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

Eberhard Karls
Universität
Tübingen

Von der Spezifikation zur Implementation

Inhaltsverzeichnis

Die vorliegende Text ist die weitgehend unveränderte Version, welche in der Vorlesung „Softwaretechnik“ im Sommer 2002 zum Einsatz kam. In der Zwischenzeit bin ich zu der Überzeugung gelangt, dass die Notation, die in diesem Handout verwendet wird, um Prädikate über die Laufzeitzustände von C-Programmen zu formulieren, verbessert werden kann und sollte. Zum einen sollte klar getrennt werden zwischen Notationen in C-Syntax (also syntaktische Konstrukten aus der Programmiersprache C) und Notationen in VDM-SL: Der Übergang zwischen beiden sollte deutlicher markiert werden. Zum anderen lässt sich die verwendete Notation präziser fassen, indem der Begriff des Laufzeitzustandes in einen formalen Rahmen eingebettet wird. Zu guter Letzt ist es auf jeden Fall nötig, Notationen vorzusehen, mit der man über Zeiger sprechen kann um Prädikate formulieren zu können, die über den topologischen Zusammenhang von Speicher zur Laufzeit Aussagen machen (etwa im Fall einer zirkulären Liste: Mit der vorhandenen Notation kann keine Darstellungsinvariante dieser Art formuliert werden).

All diese Verbesserungen sind im vorliegenden Text nicht eingearbeitet, kommen, werden jedoch zum Teil in der Lösungskizze zu Blatt 2 (TeachSWT@Tü, 2002) eingesetzt.

1 Einleitung

Ich möchte auf den folgenden Seiten demonstrieren:

- Wie man sich der Übereinstimmung zwischen Spezifikation und Implementation im Prinzip vergewissern kann,
- Wie man Nichtübereinstimmung dazu verwenden kann, systematisch Fehlerbehandlung zu konstruieren, und
- Wie man durch schrittweise Umformung eine Spezifikation näher an die Implementation bringen kann.

2 ADT *line* revisited

Als Beispiel habe ich eine vereinfachte Version des Datentyps Zeile aus dem Editor EFASS (siehe Vorlesung und andere Handouts zur Vorlesung) gewählt.

types

1.0 $line = appchar^*$

Der Datentyp *Appchar* und einige seiner Eigenschaften sind im Anhang A (Seite 12) definiert.

Ich beschränke mich auf die Betrachtungen der Operation *insert_char_after*.

functions

2.0 $\mathit{insert\text{-}char\text{-}after}(l:\mathit{line}, c:\mathit{appchar}, p:\mathbb{N})\ l':\mathit{line}$
.1 pre $p \geq 0 \wedge p \leq (\text{len } l)$
.2 post len $l' = (\text{len } l) + 1 \wedge$
.3 $\quad l'(p) = c \wedge$
.4 $\quad \forall i \in (\text{inds } l') \cdot$
.5 $\quad\quad i < p \Rightarrow$
.6 $\quad\quad l'(i) = l(i) \wedge$
.7 $\quad\quad i > p \Rightarrow l'(i+1) = l(i)$

3 Eine Implementation

Ich werde nun eine mögliche Implementation der im letzten Abschnitt zitierten Spezifikation vorführen. Eine Darstellung des abstrakten Datentyps *line* könnte in C/C++ folgendermaßen implementiert werden:

```
typedef struct{
  char* store;
  size_t  dlen;
  size_t  slen;
}                 /* ==> */     line  ;
```

Der Zeiger *store* zeigt auf einen auf dem Heap allozierten Speicherbereich (dessen Größe kann mit der Bibliotheksfunktion *realloc* bei Bedarf verändert werden). Die Komponente *slen* gibt an, wie groß der Speicherbereich ist (und muss immer korrekt mitgeführt werden, da es keine Funktion in den Systembibliotheken (Unix libc) gibt, mit der dies abgefragt werden kann). Der in *dlen* gespeicherte Wert gibt an, wie lang die im Puffer gespeicherte Zeile ist. Die ersten *dlen* Zeichen in *store[]* gehören zur Zeile. So muss die Größe des Speicherbereichs nicht nach jeder Operation angepasst werde (dies ist eine relativ teure Operation) sondern muss nur bei Bedarf vergrößert werden.

