**Softwaretechnik**
2002
Prof. Dr. H. Klaeren und M E Leypold

Eberhard Karls
Universität
Tübingen



# Spezifikation eines assoziativen Arrays in VDM-SL

## Inhaltsverzeichnis

# 1 Einleitung

Das folgende Handout stellt ein einfaches Beispiel für die Spezifikation in VDM-SL vor. Außer der Publikationssprache (mathematische Notation mit LaTeX-Schriftsatz), drucken wir zum Verständnis auch immer die Hardwaresprache (ASCII-Notation) mit ab.

Die in Schreibmaschinenschrift gesetzten Teile dieses Handouts stellen, wenn man sie in eine Datei abtippt, eine vollständige VDM-SL-Spezifikation dar, welche sowohl die Syntax- als auch die Typprüfung der VDMTools besteht.

Wir werden einen Datentyp spezifizieren, den viele Sprachen bereits nativ (in der Sprachdefinition) enthalten (Perl, Awk) oder der bei den meisten Sprachen durch eine Bibliothek verfügbar ist (C, C++). Die Rede ist von einem *assoziativen Array*, häufig auch "Hash" (Perl) genannt. Letztere Bezeichnung ist etwas unglücklich gewählt, da sie bereits den Implementationsmechanismus thematisiert, anstatt diese Frage offenzulassen (die C++ STL beispielsweise implementiert assoziative Arrays als Red-Black-Trees, aber eben nicht als Hash-Tabellen).

Das vorgestellte Beispiel ist ein sehr kleines Beispiel für die Spezifikation in VDM-SL, in dem nicht alle Konstrukte und Spezifikationstechniken zum Einsatz kommen. Es ist jedoch in der Struktur der Hausaufgabe (Spezifikation eines FiFo) sehr ähnlich.

# 2 Assoziative Arrays

Ein assoziatives Array erlaubt es, Daten eines bestimmten Typs unter einem Schlüssel abzulegen und später wieder unter diesem Schlüssel zu extrahieren. Assoziative Arrays sind Container für Daten.

Ich werde das hier spezifizierte assoziatives Array von nun an *Repository* nennen, um die von der Verwendung des Wortes "Array" ausgehenden Suggestionen zu vermeiden und so über die Operationen sprechen zu können, ohne ständig den Eindruck zu erwecken, redundante Aussagen zu machen.

Mindestens die folgenden Operationen werden benötigt:

**insert:** Mittels dieser Operation soll ein Datenelement *d* unter einem Schlüssel *k* im Repository gespeichert werden.

**retrieve:** Ein zuvor unter einem Schlüssel *k* gespeichertes Datenelement wird aus dem Repository extrahiert.

**remove:** Ein unter einem Schlüssel *k* gespeichertes Datenelement wird (unter Angabe des Schlüssels) gelöscht.

**test-contains:** Es wird getestet, ob unter einem bestimmten Schlüssel *k* ein Datenelement gespeichert ist.

**init:** Um das Repository in einen definierten Anfangszustand zu versetzen wird mindestens eine Operation zur Initialisierung benötigt.

# 3 Statische Beschreibung

Spezifikationen in modell-basierten Spezifikationsverfahren lassen sich immer in zwei Teile unterscheiden: Die Modellierung der Daten (statische Beschreibung) und die axiomatische Beschreibung der Operationen auf diesen Daten (diese Operationen entsprechen später in der Implementation Prozeduren).

Zuerst beschreiben wir die Daten, welche wir später im Repository speichern wollen, sowie die Schlüssel, unter denen die Daten gespeichert werden sollen. Über beide ist nichts Spezifisches bekannt (ja, es könnte sogar sein, dass beide Wertemengen überlappen oder identisch sind). In diesem Fall wird in VDM-SL der Typ *Token* verwendet (IFAD VDM-SL, 2000, S. 11).

```
types
  key   = token;
  data  = token;
```

types

1.0 $key = \text{token};$

2.0 $data = \text{token};$

Als Nächstes entwerfen wir ein Modell des (inneren) Zustand eines Repository. Ein Repository assoziiert jeweils Schlüssel und Datenelemente, wobei ein Schlüssel nur einmal in einem Repository vorhanden ist, jedoch ein Datenelement unter mehreren Schlüssel gespeichert sein kann. Dies legt nahe, dass wir den inneren Zustand des Repository durch eine Funktion modellieren könnten. In VDM-SL verwenden wir dazu eine Funktion mit einem endlichen Definitionsbereich:

```
  repository = map key to data;
```

3.0 $repository = key \xrightarrow{m} data$

Der Datentyp *Repository* repräsentiert die Menge aller (inneren) Zustände eines assoziativen Arrays.

