingungsauswertung ab und terminiert die Transaktion. Es werden keine Datensätze an die übergeordnete Transaktion zurückgegeben.

4. `ACTION`-**Transaktionen**
   Diese Transaktionsart wird für die Verarbeitung der Aktionskomponenten von Regeln verwendet. Sie erfüllt die ACID-Eigenschaften nicht. Für die Modellierung gibt es zwei Befehle:

   - `BOA` (Begin-Of-Action)
     Mit der Verarbeitung des `BOA`-Befehls wird eine neue `ACTION`-Transaktion als Subtransaktion erzeugt. Von dieser erhält sie die für die Ausführung der Aktionskomponente nötigen Datensätze.
   - `EOA` (End-Of-Action)
     Der `EOA`-Befehl beendet und terminiert die `ACTION`-Transaktion. Die manipulierten Datensätze werden an die übergeordnete Transaktion übergeben.

Durch diese Art der Modellierung, kann eine Befehl korrekt ausgeführt werden. Insbesondere lässt sich auch ein `cancel`-Befehl, mit dem die Verarbeitung eines regelauslösenden Befehls abgebrochen wird, korrekt verarbeiten.

Die folgenden Abschnitte zeigen, wie Befehle und Regeln im Detail zusammengesetzt sind und welche speziellen Plätze und Transitionen verwendet werden.

3.3.2 Modellierung von Befehlen

Das Petri-Netz für einen Befehl beginnt mit einem Platz und einer `BEGIN_ARFPN`-Transition, die über eine Kante verbunden sind. Der eigentliche Befehl (vgl.

Transition INSERT in Abbildung 2) wird durch die P_BOT- und die P_EOT-Transition gekapselt. Nach der P_EOT-Transition folgt der CHOICE-Platz, der abhängig vom Transaktionsstatus entweder die eine oder die andere ausgehende Transition feuert. Die Transition P_AOT wird gefeuert, wenn der Befehl durch eine ausgelöste Regel abgebrochen (aborted) worden ist. Sonst wird die P_COT-Transition gefeuert.

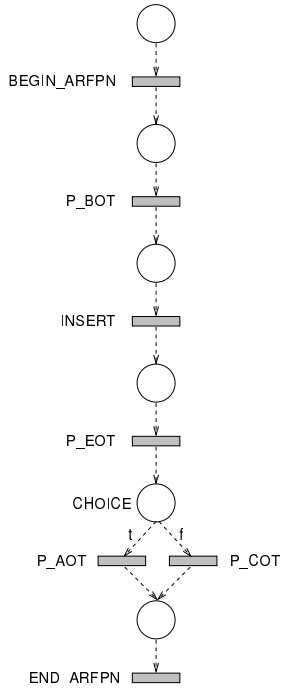


Abbildung 2: Struktur des Petri-Netzes für einen normalen Benutzerbefehl

3.3.3 Modellierung von benutzerdefinierten Transaktionen

Eine benutzerdefinierte Transaktion besteht aus mehreren Benutzerbefehlen. Das erzeugte Petri-Netz beginnt wie ein Befehl mit einem Platz und einer BEGIN_ARFPN-Transition. Alle Befehle werden zusammen durch die Transitionen BOT und EOT eingeschlossen (vgl. Abbildung 3). Jeder einzelne Befehl beginnt seinerseits mit einer BOI- und endet mit einer EOI-Transition. Dieses Konstrukt wird benötigt, um auch CANCEL-Aktionen, welche die Verarbeitung eines einzelnen Befehls abbrechen, realisieren zu können. Nach der EOT-Transition folgt der CHOICE-Platz, der analog zu oben eine der beiden

ausgehende Transition feuert. Das Netz wird ebenfalls mit einer `END_ARFPN`-Transition abgeschlossen.

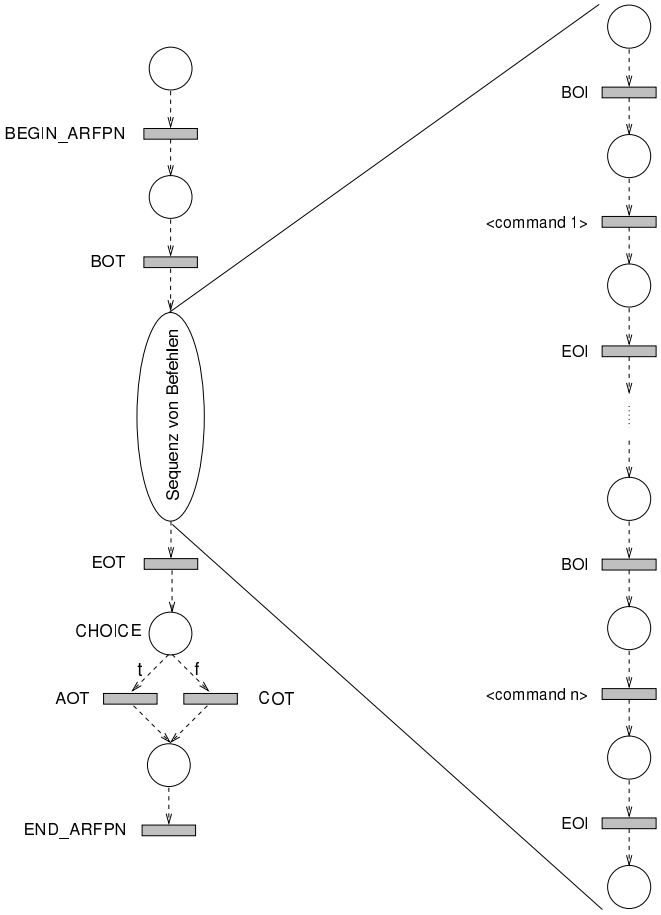


Abbildung 3: Struktur des Petri-Netzes für eine benutzerdefinierte Transaktion

3.3.4 Modellierung von Regeln

Die ARDL unterstützt die Definition von ECAA-, ECA- und EA-Regeln. Für die Verarbeitung werden diese analog zu Befehlen als ARFPN dargestellt. Anhand der in Abbildung 4 gezeigten Struktur einer ECAA-Regel werden die speziellen Konstrukte dieser Netze erläutert.

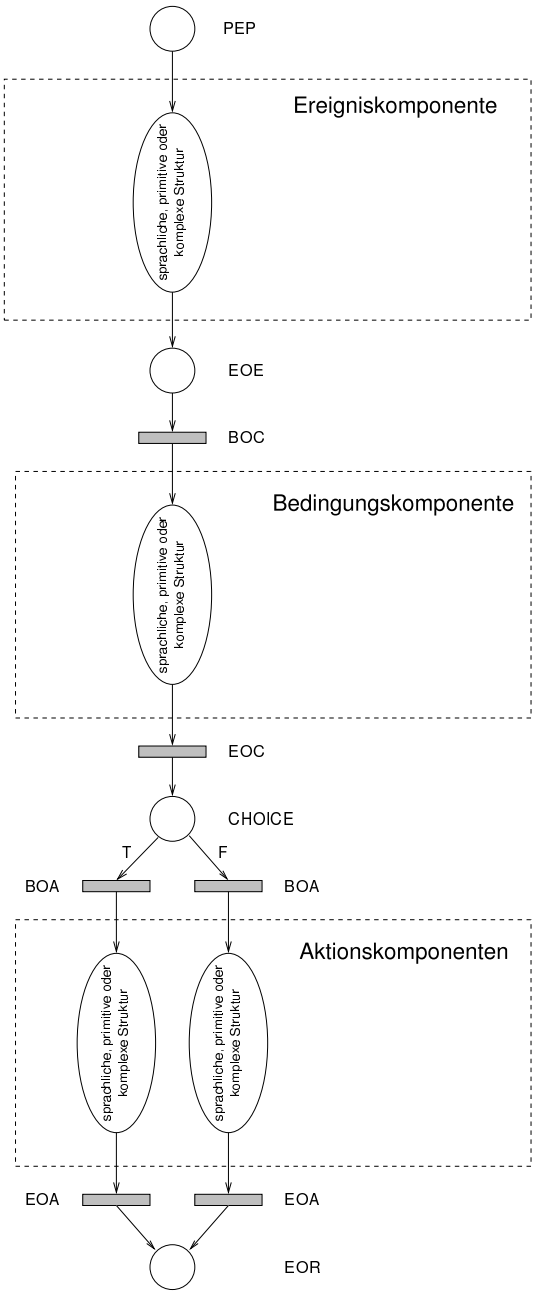


Abbildung 4: Struktur des Petri-Netzes für eine ECAA-Regel (aus [Sch98])

- **Ereigniskomponente**
  Ereignisse werden durch Plätze modelliert, denen der Ereignistyp als Name zugeordnet ist. Wenn ein Ereignis erkannt ist, erhält der entsprechende Platz eine Marke, in der u.a. die für die Verarbeitung benötigten Paramter gespeichert sind. Die Regeldefinitionssprache von ALFRED unterstützt sprachliche, primitive und zusammengesetzte Ereignisse, deren Modellierung in Abbildung 4 als Ellipse dargestellt ist. Begrenzt wird dieser Teil des Netzes durch zwei Arten von Plätzen:

  1. `Primitive Event Place (PEP)`
     Diese Plätze repräsentieren primitive Ereignisse. Sie werden mit dem Netz des auszuführenden Befehls verbunden.
  2. `End Of Event (EOE)`**-Platz**
     Dieser Platz verbindet Ereignis- und Bedingungskomponente einer Regel. Ein Ereignis gilt als eingetreten, wenn in diesem Platz eine Marke zu liegen kommt.

- **Bedingungskomponente**
  Bedingungen werden durch eine oder mehrere Transitionen repräsentiert. Auch die Bedingungskomponente einer Regel kann sprachlich, primitiv oder komplex sein. Mit zwei speziellen Transitionen und einem Platz wird sie mit der Ereignis- und den Aktionskomponenten verknüpft:

  1. `Begin Of Condition (BOC)`**-Transition**
     Diese Transition repräsentiert den Anfang der Bedingungskomponente. Sie ist mit dem `EOE`-Platz der Ereigniskomponente verbunden.
  2. `End of Condition (EOC)`**-Transition**
     Diese Transition beendet den Bedingungsteil. Sie ist mit einem `CHOICE`-Platz verbunden.
  3. `CHOICE`**-Platz**
     Der `CHOICE`-Platz ist mit genau zwei ausgehenden Transitionen verbunden. Bei der Verarbeitung wird eine Bedingung ausgewertet welche ein `true` oder `false` als Ergebnis liefert. Abhängig von diesem Ergebnis feuert die eine oder die andere Transition.

- **Aktionskomponente**
  Aktionen werden ebenfalls als Transitionen modelliert. Die Aktionskomponente kann gemäss der ARDL ebenfalls sprachlich, primitiv oder

zusammengestzt sein. Für die Integration in das ARFPN werden zwei spezielle Transitionen eingefügt:

1. `Begin Of Action (BOA)`-**Transition**
   Eine Aktionskomponente wird durch eine `BOA`-Transition eingeleitet. Diese Transition ist mit dem `CHOICE`-Platz verbunden.
2. `End Of Action (EOA)`-**Transition**
   Die `EOA`-Transition beendet eine Aktionskomponente. Sie ist mit dem Ende der Regel, dem `End Of Rule (EOR)`-Platz verbunden.

3.4 Architektur

Um alle an ALFRED gestellten Anforderungen erfüllen zu können, wurde eine erweiterte Schichtenarchitektur gewählt. Die Abbildung 5 zeigt das Modell von ALFRED, das aus zwei Subsystemen besteht: dem User-System und dem Processing-System. Ebenfalls dargestellt ist die Schnittstelle zum Datenbanksystem.

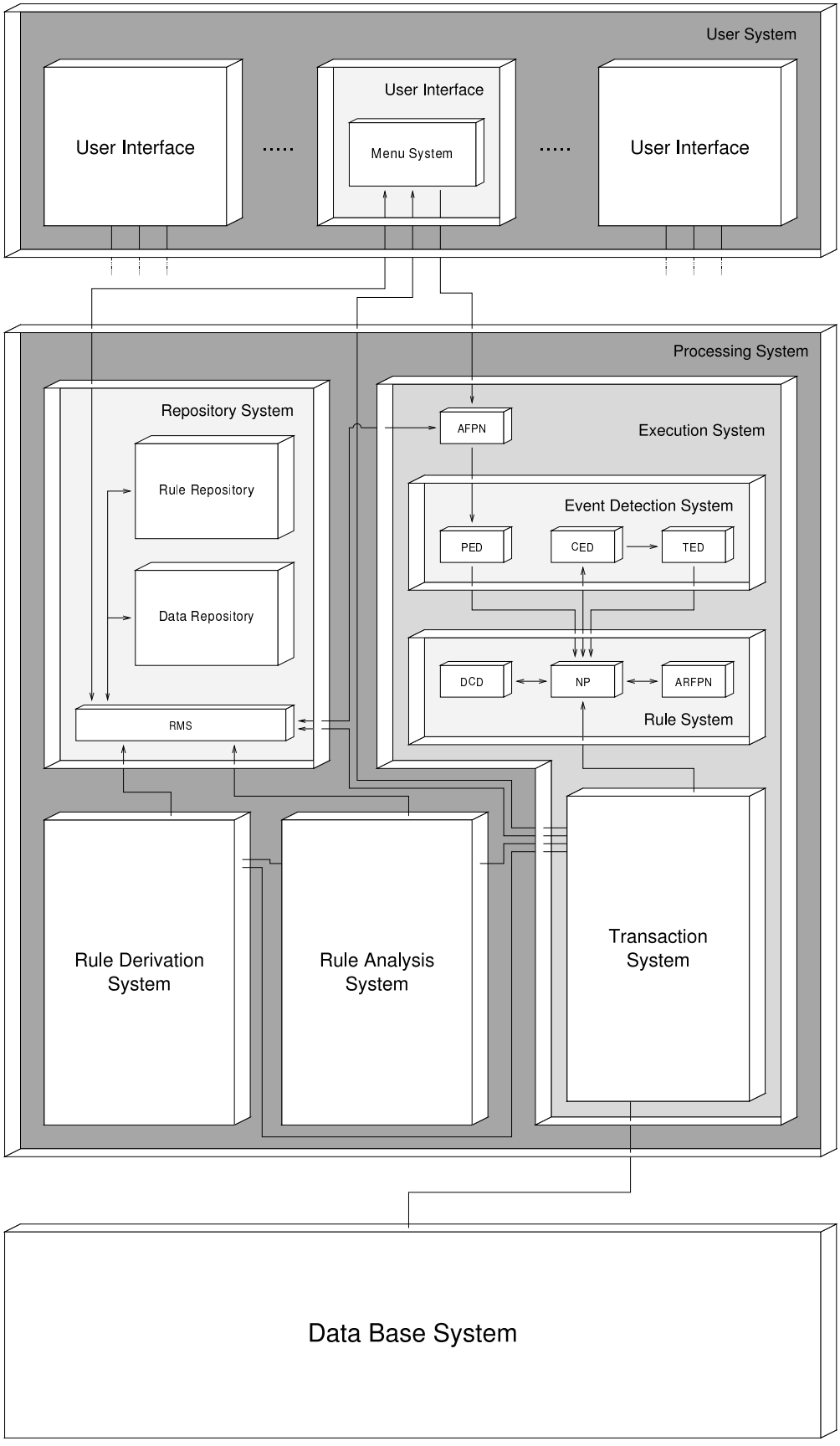
3.4.1 Benutzersystem (User System)

Das Benutzersystem beinhaltet den für den Benutzer sichtbaren Teil von ALFRED, eine grafische Benutzerschnittstelle (GUI). Dieses stellt dem Benutzer eine Oberfläche (zB. ein Menüsystem) zur Verfügung, über die er die Funktionalitäten von ALFRED, wie die Manipulation von Datenbanken, Objekten und Regeln, einfach nutzen kann (vgl. [Blu97]).

3.4.2 Verarbeitungssystem (Processing System)

Im Verarbeitungssystem werden Befehle und Regeln auf Basis dynamisch erzeugter und erweiterter ARFPN modelliert, analysiert und verarbeitet. Es besteht aus folgenden Subsystemen:

- **Ausführungsystem**
  Im Ausführungssystem (Execution System) erfolgt die eigentliche Verarbeitung der Regeln. Es besteht aus vier Modulen:
  1. *Modul 'AFPN-Generation' (AFPN)*
     Im Modul *AFPN* erfolgt die Transformation der Benutzereinga-


User System
User Interface
User Interface
Menu System
User Interface
Processing System
Repository System
Rule Repository
Data Repository
RMS
AFPN
Execution System
Event Detection System
PED
CED
TED
DCD
NP
ARFPN
Rule System
Rule Derivation System
Rule Analysis System
Transaction System
Data Base System

ben in ein erstes Ablaufmodell in Form eines einfachen Petri-Netzes, das als **A**ction **F**low **P**etri **N**et (AFPN) [Lö96] bezeichnet wird.

2. *Ereigniserkennungssystem*
   Im Ereigniserkennungssystem (Event Detection System) werden die definierten Ereignisse erkannt. Es wird unterschieden zwischen primitiven, komplexen und temporalen Ereignissen. Für jeden dieser Ereignistypen gibt es ein spezielles Modul:
   - Modul 'Primitive Event Detection' (PED)
     Das Modul PED erkennt primitive Ereignisse.
   - Modul 'Complexe Event Detection' (CED)
     Mit dem Modul CED werden komplexe Ereignisse erkannt.
   - Modul 'Time Event Detection' (TED)
     Zeitereignisse werden mit dem Modul TED erkannt.
3. *Regelsystem*
   Das Regelsystem (Rule System) besteht aus drei Moduln, durch welche die Petri-Netze verarbeitet werden:
   - Modul 'Net Processing' (NP)
     Das Modul *NP* steuert die Verarbeitung der generierten Petri-Netze.
   - Modul 'ARFPN-Generation' (ARFPN)
     Werden bei der Netzverarbeitung Regeln ausgelöst, so werden diese durch das Modul *ARFPN* ins bestehende Netz integriert. Auf diese Art entstehen aus den einfachen AFPN die komplexeren Action Rule Flow Petri Nets (ARFPN).
   - Modul 'Dynamic Cycle Detection' (DCD)
     Dieses Modul prüft, ob bei der Netzverarbeitung Zyklen auftreten und kontrolliert diese. Wenn nicht terminierende Zyklen entdeckt werden, so müssen diese auf eine bestimmte Art und Weise behandelt werden.
4. *Transaktionssystem*
   Im Transaktionssystem (Transaction System) werden Bedingungen ausgewertet und Befehle verarbeitet. Datenmanipulationsaktionen (wie das Einfügen, Ändern, Löschen und Auswählen von Datensätzen) werden an das mit ALFRED verbundene DBMS zur Ausführung weitergegeben.

- **Repository**
  Im Repository werden alle Informationen über definierte Regeln, Ob-

jekttypen, Benutzer usw. gespeichert. Das Repository besteht aus drei Teilen:

  - Data-Repository
    Das Data-Repository verwaltet alle nicht direkt zu Regeln gehörenden Informationen wie Benutzer, Privilegien, Objekttypen, usw.
  - Rule-Repository
    Dieser Teil des Repository-Systems verwaltet alle Informationen bezüglich Regeln (wie zum Beispiel Ereignisse und Bedingungen).
  - Modul 'Repository Management System' (RMS)
    Das Modul RMS koordiniert Zugriffe auf das Data- und das Rule-Repository und liefert gewünschte Informationen zurück.

- **Regelableitungsystem**
  Im Regelableitungssystem (Rule Derivation System) werden automatisch Regeln für gewisse Aufgabenbereiche erzeugt. Dazu gehören Regeln zur Überprüfung von Integritätsbedingungen und von Privilegien und Zugriffsrechten (vgl. [Blu97]).

- **Regelanalysesystem**
  Im Regelanalysesystem (Rule Analysis System) werden die Regeln analysiert, die durch den Benutzer oder durch das Regelableitungssystem definiert werden. Beispiele sind die Erkennung von Zyklen und deren Terminierung, Redundanzen und Konflikte (vgl. [Lö96]).

4 Entwurf des Prototypen

Nach diesen theoretischen Aspekten von aktiven Datenbanksystemen im allgemeinen und ALFRED im speziellen, folgen nun die Ausführungen zum praktischen Teil dieser Arbeit. Das folgende Kapitel beschreibt die Umsetzung der Konzepte des Verarbeitungssystems von ALFRED in Form eines Prototypen.

In einem ersten Teil wird erläutert, weshalb die objektorientierte Entwurfsmethode von Booch gewählt wurde. Zudem wird diese Methode kurz vorgestellt. Darauf folgt die funktionale Abgrenzung des Prototypen. Darin wird beschrieben, welche der in den Konzepten definierten Anforderungen durch den Prototypen abgedeckt werden. Anschliessend wird der Entwurf der einzelnen Subsysteme von ALFRED anhand von Klassendiagrammen und der Beschreibung wichtiger Algorithmen in Pseudo-Code erläutert. Zum Schluss werden noch einige Implementierungsaspekte dargelegt. Dies sind insbesondere die Schnittstellen zum GUI und zum DBMS.

4.1 Entwurfsmethode

Die Wahl der Entwurfsmethode ist von entscheidender Bedeutung, da diese das weitere Vorgehen im Entwicklungsprozess weitgehend bestimmt. Eine solche Methode muss ein *Vorgehen* definieren und eine *Notation* bereitstellen, um die Ergebnisse darzustellen. Das Vorgehen sollte möglichst flexibel gewählt werden können. Die Notation sollte einfach und gut verständlich sein, dabei aber genug Ausdruckskraft haben, um auch komplexere Zusammenhänge beschreiben zu können.

4.1.1 Wahl der Designmethode

Industriell oder kommerziell einsetzbare Software-Systeme beinhalten fast immer eine Vielzahl von Funktionen und Verhaltensweisen. Sie zeichnen sich zudem durch eine relativ lange Lebensdauer aus. Dies führt zu einer erhöhten Komplexität, welche häufig die Kapazität der menschlichen Intelligenz übersteigt. Diese Komplexität kann zwar überwunden, jedoch nicht ganz beseitigt werden.

> "Die Komplexität ist eine grundlegende Eigenschaft von Software und keine zufällige." [Bro87]

Booch leitet diese Komplexität, welche allen grösseren Software-Systemen inhärent ist, von vier Faktoren ab:

- **Komplexität der Problemstellung**
  Neben funktionalen Anforderungen gibt es meist auch (oft implizit) nicht-funktionale Anforderungen wie Handhabung, Performance, Kosten, usw., die sich zu einem grossen Teil konkurrenzieren.
- **Schwierigkeit der Steuerung des Entwicklungsprozesses**
  Grosse Entwicklungsprojekte können nur durch ein Team bewältigt werden. Dieses ist heute oft geografisch verteilt, woraus eine komplexe Koordination und Kommunikation resultiert.
- **Flexibilität von Software**
  In praktisch allen Bereichen der Fertigung ist es üblich, Standard-Bauteile zu verwenden. Dies gilt (noch) nicht für die Erstellung von Software-Systemen. Software bietet eine beliebige Flexibilität, welche Entwickler häufig verlockt, sämtliche Bausteine ihres Systems selbst zu entwickeln. Deshalb bleibt Software-Entwicklung stets ein arbeitsintensiver Prozess.
- **Verhalten diskreter Systeme**
  Digitale Computersysteme und somit auch Zustandsänderungen bei der Verarbeitung von Applikationen, die auf ihnen ausgeführt werden, sind *nicht stetig*, sondern *diskret*. Stetigkeit würde bedeuten, dass aus kleinen Änderungen der Eingaben auch kleine Änderungen der Ausgaben resultieren. Dies verhält sich aber bei diskreten Systemen nicht so. Sie können sich quasi sprunghaft verhalten, was ihre Beherrschung sehr schwierig macht.

Bei einer genaueren Betrachtung komplexer Systeme zeigt sich, dass diese die gleichen Eigenschaften aufweisen:

- **Hierarchie**
  Komplexität hat häufig eine hierarchische Struktur. Komplexe Systeme bestehen aus Subsystemen, die ihrerseits wiederum aus Subsystemen oder elementaren Komponenten bestehen [Cou85].
- **Wahl der primitiven Komponenten**
  Die Wahl der "primitiven" Komponenten eines komplexen Systems hängt stark vom Ermessen des Betrachters ab und ist relativ willkürlich [Rec85].

- **Starke Kohäsion**
  Die Beziehungen innerhalb einer Komponente sind stärker als die Beziehungen zwischen verschiedenen Komponenten [Sim82].
- **Nur wenige Arten von Subsystemen**
  Hierarchische Systeme setzen sich normalerweise aus wenigen Arten von Subsystemen zusammen. Diese sind auf viele verschiedene Arten kombiniert und unterscheiden sich in ihren Formen [Sim82].
- **Evolution**
  Ein komplexes System, das funktioniert, kann nicht von Grund auf entworfen werden. Es muss sich aus einem einfacheren, funktionierenden System entwickeln [Gal86].

Die menschliche Fähigkeit, mit Komplexität umzugehen, ist grundsätzlich beschränkt. Die Komplexität von Software-Systemen nimmt aber ständig zu. Laut Dijkstra ist die Technik, mit welcher Komplexität beherrscht werden kann, seit den Römern bekannt [Dij79]. Durch *divide et impera* (teile und herrsche) wird ein Problem in kleinere, überschaubarere Teilprobleme zerlegt. Auf diese Weise kann die Beschränktheit menschlicher Aufnahmefähigkeit umgangen werden.

Der traditionelle Weg ein Problem in Teilprobleme zu zerlegen, basiert auf einer *algorithmischen* Zerlegung. Ein Ergebnis davon ist ein Strukturdiagramm, das die Beziehungen zwischen einzelnen *funktionalen* Elementen einer Lösung darstellt. Diese Methode zur Lösung eines Problems wird auch *strukturierter Design* (SD) genannt [Boo97].

Ein neuerer Ansatz ist der *objektorientierte Design* (OOD). Hier stehen nicht die Algorithmen im Zentrum, sondern die *Objekte und ihre Beziehungen* zueinander. Objekte werden dabei als autonome Bestandteile verstanden. Eine Menge von Objekten führt gemeinsam eine übergeordnete Aufgabe aus.

Die Unterschiede zwischen strukturiertem und objektorientiertem Design werden verdeutlicht, wenn eine umgangsprachliche Formulierung eines Problems betrachtet wird. Beim strukturierten Design stehen die Verben, die angeben, was gemacht wird, im Vordergrund. Beim objektorientierten Design sind es die Substantive, die Auskunft darüber geben, wer was mit wem macht.

Zur Veranschaulichung wird ein Mail-Systems betrachtet. Eine übliche Funktion eines solchen Systems ist: “Verschiebe die Nachricht XY in die Ablage

der gelesenen Nachrichten".
Mit der algorithmischen Zerlegung ergibt sich aus dem Verb "verschiebe" recht schnell die Funktion, die diese Aktion durchführt: `verschiebe(Nachricht XY, Ablage Gelesene)`. Anders bei der objektorientierten Zerlegung. Durch den Fokus auf die Substantive, werden die Klassen `Nachricht` und `Ablage` gefunden. Die Aktion "verschieben" kann nun eine Operation von `Nachricht` oder `Ablage` sein.

Aus mehreren Gründen ist entschieden worden, ALFRED objektorientiert zu entwerfen und zu implementieren:

1. OOD ist besser geeignet für grosse und komplexe Problemstellungen, weil durch die objektorientierte Zerlegung kleinere, überschaubarere Systeme enstehen. Objektorientierte Systeme sind leichter anzupassen und weiterzuentwickeln, weil ihr Design auf stabilen Zwischenstufen beruht.

2. Das OO-Modell entspricht der Funktionsweise der menschlichen Kognition. Es ist daher anschaulicher und leichter nachvollziehbar.

3. Die wichtigsten Objekte (Subsysteme) von ALFRED wurden im Entwurf der Architektur (vgl. Kapitel 3.4) bereits gut identifiziert und deren Beziehungen zueinander festgelegt.

4. Die Entwicklung erfolgt aus Portabilitäts- und Performancegründen auf der Basis von C/C++ (vgl. [Blu97], Kapitel 5.1). C++ ist eine der Standard-Programmiersprachen für die objektorientierte Programmierung und durch Eigenschaften wie zB. strenger Typprüfung, C stark überlegen.

5. Als letztes gilt die Tatsache, dass OOD als Designmethode heute ein Quasi-Standard für den Entwurf von Software-Systemen darstellt.

4.1.2 Objektorientierte Methoden

Es gibt verschiedene bekannte OO-Methoden. Zu den bekanntesten gehören:

- **OMT**
  Die "Object Modeling Technique" wurde neben anderen von James Rumbaugh entwickelt. Den Kern dieser Methode bildet ein iterativer

Analyse- und Design-Prozess mit dem Schwergewicht auf der Analyse [Rum91].

- **Booch**
  Booch ist vergleichbar mit OMT insofern, dass er auch Analyse und Design als iterativen Prozess darstellt. Er legt das Schwergewicht jedoch auf den Design [Boo97].

