

10.4.2006

## Handhabung der ATMS

(Am Beispiel von AtmsTest5.java)

Aufbau und Erläuterung der ATMS anhand der Testklassen:

Zuerst erzeugen wir eine neue Klasse, der wir einen Bezeichner geben.

```
public class AtmsTest5
{
    public static void main(String[] args)
    {
        ATMS example_atms = new ATMS();
        ...
    }
}
```

Der Konstruktor erzeugt eine neue ATMS, deren Knotenmenge bereits einen "FALSUM" Knoten enthält (welcher auf dem Datum "falsum" mit der ID 0 beruht).

Neue Knoten werden der ATMS anhand des jeweiligen Datums hinzugefügt. Deswegen erzeugen wir zunächst beispielhaft drei Objekte vom Typ Datum, wobei das erste Argument in dieser ATMS Version ein String ist, das zweite Argument eine laufende Nummer vom Typ "int", um die verschiedenen Daten voneinander zu unterscheiden (und letztendlich auch die AtmsNodes, AtmsEnvironments und die AtmsJustifications).

Eine notwendige Erweiterung, um die ATMS an den GA zu binden, muss eine Schnittstelle zwischen GA und ATMS sein. Dabei soll ein Datum, welches vom GA geliefert wird, mit einem jeweiligen AtmsNode aus der ATMS (und logischen Äquivalenten dieser Data) assoziiert werden. Diese Erweiterung könnte dann auch die Vergabe der ID eines Datums übernehmen.

```
...
Datum a = new Datum("a", 1);
Datum b = new Datum("b", 2);
Datum c = new Datum("c", 3);
...
```

Alle Knoten werden mittels `createNode(Datum datum)` bzw. `createAssumption(Datum datum)` der ATMS hinzugefügt, je nachdem, ob es sich bei dem Knoten um ein Datum oder eine Assumption handelt.

```
...
    example_atms.createAssumption(a);
    example_atms.createAssumption(b);
    example_atms.createNode(c);
...
```

Nach Ausführung dieser Methode befinden sich zwei Knoten in der Knotenmenge der ATMS, die von folgender Gestalt sind (kurz geschrieben):

$\langle X, \{X\}, \{(X)\} \rangle$

D. h. es handelt sich hierbei um eine Assumption, die als Vorgängerknoten sich selbst hat (wegen der Justification  $(X \rightarrow X)$ ) und als Label (Menge von Annahmen) die `AtmsEnvironment` enthält, die definiert wird durch die Menge von `X` selbst.

Weiterhin ist in der Knotenmenge der ATMS ein Knoten für das Datum "c" mit folgender Gestalt erzeugt worden:

$\langle c, \{\}, \{()\} \rangle$

D. h. hierbei handelt es sich um Knoten mit leerer Menge an `AtmsEnvironments` und leerer Menge an `AtmsJustifications`. Diese Mengen füllen sich beispielsweise dann, wenn der ATMS neue `AtmsJustifications`, also Begründungen, `X` zu glauben, mitgeteilt werden. Diese `AtmsJustifications` werden vom GA geliefert, wobei es noch zu klären ist, wie diese der ATMS mitgeteilt werden.

Des Weiteren muss noch beachtet werden, dass es bislang nur möglich ist, der ATMS `AtmsNodes` hinzuzufügen, nicht aber, `AtmsNodes` aus der ATMS zu entfernen. Es ist also noch notwendig eine Art Garbage-Collector zu implementieren, der nicht mehr benötigte `AtmsNodes` aus der ATMS löscht. Es ist die Aufgabe des Problem Solvers, also des GA, zu entscheiden, welche `AtmsNodes` nicht mehr verwendet werden müssen und folglich aus der ATMS gelöscht werden können.

Zur weiteren Bearbeitung müssen alle Knoten noch einmal (anhand des jeweiligen Datum) eingelesen werden.

```
...
    AtmsNode A = example_atms.getNode(a);
    AtmsNode B = example_atms.getNode(b);
    AtmsNode C = example_atms.getNode(c);
...
```

Um nun die Justifications einzufügen, wird eine neue ArrayList erzeugt. Die Elemente (AtmsNodes), die dieser ArrayList hinzugefügt werden, entsprechen den Vorgängerknoten (Antecedents) eines Knotens (Consequent).

Daraufhin wird eine neue AtmsJustification erzeugt, die als Argumente einmal die ArrayList der Antecedents (stellt logisch die Konjunktion der Vorgängerknoten eines Knotens dar) und als zweites den Consequent erhält. Zuletzt wird diese unserer ATMS hinzugefügt, und zwar mittels der Methode `addJustification(AtmsJustification just)`.

