Projektbericht LiVE!

LiVE! —
Linux Verification Enterprise

Projekt WS97/98 bis SS 99

Revision : 1.38

Betreuer:
Dr. Bettina Buth, Prof. Dr. Jan Peleska

Teilnehmende:
Hans–Jürgen Ficker, Ewgeni Gisbrecht, Mark Hapke, Matthias Intemann,
Sönke Jarré , Klaus Lüttich, Matthias Riese,Raymond Scholz,
Hauke Steenbock, Harald Wagener, Klaas–Henning Zweck

`$Id: main.tex,v 1.38 2000/08/18 14:16:37 hollow Exp $`
`First Edition by: Klaus Lüttich, Harald Wagener`
`Last Update by: $Author: hollow $`

Vorwort

Eine Besonderheit des Informatik-Studiums an der Universität Bremen ist das studentische Projekt im Hauptstudium. Es stellt für Lernende und Lehrende eine Herausforderung dar, denn es verlangt die sinnvolle Zusammenarbeit in der Gruppe zu einem mehr oder weniger konkret vorgegebenen Thema über zwei Jahre hinweg. Neben allerlei Tücken birgt dieses Konzept aber auch die Möglichkeit, im Rahmen des Studiums Erfahrungen auf Gebieten zu sammeln, die auf die Arbeit nach dem Studium vorbereiten sollen; dazu zählen Projekt-Organisation ebenso wie Teamarbeit und natürlich spezielle Inhalte der Informatik.

Bei der Planung des Projektes “Live! - Linux Verification Enterprise” stand ein wichtiges Ziel im Vordergrund: die Untersuchung, inwieweit die vorhandenen formalen Methoden tatsächlich (schon) für die Verwendung im Kontext von realistischen Betriebssystemen praktisch einsetzbar sind. Die Zuverlässigkeit von Betriebsystemen oder Betriebssystemkomponenten spielt inbesondere bei sicherheitskrischen, eingebetteten Systemen eine grosse Rolle und ist damit kein rein akademisches Thema. Es war jedoch nicht das Ziel – auch wenn der Name des Projektes dies suggeriert – das Linux-System als Ganzes zu verifizieren. Die Auswahl von Linux als Grundlage für die Fallstudien, die im Projekt behandelt werden sollten, ist vor allem begründet in der großen Verbreitung des Betriebssystems (fast jeder Studierende hat schon damit gearbeitet) und der Verfügbarkeit der Quellen. Hierbei war die Idee, daß eine Modifikation des Betriebssystemes durchaus eine Option sein sollte. Vermutungen wonach wir Linux ausgewählt haben um eine möglichst große Anzahl Studierender anzulocken, weisen wir strikt von uns.

Schon vor Beginn des Projektes war uns Lehrenden klar, daß die eigentlich spannende Phase sich in die zweite Hälfte des Projektes verlagern würde, denn es war zunächst notwendig mit den Lernenden gemeinsam die Grundlage im Bereich Formale Methoden und Detailkenntnisse über Linux zu erarbeiten. Wir waren überaus erfreut darüber mit wieviel Motivation “unsere” Projektstudierenden diese Phase ertragen und mitgestaltet haben, obwohl wir ihnen in relativ kurzer Zeit eine ganze Palette von neuen Formalismen und Methoden nahegebracht haben. Dabei standen Test- und Verifikationsansätze im Vordergrund, die auf Z- und CSP-Spezifikationen basieren.

Im der zweiten Projekthälfte standen dann die Fallstudien im Vordergrund. Im Projektplenum wurden verschiedene, interessante Bereiche von Linux und Linux-Anwendungen für die Verwendung der erlernten Methoden diskutiert. Es kamen vier Teilprojekte zu sehr unterschiedlichen Themen zustande (Interprozesskommunikation, Symmetric Multi Processing, Access Control Lists und Fehlertoleranter Mailserver), wobei wir alle uns nicht immer klar darüber waren, wer schlussendlich die Auswahl des eine oder anderen Themas beschlossen hatte. Unter dem Zeitdruck des schon drohenden Projekttages gingen die Arbeiten an diesen Themen auch zügig voran - wir Lehrenden waren wieder einmal positiv überrascht, wie wenig Organisation und Druck von unserer Seite notwendig waren. Mit den Ergebnissen, die auf dem Projekttag präsentiert wurden, können wir jedenfalls alle höchst zufrieden sein.

Eines der wesentlichen Probleme stellte sich aber – wie durchaus nicht unüblich – erst nach dem Ende der inhaltlichen Arbeiten: trotz guter Vorbereitung schon zu Beginn der zweiten Phase des Projektes, brauchte die Dokumentierung der Arbeit und Ergebnisse in Form eines Projektberichtes deutlich mehr Zeit als erwartet. Da halfen auch keine regelmässigen Treffen nach Ende der Veranstaltung “Live!”. Ob es an inhaltichen oder anders gelagerten Umständen lag, bleibt eines der eher ungeklärten Projekt-Geheimnisse.

Abschliessend wollen wir Lehrenden aber eines klarstellen:

“Live!” hat uns viel Spaß gemacht!

Im August 2000

Bettina Buth Jan Peleska

Inhaltsverzeichnis

Teil I

Einleitung

1. LiVE! – The Linux Verification Enterprise

LiVE! – Kurz für Linux Verification Enterprise – ein Projekt für Studenten an der Universität Bremen, eine Möglichkeit, dem Hauptstudium einen Schwerpunkt zu geben.

Aber welchen? Allein vom Namen her gibt es drei Möglichkeiten:

1. **Linux** – und das schon 1997, also zu einer Zeit, wo die IT–Branche international noch von der übermächtigen Präsenz weniger großer Firmen geprägt war und Linux nur von Personen auf Abwegen[1] genutzt wurde. Doch für LiVE! war Linux nur eine Grundlage, die Basis, die unsere Arbeit möglich machte. Abgesehen vom Wandel zum modischen Nischenprodukt ist Linux nämlich immer noch das, was es für dieses Projekt so attraktiv gemacht hat: Ein echtes, in allen Belangen freies Betriebssystem, dessen Quellen uns zur Verfügung stehen und Grundlage für wissenschaftliche und informationstechnische Arbeit sind.

2. **Verification** – Verifikations– und Testverfahren waren schon zu Beginn des Projektes und sind immer noch interessant in Anwendung und Erforschung. Sicherlich ist in diesem Bereich der Schwerpunktbildung der größte theoretische Anteil des Projektes zu suchen, aber damit auch der, daß wir eben nicht nur langhaarige Hacker ohne Bezug zur Außenwelt sind, die „ihr “Linux mit Händen und Füßen verteidigen, sondern auch kritisch begutachten können. Was nicht heißen soll, daß die Beschäftigung mit Test– und Verifikationsmethoden den durchschnittlichen Informatiker näher an die Realität bringt.

3. **Enterprise** – Salopp mit Abenteuer, nicht Unternehmen übersetzt[2], war auch das ein nicht zu unterschätzender Anteil. Natürlich ist jedes Haupstudiumsprojekt selbst ein Abenteuer, und immer stellen sich Fragen wie *Wird was draus, wozu ich stehen kann?*, *Passen wir alle zusammen?* oder *Bekommen wir irgendwann einen vernünftigen Projektraum?* Aber deswegen muß dieses Faktum nicht im Namen verewigt werden.

   Linux selbst ist natürlich immer ein Abenteuer, so wie jedes andere Betriebssystem auch. Aber man hat hier mehr Möglichkeiten, es handelt sich um kein grafisch ummanteltes, mit Netzen und doppeltem Boden abgesichertes Unternehmen, das bei uns, die wir echten Betriebssystem–Abenteuerhunger mitbrachten, maximal zur Frustration führte. Vielmehr ist es ein unerforschtes, in vielen Bereichen unerforschtes Gebiet, das genügend Stolperfallen auch für erfahrene Code–Slinger birgt.

   Aber auch die Verifikation birgt einen besonderen Reiz. Ob es das Abenteuer ist, den eigenen Geist neuen Strukturen zu öffnen, oder

[1] also idealistischen Studenten und Firmen sowie zwanghaften Hackern
[2] Das muß man eben wie bei *Star Trek* sehen

einfach darin begründet, das man vor dem Tool sitzt wie vor dem Orakel und versucht, die Fehlermeldungen wie die berühmten Sprüche aus Delphi zu verstehen und zu ergründen, oder ob es um die Exploration geeigneter Methoden für ein bestimmtes Problem ging – auch hier bot sich uns ein weiter, unerforschter Dschungel.

Der Projektbericht ist nur ein kleiner Ausschnitt aus dem, was wir in diesen vier Semestern gesehen und erlebt haben – niemand schrieb Epen über Kohlfahrten; die Relevanz von Netzbelastungstests wird nur Eingeweihten in Erinnerung bleiben[3], Projektmanagement ist ein Ausrüstungsgegenstand gewesen, mit dem wir uns nur wenig belastet haben.

Dennoch: Vier Semester LiVE! haben uns alle maßgeblich geprägt, und auch die Ziele des Projektes haben wir im großen und ganzen erreicht. Aber nun, nach vielen Worten über Linux, Verification und Enterprise, noch mehr Worte über Access Control Lists, Fehlertolerante Systeme, Message Queues und Symmetrisches Multiprocessing.

Barcelona, im Februar 2000
Harald Wagener

[3] obwohl sich hier fachbereichsweit eine gewisse Tradition gebildet hat

2. Das Wort zum Projekttag

Bei uns hat sich für den Projekttag die besondere Schwierigkeit dargestellt, daß wir zum einen schon inhaltliches vermitteln wollten, uns aber auf der anderen Seite bewußt war, daß Teilbereiche unserer Themen ohne eine grundlegende Einführung auch in die Theorien, die hinter Test– und Verifikationsmethoden stehen, für das allgmeine Publikum nicht nur langweilig, sondern auch unverständlich bleiben würden.

Durch die Aufteilung von LiVE! in die Unterbereiche ACL, FTS, IPC und SMP hatten wir zusätzlich noch das Problem, daß – inclusive einer fünfminütigen Einführung – pro Teilprojekt gerade noch zehn Minuten übrig blieben, um „echte Inhalte“zu präsentieren.

Außerdem wollten wir dem Publikum ja auch verdeutlichen, daß wir trotz der geringen überschneidungen der Teilthemen trotzdem als Projekt gearbeitet haben und nicht etwa vier Zwergprojekte waren, die sich zufälligerweise einen Raum teilten[1].

Als Präsentationsform entschieden wir uns letztlich für die gut vorbereitete Expertenrunde, die sich in einer *TalkShow* den neugierigen Fragen eines Moderators stellt. Da sowas ohne Vorbereitung immer gründlich in die Hose geht[2], haben wir uns die Mühe gemacht, ein Script zu schreiben. Dieses Script ist hier in leicht abgewandelter Form abgedruckt, aber obwohl ihm die Standup–Comedy–Einlagen des Projekttages fehlen, hoffen wir, daß er einen schnellen Überblick über die Inhalte des Projektberichts bieten kann.

Das Projekttags–Script

```
Script: Linux Is Very Entertaining, 16.07.1999 (PDAY-LiVE)

-----------------------------------------------------------
-----------------------------------------------------------
Teilnehmer:

M - Moderator (Harald)

A - Vertreter Gruppe ACL (Hauke)

F - Vertreter Gruppe FTS (Raymond)

I - Vertreter Gruppe IPC (Hans-Jürgen)

S - Vertreter Gruppe SMP (Mark)

V - Vortragender Gruppe SMP (Klaas)
```

[1] Wir waren nämlich richtig tolle Kumpels alle, jawoll!
[2] zum Beispiel, weil die Experten nicht so gut vorbereitet sind

(BKB, JP, BB, EMM)

Hilfsmittel:

B - BEAMER
F - FOLIE
T - TAFEL
D - DEMO-SYSTEM

[B: intro-logo]

M: Guten morgen, Lehrende und Lernende! Ich freue mich, Sie alle hier zum ersten Beitrag am Projekttag begrüßen zu dürfen. Wir, das Projekt LiVE!- kurz für dem Linux Verification Enterprise - , werden ohne Showtreppe, ohne Orchester und ohne Lightshow, aber mit vielen Inhalten glaenzen.

Der Schwerpunkt des Projekt LiVE! war nicht die Verifikation des gesamten Linux-Kernels, denn wir wollten das Projekt in den normalen Zeiträumen abschließen. Vielmehr haben wir formale Verfahren und Werkzeuge benutzt, um Linux in sicherheitskritischen Bereichen einsetzen zu können. Die verschiedenen Teilbereiche des Projektes haben dabei Teile des Kerns neu- oder reimplementiert, oder bestehende Komponenten verifiziert.

[B: intro-linux]

Wir haben Linux aus verschiedenen Gründen als Basis für unsere Arbeit gewählt:

- Der Linuxkern ist eine komplette, funktionierende Beispielimplementierung für ein Unixsystem, das breite Anwendung findet. Es krankt nicht an den Nachteilen von rein akademischen System, die oft genug nur Konzepte implementieren, aber nicht wirklich einsetztbar sind.

- Wir hatten in einem Open Source Project nicht nur die Möglichkeit, in den frei verfügbaren Quellen zu arbeiten, sondern auch, unsere Arbeit an die Community zurückzugeben. Dies ist insbesondere mit den Ergebnissen der Gruppe ACL geschehen.

- Ein Tool, das wir benutzt haben, ist auch für die Linux-Plattform erhältlich. Das hatte für uns den Vorteil, das wir auf den gleichen Rechnern, die Basis unserer Entwicklung sein sollten, auch die Tests fahren konnten.

[B: intro-gruppen]

Ich möchte jetzt die Vertreter der einzelnen Teilgruppen des Projektes vorstellen. Für die Gruppe ACL ist Hauke zugegen, Hans-Jürgen zu meiner linken vertritt die IPC - Leute. Auf der anderen Seite sitzt Raymond, der uns über die Erfahrungen der Gruppe FTS berichten wird und last not least haben wir Mark dabei, der die SMP-Gruppe vertritt.

Mark, was war Euer Aufgabengebiet?

S: Ursrünglich wollten wir uns it den Linux-eigenen SMP-Eigenschaften des Kernels beschäftigen. Im Projektverlauf so ergab es sich aber so, dass wir uns mehr mit der verteilten Nutzung von Ressource beschäftigt haben.

M: Eure Zielsetzung hat sich also von einem speziellen Fall auf eine allgemeine Problematik verschoben?

S: Stimmt. Dennoch haben wir den Kern in einigen Punkten verändert, wenngleich diese Änderungen nicht so einschneidend sind, wie zu Beginn vermutet.

M: Und Eure Methoden?

S: Wir haben unser System vor der Implementierung spezifiziert, um das Kommunikationsverhalten der verschiedenen Prozesse testen zu können.

M: Wo Du Prozeßkommunikation erwähnst, mochte ich erstmal mit Hans-Jürgen weitermachen. Erzähl doch mal, womit Ihr Euch beschäftigt habt.

I: Hier muß ich weiter ausholen: In einem _richtigen_ Betriebssystem wie Linux gibt es Prozesse, die miteinander kommunizieren wollen. Seit UNIX System V gibt es dazu das IPC-Paket, wobei IPC für Inter-Process-Communication steht. Unser Teilprojekt hat sich mit der formalen Spezifikation und Verifikation des IPC-Paketes, und speziell der Message Queues beschäftigt.

M: Was bitte sind denn Message Queues?

I: Message Queues sind ein Mechanismus des IPC-Pakets, die es einem Prozess ermöglichen, Nachrichten an einen anderen Prozeß zu schicken.

M: Und wie war Euer Vorgehen?

I: Anhand der vorhandenen Dokumentation haben wir die Message Queues formal spezifiziert, und das System anschließend gegen diese Spezifikation verifiziert.

M: Welche Dokumentation habt Ihr benutzt?

I: Hauptsächlich haben wir uns auf die manpages bezogen, aber natürlich haben wir auch den Quellcode gesichtet. In Zweifelsfällen haben wir auch kleine Testprogramme geschrieben, die das momentane Verhalten zeigten.

M: Das soll hier erstmal reichen. Ich komme später nochmal darauf zurück. Hauke - mit welchem Problem habt Ihr Euch auseinandergesetzt?

A: Linux stellt bisher lediglich die Standard Unix Rechte für die Zugriffskontrolle auf Dateien zur Verfügung. Diese erlauben es nicht, Rechte für individuelle User und Gruppen festzulegen, wie es beispielsweise bei Novell Netware und auch bei verschiedenen kommerziellen UNIX-Versionen möglich ist.

M: Was kann man da nun machen?

A: Diese Unix-Versionen verwenden Access Control Lists - kurz ACLs - die so eine individuelle Rechtevergabe erlauben. Wir haben ACL-Support für Linux realisiert. Dazu gehörte eine Erweiterung des Betriebssystemkerns und die Implementierung der Dienstprogramme auf Benutzerebene.

M: Und was hat das mit Verifikation zu tun?

A: Ein großer Teil unserer Arbeit bestand darin, die Korrektheit unserer Implementierung durch einen umfangreichen automatisierten Test zu prüfen. Außerdem haben wir den Algorithmus zur Zugriffskontrolle formal verifiziert.

M: Raymond, Ihr habt Euch mit etwas anderen Dingen beschäftigt als die anderen Gruppen. Was habt Ihr gemacht?

[B: fts-intro]

F: Wir haben die konkrete Anwendung von Linux in einem sicheheitsrelevanten Bereich erprobt. Dazu haben wir einen fehlertoleranten Mailserver konzipiert und gebaut, der _bald_ hier im Fachbereich eingesetzt werden wird. Diese Lösung besteht nicht nur allein aus der Hardware - sie zeichnet sich auch durch intelligente Software-Komponenten aus.

M: Fehlertolerant?

F: Ja, denn der neue Mailserver hat eine erhöhte Ausfallsicherheit durch den Einsatz von Redundanz an mehreren Punkten. Wir haben z.B. wichtige Komponenten doppelt ausgelegt, so dass beim Ausfall der einen die andere weiter arbeiten kann.

M: Wie seid ihr vorgegangen?

[B: fts-alt]

F: Nun, wir haben einfach einen Rechner zusammengebaut, Linux installiert und geguckt, ob es funktioniert. Hoffentlich klappt alles, wie vorgesehen.

M: Das ist nicht euer Ernst, oder? [(Klaus|Matthias) einen Pseudo-MS hereinbringend]

[B: fts-altneu]

F: Ähhhh nein, das ist unser (Klaus|Matthias)... Der richtige Ansatz ist natürlich, systematisch vorzugehen. Dazu haben wir erst einmal die Schwachpunkte des alten Mailservers analysiert und anschließend ein neues, fehlertolerantes System konzipiert.

M: Aber wie kann man denn schon im voraus mögliche Risiken erkennen, die später beim Betrieb zu schwerwiegenden Fehlern und Ausfällen führen können?

[B: fts-fta1]

F: Dazu gibt es geeignete Methoden, z.B. die Fehlerbaumanalyse. Ausgehend von Fehlern (Faults) in einer Komponente lassen sich damit Aussagen über das Verhalten des Gesamtsystems machen. Die Fehler in den Teilkomponenten werden dabei mit den booleschen Operatoren UND, beziehungsweise ODER, verknüpft, je nach den tatsächlichen Zusammenhängen im System. Und wie alle Bäume der Informatiker, steht auch dieser auf dem Kopf: Ganz oben befindet sich die größte Katastrophe, der Top-Hazard. [to be changed]

M: Was kann man denn jetzt mit diesen Bäumen anfangen?

F: Wir konnten Schwachstellen des alten Mailservers identifizieren, die zum Ausfall des gesamten Systems geführt hätten. Beim Aufbau des neuen Mailservers konnten wir z.B. den Einsatz von redundant ausgelegten Komponenten auf Wirksamkeit überprüfen.

[B: fts-fta2]

Hier ein kleiner Ausschnitt aus unserem Fehlerbaum...

M: Wie sieht er denn nun aus, der neue Mailserver?

[B: fts-ms1]

F: Der Mailserver besteht aus zwei unabhängigen Teilrechnern, von denen einer der aktive Mailserver ist. Beide Teilrechner verfügen über eine doppelte Netzanbindung. Die Mails der Benutzer werden extern auf einer gespiegelten Festplatte gespeichert. Ist der aktive Teilrechner überlastet, hilft der passive beim Annehmen von Mails aus.

M: Was passiert, wenn der aktive Teilrechner nun doch einmal völlig ausfällt?

[B: fts-ms2]

F: Dann kann innerhalb von 2 Minuten der passive Teilrechner zum neuen aktiven Mailserver werden. Für die Benutzer des Mailservers fallen keine Konfigurationsänderungen an.

M: Um noch einmal auf eure Fehlerbäume zurückzukommen; habt ihr noch weitere Ergebnisse aus ihnen gewinnen können?

F: Es lassen sich sehr einfach die Bedingungen für den Betrieb des Mailservers extrahieren, die minimal erfordlich sind, damit er die spezifizierte Leistung erbringen kann. Dies ist eine sogenannte Sicherheitsspezifikation.

M: Als Fachmann für Spezifikationen ist heute auch Hans-Jürgen hier. Erzähl mal, was ist eine Spezifikation?

[B: ipc-spec $$! noch zu zeichnen!!]

I: Grob gesagt beschreibt eine Spezifikation ein System. Meistens dient sie dazu, dem Programmierer zu sagen, wie sein Programm ablaufen soll. Er hantiert dabei mit Daten und den Veränderungen dieser Daten, häufig spielt dabei die Zeit noch eine Rolle. Hier haben wir also verschiedene Betrachtungsweisen eines Systems. Auf entsprechende Arten kann das System dann auch beschreiben.

M: Und was für Arten von Spezifikationen gibt es?

I: Zuerst muß man zwichen informaler und formaler Spezifikation unterscheiden. Machen wir dazu mal ein Experiment.

I: Ray, würdest Du bitte mal einen Tisch auf dieses Blatt malen. So, und nun frage ich mal Mark, wie stellst Du Dir einen Tisch vor?

S: groß, hat 4 Beine und eine Tischplatte!

I: Da frage ich doch mal lieber nach, welche Form hat der Tisch?

S: Oval, wuerde ich mal sagen.

I: Aus welchem Material ist er?

S: Aus Holz, daher ist er auch dunkelbraun. Halt ein Wohnzimmertisch.

R: Bin fertig <hält Zeichnung hoch>.

I: Wie man nun sieht gibt es etliche Unterschiede zu dem von Mark beschriebenen Tisch: Dieser Tisch ist eckig und wesentlich heller. Danke euch beiden.

Die Beschreibung und das Bild waren eben zwei informelle Spezifikationen des einfachen Systems Tisch. Dabei haben wir schon verschiedene Probleme der informellen Spezifikation kennengelernt. Bei der verbalen Beschreibung mußte ich schon nachfragen, um wichtige Informationen zu bekommen. Die Zeichnung war da schon genauer, allerdings gibt sie immer noch keinen Aufschluß z.B. über die Größe des Tisches.