- **Shlaer/Mellor**
  "Die" Methode für den Entwurf von Echtzeit-Systemen wurde von Sally Shlaer und Steven Mellor 1991 entwickelt [SM89, SM91]. Shlaer und Mellor aktualisieren regelmässig ihre Methode und veröffentlichten kürzlich ein "white paper" über die Verwendung der UML-Notation. Die Unified Modeling Language ist eine standardisierte Notation für die Modelierung von objektorientierten Systemen.

- **Fusion**
  Fusion ist eine Entwicklung von Hewlett Packard als ein erster Versuch einer Standardisierung von objektorientierten Methoden. Fusion vereinigt OMT und Booch mit CRC-Karten und formalen Methoden [CB94]. CRC-Karten (Class / Responsibilities / Collaborators = Klasse / Verantwortlichkeiten / Beteiligte) stellen ein einfaches Hilfsmittel für den Entwurf und die Darstellung von Klassen und ihren Beziehungen dar.

- **Catalysis**
  Catalysis ist eine weitere objektorientierte Methode auf der Basis von UML, die viele der neuesten Arbeiten über objektorientierte Methoden vereinigt und zusätzlich spezifische Techniken für das Modellieren von verteilten Komponenten beinhaltet [DC98].

Für den Entwurf von ALFRED wurde die Methode von Booch gewählt, weil diese sehr praxisorientiert, leicht verständlich und nachvollziehbar ist und das Schwergewicht auf den Design eines Systems legt.

4.1.3 Die Methode von Booch

Booch definiert objektorientiertes Design als

> "...Designmethode, die den Prozess der objektorientierten Zerlegung beinhaltet sowie eine Notation für die Beschreibung der

logischen und physikalischen wie auch statischen und dynamischen Modelle des betrachteten Systems" [Boo97].

Das Ergebnis des Designs stellt für ihn ein *Modell* dar, das es ermöglicht, die Strukturen des Systems zu diskutieren und Kompromisse zu finden, wenn Anforderungen im Konflikt miteinander stehen. Dieses Modell dient auch als Grundlage für die Implementierung.

Im folgenden wird nicht die gesamte Methode von Booch erläutert, sondern nur die wichtigsten Elemente der Notation von Booch dargestellt, die auch für die Visualisierung der Designergebnisse verwendet werden:

- **Klasse**
  Booch definiert Klasse als *"... eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen"* und verwendet in seiner Notation das in Abbildung 6 gezeigte Symbol.

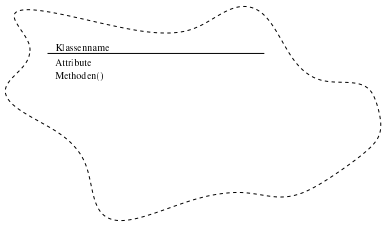


Abbildung 6: Klasse

- **Parametrisierte (generische) Klasse**
  Eine parametrisierte Klasse ist eine Klasse, die als Schablone für andere Klassen dient. Diese Schablone kann durch andere Klassen, Objekte und/oder Operationen parametrisiert werden.

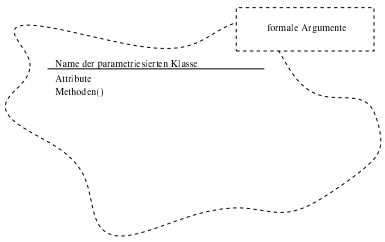


Abbildung 7: Parametrisierte Klasse

- **Assoziation**
  Die Assoziation gibt eine einseitige semantische Abhängigkeit an, aber nicht die Richtung der Abhängigkeit. Auch wird nicht genau festgelegt, auf welche Art zwei Klassen assoziiert sind. Diese Semantik ist für die Analyse eines Problems ausreichend, da zu diesem Zeitpunkt nur die Beteiligten einer Verbindung festgestellt werden sollen, sowie allenfalls die Kardinalitäten (vgl. Abbildung 8).

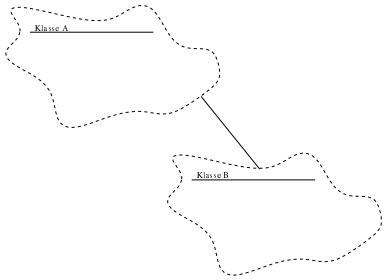


Abbildung 8: Assoziation

- **Vererbung**
  Durch Vererbung kann eine *Verallgemeinerung/Spezialisierung-Hierarchie* aufgebaut werden. Es wird unterschieden zwischen einfacher und mehrfacher Vererbung. Bei einfacher Vererbung erbt eine Klasse Attribute und Methoden von genau einer übergeordneten Klasse (Oberklasse). Bei der mehrfachen Vererbung werden von mehreren Oberklassen Attribute und Methoden übernommen. Die Abbildung 9 zeigt ein Beispiel, bei dem die Klasse B eine Spezialisierung (eine Unterklasse) der Klasse A ist. Entsprechend ist die Klasse A eine Verallgemeinerung (auch Generalisierung oder Oberklasse) der Klasse B.

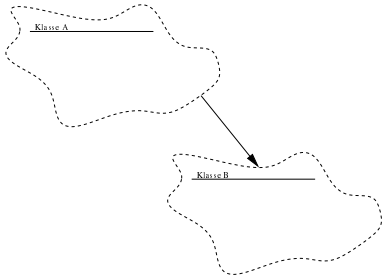


Abbildung 9: Vererbung

- **Aggregation**
  Die Aggregationsbeziehung zwischen Klassen beschreibt ein "physisches" Enthaltensein. Die Klasse, die eine andere enthält, umschreibt das Ganze, wovon die enthaltene Klasse ein Bestandteil ist. Durch die Aggregation wird eine *Ganzes/Teil-Hierarchie* dargestellt. Die Abbildung 10 zeigt ein Beispiel, in dem die Klasse B ein Bestandteil der Klasse A ist.

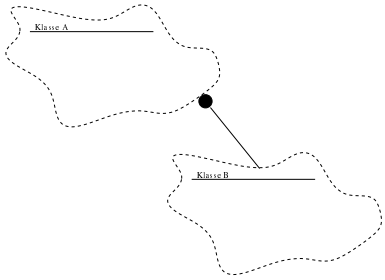


Abbildung 10: Aggregation

- **Verwendung**
  Eine Verwendungsbeziehung ist eine der möglichen Verfeinerungen der Assoziation. Es wird angegeben, welche der Klassen *Client* und welche *Supplier* ist. Im Beispiel von Abbildung 11) wird die Klasse B von der Klasse A verwendet. Das Symbol für die Verwendung ist eine Linie, die zwei Klassen verbindet. Die Richtung der Beziehung wird durch einen nicht ausgefüllten Kreis bei der Client-Klasse angegeben.

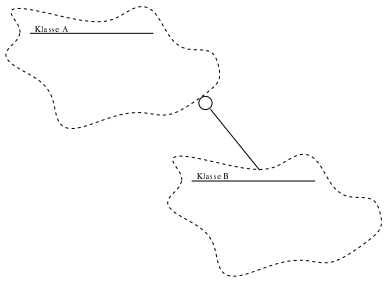


Abbildung 11: Verwendung

- **Polymorphismus**
  Polymorphismus ist ein wichtiges Konzept der objektorientierten Me-

thoden. Es erlaubt Objekten verschiedener Klassen, die eine gemeinsame Oberklasse haben, auf ihre eigene Art und Weise zu reagieren. Mit *virtuellen* Methoden kann in der Programmiersprache C++ Polymorphismus realisiert werden. Sie werden durch das in Abbildung 12 gezeigte Symbol bezeichnet.

Abbildung 12: Virtuelle Methoden

4.2 Funktionalitäten des Prototypen

In den Konzepten von ALFRED sind eine Vielzahl von Funktionalitäten kombiniert. Die Realisierung all dieser Eigenschaften sprengt aber den Rahmen einer Diplomarbeit. Aus diesem Grund ist festgelegt worden, welche Funktionalitäten und Konzepte des Systems umgesetzt werden sollen.

Ein Ziel der Prototyp-Entwicklung ist es, einen möglichst grossen Teil der in den Konzepten (vgl. Kapitel 3) festgelegten Anforderungen umzusetzen. Nur so können die Konzepte auf ihre Tauglichkeit hin untersucht werden.
Um den Entwicklungsaufwand aber besser kontrollieren zu können, wurde ein *inkrementelles* Vorgehen für die Realisierung gewählt. Ausgehend von einem Kern an minimaler Funktionalität, kann dieser schrittweise um weitere Funktionen erweitert werden.
Die folgenden Abschnitte beschreiben diesen minimalen Kern, der als Voraussetzung für die Messung des Laufzeitverhaltens, implementiert werden soll.

4.2.1 Funktionalitäten des Menüsystems

Die Anforderungen an das Menüsystem definieren eine Vielzahl von Funktionen, die vom Benutzer ausgeführt werden können. Für den Prototypen werden von diesen nur einige wenige benötigt:

1. **Datenbankbefehle**
   Um ALFRED überhaupt verwenden zu können, muss sich der Benutzer anmelden und mit einer Datenbank verbinden. Der Protyp muss

also die Befehle `logon` und `connect database` unterstützen, mit denen der Benutzer dies ausführen kann.

2. **Datenmanipulationsbefehle**
   Als Basis für benutzerdefinierte Transaktionen werden die Datenmanipulationsbefehle `retrieve`, `insert`, `update` und `delete` verwendet. Diese können aber auch einzeln ausgeführt werden.

3. **Benutzerdefinierte Transaktionen**
   Transaktionen sollen nicht nur definiert, sondern müssen vor allem auch ausgeführt werden können. Dies ist insbesondere für die Laufzeitanalyse notwendig. Also muss auch der Benutzerbefehl `execute transaction` vom Prototypen unterstützt werden.

4.2.2 Schnittstellen

Der im Rahmen dieser Arbeit implementierte Prototyp des Verarbeitungssystems hat zwei Schnittstellen. Dies sind die bestehende grafische Benutzeroberfläche und eine Datenbanksystem:

1. **Grafische Benutzerschnittstelle**
   Um die Verwendung von ALFRED möglichst einfach zu gestalten, wurde in einer Projektarbeit [Blu97] eine grafische Benutzerschnittstelle als Prototyp erstellt. Diese soll so an das Verarbeitungssystem von ALFRED angebunden werden, dass alle Funktionen des Prototypen verwendet werden können.

2. **Datenbanksystem**
   Als Datenbank für die Repositories und die Testdatenbank wird das auf dem Internet frei verfügbare DBMS *Ingres PD V 8.9* verwendet. Für die Wahl dieses einfachen DBMS sprachen vor allem die folgenden Gründe:

   - Für die Bedürfnisse von ALFRED genügt ein sehr einfaches DBMS.
   - Das verwendete DBMS braucht keine Transaktionsverwaltung zu unterstützen, da dies von ALFRED selbst übernommen wird. Somit können auch keine Seiteneffekte auftreten wie zum Beispiel, dass ALFRED eine Transaktion festschreibt, das DBMS aber nicht.

- *Ingres PD V. 8.9* ist nicht ressourcenintensiv und lässt sich auch auf einem kleinen Entwicklungssystem betreiben.
- Der vorhandene Precompiler gestattet eine einfache Einbettung der DBMS-Funktionen in ein C/C++-Programm.

4.2.3 Regelstrukturen

Von den drei ereignisbasierten Regelstrukturen, die von der ARDL unterstützt sind, werden alle (ECAA, ECA und EA) im Prototypen implementiert. Dabei werden die folgenden Arten von Regelkomponenten berücksichtigt:

1. **Ereignisse**
   Für den Prototypen werden nur ein Teil der primitiven Ereignisse als Auslöser von Regeln unterstützt:
   - Transaktionsereignisse
     Dies umfasst die Ereignisse `'begin of transaction'` und `'end of transaction'` sowie `'commit of transaction'` und `'abort of transaction'`.
   - Datenmanipulationsereignisse
     Dazu gehören die bereits weiter oben erwähnten Befehle `select`, `insert`, `update` und `delete` des Menüsystems.
2. **Bedingungen**
   Als Bedingungen können nur einfache Prädikate, wie zum Beispiel `Mieter.Mietkosten > 700` spezifiziert werden.
3. **Aktionen**
   Bei den Aktionen werden primitive und zusammengesetzte unterstützt. Dies ist notwendig, um eine gewisse Reaktion auf eingetretene Ereignisse zu ermöglichen. Beispielsweise kann eine Regel prüfen, ob ein neuer Datensatz gültige Werte für alle Attribute hat. Ist dies nicht der Fall, so soll dem Benutzer eine Nachricht geschickt werden und die Verarbeitung des Befehl abgebrochen werden. Hier liegt eine Folge von zwei Befehlen (`message` und `abort` resp. `cancel`) vor. Also müssen als Aktionskomponente von Regeln Sequenzen von primitiven Aktionen definiert werden können.

Befehle, die in den Aktionskomponenten von Regeln verwendet werden können, sind die Datenmanipulationsbefehle `select`, `insert`, `update` und `delete`, sowie die Transaktionsbefehle `cancel` und `abort` und die Meldungsaktion m̂essage.

4.2.4 Regelsemantiken

Im Rahmen der Prototyp-Entwicklung können nicht alle von der ARDL unterstützten Semantiken realisiert werden, sondern nur ein gewisser Teil daraus. Um sowohl Regeln, die sich auf einen Befehl beziehen, wie auch datensatzbezogene Regeln definieren und ausführen zu können, werden beide Granularitäten `set` und `instance` realisiert. Als Auslösungszeitpunkte für Regeln werden sowohl `pre` als auch `post` unterstützt. Um keine Nebenläufigkeiten berücksichtigen zu müssen, wird nur der Kopplungsmodus `immediate` implementiert. Aus dem gleichen Grund müssen die in ALFRED globalen Prioritäten ein Bestandteil des Prototypen sein. Mit ihnen kann, wenn mehrere Regeln durch ein Ereignis ausgelöst werden, die Reihenfolge der Ausführung festgelegt werden. Damit diese eindeutig ist, müssen die Prioritäten streng monoton festgelegt werden.

4.2.5 Analyse und Test

Zusätzlich zu den bisher erwähnten Funktionen werden einige weitere benötigt. Diese werden dazu verwendet, Fehler gezielt zu suchen und zu beheben, sowie Zeitmessungen durchzuführen:

1. **Log-File**
   Für ein effizientes Suchen und Beheben von Fehlern (Debugging) wird ein aussagekräftiges Log-File erstellt. Darin wird detailliert aufgezeichnet, welche Eingaben der Benutzer gemacht hat und wie diese vom System verarbeitet wurden. Als minimale Angaben werden die Quelle der Meldung (Modul), ein Zeitstempel sowie eine kurze Information festgehalten. Anhand dieser Log-Datei lässt sich auch überprüfen, ob das System Befehle und Regeln korrekt verarbeitet.

2. **Zeit-Log**
   Für die Gewinnung von Information bezüglich des Laufzeitverhaltens werden gewisse Zeitinformationen, wie zum Beispiel die Dauer von Transaktionen, in einem separaten Zeit-Log aufgezeichnet.

3. **Grafische Darstellung der erzeugten Petri-Netze**
   Um zu überprüfen, ob die erzeugten Petri-Netze den Vorgaben entsprechen, ist es am anschaulichsten, diese grafisch darzustellen.
   Mit dem Programm XFig, einem Vektor-Zeichenprogramm auf Unix, kann dies auf einfache Weise erfolgen. Dazu ist eine Textdatei mit den entsprechenden Kommandos zu erzeugen.

4.3 Entwurf der Klassen

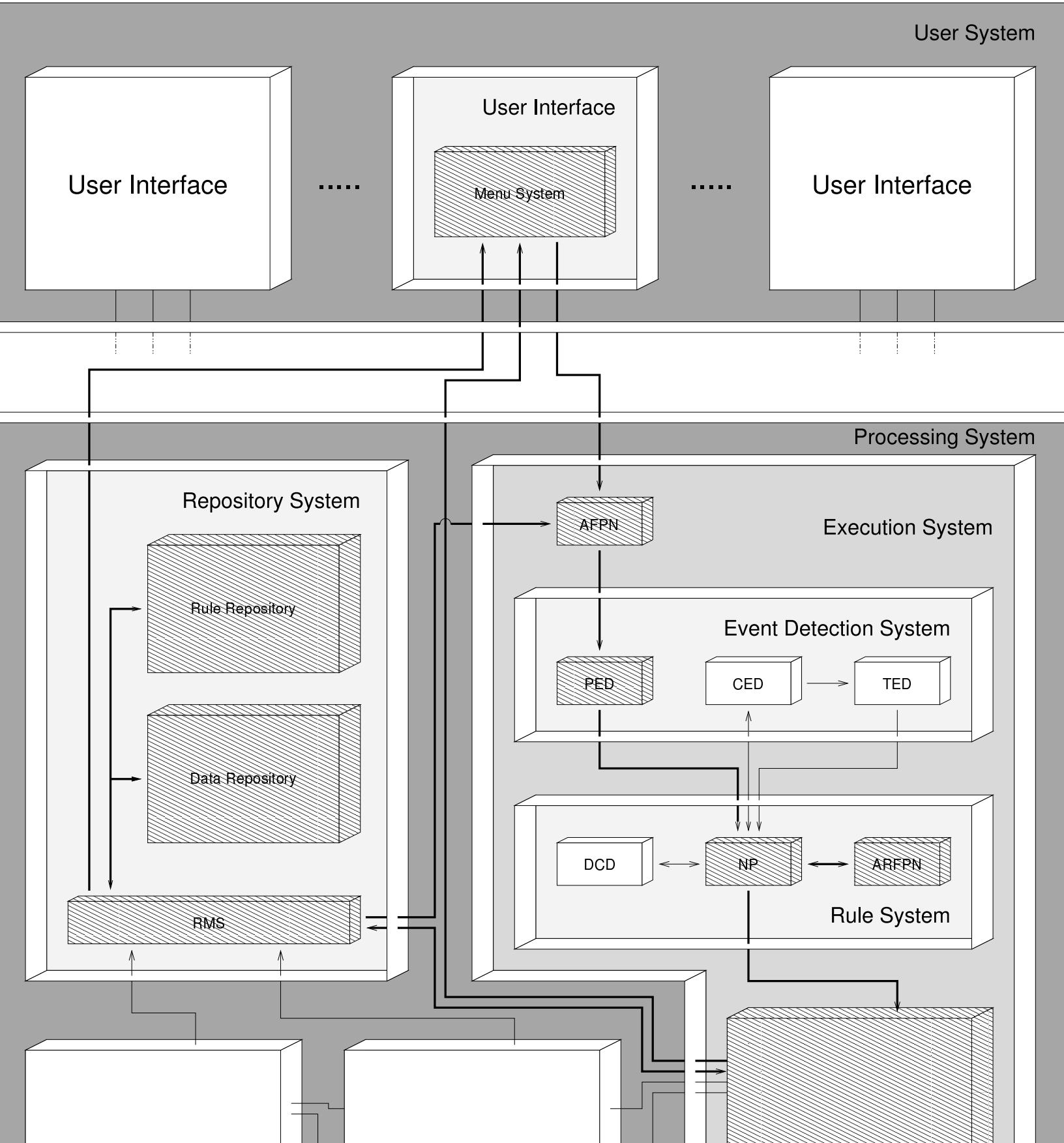
Für eine Implementierung dieser Funktionen sind nicht alle der in der Architektur aufgeführten Subsysteme und Module zu programmieren. Die Abbildung 13 verdeutlicht welche Elemente der Architektur von ALFRED Bestandteil des Prototypen sind. Diese sind schraffiert hervorgehoben. Jedes Modul ist als eine oder mehrere Klassen realisiert. Zusätzlich zu diesen Klassen werden einige weitere für spezielle Aufgaben benötigt.

Im folgenden werden die wichtigsten Eigenschaften, Operationen und Beziehungen dieser Klassen erläutert. Die Spezifikation aller Klassen ist zudem im Anhang C enthalten. Zu jeder Klasse wird ein Diagramm in der Notation von Booch [Boo97] gezeigt, das vor allem die Beziehungen zwischen den Klassen verdeutlichen soll. Wichtige, nicht-triviale Algorithmen werden in Pseudo-Code dargestellt.

4.3.1 Grundlegende Klassen

Dieser Abschnitt beschreibt zwei grundlegende Klassen des Prototypen. Sie werden für die Verwaltung von Subsystemen bzw. Modulen und Datenlisten verwendet:

1. **Klasse `subsys`**
   Die Klasse `subsys` verwaltet alle vorhandenen Subsysteme von ALFRED. Durch das Instantiieren eines Objekts dieser Klasse werden auch die Subsysteme erzeugt. Durch die Tatsache, dass sämtliche Subsysteme Zugriff auf diese Instanz der Klasse `subsys` haben, ist auch der Zugriff auf jedes andere Subsystem gewährleistet. Zudem stellt die Klasse `subsys` Dienste zur Verfügung, die von allen gemeinsam genutzt werden, wie etwa das Log-File.


User System
User Interface
User Interface
Menu System
User Interface
Processing System
Repository System
Rule Repository
Data Repository
RMS
AFPN
Execution System
Event Detection System
PED
CED
TED
DCD
NP
ARFPN
Rule System

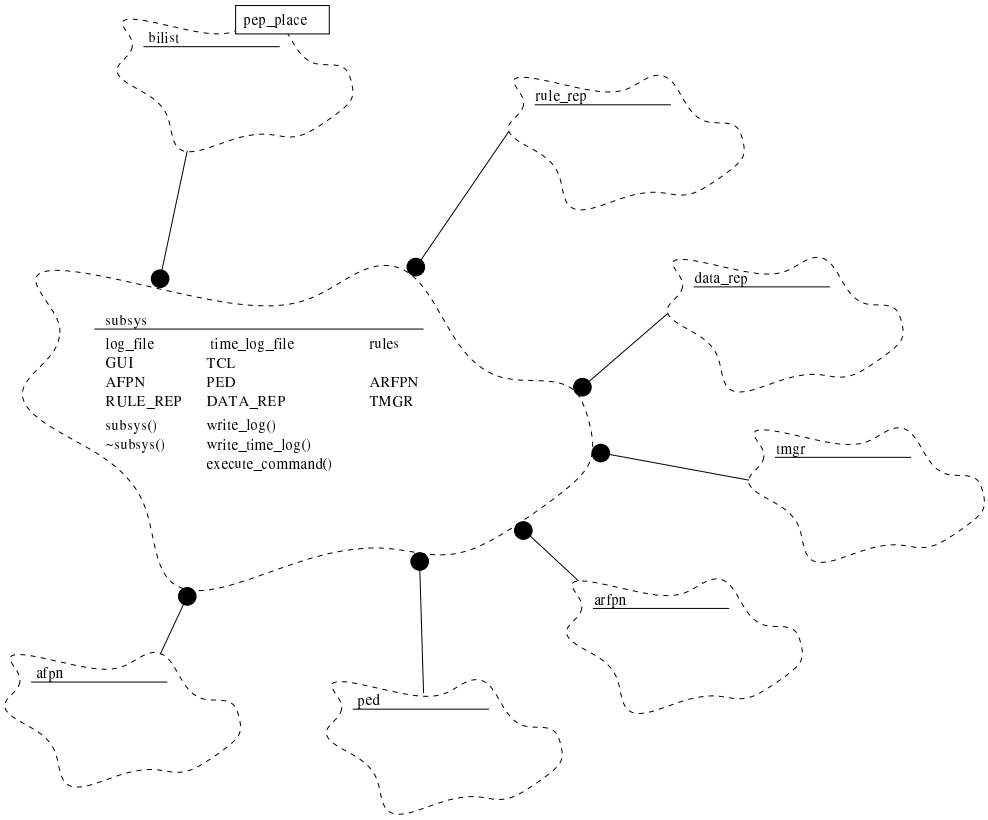


Abbildung 14: Diagramm der Klasse `subsys`

Die wichtigste Methode von `subsys` ist `execute_command()`. Der Algorithmus dieser Methode ist wie folgt:

```
Write message to log-file: STARTED
Build AFPN by calling afpn.run()
If net exists
    Detect primitive events by calling ped.run()
    If no error has occured
        Fire the first transition of the net
        Write message to log-file: TERMINATED
        Delete net
    Else if error
        Error-message
    End if
Else if no net exists
    Error-message
End if
```

2. **Klasse** `bilist`
   Die Klasse `bilist` ist eine parametrisierte Klasse, die den abstrakten Datentyp "doppelt verkettete Ringliste" implementiert. Sie wird in ALFRED immer dann verwendet, wenn es gilt, Informationen in einer Liste zu sammeln. Zu diesem Zweck wird jeweils eine Klassen-Instanz (mit dem zu speichernden Objekttypen als Argument) erzeugt. Beispiele für solche Klasseninstanzen sind Listen mit ganzen Zahlen oder Listen mit Objekten der Klasse `place` (siehe weiter unten). Zur Laufzeit werden dann Objekte dieser Klassensinstanzen instanziiert.

   Die doppelt verkettete Ringliste ist sicher nicht die effizienteste Art, Daten zu speichern. Sie ist aber sehr einfach zu implementieren und erfüllt die Anforderungen, die der Prototyp an die Datenhaltung stellt.

   Die Abbildung 15 zeigt einige Klassen, die eine Instanz der Klasse `bilist` enthalten. Wegen der Übersichtlichkeit werden nur die wichtigsten aufgeführt.

4.3.2 Modul AFPN

Das Modul AFPN erzeugt aus dem vom Benutzer spezifizierten Befehl ein Action Flow Petri Net. Dabei handelt es sich um eine einfache Kette von

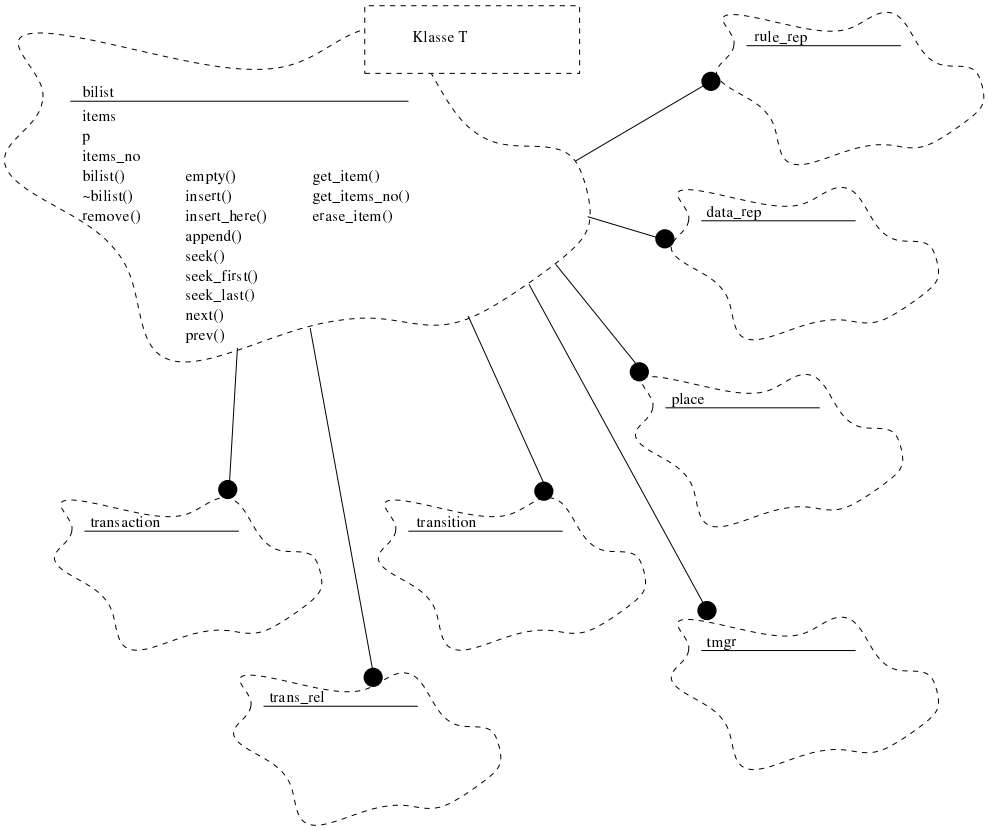


Abbildung 15: Diagramm der Klasse `bilist`

Plätzen und Transitionen, die später vom Modul PED (vgl. Abschnitt 4.3.3) um Elemente zur Regelauslösung erweitert wird.

Es gibt grundsätzlich zwei verschiedene Typen von Action Flow Petri Nets, die ebenfalls (gleich wie das Modul) als AFPN bezeichnet werden:

- **Befehle**
  Dazu gehören beispielsweise `create rule`, `delete rule`, `insert` und `delete`. Die Abbildung 16 zeigt die Struktur des erzeugten Petri-Netzes.

