
# *Anfragesprache für Workflowmodelle*

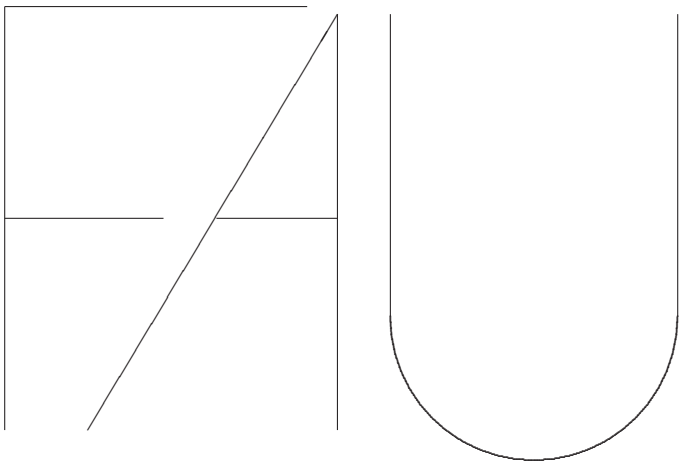
Bachelorarbeit

***Johannes C. Tenschert***

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



# Anfragesprache für Workflowmodelle

Bachelorarbeit im Fach Informatik

vorgelegt von

**Johannes C. Tenschert**

geb. 01.05.1990 in Neumarkt

angefertigt am

**Department Informatik**
**Lehrstuhl für Informatik 6 (Datenmanagement)**
**Friedrich-Alexander-Universität Erlangen-Nürnberg**

Betreuer: Univ.-Prof. Dr.-Ing. habil. Richard Lenz
Dipl.-Inf. Philipp Baumgärtel

Beginn der Arbeit: 30.03.2012
Abgabe der Arbeit: 30.08.2012

# Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30.08.2012

(Johannes C. Tenschert)

# Kurzfassung


## Anfragesprache für Workflowmodelle

Für die Simulation medizinischer Arbeitsabläufe ist eine Vielzahl von Datensätzen nötig, die sehr häufig einzelnen Arbeitsschritten zugeordnet werden können. Es wurden daher zunächst als Modell für Arbeitsabläufe UML-Aktivitätsdiagramme gewählt, die in RDF formuliert, deren Zustände mit Zeitangaben und Kosten und deren Transitionen mit Bedingungen annotiert werden. In dieser Arbeit wird die Workflow Query Language (WQL), eine Anfragesprache für Workflowmodelle, entwickelt. Diese verwendet das Konzept der Fallakten, d.h. über mehrere Abfragen verfügbare und über ihren Namen identifizierbare Sammlungen von Variablen, Zeitangaben und Entscheidungen für Workflows und unterstützt vier verschiedene Anfragearten sowie Operationen zur Verwaltung von Workflows und Fallakten. Die vorgestellte Heuristik zur Pfadverarbeitung der WQL kann dabei mit Verzweigungen, Parallelität und Zyklen umgehen. Abschließend wird die Sprache nach vorher ausgearbeiteten Anforderungen evaluiert.

# Abstract


## Query Language for Workflow Models

A multitude of data is required for simulating medical workflows. More often than not does this data directly belong to single or composite steps. UML activity diagrams have been chosen in order to offer an adequate solution to describe workflows. RDF is used to express workflows and allow annotations of states and transitions with time spans, costs and constraints. This thesis develops Workflow Query Language (WQL) which uses a concept of named case files to gather variables, timings and decisions for branches to use in multiple queries and as empirical values. The language features four different types of queries and operations to manage workflows and case files. A heuristic for processing paths is presented which is able to handle branches, parallelism and cycles. Finally, the language is evaluated against initially outlined criteria.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **API** | Application Programming Interface |
| **CRM** | Customer Relationship Management |
| **DOM** | Document Object Model |
| **DSL** | Domain Specific Language |
| **ERP** | Enterprise Resource Planning |
| **FOAF** | Friends-of-a-Friend |
| **N3** | Notation 3 |
| **OWL** | Web Ontology Language |
| **PEG** | Parsing Expression Grammar |
| **ProHTA** | Prospective Health Technology Assessment |
| **RDF** | Resource Description Framework |
| **RSS** | Really Simple Syndication (*Version 2.0*) |
| **RSS** | RDF Site Summary (*Version 1.0*) |
| **SPARQL** | SPARQL Protocol and RDF Query Language |
| **Turtle** | Terse RDF Triple Language |
| **UML** | Unified Modelling Language |
| **URI** | Uniform Resource Identifier |
| **URL** | Uniform Resource Locator |
| **WDL** | Workflow Definition Language |

**WFMC** WorkFlow Management Coalition

**WFMS** WorkFlow Management System

**WQL** Workflow Query Language

**XML** Extended Markup Language

# 1 Einführung, Zielsetzung und Aufbau

> „We don't know a millionth of one percent about anything."
> (Thomas Alva Edison)

Mit Datenbanken und Abfragesprachen kann vorhandenes Wissen angemessen strukturiert, verwaltet und durch Zusammenführen von Informationen neues Wissen generiert werden. Auch wenn, wie Thomas Alva Edison im Zitat klagt, kein Millionstel Prozent zu irgendwas bekannt sind, so ist durchaus der Versuch erlaubt, diese Barriere zu brechen und das nächste Millionstel Prozent anzustreben.

Workflows bzw. Arbeitsabläufe sind für diesen Versuch ein attraktives Ziel, da ein Plus an Informationen oder auch nur der Überblick über bereits Bekanntes nicht nur theoretisch interessant ist, sondern ganz praktisch auch zu Optimierungen von konkreten Arbeitsabläufen hinsichtlich der Kosten und erfolgreicher Resultate führen kann.

## 1.1 Motivation

Für die Simulation medizinischer Arbeitsabläufe ist eine Vielzahl von Datensätzen nötig, die sehr häufig einzelnen Arbeitsschritten zugeordnet werden können. Diese Situation zeigt sich auch in Arbeitsabläufen anderer Bereiche. So fallen z.B. für Geschäftsprozesse vielfach Informationen über Arbeitszeiten, einzelne Kostenstellen und Bearbeitungs- bzw. Produktionszeiten einzelner oder zusammengefasster Arbeitsschritte an.

Um diese vielen zu Arbeitsschritten zuordenbaren Informationen auch sinnvoll einsetzen zu können, ist ein gutes konzeptionelles Modell zur Strukturierung der Abläufe und zugehöriger Datensätze nötig, auf das mit vorhandenen oder neuen Werkzeugen zugegriffen werden kann.

## 1.2 Ziele

In dieser Arbeit soll eine angemessene Darstellung für Arbeitsabläufe gewählt werden, die mittels RDF (Resource Description Framework) beschrieben in einer Triple-Store-Datenbank abgespeichert werden. Weiterhin soll eine dazugehörige Anfragesprache entwickelt werden, die mit annotierten Arbeitsschritten umgehen und Abfragen, z.B. des Zeitverbrauchs eines Teilabschnitts eines Ablaufs, beantworten kann. Das Modell und die Anfragesprache sollen abschließend anhand von auszuarbeitenden Anforderungen evaluiert werden.

## 1.3 Aufbau der Arbeit

Diese Arbeit ist in 6 Kapitel unterteilt. In Kapitel 1 wird die Ausgangslage erläutert, welche diese Arbeit motiviert, und zusammengefasst, welche Ziele erreicht werden sollen. Kapitel 2 bietet einen Überblick zu einigen Grundlagen, die zum Modellieren und Speichern von Abläufen nötig sind, sowie zu Workflows und Entwicklung von domänenspezifischen Sprachen. In Kapitel 3 wird die entwickelte Anfragesprache Workflow Query Language (WQL), das Modell zur Formulierung von Workflows und das Konzept der Fallakte vorgestellt. Kapitel 4 betrachtet die konkrete Implementierung der WQL näher. Wie gut die Anforderungen an die Sprache und das Modell erfüllt wurden, wird in Kapitel 5 evaluiert. In Kapitel 6 werden abschließend die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

# 2 Grundlagen

In diesem Kapitel sollen zunächst der Standard RDF für die Beschreibung von Ressourcen und die dafür entwickelte Abfragesprache SPARQL vorgestellt werden. Anschließend wird die für diese Arbeit verwendete Definition des Begriffs „Workflow“ erklärt. Abschließend werden Domain Specific Languages, deren Entwicklung und bekannte Beispiele erläutert.

## 2.1 RDF

Ziel des 1999 veröffentlichten Standards **Resource Description Framework** (RDF) [W3C99] ist es, durch ein möglichst simples Datenmodell, beweisbare Rückschlüsse und Support von XML-Schema-Datentypen jedem zu ermöglichen, Aussagen zu jeder Ressource zu machen [W3C04]. Es wird dabei Wert auf eine formal definierte Semantik und Erweiterbarkeit gelegt.

Ergebnis war ein Graphdatenmodell, welches diese Voraussetzungen mit URI-basiertem Vokabular, Datentypen, Literalen, XML-Serialisierung und Formulierung simpler Fakten [W3C04] erfüllt. Dank dieser Eigenschaften wird RDF „oft als grundlegendes Darstellungsformat für die Entwicklung des Semantic Web angesehen“ [HKRS08, S. 35].

Genutzt wird RDF z.B. für die *Web Ontology Language* (OWL), mit der Ontologien mit formal definierter Bedeutung anhand von Klassen, Eigenschaften, Individuals (Instanzen von Klassen) und Datenwerten beschrieben werden [W3C09]. Mit *Friends-of-a-Friend* (FOAF) können Informationen zu Personen (Name, Spitzname, E-Mail-Adressen, Telefonnummer usw.) einheitlich beschrieben und somit leicht verteilt und ausgewertet werden [Dum02]. Der über Blogs, Nachrichtenseiten und auch sonst allgemein weit verbreitete Standard *Really Simple Syndication* (RSS) [RSS09], ehemals *RDF Site Summary* [RSS08], entspringt ebenfalls RDF.

Interessant ist es jedoch nicht, jedes Schema einzeln zu betrachten, sondern ggf. mehrere zu verwenden. Programme können alle ihnen bekannte Modelle verarbeiten und auf für andere Anwendungen relevante Informationen zugreifen. Zum Beispiel kann FOAF auch für Personendaten eigener Programme verwendet werden, wodurch alle Anwendungen, die mit diesem Format umgehen können, auch hierfür einsetzbar sind. Durch das URI-

basierte Vokabular können Ressourcen eindeutig adressiert und somit problemlos aus unterschiedlichen Quellen mit Informationen angereichert werden.

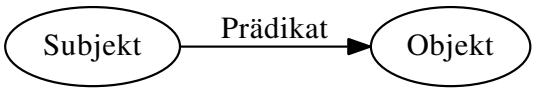



**Abbildung 2.1:** Aufbau von Fakten [W3C04]

Abbildung 2.1 zeigt den Aufbau von Fakten bzw. Tripeln in RDF. Das Subjekt ist die beschriebene Ressource, das Objekt kann entweder eine weitere Ressource oder ein Literal sein und mit dem Prädikat wird die Beziehung des Subjekts zum Objekt ausgedrückt.

Als Beispiel für die Syntax und spätere Abfragen soll ein Ausschnitt der Organisationsstruktur der Friedrich-Alexander-Universität Erlangen-Nürnberg dienen, welche in Abbildung 2.2 dargestellt wird. Die Prädikate `fau:name`, `fau:faculty`, `fau:department` und `fau:students` könnten dabei mit OWL definiert worden sein. URIs können mit vorangehendem Präfix, hier `:` und `fau:`, abgekürzt werden.

Es gibt mehrere Darstellungsformen für RDF-Graphen, so z.B. *Notation 3* (N3), *N-Triples*, *Terse RDF Triple Language* (Turtle) und XML [HKRS08, S. 40f.]. N3 erweitert RDF um Formeln (Literale, die eigentlich selbst Graphen sind), Variablen und logische Folgerungen [W3C11]. *N-Triples* ist eine stark vereinfachte Untermenge von N3 und erlaubt z.B. nur ein Tripel pro Zeile. N-Triples ist darauf optimiert, leicht von Skripten geparst und verglichen werden zu können [W3C11]. Alle Beispiele von RDF-Graphen werden hier stattdessen in Turtle angegeben, was ebenso weniger kompliziert ist als N3, jedoch nicht ganz so minimalistisch wie N-Triples aufgebaut ist. So werden z.B. Abkürzungen für Gruppen von Tripeln und kurze URL-Schreibweisen unterstützt [Bec07]. Fakten werden in Turtle durch „`Subjekt Prädikat Objekt .`“ formuliert. Sollen für das gleiche Subjekt und Prädikat mehrere Objekte angegeben werden, so können diese durch Komma getrennt abgekürzt werden. Ist für mehrere aufeinanderfolgende Fakten ausschließlich das Subjekt gleich, können Prädikate und Objekte durch Semikolon getrennt geschrieben werden. Die Abkürzungen für Fakten und URIs werden in Listing 2.1 verwendet.

Aufgrund der breiten Unterstützung von XML in nahezu allen gebräuchlicheren Programmiersprachen und Frameworks ist es von Anfang an eine wichtige Darstellungsform von RDF gewesen. Dass XML Bäume und RDF Graphen beschreibt ist unproblematisch, denn dafür können Tripel nach Subjekt gruppiert und damit wesentlich platzsparender geschrieben werden [HKRS08, S. 43].

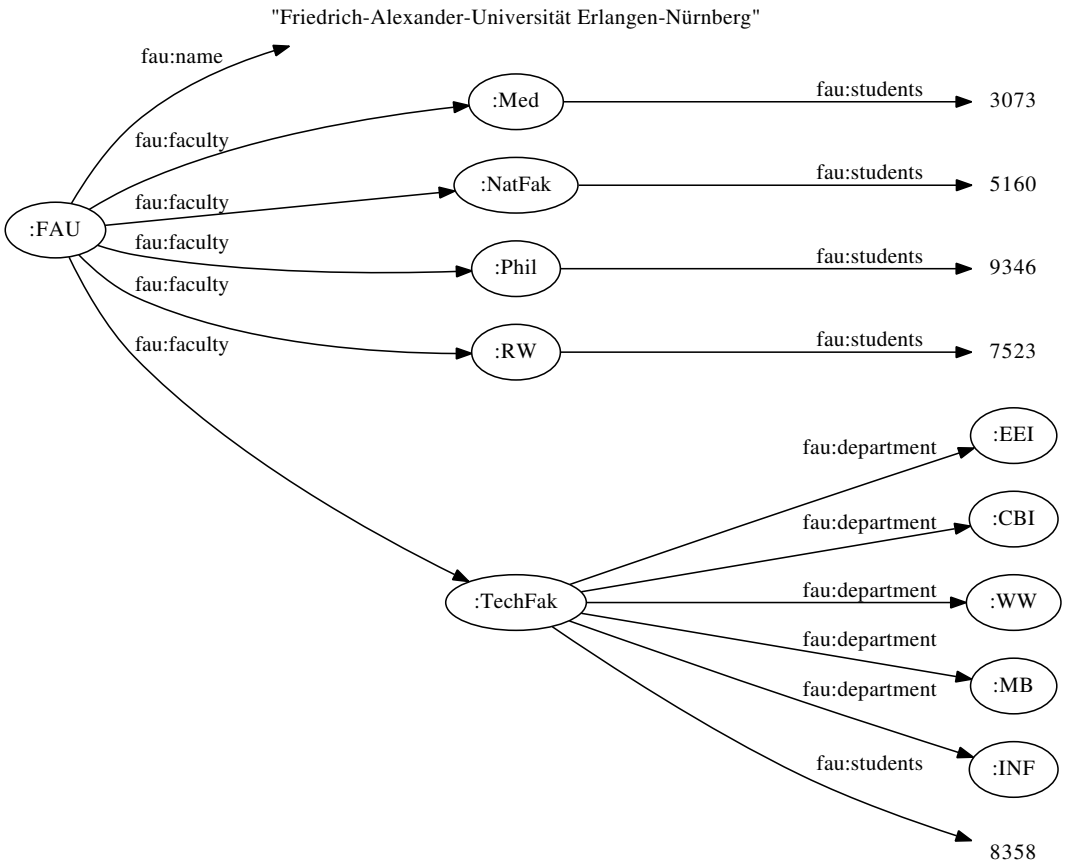



**Abbildung 2.2:** Ausschnitt der Organisationsstruktur der FAU

Interessant wird RDF jedoch erst, wenn verteilte Informationen miteinander verknüpft und durchsucht werden können. Dazu ist eine Abfragesprache wie SPARQL nötig, welche im nächsten Abschnitt näher erläutert wird.

```
@prefix : <http://www.fau.de/info#> .
@prefix fau: <http://www.fau.de/ontology#> .

:FAU        fau:name
                    "Friedrich-Alexander-Universität Erlangen-Nürnberg" ;
            fau:faculty       :Med ,
                              :NatFak ,
                              :Phil ,
                              :RW ,
                              :TechFak .
:Med        fau:students      3073 .
:TechFak    fau:students      8358 ;
            fau:department    :EEI ,
                              :CBI ,
                              :WW ,
                              :MB ,
                              :INF .
...
```

**Listing 2.1:** Teile von Abbildung 2.2 in Turtle

## 2.2 SPARQL

**SPARQL Protocol and RDF Query Language** (SPARQL) ist eine Abfragesprache für RDF [HKRS08, S. 202]. Die Syntax orientiert sich an SQL und Turtle. Folgende grundlegende Abfragearten sind vorgesehen [W3C08]:

- *SELECT*
  Alle oder eine Auswahl gebundener Variablen in einem Mustervergleich von Tripeln zurückgeben. Welche Spalten bzw. Variablen zurückgegeben werden sollen, wird nach dem SELECT-Schlüsselwort angegeben oder mit * pauschal jede aufgeführte Variable verwendet.
- *CONSTRUCT*
  Zurückgegeben wird ein Graph, der aus einem Mustervergleich resultierender gebundener Variablen und angegebener Kanten konstruiert wird.
- *ASK*
  Mit ASK kann abgefragt werden, ob ein angegebenes Muster auf den Graphen zutrifft oder nicht. Das Ergebnis ist ein boolscher Wert.
- *DESCRIBE*
  Gibt einen RDF-Graphen zurück, der gefundene Ressourcen beschreibt.

Dies kann zwar die Verwendung von Abfragen an RDF-Graphen gut vereinheitlichen, wie die gespeicherten Daten jedoch gesammelt und verwaltet werden, wird damit noch nicht erklärt. Das Problem wurde allerdings erkannt und mit *SPARQL 1.1 Update* werden zusätzliche Operationen *INSERT DATA*, *DELETE DATA*, *LOAD* und *CLEAR* definiert [W3C12].

Der grundsätzliche Aufbau einer SELECT-Abfrage wird in Listing 2.2 abgebildet. Eine minimale Abfrage, mit der alle gespeicherten Tripel zurückgegeben werden, wird in Listing 2.3 gezeigt.

```
{   BASE <...> | PREFIX prefixname: <...>   }
    SELECT [DISTINCT] Variablen, Aggregationen, Aliase
{   FROM [NAMED] prefix:name   }
    WHERE {Bedingungen auf (ungruppierte) Tripel}
[   GROUP BY Variablen, Expressions   ]
[   HAVING Auswahl nach Gruppierung   ]
[   ORDER BY Sortierung nach...   ]
[   LIMIT Begrenzen auf bestimmte Anzahl von Elementen   ]
[   OFFSET Ausgabe der Elemente nach angegebener Position   ]
```

**Listing 2.2:** Aufbau einer SELECT-Abfrage [W3C08]

Da eine WHERE-Klausel den Mittelpunkt der meisten SELECT-Abfragen darstellt, lohnt es sich, diese genauer zu betrachten. So werden hier in der Regel Muster der Art `Subjekt Prädikat Objekt` geschrieben, wobei jede einzelne Komponente eines Tripels sowohl eine Variable als auch ein Literal sein kann. Kürzere Schreibweisen für aufeinanderfolgende Tripel mit gleichem Subjekt und ggf. gleichem Prädikat sind ebenso wie in N3 mit gleicher Syntax möglich, d.h. `?x fau:faculty :NatFak, :TechFak ; fau:name ?name .` matcht `?x` auf `:FAU` und `?name` auf deren Namen. Blank Nodes verhalten sich wie Variablen, tauchen jedoch nicht in Ergebnislisten auf, was für `SELECT *` ... durchaus erwünscht sein kann. Mit ihnen können SPARQL-Abfragen etwas übersichtlicher gestaltet werden. So ist es beispielsweise möglich, den Typ eines anonymen Knotens oder ein sonstiges Muster zu dessen Auswahl direkt in diesen Knoten zu schreiben anstatt dies später in der Abfrage und damit ggf. schwerer

```
SELECT ?subject ?predicate ?object
WHERE { ?subject ?predicate ?object . }
```

**Listing 2.3:** Alle Tripel des Default-Graphen abfragen

zuzuordnen zu tun. Als Beispiel dient `[fau:department :INF] fau:department ?dept` zur Auswahl aller „Schwesterdepartments“ des Department Informatik, wobei die jeweilige Fakultät zumindest für diese Abfrage irrelevant sein soll. Auch ein komplett leerer Blank Node `[]` ist möglich und steht für eine anonyme Variable ohne weitere Einschränkungen.

Seit SPARQL 1.1 können auch Property Paths verwendet werden, die sich in der Syntax stark an XPath und regulären Ausdrücken orientieren. Mit ihnen können alternative Pfade, eine bestimmte Anzahl an Vorkommnissen von „Teilstrecken“, Folgen von Beziehungen, inverse Pfade, Negationen und mit diesen Möglichkeiten zusammengesetzte wesentlich kompliziertere Pfade dargestellt werden. Zyklen in Pfaden sind keine Seltenheit und werden daher von Implementierungen sinnvoll behandelt, d. h. die zugehörigen Ergebnisse bei unendlich häufigen Durchläufen werden nur einmal ausgegeben, bei endlichen wie z.B. `(fau:department/^fau:department){1,42}` jedoch alle. XPath erlaubt es, an Knoten in Pfaden weitere Bedingungen zu stellen, Property Paths in SPARQL unterstützen diese leider noch nicht. Dennoch sind Property Paths ein mächtiges Werkzeug mit dem sehr komplexe Anforderungen an Knoten sehr einfach und wesentlich platzsparender und mächtiger als ohne ausgedrückt werden können. Geschrieben werden sie anstatt des „normalen“ Prädikats. Weiterhin wichtig sind optionale Variablen oder Bedingungen, die bei relationalen Datenbanken dem Outer Join entsprechen würden, d. h. die jeweiligen Variablen sind leer, sofern nicht gematcht werden kann. Das ist insbesondere dann sinnvoll, wenn nicht zu allen auszugebenden Ressourcen alle Informationen verfügbar sind. Ein Beispiel dafür wäre eine Abfrage aller bisherigen Filme von Christopher Nolan, deren Ergebnis auch der Erscheinungstermin der DVD enthalten soll. Werden Informationen zur DVD nicht mittels OPTIONAL formuliert, so tauchen neue Filme ohne Angaben zur DVD nicht in den resultierenden Tripeln auf. Mit OPTIONAL darf die Variable für den Erscheinungstermin auch leer sein und alle Werke von Christopher Nolan werden gelistet. Mit FILTER können gebundene Variablen weiter überprüft und Ergebnisse ggf. aussortiert werden, z.B. wie in Listing 2.4, in der nur Fakultäten mit mindestens 8.000 Studenten ermittelt werden.

Auch Aggregatfunktionen spielen eine wichtige Rolle, unterstützt werden `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT` und `SAMPLE`. Gruppiert wird dabei in `GROUP BY` und über `HAVING` können die Ergebnisse nach Gruppierung weiter selektiert werden. Ein einfaches Beispiel für die Benutzung von Aggregatfunktionen ist Listing 2.5. Darüber hinaus zeigt Listing 2.5 auch die Syntax für Aliase in Ergebnissen, die auch komplexere Ausdrücke sein können.

```
PREFIX     : <http://www.fau.de/info#>
PREFIX fau: <http://www.fau.de/ontology#>
SELECT ?x ?students
WHERE {
    :FAU   fau:faculty   ?x .
    ?x     fau:students  ?students .
    FILTER (?students > 8000)
}
```

**Listing 2.4:** Alle Fakultäten der FAU mit mindestens 8.000 Studenten

```
PREFIX     : <http://www.fau.de/info#>
PREFIX fau: <http://www.fau.de/ontology#>
SELECT (SUM(?students) AS ?AnzahlStudenten)
WHERE {
    :FAU   fau:faculty   ?x .
    ?x     fau:students  ?students .
}
```

**Listing 2.5:** Aggregationen: Summe aller Studenten der FAU ermitteln

Neben Aliasen kann jedoch auch im WHERE-Bereich explizit eine Variable über BIND zugewiesen werden, sofern auf Aggegatfunktionen verzichtet wird. Diese zugewiesenen Variablen können auch ohne dauernd wiederholt zu werden in Filterausdrücken, als Ergebnisspalte oder zum Gruppieren/Sortieren weiter verwendet werden.

Ein weiteres neues Feature in SPARQL 1.1 sind Subqueries, deren Ergebnisse vor denen der übergeordneten Abfrage ermittelt werden. Ergebnisvariablen dieser Unterabfrage sind in der übergeordneten ebenfalls verfügbar und können dort weiterverarbeitet oder zum Matchen verwendet werden. Subqueries können eine komplexe Abfrage wesentlich einfacher strukturieren und z.B. Abfragen mit mehreren Zwischenergebnissen erst möglich machen. Mit UNION können Alternativen angegeben werden, wobei dafür sowohl Muster als auch Subqueries verwendet werden können. Ein Beispiel mit UNION und Subqueries ist Listing 2.6. Ein wichtiges Merkmal von RDF ist es, dass Informationen verteilt gespeichert und erzeugt werden können, jeder soll Aussagen zu jeder Ressource machen können.

```
PREFIX     : <http://www.fau.de/info#>
PREFIX fau: <http://www.fau.de/ontology#>
SELECT DISTINCT ?fakultaet ?anzahl
WHERE {
    {
        :FAU          fau:faculty     ?fakultaet .
        ?fakultaet    fau:students    ?anzahl .
        FILTER(?anzahl >= 9000 && ?anzahl <= 10000)
        } UNION {
            SELECT ?fakultaet ?anzahl
            WHERE {
                :FAU          fau:faculty     ?fakultaet .
                ?fakultaet    fau:students    ?anzahl .
                OPTIONAL {?fakultaet fau:department ?dept}
            }
            GROUP BY ?fakultaet ?anzahl
            HAVING (COUNT(?dept) >= 5)
       }}
```

**Listing 2.6:** Ausgabe aller Fakultäten mit 9000-10000 Studenten oder mindestens 5 gespeicherten Departments (Ergebnis: :Phil, :TechFak)

SPARQL bietet zwar eine Fülle weiterer Features, ein Überblick über die wichtigsten sollte hier jedoch gegeben sein. Implementiert wird SPARQL beispielsweise von Fuseki oder OpenLink Virtuoso[1].

1 Auf *http://www.w3.org/wiki/SparqlEndpoints* (zuletzt abgerufen am 29.08.2012) findet man eine umfangreiche Liste von SPARQL-Endpoints, die zum Experimentieren mit Abfragen oder für Aufbereitungen der gespeicherten Daten genutzt werden können.

## 2.3 Workflows

Der Begriff „Workflow“ hat verschiedene Bedeutungen, so beschreibt ihn der Duden als „Abwicklung arbeitsteiliger Vorgänge bzw. Geschäftsprozesse in Unternehmen und Behörden mit dem Ziel größtmöglicher Effizienz“ [Bib12] und die WorkFlow Management Coalition (WFMC) als „system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.“ [Wor99, S. 9]. Dagegen verwendet Van der Aalst den Begriff „Workflow“ als Synonym zum vom WFMC definierten „Business Process“ [AVH04]:

> „A set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships“ [Wor99].

Diese Definition kommt der hier eingesetzten Verwendung am nächsten, wobei besonders Parallelität und Verzweigungen ein besonderer Wert beigemessen wird und zusätzliche Informationen wie Kosten, Zeiten, Wahrscheinlichkeiten zu einzelnen Arbeitsschritten und Transitionen bereitstehen und verarbeitet werden sollen. Ein vereinfachtes Beispiel dafür könnte Abbildung 2.3 mit einem möglichen Ablauf bis zu einer Notoperation nach einem schweren Autounfall sein.

