Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2012,13
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

POMDP ModelBuilder

Gerhard Dirschl, Rainer Jäkel, Sven R. Schmidt-Rohr

2012

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Technischer Report

POMDP ModelBuilder

Gerhard Dirschl, Rainer Jäkel, Sven R. Schmidt-Rohr

dirschl@ira.uka.de, jaekel@ira.uka.de, srsr@ira.uka.de

Institut für Anthropomatik, Karlsruher Institut für Technologie (KIT)

1 Einleitung

In ein POMDP Modell fließen mehrere Modalitäten wie Position, Interaktion (Spracherkennung), Aktion des Menschen (Gesten usw.) ein. In einem faktorisierten Modell können diese Modalitäten als Zustandsmengen mit einer endlichen Anzahl von Elementen dargestellt werden. Der POMDP Solver erwartet jedoch ein flaches Modell, in dem es nur einen Zustandsraum für das Gesamtsystem gibt. Die Zustände in diesem flachen POMDP Modell entsprechen den Tupeln des kartesischen Produkts aller Modalitäten.

Der ModelBuilder generiert ein flaches POMDP Modell aus einer faktorisierten Beschreibung, den sogenannten MdlRules (svn: `srmodalbert/trunk/mdlrules/`). Das ModelBuilder Programm **buildmodel** (svn: `srmcalbert/trunk/Learner/GDLearner/ModelBuilder-v1`) erwartet eine MdlRule Datei als Eingabe und erzeugt mehrere Ausgabedateien: das POMDP Modell in verschiedenen Formaten und eine optionale FSMap. Folgende Formate für das POMDP Modell werden unterstützt:

- Svens Format (gut lesbar, solange die Anzahl der Zustände überschaubar ist).
- Tony Cassandras Dateiformat.
- PomdpX, ein von APPL unterstütztes XML Dateiformat, mit dem sich auch MOMDPs beschreiben lassen.

1.1 POMDP Datenstrukturen und Berechnungen

Zentrale Elemente des ModelBuilders sind Matrizen, mit denen die Belohnungsfunktion sowie die Observations- und Transitionswahrscheinlichkeiten des POMDP Modells berechnet werden. Eine Eingabedatei für den ModelBuilder besteht aus einer Folge von Befehlen, mit denen die Matrizen festgelegt werden. Die erzeugebaren POMDP-Modelle unterliegen einigen Einschränkungen.

- Die Beobachtungen sind von der ausgeführten Aktion unabhängig. Das Observationsmodell besteht also nur aus Wahrscheinlichkeiten $P(o \mid s)$
- Die Menge der Möglichen Beobachtungen ist gleich der Menge der möglichen Zustände, d.h. eine Beobachtung wird ebenfalls als Tupel des kartesischen Produkts aller Modalitäten dargestellt. Die Idee dahinter ist, dass der Agent seine möglichen Zustände beobachtet.