- **Benutzerdefinierte Transaktionen**
  Benutzerdefinierte Transaktionen fassen mehrere Befehle zu einem Verbund zusammenfassen. Die *ACID-Eigenschaften* solcher Transaktio-

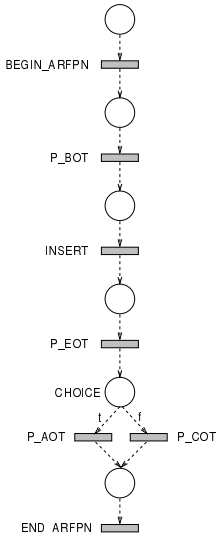


Abbildung 16: Einfacher Befehl als AFPN

nen gewährleisten, dass entweder alle oder keine Änderungen der Daten durch einen der Befehle festgeschrieben werden. Ein Beispiel für ein Modell einer Transaktion als Petri-Netz ist in Abbildung 17 gezeigt.

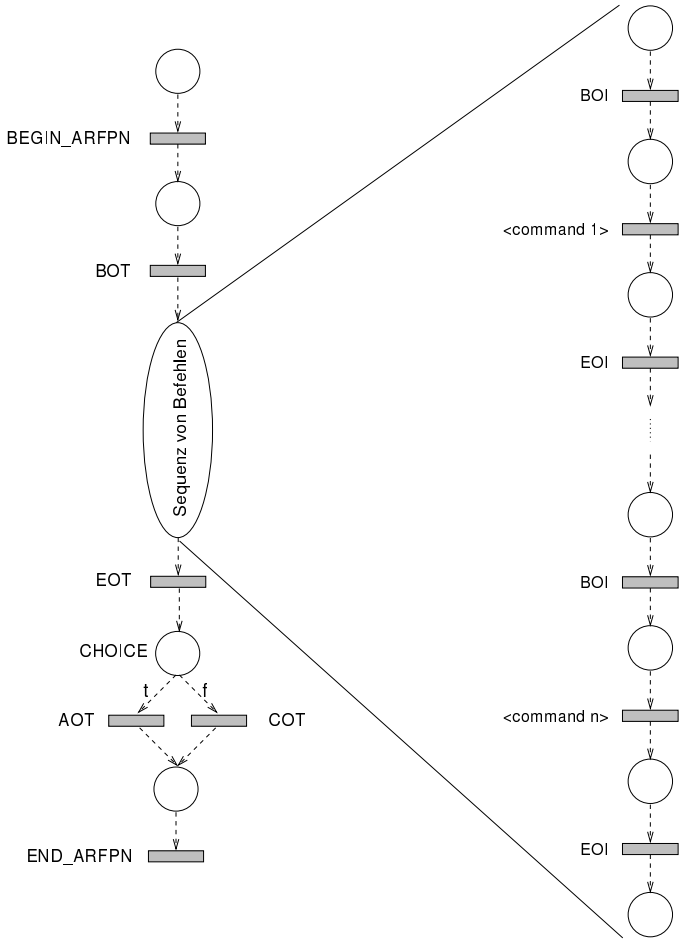


Abbildung 17: Benutzerdefinierte Transaktion als AFPN

Das Modul AFPN ist eine eigenständige Klasse und wird von `subsys` instanziiert (vgl. Abb. 18). Der Aufbau der Action Flow Petri Nets erfolgt durch den Aufruf der Methode `run()`.

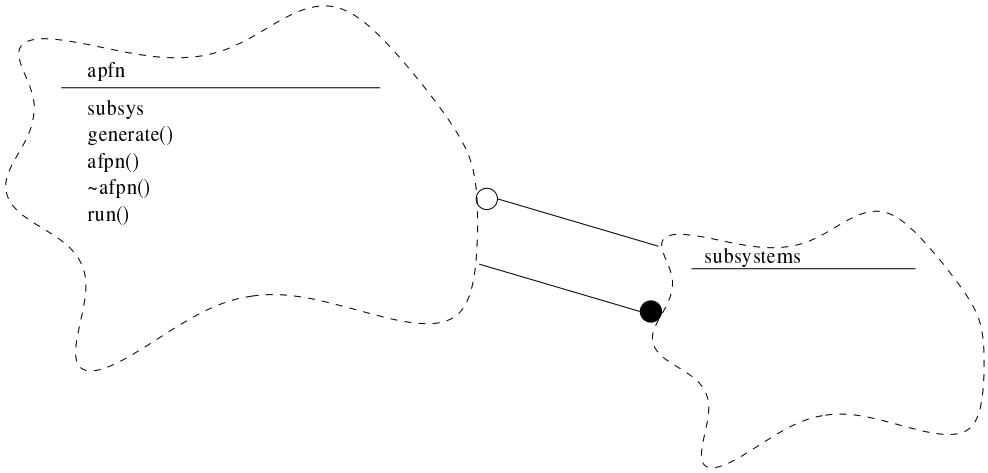


Abbildung 18: Diagramm der Klasse `afpn`

Die einzige nicht-triviale Methode des Moduls `AFPN` ist `generate()`. Mit dieser wird ein Petri-Netz erzeugt. Die Verarbeitung dieser Methode erfolgt nach dem folgenden Algorithmus:

```
Begin net with a place and a BEG_ARFPN-transition
If command is ‘‘execute transaction’’
     Load transaction from repository
     Append a place and a BOT-transition
     For each command
          Append a place and a BOI-transition
          Append a place and the command-transition
          Append a place and a EOI-transition
     End for
     Append a place and a EOT-transition
     Append a CHOICE-place
     Append a COT-transition on TRUE-branch
     Append a AOT-transition on FALSE-branch
Else if it’s a simple command
     Append a place and a P_BOT-transition
     Append a place and the command-transition
```

```
        Append a place and a P_EOT-transition
        Append a CHOICE-place
        Append a P_COT-transition on TRUE-branch
        Append a P_AOT-transition on FALSE-branch
    end if
    Append a place and a END_ARFPN-transition
    Create a token with specified data
    Put the token into the first place
    Return pointer to first place as result
```

4.3.3 Modul PED

Das Modul PED dient der Erkennung von primitiven Ereignissen. Wird ein solches erkannt, so wird eine `pre-set`, `pre-inst`, `post-inst` oder `post-set` Transition sowie ein zusätzlicher Platz (`PASSIVE-PLACE`) eingefügt. Anschliessend wird die eingefügte Transition mit dem assoziierten PEP im *Rule System* verbunden (vgl. Abbildung 19). Ist das AFPN vollständig kontrolliert, kann die erste Transition gefeuert werden.

PED ist ein eigenständiges Modul von ALFRED und wird beim Initialisieren des Objekts `subsys` ein einziges mal instanziiert (vgl. Abbildung 20). Die Erkennung der primitiven Ereignisse erfolgt durch den Aufruf der Methode `run()`:

```
For all transitions in the petri-net
     If it's a transaction-transition
          Check for rules triggered by transaction-transitions
     Else if it's a command-transition
          Check for rules triggered by command-transitions
     End if
     If place after transition is a CHOICE-place
          Check next transition in true-out-branch for rules
            triggered by transaction-transitions
          Check next transition in false-out-branch for rules
            triggered by transaction-transitions
     End if
     Goto next transition
End for
Write xfig-file
```

4.3.4 Modul NP

Die Netzverarbeitung (Modul NP) ist im Gegensatz zu AFPN-Generierung und Erkennung von primitiven Ereignissen nicht als ein Modul realisiert. Vielmehr wird die gesamte Funktionalität in die Plätze und Transitionen der Petri-Netze verlegt. Dadurch wird erreicht, dass der Fluss der Marke durch das Netz nicht permanent überwacht und gesteuert werden muss. Durch den rekursiven Aufruf der Methoden zum Transportieren der Marke wird erreicht, dass eine Marke das ganze Netz durchläuft. Dies ist möglich, weil die eigentliche Funktionalität der Petri-Netze sehr beschränkt ist und nur wenige Spezialfälle abweichend vom Standardverhalten modelliert werden müssen. Ein Beispiel dafür sind die passiven Plätze, die keine nachfolgende Transition feuern.

Petri-Netze bestehen aus den vier Elementen *Plätze*, *Transitionen*, *Kanten* und *Marken* (Token). Um eine grosse Flexibilität zu erreichen, werden bei den Plätzen und Transitionen je verschiedene Arten unterschieden, die als Klassenhierarchien entworfen sind (vgl. Abbildungen 21 und 22). Die Kanten werden nicht explizit modelliert, sondern implizit als Verbindung eines Platzes mit einer Transition (und umgekehrt) dargestellt.

1. **Klasse** `place`
   Die Klasse `place` implementiert das Standardverhalten von Plätzen.

Dies besteht im wesentlichen darin, einen erhaltenen Marken an eine ausgehende Transition weiterzugeben. Die Subklassen zeigen ein zum Teil davon abweichendes Verhalten.

- Klassen `passive_place` und `eoe_place`
  Plätze dieses Typs versuchen nicht, eine nachfolgende Transition zu feuern, sondern warten darauf, dass die Marke von einer der ausgehenden Transitionen angefordert wird.
- Klasse `choice_place`
  Plätze dieses Typs geben die Marke an eine bestimmte ausgehende Transition weiter. Dies erfolgt aufgrund einer Bedingung, die entweder `true` oder `false` liefert.
- Klasse `pep_place`
  Ein `pep_place` versucht eine ausgehende Transition nur zu feuern, wenn der Status der aktuellen Transaktion nicht `cancelled` oder `aborted` ist. Damit wird verhindert, dass Regeln ausgeführt werden, die nicht mehr verarbeitet werden dürfen, weil die Transaktion bereits abgebrochen wurde.

Der Algorithmus der Methode `fire_trans()`, die versucht eine ausgehende Transition zu feuern, ist wie folgt:

```
For all outgoing transitions
     Get the transition
     If transition can fire
          Make transition fire
          Return 0
     Else
          Find next transition
     End if
End for
Return 1 if no transition has fired
```

2. **Klasse** `transition`
   Die Klasse `transition` implementiert das Standard-Verhalten von Transitionen. Dies besteht im wesentlichen darin, auf Veranlassung durch einen eingehenden Platz, von allen eingehenden Plätzen eine Marke zu beziehen. Kann von jedem Platz eine Marke bezogen werden, so werden diese gesammelt und zur Verarbeitung an den Transaktionsmanager (vgl. Klasse `tmgr`) geschickt. Dieser liefert als Ergebnis einen neuen Token zurück. Alle ausgehenden Plätze erhalten anschliessend eine Kopie dieses Tokens.
   Die Subklassen zeigen ein zum Teil davon abweichendes Verhalten:

   - Klasse `replication_trans`
     Die Replikations-Transition folgt in Petri-Netzen von Regeln auf den PEP. Sie erzeugt einen Token für jede durch dieses Ereignis ausgelöste Regel. Nachdem jeder ausgehende Platz einen Token erhalten hat, wird das Modul ARFPN (vgl. Klasse `arfpn`) aufgerufen, welches ausgelöste Regeln in das entsprechende Netz integriert.

   Der Algorithmus der Methode `fire_token()`, die eine Transition feuert, ist wie folgt:

```
Check if every incoming place has a token to deliver
If true
     Get a token from every incoming place
     Insert token in token-list
```

```
        Call transaction-manager to process the transition
        If the transaction-manager returns a token
             For each outgoing place
                  Create a copy of the token
                  Send the copy to the outgoing place
             End for
             Delete the token received from transaction-manager
             Return 0 (Firing ok)
        Else
             Return 1 (couldn't fire)
        End if
   End if
```

3. **Klasse** `token`
   Die Marke (Token) ist das dynamische Element in den Petri-Netzen. Sie wird durch das Feuern einer Transition von einem Platz zu einem nächsten Platz verschoben. Die Marke dient dazu, Informationen zur Verarbeitung des Netzes zu speichern und liefert dem Transaktionsmanager Angaben darüber, wie die aktuelle Transition zu verarbeiten ist.

   Da die Marke nur Informationen speichert, gibt es nur Methoden mit trivialen Algorithmen zum Setzen und Abfragen von Attributen. Daher wird hier auf eine Angabe verzichtet.

4.3.5 Modul ARFPN

Das Modul ARFPN wird aufgerufen, wenn im Event-System (vgl. Abbildung 24) die Marke(n) den jeweiligen `end-of-event`-Platz (EOE) erreicht haben. Es wird für jede Regel, die in ihrem `EOE`-Platz eine Marke hat (dies bedeutet, das Ereignis ist eingetreten), eine Instanz erzeugt ((1) in Abb. 24). Diese wird dann unter Berücksichtigung von Prioritäten ((2) in Abb. 24) und Kopplungsmodi ins bestehende AFPN eingebunden ((3) in Abb. 24).

ARFPN ist ein eigenständiges Modul von ALFRED und wird beim Initialisieren des Objekts `subsys` genau einmal instanziiert (vgl. Abbildung 25).

Die wichtigste Methode des Moduls ARFPN ist `run()`. Diese wird durch eine Replikations-Transition (folgt in einer Regel auf den PEP-Platz) aufgerufen, wenn sie an alle ausgehenden Plätze eine Marke geschickt hat. Durch diese Methode wird festgestellt, welche Regeln ausgelöst sind (d.h. im EOE-Platz einen Token haben):

```
Get rule-informations from PEP
For each rule after replication-transition
     If EOE-place has a token
          Make a copy of the rule
          Move token to EOE of the copied rule
     End if
End for
Integrate rules with consideration of priorities and coupling-modes
If granularity is instance
     Set the record counter in the last place of all copied rules
     For each record
          Make a copy of the token of every rule
          Fire the first transition of the first rule
     End for
Else if granularity is set
     Fire the first transition of the first rule
End if
```

4.3.6 Subsystem Transaktionsverwaltung

Das Subsystem für die Transaktionsverwaltung bildet neben der Netzverarbeitung den eigentlichen Kern des Prototypen. Da es im Rahmen dieser Diplomarbeit nicht möglich ist, eine komplette Transaktionsverwaltung mit allen Eigenschaften wie Persistenz, Parallelität, Sperren, 'lange' Transaktionen, usw. zu realisieren, bildet die Klasse `tmgr` einen umfangreichen *Treiber*, der es erlaubt, Transaktionen zu erzeugen und zu verwalten. Weiter werden Transitionen verarbeitet und Datenbankzugriffe (lesend und schreibend) durchgeführt. Dieser Treiber muss also genügend intelligent sein, um eine Verarbeitung der Petri-Netze, wie sie für den Prototypen vorgesehen ist, sicherzustellen.

Im folgenden werden die Klassen für die Realisierung des in Kapitel 3.3 beschriebenen Transaktionskonzepts erläutert. Dabei werden nur die Basisfunktionalitäten angegeben:

1. **Klasse** `record`
   Ein Datensatz (`record`, vgl. Abb. 26) zur Speicherung von Datenmanipulationen enthält drei Versionen der Daten (Übergangswerte) und ein Attribut zur Symbolisierung des Status:

   - RET
     Dies sind die Werte, die von der Datenbank für diesen Datensatz gelesen wurden. Neu eingefügte Datensätze enthalten keine Werte, sondern einen eindeutigen Stempel, der aus der Transaktionsnummer erzeugt wird.
   - OLD
     Dies sind die Werte des Datensatzes vor der letzten Änderung.
   - NEW
     Dies sind die aktuellen Werte des Datensatzes.

   Der Status gibt Auskunft darüber, ob der Datensatz bereits geändert (`changed`) oder gelöscht (`deleted`) oder ob er neu eingefügt (`inserted`) worden ist.

   Die Klasse `record` verfügt nur über triviale Methoden zum Lesen und Setzen der Attributwerte.

2. **Klasse** `trans_rel`
   Eine Übergangsrelation (`trans_rel`, vgl. Abb. 27) verwaltet eine Liste von Datensätzen. Die wichtigsten Methoden dieser Klasse sind:

- `append_record()`
  Mit dieser Methode wird gemäss folgendem Algorithmus ein neuer Datensatz an eine Übergangsrelation angefügt:

```
Find record with same RET-values
If not found
     Find highest record-number used
     Increment it by 1
     Insert record in relation
     Return record-number
Else
     Return 0
End if
```

- `get_record_values()`
  Diese Methode bestimmt alle Bestandteile eines Datensatzes. Für die Verarbeitung wird der folgende Algorithmus verwendet:

```
Find record with given record-number
If found
     Set parameter ret-val
     Set parameter old-val
     Set parameter new-val
     Set parameter state
     Return record-number
Else
     Return 0
End if
```

3. **Klasse** `transaction`
   Eine Transaktion (`transaction`, vgl. Abb. 28) verwaltet zwei Listen von Übergangsrelationen und eine Liste von Datensätzen, die der Benutzer zum Anzeigen selektiert hat. Diese drei Listen werden durch Objekte der Klasse `trans_rel` repräsentiert:

   - CURRENT-Übergangsrelationen (CDS)
     Diese Übergangsrelationen enthalten alle Datensätze, die durch Befehle dieser Transaktion betroffen sind.
   - OTHER-Übergangsrelationen (ODS)
     Diese Übergangsrelationen enthalten die Datensätze, die durch Befehle untergeordneter Transaktionen geändert wurden. Diese

werden beim Terminieren der Subtransaktion mit den bereits vorhandenen Datensätzen 'verschmolzen'. Das bedeutet, Datensätze, die bereits in den Übergangsrelationen vorhanden sind und in einer Subtransaktion verändert wurden, werden mit den neuen Werten aktualisiert. Datensätze, die bisher nicht in den Übergangsrelationen vorhanden waren, werden neu eingefügt.

- RETRIEVED-Relationen (RDS)
  Diese Relationen enthalten die Datensätze, die dem Benutzer angezeigt werden.

Der Status gibt an, ob die Transaktion läuft (`running`), wartet (`waiting`) oder abgebrochen (`cancelled` oder `aborted`) ist. Der Unterschied zwischen `cancelled` und `aborted` liegt darin, dass entweder der gesamte Transaktionsbaum (inkl. ROOT) oder nur ein Teil (der Befehl, der ROOT-Transaktion, bei dem der `cancel`-Befehl aufgetreten ist), verworfen wird. Die übrigen Attribute der Klasse `transaction` dienen der Bildung des Transaktionsbaums. Dies sind:

- `top_ta`
  Die Transaktionsnummer der Wurzel des Transaktionsbaums.
- `sup_ta`
  Die Nummer der übergeordneten Transaktion.
- `dep_ta`
  Eine Liste von untergeordneten Transaktionen.

Wichtige Methoden der Klasse `transaction` sind:

- `merge_record_curr_data()`
  Diese Methode aktualisiert Datensätze in CDS mit Datensätzen einer Subtransaktion. Die Verarbeitung erfolgt gemäss folgendem Algorithmus:

```
Find concerned relation in CDS
If found
     Find corresponding record in this relation
     If found
          If record is to delete
               Set record state to deleted
          Else if record is to change
               Set new values
               Set record state to changed
          End if
          Set flag: record could be merged
     End if
End if
If record could not be merged
     Merge the record in ODS
End if
```

- `merge_record_other_data()`
  Diese Methode aktualisiert Datensätze in ODS mit Datensätzen einer Tochtertransaktion. Dies wird gemäss folgendem Algorithmus durchgeführt:

```
Find concerned relation in ODS
If found
     Find corresponding record in this relation
     If found
          If record is to delete
               Set record state to 'deleted'
          Else if record is to change
               Set new values
               Set record state to 'changed'
          End if
     Else
          Append the record to the relation
     End if
Else
     Append a new relation to ODS
     Append the record to the relation
End if
```

4. **Klasse** `tmgr`
   Die Klasse `tmgr` (vgl. Abb. 29) verwaltet eine Liste von Transaktionen. Der `tmgr` wird von jeder Transition im Petri-Netz aufgerufen und verarbeitet die mit den Marken gelieferten Informationen entsprechend dem Transitionstyp. Er gibt der aufrufenden Transition als Ergebnis eine Marke mit den aktuellen Werten (wie zB. Transaktions-Status) zurück.

   Wichtige Methoden der Klasse `tmgr` sind:

   - `commit_transaction()`
     Diese Methode schreibt die Änderungen, die durch einen ganzen Transaktionsbaum durchgeführt wurden, auf der Datenbank fest und zeigt dem Benutzer die Daten an, die selektiert wurden. Für diese Methode wurde der folgende Algorithmus entwickelt:

```
For all relations in CURRENT-data (CDS)
     For all records in relation
          if record-state is ‘‘deleted’’
```

```
                    delete record from database
               else if record-state is ‘‘changed’’
                    delete record from database
                    insert record into database
               else if record-state is ‘‘inserted’’
                    insert record into database
               end if
          End for all records
     End for all relations in CDS
     For all relations in OTHER-data (ODS)
          For all records in relation
               if record-state is ‘‘deleted’’
                    delete record from database
               else if record-state is ‘‘changed’’
                    delete record from database
                    insert record into database
               else if record-state is ‘‘inserted’’
                    insert record into database
               end if
          End for all records
     End for all relations in ODS
     For all relations in RETRIEVED-data (RDS)
          For all records in relation
               Append record to Tcl-list of records
          End for all records
          If user-interface is GUI
               Call Tcl-function to show selected data
          Else if user-interface is CLI
               List selected records to std-output
          End if
     End for all relations
     Delete transaction
```

- `delete_transaction()`
  Mit dieser Methode wird eine Transaktion mit allen zugehörigen Subtransaktionen gelöscht. Das Löschen erfolgt rekursiv nach folgendem Algorithmus:

```
For all transactions in sub_ta
     Call delete_transaction()
End for
```

```
Delete the transaction
```

- `create_transaction()`
  Diese Methode erzeugt abhängig von der aktuellen Transition eine neue Transaktion. Die Verarbeitung erfolgt nach folgendem Algorithmus:

  ```
  Get numbers of current and new transaction
  Determine type of new transaction
  Create new transaction and link it to superior
  Set token-infos
  Insert new transaction in transaction-list
  Return new transaction as result
  ```

4.3.7 Subystem Repository System

Um die Übersichtlichkeit zu verbessern, wurde das Repository im Konzept in zwei Teile zerlegt (vgl. Abbildung 5). Der eine Teil ist das *Data-Repository*, der andere das *Rule-Repository*. Diese Aufteilung wurde auch für den Entwurf der Datenmodelle und Klassen übernommen. Das vollständige Datenmodell ist als Entity Relationship Diagram (ERD) im Anhang B dargestellt. Die Abbildung 30 zeigt das Klassendiagramm des Repository-Systems.

1. **Klasse** `data_rep`
   Das Data-Repository enthält die Informationen zu in ALFRED definierten Datenbankobjekten (zB. Tabellen-Definitionen, Benutzer und ihre Privilegien). Mit der Klasse `data_rep` werden diese Informationen dem System "online" zur Verfügung gestellt, was einen wesentlich schnelleren Zugriff auf die Informationen gewährleistet, als ein reines DB-Repository. In der Klasse `data_rep` werden verschiedene Listentypen (Instanzen der parametrisierten Klasse `bilist`) für die Speicherung der Daten verwendet.
   Das Datenmodell des Data-Repository ist in Anhang B.1 als Entity-Relationship-Diagramm (ERD) dargestellt.

2. **Klasse** `rule_rep`
   Das Rule-Repository enthält Informationen zu den in ALFRED definierten Regeln wie zum Beispiel Ereignisse, Bedingungen und Aktionen. Der Aufbau der Klasse `rule_rep` ist analog zu derjenigen von `data_rep`. Das ERD des Datenmodells des Rule-Repository befindet sich in Anhang B.2.

4.4 Aspekte der Implementierung

Dieses Kapitel beschreibt einige Aspekte, die für die Implementierung des Prototypen wichtig sind. Dies sind insbesondere die verwendeten Werkzeuge sowie die Realisierung der Schnittstellen zu Menü- und Datenbank-System.

4.4.1 Entwicklungsumgebung

Die Entwicklung von ALFRED erfolgte auf einem Intel-Pentium-Rechner (150 MHz), der mit dem Betriebssystem Linux (Kernel 2.0.33) ausgerüstet ist. Als Entwicklungswerkzeuge wurden die folgenden verwendet:

- **Editor**
  Als Editor für alle Quell-Codes ist der XEmacs Version 20.4 mit seinen speziellen Modi für Formatierung und Ausführung von Befehlen, verwendet worden.
- **Compiler**
  Die Übersetzung der C++-Quellprogramme erfolgte mit dem GNU-C++-Compiler Version 2.7.2.1.
- **Debugger**
  Für die Fehlersuche wurde der GNU-Debugger GDB Version 4.16 verwendet.
- **Tcl/Tk**
  Das Menüsystem wurde mit Tcl Version 7.2 und Tk 4.0 entwickelt. Die notwendigen Anpassungen und Erweiterungen für den Prototypen von ALFRED wurden für die Version 8.0 von Tcl und Tk gemacht.
- **Datenbanksystem**
  Als Datenbanksystem für das Repository und für die Testdaten wurde Ingres PD V. 8.9 verwendet.

4.4.2 Einbinden des Menüsystems

Das Menüsystem verwendet verschiedene Funktionen, mit denen Daten aus den Repositories gelesen werden. Diese Funktionen müssen für die Verwendung in Tcl/Tk in einem Tcl-Interpreter registriert werden. Dazu werden sie im Hauptprogramm von ALFRED implementiert und in der Funktion

`Tcl_AppInit()`, wie am folgenden Beispiel der Funktion für das Ausführen eines Benutzerbefehls `execute_command` gezeigt, registriert:

```
Tcl_CreateCommand(
     interp,
     "execute_command",
     (Tcl_CmdProc *) execute_command,
     (Tcl_ClientData *) NULL,
     (Tcl_DeleteCmdProc *) NULL);
```

Ein auf diese Weise registrierter Befehl kann nun wie ein normales Tcl-Kommando verwendet werden.

4.4.3 Anbindung der Datenbank

Es gibt drei Klassen, die direkt auf das verwendete DBMS *Ingres PD V.8.9* zugreifen. Dies sind:

- Klasse `data_rep`
  Für die Initialisierung des Online-Data-Repository werden alle entsprechenden Daten aus der Datenbank `system` gelesen.
- Klasse `rule_rep`
  Für die Initialisierung des Online-Rule-Repository werden die Regel-Informationen aus der Datenbank `system` gelesen.
- Klasse `tmgr`
  Der Transaktionsmanager liest Daten aus einer Datenbank, wenn eine zu verarbeitende Transition dies erfordert (zB. Update-Transition).
  Beim `commit` einer Transaktion werden einzufügende und zu ändernde Daten in der Datenbank festgeschrieben, sowie zu löschende Datensätze aus der Datenbank entfernt.

Der Transaktionsmanager ist also das einzige Modul, welches Änderungen auf einer Datenbank durchführt.

Der Datenbankzugriff erfolgt mit den EQUEL-Befehlen (**E**mbedded **QUE**ry **L**anguage) von *Ingres PD V. 8.9*. Diese Befehle wurden in den übrigen C(++) Code eingebettet.

Das folgende Beispiel zeigt eine Sequenz von EQUEL-Befehlen, die einen neuen Datensatz in die Datenbank `AUTOVERMIETUNG` einfügen. Zuerst werden einige EQUEL-Variablen deklariert. Anschliessend werden diesen Werte

zugewiesen. Zum Schluss wird ein Datensatz in der Relation MIETER eingefügt:

```
## char rel_name[20]
## char attribute_list[100]
## char value_list[100]

strcpy(rel_name, "MIETER");
strcpy(attribute_list, "MNR,NAME,ORT");
strcpy(value_list, "1,Meier,Bern");

## INGRES AUTOVERMIETUNG
## PARAM APPEND TO rel_name (attribute_list, value_list)
## EXIT
```

Module, die EQUEL-Befehle enthalten, müssen vor dem Compilieren mit dem EQUEL-Precompiler übersetzt werden. Dadurch werden diese Befehle (beginnend mit "##") durch normalen C-Code ersetzt. Beim Linken des Programms muss die Bibliothek "libq", welche ein Bestandteil von Ingres PD V. 8.9 ist, mit eingebunden werden.

5 Laufzeitanalyse

Zur Beurteilung der Leistungsfähigkeit der entwickelten Konzepte und des implementierten Prototypen wurden verschiedene Laufzeitmessungen durchgeführt.

Die erzielten Ergebnisse zeigen, dass der realisierte Prototyp die an ihn gestellten Anforderungen grundsätzlich erfüllt. Das Laufzeitverhalten ist für einen Prototypen angemessen. Ein direkter Vergleich mit einem funktionell gleichwertigen System konnte allerdings nicht durchgeführt werden. Die aus den Laufzeitmessungen gewonnenen Resultate liefern aber dennoch einige Erkenntnisse, die zur Optimierung des Systems bezüglich Laufzeitverhalten und Ressourcen-Verbrauch führen können.