Für eine korrekt initialisierte Struktur dieses Typs muss immer $dlen \leq blen$ sein. Dies ist eine Invariante der implementierten Darstellung, die analog zu den Invarianten in Spezifikationen gesehen werden kann.

Ein mögliche Implementation der Operation *insert_char* könnte so aussehen:

```
void
insert_char(line* p, char c, size_t pos){
  size_t nlen;

  nlen = p->dlen+1;

  if (!(p->slen)>=nlen) {                    /* [1] */
    p->store=(char*)realloc(p->store, nlen);
    p->slen=nlen;
  }

  memmove(p->store+1+pos,p->store+pos,p->dlen-pos);
  p->dlen++;
  p->store[pos]=c;
}
```

Der mit [1] markierte Block dient dazu, die Invariante zu gewährleisten: Hier wird der allozierte Puffer vergrößert, wenn der Platz nicht ausreicht, die neue Zeile zu speichern.

„Genaues Hinschauen“ enthüllt mehrere Probleme mit dieser Implementation (abgesehen davon, dass der der klassische Fall des 0-Zeigers als Ergebnis von *malloc()* nicht abgefangen wird):

- Der Typ *size_t* ist größenbeschränkt. Die genaue obere Schranke *SIZE_T_MAX* hängt von der jeweiligen C-Implementation und der Zielmaschine ab. Sie ist aber für 16-Bit-Systeme so klein wie 32767.

- Der ANSI-C-Standard garantiert nicht, dass es möglich ist, Speicherobjekte, die größer als 32KB sind, zu verwenden. Dies trifft auch auf heapallozierte Speicherbereiche zu.

Auf beide Beschränkungen nimmt die vorliegende Implementation keine Rücksicht. Das Verhalten der Implementation ist undefiniert für den Fall, dass diese „magischen“ Grenzen überschritten werden. Die Implementation ist ganz klar mangelhaft.

Darüberhinaus gibt es in der Spezifikation keinen Hinweis, dass solche Beschränkungen existieren, noch, wie mit ihnen umzugehen ist.

Man könnte sich hier darauf berufen, dass sich die Implementation ja weitgehend (was immer das genau bedeutet) so verhält, wie es die Spezifikation vorsieht. In der Tat ist es ja sogar unmöglich, die vorliegende Spezifikation vollständig zu implementieren: Sie sieht überhaupt keine Beschränkung der Zeilenlänge vor, was auf einer realen Maschine praktisch nicht möglich ist.

Wir müssen uns jedoch die Frage stellen, wie wir solche Diskrepanzen systematisch finden können oder wie wir mit ihnen umgehen und sie eventuell berichtigen können.

4 Korrespondenz zwischen Spezifikation und Implementation

Eine Spezifikation muss man im Prinzip sehen als eine Aussage über ein zu implementierendes (oder bereits implementiertes) Stück Software. Entweder trifft die Aussage zu, dann implementiert die Software die Spezifikation, oder sie trifft eben nicht zu, dann ist die Software keine Implementation der Spezifikation.

Wie schon in anderen Handouts erwähnt, spezifiziert man mit modellbasierten Verfahren (z. B. VDM-SL) per Analogie: Auf eine Art, die wir noch näher erläutern müssen, wird eine Ähnlichkeit zwischen Spezifikation und Implementation behauptet, vor allem eine strukturelle Gleichheit zwischen den Elementen (Werten) der abstrakten Datentypen der Spezifikation und den konkreten Datenobjekten der Implementationssprache.

Die Vor- und Nachbedingungen charakterisieren axiomatisch Operationen auf den abstrakten Daten. Daraus müsste also eine Aussage über die Prozeduren der Implementation gewonnen werden können, die konform ist mit der strukturellen Analogie der Daten von Implementation und Spezifikation.

Was wir also im Folgenden entwickeln müssen, ist ein Verfahren, die strukturelle Äquivalenz zwischen Spezifikation und Implementation zu beweisen oder zumindest plausibel zu machen. Dort, wo diese Beweise nicht zu führen sind, liegen dann voraussichtlich die Implementationsgrenzen.

Die Distanz zwischen der eben formulierten Spezifikation und der vorgestellten Implementation ist allerdings (für den ersten Versuch) verhältnismäßig groß: Wir sehen im konkreten C++-Datentyp *line* mehr Komponenten als im abstrakten Datentyp *line*. Deshalb werde ich zuerst eine neue Spezifikation entwickeln, welche näher an der vorgestellten Implementation liegt.

