EFASS (A editor for a small system) Beschreibung und VDM-SL Spezifikation

Inhaltsverzeichnis

1 Stand dieser Spezifikation

1.1 Übersicht

EFASS ist ein kleiner, sehr einfacher Editor, der als Fallstudie für Spezifikation und modulare Implementation in C/C++ für die Vorlesung *Softwaretechnik* an der Universität Tübingen im Sommer 2002 entwickelt wurde.

Ich werde zuerst eine Übersicht über den Stand von Implementation und Spezifikation geben, Anomalien und Defekte ansprechen, sowie einige Bemerkung zu den mit EFASS verfolgten didaktischen Zielen machen. Im zweiten Teil dieses Dokuments werde ich die existierende Spezifikation von EFASS ohne weitere Kommentare abdrucken: Dieser Text und die Spezifikation sind als Arbeitsmaterial zu verstehen, nicht als vollständiges Tutorial darüber, wie man Applikationen der realen Welt formal spezifiziert.

1.2 Implementationssprache und Sprachmittel

Die Implementationssprache war nominell C++, tatsächlich entstammen aber fast alle eingesetzten Konstrukte der Programmiersprache C. Lediglich das *namespace*-Konstrukt wurde aus C++ entliehen, um eine Trennung der von den Modulen besetzten Namensräume zu erreichen.

Die Wahl der Sprachmittel liegt darin begründet, dass in der bereits genannten Vorlesung das Modulkonzept vor und unabhängig von den Begriffen der objektorientierten Programmierung eingeführt wurde. Dementsprechend wurde in der vorliegenden Implementation von EFASS auf die Verwendung objektorientierter Sprachmittel verzichtet – Editoren lassen sich schließlich auch in prozeduralen Sprachen implementieren.

Ebenso wurde auf den Einsatz der C++-*String*-Klasse und der Standard-Template-Library (STL) verzichtet. Beide Beschränkungen sind didaktisch motiviert. Die Implementation eines Datentyps *Zeile* mittels eines bereits vorhandenen Datentyps *String* stellt oberflächlich betrachtet keine softwaretechnisch interessante Leistung dar. Es ist zudem zu erwarten, dass die naive Implementation (mittels Aneinanderhängen von Strings im funktionalen Stil) zwar der Spezifikation syntaktisch ähnlich wäre, jedoch über alle Maßen ineffizient. Dagegen würde eine effiziente Implementation auf nicht-triviale Eigenschaften der Klasse String Bezug nehmen müssen (und wäre damit wieder softwaretechnisch interessant, allerdings nicht im Sinne einer Abstraktionsleistung, wie das für die EFASS-Implementation angestrebt wurde).

Bezüglich der STL gilt eine ähnliche Argumentation. Hinzu kommt, dass die ersten Übungsblöcke den formalen Methoden gewidmet waren. Ein besonderes Anliegen dabei war die Frage, wie Programme *systematisch* so konstruiert und beschrieben werden können, dass *Garantien* über ihr Verhalten abgegeben werden könnten: Mit der STL ist das wegen der ständig implizit stattfindenden Allokation und Reallokation von Speicher praktisch nicht möglich. Der Heapzustand ist als Nachbedingung solcher Operationen undefiniert genug, als dass auf ihm noch Garantien basiert werden können. Speicherbezogene Laufzeitfehler können unter diesen Bedingungen praktisch jederzeit auftreten und werden (vermutlich) durch *exceptions* signalisiert. Die saubere Abwicklung dieser Ausnahmebedingungen ist nichts, was man einem Anfänger als Erstes zumuten möchte. Zwar handelt es sich um Probleme, wie sie in der realen Entwicklung auch auftreten können, die auch mit geeigneten Methoden (speziellen Allokatoren, Wahrscheinlichkeitsüberlegungen) prinzipiell gelöst werden können, aber als Einstieg für Anfänger eignen sie sich schlecht.

1.3 Defekte der vorliegenden Implementation und Spezifikation

1.3.1 EFASS ist nicht perfekt

Die vorliegende Implementation und Spezifikation sind in einigen Punkten defizient (wenn auch viel besser, als die meiste real existierende Software). Ich möchte kurz erklären, wie es dazu kommt, und was unter Umständen in zukünftigen oder abgeleiteten Versionen verändert werden sollte. Diese Änderungen wurden einmal aus Gründen der „historischen Ehrlichkeit“, zum anderen aber auch aus Zeitmangel nicht mehr eingearbeitet. Es ist auch angemessener, dies mit einem konkreten Ziel zu tun, nämlich dann, wenn das Material für einen weiteren Kurs eingesetzt werden wird.

1.3.2 Defekte und Anomalien der Spezifikation

Für gewöhnlich unterhält man im Lehrbetrieb die Fiktion, dass *vor* der Implementation spezifiziert wird[1]. Der Editor EFASS hat aus praktischen Gründen den umgekehrten Weg genommen: Zuerst musste erprobt werden, ob ein einfacher Editor in für Anfänger übersichtlicher und verständlicher Weise implementiert werden kann[2], ehe man sich daran machte, diesen Editor auch zu spezifizieren. Dies hatte u. a. auch zur Folge, dass sich die Struktur der Spezifikation nahe an die Struktur der Implementation anlehnt, wie man sich anhand eines Vergleichs mit *Specification of a Display Editor* (Sufrin, 1981) überzeugen kann. In dieser Monographie erschafft sich der Autor erst ein allgemeines Vokabular um über „Text“ und Operationen auf Texten zu sprechen, ehe er dann die Anwendung spezifiziert.