5.1 Konzept

Mit den durchgeführten Messungen wird das Laufzeitverhalten des Verarbeitungssystems von ALFRED beurteilt. Dabei sollen zwei Aspekte betrachtet werden:

- **Beurteilung der Subsysteme von ALFRED**
  Um Aussagen über das Laufzeitverhalten der einzelnen Subsysteme machen zu können, ist die von den einzelnen Subsystemen verbrauchte Rechenzeit gemessen und analysiert worden. Für die Messung der Verarbeitung der Netze ist die Zeitdauer zwischen Beginn und Ende der erzeugten Transaktionen berechnet worden.
- **ALFRED vs. Ingres PD V. 8.9**
  Mit diesem Vergleich soll festgestellt werden, wie sich ALFRED im Vergleich zu direkten Manipulationen der Datenbank verhält. Um möglichst aussagekräftige Resultate zu erhalten, wurden benutzerdefinierte Transaktionen mit einer Vielzahl von Befehlen erstellt und anschliessend mit ALFRED verarbeitet. Als Vergleichsmessung wurde eine Ingres-Kommando-Datei mit entsprechenden Befehlen erstellt. Die jeweils benötigte Zeit für die Verarbeitung wurde dann verglichen.
  Die Ergebnisse dieses Vergleichs sind aber sehr stark zu relativieren, da Ingres PD V. 8.9 ein sehr einfaches DBMS mit viel geringerer Funktionalität ist. So werden zB. keine Verarbeitung auf Transaktionsbasis und auch nur wenige Integritätsbedingungen unterstützt. Es können beispielsweise keine Schlüssel und keine Not-Null-IB definiert werden.

Anhand der so gewonnen Grössen und Vergleichswerte lassen sich schliesslich Hinweise und Ansätze für Optimierungen ableiten.

5.2 Testszenario

Für die Durchführung der Laufzeitmessungen wurden ein einfaches Datenmodell, eine Anzahl von Regeln und vier Test-Transaktionen mit durchschnittlich 50 Befehlen definiert.

5.2.1 Datenmodell

Das Testbeispiel beschreibt eine Autoverleih-Firma, welche Autos verschiedener Marken und zu verschiedenen Konditionen an Kunden vermietet. Das Datenmodell für die Testdurchführung umfasst drei Relationen und ist in Abbildung 31 als ERD dargestellt.

1. Relation “Mieter”
   Die Mieter-Relation beschreibt einen Kunden und enthält die folgenden Attribute:

   - `mnr`
     Eine fortlaufende Nummer, die als Primärschlüssel fungiert.
   - `name`
     Der Name eines Kunden.
   - `ort`
     Der Wohn- oder Geschäftssitz eines Kunden.
   - `branche`
     Die Branche, in welcher der Kunde tätig ist.
   - `g_beginn`
     Der Beginn der Geschäftsbeziehungen zu diesem Kunden.
   - `g_ende`
     Das Ende der Geschäftsbeziehungen mit diesem Kunden.
   - `mietkosten`
     Das Total der Kosten für alle bisherigen Verträge mit diesem Kunden.
   - `mv_anzahl`
     Die Anzahl bisher abgeschlossener Mietverträge.

2. Relation “Mietwagen”
   Die Mietwagen-Relation beschreibt ein zu mietendes Auto. Die Attribute dieser Relation haben die folgende Bedeutung:

   - `wnr`
     Dies ist eine fortlaufende Nummer, die zur Identifizierung eines Autos dient und als Primärschlüssel der Relation fungiert.
   - `hersteller`
     Der Name des Autoherstellers.
   - `typ`
     Die Typbezeichnung des Autos.
   - `baujahr`
     Das Baujahr des Autos.
   - `mietsatz`
     Die Mietkosten für einen Tag.

- `g_beginn`
  Das Datum, ab wann das Auto zur Verfügung steht.
- `g_ende`
  Das Datum des Verkaufs des Autos.

3. Relation "Mietvertrag"
   Die Mietvertrag-Relation beschreibt einen einzelnen Mietvorgang eines Kunden bezüglich eines Autos über eine zusammenhängende Zeitdauer. Die Relation enthält 11 Attribute mit den folgenden Bedeutungen:

   - `mvnr`
     Eine fortlaufende Nummer, welche zur Identifizierung eines Mietvertrags dient und als Primärschlüssel für die Relation fungiert.
   - `miet_datum`
     Das Datum des Vertragsabschlusses.
   - `miet_beginn`
     Das Datum des Mietbeginns.
   - `miet_ende`
     Das Datum des Mietendes.
   - `bezahl_dt`
     Das Datum, an welchem die Rechnung für diesen Mietvertrag beglichen worden ist.
   - `miet_tage`
     Die Länge der Miete in Tagen.
   - `miet_satz`
     Die Kosten für den Wagen pro Tag.
   - `miet_kosten`
     Die Kosten für die Miete insgesamt.
   - `lfd_nr`
     Eine fortlaufende Nummer, welche die Anzahl Mietverträge eines Kunden angibt.
   - `mnr`
     Die Nummer des Mieters.
   - `wnr`
     Die Nummer des gemieteten Wagens.

5.2.2 Regeln

Für alle im vorherigen Abschnitt beschriebenen Relationen wurden mehrere Regeln zur Überprüfung der Integritätsbedingungen (IB) und zur Aktualisierung (Neuberechnung) von Attributwerten definiert. Zudem gibt es auch eine Anzahl Regeln, die durch die Verarbeitung von Transaktionen ausgelöst werden. Diese prüfen unter anderem, ob der Benutzer berechtigt ist, die Transaktion auszuführen.

- **Regeln für den Entitätstyp "Mieter"**
  Für diesen Entitätstypen wurden 17 Regeln zur Überprüfung der NOT-NULL-Eigenschaft verschiedener Attribute erzeugt. Ausgelöst werden diese Regeln entweder durch `insert`- oder `update`-Befehle. Ein Beispiel für eine solche Regel in der Syntax der ARDL ist:

```
rule MT_I_NN_MNR
     on pre insert in MIETER inst
     if :NEW.MIETER.MNR = NULL
     do begin
          message to CURRENT USER
               error 'MT1: Datensatz von MIETER fehlerhaft'
          cancel
     end
     is active
     has priority 90
```

  Mit dieser Regel wird, wenn die NOT-NULL-IB für das Attribut `mnr` der Relation `mieter` verletzt ist, dem Benutzer eine Meldung angezeigt und die Verarbeitung des Befehls abgebrochen.

  Da der Prototyp von ALFRED nur primitive Bedingungen unterstützt, muss für jedes Attribut, für welches eine NOT-NULL-IB definiert ist, eine eigene Regel spezifiziert werden. Mit komplexen Bedingungen könnten alle diese Regeln zu einer einzigen zusammengefasst werden.

  Für die Aktualisierung von Attributen wurden 6 Regeln definiert. Ein Beispiel für eine solche Regel in ARDL-Syntax ist:

```
rule MT_U_RU_mv_anzahl
     on event post insert in MIETER inst
     if :NEW.MIETER.MV_ANZAHL = 6
```

```
do begin
     update MIETER (MIETKOSTEN = :NEW.MIETER.MIETKOSTEN - 200)
end
is active
has priority 20
```

Mit dieser Regel wird einem Mieter beim 6. Mietvertrag ein Rabatt von Fr. 200.- gewährt.

- **Regeln für den Entitätstyp "Mietwagen"**
  Für den Entitätstypen Mietwagen wurden 20 Regeln zur Überprüfung der Integritätsbedingungen definiert. Diese sind analog zu den oben gezeigten implementiert worden.

- **Regeln für den Entitätstyp "Mietvertrag"**
  Für den Entitätstypen Mietvertrag wurden insgesamt 29 Regeln (26 für IB und 3 für Aktualisierungen) erzeugt.

- **Durch Transaktionen ausgelöste Regeln**
  Es gibt 6 Regeln, die durch Transaktionsereignisse ausgelöst werden. Davon werden drei durch BOT, eine durch EOT und zwei durch COT ausgelöst. Das folgende Beispiel zeigt eine EA-Regel, welche dem Benutzer am Beginn einer Transaktion anzeigt, dass diese nun verarbeitet wird:

```
rule BOT_MSG_1
     on post BOT
     do begin
          message to CURRENT USER hint
               'Transaktion wird verarbeitet!'
     end
     is active
     has priority 50
```

5.2.3 Transaktionen

Für die Durchführung der Laufzeitmessungen und auch für die Überprüfung der korrekten Verarbeitung der Befehle und Regeln wurden vier Test-Transaktionen erstellt. Diese beinhalten jeweils mehrere `select`, `insert`, `update` und `delete`-Befehle und haben alle mehr oder weniger den selben Aufbau.

5.3 Durchführung

Nach der Initialisierung der Test-Datenbank wird die Applikation ALFRED mit `alfred -l test.log -t` gestartet. Die Kommandozeilenparameter geben an, dass eine Log-Datei mit dem Namen 'test.log' ('-l') und ein Zeitlog ('-t') mit den Transaktionszeiten, welches anschliessend für die Beurteilung des Laufzeitverhaltens verwendbar ist, erstellt werden sollen.

Für die Ausführung der Testtransaktionen muss sich der Benutzer im Login-Fenster bei ALFRED anmelden. Anschliessend können in der grafischen Benutzeroberfläche Befehle spezifiziert und ausgeführt werden. Als erstes muss im Menü "Database" die Datenbank, mit welcher gearbeitet werden soll, ausgewählt werden (Befehl "connect"). Darauf kann mit dem Befehl "Execute" im Menü "Transaction" eine der definierten Test-Transaktionen ausgeführt werden. Nach dem Klicken auf die Schaltfläche "Ok" im Fenster "Execute transaction" (vgl. Abbildung 32) erzeugt das Verarbeitungs-System das Petri-Netz mit allen Befehlen und verarbeitet dieses anschliessend. Dabei werden in einem neuen Fenster Meldungen angezeigt, die durch die Verarbeitung von Regeln erzeugt werden (vgl. Abbildung 33). Datensätze, die zum Anzeigen selektiert worden sind, werden in weiteren speziellen Fenstern dargestellt. Die Abbildung 34 zeigt das Fenster, das die Liste aller selektierten Wagen enthält.