Diese muss natürlich äquivalent zu der ersten Spezifikation sein. Auch hierfür werden wir eine Methode finden müssen, um das zu beurteilen. Günstigerweise ist diese Methode von derselben Art, wie die, welche wir dann später benötigen, um die Korrespondenz zwischen Spezifikation und Implementation zu aufzuzeigen.

5 Äquivalenz von Spezifikationen

Ich werde nun denn Datentyp *line* durch die ganze Spezifikation durch einen Datentyp *line*ρ ersetzen, der der Implementation ähnlicher ist. Dieser Ersetzungsprozess wird in der VDM-SL-Welt als *Reifikation* bezeichnet. In der Welt der Spezifikationssprache Z dagegen ist der Begriff *Data Refinement* üblich.

Wir modellieren die Eigenschaft, dass die Zeichen in einem Puffer gespeichert werden, Information über die Länge des Puffers und die Anzahl der gültigen Zeichen mitgeführt werden muss, und Zeichen nur in gültig allozierten Speicherbereichen abgelegt werden dürfen.

types

3.0 $line\rho :: store : appchar^*$
.1 $\quad dlen : \mathbb{N}$
.2 $\quad slen : \mathbb{N}$

.3 inv $l \triangleq$
.4 $\quad (\text{len } l.store) = l.slen \wedge$
.5 $\quad l.slen \geq l.dlen$

Nun muss sich eine Entsprechung zwischen den Elementen (Werten, Zuständen) des Datentyps *line*ρ und den Elementen des Datentyps *line*ρ finden – und mathematisch formulieren – lassen. Dies erledigen wir mit einer Funktion, welche jedem Element des neueren Datentyps ein Element des ursprünglichen Datentyps zuordnet.

functions

4.0 $retr\text{-}line : line\rho \rightarrow line$
.1 $retr\text{-}line\,(l\rho) \triangleq$
.2 $\quad subseq\,(1, l\rho.dlen, l\rho.store);$

Eine solche Funktion muss immer total und surjektiv sein, aber nicht notwendig injektiv. Sie wird *Retrieve-Funktion* genannt.

Bei genauerem Hinsehen ist zu erkennen, dass die Retrieve-Funktion auch die Interpretation des Inhaltes des Puffers *store[]* beschreibt: Nur die ersten *dlen* Zeichen werden auf das Ergebnis vom Typ *line* abgebildet.

*Line*ρ ist unser neues Modell für eine „Zeile“. Wir müssen nun eine Operation *insert-char-after*ρ auf diesem neuen Modell axiomatisch charakterisieren, und zwar so dass sie äquivalent ist zu der oben charakterisierten Operation *insert-char-after* auf dem alten Modell. Diese Beschreibung erhält man, indem man die Elemente des neuen Modells mittels der Retrieve-Funktion in das alte Modell abbildet, und dann die Vor- und Nachbedingungen des alten Modells anwendet.

5.0 $insert\text{-}char\text{-}after\rho''\,(l : line\rho, c : appchar, p : \mathbb{N})\ l' : line\rho$
.1 pre pre-$insert\text{-}char\text{-}after\,(retr\text{-}line\,(l), c, p)$
.2 post post-$insert\text{-}char\text{-}after\,(retr\text{-}line\,(l), c, p, retr\text{-}line\,(l'))$;

Das Zitieren der Vor- und Nachbedingungen des alten Modells ist allerdings etwas unbefriedigend: Schließlich möchte man am Schluss eine Spezifikation erhalten, die für sich selbst steht. Ich expandiere deshalb die ursprüngliche Definition des alten Modells und erhalte vorerst:

6.0 $insert\text{-}char\text{-}after\rho'\,(l : line\rho, c : appchar, p : \mathbb{N})\ l' : line\rho$
.1 pre $p \geq 0 \wedge p \leq (\text{len } retr\text{-}line\,(l))$