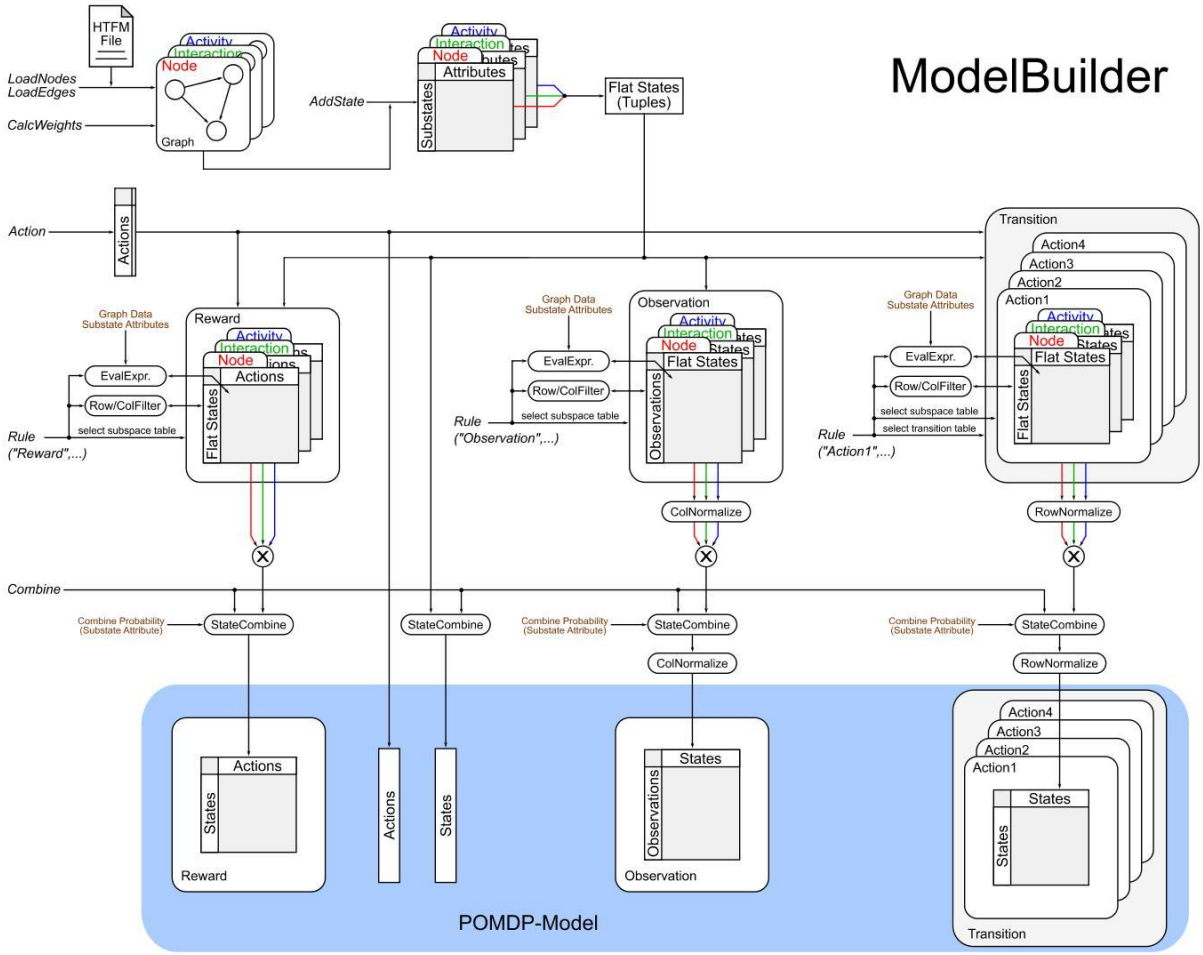

Abbildung 1: Grobe ModelBuilder Struktur

Es folgen einige Definitionen. Seien

- $S_1, \ldots, S_n$ Modalitäten (endliche, nichtleere Mengen von Teilzuständen),
- $S := S_1 \times \cdots \times S_n$ die Menge der Zustandstupel $s = (s_1, \ldots, s_n)$,
- $O := O_1 \times \cdots \times O_n = S_1 \times \cdots \times S_n$ die Menge der Beobachtungstupel,
- A eine endliche, nichtleere Menge von Aktionen,
- $\tilde{S}$ die Menge der Zustände des erzeugten POMDP Modells. $\tilde{S}$ und S werden unterschieden, da es der ModelBuilder erlaubt, mehrere Zustandstupel zu einem POMDP Zustand zusammenzufassen (*Combine*). Ein POMDP Zustand $\tilde{s} \in \tilde{S}$ kann als Äquivalenzklasse von $s \in S$ bezüglich der Äquivalenzrelation *combined* aufgefasst werden: $\tilde{s} := \{t \in S \mid \text{combined}(t, s)\}$.
- $\tilde{O}$ die Menge der Beobachtungen des erzeugten POMDP Modells (analog zu $\tilde{S}$),
- π_{S_i} eine Abbildung $S \to S_i$ mit $\pi_{S_i}(s) := s_i$ (Selektion der S_i-Komponente eines Tupels),

- $\tilde{\pi}_{S_i}$ eine Abbildung $\tilde{S} \to \mathcal{P}(S_i)$ mit $\tilde{\pi}_{S_i}(\tilde{s}) := \{\pi_{S_i}(t) \mid t \in \tilde{s}\}$ (die Menge aller Teilzustände aus S_i, die zum POMDP Zustand $\tilde{s}$ zusammengefasst werden),
- $\lambda_x(s_i)$ der (numerische) Wert des Attributs x des Teilzustands $s_i \in S_i$.
- M eine $l \times m$ Matrix mit symbolischen Zeilen- und Spaltenindizes. Dann ist
  - $M(r, c)$ der Eintrag der Zeile r und Spalte c.
  - $M(r) := (M(r, c_1), \ldots, M(r, c_m))$ der Zeilenvektor der Zeile r.
  - $\text{rnorm}(M)$ eine Funktion zu Normalisierung der Zeilensummen von M mit

$$\text{rnorm}(M)(r, c) := \begin{cases} \frac{M(r,c)}{\|M(r)\|_1} & \text{für } \|M(r)\|_1 \neq 0, \\ \frac{1}{m} & \text{sonst} \end{cases}$$

    rnorm ist nur für Matrizen ohne negative Einträge definiert.

Ein POMDP Modell wird mit Hilfe der folgenden Matrizen beschrieben, die wiederum durch MdlRule-Regeln festgelegt werden. Die Matrizen sind mit 0 initialisiert.

1.1.1 Belohnungsmatrizen

Für jede Modalität S_i gibt es eine Matrix R_{S_i} mit den Zeilen $s \in S$ (**nicht** $s \in S_i$) und Spalten $a \in A$. Aus diesen Matrizen wir die Reward Funktion des POMDP Modells wie folgt berechnet:

$$R(\tilde{s}, a) = \frac{1}{|\tilde{s}|} \sum_{t \in \tilde{s}} \left(\prod_{i=1}^{n} R_{S_i}(t, a) \right)$$

Wird *Combine* nicht verwendet:

$$R(\tilde{s}, a) = \prod_{i=1}^{n} R_{S_i}(s, a) \quad \text{mit } \tilde{s} = \{s\}$$

1.1.2 Transitionsmatrizen

Für jede Aktion $a \in A$ und jede Modalität S_i gibt es je eine Matrix $T^a_{S_i}$ mit den Zeilen (Ausgangszuständen) $s \in S$ und Spalten (Zielzuständen) $s' \in S$. Die Berechnung der Transitionswahrscheinlichkeit ist nur für Matrizen ohne negativen Einträge definiert:

$$P(\tilde{s}' \mid \tilde{s}, a) = \text{rnorm}(T^a)(\tilde{s}, \tilde{s}') \tag{1}$$

$$T^a(\tilde{s}, \tilde{s}') = \sum_{t \in \tilde{s}} \left(c_t \sum_{t' \in \tilde{s}'} \left(\prod_{i=1}^{n} \text{rnorm}(T^a_{S_i})(t, t') \right) \right) \tag{2}$$

$$c_t = \prod_{i=1}^{n} \text{comb}_{S_i}(t) \tag{3}$$

$$\text{comb}_{S_i}(t) = \begin{cases} \lambda_{\text{Combine}}(\pi_{S_i}(t)) & \text{für } |\tilde{\pi}_{S_i}(\tilde{t})| > 1 \\ 1 & \text{sonst} \end{cases} \tag{4}$$