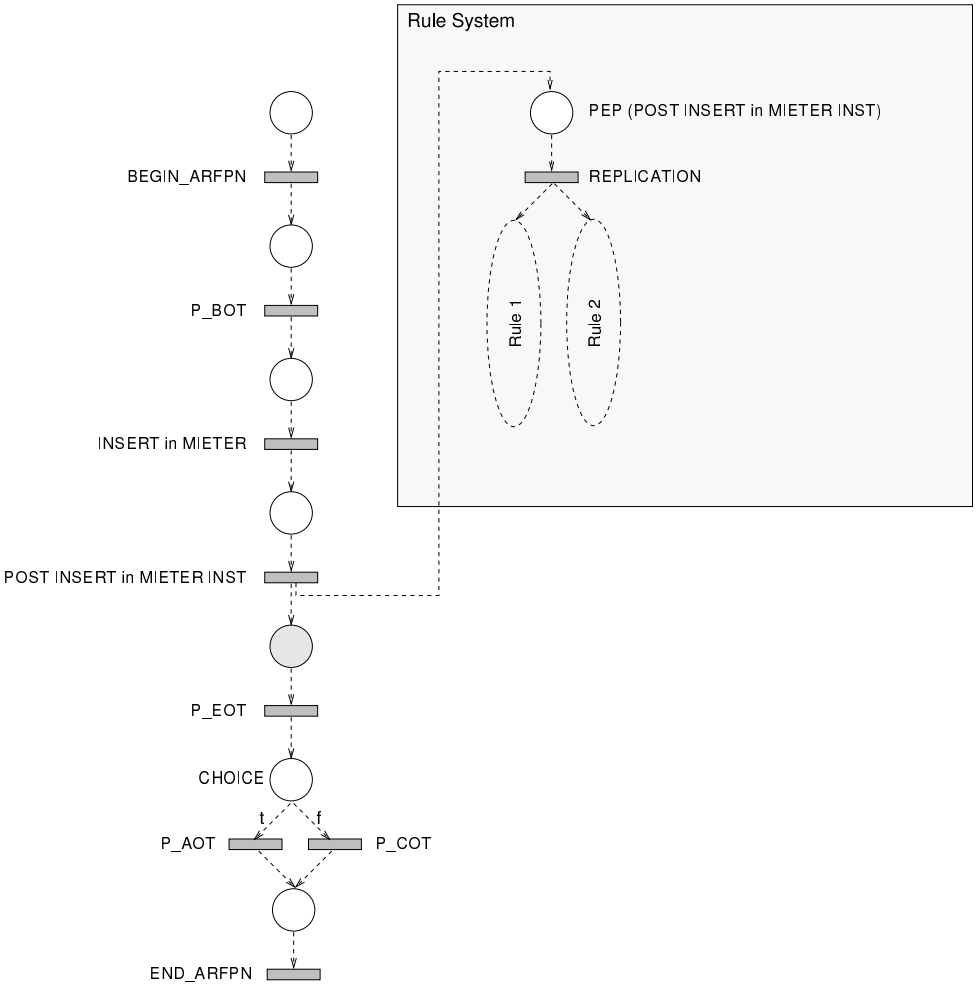


Abbildung 19: Erweiterung des AFPN durch den PED

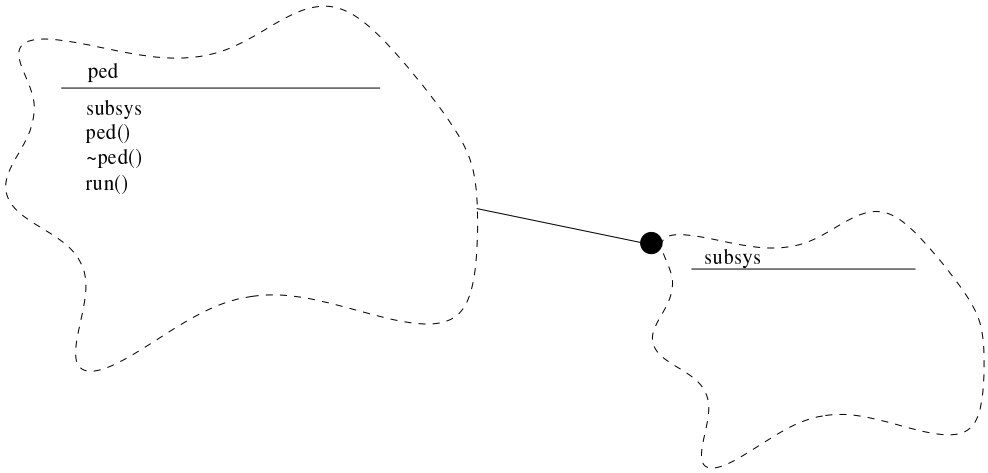


Abbildung 20: Diagramm der Klasse `ped`

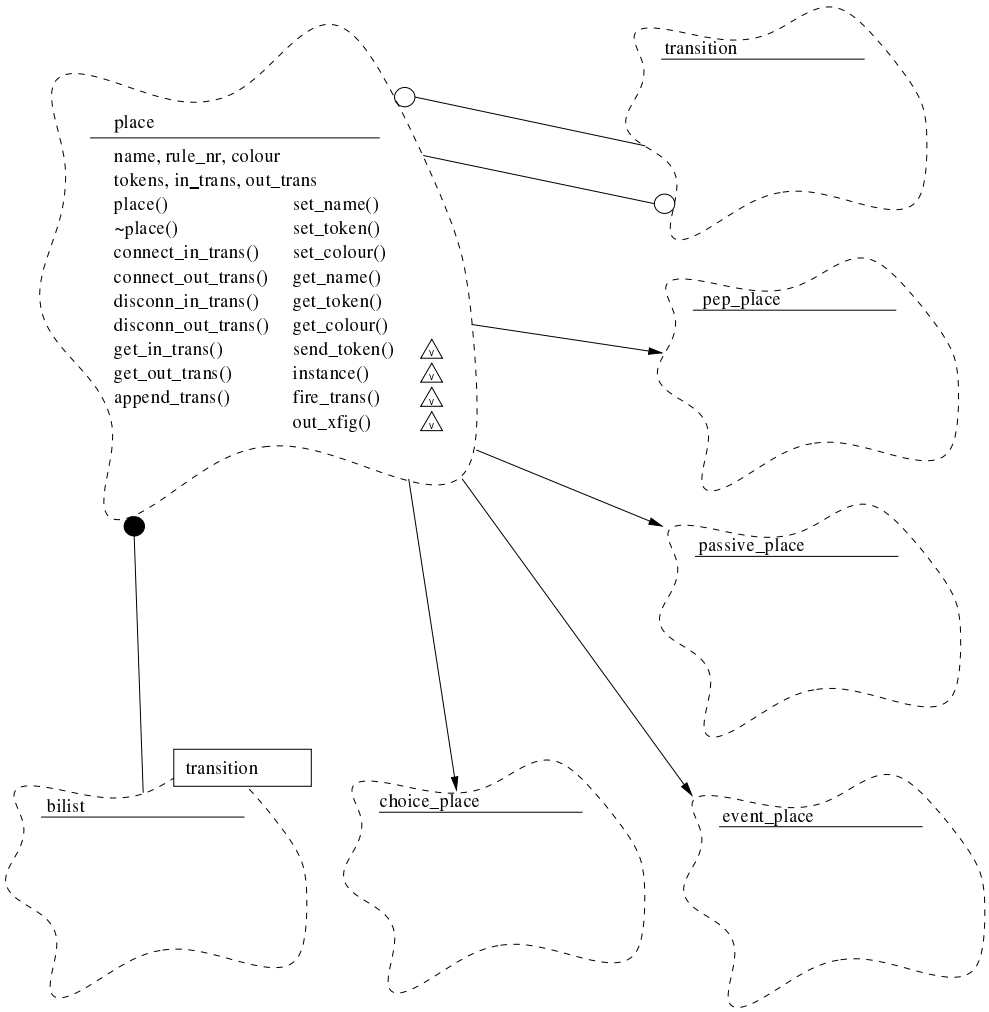


Abbildung 21: Diagramm der Klasse `place`

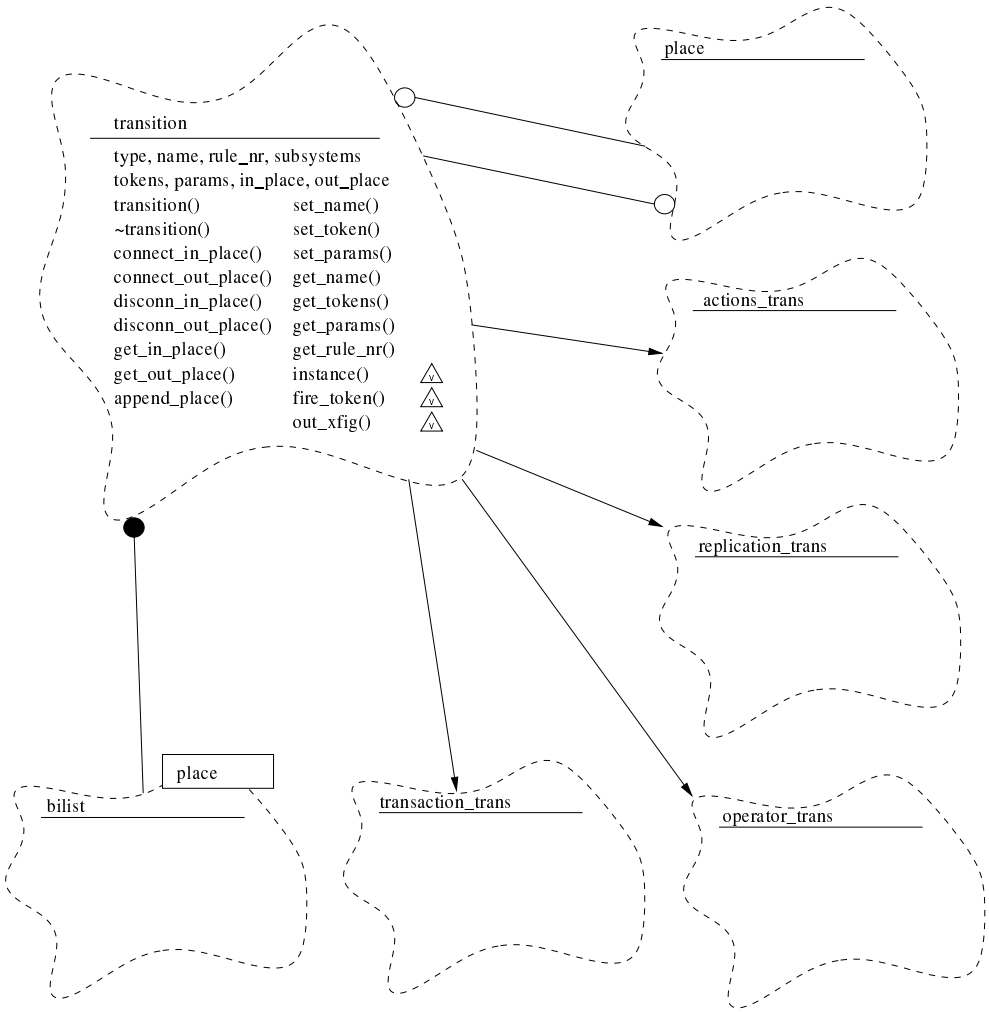


Abbildung 22: Diagramm der Klasse `transition`

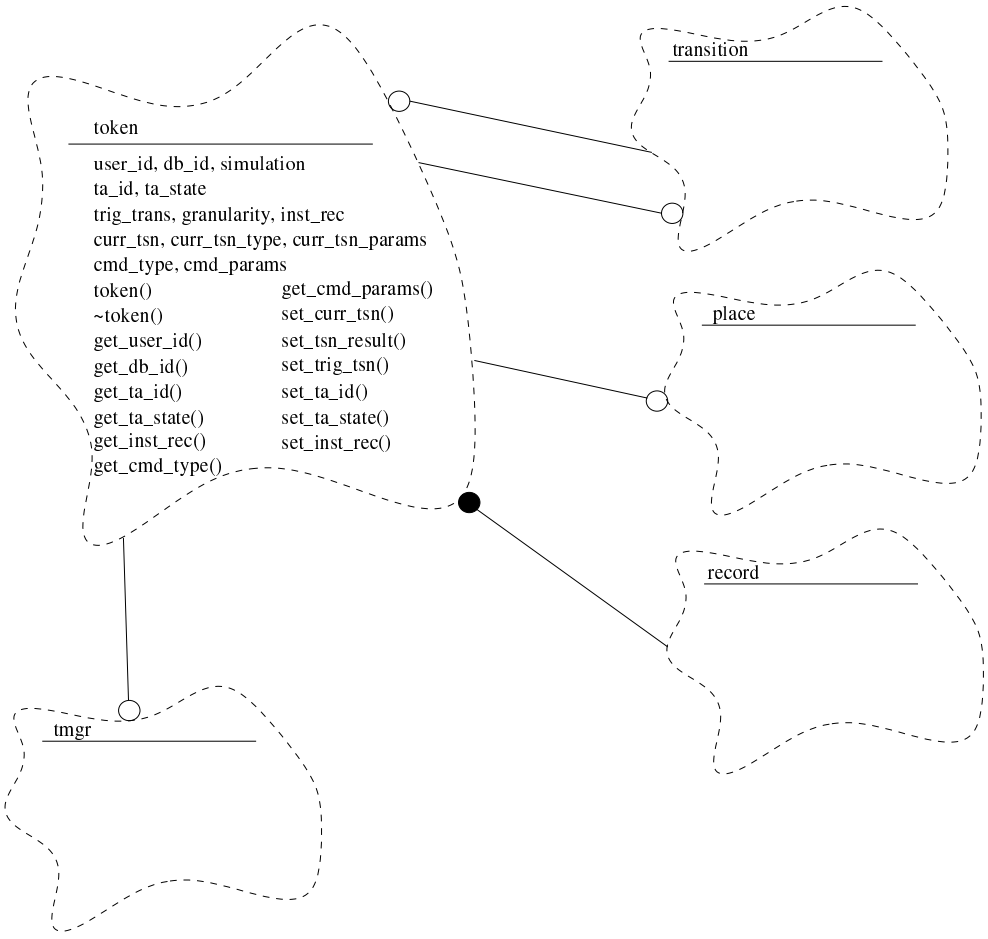


Abbildung 23: Diagramm der Klasse `token`

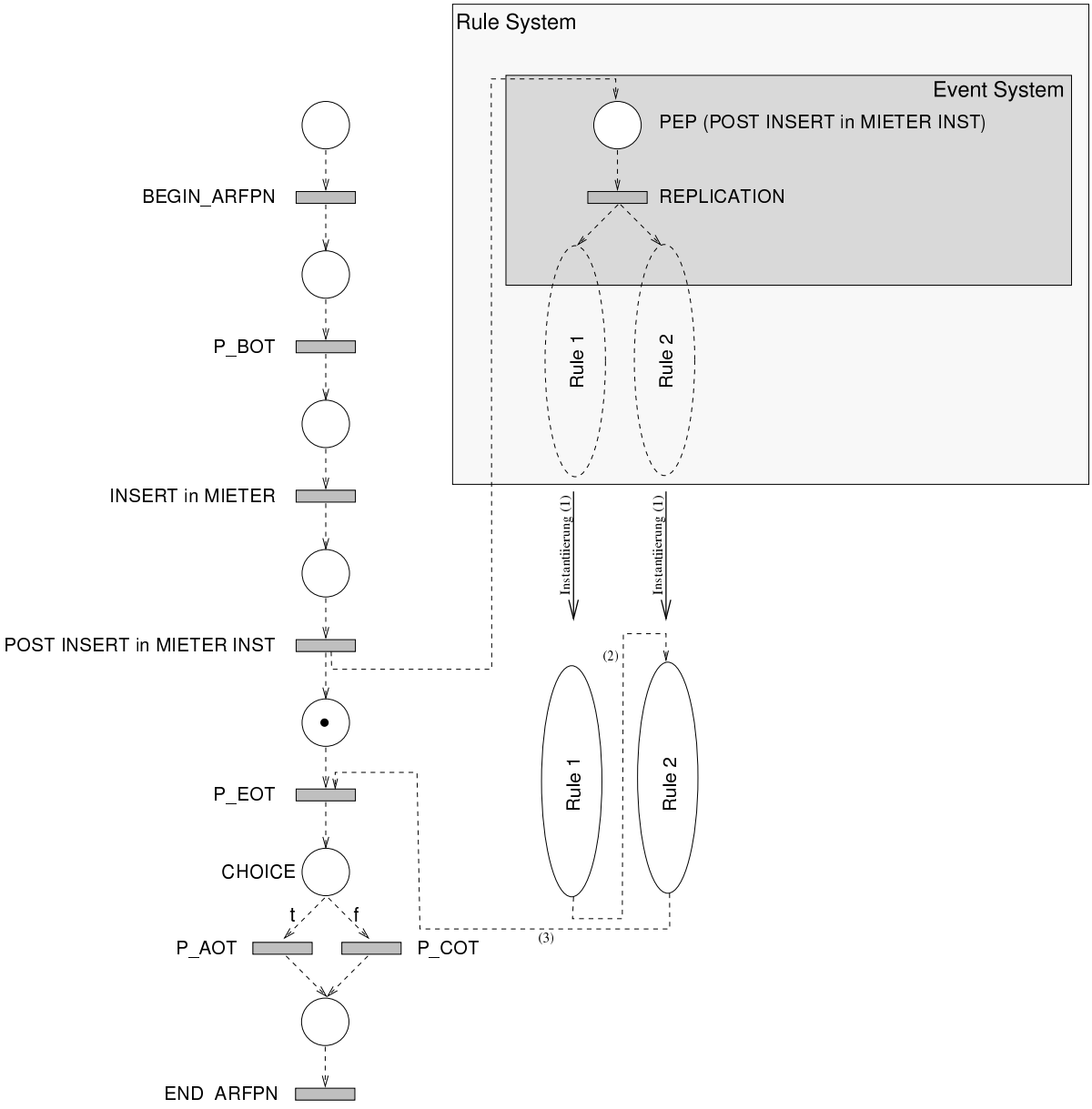


Abbildung 24: Aufgaben des Moduls ARFPN

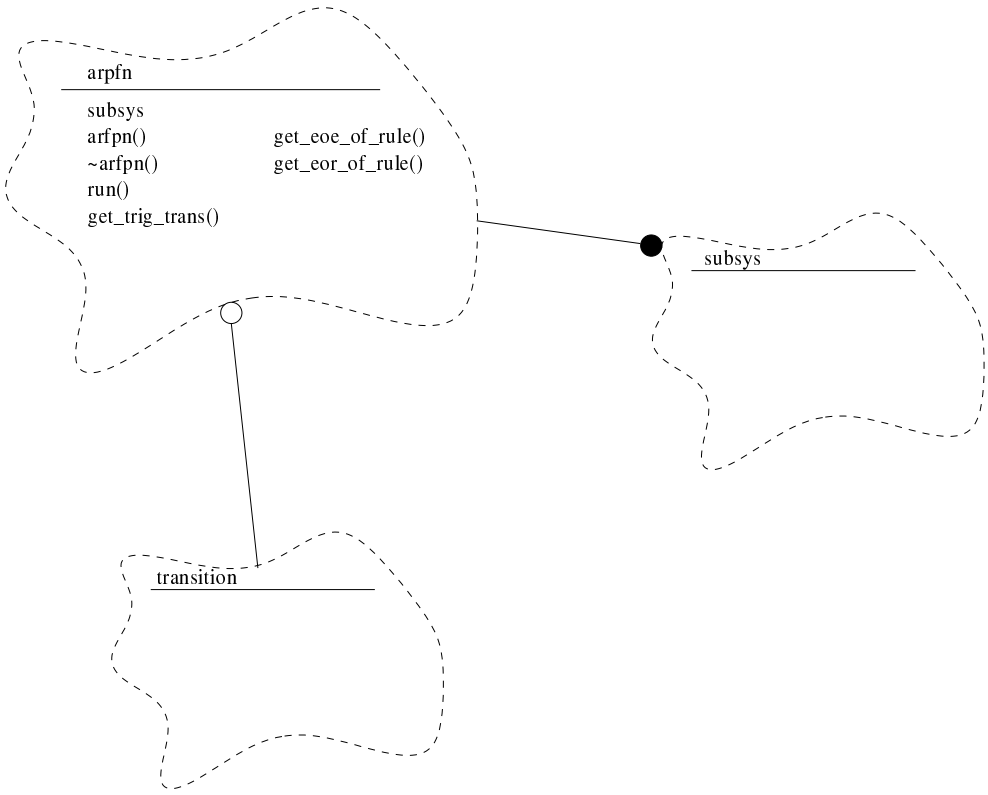


Abbildung 25: Diagramm der Klasse `arfpn`

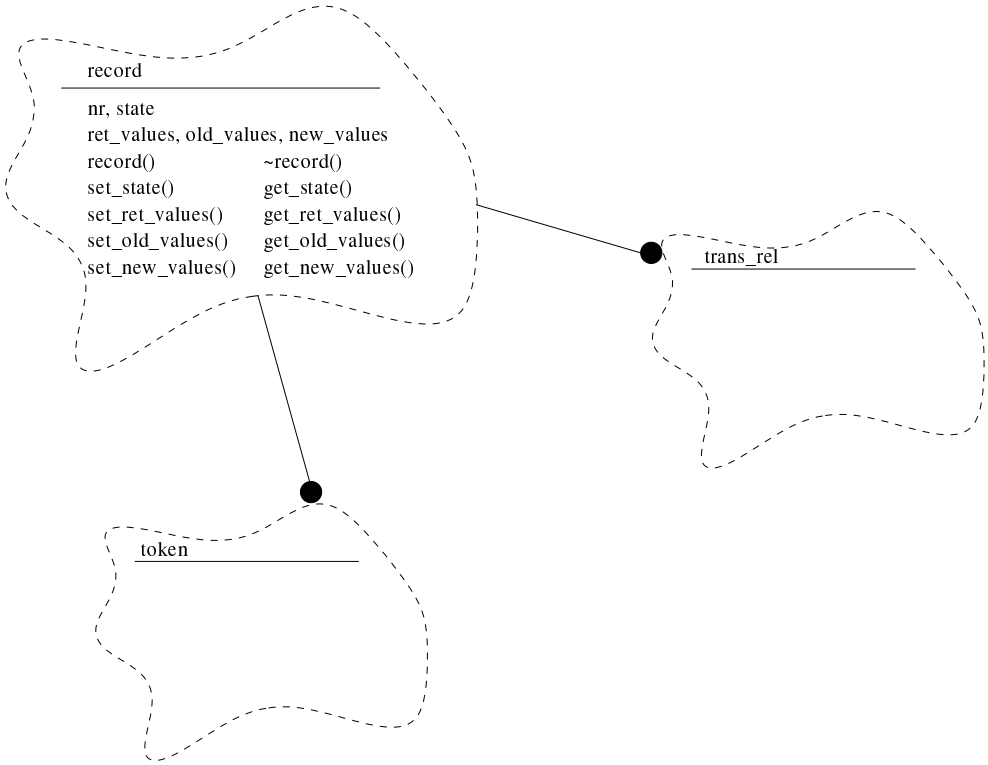


Abbildung 26: Diagramm der Klasse `record`

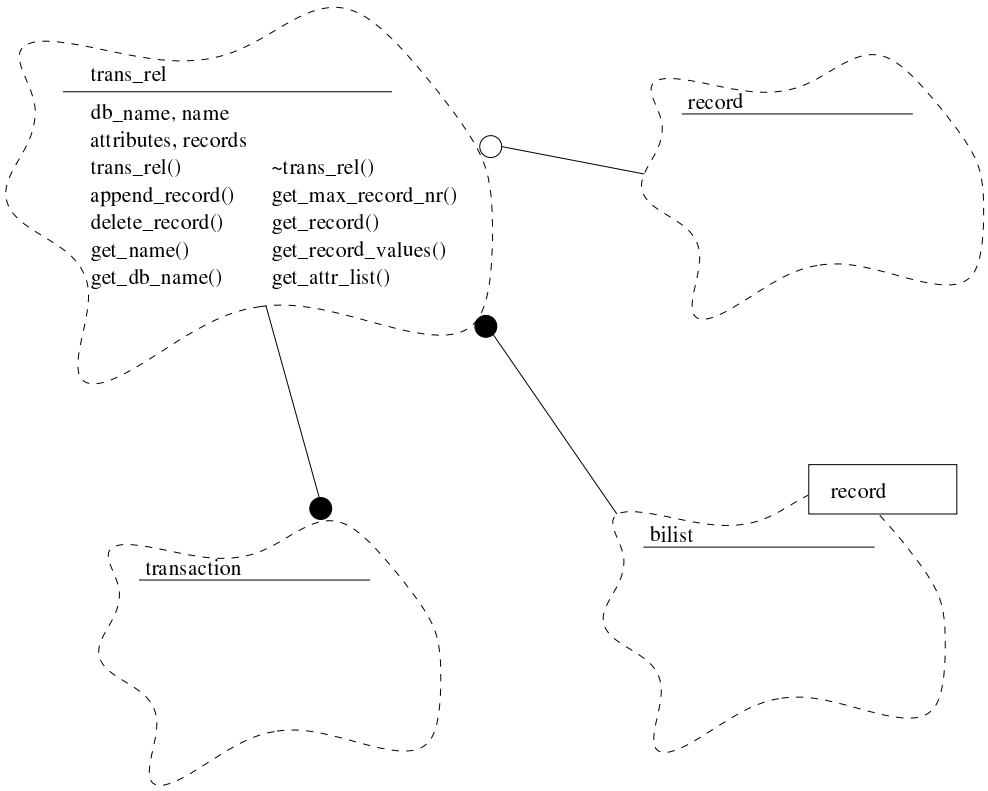


Abbildung 27: Diagramm der Klasse `trans_rel`

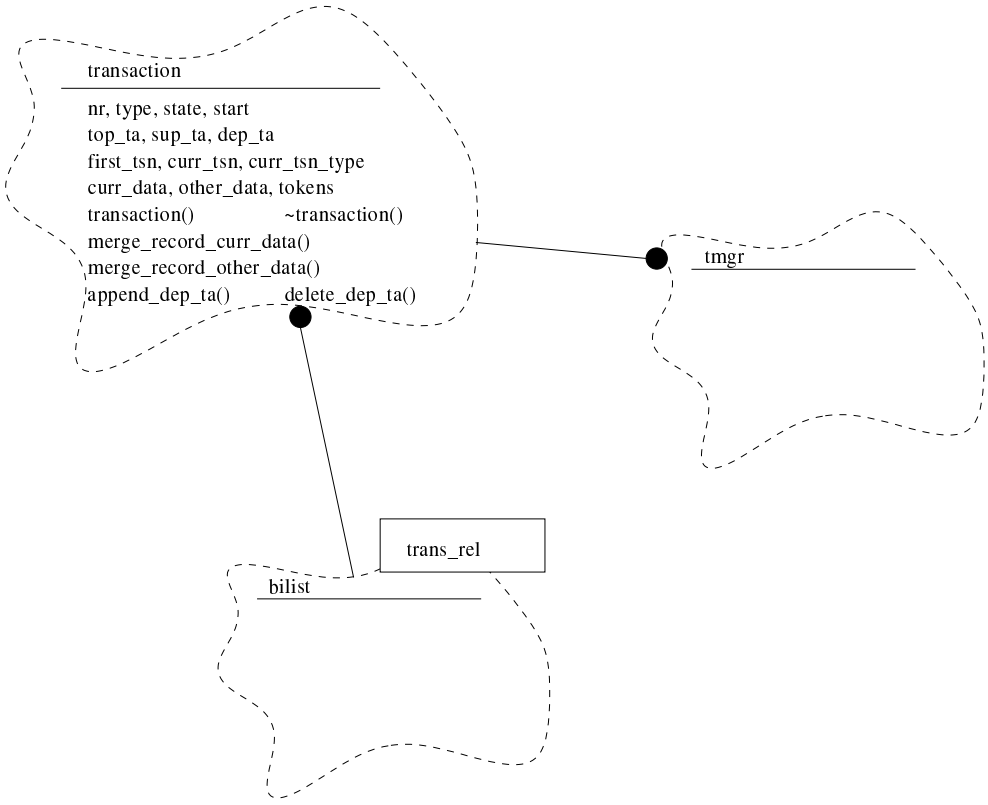


Abbildung 28: Diagramm der Klasse `transaction`

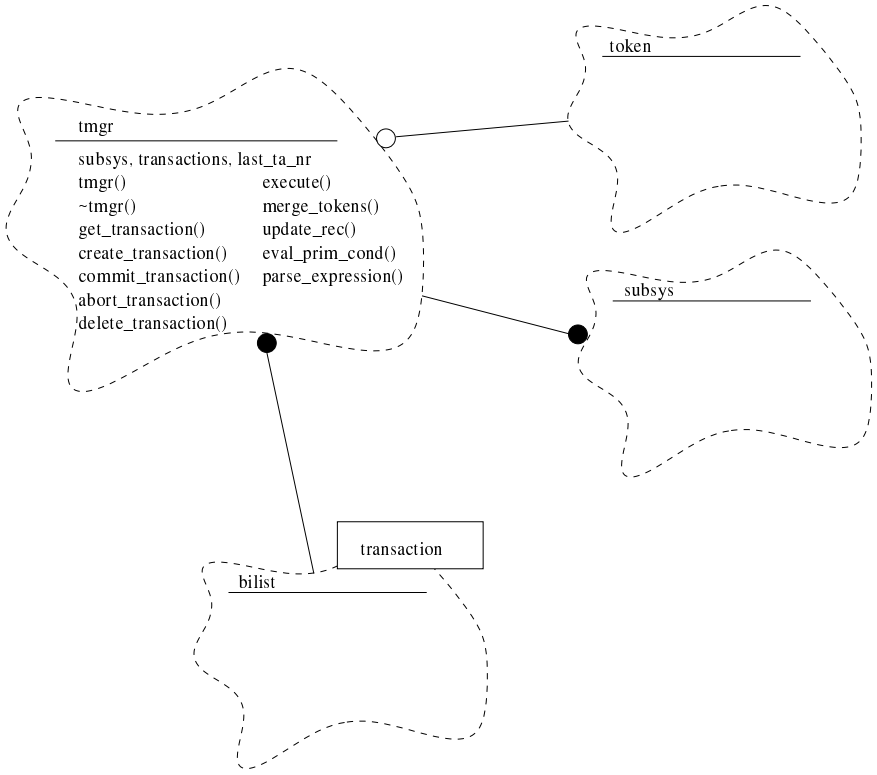


Abbildung 29: Diagramm der Klasse `tmgr`

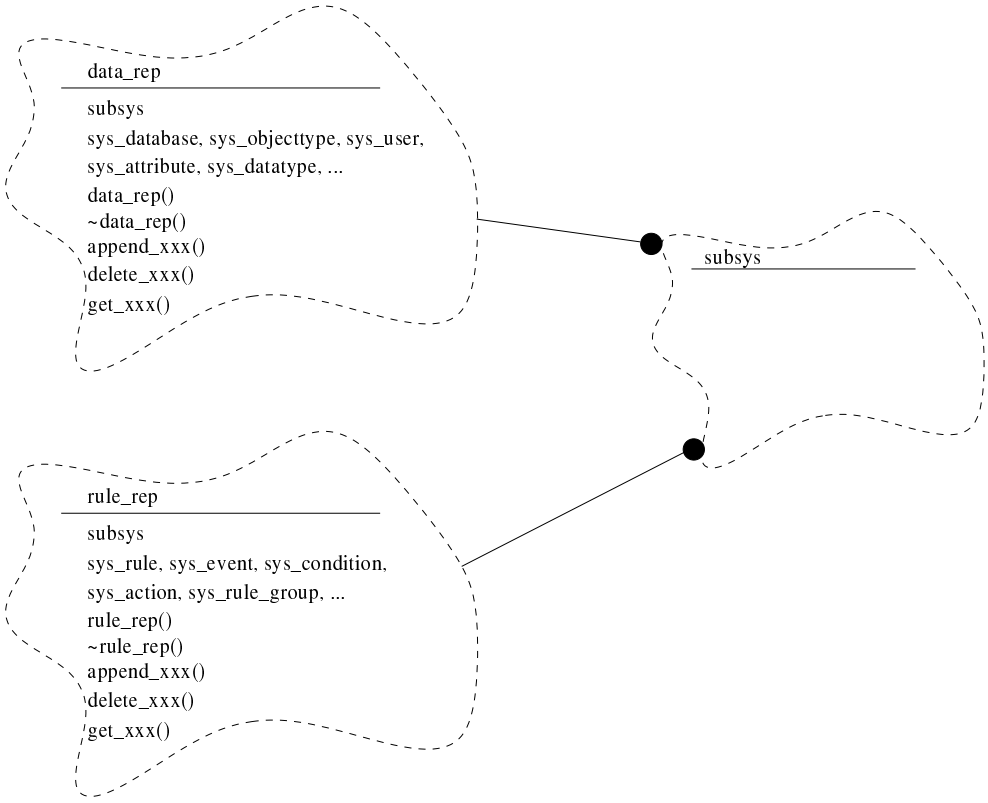


Abbildung 30: Klassendiagramm des Repository-Systems

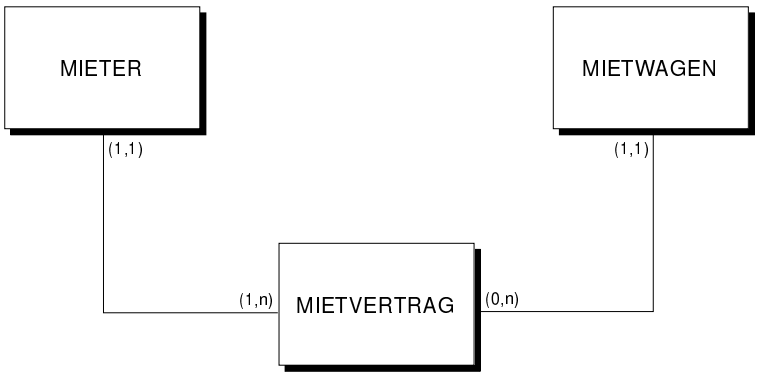


Abbildung 31: Datenmodell der Test-Datenbank

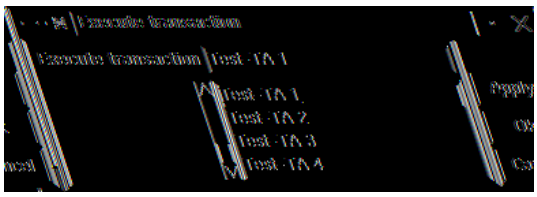


Abbildung 32: Fenster zum Ausführen einer Transaktion

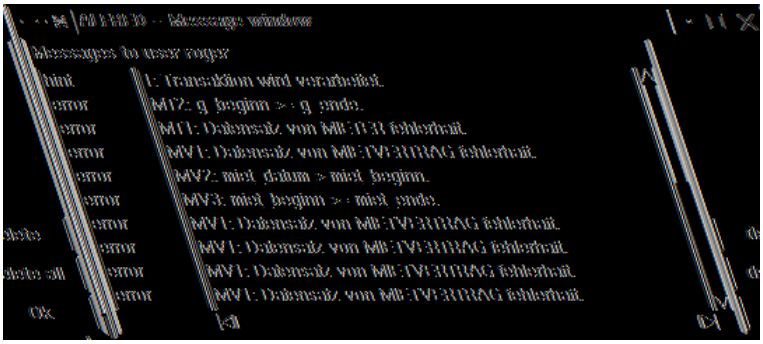


Abbildung 33: Fenster mit Liste der Meldungen an den Benutzer

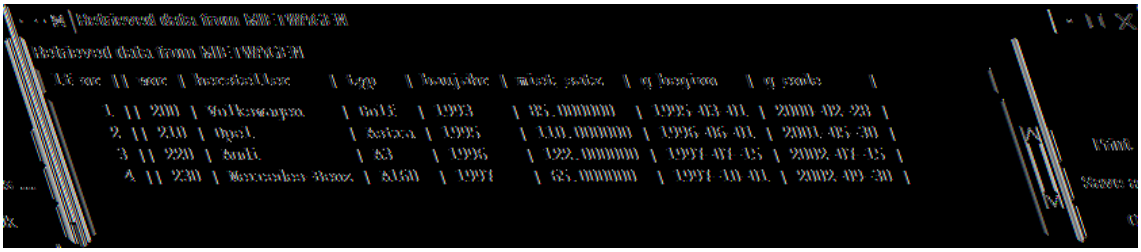


Abbildung 34: Fenster mit Liste der selektierten Datensätze

5.4 Ergebnisse der Transaktionen

Durch die Angabe des Kommandozeilenparameters `-t` wurde bei der Verarbeitung eine Log-Datei mit Zeitwerten erstellt. Aus diesen wurden für einige Grössen jeweils Mittelwert (MW) und Standardabweichung (SA) ermittelt. Alle Zahlen geben die benötigte Verarbeitungszeit in Sekunden an.
Neben den Werten aus dem Auswertungsprogramm können auch der Log-Datei, welche bei der Ausführung von ALFRED erzeugt wird, gewisse Zeitwerte entnommen werden. Dies sind unter anderem:

- Die gesamte Zeitdauer für die Verarbeitung eines vom Benutzer spezifizierten Befehls und
- die Zeit, die von den einzelnen Subsystemen verbraucht wird.

1. **Test-Transaktion 1**
   Die Test-Transaktion 1 besteht aus 15 `insert`-, 23 `update`-, 2 `delete`- und 3 `select`-Befehlen. Dies sind insgesamt 43 Kommandos. Die Ergebnisse der Auswertung sind in Tabelle 1 dargestellt.

Gemessene Zeiten	**N**	**MW**	**SA**
Root-Transaktion	1	24.230	
Alle Transaktionen (ohne Root)	1208	0.035	0.112
Befehle in Root-Transaktion	43	0.557	0.253
Condition-Transaktionen	464	0.017	0.010
Action-Transaktionen	500	0.019	0.018
Instruction-Transaktionen	244	0.107	0.235
davon in Aktionskomponenten	201	0.010	0.024
Subsystem AFPN	1	0.30	
Subsystem PED	1	0.76	
Subsystem ARFPN	43	0.009	0.004
Anzahl ausgelöste Regeln	488	11.63	4.012

Tabelle 1: Ergebnisse der Test-Transaktion 1

   Die in dieser Tabelle angegeben Werte sind wie folgt zu interpretieren:

   - Root-Transaktion
     Es wurde im Verlauf der Verarbeitung der Benutzertransaktion

eine ROOT-Transaktion erzeugt. Für die gesamte Verarbeitung wurden 24.23 Sekunden benötigt. Darin inbegriffen sind:

- die Verarbeitung aller 43 Befehle
- die Verarbeitung aller ausgelöster Regeln
- die Zugriffe (lesend / schreibend) auf die Datenbank
- die Ausgaben von Meldungen im GUI von ALFRED
- das Erstellen der Log-Dateien

- Alle Transaktionen (ohne Root)
  Insgesamt wurden ausser der Root-Transaktion 1208 Transaktionen erzeugt. Diese benötigten im Durchschnitt 0.035 Sekunden für die Verarbeitung bei einer Standardabweichung von 0.112 Sekunden. Dieser relativ hohe SA-Wert kommt daher, dass in dieser Stichprobe auch Instruction-Transaktionen der Root-Transaktion enthalten sind, die viel länger dauern als die Transaktionen in den Regeln.
- Befehle in Root-Transaktion
  Die Test-Transaktion 1 enthält 43 Befehle. Die Verarbeitung dauerte im Mittel 0.557 Sekunden bei einer Standardabweichung von 0.253 Sekunden. Darin inbegriffen sind wiederum:
  - die Verarbeitung aller ausgelöster Regeln
  - die Zugriffe (lesend / schreibend) auf die Datenbank
  - die Ausgaben von Meldungen im GUI von ALFRED
  - das Schreiben von Informationen in die Log-Dateien
- Condition-Transaktionen
  Für die Auswertung einer Bedingung in einer Regel wird jeweils eine Condition-Transaktion erzeugt. Daraus lässt sich schliessen, dass insgesamt 464 ECA-Regeln verarbeitet wurden. Die Auswertung einer Bedingung dauerte im Durchschnitt 0.017 Sekunden (SA 0.01).
- Action-Transaktionen
  Action-Transaktionen werden bei der Verarbeitung einer Aktionskomponente einer Regel erzeugt. Es wurden insgesamt 500 Aktions-Transaktionen erzeugt und daher auch total 500 Regeln verarbeitet. Die durchschnittliche Dauer einer Aktions-Transaktion liegt bei 0.019 Sekunden bei einer SA von 0.018 Sekunden. Diese annähernd gleichen Werte bedeuten, dass die Bedingungsauswertung häufig ein `FALSE` geliefert hat und somit der leere ELSE-Teil der Regel ausgeführt worden ist.

- Instruction-Transaktionen
  Instruction-Transaktionen werden für zwei unterschiedliche Aufgaben verwendet:
  (a) Für die Kapselung von Befehlen in der benutzerdefinierten Transaktion und
  (b) für die Kapselung von Befehlen in der Aktionskomponente einer Regel.

  Die Stichprobe enthält alle 244 erzeugten Transaktionen diesen Typs. Die Verarbeitung dauerte im Schnitt 0.107 Sekunden bei einer Standardabweichung von 0.235 Sekunden.
- Davon in Aktionskomponenten
  Diese Stichprobe umfasst nur die 201 Instruktionen aus den Aktionskomponenten. Diese Instruktions-Transaktionen werden im Mittel in 0.010 Sekunden bei einer Standardabweichung von 0.024 Sekunden verarbeitet.
- Susbystem AFPN
  Das Erzeugen eines Petri-Netzes bestehend aus 43 Befehlen (d.h 134 Plätze und 135 Transitionen) dauerte 0.3 Sekunden.
- Subsystem PED
  Die Erkennung der primitiven Ereignisse und die Herstellung der Verbindungen zu den zugehörigen Regeln dauerte 0.76 Sekunden.
- Subsystem ARFPN
  Das Subsystem ARFPN wurde insgesamt 43 mal (1 mal je Befehl) aufgerufen. Das Kopieren der benötigten Teile aus den Petri-Netzen der Regeln dauerte im Schnitt 0.009 Sekunden (SA 0.004 Sekunden).
- Anzahl ausgelöste Regeln
  Es wurden total 488 Regeln ausgelöst. Davon wurden 464 Regeln verarbeitet (vgl. weiter oben). Die restlichen 24 Regeln wurden zwar ausgelöst, aber nicht verarbeitet, da die Subtransaktion im Status `cancelled` war.

Die Ergebnisse der folgenden 3 Test-Transaktionen können in analoger Weise interpretiert werden. Da sie alle die Aussagen von Test-Transaktion 1 bestätigen, wird auf eine ausführliche Erläuterung der Resultate verzichtet.

2. **Test-Transaktion 2**
   Die Test-Transaktion 2 enthält 19 `insert`-, 29 `update`-, 4 `delete`- und

3 `select`-Kommandos und besteht somit aus insgesamt 55 Befehlen. Die Ergebnisse der Auswertung dieser Transaktion sind in Tabelle 2 dargestellt.

Gemessene Zeiten	**N**	**MW**	**SA**
Root-Transaktion	1	27.690	
Alle Transaktionen (ohne Root)	1380	0.036	0.108
Befehle in Root-Transaktion	55	0.498	0.262
Condition-Transaktionen	563	0.016	0.009
Action-Transaktionen	595	0.018	0.017
Instruction-Transaktionen	222	0.134	0.247
davon in Aktionskomponenten	167	0.014	0.026
Subsystem AFPN	1	0.40	
Subsystem PED	1	1.00	
Subsystem ARFPN	55	0.010	0.004
Anzahl ausgelöste Regeln	595	10.82	4.212

Tabelle 2: Ergebnisse der Test-Transaktion 2

3. **Test-Transaktion 3**
   Die dritte Test-Transaktion besteht aus 54 Befehlen (14 `insert`, 34 `update`, 3 `delete` und 3 `select`). Die Resultate der Verarbeitung dieser Test-Transaktion sind in Tabelle 3 zusammengestellt.

4. **Test-Transaktion 4**
   Die Test-Transaktion 4 schliesslich besteht aus 24 `insert`-, 31 `update`-, 2 `delete`- und 3 `select`-Kommandos. Dies ergibt ein Total von 60 Befehlen. Tabelle 4 zeigt die Ergebnisse der Auswertung dieser Transaktion.

5.5 Vergleich ALFRED mit Ingres PD V. 8.9

Die zwei Systeme ALFRED und Ingres PD V. 8.9 sind eigentlich nicht direkt vergleichbar, weil Ingres viel weniger Funktionalität enthält als ALFRED. Der Vollständigkeit halber und um einen Eindruck vom relativen Laufzeitverhalten der aktiven Schicht zu erhalten, wird der Vergleich an dieser Stelle aber trotzdem durchgeführt.

Gemessene Zeiten	**N**	**MW**	**SA**
Root-Transaktion	1	32.830	
Alle Transaktionen (ohne Root)	1517	0.038	0.120
Befehle in Root-Transaktion	54	0.600	0.267
Condition-Transaktionen	610	0.018	0.010
Action-Transaktionen	645	0.019	0.018
Instruction-Transaktionen	262	0.133	0.268
davon in Aktionskomponenten	208	0.012	0.025
Subsystem AFPN	1	0.380	
Subsystem PED	1	1.030	
Subsystem ARFPN	54	0.010	0.005
Anzahl ausgelöste Regeln	645	11.944	4.159

Tabelle 3: Ergebnisse der Test-Transaktion 3

Gemessene Zeiten	**N**	**MW**	**SA**
Root-Transaktion	1	29.400	
Alle Transaktionen (ohne Root)	1860	0.028	0.094
Befehle in Root-Transaktion	60	0.484	0.233
Condition-Transaktionen	657	0.014	0.010
Action-Transaktionen	711	0.016	0.016
Instruction-Transaktionen	491	0.065	0.177
davon in Aktionskomponenten	431	0.006	0.017
Subsystem AFPN	1	0.490	
Subsystem PED	1	1.150	
Subsystem ARFPN	60	0.010	0.005
Anzahl ausgelöste Regeln	711	11.850	3.644

Tabelle 4: Ergebnisse der Test-Transaktion 4

Einige wichtige Unterschiede, die sich stark auf das Laufzeitverhalten der zwei Systeme auswirken sind:

- Transaktionsverwaltung
  ALFRED unterstützt im Gegensatz zu Ingres eine transaktionsbasierte Verarbeitung von Befehlen, die beispielsweise asuch ein Rollback im Fehlerfall ermöglicht. Die Verwaltung von Transaktionen ist relativ zeitintensiv, weil sehr viele Informationen (wie zB. die Originalwerte von Datensätzen für das Rollback) organisiert werden müssen.

- Integritätsbedingungen
  ALFRED unterstützt bereits als Prototyp mehr Typen von Integritätsbedingungen als Ingres. In Ingres können nur IB der Form <`attribute`> <`compare-operator`> <`const`> definiert werden. Als Konstante kann aber kein NULL verwendet werden. Dies bedeutet, es können unter anderem keine NOT-NULL-IB definiert werden. Auch Primärschlüssel IB sind nicht möglich. Die Überprüfung dieser IB fällt also in Ingres ebenfalls weg.

- Benutzerschnittstelle
  ALFRED verfügt im Gegensatz zu Ingres PD V. 8.9 über ein grafisches Benutzer-Interface. Die Ausgabe von Meldungen an den Benutzer beispielsweise kostet somit einiges mehr an Zeit, als die einfachere Ausgabe auf einem zeichenorientierten Terminal.

Gemessene Zeiten	**ALFRED**	**Ingres**
Test-Transaktion 1	24.230	2.292
Test-Transaktion 2	27.690	2.664
Test-Transaktion 3	32.830	2.589
Test-Transaktion 4	29.400	3.043
Durchschnitt (1 - 4)	28.540	2.647

Tabelle 5: Insgesamt verbrauchte Zeit je Test-Einheit

Ein Vergleich der insgesamt benötigten Zeit je Test-Transaktion mit derjenigen eines Ingres Skripts der gleichen Befehle zeigt, dass ALFRED im Schnitt ca. 10 mal langsamer als Ingres ist. Diese Ergebnisse wurden auf dem Entwicklungsrechner (Intel Pentium 150 MHz mit Linux 2.0.33) (vgl. Kapitel 4.4.1) erzielt.

