SHORE

1.1.0

Anfragen in Prolog

0.2 (Vorabversion)

Klaus Mayr, Tammo Schnieder, Helge
Schulz

sd&m AG
software design & management
Thomas-Dehler-Strasse 27
81773 München
(089) 6 38 12-0
(089) 6 38 12-150

www.sdm.de

Historie

Version	Status	Datum	Autor(en)	Erläuterung
0.2	freigegeben	05.10.2001	Tammo Schnie-der	Indexbegriffe ein-gefügt
0.1	in Arbeit	14.02.2001	Tammo Schnie-der	Erstellung, Kapi-tel aus Anwender-handbuch herausgelöst.

Inhaltsverzeichnis

Historie ... **iii**
Inhaltsverzeichnis ... **v**
1 Einleitung ... **1**
2 Konzept ... **3**
2.1 Prolog-Grundlagen ... 3
2.1.1 Begriffe von Prolog ... 3
2.1.2 Aussagen in Prolog ... 5
2.1.3 Endlosschleifen in rekursiven Abfragen vermeiden ... 6
3 Beispiel ... **7**
4 Erste Schritte ... **9**
4.1 So gewinnen Sie einen Überblick über Ihr Metamodell ... 9
Hintergrund ... 9
4.2 So stellen Sie eine Prolog-Anfrage ... 10
weitere Anfragen ... 12
5 Anfragen in SHORE ... **13**
5.1 Prolog-Anfragen ... 13
5.1.1 Vordefinierte Prädikate von SHORE ... 13
5.1.2 Zusätzliche datenabhängige SHORE-Prädikate ... 15
5.1.3 Übersicht aller vordefinierter SHORE-Prädikate ... 16
5.1.4 Eigene Anfragen formulieren ... 17
Welche Dokumente verwaltet die SHORE-Instanz? ... 17
Welche Objekte kennt die SHORE-Instanz? ... 17
Welche Objekte vom Typ 'C-Funktion' gibt es? ... 17
Welche Beziehungen vom Typ 'ruft' gibt es? ... 17
Welche Beziehungen hat die C-Funktion 'eingabe'? ... 17
Was ruft die C-Funktion 'eingabe' direkt oder indirekt auf? ... 18
Welche Unterprogramme werden direkt oder indirekt sowohl von der C-Funktion 'eingabe' als auch von der C-Funktion 'ausgabe' aufgerufen? 18
Welche Unterprogramme werden direkt oder indirekt von der C-Funktion 'eingabe' aufgerufen, wobei alle besuchten Unterprogramme die globale Variable 'flag' benutzen? ... 18
Welche Methoden aus den Implementierungen des Interfaces 'A' benutzen nicht die Schnittstelle 'B'? ... 19
Welche Methoden werden durch abgeleitete Klassen überschrieben? . 19
Anhang ... **21**
Literaturverzeichnis ... 21
Links ... 21

Syntax regulärer Ausdrücke 21
Prolog-Programm: shore_base.P 22
Index 27

1 Einleitung

Eine wesentliche Stärke von SHORE besteht darin, anhand der in SHORE enthaltenen Dokumente, Objekte und Beziehungen komplexe Fragestellungen beantworten zu können.

Dazu verwendet SHORE Prolog als Anfragesprache (Programming by Logic). Prolog ist recht einfach zu erlernen und eignet sich, um komplexe Anfragen inklusive Rekursion und Pfadverfolgung beantworten zu können.

An wen richtet sich das Handbuch?

Dieses Handbuch richtet sich an Administratoren und fortgeschrittene Endanwender von SHORE. Sie wollen wissen, welche Möglichkeiten Ihnen das integrierte Prolog bietet und möchten gerne eigene Anfragen schreiben.

Wie kann ich das Handbuch verwenden?

Jeder geht auf seine ihm eigene Art und Weise mit einem Handbuch um. Abhängig von Ihren Vorkenntnissen und Ihrem Informationsbedürfnis können Sie dieses Handbuch auf verschiedene Weise nutzen. Um Ihnen den Zugang zu erleichtern, stellen wir Ihnen kurz vor, was wir uns bei der Strukturierung dieses Handbuchs gedacht haben.

Falls Sie noch nie mit Prolog gearbeitet haben oder grundsätzliches zu Prolog wissen wollen, lesen Sie 2.1 "Prolog-Grundlagen". Hier gehen wir auf die Syntax von Prolog ein.

Falls Sie noch nie eine Anfrage in SHORE formuliert haben, empfehlen wir Ihnen, das Kapitel 3 "Beispiel" mit unserem Beispiel durchzuarbeiten.

2 Konzept

Als Query-Maschine wird in SHORE ein Prolog-System eingesetzt. An diese können Sie vordefinierte oder frei definierte ad-hoc Anfragen richten. Jede Anfrage besteht letztendlich aud einer Zeichenkette, deren Aufbau wir nachfolgend beschreiben.

2.1 Prolog-Grundlagen

Prolog ist eine Turing-mächtige, deklarative Programmiersprache, die auf der Prädikatenlogik basiert. Eine Einleitung über diese Konzepte kann in der im Anhang genannten Literatur nachgelesen werden.

Hier sollen nur die Grundlagen geschaffen werden, um Prolog für die Beantwortung von Fragestellungen innerhalb von SHORE nutzen zu können.

2.1.1 Begriffe von Prolog

Im folgenden werden grundlegende Begriffe von Prolog kurz erklärt. Eine vollständige Definition der Begriffe des in SHORE verwendeten Prolog-Systems findet sich in “K.F. Sagonas, T. Swift, D. S. Warren, J. Freire, P.Rao: The XSB Programmer's Manual Version 1.8.”.

Atome

Es gibt vier Arten von Atomen in Prolog:

- Zeichenketten, die aus Buchstaben, Ziffern und dem Unterstrich bestehen und mit einem Kleinbuchstaben beginnen;
- Zeichenketten, die ausschließlich aus Sonderzeichen bestehen;
- [] und {};
- Zeichenketten, die in Hochkommata eingeschlossen sind.