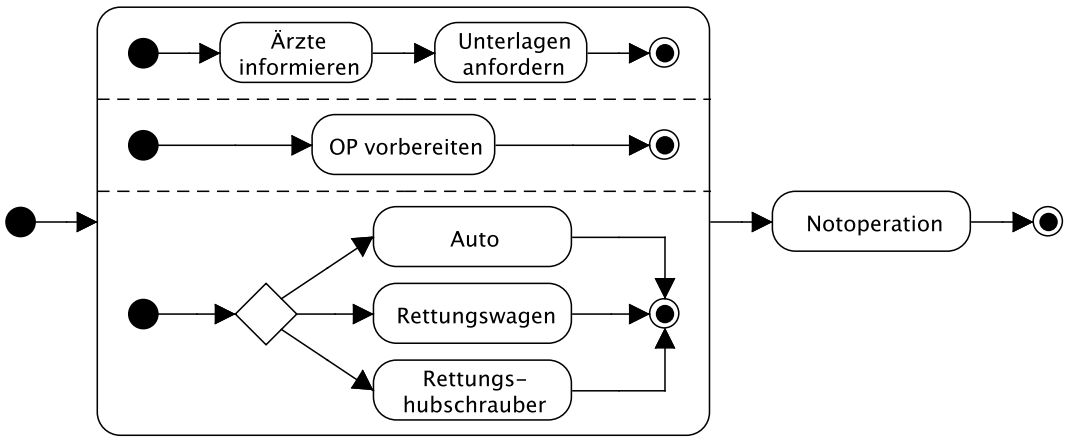



**Abbildung 2.3:** Workflow mit parallelen Vorgängen und Verzweigung

Das Modellieren von Arbeitsabläufen erlaubt es, diese zu vereinheitlichen und nach Analysen idealerweise zu optimieren. Mögliche Notationen dafür sind z.B. BPMN [Awa07]

und UML-Aktivitätsdiagramme [DH01]. Gibt es ein Diagramm eines Workflows, sorgen die Bezeichnungen seiner Arbeitsschritte für ein einheitliches Vokabular, was die Kommunikation erleichtern kann. Informationen zu Schritten können gesammelt und parallele Abläufe oder die Reihenfolge der Abarbeitung können überdacht werden. Existiert ein Modell, kann auch evaluiert werden, wie gut dieses in der Praxis umgesetzt wird und wie weit Annahmen bzgl. Zeitverbrauch und Kosten von der Realität abweichen. Workflow Management Systeme (WFMS) sollen den Nutzer hierbei unterstützen und erlauben ggf. darüber hinaus auch eine Simulation oder initiieren und überwachen eine reale Ausführung spezifizierter Workflows [Gad10, vgl. S. 254].

Begonnen haben WFMS noch ohne eine solche Bezeichnung als in Programmen fest verankerte Prozessspezifikationen, mit denen Arbeitsabläufe vereinfacht werden sollten. Die Entwicklungsphasen werden nach Schulze und Böhm in 4 und erweitert von Gadatsch in 5 Generationen aufgeteilt [Gad10, vgl. S. 256]. So sind typische Vertreter der ersten Generation von WFMS Systeme zur Schadensbearbeitung in Versicherungen. Prozessänderungen hatten Programmänderungen zur Folge. Verbessert wurden diese Systeme in der zweiten Generation durch eine Trennung von Anwendungs-Logik und Prozess-Spezifikation, welche in einer Workflow Definition Language (WDL) verfasst werden musste. Diese Trennung erweitert das Einsatzspektrum und erlaubt einfachere und sich auch verändernde Prozessmodellierung. Eine Domain Specific Language (DSL) - die verwendete WDL ist nichts anderes - kann es auch erlauben, dass Domänenexperten ohne Programmierkenntnisse Prozess-Spezifikationen ändern oder zumindest leichter überprüfen können. In der dritten Generation wurden Datenbank-Managementsysteme statt einfachen Dateisystemen zur Speicherung von Prozessdefinitions- und Prozessausführungsdaten eingeführt. Die vierte Generation wird durch den möglichen Austausch von Prozessmodellen und -ausführungsdaten zwischen Produkten verschiedener Hersteller definiert, in der fünften steht die verteilte Datenhaltung im Mittelpunkt [Gad10].

Process Mining ist ein neuer Ansatz, Workflows anhand von gesammelten realen Protokollen erkennen, analysieren und verbessern zu können [VDA12, vgl. S. 76]. Diese Daten werden z.B. in klassischen WFMS, ERP-, CRM- oder Krankenhausinformationssystemen gesammelt und nennen typischerweise Start, Ende und Kontext bzw. Informationen zu beteiligten Personen und Ressourcen eines Arbeitsschrittes [MSS+09, vgl. S. 426]. Häufige Abläufe können so erkannt und zu einem typischen Schema aggregiert werden. Gegebenenfalls werden auf diesem Weg auch Probleme festgestellt oder mögliche Lösungen ermittelt, anhand derer die untersuchten Workflows abgeändert werden sollten [VDA12, vgl. S. 83]. In vielen Bereichen wurde Process Mining bereits ausprobiert, so auch in

medizinischen Workflows [GPSD10, MSL+08, MSS+09]. Da umfangreiche Mengen von Ereignisdaten z.B. für Abrechnungszwecke ohnehin gesammelt werden, sollte man diese auch zur Verbesserung der Geschäftsprozesse und als Entscheidungshilfe einsetzen.

## 2.4 Domain Specific Languages

Eine **Domain Specific Language** (DSL) ist eine Programmiersprache mit limitierter Ausdruckskraft für ein spezielles Anwendungsgebiet [FP10, vgl. S. 27]. Diese sehr breite Definition ist nötig, da auch der Begriff DSL in der Praxis sehr ausgedehnt eingesetzt wird. Eine DSL muss dabei nicht einmal eine textuelle Programmiersprache sein, sie kann z.B. wie die nach grafischen Mustern in Geschäftsprozessen suchende Anfragesprache BPMN-Q auch eine visuelle Programmiersprache sein [Awa07]. Eine großzügige Auslegung der Definition erlaubt daher viele innovative Sprachkonzepte, die der Kategorie DSL unterzuordnen sind, aber es sollten dennoch gemeinsame Eigenschaften gelten:

Bei einer DSL handelt es sich um eine Programmiersprache. Deren Notation muss nicht weiter eingeschränkt werden, so können neben (eingeschränkten) regulären oder visuellen Programmiersprachen auch Tabellen, Abhängigkeitsnetzwerke [FP10, vgl. S. 495, S. 505] und viele weitere Formen dafür existieren. Wichtig ist dafür nur, dass sie von Menschen gelesen sowie geschrieben und von Computern ausgeführt werden kann [FP10, vgl. S. 27]. Eine DSL sollte allerdings unabhängig von ihrer Art nicht nur eine Häufung von Ausdrücken, sondern strukturiert und als Programmiersprache erkennbar sein [FP10, vgl. S. 27]. Besonders für DSLs, die sich vorwiegend an nicht oder selten programmierende Domänenexperten richten, muss eine Atmosphäre geschaffen werden, in der sich der Nutzer bewusst ist zu programmieren und daher auch möglichst eindeutige Anweisungen geben zu müssen. Da Sprachen jedoch auch mit möglichst geringer Komplexität interpretierbar sein müssen, wird diese Strukturiertheit und Erkennbarkeit als Programmiersprache sehr leicht erreicht, indem auf größeren Aufwand verzichtet wird, die DSL an natürlicher Sprache zu orientieren.

Weiterhin sollte eine DSL auf ein Minimum an Spracheigenschaften reduziert sein, die alle nötigen Funktionen der Domäne, für die sie entwickelt wurde, unterstützen. Eine DSL soll nicht dazu verwendet werden, ein komplettes Softwaresystem zu entwickeln, aber sie soll für einen Teilaspekt dieses Systems eingesetzt werden können [FP10, vgl. S. 28]. Eine DSL sollte sich darüber hinaus einzig und allein auf ihren Schwerpunkt konzentrieren, was ihre Einschränkungen nicht sinnlos erscheinen lässt, sondern zu einem Vorteil macht [FP10, vgl. S. 28.]. Für ihr Gebiet kann syntaktischer und inhaltlicher unnötiger Ballast

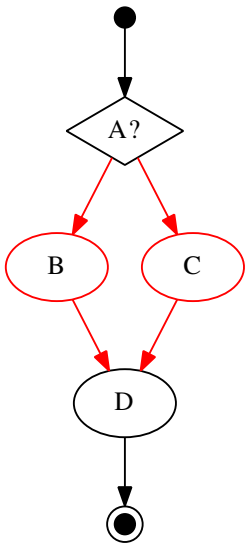
komplett ausgeblendet werden, was die Domäne in den Mittelpunkt stellt und eine Sprache stark genug vereinfachen kann, um auch von Nicht-Programmierern geschrieben oder zumindest gelesen und überprüft werden zu können. Bei unnötigem Ballast kann es sich je nach Einsatzzweck der Sprache um diverse Datentypen, Objektorientierung oder sogar für Programmiersprachen eigentlich ganz grundsätzliche Dinge wie Funktionsaufrufe und -definition handeln. Aber auch die Anbindung an andere Komponenten des Systems kann vor dem Nutzer versteckt werden, so können z. B. häufige immer gleichförmige Datenbankzugriffe der modellierten Domäne implizit über andere Angaben gegeben sein, sie müssen also nicht unbedingt explizit ausgeschrieben werden.

Universalwerkzeuge wie C++ oder Java mögen zwar sehr wohl für nahezu jedes Problem einsetzbar sein, dennoch schreibt man die Erkennung der Komponenten einer E-Mail-Adresse doch lieber in regulären Ausdrücken, einen umfangreicheren Compiliervorgang vorzugsweise in ein Makefile und Datenbankzugriffe möglichst als SQL-Statements, obwohl diese DSLs einen geringeren Funktionsumfang haben. Die in diesem Kapitel vorgestellten Werkzeuge zum Entwickeln von DSLs sind zum Großteil ebenso DSLs bzw. deren Interpreter. Eine weitere sehr bekannte DSL ist DOT, eine Sprache mit der Graphen beschrieben werden können, die von Teilen des Graphviz-Pakets (dot, neato, ...) verarbeitet werden. Dot wird auch in dieser Arbeit für die meisten Grafiken verwendet, so z.B. in Abbildung 2.4.

```
DIGRAPH G {
    init [shape=point, width=0.15];
    "A?" [shape=diamond];
    B [color=red];
    C [color=red];
    final [shape=point, peripheries=2,
           width=0.15];

    init -> "A?";
    "A?" -> B [color=red];
    "A?" -> C [color=red];
    B -> D [color=red];
    C -> D [color=red];
    D -> final;
}
```

**(a)** DOT




**(b)** Ergebnis (Abbildung 3.8)

**Abbildung 2.4:** Graphen in DOT

Aber es müssen nicht nur Werkzeuge sein, die ausschließlich für Informatiker oder Programmierer gedacht sind. Die bereits genannte visuelle Programmiersprache BPMN-Q [Awa07] ist durchaus auch für Domänenexperten der jeweiligen Workflows geeignet. Neurowissenschaftler können ihre Datenbanken über Neuronen und deren Synapsen mit Hilfe von NeuroQL durchsuchen nach Kriterien für Nachbarschaftsverhältnisse der Neuronen, Art der Verbindungen, Projektion der Neuronen zu Nerven, deren Standort und weiteren Merkmalen [TSCJ+06, S. 622f.]. Das vereinfacht die Arbeit der Neurowissenschaftler, indem sie für diesen Zweck komplizierte und umfangreiche SQL-Abfragen nicht selbst schreiben müssen, sondern diese von einem NeuroQL-Interpreter generieren lassen. Abbildung 2.5 vergleicht eine NeuroQL-Abfrage und ihre entsprechende SQL-Abfrage. Details wie die Namen von Tabellen oder Feldern werden damit vor dem Nutzer verborgen und die Schreibweise erlaubt es auch im Umgang mit Datenbanken weniger begabten Personen, mit wenig Aufwand die gesuchten Datensätze zu finden. Die Entwickler von NeuroQL sehen diese Sprache jedoch nur als konkrete Ausprägung einer allgemeineren Sprache DSC-QL, welche für jede Domäne um domänenspezifische Funktionen erweitert werden soll, anstatt eine komplett neue DSL dafür zu entwickeln [TSY07]. Zu diesem Zweck wird das System in zwei Bereiche, das konzeptionelle Datenmodell (DSC-DM) und die konzeptionelle Abfragesprache (DSC-QL) aufgeteilt. Für den Nutzer wichtige domänenspezifische Funktionen, aber auch Implementierungsdetails wie das Mapping des Schemas auf die jeweilige Datenquelle, werden als XML-Dateien angegeben und Datensätze werden mit DSC-ML angelegt, geändert oder gelöscht [She07]. Für viele Anwendungsgebiete mag der Ansatz perfekt sein, um mit dieser Abstraktion eine hohe Zahl an Datenquellen zu unterstützen und nicht für jede Domäne eine eigene Sprache entwickeln zu müssen. Ein wichtiger Vorteil, die Sprache an der jeweiligen Domäne ausrichten zu können, geht dadurch jedoch verloren. Der Ansatz einer konzeptionellen Sprache mit konkreten domänenspezifischen Funktionen kann es dennoch erlauben, einfachere Sprachen für Domänen zu haben, für die der Aufwand der Entwicklung einer eigenen DSL in keinem Verhältnis zu den Kosten steht.

Anhand dieser vielen Beispiele, die man vorher ggf. nicht als DSL wahrgenommen, aber häufig verwendet hat, können sich Gedanken über die Vor- und Nachteile der Entwicklung solcher Sprachen gemacht werden.

Häufig wird dazu angeführt, dass DSLs auch von Domänenexperten anstatt Programmierern verwendet werden könnten. Martin Fowler nennt dieses Argument die „COBOL-Täuschung“ [FP10], da hier auch zunächst angenommen wurde, dass Betriebswirte selbst kaufmännische Anwendungen entwickeln würden. SEQUEL, später als SQL

```
(neuron)[hasMolecules('5HT'), chemicalSynapse(s,'post'),
    s.itsFiringPatterns sfp, sfp.name= 'Irregular with Burst']
```

**(a)** NeuroQL

```
SELECT  n.neuronid, n.name, n.synonym
FROM    nb_neuron n, nb_molecule m,
        nb_hasmolecule hm
WHERE   n.neuronid = hm.neuronid AND
        m.moleculeid = hm.moleculeid AND
        m.name = '5HT' AND n.neuronid IN
           (SELECT  c_syn.postsyncell
            FROM    nb_neuron s, nb_csynapse c_syn
            WHERE   s.neuronid = c_syn.presyncell AND
                    s.neuronid IN
                       (SELECT  fp.neuroid
                        FROM    nb_firingpattern fp
                        WHERE   fp.name = 'Irregular with Burst'))
```

**(b)** SQL

**Abbildung 2.5:** Vergleich NeuroQL mit entsprechender SQL-Abfrage [TSCJ+06, S. 622f.]

bekannt, ist zwar auch für die weniger intensiven Datenbanknutzer gedacht, deren Autoren sind aber soweit realistisch, Kenntnisse in diesem Bereich zu fordern [CB74]. Aber dennoch sollte dieser Punkt als Vorteil angesehen werden, weil die Kommunikation mit Domänenexperten erheblich verbessert werden kann. Diese werden in einer an der Domäne orientierten Sprache nicht von Implementierungsdetails und besonders der „kryptischen“ Syntax einer General Purpose Programmiersprache abgeschreckt, sondern sehen nur, strukturiert, für die Domäne relevante formulierte Regeln, Zusammenhänge und Abläufe. Eine DSL kann die semantische Lücke der Kommunikation von Domänenexperten und Entwicklern dabei besonders durch ein einheitliches Vokabular schließen [Gho11], was dazu führen kann, dass Fehler in Formulierungen von Domänenexperten erkannt werden und sie diese gegebenenfalls sogar ausbessern. Dass jedoch auch von Leuten mit geringeren Programmierkenntnissen plötzlich Anwendungen komplett neu entwickelt oder auch nur erweitert werden sollen, scheint mit Blick auf die Vergangenheit keine realistische Annahme zu sein.

Unabhängig von der Kommunikation können aber auch weitere Fehler in mit DSLs formulierten Programmen oder Teilaspekten eines Systems vermieden werden, denn im Gegensatz zu den sonst üblicherweise verwendeten Programmiersprachen können in einer DSL Redundanzen verringert werden und Plausibilitätschecks der Eingaben sind im jeweiligen Interpreter oder Compiler einfacher möglich, da sich die Sprache an der Domäne orientiert und nicht deren Modell und Implementierung an der Sprache. Abbildung 2.5 kann hier für beide Argumente dienen, denn ein Vertipper in den zahlreichen wiederkehrenden Details der SQL-Abfrage ist nicht unwahrscheinlich, ebenso kann ohne weiteres auch eine „n.neuronid“ im Eifer des Gefechts mit einer „m.moleculeid“ verwechselt werden. Sind Details wie die erlaubten domänenspezifische Funktionen und Wertebereiche schon beim Parsen der Sprache bekannt, sind viele fehlerhafte Abläufe ausgeschlossen, die sonst von der jeweiligen Umgebung ausgeführt worden wären. Auch wird man in einer DSL gezwungen, nur die dafür definierten Spracheigenschaften zu nutzen, während man in einer General Purpose Programmierspache zwar die Option hat, sich an Vorgaben zu halten, diese aber nicht durchgesetzt werden können.

Ein weiterer Vorteil von DSLs ist, dass deren Einsatz skaliert werden kann [Gho11]. Eine einmal gelungene Kombination der Sprache und Implementierung kann danach massenhaft verwendet werden, ohne dass jeder Nutzer laufend das Rad neu erfinden und auch nicht alle Zusammenhänge und Vorgehensweisen verstehen muss. So ist nicht anzunehmen, dass jeder Programmierer für seine Projekte effiziente Lösungen zu ihrer Kompilierung und Installation findet. Formuliert er jedoch ein Makefile dafür, muss er sich nicht selbst darum kümmern, dass nicht immer alles neu übersetzt wird, sondern nur geänderte Bereiche. Auch ist es eher unwahrscheinlich, dass der typische Datenbanknutzer für jede seiner Abfragen einen zumindest guten Weg zum Ergebnis kennt und dabei alle Feinheiten des aktuellen Datenbestandes beachtet. Selbst wenn er dazu qualifiziert ist, will er dies ggf. nicht dauernd wiederholen. Aber Abfragen abstrakt und ohne Implementierungsdetails formulieren zu können ist mit SQL auch mit geringeren Kenntnissen oder Motivation möglich.

Weiterhin haben DSLs auch den Vorteil, von anderen Systemen ebenso implementiert werden zu können. Eine Motivation von NeuroQL war es, eine von System zu System unterschiedliche und sowohl in der Anzahl ihrer Typen als auch in ihrer ausdrucksweise arg eingeschränkte GUI-Abfrage zu vermeiden [TSCJ+06]. Vorgefertige Abfragen für die häufigsten Probleme können natürlich weiterhin angeboten werden und vielen Nutzern ihre Arbeit erleichtern, besonders für eine Übertragung der Kenntnisse und Automatisierung ist eine einheitliche DSL allerdings sicher vorteilhaft. Dies sind nicht alle Vorteile

von DSLs, aber vor der Entwicklung einer solchen sollten auch mögliche Nachteile in Betracht gezogen werden.

Die Entwicklung einer DSL mag zwar im Verhältnis zur Entwicklung der darunterliegenden Bibliothek ein nur geringer zusätzlicher Aufwand sein, aber dennoch ist sie ein einzukalkulierender Aufwand [FP10, vgl. S. 37]. Besonders bei der Umsetzung neuer Konzepte ist es schwierig, auf bisherige Hilfsmittel und Codegeneratoren zurückzugreifen, da diese i.d.R. nur existieren, wenn Ähnliches schon versucht wurde. Aber auch für reguläre textuelle DSLs müssen sich deren Entwickler in die entsprechenden Hilfsmittel und Compilerbau einarbeiten. Die große Auswahl verfügbarer Werkzeuge, die Abschätzung ihrer jeweiligen Vor- und Nachteile aber auch umständliches Debuggen können dabei abschreckender wirken und den Aufwand höher erscheinen lassen, als er tatsächlich ist. Umfangreiche Einarbeitungszeit sollte man dennoch einplanen. Wurde jedoch einmal der Aufwand der Einarbeitungsphase betrieben, können die erworbenen Kenntnisse auf viele weitere neue Sprachen in anderen Teilsystemen oder Projekten, sofern nach anderen Kriterien sinnvoll, angewendet werden. Was definitiv für die Kosten aller folgenden DSLs übrig bleibt, ist, den dazugehörigen Code zu schreiben und auch in den Folgejahren zu pflegen [FP10, vgl. S. 37f.].

Weiterhin ist eine domänenspezifische Sprache dafür da, den Nutzern ihre Arbeit zu erleichtern, eine neue oder vielleicht sogar erste Programmiersprache kann hier nach viel Arbeit aussehen und abschrecken. Dabei sollte jedoch zunächst überlegt werden, wie viel mehr mit einer DSL gelernt werden muss gegenüber der Alternative, auf eine solche zu verzichten, denn die hinter einem System stehenden Konzepte sollten in beiden Szenarien hinreichend verstanden werden [FP10, vgl.37]. Mit einer neuen Sprache hat man allerdings die Möglichkeit, Unwichtiges auszublenden und den zusätzlichen Aufwand zum Erlernen der Syntax mit einem einfacher zu verstehenden Modell aufzuwiegen. Ein unangenehmerer Aspekt aus Sicht der professionelleren Nutzer ist dagegen, dass Texteditoren Syntax Highlighting i.d.R. nur für eine geringe Zahl an Sprachen anbieten und die dazu nötigen Definitionen für eine neue Sprache fehlt. Viele Editoren lassen zwar zusätzliche Sprachdefinitionen zu, es kostet aber dennoch Zeit, diese zu schreiben und man wird immer unzufriedene Nutzer finden, deren Lieblingseditor keine eigene Definition für die Sprache bekommen hat.

Ein letzter wichtiger Nachteil, von Martin Fowler als „Ghetto Language“ [FP10, S. 38] bezeichnet, ist die häufig geringe und auf ein oder wenige Projekte beschränkte Verteilung einer DSL. Für Leute, die eine solche wenig verwendete Sprache gelernt haben, sind die erworbenen Kenntnisse in anderen Projekten und Unternehmen wertlos. Es

ist jedoch auch für Betriebe unmöglich, Programmierer oder Domänenexperten mit Kenntnissen zu einer In-House-Sprache zu finden, zusätzlichen Mitarbeitern müssen die nötigen Kenntnisse erst vermittelt werden. Literatur zum Erlernen dieser Sprachen ist nur vorhnden, sofern sie im Rahmen der entsprechenden Projekte verfasst wurde. Dass sich wenige Personen mit Kenntnissen zu einer selten verwendeten Sprache in einem kritischen System unentbehrlich machen können, darf dabei auch nicht unterschlagen werden.

Steuern kann man die Ausprägung der Vor- und Nachteile bereits bei der ganz grundsätzlichen Entscheidung, ob eine externe oder interne DSL entwickelt werden soll. Eine interne DSL ist dabei eine einer Sprache nachempfundene API des gewünschten Teilsystems [FP10, vgl. S. 67], die in einer bereits vorhandenen Programmiersprache - egal welcher Art - verwendet wird. Mögliche Hostsprachen können dabei anhand der Eignung ihrer Syntax für Formulierungen der jeweiligen Domäne oder pragmatischer nach vorhandenen APIs des überwiegenden Teils der bisherigen Implementierung gewählt werden. Ein bekanntes Beispiel für eine interne DSL ist jQuery, zumindest die Art der Benutzung dieser Javascript-Bibliothek. jQuery wird auf der Clientseite eingesetzt für die Navigation im Document Object Model (DOM) eines HTML-Dokuments mit einfachen Selektoren, zum Datenzugriff, für Events und in vielen Browsern darstellbare UI-Elemente sowie weitere nützliche Funktionen[1]. Eine API kann man z.B. durch Method Chaining einer Sprache nachempfinden, also durch Verkettung von Methodenaufrufen, wobei eine Methode immer ein Ergebnis - im Zweifelsfall das Objekt, auf dem die Methode aufgerufen wurde - zurückliefern sollte, auf das weitere Methoden angewendet werden können [FP10, S. 68]. Besonderheiten einer Hostsprache, wie z.B. $\lambda$-Ausdrücke oder Annotationen, können auch für eine DSL eine wichtige Rolle spielen, sollten jedoch nur eingesetzt werden, wenn eine Portierung auf eine andere Hostsprache ohne solche Eigenschaften unwahrscheinlich ist oder diese auf andere Weise akzeptabel nachgebildet werden könnten. Dass jedoch die Eigenschaften der Hostsprache teilweise oder ganz übernommen werden können, ohne sich unbedingt mit Lexern oder Parsern auseinandersetzen zu müssen, kann die Entscheidung pro interne DSL wesentlich erleichtern. Der Schritt von einer API zu einer internen DSL ist kein großer und zu einem späteren Zeitpunkt kann immer noch eine externe DSL auf dem schon vorhandenen Modell aufgebaut werden. Die einfachere Entwicklung hat natürlich auf der anderen Seite auch Einschränkungen zur Folge. So kann die Syntax der Hostsprache für mehr Unübersichtlichkeit sorgen durch

1 *http://api.jquery.com* (zuletzt abgerufen am 29.08.2012)

zusätzliche Funktionsaufrufe, Escape-Zeichen und andere Merkmale wie Anführungszeichen, Semikolons oder Klammern. Nutzern sollte die Hostsprache daher nicht fremd sein, denn viele Spracheigenschaften erscheinen sonst ggf. unlogisch. Aber eine Hostsprache kann nicht nur einengen, zur selben Zeit liegen ihre Grenzen nämlich einzig und allein in ggf. ausgemachten Regeln des Programmierstils. Da diese Regeln jedoch nur schwer bis gar nicht erzwungen werden können, sind sie nichts anderes als leicht ignorierbare Handlungsempfehlungen. Zu viel Funktionalität kann vom Nutzer in der Hostsprache geschrieben sein, anstatt eine passende Funktion der API/DSL dazu anzubieten. Das Rad wird ggf. häufiger neu erfunden. Sollen Implementierungsdetails geändert werden, so muss dies u.U. an sehr vielen Stellen passieren.

Mit einer externen DSL wird auf die Hilfsmittel der Hostsprache verzichtet und eine eigene Sprache mit geeigneter Grammatik und gegebenenfalls völlig neuen Eingabekonzepten definiert. Die eingangs aufgeführten Beispiele waren ausschließlich externe DSLs und werden hier nicht erneut vorgestellt. Neben einem semantischen Modell steht bei externen DSLs auch das Parsen der Sprache und ggf. zusätzliche Einarbeitungszeit in die dazu nötigen Hilfsmittel im Vordergrund. Die Syntax kann sich komplett an der Domäne orientieren, mangels Hostsprache müssen keine unnötigen oder sogar verwirrenden Spracheigenschaften übernommen werden [FP10, vgl. S. 89]. Eine solche Sprache zwingt deren Nutzer dazu, ausschließlich in der Sprache definierte oder erlaubte Konstrukte zu benutzen - ein wilder Mix aus eigener Implementierung und DSL-API wird damit glücklicherweise erschwert. Da Parser und Interpreter auch für andere Systeme entwickelt werden können, sind ohne den zusätzlichen Ballast des Einlesens und Verarbeitens einer Hostsprache in einer externen DSL verfasste Programme und Ausdrücke grundsätzlich portabler. Reguläre Ausdrücke z.B. sehen in den meisten Implementierungen ähnlich genug aus, so dass ein Anpassen bei Verwendung in einer anderen Umgebung selten nötig ist.