Ohne *Combine* vereinfacht sich die Berechnung von (2):

$$T^a(\tilde{s}, \tilde{s}') = \prod_{i=1}^{n} \text{rnorm}(T^a_{S_i})(s, s') \quad \text{mit } \tilde{s} = \{s\} \text{ und } \tilde{s}' = \{s'\}$$

(1) Ergebnis normalisieren. (2) Normalisierte S_i-Matrizen eintragsweise multiplizieren und Zustandstupel zusammenfassen. Die Zielzustände (Spalten) können einfach summiert werden: $P(\bigcup_{t' \in \tilde{s}'} t' \mid t) = \sum_{t' \in \tilde{s}'} P(t' \mid t)$. Da die Wahrscheinlichkeiten $P(t, a)$ nicht bekannt sind, lassen sich die Zeilen nicht mittels $P(t' \mid \bigcup_{t \in \tilde{s}} t, a) = \frac{\sum_{t \in \tilde{s}} (P(t'|t,a)P(t,a))}{\sum_{u \in \tilde{s}} P(u,a)}$ zusammenfassen. Zur Lösung des Problems wird im ModelBuilder $\frac{P(t,a)}{\sum_{u \in \tilde{s}} P(u,a)}$ durch einen vom Anwender parametrierbaren Faktor c_t ersetzt. So können die zu einer POMDP-Transition zusammengefassten Tupel-Transitionen in Abhängigkeit von den Teilzuständen gewichtet werden. (3)+(4) Berechnung des Zeilengewichts c_t als Produkt der *Combine* Attribute der Tupel-Elemente von t. Dabei werden allerdings nur die Tupel-Elemente berücksichtigt, die mit mindestens einem weiteren Teilzustand derselben Modalität zusammengefasst werden.

1.1.3 Observationsmatrizen

Für jede Modalität S_i gibt es eine Matrix B_{S_i} mit den Zeilen $s \in S$ und Spalten $o \in O$. Die Beobachtungswahrscheinlichkeiten berechnen sich wie die Transitionswahrscheinlichkeiten. Der Vollständigkeit halber:

$$P(\tilde{o} \mid \tilde{s}) = \text{rnorm}(B)(\tilde{s}, \tilde{o})$$
$$B(\tilde{s}, \tilde{o}) = \sum_{t \in \tilde{s}} \left(c_t \sum_{o \in \tilde{o}} \left(\prod_{i=1}^{n} \text{rnorm}(B_{S_i})(t, o) \right) \right)$$

Die Zeilengewichte c_t sind mit denen der Transitionsmatrizen identisch. Ohne *Combine*:

$$B(\tilde{s}, \tilde{o}) = \prod_{i=1}^{n} \text{rnorm}(B_{S_i})(s, o) \quad \text{mit } \tilde{s} = \{s\} \text{ und } \tilde{o} = \{o\}$$

1.1.4 Unzulänglichkeiten

Das Zusammenfassen von Tupeln ist aufgrund des Zusammenspiels von *Combine* Regeln und *Combine* Attributen zu kompliziert, das Ergebnis nur schwer kontrollierbar. Die Zeilengewichtung ist zu unflexibel, da sie von der Aktion unabhängig ist, auch kann nicht zwischen Transitionen und Observationen unterschieden werden kann.

1.2 Graphen

Die in einer MdlRule Datei definierten Teilzustände einer Modalität können Attribute von den Knoten eines zuvor geladenen Graphen übernehmen. Dies funktioniert jedoch nur, wenn dem Graphen und der Modalität derselbe Name zugewiesen wird. Üblicherweise besitzen auch die Knoten und Teilzustände übereinstimmende Bezeichner. Auf die Kantenattribute eines Graphen kann in den Regeln zur Festlegung der Matrizeneinträge zugegriffen werden. In den nächsten Abschnitten werden folgende Abkürzungen im Zusammenhang mit Graphen verwendet:

- V^G bezeichnet die Menge der Knoten v_k des Graphen G
- E^G bezeichnet die Menge der Kanten (v_k, v_l) des Graphen G
- $\lambda_x((v_k, v_l))$ bezeichnet den Wert des Kantenattributs x der Kante (v_k, v_l)

1.3 FSMap

Zur Ausführungszeit berechnen verschiedene *SensFilter* laufend diskrete Wahrscheinlichkeitsverteilungen über Beobachtungsfeatures. Diese müssen zu einer Verteilung über der Menge der POMDP Zustände verrechnet werden (*Belief-State*). Die dazu benötigte Abbildung von Features zu POMDP Zuständen wird mittels der generierten *FSMap* Datei festgelegt. In den MdlRules können einem Teilzustand ein oder mehrere Features eines (einzelnen) SensFilters zugeordnet werden (*FSMap* Attribut). Der ModelBuilder weist dann jedem POMDP Zustand die Vereinigung der FSMap Attribute aller zugehörigen Tupel-Elemente zu.

2 Befehlssyntax und -beschreibung

2.1 Kommentare

Ein Kommentar wird mit einer Raute (#) eingeleitet (außerhalb eines Zeichenketten-Literals) und erstreckt sich bis zum Zeilenende.

2.2 Befehle

Ein Befehl kann sich über mehrere Zeilen erstrecken, muss aber immer mit einem Semikolon (**;**) abgeschlossen werden. Für die Bezeichner von Modalitäten, Teilzuständen und Attributen gibt es folgende Einschränkungen:

- Für Teilzustände sind rein numerische Bezeichner (“1”, “12” usw.) unzulässig. Sie müssen mit einem Präfix oder Suffix versehen werden.
- “Observation” und “Reward” sind als Aktionsbezeichner unzulässig.
- Die Zustandsattribute “Combine”, “FSMap” und “UseNode” sind reserviert (siehe 2.2.7).