Ja, und wenn wir schon hierbei verschiedener Meinung sind, wie soll das erst enden, wenn wir das Kühlsystem eines Kernkraftwerks oder die Steuerung eines Flugzeugs beschreiben wollen?

M: Warum spezifiziert man nicht einfach alles formal?

I: Formale Spezifikationen sind erst mal wesentlich aufwendiger. Wenn man ein System mit seinen sämtlichen Eigenheiten beschreiben will, dann wird es einfach sehr umfangreich. Begibt man sich aber in sicherheitskritische Bereiche wie die Raumfahrt oder Atomkraft, dann werden durch die Ungenauigkeiten bei der informalen Spezifikation auch Menschenleben gefährdet. Und da lohnt sich der Aufwand.

M: Und warum habt ihr dann formale Spezifikation betrieben?

I: Wenn so etwas wichtiges wie ein Betriebssystem versagt, dann ist der ganze Computer nicht mehr sicher zu benutzen. Dabei ist nicht einmal ein Absturz das schlimmste, weil man ihn sofort bemerkt, sondern schleichende Fehler, die die Daten verfälschen. Und wir haben nun in unserem Teilprojekt versucht, einen Teil des Betriebssystems in einer formalen Spezifikationssprache zu erfassen.

M: Dann gibt es also mehrere formale Spezifikationssprachen?

I: Ja. Im Projekt haben wir CSP, Z und ASPecT benutzt. CSP eignet sich dabei für die Spezifikation von parallel arbeitenden Prozessen. Z eignet sich dagegen eher für die Modellierung von Datenstrukturen. Und ASPecT ist eine funktionale Programmiersprache, mit der man bestimmte funktionale, ähm, Aspekte gut modellieren kann.

M: Wenn ich es richtig weiß habt ihr in eurem Teilprojekt Z benutzt. Warum?

I: Wir wollten die internen Datenstrukturen der MessageQueues beschreiben, und nicht nur das nach außen sichtbare Verhalten. Was bringt es uns, wenn wir zwar wissen, daß eine Nachricht ankommt, daß dabei aber interne Kernel-Strukturen überschrieben werden, und mir dann drei Tage später die Festplatte abraucht?

M: Wie sieht denn so eine Spezifikation in Z aus?

I: Z ist eine mengenbasierte Spezifikationssprache. Alles ist auf Mengen aufgebaut und es sind nur entsprechende Mengenoperationen erlaubt. Ich habe hier ein kleines Beispiel aus unserer Spezifikation:

[B: ipc-msqdef]

Wie wir hier schon sehen, gehört zu einer formalen Spezifikation auch eine Menge informeller, beschreibender Text. Denn einfach eine Menge "KEY" zu definieren, ist sehr wenig aussagekräftig. Deshalb schreibt man an dieser Stelle dazu, daß "KEY" <ablesen> die Menge aller Messagequeue-Keys ist, damit verschiedene Prozesse die selbe Queue addressieren können </ablesen>.

Wir haben jetzt keinesfalls eingeschränkt, wie so ein Key aussieht. In der Implementierung ist dies ein Ganzzahliger Wert, aber davon müssen wir hier noch gar nichts wissen.

Außerdem haben wir uns auch einen speziellen Key IPC_PRIVATE herausgegriffen, mit der man sich eine private Messagequeue erzeugt. Auch hier haben wir uns nicht auf einen konkreten Wert festgelegt, weil uns das momentan gar nicht interessiert.

[B: ipc-msgget]

In Z wird viel mit Schemata gearbeitet. Hier haben wir ein Beispiel dafür. Hier oben wird ein anderes Schema eingebunden. Als Eingaben haben wir einen Key, und eine Prozeß-ID, und als Rückgabe bekommen wir eine Messagequeue-ID.
Im zweiten Teil haben wir zunächst die Vor- und Nachbedingungen, die gelten müssen, damit dieses Schema ausgeführt werden kann. Und ganz unten haben wir Veränderungen des Zustands.

M: Das sieht ja recht einfach aus...

Da hast du recht, aber das hier ist auch ein leicht gekürztes Bespiel. Hier haben wir mal als realistisches Beispiel Seite 13 aus unserer 35-seitigen Spezifikation.

[B: ipc-seite13]

M: Hat Z denn auch andere Nachteile?

I: Ein großer Nachteil besteht darin, daß man in Z keine zeitlichen Abläufe beschreiben kann. Wir hatten an einigen Stellen das Problem, das ein Prozeß warten muß, bis er wieder dran ist. Dies läßt sich leider in Z nicht modellieren. Weiterhin gibt es noch den Nachteil, daß es zwar eine Type-Checker gibt, aber daß man Z-Spezifikationen nicht ausführen kann.

A: Aber da kann man dann sehr gut Aspect benutzen!

M: Was genau ist Aspect?

A: ASpecT ist eine funktionale Programmiersprache, mit der sich algebraische Spezifikationen erstellen lassen. Diese Spezifikation läßt sich dann ein ausführbares Programm übersetzen.

M: Wozu kann man solche Programme verwenden?

A: Man erhält dadurch ein Referenzsystem, daß sich exakt gemäß der Spezifikation verhält. Gegen so einen Prototypen lassen sich dann konkrete Implementierungen testen.

M: Außerdem habt Ihr noch CSP verwendet.

A: Ja, mit Hilfe von CSP lassen sich Automaten modellieren. Dies haben wir benutzt, um korrekte Programmaufrufe zu spezifizieren.

M: Mark, auch Ihr habt spezifiziert, richtig?

S: Ja, das stimmt , wir haben eine Spezifikation in CSP geschrieben, die...

M: CSP? Was ist denn CSP jetzt genau? Hauke hat das auch schon erwähnt....

[B: smp-csp $$! noch zu zeichnen??]

S: CSP steht für Communicating Serquential Processes. Das ist eine Sprache, mit der man Prozesse und ihr Kommunikationsverhalten mit der Umwelt beschreiben kann. Gleichzeitig ist es eine Sammlung mathematische Modelle und Beweismethoden.

M: Und wofür wird CSP eingesetzt?

S: CSP wird insbesondere in den Bereichen eingesetzt, wo es auf Fehlertoleranz ankommt, oder wo Kontrollsysteme in Echtzeit antworten können müssen. Im letzteren Fall wird auch eine Variante von CSP, timed CSP eingesetzt werden. Damit kann man zusätzliche Aussagen über die zeitliche Abfolge von Kommunikationsereignissen treffen.

M: Welche speziellen Probleme habt Ihr mit CSP gelöst?

S: Geloest haben wir dadurch nichts, aber von vornherein vermieden. Zum Beispiel haben wir das Auftreten von Verklemmungen dadurch vermieden, dass wir unser System anhand der vorher erstellten Spezifikation implementierten.

M: Jetzt weiß ich grad nicht, ob ich Dich zuerst fragen soll, wie Ihr getestet habt, oder was Ihr denn nun implementiert habt...

S: Dann antworte ich lieber auf die Testfrage. Fuer das Testen benutzten wir FDR, womit man CSP-Specs automatisch auf aktive und passive Verklemmungen untersuchen kann. Die gefundenen Blockierungen werden dann durch schrittweise Verfeinerung aufgeloest.

M: Schrittweise Verfeinerung? Kannst Du das genauer erklären?

S: Könnte ich, aber dann muß heute leider der Projekttag ausfallen. Oder das Sommerfest.

BKB: NEIN!!!!!!

M: Gut, dann komme ich eben nochmal zurück zu Hauke. Ihr habt erwähnt, daß Ihr einen automatischen Testlauf gemacht habt, Wie seid Ihr da vorgegangen?

[B: acl-prog]

<A geht zur Projektionswand>

JP: Oh fein, jetzt wirds interessant!

<BB Schweigt sich aus>

A: Zum Testen haben wir zunächst ein Programm entwickelt, das eine Kommandozeile erhält und diese dem zu testenden System und unserem aus der Spezifikation erzeugten Prototypen zur Ausführung übergibt.
Anschließend vergleicht es die Ergebnise miteinander und meldet richtig, wenn diese übereinstimmen und falsch, wenn nicht.

Es bleibt das Problem, geeignete Kommandozeilen zu erzeugen, die möglichst alle relevanten Testfälle abdecken.

M: Wie habt Ihr das gelöst?

[B: acl-arch]

A: Wir benutzten das Programm RT-Tester, um CSP-Spezifikationen zu interpretieren und die dabei generierten CSP-Events in reale Events in Form von Zeichenketten umzuwandeln.

Dadurch lassen sich unter Verwendung einer geeigneten CSP-Spezifikation zufallsgesteuert syntaktisch korrekte Kommandozeilen erzeugen.

Diese werden dann an unser Testprogramm übergeben. Das Ergebnis wird dann dem RT-Tester zurückgemeldet. Die Testergebnisse werden fortlaufend protokolliert. Nun kann man einen Test über Tage ausführen und anschließend die durch das Testsystem festgestellten Fehlerfälle analysieren.

M: Habt Ihr dadurch noch Fehler gefunden?

A: Ja, wir entdeckten noch einige Fälle, in denen sich unser Programm nicht der Spezifikation gemäß verhielt. Viele dieser Fälle hätten wir so nicht entdeckt, da es sich hier um exotische Kommandozeilen handelte, die wir vermutlich so nicht eingegeben hätten.

Um ehrlich zu bleiben, müssen wir auch erwähnen, daß wir zu Beginn auch einige Fehler in unserer Spezifikation entdeckten.

Unsere jetzige Implementierung haben wir mit über 50.000 Kommandozeilen getestet, ohne daß dabei Fehler aufgetreten sind.

<A kehrt zurück auf seinen Platz>

M: Danke, Hauke. Raymond, Ihr habt Euren Mailserver natürlich auch getestet?

F: Wir haben ebenfalls das Programm RT-Tester benutzt. Damit werden, ausgehend von einer CSP-Spezifikation, verschiedene Mails versandt und empfangen. Die empfangenen Mails werden dabei auf Konsistenz überprüft.

M: Habt ihr so Fehler gefunden?

F: Ja, es wurden Probleme beim konkurrierenden Zugriff auf die Maildateien von Benutzern deutlich. Wir haben z.B. einen Fehler in der Implementierung des Post Office Protocol-Daemons, auch POP-Daemon genannt, gefunden - natürlich nicht unser Fehler!

M: Was ist mit möglichen Bedrohungen für den Mailserver?

F: Die von uns erstellten Fehlerbäume haben uns die Ausfälle in Teilkomponenten aufgezeigt, die zu katastrophalen Fehlern, den Top-Hazards, führen können. Unsere Test-Spezifikation kann diese Fehler stimulieren. So können wir das Verhalten in Extrem-Situationen beobachten.

M: Danke, Raymond. Da fällt mir ein - Mark: Ihr habt ja auch was implementiert.

S: Ja, das PiM-System, das...

M: PiM?

[B: PiM]

S: Parvo in Multum, das Kleine im Vielen. Wie ich gerade sagen wollte: PiM ist ein Treiberaufsatz. Mit PiM ist es möglich, asynchron verschränkt auf eine Ressource zuzugreifen, zum Beispiel eine Schnittstelle.

M: Zum Beispiel...kannst Du da was vorführen?

S: Sicher [geht zur Rechnerbatterie]. Ähh, Klaas? Willst Du das machen?

V: OK. Wir haben hier zwei Rechnerpaare, die über die serielle Schnittstelle kommunizieren. Das Rechnerpaar links verfügt über die normale I/O-Funktionalität eines Linux-Systems. Das Rechnerpaar rechts ist mit den PiM-Erweiterungen versehen.

Unsere Testanwendung verarbeitet mehrere Datenströme, um ein Ergebnis zu berechnen. Dabei ist es so, daß sie um Standardsystem auf alle Daten warten muß, bis sie ein Ergebnis berechnen kann.

Beim PiM-System ist das ganze etwas komplizierter.

[T: smp-funktionsdiagramm]

Die user-Prozesse geben ihre Absichten, mit der Schnittstelle zu kommunizieren, an Clients weiter. Diese kommunizieren mit dem Server, der für jede Anfrage ein Servlet startet, das entweder einen Lese- oder einen Schreibvorgang bearbeitet.
Dabei laufen die Prozesse, die ihre Anfragen abgestzt haben weiter, so sie zum Beispiel Berechnungen ausführen können, ohne

alle Daten vorliegen haben zu müssen.

M: Und CSP hat Euch auch bei der Implementierung direkt geholfen?

V: Sicher. Die Kommunikation zwischen Clients, Servern und Servlets läuft über Message Queues, der Datentransport mit Shared Memory. Man sieht ganz deutlich: Jede Menge Kommunikation. Deswegen ist unser Funktionsdiagramm voll mit diesen Kanten: Jede von ihnen stellt einen Kommunikationskanal dar.

Aber schauen wir mal unsere Anwendung an. Hier links ist die Kommunikation zwar schon fortgeschritten, aber der Prozeß hatte noch nicht die Gelegenheit, auf den ankommenden Daten zu arbeiten. Auf der rechten Seite sehen wir zwei Dinge: Erstens sind schon mehr Daten angekommen, und zweitens hat der empfangende Prozeß schon mit der Bearbeitung begonnen und man kann erste Ergebnisse sehen.

[B: intro-gruppen]

M: Ich glaube das reicht fürs erste. Hauke, was könnt Ihr über Eure Implementierung berichten?

A: Wir haben anfangs ein Alpha-Version im Internet gefunden. Diese stammte von Remy Card, dem Entwickler des Second Extended Filesystems.

Dies ist das Dateisystem, das standardmäßig unter Linux verwendet wird.

Diese Alpha-Version war übersetzbar jedoch noch nicht auf ihre Funktion hi getestet.
Sie hielt sich an den POSIX Standard Entwurf auf dem die Digital Unix ACL Implementierung basiert.

M: Und was habt Ihr mit dieser ALPHA Version getan.

A: Zuerst haben wir eine Reihe von Fehlern beseitigt und sie damit lauffähig gemacht. Das Ergebnis haben wir dann im Internet veröffentlicht.

M: Gab es darauf Reaktionen?

A: Einige Leute fanden es nicht so gut, daß sich die Implementierung nach dem POSIX-Standard-Entwurf richtete - die Kritik an diesem Standard ist auch nicht ganz unberechtigt.

M: Aha. Kannst Du das genauer ausführen?

A: Man merkt, wie hier die einzelnen Firmen auf einen Standard einwirken - und dann eine Einigung verhinderten. Der Standardentwurf war so ausgelegt, daß die einflußnehemenden Firmen, möglichst wenig an ihrer schon vorhandenen Version ändern mußten und nicht daran, daß eine optimale Lösung dabei herauskommt.

M: Aber abgesehen von der Kritik am Standard?

A: Abgesehen davon war die Resonanz durchaus positiv. Auch Remy Card meldete sich bei uns und vergaß auch nicht, uns auf die neueste Version des Standards hinzuweisen.

M: Es gab also eine Rückmeldung aus der Community?

A: Ja. Es war sehr motivierend zu erfahren, daß die Linux-Comunity an unserer Arbeit interessiert ist. Man muß jedoch erst etwas einbringen, bevor man beachtet wird.

M: Aber Ihr hättet auch ein Ergebnis vorweisen können, wenn die Community Eure Arbeit nicht angenommen hätte?

A: Abgesehen davon, daß unsere Implementierung nun konform zum neuesten POSIX-Standard-Entwurf und fehlerfrei ist, haben wir eine Test Suite entwickelt, mit der wir ACL-Implementierungen automatisch auf ihre Korrektheit hin überprüfen können.

M: Ein Blick auf die Organisatoren sagt mir, daß wir unsere Zeit großzügig ausgeschöpft haben. Zum Abschluß möchte ich aber nochmal jeden auf der Bühne bitten, ein Schliußwort anzubringen. Raymond, möchtest Du anfangen?

F: Grundsätzlich ist Linux für den Einsatz in sicherheitskritischen Anwendungen geeignet. Allerdings sind nur wenige Bereiche des Kerns eingehend untersucht worden. Der große Vorteil ist aber die freie Verfügbarkeit der Quellen zu Linux. Bei Problemen kann man die Fehler selbst finden und beheben. Schneller und direkter geht es nicht.

M: Hans-Jürgen?

I: Bei unserer Arbeit haben wir ja als Spezifikationssprache Z verwendet. Diese erschien uns führ unsere Arbeit am besten geeignet zu sein. Allerdings haben wir an einigen Stellen die Nachteile dieser Sprache kennengelernt.

Aber unser wichtigstes Ergebnis ist, daß wir in der vorhandenen Implementierung der Message-Queues keinen Fehler gefunden haben. Man kann also davon ausgehen, daß MessageQueues korrekt arbeiten.

M: Mark?

S: Spezifikation und Implementierung eines Systems sollten gleichzeitig entwickelt werden. Dadurch ergibt sich die Möglichkeit, bei beiden früh Fehler zu entdecken, sowohl konzeptioneller als auch technischer Art.

M: Hauke?

A: Wir haben festgestellt, daß das Prüfen unserer Implementierung auf Korrektheit hin genauso aufwendig ist, wie das Implementieren selbst. Die gilt auch für das automatisierte Testen. Hat man hier jedoch eine geeignetes Test-System geschaffen, geht der Rest wirklich automatisch. Außerdem ist das Testsystem wiederverwendbar. Unser Testsystem läßt sich auch verwenden, um andere Implementierungen zu testen.

[B: intro-logo]

M: Danke. Ich hoffe, alle unsere Zuhörer können etwas aus dem Vortrag mit nach Hause nehmen.

Zum Abschluß möchte ich noch ein paar Worte zum Projekt "an sich" sagen.

Für uns war einer der wichtigsten Punkte, daß wir uns unsere Aufgaben weitestgehend selbst bestimmen konnten, ohne daß wir die Vorstellungen und Ideen der Projektleitung vernachlässigen mußten.

Organisatorische Schwierigkeiten wurden mit Unterstützung der Projektleitung unkompliziert gelöst. Hier gilt unser Dank Bettina Buth, die ihre Aufgabe als Betreuerin stets ernst nahm und uns nicht nur mit klugen Ratschlägen, sondern auch frischem Tee und Schokolade beiseite stand.

Auch außerhalb der Projektarbeit und über die Teilprojekte hinaus gab es Kooperation und gemeinschaftliche Unternehmungen. Hier ein besonderer Dank an Oliver Vogel (in Abwesenheit), der einem Teil des Projektes immer wieder spannende und entspannende Stunden bescherte.

Die größten Schwierigkeiten hatten wir mit dem Projektraum, aber auch hier wurde mit Unterstützung aller Beteiligten doch eine Lösung gefunden - unser besonderer Dank geht hier an Bernd Krieg-Brückner für sein persönliches Engangement und kreative Beteiligung am Lösungsprozeß.

Insbesondere in den letzten sommerlichen Wochen lernten wir die Klimaanlage im Projektraum schätzen...

[B: intro-ende]
<ALLE kommen auf die Bühne>

Die letzte Danksagung geht insbesondere an die, denen wir die erstklassige Ausstattung im Projekt zu danken haben: Cornelia Zahlten für ihre Unterstützung der Gruppe FTS, und natürlich Jan Peleska, der neben seinen anderen Aufgaben immer wieder Zeit für uns gefunden hat.

<ALLE: ziehen ihre Jackets aus, drehen sich um....>

-= FINIS =-

3. Grundlagen

3.1 Am Anfang war ...

...die Zielsetzung. Sie stellte sich folgendermaßen dar:

> Die Ziele des Projektes ergeben sich aus der speziellen Problematik des Einsatzes von Betriebssystemen in sicherheitskritischen Anwendungen:
>
> - die Anforderungen sicherheitsrelevanter Anwendungen an die Zuverlässigkeit von Betriebssystemen sollen verstanden werden;
> - das Betriebssystem Linux soll im Detail besprochen werden;
> - für eine Bespielanwendung sollen sichere Betriebssystemkernkomponenten aus Linux entwickelt werden;
>
> Darüberhinaus soll ein grundlegendes Verständnis für das Arbeiten mit großen, in der Entwicklung befindlichen Systemen vermittelt werden, sowie Kenntnisse über Methoden und Werkzeuge zur Unterstützung der Entwicklung komplexer Systeme. Es werden grundlegende Kenntnisse in den Gebieten Betriebsysteme und Sicherheitskritische Anwendungen vermittelt. Dazu gehören
>
> - Architekturen für Betriebssysteme, insbesondere in sicherheitskritischen Anwendungen;
> - Einsatz Formaler Methoden zur Spezifikation und Entwicklung sicherheitskritischer Anwendungen;
> - CASE Techniken im Bereich von UNIX, insbesondere die Verwaltung von großen Projekten im Hinblick auf Configuration Management, Reengineering und Wiederverwendbarkeit von Ergebnissen aus Verifikation, Validation und Test;
> - Implementierung von Betriebssystemkomponenten; hier sollen für die Anwendung ungeeignete Anteile durch geeignete ersetzt werden (Schwerpunkt werden hierbei Zuverlässigkeit, Echtzeit-Aspekte und Prozesskommunikation sein);
> - Fallstudien, die für den Entwurf und die Entwicklung des sicheren Betriebssystemkerns als Ausgangspunkt genommen werden sollen.
>
>   Als Implementierungssprache soll dabei die Sprache C verwendet werden, die durch die Implementierung des LINUX–Systems vorgegeben ist.

3.2 Was daraus wurde

Der Verlauf des Projektes läßt sich in drei größere Abschnitte unterteilen:

Das erste Semester, in dem grundsätzliche Techniken und Handwerkszeug für Tester und Verifzierer vorgestellt wurden

Das zweite Semester, in dem sich die eigentliche Zielsetzung sowie die einzelnen Teilgruppen herausgebildet haben

Das dritte und vierte Semester, deren Hauptaugemerk auf Anwendung der früher vorgestellten Techniken in den einzelnen Teilgebieten lag.

Durch die breitgefächerte Themenwahl hat sich eine große Abdekcung verschiedener Spezifikations– und Test–Verfahren ergeben, in deren Genuß durch regelmäßige Berichte in den Plena alle Teilnehmenden des Projektes kamen.

Der Projektbericht beschreibt hauptsächlich die Ergebnisse aus der zweiten Projekthälft, aber im aktuellen Kapitel werden kurz die Verfahren, Werkzeuge und Grundlagen[1] vorgestellt, die zum Verständnis des Berichts vorausgesetzt werden. Wem die Begriffe FDR, CSP, AsPECT und Z geläufig sind, kann den Rest dieses Kapitels beruhigt überspringen und sich den selbsterarbeiteten Inhalten des Projektes widmen.

Viel Vergnügen!

[1] wer hätte das gedacht?