Entscheidet man sich für eine externe DSL, sollte man die Anforderung, dass die Sprache sich an der Domäne *orientiert*, genauer betrachten. Die schönste für eine Domäne perfekt passende Grammatik bringt nicht viel, wenn der Aufwand, den dazugehörigen Parser zu schreiben oder zu generieren, in keinem Verhältnis zum Nutzen der Sprache steht. Und selbst wenn die Entwicklung des Parsers kein Problem sein sollte, die Laufzeit- oder Speicherkomplexität beim Einlesen kann ein Problem sein. Es ist daher sinnvoll, die Sprache ggf. mit zusätzlichen Zeichen oder Regeln zu versehen, die es ermöglichen, Lexer und Parser mit bekannter und sinnvoller Komplexität erzeugen zu können. Häufig wird wie in Abbildung 2.6 eingelesen, v. a. weil es sehr mächtige Werkzeuge aus dem

Compilerbau gibt, Lexer und Parser zu generieren. Ein Lexer zerlegt die Eingabe in Token und kann für den Parser Unwichtiges wie Kommentare und Whitespaces komplett entfernen. Der Parser erhält die resultierenden Token und generiert daraus i. d. R. einen (attributierten) Strukturbaum. Was danach anhand dieses Baumes ausgeführt werden soll, ist leider nicht oder zumindest nicht im Regelfall mit vorhandenen Werkzeugen generierbar, z.B. könnte die DSL direkt interpretiert oder zunächst in eine andere Sprache übersetzt werden, weshalb auch die Grafik an diesem Punkt einen Platzhalter hat.

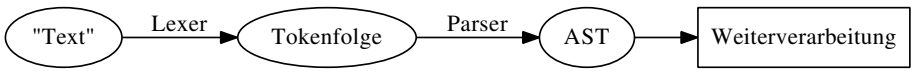



**Abbildung 2.6:** Typisches Einlesen einer externen DSL

Typische Parsergeneratoren erzeugen Top-Down- bzw. Bottom-Up-Parser. Grammatiken für Bottom-Up-Parser werden häufig als komplizierter wahrgenommen, Top-Down-Parser arbeiten leichter verständlich nach dem Prinzip des rekursiven Abstiegs [FP10, vgl. S. 99]. Bekannt sind für diese `yacc(1)` bzw. ANTLR[1] und viele weitere, welche Generatoren dabei für das eigene System relevant sind hängt vom groben Aufbau der eigenen DSL ab, die ggf. für den Parser angepasst werden muss. Eine Einschränkung von Top-Down-Parsern ist, dass Linksrekursionen in der Grammatik zu Endlosschleifen beim Einlesen führen können [FP10, vgl. S. 99]. Will man z.B. in arithmetischen Ausdrücken unnötige Klammern weglassen, so können die Präzedenzregeln nicht bereits in einem Top-Down-Parser, sondern erst zwingend danach, angewendet werden.

Eine *Parsing Expression Grammar* (PEG) kann als formale Definition eines Top-Down-Parsers gesehen werden, alle PEG-Sprachen können in linearer Laufzeit geparst werden [For04]. Der größte Unterschied zu „normalen“ Top-Down-Parsergeneratoren ist, dass anstatt ungeordneter Alternativen, wie sie z. B. in Backus-Naur-Form durch das „|“-Zeichen ausgedrückt werden, nur Alternativen mit fester Versuchsreihenfolge im rekursiven Abstieg angegeben werden [For04]. Durch diese feste Reihenfolge sind eindeutige Interpretationen für mehrdeutige Grammatiken möglich. Eine Implementierung für Python - pyPEG[2] - erlaubt die Formulierung der Grammatik in Funktionen als Nichtterminale, deren Funktionsname auch die Bezeichnung im Strukturbaum ist, und

1 *http://www.antlr.org* (zuletzt abgerufen am 29.08.2012)
2 *http://fdik.org/pyPEG/* (zuletzt abgerufen am 29.08.2012)

regulären Ausdrücken und Literalen als Terminale. Übergeben an den Generator wird nur die Funktion, die das Wurzelelement der Grammatik darstellen soll.

In Zukunft können Language Workbenches vielleicht das Erstellen von Sprachen und hilfreichen Werkzeugen für Benutzer vereinfachen. Noch beschreibt der Begriff ungenau, welche Art von Programmen damit zusammengefasst werden sollte, gemein sollten sie jedoch haben, dass die Entwicklung einer DSL mit Hilfe einer Language Workbench nicht nur auf reinem Text, ggf. noch mit Syntax Highlighting in einem Editor, beschränkt werden sollte. Stattdessen sollte der Nutzer an einer abstrakten Repräsentation des Modells arbeiten, also z.B. wie in bekannten integrierten Entwicklungsumgebungen eine intelligente, Objekttypen und Inhalte beachtende Auto-Vervollständigung und ggf. mehrere Typen von Editoren (grafisch, textuell, Eigenschafts-Fenster, ...) für den Quelltext verwenden können [Fow05]. Für eine selten benutzte DSL so viel Aufwand zu betreiben, würde unverhältnismäßig hohe Kosten verursachen. Die Anbindung an oder Generierung von Entwicklungsumgebungen für viele Sprachen kann die Relevanz von DSLs für die Softwareentwicklung jedoch stark erhöhen.

# 3 Workflow Query Language (WQL)

In diesem Kapitel wird die für die Arbeit entwickelte Workflow Query Language (WQL) vorgestellt. Zunächst werden dafür alle Anforderungen an die Sprache, das Modell und die Implementierung ausgearbeitet. Danach wird anhand der Anforderungen versucht, die WQL in Kategorien von Anfragesprachen von Workflowmodellen einzuordnen, denen auch verwandte Arbeiten zugeordnet sind. Anschließend wird das Modell für die Darstellung von Workflows erläutert und das Konzept der Fallakte eingeführt. Im nächsten Abschnitt wird erklärt, wie Pfade mit Verzweigungen, Zyklen und Parallelität behandelt werden und daraufhin wird die Syntax von Bedingungen von Transitionen beschrieben. Mit diesen Voraussetzungen kann im Anschluss die Syntax und Semantik der WQL vorgestellt werden. Abschließend wird die Verwendung der WQL anhand von Beispielen gezeigt, die das Modellieren und Verwalten von Workflows, den Einsatz von Fallakten und die verschiedenen Anfragearten behandeln.

## 3.1 Anforderungen

Die Anforderungen für die in dieser Arbeit entwickelte WQL wurden in der Entwurfsphase unter anderem auch in einem Interview mit einem Simulationsexperten ausgearbeitet. Um in der Evaluation alle Punkte erneut betrachten und bewerten zu können, wurden diese hier nummeriert aufgeführt:

1. **Zyklen**
   Zyklen sind in realen Workflows keine Seltenheit. Es ist durchaus möglich, eine Aktion oder eine Folge von Schritten so lange wiederholen zu müssen, bis ein Kriterium erfüllt ist. Ein Beispiel dafür wäre Wiederbelebungsversuche durchzuführen, bis sie erfolgreich sind oder es nicht mehr werden können.
   Die Anforderung hierbei ist, dass die Verarbeitung einer Abfrage mit Zyklen in Workflows möglich sein muss und keine Endlosschleifen verursacht.

2. **Parallelität**
   Auch Parallelität ist für Workflows wichtig, denn Arbeitsschritte müssen nicht unbedingt sequenziell erfolgen. Parallele Abläufe sind dabei keine vernachlässigbaren Ausnahmefälle und müssen sowohl modelliert als auch sinnvoll abgefragt werden können.
   Unterstützen muss die WQL zwei Arten von Parallelität:
   - Sofortiger Abbruch der Parallelität, sobald der erste Faden sein Ziel erreicht.
   - Alle fertigen Fäden warten bis der komplette parallele Vorgang abgeschlossen wurde.

   Vorstellbar ist es auch, eine gewisse Anzahl fertiger Fäden zu fordern für einen Abbruch des Vorgangs, die Implementierung soll jedoch zunächst auf diese zwei Arten beschränkt werden.

3. **Skalierungsstufen**
   Zeitangaben und Kosten sollen in verschiedenen Skalierungsstufen angegeben werden können, z.B. die komplette Zeit einer Rettung mit Hubschrauber anstatt vieler einzelner Angaben und falls gewünscht dennoch Details.
   Angaben zu Zeiten in aggregierten übergeordneten Arbeitsschritten sollen im Modell und in einer Abfrage möglich sein. In den Kapiteln 3.3 und 3.4 werden diese Skalierungsstufen noch auf Kostenarten ausgeweitet.

4. **Datenqualität**
   Die Qualität der Ergebnisse hängt zwangsläufig stark mit der Qualität der bereitgestellten Informationen zusammen. Es sollen daher eine oder mehrere Kennzahlen zur Verfügung gestellt werden, um die Vollständigkeit der Eingabedaten beurteilen zu können.

5. **Entscheidungen an Verzweigungen**
   Ein Workflow ohne Verzweigungen wäre eine Sequenz von Arbeitsschritten, ggf. mit parallelen Teilbereichen, die jedoch ebenso sequenziell verlaufen. Es ist zwar durchaus erlaubt, ein solches Workflow-Modell zu formulieren, Abfragen an dieses wären jedoch gänzlich uninteressant. Mit Verzweigungen ergeben sich ggf. sehr viele mögliche Pfade, es soll aber möglich sein, bekannte Entscheidungen für eine konkrete Instanz anzugeben und nur Ergebnisse zu erhalten, die sich auch an diese Wahl halten.
   Für den Krankentransport z.B. ist die Wahl eines Verkehrsmittels aus mehreren Alternativen eine Entscheidung, die individuell für jede Instanz des Workflows

getroffen werden muss und ggf. ausserhalb des Abfragesystems gefällt wird, aber für eine Abfrage angegeben werden sollte. Da die erste Anforderung auch eine sinnvolle Verarbeitung bei Zyklen voraussetzt, sollten Entscheidungen für den gleichen Knoten auch unterschiedlich getroffen werden können in der gleichen Instanz, was für das Beenden eines Zyklus wichtig ist.
Pfade müssen für Aggregationen gewichtet werden können, wobei ihre Wahrscheinlichkeiten als Gewichtungsfaktor eingesetzt werden sollten. Um diese Wahrscheinlichkeiten möglichst exakt ermitteln zu können, müssen unterschiedliche Wahrscheinlichkeiten für von Arbeitsschritten ausgehende Transitionen möglich sein.

6. **Erweiterbarkeit der Modelle / Datentypen**
   Nicht nur exakte Werte für Zeiten oder Kosten sind interessant und als einzige Datenquelle muss nicht unbedingt eine SPARQL-Datenbank dienen. Histogramme, z.B. für Zeitangaben oder Kosten, wurden zwar als möglichen und sinnvollen Datentypen vorgesehen, um den Rahmen der Arbeit nicht zu sprengen allerdings zurückgestellt. Eine Implementierung könnte zukünftig mit den in [DLBMW10] aufgeführten Aggregatfunktionen erfolgen, es muss daher unbedingt möglich sein, auch später neue Datentypen implementieren zu können. Intervalle wären ebenso ein leicht implementierbarer Datentyp, der später hinzugefügt werden könnte. Auch ist es unter Umständen sinnvoll, Angaben in mehreren verschiedenen Datentypen zu machen und während der Verarbeitung zu entscheiden, welche der Informationen verwendet werden. Zusammen nicht kompatible Zeit- oder Kostenangaben mehrerer Knoten könnten so z.B. auf einen kleinsten gemeinsamen Nenner gebracht und mit geringerer Genauigkeit dennoch verarbeitet werden. Datenquelle für Informationen könnten andere (relationale) Datenbanken oder auch ein mehrdimensionaler Datenwürfel eines Data Warehouse sein. Diese Informationsquellen später hinzufügen zu können soll möglich sein, ohne große Teile der WQL neu implementieren zu müssen.

7. **Erweiterbarkeit der Sprache**
   Mit der Erweiterbarkeit der Datentypen und -quellen ist auch eine Erweiterbarkeit der Sprache sinnvoll. Die Sprache und Implementierung soll so aufgebaut sein, dass nachträglich hinzugefügte Datentypen zwar ggf. zusätzliche Funktionen zum Parsen und zur Ausgabe benötigen, aber außer diesen Punkten keine allzu großen Änderungen der Implementierung nötig sind.
   Neben Datentypen soll eine Erweiterung der angebotenen Funktionen, wie z.B.

neue Renderer von Pfaden oder Konkatenation von Werten oder Ausdrücken, möglich sein. Auch die in Kapitel 3.6 eingeführten Ausdrücke für Bedingungen von Transitionen bieten Spielraum für weitere sinnvolle Funktionen und sollten leicht erweiterbar sein.

8. **Anfragearten**
   Es wurden vier unterschiedliche Arten von Anfragen an einen Workflow ausgearbeitet, welche alle unterstützt werden müssen:

   a) Der Nutzer möchte komplett aggregierte Zeiten oder Kosten für eine Abfrage erhalten um z.B. die geschätzte Gesamtzeit eines Workflows oder Abschnitts zu ermitteln.

   b) Der Nutzer möchte Zeiten und Kosten für alle Zustände erhalten um z.B. herauszufinden, welcher Teil einer Behandlung voraussichtlich die meiste Zeit beansprucht oder die größten Kosten verursacht.

   c) Der Nutzer möchte den wahrscheinlichsten Ablauf anstatt gewichteter Schätzwerte über alle Möglichkeiten haben. Geschätzte Zeiten und Kosten können je nach modelliertem Workflow ggf. näher an der Realität liegen und der wahrscheinlichste Ablauf von Arbeitsschritten kann auch ohne geforderte Schätzwerte interessant sein.

   d) Der Nutzer möchte die wahrscheinlichsten Abläufe sowie deren Kosten und Zeiten ermitteln, weil er ggf. herausfinden möchte, wie stark sich deren Ergebnisse unterscheiden oder auch nur, welches die wahrscheinlichsten Pfade sind und wie diese gewichtet werden.

9. **Einfüge-, Änderungs- und Löschoperationen**
   In Kapitel 3.4 werden Fallakten eingeführt, welche Variablen, Zeiten und Entscheidungen für Instanzen von Workflows festlegen und damit als dynamischer Teil eines Workflows gesehen werden können. Für diese Fallakten sollten Einfüge-, Änderungs- und Löschoperationen angeboten werden, um deren Aussagen nicht in jeder Abfrage zu Workflows erneut formulieren zu müssen. Darüber hinaus könnten auch Operationen bereitgestellt werden, mit denen ohne direkten Zugriff auf die SPARQL-Datenbank Workflows erstellt, geändert oder gelöscht werden können.

10. **Laufzeit**
    Konkrete Anforderungen an die Laufzeit zu stellen ist schwierig, da diese sehr stark vom tatsächlich abgefragten Workflow abhängt. Was jedoch gefordert werden

kann, ist, für die wichtigsten Abfragen in sinnvollen Workflows auch tolerable Antwortzeiten zu ermöglichen. In Kapitel 3.5 werden Beispiele für eine unendliche Zahl an Pfaden oder ein exponentielles Wachstum der Möglichkeiten mit wachsender Größe eines Modells gegeben. Für tolerable Antwortzeiten gilt daher, dass es keine Endlosschleife geben darf bei der Ermittlung der Ergebnisse und dass eine vernünftige Zahl an relevanten und sehr wahrscheinlichen Pfaden betrachtet wird, d. h. nicht unbedingt alle Pfade verarbeitet werden.

## 3.2 Verwandte Arbeiten

Anfragesprachen von Workflowmodellen können in drei Arten aufgeteilt werden [Awa07]:

1. *Abfragen auf die Workflowdefinition.* In einer solchen Abfrage sollen Informationen über den Aufbau von Workflows ermittelt werden, d. h. über Muster von Aktivitäten und deren Abfolgen und Beziehungen zueinander.
2. *Abfragen zu laufenden Instanzen von Workflows.* Der Status einer laufenden Instanz soll ermittelt und der Weg zu diesem Status verfolgt werden. Daneben können auch zusätzliche Informationen dieses Weges ausgewertet und aggregiert werden.
3. *Abfragen zum Ablauf abgeschlossener Instanzen.* Auch bekannt als „Process Mining“, da Workflowdefinitionen aus bisherigen Abläufen bzw. deren Logs heraus generiert werden sollen.

Die Art der in dieser Arbeit entwickelten Anfragesprache ergibt sich aus den Anforderungen an die Sprache. Die WQL orientiert sich an Punkt 2, denn Informationen zu Abläufen werden ermittelt und auch Pfade können verfolgt werden. Es werden jedoch weder eine laufende Instanz noch ein eindeutiger Pfad bis zum aktuellsten Status betrachtet, sondern eine ggf. sehr hohe Zahl wahrscheinlicher Abläufe ohne einen aktuellen Status, der verfolgt oder simuliert werden könnte. Auch greift die WQL nicht in Abläufe ein oder überwacht diese. Die WQL kann daher als eine neue vierte Art einer Anfragesprache gesehen werden.

Der WQL am nähesten kommt BPMN-Q [Awa07]. In dieser visuellen Abfragesprache wird in Workflowdefinitionen nach Mustern gesucht. Die hier entwickelte Sprache verfolgt jedoch einen anderen Ansatz und ermittelt Schätzwerte für Zeiten und Kosten von Abläufen. Verwandte Arbeiten von BPMN-Q, die ebenso verwandte Arbeiten der WQL sind, werden in [Awa07] bereits in die o.g. Kategorien eingeordnet und aufgelistet, weshalb hier darauf verzichtet werden soll.

## 3.3 Workflow-Modell

Für die Darstellung von Workflows wurden für die WQL UML-Aktivitätsdiagramme gewählt. Aktivitätsdiagramme eignen sich gut zur Beschreibung von Workflows. Die Syntax ist zwar nicht in allen Punkten eindeutig, die wichtigsten Muster von Workflows werden jedoch unterstützt [DH01, vgl. S. 89]. Nicht eindeutige Stellen werden hier größtenteils vermieden durch eine geringere Zahl an modellierten Elementen und genauere Spezifizierung des Verhaltens der vorhandenen. Die verwendete Ontologie für die Darstellung in RDF ist eine vereinfachte und angepasste Version der in [Dol04] enthaltenen. Es soll daher beschrieben werden, welche Elemente verwendet wurden und wofür sie eingesetzt werden können:

Zunächst muss unterschieden werden zwischen Zuständen und Transitionen. Transitionen führen von einem Zustand (`fsm:Source`) in einen anderen (`fsm:Target`). Sie haben gegebenenfalls einen Namen (`fsm:TransitionName`), der in Abfragen als Bezeichner verwendet werden kann, und in Ausgaben an Nutzer lesbarer als ein URI sein sollte. Der optionale Name ist jedoch kein eindeutiger Bezeichner und sollte daher vorsichtig eingesetzt werden. Eine Transition kann eine Bedingung haben (`fsm:Condition`), die einem Guard einer Behavioral State Machine [Obj11, S. 584] entspricht. Die in der Implementierung unterstützte Syntax einer Bedingung wird in Abschnitt 3.6 beschrieben. Weiterhin können an Verzweigungen individuelle Wahrscheinlichkeiten für Transitionen angenommen werden. Listing 3.1 zeigt daher deren Aufbau beispielhaft, relevant dafür sind `fd:Probability`, `fd:probValue` und `fd:probability`. Zur Vereinfachung verursachen Transitionen keine Kosten jeglicher Art und benötigen keine Zeit für den Übergang. Dies schränkt die Beschreibung von Workflows nicht ein, denn sollten für eine Transition dennoch Kosten und Zeiten modelliert werden, so kann diese durch einen neuen Zustand mit den entsprechenden Kosten und Zeiten sowie zwei Transitionen ersetzt werden.

```
1 :T1              a                   fsm:Transition ;
2                  fsm:Source          :X ;
3                  fsm:Target          :Y ;
4                  fd:probability      :Prob .
5
6 :Prob            a                   fd:Probability ;
7                  fd:probValue        0.3 .
```

**Listing 3.1:** Eine Wahrscheinlichkeit von 30% für Transition `:T1`

Arbeitsschritte, parallele Abläufe und Gruppierungen von Arbeitsschritten werden als Zustände modelliert. Zustände können eingehende und ausgehende Transitionen haben und führen optional einen Namen (`StateName`). Weiterhin kann es zusätzliche Informationen, wie Zeiten und Kosten, zu ihnen geben.

Da das Modell auf Erweiterungen ausgelegt ist, einfache Angaben jedoch nicht unnötig kompliziert geschrieben werden sollen, können exakte Werte für Zeiten oder Kosten direkt angegeben werden. Zeiten werden grundsätzlich in Sekunden geschrieben, Kosten ohne Einheit, d. h. deren Verwendung kann beim Modellieren frei entschieden und muss bei Abfragen jedoch beachtet werden. Soll ein Arbeitsschritt z.B. 30 Minuten dauern und eine frei gewählte Kostenart „Arbeitseinheiten“ mit dem Wert 2 sowie eine weitere Kostenart „cost“ mit dem Wert 50 verursachen, so würde dies wie in Listing 3.2 formuliert. Anstatt exakter Werte dürften für die Eigenschaft `c:value` jederzeit auch Angaben vom Typ `ds:DataStructure` stehen. Einzusetzende neue Datentypen werden daher als Unterklasse von `ds:DataStructure` definiert weshalb im Anhangskapitel A.2 neben der Definition von `ds:DataStructure` exemplarisch auch eine Unterklasse für Histogramme dargestellt wird.

```
1  :S              a                   fsm:Simple ;
2                  fsm:StateName       "Name" ;
3                  fd:time             1800 ;
4                  c:cost              :c1 , :c2 .
5
6  :c1             a                   c:Cost ;
7                  c:name              "Arbeitseinheiten" ;
8                  c:value             2 .
9  :c2             a                   c:Cost ;
10                 c:name              "cost" ;
11                 c:value             50 .
```

**Listing 3.2:** Benannter Arbeitsschritt mit Kosten- und Zeitangabe

Nachfolgend werden die verschiedenen Typen von Zuständen beschrieben. Jeder Workflow beginnt mit einem initialen Zustand (`fsm:Initial`) und endet mit einem finalen Zustand (`fsm:Final`). Diese Zustandstypen werden auch für den Beginn bzw. das Ende von zusammengesetzten Zuständen und parallelen Regionen verwendet. Der nächste wichtige Typ ist ein einfacher Zustand (`fsm:Simple`), der einen nicht weiter unterteilten Arbeitsschritt beschreiben soll. Verzweigungen (`fsm:Branch`) erlauben mehrere ausgehende Transitionen, wobei der tatsächlich eingeschlagene Pfad idealerweise letztendlich eindeutig ist durch die an Transitionen angegebenen Bedingungen oder direkt angegebene

Entscheidungen. Ist er nicht eindeutig, müssen gegebenenfalls viele Pfade untersucht werden, Abschnitt 3.5 beschreibt das hier konkret angewendete Verfahren. Die Syntax der Bedingungen von Transition wird in Abschnitt 3.6 beschrieben, angegeben wird eine solche wie in Listing 3.3.

```
1 :T2              a                    fsm:Transition ;
2                  fsm:Condition        "age >= 50 && fever = 1" ;
3                  fsm:Source           :X ;
4                  fsm:Target           :Y .
```

**Listing 3.3:** Bedingung einer Transition

Mehrere Arbeitsschritte können zu einem zusammengesetzten Zustand (`fsm:Composite`) zusammengefasst werden, der sowohl einen Beginn als auch ein Ende haben sollte. Transitionen müssen jedoch nicht unbedingt letztendlich zum Endzustand des zusammengesetzten Zustands führen, sondern können auch vorzeitig nach außen gehen. Dies macht die Darstellung zwar flexibler, anhand der Transitionen können jedoch nicht die „Kindzustände“ erkannt werden, weshalb diese mit `fsm:hasStateMachineElement` aufgelistet werden müssen. Parallele Abläufe werden ebenfalls als zusammengesetzte Zustände modelliert, jeder parallele Zweig ist eine `fsm:Region`. Anstatt initialen und finalen Zuständen hat ein solcher Zustand mit parallelen Vorgängen nur die Regionen als Kindknoten, Regionen werden wie zusammengesetzte Zustände aufgebaut, deren Arbeitsabläufe sind also ihre mit `fsm:hasStateMachineElement` aufgeführten Elemente. Parallele Abläufe wurden stärker eingeschränkt, so gibt es nur zwei Möglichkeiten für deren Ende: Entweder erreichen alle Zweige ihren finalen Zustand oder ein Zweig sorgt durch eine Transition nach außen für ein vorzeitiges Ende aller parallelen Zweige. Abbildung 3.1 zeigt einen Beispiel-Workflow mit Parallelität und einem zusammengesetzten Zustand, deren RDF-Darstellung in Listing 3.4 auf die wesentlichen Besonderheiten gekürzt und ohne Transitionen gezeigt wird.

Für parallele Abläufe wurden Fork, Join und SynchStates [Dol04] jedoch verworfen, da ihre Verwendung zu mehrdeutigen Diagrammen führen und ein Ende von parallelen Vorgängen ggf. nicht eindeutig bestimmt werden kann. Im Gegensatz zum Ansatz mit zusammengesetzten Zuständen und Regionen wäre ein vorzeitiger Abbruch der Parallelität nur schwer oder nicht darstellbar.

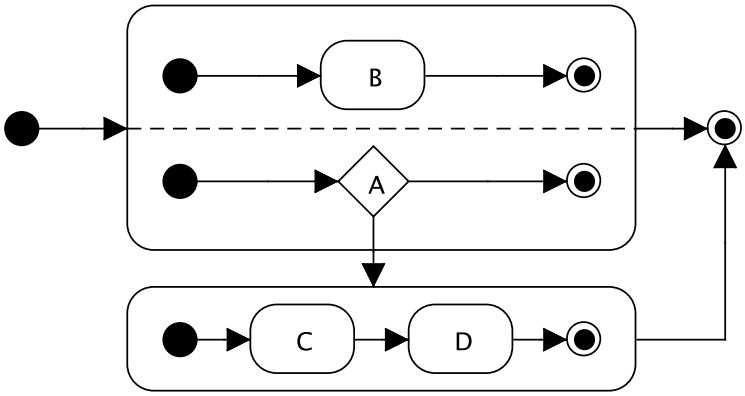



**Abbildung 3.1:** Workflow mit parallelen Regionen und zusammengesetztem Zustand

```
# Initial-Knoten: i1,i2,i3
# Final-Knoten: f1, f2, f3
# 1/2 Regionen, 3 zusammengesetzter Zustand
:Par            a                           fsm:Composite ;
                fsm:hasStateMachineElement  :R1, :R2 .
:R1             a                           fsm:Region ;
                fsm:hasStateMachineElement  :i1, :b, :f1 .
:R2             a                           fsm:Region ;
                fsm:hasStateMachineElement  :i2, :a, :f2 .

:Comp           a                           fsm:Composite ;
                fsm:hasStateMachineElement  :i3, :c, :d, :f3 .
```

**Listing 3.4:** Workflow mit parallelen Regionen und zusammengesetztem Zustand

## 3.4 Fallakte

Als Besonderheit der WQL wird nun das Konzept der Fallakte für Abfragen vorgestellt. Bedingungen von Kanten sind in Diagrammen von Workflows keine Seltenheit und auch im vorherigen Kapitel zum Workflow-Modell enthalten. Ein medizinischer Workflow, dessen Ablauf von konkreten Patientendaten, wie z.B. dem Alter oder dem Vorhandensein diverser Vorerkrankungen, abhängig ist, sollte sinnvollerweise auch bei Abfragen diese Kriterien möglichst einfach einbeziehen können.

Die Syntax für die dazugehörigen Bedingungen an Kanten wird in Kapitel 3.6 beschrieben, wichtig für dieses Kapitel ist dabei jedoch ausschließlich die Verwendung von Variablen in Ausdrücken. Diese sind für konkrete Patienten bzw. allgemeiner für

Instanzen des Workflows relevant und sollten daher bei einer Abfrage angegeben werden können.

