Eberhard Karls
Universität
Tübingen

Lösungsskizze 1 : Spezifikation eines FiFo

Inhaltsverzeichnis

1 Einleitung

Im Folgenden gebe ich eine mögliche Spezifikation eines FiFo in VDM-SL. Zu jeder Deklaration wird einmal die ASCII- und einmal die mathematische Notation abgedruckt.

In der VDM-SL-Toolbox wird die Spezifikation immer in der ASCII-Notation eingegeben. Unter Unix wird die Ausgabe der mathematischen Notation auf dem Umweg über LaTeX unterstützt – das VDMTools Manual redet hier von *pretty printing*.

Weitere Informationen über diese beiden Notationen finden sich im Handbuch *IFAD VDM-SL Language* (IFAD VDM-SL, 2000, S. 161 f).

2 Statisches Modell

```
types
  job = token;
  fifo = seq of job;
```

types

1.0 $job = \text{token};$

2.0 $fifo = job^*$

Über den Typ *job* ist nichts bekannt, und wir benötigen auch keine weiteren Informationen. Er wird deswegen als *token* modelliert. Die einzige zur Verfügung stehende Operation auf Werten des Typs *token* ist der Test auf Gleichheit.

Dann müssen wir uns über die Menge aller möglichen inneren Zustände der Warteschlange Gedanken machen. Diese Menge konstituiert einen Typ – *fifo*. Ein Typ ist nämlich nichts weiter als eine Wertemenge, aus der die einzelnen Werte von Ausdrücken, Variablen oder Namen dieses Typs kommen können.

Wir *modellieren* die inneren Zustände der Warteschlange durch Folgen von Werten des Typs *job*. *Modellieren* bedeutet hier, dass wir alle Operationen auf den Zuständen der Warteschlange durch Funktionen auf diesem Typ (also auf Elementen aus dieser Wertemenge) erklären werden, und wir können uns den Effekt aller Operationen dadurch veranschaulichen, dass wir uns vorstellen, dass in der Warteschlange eine Sequenz von Werten des Typs *job* gespeichert ist.

Es bedeutet nicht, dass die Implementationssprache einen Datentyp *Liste* oder *Folge* zur Verfügung stellen muss. Dies wäre eine rein oberflächliche, syntaktische Entsprechung. Wie wir in späteren Aufgaben sehen werden, darf diese Sequenz z. B. durch ein Array implementiert werden – vorausgesetzt man schafft es, eine mathematische Äquivalenz zu zeigen (siehe Lösung und Anleitung zu Blatt 2, TeachSWT@Tü (2002a,d)).

3 Operationen

3.1 initf

```
functions
  initf () f':fifo
    post f' = [];
```

functions

3.0 $initf\ ()\ f' : fifo$
.1 post $f' = [\,]$;

Bei der Initialisierung wird der Zustand der Warteschlange von einem undefinierten (deshalb *kein* Eingabeparameter vom Typ *fifo*) in einen definierten, leeren, Zustand überführt.

3.2 extract

```
extract (f:fifo) f':fifo, j:job
  pre  (not isEmpty(f))
  post      j  = hd f
        and f' = tl f;
```

4.0 $extract\,(f : fifo)\ f' : fifo, j : job$
.1 pre $(\neg\, isEmpty\,(f))$
.2 post $j = \mathsf{hd}\, f \wedge$
.3 $f' = \mathsf{tl}\, f$;

Die Vorbedingung charakterisiert den Definitionsbereich der Prozedur: „Legaler“ Input sind alle Werte aus dem Wertebereich der Eingabetypen (*fifo*), welche die Vorbedingung erfüllen.

Die Nachbedingung charakterisiert für *jede einzelne legale* Eingabe indirekt die Menge der *möglichen* Resultate der Operation. Das Resultat muss nicht immer eindeutig (determiniert) sein, d. h. bestimmte Eingabewerte können das eine oder das andere Resultat liefern. (siehe auch Blatt 3 und die dazugehörige Lösung: TeachSWT@Tü, 2002b,e). Als Resultate sind alle Ergebnisse zugelassen, die für eine beliebige aber feste legale Eingabe die Nachbedingung erfüllen.

Hervorzuheben ist weiterhin, dass *fifo* sowohl in der Eingabe, als auch in der Ausgabe vorkommt, obwohl für die Implementation eher die folgende Prozedur

```
void extract(fifo* f, jobs::desc* jp);
```

oder die folgende Klassendefinition zu erwarten ist:

```
class Fifo {
public
  void extract(jobs::desc* jp);
  /* ... */
}
```

Dies rührt daher, dass wir es hier mit *Funktionen* zu tun haben, die nichtsdestotrotz die Prozeduren der späteren Implementation beschreiben sollen. Dies geschieht, indem die Funktion jeder legalen Eingabe der Prozedur eine mögliche Ausgabe bzw. ein mögliches Resultat zuordnet[1].