Spezifizieren bedeutet in VDM-SL, eine Analogie aufzustellen. Die Aussage der obigen Definition ist im Grunde die, daß wir alle Operationen auf dem *Repository* verstehen können, wenn wir uns vorstellen, der innere Zustand des *Repository* bestünde aus eine Abbildung von der Menge der Schlüssel in die Menge der Datenelemente (oder aus einer entsprechenden Tabelle).

Das gewählte Modell suggeriert auch schon, wie die *Retrieve*-Operation spezifiziert werden kann. In Vorwegnahme von Dingen, die erst in den folgenden Abschnitten erklärt werden, soll sie hier schon einmal angegeben werden:

```
functions
  retrieve (r:repository, k:key) d:data
    pre    contains(r,k)
    post   d = r(k);
```

functions

4.0 $retrieve\,(r : repository, k : key)\ d : data$

.1 pre $contains\,(r,k)$

.2 post $d = r\,(k)$ ;

Das Prädikat *Contains* werden ich im folgenden Abschnitt nachliefern.

# 4 Attribute und Prädikate

Wie wir im letzten Abschnitt sahen, ist die Vorbedingung für eine *Retrieve*-Operation, dass unter dem betreffenden Schlüssel tatsächlich ein Datenelement gespeichert ist. Wir hätten diese Bedingung explizit in der *Pre*-Klausel vermerken können. Stattdessen definieren wir eine boolesche Funktion, die diese Bedingung wiedergibt:

```
contains: repository * key -> bool
contains  (r,k)       == k in set (dom r);
```

$$
\begin{array}{ll}
5.0 & \mathit{contains} : \mathit{repository} \times \mathit{key} \to \mathbb{B} \\
.1 & \mathit{contains}\,(r,k) \triangleq \\
.2 & \quad k \in (\mathsf{dom}\ r);
\end{array}
$$

Bedingungen in VDM-SL sind schlicht Ausdrücke mit booleschen Werten (das ist nicht bei allen Spezifikationssprachen so, insbesondere nicht bei denen, die auf Beweiskalkülen basieren, wie etwa Z).

# 5 Operationen 1

Operationen, die einen Zustand verändern, werden beschrieben durch eine Funktion, welche ein Element aus der Zustandsmenge (hier *Repository*) – den Anfangszustand – und die Eingangsparameter der Operation auf ein weiteres Element der Zustandsmenge – den Endzustand – und eventuell weitere Resultatwerte abbildet.

```
insert (r:repository, k:key, d:data) r':repository
  post      k in set (dom r')
       and  r'(k)=d                              -- [1]
       and  (dom r) \ {k} = (dom r') \ {k}        -- [2]
       and  forall i in set ((dom r)  \ {k})     -- [3]
                 & r'(i)=r(i) ;                  -- [4]
```

$$
\begin{array}{ll}
6.0 & \mathit{insert}\,(r : \mathit{repository}, k : \mathit{key}, d : \mathit{data})\ r' : \mathit{repository} \\
.1 & \mathsf{post}\ k \in (\mathsf{dom}\ r') \wedge \\
.2 & \qquad r'(k) = d \wedge \\
.3 & \qquad (\mathsf{dom}\ r) \setminus \{k\} = (\mathsf{dom}\ r') \setminus \{k\} \wedge \\
.4 & \qquad \forall\, i \in ((\mathsf{dom}\ r) \setminus \{k\}) \cdot \\
.5 & \qquad\quad r'(i) = r(i)\ ;
\end{array}
$$

Eine häufige Konvention ist es, den Endzustand durch dasselbe Symbol wie den Anfangszustand (hier $r$) zu bezeichnen, aber ergänzt um einen Strich (etwa $r'$).

# 6 Being Clever

Es gibt mehrere Möglichkeiten, die im letzten Abschnitt genannte Spezifikation von *Insert* zu verbessern, d. h. kürzer und verständlicher zu schreiben. Die Offenkundige ist die, über einen Teil der Nachbedingung zu abstrahieren und diese unter einer geeignet benannten booleschen Funktion zusammenzufassen.