.2 post len *retr-line* $(l') = (\text{len } \textit{retr-line}\,(l)) + 1 \wedge$
.3 $\quad \textit{retr-line}\,(l')\,(p) = c \wedge$
.4 $\quad \forall\, i \in (\text{inds } \textit{retr-line}\,(l')) \cdot$
.5 $\quad\quad i < p \;\Rightarrow$
.6 $\quad\quad \textit{retr-line}\,(l')\,(i) = \textit{retr-line}\,(l)\,(i) \wedge$
.7 $\quad\quad i > p \;\Rightarrow\; \textit{retr-line}\,(l')\,(i+1) = \textit{retr-line}\,(l)\,(i)$;

Glücklicherweise lassen sich die Ausdrücke die *retr-line* enthalten, stark vereinfachen, wenn wir Eigenschaften von *subseq* und des VDM-SL-Datentyps *seq* anwenden:

7.0 *insert-char-after*ρ $(l : \textit{line}\rho, c : \textit{appchar}, p : \mathbb{N})\; l' : \textit{line}\rho$
.1 pre $p \geq 0 \wedge p \leq l.\textit{dlen}$
.2 post $l'.\textit{dlen} = l.\textit{dlen} + 1 \wedge$
.3 $\quad l'.\textit{store}\,(p) = c \wedge$
.4 $\quad \forall\, i \in \{1, \ldots, (l'.\textit{dlen})\} \cdot$
.5 $\quad\quad i < p \;\Rightarrow$
.6 $\quad\quad l'.\textit{store}\,(i) = l.\textit{store}\,(i) \wedge$
.7 $\quad\quad i > p \;\Rightarrow\; l'.\textit{store}\,(i+1) = l.\textit{store}\,(i)$;

Die so erhaltene Spezifikation über den Typ *line*ρ ist äquivalent zu der ursprünglichen über den Typ *line*. Voraussetzung dafür war die Existenz einer *Retrieve-Funktion*, d. h. einer *surjektiven* und *totalen* Abbildung vom neuen Datenmodell zurück in das alte.

6 Korrespondenz zur Implementation

Bei der Frage der Korrespondenz zwischen der Spezifikation und der Implementation kann man sich prinzipiell vom derselben Idee leiten lassen, wie bei der Reifikation: Zuerst definiert man mittels einer geeigneten Funktion die Entsprechung zwischen Datenkonstrukten der Implementation und den abstrakten Datentypen der Spezifikation und gewinnt dann mit Hilfe dieser Funktion Aussagen über die implementierten Operationen. Diese müssen zutreffen, damit die Implementation korrekt ist.

Ich werde diese Funktion hier *abstr-line*ρ nennen, da aus einem C-Konstrukt das dazugehörige Element der Wertemenge des abstrakte Datentyps *line*ρ ermittelt wird. Die Funktion abstrahiert also aus der konkreten Darstellung.

Zur obigen C-Implementation kann *abstr-line*ρ folgendermaßen definiert werden (wobei X ein C-Konstrukt vom Typ *line* ist):

$$\begin{aligned} \textit{abstr-line}\rho(\mathbf{X}) = l \quad \Longleftrightarrow \quad & l.\textit{dlen} = \mathbf{X.dlen} \wedge \\ & l.\textit{slen} = \mathbf{X.slen} \wedge \\ & \forall i \in [1..l.\textit{dlen}] \cdot \textit{ord}(l.\textit{store}(i)) = \mathbf{X.store[}\texttt{i - 1}\mathbf{]} \end{aligned}$$

Das formale Problem ist an dieser Stelle, dass sich die Funktion *abstr-line*ρ nicht innerhalb von VDM-SL definieren lässt, da C-Konstrukte nicht in VDM-SL enthalten sind. Ich helfen mir damit, indem ich eine gemischte Notation verwende, also C-Konstrukte und VDM-SL-Notation zusammen verwende, wo ich glaube, dass kein Raum zu Missverständnissen bleibt. Insbesondere verwende ich C-Konstrukte so, dass sie ganze Zahlen bedeuten und als solche auch in VDM-SL interpretiert werden können – und umgekehrt. C-Ausdrücke werde ich im Folgenden immer fett abdrucken, um sie von mathematischer oder VDM-SL-Notation zu unterscheiden.

Die Funktion *ord* ist übrigens eine Zuordnung zwischen dem Datentyp *appchar* und den natürlichen Zahlen, die im Anhang etwas genauer beschrieben ist.