2.2.1 SetName(name)

name : str

Setzt den Namen des POMDP-Modells und der Ausgabedatei. Fehlt dieser Befehl, so wird “model” als Name verwendet.

```
SetName("MyModel");
```

2.2.2 SetFSMapFile(filename)

filename : str

Legt den FSMap Dateinamen fest. Fehlt diese Angabe, wird keine FSMap Datei erzeugt.

```
SetFSMapFile("MyModel.fsmap");
```

2.2.3 LoadNodes(graph, filepath, attr1, *attr2*, ...)

graph : str – Graph-Bezeichner
filepath : str – HTFM Datei
attr1 : str – Attributname
attr2 : str (optional) – Attributname

Lädt alle Knoten sowie die angegebenen Attribute aus einer HTFM Datei und legt sie in einer internen Graph-Datenstruktur unter dem Namen **graph** ab. Alle angegebenen Attribute müssen für einen Knoten ermittelbar sein, ansonsten wird der Knoten ignoriert (Default-Attribute werden unterstützt).

```
LoadNodes("Node", "kiki.htfm", "Pose.X","Pose.Y");
```

2.2.4
LoadEdges(graph, filepath, *attr1*, ...)

graph : str – Graph-Bezeichner
filepath : str – HTFM Datei
attr1 : str (optional) – Attributname

Lädt alle Kanten sowie die angegeben Attribute aus einer HTFM-Datei und legt sie in einer internen Graph-Datenstruktur unter dem Namen **graph** ab. Alle angegebenen Attribute müssen für eine Kante ermittelbar sein, ansonsten wird die Kante ignoriert (Default-Attribute werden unterstützt).

```
LoadEdges("Node", "kiki.htfm");
```

2.2.5
CalcWeights(graph, resultIdentifier, attr1, *attr2*, ...)

graph : str – Graph-Bezeichner
resultIdentifier : str – Attributname für Ergebnis
attr1 : str – Vektorkomponente 1
attr2 : str (optional) – Vektorkomponente 2

Berechnet den euklidischen Abstand zwischen den mit einer Kante verbundenen Knoten des Graphen **graph**. Die ab dem 3. Parameter angegebenen Attribute bilden die Komponenten der jeweiligen Ortsvektoren. Das Ergebnis wird als Kantenattribut **resultIdentifier** in der Graph-Datenstruktur abgelegt.

```
CalcWeights("Node", "Length", "Pose.X", "Pose.Y");
```

2.2.6
Action(identifier)

identifier : str

Legt eine neue Aktion mit dem Namen **identifier** an.

```
Action("GotoCore");
```

2.2.7
State(stateSet, state, *attr:val*, ...)

stateSet : str – Modalität (Teilzustandsmenge)
state : str – Teilzustand
attr : str (optional) – Attributname
val : num (optional) – Attributwert

1. Erzeugt eine Modalität (Teilzustandsmenge) mit dem Bezeichner **stateSet**.

2. Fügt einen neuen Teilzustand **state** zu **stateSet** hinzu.

3. Schlägt den Namen des Teilzustands in der Graph-Datenstruktur **stateSet** nach. Falls ein passender Knoten gefunden wird, so werden dessen Attribute aus dem Graphen übernommen.

Über das optionale Schlüsselattribut ***UseNode*** kann ein vom Teilzustand-Bezeichner abweichender Knotenname angegeben werden.

4. Die optionalen (***attr:val***) Tupel setzen/überschreiben die numerischen Attribute des hinzugefügten Teilzustands. **Wichtig:** Alle Teilzustände einer Modalität besitzen dieselbe Menge von Attributen. Wird ein Attribut für einen Teilzustand nicht explizit festgelegt, so bekommt es den Wert 0.

Es gibt drei optionale Schlüsselattribute, die eine besondere Bedeutung haben:

- "***Combine***": Dieses Attribut erwartet einen numerischen Wert, der zur Berechnung der Zeilengewichte beim Zusammenfassen von Zustandstupel verwendet wird (siehe Abschnitt 1.1.2). Alle Teilzustände besitzen dieses Attribut. Wird es nicht explizit festgelegt, so hat es den Standardwert 1.0.
- "***FSMap***": Das Attribut erwartet eine Zeichenkette als Wert, der bei der Generierung der FSMap-Datei verwendet wird. Damit werden dem Zustand ein oder mehrere Beobachtungsfeatures eines SensFilters zugeordnet (siehe 1.3). Das Attribut hat normalerweise das Format **SENSFILTER::feature1:feature2:...:**
- "***UseNode***": Knotenname, der anstelle des Teilzustand-Bezeichners für den Zugriff auf die Graph-Datenstruktur verwendet wird. Die Substitution greift, wann immer ein Teilzustand für den Zugriff auf einen Graphen verwendet wird (in den nachfolgenden Abschnitten wird dies nicht mehr explizit erwähnt).

```
State("Node", "Other", "Pos.X":0, "Pos.Y":0, "Combine":0.5,
    "FSMap":"SELFLOC::sl_coreNW:sl_coreNE:");
```

2.2.8
Rule(action, s-Filter, s'-Filter, stateSet, *flag1*, ..., valueExpression)

action	: str – Aktionsbezeichner, oder ""
s-Filter	: str – Zustandsfilter (Ausgangszustand)
s'-Filter	: str – Zustandsfilter (Folgezustand)
stateSet	: str – Modalität oder ""
flag1	: str (optional) – Schlüsselwörter, die den Rule Befehl variieren
valueExpression	: expr – Ausdruck aus Konstanten, arithmetischen Operationen und Funktionen

Matrizenauswahl

Der Befehl manipuliert die Transitionsmatrizen $T^a_{S_i}$ (für die Observations- und Belohnungsmatrizen gibt es Varianten dieses Befehls). Die zu bearbeitende Transitionsmatrix wird mittels der Parameter **action** und **stateSet** ausgewählt.