Beispiele für Atome:

platzhalter
linke_Seite
'hallo'
abs_Nullpunkt_Grad_Celsius
:=

Zeichenketten

Zeichenketten, die mit einem Großbuchstaben beginnen oder Leerzeichen oder Sonderzeichen enthalten, müssen in einfache Hochkommas eingeschlossen werden:

```
'BLZ der Bundesbank'
```

Wenn das einfache Hochkomma selbst vorkommen soll, so wird es verdoppelt:

```
'_H4qr''z 5£4n6_'
```

Zahlen

Zahlen können wie folgt dargestellt werden:

```
1
-345
-34.56
2.0E-1
```

Konstanten

Konstanten sind Zahlen oder Atome.

Variablen

Variablen werden als Folgen alphanumerischer Zeichen oder Unterstrichen ('_') geschrieben, die mit einem Großbuchstaben oder einem Unterstrich beginnen.

```
ErgebnisObjekt
_erstes
_123
```

anonyme Variable '_'

Eine Besonderheit stellt die anonyme Variable '_' dar, die für Variablen verwendet werden kann, die nur einmal in einer Prolog-Regel vorkommen. Mehrere Variablen '_' innerhalb einer Prolog-Klausel werden als verschieden angesehen (sogenannte anonyme oder 'don't-care'-Variablen).

Terme

Ausdrücke in Prolog heißen Terme und bestehen aus Konstanten, Variablen oder zusammengesetzten Termen.

Zusammengesetzte Terme bestehen aus

- einem Funktionssymbol oder Prädikat und
- einer Folge von Termen als Argumenten in runden Klammern eingeschlossen.

Beispiele:

```
foo(bar)
prolog(a, X)
ergebnisObjekt(_)
```

f(a, b((c))(d))
map(double)([], [])
X(Y,name)

Terme repräsentieren (Daten-) Objekte, die durch Funktionssymbole strukturiert werden können, z.B.

Funktionssymbole

```
adresse( strasse( 'Thomas-Dehler-Straße'),
hausnummer(27), ort('München') )
```

.

Funktionssymbole bestehen aus einem Atom als Namen und, in Klammern, einem oder mehreren Argumenten. Argumente von Funktionssymbolen sind beliebige zusammengesetzte Terme. Zweistellige Funktionssymbole können (nach entsprechender Deklaration) als Infix-Operatoren geschrieben werden, einstellige Funktionssymbole als Präfix- oder Postfix-Operatoren.

Listen

Eine spezielle Syntax erleichtert den Umgang mit der Datenstruktur Liste. `[]` bezeichnet die leere Liste, `[Head|Tail]` bezeichnet die Liste, die als Kopf den zusammengesetzten Term `Head` und als Schwanz die Liste `Tail` hat. Eine Liste kann aber auch durch Aufzählung ihrer Elemente in eckigen Klammern definiert werden. Beispielsweise kann die Liste mit den drei Elementen `a`, `b` und `c` als `[a|[b|[c|[]]]]` oder als `[a,b,c]` geschrieben werden.

Prädikate

Prädikate besitzen die gleiche Syntax wie Funktionssymbole.

Prolog-Klauseln

Eine Prolog-Klausel hat die Form `p(..):-q1(..),..,qm(..).` mit Prädikaten `p`, `q1`, ..., `qm`.

Prolog-Anfragen und Prolog-Programme

Prolog-Anfragen sehen wie Prolog-Klauseln ohne Kopf-Prädikat aus, haben also die Form `q1(..),..,qm(..).` mit Prädikaten `q1`, ..., `qm`.

Syntaktisch besteht ein Prolog-Programm oder eine Prolog-Anfrage aus Konstanten, Variablen, Funktionssymbolen und Prädikaten.

2.1.2 Aussagen in Prolog

Prolog liefert Aussagen über das gespeicherte Wissen zurück. Eine Aussage ist die Antwort von Prolog auf eine Anfrage.

Eine Prolog-Anfrage kann als Prädikat oder Prolog-Klausel formuliert werden.

Prädikate repräsentieren das Wissen über die in Prolog bekannten Objekte, d.h. eine Aussage `p(t1,..,tn)` ist wahr, wenn `p` für die Terme `t1`, ..., `tn` gilt.

Eine Prolog-Klausel ist eine wenn-dann-Aussage: wenn die Prädikate der rechten Seite (der Rumpf) wahr sind, dann gilt auch die linke Seite. Dadurch kann aus bekanntem Wissen neues Wissen abgeleitet werden.

Prolog-Anfragen beantworten bei konstanten Argumenten die Frage, ob die verwendeten Prädikate für die angegebenen Argumente gelten (Ausgabe ja oder nein).

Bei variablen Argumenten der Prädikate liefern sie als Ergebnismenge alle Variablenbelegungen, die die Prädikate erfüllen, also wahr machen.

2.1.3 Endlosschleifen in rekursiven Abfragen vermeiden

Ein Mangel von Prolog ist das Unvermögen, bestimmte rekursive Anfragen zu berechnen. Das Prolog-Programm

```
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-ancestor(X,Y):-ancestor(X,Z), parent(Z,Y).
```

zusammen mit der Anfrage

```
ancestor(1,Y).
```

führt in Prolog zu einer Endlosschleife. Diesem Problem können Sie begegnen, indem Sie für rekursive Prädikate eine andere Auswertungsmethode wählen, die diese Schwäche nicht aufweist. Dazu stellen Sie dem Programm eine Deklaration

```
:-table( ancestor/2 ).
```

voran. Die allgemeine Syntax lautet

-table(<Prädikat>/<Stelligkeit>).

Details zu diesem Thema können Sie der Dokumentation des verwendeten Prolog-Systems entnehmen.

3 Beispiel

Das Beispiel soll Ihnen zeigen, wie man eine Fragestellung als Anfrage in SHORE formuliert.

