Spezifikation des UNIX Datei- und Variablensystems. Eine SPECTRUM-Fallstudie*

Rudolf Hettler

30. Oktober 1992

*Diese Arbeit wurde unterstützt durch das BMFT-Projekt KORSO.

Zusammenfassung

In dieser Arbeit wird eine abstrakte Spezifikation des Dateisystems und Variablen-Environments des Betriebssystems UNIX gegeben. Dabei wird das Verhalten der Kommandos `pwd`, `cd`, `ls`, `mkdir`, `rmdir`, `touch`, `rm`, `setenv` und `$` spezifiziert. Besonderes Augenmerk wird dabei auf gute Verständlichkeit und leichte Erweiterbarkeit der Spezifikation um weitere UNIX-Konzepte gerichtet.

Inhaltsverzeichnis

Kapitel 1

Einführung

Dieses Papier stellt eine abstrakte Spezifikation zweier Komponenten des Betriebssystems UNIX (vgl. [Bou83]) vor, das Dateisystem (mit den Kommandos `pwd`, `cd`, `ls`, `mkdir`, `rmdir`, `touch` und `rm`) und die Variablenverwaltung (mit den Kommandos `setenv` und `$`). Als Spezifikationssprache findet dabei die an der TU München entwickelte Sprache SPECTRUM [BFG+92] Verwendung.

Spezifikationen des UNIX-Dateisystems wurden bereits von verschiedenen Autoren in unterschiedlichen Formalismen gegeben (siehe zum Beispiel [BGM89], [MS87]). Die vorliegende Arbeit wurde angeregt durch eine in der Sprache OBSCURE gegebene Spezifikation von Prof. J. Loeckx, R. Heckler und S. Uhrig (vgl. [HLU91]).

Die in [HLU91] angegebene Spezifikation ist ausführbar und kann mit dem OBSCURE-System [FHL+91] getestet werden. Sie versucht die dem Benutzer zur Verfügung stehende Schnittstelle möglichst exakt wiederzugeben. Insbesondere sollen auch die in UNIX zu beobachtenden unerwarteten Verhaltensweisen nachgebildet werden[1].

Im Gegensatz zu [HLU91] versucht die vorliegende Arbeit, eine möglichst abstrakte Anforderungsspezifikation des UNIX Dateisystems und Variablen-Environments zu geben. Die Spezifikation soll knapp, gut lesbar und verständlich sein. Als abstrakte Anforderungsspezifikation soll sie die wesentlichen Eigenschaften der spezifizierten Komponenten herausarbeiten und auf Implementierungsdetails verzichten. Der verwendete Spezifikationsstil ist nicht konstruktiv, da zur Beschreibung der Eigenschaften des spezifizierten Systems teilweise nicht ausführbare Konstrukte wie zum Beispiel Existenzquantoren verwendet werden.

Der Rest dieser Arbeit gliedert sich wie folgt. Für die restliche Spezifikation wichtige primitive Spezifikationen sind in Kapitel 2 zusammengefaßt. Die abstrakte Spezifikation des Dateisystems und des Variablen-Environments sowie der Kommandos `pwd`, `cd`, `ls`, `mkdir`, `rmdir`, `touch`, `rm`, `setenv` und `$` erfolgt

[1] So führt zum Beispiel ein `'cd ..'` im Root-Directory `/` nicht zu einer Fehlermeldung, sondern wird ignoriert. Das Kommando `'cd //'` wird nicht als inkorrekt zurückgewiesen, sondern wie `'cd /'` behandelt.

in Kapitel 3, während in Kapitel 4 Eigenschaften einer Shell spezifiziert werden. Kapitel 5 enthält einige abschließende Bemerkungen über die angegebene Spezifikation und gibt Ausblicke auf mögliche Erweiterungen.

Kapitel 2

Primitive Spezifikationen

Dieses Kapitel gibt eine Übersicht über die in der restlichen Spezifikation verwendeten primitiven Spezifikationen. Da es sich dabei um Spezifikationen von allgemein bekannten Grunddatentypen handelt, werden sie hier ohne weitere Erläuterungen lediglich aufgelistet.

2.1 Zeichen

CHAR[1] = {
    sort Char = A | ... | Z | a | ... | z | 0 | ... | 9 | / | ~ | DOT | ...;
}

2.2 Natürliche Zahlen

NAT = {

    sort Nat;

    zero : Nat;
    succ : Nat ⟶ Nat **strict total**;

    .≤. : Nat × Nat ⟶ Bool **strict total prio** 6;
    .<. : Nat × Nat ⟶ Bool **strict total prio** 6;
    .+. : Nat × Nat ⟶ Nat **strict total prio** 6:**left**;
    .∗. : Nat × Nat ⟶ Nat **strict total prio** 7:**left**;
    pred : Nat ⟶ Nat **strict**;

[1] Die ... in dieser Spezifikation dienen lediglich der Schreibabkürzung. Sie sind kein Teil der Sprache SPECTRUM.

$.-. : \mathsf{Nat} \times \mathsf{Nat} \longrightarrow \mathsf{Nat}$ **strict prio 6:left**;
.div., .mod. : $\mathsf{Nat} \times \mathsf{Nat} \longrightarrow \mathsf{Nat}$ **strict prio 7:left**;

Nat **generated by** zero, succ;

axioms $\forall$ n, m : Nat **in**

$\mathsf{zero} \neq \mathsf{succ}\ \mathsf{n}$;
$\mathsf{succ}\ \mathsf{n} = \mathsf{succ}\ \mathsf{m} \Rightarrow \mathsf{n} = \mathsf{m}$;

$\delta\ (\mathsf{pred}\ \mathsf{n}) \Leftrightarrow \mathsf{n} \neq \mathsf{zero}$;
$\mathsf{pred}(\mathsf{succ}\ \mathsf{n}) = \mathsf{n}$;

$\mathsf{zero} \leq \mathsf{n}$;
$\neg(\mathsf{succ}\ \mathsf{n} \leq \mathsf{zero})$;
$\mathsf{succ}\ \mathsf{n} \leq \mathsf{succ}\ \mathsf{m} \Leftrightarrow \mathsf{n} \leq \mathsf{m}$;
$\mathsf{n} < \mathsf{m} = (\mathsf{n} \leq \mathsf{m} \wedge \mathsf{n} \neq \mathsf{m})$;