- **action** akzeptiert eine leere Zeichenkette oder einen Aktionsbezeichner. Ein Aktionsbezeichner selektiert die Transitionsmatrizen $T^a_{S_i}$ für die Aktion $a =$ **action**. Die leere Zeichenkette wendet die Regel nacheinander für alle Aktionen $a \in A$ an.
- **stateSet** legt die Modalität (Teilzustandsmenge) S_i für diese Regel fest und vervollständigt damit unter anderem die Auswahl einer Matrix. Wird eine leere Zeichenkette angegeben, so wird die Regel nacheinander für alle Modalitäten $S_1, \ldots, S_n$ angewendet.

Zustandsfilter

Die zu bearbeitenden Einträge der Matrix werden durch die Zustandsfilter s-**Filter** und s'-**Filter** bestimmt.

- s-**Filter** legt die zu bearbeitenden Zeilen (Ausgangszustände) fest.
- s'-**Filter** legt die zu bearbeitenden Spalten (Zielzustände) fest.

Ein Zustandsfilter wird durch eine Zeichenkette definiert, einem Zustandstupel, dessen Zustandsbezeichner durch / getrennt werden. Ein leerer Bezeichner dient als Platzhalter für alle Teilzustände der entsprechenden Modalität. Wird einem Tupel-Element ein **!** vorangestellt, so wird die Auswahl für die entsprechende Modalität **invertiert** (alle Teilzustände außer dem angegebenen). Leere Tupelelemente am Ende der Zeichenkette werden ignoriert und können weggelassen werden, d.h. **"/foo//"** = **"/foo/"** = **"/foo"**. Jedoch ist **"/foo/"** ≠ **"foo/"**. Beispiele:

- **"/foo//bar"** entspricht $\{s \in S \mid \pi_{S_2}(s) = \text{“foo”} \ \wedge \ \pi_{S_4}(s) = \text{“bar”}\}$.
- **"foo//!bar"** entspricht $\{s \in S \mid \pi_{S_1}(s) = \text{“foo”} \ \wedge \ \pi_{S_3}(s) \neq \text{“bar”}\}$.
- **""**, **"/"**, **"//"** usw. sind gleichbedeutend und selektieren alle $s \in S$ (d.h. alle Zeilen bzw. Spalten)

Eine einzelne Zeile (oder Spalte) lässt sich alternativ auch über die Zeilen- bzw. Spaltennummer angeben (beginnend bei "1"). Die Nummer ist in Anführungszeichen zu setzen.

Zellenwert

valueExpression ist ein Ausdruck bestehend aus Konstanten, arithmetischen Operationen und Funktionen, der für jede zu bearbeitende Zelle ausgewertet wird und den alten Eintrag überschreibt. Sei $\mathcal{E}_{rc}(\texttt{a})$ die Auswertung des Ausdrucks `a` für die Zelle (r, c) (Zeile r, Spalte c). Zulässige Ausdrücke sind:

- Eine nummerische Konstante.
- Das interne Schlüsselwort `x`. Es wird bei der Auswertung des Ausdrucks durch den aktuellen Eintrag ersetzt. Formal: $\mathcal{E}_{rc}(\texttt{x}) = T^a_{S_i}(r, c)$
- Die Negation `-a`, wobei `a` ein Ausdruck ist.
- Die arithmetische Operation `op(a1␣a2)`, wobei `a1`, `a2` wiederum Ausdrücke sind und `op` einer der Operatoren `+`, `-`, `*`, `/` oder `%` (Fließkomma-Rest).
- Folgende Funktionen:

  `Equals(a)`
  erwartet einen Ausdruck als Parameter und ist definiert als

  $$\mathcal{E}_{rc}(\texttt{Equals(a)}) := \begin{cases} \mathcal{E}_{rc}(\texttt{a}) & \text{falls } \pi_{S_i}(r) = \pi_{S_i}(c) \\ 0 & \text{sonst} \end{cases}$$

  Die Funktion liefert also den Wert von `a`, falls die aktuellen Zeilen- und Spalten-Tupel aus dem gleichen S_i-Teilzustand bestehen und 0 sonst. Dabei ist nicht zu vergessen, dass die Funktion nur auf die zu s-**Filter** und s'-**Filter** passenden Zellen angewendet wird.

`EqualsSub("sset1/sset2/..."␣a)`

Der erste Parameter beschreibt eine Menge von durch / getrennten Modalitäten und `a` ist ein Ausdruck. Die Funktion ist eine Verallgemeinerung von `Equals`, bei der die entscheidenden Modalitäten durch den ersten Parameter und nicht durch **stateSet** festgelegt werden.

$$\mathcal{E}_{rc}(\texttt{EqualsSub}(M\ \texttt{a})) := \begin{cases} \mathcal{E}_{rc}(\texttt{a}) & \text{falls } \bigwedge_{m\in M}(\pi_m(r) = \pi_m(c)) \\ 0 & \text{sonst} \end{cases}$$

M ist die durch den ersten Parameter gegebene Menge von Modalitäten. Für $M = \{S_i\}$ ist EqualsSub identisch mit Equals.