Dazu nutzen wir wieder das aus dem Anwenderhandbuch bekannte Taschenrechnerbeispiel. Wir gehen davon aus, dass Sie mit der Oberfläche von SHORE vertraut sind, und dass Ihnen eine SHORE-Instanz mit dem Taschenrechnerbeispiel zur Verfügung steht.

Das Taschenrechnerbeispiel besteht aus einer Anwendungsfallbeschreibung und seiner Implementierung.

Die Implementierung besteht aus einer in Java realisierten Dialogkomponente und einem über eine Middleware verbundenen Anwendungskern, der in Cobol realisiert wurde.

In dem folgenden Kapitel 4 "Erste Schritte" wollen wir erste einfache Prolog-Anfragen formulieren, damit Sie ein Gefühl dafür bekommen, wie Anfragen grundsätzlich funktionieren.

4 Erste Schritte

Ihnen steht eine SHORE-Instanz mit dem Taschenrechnerbeispiel zur Verfügung und Sie möchten einfache Fragestellungen bezüglich der in der Datenbank enthaltenen Dokumente, Objekte und Beziehungen formulieren. Wir empfehlen Ihnen die Kapitel “Beispiel” auf Seite 7 und “Konzept” auf Seite 3 vorher zu lesen, sie sind aber nicht Voraussetzung für das folgende Kapitel.

Das folgende Kapitel soll Ihnen am Beispiel Taschenrechner den Einstieg erleichtern.

4.1 So gewinnen Sie einen Überblick über Ihr Metamodell

Sie kennen das Metamodell noch nicht aus dem ff und möchten ersteinmal einen Überblick gewinnen, welche Dokument-, Objekt- und Beziehungstypen existieren und wie sie zusammenhängen.

1. Sie klicken im Menü „Metamodell“auf irgendeinen Dokument-, Objekt- oder Beziehungstypen.`Beispiel: Metamodell>Dokumenttypen>Middlewarebeschreibung`
   Die Definition des Metamodells öffnet sich im Dokumenten-Bereich.
2. Nutzen Sie die Suchen-Funktion Ihres Browsers um einen Überblick zu gewinnen, wie die Typen zusammenhängen. `Beispiel: suchen Sie nach "Middlewarebeschreibung". Sie finden folgende Definitionen: Dokumenttyp Middlewarebeschreibung`

```
   ...
   Beziehungstyp NamensraumRuftViaMiddlewareFunktion
 alias ruft_via_Middleware
       von 0 bis * Namensraum
       nach 0 bis * Funktion
       ist Teilmenge ruft
       ist definiert in Middlewarebeschreibung
```

Hintergrund

Sie benötigen eine möglichst gute Kenntnis über das Metamodell, damit Sie Anfragen formulieren können.

4.2 So stellen Sie eine Prolog-Anfrage

Sie möchten gerne eine Prolog-Anfrage an Ihre SHORE-Instanz stellen.

Als Beispiel nehmen wir an, dass sie gerne wissen möchten, welche Programme die Anwendungsfallbeschreibung "Taschenrechner" direkt implementieren.

Wenn Sie eine Anfrage stellen wollen, rufen Sie nicht zuerst den entsprechenden Dialog auf, sondern Sie müssen sich ersteinmal im Klaren darüber sein, welche Möglichkeiten Ihnen anhand des Metamodells gegeben sind.

Dies entspricht dem Verfolgen der "implementiert"-Beziehungen der Anwendungsfälle (Addieren, Subtrahieren, Multiplizieren und Dividieren), die Sie sehen, wenn Sie die Anwendungsfallbeschreibung "Taschenrechner" im Dokument-Bereich sehen.

Dazu müssen Sie sich zunächst mit Hilfe des Metamodells mit den beteiligten Dokument-, Objekt- und Beziehungstypen vertraut machen. Dann können Sie eine entsprechende Anfrage formulieren.

1. Klicken Sie im Menü „Metamodell" auf Dokumenttypen > Anwendungsfallbeschreibung.
   Das Metamodell wird im „Dokument-Bereich" angezeigt.
2. Machen Sie sich mit den Definitionen des Metamodells vertraut. Für unser Beispiel stehen alle Definitionen dicht beisammen. Uns interessiert in der Anwendungsfallbeschreibung der Objekttyp System. Dieser Typ hat eine Beziehung zu den Objekttypen Anwendungsfall und jeder Anwendungsfall hat eine Beziehung zu einer Funktion.
   Wir müssen also den Weg von der Anwendungsfallbeschreibung zur Funktion verfolgen.
3. Stellen Sie sicher, dass der Dokument-Bereich den Fokus hat. Dazu können Sie sicherheitshalber auf irgendeine freie Stelle - hinter der kein Link liegt - im Dokument-Bereich klicken.
4. Öffnen Sie den „Suchen"-Dialog des Browsers (je nach Browser beispielsweise mit der Maus Bearbeiten > Rahmen durchsuchen... oder Bearbeiten > Suchen (aktuelle Seite)... oder mit der Tastenkombination `<Strg-F>`).
5. Geben Sie als Suchbegriff `Anwendungsfall` ein und suchen Sie solange, bis Sie einen Beziehungstyp gefunden haben, in dem `Anwendungsfall` in der `von` oder `nach`-Deklaration steht. Wir suchen nach diesem Eintrag:

```
Beziehungstyp FunktionRealisiertAnwendungsfall  alias realisiert
      von 1 bis 1 Funktion
      nach 1 bis 1 Anwendungsfall
      ist definiert in Quelltext
```