$\mathsf{n} + \mathsf{zero} = \mathsf{n}$;
$\mathsf{n} + \mathsf{succ}\ \mathsf{m} = \mathsf{succ}\ (\mathsf{n} + \mathsf{m})$;

$\delta\ (\mathsf{n} - \mathsf{m}) \Leftrightarrow \mathsf{m} \leq \mathsf{n}$;
$(\mathsf{n} + \mathsf{m}) - \mathsf{m} = \mathsf{n}$;

$\mathsf{n} * \mathsf{zero} = \mathsf{zero}$;
$\mathsf{n} * \mathsf{succ}\ \mathsf{m} = \mathsf{n} + \mathsf{n} * \mathsf{m}$;

$\delta\ (\mathsf{n}\ \mathsf{div}\ \mathsf{m}) \Leftrightarrow \mathsf{m} \neq \mathsf{zero}$;
$\delta\ (\mathsf{n}\ \mathsf{mod}\ \mathsf{m}) \Leftrightarrow \mathsf{m} \neq \mathsf{zero}$;
$\mathsf{m} \neq \mathsf{zero} \Rightarrow \mathsf{n}\ \mathsf{mod}\ \mathsf{m} < \mathsf{m}$;
$\mathsf{m} \neq \mathsf{zero} \Rightarrow \mathsf{n} = (\mathsf{n}\ \mathsf{div}\ \mathsf{m}) * \mathsf{m} + \mathsf{n}\ \mathsf{mod}\ \mathsf{m}$;

endaxioms;
}

2.3 Sequenzen

SEQ = {

enriches NAT;

sort Seq α;

-- Generatoren
ε : Seq α;
.^. : $\alpha \times$ Seq $\alpha \longrightarrow$ Seq α **strict total prio 9:right**;

Seq α **freely generated by** ε, .^.:$\alpha \times$ Seq $\alpha \longrightarrow$ Seq α;

-- Funktionen auf Sequenzen
first : Seq $\alpha \longrightarrow \alpha$ **strict**;
rest : Seq $\alpha \longrightarrow$ Seq α **strict**;
mkseq : $\alpha \longrightarrow$ Seq α **strict total**;
.^. : Seq $\alpha \times \alpha \longrightarrow$ Seq α **strict total prio 9:right**;
last : Seq $\alpha \longrightarrow \alpha$ **strict**;
lead : Seq $\alpha \longrightarrow$ Seq α **strict**;
.^. : Seq $\alpha \times$ Seq $\alpha \longrightarrow$ Seq α **strict total prio 9:right**;
% : Seq $\alpha \longrightarrow$ Nat **strict total**;
.$\in$., .$\notin$. : $\alpha \times$ Seq $\alpha \longrightarrow$ Bool **strict total prio 6**;
.is_prefix_of., .is_postfix_of., .contains. :
Seq $\alpha \times$ Seq $\alpha \longrightarrow$ Bool **strict total prio 6**;

axioms $\forall$ r, s, t : Seq α; e : α **in**
δ(first z) = (z $\neq \varepsilon$);
δ(rest z) = (z $\neq \varepsilon$);
first(e^s) = e;
rest(e^s) = s;

r^(s^t) = (r^s)^t;
s^ε = s;
ε^s = s;

e^s = mkseq(e)^s;
s^e = s^mkseq(e);

δ(last z) = (z $\neq \varepsilon$);
δ(lead z) = (z $\neq \varepsilon$);
last(s^e) = e;
lead(s^e) = s;

% ε = zero;
%(e^s) = succ(% s);
e $\in$ s = $\exists$ s1, s2 : Seq α. s = s1^e^s2;

e $\notin$ s = $\neg$(e $\in$ s);

r is_prefix_of s = $\exists$ s' : Seq α. s = r^s';

r is_postfix_of s = $\exists$ s' : Seq α. s = s'^r;

s contains r = $\exists$ s', s'' : Seq α. s = s'^r^s'';

endaxioms;

}

2.4 Mengen

SET = {

enriches NAT;

sort Set α;

$\emptyset$: Set α;
add, del : Set $\alpha \times \alpha \longrightarrow$ Set α **strict total**;
.$\in$., .$\notin$. : $\alpha \times$ Set $\alpha \longrightarrow$ Bool **strict total prio** 6;
choose : Set $\alpha \longrightarrow \alpha$ **strict**;
card : Set $\alpha \longrightarrow$ Nat **strict total**;
.$\cup$., .$\cap$., .\. : Set $\alpha \times$ Set $\alpha \longrightarrow$ Set α **strict total prio** 7:**left**;
.$\subseteq$. : Set $\alpha \times$ Set $\alpha \longrightarrow$ Bool **strict total prio** 6;

Set α **generated by** $\emptyset$, add;

axioms $\forall$ a, b : α; s, s' : Set α **in**

add(add(s,a),a) = add(s,a);
add(add(s,a),b) = add(add(s,b),a);

$\neg$(a $\in$ $\emptyset$);
a $\in$ add(s,b) $\Leftrightarrow$ (a = b) $\vee$ a $\in$ s;
a $\in$ del(s,b) $\Leftrightarrow$ (a $\neq$ b) $\wedge$ a $\in$ s;

a $\notin$ s = $\neg$(a $\in$ s);

δ (choose s) $\Leftrightarrow$ s $\neq$ $\emptyset$;
s $\neq$ $\emptyset$ $\Rightarrow$ choose s $\in$ s;

card $\emptyset$ = zero;
a $\in$ s $\Rightarrow$ card(add(s, a)) = card s;
a $\notin$ s $\Rightarrow$ card(add(s, a)) = succ(card s);

s $\cup$ $\emptyset$ = s;
s $\cup$ add(s', a) = add(s, a) $\cup$ s';

s $\cap$ $\emptyset$ = $\emptyset$;
a $\in$ s $\Rightarrow$ s $\cap$ add(s', a) = add(s $\cap$ s', a);
a $\notin$ s $\Rightarrow$ s $\cap$ add(s', a) = s $\cap$ s';

s \ $\emptyset$ = s;
s \ add(s', a) = del(s, a) \ s';