`Gauss(["attr1"␣...␣"attrn"]␣[s1␣...␣sn])`

Die Funktion erwartet zwei n-dimensionale Vektoren als Parameter und liefert die Auswertung einer n-dimensionalen Gauss-Funktion (Dichtefunktion einer $\mathcal{N}(\mu, \Sigma)$ Verteilung). Der erste Vektor darf nur Attributnamen, der zweite nur numerische Werte als Komponenten enthalten. `Gauss` ist definiert als

$$\mathcal{E}_{rc}(\texttt{Gauss}(\texttt{["attr1"␣...␣"attrn"]␣[s1␣...␣sn]})) := f(x;\ \mu, \Sigma)$$

Der Erwartungswert μ wird aus den Attributen des Zeilen-Teilzustand $\pi_{S_i}(r)$ gebildet, die Auswertungsstelle x der Dichtefunktion f aus den Attributen des Spalten-Teilzustands $\pi_{S_i}(c)$. Die Elemente des Zufallsvektors werden als paarweise unkorreliert angenommen, ihre Standardabweichungen werden durch den zweiten Vektor $(s_1, \ldots, s_n)$ festgelegt. Die zugehörige Kovarianzmatrix Σ ist deswegen eine Diagonalmatrix.

$$\mu = \begin{pmatrix} \lambda_{\text{attr}_1}(\pi_{S_i}(r)) \\ \vdots \\ \lambda_{\text{attr}_n}(\pi_{S_i}(r)) \end{pmatrix} \qquad x = \begin{pmatrix} \lambda_{\text{attr}_1}(\pi_{S_i}(c)) \\ \vdots \\ \lambda_{\text{attr}_n}(\pi_{S_i}(c)) \end{pmatrix} \qquad \Sigma = \begin{pmatrix} s_1^2 & & \\ & \ddots & \\ & & s_n^2 \end{pmatrix}$$

`LinearFixed("targetState"␣"edgeAttr"␣m␣n)`

Der erste Parameter bezeichnet einen Knoten, der zweite ein Kantenattribut des Graphen S_i. Die Parameter `m` und `n` sind Ausdrücke. Mit LinearFixed lässt sich z.B. die Wahrscheinlichkeit, dass der Agent bei einer Fahr-Aktion zu früh stoppt, als lineare Funktion in Abhängigkeit von der Anzahl der verbleibenden Kanten bis zum Ziel modellieren.

Zuerst wird versucht, den kürzesten Pfad $p = (v_0, \ldots, v_k)$ zwischen dem Zeilen-Teilzustand $\pi_{S_i}(r)$ und **targetState** im Graphen S_i zu ermitteln ($\pi_{S_i}(r) = v_0 \in V^{S_i}$ und **targetState** $= v_k \in V^{S_i}$). Als Gewicht für die Berechnung von p dient das Kantenattribut **edgeAttr**. Das Ergebnis von `LinearFixed` berechnet sich dann wie folgt:

$$\mathcal{E}_{rc}(\texttt{LinearFixed("targetState"␣"edgeAttr"␣m␣n)}) := \begin{cases} \max(0,\ \mathcal{E}_{rc}(\texttt{m}) + \mathcal{E}_{rc}(\texttt{n}) \cdot (k-j)) & \text{falls } p = (v_0, \ldots, v_k) \text{ existiert und } \pi_{S_i}(c) = v_j \in p \\ 0 & \text{sonst} \end{cases}$$

Es ist zu beachten, dass nur die Anzahl der verbleibenden Kanten, aber nicht deren Gewichte in das Ergebnis eingehen (siehe dazu die Funktion `LinearRelative`).

`LinearRelative("targetState"␣"edgeAttr"␣m␣n)`

NOCH NICHT IMPLEMENTIERT

Im Unterschied zu `LinearFixed` geht bei dieser Funktion nicht die Anzahl der verbleibenden Kanten, sondern die Summe deren Gewichte in die lineare Gleichung ein.

$$\mathcal{E}_{rc}(\texttt{LinearRelative("targetState"␣"eAttr"␣m␣n)}) := \begin{cases} \max\left(0,\ \mathcal{E}_{rc}(\texttt{m}) + \mathcal{E}_{rc}(\texttt{n}) \cdot \sum_{l=j}^{k-1} \lambda_{\text{eAttr}}((v_l, v_{l+1}))\right) & \text{falls } p = (v_0, \ldots, v_k) \text{ existiert und } \pi_{S_i}(c) = v_j \in p \\ 0 & \text{sonst} \end{cases}$$

`Edge("attr")`

sucht die gerichtete Kante $(\pi_{S_i}(r), \pi_{S_i}(c))$ im Graphen S_i und liefert deren Attribut **attr**. Der Graph und das Kantenattribut müssen zuvor mit dem Befehl **LoadEdges** geladen worden sein. Wird der Graph, die Kante oder das Attribut nicht gefunden, liefert die Funktion 0 als Ergebnis.

$$\mathcal{E}_{rc}(\texttt{Edge("attr")}) := \begin{cases} \lambda_{\text{attr}}((\pi_{S_i}(r), \pi_{S_i}(c))) & \text{falls } (\pi_{S_i}(r), \pi_{S_i}(c)) \in E^{S_i} \text{ und } \lambda_{\text{attr}}((\pi_{S_i}(r), \pi_{S_i}(c))) \text{ existiert} \\ 0 & \text{sonst} \end{cases}$$

`Edge("gr"␣"attr")`

Variante der vorherigen Funktion, bei der der zu verwendende Graph nicht durch **stateSet**, sondern durch den ersten Parameter festgelegt wird (d.h. es wird überprüft, ob $(\pi_{S_i}(r), \pi_{S_i}(c)) \in E^{\text{gr}}$).