5.5.1 Beurteilung des Laufzeitverhaltens

Drei wichtige Ergebnisse dieser Laufzeitmessungen sind die von den Subsystemen AFPN, PED und ARFPN benötigten Verarbeitungszeiten. Das Subsystem AFPN benötigt für die Erzeugung eines Netzes mit etwa 180 Plätzen und Transitionen knapp 0.5 Sekunden. Darin inbegriffen ist die Zeit zum Lesen der benutzerdefinierten Transaktion aus dem Repository. Das Subsystem PED benötigt für die Erkennung von über 60 primitiven Ereignissen etwa 1 Sekunde. Das Subsystem ARFPN kopiert im Schnitt ca. 10 Regeln pro Aufruf und benötigt dafür etwa eine zehntel Sekunde. Alle diese Werte können als gut bis sehr gut betrachtet werden.

Ein weiteres wichtiges Ergebnis ist die durchschnittliche Verarbeitungsdauer eines Befehls in einer benutzerdefinierten Transaktion. Diese liegt bei ca. einer halben Sekunde. Darin inbegriffen ist die Verarbeitung von durchschnittlich etwa 10 Regeln. Dies bestätigt sich auch, wenn einzelne Datenmanipulationsbefehle (zB. `insert`) ausgeführt werden. Das Antwortverhalten des Systems kann für einen Prototypen demnach als befriedigend bis gut bewertet werden.

Das Laufzeitverhalten von ALFRED und von Ingres-Scripts lässt sich nur sehr bedingt vergleichen. Da in Ingres PD V. 8.9 keine NOT-NULL- und keine Schlüssel-IB definiert werden können, ist die Funktionalität von ALFRED viel grösser. In ALFRED werden neue Datensätze, die diese IB verletzen, zurückgewiesen. Ingres fügt neue Datensätze ungeprüft ein, was natürlich mit viel weniger Aufwand verbunden ist und somit schneller erfolgen kann. Für Vergleichsmessungen wäre ein anderes (aktives) DBMS, das ALFRED in den Funktionen mindestens ebenbürtig ist, besser geeignet. Diese Messungen konnten aber wegen des grossen Aufwands nicht mehr im Rahmen dieser Arbeit durchgeführt werden.

5.6 Schlussfolgerungen

Das Laufzeitverhalten des Prototypen hat sich als befriedigend bis gut gezeigt. Mit dem gewählten Messverfahren lässt sich jedoch nicht exakt bestimmen, wieviel Zeit effektiv für die Verarbeitung der Netze und wieviel für das Ausgeben von Meldungen ins Log-File und an den Benutzer verwendet wird. Die gemessenen Zeiten verringern sich klar, wenn keine Log-Dateien geschrieben werden. Allerdings wäre dann keine detaillierte Messung mehr möglich.

Anhand der oben dargestellten Ergebnisse lassen sich aber bezüglich Laufzeitverhalten und Optimierungsmöglichkeiten einige Aussagen machen.

5.6.1 Mögliche Optimierungen

Aufgrund der Erkenntnisse aus den Laufzeittests ergeben sich die folgenden Optimierungsmöglichkeiten:

- **Anzahl Regeln**
  Dadurch, dass im Prototypen nur primitive Ereignisse und Bedingungen spezifiziert werden können, müssen viel mehr Regeln verarbeitet werden, als mit komplexen Komponenten nötig wären. Zum Beispiel können sämtliche Regeln für die Überprüfung der NOT-NULL-Integritätsbedingungen zu einer Regel zusammengefasst werden. Dadurch reduziert sich bei 10 solchen IB die Anzahl der zu verarbeitenden Transitionen um 90%. Dies kann zu einer beträchtlichen Verminderung der benötigten Laufzeit führen.

- **Petri-Netze**
  Bei der Modellierung der Petri-Netze stand nicht der Performance-Aspekt im Vordergrund. Zum Beispiel sind in ECA-Regeln auch für den `False`-Fall `Begin-Of-Action (BOA)` und `End-Of-Action (EOA)`-Transitionen in der Aktionskomponente vorhanden. Dies bedeutet, dass auch in diesem Fall eine Action-Transaktion erzeugt werden muss. Vorteilhafter wäre die Verbindung des `CHOICE`-Platzes mit dem `End-Of-Rule (EOR)`-Platz über eine (nicht zu verarbeitende) `DUMMY`-Transition (vgl. Abbildung 35). Ähnliche Optimierungen lassen sich auch in der Modellierung von anderen Teilen der Petri-Netze finden.

- **Datenstrukturen**
  Die verwendeten Listen-Datenstrukturen (zB. für das Repository) zeichnen sich vor allem durch ihre Einfachheit aus und sind nur für die Implementierung eines Prototypen geeignet. Durch die Verwendung komplexerer Datenstrukturen (zB. Hashes) ist insbesondere bei grösseren Datenbeständen eine Verbesserung der Performance zu erreichen.

- **Programmierung**
  Das primäre Ziel einer Prototyp-Entwicklung ist es nicht, ein möglichst leistungsfähiges System zu bauen. Ein Prototyp dient lediglich dazu, zu belegen, dass die entwickelten Konzepte grundsätzlich umgesetzt

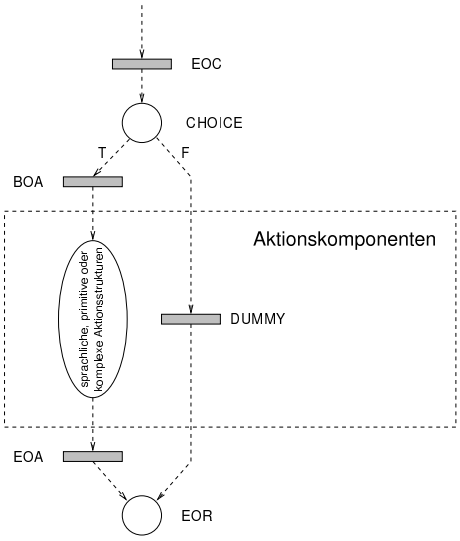


Abbildung 35: Optimierte Aktionskomponente einer ECA-Regel

werden können. Aus diesem Grund kann auch die Implementierung dieses Prototypen nicht optimal sein. In der Verbesserung gewisser Datenstrukturen und Algorithmen liegt ein relativ grosses Optimierungspotential.

Leider ist es nicht möglich, einen genauen Faktor anzugeben, um den die Leistung von ALFRED mit all diesen Verbesserungen erhöht werden könnte. Die Grössenordnung liegt aber schätzungsweise irgendwo zwischen 2 und 3.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Aktive Datenbanksysteme stellen eine Erweiterung zu traditionellen DBMS dar. Sie beinhalten Mechanismen, mit denen sie automatisch auf gewisse Ereignisse reagieren können. Um aktives Verhalten zu definieren, werden Regeln verwendet, die dem ECA-Paradigma folgen. Neben verschiedenen Regelstrukturen werden auch diverse ausführungsbezogene Eigenschaften (Semantiken) unterschieden.

Am Institut für Wirtschaftsinformatik der Universität Bern (Abteilung Information Engineering) sind Konzepte für eine aktive Schicht zur Erweiterung beliebiger (passiver) Datenbankmanagementsysteme zu aktiven DBMS erarbeitet worden. Diese aktive Schicht trägt den Namen ALFRED.
Die Hauptaufgabe der vorliegenden Arbeit bestand im Entwurf und in der Implementierung eines Prototypen für das Verarbeitungssystem von ALFRED. Es wurde entschieden, den Prototypen objektorientiert zu entwerfen, unter anderem weil die objektorientierte Methoden besser für grosse und komplexe Problemstellungen sind, als die strukturierten. Die verwendete Methode von Booch ist praxisorientiert und hat sich für den Entwurf eines relativ komplexen Softwaresystems bestens bewährt.

Die entworfenen Klassen bilden den Teil der Architektur von ALFRED ab, der für die Verarbeitung von Befehlen und benutzerdefinierten Transaktionen unbedingt benötigt wird. Dies sind die Module “AFPN”, “PED”, “NP” und “ARFPN”. Das Subsystem “Transaction System” wurde als Menge von Klassen entworfen, welche zusammen einen Treiber bilden. Dieser erlaubt es, Transaktionen zu verwalten und Transitionen zu verarbeiten. Für das Susbystem “Repository System” wurde ein Datenmodell, mit dem alle nötigen Informationen gespeichert werden können, entworfen. Die Realisierung des Datenmodells erfolgt durch zwei weitere Klassen.

Die Implementierung erfolgte vor allem aus Gründen der Portabilität mit Tcl/Tk (Benutzerschnittstelle) und C++ (Verarbeitungssystem). Diese Kombination hat sich als gut geeignet erwiesen. Insbesondere die Einfachheit der Schnittstelle und die gute Verarbeitungsgeschwindigkeit haben dabei überzeugt.

Nicht zufriedenstellend ist die Wahl von INGRES PD V. 8.9 als DBMS. Die Funktionalität dieser Version von Ingres genügt zwar als Datenhal-

tungssystem für ALFRED, erlaubt aber keinen Vergleich bezüglich der Leistungsfähigkeit, da auch grundlegende Eigenschaften eines passiven DBMS (wie zB. Primärschlüssel-IB) fehlen. Die Gründe für die Wahl dieses DBMS waren vor allem die freie Verfügbarkeit und der geringe Ressourcen-Bedarf.

Die für diese Diplomarbeit gesteckten Ziele konnten in unterschiedlichem Mass erfüllt werden:

- **Adaption der Konzepte**
  Durch die Realisierung des Prototypen konnten einige wichtige Erkenntnisse gewonnen werden. Dies betrifft vor allem die Bereiche Modellierung, Transaktionsverwaltung und Verarbeitung der Petri-Netze.
- **Laufzeitanalysen**
  Wegen der beschränkten Funktionalität des gewählten DBMS konnte nur der eine Teil der Laufzeitanalyse mit aussagekräftigen Ergebnissen durchgeführt werden. Der Vergleich des Prototypen mit einem Datenbankmanagementsystem, welches eine ähnliche Funktionalität bietet, konnte nicht durchgeführt werden. Trotzdem kann die Leistungsfähigkeit des implementierten Prototypen auf Grund der erzielten Resultate als gut beurteilt werden, da insbesondere die Erkennung von primitiven Ereignissen und die Einbindung von ausgelösten Regeln sehr effizient erfolgt:
  - Es können zum Beispiel mehr als 40 primitive Ereignisse in ca. einer halben Sekunde erkannt werden.
  - Das Integrieren von etwa 10 Regeln in das zu verarbeitende Petri-Netz benötigt durchschnittlich etwa 0.1 Sekunden.
  - Die Verarbeitung eines Befehls (inklusive der im Durchschnitt 10 Regeln) erfolgt in etwa 0.5 Sekunden.

Die Umsetzung der für ALFRED entwickelten Konzepte als Prototyp hat gezeigt, dass die Erwartungen an die potentielle Leistungsfähigkeit des Systems gerechtfertigt waren.

6.2 Ausblick

Bei der Definition der Regeln für das Testszenario (vgl. Kapitel 5.2) musste insbesondere darauf geachtet werden, dass keine nicht-terminierenden Zyklen entstanden, da diese im Prototypen während der Verarbeitung nicht

überwacht und behandelt werden. Diese Aufgabe hat sich bereits bei dieser relativ kleinen Anzahl von Regeln als recht schwierig erwiesen. Ein Regelanalyse-System, das solche Zyklen erkennt und den Benutzer darauf hinweist, wäre daher ein unverzichtbares Werkzeug bei der Entwicklung komplexerer Regelmengen.

Beim Testen hat sich gezeigt, dass es manchmal recht umständlich ist, eine einzelne Funktion immer über das Menüsystem starten zu müssen. Eine Vereinfachung der Handhabung ergäbe sich für solche sich wiederholenden Tätigkeiten, wenn auch eine Kommandozeilen-Schnittstelle zur Verfügung stehen würde.

Für die Verarbeitung der Petri-Netze wurde ein rekursiver Algorithmus entwickelt. Dieser basiert auf einigen wenigen Methoden der Klassen `place` und `transition`. Ein Vorteil dieses Algorithmus ist sicher die Einfachheit. Dem gegenüber stehen aber einige grosse Nachteile. Zum Beispiel ist es sehr schwierig, den Zustand des Petri-Netzes und der Verarbeitung zu kontrollieren. In der Entwicklungsphase ist das Finden von Fehlern nur mit einem viel grösseren Aufwand als bei nicht-rekursiven Algorithmen möglich. Der Entwurf des Moduls 'Net Processing' muss also so überarbeitet werden, dass eine Instanz geschaffen wird, welche die Kontrolle über die zu bewegende(n) Marke(n) im Netz hat.

Die verwendeten Listendatenstrukturen für das Repository sind für kleine Datenmengen, wie sie etwa im Rahmen der Prototyp-Entwicklung anfallen, sicher geeignet. Hier bestehen vor allem Optimierungsmöglichkeiten, indem Datenstrukturen mit direkterem und somit schnellerem Zugriff verwendet werden.

Zusammenfassend kann gesagt werden, dass der entwickelte Prototyp eine gute Basis für weitere Arbeiten darstellt. Eine Erweiterung der bestehenden Module, sowie eine Ergänzung mit weiteren Subystemen könnten den Nutzen, den aktive DBMS für Entwickler und Benutzer von Datenverarbeitungsanwendungen haben, weiter verdeutlichen.

A Syntax der ALFRED Rule Definition Language (ARDL)

A.1 Regelstrukturen

```
<rule_structure> ::= ('ON'   | 'ON'      'EVENT'        ) <rule_event>
                     ('IF'   | 'CHECK'   'CONDITION'    ) <rule_condition>
                     ('DO'   | 'EXECUTE' 'TRUE_ACTION'  ) <rule_action>
                     ('ELSE' | 'OR'      'FALSE_ACTION' ) <rule_action>

                   | ('ON'   | 'ON'      'EVENT'    ) <rule_event>
                     ('IF'   | 'CHECK'   'CONDITION') <rule_condition>
                     ('DO'   | 'EXECUTE' 'ACTION'   ) <rule_action>

                   | ('ON'   | 'ON'      'EVENT' ) <rule_event>
                     ('DO'   | 'EXECUTE' 'ACTION') <rule_action>
```

A.2 Ereigniskomponenten

```
<rule_event>    ::= <event>
                  | <lingual>

<event>         ::= '(' <event> ')'
                  | <primitive_event>
                  | <complex_event> [ <param_context> ]

<param_context> ::= 'RECENT'
                  | 'CHRONICAL'
                  | 'CONTINUOUS'
                  | 'CUMULATIVE'
```

A.2.1 Primitive Ereignisse

```
<primitive_event> ::= <db_event>
                    | <dd_event>
                    | <dm_event>
                    | <trans_event>
                    | <proc_event>
```

```
                    | <prog_event>
                    | <e_prog_event>
                    | <simu_event>
                    | <time_event>
                    | <abst_event>

<pre_post>        ::= 'PRE' | 'POST'

<db_event>        ::= <pre_post> <db_cmd>




<db_cmd>          ::= ( 'create'     'database'         )
                    | ( 'destroy'    'database' <ident> )
                    | ( 'connect'    'database' <ident> )
                    | ( 'disconnect' 'database' <ident> )

<dd_event>        ::= <pre_post> <dd_cmd>

<dd_cmd>          ::= ( 'create' 'object' 'type'      )
                    | ( 'create' 'view'              )
                    | ( 'create' 'rule'              )
                    | ( 'create' 'rule_set'          )
                    | ( 'create' 'user'              )
                    | ( 'create' 'transaction'       )
                    | ( 'create' 'stored' 'procedure' )
                    | ( 'create' 'program'           )
                    | ( 'create' 'embedded' 'program' )
                    | ( 'create' 'simulation'        )

                    | ( 'alter' 'object' 'type'      <ident> )
                    | ( 'alter' 'view'               <ident> )
                    | ( 'alter' 'rule'               <ident> )
                    | ( 'alter' 'rule_set'           <ident> )
                    | ( 'alter' 'user'               <ident> )
                    | ( 'alter' 'transaction'        <ident> )
                    | ( 'alter' 'stored' 'procedure' <ident> )
                    | ( 'alter' 'program'            <ident> )
                    | ( 'alter' 'embedded' 'program' <ident> )
```

```
                    | ( 'alter' 'simulation'         <ident> )

                    | ( 'destroy' 'object' 'type'      <ident> )
                    | ( 'destroy' 'view'               <ident> )
                    | ( 'destroy' 'rule'               <ident> )
                    | ( 'destroy' 'rule_set'           <ident> )
                    | ( 'destroy' 'user'               <ident> )
                    | ( 'destroy' 'transaction'        <ident> )
                    | ( 'destroy' 'stored' 'procedure' <ident> )
                    | ( 'destroy' 'program'            <ident> )
                    | ( 'destroy' 'embedded' 'program' <ident> )
                    | ( 'destroy' 'simulation'         <ident> )

                    | ( 'grant' 'object' 'type' 'access' )
                    | ( 'grant' 'file'          'access' )
                    | ( 'grant' 'privilege'              )

                    | ( 'revoke' 'object' 'type' 'access' )
                    | ( 'revoke' 'file'          'access' )
                    | ( 'revoke' 'privilege'              )


<dm_event>        ::= <pre_post> <dm_cmd> ( 'set' | 'inst' )

<dm_cmd>          ::= ( 'select'   'from' <ident> )
                    | ( 'insert'   'in'   <ident> )
                    | ( 'update'   'on'   <ident> )
                    | ( 'delete'   'from' <ident> )

<trans_event>     ::= <pre_post> <trans_cmd>

<trans_cmd>       ::= ( 'begin'  'of' 'transaction' | 'BOT' )
                    | ( 'abort'  'of' 'transaction' | 'AOT' )
                    | ( 'end'    'of' 'transaction' | 'EOT' )
                    | ( 'commit' 'of' 'transaction' | 'COT' )
                    | ( 'execute'     'transaction' <ident> )

<proc_event>      ::= <pre_post> 'exec' 'stored' 'procedure' <ident>
```

```
<prog_event>      ::= <pre_post> 'exec' 'program' <ident>

<e_prog_event>    ::= <pre_post> 'exec' 'embedded' 'program' <ident>

<simu_event>      ::= <pre_post> 'exec' 'simulation' <ident>

<time_event>      ::= [ <date> '@' ] <time>

<abst_event>      ::= 'abstract' 'event' <ident>
```

A.2.2 Komplexe Ereignisse

```
<complex_event> ::= <bool_opr>
                  | <choice_opr>
                  | <seq_opr>
                  | <rep_opr>
                  | <time_opr>
                  | <intvl_opr>

<bool_opr>      ::= <event> <bool_opr> <event>

<choice_opr>    ::= 'WEAK'   'CHOICE' <num> 'OF' '(' <event_list> ')'
                  | 'STRONG' 'CHOICE' <num> 'OF' '(' <event_list> ')'

<seq_opr>       ::= 'WEAK'   'SEQUENCE' '(' <event_list> ')'
                  | 'STRONG' 'SEQUENCE' '(' <event_list> ')'

<rep_opr>       ::= 'EVERY'         <num> 'OF' <event>
                  | 'EVERY' 'AFTER' <num> 'OF' <event>

<intvl_opr>     ::= [ 'NOT' ]   <event> 'IN' <interval>
                  | 'THE' <num> <event> 'IN' <interval>
                  | 'LAST'      <event> 'IN' <interval>

<interval>      ::= '(' <event> ';' <event> ')'

<time_opr>      ::= 'TIMESPAN' ( <time> | <num> <time_unit> ) 'AFTER'
                       <event> <start_pnt>
                  | 'EVERY' <time> <start_pnt>
```

```
                    | 'EVERY' <num> <time_unit> [ 'AT' <time> ] <start_pnt>
                    | 'EVERY' <num> <week_day>  [ 'AT' <time> ] <start_pnt>
                    | 'EVERY' <num> 'DAY' 'IN' 'MONTH' [ 'AT' <time> ]
                         <start_pnt>

<start_pnt>         ::= 'BEGINNING' ( 'NOW' | 'WITH' <event> )
```

A.3 Bedingungskomponenten

```
<rule_condition> ::= <condition>
                   | <lingual>

<condition>      ::= '(' [ 'not' ] <condition> ')'
                   | <primitive_condition>
                   | <complex_condition>
```

A.3.1 Primitive Bedingungen

```
<primitive_condition> ::= <predicate>
                        | <pred_query>
                        | <query>
                        | 'TRUE'
                        | 'FALSE'
```

A.3.2 Komplexe Bedingungen

```
<complex_condition> ::= <condition> <bool_opr> <condition>
                      | <bool_function>

<bool_function>     ::= 'function' <ident> '(' [ <par_list> ] ')'
```

A.4 Aktionskomponenten

```
<rule_action> ::= <action>
                | <lingual>

<action>      ::= <primitive_action>
```

```
                | <complex_action>
```

A.4.1 Primitive Aktionen

```
<primitive_action> ::= <dm_action>
                     | <mess_action>
                     | <rule_action>
                     | <canc_action>
                     | <raise_action>

<dm_action>        ::= <select_cmd>
                     | <insert_cmd>
                     | <update_cmd>
                     | <delete_cmd>

<mess_action>      ::= <message_cmd>

<rule_action>      ::= <enable_rule_cmd>
                     | <disable_rule_cmd>

<canc_action>      ::= <cancel_cmd>

<raise_action>     ::= <raise_cmd>
```

A.4.2 Komplexe Aktionen

```
<complex_action> ::= 'BEGIN' [ 'PARALLEL' ]
                        ( <prim_act_block>
                          | <trans_block>
                          | <proc_cmd>
                          | <prog_cmd>
                          | <e_prog_cmd> )
                     'END' [ 'PARALLEL' ] ;

<prim_act_block> ::= <primitive_action>  <prim_act_block>
                   | <primitive_action> <primitive_action>

<trans_block>    ::= <primitive_action> <trans_block>
```

```
                    | <ta_block>
                    | <exec_ta_cmd>

<proc_cmd>        ::= <exec_sp_cmd> <proc_cmd>
                    | <exec_sp_cmd>

<prog_cmd>        ::= <exec_pg_cmd> <prog_cmd>
                    | <exec_pg_cmd>

<e_prog_cmd>      ::= <exec_ep_cmd> <e_prog_cmd>
                    | <exec_ep_cmd>
```

A.5 Regelsemantiken

```
<rule_semantics> ::= 'IS'        <rule_state>
                     'PRIORITY'  <rule_order>
                     <rule_coupling>

<rule_state>     ::= ( 'ACTIVE' | 'PASSIVE' ) <rule_tmp_state>

<rule_tmp_state> ::= 'BETWEEN' '(' ( 'NOW' | <event> ) 'AND' <event> ')'

<rule_order>     ::= <integer>

<rule_coupling>  ::= [ 'WITH' ] 'EC_COUPLING'  <coupling_mode>
                     [ 'WITH' ] 'CtA_COUPLING' <coupling_mode>
                     [ 'WITH' ] 'CfA_COUPLING' <coupling_mode>

                   | [ 'WITH' ] 'EC_COUPLING' <coupling_mode>
                     [ 'WITH' ] 'CA_COUPLING' <coupling_mode>

                   | [ 'WITH' ] 'EA_COUPLING' <coupling_mode>


<coupling_mode>  ::= 'IMMEDIATE'
                   | 'DEFERRED'
                   | 'DETACHED' 'INDEPENDENT'
                   | 'DETACHED' 'SEQUENTIAL' 'DEPENDENT'
                   | 'DETACHED' 'PARALLEL'   'DEPENDENT'
                   | 'DETACHED' 'EXCLUSIVE'  'DEPENDENT'
```

B Entity-Relationship-Diagramme der Repositories

Die folgenden Abschnitte beschreiben das Datenmodell der Repositories. Um die Diagramme nicht zu überladen, wird auf die Angabe der Attribute der Entitäten und Relationen verzichtet.

B.1 Data-Repository

Die Abbildungen 36 und 37 beschreiben das Data-Repository.

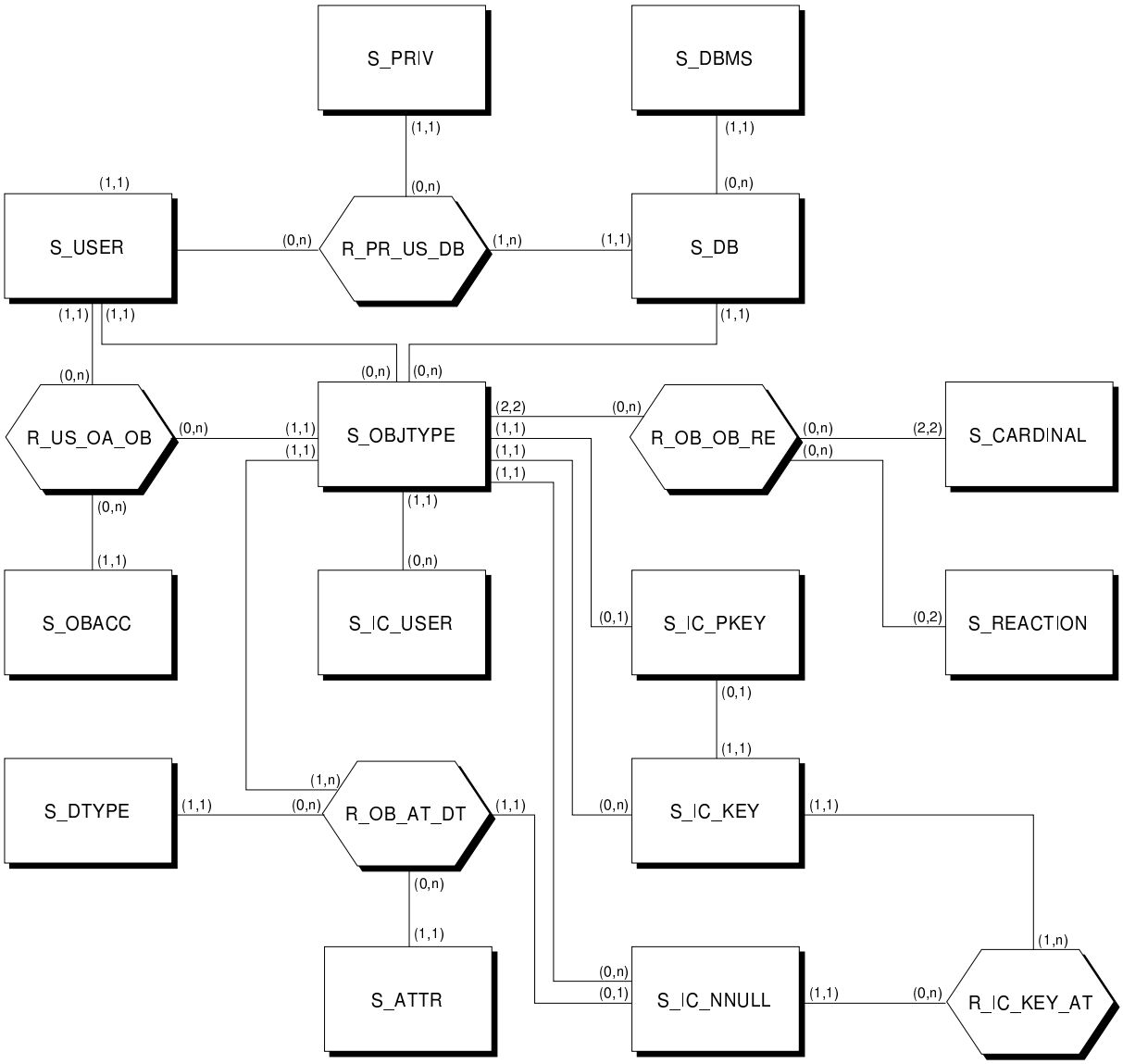


Abbildung 36: Data-Repository

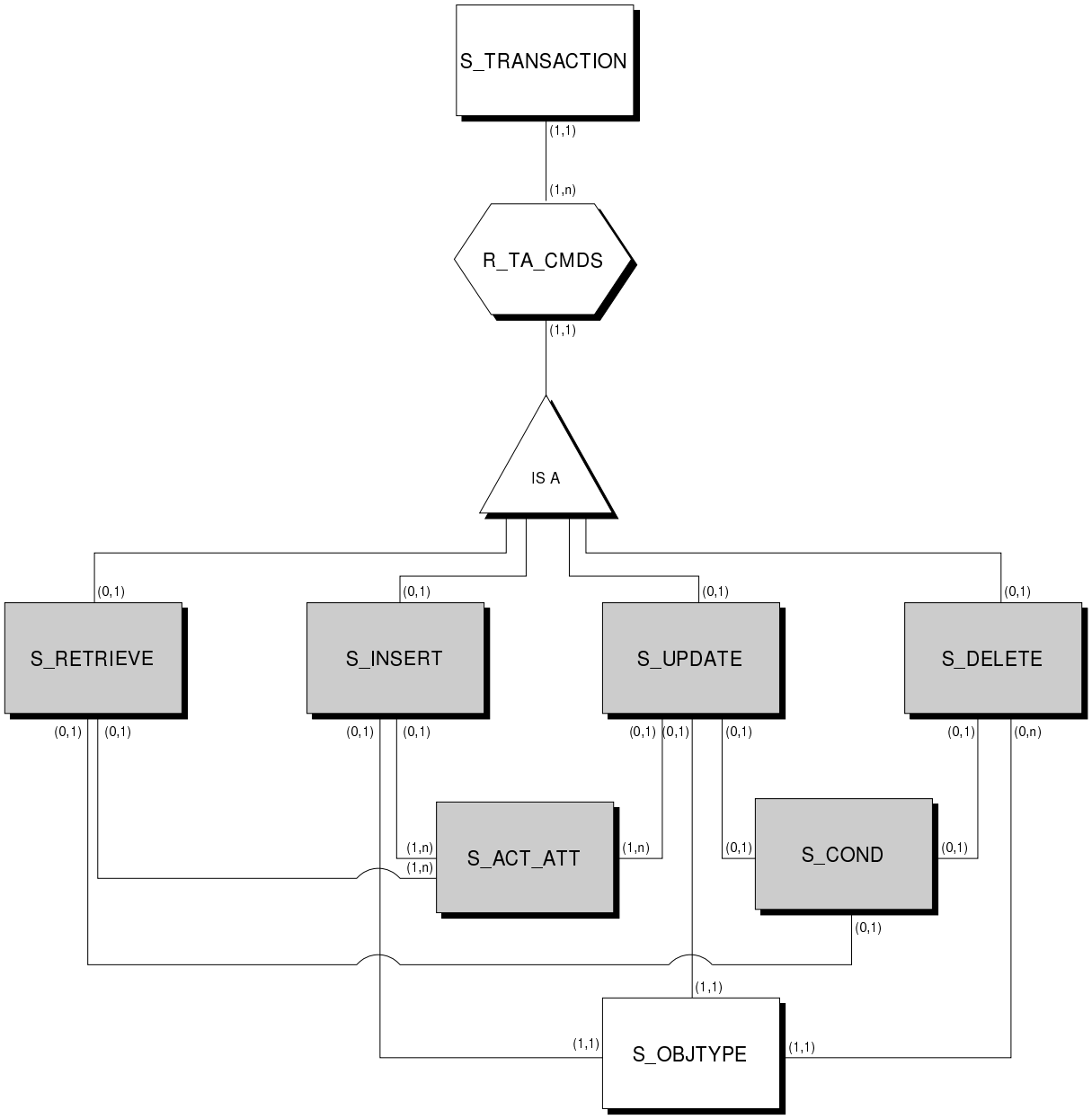


Abbildung 37: Transaktionen

B.2 Rule-Repository

Die Abbildungen 38 bis 43 beschreiben das Rule-Repository.