$\emptyset$ $\subseteq$ s;
add(s, a) $\subseteq$ s' = ((a $\in$ s') $\wedge$ (s $\subseteq$ s'));

endaxioms;
}

2.5 Endliche Abbildungen

MAP = {

sort Map α β;

$\emptyset$: Map α β;
update : Map α β $\times$ α $\times$ β $\longrightarrow$ Map α β **strict total**;
.?. : Map α β $\times$ α $\longrightarrow$ β **strict prio** 6;
isdef : Map α β $\times$ α $\longrightarrow$ Bool **strict total**;

Map α β **generated by** $\emptyset$, update;

axioms $\forall$ a, a' : α; b : β; m : Map α β **in**

¬(isdef($\emptyset$, a));
isdef(update(m, a, b), a') = (a = a' $\vee$ isdef(m, a'));

δ(m ? a) $\Leftrightarrow$ isdef(m, a);
(update(m, a, b) ? a) = b;
a $\neq$ a' $\Rightarrow$ (update(m, a, b) ? a') = (m ? a');

endaxioms;
}

2.6 Strings

STRING = {

enriches SEQ + CHAR;

sortdef String = Seq Char;

}

Da Strings Sequenzen über Zeichen sind, werden sie mit den in SEQ definierten Konstruktoren ε und .^. aufgebaut. Um diese aufwendige Schreibweise zu vermeiden, wird für den Rest dieser Arbeit folgende Schreibabkürzung vereinbart:

> *Ein String* a^b^c^ε *wird abgekürzt als* 'abc' *geschrieben. Innerhalb einer solchen Abkürzung kann statt* DOT *auch ein normaler Punkt geschrieben werden. '..' ist also eine Abkürzung für* DOT^DOT^ε.

Diese Konvention wird auch innerhalb von Spezifikationen eingehalten, da dadurch eine wesentlich bessere Lesbarkeit gegeben ist. Formal können solche Abkürzungen wie vordefinierte Konstanten der Sorte String behandelt werden.

2.7 Projektionsfunktionen

In den folgenden Spezifikationen werden Projektionsfunktionen auf zweistelligen Tupeln benötigt. Diese werden in PROJ polymorph spezifiziert.

PROJ = {

fst : $\alpha \times \beta \rightarrow \alpha$ **total**;
snd : $\alpha \times \beta \rightarrow \beta$ **total**;
axioms $\forall^{\perp}$ a : α; b : β **in**

fst(a,b) = a;
snd(a,b) = b;

endaxioms;
}

Kapitel 3

Das Variablen- und Dateisystem

Das System besteht aus den Komponenten Variablen-Environment und Dateiverwaltung. Diese beiden Komponenten werden in den Abschnitten 3.1 und 3.2 spezifiziert. Abschnitt 3.3 stellt die Spezifikation SYSTEM_CORE vor, in der die beiden Komponenten zu einem Systemkern zusammengefaßt werden. Diese Spezifikation enthält auf internen Sorten (wie Path) arbeitende Versionen der in Kapitel 1 genannten UNIX-Kommandos. Aufbauend auf diesen Systemkern werden in Abschnitt 3.4 die eigentlichen Systemkommandos spezifiziert.

3.1 Das Variablen-Environment

In UNIX-Systemen hat der Benutzer die Möglichkeit, Information in Form von Strings in sogenannten *Environment Variablen* zu speichern und wieder auszulesen. Obwohl [HLU91] das UNIX Variablensystem mit den Kommandos `$` und `setenv` nicht spezifizieren, soll die hier gegebene Spezifikation diese Komponente enthalten, da sich damit die Eigenschaft des UNIX-Systems, die Pfade des *current working directory* und des *HOME-Directory* des jeweiligen Benutzers in Variablen zu speichern, angemessen nachbilden läßt.

Das Variablen-Environment ist eine endliche (partielle) Abbildung von Variablen nach Strings. Variablen selbst werden durch Strings bezeichnet. Wie die folgende Spezifikation VAR zeigt, gibt es keine Variable, die durch den leeren String ε bezeichnet wird. Zusätzlich wird eine Variable CWD definiert, die keine Bezeichnung als String besitzt und deshalb vom Benutzer nicht direkt (aus der Shell) angesprochen werden kann. Diese Variable wird vom Dateisystem benutzt, um das *current working directory* zu speichern (siehe Kapitel 3.3).

VAR = {

enriches STRING;

sort Var;

```
    CWD : Var;
    String2Var : String ⟶ Var strict;
    Var2String : Var ⟶ String strict;

    axioms ∀ s : String; v : Var in

        δ(String2Var s) ⇔ s ≠ ε;
        δ(Var2String v) ⇔ v ≠ CWD;
        s ≠ ε ⇒ Var2String(String2Var s) = s;
        v ≠ CWD ⇒ String2Var(Var2String v) = v;

    endaxioms;
}

VARENV = {

enriches MAP + VAR + STRING;

sortdef Varenv = Map Var String;

}
```

Auf Variablen-Environments sind also die üblichen MAP-Funktionen definiert. Dabei wird `update` später zur Implementierung des Kommandos `setenv` dienen (siehe Abschnitt 3.3 und 3.4) und .?. zur Implementierung von `$`.

3.2 Die Dateiverwaltung

Dateien werden in UNIX nicht flach verwaltet, sondern zu Directories zusammengefaßt, welche im Dateisystem baumartig verwaltet werden[1]. Ausgehend von einem Wurzeldirectory kann jedes Directory beliebig viele Söhne (Subdirectories) besitzen. Sowohl Dateien als auch Subdirectories werden durch Strings bezeichnet, die folgende Eigenschaften erfüllen:

1. Es gibt keine leeren Bezeichner.
2. Kein Bezeichner darf das Zeichen / enthalten.
3. Die Strings '.' und '..' sind nicht als Bezeichner zugelassen.

[1]Diese Fallstudie berücksichtigt keine symbolischen Links, die in UNIX-Systemen die strenge Baumstruktur durchbrechen. Sie kann aber um dieses Konzept erweitert werden. Näheres dazu findet sich in Kapitel 5.

Um diesen Eigenschaften zu genügen, wir für diese Bezeichner eine Sorte Nodeid eingeführt, deren Elemente isomorph zu den erlaubten Strings sind.