Dagegen ist es in der vorliegenden Spezifikation so, dass fast jeder Funktion aus der Spezifikation eine Prozedur der Implementation entspricht. Lediglich sich wiederholende Teilterme sind ausfaktorisiert worden. Dies ist zwar eine brauchbare Dokumentation des Ist-Standes der Implementation, versperrt jedoch den Weg zu neuen implementationsunabhängigen Sichten auf das Problem. Ein gutes Beispiel für solche versäumten Chancen sind die Funktionen zum Laden und Speichern des Textes. Hier wurde in komplizierter und sehr mühevoller Art jedes Detail des Verfahrens nachgebildet, wie sich die Module *lines* und *texts* die Arbeit beim Einlesen des Textes aus einer Datei aufteilen. Im Nachhinein muss man sagen, dass das einfacher gegangen wäre: Es hätte genügt zu definieren, was eine korrekte Kodierung einer Zeile und, darauf basierend, eines Textes ist. Die Nachbedingung für das Einlesen eines Textes besteht dann im Wesentlichen – Komplikationen wegen eines eventuell fehlenden *newline* am Ende einer Zeile abgerechnet – einfach darin, dass (a) bis zum Zeilenende gelesen wird und (b) die gelesenen Bytefolge eine korrekte Kodierung des beim Einlesen erhaltenen Textes ist.

Dass es darüber hinaus auch eine Prozedur gibt, welche genau eine einzelne Zeile aus der Datei parst, wäre entweder eine Bequemlichkeit, die man sich dann bei der Implementation gönnt, oder Bestandteil eines späteren Entwurfsschrittes. Jedenfalls lassen sich für einen ersten Entwurf die Prozesse des Ladens und des Speicherns einer Datei einfacher und übersichtlicher charakterisieren.

[1] Dies vernachlässigt eigentlich auch, dass in der Praxis gerade Spezifikationen existierender Systemteile nützlich sein können (wenn sie möglich sind), nämlich um in eindeutiger Weise darüber zu reden, wie diese Systemteile beschaffen sind, und was sie leisten.

[2] Ich bin mir darüber noch nicht im Klaren: Da EFASS nur als Anschauungsmaterial nicht als Übungsmaterial zum Einsatz kam. Zuweilen hege ich den Verdacht, dass sich kein Teilnehmer die Mühe gemacht hat, EFASS Quelltext oder Spezifikation anzusehen, für die Mehrheit der Teilnehmer lässt sich das mit Sicherheit aussagen.

1.3.3 Defekte und Anomalien der Implementation

Auch die Art wie die einzelnen Module miteinander interagieren ist nicht immer günstig. Einige Prozeduren, wie z. B. *views*::*save*() und *views*::*load*(), reichen den Aufruf fast ohne eigene Beteiligung an das unterliegende Modul (hier *points*) durch. Der ursprüngliche Gedanke war, das unterliegende Modul vollständig zu verbergen. Inzwischen erkenne ich, dass das keine gute Idee war.

Nach der reinen Lehre sind solche Durchreicheprozeduren negativ zu bewerten. Andererseits plädiere ich dafür, dass es zuweilen wirklich nötig ist, Operationen auf dem Zustand des unterliegenden Moduls durch die übergeordneten Module zu kanalisieren. Ich will kurz versuchen klarzustellen, wann welche Methode anzuwenden ist, und wie nach meinem Dafürhalten *load*() und *save*() richtig implementiert worden wären. Zwar sind die Module hier als ADOs ausgeführt, jedoch gelten sinngemäß analoge Ausführungen auch für ADTs.

Das Modul *points* basiert auf dem Module *texts*, das Modul *views* wiederum auf dem Modul *points*. Gemeinsam ist in diesen Benutzungsbeziehungen, dass das übergeordnete Modul zum Zustand des jeweils untergeordneten Moduls etwas weiteren Zustand hinzufügt, und den Gesamtzustand in einer Abstraktion anbietet. Konsequente Abstraktion war der ursprüngliche Zweck der Durchreichefunktionen. Der zusätzliche Zustand ist in dem einen Fall die Position des Cursors, und im anderen Fall die Position eines Fensterausschnitts zur Durchsicht auf den unterliegenden Text. Beide supplementären Zustände sind mit dem Zustand, den sie ergänzen, durch Invarianten „verklammert“: Im ersten Fall ist die Invariante, dass der Cursor den Text nicht verlassen darf, im zweiten Fall, dass sich der Cursor innerhalb des sichtbaren Textteils, des Fensterausschnittes, befinden muss.

Wo es eine Invariante gibt, darf es, um programmtechnisch sicherzustellen, dass die Invariante nicht verletzt wird, keine separaten Operationen auf einzelne Teile des Zustandes mehr geben, stattdessen müssen Prozeduren zur Verfügung gestellt werden, die potentiell immer nur auf den Gesamtzustand wirken, und dies so tun, dass die Invariante als Nachbedingung gilt. Wären die hinzugekommenen Zustandsteile (in unserem Fall Cursorposition oder Fenster) selbst reichhaltiger, müsste man das wohl so machen, dass man diesen neuen Zustandsteil selbst als ADO (oder ADT) in einem separaten Modul mit eigenen Operationen versieht, und die beiden Module dann mittels eines dritten Moduls zu einer Einheit zusammenfasst. Die Prozeduren des dritten Moduls würden dann jeweils so auf den beiden unterliegenden Modulen operieren, daß die Invariante über beide Zustände erhalten bliebe.

Nun sind jedoch sowohl der Cursor selbst, als auch der Fensterausschnitt für sich betrachtet recht simple Zustände. Für sie selbst gibt es keine eigene Invariante. Die Mühe, beispielsweise den Cursor – und nur diesen allein (!) – in ein eigenes Modul zu abstrahieren, erscheint also zu Recht zu groß, weswegen der zusätzlich Zustand gleich in dem Modul (*points*) angesiedelt ist, das den neuen erweiterten Datentyp konstituiert. Dasselbe gilt für den Fensterausschnitt in *views*.