Abbildung 1: Ausschnitt aus dem Metamodell

6. Klicken Sie im Menü „Anfragen“ auf „Individuelle Anfragen“.
   Der Dialog „Individuelle Anfrage stellen“ öffnet sich.

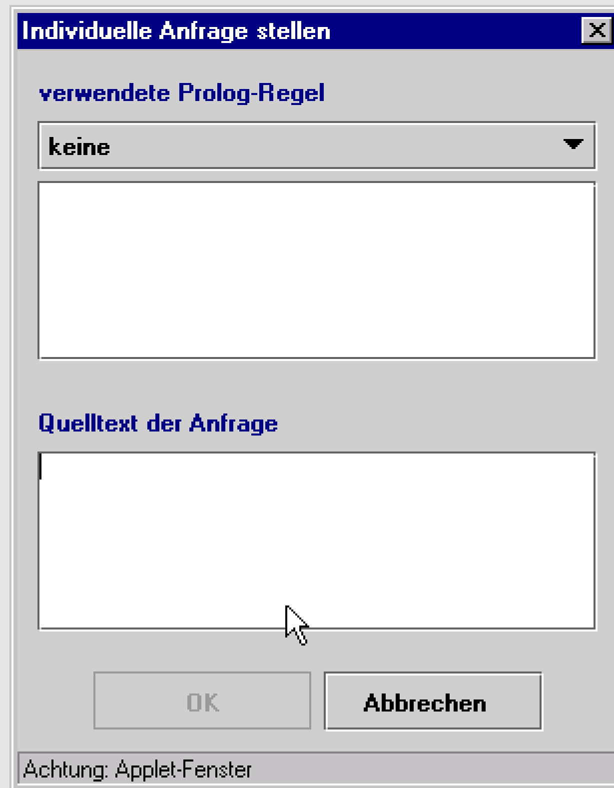

Abbildung 2: Der Dialog "Individuelle Anfrage stellen"

7. Im unteren Teil des Fensters können Sie Ihre Anfrage eingeben.
   Formulieren Sie die entsprechende Prolog-Anfrage:
   ```
   system(ObjIDsys,'Taschenrechner'),
   systemImplementiertAnwendungsfall(_,ObjIDsys,ObjIDawf),
   funktionRealisiertAnwendungsfall(_,ObjIDfun,ObjIDawf),
   ist_definiert_in(ObjIDfun,_).
   ```
   Die letzte Klausel (ist_definiert_in) dient dazu, das Dokument auszugeben, in dem die Implementierung enthalten ist.
8. Die Anfrage wird an das System gestellt, wenn Sie mit „ok“ bestätigen.

Das Ergebnis Ihrer Anfrage sehen Sie im Dokumentbereich:

Anfrage 'system(ObjIDsys,'Taschenrechner'), systemImplementiertAnwendungsfall(_,ObjIDsys,ObjIDawf), funktionRealisiertAnwendungsfall(_,ObjIDfun,ObjIDawf), ist_definiert_in (ObjIDfun,_).'

```
[sysinitrc loaded]

[shore_base loaded]

[Dynamically loading F:/Programme/shore/xsb/oop-bsp]

[F:/Programme/shore/xsb/oop-bsp.P dynamically loaded, cpu time used: 0.2030 seconds]
```

-- implementiert --> , System Taschenrechner, -- realisiert --> , Anwendungsfall Taschenrechner-Addieren, JavaMethode JBC1Main.add(), JavaQuelltext JBC1Main
-- implementiert --> , System Taschenrechner, -- realisiert --> , Anwendungsfall Taschenrechner-Subtrahieren, JavaMethode JBC1Main.subtract(), JavaQuelltext JBC1Main
-- implementiert --> , System Taschenrechner, -- realisiert --> , Anwendungsfall Taschenrechner-Multiplizieren, JavaMethode JBC1Main.multiply(), JavaQuelltext JBC1Main
-- implementiert --> , System Taschenrechner, -- realisiert --> , Anwendungsfall Taschenrechner-Dividieren, JavaMethode JBC1Main.divide(), JavaQuelltext JBC1Main

---

4 Ergebnisse

Abbildung 3: Ergebnis einer Prolog-Anfrage.

Das Ergebnis ist in folgende Bereiche gegliedert:

1. Als erstes wird der Text der Anfrage angezeigt.
2. Danach folgen Meldungen das in SHORE integrierten Prolog-Systems. Falls Fehler auftreten, so werden diese hier angezeigt.
3. Anschließend folgt das Ergebnis der Anfrage. Falls die Anfrage kein Ergebnis geliefert hat oder ein Fehler aufgetreten ist, steht hier als Ergebnis Nein.
   In unserem Beispiel sehen Sie, dass die Anfrage erfolgreich beantwortet wurde.
4. Zuletzt steht gegebenenfalls noch die Anzahl der gefundenen Ergebnisse.

weitere Anfragen

In diesem kurzen Beispiel wurde nur eine relativ einfache Anfrage gestellt. Damit Sie selbst Anfragen formulieren können, lesen Sie bitte das Kapitel 5.1 "Prolog-Anfragen"

5 Anfragen in SHORE

Die ersten Schritte sind gemacht, und das Beispiel können Sie jetzt nachvollziehen. Nun können Sie eigene Anfragen formulieren.

Wenn Sie das Kapitel “Beispiel” auf Seite 7 nicht gelesen haben, möchten wir es Ihnen empfehlen. Falls Sie sich nicht mit Prolog auskennen, sollten Sie das Kapitel “Konzept” auf Seite 3 lesen, da Prolog-Know-How Voraussetzung für das folgende Kapitel ist.

Dieses Kapitel soll Ihnen die Möglichkeit geben, eigene - auf Ihr Metamodell zugeschnittene - Anfragen zu formulieren.

5.1 Prolog-Anfragen

Dem Prolog-System in SHORE sind alle Informationen bekannt, die in den Dokumenten der SHORE-Instanz durch entsprechenden durch das Metamodell definierten XML-Markup ausgezeichnet sind.

Genauer gesagt, kennt das Prolog-System alle in SHORE gespeicherten Objekte, Dokumente und Beziehungen, ihre jeweiligen Typen und die Quellen und Ziele von Beziehungen. Darüberhinaus weiß es, welche Objekte und Beziehungen in welchem Dokument definiert sind. Die Prädikate, die dieses Wissen repräsentieren, werden im Abschnitt “Vordefinierte Prädikate von SHORE” auf Seite 13 vorgestellt.