Am Beispiel der AtmsJustification  $A \wedge B \rightarrow C$  bedeutet dies:

```
...
    ArrayList all = new ArrayList();
    all.add(A);
    all.add(B);

    AtmsJustification justification1 =
        new AtmsJustification(all, C);

    example_atms.addJustification(justification1);
...

```

Fakten, welche keine Antecedents besitzen (aber aus der leeren Menge resultieren, also immer wahr sind), werden mittels des AtmsJustification-Konstruktors `AtmsJustification (AtmsNode node)` erzeugt.

Mittels des Konstruktors `AtmsJustification (AtmsNode antecedent, AtmsNode Consequent)` kann eine AtmsJustification erzeugt werden, deren Consequent nur einen Antecedent hat.

Der Algorithmus, welcher der Methode `addJustification` zugrunde liegt, berechnet aus dem jeweiligen Label jedes Antecedents eines Consequent das neue Label des Consequents. Diese Methode ist noch sehr ineffizient, da bei jedem Aufruf, also jedem Hinzufügen einer AtmsJustification, sämtliche Antecedents durchlaufen werden. Nachdem ein ggf. neues Label eines Consequent berechnet worden ist, muss diese Änderung all den Knoten mitgeteilt werden, die von dem Consequent selbst abhängen. Diese "Mitteilung" erfolgt bislang mit dem rekursiven Aufruf der `addJustification` Methode (mittels der `propagateNewLabels` Methode). Auch diese Vorgehensweise ist bislang ineffizient, da eine "Mitteilung" an die vom Consequent abhängenden AtmsNodes über die berechnete Änderung eines Knotens ausreichen müsste.

Nach Ausführung der obigen `addJustification` Methode ergibt sich nun folgendes für den Inhalt der Knotenmenge der ATMS:

```
< falsum, {}, {} >
< a, {{a}}, {(a)} >
< b, {{b}}, {(b)} >
< c, {{a, b}}, {(a, b)} >
```

Die nogood-Datenbank enthält alle AtmsEnvironments (Annahmemengen), welche sich gegenseitig ausschließen. Diese führen somit, wenn sie gleichzeitig gelten würden, zur Kontradiktion. Um beispielsweise zu erfassen, dass die Annahmen "a" und "not\_a" nicht gleichzeitig gelten können oder sollen, müssen für diesen Fall zwei AtmsAssumptions A und NOT\_A der ATMS hinzugefügt werden. (Und zwar wieder zunächst über Erzeugung des jeweiligen Datums a bzw. not\_a). Nach Erzeugung dieser beiden Assumptions muss der ATMS nun noch mitgeteilt werden, dass die Annahme von "a" und "not\_a" kontradiktorisch ist, und zwar, indem man folgende AtmsJustification der ATMS hinzufügt:  $A \wedge \text{NOT\_A} \rightarrow \text{falsum}$

D.h. es muss eine AtmsJustification mit dem Konstruktor

```
new AtmsJustification(Arraylist arraylist, FALSUM)
```

erzeugt werden, die der ATMS mittels addJustification hinzuzufügen ist.

Um falsum als Knoten zu benutzen, kann man ihn sich wie andere Knoten aus der ATMS holen:

```
...
Datum falsum = new Datum("falsum", 0);
AtmsNode FALSUM = example_atms.getNode(falsum);
...
```

Sollen sich nun beispielsweise die Annahmen a und b ausschließen:

```
...
ArrayList al2 = new ArrayList();
al2.add(A);
al2.add(B);
AtmsJustification justification2 =
    new AtmsJustification(al2, FALSUM)
example_atms.addJustification(justification_2);
...
```

Der Inhalt der ATMS lautet nun:

```
< falsum, {}, {(a, b)} >
< a, {{a}}, {(a)} >
< b, {{b}}, {(b)} >
< c, {}, {(a, b)} >
```

Nogood DB:  
{a, b},

Man beachte, dass die inkonsistente AtmsEnvironment {a, b} automatisch aus allen Labels aller AtmsNodes der ATMS entfernt wurde und der nogoodDB hinzugefügt worden ist.

Die hierbei hinzugezogene und verwendete Literatur:

- **Doyle, Jon: A Truth Maintenance System. *Artificial Intelligence* 12 (1979): 231-272**
- **de Kleer, Johan: An assumption-based TMS. *Artificial Intelligence* 28 (1986): 127-162 (online verfügbar)**

Für weitere Fragen wendet euch einfach an die WSV Arbeitsgruppe oder kontaktiert uns unter [2roeder@informatik.uni-hamburg.de](mailto:2roeder@informatik.uni-hamburg.de) bzw. [1zimmerm@informatik.uni-hamburg.de](mailto:1zimmerm@informatik.uni-hamburg.de)