4. Einführung in CSP

CSP ist eine Sprache, die 1985 von C. A. R. Hoare veröffentlicht wurde. Sie dient der Beschreibung komplexer Systeme mit mehreren miteinander kommunizierenden Prozessen.

Ein CSP-Prozeß wird durch seine Kommunikation mit der externen Umgebung komplett beschrieben. Diese Kommunikation wird in CSP mittels Kommunikations-*Events* beschrieben. Die Menge aller Events eines Prozesses heißt Σ. Events müssen als *channels* deklariert werden.

```
channel a, b

P = a -> b -> STOP
```

Es kommt hierbei nicht auf die kommunizierten Daten an, sondern auf das Ereignis (engl. *event*) an sich. Dennoch können Daten über diese Kanäle bzw. Events übertragen werden, was bei der Deklaration des Events angegeben werden muß.

```
channel a:{0..1}

P = a!0 -> STOP
Q = a?x -> STOP
```

Dies bedeutet, daß die Zahlen 0 und 1 über *a* kommuniziert werden sollen. Der Prozeß P versendet(*!*) eine 0 über *a* und der Prozeß Q empfängt (*?*) das Datum und speichert es im Objekt *x*, welches ein freier Bezeichner sein kann.

4.1 Spezielle CSP-Prozesse und Events

STOP — *STOP* ist der einfachste CSP-Prozeß, denn er macht nichts und kommuniziert auch nicht. Als Konsequenz sind die Prozesse $STOP \Box P$ und $STOP$ äquivalent, da $STOP$ nicht erfolgreich terminiert[1] und der Prozeß somit keine Gelegenheit hat, sich nach $STOP$ wie P zu verhalten.

SKIP und $\surd$ — *SKIP* ist der Prozeß, der sich sofort beendet. Er produziert vor seinem endgültigen Ende den speziellen Event $\surd$, welcher nicht explizit benutzt werden kann ($\surd \notin \Sigma$). Also ist

$$SKIP = \surd \rightarrow STOP,$$

aber so so darf es nicht notiert werden.
Das $\surd$ zeigt der Umgebung an, daß der entsprechende Prozeß sich nun sauber terminiert.

RUN — Dieser Prozeß tritt in der Form RUN_A auf, wobei $A \subseteq \Sigma$ gilt.
Er kann immer einen beliebigen Event aus A kommunizieren, wenn dieser von der Umgebung angefordert wird.

[1] näheres zur Terminination in CSP im Abschnitt *Termination*

Chaos Auch dieser Prozeß arbeitet auf einer Untermenge A aus Σ, ($Chaos_A$). Er kann sich aber im Gegensatz zu RUN_A auch beliebig weigern, den gefragten Event zu kommunizieren; er verhält sich eben chaotisch.

div *div* divergiert, nichts anderes.

4.2 Operatoren

Zur Beschreibung der Kommunikation eines Prozesses werden die *Events* eines Prozesses mithilfe diverser Operatoren in Beziehung gesetzt.

4.2.1 Allgemeine Operatoren

Prefixoperator Ein Beispiel:

```
P = a -> b -> STOP
```

Dieser Operator, welcher durch ein $\rightarrow$ symbolisiert wird, zeigt an, daß sich ein Prozeß *P* bei einem gegebenen Event *a*, nach dessen geschehen wie der Prozeß $b \rightarrow STOP$ verhält. Hier sieht man auf der linken und rechten Seite des Opertors *Events*, obwohl auf der rechten Seite des Prefixoperators ausschließlich Prozesse folgen dürfen. Das erklärt sich dadurch, daß der folgende Rest des Prozesses (hier $b \rightarrow STOP$) selbst ein Prozeß ist.
Sprich: nach *a* verhält sich *P* wie der Prozeß $b \rightarrow STOP$.
Dieser Operator wird am häufigsten benutzt.

Sequential Composition Wie in mehreren Programmiersprachen auch gibt es in CSP den Operator für die sequentielle Komposition (*;*).

```
P; Q
```

Nachdem P erfolgreich terminiert, verhält sich der Prozeß wie Q.

Recursion Rekursion läßt sich durch eine einfache Maßnahme herstellen: auf der rechten Seite eines Prefixoperators wird anstelle weiterer Events, oder eines anderen Prozesses, der Bezeichner des Prozesses angegeben, dessen Kommunikation gerade beschrieben wird.

```
P = a -> b -> P
```

Das Ergebnis des obigen Beispiels läßt sich auch für gegenseitige Rekursion erzeugen:

```
P = a -> Q
Q = b -> P
```

Guarded Alternative Dieser Operator wählt abhängig von der Umgebung des Prozesses eine von mehreren Möglichkeiten aus.

```
P = a -> STOP
Q = b -> (a -> STOP | c -> STOP)
```

Die Umgebung von Prozeß *Q* (Prozeß *P*) bietet ihm den Event *a* an. Da *P* und *Q* synchronisiert sind, wartet *P* bis alle beteiligten Prozesse *a* erlauben.

External Choice siehe *Guarded Alternative*.

```
P = a -> STOP
Q = b -> (a -> STOP [] c -> STOP)
```

Q hat nach *b* die Wahl zwischen $a \rightarrow STOP$ und $c \rightarrow STOP$. Da *P* den Event *a* anbietet, wird sich *Q* wie der Prozeß $a \rightarrow STOP$ verhalten. Dieses Notation ist äquivalent zu der oben angegebenen, ist aber gebräuchlicher.

Internal Choice oder *Nondeterministic Choice*.

```
P = a -> STOP
Q = b -> (a -> STOP |~| c -> STOP)
```

Im Gegensatz zum *External Choice*-Operator wird die Wahl nicht von der Umwelt beeinflusst, sondern unterliegt dem Zufall; sie bleibt innerhalb des Prozesses *Q* verborgen, daher die Bezeichnung *internal*, und ist nicht weiter beeinflussbar.

4.2.2 Paralleloperatoren

Viel wichtiger ist die Synchronisation mehrerer Prozesse mit Hilfe dieser *Events*. Diese Synchronisation führt dazu, daß ein *Event* nur dann auftritt, wenn alle beteiligten Prozesse bereit sind, dieses Ereignis geschehen zu lassen.

Synchronous parallel `P || Q`

Dies ist die einfachste Methode, um Parallelität zwischen Prozessen herzustellen. Es setzt voraus, daß alle beteiligten Prozesse immer ein auftretendes Ereignis geschehen lassen.

Alphabetized parallel Im Normalfall muß nicht jeder Prozeß einem Event eines parallelen Prozesses zustimmen. Bei steigender Anzahl von beteiligten Prozessen ist es schwer, diese so zu gestalten, daß sie immer jeder Kommunikation zustimmen, und dieses Verhalten ist nicht immer gewünscht.
In dieser Form der Parallelität wird für die beteiligten Prozesse je ein Alphabet angeben, welches eine Teilemenge des Alphabets des jeweiligen Prozesses ist. Diese Alphabete sind die Events, über die mit anderen Prozessen kommuniziert wird. Das heißt, alle beteiligten Prozesse müssen die Events in der Schnittmenge der Alphabete geschehen lassen, sonst blockieren sie. Beispiel:

$$(a \rightarrow b \rightarrow b \rightarrow STOP)_{\{a,b\}} \,||_{\{b,c\}}\, (b \rightarrow c \rightarrow b- > STOP)$$

Der erste erscheinende Event ist *a* des linken Prozesses. *b* ist solange blockiert, wie kein anderer beteiligter Prozeß *b* geschehen läßt. Der linke Prozeß erreicht dann *b* und nun kann auch der rechte Prozeß diesen Event geschehen lassen.
Danach will der linke Prozeß *a* kommunizieren, kann das aber noch nicht, weil der rechte zuerst *c* geschehen läßt. Danach erreicht der Prozeß das nächste *b* und somit tritt dieser Event auch beim linken Prozeß auf. Beide Prozesse beenden sich. Die Event-Abfolge ist also:

```
a -> b -> c -> b -> STOP
```

Interface parallel Während bei *Alphabetized parallel* für die beteiligten Prozesse je ein eigenes Alphabet der zu synchronisierenden Prozesse angegeben wird, wird hier lediglich die Schnittmenge selbst angegeben.

```
(a -> b -> b -> STOP) {|b|} (b -> c -> b -> STOP)
```

Davon abgesehen ist diese Version vergleichbar mit *Alphabetized parallel*. Es ist die in *FDR*[2] am häufigsten genutzte Variante.

Interleaving `P ||| Q`

Im Gegensatz zur bisherigen Parallelität steht das *Interleaving*; die zwei in Beziehung gesetzten Prozesse laufen völlig unabhängig voneinander ab.
Sollte zufällig in P und Q der gleiche Event auftauchen, so wird nicht-deterministisch festgelegt, welcher Prozeß diesen Event erzeugt. Der fragliche Event taucht aber nur **ein** Mal auf.

4.2.3 weitere Operatoren

Hiding operator Es gibt Fälle, bei denen man bestimmte Events vor der Umgebung verstecken möchte.
Zum Beispiel möchte man eine Spezifikation, in der das optimale Verhalten eines Systems so beschrieben ist, daß es eine feste, korrekte Eventabfolge gibt, gegen eine Implementierung testen. Dafür fügt man die Events der Implementierung hinzu, und *versteckt* die anderen, für diesen Zweck unwichtigen Events.

Ein Testtool wie z.B. *FDR* kann dann anhand des Vergleichs der Eventabfolgen der Spezifikation und der Implementierung, da hier nur die nicht-versteckten beobachtbar sind, feststellen, ob sich die Implementierung der Spezifikation gegenüber korrekt verhält.

4.3 Termination

In CSP stellt sich Termination dadurch dar, daß ein Prozeß keine Kommunikation mehr aufweist, d.h. keine Events mehr erzeugt.

Im vorherigen Teil wurden die Prozesse $STOP$, div und $SKIP$ genannt, die die Prozesse sind, welche nicht mehr kommunizieren. $STOP$ ist als eine *deadlock*-Situation zu betrachten, in der der Prozeß keine Aktivität mehr zeigt, und div ist im Gegensatz dazu die *lifelock*-Situation, wo der Prozeß unendlich viele Schritte durchführt, aber kein Ergebnis absehbar ist. Diese beiden Prozesse zeigen eine nicht-erfolgreiche Termination eines Prozesses an. $SKIP$ ist dagegen ein Prozeß der eben dieses durch das Auftauchen von $\surd$ macht. Bei Systemen mit Parallelkomposition terminiert das System erst, wenn auch der letzte beteiligte Prozeß terminiert, und zwar zu genau diesem Zeitpunkt.

Das System $P_X \;||_Y\; Q$ terminiert erst, wenn sowohl P, als auch Q erfolgreich terminieren. Kann einer der Prozesse nicht erfolgreich terminieren, sondern z.B. durch $STOP$, oder div nicht mehr kommunizieren, terminiert das ganze System, das mit dem fraglichen Prozeß in Beziehung steht, nicht. Dieses Verhalten wird als *verteilte Termination (distributed termination)* bezeichnet.

[2] siehe dort

5. Einführung in FDR

5.1 FDR und Refinement

FDR (Failures-Divergence Refinement) ist ein Tool, das erlaubt auf Basis von CSP durch Verfeinerung des Systems, bestimmte Eigenschaften dieses Systems zu beweisen. Mit dem Tool ist es auch möglich, das System auf Dead- und Livelocks zu prüfen und auch zu testen, ob es deterministisch ist. *FDR* unterstützt drei verschiedene Verfeinerungsmodelle:

- Traces Refinement
- Failures Refinement
- Failures-Divergences Refinement

5.1.1 Traces Refinement

Ein Trace eines Prozesses ist eine Folge von Events, die der Prozeß durchführen kann ($traces(P)$). Der Prozeß Q ist eine Verfeinerung von Prozeß P im Trace Model, wenn alle möglichen Traces von Q auch Traces von P sind. Wenn wir P als Spezifikation betrachten, welche sichere Zustände von unserem System beschreibt, dann ist Q die sichere Implementierung im Sinne von Verfeinerung im Trace Modell. Die Implementierung darf nichts tun, was in der Spezifikation nicht erlaubt ist. Der Prozeß $STOP$, der keine Events ausführt, ist damit die Verfeinerung von jedem Prozeß im Trace Model.

5.1.2 Failures Refinement

Failure von einem Prozeß P ist ein Paar (s, X), wobei s ein Trace von P ist und X die Menge von Events ($Refusals$), die nach der Ausführung von s verweigert werden. $failures(P)$ ist die Menge aller Failures von Prozeß P. Der Prozeß Q ist eine Verfeinerung von Prozeß P im Failures Model, wenn alle möglichen Failures von Q auch Failures von P sind. Die Implementierung darf nur die Events verweigern, die die Spezifikation verweigert. Ein Prozeß befindet sich im Deadlock-Zustand, wenn er alle Events verweigert. Das einfachste Beispiel für einen solchen Prozeß ist $STOP$. Damit ist er keine Verfeinerung von jedem Prozeß (außer sich selbst) im Failures Model.

5.1.3 Failures-Divergences Refinement

Divergences von einem Prozeß P ist eine Menge von Traces P, nach deren Ausführung P in den Lifelock-Zustand gerät. Ein Prozeß befindet sich im Lifelock-Zustand, wenn er eine unendliche Folge interner und damit keine nach außen sichtbaren Events ausführt. Der Prozeß Q ist eine Verfeinerung von Prozeß P im Failures-Divergences Model, wenn alle möglichen Failures von Q auch Failures von P sind und alle möglichen Divergences von Q auch Divergences von P sind. Nach der Divergence verhält sich der Prozeß chaotisch, kann also alle Events ausführen oder auch verweigern.

Zahlen

10	ganze Zahl
$m + n$, $m - n$	Summe und Differenz
$-m$	unäres Minus
$m * n$	Multiplikation
m/n, $m\%n$	Division und Modulo

Sequenzen

$<>$,$< 1, 2, 3 >$	Sequenzenbezeichnung
$< m..n >$	Geschlossene Sequenz von m bis n
$< m.. >$	Offene Sequenz aufwärts m
$s\hat{}t$	Sequenzverkettung
$\#s$, $length(s)$	Die Länge der Sequenz
$null(s)$	Test, ob die Sequenz s leer ist
$head(s)$	Das erste Element der Sequenz s
$tail(s)$	Teilsequenz von s, außer dem ersten Element
$elem(x, s)$	Test, ob x Element der Seuenz s ist

5.2 Syntax

5.2.1 Identifikatoren

Identifikatoren in *FDR-Skripten* müssen mit einem Buchstaben beginnen, danach kann eine beliebige Anzahl von alphanumerischen Zeichen oder Unterstrichen folgen. Für ein besseres Verständnis von Skripten sind folgende Regelen zu beachten:

- Prozesse in großen Buchstaben
- Typen und Typkonstruktoren beginnend mit großem Buchstaben
- Funktionen oder Channels in kleinen Buchstaben
- Vollständige Namen verwenden($COPY$ statt C)

Gleitkommazahlen werden nicht direkt unterstützt.

Mengen

$\{\}$	Mengenbezeichnung
$\{m..n\}$	Geschlossene Menge von m bis n
$\{m..\}$	Offene Menge aufwärts von m
$union(a1, a2)$	Vereinigung von Mengen
$inter(a1, a2)$	Schnitt von Mengen
$diff(a1, a2)$	Differenz von Mengen
$member(x, a)$	Test, ob x Element der Menge a ist
$card(a)$	Kardinalität von der Menge a
$empty(a)$	Test, ob die Menge a leer ist
$set(s)$	Konvertiert Sequenz s zu eine Menge
$Set(a)$	Die Menge aller Teilmengen der Menge a
$Seq(a)$	Die Menge aller Sequenzen über die Menge a

Boolesche Ausdrücke

$true, false$	boolesche Konstanten
$b1\ and\ b2$	und
$b1\ or\ b2$	oder
$not\ b$	nicht
$x1 == x2$, $x1! = x2$	gleich, ungleich
$x1 < x2$, $x2 > x1$, $x1 <= x2$, $x1 >= x2$	Ordnung Operatoren
$if\ b\ then\ x1\ else\ x2$	bedingte Anweisung

Tupel

$(1, 2)$, $(4, <>, \{7\})$	Paar und Tripel

Funktionen (Beispiele)

$f(x, y) = x + y$	Liefert $x + y$ als Ergebnis
$reverse(s) = if\ null(s)\ then\ <>\ else\ reverse(tail(s))\hat{}< head(s) >$	Liefert die reverse Sequenz zu s

Einfache Typen

$\{0..3\}$	Integer Datentyp
$\{true, false\}$	Boolesche Datentyp

Namenstypen

$nametype\ Values = \{0..199\}$	Namenstyp Values
$nametype\ Ranges = Values.Values$	Tupel aus Values

Datentypen

$datatype\ SimpleColor = Red \mid Green \mid Blue$	Aufzählbarer Datentyp SimpleColor

Channels

$channel\ flip, flop$	Einfache Channels
$channel\ c, d : \{0..3\}$	Komplexe Channels. $c.1$, $c!2$, $d?3$ sind Beispielevents
$\{\mid c \mid\}$	Alle Events auf dem Channel c

Prozesse

$STOP$	Prozeß, der keine Events ausführt
$SKIP$	Prozeß, der erfolgreich terminiert
$CHAOS(a)$	Chaos-Prozeß auf der Eventsmenge a
$c- > P$	Einfaches Prefix
$P;\ Q$	Sequentielle Komposition von P und Q
$P/\backslash Q$	Interrupt
$P\backslash Q$	Hiding
$P[\,]Q$	External Choice
$P \mid \sim \mid Q$	Internal Choice
$b\&P$	Boolean Guard
$P \mid\mid\mid Q$	Interleaving(Parallele Ausführung nicht synchronisiert)
$P[\{\mid a \mid\}]Q$	Sharing (Parallele Ausführung synchronisiert über Events aus a)

Refinement

$assert\ SPEC[m = SYSTEM$	$SPEC$, $SYSTEM$-Prozesse, m = Refinement-Modell(T, F oder FD)
$assert\ P[deterministic[FD]]$	Test, ob der Prozeß P deterministisch ist
$assert\ P[deadlock\ free[F]]$	Test, ob der Prozeß P deadlock frei ist
$assert\ P[divergence\ free[D]]$	Test, ob der Prozeß P lifelock frei ist

6. Validation, Verifikation und Test

Um zu überprüfen, ob sich ein Hard- oder Softwaresystem gemäß der spezifizierten Eigenschaften verhält, also die spezifizierte Leistung erbringt, bietet sich eine (automatisierte) Überprüfung des konkreten Systems gegen die spezifizierten Eigenschaften an.

Ebenso sinnvoll ist es, bereits vor der eigentlichen (eventuell kostspieligen) Konstruktion des Systems eine Überprüfung der Systemspezifikation gegen die Anforderungsdefinition vorzunehmen. Für diese Anwendungen eignen sich Validations-, Verifikations- und Testmethoden. Die Begrifflichkeiten sollen im folgenden erläutert werden.

6.1 Validation

Mit Validation bezeichnet man den Prozeß der Überprüfung der geplanten Systemanforderungen (engl. system requirements), ob sie den wirklich erforderlichen Anforderungen entsprechen. Dabei liegt es in der Natur der Sache, daß die Validation ohne ein wirklich greifbares Referenzobjekt auskommen muß, da das zu überprüfende Dokument ja eine *vorläufige* Zusammenfassung gerade dieser gewünschten Eigenschaften darstellt. Es ist aber möglich, die Systemanforderungen (auch formal) zu validieren. So kann sichergestellt werden, daß die Systemanforderungen in sich konsistent sind, also beispielsweise Widerspruchsfreiheit vorliegt und die Anforderungen eindeutig formuliert sind.

6.2 Verifikation

Von einer Verifikation spricht man, wenn eine — wie auch immer geartete — Korrektheitsuntersuchung eines konkreten Systems **gegen ein vorhandenes Referenzobjekt** vorgenommen wird. Das Referenzobjekt stellt dabei im Idealfall die Systemspezifikation dar, die aus einer Anforderungsanalyse hervorgegangen ist.

Diese Verifikation kann formal geschehen, indem eine formale Beschreibung des konkreten Systems gegen eine formale Spezifikation des Systems mit mathematischen Methoden verifiziert wird. Ebenso ist eine Modellprüfung (engl. model check) des konkreten Systems auf Basis der Systemspezifikation möglich, bei der das von der Spezifikation vorgesehene Systemverhalten (modelliert durch parallele Zustandsmaschinen) in jedem Zustand des ausgefalteten Zustandsraums gegen eine oder mehrere Korrektheitsbedingungen abgeglichen wird.

6.3 Test

Der allgemeine Begriff des Tests bezeichnet in diesem Kontext eine überwachte Ansteuerung des konkreten Systems mit Testdaten als Eingabe und der Aufzeichnung der Ausgaben oder internen Zustände des Systems währenddessen.

Die Interpretation der so gewonnenen Daten kann Aufschluß darüber geben, inwieweit sich das System gemäß der Spezifikation verhalten hat. Dazu werden die Eingabedaten aus der Spezifikation abgeleitet und Ausgaben oder beobachtbare interne Zustände des Systems mit den in der Spezifikation festgelegten verglichen. Diese Interpretation kann als eine Methode der Verifikation aufgefaßt werden.

6.4 Hardware vs. Software

Bei den oben genannten Definitionen ist neutral von „System“ die Rede, in der konkreten Anwendbarkeit der Methoden ist aber eine Unterscheidung von Hardware und Software notwendig.

Hardware ist im Gegensatz zu Software einem physikalischen Alterungsprozeß ausgesetzt, der nur bedingt zu Überprüfungszwecken beschleunigt werden kann. Dies hat zur Folge, daß das altersabhängige Verhalten von Hardware mit empirischen Methoden vorausgesagt werden muß. Für die Hardwareverifikation muß also ein zeitlicher Faktor in die Spezifikation einfließen und in den Anforderungen berücksichtigt werden. Dies macht die Verifikation von Hardware aufwendiger und vor allem teurer.

Software ist dagegen im Idealfall aus Anforderungsdefinitionen und Spezifikationen abgeleitet worden. Dieser direkte Zusammenhang macht eine formale Verifikation des Softwaresystems einfacher.

6.5 Automatisierung

Die Methoden der Validation, Verifikation und der Erstellung und Auswertung von Tests sind ursprünglich manueller Natur. Es existieren allerdings einige Verfahren, die dem Anwender Arbeit abnehmen oder gar neue Möglichkeiten bieten.

6.6 Das Testwerkzeug RT-Tester

Die Teilgruppen FTS und ACL führten einen automatisierten Test mit dem Testwerkzeug RT-Tester durch. Daher wird an dieser Stelle die Arbeitsweise von RT-Tester anhand dessen Software-Architur erläutert. Abbildung 6.1 zeigt diese Architektur.