Übrigens dient der supplementäre Zustand in *points* dazu, die ausgeführten Operationen in andere Operationen auf *texts* zu übersetzen, während die Operationen auf *views* wirklich direkt an *points* übergeben werden. Es führt aber auch dort kein Weg daran vorbei, dass die Operationen durch das Modul *views* kanalisiert werden müssen, da nach der Operation auf *points* der supplementäre Zustand (die Fensterposition) unter Umständen korrigiert werden muss, so dass die Invariante wieder gilt (Cursor im Fenster).

Das Ergebnis der Diskussion der letzten Seiten lässt sich folgendermaßen zusammenfassen: Werden Zustände zu einem Gesamtzustand so zusammengefasst, so dass eine Invariante gilt, welche die Teilzustände in voneinander abhängiger Weise einschränkt, so muss der Gesamtzustand zu einem neuen ADO (bzw. ADT) abstrahiert werden, mit Operationen, die die Einhaltung der Invariante garantieren. Ist der hinzukommende Zustandsteil trivial, so werden die Operationen des erweiterten ADO denen des ursprünglichen recht

ähnlich sein, und es ist außerdem zweckmäßig, diese Operationen im gleichen Modul wie den supplementären Zustand anzusiedeln. Wegen der Ähnlichkeit der Operationen des erweiterten und denen des ursprünglichen ADO scheinen die Prozeduren dieses Moduls nur Durchreichefunktion zu haben – sie sind jedoch wegen der Invarianten unverzichtbar.

Die vorangegangenen Ausführungen gelten für Operationen, nicht jedoch für die Initialisierung (Konstruktoren), welche sich von den Operationen ja dadurch unterscheiden, dass vor dem Aufruf des Initialisierers der Gesamtzustand gar nicht existiert, also auch keine Invariante gelten muss. Man verliert also nichts, wenn man z. B. erst das eine Modul initialisiert (etwa *texts*) und dessen Zustand dann (explizit oder implizit) dem Initialisierer des anderen Moduls übergibt (z. B. *points*), welches dann in der Regel den initialen supplementären Zustand so einrichten kann, dass die Invariante gilt.

Nun sind im vorliegenden Fall die *load*-Prozeduren in den diversen Modulen eigentlich als Initialisierer angelegt, da sie nur einmal in der Programmabarbeitung aufgerufen werden, d. h. entsprechend den gemachten Ausführungen sollte es *keine* Durchreicheprozedur *points*::*load*() geben, stattdessen sollte die Initialisierungssequenz so ablaufen:

```
texts::load(filename);
points::init();
views::init();
```

Während also Operationen auf dem Zustand des untergeordneten Moduls wegen der Invariante verborgen werden müssen, gilt das für Initialisierer (aka Konstruktoren) nicht. Ein analoges Argument lässt sich für Deinitialisierer führen.

Auch Operationen, die lediglich einen Teilzustand abfragen (Attributfunktionen), müssen nicht zu einer Operation auf dem Gesamtzustand erhoben werden, wenn der Teilzustand Bestandteil des öffentlich gemachten Modells des Gesamtzustandes ist. Dies ist der Fall im Verhältnis von *texts* zu *points* – man sollte dort als eher *texts*::*save*() direkt aufrufen, und *points*::*save*() ersatzlos streichen.

Hätten wir es mit ADTs statt mit ADOs zu tun, müsste man sich noch zusätzlich darum bemühen, einen Verweis (Handle, Zeiger) auf den betreffenden Teilzustand zu bekommen, etwa so:

```
this_text_ref = points::get_text(this_point_ref);
texts::save(this_text_ref, filename);
```

Die eben aufgeführten Fälle von Durchreicheprozeduren (*load*, *save*, Initialisierung, Deinitialisierung) wurden erst im Laufe des Semesters mit der immer stärker zu Tage tretenden entwurfssteuernden Rolle der Invarianten als Anomalien erkannt, die in der nächsten Version von EFASS korrigiert werden sollten. Der Leser und hoffentlich interessierte Student von EFASS sollte dies im Auge behalten: Einige Eigenschaften von EFASS sind bewusster Entwurf, andere historischer Zufall und andere haben sich schlicht als Fehler herausgestellt, die in diesem Durchgang nicht mehr korrigiert werden.

1.4 Didaktischer Gehalt

1.4.1 Vorhandene Spezifikationsteile

Es wird hier hier keine vollständige Anleitung gegeben werden, wie EFASS in der Lehre (oder im Unterricht) am Besten eingesetzt werden kann. Dies muss sich nach den jeweilige Bedürfnissen richten und für allgemeine Regeln liegen noch nicht genügend Erfahrungen vor. Ich bin jedoch der Überzeugung, dass EFASS ein einfaches und gutes, wenn auch nicht perfektes Beispiel für modulare Programmierung ist, das auch von einem Anfänger verstanden und in Gänze überblickt werden kann, und das abstrakte Datentypen, welche über den üblichen Stack oder FiFo hinausgehen, in der Anwendung zeigt.

Ursprünglich war geplant, die gesamte abstrakte Maschine des Editors (also die Module *lines*, *texts*, *points*, *views*) und einen Teil des Eingabe-Tiers (*keyboard*, *events*) durchzuspezifizieren. Dieses scheiterte leider durch einen Fehler des verwendeten Werkzeugs[3], aufgrund dessen die Spezifikationen nochmals überarbeitet werden mussten. Dies konnte aus Zeitmangel nur für *texts* und *lines* geschehen. Was so überlebt hat, sind die Spezifikationsteile *sys*, *appchars*, *texts* und *lines*, die demonstrieren,

- wie man die „Außenwelt“, d. h. das System in dem die Applikation operiert, beschreiben kann (*sys* dies entspricht einem Teil der Ist-Analyse),
- wie man Kodierungsfragen klärt (*appchars*),
- wie man einen ADT spezifiziert (*lines*), und
- wie man einen abstrakten Datentyp aus einem anderen aufbaut (*texts*).