NODEID = {

enriches STRING;

sort Nodeid;

String2Nodeid : String $\rightarrow$ Nodeid **strict**;
Nodeid2String : Nodeid $\rightarrow$ String **strict total**;

axioms $\forall$ s : String; n : Nodeid **in**

δ(String2Nodeid s) $\Leftrightarrow$ s $\neq \varepsilon \wedge$ s $\neq$ '.' $\wedge$ s $\neq$ '..' $\wedge$ (/ $\notin$ s);
δ(String2Nodeid s) $\Rightarrow$ Nodeid2String(String2Nodeid s) = s;
String2Nodeid(Nodeid2String n) = n;

endaxioms;
}

Dateien Dateien sind die kleinsten Informationseinheiten, die im Dateisystem gespeichert werden können. Diese Arbeit macht keine Aussage darüber, welche Eigenschaften Dateien besitzen. Sie werden hier als zu verwaltende primitive Elemente gesehen. Es wird lediglich gefordert, daß es eine ausgezeichnete leere Datei gibt, die durch das Systemkommando `_touch` erzeugt wird (siehe Abschnitt 3.3).

FILE = {

sort File;

empty : File;

}

Directories Directories sind die Verwaltungseinheiten des Dateisystems. Ein Directory enthält eine Menge von Dateien, die durch im jeweiligen Directory eindeutige Bezeichner (Sorte Nodeid) identifiziert werden. Man kann sich also Directories als endliche Abbildungen von Bezeichnern in Dateien vorstellen.

DIRECTORY = {

```
    enriches NODEID + MAP + FILE;

    sortdef Directory = Map Nodeid File;

endaxioms;
}
```

Pfade Da die Directories im Dateisystem baumartig verwaltet werden, kann jedes Directory durch die Angabe der Bezeichner der Subdirectories, die von der Wurzel bis zu diesem Directory durchlaufen werden müssen, bezeichnet werden. Eine solche Sequenz heißt *(absoluter) Pfad*. Das Wurzeldirectory wird durch die leere Sequenz bezeichnet, die in der folgenden Spezifikation den Namen ROOT erhält.

```
PATH = rename {
    enriches SEQ + NODEID;
    sortdef Path = Seq Nodeid;
}
by [ ε to ROOT ]
```

Pfade werden nicht nur dazu verwendet, Directories zu bezeichnen. Ist das letzte Element eines Pfades der Bezeichner einer Datei, so bezeichnet er eine Datei innerhalb des gesamten Dateisystems.

Für Pfade kann nun eine Stringdarstellung definiert werden. Die Repräsentation eines absoluten Pfades als String beginnt mit dem Zeichen /. Daran schließen sich, durch / getrennt, die Stringdarstellungen der im Pfad enthaltenen Knotenbezeichner an.

```
PATH2STRING = {

    enriches PATH + STRING;

    Path2String : Path ⟶ String strict total;
    legal_pathid : String ⟶ Bool strict total;

    axioms ∀ p : Path; n : Nodeid; s : String in

        Path2String ROOT = '/';
        Path2String (p^n) = Path2String p ^ '/' ^ Nodeid2String n;

        legal_pathid s = ∃ p' : Path. s = Path2String p';

    endaxioms;
}
```

Im Rest dieser Arbeit wird nur noch in den Spezifikationen zwischen Pfaden und ihrer Stringdarstellung unterschieden. Im Text wird aufgrund der besseren Lesbarkeit immer die Stringdarstellung verwendet.

Dateisystem Wie bereits weiter oben erwähnt, werden Directories im UNIX-Dateisystem hierarchisch verwaltet und durch Pfade bezeichnet. Anstatt das Dateisystem als Baum von Directories zu modellieren, wird es in dieser Arbeit als Zuordnung (Abbildung) von Pfaden zu Directories angesehen. Diese Entscheidung wurde getroffen, um eine Erweiterung der Spezifikation um sogenannte *harte Links* zu erleichtern, da die Einführung von Links die strenge Baumstruktur des Dateisystems durchbricht (siehe dazu Kapitel 5). Die Abbildung von Pfaden in Directories ist partiell, da in einem Dateisystem nicht allen Pfaden Directories zugeordnet sind. Damit diese Abbildung das vom UNIX-System her bekannte hierarchische Dateisystem beschreibt, muß sie folgende Eigenschaften erfüllen:

1. Jedes Directory außer dem Wurzeldirectory ROOT besitzt ein Vaterdirectory, d.h. ist die Abbildung auf dem Pfad p definiert, so ist sie auch auf lead p definiert.

2. Bei jedem Directory müssen die Menge der Bezeichner der darin enthaltenen Dateien und die Menge der Bezeichner seiner Subdirectories disjunkt sein.

Die folgende Spezifikation beschreibt diese Zuordnung, indem sie partielle Konstruktoren verwendet, die immer dann undefiniert sind, wenn eine der obigen Bedingungen verletzt würde. Dabei erlaubt der Konstruktor addir das Anlegen eines noch nicht im System vorhandenen leeren Directory, während updatedir das Verändern eines bereits existierenden Directory ermöglicht.

FILESYS = {

enriches SET + PATH + DIRECTORY;

sort Filesys;

emptyfs : Filesys;
addir : Filesys × Path ⟶ Filesys **strict**;
updatedir : Filesys × Path × Directory ⟶ Filesys **strict**;

axioms ∀ fs, fs' : Filesys; p, p' : Path; d : Directory **in**

emptyfs ≠ addir(fs, p);
emptyfs ≠ updatedir(fs, p, d);
addir(fs, p) ≠ updatedir(fs', p', d);

endaxioms;

subdirs : Filesys × Path ⟶ Set Nodeid **strict**;
has_subdirs : Filesys × Path ⟶ Bool **strict**;
exists_dir : Filesys × Path ⟶ Bool **strict total**;
exists_file : Filesys × Path ⟶ Bool **strict total**;
.?. : Filesys × Path ⟶ Directory **strict prio** 6;
deldir : Filesys × Path ⟶ Filesys **strict**;

axioms ∀ fs : Filesys; p, p' : Path; d : Directory; dn : Nodeid **in**

-- *Definiertheit der Konstruktoren*
δ(addir(fs, p)) ⇔ ¬(exists_dir(fs, p)) ∧ exists_dir(fs, lead p) ∧
last p ∉ (fs ? lead p);