Als nächstes müssen nun die axiomatischen Charakterisierungen des Modells in Beschreibungen über die von der Implementation erbrachten Berechnungsleistungen umgeformt werden: Die Vor- und Nachbedingungen werden zu Aussagen über den Zustand von Variablen, Speicherplätzen und über Funktionsergebnisse zu bestimmten Zeiten der Programmausführung.

Dabei ergibt sich allerdings das Problem, dass die C-Konstrukte Ihren Wert ja mit der Ausführung des Programmes ändern, wir also jeweils noch anmerken müssen, wann das betreffende Konstrukt ausgewertet werden soll. Ich werde dies durch einen Index tun, der Ort und eventuell Zeit der Auswertung anzeigt. Insbesondere wird im Folgenden der Index „0“ für Auswertung beim Eintritt in die Prozedur und „1“ für die Auswertung beim Austritt aus der Prozedur stehen.

Nach Anwendung der Abstraktionsfunktion *abstr-line*ρ erhalten wir die Vorbedingung

$$\mathbf{p} \geq 0 \wedge \mathbf{p} \leq \mathbf{(l\text{->}dlen)}$$

für die Prozedur *insert-char*.

Wenn man möchte, kann man das in diesem Fall direkt in ein *assert* Makro übersetzen:

```
assert(p>=0 && p <= l->dlen);
```

So etwas ist allerdings nicht immer mit vertretbarem Aufwand möglich. Als Nachbedingung für *insert-char* erhalten wir:

$$\begin{aligned}
\mathbf{(l\text{->}dlen)}_1 &= \mathbf{(l\text{->}dlen)}_0 + 1 \\
\mathbf{(l\text{->}store[p}_0\mathbf{])}_1 &= \mathbf{c}_0 \\
\forall i \in \{0,\ldots,\mathbf{p}_0\text{-}1\} \cdot \mathbf{l\text{->}store[}i\mathbf{]}_1 &= \mathbf{l\text{->}store[}i\mathbf{]}_0 \\
\forall i \in \{\mathbf{p}_0+1,\ldots,\mathbf{(l\text{->}dlen)}_1\text{-}1\} \cdot \mathbf{l\text{->}store[}i\mathbf{]}_1 &= \mathbf{l\text{->}store[}i\mathbf{]}_0
\end{aligned}$$

Dies ist im Wesentlichen wieder dieselbe Nachbedingung, die wir schon kennen, nur, dass wir hier über den Laufzeitzustand von Speicherobjekten der Sprache C bzw. C++ reden (statt über abstrakte Werte, d. h. Elemente von mathematischen Mengen).

Als wir *retr-line* formulierten, gaben wir zuerst die Invariante *inv-line*ρ an und mussten uns dann davon überzeugen, dass die Retrieve-Funktion total ist. Diesesmal müssen wir die Invariante des Modells in eine Invariante der Implementation übersetzen, *damit* wir *abstr-line*ρ als total auffassen können:

- Der zweite Teil der Invariante kann übersetzt werden zu

  $$\mathbf{l.slen} \geq \mathbf{(l.dlen)}$$

- Der erste Teil der Invariante $(\mathit{len}\ \mathit{l.store}) = \mathit{l.slen}$ dagegen bereitet mehr Probleme: Er drückt ja aus, dass der auf dem Heap allozierte Speicherblock, auf den **l.store** zeigt, die Länge **l.dlen** haben muss. Wir haben ja leider überhaupt kein C-Konstrukt, mit dem wir auf diese Eigenschaft des Heaps Bezug nehmen können. Hier bleibt uns nichts als eine verbale Formulierung.

  Eine Alternative wäre allerdings hier die explizite Modellierung des Heap.

Diese Invarianten müssen begriffen werden als Regeln, welche festlegen, wann der Speicherinhalt einer C-Struktur des Typs *line* tatsächlich eine Zeile im Sinne der abstrakten

Spezifikation kodiert – und wann wir umgekehrt den Inhalt der Variable nicht als Zeile interpretieren können.

Tatsächlich handelt es sich um die bereits im Abschnitt 3 (Seite 3) erläuterten Invarianten für die dort definierte C-Struktur *line*.