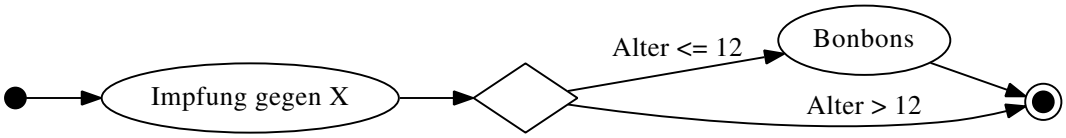



**Abbildung 3.2:** Die wahrgenommenen Arbeitsschritte einer Impfung

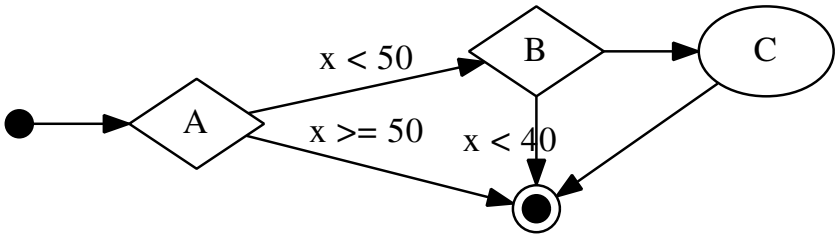



**Abbildung 3.3:** Verwendung einer Variable x

Wie Abbildung 3.3 zeigt, kann eine einzelne, gezielt eingesetzte Variable sehr großen Einfluss auf mögliche Abfolgen haben. Es gibt für dieses Beispiel vier mögliche Angaben für $x$: keine, $x < 40$, $x \geq 40 \wedge x < 50$ und $x \geq 50$ mit unterschiedlicher Anzahl möglicher Pfade. Statt Intervallen wurden stellvertretend Werte aus diesen gewählt. Was in der folgenden Tabelle auffallen wird, ist, dass auch bei Erfüllen der Bedingung von $x < 40$ der Transition $B \to \odot$ die Transition $B \to C$ beachtet wird. Bedingungen werden nur als Ausschlusskriterium eingesetzt, nie als Werkzeug zur Auswahl, was insbesondere bei mehreren möglichen Kanten an einer Verzweigung nützlich ist, aber damit auch ein auf den ersten Blick unintiuitives Resultat für $x = 30$ zur Folge haben kann. Die Bedingung einer „leeren Kante“ gilt immer als erfüllt.

| | | | |
|---|---|---|---|
| $x$ fehlt: | 1. | $\bullet\ A\ \odot$ | 50 % |
| | 2. | $\bullet\ A\ B\ \odot$ | 25 % |
| | 3. | $\bullet\ A\ B\ C\ \odot$ | 25 % |
| $x = 30$: | 1. | $\bullet\ A\ B\ \odot$ | 50 % |
| | 2. | $\bullet\ A\ B\ C\ \odot$ | 50 % |
| $x = 40$: | | $\bullet\ A\ B\ C\ \odot$ | 100 % |
| $x = 50$: | | $\bullet\ A\ \odot$ | 100 % |

Aber auch tatsächliche Entscheidungen für Verzweigungen und gemessene Zeitangaben für einzelne oder aggregierte Zustände sollten einer Fallakte zugeordnet werden. Entscheidungen können zwar als Variable mit Zahlenwert für jeden möglichen Ausgang und entsprechende Bedingungen an allen zugehörigen Kanten ausgedrückt werden, sie werden aber trotzdem durch eine direkte Zuweisung mit optionalem Intervall für Besuche in einer Fallakte angenehmer unterstützt. Abbildung 3.4 zeigt vier mögliche Transportmittel für den Weg zum Krankenhaus in einem Workflow. Der intuitivste Weg, das Transportmittel zu nennen, wäre die Angabe der Entscheidung Transportmittel $\rightarrow$ Auto. Mächtiger ist diese Vorgehensweise jedoch in Zyklen, denn z.B. für den ersten bis fünften Durchlauf durch eine Verzweigung kann eine Entscheidung gewählt werden, die letztenendes wieder zur Verzweigung führt und abschließend eine aus dem Zyklus herausführende Transition.

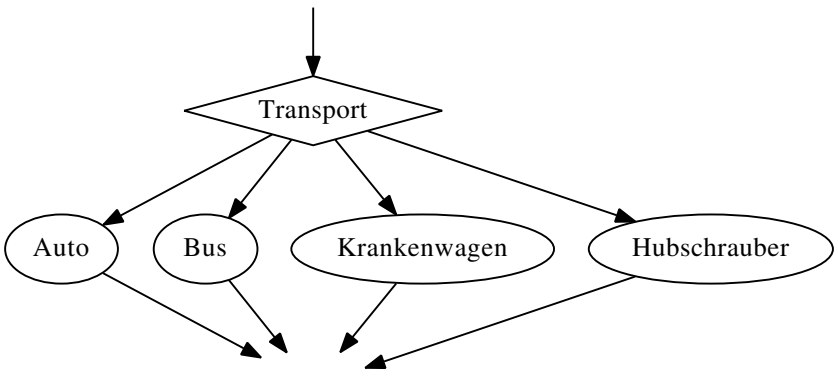



**Abbildung 3.4:** Beispiel für Entscheidung in Workflow

Konkrete Zeitangaben für alle beteiligten Arbeitsschritte zu fordern ist häufig nicht möglich. So ist es unwahrscheinlich, dass z.B. die Besatzung eines Rettungshubschraubers (siehe Abbildung 3.5) alle Einzelheiten eines Einsatzes notiert anstatt sich vollständig auf den Patienten zu konzentrieren und der behandelnde Arzt wird sich eher um die gesamte verstrichene Zeit seit einem Notruf sorgen. Ein Mix aus aggregierten und exakten Werten kann in diesem und vielen anderen Beispielen nicht ausgeschlossen werden, weshalb Abfragen dies unterstützen sollten. Darüber hinaus eignet sich das Sammeln von aggregierten tatsächlichen Werten und ein Vergleich mit im Modell angenommenen Informationen gut für Plausibilitätsprüfungen, um zu prüfen, inwiefern ein Workflow sich mit der Realität vereinbaren lässt. Zeitangaben können verallgemeinert jedoch auch als Kosten gesehen werden, weshalb in zukünftigen Implementierungen auch verschiedene Kostenarten mit Namen unterstützt werden sollten.

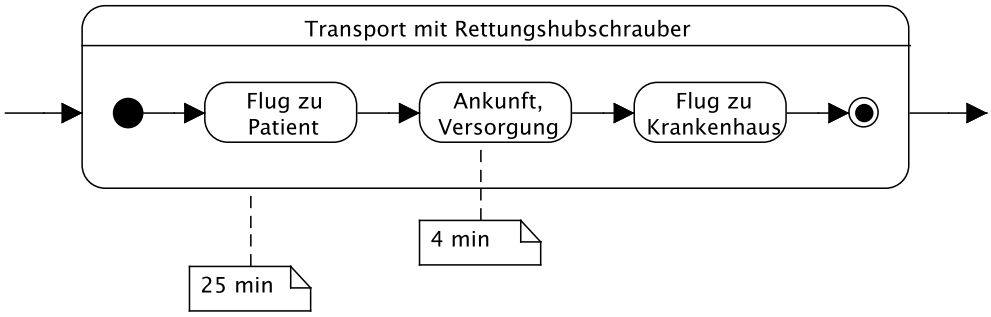



**Abbildung 3.5:** Aggregierte und direkte Zeitangaben

Eine Fallakte besteht zusammengefasst aus folgenden Elementen:

1. Variablen
   Tupel (Name, Wert)
2. Entscheidungen
   Eine Entscheidung wird eindeutig beschrieben durch eine Transition und ein Intervall für die jeweiligen Durchläufe, für die die Entscheidung gelten sollen. Da die WQL für Transitionen weder Kosten noch einen Zeitverbrauch annimmt, werden Entscheidungs- und Zielknoten sowie optional ein Intervall für Besuche angegeben. Ohne Beschränkung gilt die Entscheidung für jeden Besuch des Entscheidungsknotens.
   Tupel (Entscheidungsknoten, Zielknoten, Intervall für Besuche)

3. Zeiten
   Grundsätzlich wären auch für Zeitangaben Intervalle für Besuche, in denen diese gültig sind, sinnvoll. Vorerst wurde jedoch davon abgesehen, um den Zeitverbrauch von Knoten in Zyklen nicht jedes Mal neu ermitteln zu müssen. Tupel ((aggregierter) Arbeitsschritt, Zeitverbrauch)

Fallakten können sowohl mit eindeutigem Namen als auch anonym in Abfragen angegeben werden. Wo und wie diese Fallakten gespeichert werden, soll hier nicht festgelegt werden. In der Implementierung dieser Arbeit werden Fallakten ebenso im Graphen einer SPARQL-Datenbank abgelegt, in der auch die modellierten Workflows gespeichert werden. Statt „Fallakte“ wurde ein allgemeinerer Begriff „Record“ gewählt, da die WQL nicht nur auf medizinische Workflows beschränkt werden soll. Folgende Operationen sollen unterstützt werden, die konkrete Syntax zu diesen wird im nächsten Kapitel beschrieben:

1. Anlegen
2. Ändern
3. Ausgeben
4. Löschen
5. Bestimmte Zeitangabe entfernen
6. Bestimmte Variable entfernen
7. Bestimmte Entscheidung entfernen

Eine einzelne Fallakte, ggf. durch Ergänzungen einer anonymen, wird in Abfragen komplett mit all ihren Variablen, Entscheidungen und Zeitangaben verwendet. Da Erfahrungswerte von Zeitangaben jedoch auch ein guter Indikator für zukünftige Instanzen von Workflows sein können, gibt es in Abfragen auch die Möglichkeit, Zeitangaben bisheriger Fallakten des Workflows einzubeziehen. Diese werden entweder komplett oder als Auswahl durch direkte Angabe ihrer Namen berücksichtigt.

## 3.5 Pfade in Workflows

Die tatsächliche Abfolge von Arbeitsschritten in Abfragen zu einem Workflow kann uneindeutig sein oder nicht mit SPARQL ermittelt werden, wie beispielsweise Ausdrücke an von Verzweigungen ausgehenden Transitionen, nicht spezifizierte Entscheidungen oder Parallelität mit vorzeitigen Abbrüchen. Ebenso kann es dank Zyklen mit unvollständigen Angaben unendlich viele oder eine viel zu hohe, nicht zu verarbeitende endliche Zahl möglicher Pfade geben.

Um diese Probleme zu veranschaulichen wurden vier sehr einfache Workflows konstruiert, die so auch in der Realität auftreten können. Abbildung 3.6 zeigt einen Zyklus im Workflow an, es gibt unendlich viele Pfade. Gibt man den Transitionen an der Verzweigung jedoch Wahrscheinlichkeiten, z.B. $B \to \odot$ und $B \to A$ mit jeweils 50%, kann man dennoch durchschnittliche Kosten annähern, indem man sich eine beliebige Anzahl wahrscheinlicher Pfade heraussucht. Arbeitsschritt A koste 10€, Arbeitsschritt B sei kostenlos. Da dieses Beispiel sehr klein ist kann man die durchschnittlichen Kosten aller Pfade von $\bullet$ nach $\odot$ hier sehr leicht exakt berechnen: $cost = \sum_{i=1}^{\infty} \frac{1}{2^i} \cdot 10i = 10 \sum_{i=1}^{\infty} i \cdot 2^{-i} = 10 \cdot 2 = \underline{20€}$.

Da jedoch eine exakte Berechnung in den meisten Fällen schwieriger ist, kann dies damit umgangen werden, wie in Tabelle 3.1 die *wahrscheinlichsten* Abfolgen herauszufinden und deren Kosten mit der Wahrscheinlichkeit zu gewichten und zu Gesamtkosten aufzusummieren. Da dies nur angewendet werden muss, wenn ohnehin zu wenige Daten für einen eindeutigen Ablauf vorhanden sind und damit geschätzt wird, können ggf. auch höhere relative Fehler unproblematisch sein. Summiert man die Wahrscheinlichkeiten aller Pfade, kann dieser Wert als Kennzahl für die Sicherheit des Ergebnisses dienen.

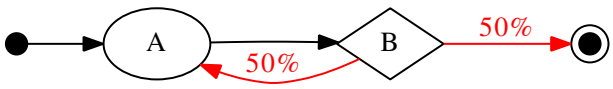



**Abbildung 3.6:** Zyklus in Graph

In Abbildung 3.7 gibt es auch mit geringer Anzahl an Knoten und Kanten sehr viele mögliche Pfade durch den Workflow. Das Beispiel ist so gewählt, dass immer nur maximal ein Knoten übersprungen werden kann, ansonsten ist es eine zyklenfreie Abfolge von Arbeitsschritten. Betrachtet man den Workflow genauer, wird man feststellen, dass die Anzahl möglicher Pfade von einem Arbeitsschritt zum Ende hin hier eine Fibonacci-Zahl ist. Fügt man einen Arbeitsschritt und zwei Transitionen hinzu, so wird die Anzahl der Möglichkeiten in etwa um den goldenen Schnitt $\Phi \approx 1{,}618$ vervielfacht, d. h. die Anzahl

| Anzahl Pfade | ∅Kosten | rel. Fehler | ∑ Wahrscheinlichkeiten | zusätzlicher Ablauf |
|---|---|---|---|---|
| 1 | 5.00000 | 75.0000% | 50.000000% | $\bullet AB\odot$ |
| 2 | 10.00000 | 50.0000% | 75.000000% | $\bullet ABAB\odot$ |
| 3 | 13.75000 | 31.2500% | 87.500000% | $\bullet ABABAB\odot$ |
| 4 | 16.25000 | 18.7500% | 93.750000% | $\bullet ABABABAB\odot$ |
| 5 | 17.81250 | 10.9375% | 96.875000% | ... |
| 6 | 18.75000 | 6.2500% | 98.437500% | |
| 7 | 19.29688 | 3.5156% | 99.218750% | |
| 8 | 19.60938 | 1.9531% | 99.609375% | |
| 9 | 19.78516 | 1.0742% | 99.804688% | |
| 10 | 19.88281 | 0.5859% | 99.902344% | |
| 15 | 19.99481 | 0.0259% | 99.996948% | |
| 20 | 19.99979 | 0.0010% | 99.999905% | |
| 25 | 19.99999 | 0.0000% | 99.999997% | |
| 26 | 20.00000 | 0.0000% | 99.999999% | |

**Tabelle 3.1:** Durchschnittliche Kosten und Fehler bei fester Anzahl betrachteter wahrscheinlichster Pfade für Abbildung 3.6

möglicher Pfade kann auch bei überschaubaren Workflows mit zusätzlichen Arbeitsschritten und Transitionen exponentiell wachsen. Somit kann es auch bei einer endlichen Zahl an Möglichkeiten notwendig sein, nur die Wahrscheinlichsten zu betrachten. Der gezeigte Workflow würde 233 mögliche Pfade bieten, wobei es 7 gleich wahrscheinliche wahrscheinlichste Lösungen gibt ($\bullet LKIGECA\odot$, $\bullet LJIGECA\odot$, $\bullet LJHGECA\odot$, $\bullet LJHFECA\odot$, $\bullet LJHFDCA\odot$, $\bullet LJHFDBA\odot$, $\bullet LJHFDB\odot$). Würde man stattdessen ein Überspringen von maximal zwei aufeinanderfolgenden Arbeitsschritten erlauben, so gäbe es bereits 927 Lösungen.

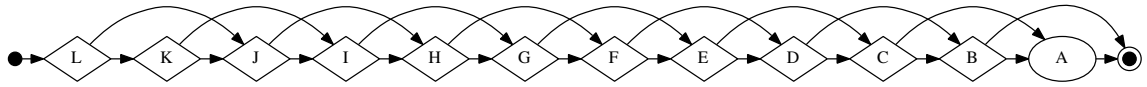



**Abbildung 3.7:** Explodierende Anzahl Möglichkeiten

Gibt es wie in Abbildung 3.8 nur sehr wenige Pfade, so kann das Ermitteln der wahrscheinlichsten alle finden und Endergebnisse durch Wahrscheinlichkeiten gewichtet errechnen. Dies ist ideal, wenn noch nicht alle Details eines durchlaufenen Workflows feststehen, aber dennoch Aussagen dazu getätigt werden sollen. Ein Problem für eine SPARQL-Abfrage wäre dieser Workflow dennoch, denn es muss gewichtet oder die Bedingungen der Ausgangstransitionen von A müssten getestet werden.

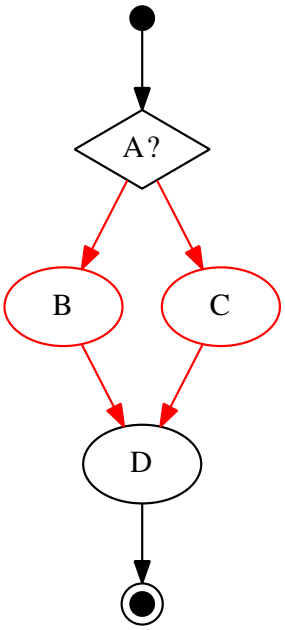



**Abbildung 3.8:** Geringe Anzahl möglicher Pfade

Das vierte Beispiel wie in Abbildung 3.9 behandelt nun Parallelität in Workflows. Es gibt zwei Möglichkeiten, wie sich ein Arbeitsablauf verhalten soll: Es wird gewartet, bis alle parallelen Zweige fertig sind oder sobald einer oder eine gewisse Zahl an Zweigen abgeschlossen, wird der Zustand verlassen. Da im Abschnitt 2.3 Workflows so eingegrenzt wurden, dass sie warten, bis alle Zweige abgearbeitet sind und bei einer ausgehenden Transition vorzeitig zum nächsten Zustand gewechselt wird, wird dieses Beispiel genauso behandelt. In der zweiten Region des parallelen Zustandes bestimmt die Verzweigung, ob dieser Zustand komplett läuft oder ob vorzeitig zu C gewechselt wird, d. h. es gibt folgende mögliche Abläufe:

1. $\bullet\ (\ \bullet\ B\ \odot\ |\ \bullet\ A\ \odot\ )\ \odot$
2. $\bullet\ (\ \bullet\ B\ \odot\ |\ \bullet\ A\ )\ C\ \odot$
3. $\bullet\ (\ \bullet\ B\ |\ \bullet\ A\ )\ C\ \odot$
4. $\bullet\ (\ \bullet\ |\ \bullet\ A\ )\ C\ \odot$

Dem ersten Pfad kann aufgrund der Verzweigung A eine Wahrscheinlichkeit zugewiesen werden, die Möglichkeiten 2-4 sollten jedoch zusammengefasst werden: Bekommen sie eine Wahrscheinlichkeit zugewiesen, bevor anhand der Zeitangaben der genaue Ablauf

bestimmt wird, fallen zwei ohnehin unmögliche Lösungen heraus. Da die Summe der Wahrscheinlichkeiten der betrachteten Pfade jedoch als Kennzahl für die Genauigkeit verwendet werden kann, würde diese ohne Not verringert und das Ergebnis scheint ungenauer zu sein, als es tatsächlich ist.

Da dieses Beispiel sehr nahe an realen Workflows ist, sei z.B. B eine Standardbehandlung gegen Grippe und A eine parallele Blutuntersuchung, die Verzweigung dabei das Handeln nach dem Ergebnis, wird im Falle einer normalen Grippe dieser Zweig beendet mit Warten und bei einer unerwarteten schwereren Erkrankung vorzeitig zu Behandlung C gewechselt. Damit muss die Anfrageverarbeitung Parallelität auch mit vorzeitigem Abbruch beherrschen, um einen solchen Workflow untersuchen zu können.

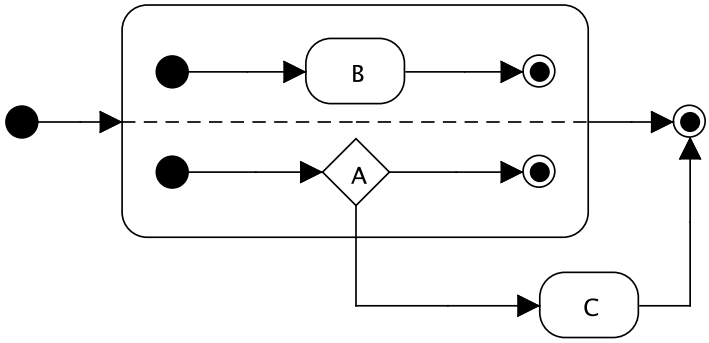



**Abbildung 3.9:** Parallelität in Workflows

Da nun mögliche Probleme beispielhaft gezeigt wurden, wird nun ein Verfahren vorgeschlagen, welches damit umgehen kann. Zunächst wird dazu auf Parallelität verzichtet, was die Lösung übersichtlicher macht, eine Verarbeitung und Gewichtung über Markow-Ketten ist jedoch wegen der Parallelität trotzdem ausgeschlossen. Später werden geringe Änderungen vorgenommen, um das hier gewählte Verfahren auch auf parallele Zweige anzupassen. Das grundsätzliche zunächst vereinfachte Vorgehen ist folgendes:

1. Beteiligte Knoten finden
2. Eine maximale Anzahl wahrscheinlicher Pfade ausprobieren, Knoten gewichten
3. Zeitangaben oder Kosten der gefundenen Knoten herausfiltern
4. Gewünschte Werte (Gesamtzeit/-kosten) ermitteln

Punkt 1 kann bequem über SPARQL-Abfragen erfolgen, deren Ergebnisse entweder alle oder nur relevante Arbeitsschritte und Transitionen des gewählten Workflows sind. Mit

relevant ist hierbei gemeint, dass nur Knoten und Kanten weggelassen werden dürfen, die definitiv nicht gebraucht werden. Mit unnötigen Angaben kann die vorgestellte Heuristik problemlos umgehen, es muss also nicht unbedingt gefiltert werden.

Zwischen Punkt 1 und 2 muss Vorarbeit geleistet werden: Ausgehend vom gewünschten Endzustand der Abfrage werden Kanten rückwärts besucht und ein Flag gesetzt, dass diese Kante den Endzustand erreicht. Erreicht dieser Vorgang die selbe Kante später wieder, wird deren Ausgangsknoten nicht noch einmal durchlaufen, d.h. kein zusätzliches Besucht-Flag ist nötig. Kanten bzw. Transitionen, die aufgrund vorhandener Bedingungen an einer Verzweigung nicht benutzt werden könnten, sollten hier bereits aussortiert werden, da sonst später unmögliche Abläufe produziert werden können, die nicht den gewünschten Endzustand erreichen und damit aus der Lösung fallen, ihre Wahrscheinlichkeit aber ebenso aus der Kennzahl für Genauigkeit. Wurden alle Kanten identifiziert, die den gewünschten Endzustand erreichen können, so wird diesen Kanten an Verzweigungen eine Wahrscheinlichkeit zugewiesen, die in Schritt 2 benötigt wird.

Da die Heuristik nur die wahrscheinlichsten Pfade heraussucht, die vorher genannte explodierende oder unendliche Anzahl an Möglichkeiten macht dies nötig, wird dafür ein nach Wahrscheinlichkeit sortiertes Array an Abläufen erstellt. Ein Array wurde gewählt, um die Benutzung ein wenig zu erleichtern. Da zum Verschieben und Löschen nur die Zeiger auf die jeweiligen Elemente der Liste geändert werden müssen, ist dieser Vorgang für kleine Anzahlen unproblematisch und vermutlich schneller als die meisten Alternativen. Eine Halde eignet sich nicht, denn es wird in jedem Durchlauf das wahrscheinlichste und zum Einreihen in die Liste das unwahrscheinlichste Element benötigt, eine Halde würde nur eine der beiden Wünsche in hoher Geschwindigkeit erfüllen. Sollen sehr viele verschiedene Pfade ausprobiert werden können, kann ein AVL-Baum gewählt werden, der für alle hier relevanten Operationen (Minimum, Maximum, Einfügen, Löschen) eine Komplexität von $\mathcal{O}(\log(n))$ hat. Das sortierte Array kann das Minimum/Maximum zwar in $\mathcal{O}(1)$ ermitteln und das Entfernen des letzten Elementes braucht ebenso konstante Zeit, ein Einreihen in die Liste oder das Löschen des wahrscheinlichsten Elementes geschieht jedoch in $\mathcal{O}(n)$. Das Array bekommt den Vorzug daher, weil die konstanten Faktoren der $\mathcal{O}$-Notation nur gegen unendlich ignoriert werden können, nicht jedoch für kleine $n$.

Ein in diese Liste oder ggf. den Baum einzuordnender Ablauf benötigt folgende Informationen: Ausgangsknoten, Zielknoten, derzeitige Wahrscheinlichkeit und bisherige Abfolge. Beschränkt werden kann die die Abläufe speichernde Struktur entweder in der maximalen Anzahl der betrachteten Pfade, einer Mindestwahrscheinlichkeit jedes betrachteten Pfades (dies kann wesentlich restriktiver sein als die Begrenzung der Anzahl)

oder einer Verbindung dieser beiden Kriterien. Um Endlosschleifen zu vermeiden für den Fall, dass nur die maximale Anzahl an Pfaden betrachtet wird, sollte ein Counter mitlaufen, der fehlgeschlagene Pfade mitzählt und diese von den maximal erlaubten Möglichkeiten abzieht. Für diese Anfrageverarbeitung wurde eine Mischung der beiden Kriterien gewählt, die Anzahl der Pfade kann hierzu begrenzt werden und ein Pfad muss eine angegebene Mindestwahrscheinlichkeit haben oder wird aussortiert.

Wenn die Vorarbeiten bzgl. Wahrscheinlichkeiten und Erreichbarkeit getätigt sind und die Struktur der zwischengespeicherten Abläufe geklärt ist, wird zunächst ein initialer Ablauf mit gewünschtem Anfangs- und Endknoten und der Wahrscheinlichkeit 1,0 und danach Schritt 2 ausgeführt: Es wird der wahrscheinlichste Ablauf gewählt und fortgeführt, bis der Zielknoten, eine Verzweigung oder ein unlösbares Problem für den Pfad erreicht ist. Um diese Fälle herauszufinden werden alle relevanten Kanten, hier bedeutet relevant alle möglichen, die wahrscheinlich genug sind, herausgesucht. Jeder Ablauf startet mit einer Eingangswahrscheinlichkeit, die Wahrscheinlichkeiten der Kanten an Verzweigungen werden damit multipliziert. Weiterhin muss an diesem Punkt für jede Transition feststehen, ob sie benutzt werden kann, auch Variablen eines Workflows werden spätestens hier betrachtet. Werden keine relevanten Kanten gefunden, ist dieser Ablauf ungültig und wird aussortiert. Gibt es nur eine Kante, läuft der Vorgang über diese weiter, und werden an Verzweigungen mehrere relevante gefunden, so wird der aktuelle Ablauf kopiert und bekommt als neuen Startknoten das Ziel der jeweiligen Kante.

Ist die erlaubte Anzahl möglicher Pfade hinreichend groß, so werden tatsächlich die wahrscheinlichsten ermittelt. Da es sich bei diesem Ansatz zwangsläufig nur um eine Heuristik handelt, können auch Beispiele konstruiert werden, in denen diese Heuristik nicht optimal arbeitet. Ein mögliches Beispiel dafür wäre z.B. eine frühe Verzweigung, bei der die tatsächlich wahrscheinlichsten Pfade durch zunächst geringe Wahrscheinlichkeiten benachteiligt werden. Die Unwichtigeren treten zunächst mit höherer Wahrscheinlichkeit auf und werden durch viele Verzweigungen immer unwahrscheinlicher. Bis das jedoch erkannt wurde, können die wahrscheinlichsten Pfade bereits verdrängt sein. Die Summe der Wahrscheinlichkeiten aller gefundenen und bearbeiteten Pfade kann hier Informationen darüber liefern, wie gut man dem Ergebnis trauen kann und wie viel aussortiert wurde.

Abschließend, oben als Schritt 3 gelistet, können die Knoten aus den Wahrscheinlichkeiten der Abläufe heraus gewichtet werden und in Schritt 4 zusätzliche Informationen wie Gesamtkosten oder -zeiten gewichtet ermittelt werden.

Um auch Parallelität in das Verfahren bringen zu können sind nicht große Änderungen nötig. Die Anforderungen an diese Änderungen sind zunächst folgende:

1. Ein Ablauf darf nicht mehr eine einfache Liste besuchter Knoten sein, sondern muss auch parallele (Teil-)Abläufe darstellen können.
2. Parallele Abläufe können auch verschachtelt sein.
3. Vorzeitige Abbrüche von Parallelität sollten nicht alle parallelen Abläufe abbrechen, sondern nur solche, aus denen ausgebrochen wurde.