δ(updatedir(fs, p, d)) ⇔ exists_dir(fs, p) ∧
∀ n : Nodeid. isdef(d,n) ⇒ n ∉ subdirs(fs,p);

-- *exists_dir*
exists_dir(emptyfs, p) = (p = ROOT);
δ(addir(fs, p)) ⇒
exists_dir(addir(fs, p), p') = (p = p' ∨ exists_dir(fs, p'));
δ(updatedir(fs, p, d)) ⇒
exists_dir(updatedir(fs, p, d), p') = exists_dir(fs, p');

-- *exists_file*
¬(exists_file(fs, ROOT));
exists_file(fs, p^dn) = (exists_dir(fs, p) ∧ dn isin (fs?p));

-- *subdirs*
δ(subdirs(fs, p)) ⇔ exists_dir(fs, p);
exists_dir(fs, p) ⇒ (dn ∈ subdirs(fs, p) ⇔ exists_dir(fs, p^dn));

-- *has_subdirs*
δ(has_subdirs(fs, p)) ⇔ exists_dir(fs, p);
exists_dir(fs, p) ⇒ has_subdirs(fs, p) = (subdirs(fs, p) ≠ ∅);

-- *deldir*
δ(deldir(fs, p)) ⇔
p ≠ ROOT ∧ exists_dir(fs, p) ∧ subdirs(fs, p) = ∅ ∧ fs?p = ∅;
δ(deldir(fs, p) ⇒
exists_dir(deldir(fs, p), p') = (p ≠ p' ∧ exists_dir(fs, p');

exists_dir(deldir(fs, p), p') ⇒ deldir(fs, p)?p' = fs?p';

-- .?.
δ(fs ? p) ⇔ exists_dir(fs, p);
emptyfs ? ROOT = ∅;
δ(addir(fs, p)) ⇒ addir(fs, p) ? p = ∅;
δ(addir(fs, p)) ∧ p ≠ p' ⇒ addir(fs, p) ? p' = fs ? p';
δ(updatedir(fs, p, d)) ⇒ updatedir(fs, p, d) ? p = d;
δ(updatedir(fs, p, d)) ∧ p ≠ p' ⇒ updatedir(fs, p, d) ? p' = fs ? p';

endaxioms;
}

3.3 Der Systemkern

In diesem Abschnitt werden die in 3.1 und 3.2 spezifizierten Komponenten Variablen-Environment und Dateiverwaltung zu einem Systemkern zusammengefaßt. In diesem Systemkern werden die Kommandos `_pwd`, `_cd`, `_ls`, `_mkdir`, `_rmdir`, `_touch`, `_rm`, `_setenv` und `_$` definiert. Der Underscore _ als erstes Zeichen des Kommandonamens deutet an, daß diese Kommandos auf abstrakten Sorten wie Varenv, Filesys, Path, ... arbeiten, während ihre Äquivalente, wie sie der UNIX-Benutzer kennt, auf Strings operieren. Es werden also über die Kommandos des Variablensystems (`_$` und `_setenv`) und die auch in [HLU91] spezifizierten Kommandos `_pwd`, `_cd`, `_ls`, `_mkdir` und `_rmdir` hinaus die Kommandos `_touch` und `_rm` zum Erzeugen und Löschen von Dateien spezifiziert, da sich die Spezifikation des Dateisystems wesentlich auf das Konzept der Datei stützt. Im folgenden werden die Kernkommandos kurz informell vorgestellt:

`_setenv` erlaubt es, eine Environment-Variable mit einem String zu belegen, unabhängig davon, ob diese Variable bereits definiert war oder nicht.

`_$` liest den Inhalt einer Environment-Variablen aus.

`_mkdir` erzeugt, falls möglich, unter dem angegebenen Pfad ein leeres Directory.

`_rmdir` löscht ein leeres Directory, falls es keine Subdirectories hat.

`_touch` erzeugt die angegebene Datei[2].

`_rm` löscht die angegebene Datei, falls vorhanden.

[2] `touch` wird in UNIX dazu benutzt, das Modifikationsdatum von Dateien zu verändern. Ist eine Datei nicht vorhanden, so wird sie durch `touch` angelegt. Da dieses Fallbeispiel zusätzliche Informationen zu Dateien wie das Modifikationsdatum nicht berücksichtigt, wird hier nur der zweite Effekt zum Erzeugen von Dateien benutzt.

`_ls` listet die im angegebenen Directory enthaltenen Dateien und die Subdirectories dieses Directory auf.

`_cd` wechselt das aktuelle Directory (current working directory).

`_pwd` gibt das current working directory aus.

Alle diese Kommandos liefern einen Rückgabewert, der entweder erfolgreiche Bearbeitung oder das Auftreten eines Fehlers signalisiert. Der Systemzustand, auf dem diese Funktionen operieren, ist das Tupel bestehend aus der Sorte Filesys und der Sorte Varenv. Es sind jedoch nicht alle Paare aus Varenv und Filesys korrekte Systemzustände, sondern nur diejenigen, bei denen die Variable CWD die Pfadbezeichnung eines im Dateisystem existierenden Pfades enthält. Dies ist in der Spezifikation SYSTEM_CORE durch Partialität des Sortenkonstruktors state modelliert, der $\perp$ liefert, wenn die obige Bedingung verletzt ist.