Im letzteren Fall lässt sich auch gut an der Spezifikation von Operationen wie *lines‘insert-char* und *lines‘remove-char* jenes Spezifikationsmuster beobachten, das in der Z-Community *Promotion* genannt wird (siehe Woodcock und Davies, 1996): Operationen auf einem Aggregatstyp (hier *text*), lassen sich in fast trivialer Weise durch Rückbezug auf die Operationen des Elementtyps spezifizieren, wobei ein zusätzlicher Parameter (*line-no*) als Selektor des zu manipulierenden Elements dient.

1.4.2 Potential noch nicht vorhandener Spezifikationsteile

Die Spezifikation von *points* und *views* würde kaum nennenswert Neues mit sich bringen, vielleicht mit der Ausnahme, dass es aufschlussreich wäre, zu versuchen, die Repositionierung des Fensterausschnittes in eine separate Strategie auszulagern[4].

Die noch unspezifizierten Module *events* und *display* bieten die Chance, zu demonstrieren, dass es durchaus möglich ist, reale Anwendungen auch in ihrem Gesamtverhalten vollständig und sinnvoll zu spezifizieren. Dies muss, so wie die Dinge stehen, künftigen Benutzern – ob Studenten oder Dozenten – dieses Materials überlassen bleiben. Nur soviel als Hinweis: Im Wesentlichen nimmt der Editor als Eingabe einen initialen Text, und eine Folge von Tastendrücken, und produziert dafür eine Folge von Anzeigen und einen finalen Text. Es ist durchaus möglich, sich hier auf die wesentlichen Ausschnitte zu konzentrieren, und lästige Details, wie die Rolle des Dateisystems, entweder später hinzuzufügen, oder ganz darauf zu verzichten, da man davon ausgeht, dass sich diese Dinge sowieso von allein verstehen und ihre detaillierte Spezifikation in der Praxis nicht lohnt.

[3] Die ansonsten ausgezeichneten VDMTools können, soweit ich feststellen konnte, die in einem Modul *B* zitierten Nachbedingungen von Operationen aus einem anderen Modul *A* nicht korrekt typ-prüfen.

[4] Man hat hier beispielsweise die Entscheidung zwischen minimalem Scrollen, wenn der Cursor das Fenster verlässt, und jeweils halb- oder viertelseitigem Scrollen: Ein schönes Beispiel dafür, wie ein sauberes Durchspezifizieren manchmal fast zwangsläufig auch zu den noch existierenden Freiheiten des Entwurfs hinführt.

1.4.3 Warnung vor VDM-SL *operations*

Ursprünglich war geplant, *points* und *views* in der *state/operation*-Syntax von VDM-SL zu beschreiben. Aufgrund der erwähnten technischen Problem musste darauf verzichtet werden und im Nachhinein möchte ich auch davon abraten, diese VDM-SL-Konstrukte bei der Arbeit mit Anfängern in formalen Methoden einzusetzen: Zum einen ist diese Notation nur syntaktischer Zucker für die funktionale Notation, zudem aber mit dem Mangel behaftet, dass sie eine bestimmte *Speicherstrategie* suggeriert, nämlich dass das Resultat eines Verarbeitungsprozesses die Eingabe des Prozesses *ersetzt*.

Im Laufe des Kurses hat sich aber bei mir die Überzeugung herauskristallisiert, dass man gut beraten ist, Entscheidungen über Speicherstrategien möglichst ans Ende des Entwurfsprozesses zu verlagern. Zum einen ist dies wirklich immer ohne großen Verlust möglich, zum anderen handelt es sich dabei um Entscheidungen, die vom Bedürfnis nach Effizienz bestimmt sind – und solche gehören soweit ans Ende eines Entwicklungsschrittes als möglich, um das eigentliche Kernanliegen nicht zu vernebeln.

Darüberhinaus konnte ich beobachten, dass viele Teilnehmer, die nur einen recht dürftigen (möglicherweise auch wieder vergessenen) mathematischen Hintergrund hatten, sich nur sehr schwer an den Gedanken gewöhnen konnten, dass VDM-SL keine Programmiersprache ist, dass die Ausdrücke, die in VDM-SL hingeschrieben werden, Funktionen (bzw. Funktionsfamilien) im mathematischen Sinne und logische Aussagen sind, jedoch keine Prozeduren, und dass Namen schlicht *Bezeichnungen für Werte* sind, keine *Variablennamen*. Da viele Teilnehmer nur imperative Programmiersprachen kannten, wäre die schiefe Ebene in diese „falsche“ Gedankenwelt einfach zu abschüssig, wenn in VDM-SL dann auch noch Zustände (*states*) und Operationen (*operations*) behandelt werden. Von der Verwendung diese Mittel in einem Anfängerkurs muss deshalb dringend abgeraten werden.

1.4.4 Assertions

Vor allem das Modul *points* ist relativ großzügig mit Assertions durchsetzt um zu illustrieren, wie die Vor- und Nachbedingungen der formalen Spezifikation verwendet werden können, um systematisch zur Laufzeit selbstprüfenden Code für eine Debugging-Version herzuleiten.

1.4.5 ADOs vs. ADTs

Balzert (1996) unterscheidet zwischen *abstrakten Datenobjekten* (*ADO*), das sind Module mit einem gekapselten Zustand, und *abstrakten Datentypen* (*ADT*), das sind gekapselte Daten, die nur über die Prozeduren eines bestimmten Moduls manipuliert werden können, von denen jedoch beliebig viele Instanzen in entsprechend definierten Variablen erstellt werden können. Wendet man die Definitionen auf die EFASS-Module an, so findet man, dass *lines* einen ADT exportiert, dagegen die Module *texts*, *views* und *points* jeweils als ADO ausprogrammiert sind.