7 Der Bruch

Wo bleibt nun die anfangs angesprochene Diskrepanz zwischen Implementation und Spezifikation? Für $\mathit{abstr\text{-}line}\rho$ gelten dieselben Regeln wie für die Retrieve-Funktion: Damit die Entsprechung vollständig ist, muss die Funktion total und surjektiv sein. Gerade an Letzterem scheitert es aber: *abstrline* ist nicht surjektiv, es lassen sich leicht $l : \mathit{line}\rho$ finden mit $\mathit{len}\ l > \mathit{SIZE\_T\_MAX}$ oder welche andere Längenbeschränkung in der Implementationsumgebung des Programms auch immer existieren mag.

Die Implementation ist aber insoweit richtig, als sie, wenn (a) die Eingabe im C-Datentyp kodiert werden kann, und (b) das Resultat kodiert werden kann, genau die Leistung erbringt, die von der Spezifikation beschrieben wird. Insbesondere (b) ist aber nicht immer möglich, z. B. dann, wenn die als Eingabe in *Insert-char* eingehende Zeile *l* bereits maximale Länge hat.

Dieses Problem entsteht dadurch, dass die Spezifikation zu stark idealisiert (Datentyp mit unendlichem Wertebereich) und auf die Gegebenheiten der Zielumgebung (reale, endliche Maschine) keine Rücksicht nimmt.

Wir können damit auf drei verschiedene Arten umgehen:

1. Wir können die Bereiche undefinierten Verhaltens im Rahmen der Implementation als Implementationsbeschränkungen dokumentieren, ohne die Spezifikation zu ändern.
2. Wir können ad-hoc eine Fehlerbehandlung in die Implementation einfügen.
3. Wir können die Spezifikation so anpassen, dass sie zum einen soweit als möglich der idealisierten, ursprünglichen entspricht, aber zum anderen implementierbar wird.

Die Beschränkungen in der vorliegenden Implementation sind weitgehend harmlos: Sie sind zu weit weg von üblichen „Betriebsbedingungen“ des Programms, als dass sie wirklich stören würden. In diesem Fall bietet sich an, das erstgenannte Verfahren (Dokumentation bei Implementation) anzuwenden.

In anderen Fällen könnte die Situation jedoch komplex genug sein, um die Konsequenzen der in der Implementation hinzutretenden Einschränkungen weniger klar zu machen.

Und manchmal möchte man auch nicht, dass bei Verlassen des „sicheren“ Bereichs der Implementation undefinierte (und meist destruktive) Dinge geschehen: Fehlerbehandlung bzw. Behandlung von Ausnahmesituationen ist angesagt. Auch dies könnte ad-hoc geschehen (Verfahren 2), ich möchte jedoch vorführen, wie die Spezifikation systematisch so korrigiert werden kann, dass sich (praktisch automatisch) eine sinnvolle Behandlung der Ausnahmesituationen ergibt.

8 Spezialisierung

Die Eigenschaft der Zielmaschine (und Implementationssprache), dass nur Zeilen einer beschränkten Länge implementiert werden können, beschreibe ich ad-hoc durch eine Konstante, über deren genaue Grösse nichts bekannt ist, die aber deutlich grösser als 0,1,2,3, … ist.

```
8.0  memblk-maxlen() n : ℕ
.1   post n ≥ 127
```

Die Definition von *line*ρ schränke ich so ein, dass *abstr-line*ρ darauf surjektiv definiert werden könnte. Ich füge dazu eine weitere Bedingung zur Invariante *inv-line*ρ hinzu, mache die Invariante also restriktiver.

types

```
9.0  lineσ' :: store : appchar*
.1           dlen : ℕ
.2           slen : ℕ

.3   inv l ≜
.4     inv-lineρ (mk-lineρ (l.store, l.dlen, l.slen)) ∧
.5     l.slen ≤ memblk-maxlen();
```

Expandiert sieht die Typdefinition so aus:

```
10.0 lineσ :: store : appchar*
.1           dlen : ℕ
.2           slen : ℕ

.3   inv l ≜
.4     (len l.store) = l.slen ∧
.5     l.slen ≥ l.dlen ∧
.6     l.slen ≤ memblk-maxlen()
```

Der Typ *line*σ bedeutet lediglich die Einschränkung der Wertemenge von *line*ρ auf eine Untermenge. Auch hier kann die Korrespondenz (welches Element des einen Typs entspricht welchem Element des anderen Typs) wieder (wie zuvor mit der Retrieve-Funktion) durch eine geeignete Abbildung hergestellt werden. Diese fällt verhältnismäßig trivial aus:

functions

```
11.0 spec-lineρ : lineσ → lineρ
.1   spec-lineρ (l) ≜
.2     mk-lineρ (l.store, l.dlen, l.slen)
```

Allerdings ist *spec-line*ρ nicht surjektiv (wenn auch total) – das markiert eben den oben erläuterten Bruch zwischen der Idealisierung der ersten beiden Spezifikationen und den Beschränkungen der Zielmaschine. Dafür kann der neue Typ nun vollständig implementiert werden: Sein Wertebereich ist endlich.