ERROR = {

sort Error = OK
| Variable_not_defined
| No_such_Directory
| Directory_exists
| Cannot_make_Directory
| Cannot_remove_Directory
| No_such_File
| File_exists
| Cannot_touch_File
| Illegal_Variable_Identifier
| Illegal_Path_Identifier;

}

SYSTEM_CORE = {

enriches SET + ERROR + PATH + PATH2STRING + VARENV + FILESYS;

sort State = state(Ve : Varenv, Fs : Filesys) **strict**;

axioms $\forall$ ve : Varenv; fs : Filesys in

δ(state(ve, fs)) $\Leftrightarrow$ $\exists$ p : Path. Path2String p = ve?CWD $\wedge$ exists_dir(fs, p);

endaxioms;

isdir, isfile : State × Path ⟶ Bool **strict total**;

_$: State × Var ⟶ String × Error **strict total**;
_setenv : State × Var × String ⟶ State × Error **strict total**;
_cd : State × Path ⟶ State × Error **strict total**;
_pwd : State ⟶ String × Error **strict total**;
_ls : State × Path ⟶ Set String × Error **strict total**;
_mkdir, _rmdir : State × Path ⟶ State × Error **strict total**;
_touch, _rm : State × Path ⟶ State × Error **strict total**;

axioms ∀ st : State; v : Var; p : Path; s : String; n : Nodeid **in**

-- isdir
isdir(st, p) = exists_dir(Fs st, p);

-- isfile
isfile(st, p) = exists_file(Fs st, p);

-- _$
_$(st, v) = **if** isdef(Ve st, v)
then ((Ve st)?v, OK)
else (ε, Variable_not_defined)
endif;

-- *_setenv*
_setenv(st, v, s) = (state(update(Ve st, v, s), Fs st), OK);

-- *_cd*
_cd(st, p) = **if** isdir(st, p)
then (state(update(Ve st, CWD, Path2String p), Fs st), OK)
else (st, No_such_Directory)
endif;

-- *_pwd*
_pwd st = ((Ve st)?CWD, OK);

-- *_ls*
¬(isdir(st, p)) ⇒ _ls(st, p) = (∅, No_such_Directory);
isdir(st, p) ⇒ ∀ res : Set String. _ls(st, p) = (res, OK) ⇔
∀ s : String. s ∈ res ⇔
∃ n : Nodeid. s = Nodeid2String n ∧
(isdef(Fs st?p, n) ∨ n ∈ subdirs(Fs st, p));

```
-- _mkdir
_mkdir(st, p) = if isdir(st, p)
                then (st, Directory_exists)
                else if isfile(st, p)
                     then (st, File_exists)
                     else if ¬(isdir(st, lead p))
                          then (st, Cannot_make_Directory)
                          else (state(Ve st, addir(Fs st, p)), OK)
                          endif
                     endif
                endif;

-- _rmdir
_rmdir(st, p) = if ¬(isdir(st, p))
                then (st, No_such_Directory)
                else if has_subdirs(Fs st, p) ∨ (Fs st)?p ≠ ∅
                     then (st, Cannot_remove_Directory)
                     else (state(Ve st, deldir(Fs st, p)), OK)
                     endif
                endif;

-- _touch
_touch(st, ROOT) = (st, Directory_exists);
_touch(st, p^n) = if isfile(st, p^n)
                  then (st, File_exists)
                  else if isdir(st, p^n)
                       then (st, Directory_exists)
                       else if ¬(isdir(st, p))
                            then (st, Cannot_touch_File)
                            else (state(Ve st,
                                        update(Fs st,
                                               p,
                                               add((Fs st)?p,
                                                   mkFile(n, empty)))),
                                  OK)
                            endif
                       endif
                  endif;

-- _rm
_rm(st, ROOT) = (st, No_such_File);
_rm(st, p^n) = if ¬(isfile(st, p^n))
```

```
                then (st, No_such_File)
                else (state(Ve st,
                        update(Fs st,
                                p,
                                del_file((Fs st)?p, n))),
                      OK)
                endif;

    endaxioms;
}
```

3.4 Die Systemkommandos

Aufbauend auf den Systemkern können nun die für den Benutzer sichtbaren Kommandos `pwd`, `cd`, `ls`, `mkdir`, `rmdir`, `touch`, `rm`, `setenv` und `$` definiert werden. Diese Kommandos erhalten als Eingabe Stringrepräsentationen von internen Objekten (Variablen und Pfade), nehmen die Übersetzung dieser Strings in die internen Objekte vor und rufen dann die zugehörigen Kernkommandos auf. Als Ergebnis reichen sie das Resultat des aufgerufenen Kernkommandos durch.

Neben dieser einfachen Umsetzung wird in dieser Schicht das Konzept der relativen Pfade als Abkürzungsmöglichkeit für absolute Pfade eingeführt.

Relative Pfade Ein absoluter Pfad bezeichnet ein Directory (bzw. eine Datei) durch Angabe aller Subdirectories, die von der Wurzel bis zu diesem Directory (dieser Datei) durchlaufen werden müssen. Ein relativer Pfad dagegen startet nicht beim Wurzeldirectory, sondern beim 'current working directory'. Der absolute Pfad dieses Directory ist in der Variablen CWD gespeichert. Bezeichner für absolute Pfade und relative Pfade können durch ihr erstes Zeichen unterschieden werden. Absolute Pfade beginnen mit dem (in Elementen der Sorte Nodeid nicht zugelassenen) Zeichen /, relative Pfade beginnen sofort mit einem Subdirectory-Bezeichner. Der absolute Pfad eines durch einen relativen Pfad bezeichneten Directory entsteht durch Konkatenation des Inhalts von CWD mit der relativen Pfadbezeichnung (die beiden Teile werden durch ein / getrennt). Ist zum Beispiel '/usr' das current working directory, dann ist der relative Pfad 'include' gleichbedeutend mit dem absoluten Pfad '/usr/include'. Die Umsetzung von relativen Pfaden in absolute Pfadbezeichnungen erfolgt durch die Funktion make_abs.

'.' und '..' Durch relative Pfade können lediglich Directories und Dateien bezeichnet werden, die in dem Teilbaum liegen, der das current working directory als Wurzel hat. Um diesen Nachteil zu umgehen, gibt es in UNIX die speziellen Directory-Bezeichner '.' und '..'. Diese in der Spezifikation des Systemkerns nicht zugelassenen Bezeichner werden nun zusammen mit den relativen Pfaden

eingeführt. Sie sind in jedem Directory als Subdirectory-Bezeichner gültig, bezeichnen jedoch keine echten Söhne dieses Directory. Vielmehr ist '.' eine Bezeichnung für das Directory selbst, während '..' das Vaterdirectory bezeichnet. Beide Bezeichner können in der Stringrepräsentation sowohl von relativen Pfaden als auch von absoluten Pfaden vorkommen. So bezeichnen zum Beispiel die Pfade '/usr/./include', '/usr/include/../include' und '/usr/include' das gleiche Directory.