Die Entscheidung, dies so zu implementieren fiel aus didaktischen Gründen: Es sollte der Umgang mit ADTs und ADOs sowohl in die Implementation (statische Variablen bzw.. programmexterne Zustände), als auch in der Spezifikation (kein Unterschied!) demonstriert werden[5].

[5] Eine aufschlussreiche, aber nicht sehr schwierige Aufgabe könnte darin bestehen, EFASS soweit als möglich ADTs umzuschreiben.

1.5 Der Umgang mit Speicher

Zum Schluss noch ein nötiges Wort zum Umgang mit Speicher in EFASS: Die vorliegende Implementation geht davon aus, dass unendlich viel Speicher vorhanden ist, und versucht gar nicht erst ein Fehlschlagen von Allokation zu behandeln.

Speicher ist, ebenso wie Prozessorzeit, eine fundamentale Ressource für ein Programm. Aus diesem Grunde muss sowieso die Frage gestellt werden, ob eine Behandlung eines solchen Ressource-Engpasses (ob temporär oder endgültig) überhaupt sinnvollerweise vom Programm selbst übernommen werden kann, oder besser vollständig im Betriebssystem geregelt werden sollte[6]. Zeitgenössische Betriebssystem jedoch sind in dieser Hinsicht sehr unkooperativ und bürden den Programmen die Notwendigkeit auf, auch temporäre Verknappungen der Ressource Speicher zu behandeln.

Wie gesagt, wird in EFASS überhaupt nicht versucht, mit einer Situation umzugehen, in der kein Speicher mehr alloziert werden kann. Ich will hier kurz einige Ansätze skizzieren, wie dieser Mangel behoben werden könnte. Das mindeste wäre wohl ein „ehrenvoller Abgang“ (graceful exit), oder alternativ das Blockieren aller Funktionen, die die Allokation weiteren Speichers erfordern. Damit Letzteres sinnvoll ist, muss die Funktion zum Speichern des Textes in eine Datei so implementiert werden, dass man dafür keinen weiteren Speicher mehr belegen muss. Der „ehrenvolle Abgang“ lässt sich schon durch Kapselung des *malloc*()-API in spezielle Prozeduren erreichen, die ein neues API zur Speicherbelegung bilden, das entweder immer erfolgreich Speicher belegt, oder gar nicht zurückkehrt („Terminieren durch die Hintertür“).

Wie man ein Programm zuverlässiger macht, indem man systematisch Stellen erkennt, an denen eine Programm in einen undefinierten Zustand übergehen könnte, wird im Handout *Von der Spezifikation zur Implementation* (TeachSWT@Tü, 2002c) und in der Lösungsskizze 2 (TeachSWT@Tü, 2002a) zur Softwaretechnik-Vorlesung 2002 (TeachSWT@Tü, 2002b) thematisiert.

[6] Vollkommen konfus wird die Situation, wenn man Memory-Over-Commit-Strategien diverser Betriebsystem mitberücksichtigen muss, die zwar eine Allokation immer gelingen lassen, jedoch wird, wenn denn zur Zeit des ersten Schreibens auf die (virtuelle) Speicherkachel keine reale freie Kachel mehr zur Verfügung steht, ein SIGSEGV-Signal an den Prozess geschickt. Meines Erachtens ist das genau die falsche Strategie, stattdessen sollte der Prozess angehalten werden, bis wieder eine Speicherseite (durch Beendigung anderer Prozesse) verfügbar wird. Ausnahmen müssten sich pro Prozess durch entsprechende Laufzeitoptionen regeln lassen, und wären wirklich nur für eine Minderheit von Programmen nötig (Systemdienste, Serverprozesse).

2 Systemanbindung

module *sys*

imports

1.0 from *seqtools* all

exports all

definitions

functions

2.0 $newline\,()\ c : c\text{-}char$

.1 post $(c = c)$;

3.0 $space\,()\ c : c\text{-}char$

.1 post $c \neq newline\,()$;

4.0 $ord\,()\ m : c\text{-}char \xleftrightarrow{m} \mathbb{N}$

.1 post $m \neq \{\mapsto\}$;

5.0 $chr\,()\ m : \mathbb{N} \xleftrightarrow{m} c\text{-}char$

.1 post $m = (ord\,()^{-1})$;

6.0 $C\text{-}CHARS\,()\ v :$ token-set

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

types

7.0 $c\text{-}char =$ token

.1 inv $c \triangleq c \in (C\text{-}CHARS\,())$

functions

8.0 $PrintableCHARS\,()\ v :$ token-set

.1 post $v \subset C\text{-}CHARS\,() \wedge$

.2 $\neg\,((newline\,()) \in v) \wedge$

.3 $(space\,()) \in v \wedge$

.4 $v \neq \{\}$;

9.0 $isPrintable\,(c : c\text{-}char)\ b : \mathbb{B}$

.1 post $b = (c \in (PrintableCHARS\,()))$;

10.0 $isNewline\,(c : c\text{-}char)\ b : \mathbb{B}$

.1 post $b = (c = newline\,())$

types

11.0 $stream = c\text{-}char^*$

values

12.0 $prefix = seqtools`prefix[c\text{-}char];$

13.0 $suffix = seqtools`suffix[c\text{-}char]$

functions

14.0 $write\,(s : stream, d : c\text{-}char^*)\ s' : stream$
.1 post $s' = s \frown d$;

15.0 $read\,(s : stream, n : \mathbb{N})\ s' : stream, d : c\text{-}char^*$
.1 pre $(\text{len } s) \geq n$
.2 post $d \frown s' = s \wedge (\text{len } d) = n$;