Auf oberster Ebene, dem Abstract Machine Layer befinden sich die Abstrakten Maschinen. Diese interpretieren die aus einer CSP-Spezifikation mit Hilfe des FDR-Tools erzeugten Transitionsgraphen. Dabei werden abstrakte Events von darunter liegenden Ebenen gelesen (`AM_INPUT` Events) und auch abstrakte Events an die darunter gelegene Ebene ausgegeben (`AM_OUTPUT` Events). Die Kommunikation zwischen AML und der Ebene darunter findet dabei über Shared Memory statt.

Darunter befindet sich der Communication Control Layer (CCL)Der Communication Control Layer besteht in der Regel aus zwei Prozessen: `rttsndccl` und `rttrcvccl`. `rttsndccl` empfängt `AM_OUTPUT` Events von dem abstrakten Maschinen und leitet sie entweder an eine andere abstrakte Maschine oder an das Interface Modul weiter. Im letzteren Fall werden daraus Eingabewerte des zu testenden System gebildet. Dies kann

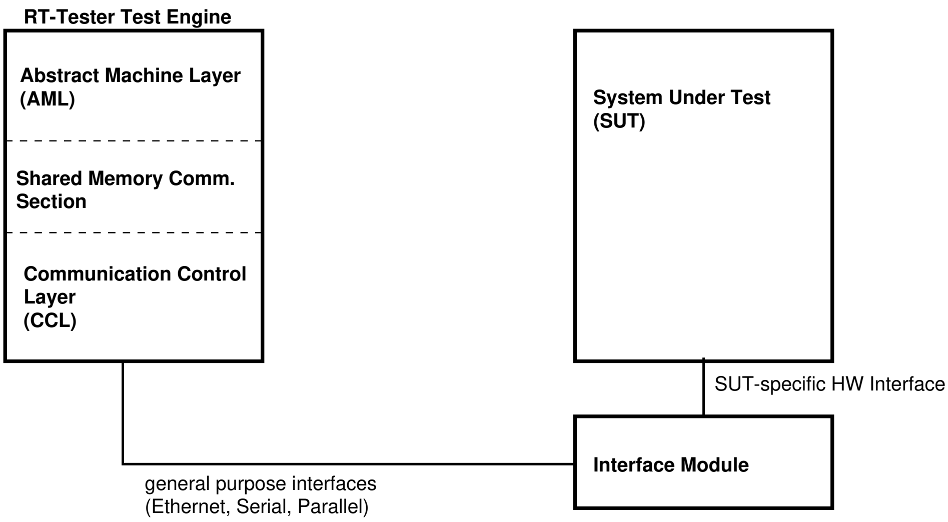


Abbildung 6.1: RT-Tester Architektur

eine einfache Wert-zu-Wert-Transformation sein aber auch eine komplexe Umwandlung in ein anderes Datenformat. Die Abstrakten Events, mit denen die Abstrakten Maschinen arbeiten, sind Integer-Werte, wobei jeder Wert einem CSP-Event entspricht. Die Zuordnung von CSP-Event zu Zahlenwert und Eventtyp ist in sogenannten Alphabet-Dateien festgehalten, die direkt aus der CSP-Spezifikation erzeugt werden. `rttrcvccl` empfängt die Ausgaben des Interface-Modules und wandelt diese in Abstrakte `AM_INPUT` Events um. Diese werden dann an die zuständige Abstrakte Maschine weitergeleitet.

Das Communication Control Layer kommuniziert über eine Socket-Verbindung mit dem Interface Module (IFM) Dieses dient dazu, das konkrete System Under Test (SUT) anzusteuern. Es empfängt die Ausgaben des CCL und übergibt sie dem System under Test als Eingaben. Andererseits sendet es die Ausgaben des SUT an das CCL weiter.

7. Spezifikationssprache Z

Z ist eine Spezifikationssprache, die auf der Mengentheorie und der Prädikatenlogik erster Stufe aufbaut. Sie wurde seit dem Ende der 70er Jahre von der Programming Research Group an der Oxford University entwickelt.

Ein wichtiges Merkmal von Z sind die Schemata, mit denen eine Spezifikation in kleinere Teile zerlegt werden kann. Die Schemata enthalten sowohl statische Aspekte der Spezifikation, wie die Zustände des Schemas, als auch die dynamischen Aspekte, wie die möglichen Operationen oder die möglichen Zustandswechsel.

Zu einer formalen Spezifikation gehört immer auch informeller Text, um zu erläutern, was man spezifiziert[1]. Da Z im Gegensatz beispielsweise zu CSP kein Konstrukt für Kommentare bietet, werden diese Erläuterungen außerhalb der Schemata geschrieben.

7.1 Beispiel: Birthdaybook

Am besten lassen sich die Grundzüge von Z an einem anschaulichen Beispiel erläutern. Hierfür nehmen wir einen die Spezifikation eines Geburtstagskalenders. Dieses Beispiel ist aus [Spi92] entnommen.

Es gibt dabei verschiedene Leute, deren Geburtstag wir kennen, und es gibt eine Zuordnung zwischen den Personen und den Geburtstagen. Die Personen werden hierbei repräsentiert durch ihre Namen.

Da Z eine streng getypte Sprache ist, müssen wir die Basisdatentypen definieren:

$$[NAME, DATE]$$

Wie man hier deutlich sieht, ist durch die Definition der Typen noch keine Aussage über das konkrete Aussehen oder die spätere Implementierung gemacht worden. So ist nicht festgelegt, daß beispielsweise $NAME$ ein String, bestehend aus einzelnen Zeichen sein soll, oder $DATE$ die Anzahl der Sekunden seit dem 1.1.1970 zählt, wie in den meisten UNIX-Systemen[2].

Gerade aus diesem Grund ist erläuternder Text wichtig, da nur aus ihm hervorgeht, was denn alles mit diesen Typen gespeichert werden soll. So soll in diesem Falle mit $NAME$ die Namen der Personen gespeichert werden, und mit $DATE$ das Geburtsdatum.

Der Zustandsraum unseres Geburtstagskalenders sieht folgendermaßen aus:

[1] Guter Quelltext ist zwar selbsterklärend, aber Erläuterungern können nie schaden
[2] Was für einen Geburtstagskalender allerdings eher ungünstig wäre…

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{BirthdayBook} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{known} : \mathbb{P}\, \mathit{NAME} \\
\mathit{birthday} : \mathit{NAME} \nrightarrow \mathit{DATE} \\
\hline
\mathit{known} = \mathrm{dom}\, \mathit{birthday} \\
\hline
\end{array}
$$

Er besteht im Wesentlichen aus einer Menge von Namen, deren Geburtstag ich weiß, und der Zuordnung von Namen und Geburtstagen. known ist dabei eine Teilmenge von NAME, bzw. ist ein Element aus der Potenzmenge über NAME. $\mathit{birthday}$ ist eine partielle Funktion, die einem Namen aus NAME ein Datum aus DATE zuordnet.

Im unteren Teil des Schemas steht eine Konsistenzbedingung. Sie besagt, daß im Domain, das heißt „auf der linken Seite", der Funktion $\mathit{birthday}$ nur Werte stehen dürfen, die in known vorhanden sind, oder anschaulicher, daß ich nur eine Zuordnung vom Namen auf den Geburtstag machen kann, wenn mir der Geburtstag überhaupt bekannt ist.

Das klingt so selbstverständlich, daß man meinen könnte, diese Bedingung bräuchte nicht aufgeschrieben werden. Man muß allerdings beachten, daß bei diesen Überlegungen schon die Bedeutung von known und $\mathit{birthday}$ berücksichtigt wurden. In der formalen Spezifikation sind dies allerdings nur abstrakte Mengen bzw. Funktionen, und deren Verhältnis zueinander muß explizit aufgeführt werden.

Ein zulässiger, konsistenter Zustand könnte jetzt beispielsweise so aussehen:

$$
\begin{aligned}
\mathit{known} &= \{\, \mathrm{John}, \mathrm{Mike}, \mathrm{Susan} \,\} \\
\mathit{birthday} &= \{\, \begin{array}[t]{l@{\ }l} \text{John} & \mapsto \text{25–Mar,} \\ \text{Mike} & \mapsto \text{20–Dec,} \\ \text{Susan} & \mapsto \text{20–Dec} \,\} \end{array}
\end{aligned}
$$

Ein Geburtstagskalender wäre ziemlich langweilig, wenn er nur aus einem Zustandsraum bestehen würde, und keine Zugriffsmöglichkeiten bieten würde. Daher kann man sich jetzt eine Operation $\mathit{AddBirthday}$ definieren:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{AddBirthday} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{BirthdayBook} \\
\mathit{name?} : \mathit{NAME} \\
\mathit{date?} : \mathit{DATE} \\
\hline
\mathit{name?} \notin \mathit{known} \\
\mathit{birthday}' = \mathit{birthday} \cup \{\mathit{name?} \mapsto \mathit{date?}\} \\
\hline
\end{array}
$$

Dieses Schema soll es uns ermöglichen, einen neuen Geburtstag in unseren Kalender aufzunehmen. Dazu muß sich der Zustand von $\mathit{BirthdayBook}$ verändern. Dies wird durch die erste Zeile im Schema erreicht, in der durch den Operator Δ das Schema $\mathit{BirthdayBook}$ eingebunden wird. Damit werden die Variablen known, known', $\mathit{birthday}$, und $\mathit{birthday}'$ eingeführt. known und $\mathit{birthday}$ kennzeichnen den Zustand des Geburtstagskalenders vor der Operation $\mathit{AddBirthday}$, und known' und $\mathit{birthday}'$ den Zustand danach.

$name?$ und $date?$ sind zwei Eingaben in diese Operation. In einer Programmiersprache wie zum Beispiel C würde man sie als Argumente der Funktion bezeichnen.

In der ersten Zeile des zweiten Teils des Schemas sehen wir eine Vorbedingung des Schemas. Damit wir einen Namen und einen Geburtstag hinzufügen können, dürfen sie vorher noch nicht bekannt (bzw. der Name Element von $known$) sein.

In der letzten Zeile wird dann der Nachzustand des Schemas beschrieben. Dadurch gibt es im Nachzustand eine neue Zuordnung von $name?$ nach $date?$. Dabei macht man sich zunutze, daß man eine Funktion als eine Menge von geordneten Paaren darstellen kann, bei denen allerdings die linken Seiten verschieden sein müssen.

Implizit wird in diesem Schema allerdings noch eine weitere Gleichung $known' = known \cup name?$ definiert, wie man leicht zeigen kann[3]:

$$\begin{aligned}
known' &= \operatorname{dom} birthday' && [\text{Invariante aus } BirthdayBook'] \\
&= \operatorname{dom}(birthday \cup \{name? \mapsto date?\}) && [\text{Spez. von } AddBirthday] \\
&= \operatorname{dom} birthday \cup \operatorname{dom}\{name? \mapsto date?\} && [\text{Definition von „dom“}] \\
&= \operatorname{dom} birthday \cup \{name?\} && [\text{Definition von „dom“}] \\
&= known \cup \{name?\} && [\text{Invariante aus } BirthdayBook]
\end{aligned}$$

Allerdings wird es häufig bevorzugt, wenn man solche impliziten Definitionen im Nachzustand vermeidet, und sie statt dessen explizit aufschreibt. Auch dann muß die Konsistenz noch bewiesen werden, die Spezifikation wird dadurch jedoch wesentlich besser zu durchschauen, da man die ganzen Effekte eines Schemas gleich im Auge hat, und sie nicht erst aus anderen Schemata herleiten muß.

Häufig hat man Operationen, die den Zustandsraum nicht verändern. In unserem Beispiel wäre das etwa das Suchen eines Geburtstages anhand des Namens. Man könnte, wie in der Operation $AddBirthday$, $\Delta BirthdayBook$ einbinden, und in den unteren Teil des Schemas die zwei Gleichungen $known' = known$ und $birthday' = birthday$ schreiben. Dies ist jedoch nicht praktikabel. Statt dessen verwendet man eine neue Notation, es wird $\Xi BirthdayBook$ eingebunden, um zu zeigen, daß der Zustandsraum von $BirthdayBook$ nicht verändert wird:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindBirthday}} \\ \hline
\Xi BirthdayBook \\
name? : NAME \\
date! : DATE \\ \hline
name? \in known \\
date! = birthday(name?) \\ \hline
\end{array}$$

In diesem Schema gibt es eine weitere, neue Notation: das Ausrufungszeichen am Ende von $date!$ zeigt, daß dies eine Ausgabe des Schemas ist. Im unteren Teil wird dann der Wert dieser Ausgabe $date!$ festgelegt. Die Vorbedingung ist natürlich, daß der Geburtstag dieser Person überhaupt bekannt ist.

[3]und schon haben wir ein Beispiel für einen Konsistenzbeweis

Um die Spezifikation vollständig zu machen, muß noch der Initialzustand angegeben werden:

$$
\begin{array}{|l}
\hline
\textit{InitBirthdayBook} \\
\textit{BirthdayBook}' \\
\hline
\textit{known}' = \varnothing \\
\textit{birthday}' = \varnothing \\
\hline
\end{array}
$$

Da man hier keinen Vorzustand hat, braucht er auch nicht eingebunden zu werden. Statt dessen wird nur der Nachzustand $BirthdayBook'$ eingebunden[4]

In unserem Anfangszustand gibt es also noch keine Personen, deren Geburtstag wir kennen, und infolgedessen gibt es auch keine Zuordnung von Personen zu Geburtstagen. Die zweite Gleichung hätte man auch weglassen können, weil sie aus der ersten folgt. Wie oben aber schon erläutert, hat man so alle Effekte, die durch dieses Schema hervorgerufen werden, auf einen Blick.

7.2 Robustheit

Bis jetzt haben wir bei unserem Geburtstagskalender nur den „normalen" Fall spezifiziert, in dem keine Fehler auftreten. Was allerdings passiert, wenn beim Eintragen des eines Geburtstages der Name schon vorhanden ist, oder wenn beim Suchen der Name unbekannt ist, ist undefiniert. Im besten Fall würde eine Implementierung mit einer Fehlermeldung reagieren, im ungünstigeren Fall allerdings die Daten korrumpieren.[5]

Aus diesem Grund sollten wir in unserer Spezifikation auch festlegen, was in solchen Fehlerfällen passieren soll. Dazu brauchen wir zunächst einmal eine Rückmeldung, ob unsere Operation erfolgreich war, oder nicht:

$$REPORT ::= ok \mid already\_known \mid not\_known$$

Weiterhin können wir ein Schema definieren, daß uns nur sagt, daß alles OK ist, aber nicht sagt, was sich am Zustandsraum geändert hat:

$$
\begin{array}{|l}
\hline
\textit{Success} \\
\textit{result}! : \textit{REPORT} \\
\hline
\textit{result}! = \textit{ok} \\
\hline
\end{array}
$$

Durch Schemakonjunktion erhalten wir eine Funktion, die einen Namen in unseren Geburtstagskalender einfügt, und einen Erfolg meldet:

$$AddBirthday \land Success$$

Weiterhin müssen wir jetzt noch den Fehlerfall behandeln. Das heißt, wir müssen definieren, was passieren soll, wenn der Name, den wir in den

[4] Es gibt noch eine andere Notation, die $BirthdayBook$ (also ohne (')) einbindet. Wir halten sie aber für ein wenig inkonsistent, da dies ansonsten für den Vorzustand verwendet wird.

[5] Diese Implementierung ist zwar nicht besonders schlau, aber immer noch korrekt nach unserer Spezifikation

Geburtstagskalender eingetragen werden soll, schon vorhanden ist. Wir könnten als erste Möglichkeit den alten Geburtstag mit dem neuen überschreiben, oder als zweite Möglichkeit nichts tun, und eine erfolgreiche Bearbeitung melden[6], oder als dritte Möglichkeit mit der Fehlermeldung $already\_known$ und ohne Änderungen am Zustandsraum zurückkehren. Die dritte Möglichkeit erscheint mir am sinnvollsten, und damit sieht unser Schema folgendermaßen aus:

$$
\begin{array}{|l}
\multicolumn{1}{l}{AlreadyKnown} \\
\hline
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \in known \\
result! = already\_known \\
\hline
\end{array}
$$

Zusammen mit $AddBirthday$ und $Success$ kann man mit diesem Schema für den Fehlerfall jetzt eine neue Operation $RAddBirthday$ (für **R**obustes $AddBirthday$) spezifizieren:

$$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \land Success) \lor AlreadyKnown.$$

Somit haben wir jetzt eine neue Operation zum Hinzufügen eines Geburtstages, bei der auch im Fehlerfall genau spezifiziert ist, wie sie sich verhalten soll. Insbesondere brauchten wir dazu unsere alte, nicht robuste Version nicht wegwerfen, sondern konnten auf ihr aufbauen.

Dasselbe müßte jetzt auch noch für $FindBirthday$ gemacht werden, worauf ich aber verzichten möchte, da das Vorgehen analog ist. Bei $InitBirthdayBook$ ist dies nicht nötig, da diese Operation nicht schiefgehen kann.

7.3 Von der Spezifikation zur Implementierung: Verfeinerung

Nun will man den Geburtstagskalender nicht nur spezifizieren, sondern auch eine Implementation davon haben. Dazu müssen zunächst einmal konkrete Datentypen für die abstrakten gefunden werden. In der Spezifikation haben wir Repräsentationen für die Daten gewählt, die möglichst aussagekräftig sind, ohne Blick darauf, ob sie gut im Computer, oder in einer Programmiersprache darstellbar sind. In der Implementation müssen wir jetzt aber auch darauf achten. Für eine Implementierung in Pascal könnte man man den Geburtstagskalender mit zwei Arrays repräsentieren[7]:

```
names : array[1..] of NAME;
dates : array[1..] of DATE;
```

Dabei sind $NAME$ und $DATE$ noch nicht weiter spezifiziert. Auch die Arraygrenzen wären noch festzulegen.

Diese Arrays lassen sich mathematisch als Abbildungen von den natürlichen Zahlen in die Mengen $NAME$ beziehungsweise $DATE$ modellieren:

[6] Das würde allerdings nicht viel Sinn machen

[7] Ich weiß, in Perl wäre es einfacher mit Hashes zu implementieren gewesen

$$\begin{array}{l} names : \mathbb{N}_1 \rightarrow NAME \\ dates : \mathbb{N}_1 \rightarrow DATE \end{array}$$

Somit ist das Element $names[i]$ des Arrays exakt dasselbe wie der Wert der Funktion $names(i)$, und die Zuweisung $names[i] := v$ läßt sich wiedergeben durch

$$names' = names \oplus \{i \mapsto v\}$$

Um noch zu zeigen, wieviel von den Arrays benutzt ist, wird noch eine zusätzliche Variable hwm (für High Water Mark) eingeführt. Damit sieht der konkrete Zustandsraum folgendermaßen aus:

$$\begin{array}{|l} \multicolumn{1}{l}{BirthdayBook1} \\ \hline names : \mathbb{N}_1 \rightarrow NAME \\ dates : \mathbb{N}_1 \rightarrow DATE \\ hwm : \mathbb{N} \\ \hline \forall\, i, j : 1 \mathinner{..} hwm \bullet \\ \qquad i \neq j \Rightarrow names(i) \neq names(j) \\ \hline \end{array}$$

Um jetzt zu zeigen, wie unsere abstrakte Spezifikation mit dieser neuen, konkreten Z–Spezifikation zusammenhängt, bildet man eine Abstraktionsrelation:

$$\begin{array}{|l} \multicolumn{1}{l}{Abs} \\ \hline BirthdayBook \\ BirthdayBook1 \\ \hline known = \{\, i : 1 \mathinner{..} hwm \bullet names(i) \,\} \\ \forall\, i : 1 \mathinner{..} hwm \bullet \\ \qquad birthday(names(i)) = dates(i) \\ \hline \end{array}$$

Hierbei wird festgelegt, daß in der Menge $known$ genau die Namen sein sollen, die im Array $names$ vom 1 bis hwm gespeichert sind, und daß die Geburtstage im Array $dates$ an genau dem Index gespeichert sind, an dem auch die korrespondierenden Namen im Array $names$ gespeichert sind.

Es ist noch festzuhalten, daß es zu einem abstrakten Zustand mehrere korrespondierende konkrete Zustände geben kann. Dies liegt zum Beispiel daran, daß die Reihenfolge der Namen nicht festgelegt ist. Andererseits gibt es zu jedem konkreten Zustand nur einen einzigen korrespondierenden abstrakten Zustand.

Nun können wir auch eine Operation $AddBirtday1$ schreiben, die eine konkrete Version von $AddBirtday1$ ist:

$$\begin{array}{|l} \multicolumn{1}{l}{AddBirthday1} \\ \hline \Delta BirthdayBook1 \\ name? : NAME \\ date? : DATE \\ \hline \forall\, i : 1 \mathinner{..} hwm \bullet name? \neq names(i) \\ hwm' = hwm + 1 \\ names' = names \oplus \{hwm' \mapsto name?\} \\ dates' = dates \oplus \{hwm' \mapsto date?\} \\ \hline \end{array}$$

Der Allquantor könnte in einer konkreten Pascal-Implementierung zum Beispiel mit einer `for`–Schleife realisiert werden, und die restlichen Anweisungen sind einfache Zuweisungen.

7.4 Verifikation der Äquivalenz

Für die Korrektheit dieser Verfeinerung sind zwei Bedingungen nachzuweisen:

- Wann immer $AddBirthday$ in der abstrakten Spezifikation durchgeführt werden kann, kann auch $AddBirthday1$ in der Implementation durchgeführt werden.
- Der Endzustand von $AddBirthday1$ repräsentiert einen Nachzustand von $AddBirthday$

Vereinfacht kann man also sagen: Was im konkreten Zustandsraum funktioniert, muß auch im abstrakten Zustandsraum genau so funktionieren. Dies kann man auch in der Abbildung 7.1 wiederfinden.