Durch die Einführung relativer Pfade und der Bezeichner '.' und '..' ist die Bezeichnung von Directories und Dateien durch Pfade nicht mehr eindeutig. Zu jedem (möglicherweise relativen) Pfad, der '.' und '..' enthält, kann aber (eindeutig) eine äquivalente absolute Pfadbezeichnung gefunden werden, die '.' und '..' nicht mehr enthält. Dies geschieht in der folgenden Spezifikation in den Funktionen make_abs,DOTelim und DOTDOTelim. Diese Pfadbezeichnung kann mittels der Funktion path in die interne Sorte Path übersetzt werden. Mit Hilfe von make_abs, normalize und path kann also eine Funktion String2Path definiert werden, die die Stringrepräsentation eines Pfades in die Sorte Path übersetzt und dabei sowohl relative Pfadbezeichnungen als auch die Verwendung von '.' und '..' erlaubt.

Man beachte in der folgenden Spezifikation, daß '..' auch im Wurzeldirectory verwendet werden darf, obwohl dieses Directory keinen Vater besitzt. Die Pfadbezeichnung '/..' erzeugt jedoch keinen Fehler, sondern ist gleichbedeutend mit '/'. Dies entspricht dem bei UNIX-Systemen beobachtbaren Verhalten und ist in der Funktion DOTDOTelim berücksichtigt.

STRING2PATH = {

enriches PROJ + PATH2STRING + SYSTEM_CORE;

path : String $\longrightarrow$ Path **strict**;
DOTelim : String $\longrightarrow$ String **strict total**;
DOTDOTelim : String $\longrightarrow$ String **strict total**;
make_abs : String $\times$ State $\longrightarrow$ String **strict total**;
String2Path : String $\times$ State $\longrightarrow$ Path **strict**;

axioms $\forall$ p : Path; s, s1, s2, s3 : String **in**

δ(path s) = legal_pathid s;
path(Path2String p) = p;

$\neg$(s contains '/../') $\Rightarrow$ DOTDOTelim s = s;
DOTDOTelim ('/../'^s) = DOTDOTelim (/^s);
s2 $\neq$ '..' $\wedge$ (/ $\notin$ s2) $\Rightarrow$
DOTDOTelim (s1^'/'^s2^'/../'^s3) = DOTDOTelim (s1^'/'^s3);

```
        ¬(s contains '/./') ⇒ DOTelim s = s;
        DOTelim (s1^'/./'^s2) = DOTelim (s1^'/'^s2);

        make_abs(ε, st) = fst(_$(st, CWD));
        s ≠ ε ⇒ make_abs(s, st) = if first s = /
                                  then s
                                  else fst(_$(st, CWD)^'/'^s)
                                  endif;

        δ(String2Path(s, st)) ⇔ ¬(s contains '//');
        String2Path(s, st) = path(DOTDOTelim(DOTelim(make_abs(s, st))));

    endaxioms;
}
```

Mit Hilfe von STRING2PATH können die Systemkommandos nun leicht spezifiziert werden.

```
SYSTEM = {

    enriches STRING2PATH + SYSTEM_CORE;

    legal_varid : String ⟶ Bool strict total;
    legal_relpath : String ⟶ Bool strict total;

    $ : State × String ⟶ String × Error strict total;
    setenv : State × String × String ⟶ String × Error strict total;
    cd : State × String ⟶ String × Error strict total;
    pwd : State ⟶ String × Error strict total;
    ls : State × String ⟶ String × Error strict total;
    mkdir, rmdir : State × String ⟶ String × Error strict total;
    touch, rm : State × String ⟶ String × Error strict total;

    axioms ∀ s, s' : String; st : State in

        legal_varid s = (s ≠ ε);
        legal_relpath s = (s ≠ ε ∧ ¬(s contains '//'));

        $(st, s) = if legal_varid s
                   then _$(st, String2Var s)
                   else (st, Illegal_Variable_Identifier)
                   endif;
```

```
        setenv(st, s, s') = if legal_varid s
                            then _setenv(st, String2Var s, s')
                            else (st, Illegal_Variable_Identifier)
                            endif;

        cd(st, s) = if legal_relpath s
                    then _cd(st, String2Path(s, st))
                    else (st, Illegal_Path_Identifier)
                    endif;

        pwd st = _pwd st;

        ls(st, s) = if legal_relpath s
                    then _ls(st, String2Path(s, st))
                    else (st, Illegal_Path_Identifier)
                    endif;


        mkdir(st, s) = if legal_relpath s
                       then _mkdir(st, String2Path(s, st))
                       else (st, Illegal_Path_Identifier)
                       endif;

        rmdir(st, s) = if legal_relpath s
                       then _rmdir(st, String2Path(s, st))
                       else (st, Illegal_Path_Identifier)
                       endif;

        touch(st, s) = if legal_relpath s
                       then _touch(st, String2Path(s, st))
                       else (st, Illegal_Path_Identifier)
                       endif;

        rm(st, s) = if legal_relpath s
                    then _rm(st, String2Path(s, st))
                    else (st, Illegal_Path_Identifier)
                    endif;

    endaxioms;
}
```

Kapitel 4

Bemerkungen zur Shell

Unter UNIX stehen dem Benutzer gewöhnlich mehrere verschiedene Shells zur Verfügung, zum Beispiel die Bourne-Shell, die C-Shell oder die Korn-Shell. Der folgende Abschnitt enthält einige kurze Bemerkungen zu Aufgaben und Eigenschaften solcher Shells. Es ist jedoch nicht beabsichtigt, dieses Thema hier vollständig zu diskutieren. Insbesondere wird keine Spezifikation einer der gebräuchlichen UNIX-Shells gegeben.

Shells dienen zur Kommunikation des Benutzers mit dem UNIX-System. Eine Shell ist ein Kommandozeileninterpreter, der Benutzereingaben liest, in Systemaufrufe übersetzt und die Ergebnisse der Aufrufe für den Benutzer aufbereitet und ausgibt. Standardmäßig erfüllt jede Shell die folgenden Aufgaben:

Eingabe Einlesen und syntaktische Analyse der Benutzereingabe.