Wenn nur die Spezifikation von *insert-char-after*ρ mit Hilfe von *spec-line*ρ in gleicher Weise transformiert werden würde, wie zuvor *insert-char-after* mittels *abstr-line*ρ transformiert wurde, um *insert-char-after*ρ zu erhalten, so erhielten wir Vor- und Nachbedingungen, bei denen die Nachbedingung und Invarianten nicht in allen Fällen erfüllt werden könnte, in denen die Vorbedingung erfüllt wäre.

Dies wäre zum Beispiel der Fall, wenn $l.dlen = memblk\text{-}maxlen()$ wäre. Die Bedingungen

$$\begin{aligned} l'.dlen &\leq l'.slen \\ l'.dlen &= l.dlen + 1 \\ l'.slen &< memblk\text{-}maxlen() \end{aligned}$$

sind unter diesen Umständen schlicht nicht gleichzeitig erfüllbar.

Das sollte eigentlich nicht überraschen, da wir die Wertemenge des Typs, auf dem wir hier operieren, auf eine Untermenge eingeschränkt haben. Bezüglich der Vorbedingung ist dies zuerst harmlos, aber in der Nachbedingung entsteht genau dann ein Problem, wenn die ursprüngliche Operation (*insert-char-after*ρ) aus der in Frage stehenden Untermenge hinausführt.

Es gibt wiederum zwei prinzipielle Strategien mit diesem Problem umzugehen:

1. Die Vorbedingung kann so eingeschränkt werden, dass sie einen Definitionsbereich der Operation beschreibt, in dem die Nachbedingung erfüllt werden kann.
2. Die Ergebnismenge der Operation kann um Werte, die bedeuten, dass die gewünschte Leistung nicht erbracht werden kann erweitert werden, und die Nachbedingung entsprechend modifiziert (erweitert) werden.

Die erste Methode bürdet die Last, zu erkennen, wann die Operation nicht durchgeführt werden kann, weil eine Beschränkung durch die Implementation vorliegt, dem Benutzer der Operation auf. Er hat hier eine entsprechend erweiterte Beweispflicht zu erfüllen, wenn er die Prozedur aufruft.

Die zweite Strategie läuft darauf hinaus, in eine Prozedur eine Fehler- oder besser Ausnahmebehandlung zu implementieren, die z. B. durch spezielle Resultatwerte oder Fehlerflags der aufrufenden Routine anzeigt, daß die Operation nicht ausgeführt werden kann, und überlässt der aufrufenden Routine die Entscheidung, was dann geschehen soll.

Es sei ausdrücklich darauf hingewiesen, dass beide Möglichkeiten eine „kaputtgegangene“ Spezifikation zu reparieren, ihre Meriten haben: Für welche man sich letztlich entscheiden sollte, hängt davon ab, wie die Prozedur verwendet werden soll und was im jeweiligen Kontext Sinn macht.

Im vorliegenden Fall möchte ich eine Fehlerbehandlung implementieren. Der entscheidende Punkt ist hier der, dass sich in einem Texteditor einfach nicht garantieren lässt, dass eine Zeile eine bestimmte Maximallänge nicht überschreitet – dieser Fall muss also explizit behandelt werden: Die Operation soll dann ein Fehlerflag zurückgeben, aber die Zeile unverändert lassen. Die aufrufende Prozedur kann dann dieses Fehlerflag an ihre aufrufende Prozedur hochreichen, oder einen Warnton für den Benutzer ausgeben, der diesen darauf hinweist, dass sein Tastendruck nicht angenommen wurde.