`RowVar("attr")`

Die Funktion liefert den Wert des **attr** Attributs des S_i-Teilzustands des aktuellen Zeilentupels.

$$\mathcal{E}_{rc}(\texttt{RowVar("attr")}) := \lambda_{\text{attr}}(\pi_{S_i}(r))$$

`RowVar("sset"␣"attr")`

Wie zuvor, jedoch wird hier der Teilzustand der als Parameter übergebenen Modalität verwendet.

$$\mathcal{E}_{rc}(\texttt{RowVar("sset"␣"attr")}) := \lambda_{\text{attr}}(\pi_{\text{sset}}(r))$$

`ColVar("attr")`

Die Funktion liefert den Wert des **attr** Attributs des S_i-Teilzustands des aktuellen Spaltentupels.

$$\mathcal{E}_{rc}(\texttt{ColVar("attr")}) := \lambda_{\text{attr}}(\pi_{S_i}(c))$$

`ColVar("sset"␣"attr")`

Wie zuvor, jedoch wird hier der Teilzustand der als Parameter übergebenen Modalität verwendet.

$$\mathcal{E}_{rc}(\texttt{ColVar("sset"␣"attr")}) := \lambda_{\text{attr}}(\pi_{\text{sset}}(c))$$

Flags

Mit den optionalen Flags kann das Verhalten des **Rule** Befehls modifiziert werden.

- "***ClearCols***"
  Ist dieses Flag gesetzt, so werden die **nicht** von einem s'-, o- oder a-Filter selektierten Spalten einer Zeile auf 0 gesetzt. Die Zeilenauswahl bleibt unbeeinflusst, d.h. nicht selektierte Zeilen werden nicht angerührt. Das Flag wird in Transitionsmatrizen häufig dafür verwendet, eine vorausgehende allgemeine Regel für bestimmte Ausgangszustände zu spezialisieren.

- "***AllMod***"
  Ist dieses Flag gesetzt, so wird **valueExpression** auf die Einträge der durch **stateSet** festgelegten S_i-Matrix angewendet. In den restlichen Matrizen ($S_j,\ j \neq i$) werden die entsprechenden Einträge dagegen auf 1 gesetzt.

  Hintergrund: Häufig ist es erwünscht, die von einer Regel erzeugten Werte möglichst direkt (nur normalisiert) als Transitionswahrscheinlichkeiten zu übernehmen. Damit einem dabei die eintragsweise Multiplikation der S_i-Matrizen nicht in die Quere kommt, müssen die entsprechenden Einträge der restlichen Matrizen auf 1 gesetzt werden. Ohne das ***AllMod*** Flag bräuchte man dazu deutlich mehr Regeln.

```
Rule("GotoCore", "PosManip/", "PosCore", "Node", "ClearCols", 0.99);
Rule("GotoCore", "PosManip/", "!PosCore/", "Node", 0.01);
Rule("GotoCore", "2", "Outer", "Node", +(*(2 x) 0.01));
```

2.2.9

Rule("Reward", *s*-Filter, *a*-Filter, stateSet, *flag1*, ..., valueExpression)

s-Filter	: str – Zustandsfilter
a-Filter	: str – Aktionsfilter
stateSet	: str – Modalität oder ""
flag1	: str (optional) – Schlüsselwörter, die den Rule Befehl variieren
valueExpression	: expr – Ausdruck aus Konstanten, arithmetischen Operationen und Funktionen

Diese Variante des Rule Befehls wird über das Schlüsselwort **Reward** anstelle eines Aktionsbezeichners identifiziert und dient der Manipulation der Belohnungsmatrizen R_{S_i}.

- **stateSet**, ***flag1*** und **valueExpression**: siehe 2.2.8.

- ***s*-Filter**: Zustandsfilter, der die zu bearbeitenden Zeilen (Zustandstupel) der Matrix bestimmt. Details zum Zustandsfilter unter 2.2.8.

- ***a*-Filter**: Bestimmt die zu bearbeitende Spalte (Aktion) der Matrix. Zulässige Eingaben sind entweder eine Spaltennummer (in Anführungszeichen, mit "1" beginnend), genau ein Aktionsbezeichner oder eine leere Zeichenkette. Die leere Zeichenkette selektiert alle Spalten. Einem Aktionsbezeichner kann ein **!** vorangestellt werden. Dadurch wird die Auswahl invertiert (d.h. es werden alle Aktionen mit Ausnahme der angegebenen selektiert).

```
Rule("Reward", "PosCore/CupPresent", "GotoTable", "Node", +(x 0.5));
```

2.2.10
Rule("Observation", *o*-Filter, *s*-Filter, stateSet, *flag1*, ..., valueExpression)

o-Filter	: str – Observationsfilter (= Zustandsfilter)
s-Filter	: str – Zustandsfilter
stateSet	: str – Modalität oder ""
flag1	: str (optional) – Schlüsselwörter, die den Rule Befehl variieren
valueExpression	: expr – Ausdruck aus Konstanten, arithmetischen Operationen und Funktionen

Diese Variante des Rule Befehls wird über das Schlüsselwort **Observation** anstelle eines Aktionsbezeichners identifiziert. Er dient der Manipulation der Observationsmatrizen B_{S_i}.

Wichtig: Aus Kompatibilitätsgründen sind bei dieser Variante der zweite und dritte Parameter vertauscht. Der zweite Parameter bestimmt die Spalten der Matrix, der dritte Parameter die Zeilen.

- **stateSet**, ***flag1*** und **valueExpression**: siehe 2.2.8.
- ***s*-Filter**: Zustandsfilter, der die zu bearbeitenden Zeilen (Zustandstupel) der Matrix bestimmt. Details zum Zustandsfilter unter 2.2.8.
- ***o*-Filter**: Zustandsfilter, der die zu bearbeitenden Spalten (Observationstupel) der Matrix bestimmt. Details zum Zustandsfilter unter 2.2.8.

```
Rule("Observation", "", "", "", Equals(1.0));
```

2.2.11
Combine(filter1, *filter2*, ...)

filter1	: str – Zustandsfilter
filter2	: str (optional) – Zustandsfilter

Ein **Combine** Befehl legt eine Menge zu kombinierender Zuständen fest (siehe Abschnitt 1.1). Die Auswahl der Zustände erfolgt über einen oder mehrere Zustandsfilter (siehe 2.2.8). Die Selektionen der Zustandsfilter eines Befehls werden vereinigt. Die durch mehrere **Combine** Befehle definierten Mengen müssen paarweise disjunkt sein.