16.0 $EOF\,()\ s' : stream$
.1 post $s' = []$

end *sys*

3 Zeichen innerhalb der Applikation

module *appchars*

imports

17.0 from *seqtools* all ,

18.0 from *sys* all

exports all

definitions
functions

19.0 $encoding\,()\ v : \text{token} \xleftrightarrow{m} sys`c\text{-}char$
.1 post $(\text{rng } v) = sys`PrintableCHARS\,()$;

20.0 $APPCHARS\,()\ s : \text{token-set}$
.1 post $s = (\text{dom } encoding\,())$

types

21.0 $appchar = \text{token}$
.1 inv $a \triangleq a \in (APPCHARS\,())$

functions

22.0 $encode\,(s : appchar^*)\ s' : sys`c\text{-}char^*$
.1 post $s' = seqtools`mapped[appchar, sys`c\text{-}char](encoding\,(), s)$;

23.0 $decode\,(s : sys`c\text{-}char^*)\ s' : appchar^*$
.1 pre $(\text{elems } s) \subseteq (\text{rng } encoding\,())$
.2 post $s = encode\,(s')$;

24.0 $space\,()\ c : appchar$
.1 post $encoding\,()\,(c) = sys`space\,()$

end *appchars*

4 ADT Zeile

module *lines*

imports

25.0 from *appchars* all ,

26.0 from *seqtools* all ,

27.0 from *sys* all

exports all

definitions

types

28.0 $line = appchars`appchar^*$

values

29.0 $prefix = seqtools`prefix[appchars`appchar];$

30.0 $suffix = seqtools`suffix[appchars`appchar]$

functions

31.0 $empty : () \to line$

.1 $empty() \triangleq$

.2 $[];$

32.0 $length : line \to \mathbb{N}$

.1 $length(l) \triangleq$

.2 $(\text{len } l)$

values

33.0 $decodeable = \lambda s : sys`c\text{-}char^* \cdot$

.1 $\text{elems } s \subseteq sys`PrintableCHARS()$

functions

34.0 $completely\text{-}parsed : sys`stream \times sys`stream \times sys`c\text{-}char^* \to \mathbb{B}$

.1 $completely\text{-}parsed(s, s', d) \triangleq$

.2 $\forall c \in (\text{elems } d) \cdot$

.3 $\neg\, sys`isNewline(c) \wedge$

.4 $\text{let } n = \text{len } d \text{ in}$

.5 $(s' = sys`EOF() \wedge sys`\text{post-}read(s, n, \text{mk-}(sys`EOF(), d)) \vee$

.6 $sys`\text{post-}read(s, n+1, \text{mk-}(s', d \overset{\frown}{\to} [sys`newline()])));$

35.0 $could\text{-}be\text{-}parsed\text{-}from : sys`stream \times sys`c\text{-}char^* \to \mathbb{B}$

.1 $could\text{-}be\text{-}parsed\text{-}from(s, d) \triangleq$

.2 $\exists s' : sys`stream \cdot$

.3 $completely\text{-}parsed(s, s', d);$

36.0 $next\text{-}stream\text{-}section : sys`stream \to sys`c\text{-}char^*$

.1 $next\text{-}stream\text{-}section(s) \triangleq$

.2 $\iota p : sys`c\text{-}char^* \cdot could\text{-}be\text{-}parsed\text{-}from(s, p);$

37.0 $load\,(s : sys`stream)\ s' : sys`stream, l : line$
.1 pre $decodeable\,(next\text{-}stream\text{-}section\,(s))$
.2 post let $d = (appchars`encode\,(l))$ in
.3 $completely\text{-}parsed\,(s, s', d)$;

38.0 $save\,(s : sys`stream, l : line)\ s' : sys`stream$
.1 post $sys`$post-$write\,(s, s \frown appchars`encode\,(l) \frown [sys`newline\,()], s')$

types

39.0 $pos = \mathbb{N}$

functions

40.0 $addresses\text{-}boundary\text{-}in : line \times pos \to \mathbb{B}$
.1 $addresses\text{-}boundary\text{-}in\,(l, p) \triangleq$
.2 $p \geq 0 \wedge p \leq (\text{len } l)$;

41.0 $addresses\text{-}character\text{-}in : line \times pos \to \mathbb{B}$
.1 $addresses\text{-}character\text{-}in\,(l, p) \triangleq$
.2 $p \geq 1 \wedge p \leq (\text{len } l)$;

42.0 $insert\text{-}char\text{-}after\,(l : line, c : appchars`appchar, p : pos)\ l' : line$
.1 pre $addresses\text{-}boundary\text{-}in\,(l, p)$
.2 post $prefix\,(l, p) \frown [c] \frown suffix\,(l, (\text{len } l) - p) = l'$;

43.0 $remove\text{-}char\,(l : line, p : pos)\ l' : line$
.1 pre $addresses\text{-}character\text{-}in\,(l, p)$
.2 post $prefix\,(l', p - 1) \frown [l\,(p)] \frown suffix\,(l', (\text{len } l') - p) = l$;

44.0 $split\text{-}after\,(l : line, p : pos)\ l' : line, l'' : line$
.1 pre $addresses\text{-}boundary\text{-}in\,(l, p)$
.2 post $l' = prefix\,(l, p) \wedge$
.3 $l' \frown l'' = l$;

45.0 $merge\text{-}lines\,(l : line, l' : line)\ l'' : line$
.1 post $l'' = l \frown l'$

end *lines*

5 ADT Text

module *texts*

imports

46.0 from *appchars* all ,

47.0 from *lines* all ,

48.0 from *seqtools* all ,

49.0 from *sys* all

exports all

definitions

types

50.0 $text = lines`line^*$

functions

51.0 $addresses\text{-}a\text{-}line : \mathbb{N} \times text \to \mathbb{B}$

.1 $addresses\text{-}a\text{-}line(n,t) \triangleq$

.2 $seqtools`addresses\text{-}an\text{-}item[lines`line](n,t);$

52.0 $length : text \to \mathbb{N}$

.1 $length(t) \triangleq$

.2 len t;