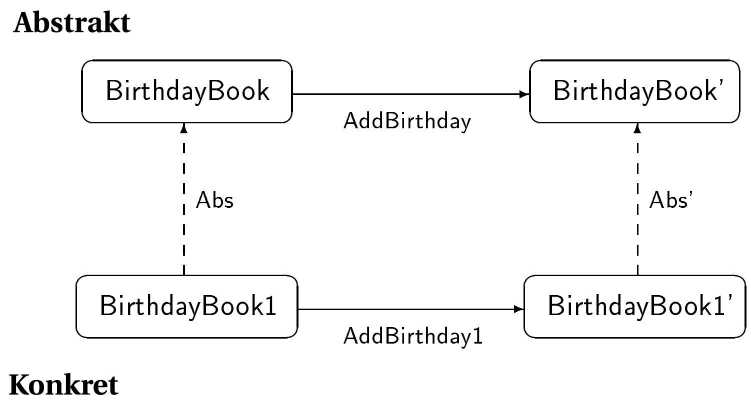


Abbildung 7.1: Verfeinerung einer Operation

Zur ersten Bedingung: $AddBirthday$ ist erlaubt genau dann, wenn $name? \notin known$ gilt. Falls dies gilt, dann können wir mit Hilfe des Prädikates

$$known = \{\, i : 1 \mathrel{..} hwm \bullet names(i) \,\}$$

aus der Abstraktionsrelation Abs folgern, daß $name?$ nicht in den Elementen $names[i]$ ist:

$$\forall\, i : 1 \mathrel{..} hwm \bullet name? \neq names(i).$$

Dies ist die Vorbedingung zu $AddBirthday1$

Um die zweite Bedingung nachzuweisen, müssen wir den konkreten Vorzustand und den konkreten Nachzustand von $AddBirtday1$ betrachten, und den abstrakten Vor– beziehungsweise Nachzustand, den sie nach der Abstraktionsrelation Abs repräsentieren. Es ist dabei zu zeigen, daß sich die abstrakten Zustände so verhalten, wie sie in $AddBirthday$ beschrieben sind:

$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Auch wenn man die Transformation auf der konkreten Seite vornimmt, bleiben die Domains[8] der abstrakten Repräsentation gleich:

$$\operatorname{dom} birthday' = known' \qquad \text{[Nachbedingung, aus } BirthdayBook\text{]}$$

[8]Wie wir schon wissen, sind das die linken Seiten von Funktionen

$$
\begin{aligned}
&= \{\, i : 1 \mathinner{\ldotp\ldotp} hwm' \bullet names'(i) \,\} && [\text{aus } Abs'] \\
&= \{\, i : 1 \mathinner{\ldotp\ldotp} hwm \bullet names'(i) \,\} \cup \{names'(hwm')\} \\
& && [hwm' = hwm + 1] \\
&= \{\, i : 1 \mathinner{\ldotp\ldotp} hwm \bullet names(i) \,\} \cup \{name?\} \\
& && [names' = names \oplus \{hwm' \mapsto name?\}] \\
&= known \cup \{name?\} && [\text{aus } Abs] \\
&= \operatorname{dom} birthday \cup \{name?\} && [\text{Vorbedingung}]
\end{aligned}
$$

Weiterhin wird der Teil des Arrays, der schon vorher benutzt wurde, nicht verändert, so daß gilt:

$$\forall\, i : 1 \mathinner{\ldotp\ldotp} hwm \bullet names'(i) = names(i) \land dates'(i) = dates(i)$$

Damit gilt für jedes i in diesem Bereich, daß

$$
\begin{aligned}
& birthday'(names'(i)) \\
&\quad = dates'(i) && [\text{aus } Abs'] \\
&\quad = dates(i) && [dates \text{ unverändert}] \\
&\quad = birthday(names(i)) && [\text{aus } Abs]
\end{aligned}
$$

und für den einen, neu hinzugekommenen Namen, der unter dem Index $hwm' = hwm + 1$ gespeichert ist:

$$
\begin{aligned}
& birthday'(name?) \\
&\quad = birthday'(names'(hwm')) && [names'(hwm') = name?] \\
&\quad = dates'(hwm') && [\text{aus } Abs'] \\
&\quad = date? && [\text{Spez. von } AddBirthday1]
\end{aligned}
$$

Aus den letzten beiden Gleichungen können wir jetzt folgern, daß $birtday'$ und $burthday \cup \{name? \mapsto date?\}$ gleich sind, selbst wenn die Zustandsänderung im konkreten Zustandsraum durch die Operation $AddBirthday1$ vorgenommen wurde.

8. ASpecT

ASpecT wurde anfänglich 1986 als ein Versuch, eine Teilmenge von Algebraischen Spezifikationen für abstrakte Datentypen zu implementieren, entwickelt. Auch parametrisierte Module wurden unterstützt. Von Anfang an wurde das System dazu entworfen, so benutzerfreundlich wie möglich zu sein, eingeschlossen Möglichkeiten zum Überladen und ein source-level debugger. Effizienz verlangte nach call-by-value Auswertung und Speicherverwaltung mit Referenzzählung.

Über die Jahre wurden mehr und mehr Erweiterungen hinzugefügt, z.B. Subsorten, Funktionale und eingeschränkter Polymorphismus.

Die Entwicklung der Sprache wurde eingestellt, als die Autoren, Dr. Richard Seifert und Dr. Jörn von Holten, die Universität Bremen verließen.

8.1 Module

Ein Modul (Spezifikation) besteht aus den importierten Modulen, Signaturen und Axiomen. Ein Modul läßt sich in globale Schnittstelle und lokale Implementation aufteilen. Parametrisierte Module haben zusätzlich einen formalen Parameter. Der Teil „GLOBAL“, der die globale Schnittstelle beschreibt, darf nicht leer sein. Die Signatur eines Moduls besteht aus Deklarationen von Datentypen (Sorten) und Funktionen (Operationen).

SPEC TEST =
GLOBAL
< *Globale Schnittstelle*
LOCAL
< *Lokale Implementation*
END.

8.2 Datentypen (Sorten)

Datentypen (Sorten) werden durch ihren Namen, gefolgt von den Funktionen zur Kreierung ihrer Daten, Konstruktoren, eingeführt. Anders als bei objektorientierten Sprachen sind Konstruktoren in funktionalen Sprachen mehr als nur Funktionen zur Initialisierung von Variablen, sondern erschaffen unterscheidbare Werte des Typs. Namen von Sorten und Konstruktoren beginnen mit einem Kleinbuchstaben. Die Konstruktor-Funktionen besitzen eine Typdeklaration.

Alle Sorten, formale, externe und im voraus deklarierte ausgenommen, müssen immer mindestens einen Konstruktor haben.

SORTS
nat ::= zero | succ nat.

Hierdurch wird die Sorte nat mit den zwei Konstruktoren zero :: nat

und

succ :: nat → nat

deklariert. (Die Deklarationen der Konstruktoren sind abgekürzt.)

Das bedeutet, daß jeder Term der Sorte nat entweder als

zero

(für 0) oder

succ(succ(...(zero)...))

(n mal succ für n) ausgedrückt werden kann.

Die vielfach gebrauchte Sorte "Liste von xx" kann als [xx] abgekürzt werden.

nats ::= [nat].

Dies erzeugt die Konstruktoren

[] :: nats

und

(:) :: nat → nats → nats

Der Infix-Konstruktor (:) ist eine Curry-Funktion, die zwei Argumente benötigt, um tatsächlich eine Liste zu erzeugen. Eine Liste mit den Elementen

x1:(x2:(x3:[]))

kann auch als

[x1,x2,x3]

geschrieben werden und

x1:(x2:xs)

kann auch als

[x1,x2|xs]

geschrieben werden.

Um strukturierte Typen abzukürzen können Typsynonyme eingeführt werden:

SORTS
relation ::= (data → datas,datas).

Hierbei ist relation synonym für ein Tupel aus dem Funktional

data → datas

und dem Datentyp

datas.

Sorten können auch durch Vereinigung bestehender Sorten (Subsorten) erzeugt werden:

SORTS
  int ::= (pos) | (neg).

Hierbei setzt sich die Sorte int aus den beiden Sorten pos und neg zusammen. Tatsächlich werden zwei unbenannte Konstruktoren (Umwandlungen) mit den Typen

SORTS
  ::= pos → int

oder auch

SORTS
  ::= neg → int

deklariert. Diese Umwandlungen sollten mit Vorsicht benutzt werden, da sie leicht Mehrdeutigkeiten beim Schreiben von Axiomen erzeugen können.

Es lassen sich zusätzlich Datenselektoren und -modifikatoren deklarieren, um die Komponenten eines Konstruktor-Terms zu extrahieren. Ein Daten-Selektor (Projektion) extrahiert eine Komponente aus einem Konstruktor-Term, ein Modifikator ändert eine Komponente eines Konstruktor-Terms.

Ein Selektor/Modifikator muß in jedem gegebenen Konstruktor deklariert werden.

SORTS
  chunk ::=
    small (hd :: data,datas)
  | big (hd :: data,data,datas)

Hierduch wird eine Sorte chunk mit den beiden Konstruktoren

small :: (data,datas) → chunk

und

big :: (data,data,datas) → chunk

deklariert sowie der Selektor

hd :: chunk → data

und der Modifikator

hd :: (chunk,data) → chunk

8.3 Funktionen

Deklaration von Funktionen bezeichnen abgeleitete Funktionen (Operationen), die auf Konstruktor-Termen arbeiten und einen Konstruktor-Term erzeugen.

Funktions-Namen beginnen mit einem Kleinbuchstaben. Ausgenommen sind Infix-Funktionen, wie das folgende Beispiel zeigt.

OPNS
  nat + nat :: nat.
  split :: nat → nats → (nats,nats).

Hierdurch werden zwei abgeleitete Funktionen

(+) :: nat → nat → nat

und split deklariert. (+) ist eine Infix-Funktion, für die Infixnotation verfügbar ist.

Es wird angenommen, daß später Axiome angegeben werden, die das Verhalten der Funktionen beschreiben.

Namen von Infix-Funktionen sind Folgen von beliebigen Sonderzeichen. Mit Sonderzeichen sind Zeichen, die keine Zahlen oder Buchstaben sind, gemeint. Ausgenommen sind die Zeichen `.,(){}#_"`. Darüberhinaus ist für jede Curry-Funktion die Infixnotation durch Voranstellung eines Unterstrichs „_" vor dem tatsächlichen Funktionsnamen verfügbar.

Zur Vereinfachung werden implizit Tupel-Selektoren zur Extraktion der Komponenten der von Funktionen berechneten Tupel-Ergebnissen deklariert.

Diese Selektoren sind entsprechend dem Namen der Sorte des Ergebnisses, das sie extrahieren, benannt.

Für den Fall der Mehrdeutigkeit dieser Selektoren können sie durch zusätzliche Nummern unterschieden werden. Im obigen Beispiel wären das die impliziten Selektoren

nats[.1] :: (nats,nats) → nats

und

nats.2 :: (nats,nats) → nats,

die die erste bzw. zweite Komponente des Tupels im Argument extrahieren.

Mehrere Funktionen mit identischem Typ können durch Kommata getrennt aufgezählt werden.

8.4 Axiome

Die Axiome beschreiben das Verhalten abgeleiteter Funktionen. Sie sind als Regeln für Ersetzungen der linken Seite durch die rechte Seite zu verstehen. Es gibt mehrere Einschränkungen, die nur für nicht operationelle Theoreme gelockert werden:

1. Die Axiome müssen nach Beseitigung von Typvariablen eindeutig typbar sein.
2. Nur Axiome für abgeleitete Funktionen können angegeben werden.
3. Die Argumente für die abgeleitete Funktion auf der linken Seite darf nur aus Variablen und Konstruktoren bestehen.
4. Die Axiome müssen links-linear sein. Das heißt: Keine Variable darf mehr als einmal auf der linken Seite eines Axioms vorkommen.
5. Alle Variablen der rechten Seite des Axioms müssen auch auf der linken Seite vorkommen.
6. Alle Variablen auf der linken Seite des Axioms müssen auch auf der rechten Seite vorkommen. Das heißt: Variablen, die nur auf der linken Seite vorkommen, müssen anonymisiert werden (als _ geschrieben werden).

7. Axiome für eine abgeleitete Funktion müssen in einer Gruppe geschrieben werden.
8. Die linken Seiten von Axiomen für eine abgeleitete Funktion müssen alle möglichen Fälle für Argumentterme abdecken.
9. Die linken Seite der Axiome für eine abgeleitete Funktion dürfen sich nicht überlappen. Es dürfen also keine Argumentterme existieren, für die mehr als ein Axiom anwendbar ist. Variablennamen beginnen immer mit einem Großbuchstaben.

```
EQNS
   X + zero = X
   X + succ Y = succ (X + Y)
```

Hierbei wird die Funktion (+) durch zwei Axiome beschreiben. Das erste besagt, wenn man einen beliebigen Term (bezeichnet durch die Variable X) zu zero addiert, so erhält man X. Das zweite Axiom besagt, wenn man einen beliebigen Term X zu einem anderen Term der Form succ Y addiert, so erhält man einen Term succ Z, wobei man Z durch Addition von X und Y erhält.

Es gibt keine eingebauten Vorrangsregeln für Operator in ASpecT, Infix-Terme werden von links nach rechts ausgewertet. Funktionsanwendung hat immer Vorrang vor Infix-Operatoren.

Um Mehrdeutigkeiten aufzulösen ist es möglich, Typen von Funktionen und Variablen zu deklarieren. Zusätzlich kann jeder Term mit seinem TODO:presumed Typ annotiert werden.

```
FORALL X :: nat.
EQNS
   X + (zero :: nat) = X.
```

Für den Bedingungs-Operator

```
if_then_else ::
   (boolean,data,data) → data
```

ist eine „Mixfix"-Notatation verfügbar. Die if-then-else Namensteile binden weniger stark als andere Funktionsanwendungen oder Infix-Operatoren.

```
EQNS
   abs X = if X >= zero then X else neg X.
```

Als Abkürzung gibt es eine case-Konstruktion für mehrere Bedingungen über die eingebaute Gleichheit.

```
EQNS
   g X = case X of
         0:    0
       | 1,2:  1
       | 3:    2
       else X.
```

Manchmal ist es bequemer, überlappende Axiome zu schreiben, und die 6. Einschränkung zu verletzen. Zu diesem Zwecke können Axiome als *Catch-All*-Axiome gekennzeichnet werden (durch ein vorangestelltes

„$“). Diese Catch-All-Axiome können sich mit normalen Axiomen überlappen und behandeln per Definiton nur Fälle, die nicht von normalen Axiomen abgedeckt werden.

Catch-All-Axiome dürfen sich nicht mit anderen Catch-All-Axiomen überlappen.

```
EQNS
  f zero = zero
  $f X = X
```

Teil II

GEN

Vorspann

Das Teilprojekt GEN setzt sich aus Klaus Lüttich und Harald Wagener zusammen. Als einziges Teilprojekt hat es keine systemspezifischen Ziele. Vielmehr ist es der Erstellung eines „Gerippes" für den Projektbericht verschrieben.

Eine weitere wichtige Aufgabe ist die Verwaltung der Bibliographie, die aufgrund der Möglichkeiten von BibTeX zentral gehalten werden kann. Dies hat insbesondere den Vorteil, daß die automatisch generierten Labels eindeutig bleiben.

Die Erstellung eines Projektberichtes muß auch in einem locker organisierten Projekt wie LiVE! gut vorbereitet sein.

9. Grundlagen

9.1 Section

9.1.1 Subsection

Für das einfache Verständnis dieses Teils des Projektberichts sollte der Leser mit LaTeX, BibTeX, CVS und EMACS vertraut sein. Weitere Voraussetzungen gibt es nicht.

10. Ziele

Uns war es insbesondere wichtig, das alle Gruppen selbst ihren Teil des Projektberichtes erstellen können, und dabei nicht die Arbeit an anderen Teilen stören.

Die bibliographischen Verweise sollten eindeutig sein, aber in einer zentralen Datenbasis gesammelt werden. Auf diese Art können in der Endphase redundante Einträge zusammengefaßt werden.

Der Bericht über GEN mag etwas künstlich erscheinen, aber er hat immerhin die Aufgabe, anderen Teilprojekten als Vorbild zu dienen. So können wir uns sicher sein, daß zumindest die anderen Projektmitglieder einmal lesen werden, was hier steht.

11. Optionen

Abgesehen von realitätsfernen Möglichkeiten war für uns die Wahl der Werkzeuge von vornherein klar. Die einzige Unsicherheit hatten wir im Umgang mit BibTeX, wwas aber eher damit zusammenhing, daß keiner von uns dieses Tool bisher benutzt hatte.

Letztendlich blieb uns hier noch die Wahl eines vernünftigen Styles für die Einträge im Literaturverzeichnis (siehe 12 auf Seite 57)

12. Entscheidungen

Das Grundgerüst des Projektberichtes sollte uns die Möglichkeit geben, einen Bericht zu erstellen, der nicht nur von der Verwaltung her einfach ist, neben dem Inhalt auch durch die Form besticht.

Auf eine Empfehlung hin haben wir eine Abwandlung des `refman`–Styles als Grundlage genommen, natürlich auf die deutsche Sprache angepaßt, inklusive des `zed-csp`–Styles, der regelmäßig Verwendung finden wird und deswegen von vornherein eingebunden wird.

Als Schriftart haben wir die Palatino gewählt, da diese eine willkommene Abwechslung zur althergebrachten ComputerModern ist und als Serifenschrift immer noch gut lesbar ist.

Bei unserer Suche nach deutschen BIBTEX–Styles hatten wir die Auswahl zwischen mehreren Angeboten, die wir der TEX–FAQ entnommen hatten. Im Sinne einer vernünftigen Arbeit erschien es uns sinnvoll, den `dinalpha`–Style zu nehmen, da dieser Referenzen nach deutscher Industrienorm ermöglicht, die im Gegensatz zu den internationalen Styles auch vorsieht, eine ISBN anzugeben. Zwei Beispiele für solche Eintrage sind [BBD 97] und [CAR94]

13. Ergebnisse

Mit CVS, BibTeX und LaTeX stehen technisch einwandfreie Werkzeuge zur Verfügung, die eine unkomplizierte verteilte Erzeugung größerer Dokumente ermöglichen. Dem Projekt kann eine Plattform angeboten werden, die eine allmählich fortschreitende Erstellung des Projektberichts erlaubt, die sich in den meisten Fällen begleitend zum letzten Projektsemester ergeben wird.

Wir haben somit die Möglichkeit, auch schon zum Projekttag einen Bericht vorzulegen, der Interessierten einen nachhaltigen Einblick in die Arbeit des Projektes ermöglicht.

Wir empfehlen ähnlich großen und auch größeren Gruppen, ein ähnlich konzipiertes System einzurichten.

14. To-Be-Dones

Mit dem Abschluß des Projektberichts ist dieser Teil des Projektberichts eigentlich nicht mehr notwendig. Zu Dokumentationszwecken bleibt er aber erhalten.

Teil III

ACL

15. Einleitung

15.1 Projektziel

Das herkömmliche UNIX Modell für die Zugriffskontrolle von Dateien erweist sich für manche Zwecke als unzureichend, weil die Unterteilung aller Benutzer in einen Besitzer der Datei, Angehörige der Gruppe, der die Datei gehört, und sonstige Benutzer nicht fein genug ist. In einer Organisation mit einer komplexen Struktur kann sich die Vergabe von Rechten auf diese Weise sehr schwierig gestalten. Eine eins-zu-eins Abbildung von "realen" Arbeitsgruppen auf "virtuelle" Systemgruppen ist nicht möglich, da es in der Realität z.B. oft vorkommt, daß zwei Gruppen mit verschiedenen Rechten auf Dokumente zugreifen und noch andere Zugriffsrechte für andere Gruppen gelten. Beispielweise könnte es sein, daß die Mitarbeiter eines Lagers eine Datei lesend und schreibend bearbeiten dürfen, um Lagerbewegungen verbuchen zu können. Die Personen des Einkaufs dürfen lesend zugreifen, um die Lagerbestände zu kontrollieren. Wenn nun alle weiteren Personen gar keine Rechte haben sollen ist diese Funktionalität mit Standard-UNIX-Rechten nicht mehr direkt zu erreichen. Weiter ist es nur schwer möglich einzelnen Benutzern Zugriffsrechte zu verweigern - man müßte dazu beispielsweise alle Benutzer, die nicht zugreifen dürfen, in die dateibesitzende Gruppe aufnehmen und die Zugriffsrechte dann nur allen übriggebliebenen gewähren. Die Rechtevergabe ist in solchen komplizierteren Fällen zu wenig transparent als daß sie von normalen Benutzern zuverlässig gehandhabt werden kann was wiederum ein Sicherheitsrisiko darstellt. Dazu kommt, daß das Gruppenmanagement nur durch den Systemverwalter durchgeführt werden kann - hier sei abgesehen von den Erweiterungen, die beispielsweise das NIS (Network Information System) und das Shadow Password System bietet.

Mit Hilfe von Access Control Lists (ACLs) können für verschiedene Benutzer und Gruppen Zugriffsrechte einzeln festgelegt werden, d.h. eine diskrete Zugriffskontrolle (engl. Discretionary Access Control(DAC)) auszuüben. Dies bietet die Möglichkeit die Systemgruppen sehr viel stärker an den realen Arbeitsgruppen auszurichten. Außerdem kann die Rechtevergabe viel intuitiver erfolgen, was das durch Fehlbedienung ausgehende Sicherheitsrisiko verringert. Vorteilhaft ist auch, daß die Rechtevergabe nicht mehr so abhängig vom Gruppenmanagement ist.

Mittlerweile werden ACLs von vielen UNIX Derivaten, z. B. HP-UX, Solaris und Digital UNIX, unterstützt. Linux hingegen bietet zu Projektbeginn noch keine ACL Unterstützung an. Ziel dieser Arbeitsgruppe ist daher die Implementierung von ACLs für Linux. Die Implementierung soll unterstützt von formalen Methoden durchgeführt werden, die Korrektheit des Ergebnisses formal geprüft werden.

15.2 Projektablauf

Das native Dateisystem von Linux ist das „Second Extended Filesystem", kurz EXT2. Zwar unterstützt Linux noch eine Vielzahl anderer Dateisysteme, doch verwendet man für gewöhnlich das EXT2. Daraus folgte, daß wir die ACL Unterstützung in das EXT2 einbauten. EXT2 ist der Nachfolger von EXT, dem „Extended Filesystem“, das sozusagen eine Erweiterung des ersten Dateisystems von Linux, dem „Minix“ von Andrew S. Tanenbaum, ist.

Wir entdeckten, daß die Implementierung von ACLs im EXT2-Dateisystem anscheinend bereits vorbereitet worden war. Daher verfolgten wir zunächst, ob an einer Linux Implementierung für ACL Unterstützung bereits gearbeitet wurde. Bei unserer Recherche entdeckten wir eine Implementierung[1] von Remy Card im Alpha-Stadium. Er ist der Maintainer des EXT2-Dateisystems, d. h. er ist zur Zeit für die Pflege und Entwicklung des EXT2 zuständig. Außerdem analysierten wir die Funktionsweise bestehender ACL Implementierungen von verschiedenen UNIX-Versionen.

Die Implementierung von Remy Card war in der vorliegenden Form übersetzbar, aber nicht funktionsfähig. Wir beschlossen, sie als Grundlage unserer Implementierung zu verwenden und uns auf die Prüfung der Korrektheit durch formale Verifikation und automatisierten Test zu konzentrieren. Da eine komplette formale Verifikation zu aufwendig gewesen wäre, führten wir diese nur beispielhaft an der Funktion zur Zugriffskontrolle durch. Dazu spezifizierten wir das gewünschte Verhalten formal in Z und führten damit eine formale Verifikation gemäß dem Kalkül von Vor- und Nachbedingungen durch, dieses Kalkül ist in [AO91] beschrieben.