2.3 Debugging

Mit den folgenden MdlRule Befehlen lassen sich ein paar der internen Datenstrukturen ausgeben.

2.3.1
DumpGraph(graph)

graph	: str – Graph-Bezeichner

Gibt die Knoten, Kanten und Attribute des Graphen **graph** aus, der mit den Befehlen **LoadNodes**, **LoadEdges** und **CalcWeights** eingelesen bzw. erweitert wurde.

2.3.2
DumpTable(matrixSelect, stateSet)

matrixSelect : str – Aktion, "**StateSet**", "**Reward**" oder "**Observation**"
stateSet : str – Modalität (Teilzustandsmenge)

Der Befehl gibt jeweils eine interne Matrix aus, die durch die beiden Parameter festgelegt wird.

- "**StateSet**": Mit diesem Schlüsselwort für **matrixSelect** werden die Teilzustände und deren Attribute der Modalität **stateSet** ausgegeben.
- "**Reward**": Mit diesem Schlüsselwort für **matrixSelect** wird die Belohnungsmatrix R_{stateSet} ausgegeben.
- "**Observation**": Mit diesem Schlüsselwort für **matrixSelect** wird die Observationsmatrix B_{stateSet} ausgegeben.
- Wird für **matrixSelect** ein Aktionsbezeichner angegeben, so wird die Transitionsmatrix $T^{\text{matrixSelect}}_{\text{stateSet}}$ ausgegeben.

3 Beispiele

Hier einige Beispiele, die die Wirkung verschiedener Rule Befehle veranschaulichen sollen. Die Zeilen wurden dabei nicht normalisiert. Leere Matrizeneinträge stellen eine 0 dar. "*" repräsentiert den alten Zellenwert, d.h. der Eintrag wurden vom Rule Befehl nicht verändert.

- 2 Modalitäten $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2\}$
- 1 Aktion "a"

```
Rule("a", "", "", "", Equals(1.0));
```

T_X^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1	1				
x_1y_2	1	1				
x_2y_1			1	1		
x_2y_2			1	1		
x_3y_1					1	1
x_3y_2					1	1

T_Y^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1		1		1	
x_1y_2		1		1		1
x_2y_1	1		1		1	
x_2y_2		1		1		1
x_3y_1	1		1		1	
x_3y_2		1		1		1

Die eintragsweise Multiplikation ergibt

T^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1					
x_1y_2		1				
x_2y_1			1			
x_2y_2				1		
x_3y_1					1	
x_3y_2						1

```
Rule("a", "", "", "X", EqualsSub("Y/X" 1.0));
```

T_X^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1					
x_1y_2		1				
x_2y_1			1			
x_2y_2				1		
x_3y_1					1	
x_3y_2						1

T_Y^a bleibt unverändert.

```
Rule("a", "x2/y1", "x2", "X", "AllMod", 0.1);
Rule("a", "x3/y1", "x2", "X", "AllMod", 0.8);
```

T_X^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	*	*	*	*	*	*
x_1y_2	*	*	*	*	*	*
x_2y_1	*	*	0.1	0.1	*	*
x_2y_2	*	*	*	*	*	*
x_3y_1	*	*	0.8	0.8	*	*
x_3y_2	*	*	*	*	*	*

T_Y^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	*	*	*	*	*	*
x_1y_2	*	*	*	*	*	*
x_2y_1	*	*	1	1	*	*
x_2y_2	*	*	*	*	*	*
x_3y_1	*	*	1	1	*	*
x_3y_2	*	*	*	*	*	*

Ohne das "AllMod" Flag bräuchte man 4 Regeln, um das gleiche Ergebnis zu erzielen.

```
Rule("a", "x2/y1", "x2", "X", 0.1);
Rule("a", "x3/y1", "x2", "X", 0.8);
Rule("a", "x2/y1", "x2", "Y", 1.0);
Rule("a", "x3/y1", "x2", "Y", 1.0);
```

```
Rule("a", "!x2", "/y1", "X", EqualsSub("Y" 1.0));
```

T_X^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1	*	1	*	1	*
x_1y_2		*		*		*
x_2y_1	*	*	*	*	*	*
x_2y_2	*	*	*	*	*	*
x_3y_1	1	*	1	*	1	*
x_3y_2		*		*		*

T_Y^a bleibt unverändert.

```
Rule("a", "!x2", "/y1", "X", "ClearCols", EqualsSub("Y" 1.0));
```

T_X^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1		1		1	
x_1y_2						
x_2y_1	*	*	*	*	*	*
x_2y_2	*	*	*	*	*	*
x_3y_1	1		1		1	
x_3y_2						

T_Y^a bleibt unverändert.

```
Rule("a", "!x2", "/y1", "X", "ClearCols", "AllMod", EqualsSub("Y" 1.0));
```

T_X^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1		1		1	
x_1y_2						
x_2y_1	*	*	*	*	*	*
x_2y_2	*	*	*	*	*	*
x_3y_1	1		1		1	
x_3y_2						

T_Y^a	$x'_1y'_1$	$x'_1y'_2$	$x'_2y'_1$	$x'_2y'_2$	$x'_3y'_1$	$x'_3y'_2$
x_1y_1	1		1		1	
x_1y_2	1		1		1	
x_2y_1	*	*	*	*	*	*
x_2y_2	*	*	*	*	*	*
x_3y_1	1		1		1	
x_3y_2	1		1		1	