53.0 $line\text{-}n : text \times \mathbb{N} \to lines`line$

.1 $line\text{-}n(t,n) \triangleq$

.2 $t(n)$

.3 pre $addresses\text{-}a\text{-}line(n,t)$;

54.0 $isDecodeable : sys`stream \to \mathbb{B}$

.1 $isDecodeable(s) \triangleq$

.2 $\exists l : lines`line, s' : sys`stream \cdot$

.3 $lines`\text{post-}load(s, \text{mk-}(s', l)) \wedge$

.4 $((s' = sys`EOF()) \vee isDecodeable(s'));$

55.0 $isDecoding : text \times sys`stream \to \mathbb{B}$

.1 $isDecoding(t,s) \triangleq$

.2 $\exists s' : sys`stream \cdot$

.3 $lines`\text{post-}load(s, \text{mk-}(s', \text{hd } t)) \wedge$

.4 $((s' = sys`EOF() \wedge t = []) \vee isDecoding(\text{tl } t, s'));$

56.0 $load(s : sys`stream)\ s' : sys`stream, t : text$

.1 pre $isDecodeable(s)$

.2 post $isDecoding(t,s) \wedge s' = []$;

57.0 $get\text{-}line(n : \mathbb{N}, t : text)\ l : lines`line$

.1 pre $\text{pre-}line\text{-}n(t,n)$

.2 post $l = \mathit{line\text{-}n}(t,n)$;

58.0 $\mathit{AppendedEncoding} : \mathit{sys}\text{`}\mathit{stream} \times \mathit{sys}\text{`}\mathit{stream} \times \mathit{text} \to \mathbb{B}$

.1 $\mathit{AppendedEncoding}(s,s',t) \triangleq$
.2 $(t = [] \wedge s = s') \vee$
.3 $\exists s'' : \mathit{sys}\text{`}\mathit{stream} \cdot$
.4 $\mathit{lines}\text{`}\mathsf{post}\text{-}\mathit{save}(s, \mathsf{hd}\ t, s'') \wedge$
.5 $\mathit{AppendedEncoding}(s'', s', \mathsf{tl}\ t)$;

59.0 $\mathit{save}(s : \mathit{sys}\text{`}\mathit{stream}, t : \mathit{text})\ s' : \mathit{sys}\text{`}\mathit{stream}, t' : \mathit{text}$
.1 post $t = t' \wedge \mathit{AppendedEncoding}(s,s',t)$

values

60.0 $\mathit{prefix} = \mathit{seqtools}\text{`}\mathit{prefix}[\mathit{lines}\text{`}\mathit{line}]$;

61.0 $\mathit{suffix} = \mathit{seqtools}\text{`}\mathit{suffix}[\mathit{lines}\text{`}\mathit{line}]$

functions

62.0 $\mathit{unchanged\text{-}except} : \mathit{text} \times \mathit{text} \times \mathbb{N}\text{-set} \to \mathbb{B}$
.1 $\mathit{unchanged\text{-}except}(t,t',s) \triangleq$
.2 $\forall i : \mathbb{N} \cdot$
.3 $i \notin s \Rightarrow$
.4 $(i \in (\mathsf{inds}\ t) \Leftrightarrow$
.5 $i \in (\mathsf{inds}\ t') \wedge$
.6 $i \in (\mathsf{inds}\ t) \Rightarrow$
.7 $t'(i) = t(i))$;

63.0 $\mathit{insert\text{-}char}(t : \mathit{text}, c : \mathit{appchars}\text{`}\mathit{appchar}, \mathit{line\text{-}no} : \mathbb{N}, p : \mathit{lines}\text{`}\mathit{pos})\ t' : \mathit{text}$
.1 pre $\mathit{addresses\text{-}a\text{-}line}(\mathit{line\text{-}no}, t) \wedge$
.2 $\mathit{lines}\text{`}\mathsf{pre}\text{-}\mathit{insert\text{-}char\text{-}after}(t(\mathit{line\text{-}no}), c, p)$
.3 post $\mathit{lines}\text{`}\mathsf{post}\text{-}\mathit{insert\text{-}char\text{-}after}$
.4 $($
.5 $t(\mathit{line\text{-}no}), c, p, t'(\mathit{line\text{-}no})) \wedge$
.6 $\mathit{unchanged\text{-}except}(t,t',\{\mathit{line\text{-}no}\})$;

64.0 $\mathit{remove\text{-}char}(t : \mathit{text}, \mathit{line\text{-}no} : \mathbb{N}, p : \mathit{lines}\text{`}\mathit{pos})\ t' : \mathit{text}$
.1 pre $\mathit{addresses\text{-}a\text{-}line}(\mathit{line\text{-}no}, t) \wedge$
.2 $\mathit{lines}\text{`}\mathsf{pre}\text{-}\mathit{remove\text{-}char}(t(\mathit{line\text{-}no}), p)$
.3 post $\mathit{lines}\text{`}\mathsf{post}\text{-}\mathit{remove\text{-}char}(t(\mathit{line\text{-}no}), p, t'(\mathit{line\text{-}no})) \wedge$
.4 $\mathit{unchanged\text{-}except}(t,t',\{\mathit{line\text{-}no}\})$;

65.0 $\mathit{split\text{-}line}(t : \mathit{text}, \mathit{line\text{-}no} : \mathbb{N}, p : \mathit{lines}\text{`}\mathit{pos})\ t' : \mathit{text}$
.1 pre $\mathit{addresses\text{-}a\text{-}line}(\mathit{line\text{-}no}, t) \wedge$
.2 $\mathit{lines}\text{`}\mathsf{pre}\text{-}\mathit{split\text{-}after}(t(\mathit{line\text{-}no}), p)$