Für Anfragen in SHORE kann man Prolog-Regeln verwenden. Das sind beliebige Prolog-Programme (= Menge von Prolog-Klauseln), die dazu dienen, zusätzliche Prädikate aus den Grundprädikaten (`document`, `object` und `relation`) abzuleiten.

5.1.1 Vordefinierte Prädikate von SHORE

Sie können in Prolog-Anfragen durch vordefinierte Prädikate auf die in den Dokumenten der SHORE-Instanz enthaltenen und durch XML-Markup ausgezeichneten Informationen zugreifen.

Als solche vordefinierte Grundprädikate stehen Ihnen die drei Prädikate `object`, `relation` und `document` zum Zugriff auf die Objekte, Beziehungen bzw. Dokumente der Datenbasis zur Verfügung.

Jedes dieser Prädikate ist in verschiedenen Stelligkeiten ausgestaltet, das heißt es wird eine unterschiedliche Anzahl von Argumenten erwartet.

- `object` bzw. `document` als einstelliges Prädikat hat alle Objekte bzw. Dokumente der Datenbank als Erfüllungsmenge.
- Dieselben Prädikate als dreistellige Prädikate bilden Objekte bzw. Dokumente auf ihren Typ und Namen ab.

Hinweis: In Prolog-Schreibweise unterscheidet man die Stelligkeit so: `object/1` erwartet ein Argument und `object/3` erwartet drei Argumente.

Zum Beispiel liefert die Anfrage

```
object(X).
```

alle in der Datenbasis enthaltenen Objekte, und

```
document(X,'Quellcode',Name).
```

hat als Ergebnis alle Dokumente des Typs `Quellcode` und ihre Namen in der Datenbank.

Das Prädikat `relation` gibt es in den Stelligkeiten eins, vier und sechs:

```
relation(X).
```

liefert alle Beziehungen,

```
relation(X, Typ, Quelle, Ziel).
```

ergibt die Menge der Beziehungen mit Beziehungstyp, Quelle und Ziel, und

```
relation(X, Typ, Quelltyp, Quellname, Zieltyp, Ziel-
name).
```

schließlich bildet Beziehungen auf ihren Typ und den Name und Typ von Quelle und Ziel ab.

Zusätzlich existiert ein Prädikat `relationClosure/3`, welches die transitive Hülle einer Beziehung eines bestimmten Typs zwischen Quelle und Ziel berechnet.

Hinweis: Alle Ergebnisse, die direkt und indirekt die gewünschte Beziehung zwischen Quelle und Ziel liefern.Beispiel: Die transitive Hülle der Beziehung A "hat als Kind" B für einen vorgegebenen Personenkreis umfasst alle A und B inklusive aller Nachfahren.

Darüberhinaus stellt SHORE ein Prädikat `match/2` zur Verfügung, mit dem reguläre Ausdrücke und Zeichenketten verglichen werden können. Beispielsweise ist

```
match( 'a*', 'aaaa' ).
```

wahr. Dieses Prädikat kann nicht mit freien Variablen verwendet werden, da eine Anfrage wie `match( 'a*', X )` zum Beispiel nach allen Zeichenketten fragen würde, die auf das Muster `a*` passen und damit eine unendliche Ergebnismenge liefert.

Alle bisher aufgeführten Prädikate stehen Ihnen für Anfragen automatisch zur Verfügung. Daneben stellt das Prolog-System aber noch weitere nützliche 'eingebaute' Prädikate zur Verfügung, die zum Beispiel den Umgang mit Listen oder Strings erleichtern. Diese Prädikate befinden sich zum Teil in eigenen Bibliotheken, die vor der Benutzung importiert werden müssen. Es existiert beispielsweise ein Prädikat `merge/3`, das zwei Listen zusammenfügt. Dieses befindet sich in einer Bibliothek namens '`listutils`' und muss vor der Benutzung mit der Anweisung `import merge/3 from listutils.` geladen werden. Die zur Verfügung stehenden Prädikate und die jeweils benötigten Bibliotheken lassen sich im Kapitel 'Library Utilities' von “K.F. Sagonas, T. Swift, D. S. Warren, J. Freire, P.Rao: The XSB Programmer's Manual Version 1.8.” nachlesen.

5.1.2 Zusätzliche datenabhängige SHORE-Prädikate

Damit Sie Anfragen intuitiver formulieren können, leitet SHORE Prädikate mit den Namen des projektspezifischen Metamodells automatisch ab.

Gibt es im Ihrem Metamodell zum Beispiel einen Objekttyp `Variable`, so erzeugt SHORE beim Import des Metamodells automatisch eine Prolog-Regel, in der die Prädikate `variable/1` und `variable/2` definiert sind. Das einstellige Prädikat liefert alle Objekte des Typs `Variable`, das zweistellige liefert alle Objekte des Typs 'Variable' mit ihren Namen.

Diese Prädikate gibt es zu jedem Typ des Metamodells. Falls eine Stelle des Grundprädikats (`object`, `relation` oder `document` in den verschiedenen Stelligkeiten) den Typ als Argument beinhaltet, so fällt diese weg, da die Typinformation bereits im Namen des Prädikats kodiert ist. Z.B. wird
`object(X,'Funktion',Name)`
zu
`funktion(X,Name)`
Bei der Verwendung dieser Prädikate ist zu beachten, dass alle Prädikate mit einem Kleinbuchstaben beginnen müssen, um nicht als Variablen interpretiert zu werden (daher im Beispiel 'funktion' statt 'Funktion'). Außerdem werden '-'-Zeichen in Underscores umgewandelt, um Verwechslungen mit dem Minus-Operator zu vermeiden.