Interpretation Umsetzen der durch die syntaktische Analyse aufbereiteten Eingabe in Aufrufe von Systemkommandos. Dabei ist es in sinnvollen Fällen gestattet, ein Kommando durch Angabe von mehreren Argumenten mehrfach ausführen zu lassen. Zum Beispiel sollte die Benutzereingabe `'ls /usr /usr/include'` durch die Shell in einen zweifachen Aufruf des Systemkommandos `ls` umgesetzt werden.

Ausgabe Aufbereiten der Ausgabe der aufgerufenen Systemkommandos und Ausgabe an den Benutzer. Sollte das Systemkommando einen Fehler ungleich OK liefern, generiert die Shell eine Fehlermeldung und gibt sie an den Benutzer aus.

Dieses Grundverhalten eines Kommandozeileninterpreters wird in der vorliegenden Arbeit nicht spezifiziert. Eine vollständige Spezifikation eines solchen Kommandozeileninterpreters könnte jedoch als eigenständige Fallstudie von Interesse sein.

Über das oben beschriebene Standardverhalten hinaus stellen Shells üblicherweise eine Reihe von Funktionen zur Verfügung, die dem Benutzer den Umgang

mit dem System erleichtern sollen. Zwei einfache Beispiele für solche Funktionen sollen hier vorgestellt und spezifiziert werden:

'//' **in Pfadbezeichnungen** Strings, die zwei aufeinanderfolgende /-Zeichen enthalten, sind normalerweise keine legalen Pfadbezeichner (siehe Spezifikation SYSTEM). Die meisten Shells akzeptieren jedoch auch solche Strings als Pfadbezeichner und interpretieren mehrere aufeinanderfolgende / als eines.

~**-Expansion** Neben absoluten und relativen Pfadbezeichnungen erlauben manche Shells auch die Angabe von Pfaden relativ zum HOME-Directory des jeweiligen Benutzers. Solche Pfadbezeichnungen beginnen mit '~/'. Ist zum Beispiel das HOME-Directory eines Benutzers '/usr/meier', so steht '~/bin' für den Pfad '/usr/meier/bin'.

Diese beiden Eigenschaften werden in der Spezifikation SHELLFEATURES formal spezifiziert.

SHELLFEATURES = **{**

enriches SYSTEM + STRING + PROJ;

mk_legal : String ⟶ String **strict total**;
tilde_expand : State × String ⟶ String **strict total**;

axioms ∀ a : α; b : β; s1, s2 : String; st : State **in**

¬(s1 contains '//') ⇒ mk_legal s1 = s1;
mk_legal (s1^'//'^s2) = mk_legal (s1^'/'^s2);

¬('~/' is_prefix_of s1) ⇒ tilde_expand (st, s1) = s1;
tilde_expand (st, '~/') = fst($(st, String2Var 'HOME'))^'/'^s1;

endaxioms;
}

Kapitel 5

Abschließende Bemerkungen

Diese Arbeit gibt eine SPECTRUM-Anforderungsspezifikation des UNIX Datei- und Variablensystems. Durch Modularisierung und die Schichtung in Systemkern, Systemkommandos und Eigenschaften der Shell wurde versucht, die Spezifikation übersichtlich und verständlich zu halten. Es wurde auch versucht, die Spezifikation so zu gestalten, daß sie um weitere Konzepte von UNIX-Systemen erweitert werden kann.

Zum Abschluß dieser Arbeit soll noch auf einige mögliche Erweiterungen der hier gegebenen Spezifikation kurz eingegangen werden:

Links In UNIX-Systemen existiert das Kommando `ln`, das es erlaubt, sogenannte *Links* auf Dateien oder Directories einzurichten. Ein Link ist ein Eintrag in einem Directory (ein in diesem Directory bekannter Bezeichner), der auf eine Datei oder ein Directory in einem anderen Teil des Dateisystems verweist. Durch Links wird die Baumstruktur des Dateisystems durchbrochen, da es nun möglich ist, Dateien und Directories auf vollständig unterschiedlichen Wegen zu erreichen.

Man unterscheidet in UNIX-Systemen zwei verschiedene Arten von Links, *harte* und *weiche Links*. Weiche Links werden oft auch *symbolische Links* genannt. Harte Links können nur auf Objekte (Dateien bzw. Directories) zeigen, die im Dateisystem vorhanden sind. Objekte, auf die ein harter Link zeigt, können nicht gelöscht werden. Im Gegensatz dazu können symbolische Links 'ins Leere zeigen', d.h. die Objekte, auf die sie verweisen, müssen nicht notwendigerweise existieren.

Aus diesem Grund sollten bei einer Erweiterung der Spezifikation um das Kommando `ln` harte Links in den Systemkern aufgenommen werden, wogegen symbolische Links auf der Ebene der Systemfunktionen spezifiziert werden sollten, zum Beispiel als spezielle Art von Dateien, deren Inhalt ein Pfadbezeichner ist.

Attribute für Dateien Man könnte in einer weiteren Ausbaustufe Dateien und Directories mit Attributen wie Zugriffsrechten, Zeitstempel und Größe versehen, um realen UNIX-Systemen näherzukommen.

Shell Die Spezifikation eines Kommandozeileninterpreters (siehe Kapitel 4) stellt sicher eine interessante SPECTRUM-Fallstudie dar.

Für gründliches Lesen und wertvolle Hinweise beim Erstellen dieses Berichts bin ich meinen Kollegen H. Hußmann, F. Regensburger und B. Rumpe zu Dank verpflichtet.

Literaturverzeichnis

[BFG+92] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 0.3. Technical Report TUM-I9140, Technische Universität München, 1992.

[BGM89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of Computer Programming*, 12(1):1–38, 1989.

[Bou83] S. R. Bourne. *The UNIX System*, volume 6 of *International computer science series*. Addison-Wesley, London, 1983.

[FHL+91] J. Fuchs, A. Hoffmann, J. Loeckx, L. Meiss, J. Philippi, M. Stolz, M. Wolf, and J. Zeyer. *The OBSCURE Manual. Part 1: Editing and Rapid Prototyping*, 1991.

[HLU91] Ramses A. Heckler, Jacques Loeckx, and Stephan Uhrig. Ein Fallbeispiel: Das Dateisystem von UNIX. Working Paper WP91/30, October 1991.

[MS87] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. In Ian Hayes, editor, *Specification Case Studies*, pages 91–140. Prentice Hall, 1987.