Der Dokumentation der Implementierung von Remy Card entnahmen wir, daß die Implementierung den Anspruch hatte, konform zu den Standardentwürfen POSIX.1e [Sec97a] und POSIX.2c [Sec97b], Draft 13, zu sein. Da wir zu den Standardentwürfen anfangs noch keinen Zugang hatten, begannen wir, offensichtliche Fehler in der ACL Implementierung von Remy Card zu beseitigen. Parallel beschrieben wir mit Z Spezifikationen in Teilen das Verhalten der Implementierung. Wir setzten also indirekt den Draft 13 in eine Z-Spezifikation um. Allerdings konnten wir nicht alle relevanten Teile in Z-Spezifikationen umsetzen, da dies zu aufwendig gewesen wäre.

Parallel zur Spezifikation entwickelten wir aus der Alpha-Implementierung eine lauffähige Implementierung. Durch Analyse der Z-Spezifikationen entdeckten wir zudem Inkonsistenzen und beseitigten diese. Das Ergebnis veröffentlichten wir im Internet. Dadurch kamen wir unter anderem mit Remy Card in Kontakt. Er begrüße unsere Arbeit und teilte uns mit, daß er die Implementierung nach der neuesten Version des Entwurfs von POSIX.1e [Sec97a] und POSIX.2c [Sec97b], Draft 17, überarbeite. Dabei gelangten wir auch an diese Standardentwürfe.

Wir hatten nun hohe Erwartungen an den Standardentwurf als Referenzdokument. Außerdem hielten wir unsere Arbeit für nützlicher für die Linux Community, wenn wir dem Maintainer des EXT2 nacheiferten. Daher überarbeiteten wir auch unsere Implementierung entsprechend dieser Version. Wir spezifizierten nun umgekehrt das Verhalten gemäß POSIX Standard und überarbeiteten die Implementierung entsprechend der Spezifikation. Bei dem größeren Teil dieser Arbeit mußten wir jedoch wegen Masse wiederum auf eine formale Spezifikation verzichten.

[1] Der Quellcode ist zu finden unter: ftp://tsx-11.mit.edu/pup/linux/ALPHA/ext2fs

Um die Korrektheit der Implementierung zu prüfen, unterzogen wir sie zunächst einem funktionalen Test. Dazu prüften wir die durch den Standardentwurf spezifizierten Eigenschaften einzeln. Auf eine solche Weise wird die Implementierung allerdings nur in Stichproben getestet. Dies ist gut geeignet, um grobe Fehler zu entdecken. Diese Methode ist jedoch unzureichend, um die Korrektheit eines Programmes zu gewährleisten. Genau dieses wollten wir aber für unsere Implementierung der Linux-ACL-Unterstützung zumindest partiell erreichen. Deshalb testeten wir zudem eine Teilfunktionalität unserer Implementierung, nämlich das Setzen einer ACL mit dem Kommando `setfacl`, mit einem umfangreichen automatisierten Test. Dazu verwendeten wir das Testwerkzeug RT-Tester und die Spezifikationssprache ASpecT, die wir benutzten, um ein korrektes Referenzsystem zu generieren.

Mit der Behebung der durch die Tests entdeckten Fehler und dem erneuten, nun fehlerfrei durchgeführten Test vervollständigten wir unsere Arbeit. Im Kapitel 20 wird dargestellt, an welchen Stellen sinnvoll weitergearbeitet werden könnte.

16. Implementationen von ACL

16.1 POSIX Standard P1003

Die Standardisierung von ACLs fand innerhalb des *Institute of Electrical and Electronics Enineers* (IEEE) durch die POSIX Arbeitsgruppe statt. Die Standards sind mit der P1003 Serie bezeichnet. Für unsere Implementierung verwandten wir P1003.1e [Sec97a] und P1003.2c [Sec97b] in der letzten Version des Standards, Draft 17, vom Oktober 1997. Die Standardisierung wurde eingestellt, die vorläufigen Versionen des Standards zurückgezogen.

P1003.1e [Sec97a] beschreibt unter anderem die Schnittstellen für Access Control Lists (ACLs) sowie die Namensgebung für Informationen. Dazu gehört

1. die Spezifikation des Algorithmus für die Zugriffskontrolle,
2. die Definition und der Umgang mit Access und Default ACLs,
3. die Definition der initialen Zugriffsrechte bei der Erzeugung eines Objekts[1],
4. die Funktionen zur Manipulation von ACLs.

P1003.2c [Sec97b] beschreibt die Systemprogramme für den Umgang mit ACLs.

Im folgenden werden diese Standardentwürfe genauer beschrieben. Hauptsächlich wird an dieser Stelle Draft 17 beschrieben. Daran schließt sich eine Kurzbeschreibung an, wie sich dieser Entwurf von Draft 13 unterscheidet, auf dem der Prototyp von Remy Card basiert.

16.1.1 Definition und Gebrauch von Access Control Lists

Jedes Objekt kann mit einer Access Control List assoziiert sein. Diese bestimmt die Zugriffsrechte auf dieses Objekt. Bislang sind die einzigen Objekte, für die dieses definiert ist, UNIX-Dateien. Eine solche ACL, die den Zugriff kontrolliert, wird als Access ACL[2] bezeichnet. Darüber hinaus können Verzeichnisse mit Default ACLs assoziiert werden. Diese bestimmen die initialen Zugriffsrechte von Dateien, die in diesem Verzeichnis erzeugt werden. P1003.1e[Sec97a] legt nicht die tatsächliche Implementierung von ACLs oder des Standard-Unix Rechtemechanismus nach P1003.1 fest. Genausowenig wird ein bestimmter Aufbau von internen Datenstrukturen oder eine festgelegte Reihenfolge der ACL-Einträge in einer ACL vorgeschrieben.

[1] POSIX definiert Objekte hier als Dateien

[2] POSIX bezeichnet diesen ACL-Typ wirklich als Access Access Control List, dieser Typ regelt nämlich tatsächlich den Zugriff, während die Default ACL nur zur Veerbung benutzt wird

16.1.2 Aufbau eines ACL-Eintrags

Ein ACL-Eintrag besteht aus drei Feldern:

- **Tag Type** Bestimmt den Typ des ACL-Eintrags. Hier sind die folgenden Typen definiert:
  - **ACL_USER_OBJ** Ein ACL-Eintrag von diesem Typ spezifiziert die Rechte für den Dateibesitzer. In einer gültigen ACL muß genau ein Eintrag von diesem Typ vorhanden sein.
  - **ACL_GROUP_OBJ** Ein ACL-Eintrag von diesem Typ spezifiziert die Rechte für die besitzende Gruppe der Datei. In einer gültigen ACL muß genau ein Eintrag von diesem Typ vorhanden sein.
  - **ACL_OTHER** Ein ACL-Eintrag von diesem Typ spezifiziert die Rechte für alle Prozesse für die kein anderer ACL-Eintrag maßgeblich ist. In einer gültigen ACL muß genau ein Eintrag von diesem Typ vorhanden sein.
  - **ACL_USER** Ein ACL-Eintrag von diesem Typ spezifiziert die Rechte für einen bestimmten Benutzer, der im Qualifier-Feld spezifiziert ist.
  - **ACL_GROUP** Ein ACL-Eintrag von diesem Typ spezifiziert die Rechte für eine bestimmte Gruppe, die im Qualifier-Feld spezifiziert ist.
  - **ACL_MASK** Ein ACL-Eintrag von diesem Typ (Mask Entry) spezifiziert die maximalen Rechte die einem Prozeß in der File Group Class gewährt werden. Zu der File Group Class gehören Prozesse, die nicht dem Dateibesitzer gehören und die auf einen der Einträge vom Typ **ACL_USER**, **ACL_GROU_OBJ** oder **ACL_GROUP** passen. Sofern eine ACL Einträge vom Typ **ACL_USER** oder **ACL_GROUP** enthält muß sie auch genau einen Eintrag vom **ACL_MASK** enthalten.
- **Qualifier** Qualifiziert den Tag Type. Für die Typen **ACL_USER** und **ACL_GROUP** wird hier die UID bzw. GID angegeben. Für alle anderen Typen bleibt dieses Feld leer.
- **Rechte** Dieses Feld spezifiziert die Rechte, die durch den Eintrag den durch Tag Type und Qualifier spezifizierten Prozessen gewährt werden.

Einträge vom Typ **ACL_USER_OBJ**, **ACL_GROUP_OBJ** oder **ACL_OTHER** werden *required ACL entries* genannt. Eine ACL, die nur die drei required entries beinhaltet wird *minimum ACL* genannt. Eine ACL, die weitere Einträge enthält wird *extended ACL* genannt.

POSIX erlaubt weitere implementationsspezifische Tag Types und auch über den Standard-UNIX-Rechte-Mechanismus hinausgehende Rechte.

16.1.3 Beziehung zum Standard-UNIX-Rechte-Mechanismus

POSIX.1 standardisert den Standard-UNIX-Rechte-Meschanismus. ACL-Schnittstellen erweitern die Schnittstelle der File Permission Bits. Die Kompatibilität mit Anwendungen, die POSIX.1-Schnittstellen benutzen, um File Permission Bits zu lesen und zu ändern - wie beispielsweise `chmod()` und `stat()`, - bleibt erhalten.

Die File Permission Bits korrespondieren folgendermaßen zu den ACL-Entries:

- Die Rechte für den Dateibesitzer korrespondieren mit den durch den **ACL_USER_OBJ**-Entry festgelegten Rechten.
- Die Rechte für die besitzende Gruppe korrespondieren mit den durch den **ACL_GROUP_OBJ**-Entry festgelegten Rechten bzw. mit den durch den **ACL_MASK**-Entry festgelegten Rechten, wenn die ACL einen Eintrag dieses Typs enthält.
- Die Rechte für Others korrespondieren mit den durch den **ACL_OTHER_OBJ**-Entry festgelegten Rechten.

16.1.4 Default ACLs

Eine Default ACL kann einem Verzeichnis zugeordnet sein, hat aber keinen Einfluß auf die Zugriffskontrolle auf dieses Verzeichnis. Eine solche Default ACL wird dazu benutzt die initialen ACLs der in diesem Verzeichnis erzeugten Objekte festzulegen.

Wenn ein Verzeichnis eine Default ACL besitzt, erhalten alle Dateien, die in diesem Verzeichnis erstellt werden, diese als Access ACL. Verzeichnisse erhalten zusätzlich die Default ACL vererbt.

Eine Default ACL ist nach den gleichen Regeln aufgebaut, wie eine Access ACL.

16.1.5 Algorithmus zur Zugriffskontrolle

In P1003.1e [Sec97a] wird zunächst die Reihenfolge festgelegt, in der die ACL-Einträge (Entries) durch den Algorithmus zur Zugriffskontrolle überprüft werden:

1. Den **ACL_USER_OBJ** Entry.
2. Die **ACL_USER** Entries.
3. Den **ACL_GROUP_OBJ** Entry und die **ACL_GROUP** Entries.
4. Den **ACL_OTHER** Entry.

Anschließend spezifiziertP1003.1e [Sec97a] den Algorithmus zur Zugriffkontrolle folgendermaßen:

1. **If** the effective user ID of the process matches the user ID of the object owner
then
set matched entry to ACL_USER_OBJ entry

2. **else if** the effective user ID of the process matches the user ID specified in any ACL_USER tag type ACL entry,
then
set matched entry to the matching ACL_USER entry

3. **else if** the effective group ID or any of the supplementary group IDs of

the process match the group ID of the object or match the group ID specified in any ACL_GROUP or ACL_GROUP_OBJ tag type ACL entry
then
if the requested access modes are granted by at least one entry matched by the effective group ID or any of the supplementary group IDs of the process
then
set matched entry to a granting entry
else
access is denied
endif

4. **else if** the requested access modes are granted by the ACL_OTHER entry of the ACL,
then
set matched entry to the ACL_OTHER entry
endif

5. **If** the requested access modes are granted by the matched entry
then
if the matched entry is an ACL_USER_OBJ or ACL_OTHER entry
then
access is granted
else if the requested access modes are also granted by the ACL_MASK entry or no ACL_MASK entry exists in the ACL
then
access is granted
else
access is denied
endif
else
access is denied
endif

16.1.6 ACL Funktionen

Eine Funktionsaufrufschnittstelle ist definiert, um ACLs und ACL-Entries zu verändern. Es werden vier Gruppen von Funktionen definiert:

1. Für das Management des ACL-Arbeitsspeichers.
2. Um ACL-Entries zu bearbeiten.
3. Eine ACL zu einem Objekt zu bearbeiten.
4. Eine ACL in verschiedene Formate umzuwandeln.

Jede dieser Funktionen ist in P1003.1e [Sec97a] - in Form und Umfang vergleichbar mit einer Manual-Page - beschrieben. Auf eine Beschreibung dieser Funktionen wird an dieser Stelle verzichtet, da diese zwar für unsere Implementierungsarbeiten wichtig waren, aber für den automatisierten Test keine Rolle spielten, da wir nur auf Anwendungsebene getestet haben.

16.1.7 ACL Dienstprogramme

Zum Umgang mit ACLs beschreibt POSIX.2c [Sec97b] zwei Dienstprogramme:

getfacl

Syntax

```
getfacl [-d] [file...]
```

Beschreibung

Das Kommando **getfacl** dient dazu Access und Default ACLs von Dateien und Verzeichnissen anzeigen zu lassen.

Optionen

- **-d** Anstatt der Access ACL wird die Default ACL angezeigt.

setfacl

Syntax

```
setfacl [-kbdn] [-x entries] [-X file1]
                [-m entries] [-M file2] [file...]
```

Beschreibung

`setfacl` dient dazu Access und Default ACLs von Dateien und Verzeichnissen zu verändern. Das Programm `setfacl` stellt folgende Funktionen zur Verfügung:

- Löschen aller Einträge außer den drei nicht optionalen.
- Löschen der Default ACL.
- Verändern einer ACL durch Hinzufügen bzw. Ersetzen von Einträgen. Die Einträge können in der Kommandozeile und/oder durch Angabe einer Datei, die die Einträge beinhaltet, spezifiziert werden.
- Verändern einer ACL durch Löschen von Einträgen. Die Einträge können in der Kommandozeile und/oder durch Angabe einer Datei, die die Einträge beinhaltet, spezifiziert werden.
- Automatische Neuberechnung des Mask-Entries.
- Bei Verzeichnissen kann wahlweise die Default oder die Access ACL, bei anderen Dateien lediglich die Access ACL bearbeitet werden.

Optionen

- **-k** Löscht die Default ACL der angegebenen Verzeichnisse.
- **-b** Löscht alle Einträge der ACL bis auf die required entries `(u::, g::, o::)`.

- **-d** Die Operation bezieht sich auf die Default ACL statt der Access ACL.
- **-n** Bewirkt, daß der Mask Entry nicht automatisch neu berechnet wird.
- **-x entries**
  Die angegebenen Einträge werden aus der ACL gelöscht, sofern Tag, Type und Qualifier übereinstimmen. Die Einträge werden in der Form `Type:Qualifier:Permission[,Type:Qualifier:Permission...]` angegeben.
- **-X file1**
  Wie `-x`, jedoch werden die zu löschenden Einträge aus der Datei `file1` gelesen.
- **-m entries**
  Die angegebenen Einträge werden zu der bestehenden ACL hinzugefügt, bzw. die Rechte der bestehenden Einträge werden angepaßt, sofern Tag Type und Qualifier übereinstimmen. Die Einträge werden in der Form `Type:Qualifier:Permission[,Type:Qualifier:Permission...]` angegeben.
- **-M file2**
  Wie `-m`, jedoch werden die zu modifizierenden Einträge aus der Datei `file2` gelesen.

16.1.8 Unterschiede von Draft 13 zu Draft 17

- Draft 13 unterstützt keine Mask Entries.
- Draft 13 verwendet die Kommandonamen `getacl` und `setacl` für die Dienstprogramme[3].
- Bei `setacl` gibt es Unterschiede in den Optionen:
  - Bei `setacl` wird mit der Option `-a` expliziet die Access ACL spezifiziert, standardmäßig wird bei Verzeichnissen die Default ACL bearbeitet. `setfacl` bearbeitet bei Verzeichnissen standardmäßig die Access ACL, die Default ACL muß mit der Option `-d` spezifiziert werden.
  - `setacl` verwendet die Option `-u` zum verändern einer ACL, `setfacl` verwendet die Option `-m`.
  - Da in Draft 13 keine Mask Entries unterstützt werden, gibt es auch keine Option `-n`, um die Neuberechnung des Mask Entry einer ACL zu unterbinden.

16.2 Unix Derivate mit ACL Unterstützung

Die ACL Implementierungen verschiedener Unix Derivate, namentlich HP, Solaris und Digital Unix, sollen hier vorgestellt werden. Dabei wird mit dem POSIX Standard P1003 Draft 17 und untereinander verglichen.

[3] Vermutlich wurde in Draft 17 die Namensgebung von Solaris übernommen. Was das “f” bedeuten könnte weiß wohl nur der Sonnengott

16.2.1 Digital Unix

Zur Untersuchung der Digital Unix Implementierung diente [Dig96]. Die aktuelle Implementierung von ACLs bei Digital UNIX basiert auf dem Standard POSIX P1003 Draft 13 und erweitert diesen. Folgende Abweichungen, soweit sie hier von Interesse sind, bestehen gegenüber Draft 17:

ACLs haben keinen *mask* Eintrag. Entsprechend existiert keine Einschränkung der gewährten Rechte durch diesen Eintrag. Ansonsten ist der **Zugriff** wie bei Draft 17 geregelt.

Dateien haben wie bei Draft 17 eine Access ACL, die den Zugriff auf die Datei regelt. Verzeichnisse können hingegen drei ACLs haben:

1. **Access ACL** Diese ACL kontrolliert wie bei Draft 17 den Zugriff auf das Verzeichnis.
2. **Default Directory ACL** Diese ACL wird an Unterverzeichnisse, die in einem Verzeichnis kreiert werden, als Access ACL und Default Directory ACL vererbt. Dies ist eine Digital-Unix Erweiterung, die nicht in Draft 13 spezifiziert ist.
3. **Default Access ACL** Diese ACL wird an Dateien und Unterverzeichnisse, die in einem Verzeichnis kreiert werden, als Access ACL vererbt. Diese ACL wird an Unterverzeichnisse, die in einem Verzeichnis kreiert werden, als Default Access ACL vererbt.

16.2.2 Solaris

Zur Untersuchung der Solaris Implementierung dienten die Manuals [Sun96], [Sun94a], [Sun94b], [Sun94c], [Sun94d], [Sun94e], [Sun94f], [Sun97]. Der wesentliche Unterschied der Solaris Implementierung zu Draft 17 besteht in der Behandlung der Gruppenrechte. Unter Solaris erhält ein Benutzer die Vereinigung der Rechte aller Gruppeneinträge der ACL, in denen er Mitglied ist. Im Unterschied dazu werden die Rechte nach Draft 17 nur dann gewährt, wenn ein ACL-Eintrag für eine der Gruppen des Benutzers alle angeforderten Rechte gewährt. Weiterhin unterscheidet sich die Solaris Implementierung in der Syntax des Aufrufs des Kommandos `setfacl`:

- Bei Solaris hat die Option `-d` eine andere Bedeutung als bei P1003.2c [Sec97b].Dies ist weiter unten im Text beschrieben. Einträge für die Default ACL werden bei Solaris spezifiziert, indem vor den Eintrag `d:` bzw. `default:` vorangestellt wird.
- Es ist bei Solaris möglich, die Rechte in oktaler Schreibweise anzugeben.
- Es gibt keine Optionen `-k` und `-b` wie bei P1003.2c [Sec97b], stattdessen gibt es die Option `-s acl entries`, die wie die Option `-m acl entries` arbeitet, also die durch die Zeichenkette `acl entries` spezifizierten Einträge hinzufügt bzw. ändert, jedoch zuvor alle vorhandenen Entries löscht.
- Die Option `-d acl entries` hat die Funktionalität der Option `-x` wie bei P1003.2c [Sec97b], löscht also die durch die Zeichenkette `acl entries` spezifizierten Einträge.

- Solaris unterstützt keine Funktionalität, wie sie die Optionen `-X acl file` und `-M acl file` bei P1003.2c [Sec97b] bieten. Stattdessen gibt es bei Solaris eine Otion `-f acl file`, die es erlaubt eine ACL entspechend der ACL-Einträge aus der durch den Dateinamen `acl file` spezifizierten Datei zu setzen.
- Statt der Option `-n`, die bei P1003.2c [Sec97b] die Neuberechnung des Mask Entries unterbindet, stellt Solaris die Option `-r` zur Verfügung, die eine Neuberechnung spezifiziert. Ohne diese Option, wird der Mask Entry nicht neuberechnet.

16.2.3 HP-UX 10.20

Zur Untersuchung der HP-UX Implementierung dienten die Manuals [Hew96a], [Hew96b], [Hew96c]. Die ACL Implementierung von HP-UX10.20 unterscheidet sich wesentlich von den auf den Standardentwürfen von POSIX basierenden Implementierungen von Solaris und Digital Unix. Daher wird hier eine genauere Beschreibung und ein Vergleich der Implementierungen gegeben.

Aufbau der ACL

Eine ACL besteht aus einer Reihe von Einträgen. Einträge lassen sich in vier Klassen mit verschiedenen Prioritäten einteilen:

- *u.g* steht für Benutzer *u* in Gruppe *g*.
- *u.%* steht für Benutzer *u* in einer beliebigen Gruppe.
- *%.g* steht für einen beliebigen Benutzer *u* in Gruppe *g*.
- *%.%* steht für einen beliebigen Benutzer in einer beliebigen Gruppe.

Einträge in der ACL müssen eindeutig sein; keine zwei Einträge mit derselben Benutzer- und Gruppen-ID dürfen vorkommen. Jede ACL hat drei Basis Einträge, die nicht hinzugefügt oder gelöscht, sondern nur geändert werden können. Auf diese Basis ACL werden die Standard Zugriffsrechte direkt abgebildet:

- `<Datei Besitzer>`*.% user* Zugriffsrechte.
- *%.*`<Datei besitzende Gruppe>` *group* Zugriffsrechte.
- *%.% other* Zugriffsrechte.

Werden die Basis ACL Einträge mit dem HP-UX-Kommando `setacl()` geändert, so werden auch die entsprechenden Standard Zugriffsrechte geändert. Durch einen erfolgreichen Aufruf von `setacl()` werden, wenn vorhanden, alle vorherigen, optionalen (nicht-Basis) Einträge gelöscht. Wird ein Basis Eintrag in der neuen ACL nicht angegeben, werden die entsprechenden Zugriffsrechte auf 0 gesetzt. Dadurch hinterlassen Aufrufe von `setacl()` eine vollständig definierte ACL.

Zugriffskontrolle

Die Rechte zum Lesen, Schreiben und Ausführen/Durchsuchen werden nacheinander einzeln gegen die ACL geprüft.