Erinnern Sie sich noch an das Beispiel zur individuellen Anfrage? Dort haben wir die Anfrage `document(_,Anwendungsfallbschreibung,_).` formuliert. Sie können die Anfrage also auch als `anwendungsfallbeschreibung(_,_).` formulieren und erhalten dasselbe Ergebnis. Beachten Sie dabei, die Kleinschreibung von Prädikaten.

Neben den Prädikaten `object`, `relation` und `document` in ihren verschiedenen Stelligkeiten kann man mit den Prädikaten `kommt_vor_in/2` und `ist_definiert_in/2` abfragen, welche Beziehungen bzw. Objekte in welchem Dokument enthalten sind. Beispielsweise liefert die Anfrage
`object(X, 'Funktion', 'main'), ist_definiert_in(X, Dokument).`
das Dokument, in dem die Funktion `main` definiert ist.

5.1.3 Übersicht aller vordefinierter SHORE-Prädikate

In der folgenden Tabelle sind noch einmal alle vordefinierten Prädikate von SHORE zusammengefasst. Bei der Angabe der Typen der Argumente werden die Abkürzungen

- S für String-Argumente und
- R für Datenbankreferenzen verwendet.

Datenbankreferenzen können nur durch SHORE-eigene Prädikate erzeugt werden, sie werden bei der Ausgabe des Ergebnisses als Hyperlinks dargestellt. Sie können bei Anfragen nicht direkt angegeben werden, da ihre Werte nach außen nicht sichtbar sind, und werden typischerweise mir einer Variablen belegt (oft auch mit der anonymen Variablen "_").

Prädikat	Bedeutung (und Typ) der Variablen
object/1	Objekt(R)
object/3	Objekt(R), Objekttyp(S), Objektname(S)
relation/1	Beziehung(R)
relation/4	Beziehung(R), Beziehungstyp(S), Quelle(R), Ziel(R)
relation/6	Beziehung(R), Beziehungstyp(S), Quelltyp(S), Quellname(S), Zieltyp(S), Zielname(S)
relationClosure/3	Beziehungstyp(R), Quelle(R), Ziel(R)
document/1	Dokument(R)
document/3	Dokument(R), Dokumenttyp(S), Dokumentname(S)
ist_definiert_in/2	Objekt(R), Dokument(R)
kommt_vor_in/2	Beziehung(R),Dokument(R)
match/2	regulärer Ausdruck(S), Text(S)
für jeden Objekttyp 'x' des Metamodells	

Prädikat	Bedeutung (und Typ) der Variablen
x/1	Objekt des Typs 'x'(R)
x/2	Objekt des Typs 'x'(R), Objektname(S)
für jeden Beziehungstyp 'y' des Metamodells	
y/1	Beziehung des Typs 'y'(R)
y/3	Beziehung des Typs 'y'(R), Quelle(R), Ziel(R)
y/5	Beziehung des Typs 'y'(R), Quelltyp(S), Quellname(S), Zieltyp(S), Zielname(S)
y_rekursiv/2	Quelle(R),Ziel(R)
für jeden Dokumenttyp des Metamodells 'z'	
z/1	Dokument des Typs 'z'(R)
z/2	Dokument des Typs 'z'(R), Dokument-name(S)

5.1.4 Eigene Anfragen formulieren

Um die Anfragesprache und die Benutzung der vordefinierten Prädikate zu verdeutlichen, hier zunächst einige Fragestellungen und wie sie in Prolog formuliert werden.

Welche Dokumente verwaltet die SHORE-Instanz?

```
document(X).
```

Welche Objekte kennt die SHORE-Instanz?

```
object(X).
```

Welche Objekte vom Typ 'C-Funktion' gibt es?

```
object(X, ´C-Funktion´, _).
```

oder `c_funktion(X, _).`

Welche Beziehungen vom Typ 'ruft' gibt es?

```
relation(X, ´ruft´, _, _).
```

oder `ruft(X, _, _).`

Welche Beziehungen hat die C-Funktion 'eingabe'?

Hier werden Ein- und Ausgehende Beziehungen gesucht, die durch die Kombination von zwei Anfragen ermittelt werden müssen. Dazu muss zuerst eine Prolog-Regel geschrieben werden:

Prolog-Regel:`info(Type, Name, RelId) :-`
```
  object(X, Type, Name),
  relation(RelId, _, X, _).

info(Type, Name, RelId) :-
  object(X, Type, Name),
  relation(RelId, _, _, X).
```
Anschließend kann die Regel genutzt werden:

Anfrage:`info(´C-Funktion´, ´eingabe´, X).`

Was ruft die C-Funktion ’eingabe’ direkt oder indirekt auf?

Die Regel `ruft_rekursiv` muss definiert sein.
```
c-funktion(X, ´eingabe´), ruft_rekursiv(X, Y).
```

Welche Unterprogramme werden direkt oder indirekt sowohl von der C-Funktion ’eingabe’ als auch von der C-Funktion ’ausgabe’ aufgerufen?

Anfrage:
```
gerufen_von_beiden(´eingabe´, ´ausgabe', X).
```
Prolog-Regel:
```
gerufen_von_beiden(CName1,  CName2, Obj) :-
c-funktion(X1, CName1),
c-funktion(X2, CName2),
ruft_rekursiv(X1, Obj),
ruft_rekursiv(X2, Obj).
```

Welche Unterprogramme werden direkt oder indirekt von der C-Funktion ’eingabe’ aufgerufen, wobei alle besuchten Unterprogramme die globale Variable ’flag’ benutzen?

Anfrage:
```
unterprogramme_via_flag(´eingabe´, X).
```
Prolog-Regel:
```
:- table(ruft_rekursiv_mit_var/3).
ruft_rekursiv_mit_var(Var, Obj1, Obj2) :-
ruft(_, Obj1, Obj2),
benutzt(_, Obj2, Var).

ruft_rekursiv_mit_var(Var, Obj1, Obj2) :-
ruft_rekursiv_mit_var(Var, Obj1, X),
ruft(_, X, Obj2),
benutzt(_, Obj2, Var).

unterprogramme_via_flag(CName, Obj) :-
variable(Y, ´flag´),
c-funktion(Z, CName),
ruft_rekursiv_mit_var(Y, Z, Obj).
```

Welche Methoden aus den Implementierungen des Interfaces 'A' benutzen nicht die Schnittstelle 'B'?