[1] Ich bin hier etwas ungenau: Eine Funktion ordnet natürlich einer Eingabe genau eine Ausgabe zu. Wir sprachen jedoch weiter vorne von mehreren möglichen Ausgaben. Der scheinbare Widerspruch erklärt sich daraus,

Der Aufruf der Prozedur *extract* hat als Ergebnis einen Jobdescriptor *j* und zugleich einen neuen Zustand der Warteschlange. Der neue Zustand ist also ein Ergebnis der Prozedur und muss deshalb unter den Resultaten der spezifizierenden Funktion auftauchen, hier als *f'*. Beide Ergebnisse *f'* und *j* sind abhängig vom anfänglichen Zustand der Warteschlange. Der anfängliche Zustand zählt also zu den Eingaben der Prozedur und muss unter den Eingaben der spezifizierenden Funktion auftauchen.

In VDM-SL gibt es (zuerst einmal) keinen Speicher und keine „Variablen". Die in der Spezifikation auftretenden Namen sind – ganz im Sinne der üblichen *mathematischen* Notation Namen für Werte (Elemente von Mengen), keine Namen für Speicherplätze (wie in imperativen Sprachen). VDM-SL ist eine mathematische Notation, keine Programmiersprache.

Weitere Informationen zu diesem Thema finden sich im begleitenden Handout zu Blatt 3: *Spezifikation durch Funktionen und die Behandlung von Speicher* (TeachSWT@Tü, 2002f).

3.3 enqueue

```
enqueue (f:fifo, j:job) f':fifo
  post f' = f^[j];
```

5.0 $enqueue\,(f : fifo, j : job)\; f' : fifo$
.1 post $f' = f \curvearrowright [j]$;

3.4 isEmpty

```
isEmpty (f:fifo) b:bool
  post b = ( f=[] );
```

6.0 $isEmpty\,(f : fifo)\; b : \mathbb{B}$
.1 post $b = (f = [])$

Eine direkte, explizite Definition wäre hier kürzer gewesen. Es erscheint mir jedoch suggestiver, Implementationsobjekte (d. h. Prozeduren) immer durch Vor- und Nachbedingungen zu charakterisieren, auch wenn eine explizite Definition möglich wäre. Dies bringt auch einige Vorteile, wenn die Darstellung (das Modell) gewechselt wird. Explizite Definitionen sollten reserviert bleiben für Hilfsfunktionen.

Literatur

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

[TeachSWT@Tü 2002a] LEYPOLD, M E.: Übungsblatt 2: Implementation eines FiFo. In: (TeachSWT@Tü, 2002c). – `handouts/uebung-02.tex`

[TeachSWT@Tü 2002b] LEYPOLD, M E.: Übungsblatt 3: Modellierung eines Heap in VDM-SL. In: (TeachSWT@Tü, 2002c). – `handouts/uebung-03.tex`

dass Vor- und Nachbedingungen *Mengen* von partiellen Funktionen charakterisieren (eben nicht nur eine) – eben alle Funktionen, deren Definitionsbereiche durch die Vorbedingung charakterisiert und deren Graphen die Nachbedingung erfüllen. Eine Prozedur ist eine Implementation der Spezifikation, wenn die von Ihr vollbrachte datenverarbeitende Leistung eine Abbildung aus dieser erwähnten Menge von Funktionen ist.

[TeachSWT@Tü 2002c] Wilhelm Schickart Institut (Veranst.): *Übungsmaterial, Fallbeispiele und Ergänzungen zur Softwaretechnikvorlesung 2002 in Tübingen*. Sand 13, D 72076 Tübingen, 2002. (Materialien zur Softwaretechnik). – URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `teachswt-tue-2002-a.tar.gz`

[TeachSWT@Tü 2002d] LEYPOLD, M E.: Lösungsskizze 2: Implementation eines FiFo. In: (TeachSWT@Tü, 2002c). – `handouts/loesung-02.vdm`

[TeachSWT@Tü 2002e] LEYPOLD, M E.: Lösungsskizze 3: Spezifikation eines Heap in VDM-SL. In: (TeachSWT@Tü, 2002c). – `handouts/loesung-03.vdm`

[TeachSWT@Tü 2002f] LEYPOLD, M E.: Spezifikation durch Funktionen und die Behandlung von Speicher. In: (TeachSWT@Tü, 2002c). – `handouts/functionale-spec-und-speicher.vdm`