Die echte Benutzer ID wird mit der echten Gruppen ID des Prozesses und jeder Gruppen ID in der Liste zusätzlicher Gruppen kombiniert. Für diese Kombination werden passende Einträge in der ACL gesucht.

Die Suche erfolgt in der Reihenfolge der Priorität der Einträge. Hier hat ein Eintrag, der die Benutzer und Gruppen genauer spezifiziert, eine höhere Priorität. Beispielweise ist ein Eintrag der Form *u.g* höher priorisiert als *u.%* oder *%.g*. Die Suche wird beendet, wenn mindestens ein passender Eintrag in einer Klasse gefunden wurde. Durch die zusätzlichen Gruppen können mehr als ein *u.g* oder *%.g* Eintrag passen. Enthält einer der passenden Einträge das angeforderte Zugriffsrecht, wird der Zugriff gewährt.

16.3 Prototyp für Linux von Remy Card

Die Implementierung von Remy Card bestand aus einem Patch auf den Linux-Kernel Version 2.1.99, einer Bibliothek und einigen Werkzeugen u.a. zum Lesen und Setzen von ACLs. Die Implementierung war von der Intention her konform zu Draft 13. Dieser Standard sah auch die Bibliothek und einen Teil der Werkzeuge zum Manipulieren von ACLs vor. Der Patch von Remy Card war nach nach eigener Dokumentation zwischen zwei Vorträgen auf einem Kongreß entstanden. Dementsprechend stellten wir bei ersten Tests fest, daß die Implementierung Alpha-Qualität hatte.

16.3.1 Kernel

Zur Speicherung der ACLs sind pro Dateisystem zwei spezielle Inodes reserviert. In den Datenblöcken des **ACL Index Inode** wird für jede ACL ein Header gespeichert der u.a. den Index des ersten ACL Entries im **ACL Data Inode** enthält.

Die durch diese Inodes referenzierten Datenblöcke sind folgendermaßen aufgebaut:

```
+----------------------------------+
|Block descriptor (ext2_acl_desc) |
+----------------------------------+
| Bitmap                           |
+----------------------------------+
| Header or Entry #1               |
+----------------------------------+
| ...                              |
+----------------------------------+
```

Blockdeskriptor und Bitmap enthalten Informationen für die Freispeicherverwaltung. Alle Datenstrukturen haben die gleiche Größe, damit läßt sich die Position eines Eintrags leicht aus einem Index berechnen:
Datenblocknummer = index / max. Anzahl von Einträgen pro Block
Position im Block = index % max. Anzahl von Einträgen pro Block.

Jeder Inode im EXT2-Dateisystem besitzt zwei Felder, das eine enthält einen Verweis auf eine File ACL, das andere auf eine Directory ACL. Beide sind jeweils NULL, wenn keine entsprechende ACL assoziiert ist. Wenn eine ACL assoziiert ist, enthält das entsprechende Feld den Index des ACL

Headers im ACL Index Inode. Dieser Header enthält nun den Index des ersten ACL Entries der ACL im ACL Data Inode. In der jetzigen Version werden die ACL Entries fortlaufend innerhalb eines Datenblockes abgespeichert. Daraus ergibt sich eine maximale ACL-Größe, die der maximalen Anzahl von ACL Entries, die in einem Block gespeichert werden können entspricht. Dies sind etwa dreißig Einträge bei der Standard-Blockgröße von 1024 Byte. Die Datenstruktur eines ACL Entries sieht bereits eine einfache Verkettung vor, so daß eine Fragmentierung einer ACL über mehrere Datenblöcke hinweg keine größeren Probleme im Weg stehen.

Im Kernelcode wurden hierzu die folgenden Funktionen in der Datei `*/fs/ext2/acl.c` definiert.

- **check_acl_descriptor()** Prüft in einem Datenblock, ob die Daten im Block Descriptor und in der Bitmap zueinander konsistent sind.
- **update_descriptor()** Setzt in einem Datenblock die Daten im Block Descriptor entsprechend der Daten in der Bitmap.
- **allocate_acl_header()** Allokiert einen ACL Header im ACL Index Inode und gibt einen Zeiger auf den Header zurück. Enthält der Inode keinen Datenblock mit freien Einträgen mehr, wird ein neuer allokiert und die Datenstrukturen geschrieben.
- **allocate_acl_entries()** Allokiert die angegebene Anzahl von ACL Entries im ACL Data Inode und gibt einen Zeiger auf den ersten Entry zurück. Enthält der Inode keinen Datenblock mit genügend freien Einträgen mehr, wird ein neuer allokiert und die Datenstrukturen geschrieben. Ist die Anzahl der zu allokierenden Entries größer als die maximal in einem Block speicherbaren Entries gibt die Funktion NULL zurück.
- **free_acl_header()** Markiert den freizugebenden Header in dem Datenblock als frei, sofern er nicht noch von anderen Inodes referenziert wird. Werden dadurch alle Header in dem Datenblock frei, wird der Datenblock selbst **nicht** frei gegeben. Dies hat zur Folge, daß zum Speichern von ACL Headern allokierte Datenblöcke nicht wieder zum Speichern von anderen Daten zur Verfügung gestellt werden.
- **free_acl_entries()** Markiert die freizugebenden Entries in dem Datenblock als frei. Werden dadurch alle Entries in dem Datenblock frei, wird der Datenblock selbst **nicht** frei gegeben. Dies hat zur Folge, daß zum Speichern von ACL Entries allokierte Datenblöcke nicht wieder zum Speichern von anderen Daten zur Verfügung gestellt werden.
- **get_acl_header()** Gibt einen Zeiger auf den ACL Header des angegebenen ACL Typs (Access/Default) zu einem Inode zurück. Wenn keine entsprechende ACL assoziiert ist oder im Fehlerfall gibt die Funktion NULL zurück.
- **get_acl_entries()** Gibt einen Zeiger auf den ersten ACL Entry einer ACL zu dem angegebenen ACL Header und Inode zurück. Im Fehlerfall gibt die Funktion NULL zurück.

Wird eine Datei in einem Verzeichnis mit Default ACL erzeugt, erhält die Datei diese Default ACL als Access ACL und, wenn es sich um ein Verzeichnis handelt, zusätzlich als Default ACL. Dies wird im Kernelcode so geregelt, daß die ACL-Felder im Inode der neu erzeugten Datei auf die Default ACL des Verzeichnisses verweisen, in dem diese erzeugt wurde. Weiter enthält der ACL Header zu jeder ACL einen Referenz-Zähler. Dieser enthält die Anzahl der Inodes, die diese Default ACL referenzieren. Wird die ACL zu einem dieser Inodes verändert, erhält dieser Inode eine Kopie der Default ACL, und auf dieser ACL werden die Veränderungen vollzogen. Weiter wird der Referenzzähler dekrementiert, wenn Inode oder ACL zu dem Inode gelöscht werden. Diesen Mechanismus unterstützt der Kernelcode mit den folgenden Funktionen:

- **ext2_inherit_acl()** Diese Funktion vererbt eine ACL eines Inodes an einen anderen Inode. Dazu wird der Referenzzähler im ACL Header der zu vererbenden ACL erhöht und ein Verweis in den Inode eingetragen, der die ACL erbt. Die Funktion wird aufgerufen, wenn eine Datei oder ein Verzeichnis in einem Verzeichnis mit Default ACL erzeugt wird.
- **ext2_copy_acl()** Wenn ein Inode auf eine gemeinsam benutzte ACL verweist, erzeugt diese Funktion eine Kopie der gemeinsam genutzten ACL und ersetzt den Verweis im Inode auf die gemeinsam genutzte ACL durch einen Verweis auf die Kopie. Weiter wird der Referenz-Zähler im ACL Header der gemeinsam benutzten ACL dekrementiert. Diese Funktion wird aufgerufen, wenn die ACL zu einem Inode geändert wird.

POSIX verlangt, daß die Rechte-Bits im Inode konsistent zu den required entries in einer ACL sind. Dies wird durch die folgenden Funktionen realisiert:

- **ext2_update_acl_from_mode()** Diese Funktion wird aufgerufen, wenn die Rechte eines Inodes mittels `chmod()` verändert werden und verändert die `ACL_USER_OBJ`, `ACL_GROUP_OBJ` und `ACL_OTHER_OBJ`-Einträge der ACL, auf die der Inode verweist, entsprechend der eingestellten Rechte.
- **ext2_update_mode_from_acl()** Durch diese Funktion werden die Rechte des Inodes entsprechend einer als Parameter übergebenen, nur aus den drei required entries bestehenden ACL eingestellt.

Beim Zugriff auf die Daten eines Inodes wird - auch ohne ACL-Unterstützung - im Kernelcode die Funktion `ext2_permission` aufgerufen, um die Zugriffsrechte zu überprüfen. Mit ACL-Unterstützung ist diese Funktion derart erweitert, daß sie die Funktion `ext2_acl_permission()` aufruft, sofern die ACL-Unterstützung aktiviert ist und der Inode auf eine Access ACL verweist. Diese führt die Zugriffskontrolle anhand der ACL durch. Dazu durchsucht diese Funktion die ACL-Einträge iterativ, bis ein passender Eintrag gefunden wird. Bestimmt durch die Rechte dieses Eintrags wird dann der Zugriff gewährt oder verweigert. Dieser Algorithmus setzt voraus, daß die Einträge der ACL nach ihren Prioritäten sortiert gespeichert wurden. Dies vereinfacht den Aufwand und erhöht die Schnelligkeit des Algorithmus, da nicht alle Einträge durchsucht werden müssen, sondern nach dem ersten passenden abgebrochen werden kann, da alle folgenden passenden Einträge mit niedriger Priorität versehen sind.

Für den Fall, daß ACL-Unterstützung aktiviert ist, einem Inode aber keine ACL zugewiesen wurde, werden die folgenden beiden Funktionen benötigt:

- **copy_mode_to_acl()** Diese Funktion erzeugt eine ACL mit den drei required entries. Die Rechte der Einträge werden entsprechend den im Inode gespeicherten Rechten gesetzt. Wenn ein Inode auf keine Access ACL verweist, wird diese Funktion benutzt, um eine ACL entsprechend der Standard Unix Rechte zu erzeugen.
- **make_null_acl()** Diese Funktion erzeugt eine leere ACL und wird benutzt, wenn die Default ACL eines Inodes gelesen wird und dieser Inode auf keine Default ACL verweist.

Zum Setzen, Löschen und Lesen einer ACL bietet der Kernelcode die folgenden Funktionen:

- **copy_acl_out()** Liest eine ACL eines Inodes und kopiert deren Inhalt in den Datenbereich des User Space.
- **copy_acl_in()** Speichert eine ACL aus dem User Space, trägt einen Verweis darauf in einen Inode ein und stellt die Rechte in den File Permission Bits dieses Inodes entsprechend ein.
- **ext2_remove_acl()** Löscht die Access ACL bzw. Default ACL eines Inodes und den Verweis auf die ACL im Inode. Zum Löschen wird lediglich der Referenzzähler im ACL-Header dekrementiert. Erst wenn dieser Null ist, wird die ACL tatsächlich gelöscht.
- **set_acl()** Ersetzt eine Access oder Default ACL eines Inodes. Dazu wird eine eventuell vorhandene ACL mit `ext2_remove_acl()` gelöscht. Handelt es sich um eine Access ACL und besteht diese lediglich aus den drei required entries, werden ausschließlich die Rechte im Inode selbst geändert. Ansonsten wird die neue ACL mittels `copy_acl_in()` gespeichert.

Der eigentliche Systemaufruf ist durch die Funktionen **sys_acl_ctl()** und **ext2_acl_ctl()** realisiert. `sys_acl_ctl()` befindet sich im Virtual File System Layer und ist der eigentliche Systemaufruf. Dieser ruft dann die Funktion `ext2_acl_ctl()` auf, sofern sich der zu bearbeitende Inode auf einem EXT2-Dateisystem befindet. Befindet sich der Inode auf einem anderen Dateisystem, wird eine entsprechende Funktion des jeweiligen Dateisystems aufgerufen. Da andere Dateisysteme unter Linux diese Funktion nicht unterstützen, bleibt dieser Aufruf wirkungslos. Weiterhin überprüft `sys_acl_ctl()` für den Fall, daß eine ACL gesetzt werden soll, ob die zu setzende ACL gültig ist. Dazu müssen die folgenden Bedingungen erfüllt sein:

1. Jede ACL und jeder Eintrag einer ACL haben ein Feld in dem der konstante Wert `LINUX_ACL_VALID` eingetragen sein muß. Dies wird überprüft.
2. Die ACL muß mindestens die drei required entries beinhalten.
3. Die Einträge der ACL müssen nach Priorität sortiert sein: `ACL_USER_OBJ`, `ACL_USER`, `ACL_GROUP_OBJ`, `ACL_GROUP`, `ACL_OTHER_OBJ`.
4. Die Rechte, die in den Entries gespeichert sind, müssen gültig sein. Das bedeuted, daß keine anderen Rechte als read, write und execute Recht bzw. deren Kombinationen zugelassen sind.

Weiter enthält der Kernelcode in der Datei `arch/i386/kernel/entry.S` einen neuen Eintrag für den Systemaufruf `sys_acl_ctl()`.

`ext2_acl_ctl()` stellt die folgenden Dienste zur Verfügung, die sich durch den Parameter `operation` auswählen lassen:

- Lesen der Anzahl der Einträge einer ACL.
- Lesen einer ACL.
- Setzen einer ACL.
- Löschen einer ACL.

Durch weitere Parameter wird angegeben, auf welchen Inode sich die Operation bezieht und ob sich die Operation auf die Access ACL oder die Default ACL bezieht. Diese Funktionalität werden durch den Aufruf der entsprechenden oben beschriebenen Funktionen implementiert.

Neben der erweiterten Zugriffskontrolle in `ext2_permission` und der Realisierung des Systemaufrufs `sys_acl_ctl()` wurden unter anderem folgenden Funktionen des EXT2-Dateisystems verändert, um den ACL Support zu realisieren:

- **ext2_new_inode** Wenn sich das Verzeichnis, in dem der neue Inode erzeugt wird, auf einem Dateisystem mit aktivierten ACL Support befindet, wird die Access ACL und ggf. auch die Default ACL des Verzeichnisses an den neu erzeugten Inode vererbt. Dazu wird die Funktion `ext2_inherit_acl()` benutzt.
- **ext2_notify_change()** Diese Funktion wird aufgerufen, wenn sich die Dateigröße, ein Zeiteintrag, der Modus (d.h. die Rechte), die UID oder GID eines Inodes geändert hat. Für den Fall, daß ACL Support aktiv ist und sich die Rechte des Inodes geändert haben, wird nun die Funktion `ext2_update_acl_from_mode()` aufgerufen, um die Änderungen auch in der ACL zu vollziehen.
- **ext2_put_super()** Index Inode und Data Inode werden zurückgeschrieben.
- **parse_options** Beim Mounten eines EXT2-Dateisystems wird die Option "`acl=yes`" unterstützt und zeigt das Aktivieren des ACL Support an.
- **ext2_read_super()** Index Inode und Data Inode werden gelesen. Weiter wird die Anzahl der maximalen Einträge von ACL-Headern und ACL-Einträgen per Datenblock bestimmt.

16.3.2 Benutzerebene

Auf Benutzerebene implementierte Remy Card zunächst eine Funktionsbibliothek mit den durch POSIX 1003.1e Draft 13 spezifizierten Funktionen. Diese sind in wesentlichen eine Teilmenge der in Draft 17 beschriebenen Funktionen und werden hier nicht näher erläutert. Die eigentlichen Dienstprogramme **getacl** und **setacl** wurden dann unter Verwendung dieser Bibliothek implementiert.

getacl

Syntax

```
getacl [-d] [file...]
```

Beschreibung

Das Programm **getacl** zeigt die ACLs der als Parameter angegebenen Dateien an. Durch Angabe der Option "-d" kann bestimmt werden, daß anstatt der standardmäßig angezeigten Access ACL die Default ACL der Dateien angezeigt wird. Dazu werden die folgenden Funktionen verwendet:

- **usage()** Im Falle einer syntaktisch nicht korrekten Kommandozeile wird die Funktion `usage()` aufgerufen, die die korrekte Syntax angibt und das Programm abbricht.
- **main()** Wertet die angegeben Optionen und Parameter aus. Für jeden angegebenen Dateinamen wird dann die Funktion **getacl()** aufgerufen. Ist statt eines Dateinamens "`-`" angegebenen ruft `main()` die Funktion **get_acl_from_stdin()** auf.
- **getacl()** Liest die ACL zu einer Datei, wandelt diese ACL in Textform um und gibt sie aus.
- **get_acl_from_stdin()** Liest Dateinamen von der Standardeingabe und ruft für jeden dieser Namen dann `getacl()` auf.

setacl

Syntax

```
setacl [-k] [-b] [-d] [-x entries] [-X file1]
                      [-u entries] [-U file2] [file...]
```

Beschreibung

Das Programm **setacl** wertet die Optionen aus, liest die ACL zu den angegebenen Dateien und modifiziert diese entsprechend der Optionen und schreibt die modifizierten ACLs dann zurück.

Optionen

- **-k** Löscht die Default ACL der angegebenen Verzeichnisse.
- **-b** Löscht alle Einträge der ACL bis auf die required entries `(u::, g::, o::)`.
- **-d** Die Operation bezieht sich auf die Default ACL statt der Access ACL.
- **-x entries**
  Die angegebenen Einträge werden aus der ACL gelöscht, sofern Tag Type und Qualifier übereinstimmen. Die Einträge werden in der Form `Type:Qualifier:Permission[,Type:Qualifier:Permission...]` angegeben.
- **-X file1**
  Wie `-x`, jedoch werden die zu löschenden Einträge aus der Datei `file1` gelesen.

- **-u entries**
  Die angegebenen Einträge werden zu der bestehenden ACL hinzugefügt, bzw. die Rechte der bestehenden Einträge werden angepaßt, sofern Tag Type und Qualifier übereinstimmen. Die Einträge werden in der Form `Type:Qualifier:Permission[,Type:Qualifier:Permission...]` angegeben.

- **-U file2**
  Wie `-u`, jedoch werden die zu modifizierenden Einträge aus der Datei `file2` gelesen.

Dazu wurden zusätzlich zu den Funktionen aus der Funktionsbibliothek die folgenden Funktionen implementiert:

- **usage()** Im Falle einer syntaktisch nicht korrekten Kommandozeile wird die Funktion `usage()` aufgerufen, die die korrekte Syntax angibt und das Programm abbricht.

- **remove_entries_but_three()** Entfernt aus einer ACL alle Einträge außer den drei required entries.

- **remove_entries()** Entfernt aus einer ACL die in einer zweiten ACL angegebenen Einträge.

- **update_entry()** Verändert die Rechte eines Eintrags entsprechend denen eines zweiten. Mit einem weiteren Parameter wird bestimmt, ob die Rechte hinzugefügt, ersetzt oder gelöscht werden sollen.

- **add_entry()** Fügt einer ACL einen als weiteren Parameter angegeben Eintrag hinzu.

- **update_entries()** Fügt einer ACL die Einträge einer zweiten hinzu bzw. ersetzt diese, wenn sie bereits vorhanden sind. Letzteres ist der Fall, wenn in der ersten AVL ein Eintrag mit gleichem Tag und Qualifier existiert.

- **acl_from_file()** Liest eine ACL aus einer Datei. Die Datei enthält die ACL in Textform. Dies wird für die Optionen -U und -X benötigt.

- **set_acl** Liest für den angegebenen Dateinamen die ACL, modifiziert diese entsprechend der durch die Optionen im Kommandoaufruf angegebenen Regeln und schreibt die ACL zurück

- **set_acl_from_stdin()** Ist in der Kommandozeile als Parameter statt eines Dateinamens "`-`" angegeben list die Funktion Dateinamen von stdin und ruft für jeden dieser Namen setfacl() auf.

- **main()** Wertet die angegeben Optionen aus. Dabei werden für Schalteroptionen Variablen gesetzt. Weiter werden die zu modifizierenden Einträge von der Kommandozeile aus der Textform in eine ACL übersetzt bzw. aus einer Datei mit Hilfe von `acl_from_file()` gelesen. Für jeden der nach den Optionen angegeben Dateinamen ruft `main()` die Funktion `setacl()` bzw. `set_acl_from_stdin()` auf, letzteres, wenn statt eines Dateinamens "`-`" angegeben wurde.

16.4 Implementierung des Projekts

16.4.1 Kernel

Die ACLs werden in zwei separaten, sonst nicht zugänglichen Dateien gehalten. Das hat den Vorteil, daß die Änderung am Dateisystem abwärtskompatibel und aufwärtskompatibel ist: Ein EXT2-Dateisystem mit ACLs kann von einer Version des Dateisystemtreibers ohne ACL Unterstützung gelesen werden. Dabei können die ACLs natürlich nicht genutzt werden. Ein EXT2-Dateisystem ohne ACLs kann von dem Dateisystemtreiber mit ACL Unterstützung gelesen werden. Die nötigen Änderungen am Dateisystem werden automatisch ausgeführt.

Lediglich das Programm, das die Konsistenz eines EXT2-Dateisystems überprüft (e2fsck) ist nicht aufwärtskompatibel: Bei der Überprüfung eines EXT2-Dateisystems mit ACLs meldet die Version des Programmes ohne ACL Unterstützung Fehler. Das bedeutet, daß eine Version des Programmes mit ACL Unterstützung verwendet werden sollte, auch wenn man vorhandene ACLs nicht verwenden will. Hier kann man also nicht einfach zurückschreiten auf ein System ohne ACL Unterstützung: Tut man dies doch, wird man von e2fsck mit einer langen Liste an Abfragen konfrontiert oder man kann sich entscheiden, alle ACLs unwiederbringlich zu löschen.

Die erste Datei dient als Tabelle mit Verweisen in die zweite Datei, die die ACLs enthält. Dadurch wird Copy-on-Write realisiert: Wird eine ACL dupliziert, so wird unmittelbar nur der Verweis in der ersten Datei auf die ACL in der zweiten Datei dupliziert. Erst wenn die ACL verändert wird, wird die ACL selbst in der zweiten Datei dupliziert.

Die Verwaltung der ACLs in diesen beiden Dateien war grundsätzlich in der Version von Remy Card vorhanden; einige Fehler waren zu beheben. Speziell die Konvertierung zwischen Byte-Endian des Dateisystems (klein) und der Byte-Endian der CPU (klein bei Intel) fehlte teilweise. Dieser Fehler hätte sich jedoch nur auf einer Architektur mit großer Byte-Endian gezeigt. Leider konnten wir unsere Fehlerkorrektur nicht überprüfen, da eine entsprechende Architektur nicht zur Verfügung stand. Wir hoffen, dies baldmöglichst nachholen zu können.