Die Teile 3 und 4 der Nachbedingung dienten ja vor allem dem Zweck, festzulegen, dass alle Assoziationen zwischen Schlüsseln und Daten im Repository unverändert sein sollten, mit Ausname derer zum Schlüssel *k*. Wir können also ein Prädikat definieren, dass wir "*unchanged-but*" nennen, und mit dessen Hilfe wir genau diesen Sachverhalt ausdrücken können.

```
unchanged_but: repository * repository * set of key -> bool
unchanged_but  (r,r',ks) ==
                              (dom r) \ ks = (dom r') \ ks
                        and   forall i in set ((dom r) \ ks)
                                   & r'(i)=r(i) ;
```

7.0 $unchanged\text{-}but : repository \times repository \times key\text{-set} \to \mathbb{B}$
.1 $unchanged\text{-}but(r, r', ks) \triangleq$
.2 $(\text{dom } r) \setminus ks = (\text{dom } r') \setminus ks \wedge$
.3 $\forall i \in ((\text{dom } r) \setminus ks) \cdot$
.4 $r'(i) = r(i);$

Damit kann die *Insert*-Operation nun folgendermaßen geschrieben werden:

```
insert' (r:repository, k:key, d:data) r':repository
  post      k in set (dom r)
       and  r'(k)=d
       and  unchanged_but  (r,r',{k}) ;
```

8.0 $insert'(r : repository, k : key, d : data)\ r' : repository$
.1 $\text{post } k \in (\text{dom } r) \wedge$
.2 $r'(k) = d \wedge$
.3 $unchanged\text{-}but(r, r', \{k\})$ ;

Die weniger offenkundige Methode verwendet den "Map Override"-Operator (IFAD VDM-SL, 2000, S. 19), um im Prinzip denselben Sachverhalt zum Ausdruck zu bringen.

```
insert'' (r:repository, k:key, d:data) r':repository
  post r' = r ++ {k|->d} ;
```

9.0 $insert''(r : repository, k : key, d : data)\ r' : repository$
.1 $\text{post } r' = r \dagger \{k \mapsto d\}$ ;

Im Grunde bedeutet diese Formulierung bereits einen Schritt in Richtung Implementation. Zudem ist die Notation zwar kompakter, aber nur für den "Eingeweihten" lesbar.

Ich möchte hier noch bemerken, dass es relativ schwierig und vom Vorgehen vermutlich auch unsinnig ist, danach zu streben, gleich die perfekte Formulierung einer Spezifikation anzustreben. In der Praxis hat es sich bewährt, zuerst einmal alle bekannten Fakten (bzw. Desiderata) über das zu spezifizierende Objekt aufzuschreiben, so wie wir das für *Insert* getan haben. Nachdem man sich von der Vollständigkeit (und Korrektheit) der angegebenen Bedingungen überzeugt hat, kann man sich daran machen, diese in eine äquivalente, aber mathematisch elegantere und kürzere Form umzuschreiben, so wie wir das für die alternativen Versionen von *Insert* getan haben.

# 7 Operationen 2

Die Spezifikation der Operation *Remove* bietet nun nicht viel Neues. Wir greifen auf das Prädikat *unchanged-but* aus dem letzten Abschnitt zurück:

```
remove (r:repository, k:key) r′:repository
  post     k not in set (dom r′)
       and unchanged_but(r,r′,{k}) ;
```

10.0 $remove\,(r : repository, k : key)\ r' : repository$
.1 post $k \notin (\mathsf{dom}\ r') \wedge$
.2 $\quad unchanged\text{-}but\,(r, r', \{k\})$ ;

Der Zweck der Initialisierungoperation ist es, ein *Repository* in einen definierten Anfangszustand zu bringen.

```
empty  () r′:repository
  post (dom r′) = {} ;
```

11.0 $empty\,()\ r' : repository$
.1 post $(\mathsf{dom}\ r') = \{\}$ ;

Wie einleitend erwähnt, soll auch eine Funktion implementiert und spezifiziert werden, mit der getestet werden kann, ob zu einem Schlüssel eine Datenelement im Repository vorhanden ist. Das Prädikat (*contains*), mit welchem wir über diesen Sachverhalt reden können, haben wir bereits direkt definiert. Die zu implementierende Prozedur jedoch spezifizieren wir über die Angabe von Vor- und Nachbedingung.

```
test_contains (r:repository, k:key) b:bool
  post b=contains(r,k) ;
```

12.0 $test\text{-}contains\,(r : repository, k : key)\ b : \mathbb{B}$
.1 post $b = contains\,(r, k)$

# Literatur


[c.se FAQ ] *Comp.software-eng FAQ (Part 3): readings*. – `http://www.faqs.org/faqs/software-eng/part3`

[IFAD VDM-SL 2000] *VDMTools: The IFAD VDM-SL Language*. Forskerparken 10A, DK - 5230 Odense M, 2000. – URL `http://www.ifad.dk`

[VDM FAQ ] *Formal Methods Europe: VDM FAQ list*. – `http://www.cs.tcd.ie/FME/original/FAQ/vdm/Master.html`