.3 post $(\text{len } t') = (\text{len } t) + 1 \wedge$
.4 $\quad$ *lines*‘post-*split-after*$(t(\textit{line-no}), p,$
.5 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ mk-$(t'(\textit{line-no}), t'(\textit{line-no}+1))) \wedge$
.6 $\quad \forall\, i \in (\text{inds } t) \cdot$
.7 $\quad\quad i < \textit{line-no} \;\Rightarrow$
.8 $\quad\quad t'(i) = t(i) \wedge$
.9 $\quad\quad i > \textit{line-no}+1 \;\Rightarrow$
.10 $\quad\quad t'(i) = t(i-1)$;

66.0 *merge-lines*$(t : \textit{text}, \textit{line-no} : \mathbb{N})\ t' : \textit{text}$
.1 pre $\textit{line-no} \neq (\text{len } t) \wedge$
.2 $\quad$ *addresses-a-line*$(\textit{line-no}, t)$
.3 post $(\text{len } t') = (\text{len } t) - 1 \wedge$
.4 $\quad$ *lines*‘post-*merge-lines*$(t(\textit{line-no}), t(\textit{line-no}+1), t'(\textit{line-no})) \wedge$
.5 $\quad \forall\, i \in (\text{inds } t) \cdot$
.6 $\quad\quad i < \textit{line-no} \;\Rightarrow$
.7 $\quad\quad t'(i) = t(i) \wedge$
.8 $\quad\quad i > \textit{line-no} \;\Rightarrow$
.9 $\quad\quad t'(i) = t(i+1)$;

67.0 *ensure-length*$(t : \textit{text}, n : \mathbb{N})\ t' : \textit{text}$
.1 post $n \leq (\text{len } t') \wedge$
.2 $\quad$ *unchanged-except*$(t, t', (\text{inds } t') \setminus (\text{inds } t)) \wedge$
.3 $\quad \forall\, i \in (\text{inds } t') \setminus (\text{inds } t) \cdot$
.4 $\quad\quad t'(i) = \textit{lines}\text{‘}\textit{empty}() \wedge$
.5 $\quad\quad i \leq n$

end *texts*

A Funktionen zur Beschreibung und Manipulation von Sequenzen

module *seqtools*

exports all

definitions

functions

68.0 $prefix[@item]\,(s:@item^*, n:\mathbb{N})\ p:@item^* \triangleq$
.1 if $n = 0$
.2 then $[]$
.3 else $[\mathsf{hd}\ s] \frown prefix[@item]\,((\mathsf{tl}\ s), n-1)$
.4 pre $(\mathsf{len}\ s) \geq n$
.5 post $(\mathsf{len}\ p) = n \wedge$
.6 $\forall i \in \mathsf{inds}\ p \cdot p(i) = s(i)$;

69.0 $suffix[@item]\,(s:@item^*, n:\mathbb{N})\ sf:@item^* \triangleq$
.1 if $n = (\mathsf{len}\ s)$
.2 then s
.3 else $suffix[@item]\,((\mathsf{tl}\ s), n-1)$
.4 pre $(\mathsf{len}\ s) \geq n$
.5 post $(\mathsf{len}\ sf) = n \wedge$
.6 $prefix[@item]\,(s, (\mathsf{len}\ s) - n) \frown sf = s$;

70.0 $isPrefix[@item]:@item^* \times @item^* \to \mathbb{B}$
.1 $isPrefix\,(p,s) \triangleq$
.2 $prefix[@item]\,(s, (\mathsf{len}\ p)) = p$;

71.0 $isSuffix[@item]:@item^* \times @item^* \to \mathbb{B}$
.1 $isSuffix\,(sf,s) \triangleq$
.2 $suffix[@item]\,(s, (\mathsf{len}\ sf)) = sf$;

72.0 $mapped[@item, @res]\,(m:(@item \xrightarrow{m} @res), s:@item^*)\ r:@res^* \triangleq$
.1 if $s = []$
.2 then $[]$
.3 else $[m\,(\mathsf{hd}\ s)] \frown (\mathsf{tl}\ s)$
.4 post $(\mathsf{len}\ r) = (\mathsf{len}\ s) \wedge$
.5 $\forall i \in \mathsf{inds}\ s \cdot r(i) = m(s(i))$;

73.0 $addresses\text{-}an\text{-}item[@item]:\mathbb{N} \times @item^* \to \mathbb{B}$
.1 $addresses\text{-}an\text{-}item\,(n,s) \triangleq$
.2 $(n \in (\mathsf{inds}\ s))$

end *seqtools*

Literatur

[Balzert 1996] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Entwicklung*. Spektrum, Akademischer Verlag, 1996 (Lehrbücher der Informatik). – m. CD-ROM.. – ISBN 3-8274-0042-2

[Sufrin 1981] SUFRIN, Bernard: Specification of a Display Editor / Oxford University Computing Laboratory, Programming Research Group. 45, Banbury Road, Oxford. OX2 6PE, Juni 1981 (PRG-21). – Technical Monograph. Vergriffen, jedoch vermutlich 1982 in *Science of Computer Programming* veröffentlicht.

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

[TeachSWT@Tü 2002b] Wilhelm Schickart Institut (Veranst.): *Softwaretechnikvorlesung 2002*. 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 2002c] LEYPOLD, M E.: Von der Spezifikation zur Implementation. In: (TeachSWT@Tü, 2002b). – `handouts/example-specification-to-implementation.vdm`

[Woodcock und Davies 1996] WOODCOCK, Jim ; DAVIES, Jim: *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996 (Prentice-Hall international series in computer science). – ISBN 0-13-948472-8