Zuerst definiere ich das Fehlerflag:

types

12.0 $errorflag = \mathbb{B}$

Die neue Spezifikation sieht nun so aus:

functions

13.0 $insert\text{-}char\text{-}after\sigma(l : line\sigma, c : appchar, p : \mathbb{N})\ l' : line\sigma, err : errorflag$

.1 pre $p \geq 0 \wedge p \leq l.dlen$

.2 post if $l.dlen < memblk\text{-}maxlen()$
.3 then $\neg\, err \wedge$
.4 $l'.dlen = l.dlen + 1 \wedge$
.5 $l'.store(p) = c \wedge$
.6 $\forall\, i \in \{1, \ldots, (l'.dlen)\} \cdot$
.7 $i < p \;\Rightarrow$
.8 $l'.store(i) = l.store(i) \wedge$
.9 $i > p \;\Rightarrow\; l'.store(i+1) = l.store(i)$
.10 else $(err \wedge l' = l)$;

Die Ergebnismenge von *Insert-char* wurde (im Vergleich zu *insert-char-after*ρ) erweitert, der Definitionsbereich eingeschränkt. Überall dort jedoch, wo dies im eingeschränkten Definitionsbereich möglich ist, erbringt *insert-char-after*σ die ursprüngliche Leistung, wo nicht, bleibt *line*σ unverändert und es wird ein Fehlerflag zurückgegeben.

Was hier vielleicht überflüssig zu bemerken ist: Natürlich ließe sich auch diese Spezialisierung auf eine eingeschränkte Untermenge in analoger Weise zu den bereits behandelten Transformationen formalisieren und stereotypisieren.

9 Eine sicherere Implementation

Eine Implementation mit Fehlerbehandlung, die die obige Spezifikation erfüllt, könnte so aussehen:

```
int
insert_char(line* p, char c, size_t pos){
  size_t nlen;

  nlen = p->dlen+1;

  if (!(p->slen)>=nlen) {
    if (nlen>=LINE_MAXLEN) return -1;

    p->store=(char*)realloc(p->store, nlen);
    p->slen=nlen;
  }

  memmove(p->store+1+pos,p->store+pos,p->dlen-pos);
  p->dlen++;
  p->store[pos]=c;

  return 0;
}
```

Literatur

[TeachSWT@Tü 2002] LEYPOLD, M E.: Lösungsskizze 2: Implementation eines FiFo. Sand 13, D 72076 Tübingen, 2002 (Materialien zur Softwaretechnik). – `handouts/loesung-02.vdm`

A Ergänzende VDM-SL-Deklarationen

14.0 $ord\,()\ m : appchar \xleftrightarrow{m} \mathbb{N}$

.1 post $m \neq \{\mapsto\} \wedge \forall n \in (\text{rng } m) \cdot n < 256$;

15.0 $APPCHARS\,()\ v :$ token-set

.1 post $v = \text{dom } ord\,()$

types

16.0 $appchar =$ token

.1 inv $c \triangleq c \in (APPCHARS\,())$

functions

17.0 $subseq\,(p : \mathbb{N}, l : \mathbb{N}, s : appchar^*)\ s' : appchar^*$

.1 pre $p + l - 1 \leq \text{len } s$

.2 post len $s' = l \wedge$

.3 $\forall i \in (\text{inds } s') \cdot s'(i) = s(i - p + 1)$

B Teilweise Implementation des ADT *line*

```
/*
  Partial implementation of the ADT 'line'.

  Adapted from: EFASS (editor for a small system).

  Copyright (C) 2002 M E Leypold.

  This file is distributed under the GNU GENERAL PUBLIC LICENSE.
*/

#include <sys/types.h> /* need size_t                    */
#include <stdlib.h>    /* need malloc() and friends      */
#include <string.h>    /* need memcpy() and friends      */

namespace lines {

  typedef struct{    /* PRIVATE. You're not supposed to  */
    char* store;     /* use any knowledge about the      */
    size_t  dlen;    /* components of this structure.    */
    size_t  slen;
  }                  /* ==> */     line  ;

  void
  insert_char(line* p, char c, size_t pos){
    size_t nlen;

    nlen = p->dlen+1;

    if (!(p->slen)>=nlen) {
      p->store=(char*)realloc(p->store, nlen);
      p->slen=nlen;
    }

    memmove(p->store+1+pos,p->store+pos,p->dlen-pos);
    p->dlen++;
    p->store[pos]=c;
  }
};
```