Die einfachste Lösung dafür wäre es, Parallelität als zusätzlichen Punkt zwischen Schritt 1 und 2 des bisherigen Ansatzes einzuführen und zu einem seriellen Ablauf zu übersetzen. Dabei würde ein Kreuzprodukt aus den Knoten paralleler Zweige erstellt und vorherige Transitionen der einzelnen Knoten übernommen. Problematisch dabei ist allerdings, dass die Anzahl der Knoten und vor allem der Verzweigungen dramatisch zunimmt, obwohl eine große Zahl dieser neuen Knoten real so nicht auftreten wird und die verwendete Heuristik die Anzahl getesteter Pfade begrenzt. Ein sehr einfacher Workflow wie in Abbildung 3.10 artet damit zum wesentlich schwerer handhabbaren Workflow der Abbildung 3.11 aus, eventuelle Zusatzinformationen zum übergeordneten parallelen Arbeitsschritt oder der jeweiligen Region wären, falls vorhanden, auch wesentlich schwieriger zuzuordnen. Die benötigte Zeit kann stark verfälscht werden, da für jeden Knoten das Maximum aller ihm zugrunde gelegten vorher parallelen Knoten gewählt wird oder eine weitere Aufteilung mit noch mehr Knoten und Kanten vorgenommen werden muss.

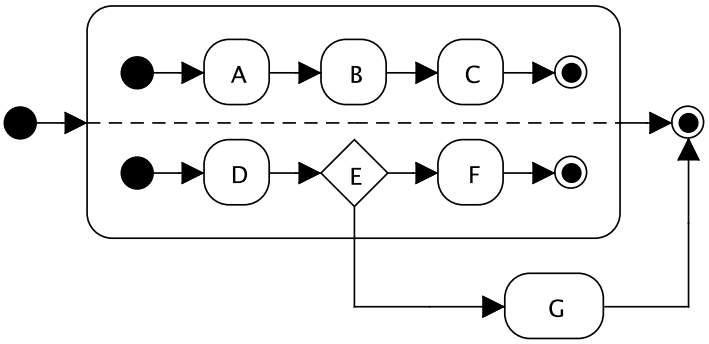



**Abbildung 3.10:** Einfacher Workflow mit Parallelität

Soll wie im Beispiel jedoch ausgenutzt werden, dass es real wesentlich weniger grundsätzlich verschiedene Pfade gibt, so ist folgender Ansatz sicher die bessere Wahl:

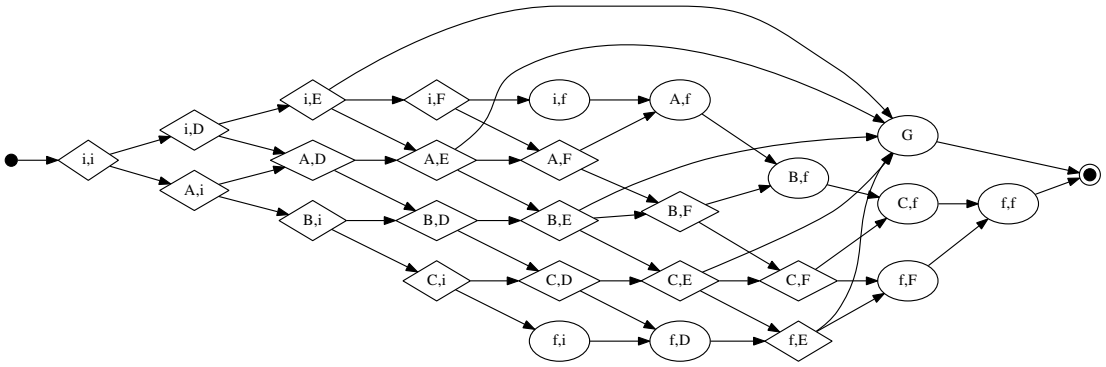



**Abbildung 3.11:** Abbildung 3.10 nach Kreuzprodukt für Parallelität

Ein Pfad ist keine einfache Liste aufeinanderfolgender Knoten mehr, sondern parallele Abschnitte verhalten sich wie ein Knoten, beinhalten jedoch alle parallelen Regionen, die ebenso eine abgewandelte Struktur enthalten. Da parallele Regionen entweder aufeinander warten müssen oder ein vorzeitiges Abbrechen auftritt und damit für die aufgebrachte Zeit entweder das Maximum oder das Minimum der nötigen Zeit paralleler Regionen ermittelt werden muss, beinhaltet ein solcher paralleler Knoten seinen Typ als Zusatzinformation.

Weiterhin wird es ein wenig schwieriger zu erkennen, welche Pfade den Zielknoten erreichen. So kann eine Kante sowohl vom übergeordneten Arbeitsschritt ausgehen als auch von einem vorzeitigen Abbruch. Für eine Kante des übergeordneten Arbeitsschrittes ist es dabei wichtig, dass alle parallelen Regionen des Arbeitsschrittes ihren finalen Knoten erreichen. Für einen vorzeitigen Abbruch gilt jedoch, dass alle anderen Regionen dennoch laufen, auch wenn das Ziel nur über diese Abbruchkante heraus erreicht werden kann.

Nun da die Erreichbarkeit geklärt und die Datenstruktur für den jeweils aktuellen Pfad an die neuen Gegebenheiten angepasst worden ist, kann die Veränderung von Schritt 2, dem Ausprobieren der wahrscheinlichsten Pfade, vorgestellt werden. Es wird nun rekursiv gearbeitet: Für parallele Abläufe wird abgestiegen und es werden die wahrscheinlichsten zum Ziel führenden Pfade jeder Region herausgesucht, falls auch dort parallele Abläufe enthalten sind genauso. Zurückgegeben wird im Erfolgsfall ein Pfad, bei der der letzte Knoten angibt, ob vorzeitig abgebrochen wurde (externer Knoten ausserhalb der Region) oder ob bis zum Ende gelaufen wird (finaler Knoten). Punkt 3 der aufgestellten Anforderungen, also nur parallele Abläufe abzubrechen, aus denen ausgebrochen wurde, ist somit möglich, denn externe Ziele werden an übergeordnete

Regionen weitergegeben und Parallelität abgebrochen, bis sie keine externen Ziele mehr sind.

Dennoch muss es ein Kreuzprodukt der Ergebnisse geben und ein Ausgangsknoten gewählt werden, bei vorzeitigem Abbruch der am frühesten abbrechende Zweig und, falls gewartet werden, soll das Ziel der Ausgangskante des übergeordneten Arbeitsschrittes. Es können hierbei allerdings direkt die wahrscheinlichsten Lösungen herausgesucht werden. Zusätzliche Informationen zu übergeordneten Regionen bleiben erhalten, sind leicht zuzuordnen und es muss eine wesentlich geringere Anzahl an Verzweigungen beachtet werden. Für vorzeitigen Abbruch kann das Kreuzprodukt weiter verfeinert werden, denn für alle Zweige, ausser dem vorzeitig abgebrochenen, sind zeitlich nur die Knoten bis zum Abbruch relevant. Diese Pfade müssen jedoch keinen Endzustand erreichen, der abbrechende Zweig bestimmt den nächsten Knoten. Diese Zweige sollten daher ohne größere Einschränkung, insbesondere auch auf Kanten, die keinen Endzustand erreichen, laufen dürfen, bis die Abbruchzeit erreicht ist. Vorher vorzeitig abbrechende Pfade werden aussortiert und auch in der Wahrscheinlichkeit nicht berücksichtigt oder kompensiert. Pfade, die den Endzustand des Zweiges erreichen, warten und dürfen nicht entfernt werden. Abschließend werden die Knoten gewichtet, wobei nun nach Zeiten und Kosten unterschieden werden muss, denn für parallele Abläufe fällt nur der maximale Zeitverbrauch der Zweige an, für Kosten müssen die Gewichte der Zweige jedoch summiert werden.

## 3.6 Bedingungen an Transitionen

Der Übergang von einem Zustand in einen anderen ist häufig an Bedingungen geknüpft. Lassen sich diese Bedingungen gut in logische Ausdrücke übertragen und hängen vorwiegend von leicht quantifizierbaren Merkmalen eines Patienten, Nutzers, Systems oder sonstigen Informationen ab, so können Entscheidungen als unumgänglich, möglich oder ausgeschlossen eingestuft werden ohne explizite Angabe der tatsächlichen Abfolge von Arbeitsschritten. Selbst wenn es für einen konkreten Workflow unmöglich sein sollte, durch logische Ausdrücke allein eindeutige Pfade zu ermitteln, so ist es dennoch denkbar, nur quantifizierbare Bedingungen zu verwenden und zumindest Entscheidungen auszusortieren, die gegen diese verstoßen würden.

Da es für Diagramme nicht unüblich ist, Bedingungen sehr frei zu formulieren, werden nicht verarbeitbare Voraussetzungen ignoriert und die dazugehörenden Transitionen als zumindest möglich eingestuft. Ausgewertet werden sie nur unmittelbar nach Verzweigun-

gen. Erzwungen werden kann dieses Verhalten auch durch Verzweigungen mit nur einer Ausgangstransition. Die in Listing 3.5 dargestellte Grammatik zeigt die unterstützte Syntax:

```
expression = '(', expression, ')' | '!', expression
           | condition | expression, boolOp, expression ;

condition = arithExpr, condOp, arithExpr ;

arithExpr = '(', arithExpr, ')' | value
          | arithExpr, arithOp, arithExpr ;

value = function | variable | constant ;

function = name, '(', [arithExpr, {',', arithExpr}], ')' ;
variable = name ;
constant = number, ['.', number] | '.', number ;

name = alpha, {alpha, digit} ;
number = digit, {digit} ;
alpha = 'a', 'b', ..., 'z', 'A', ... 'Z' ;
digit = '0', '1', ..., '9' ;

boolOp = '&' | '&&' | '|' | '||' ;
condOp = '=' | '<=' | '<' | '>=' | '>' | '/=' ;
arithOp = '+' | '-' | '*' | '/' ;
```

**Listing 3.5:** Grammatik für Bedingungen in EBNF

Die Präzedenzregeln stimmen mit denen der Grundrechenarten überein, bei den boolschen Operatoren bindet ein UND (& bzw. &&) stärker als ein ODER (| bzw. ||). Eine Negation mit '!' gehört zum nächsten je nach Klammerung kleinsten logischen Ausdruck. Als Symbol für $A \neq B$ wurde /= gewählt, die restlichen arithmetischen und vergleichenden Operatoren sollten selbsterklärend sein.

Tabelle 3.2 beschreibt alle definierten Funktionen. In der Signatur werden keine Typen angegeben, weil alle arithmetischen Operationen als Gleitkommazahlen verwendet werden, was auch die Funktionen `ceil`, `floor` und `round` nötig macht. Ein Spezialfall ist die Funktion `occurence`, mit der die Anzahl bisheriger Besuche des Ursprungsknotens einer Transition zurückgegeben wird. Nur `occurence` nutzt derzeit Informationen zum aktuell verarbeiteten Pfad. Soll erreicht werden, dass drei Transitionen nach jedem Durchlauf eines Arbeitsschrittes abwechselnd in gleicher Reihenfolge benutzt werden, so kann dies

wie in Abbildung 3.12 erreicht werden. Auch nicht definierte Funktionen oder eine falsche Anzahl an Parametern können verwendet werden, womit nur der komplette Teilausdruck bis hin zum nächsthöheren Vergleich, nicht jedoch der komplette Ausdruck, ungültig ist. Ebenso verhält es sich mit noch nicht gesetzten, aber im Ausdruck verwendeten Variablen. Ein Ausdruck `test(42) < foo` würde daher als Ergebnis *vielleicht* liefern, `test(42) < foo || 1 = 1` wird jedoch als *richtig* erkannt. Somit wird erlaubt, auch Bedingungen zu schreiben, die so noch nicht evaluiert werden können, aber vielleicht in Zukunft, um das Modellieren eines Workflows nicht künstlich einzuschränken.

| Signatur | Anmerkungen |
|---|---|
| occurence() | Anzahl bisheriger Besuche des Ursprungsknotens |
| abs(x) | |
| mod(a, b) | $a$ mod $b$ |
| ceil(x) | $\lceil x \rceil$ |
| floor(x) | $\lfloor x \rfloor$ |
| round(x) | |
| max(a, b) | |
| min(a, b) | |
| pow(a, b) | $a^b$ |
| pi() | $\pi$ |
| e() | $e$ |
| exp(x) | $e^x$ |
| ln(x) | natürlicher Logarithmus |
| log(x) | Logarithmus zur Basis 10 |
| sqrt(x) | Wurzel von x |
| sin(x), cos(x), tan(x) | Parameter x in Bogenmaß |

**Tabelle 3.2:** Unterstützte Funktionen

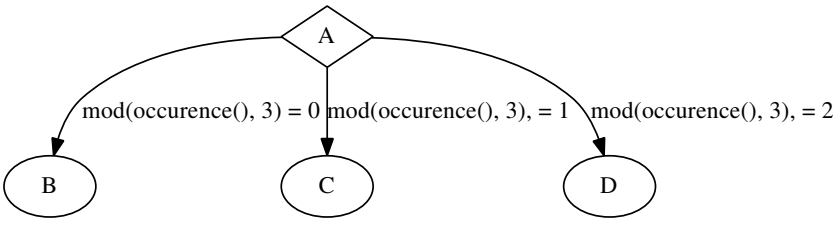



**Abbildung 3.12:** Abwechselndes Verwenden von Transitionen

## 3.7 Syntax & Semantik

Für die WQL werden in diesem Kapitel eine Reihe von Abfragen und Änderungsoperationen spezifiziert und deren Verhalten beschrieben. Eine Implementierung muss die in Abschnitt 3.7.1 beschriebenen `ESTIMATE`-Abfragen unterstützen, sollte Änderungsbefehle für Fallakten erlauben und kann optional auch die vorgeschlagenen Änderungsbefehle für Workflows umsetzen. Abschnitt 3.7.4 legt die dafür vorgesehene Grammatik fest und erläutert, an welchen Stellen Erweiterungen sinnvoll einzubringen sind.

### 3.7.1 ESTIMATE

`ESTIMATE`-Abfragen sind der Schwerpunkt der WQL und erlauben es, Zeiten, Kosten und Pfade von Abläufen eines gewählten Start- und Endzustandes zu ermitteln. Ihr allgemeiner Aufbau wird in Listing 3.6 gezeigt. Da auch mit genauen Angaben häufig keine eindeutigen Abläufe ermittelt werden können oder genauere Informationen zu Zeiten, Kosten und tatsächlichen Entscheidungen fehlen, kann diese Abfrageart konsequenterweise mit „Schätzung“ übersetzt werden. Eine `ESTIMATE`-Abfrage kann weiter unterteilt werden in folgende Fälle:

1. Durchschnittliche Zeiten/Kosten aller Pfade
   Alle besuchten Zustände werden gewichtet und abschließend werden mit diesen Gewichten Zeiten und Kosten summiert und zurückgegeben, z.B. die durchschnittliche Gesamtzeit aller gefundenen möglichen Abläufe.
2. Zustände und deren durchschnittliche Zeiten/Kosten über alle Pfade:
   Im Gegensatz zur ersten Art werden einzelne Zustände zurückgegeben. Durchschnittliche Zeiten und Kosten werden nicht für Abläufe, sondern für Zustände, ermittelt.
3. Zustände und deren Zeiten/Kosten des wahrscheinlichsten Pfades:
   Eventuell soll nur der wahrscheinlichste Pfad betrachtet werden, für den auch eine Angabe der Reihenfolge von Zuständen und zugeordnete Zeiten/Kosten möglich ist.
4. Kosten und Zeiten kompletter Pfade:
   Zurückgegeben wird neben angeforderten Kosten/Zeiten eines Pfades auch eine Repräsentation der tatsächlichen Abfolge. Da eine Darstellung in verschiedenen Spalten insbesondere wegen parallelen Vorgängen nicht möglich ist, wird ein Pfad in

eine einfache textuelle Darstellung, die Parallelität unterstützt, umgewandelt und in eine einzige Ergebnisspalte eingeordnet. Zusätzlich kann auch die Wahrscheinlichkeit jedes ausgegebenen Pfades zurückgegeben werden.

Diese Fälle werden trotzdem in einer Syntax zusammengefasst, da sie sich nur in der Darstellung der Endergebnisse unterscheiden, für alle vier müssen die wahrscheinlichsten gültigen Abläufe ermittelt werden. Zur Erkennung des jeweils zutreffenden Abfragetyps werden die Ergebnisspalten herangezogen. Fall 1 ist dabei der Basisfall und tritt nur auf, wenn ausschließlich Kosten und/oder Zeiten in den Ergebnisspalten gefordert sind. Fall 2 und 3 sind ähnlich, es werden einzelne Zustände zurückgegeben. Werden dabei Zustände gefordert (`ESTIMATE node, time, ...`) ist klar, dass einer dieser Fälle vorliegt, die Unterscheidung liegt in der Positionsangabe: die Funktion `pos` ist nur sinnvoll, wenn ein tatsächlicher und eindeutiger Ablauf zurückgegeben werden soll (`ESTIMATE node, pos(node), ...`), ohne sie wird Fall 2 gewählt. Ergebnisspalten mit Kosten/Zeiten sind hier genauso möglich und beziehen sich auf einzelne Zustände. Der vierte Fall wird durch die geforderte Ergebnisspalte path oder eine Renderfunktion für Abläufe, die path enthält, erkannt. Renderfunktionen sollen Pfade oder einzelne Knoten in verschiedenen Formaten als Zeichenkette zurückgeben. Die angebotenen Funktionen und deren gültige Abfragetypen werden in Tabelle 3.3 aufgeführt, Implementierungen können aber auch weitere anbieten. Ist eine Ergebnisspalte nur zum Erkennen des Typs nötig, soll jedoch nicht tatsächlich ausgegeben werden, kann diese zu *nichts* umbenannt werden, z.B. `path AS ""`, und wird bei der Rückgabe ignoriert. Dieser Ansatz dürfte durch ein eher seltenes Ignorieren wesentlich angenehmer zu verwenden sein als eine explizite Trennung der Fälle z.B. mit `ESTIMATE PATH` oder `ESTIMATE MOST LIKELY PATH`.

Häufig werden URIs benötigt, die in der Praxis meist mit gleichem Präfix beginnen. Alle Abfragen unterstützen daher für ihren Anfang optional eine Definition des Prä-

```
[ CONTEXT <URI> ]
  ESTIMATE Zeiten, Kosten, Funktionen
  OF Name/URI des Workflows
[ FROM Start-Zustand: URI oder Name ] [ TO Ziel: URI oder Name ]
[ USING RECORD Benannte Fallakte ]
[ USING RECORDS Fallakten für Durchschnittszeiten realer Abläufe ]
[ USING ALL RECORDS Kein Parameter, alle bekannten Fallakten für Vergleichswerte einbeziehen ]
[ WHERE {Variablen, Zeitangaben, Entscheidungen, ...} ]
[ ORDER BY Zeiten, Kosten, Funktionen ASC bzw. DESC ]
```

**Listing 3.6:** Allgemeiner Aufbau einer `ESTIMATE`-Abfrage

| Signatur | Gültige Typen | Anmerkungen |
|---|---|---|
| name(node) | 2,3 | Name des Zustandes |
| uri(node) | 2,3 | URI des Zustandes |
| uri(path) | 4 | Renderer für Ablaufdarstellung mit URIs als Knotenbezeichnung |
| compact(node) | 2,3 | Renderer für kompakte Ablaufdarstellung auf einen Zustand angewendet |
| compact(path) | 4 | Renderer für kompakte Ablaufdarstellung |
| probability(path) | 4 | Wahrscheinlichkeit des aktuellen Ablaufes |
| pos(node) | 3 | Position in Ablauf (wegen parallelen Pfaden ggf. nicht exakt und `float`) |

**Tabelle 3.3:** Unterstützte Funktionen in Ergebnisspalten

fixes mit `CONTEXT <`*`Präfix`*`>`. Eine abgekürzte URI beginnt mit einem Doppelpunkt, `:BspURI` wird daher mit vorhangehendem `CONTEXT <http://www6.cs.fau.de/Bsp#>` zu `http://www6.cs.fau.de/Bsp#BspURI` erweitert.

Mit `USING RECORD`, `USING RECORDS`, `USING ALL RECORDS` und `WHERE` können Datenquellen angegeben werden, die neben dem reinen Workflow auch für Zeitangaben, Variablen und Entscheidungen beachtet werden. Mit `USING RECORD` kann eine gespeicherte Fallakte ausgewählt werden, deren Informationen komplett verwendet werden. `USING ALL RECORDS` benutzt Zeitangaben aller zum jeweiligen Workflow gespeicherten Zeitangaben für realitätsnähere Werte, mit `USING RECORDS` und einer Liste gewünschter Fallakten werden nur diese für Zeitangaben beachtet. Im `WHERE`-Abschnitt wird eine anonyme Fallakte formuliert, welche Variablen, Zeitangaben und Entscheidungen beinhalten kann. Variablen werden dabei nach dem Schema `Name = Wert` aufgelistet. Zeitangaben können für Knoten entweder über deren URI oder ggf. mehrdeutig über den Namen zugewiesen werden. Ohne Suffix werden Sekunden als Einheit einer Zeitangabe angenommen. Zeiten werden nach dem Schema `time(`Name oder URI`) = Zeit mit Suffix` ausgedrückt. Entscheidungen können für jeden Besuch des Verzweigungsknotens getroffen werden, aber auch für einzelne Besuche. Mögliche Zielknoten der Entscheidungen können dabei über deren URI oder auch ggf. mehrdeutig über den Namen angegeben werden. Um zu spezifizieren, für welche Besuche die Entscheidung gelten soll, kann optional ein Intervall oder eine einzelne Zahl in Klammern angegeben werden. Der Platzhalter `*` kann für halboffene Intervalle benutzt werden. Eine Entscheidung wird nach dem Schema

`Knoten -> Knoten` oder mit Einschränkungen `Knoten -> Knoten (von - bis)` formuliert. Mit verschiedenen Datenquellen spielt auch die Reihenfolge der Berücksichtigung von Werten eine Rolle:

1. Anonyme Fallakten (`WHERE`) sind die wichtigste Quelle für Informationen und ergänzen oder überschreiben alle anderen Angaben.
2. Gespeicherte Fallakten (`USING RECORD`) können Werte des Modells und Angaben von `USING (ALL) RECORDS` überschreiben.
3. Mehrere oder alle gespeicherten Fallakten eines Workflows, über `USING ALL RECORDS` oder `USING RECORDS` sind ausschließlich für durchschnittliche Zeitangaben relevant, Entscheidungen und Variablen dieser Fallakten werden ignoriert. Durchschnittliche Zeitangaben können nur die Zeiten des Workflow-Modells überschreiben.
4. Angaben im Workflow werden zuletzt betrachtet, falls keine Informationen in Fallakten vorliegen.

Abschließend ist auch ein Sortieren der Ergebnisse über `ORDER BY` möglich. Die hier verwendeten Felder oder Ausdrücke werden nicht für das Erkennen des Abfragetyps verwendet, müssen sich aber an den Typ halten. Ein aufsteigendes und absteigendes Sortieren ist mit `ASC` bzw. `DESC` möglich, mehrere Kriterien werden durch Komma getrennt.

### 3.7.2 Fallakten

Das Schema zum Anlegen einer Fallakte wird in Listing 3.7 dargestellt. Die Syntax der Angaben stimmt mit der anonymer Fallakten überein, d.h. mit `Variablenname = Wert`, `time(Knoten) = Zeit` und `Knoten -> Knoten` bzw. `Knoten -> Knoten (von - bis)` werden Variablen, Zeiten und Entscheidungen beschrieben. Soll eine Fallakte geändert bzw. ergänzt werden, muss eine `ALTER`-Operation nach dem Schema von Listing 3.8 verwendet werden. Ist der Name der Fallakte eindeutig über alle Workflows, so kann auf die Angabe des Workflows verzichtet werden.

Sollen Angaben entfernt werden, zeigen sich jedoch Nachteile der Syntax einer `ALTER`-Operation, denn als Ergänzung oder Überschreibung kann ein Entfernen schlecht formuliert werden. Daher müssen getrennte Löschoperationen für Variablen, Zeiten und Entscheidungen angeboten werden. Beim Löschen der Entscheidungen einer Verzweigung werden für eine einfachere Syntax alle entfernt. Neben den Elementen einer Fallakte kann auch eine komplette Fallakte entfernt werden. Für alle Löschoperationen gilt, dass

```
[ CONTEXT <URI> ]
  CREATE Record "Name der Fallakte"
                of "Name" oder <:URI> eines Workflows
  {
       Variablen, Zeitangaben, Entscheidungen, ...
  }
```

**Listing 3.7:** Anlegen einer Fallakte

```
[ CONTEXT <URI> ]
  ALTER Record "Name der Fallakte"
               [ of "Name" oder <:URI> eines Workflows ]
  {
       Variablen, Zeitangaben, Entscheidungen, ...
  }
```

**Listing 3.8:** Ändern einer Fallakte

unbedingt der Name der Fallakte angegeben werden muss und optional auch der jeweilige Workflow, falls der Name der Fallakte nicht über alle Workflows eindeutig sein sollte. Zur Wahl eines Workflows kann entweder dessen URI oder, falls eindeutig, der Name verwendet werden. Die Syntax der vier Löschoperationen wird in Listing 3.9 dargestellt.

Eine Rückgabe aller Informationen der Fallakte soll über die in Listing 3.10 beschriebene `RETRIEVE`-Operation möglich sein.

```
[ CONTEXT <URI> ]

  DELETE variable Variablenname in Record "Name" [ of Workflow ]

  DELETE time of Knoten in Record "Name" [ of Workflow ]

  DELETE decisions of Knoten in Record "Name" [ of Workflow ]

  DELETE Record "Name" [ of Workflow ]
```

**Listing 3.9:** Löschoperationen

```
[ CONTEXT <URI> ]
  RETRIEVE Record "Name" [ of Workflow ]
```

**Listing 3.10:** Rückgabe einer Fallakte

### 3.7.3 Workflows

Operationen zum Anlegen, Ändern und Löschen von Workflows sollten nur als Vereinfachung gesehen werden. Nützliche zusätzliche Informationen ausserhalb des Workflow-Modells, aber auch Knoten, Kanten, ganze Workflows oder in neuen Datentypen formulierte Fakten können weiterhin durch Zugriff auf die verwendete SPARQL-Datenbank hinzugefügt werden. Die nachfolgend genannten Operationen sollten daher als optional gesehen werden und es steht jedem Nutzer frei, einen Workflow in SPARQL- oder WQL-Operationen aufzubauen.