Anfrage:
`methode_benutzt_nicht(´A´, ´B´, X).` Prolog-Regel:

```
methode_benutzt_nicht(IName1, IName2, Methode) :-
  interface(I1, IName1),
  interface(I2, IName2),
  implementiert(_, Klasse, I1),
  kann(_, Klasse, Methode),
  not( benutzt_rekursiv(Methode, I2) ).
```

Welche Methoden werden durch abgeleitete Klassen überschrieben?

In einem Projekt, in dem Methoden per Namenskonvention durch Klassenname::Methodenname benannt werden, fragt die folgende Anfrage nach allen Methoden, die in einer abgeleiteten Klasse überschrieben werden:

Anfrage:`same_method( M1, M2), ist_Methode_von(_,M1,K1), ist_Methode_von(_,M2,K2),erbt_rekursiv(K1,K2).`
Prolog-Regel:`:-[string].`

```
:-import str_length/2, str_substring/4 from string.

/* extrahiert den Teilstring ab der Position Pos bis zum Ende  */
str_substring(X,Pos,Y):-
   str_length(X,L),
   Pos<L,
   Rest is L-Pos,
   str_substring(X,Pos,Rest,Y).

/* liefert das letzte Feld, wobei Felder durch '::' getrennt sind */
lastfield(X,Y):-
   str_length(X,L),
   Start is L-2,
   lf(X,Z,Start),
   Z = Y.

lf(X,Y,Pos):-
   Pos >= 0,
   str_substring(X,Pos,2,S),
   S == '::',
   NewPos is Pos + 2,
   str_substring(X,NewPos,Y),!.

lf(X,Y,Pos):-
   Pos>0,
   NewPos is Pos - 1,
   lf(X,Y,NewPos).

lf(X,X,0).
```

```
same_method(X,Y):-
method(X,_,Name1),
method(Y,_,Name2),
X\==Y,
lastfield(Name1, L),
lastfield(Name2, L).
```

Anhang

Literaturverzeichnis

1. K.F. Sagonas, T. Swift, D. S. Warren, J. Freire, P.Rao: The XSB Programmer's Manual Version 1.8.
2. Michael A. Covington (1994): Natural Language Processing for Prolog Programmers. Englewood Cliffs, New Jersey: Prentice Hall
3. Yoav Shoham (1994): Artificial Intelligence Techniques in Prolog. San Francisco: CAL
4. Leon Sterling und E. Shapiro (1986): The Art of Prolog. Advanced Programming Techniques. Cambridge, Mass.; London, England: The MIT Press
5. Ramin Yasdi (1995): Logik und Programmieren in Logik. München

Links

Einführung in logische Programmiersprachen (und Prolog)	http://ivs.cs.uni-magdeburg.de/~dumke/PSK/LP.html
Prolog-FAQ	http://www.cs.uu.nl/wais/html/na-dir/prolog/faq.html
Eine kompakte Einführung in reguläre Ausdrücke	http://www.php.comzept.de/regexp.htm
Eine weitere Einführung in reguläre Ausdrücke, mit zugehöriger Theorie.	http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/

Syntax regulärer Ausdrücke

Regulärer Ausdruck	Passender Text
ein 'normales' Zeichen	ein 'normales' Zeichen
.	ein beliebiges Zeichen
Hintereinanderschreiben	Verkettung
|	Alternative
[...]	einer der aufgelisteten Zeichen
[^...]	ein Zeichen, das nicht aufgelistet ist
(...)	Gruppierungs-Operator

Regulärer Ausdruck	Passender Text
*	beliebig viele Vorkommen (des Voran-gehenden)
+	mindestens ein Vorkommen (des Voran-gehenden)
?	Null oder ein Vorkommen (des Voran-gehenden)