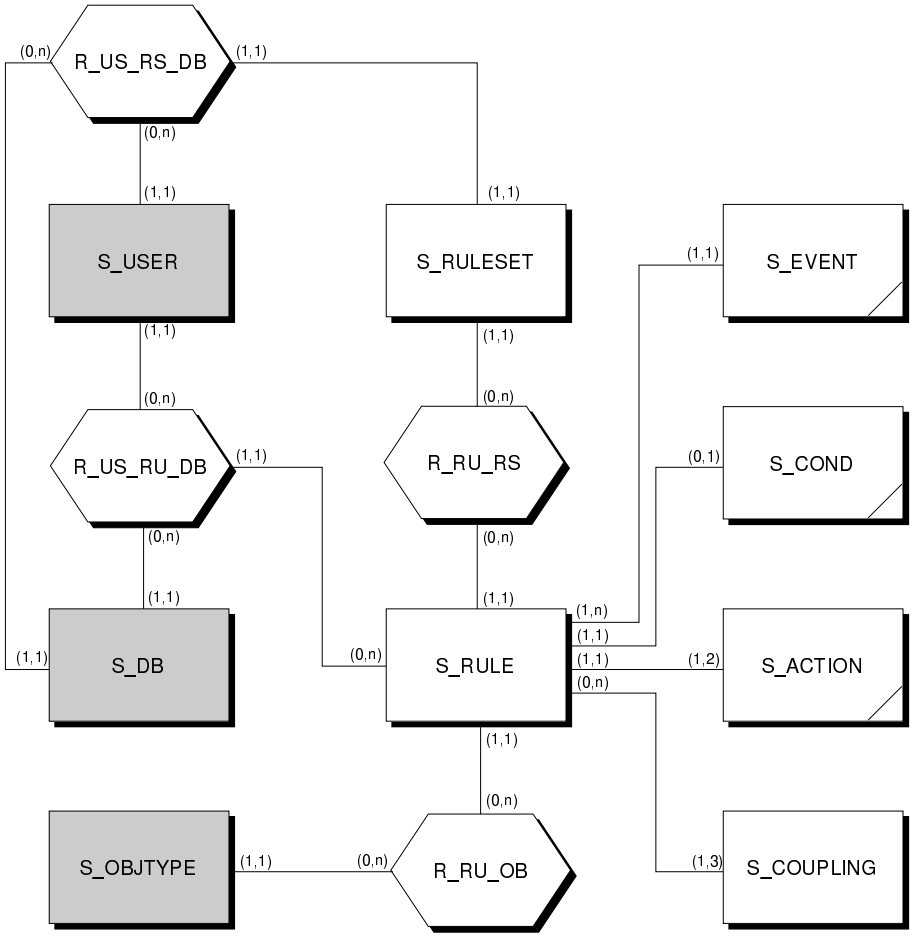


Abbildung 38: Rule-Repository

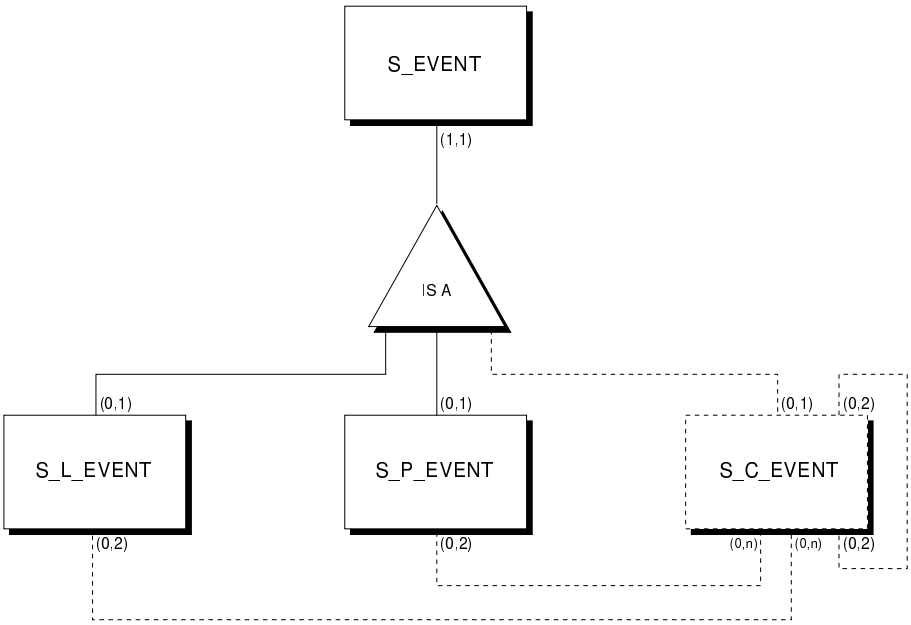


Abbildung 39: Ereignisse

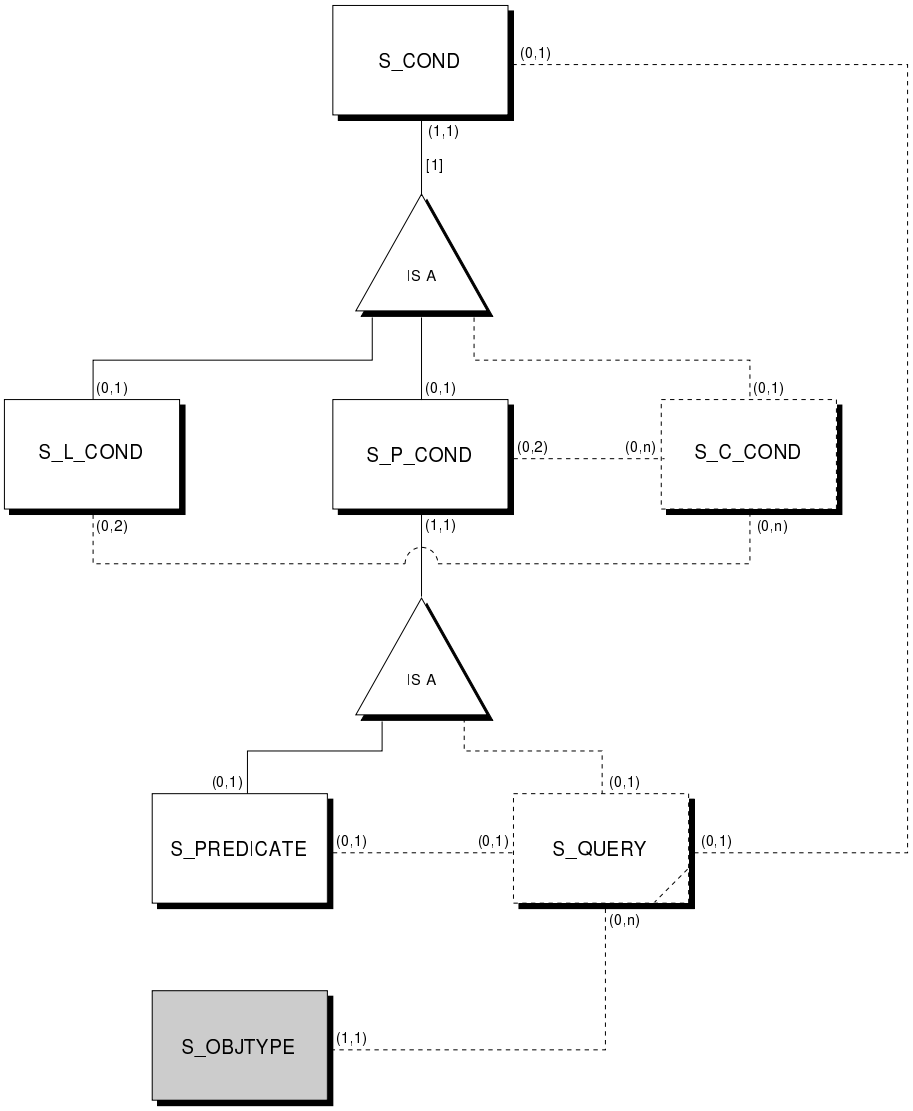


Abbildung 40: Bedingungen

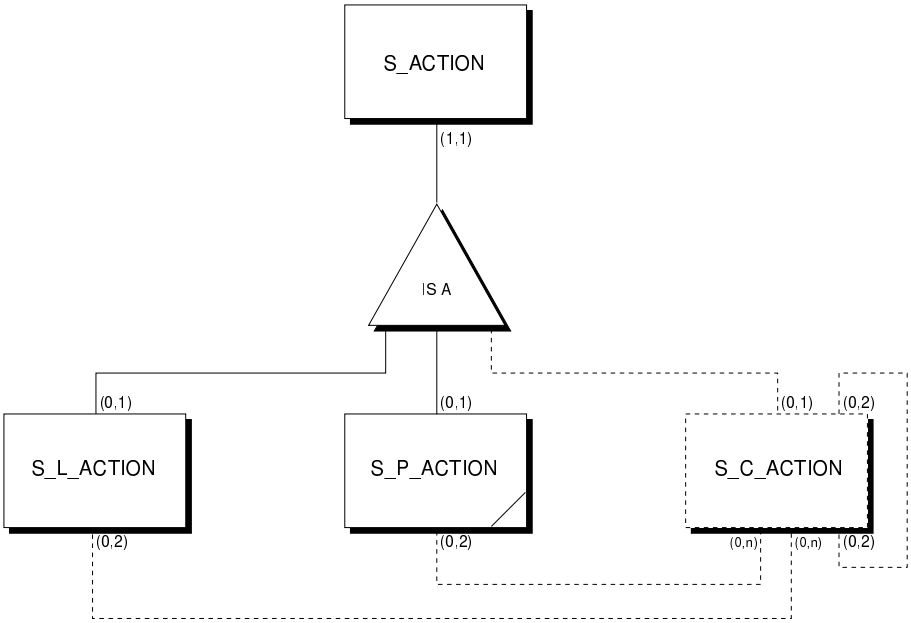


Abbildung 41: Aktionen

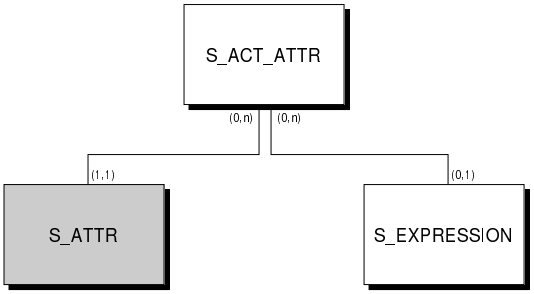


Abbildung 42: Attribute von Aktionen

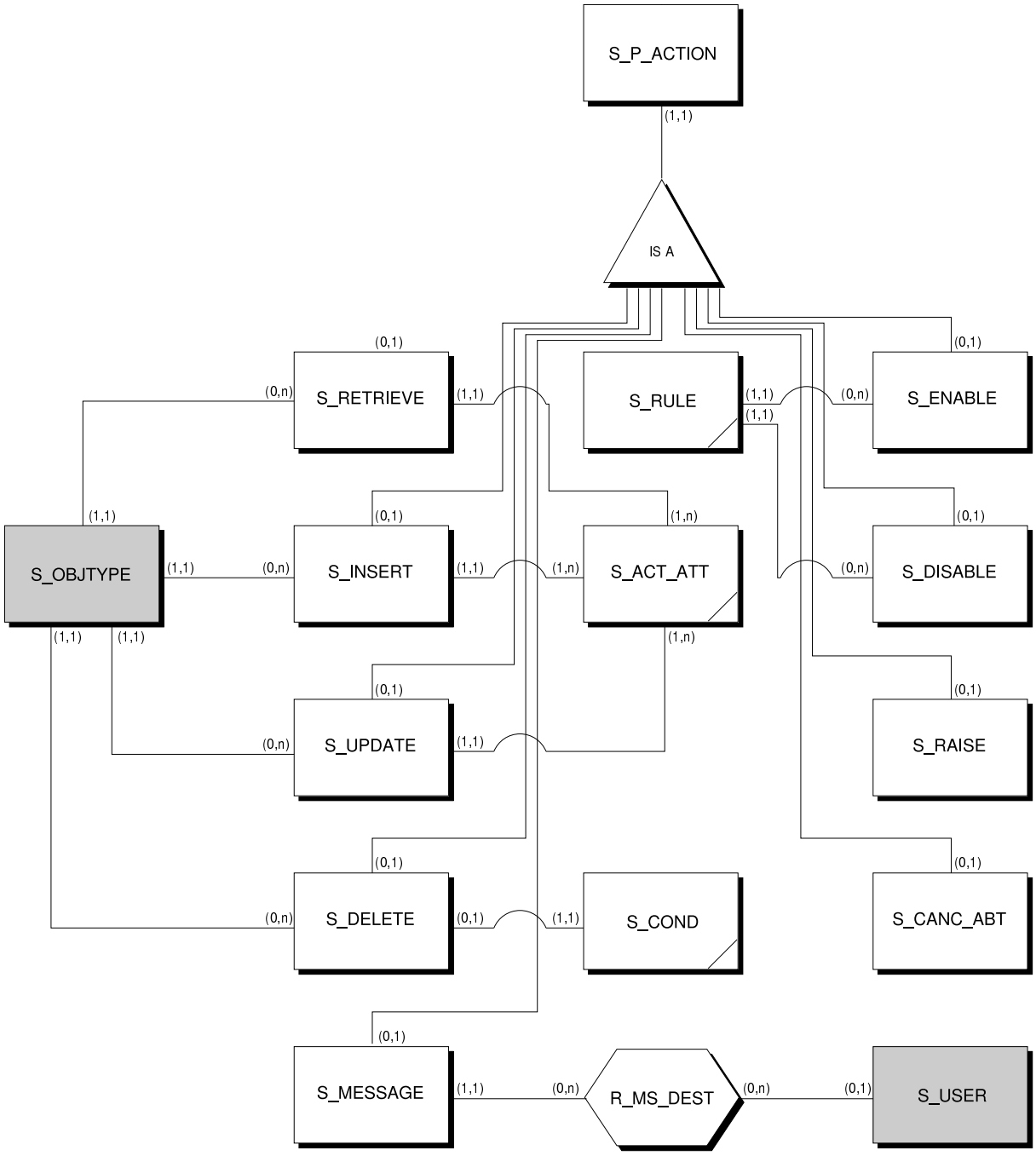


Abbildung 43: Primitive Aktionen

C Spezifikation der Klassen

Im diesem Anhang werden die Spezifikationen der wichtigsten Klassen aufgeführt. Jedes Attribut und jede Methode wird kurz beschrieben.

C.1 Klasse `subsys`

- Attribute
  - `afpn *AFPN`
    Ein Zeiger auf das Objekt und Subsystem AFPN.
  - `ped *PED`
    Ein Zeiger auf das Objekt und Subsystem PED.
  - `arfpn *ARFPN`
    Ein Zeiger auf das Objekt und Subsystem ARFPN.
  - `tmgr *TMGR`
    Ein Zeiger auf das Objekt und Subsystem TMGR (Transaktionsmanager).
  - `data_rep *DATA_REP`
    Ein Zeiger auf das Objekt und Subsystem DATA_REP (Data-Repository).
  - `rule_rep *RULE_REP`
    Ein Zeiger auf das Objekt und Subsystem RULE_REP (Regel-Repository).
  - `place_list *rules`
    Die Liste der als Petri-Netz dargestellten Regeln.
- Methoden
  - `subsys()`
    Öffnet die Log-Dateien und erzeugt die Instanzen der Subsysteme.
  - `~subsys()`
    Schliesst offene Log-Dateien und löscht die Instanzen der Subsysteme.
  - `execute_command()`
    Mit dieser Methode wird ein vom Benutzer spezifizierter Befehl im Verarbeitungssystem von ALFRED verarbeitet.

- `write_log()`
  Schreibt einen Eintrag in die Log-Datei.
- `write_time_log()`
  Schreibt einen Eintrag in die Zeit-Log-Datei.
- `get_sublist()`
  Bestimmt einen Listeneintrag in einer Tcl-Listendatenstruktur und gibt diesen als String zurück.

C.2 Klasse `bilist`

- Attribute
  - `_item *items`
    Das Basiselement der Liste.
  - `_item *p`
    Ein Zeiger auf das aktuelle Element in der Liste.
  - `int items_no`
    Die Anzahl Elemente in der Liste.
- Methoden
  - `bilist()`
    Erzeugt eine neue Liste.
  - ∼`bilist()`
    Gibt eine Liste und alle in ihr enthaltenen Elemente frei.
  - `remove()`
    Löscht das aktuelle Element aus der Liste und dekrementiert den Zähler `items_no`.
  - `empty()`
    Löscht alle Elemente aus der Liste und setzt den Zähler `items_no` auf 0.
  - `insert()`
    Fügt ein neues Element aufsteigend sortiert in die Liste ein.
  - `insert_here()`
    Fügt ein neues Element an der aktuellen Position ein.
  - `append()`
    Fügt ein neues Element ans Ende der Liste an.
  - `seek()`
    Setzt den Zeiger p auf ein zu suchendes Element.
  - `seek_first()`
    Setzt den Zeiger p auf das erste Element.
  - `seek_last()`
    Setzt den Zeiger p auf das letzte Element.
  - next()
    Setzt den Zeiger p auf das nächste Element.

- `prev()`
  Setzt den Zeiger p auf das vorhergehende Element.
- `get_item()`
  Bibt eine Referenz auf das aktuelle Element zurück.
- `get_items_no()`
  Gibt den Wert des Zählers `items_no` zurück.
- `erase_item()`
  Entfernt ein Element aus der Liste ohne es zu löschen. Dekrementiert den Zähler `items_no`.

C.3 Klasse `afpn`

- Attribute
  - `subsys *subsystems`
    Das einzige Attribut dieser Klasse ist ein Zeiger auf die Klasse `subsys`, die der Verwaltung aller Subsysteme dient. Dieser Zeiger wird verwendet für Zugriffe auf das Repository, sowie für die Ausgabe von Meldungen in das Log-File.
- Methoden
  - `afpn()`
    Dies ist der Konstruktor der Klasse `afpn`. Mit diesem wird eine Instanz der Kasse erzeugt und initialisiert. Er benötigt als Eingabe einen Zeiger auf das Objekt zur Verwaltung der Subsysteme.
  - `~afpn()`
    Der Destruktor löscht die Instanz und gibt reservierten Speicher wieder frei. Der Zeiger auf das Objekt `subsys` wird gelöscht (auf `NULL` gesetzt).
  - `run()`
    Diese Methode wird aufgerufen, um einen vom Benutzer spezifizierten Befehl in ein Petri-Netz umzusetzen. Dabei wird die benötigte Zeit gemessen und gewisse Informationen ins Log-File geschrieben.
  - `generate()`
    Diese Methode erzeugt das eigentliche Netz, das heisst sie baut Element für Element ein AFPN auf.

C.4 Klasse `ped`

- Attribute
  - `subsys *subsystems`
    Das einzige Attribut dieser Klasse ist ein Zeiger auf die Klasse `subsys`, die der Verwaltung aller Subsysteme dient. Dieser Zeiger wird verwendet für die Ausgabe von Meldungen in das Log-File.
- Methoden
  - `ped()`
    Dies ist der Konstruktor der Klasse `ped`. Mit diesem wird eine Instanz der Kasse erzeugt und initialisiert. Er benötigt als Eingabe einen Zeiger auf das Objekt zur Verwaltung der Subsysteme.
  - `~ped()`
    Der Destruktor löscht die Instanz und gibt reservierten Speicher wieder frei. Der Zeiger auf das Objekt `subsys` wird gelöscht (auf `NULL` gesetzt).
  - `check_command_transition()`
    Prüft, ob es sich bei einer Transition des Petri-Netzes um eine Transaktions-Transition handelt. Wenn ja wird das Petri-Netz nötigenfalls um gewisse Plätze und Transitionen erweitert und mit der entsprechenden Regel verbunden.
  - `check_transaction_transition()`
    Prüft, ob es sich bei einer Transition des Petri-Netzes um eine Befehls-Transition handelt. Wenn ja wird das Petri-Netz nötigenfalls um gewisse Plätze und Transitionen erweitert und mit der entsprechenden Regel verbunden.
  - `write_xfig()`
    Erzeugt eine XFig-Datei im XFig-Format 3.2.
  - `run()`
    Diese Methode wird aufgerufen, um nach der Erstellung eines Petri-Netzes, dieses auf primitive Ereignisse zu untersuchen. Dabei wird die benötigte Zeit gemessen und gewisse Informationen ins Log-File geschrieben sowie eine XFig-Datei erzeugt.

C.5 Klasse `place`

- Attribute
  - `name`
    Der Name eines Platzes zur Information.
  - `colour`
    Die Farbe eines Platzes. Diese bestimmt, wie viele Token auf einmal an die nächste Transition weitergegeben werden.
  - `rule_nr`
    Dieses Attribut ordnet einen Platz einer Regel zu. Die Regelnummer ist für das AFPN (d.h für das Netz das aus dem Benutzerbefehl erzeugt wird) 0. Plätze im Event-System (vgl. Kapitel 3.4) und Abbildung 5 haben die Regelnummer -1.
  - `tokens`
    Eine Liste von erhaltenen Marken.
  - `in_trans`
    Eine Liste von eingehenden Transitionen.
  - `out_trans`
    Eine Liste der ausgehenden Transitionen.
- Methoden
  - `place()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - ∼`place()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `connect_in_trans()`
    Diese Methode verbindet eine Transition als eingehende Transition.
  - `connect_out_trans()`
    Diese Methode verbindet eine Transition als ausgehende Transition.
  - `disconn_in_trans()`
    Diese Methode trennt eine eingehende Transition vom Platz.
  - `disconn_out_trans()`
    Diese Methode trennt eine ausgehende Transition vom Platz.

- `append_trans()`
  Mit dieser Methode wird eine Tansition (als ausgehende) an den Platz angefügt.
- `set_token()`
  Fügt ein Token an die Liste der bereits vorhandenen Token an.
- `set_name()`
  Ersetzt den Namen des Platzes.
- `set_colour()`
  Setzt die Farbe des Platzes neu.
- `get_out_transition()`
  Sucht eine ausgehende Transition.
- `get_in_transition()`
  Sucht eine eingehende Transition.
- `has_token()`
  Liefert `true` zurück, wenn ein Token vorhanden ist.
- `get_token()`
  Liefert das angeforderte Token zurück.
- `get_name()`
  Liefert den Namen des Platzes zurück.
- `get_rule_nr()`
  Liefert die Nummer der Regel, zu der der Platz gehört, zurück.
- `get_colour()`
  Liefert die Farbe des Platzes zurück.
- `instance()`
  Erzeugt eine Kopie von sich selbst und allen ausgehenden Transitionen.
- `send_token()`
  Liefert die erste Marke zurück. Diese wird dabei aus der Token-Liste entfernt.
- `fire_trans()`
  Versucht eine ausgehende Transition zu feuern, bis dies bei einer klappt.
- `out_xfig()`
  Schreibt XFig-Befehle zu Darstellung dieses Platzes in die Datei. Anschliessend wird `out_xfig` für alle ausgehenden Transitionen aufgerufen.

C.6 Klasse `transition`

- Attribute
  - `transition_type type`
    Der Typ der Transition.
  - `string name`
    Der Name der Transition.
  - `int rule_nr`
    Die Nummer der Regel, zu welcher die Transition gehört.
  - `token_list *tokens`
    Eine Liste zum sammeln aller Token, welche von den eingehenden Plätzen geliefert werden.
  - `string parameters`
    Parameter für die Verarbeitung der Transition durch den Transaktionsmanager.
  - `place_list *in_places`
    Eine Liste von eingehenden Plätzen.
  - `place_list *out_places`
    Eine Liste von ausgehenden Plätzen.
  - `subsys *subsystems`
    Ein Zeiger auf das Objekt `subsystems`. Über diesen Zeiger ist es auch Transitionen möglich, Meldungen ins Log-File zu schreiben.
- Methoden
  - `transition()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - `~transition()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `connect_in_place()`
    Verbindet einen eingehenden Platz.
  - `connect_out_place()`
    Verbindet einen ausgehenden Platz.
  - `disconn_in_place()`
    Löst die Verbindung zu einem eingehenden Platz.

- `disconn_out_place()`
  Löst die Verbindung zu einem ausgehenden Platz.
- `append_place()`
  Fügt einen ausgehenden Platz an.
- `set_params()`
  Setzt die Verarbeitungs-Parameter dieser Transition.
- `get_in_place()`
  Gibt einen Zeiger auf einen bestimmten eingehenden Platz zurück.
- `get_out_place()`
  Gibt einen Zeiger auf einen bestimmten ausgehenden Platz zurück.
- `get_name()`
  Gibt den Namen der Transition zurück.
- `get_rule_nr()`
  Gibt die Nummer der Regel, zu welcher die Transition gehört, zurück.
- `get_params()`
  Gibt die Verarbeitungs-Parameter für diese Transition zurück.
- `get_tokens()`
  Gibt eine Liste von Tokens zurück.
- `get_type()`
  Gibt den Typ der Transition zurück.
- `can_fire()`
  Prüft, ob alle Vorbedingungen für das Feuern erfüllt sind.
- `fire_token()`
  Feuert die Transition. Dabei werden von allen eingehenden Plätzen Tokens bezogen. Diese werden anschliessend an den Transaktionsmanager zur Verarbeitung weitergereicht. Der Transaktionsmanager seinerseits liefert einen einzigen aktualisierten Token zurück. Eine Kopie dieses Tokens wird nun an jeden ausgehenden Platz geschickt.
- `instance()`
  Erzeugt eine Kopie von sich selbst und allen ausgehenden Plätzen.
- `out_xfig()`
  Schreibt XFig-Befehle zu Darstellung dieser Transition in die Datei. Anschliessend wird `out_xfig` für alle ausgehenden Plätze aufgerufen.

C.7 Klasse `token`

- Attribute
  - `int user_id`
    Der Identifikator des aktuellen Benutzers.
  - `int db_id`
    Der Identifikator der verbunden Datenbank.
  - `event_time`
    Der Zeitpunkt, zu dem eine Regel ausgelöst worden ist.
  - `ta_id`
    Die Nummer der aktuellen Transaktion.
  - `ta_state`
    Der Status der aktuellen Transaktion.
  - `trig_tsn`
    Ein Zeiger auf die Transition, welche eine Regel ausgelöst hat.
  - `granularity`
    Die Granularität einer ausgelösten Regel.
  - `curr_tsn`
    Ein Zeiger auf die aktuelle Transition.
  - `curr_tsn_type`
    Der Typ der aktuellen Transition.
  - `cond_result`
    Das Ergebnis einer Bedingungsauswertung.
  - `cmd_type`
    Der Typ des auszuführenden Befehls.
  - `cmd_params`
    Die Parameter des auszuführenden Befehls.
- Methoden
  - `token()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - ∼`token()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `get_user_id()`
    Liefert die Benutzer-Identifikation zurück.