Zunächst müssen Workflows angelegt werden. Diese können einen optionalen Namen haben, eine URI zur eindeutigen Identifikation muss jedoch angegeben werden. Listing 3.11 zeigt dabei das allgemeine Schema zum Erstellen des Workflows.

```
[ CONTEXT <URI> ]
  CREATE WORKFLOW <URI> [ named "Name" ]
```

**Listing 3.11:** Erstellen eines Workflows

Neue Knoten und Kanten werden über die `ADD`-Operation hinzugefügt. Dabei kann entweder jeweils ein Element oder durch Komma getrennt eine größere Anzahl an Knoten und Kanten angegeben werden. Für einen Knoten muss dessen Typ vorangehend angegeben werden. Nachfolgende Definitionen von Knoten in der selben Abfrage haben ohne Angabe des Typs den selben wie der vorangehende. Neben dem Typ ist die URI des Knotens eine Pflichtangabe, der Name optional. Die Definition eines Knotens hat daher die Syntax `[ Typ ] <URI> [ ("Name") ]`. Mögliche Angaben für Typen sind `Simple`, `Initial`, `Final`, `Branch`, `Composite` und `Region`. Für Kanten werden die beteiligten Knoten entweder mit Namen oder URI angesprochen. Ist der angegebene Name für den Workflow nicht eindeutig, so werden alle möglichen Kanten hinzugefügt. Weiterhin kann für Kanten auch eine Bedingung gesetzt werden, deren Syntax in Kapitel 3.6 erläutert wird. Eine Kante wird daher nach dem Schema `edge(`*von, nach* `[, 'Bedingung' ] [, Wahrscheinlichkeit])` definiert. Das allgemeine Schema für das Hinzufügen von Knoten und Kanten wird in Listing 3.12 zusammengefasst.

```
[ CONTEXT <URI> ]
  ADD durch Komma getrennt Knoten und Kanten TO Workflow
```

**Listing 3.12:** Hinzufügen von Knoten und Kanten zu einem Workflow

Manche hinzugefügte Knoten können zusammengesetzte Zustände oder parallele Zustände sein. Welche Kindknoten oder parallelen Zweige diese beinhalten, wird in einer ADD-Operation nicht formuliert. Weiterhin kann es durch zusammengesetzte Zustände auch mehrere Start- oder Endzustände geben. Der Start- und Endzustand des Workflows ist damit auch nicht eindeutig gegeben. Eine von zwei mit SET benannten Operationen weist Zuständen ihren Elternknoten zu. Für einen Elternknoten kann nur dessen URI angegeben werden, um mehrdeutige Angaben ausschließen zu können. Ist die URI des angegebenen Elternknotens leer, wird der Workflow als „Elternknoten“ angenommen, womit Start- und Endzustände des Workflows eindeutig identifiziert werden können. Kindknoten werden mit URI oder (mehrdeutigem) Namen durch Komma getrennt aufgelistet. Listing 3.13 zeigt den Aufbau einer solchen Abfrage.

```
[ CONTEXT <URI> ]
  SET <Elternknoten> as parent for Kindknoten
```

**Listing 3.13:** Zuweisen von Eltern- bzw. Kindknoten

Die zweite SET-Operation soll Zuständen Zeiten und Kosten zuweisen. Das Format für beide ist gleich, Zeiten werden als Kosten mit Namen „time“ verwendet. Verschiedene Kosten werden nach dem Schema *Kostenbezeichnung*(*Knoten*) = *Wert* zugeordnet. Für den Knoten kann eine URI oder dessen Name angegeben werden. Da die Syntax der SET-Operation keine Angabe des Workflows erfordert und auch eine Zuweisung von Werten zu Knoten mehrerer Workflows erlaubt ist, muss für eine Knotenbezeichnung über den Namen um eine Angabe des jeweiligen Workflows mit of <URI> bzw. of "Workflowname" ergänzt werden. In einer in Listing 3.14 dargestellten SET-Operation können durch Komma getrennt beliebig viele Eigenschaften gesetzt werden.

```
[ CONTEXT <URI> ]
  SET Zeiten, Kosten
```

**Listing 3.14:** Zuweisen von Zeitangaben und Kosten

Abschließend wird das Löschen von Workflows oder deren Knoten, Kanten und Attribute beschrieben, was allgemein in den Listings 3.15, 3.16, 3.17 und 3.18 dargestellt wird. Bezeichner eines Workflows kann in allen Löschoperationen dessen URI oder Name sein. Knoten können generell über ihre URI oder den gegebenenfalls mehrdeutigen Namen angegeben werden. Soll eine Kante gelöscht werden, so muss deren Quell- und Zielknoten angegeben werden. Für das Löschen von zusammengesetzten Knoten und

ganzen Workflows werden alle untergeordneten „Kindknoten“ und nicht mehr sinnvollen Transitionen ebenso entfernt.

```
[ CONTEXT <URI> ]
  DELETE Workflow Workflow
```

**Listing 3.15:** Löschen eines Workflows

```
[ CONTEXT <URI> ]
  DELETE node Knoten of Workflow
```

**Listing 3.16:** Löschen eines Knotens

```
[ CONTEXT <URI> ]
  DELETE edge( Quellknoten , Zielknoten ) of Workflow
```

**Listing 3.17:** Löschen einer Transition

```
[ CONTEXT <URI> ]
  DELETE Kostenname of node <URI>
  DELETE Kostenname of node "Name" of Workflow
```

**Listing 3.18:** Löschen von Kosten und Zeiten

### 3.7.4 Grammatik

In Listing 3.19 wird die Grammatik der WQL in EBNF aufgeführt. Die Grammatik ist so aufgebaut, dass auch Erweiterungen möglich sind. Neue Abfragearten sollten in den Nichtterminalsymbolen `query` bzw. bei Ähnlichkeit zu bereits vorhandenen Arten in `add`, `alter`, `create`, `delete`, `retrieve` und `set` formuliert werden. Neue Inhalte von Fallakten können im Nichtterminalsymbol `assignment` eingefügt werden. Neue Datentypen für Kosten können als zusätzliche Punkte in `costdata` geschrieben werden.

Da Funktionen für Ergebnisspalten von `ESTIMATE`-Anfragen in der Grammatik abstrakt beschrieben werden, können die meisten Erweiterungen der Sprache ohne ein Ändern der Grammatik und des Parsers erfolgen.

```
query = [ context ], ( estimate | add | alter | create | delete
                     | retrieve | set ) ;
context = "CONTEXT", uri ;
add = addToWorkflow ;
```

```
alter = alterRecord ;
create = createRecord | createWorkflow ;
delete = deleteRecord | deleteTime | deleteDecision | deleteVariable
       | deleteWorkflow | deleteNode | deleteAttribute | deleteEdge ;
set = setParent | setValue ;

(* Estimate-Abfrage *)
estimate = "ESTIMATE", namedFields,
           "OF", workflow,
           [ "FROM", node ], [ "TO", node ],
           { "USING", ( records | record ) },
           [ "WHERE", '{', assignments, '}' ]
           [ "ORDER BY", orderby, { ',', orderby } ] ;

namedFields = namedField, { ',', namedField } ;
namedField = field, [ "AS", name ] ;
field = functioncall | name ;

records = "RECORDS", name, { ',', name } | "ALL RECORDS" ;
record = "RECORD", name ;
orderby = field, [ "DESC" | "ASC" ] ;

(* Fallakten *)
recordID = 'Record', name, [ 'of', workflow ] ;
createRecord = 'CREATE', recordID, '{', assignments, '}' ;
alterRecord = 'ALTER', recordID, '{', assignments, '}' ;
deleteRecord = 'DELETE', recordID ;
deleteTime = 'DELETE', 'time', 'of', node, 'in', recordID ;
deleteDecision = 'DELETE', 'decisions', 'of', node, 'in', recordID ;
deleteVariable = 'DELETE', 'variable', name_simple, 'in', recordID ;
retrieve = 'RETRIEVE', recordID ;

(* Attribute von (anonymen) Fallakten *)
assignment = setVar | setTime | setDecision ;
assignments = [ assignment, { ',', assignment } ] ;
setVar = name_simple, '=', float ;
setTime = "time", '(', node, ')', '=', time ;
time = float, [timesuffix] ;
timesuffix = "millisecond" | "second" | "minute" | "hour" | "day"
           | "ms" | "sec" | "s" | "min" | "m" | "h" | "d" ;
setDecision = node, "->", node,
             [ '[', decExpr, [ '-', decExpr ], ']' ] ;
```

```
decExpr = { number | '*' } ;

(* Workflows *)
createWorkflow = 'CREATE', 'WORKFLOW', uri, [ 'named', name ] ;
addToWorkflow = 'ADD', addElem, { ',', addElem }, 'TO', workflow ;
addElem = addEdge | addNode ;
addEdge = 'edge(', node, ',', node, [ ',', ''', string, ''' ],
          [ ',', float ] ;
addNode = name_simple, uri, [ '(', name, ')' ] ;
setParent = 'SET', setBez, 'as', 'parent', 'for', setBez, { ',',
   setBez } ;
setValue = 'SET', val, { ',', val } ;
val = name_simple, '(', setBez, ')', = costdata ;
deleteWorkflow = 'DELETE', 'Workflow', workflow ;
deleteNode = 'DELETE', 'node', setBez ;
deleteEdge = 'DELETE', addEdge ;
deleteAttribute = 'DELETE', name_simple, 'of', 'node', setBez ;
setBez = uri | name, 'of', workflow ;

functioncall = name_simple, '(', name, { ',', name }, ')' ;
name = '"', {alpha | digit | '.' | ' '}, '"' | nameSimple ;
name_simple = alpha, { alpha | digit } ;
node = uri | name ;
workflow = uri | name ;

costdata = float ;
float =  number, [ '.', number ] | '.', number ;
number = digit, {digit} ;
alpha = 'a', 'b', ..., 'z', 'A', ... 'Z' ;
digit = '0', '1', ..., '9' ;
uri = '<', ..., '>' ;
```

**Listing 3.19:** Grammatik der WQL in EBNF

## 3.8 Beispiele

Neben der Syntax ist auch die tatsächliche Verwendung einer Sprache interessant. In diesem Kapitel soll die WQL daher beispielhaft anhand des Ablaufs der Erstbehandlung eines Schlaganfalls nach Abbildung 3.13 vorgeführt werden, welches zunächst noch an das in Kapitel 3.3 vorgestellte Workflow-Modell angepasst werden muss.

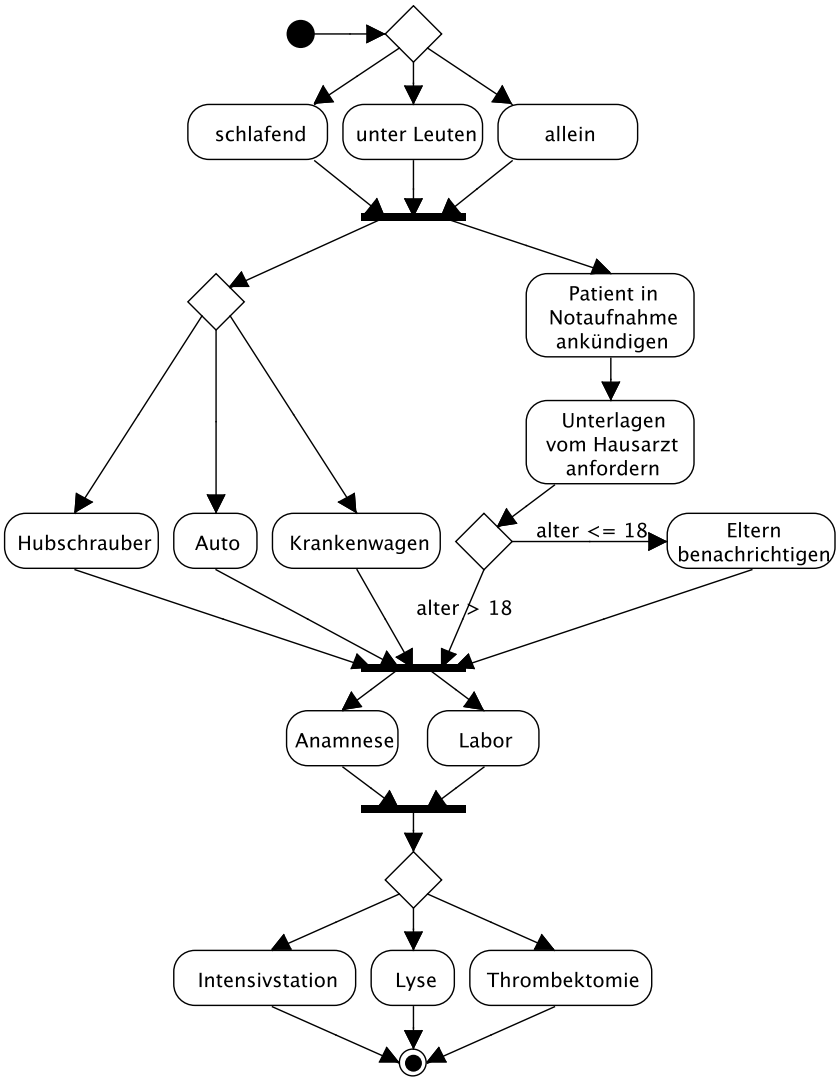



**Abbildung 3.13:** Workflow eines Schlaganfalls

### 3.8.1 Modellierung

Ob in diesem Workflow ein Patient mit Hubschrauber, Krankenwagen oder von einem Familienmitglied mit dem Auto ins Krankenhaus gebracht wird, kann in manchen Punkten, wie der Gesamtzeit, irrelevant sein. Die Information „Der Transport wurde in 30 Minuten durchgeführt.“ kann ohne explizite Angabe des Transportmittels vorliegen. Da Werte für Zeiten oder Kosten aggregiert angegeben werden können, sollen diesem Workflow nun passende Überzustände für Aggregationen hinzugefügt werden. Der bereits angesprochene Transport wurde dabei neben Aktionen während des Transportes, Diagnose, Behand-

lungsarten und möglichen Zuständen vor einem Notruf ausgewählt. Kapitel 3.3 setzt für parallele Abläufe einen übergeordneten Zustand mit mehreren parallelen Regionen voraus, auch dieser muss im ursprünglichen Modell ergänzt werden. Die angepasste Abbildung 3.14 soll daher in allen folgenden Abfragen eingesetzt werden.

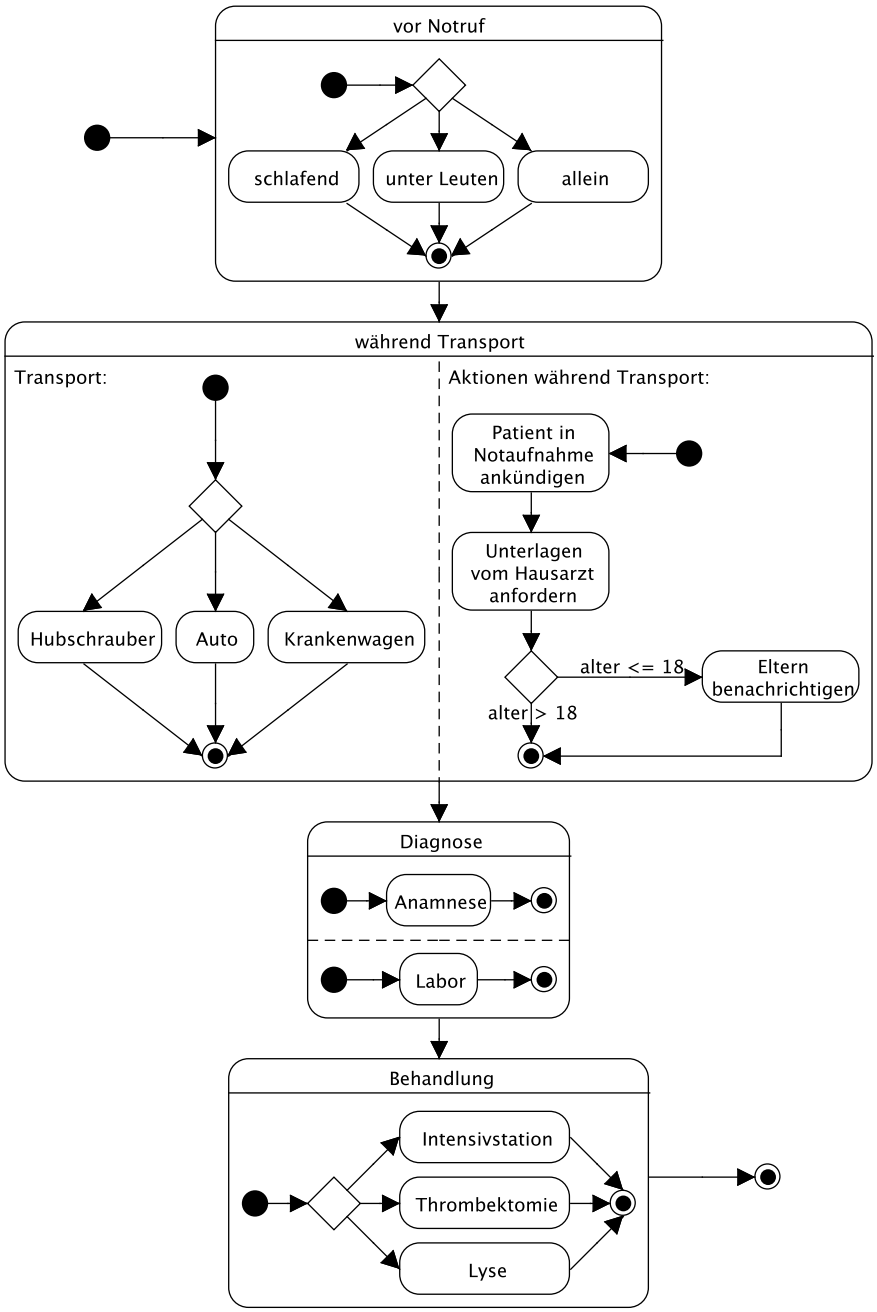



**Abbildung 3.14:** Verbesserter Workflow eines Schlaganfalls

### 3.8.2 Workflow anlegen

Auch wenn das Anlegen eines Workflows ein optionaler Teil der WQL-Implementierung ist, soll trotzdem die Syntax anhand des Beispielworkflows 3.14 gezeigt werden. Die hier vorgestellten Befehle sollen in jedem Fall allerdings nur eine Vereinfachung für Änderungen sein, weitere Informationen zu Knoten und Kanten können jederzeit beim Zugriff auf die verwendete SPARQL-Datenbank hinzugefügt werden. Für neue, ggf. komplizierte Datentypen kann daher grundsätzlich auf eine WQL-Syntax zum Anlegen, Ändern und Löschen verzichtet werden.

Alle URIs von Elementen des Workflows sollen mit dem Präfix `http://www6.cs.fau.de/prohta/workflows/Schlaganfall#` beginnen. Da es aufwändig wäre, diese URI laufend zu wiederholen und auch häufiges Copy&Paste nicht unbedingt sinnvoll ist, kann ein solches Präfix mit **CONTEXT** *<http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>* zu Beginn jeder Abfrage genannt werden. Eine abgekürzte URI beginnt mit einem Doppelpunkt, `:BspWorkflow` wird daher zu `http://www6.cs.fau.de/prohta/workflows/Schlaganfall#BspWorkflow` erweitert.

Zunächst muss ein Grundgerüst für einen Workflow angelegt werden - noch komplett ohne Zustände und Transitionen. Listing 3.20 zeigt dabei die Syntax zum Erstellen des Schlaganfall-Workflows mit der WQL, was übersetzt wird in eine SPARQL-Abfrage wie in Listing 3.21.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
CREATE WORKFLOW <:BspWorkflow> named "Schlaganfall"
```

**Listing 3.20:** Anlegen eines benannten Workflows mit der WQL

```
PREFIX w: <http://www6.cs.fau.de/prohta/workflow#>
PREFIX : <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>

INSERT DATA {
    :BspWorkflow    a           w:Workflow ;
                    w:name      "Schlaganfall" .
}
```

**Listing 3.21:** Anlegen eines benannten Workflows mit SPARQL

Als nächstes sollten Zustände und Transitionen hinzugefügt werden. Dies kann entweder einzeln oder durch Komma getrennt in einer `ADD`-Operation geschehen. Für Knoten muss

neben der URI und einem optionalen Namen auch der Typ angegeben werden, z.B. Initial, Final oder Simple. Wird für den nächsten Knoten kein Typ angegeben, wird der vorherige angenommen. Die Syntax dafür ist `Typ <URI>("Name")`. Die beteiligten Knoten einer Kante werden entweder mit `<URI>` oder `"Name"` angegeben. Mehrdeutige Ziele verursachen dabei mehrere Kanten. Namen können dabei jedoch nur pro Workflow mehrdeutig sein, d. h. haben die Workflows „Schlaganfall“ und „Blinddarmentzündung“ beide jeweils nur einmal einem Zustand den Namen „Diagnose“ zugeteilt, so ist dieser nicht mehrdeutig. In diesem Beispiel kann das Verhalten zwar nicht ausgenutzt werden, für größere Workflows kann jedoch Schreibarbeit gespart werden durch gezielten Einsatz uneindeutiger Namen. In der Syntax für Kanten sind auch optionale Bedingungsausdrücke vorgesehen. Formuliert werden Kanten durch `edge(<von>, <nach>)` bzw. mit Bedingung `edge(<von>, <nach>, 'Bedingung')` und/oder in Prozent angegebener Wahrscheinlichkeiten `edge(<von>, <nach>, 'Bedingung', Wahrscheinlichkeit)`. Listing 3.22 zeigt dabei einen einzelnen neuen Zustand, Listing 3.23 alle benötigten Knoten des Composite-Zustandes „vor Notruf“ mit den beteiligten Kanten, Listing 3.24 Zustände und Transitionen des kompletten parallelen Ablaufs der Diagnose und Listing 3.25 die korrekte Verwendung von Bedingungen in Transitionen.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ADD Initial <:Init> TO WORKFLOW <:BspWorkflow>
```

**Listing 3.22:** Hinzufügen eines einzelnen Zustandes

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ADD Composite <:PreCall> ("vor Notruf"),
    Initial <:Init1>,
    Branch <:stateBranch>,
    Simple <:sleeping> ("schlafend"),
           <:inCompany> ("unter Leuten"),
           <:alone> ("allein"),
    Final <:Final1>,
    edge(<:Init1>, <:stateBranch>), edge(<:stateBranch>, "schlafend"),
    edge(<:stateBranch>, "unter Leuten"), edge(<:stateBranch>, "allein"),
    edge("schlafend", <:Final1>), edge("unter Leuten", <:Final1>),
    edge("allein", <:Final1>)
TO WORKFLOW <:BspWorkflow>
```

**Listing 3.23:** Hinzufügen mehrerer Zustände und Transitionen

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ADD Composite <:diagnosis> ("Diagnose"),
    Region <:anamnesisThread>, <:labThread>,
    Initial <:Init4>, <:Init5>,
    Simple <:anamnesis> ("Anamnese"), <:lab> ("Labor"),
    Final <:Final4>, <:Final5>,
    edge(<:Init4>, "Anamnese"), edge("Anamnese", <:Final4>),
    edge(<:Init5>, "Labor"), edge("Labor", <:Final5>)
TO WORKFLOW <:BspWorkflow>
```

**Listing 3.24:** Hinzufügen der Zustände und Transitionen des Bereichs „Diagnose“

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ADD Branch <:edgeBranch>,
    Simple <:notifyParents> ("Eltern benachrichtigen"),
    Final <:Final3>,
    edge(<:edgeBranch>, "Eltern benachrichtigen", 'alter <= 18'),
    edge(<:edgeBranch>, <:Final3>, 'alter > 18')
TO WORKFLOW <:BspWorkflow>
```

**Listing 3.25:** Bedingungen in Kanten

Ein Workflow muss einen Start- und Endzustand haben. Formuliert werden können diese durch das Zuweisen eines leeren Parent-Knotens oder der URI des Workflows. Anhand des Typs wird entschieden, ob es sich um einen Initial- oder Final-Knoten handelt, Listing 3.26 zeigt dies für den Schlaganfall-Workflow. Das Zuweisen eines übergeordneten Zustandes wird auf die gleiche Weise durchgeführt, wobei der Kindzustand eine parallele Region oder ein Zustand sein darf, der Elternzustand jedoch nur eine parallele Region oder ein zusammengesetzter Zustand. Initial- und Final-Knoten für Regionen und zusammengesetzte Zustände werden anhand des Typs gewählt. Werden Regionen einem übergeordneten Zustand zugeordnet, so handelt es sich dabei um parallele Vorgänge. Listing 3.27 zeigt das Zuweisen der Kindzustände zum übergeordneten Zustand „vor Notruf“, Listing 3.28 setzt die parallen Vorgänge der Diagnose zusammen.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
SET <> as parent for <:Init>, <:Final>
```

**Listing 3.26:** Initial- und Endzustände des Workflows

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
SET <:PreCall> as parent for <:Init1>, <:stateBranch>, "schlafend",
                                "unter Leuten", "allein", <:Final1>
```

**Listing 3.27:** Kindknoten eines zusammengesetzten Zustandes

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
SET <:diagnosis> as parent for <:anamnesisThread>, <:labThread>

SET <:anamnesisThread> as parent for <:Init4>, "Anamnese", <:Final4>

SET <:labThread> as parent for <:Init5>, "Labor", <Final5>
```

**Listing 3.28:** Kindknoten eines parallelen Vorganges

Abschließend können Knoten, ebenfalls durch Komma getrennt, auch Eigenschaften wie Zeiten und Kosten durch eine SET-Operation zugewiesen werden. Zusätzliche Datentypen können dabei in die Sprache aufgenommen werden oder ausserhalb der WQL in SPARQL verwendet werden. Die Syntax dafür sieht wie ein Funktionsaufruf aus, dessen einziger Parameter ein Knoten ist und dem nach einem „=“ der Wert folgt. Der Knoten wird entweder als URI angegeben oder benannt mit zusätzlicher Angabe des zugehörigen Workflows. Mehrdeutige Namen sorgen, wie bereits bei Kanten, für mehrere Zuweisungen. Für Listing 3.29 wird als durchschnittliche Rettungszeit mit Hubschrauber 10 Minuten, mit Krankenwagen 15 Minuten und Auto 17 Minuten angenommen. Für den Hubschrauber sollen 200 € Rettungskosten angesetzt werden, für den Krankenwagen 80 €. Für den Hubschrauber soll ein externer Dienstleister verantwortlich sein, der pro Einsatz 150 € verlangt. Um die Kosten für den Dienstleister ebenso abbilden zu können, soll die zusätzliche Kostenart „RescueCorp“ verwendet werden.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
SET time(<:helicopter>) = 10 min, time(<:ambulance>) = 15 min,
    time(<:car>) = 17 min,
    cost("Hubschrauber" of <:BspWorkflow>) = 200,
    cost("Krankenwagen" of "Schlaganfall") = 80,
    cost(<:car>) = 0, RescueCorp(<:helicopter>) = 150
```

**Listing 3.29:** Setzen von Zeiten und verschiedene Kosten für Transportarten

Die WQL sieht auch Operationen zum Löschen von Workflows oder ihren Elementen vor. Listing 3.30 löscht den kompletten Workflow, Listing 3.31 nur den Knoten für einen Transport mit Auto und Listing 3.32 eine irrtümlich angelegte Transition von Hubschrauber zu Krankenwagen. Auch Attribute können gelöscht werden, die in Listing 3.29 angelegte Kostenart "RescueCorp für den Helikopter wird in Listing 3.33 wieder entfernt.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE Workflow <:BspWorkflow>
```

**Listing 3.30:** Löschen eines kompletten Workflows

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE node "Auto" of <:BspWorkflow>
```

**Listing 3.31:** Löschen eines Knotens

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE edge("Hubschrauber", <:ambulance>) of <:BspWorkflow>
```

**Listing 3.32:** Löschens einer Kante

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE RescueCorp of node <:helicopter>
```

**Listing 3.33:** Löschen von Kosten oder Zeiten

### 3.8.3 Fallakte

Sobald ein Workflow fertig angelegt ist, können Abfragen für diesen formuliert werden. Ein wichtiger Bestandteil von Abfragen, z.B. für Vergangenheitswerte oder Details eines konkreten Ablaufes, sind Fallakten, deren Syntax vorher gezeigt werden sollte. Anhand der Fallakte von *John Doe*, siehe Tabelle 3.4, wird nun die Syntax zum Anlegen einer Fallakte und deren Inhalten gezeigt.

Fallakten werden anhand ihres Namens identifiziert, diese soll „John Doe“ heißen. Zeiten, Entscheidungen oder Variablen können bereits beim Anlegen komplett hinzugefügt werden. In Listing 3.34 werden nur die Variablen und das Transportmittel gesetzt. Nach Ankunft des Krankenwagens ist die Rettungsdauer bekannt und der Fahrer teilt mit, dass der Patient bis zum Notruf allein war, weshalb mit Listing 3.35 die Fallakte um diese Informationen erweitert wird.

| **John Doe** | |
|---|---|
| Alter | 42 Jahre |
| Gewicht | 80 kg |
| vor Notruf | allein |
| Transportmittel | Krankenwagen (12 Minuten) |

**Tabelle 3.4:** Fallakte von John Doe

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
CREATE Record "John Doe" of <:BspWorkflow> {
    Alter = 42
    Gewicht = 80
    <:transportBranch> -> "Krankenwagen"
}
```

**Listing 3.34:** Anlegen der Fallakte

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ALTER Record "John Doe" of <:BspWorkflow> {
    time("Krankenwagen") = 12 min
    <:stateBranch> -> "allein"
}
```

**Listing 3.35:** Ändern der Fallakte

Variablen, Zeiten und Entscheidungen einer Fallakte können auch wieder gelöscht werden. In Listing 3.36 wird die Variable für das Alter des Patienten entfernt, in Listing 3.37 die Dauer der Rettung mit Krankenwagen. Entscheidungen sind beim Löschen ein Sonderfall, denn für einen Zustand kann es mehrere Entscheidungen geben, die nach der Anzahl der bisherigen Besuche des Zustands gesetzt werden. Um eine verwirrende Syntax zum Löschen zu vermeiden, werden nach Angabe eines Zustandes alle Entscheidungen dazu entfernt, d. h. mit `ALTER` müssen ggf. eigentlich nicht zu löschende Entscheidungen wieder hinzugefügt werden. Alle Entscheidungen zum Zustand vor dem Notruf werden in Listing 3.38 gelöscht, in diesem Beispiel handelt es sich bei allen Entscheidungen allerdings nur um eine. Eine komplette Fallakte „Jane Doe“ wird in Listing 3.39 entfernt.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE variable Alter in Record "John Doe" of <:BspWorkflow>
```