Prolog-Programm: shore_base.P

Definition der SHORE-Standardausdrücke

```
:- export object/3, object/1.
:- export document/3, document/1.
:- export relation/6, relation/4, relation/1.
:- export ist_definiert_in/2, kommt_vor_in/2.
:- export relationClosure/3.
:- export match/2.

:- import object_iter/5, object_get/5 from object.
:- import document_iter/5, document_get/5 from object.
:- import relation_iter/6, relation_get/6 from object.

:- import check_ist_definiert_in/3, get_document_to_object/2,
  object_iter_2/4, object_get_2/3 from object.
:- import check_kommt_vor_in/3, get_document_to_relation/2,
  relation_iter_2/4, relation_get_2/3 from object.


shore( _Function ) :- '_$builtin'(63).
shore( _Function, _Arg1 ) :- '_$builtin'(63).
shore( _Function, _Arg1, _Arg2 ) :- '_$builtin'(63).
shore( _Function, _Arg1, _Arg2, _Arg3 ) :- '_$builtin'(63).
shore( _Function, _Arg1, _Arg2, _Arg3, _Arg4 ) :- '_$builtin'(63).
shore( _Function, _Arg1, _Arg2, _Arg3, _Arg4, _Arg5 ) :- '_$buil-
tin'(63).
shore( _Function, _Arg1, _Arg2, _Arg3, _Arg4, _Arg5, _Arg6 ) :-
'_$builtin'(63).


/* -------------------- Objekte */
```

```
object(ObjId, Typ, Name, _, IterPos) :-
    (nonvar(ObjId); nonvar(IterPos)),
    shore('object_get', ObjId, IterPos, Typ, Name, Ok),
    nonvar(Ok).

object(ObjId, Typ, Name, IterId, _) :-
var(ObjId),
shore('object_iter', ObjId, Typ, Name, IterId, NextIterPos),
nonvar(NextIterPos),
object(ObjId, Typ, Name, IterId, NextIterPos).


object(ObjId, Typ, Name) :-
object(ObjId, Typ, Name, _, _).

object(ObjId):-
 object(ObjId, _, _, _, _).

/* -------------------- Dokumente */

document(DocId, Typ, Name, _, IterPos) :-
    (nonvar(DocId); nonvar(IterPos)),
shore('document_get', DocId, IterPos, Typ, Name, Ok),
    nonvar(Ok).

document(DocId, Typ, Name, IterId, _) :-
var(DocId),
shore('document_iter', DocId, Typ, Name, IterId, NextIterPos),
nonvar(NextIterPos),
document(DocId, Typ, Name, IterId, NextIterPos).


document(DocId, Typ, Name) :-
document(DocId, Typ, Name, _, _).

document(DocId):-
document(DocId, _, _, _, _).

/* -------------------- Relationen */

relation_(RelId, Typ, Source, Target, _, IterPos) :-
```

```
    (nonvar(RelId); nonvar(IterPos)),
shore('relation_get', RelId, IterPos, Typ, Source, Target, Ok),
    nonvar(Ok).

relation_(RelId, Typ, Source, Target, IterId, _) :-
var(RelId),
shore('relation_iter', RelId, Typ, Source, Target, IterId, NextIter-
Pos),
nonvar( NextIterPos ),
relation_(RelId, Typ, Source, Target, IterId, NextIterPos).

relation( RelId, Typ, SourceType, SourceName, TargetType, TargetName
):-
relation_(RelId, Typ, Source, Target, _, _),
( object( Source, SourceType, SourceName );
  document( Source, SourceType, SourceName ) ),
( object( Target, TargetType, TargetName );
  document( Target, TargetType, TargetName ) ).

relation(RelId, Typ, Source, Target) :-
relation_(RelId, Typ, Source, Target, _, _).

relation(RelId):-
relation_(RelId, _, _, _, _, _).

/* -------------------- ist_definiert_in */

ist_definiert_in(Objekt, _, _, IterPos):-
(nonvar(Objekt); nonvar(IterPos)),
shore('object_get_2', Objekt, IterPos, OK),
nonvar(OK).

ist_definiert_in(Objekt, Dokument, IterId, _):-
var(Objekt),
shore('object_iter_2', Objekt, Dokument, IterId, NextIterPos),
nonvar(NextIterPos),
ist_definiert_in(Objekt, Dokument, IterId, NextIterPos).

ist_definiert_in(Objekt, Dokument):-
nonvar(Objekt),
nonvar(Dokument),
```

```
shore('check_ist_definiert_in', Objekt, Dokument, OK),
nonvar(OK).

ist_definiert_in(Objekt, Dokument):-
var(Objekt),
nonvar(Dokument),
ist_definiert_in(Objekt, Dokument, _, _).

ist_definiert_in(Objekt, Dokument):-
nonvar(Objekt),
var(Dokument),
shore('get_document_to_object', Objekt, Dokument).

ist_definiert_in(Objekt, Dokument):-
var(Objekt),
var(Dokument),
object(Objekt,_,_),
ist_definiert_in(Objekt, Dokument).

/* -------------------- kommt_vor_in */

kommt_vor_in(Relation, _, _, IterPos):-
(nonvar(Relation); nonvar(IterPos)),
shore('relation_get_2', Relation, IterPos, OK),
nonvar(OK).

kommt_vor_in(Relation, Dokument, IterId, _):-
var(Relation),
shore('relation_iter_2', Relation, Dokument, IterId, NextIterPos),
nonvar(NextIterPos),
kommt_vor_in(Relation, Dokument, IterId, NextIterPos).

kommt_vor_in(Relation, Dokument):-
nonvar(Relation),
nonvar(Dokument),
shore('check_kommt_vor_in', Relation, Dokument, OK),
nonvar(OK).

kommt_vor_in(Relation, Dokument):-
var(Relation),
nonvar(Dokument),
```

```
kommt_vor_in(Relation, Dokument, _, _).

kommt_vor_in(Relation, Dokument):-
nonvar(Relation),
var(Dokument),
shore('get_document_to_relation', Relation, Dokument).

kommt_vor_in(Relation, Dokument):-
var(Relation),
var(Dokument),
relation(Relation),
kommt_vor_in(Relation,Dokument).

/* -------------------- relationClosure */

:- table(relationClosure/3).

relationClosure(Typ, Source, Target) :-
relation(_, Typ, Source, Target).

relationClosure(Typ, Source, Target) :-
nonvar(Target),
relationClosure(Typ, X, Target),
relation(_, Typ, Source, X).

relationClosure(Typ, Source, Target) :-
var(Target),
relationClosure(Typ, Source, X),
relation(_, Typ, X, Target).

/* --------------------- match (reguläre Ausdrücke) */

match( Regex, String ):-
nonvar( Regex ),
nonvar( String ),
var( Ok ),
shore( 'match', Regex, String, Ok ),
nonvar( Ok ).
```

Index

Symbole
[ ] 5
_ (Unterstrich) 4

A
Abfragen
  rekursive 6
Anfragen 5
Argumente 13
Atome 3
Aussagen 5

D
Datenbankreferenzen 16
Deklaration
  table 6

E
Endlosschleifen 6

F
Funktionssymbole 5

I
Individuelle Anfrage stellen 11

K
Klauseln 5
Konstanten 4

L
Liste 5
listutils 15

M
Metamodell 9, 15

P
Prädikate 5
  Bibliothek 15
  datenabhängige 15
  document 13
  match/2 14
  merge/3 15
  object 13
  relation 13
  relationClosure/3 14
  Übersicht vordefinierter 16
  vordefinierte 13
Programme 5

R
Rekursion 6

S
Sonderzeichen 3
Stelligkeiten 13

T
Taschenrechnerbeispiel 7
Terme 4
transitive Hülle 14

V
Variablen 4
  anonyme 4
Vordefinierte Prädikate 13

Z
Zahlen 4
Zeichenketten 3