- `get_db_id()`
  Liefert die Datenbank-Identifikation zurück.
- `get_ta_state()`
  Liefert den Transaktionsstatus zurück.
- `get_ta_id()`
  Liefert die Transaktionsnummer zurück.
- `get_curr_tsn()`
  Liefert einen Zeiger auf die aktuelle Transition zurück.
- `get_curr_tsn_type()`
  Gibt den Typ der aktuellen Transition zurück.
- `get_trig_tsn()`
  Liefert einen Zeiger auf die regelauslösende Transition zurück.
- `get_cmd_type()`
  Gibt den Typ des aktuellen Befehls zurück.
- `get_cmd_params()`
  Liefert die Parameter für den aktuellen Befehl zurück.
- `get_granularity()`
  Gibt die Verarbeitungsgranularität für Regeln zurück.
- `set_ta_state()`
  Setzt den Transaktionsstatus.
- `set_ta_id()`
  Setzt die Transaktionsnummer.
- `set_curr_tsn()`
  Setzt einen Zeiger und den Typ der aktuellen Transition.
- `set_trig_tsn()`
  Setzt einen Zeiger auf die regelauslösende Transition.
- `set_cmd_type()`
  Setzt den Typ des aktuellen Befehls.
- `set_cmd_params()`
  Setzt die Parameter für den aktuellen Befehl.
- `set_granularity()`
  Setzt die Verarbeitungsgranularität für Regeln.
- `set_event_time()`
  Setzt die Zeit, wann ein Ereignis eingetreten ist.

C.8 Klasse `arfpn`

- Attribute
  - `subsys *subsystems`
- Methoden
  - `arfpn()` Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - ~`arfpn()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `run()`
    Diese Methode führt die eigentliche Verarbeitung durch. Zuerst wird für jede Regel, die in ihrem EOE-Platz einen Token hat, eine Instanz, das heisst, eine Kopie der Regel erzeugt. Anschliessend werden diese Regeln gemäss ihren Prioritäten miteinander und mit dem regelauslösenden Netz verbunden. Darauf werden die Token von den EOE-Plätzen aus weitergefeuert.
  - `get_triggering_trans()`
    Bestimmt, von welcher Transition die Regel ausgelöst worden ist.
  - `get_eoe_of_rule()`
    Bestimmt den EOE-Patz einer Regel.
  - `get_eor_of_rule()`
    Bestimmt den EOR-Platz einer Regel.

C.9 Klasse `tmgr`

- Attribute
  - `subsys *subsystems`
    Dieses Attribut ist ein Zeiger auf die Klasse `subsys`, die der Verwaltung aller Subsysteme dient. Dieser Zeiger wird verwendet für die Ausgabe von Meldungen in das Log-File sowie für den Zugriff auf andere Subsysteme (z.B. die Repositories).
  - `transaction_list *transactions`
    Dies ist ein Zeiger auf eine Liste aller noch laufenden Transaktionen.
  - `int last_ta_nr`
    Dies ist die Nummer der zuletzt erzeugten Transaktion.
- Methoden
  - `tmgr()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - ∼`tmgr()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `execute()`
    Diese Methode führt die Verarbeitung einer Transition durch.
  - `create_transaction()`
    Diese Methode erzeugt eine neue Transaktion.
  - `get_transaction()`
    Diese Methode liefert einen Zeiger auf eine bestimmte Transaktion aus der Transaktionsliste zurück.
  - `delete_transaction()`
    Diese Methode löscht eine Transaktion samt allen ihren Subtransaktionen.
  - `commit_transaction()`
    Mit dieser Methode wird eine Transaktion auf der Datenbank festgeschrieben.
  - `abort_transaction()`
    Diese Methode verwirft alle Transaktionsergebnisse.

- `get_top_ta_nr()`
  Diese Methode bestimmt die Nummer der Wurzel des Transaktionsbaumes.
- `parse_expression()`
  Durchsucht einen Ausdruck auf Konstanten- und Variablennamen und ersetzt diese durch ihre Werte. Dazu werden die Informationen im Token verwendet.
- `eval_prim_cond()`
  Bestimmt den Wert einer primitiven Bedingung. Varibalen- und Konstantennamen werden dabei durch iher Werte ersetzt.
- `update_rec()`
  Aktualisiert gewisse Attribute eines Datensatzes.
- `db_retrieve()`
  Liest Datensätze aus einer Tabelle einer Datenbank.
- `db_insert()`
  Fügt einen Datensatz in eine Tabelle einer Datenbank ein.
- `db_update()`
  Aktualisiert einen oder mehrere Datensätze einer Tabelle einer Datenbank.
- `db_delete()`
  Löscht Datensätze aus einer Tabelle einer Datenbank.

C.10 Klasse `transaction`

- Attribute
  - `int nr`
    Ein eindeutiger Identifikator für Transaktionen.
  - `clock_t start`
    Der Zeitpunkt der Erzeugung der Transaktion.
  - `ta_type type`
    Der Typ der Transaktion. Dies kann `ROOT`, `CONDITION`, `ACTION` oder `COMMAND` sein.
  - `ta_state_type state`
    Der Zustand der Transaktion. Es gibt vier mögliche Zustände: `RUNNING`, `WAITING`, `CANCELLED` und `ABORTED`.
  - `transition *first_tsn`
    Dies ist die erste Transition im Netz, die zu dieser Transaktion gehört.
  - `transition *curr_tsn`
    Dies ist die aktuell zu feuernde Transition im Netz.
  - `transition_type curr_tsn_type`
    Dies ist der Typ der aktuellen Transition.
  - `int top_ta`
    Die Nummer der Wurzel-Transaktion des Transaktionsbaumes.
  - `int sup_ta`
    Die Nummer der Muttertransaktion im Transaktionsbaum.
  - `int_list dep_ta`
    Eine Liste von Nummern von Tochtertransaktionen.
  - `trans_rel_list curr_data`
    Eine Liste von Übergangsrelationen. Diese enthalten alle Datensätz, die in dieser Transaktion bearbeitet werden. Sie wird CDS genannt.
  - `trans_rel_list other_data`
    Eine Liste von Übergangsrelationen. Diese enthalten alle Datensätz, die in Tochtertransaktion bearbeitet worden sind. Sie wird ODS genannt.

  - `trans_rel_list ret_data`
    Eine Liste von Übergangsrelationen. Diese enthalten alle Datensätz, die für das Anzeigen (mit dem `select`-Befehl) selektiert worden sind. Sie wird RDS genannt.
  - `token_list *tokens`
    Eine Liste von Tokens.

- Methoden
  - `transaction()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - ∼`transaction()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `get_ta_state()`
    Liefert den Transaktionsstatus zurück.
  - `get_ta_type()`
    Gibt den Typ der Transaktion zurück.
  - `get_ta_nr()`
    Gibt die Nummer der Transaktion zurück.
  - `get_top_ta()`
    Gibt die Nummer der Wurzeltransaktion zurück.
  - `get_sup_ta()`
    Liefert die Nummer der Muttertransaktion zurück.
  - `get_dep_ta()`
    Gibt die Liste mit den Nummern der Tochtertransaktionen zurück.
  - `get_curr_tsn()`
    Gibt einen Zeiger auf die aktuell zu feuernde Transition im Netz zurück.
  - `get_start_time()`
    Gibt die Zeit der Erzeugung der Transaktion zurück.
  - `set_ta_state()`
    Setzt den Transaktionsstatus.
  - `set_curr_tsn()`
    Setzt den Zeiger auf die aktuell zu verarbeitende Transition neu.
  - `append_dep_ta()`
    Fügt die Nummer einer neuen Tochtertransaktion in die Liste ein.

- `delete_dep_ta()`
  Löscht die Nummer einer Tochtertransaktion aus der Liste.
- `merge_record_curr_data()`
  Verschmilzt einen Datensatz einer Tochtertransaktion mit CDS.
- `merge_record_other_data()`
  Verschmilzt einen Datensatz einer Tochtertransaktion mit ODS.
- `append_curr_data()`
  Fügt den Datensätzen in CDS einen neuen Datensatz hinzu, wenn er noch nicht existiert.
- `append_curr_data_ret()`
  Fügt den Datensätzen in CDS einen neuen Datensatz, der dem Benutzer angezeigt werden soll, hinzu, wenn er noch nicht existiert.
- `set_curr_data_new()`
  Setzt die neuen (`:NEW`) Werte eines Datensatzes in einer Übergangsrelation in CDS.
- `set_curr_data_old()`
  Ersetzt die alten (`:OLD`) Werte eines Datensatzes in einer Übergangsrelation in CDS.
- `append_other_data()`
  Fügt den Datensätzen in ODS einen neuen Datensatz hinzu, wenn er noch nicht existiert.
- `append_other_data_ret()`
  Fügt den Datensätzen in ODS einen neuen Datensatz, der dem Benutzer angezeigt werden soll, hinzu, wenn er noch nicht existiert.
- `set_other_data_new()`
  Setzt die neuen (`:NEW`) Werte eines Datensatzes in einer Übergangsrelation von ODS.
- `set_other_data_old()`
  Ersetzt die alten (`:OLD`) Werte eines Datensatzes in einer Übergangsrelation von ODS.
- `append_other_rel()`
  Fügt der Liste der Übergangsrelationen in ODS eine neue hinzu.
- `get_curr_max_rel()`
  Liefert die Anzahl Übergangsrelationen in CDS zurück.

- `get_curr_rel_data()`
  Liefert Daten wie Name, Datenbank etc. einer Übergangsrelation in CDS zurück.
- `get_curr_max_rec()`
  Gibt die Anzahl Datensätze in einer Übergangsrelation von CDS an.
- `get_other_max_rel()`
  Liefert die Anzahl Übergangsrelationen in ODS zurück.
- `get_other_rel_data()`
  Liefert Daten wie Name, Datenbank etc. einer Übergangsrelation in ODS zurück.
- `get_other_max_rec()`
  Gibt die Anzahl Datensätze in einer Übergangsrelation von ODS an.
- `get_copy_curr_data()`
  Erzeugt eine Kopie aller Übergangsrelationen in CDS.

C.11 Klasse `trans_rel`

- Attribute
  - `string db_name`
    Der Name der Datenbank, zu welcher die Übergangsrelation gehört.
  - `string name`
    Der Name der Übergangsrelation.
  - `string attributes`
    Eine Liste von Attributen.
  - `record_list *records`
    Eine Liste von Datensätzen.
- Methoden
  - `trans_rel()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - `~trans_rel()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `get_max_record_nr()`
    Liefert die Anzahl vorhandener Datensätze zurück.
  - `append_record()`
    Fügt einen Datensatz hinzu.
  - `delete_record()`
    Löscht einen Datensatz.
  - `get_record()`
    Gibt einen Zeiger auf einen Datensatz zurück.
  - `get_record_values()`
    Liefert die Werte eines Datensatzes zurück.
  - `get_name()`
    Gibt den Namen der Übergangsrelation zurück.
  - `get_db_name()`
    Gibt den Namen der Datenbank zurück.
  - `get_attr_list()`
    Liefert die Liste der Attribute dieser Übergangsrelation.
  - `get_rec_list()`
    Gibt die Liste der Datensätze zurück.

C.12 Klasse `record`

- Attribute
  - `int nr`
    Die Nummer ist ein eindeutiger Identifikator für einen Datensatz.
  - `string ret_values`
    Dieses Attribut enthält die Werte eines Datensatzes, wie sie in der Datenbank gespeichert sind und von dort gelesen worden sind. Die Werte sind als Tcl-Liste in einem String gespeichert.
  - `string new_values`
    Dieses Attribut entält die aktuellen Werte des Datensatzes. Die Werte sind als Tcl-Liste in einem String gespeichert.
  - `string old_values`
    Dieses Attribut entält die Werte, die vor den letzten Änderungen aktuell waren, also die vorletzten Werte. Die Werte sind als Tcl-Liste in einem String gespeichert.
  - `string state`
    Der Status gibt an, ob ein Datensatz verändert (`CHANGED`), gelöscht (`DELETED`) oder unverändert (`UNCHANGED`) ist.
- Methoden
  - `record()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - `~record()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `get_number()`
    Gibt die Nummer des Datensatzes zurück.
  - `get_ret_values()`
    Gibt die von der Datenbank gelesenen Werte zurück.
  - `get_new_values()`
    Gibt die neuen Werte (`:NEW`) des Datensatzes zurück.
  - `get_old_values()`
    Gibt die alten Werte (`:OLD`) des Datensatzes zurück.
  - `get_state()`
    Gibt den Status des Datensatzes zurück.

- `set_ret_values()`
  Setzt die von der Datenbank gelesenen Werte.
- `set_new_values()`
  Setzt die neuen Werte (`:NEW`) des Datensatzes.
- `set_old_values()`
  Setzt die alten Werte (`:OLD`) des Datensatzes.
- `set_state()`
  Setzt den Status des Datensatzes.
- `upd_new_values()`
  Aktualisiert die neuen (`:NEW`) Werte des Datensatzes. Die alten (`:OLD`) werden dabei durch die bisherigen neuen ersetzt.
- `replace_old_new()`
  Ersetzt `:OLD` und `:NEW` Werte durch identische neue Werte.

C.13 Klasse `data_rep`

- Attribute
  - `subsys *subsystems`
    Ein Zeiger auf die Klasse `subsys`, die der Verwaltung aller Subsysteme dient. Dieser Zeiger wird verwendet für Zugriffe auf das Repository, sowie für die Ausgabe von Meldungen in das Log-File.
  - `str_item_list *sys_dbms`
    Eine Liste von Datenbanksystemen.
  - `str_item_list *sys_database`
    Eine Liste von Datenbanken.
  - `user_list *sys_user`
    Eine Liste von Benutzern.
  - `str_item_list *sys_user_group`
    Eine Liste von Benutzergruppen.
  - `str_item_list *sys_privilege`
    Eine Liste von Privilegien.
  - `str_item_list *sys_objtype`
    Eine Liste von Objekttypen.
  - `str_item_list *sys_attribute`
    Eine Liste von Attributen.
  - `str_item_list *sys_datatype`
    Eine Liste von Datentypen.
  - `ic_list *sys_ic_nn`
    Eine Liste von Not-Null-Integritätsbedingungen (IB).
  - `ic_list *sys_ic_key`
    Eine Liste von Schlssel-IB.
  - `ic_list *sys_ic_pkey`
    Eine Liste von Primärschlüssel-IB.
  - `ic_list *sys_ic_userdef`
    Eine Liste von benutzerdefinierten IB.
  - `str_item_list *sys_cardinality`
    Eine Liste von Kardnalitäten für Beziehungen zwischen Objekttypen.
  - `str_item_list *sys_reaction`
    Eine Liste von Reaktionen auf Datenmanipulationen.

- `pred_list *sys_predicate`
  Eine Liste von Prädikaten.
- `expr_list *sys_expression`
  Eine Liste von Ausdrücken.
- `str_item_list *sys_simulation`
  Eine Liste von Simulationen.
- `str_item_list *sys_transaction`
  Eine Liste von benutzerdefinierten Transaktionen.
- `tupel_tstamp_list *db_dbms`
  Eine Liste von Beziehungen zwischen Datenbanken und Datenbankmanagementsystemen.
- `tripel_tstamp_list *user_priv_db`
  Eine Liste von Beziehungen zwischen Benutzern und Privilegien.
- `tripel_tstamp_list *user_usergroup_db`
  Eine Liste von Beziehungen zwischen Benutzern und Benutzergruppen.
- `tripel_tstamp_list *user_objtype_db`
  Eine Liste von Beziehungen zwischen Benutzern und Objekttypen.
- `us_oa_list *user_obj_access`
  Eine Liste von Beziehungen zwischen Benutzern, Objekttype und Zugriffsrechten.
- `ob_attr_list *ob_attr_dt`
  Eine Liste von Beziehungen zwischen Objekttypen, Attributen und Datentypen.
- `tupel_list *key_notnull`
  Eine Liste von Beziehungen zwischen Schlssel-IB und Not-Null-IB
- `relship_list *ob_ob_re`
  Eine Liste von Beziehungen zwischen zwei Objekttypen und einem Beziehungstyp.
- `tripel_tstamp_list *user_sim_db`
  Eine Liste von Beziehungen zwischen Benutzern, Simulationen und Datenbanken.
- `si_ta_cmd_list *sim_commands`
  Eine Liste von Beziehungen zwischen Simulationen und ihren Kommandos.

  - `us_fa_list *user_sim_access`
    Eine Liste von Beziehungen zwischen Benutzern, Simulationen und Zugriffsrechten.
  - `tripel_tstamp_list *user_ta_db`
    Eine Liste von Beziehungen zwischen Benutzern, benutzerdefinierten Transaktionen und Datenbanken.
  - `si_ta_cmd_list *ta_commands`
    Eine Liste von Beziehungen zwischen benutzerdefinierten Transaktionen und ihren Kommandos.
  - `us_fa_list *user_ta_access`
    Eine Liste von Beziehungen zwischen Benutzern, benutzerdefinierten Transaktionen und Zugriffsrechten.

- Methoden

  - `data_rep()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - `~data_rep()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `append_db()`
    Fügt eine Datenbank an die Liste der Datenbanken an.
  - `append_user()`
    Fügt einen Benutzer an die Liste der Benutzer an.
  - `append_objtype()`
    Fügt einen Objekttypen an die Liste der Objekttypen an.
  - `append_transaction()`
    Fügt eine benutzerdefinierte Transaktion an die Liste der Transaktionen an.
  - `append_simulation()`
    Fügt eine Simulation an die Liste der Simulationen an.
  - `append_priv_us()`
    Fügt ein Privileg fr einen Benutzer an die Liste der Benutzerprivilegien an.
  - `delete_db()`
    Löscht eine Datenbank aus der Liste der Datenbanken.
  - `delete_user()`
    Löscht einen Benutzer aus der Benutzerliste.

- `delete_objtype()`
  Löscht einen Objekttypen aus der Liste der Objekttypen.
- `delete_transaction()`
  Löscht eine benutzerdefinierte Transaktion aus der Liste der Transaktionen.
- `delete_simulation()`
  Löscht eine Simulation aus der Liste der Simulationen.
- `get_dbms_list()`
  Liest die Namen aller DBMS aus der Liste.
- `get_db_list()`
  Liest die Namen aller Datenbanken aus der Liste.
- `get_us_list()`
  Liest die Namen aller Benutzer aus der Liste.
- `get_ob_list()`
  Liest die Namen aller Objekttypen aus der Liste.
- `get_ob()`
  Liest einen kompletten Objekttypen aus dem Repository.
- `get_priv_list()`
  Liest die Namen aller Privilegien aus der Liste.
- `get_privs()`
  Liest die Privilegien eines Benutzers.
- `get_relship_name_list()`
  Liest die Namen aller Beziehungen zwischen Objekttypen aus der Liste.
- `get_oa_rights()`
  Liest alle Zugriffsrechte eines Benutzers aus der Liste.
- `get_attr_list()`
  Liest die Namen aller Attribute aus der Liste.
- `get_ob_attr_list()`
  Liest alle Attribute eines Objekttypen.
- `get_dt_list()`
  Liest die Namen aller Datentypen aus der Liste.
- `get_predicate_list()`
  Liest alle Prädikate aus der Liste.
- `get_simulation_list()`
  Liest eine komplette Simulation inkl. allen ihren Kommandos.

- `get_transaction_list()`
  Liest die Namen aller benutzerdefinierten Transaktionen aus der Liste.
- `get_transaction()`
  Liest eine komplette benutzerdefiniere Transaktion inkl. allen ihren Kommandos.

C.14 Klasse `rule_rep`

- Attribute
  - `subsys *subsystems`
    Ein Zeiger auf die Klasse `subsys`, die der Verwaltung aller Subsysteme dient. Dieser Zeiger wird verwendet für Zugriffe auf das Repository, sowie für die Ausgabe von Meldungen in das Log-File.
  - `str_item_list *sys_couplings`
    Eine Liste von Kopplungsmodi.
  - `str_item_list *sys_ruleset`
    Eine Liste von Regelmengen.
  - `rule_list *sys_rule`
    Eine Liste von Regeln.
  - `rule_comp_list *sys_event`
    Eine Liste von Ereignissen.
  - `ling_list *sys_ling_event`
    Eine Liste von sprachlichen Ereignissen.
  - `prim_event_list *sys_prim_event`
    Eine Liste von primitiven Ereignissen.
  - `comp_list *sys_comp_event`
    Eine Liste von komplexen Ereignissen.
  - `rule_comp_list *sys_condition`
    Eine Liste von Bedingungen.
  - `ling_list *sys_ling_cond`
    Eine Liste von sprachlichen Bedingungen.
  - `prim_cond_list *sys_prim_cond`
    Eine Liste von primitiven Bedingungen.
  - `comp_list *sys_comp_cond`
    Eine Liste von komplexen Bedingungen.
  - `rule_comp_list *sys_action`
    Eine Liste von Aktionen.
  - `ling_list *sys_ling_action`
    Eine Liste von sprachlichen Aktionen.
  - `prim_action_list *sys_prim_action`
    Eine Liste von primitiven Aktionen.

- `comp_action_list *sys_comp_action`
  Eine Liste von komplexen Aktionen.
- `action_attr_list *sys_action_attribute`
  Eine Liste von Attributen für eine primitive Aktion.
- `tripel_list *sys_retrieve`
  Eine Liste von Retrieve-Aktionen.
- `tripel_list *sys_update`
  Eine Liste von Update-Aktionen.
- `tripel_list *sys_delete`
  Eine Liste von Delete-Aktionen.
- `tupel_list *sys_insert`
  Eine Liste von Insert-Aktionen.
- `msg_list *sys_message`
  Eine Liste von Meldungsaktionen.
- `tupel_list *sys_enable`
  Eine Liste von Enable-Aktionen.
- `tupel_list *sys_disable`
  Eine Liste von Disable-Aktionen.
- `tupel_list *sys_raise`
  Eine Liste von Ereignis-Auslöseaktionen.
- `seq_action_list *seq_action`
  Eine Liste sequentiellen Aktionen fr komplexe Aktionen.
- `tripel_tstamp_list *user_rule_db`
  Eine Liste von Beziehungen zwischen Benutzern, Regeln und Datenbanken.
- `tripel_tstamp_list *user_ruleset_db`
  Eine Liste von Beziehungen zwischen Benutzern, Regelmengen und Datenbanken.
- `tupel_list *rule_objtype`
  Eine Liste von Beziehungen zwischen Regeln und Objekttypen.
- `tupel_list *rule_ruleset`
  Eine Liste von Beziehungen zwischen Regeln und Regelmengen.
- `tupel_list *msg_dest`
  Eine Liste von Beziehungen zwischen Meldungen und Benutzern oder Benutzergruppen.

- Methoden
  - `rule_rep()`
    Dies ist der Konstruktor der Klasse. Er initialisiert die Attribute gemäss Input. Für die übrigen wird ein Standard-Wert gesetzt.
  - `~rule_rep()`
    Mit dem Destruktor wird eine Instanz gelöscht.
  - `get_condition_string()`
    Liest ein Bedingung und gibt diese als String zurück.
  - `append_rule()`
    Fügt eine Regel an die Liste der Regeln an inkl. alle abhängigen Objekte.
  - `append_ruleset()`
    Fügt eine Regelmenge an die Liste der Regelmengen an.
  - `delete_rule()` Löscht eine Regel inklusive alle abhängigen Objekte.
  - `delete_rules_in_db()`
    Löscht alle Regeln, die in einer Datenbank definiert sind.
  - `delete_rules_of_user()`
    Löscht alle Regeln eines Benutzes.
  - `delete_rules_of_objtype()`
    Löscht alle Regeln, die zu einem Objekttypen gehören.
  - `delete_ruleset()`
    Löscht eine Regelmenge.
  - `get_ru_list()`
    Liest alle Regelnamen aus der Liste der Regeln.
  - `get_ru()`
    Liest eine komplette Regel.
  - `get_rs_list()`
    Liest eine Liste aller Regelmengennamen aus der Liste der Regelmengen.
  - `get_coupling_list()`
    Liest eine Liste aller Kopplungsmodi.
  - `build_PEP_list()`
    Erzeugt eine Liste von Plätzen primitiver Ereignisse (PEP) als Ausgangspunkt für die Repräsentation der Regeln als Petri-Netze.

- `build_all_rules()`
  Erzeugt für alle Regeln die entsprechenden Petri-Netze.
- `build_rule()`
  Erzeut für eine Regel ein Petri-Netz.
- `append_prim_ac_to_net()`
  Fügt an ein bestehendes Teilnetz eine primitive Aktion an.
- `append_comp_ac_to_net()`
  Fügt an ein bestehendes Netz eine komplexe Aktion an.

Literatur

[AEA95] D. Agrawal and A. El Abbadi. Transaction Management in Database Systems. In A.K. Elmagarmid, editor, *Database Transaction Models*, pages 1 – 31. Morgan Kaufmann, San Mateo, 1995.

[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.

[Blu97] R. Blum. Entwurf und Implementierung einer Benutzerschnittstelle für ALFRED. Projektarbeit, Institut für Informatik und Angewandte Mathematik, Universität Bern, 1997.

[Boo97] G. Booch. *Objektorientierte Analyse und Design (mit praktischen Anwendungsbeispielen)*. Addison-Wesley, 1997.

[Bro87] F. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):11, April 1987.

[CB94] D. Coleman and S. Bodoff. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[Cou85] P. Courtois. On Time and Space Decomposition of Complexe Structures. *Communications of the ACM*, 28(6):596, Juni 1985.

[Day95] U. Dayal. Ten Years of Activity in Active Database Systems. In M. Berndtsson and J. Hansson, editors, *Active and Real-Time database Systems (ARTDB-95)*, pages 3 – 22. Springer, London et al., 1995.

[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.

[DC98] D. F. D'Souza and Wills A. C. *Objects, Components and Frameworks with UML: The Catalysis Appraoch.* Addison-Wesley, 1998.

[Deu94] A. Deutsch. Method and Composite Event in the "REACH" Active Database System. Master's thesis, Technical University Darmstadt, 1994.

[DGG95] K.R. Dittrich, S. Gatziu, and A. Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS. In T. Sellis, editor, *Rules on Database Systems, Lecture Notes in Computer Science 985*, pages 3 – 17. Springer, Berlin et al., 1995.

[Dij79] E. Dijkstra. *Programming Considered as a Human Activity. Classics in Software Engineering.* Yourdon Press, New York, 1979.

[Fri93] H. Fritschi. Entdeckung zusammengesetzter Ereignisse unter Berücksichtigung der Ereignisparameter. Master's thesis, Institut für Informatik, Universität Zürich, 1993.

[Gal86] J. Gall. *Systemantics: How Systems Really Work and How They Fail.* The General Systemantics Press, Ann Arbor, MI, 2. aufl. edition, 1986.

[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.

[Han92a] E.N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proceedings of the ACM SIGMOD International Con-*

ference 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.

[Han96] E.N. Hanson. The Ariel Project. In J. Widom and S. Ceri, editors, *Active Database Systems*, pages 63 – 86. Morgan Kaufmann, San Francisco, 1996.

[KHS94] G. Knolmayer, H. Herbst, and M. Schlesinger. Enforcing Business Rules by the Application of Trigger Concepts. In *Priority Programme Informatics Research, Information Conference Module 1, Secure distributed systems*, pages 28 – 31. SNF, Bern, 1994.

[Lö96] G. Lörincze. Modellierung, Analyse und Simulation von Regeln in der aktiven Schicht ALFRED. Master's thesis, Diplomarbeit, Institut für Informatik und Angewandte Mathematik, Universität Bern, 1996.

[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.

[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.

[Rec85] E. Rechtin. The Art of Systems Architecting. *IEEE Spectrum*, 29(10):66, Oktober 1985.

[Rum91] J. Rumbaugh. *Object-oriented modeling and design.* Prentice Hall, 1991.

[Sch95] M. Schlesinger. Vergleich aktiver Mechanismen in Ingres V6.4, Oracle V7.0 und Sybase V10.0. In H.-J. Scheibl, editor, *Softwareentwicklung - Methoden, Werkzeuge, Erfahrungen '95*, pages 41 – 53. Technische Akademie Esslingen, Esslingen, 1995.

[Sch98] M. Schlesinger. *ALFRED: Konzept und Implementation einer aktiven Regelschicht für Datenbanksysteme*. PhD thesis, Institute of Information Systems, University of Bern, (to be published) 1998.

[Sim82] H. Simon. The Sciences of the Artificial. Technical report, Cambridge, MA: The MIT Press, 1982.

[SM89] S. Shlaer and S.J. Mellor. *Object-Oriented System Analysis: Modeling the World in Data*. Yourdon, 1989.

[SM91] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon, 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.

[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.