**Listing 3.36:** Löschen einer Variable

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE time of <:ambulance> in Record "John Doe" of <:BspWorkflow>
```

**Listing 3.37:** Löschen einer Zeitangabe

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE decisions of <:stateBranch> in Record "John Doe" of <:BspWorkflow>
```

**Listing 3.38:** Löschen von Entscheidungen

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
DELETE Record "Jane Doe" of <:BspWorkflow>
```

**Listing 3.39:** Löschen einer kompletten Fallakte

### 3.8.4 ESTIMATE

Der eigentliche Schwerpunkt der WQL kann nun nach dem Erstellen von Workflows und Fallakten sinnvoll verwendet werden. Dabei soll die Verwendung anonymer und gespeicherter Fallakten in allen vier in 3.7.1 aufgeführten Abfragetypen anhand des Schlaganfall-Workflows gezeigt werden. Den Anfang macht dabei in Listing 3.40 die Ermittlung der geschätzten Dauer und Kosten des Workflows für den Patienten „John Doe“ (siehe Tabelle 3.4).

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ESTIMATE time, cost
OF <:BspWorkflow>
USING RECORD "John Doe"
```

**Listing 3.40:** Geschätzte Dauer und Kosten (Abfragetyp 1)

Anonyme Fallakten sind auch bei vorhandenen tatsächlichen sinnvoll einsetzbar, z.B. um die Auswirkungen von Änderungen einzelner Parameter zu testen. Da für *John Doe* eine Thrombektomie als Behandlung vermutet wird, aber noch nicht durchgeführt

wurde, sollen die angefallenen Kosten des kompletten Workflows mit der Abfrage in Listing 3.41 ermittelt werden. Es wäre auch möglich, eine andere Entscheidung für das Transportmittel oder den Zustand vor dem Notruf anzugeben, denn Informationen in anonymen Fallakten werden vor allen anderen beachtet.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ESTIMATE cost
OF <:BspWorkflow>
USING RECORD "John Doe"
WHERE {
    <:therapyBranch> -> "Thrombektomie"
}
```

**Listing 3.41:** Geschätzte Gesamtkosten bei zusätzlicher Wahl der Behandlung (Abfragetyp 1)

Für die nächste Abfrage in Listing 3.42 ist eine Angabe in zusammengefassten Kosten und Zeiten nicht genau genug, diese sollen aufgeschlüsselt nach Knoten ermittelt werden. Relevant sollen diese Angaben nur bis zur Diagnose sein, da in der Fallakte auch noch keine gewählte Behandlungsmethode feststeht. Die Bezeichnung jedes Knotens soll deren URI sein. Da die Ausgangslage der Fallakten „Jane Doe“ und „Joe Average“ sehr ähnlich ist, sollen deren Angaben auch in dieser Abfrage nützlich sein und genauere Zeitschätzungen ermöglichen.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ESTIMATE uri(node), time, cost
OF <:BspWorkflow>
FROM <:Init> TO "Diagnose"
USING RECORD "John Doe"
USING RECORDS "Jane Doe", "Joe Average"
```

**Listing 3.42:** Informationen zu Zuständen bis zur Diagnose (Abfragetyp 2)

Da es im Modell drei Behandlungsmethoden gibt und bis auf deren Auswahl alle Entscheidungen bereits feststehen, sollen nun die Gesamtkosten und deren komplette Abläufe zurückgegeben werden. Um die Relevanz der Ergebnisse einordnen zu können, sollen auch die Wahrscheinlichkeiten der gefundenen Pfade im Ergebnis enthalten sein. Da Zeiten in dieser Abfrage nicht benötigt und Kosten in Fallakten nicht aufgeführt werden, kann auf ähnliche Fallakten in der Formulierung in Listing 3.43 verzichtet werden.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ESTIMATE path, probability(path), cost
OF <:BspWorkflow>
USING RECORD "John Doe"
```

**Listing 3.43:** Abfrage der Kosten aller möglichen Pfade (Abfragetyp 4)

Abschließend soll mit der Abfrage in Listing 3.44 der wahrscheinlichste Ablauf des Workflows und der durchschnittliche Zeitverbrauch der einzelnen Zustände ermittelt werden. Für die Zeiten sollen allerdings nicht die im Modell angenommenen Werte eingesetzt werden, sondern alle für den Workflow Schlaganfall vorhandenen Fallakten.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>
ESTIMATE pos(node), node, time
OF <:BspWorkflow>
USING ALL RECORDS
```

**Listing 3.44:** Abfrage des wahrscheinlichsten Pfades (Abfragetyp 3)

# 4 Implementierung

Die WQL wurde in *Python* programmiert und in der Version 2.7.2 getestet. Als SPARQL-Server wurde die Version 0.2.1-incubating von *Fuseki*, ein Teil des Apache Jena Projekts, gewählt. Die Dokumentation wird mit Version 3.0.1 von *epydoc* aus dem Quellcode der WQL generiert, dessen Funktionen mit sog. Python Docstrings beschrieben wurden.

In der Implementierung unterstützt werden alle vorgestellten `ESTIMATE`-Anfragen sowie Einfüge-, Änderungs- und Löschoperationen von Fallakten. Es soll nun ein kurzer Überblick über die nötigen Schritte des Einlesens von Abfragen bis zum Erhalt der Ergebnisse gegeben werden.

## 4.1 Interpretieren der Abfrage

Zunächst muss eine gestellte Abfrage eingelesen werden. Zum Parsen wurde dabei *PyPEG* in der Version 1.5 gewählt. Sobald der Syntaxbaum zurückgegeben wurde, wird erst die geforderte Operation, z.B. `ESTIMATE` oder `CREATE`, ermittelt. Für manche Einfüge-, Änderungs- und Löschoperationen muss anschließend deren Art weiter verfeinert werden auf Operationen für Workflows oder Fallakten und bei Löschoperationen noch auf die vielen löschbaren Elemente.

Der Typ von `ESTIMATE`-Abfragen wird anhand der angegebenen Ergebnisspalten, wie in Abschnitt 3.7.1 beschrieben, erkannt. Für Plausibilitätsprüfungen sollte dies bereits vor dem Laden von Workflows und der Pfadverarbeitung geschehen. Weil die Art der jeweiligen `ESTIMATE`-Abfrage jedoch erst bei der Ermittlung des Endergebnisses nach der Pfadverarbeitung relevant ist, könnte dies auch später oder parallel geschehen.

Die folgenden Abschnitte sind ausschließlich für `ESTIMATE`-Abfragen relevant.

## 4.2 Workflow laden

Für die Pfadverarbeitung müssen Workflows zunächst durch SPARQL-Abfragen angefordert werden. Stellvertretend für einen beliebigen Workflow soll in allen Listings der aus Kapitel 3.8 bekannte Workflow `:BspWorkflow` verwendet werden.

Den Anfang bildet mit Listing 4.1 die Suche nach allen Knoten eines Workflows. Anschließend werden in Listing 4.2 alle Kanten mit ihren eventuell vorhandenen Bedingungen und Wahrscheinlichkeiten ermittelt. Im Anschluss müssen wie in Listing 4.3 die Beziehung zwischen Eltern- und Kindknoten ermittelt werden, d.h. Unterzustände von zusammengesetzten Knoten und parallelen Abläufen.

```
PREFIX fsm: <http://www6.cs.fau.de/prohta/fsm#>
PREFIX   w: <http://www6.cs.fau.de/prohta/workflow#>
PREFIX : <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>

SELECT ?uri ?type ?name
WHERE {
    :BspWorkflow    w:hasStateMachineElement    ?uri .
    ?uri            a                           ?type .
    OPTIONAL { ?uri fsm:StateName ?name . }
}
```

**Listing 4.1:** Alle Knoten eines Workflows anfordern

```
PREFIX  fd: <http://www6.cs.fau.de/prohta/fsmdata#>
PREFIX fsm: <http://www6.cs.fau.de/prohta/fsm#>
PREFIX   w: <http://www6.cs.fau.de/prohta/workflow#>
PREFIX : <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>

SELECT ?source ?dest ?condition ?probability
WHERE {
    :BspWorkflow    w:hasStateMachineElement    ?source, ?dest.
    ?edge           a                           fsm:Transition;
                    fsm:Source                  ?source ;
                    fsm:Target                  ?dest .
    OPTIONAL { ?edge    fsm:Condition   ?condition . }
```

```
    OPTIONAL {
        ?edge       fd:probability      ?prob .
        ?prob       a                   fd:Probability ;
                    fd:probValue        ?probability .
    }
}
```

**Listing 4.2:** Alle Kanten eines Workflows ermitteln

```
PREFIX fsm: <http://www6.cs.fau.de/prohta/fsm#>
PREFIX   w: <http://www6.cs.fau.de/prohta/workflow#>
PREFIX : <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>

SELECT ?parent ?child
WHERE {
    :BspWorkflow   w:hasStateMachineElement   ?parent, ?child .
    ?parent        fsm:hasStateMachineElement ?child .
}
```

**Listing 4.3:** Unterzustände von zusammengesetzten Zuständen ermitteln

Für Zeitangaben und Kosten können Werte entweder als Zahl oder URI einer `DataStructure` gegeben sein. Da für neue Datenstrukturen ohnehin eigene Abfragen erstellt werden, können beide Fälle bei der Rückgabe der gebundenen Variablen `?time` bzw. `?value` der Listings 4.4 und 4.5 unterschieden werden. Diese Abfragen wurden nicht in Listing 4.1 zur Abfrage von Knoten integriert, da für beide ggf. durch unterschiedliche Datentypen und Namen Mehrfachangaben gemacht werden können, die durch häufige Nennung aller weiteren Angaben von Knoten das Ergebnis der Abfrage stark aufblähen würden.

```
PREFIX  fd: <http://www6.cs.fau.de/prohta/fsmdata#>
PREFIX fsm: <http://www6.cs.fau.de/prohta/fsm#>
PREFIX   w: <http://www6.cs.fau.de/prohta/workflow#>
PREFIX : <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>

SELECT ?node ?time
WHERE {
    :BspWorkflow    w:hasStateMachineElement    ?node .
    ?node           fd:time                     ?time .
}
```

**Listing 4.4:** Zeitangaben von Knoten anfordern

```
PREFIX c: <http://www6.cs.fau.de/prohta/cost#>
PREFIX w: <http://www6.cs.fau.de/prohta/workflow#>
PREFIX : <http://www6.cs.fau.de/prohta/workflows/Schlaganfall#>

SELECT ?node ?cost ?value
WHERE {
    :BspWorkflow    w:hasStateMachineElement    ?node .
    ?node           c:cost                      ?costObj .
    ?costObj        c:name                      ?cost ;
                    c:value                     ?value .
}
```

**Listing 4.5:** Kosten von Knoten anfordern

Ist der Ergebniswert von Zeitangaben oder Kosten eine URI, müssen ggf. beim Laden oder während der Ermittlung des Endergebnisses weitere hier nicht aufgeführte Abfragen für deren Werte gestellt werden.

## 4.3 Pfadverarbeitung

Die Pfadverarbeitung arbeitet exakt wie in Abschnitt 3.5 beschrieben. Um Bedingungen an Kanten auswerten zu können, müssen zunächst ggf. noch Fallakten geladen werden. Die Bearbeitungsreihenfolge zur Auswahl von ausgehenden Kanten kann bei der Verwendung der WQL wichtig sein, weshalb sie in Listing 4.6 gezeigt wird.

```
def suitableEdges(wp, p, node):
    """
    Return all possible outgoing edges of node.

    @param wp: Current L{WorkflowProcessor}.
    @param p: Incoming probability of node.
    @type p: number
    @param node: Node which edges are checked.
    @type node: L{Node}
    @return: All possible outgoing edges.
    @rtype: [(Edge, probability)]
    """

    result = []
    summe = 0.0

    for k in node.outgoing:
        # Prüfen, ob Verwendung dieser Kante erlaubt:

        if not k.reachEnd:
            # Kante kann das Ziel nicht erreichen
            continue

        if node.hasDecision(wp) and not node.isDecision(k, wp):
            # Entscheidung auf andere Kante gefallen
            continue

        # Bedingung auswerten
        if k.test(wp).isFalse():
            # ausgewertete Bedingung schließt Knoten aus
            continue

        summe = summe + k.p

```

```
35          # Wahrscheinlichkeit für mögliche neue Abläufe
               ermitteln
36          neu_p = p * k.p
37          if neu_p > wp.p:
38              result.append((k, neu_p))
39
40      # Wahrscheinlichkeit ausgehender Kanten auf 1 normieren
41      if abs(1.0 - summe) > max_round_error:
42          l = map(lambda x: (x[0], x[1] / summe), result)
43          result = l
44
45      return sorted(result, lambda x, y: -cmp(x[1], y[1]))
```

**Listing 4.6:** Zeitangaben von Knoten anfordern

Für Kanten wird also zunächst geprüft, ob das Ziel grundsätzlich erreicht werden kann, ob die Entscheidung auf die jeweilige Kante gefallen ist und abschließend, ob die Bedingung der Kante erfüllt wurde. Entscheidungen können daher nicht widersprüchlich zu Bedingungen angegeben werden. Die Unterteilung in `hasDecision` und `isDecision` erlaubt der Pfadverarbeitung gründsätzlich auch mehrere „Entscheidungen“, was für zukünftige Versionen der WQL z.B. für eine Auswahl an möglichen Entscheidungen genutzt werden könnte. Der Parameter `wp` wird für die bisherigen Besuche des Knotens benötigt. Da durch die drei Kriterien Kanten aussortiert werden können, die Wahrscheinlichkeit für übrig gebliebene allerdings 100% betragen sollte, wird abschließend auf diesen Wert normiert.

## 4.4 Ergebnis

Zuletzt muss aus den vorher ermittelten möglichen Abläufen ein Ergebnis errechnet werden. Die vier Anfragetypen werden dabei unterschiedlich behandelt:

1. Durchschnittliche Zeiten/Kosten aller Pfade:
   Alle besuchten Zustände werden nach Wahrscheinlichkeiten und Auftreten in Abläufen gewichtet und mit den resultierenden Gewichten werden Kosten und Zeiten angemessen summiert und zurückgegeben. Bei Parallelität bedeutet „angemessen“, dass für Zeitangaben das Maximum paralleler Vorgänge verwendet wird, Kosten werden ohne diese Besonderheit summiert.

2. Zustände über deren durchschnittliche Zeiten/Kosten über alle Pfade:
   Exakt wie in der ersten Art wird zunächst gewichtet, eine abschließende Summe wird jedoch nicht gebildet, sondern alle Funktionen auf das „Zwischenergebnis“ angewendet.
3. Zustände und deren Zeiten/Kosten des wahrscheinlichsten Pfades:
   Nur der wahrscheinlichste Ablauf wird gewählt und Funktionen beziehen sich generell auf einen einzelnen Punkt Zustand des Ablaufes, d. h. nur Informationen zusammengesetzter Zustände werden aggregiert. Ausser explizit ausgeblendet wird in dieser Anfrageart auch eine Position zurückgegeben. Die Position ist für einfache Abläufe ohne Parallelität eine fortlaufende Nummer, für Abläufe mit Parallelität erscheint sie jedoch heikel. Um diesen Punkt unterstützen zu können wird daher eine Gleitkommazahl zurückgegeben, bei der parallele Vorgänge die fortlaufende Nummer um 1 erhöhen und alle Kindzustände zwischen dem aktuellen und nächsten Zählerstand anhand ihrer geschätzten Zeitwerte angesiedelt werden.
4. Kosten und Zeiten kompletter Pfade: In diesem Anfragetyp werden Informationen kompletter Pfade ermittelt. Es wird daher nicht gewichtet, stattdessen werden für jeden einzeln betrachteten Ablauf aggregierten Zeiten und Kosten ermittelt. Offensichtlich ist eine Rückgabe der beteiligten Knoten schwierig, wenn ein Ablauf eine „Ergebniszeile“ sein soll, daher werden Pfade als eine einzelne Zeichenkette zusammengefasst, wofür mehrere Renderfunktionen bereitgestellt sind.

Abschließend werden die Ergebnisse, falls gewünscht, sortiert.

# 5 Evaluation

In Kapitel 3.1 wurden 10 Anforderungen an die WQL, deren Implementierung und das Workflow-Modell gestellt, deren Erfüllung nun untersucht werden soll.

## 5.1 Zyklen

Gibt es Zyklen in Workflows, sollen Abfragen auf Pfade dennoch möglich sein und deren Verarbeitung darf keine Endlosschleifen verursachen. Da ein Zyklus ohne oder mit schlecht gewählten Bedingungen an Transitionen zu einer unendlichen Anzahl an Pfaden führen kann, verlangt diese Anforderung nicht, dass alle möglichen Pfade gefunden werden müssen.

Die in Kapitel 3.5 vorgestellte und implementierte Heuristik erfüllt diese Anforderung, indem bei der Verarbeitung nur die wahrscheinlichsten Pfade gesucht werden sollen. Dafür wird konkret eine nach Wahrscheinlichkeit sortierte Warteschlange mit Abläufen verwaltet, für die ein Pfad an einer Verzweigung kopiert und alle möglichen Ausgänge erneut in die Warteschlange eingereiht werden. Die mögliche Anzahl an Pfaden in der Warteschlange ist begrenzt und ergibt sich aus einer maximalen Anzahl berücksichtigter Pfade, von denen bereits gefundene Ergebnisse und Fehlschläge abgezogen werden. Weiterhin gibt es auch eine Minimalwahrscheinlichkeit von Abläufen, die aufgrund der nach oben begrenzten Anzahl untersuchter Pfade beliebig klein gesetzt werden kann. Bis die Warteschlange leer ist, wird jeweils der wahrscheinlichste Ablauf aus der Warteschlange heraus genommen und verarbeitet. Auch wenn Workflows konstruiert werden können, für welche die Heuristik versagt und keine Ergebnisse gefunden werden, so wird durch die Maximalgrenze für fehlgeschlagene, gefundene oder in die Warteschlange eingereihte Abläufe sichergestellt, dass die Suche nach Pfaden im schlimmsten Fall ohne Ergebnisse endet, aber tatsächlich ohne Endlosschleifen dieses Ergebnis liefert. Im Regelfall und bei hinreichend großer Grenze zu untersuchender Abläufe werden die wahrscheinlichsten Pfade gefunden.

Die Verarbeitung von Abfragen über Workflows mit Zyklen ist daher möglich und wird ohne Endlosschleifen durchgeführt, die Anforderung wird demnach erfüllt.

## 5.2 Parallelität

Parallelität in Workflows ist kein vernachlässigbarer Ausnahmefall und soll daher sowohl modelliert und bei der Pfadverarbeitung von Abfragen unterstützt werden. Dabei wurden zwei Arten von Parallelität gefordert:

1. Sofortiger Abbruch der Parallelität, sobald der erste Faden sein Ziel erreicht.
2. Warten auf den Abschluss aller parallelen Zweige.

Transitionen von Unterzuständen eines zusammengesetzten Zustandes oder parallelen Ablaufs zu externen Knoten erlauben einen vorzeitigen Abbruch der Parallelität. Das erste Kriterium wird daher unterstützt, indem anstatt eines Endzustandes solche vorzeitigen Abbrüche verwendet werden. Das zweite Kriterium beschreibt das Standardvorgehen der Pfadverarbeitung und wird ebenso erfüllt durch die in Kapitel 3.5 vorgestellte und implementierte Heuristik.

## 5.3 Skalierungsstufen

Angaben zu Zeiten und Kosten können an verschiedenen Stellen getätigt werden. So sieht das in Kapitel 3.3 beschriebene Workflow-Modell Angaben zu Zeiten und Kosten für *alle* Typen von Zuständen, daher auch zusammengesetzte vor. In (anonymen) Fallakten wird zur Angabe von Zeiten ebenso keine spezielle Art von Knoten gefordert, womit auch hier alle erlaubt sind. Die Syntax für die Zuweisung von Attributen (Zeiten, Kosten) mit einer `SET`-Operation in der WQL erlaubt gleichermaßen alle Knoten als zu beschreibendes Ziel. Die Anfrageverarbeitung verwendet generell die für einen Knoten genauesten Angaben, d. h. liegt ein aggregierter Wert vor, so wird dieser verwendet.

Da das Workflow-Modell, die Sprache und die Implementierung Angaben von Informationen unterschiedlicher Skalierungsstufen unterstützen, wird auch dieser Anforderung nachgekommen.

## 5.4 Datenqualität

Für jede `ESTIMATE`-Abfrage wird die Summe der Wahrscheinlichkeiten aller gefundenen Pfade gebildet und als Teil des Ergebnisses zurückgegeben. Werden die wichtigsten oder alle möglichen Abläufe gefunden, nähert sich dieser Wert stark der 1. Ein wesentlich geringerer Wert der Summe spricht für fehlende Entscheidungen und eine zu hohe Anzahl an Möglichkeiten. Die Qualität der gefundenen Schätzwerte für Zeiten und Kosten kann mit dieser Summe bewertet werden. Da jedoch die Anzahl gefundener Pfade ein besserer Indikator für eine zu geringe Anzahl an Angaben ist, wird auch diese zurückgegeben.

Soll stattdessen ermittelt werden, ob für alle Knoten ausreichend Zeitangaben und Kosten annotiert wurden, so kann man sich über einen Zugriff auf die Definition oder eine `ESTIMATE`-Abfrage für einzelne Zustände ohne Angabe von Fallakten und damit ohne weitere Zeitangaben manuell einen Überblick verschaffen. Da jederzeit neue benannte Kostenarten für Workflows formuliert werden können, wird bei der Verarbeitung nicht verlangt, dass diese für alle Zustände spezifiziert wurden, sondern ohne Angabe generell 0 angenommen. Auch Zeitangaben sind optional, da diese für Instanzen von Workflows ohnehin in Fallakten gegeben werden können. Wie bei Kosten wird auch hier daher ohne Angabe 0 angenommen. Nicht vorhandene Annotationen sind daher keine Fehler.

Die Kennzahlen der Summe von Wahrscheinlichkeiten und der Anzahl an Workflows zusammen erlauben eine Beurteilung der Vollständigkeit der für Entscheidungen relevanten Eingaben und werden in Ergebnissen jeder `ESTIMATE`-Abfrage zur Verfügung gestellt, womit die Anforderung erfüllt ist.

## 5.5 Entscheidungen an Verzweigungen

Für Entscheidungen in Workflows wird gefordert, dass diese auf einfache Weise formuliert werden können und mehrdeutige Ergebnisse bzw. mehrere Pfade nach ihrer Wahrscheinlichkeit für Aggregationen gewichtet werden. Entscheidungen werden dafür entweder durch eine direkte Angabe in einer (anonymen) Fallakte, gegebenenfalls für unterschiedliches Auftreten des Bedingungsknotens mit unterschiedlichen Ergebnissen, oder durch Auswertung von Bedingungen von Transitionen gefällt. Im Workflow-Modell ist für Transitionen eine optionale Angabe der Wahrscheinlichkeit vorgesehen, welche zur Auswahl der wahrscheinlichsten Pfade und für die Gewichtung hergenommen wird. Bleiben an einer Verzweigung Transitionen mit einer Summe der Wahrscheinlichkeiten $\neq 100\%$ übrig, so wird auf 100% normiert.

Abbildung 5.1 soll exemplarisch die Möglichkeiten der WQL bei der Verarbeitung von Pfaden zeigen. Nicht eingezeichnet, sondern nachfolgend aufgelistet, sind dabei die Wahrscheinlichkeiten für Transitionen an Verzweigungen:

| | | |
|---|---|---|
| $B \to A$: 25% | $E \to F$: 70% | $H \to I$: 50% |
| $B \to C$: 50% | $E \to G$: 30% | $H \to J$: 50% |
| $B \to D$: 25% | | |

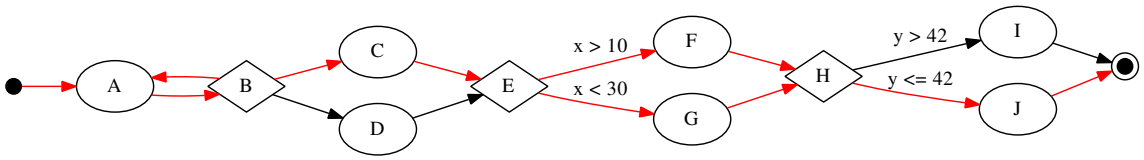



**Abbildung 5.1:** Entscheidungen in Workflows

Durch die Verzweigung B gibt es offensichtlich einen Zyklus. In der Abfrage soll festgelegt sein, dass die ersten drei Entscheidungen bei B ein Zurücklaufen zu A sein sollen und anschließend Knoten C das Ziel ist. Die Variablen werden mit $x = 20$ und $y = 7$ angegeben, womit es keine feste Entscheidung bei Knoten E gibt und der Pfad nach der Verzweigung H eindeutig ist. Listing 5.1 zeigt die Syntax dieser Abfrage und Tabelle 5.1 deren Ergebnis. Da die Syntax, das Workflow-Modell und die Implementierung Entscheidungen unterstützen, wird die Anforderung als erfüllt angesehen.

```
CONTEXT <http://www6.cs.fau.de/prohta/workflows/evaluation#>
ESTIMATE path, probability(path)
OF <:Entscheidungen>
WHERE {
    "B" -> "A" (1-3)
    "B" -> "C" (4)
    x = 20
    y = 7
}
```

**Listing 5.1:** Abfrage von Pfaden mit Angabe von Entscheidungen

| path | probability(path) |
|---|---|
| $\bullet ABABABABCEFHJ\odot$ | 70% |
| $\bullet ABABABABCEGHJ\odot$ | 30% |

**Tabelle 5.1:** Ergebnis der Abfrage

## 5.6 Erweiterbarkeit der Modelle / Datentypen

Gefordert ist, dass im Nachhinein neue Datentypen modelliert und zur WQL hinzugefügt werden können. Für die Zuweisung von Kosten und Zeiten wird eine Zahl oder eine Unterklasse von `ds:DataStructure` (Listing A.2) gewählt, d.h. es ist von Anfang an vorgesehen, noch nicht bekannte Typen ebenso zu verwenden. Zumindest für Kosten hält sich der Aufwand zur Änderung der Implementierung in Grenzen, da diese erst ausgewertet werden, sobald Abläufe und Knoten gewichtet sind. Zeitangaben dagegen spielen bei der Ermittlung der Pfade eine Rolle und müssen entweder leicht implementierbar einen fixen Zahlenwert anstatt eines komplexeren Datentyps zurückliefern oder es muss tatsächlich die Pfadermittlung geändert werden. Zeiten und Kosten müssen gleichermaßen beim initialen Laden des Workflows bei `ESTIMATE`-Abfragen beachtet und komplett mitgeladen werden oder z.B. für größere Datentypen einen Platzhalter liefern, für den erst bei Bedarf tatsächlich Werte ermittelt und verarbeitet werden. Eine Änderung der Sprache ist optional, da ein Zugriff auf Workflows über SPARQL generell möglich sein sollte.

Die Erfüllung dieser Anforderung kann anhand Listing A.4 zur Definition von Histogrammen gesehen werden. Neben dem Modell müssen Änderungen nur beim Laden des Workflows und bei der Verarbeitung nach Ermitteln der gewichteten Pfade oder Knoten erfolgen, für Zeitangaben sollen für die Pfadverarbeitung fixe Werte zurückgeliefert werden. Der Aufwand für die Programmierung der Operationen des Datentyps soll vernachlässigt werden, da er nichts mit der Erweiterbarkeit, sondern ausschließlich mit dem Datentyp zu tun hat.

## 5.7 Erweiterbarkeit der Sprache

Für die Erweiterbarkeit der Sprache wurden drei Kriterien gefordert:

1. Erweiterbarkeit der Sprache um neue Datentypen
2. Erweiterbarkeit der Sprache um neue Funktionen
3. Erweiterbarkeit der Ausdrücke für Bedingungen von Transitionen um neue Funktionen

Die Grammatik der WQL, siehe Listing 3.19, ist so strukturiert, dass Erweiterungen an sinnvollen Stellen nicht nur möglich, sondern vorgesehen sind. Datentypen werden im Nichtterminalsymbol `costdata` zusammengefasst und an allen Stellen mit Angaben von Kosten verwendet. Für Fallakten können im Nichtterminalsymbol `assignment` zusätzliche Zuweisungsarten neben Zeitangaben, Variablen und Entscheidungen definiert werden. Komplett neue Abfragearten können zwar nicht vorgesehen sein, in der Grammatik lassen sich diese jedoch in den Symbolen `query`, `add`, `alter`, `create`, `delete`, `retrieve` oder `set` einordnen. Änderungen der Grammatik erfordern natürlich auch kleine Änderungen am Parser und der Verarbeitung, diese halten sich jedoch in Grenzen, womit das erste Kriterium abgehakt werden kann.

Die Sprache um neue Funktionen erweitern ist leichter, denn die Grammatik sieht für Funktionen bereits eine allgemeinere Form vor. Angegeben werden muss bei Funktionen deren Name, Anzahl an Parametern und ob sie sich auf Pfade, Knoten oder sonstige Werte beziehen, womit die Art der jeweiligen `ESTIMATE`-Abfrage erkannt werden soll und widersprüchliche Angaben zu Fehlermeldungen statt Fehlern führen sollen. Auch für bereits in Tabelle 3.3 definierte Funktionen wurden diese Angaben gemacht. Ausgeführt werden sie erst nach der Pfadverarbeitung auf aggregierten Werten, vorher muss nur deren Definition bekannt sein, d. h. der Aufwand zur Erstellung neuer Funktionen in der WQL ist begrenzt und das zweite Kriterium erfüllt.

Auch die in Tabelle 3.2 vorgestellten Funktionen für Bedingungen von Transitionen sind leicht erweiterbar. Nötig ist auch hier die Angabe des Namens, der Funktion und der Anzahl an Parametern. Daneben gibt es allerdings auch ein Flag dafür, dass bei gleichen Eingaben das gleiche Ergebnis herauskommen muss, d. h. es kann einen Cache geben bzw. könnten auf der anderen Seite auch ein Zufallsgenerator oder veränderliche externe Informationen für Bedingungen relevant sein. Weiterhin gibt es ein Flag, mit dem geregelt wird, ob eine aufgerufene Funktion Einfluss auf die Pfadverarbeitung nehmen darf, was für die Funktion `occurence()` bereits sehr wichtig war, aber besonders Spielraum für

mehr Einfluss auf die Pfadverarbeitung bietet. Es kann daher das dritte Kriterium und somit auch die Anforderung eingehalten werden.

## 5.8 Anfragearten

Es wurden vier Anfragearten gefordert, deren Unterstützung in Kapitel 3.8 anhand der angegebenen Listings verdeutlicht werden soll:

1. Komplett aggregierte Gesamtzeiten und -kosten über alle möglichen Abläufe. In Listing 3.40 (S. 65) werden diese unter Einbeziehung einer Fallakte ermittelt.
2. Zeiten und Kosten für alle beteiligten Zustände. In Listing 3.42 (S. 66) wird dabei die URI der beteiligten Zustände angefordert, welche diesen Abfragetyp erforderlich macht.
3. Wahrscheinlichster Ablauf. In Listing 3.44 (S. 67) wird durch die geforderte Position von Zuständen in Abläufen dieser Abfragetyp von Punkt 2 unterschieden.
4. Wahrscheinlichste Abläufe. Listing 3.43 (S. 67) soll die wahrscheinlichsten Pfade eines Workflows sowie deren Wahrscheinlichkeit ermitteln.

Alle vier Anfragen werden in einer `ESTIMATE`-Operation zusammengefasst und anhand der geforderten Ergebnisspalten wird der Anfragetyp ermittelt, die Anforderung wird somit erfüllt.

## 5.9 Einfüge-, Änderungs- und Löschoperationen

Es sollen Operationen zum Einfügen, Ändern und Löschen von Fallakten und optional Workflows bereitgestellt werden. Um nicht viele bereits in den Kapiteln 3.7 und 3.8 aufgeführte Listings erneut angeben zu müssen, soll in Tabelle 5.2 daher eine Übersicht über Listings unterstützter Funktionen und passender Beispiele gegeben werden.

**Fallakten**

| *Beschreibung* | *Operation* | *Listing Schema* | *Listings Beispiele* |
|---|---|---|---|
| anlegen | `CREATE` | 3.7 (S. 51) | 3.34 (S. 64) |
| ändern | `ALTER` | 3.8 (S. 51) | 3.35 (S. 64) |
| löschen | `DELETE Record` | 3.9 (S. 51) | 3.39 (S. 65) |
| löschen (Variablen) | `DELETE variable` | 3.9 (S. 51) | 3.36 (S. 65) |
| löschen (Zeiten) | `DELETE time` | 3.9 (S. 51) | 3.37 (S. 65) |
| löschen (Entscheidungen) | `DELETE decisions` | 3.9 (S. 51) | 3.38 (S. 65) |

**Workflows**

| *Beschreibung* | *Operation* | *Listing Schema* | *Listings Beispiele* |
|---|---|---|---|
| anlegen | `CREATE` | 3.11 (S. 52) | 3.20 (S. 59) |
| neue Knoten/Kanten | `ADD` | 3.12 (S. 52) | 3.22 (S. 60) bis 3.25 (S. 61) |
| Elternknoten setzen | `SET` | 3.13 (S. 53) | 3.26 (S. 61) bis 3.28 (S. 62) |
| Zeiten/Kosten setzen | `SET` | 3.14 (S. 53) | 3.29 (S. 62) |
| löschen | `DELETE Workflow` | 3.15 (S. 54) | 3.30 (S. 63) |
| löschen (Knoten) | `DELETE node` | 3.16 (S. 54) | 3.31 (S. 63) |
| löschen (Kanten) | `DELETE edge` | 3.17 (S. 54) | 3.32 (S. 63) |
| löschen (Attribute) | `DELETE` *`attr`* | 3.18 (S. 54) | 3.33 (S. 63) |

**Tabelle 5.2:** Unterstützte Einfüge-, Änderungs- und Löschoperationen von Fallakten und Workflows

## 5.10 Laufzeit

Da die meisten `ESTIMATE`-Abfragen mit auszugebenden Pfaden und Knoten oder Gewichtung in SPARQL sehr schwer oder nicht formulierbar sind und frei wählbare Parameter der Pfadverarbeitung einen sehr großen Einfluss auf die Laufzeit haben können, wird auf einen Geschwindigkeitsvergleich von WQL- und SPARQL-Abfragen verzichtet.

Wie bereits in Abschnitt 5.1 erwähnt umgeht die in Kapitel 3.5 vorgestellte und implementierte Heuristik Endlosschleifen durch eine Suche und Verarbeitung einer maximalen Zahl wahrscheinlichster Pfade. Die Summe gefundener, fehlgeschlagener und in einer Warteschlange vorgehaltener Pfade ist begrenzt. Durch Einbeziehung fehlgeschlagener Abläufe, die durch zu geringe Wahrscheinlichkeit oder nicht erfüllbare Bedingungen auftreten, ist die Summe tatsächlich eine leicht einzuhaltende Höchstgrenze und mit einer maximalen Anzahl an Knoten kann es auch keine endlosen Pfade geben. Da End-

losschleifen auch in den anderen, weniger kritischen Bereichen der Implementierung u.a. durch Flags und Listen besuchter Knoten und Kanten vermieden werden und bei der Suche nach Pfaden i.d.R. tatsächlich die relevantesten gefunden werden, wird auch diese letzte Anforderung erfüllt.

# 6 Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

Zu Beginn wurden in der Einleitung die Motivation und Ziele der Arbeit vorgestellt.

Nachfolgend wurden die Standards RDF und SPARQL als nötige Grundlagen zum Speichern von und den Zugriff auf Workflows erläutert. Weiterhin wurde die für diese Arbeit verwendete Definition des Begriffs geschildert und die Entwicklung von domänenspezifischen Sprachen sowie deren Vor- und Nachteile skizziert.

Im Anschluss wurden die ausgearbeiteten Anforderungen an die Sprache aufgelistet und versucht, die WQL in Kategorien von Anfragesprachen von Workflowmodellen einzuordnen, denen auch verwandte Arbeiten zugeordnet sind. Als Modell für Arbeitsabläufe wurden UML-Aktivitätsdiagramme gewählt und deren RDF-Darstellung erklärt. Weiterhin wurde das Konzept der Fallakte als über mehrere Abfragen verfügbarer Container für Variablen, Zeitangaben und Entscheidungen eingeführt sowie deren Einsatz erläutert. Anschließend wurde die Pfadverarbeitung in Workflows erklärt und die Syntax von Bedingungen von Kanten definiert und deren Verwendung erklärt. Nach der Beschreibung aller Voraussetzungen und Konzepte wurde abschließend die Syntax und Semantik der WQL vorgestellt und deren Einsatz anhand von Beispielen gezeigt.

Auch Implementierungsdetails zum Laden von Workflows, der Auswahl von Kanten an Verzweigungen und zum Ermitteln der Endergebnisse nach der Pfadverarbeitung wurden geschildert und verwendete Programme sowie deren geteste Versionen genannt. Abschließend wurden die ausgearbeiteten Anforderungen auf deren Erfüllung hin evaluiert.

## 6.2 Übereinstimmung mit den gegebenen Zielen

Das Ziel der Arbeit - die Entwicklung einer Anfragesprache für Workflowmodelle - wurde erreicht. Arbeitsabläufe werden als UML-Aktivitätsdiagramme in RDF beschrieben, einzelne oder zusammengefasste Arbeitsschritte können dabei mit Zeiten und Kosten annotiert werden.

Die Anfragesprache verwendet diese Annotationen für Ergebnisse von vier verschiedenen Anfragearten. Weiterhin wird für konkrete Instanzen von Arbeitsabläufen das Konzept der Fallakte eingeführt, womit Variablen, Zeitangaben und Entscheidungen über mehrere Abfragen und als Erfahrungswert für weitere Instanzen zusammengefasst werden.

Anforderungen an das Modell und die Sprache wurden ausgearbeitet und deren Erfüllung evaluiert.

## 6.3 Ausblick

Da Erweiterbarkeit eine wichtige Anforderung an die WQL war, wären in Zukunft mächtigere Datentypen wie Histogramme zur Beschreibung von Zeiten und Kosten sinnvoll und möglich. Auch wurde bisher in Fallakten darauf verzichtet, Kosten angeben zu können. Darüber hinaus kann die Verarbeitung von Bedingungen an Transitionen schon jetzt auf Informationen zur Pfadverarbeitung, derzeit nur die Anzahl an Besuchen des Ursprungsknotens einer Transition, zugreifen, was auch auf Zeitangaben und Kosten ausgeweitet werden könnte.

Andere Konzepte aus verwandten Arbeiten können auch in der WQL sinnvoll sein. Sehr vereinfacht wird Process Mining bereits jetzt einbezogen durch Zeitangaben bisheriger Fallakten. Die Wahrscheinlichkeiten von Entscheidungen an Verzweigungen könnten jedoch ebenso aus bisherigen Abläufen ermittelt werden. Das eingangs gewünschte Millionstel Prozent Wissen zu etwas kann vielleicht mit Data Mining schneller erreicht werden.

Weiterhin kann die WQL Teil einer multidimensionalen Anfragesprache für ProHTA (Prospective Health Technology Assessment) [BL12] werden. Die WQL würde dabei eine Dimension - Abfragen zu geschätzten Zeiten und Kosten zusammengefasst, detaillierter für Knoten, Abläufe oder den wahrscheinlichsten Pfad - übernehmen und Ergebnisse für Abfragen in anderen Teilaspekten bzw. Dimensionen verfügbar machen.

# Appendices

# A Ontologien

In diesem Kapitel sollen die verwendeten Ontologien für das in Kapitel 3.3 vorgestellte Workflow-Modell mit zugehörigen Datentypen und Definitionen für Attribute aufgeführt werden.

## A.1 UML-Diagramm

Die verwendete und in Listing A.1 spezifizierte Ontologie für die Darstellung von UML in RDF ist eine vereinfachte und angepasste Version der in [Dol04] enthaltenen.

```
1  @prefix  owl: <http://www.w3.org/2002/07/owl#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3  @prefix  xsd: <http://www.w3.org/2001/XMLSchema#>
4  @prefix     : <http://www6.cs.fau.de/prohta/fsm#> .
5
6      <>      a owl:Ontology .
7
8  #Classes
9
10 #Basics
11
12 :StateMachine    a                    owl:Class;
13                   rdfs:subClassOf      :StateMachineElement.
14
15 :StateMachineElement      a           owl:Class.
16
17 :Transition      a                    owl:Class;
18                   rdfs:subClassOf      :StateMachineElement.
19
20 :State           a                    owl:Class;
21                   rdfs:subClassOf      :StateMachineElement.
22
```

```
23 #States
24
25 :Simple          a                    owl:Class;
26                   rdfs:subClassOf      :State.
27
28 :Pseudostate     a                    owl:Class;
29                   rdfs:subClassOf      :State.
30
31 :Composite       a                    owl:Class;
32                   rdfs:subClassOf      :State.
33
34 :Region          a                    owl:Class;
35                   rdfs:subClassOf      :Composite.
36
37 :Final           a                    owl:Class;
38                   rdfs:subClassOf      :State.
39
40 :Initial         a                    owl:Class;
41                   rdfs:subClassOf      :State.
42
43 :SynchState      a                    owl:Class;
44                   rdfs:subClassOf      :State.
45
46 #Pseudostates
47
48 :Branch          a                    owl:Class;
49                   rdfs:subClassOf      :Pseudostate.
50
51 :Join            a                    owl:Class;
52                   rdfs:subClassOf      :Pseudostate.
53
54 :Junction        a                    owl:Class;
55                   rdfs:subClassOf      :Pseudostate.
56
57 :Split           a                    owl:Class;
58                   rdfs:subClassOf      :Pseudostate.
59
```

```
#Properties

:Count          a                   owl:DatatypeProperty,
                                    owl:FunctionalProperty;
                rdfs:domain         :SynchState;
                rdfs:range          xsd:int .

:Source         a                   owl:FunctionalProperty,
                                    owl:ObjectProperty,
                                    owl:TransitiveProperty;
                rdfs:domain         :Transition;
                rdfs:range          :State .

:StateName      a                   owl:DatatypeProperty,
                                    owl:FunctionalProperty;
                rdfs:range          xsd:string .

:Target         a                   owl:ObjectProperty;
                rdfs:domain         :Transition;
                rdfs:range          :State .

:TransitionName     a               owl:DatatypeProperty,
                                    owl:FunctionalProperty;
                rdfs:range          xsd:string .

:contains       a                   owl:ObjectProperty;
                rdfs:domain         :StateMachine;
                rdfs:range          :StateMachineElement .

:hasStateMachineElement     a       owl:ObjectProperty;
                rdfs:domain         :Composite;
                rdfs:range          :StateMachineElement .

:isConcurrent   a                   owl:DatatypeProperty,
                                    owl:FunctionalProperty;
                rdfs:domain         :Composite;
                rdfs:range          xsd:boolean .
```

```
97
98  :isRegion       a                   owl:DatatypeProperty,
99                                      owl:FunctionalProperty;
100                 rdfs:domain         :Composite;
101                 rdfs:range          xsd:boolean .
102
103 :Condition      a                   owl:ObjectProperty,
104                 rdfs:domain         :Transition;
105                 rdfs:range          xsd:string .
```

**Listing A.1:** UML in RDF

## A.2 Datentypen

Für die Zuweisung von Attributen zu Workflows wird eine Klasse „DataStructure“ in Listing A.2 definiert, von der sich ein neuer, in Implementierungen der WQL verwendbarer Datentyp ableiten muss. Listing A.3 weist dabei abstrakt diese Daten Zuständen zu, was im Abschnitt zu Kosten verfeinert wird. Um die Erweiterbarkeit mit neuen Datentypen zu verdeutlichen, wird in Listing A.4 der Datentyp für Histogramme aufgeführt.

```
1  @prefix  owl: <http://www.w3.org/2002/07/owl#> .
2  @prefix  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4  @prefix   ds: <http://www6.cs.fau.de/prohta/datastructures#> .
5
6  ds:DataStructure    a               owl:Class.
7  ds:Element          a               owl:Class.
8
9  ds:elementOf        a               rdf:Property ;
10                     rdfs:domain     ds:Element ;
11                     rdfs:range      ds:DataStructure .
```

**Listing A.2:** Definition für „DataStructure“

```
@prefix  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix   ds: <http://www6.cs.fau.de/prohta/datastructures#> .
@prefix  fsm: <http://www6.cs.fau.de/prohta/fsm#> .
@prefix   fd: <http://www6.cs.fau.de/prohta/fsmdata#> .

fd:data         a                   rdf:Property ;
                rdfs:domain         fsm:StateMachineElement ;
                rdfs:range          ds:DataStructure , xsd:float.

fd:time         rdfs:subPropertyOf  fd:data ;
                rdfs:domain         fsm:StateMachineElement ;
                rdfs:range          ds:DataStructure , xsd:float .


fd:Probability  rdfs:subClassOf     ds:DataStructure .

fd:probValue    a                   rdf:Property ;
                rdfs:domain         fd:Probability ;
                rdfs:range          xsd:float .

fd:probability  rdfs:subPropertyOf  fd:data ;
                rdfs:domain         fsm:Transition ;
                rdfs:range          fd:Probability .
```

**Listing A.3:** Zuweisung von Attributen zu Zuständen und Wahrscheinlichkeiten zu Transitionen

```
@prefix  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix   ds: <http://www6.cs.fau.de/prohta/datastructures#> .
@prefix    h: <http://www6.cs.fau.de/prohta/histogram#> .


h:Histogram     rdfs:subClassOf     ds:DataStructure .
h:Bin           rdfs:subClassOf     ds:Element .


h:min           a                   rdf:Property ;
                rdfs:domain         h:Bin ;
                rdfs:range          xsd:float .

h:max           a                   rdf:Property ;
                rdfs:domain         h:Bin ;
                rdfs:range          xsd:float .

h:observations  a                   rdf:Property ;
                rdfs:domain         h:Bin ;
                rdfs:range          xsd:int .
```

**Listing A.4:** Histogramme

## A.3 Kosten

Verschiedene Kostenarten haben eigene Namen, sollen aber dennoch unter einem einzelnen Attribut für Kosten gefunden werden. Deshalb wird in Listing A.5 eine Klasse „Cost“ definiert, die Zuständen benannte Kosten mit Zahlenwerten oder in einer Unterklasse von DataStructure formulierte Informationen zuweist.

```
@prefix  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  owl: <http://www.w3.org/2002/07/owl#> .
@prefix  xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix   ds: <http://www6.cs.fau.de/prohta/datastructures#> .
@prefix   fd: <http://www6.cs.fau.de/prohta/fsmdata#> .
@prefix    c: <http://www6.cs.fau.de/prohta/cost#> .

# Unterklasse von DataStructure zur Verwendung in fd:data
c:Cost      a                   owl:Class ;
            rdfs:subClassOf     ds:DataStructure .

c:cost      rdfs:subPropertyOf  fd:data ;
            rdfs:domain         fsm:StateMachineElement ;
            rdfs:range          c:Cost .

c:name      a                   rdf:Property ;
            rdfs:domain         c:Cost ;
            rdfs:range          xsd:string .

c:value     a                   rdf:Property ;
            rdfs:domain         c:Cost .
            rdfs:range          ds:DataStructure, xsd:float .
```

**Listing A.5:** Kosten von Zuständen

## A.4 Workflow

Listing A.6 bestimmt die Definition eines Workflows mit dessen Attributen für einen Namen, beteiligte Knoten sowie Initial- und Endzustand.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  owl: <http://www.w3.org/2002/07/owl#> .
@prefix  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix    w: <http://www6.cs.fau.de/prohta/workflow#> .
@prefix  fsm: <http://www6.cs.fau.de/prohta/fsm#> .

w:Workflow          a                   owl:Class .

w:name              a                   rdf:Property ;
                    rdfs:domain         w:Workflow ;
                    rdfs:range          xsd:string .

w:hasStateMachineElement        a       owl:ObjectProperty;
                    rdfs:domain         w:Workflow;
                    rdfs:range          fsm:StateMachineElement .

w:init              a                   rdf:Property ;
                    rdfs:domain         w:Workflow ;
                    rdfs:range          fsm:Initial .

w:final             a                   rdf:Property ;
                    rdfs:domain         w:Workflow ;
                    rdfs:range          fsm:Final .
```

**Listing A.6:** Definition von Workflows

# Literaturverzeichnis


[AVH04] Aalst, W. Van d. ; Van Hee, K.M.: *Workflow management: models, methods, and systems*. The MIT press, 2004

[Awa07] Awad, A.: BPMN-Q: A language to query business processes. In: *EMISA* Bd. 119, 2007, 115–128

[Bec07] Beckett, David: *Turtle - Terse RDF Triple Language*. `http://www.dajobe.org/2004/01/turtle/2007-11-20/`. Version: 11 2007, Abruf: 29.08.2012

[Bib12] Bibliographisches Institut GmbH: *Duden online*. `http://www.duden.de/node/778545/revisions/778550/view`. Version: 2012, Abruf: 29.08.2012

[BL12] Baumgärtel, Philipp ; Lenz, Richard: Towards Data and Data Quality Management for Large Scale Healthcare Simulations. In: Conchon, Emmanuel (Hrsg.) ; Correia, Carlos (Hrsg.) ; Fred, Ana (Hrsg.) ; Gamboa, Hugo (Hrsg.): *Proceedings of the International Conference on Health Informatics*, SciTePress - Science and Technology Publications, 2012, S. 275–280. – ISBN: 978-989-8425-88-1

[CB74] Chamberlin, D.D. ; Boyce, R.F.: SEQUEL: A structured English query language. In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* ACM, 1974, 249–264

[DH01] Dumas, Marlon ; Hofstede, Arthur ter: UML Activity Diagrams as a Workflow Specification Language. In: Gogolla, Martin (Hrsg.) ; Kobryn, Cris (Hrsg.): *ÇUMLÈ 2001 Ñ The Unified Modeling Language. Modeling Languages, Concepts, and Tools* Bd. 2185. Springer Berlin / Heidelberg, 2001. – ISBN 978–3–540–42667–7, S. 76–90. – 10.1007/3-540-45441-1_7

[DLBMW10] Daum, Michael ; Lauterwald, Frank ; Baumgärtel, Philipp ; Meyer-Wegener, Klaus: Propagation of Densities of Streaming Data within Query Graphs. In: Gertz, Michael (Hrsg.) ; Ludäscher, Bertram (Hrsg.): *Scientific and Statistical Database Management: 22nd International Conference*. Heidelberg, 2010. – ISBN 978–3–642–13817–1, 584–601

[Dol04] Dolog, P.: Model-driven navigation design for semantic web applications with the UML-guide. In: *Engineering Advanced Web Applications* (2004)

[Dum02] Dumbill, Edd: XML Watch: Finding friends with XML and RDF. (2002), 6. `http://www.ibm.com/developerworks/xml/library/x-foaf/index.html`, Abruf: 29.08.2012

[For04] Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: *ACM SIGPLAN Notices* Bd. 39 ACM, 2004, 111–122

[Fow05] Fowler, M.: Language workbenches: The killer-app for domain specific languages. (2005). `http://www.martinfowler.com/articles/languageWorkbench.html`, Abruf: 29.08.2012

[FP10] Fowler, M. ; Parsons, R.: *Domain-specific languages*. Addison-Wesley Professional, 2010

[Gad10] Gadatsch, A.: Prozessmanagement mit Workflow-Management-Systemen. In: *Grundkurs Geschäftsprozess-Management* (2010), S. 253–284

[Gho11] Ghosh, Debasish: DSL for the uninitiated. In: *Communications of the ACM* 54 (2011), Juli, Nr. 7, S. 44–50. `http://dx.doi.org/10.1145/1965724.1965740`. – DOI 10.1145/1965724.1965740. – ISSN 0001–0782

[GPSD10] Ghattas, J. ; Peleg, M. ; Soffer, P. ; Denekamp, Y.: Learning the context of a clinical process. In: *Business Process Management Workshops* Springer, 2010, 545–556

[HKRS08] Hitzler, P. ; Krötzsch, M. ; Rudolph, S. ; Sure, Y.: *Semantic Web: Grundlagen*. Springer-Verlag Berlin Heidelberg, 2008

[MSL+08] Mans, R. ; Schonenberg, H. ; Leonardi, G. ; Panzarasa, S. ; Cavallini, A. ; Quaglini, S. ; AALST, W. van d.: Process mining techniques: an application to stroke care. In: *Studies in health technology and informatics* 136 (2008), S. 573

[MSS+09] Mans, RS ; Schonenberg, MH ; Song, M. ; Aalst, W.M.P. ; Bakker, PJM: Application of process mining in healthcare–a case study in a dutch hospital. In: *Biomedical Engineering Systems and Technologies* (2009), S. 425–438

[Obj11] Object Management Group: *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1*. `http://www.omg.org/cgi-bin/doc?formal/2011-08-06.pdf`. Version: 8 2011, Abruf: 26.06.2012

[RSS08] RSS: *RDF Site Summary (RSS) 1.0*. `http://web.resource.org/rss/1.0/spec`. Version: 6 2008, Abruf: 29.08.2012

[RSS09] RSS Advisory Board: *RSS 2.0 Specification*. `http://www.rssboard.org/rss-specification`. Version: 3 2009, Abruf: 29.08.2012

[She07] Shen, Xiuyun: *A Domain-Specific Conceptual Query System*, Georgia State University, Diplomarbeit, 2007

[TSCJ+06] Tian, Hao ; Sunderraman, Rajshekhar ; Calin-Jageman, Robert ; Yang, Hong ; Zhu, Ying ; Katz, Paul: NeuroQL: A Domain-Specific Query Language for Neuroscience Data. Version: 2006. `http://dx.doi.org/10.1007/11896548_46`. In: Grust, Torsten (Hrsg.) ; HŠpfner, Hagen (Hrsg.) ; Illarramendi, Arantza (Hrsg.) ; Jablonski, Stefan (Hrsg.) ; Mesiti, Marco (Hrsg.) ; MŸller, Sascha (Hrsg.) ; Patranjan, Paula-Lavinia (Hrsg.) ; Sattler, Kai-Uwe (Hrsg.) ; Spiliopoulou, Myra (Hrsg.) ; Wijsen, Jef (Hrsg.): *Current Trends in Database Technology Ð EDBT 2006* Bd. 4254. Springer Berlin / Heidelberg, 2006. – ISBN 978–3–540–46788–5, 613-624. – 10.1007/11896548_46

[TSY07] Tian, H. ; Sunderraman, R. ; Yang, H.: A Domain-Specific Conceptual Data Modeling and Querying Methodology. In: *1st International Conference on Information Systems, Technology and Management*, 2007

[VDA12] Van Der Aalst, Wil: Process mining. In: *Commun. ACM* 55 (2012), August, Nr. 8, S. 76–83. `http://dx.doi.org/10.1145/2240236.2240257`. – DOI 10.1145/2240236.2240257. – ISSN 0001–0782

[W3C99] W3C: *Resource Description Framework (RDF) Model and Syntax Specification*. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`. Version: 2 1999, Abruf: 29.08.2012

[W3C04] W3C: *Resource Description Framework (RDF): Concepts and Abstract Syntax*. http://www.w3.org/TR/rdf-concepts/. Version: 2 2004, Abruf: 29.08.2012

[W3C08] W3C: *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/. Version: 1 2008, Abruf: 29.08.2012

[W3C09] W3C: *OWL 2 Web Ontology Language Document Overview*. http://www.w3.org/TR/owl2-overview/. Version: 10 2009, Abruf: 29.08.2012

[W3C11] W3C: *Notation3 (N3): A readable RDF syntax*. http://www.w3.org/TeamSubmission/n3/. Version: 3 2011, Abruf: 29.08.2012

[W3C12] W3C: *SPARQL 1.1 Update*. http://www.w3.org/TR/sparql11-update/. Version: 1 2012, Abruf: 29.08.2012

[Wor99] Workflow Management Coalition: *Terminology & Glossary*. http://www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html. Version: 1999, Abruf: 29.08.2012