Bei der Portierung der ACL Unterstützung auf Draft 17 mußte die Zugriffskontrolle und die Validation von ACLs neu geschrieben werden. Die Manipulation von ACLs geschieht durch einen neuen Systemaufruf. Dieser verlangt auch zwei neue Fehlernummern.

Die ACL Unterstützung wurde in zweierlei Weise optional gestaltet:

- Die ACL Unterstützung läßt sich bei der Konfiguration des Linux-Kernel, also zur Kompilierzeit, ein- und ausschalten.
- Ein Dateisystem kann beliebig mit und ohne ACL Unterstützung gemountet werden.

Allerdings kann es zu Inkonsistenzen zwischen den Standard Zugriffsrechten und den ACLs kommen, wenn ein Dateisystem mal mit ACLs und mal ohne ACL Unterstützung gemountet wird und dann die Standard-UNIX-Zugriffsrechte geändert werden. Beispiel: Eine Datei hat eine ACL erhalten, als das Dateisystem mit ACL Unterstützung gemountet war, die Standard-UNIX-Zugriffsrechte wurden angepaßt.

Wird dasselbe Dateisystem jetzt ohne ACL Unterstützung gemountet und die Standard-UNIX-Zugriffsrechte der Datei geändert, sind ACL und Standard-UNIX-Zugriffsrechte nicht mehr konsistent.

Der Code zum Remounten von EXT2-Dateisystemen mußte adaptiert werden, um Änderungen bzgl. der Unterstützung von ACLs zu reflektieren. Dies war vor allem wichtig, um die ACL Unterstützung für das root Dateisystem zu nutzen: Es bedarf einer Sonderbehandlung, um für das root Dateisystem mount Optionen anzugeben. Die ACL Unterstützung wird daher erst nach dem initialen mount des root Dateisystem durch einen entsprechenden remount aktiviert.

Die Ausführbarkeit von Dateien mit ACLs wurden speziell gekennzeichnet. Für gewöhnliche Benutzer wird ganz normal eine Überprüfung der Zugriffsrechte vorgenommen. Dies hat für den Superuser keinen Sinn, da dem Superuser sowieso immer alle Rechte gewährt werden. Also wären alle Dateien für den Superuser ausführbar. Die POSIX Amendments schränkt daher ein, daß nur Dateien mit ACLs, die mindestens einen ACL Eintrag mit gesetzten execute Bit haben, vom Superuser ausführbar sein sollen. Dies entspricht der Behandlung nach POSIX von Dateien ohne ACLs: Nur wenn mindestens eine Klasse der Standard Zugriffsrechte das execute Bit gesetzt haben, wird dem Superuser die Ausführung einer Datei gestattet. Um diese Überprüfung zu beschleunigen, wird allen geladenen Inodes, deren ACL die Ausführbarkeit durch den Superuser gestatten, ein spezielles Flag gesetzt. Der Zeitgewinn ist aber in einem echten System wahrscheinlich nur gering und die Lösung ist sehr unschön, weil sie anfällig für Programmierfehler ist.

Beim Fortgang der Implementierung bemerkten wir, daß die Dokumentation eines Feature großen Einfluß auf die Annahme dieses Feature hat. Abgemildert wird dies wohl dadurch, daß viele Linux-Benutzer Systemadministratoren oder Entwickler sind. Die Implementierung befindet sich nunmehr in der Beta Phase. Zusammengefaßt hat die Implementierung folgende Eigenschaften: Die Implementierung . . .

- bewirkt eine nur unmerkliche Performanzeinbuße des EXT2. Diese erreicht auch bei extremen Einsatz von ACLs nur maximal ca. 5%.
- ist abwärtskompatibel: Ein EXT2-Dateisystem ohne ACLs kann von einem Linux-Kernel mit ACL-Unterstützung benutzt werden.
- ist mit Ausnahme des Programms `e2fsck` aufwärtskompatibel: Ein EXT2-Dateisystem mit ACLs kann von einem Linux-Kernel mit ACL-Unterstützung benutzt werden.
- läuft nach bisherigen Erfahrungen stabil.
- ist zur Zeit nur für Intel Architektur verfügbar.
- unterstüzt ACLs mit einer Länge von bis zu 30 Einträgen.

16.4.2 Benutzer-Programme

Die Implementierung des Projekts wurde erst einem manuellen, dann einem automatisierten Test unterworfen. Beim automatisierten Test erhielten wir zunächst eine Vielzahl von Fehlern. Die Ursache lag darin, daß das Test-Orakel von Matthias Riese in ASpecT geschrieben worden war und die Implementierung von Hauke Steenbock in C. Dabei zeigte sich ganz deutlich, daß natürlichsprachliche Spezifikation niemals eindeutig oder wenigstens unmißverständlich sein können. Auch die großen konzeptionellen Unterschiede beider Sprachen trugen sicherlich dazu bei, daß potentielle Mißdeutungen auch zum Tragen kamen. Bei zwei Implementierungen in derselben Sprache, die die Entwickler der Spezifikation im Hinterkopf hatten, C, wäre dies weniger wahrscheinlich gewesen.

In einer Art Spiralmodell verbesserten wir daher nach und nach sowohl das Test Orakel als auch die Implementierung. Wir fanden in der Implementierung hauptsächlich Fehler in der Abarbeitung der Kommandozeile. Die eigentlichen Algorithmen waren meist fehlerfrei. Schon eine Kommandozeilen Schnittstelle muß, um benutzbar zu sein, viele Möglichkeiten der Bedienung zulassen. Das mag der Grund dieses Mißverhältnis der Zahl der Fehler sein.

Wie ein Fehlerbericht eines Benutzers zeigte, gab es jedoch Möglichkeiten die C Implementierung durch eine teilweise ungültige Kommandozeile zum Absturz zu bringen. Leider muß die Implementierung nach POSIX den validen Teil der Kommandozeile ohne Absturz ausführen. Der Fehler wurde jedoch schnellstmöglich korrigiert.

17. Spezifikation der Zugriffskontrolle

Die folgende Spezifikation beschreibt die Zugriffskontrolle durch ACLs entsprechend POSIX.1e und POSIX.2c Draft 17. Die Spezifikation ist in Z angegeben. Da ein vollständiger Test oder eine vollständige Formalisierung den Umfang eines studentischen Projektes übersteigen würde, sahen wir von der Spezifikation oder sogar Verifikation des gesamten EXT2 ab. Statt dessen beschränken wir uns darauf, wichtige Elemente des EXT2, die für die ACL Unterstützung hinzukommen oder sich ändern, zu spezifizieren und verifizieren.

17.1 Datentypen

17.1.1 ACL

Zunächst wird eine ACL wie folgt von POSIX definiert:

Eine **ACL** ist eine Einheit zur diskreten Zugriffskontrolle verbunden mit einem Objekt. Eine ACL besteht aus Einträgen, wobei jeder Eintrag aus der Bezeichnung eines Benutzers oder eine Gruppe von Benutzern und einer Menge von Zugriffsrechten besteht.

Diskrete Zugriffskontrolle (DAC - Discretionary Access Control) ist ein Mittel, den Zugriff auf ein Objekt basierend auf der Identität eines Benutzers, eines Prozesses und/oder Gruppen, zu denen das Objekt gehört, einzuschränken. Die Kontrolle ist insofern diskret, als daß es einem Subjekt mit gewissen Zugriffsrechten möglich ist, diese Rechte u. U. indirekt an andere Subjekte weiterzugeben.

Nicht diskret ist beispielsweise der Rechnerzugang mit Name und Paßwort: Hier kann man Rechte nur weitergeben, indem man die Authentifikation „täuscht". Ebenfalls ist eine abgeschlossene Tür vor dem Rechnerraum nicht diskret.

Benutzer und Gruppen von Benutzern werden unter UNIX durch natürliche Zahlen bezeichnet. Je nach Darstellung werden diese wegen Überlauf vorzeichenbehafteter Variablen unter Linux auch teilweise als negative Zahlen dargestellt. Da auf diesen IDs jedoch keine Arithmetik ausgeführt wird, schadet diese Inkonsistenz nicht. Benutzer- (UID) und Gruppen ID (GID) werden durch den Z-Typ *ID* repräsentiert:

$$ID == \mathbb{Z}$$

Unter Linux werden die folgenden Zugriffsrechte für Dateien unterschieden:

- Lesen der Datei (<u>r</u>ead).
- Schreiben in die Datei (<u>w</u>rite).
- Ausführen der Datei (e<u>x</u>ecute).

POSIX läßt zwar weitere Zugriffsrechte in einer ACL Implementierung zu, jedoch wird davon in der Implementierung kein Gebrauch gemacht. Die Bedeutung dieser Rechte ist für reguläre Dateien und Verzeichnisse unterschiedlich, die Semantik ist hier jedoch unerheblich. Diese Rechte werden durch den Typ $PERMISSION$ repräsentiert:

$$PERMISSION ::= r \mid w \mid x$$

Folgenden Typ kann ein ACL Eintrag haben:

acl_user_obj Die UID des Prozesses ist identisch mit derjenigen der Datei. Dies entspricht den user Rechten.

acl_user Die UID des Prozesses ist identisch mit der angegebenen UID.

acl_mask Bestimmt die maximalen Rechte, die einem Prozeß durch acl_group_obj- und acl_group- Einträge gewährt werden. Nur die Rechte der acl_group_obj- und acl_group- Einträge werden gewährt, die auch durch den acl_mask- Eintrag gewährt werden.

acl_group_obj Die GID der Datei ist identisch mit einer der zusätzlichen Gruppen des Prozesses. Dies entspricht den group Rechten.

acl_group Die angegebene GID ist identisch mit einer der zusätzlichen Gruppen des Prozesses.

acl_other_obj Prozesse, auf die die beiden obigen Bedingungen nicht zutreffen. Dies entspricht den other Rechten.

Der entsprechende Z-Typ ist $ACLE\_TYPE$:

$$\begin{array}{l} ACLE\_TYPE ::= \\ \quad acl\_user\_obj \mid \\ \quad acl\_group\_obj \mid \\ \quad acl\_other\_obj \mid \\ \quad acl\_mask \mid \\ \quad acl\_user \mid \\ \quad acl\_group \end{array}$$

Ein ACL Eintrag ist nun aus folgenden Komponenten aufgebaut:

$acle\_type$ ist der Typ des Eintrags.

$acle\_tag$ ist die UID oder GID, auf die sich der Eintrag bezieht.

$acle\_perms$ gibt die Rechte an, die dieser Eintrag gewährt.

$$\begin{array}{|l} \hline ACLENTRY \\ \hline acle\_type : ACLE\_TYPE \\ acle\_tag : ID \\ acle\_perms : \mathbb{P}\, PERMISSION \\ \hline \end{array}$$

Die Benennung der Komponenten wurden aus der Struktur $linux\_acl\_entry$ übernommen. Eine ACL ist also eine Sequenz von ACL-Einträgen:

$$ACL == \mathrm{seq}\, ACLENTRY$$

17.1.2 Prozeß

Der Begriff des **Subjekts** ist wie folgt definiert:

Ein **Subjekt** ist eine aktive Einheit, die den Informationsfluß zwischen Objekten verursacht oder den Systemzustand verändert; dies kann z.B. ein Prozeß sein, der von einem Benutzer gestartet worden ist. Als Benutzer wird jede Person bezeichnet, die mit dem Computersystem interagiert.

Wie auch POSIX konstatiert, sind die einzigen sichtbaren **Subjekte** die Prozesse. Die für die Zugriffskontrolle erheblichen Attribute eines Prozesses sind die Benutzer Identifikation (UID), die Gruppen Identifikation (GID) und einer Reihe von zusätzlichen Gruppen (supplementary groups). Meist ist die GID des Prozesses auch in der Liste zusätzlicher Gruppen enthalten. Doch der Linux-Kernel erlaubt auch, beides unabhängig voneinander zu setzen.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{PROCESS}} \\
\mathit{fsuid} : \mathit{ID} \\
\mathit{fsgid} : \mathit{ID} \\
\mathit{groups} : \mathbb{P}\,\mathit{ID} \\
\hline
\end{array}
$$

Die Benennung der Komponenten ist aus der Struktur $task\_struct$ übernommen. Die hier betrachteten IDs eines Prozesses ($fsuid$ und $fsgid$) sind die für das Dateisystem gültigen IDs. Diese sind unter Linux u. U. verschieden von den effektiven IDs ($euid$ und $egid$). Diese können wiederum verschieden von den echten (real) IDs (uid und gid) sein. Initial werden sie von der Login Shell auf die IDs des sich einloggenden Benutzers gesetzt. Sämtliche IDs werden beim Erzeugen eines neuen Prozesses von dem Vater- an den Kindprozeß vererbt. Daher sind die IDs eines Prozesses im allgemeinen identisch mit denen des Benutzers, der den Prozeß gestartet hat. Diese IDs können auf verschiedenste Weisen geändert werden. Die dabei verwendeten Sicherheitsmechanismen sollen jedoch nicht Inhalt dieses Projektes sein. Prozesse mit einer bestimmten UID und der Benutzer mit der gleichen UID werden im folgenden synonym verwendet.

Für den Superuser entfallen alle genannten Einschränkungen und Zugriffskontrollen. Unter anderem kann der Superuser die genannten IDs frei wählen. Dies ist notwendig, damit der Superuser ein defektes System reparieren oder ein neues System aufsetzen kann.

17.1.3 Datei

Ziel eines Zugriffs ist ein **Objekt**:

Ein **Objekt** ist hier eine passive Einheit, die Daten trägt oder empfängt. Der Zugriff auf ein Objekt impliziert potentiell den Zugriff auf die Daten des Objektes.

Wird beispielsweise unter Linux einem Benutzer gestattet, eine Datei zum Lesen bzw. zum Schreiben zu öffnen, kann er ohne weitere Kontrolle den Inhalt der Datei lesen bzw. schreiben. Dem Begriff des Objekts entspricht ein Inode des EXT2: Jedes Objekt des Dateisystems wird durch einen Inode repräsentiert, über den auch der Zugriff geregelt wird. Der Zugriff auf Daten, z. B. den Inhalt einer Datei, ist normalen Benutzern nur über das

Öffnen des Inode möglich. Inodes haben folgende Attribute, die den Zugriff regeln:

Besitzer Diese UID wird beim Anlegen der Datei mit derjenigen des anlegenden Prozesses gesetzt. Nur der Superuser kann den Besitzer einer Datei ändern.

Gruppe Diese GID wird beim Anlegen der Datei mit der primären GID des Prozesses gesetzt. Nur der Besitzer oder Superuser kann die Gruppe einer Datei ändern. Die GID kann auf eine beliebige andere Gruppe des Prozesses gesetzt werden. Der Superuser kann eine beliebige Gruppe setzen.

Rechte Die Standard-Zugriffsrechte. Nur der Besitzer oder der Superuser kann die Zugriffsrechte ändern.

ACL Die ACL steht zwar nicht direkt im Inode im EXT2, ist aber eindeutig dem Inode zugeordnet.

Dies ergibt das folgendes Schema:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{INODE}} \\
\mathit{i_uid} : \mathit{ID} \\
\mathit{i_gid} : \mathit{ID} \\
\mathit{uperms} : \mathbb{P}\,\mathit{PERMISSION} \\
\mathit{gperms} : \mathbb{P}\,\mathit{PERMISSION} \\
\mathit{operms} : \mathbb{P}\,\mathit{PERMISSION} \\
\mathit{i_file_acl} : \mathit{ACL} \\
\hline
\end{array}$$

Die Benennung der ersten beiden Komponenten ist aus der Struktur inode übernommen. Die Benennung der ACL wurde aus der Struktur $\mathit{ext2\_inode\_info}$ übernommen.

17.2 Operationen

17.2.1 Standard Zugriffskontrolle

Durch die Standard-UNIX-Zugriffsrechte von Dateien werden Prozesse in drei Gruppen unterteilt:

Kategorie	Bedingung
user	Die UID des Prozesses ist identisch mit derjenigen der Datei.
group	Die GID der Datei ist identisch mit der Gruppe des Prozesses oder einer der zusätzlichen Gruppen des Prozesses.
other	Prozesse, die nicht in die vorigen Kategorien fallen.

Jeder Kategorie kann eine beliebige Kombination von Rechten gewährt werden. Einem Prozeß werden durch die Standardzugriffsrechte jeweils die Rechte für seine Kategorie gewährt.

Das folgende Schema drückt aus, ob die Standard-UNIX-Zugriffsrechte einer Datei $\mathit{inode}?$ dem aktuellen Prozeß $\mathit{current}?$ eine Menge von Zugriffsrechten $\mathit{mask}?$ gewähren. Das Ergebnis $\mathit{result}!$ ist vom Typ

$$\mathit{ERRNO} ::= \mathit{deny} \mid \mathit{success}$$

$deny$ entspricht dem Fehlercode `EACCESS`. $success$ entspricht `0`, sprich „kein Fehler“. Dieses Schema spezifiziert das Verhalten der gleichnamigen Funktion im EXT2 ohne ACL Unterstützung.

$$
\begin{array}{l}
\textit{ext2\_permission} \\
\hline
current? : PROCESS \\
inode? : INODE \\
mask? : \mathbb{P}\, PERMISSION \\
result! : ERRNO \\
\hline
result! = \\
\mathbf{if}\ inode?.i\_uid = current?.fsuid\ \mathbf{then} \\
\quad (\mathbf{if}\ mask? \subseteq inode?.uperms\ \mathbf{then}\ success\ \mathbf{else}\ deny) \\
\mathbf{else\ if}\ inode?.i\_gid \in current?.groups \cup \{current?.fsgid\}\ \mathbf{then} \\
\quad (\mathbf{if}\ mask? \subseteq inode?.gperms\ \mathbf{then}\ success\ \mathbf{else}\ deny) \\
\mathbf{else} \\
\quad (\mathbf{if}\ mask? \subseteq inode?.operms\ \mathbf{then}\ success\ \mathbf{else}\ deny)
\end{array}
$$

17.2.2 ACL-Validation

Eine valide ACL muß nach POSIX folgende Bedingungen erfüllen:

- Einträge des Typs acl_user_obj, acl_group_obj, acl_other_obj müssen genau einmal vorhanden sein.
- Jedes Paar aus Typ und Tag darf nur einmal vorkommen.
- Wenn eine ACL Einträge vom Typ acl_user oder acl_group enthält, muß sie genau einen Eintrag vom Typ acl_mask enthalten.

Die Implementierung legt darüberhinaus fest, daß die Einträge einer validen ACL immer in der Reihenfolge acl_user_obj, acl_mask, acl_user, acl_group_obj, acl_group, acl_other_obj sortiert sind. Diese Sortierung gilt jedoch nur für die Schnittstelle zum Kernel und im Kernel. Die ACL-Bibliothek kann mit beliebig geordneten ACLs umgehen. Erst bei der Übergabe an den Kernel stellt sie die korrekte Sortierung her.

Die folgende Sequenz vom Typ $\mathrm{seq}\, ACLE$ gibt die Reihenfolge wieder:

$$ato == \langle acl\_user\_obj, acl\_mask, acl\_user, acl\_group\_obj, acl\_group, acl\_other\_obj \rangle$$

Damit läßt sich die folgende Ordnungsrelation definieren:

$$
\begin{array}{l}
\_ \leq_{ACL} \_ : ACLE\_TYPE \leftrightarrow ACLE\_TYPE \\
\hline
\forall\, at1, at2 : \\
\quad ACLE\_TYPE \bullet (at1 \leq_{ACL} at2) \Leftrightarrow \\
\quad (\exists\, i, j : \mathrm{dom}\, ato \bullet (ato\ i) = at1 \wedge (ato\ j) = at2 \wedge i \leq j)
\end{array}
$$

Das folgende Schema prüft die Bedingungen laut POSIX und die Sortierung. Die ACL, die als Eingabe dient, wird genau dann als valide bezeichnet, wenn das Schema erfüllt ist.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{check\_acl}} \\
acl? : ACL \\
\hline
\exists_1\, e : \operatorname{ran} acl? \mid e.type = acl\_user\_obj \\
\exists_1\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \\
\exists_1\, e : \operatorname{ran} acl? \mid e.type = acl\_other\_obj \\
\forall\, i, j : \operatorname{dom} acl? \bullet i < j \Rightarrow (acl?\, i).type \leq_{ACL} (acl?\, j).type \\
\forall\, i, j : \operatorname{dom} acl? \bullet \\
\quad ((acl?\, i).type = (acl?\, j).type \land (acl?\, i).tag = (acl?\, j).tag) \\
\quad \Rightarrow i = j \\
\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_user \lor e.type = acl\_group \bullet \\
\quad \exists_1\, e : \operatorname{ran} acl? \mid e.type = acl\_mask \\
\hline
\end{array}
$$

Das Schema entspricht der gleichnamigen Funktion check_acl in der Implementierung. Diese Funktion testet jede ACL, die für eine Datei gesetzt werden sollen. Stellt die Funktion fest, daß die ACL nicht valide ist, wird sie nicht gesetzt. Der Korrektheit dieser Funktion haben wir uns durch Code-Inspektion versichert. Daher können wir vorraussetzen, daß die ACL einer Datei immer valide ist, und die Eigenschaften einer validen ACL als Invarianten für die Beweisführung voraussetzen.

17.2.3 ACL Zugriffskontrolle

Die Funktion `ext2_acl_permission` des EXT2 testet, ob die ACL eines EXT2 Inodes dem aktuellen Prozeß gewisse Zugriffsrechte auf den Inode gewährt. Der Prototyp der Funktion ist

```
int ext2_acl_permission (struct inode *inode,
  struct ext2_acl_entry *acle, int n, int mask)
```

Das gleichnamige Z-Schema beschreibt das laut POSIX korrekte Verhalten dieser Funktion. Die Komponenten des Schemas entsprechen weitgehend den Parametern von `ext2_acl_permission`.

$current?$ Dies entspricht dem globalen Zeiger `current` auf die `task_struct` des aktuellen Prozesses. Genaugenommen ist `current` ein unter den verschiedenen Architekturen mehr oder weniger kompliziertes Assembler Konstrukt.

$inode?$ Der Inode, dessen ACL daraufhin überprüft werden soll, ob die verlangten Zugriffsrechte gewährt werden (`inode`). Die ACL selbst ist hier eine Komponente von $inode?$. Auch in der Implementierung könnte die ACL dem Inode entnommen werden, wird jedoch aufgrund technischer Details separat übergeben: als Zeiger auf die Einträge (`acle`) und der Anzahl der Einträge (`n`).

$mask?$ Die verlangten Zugriffsrechte als Menge von $PERMISSION$ entsprechend der Bit-Maske `mask`.

$result!$ Dieser Komponente wird das Ergebnis zugewiesen entsprechend dem Rückgabewert der Funktion.

$matched$ Diese Hilfsfunktion dient lediglich der Vereinfachung der Spezifikation.

$$
\begin{array}{l}
\textit{ext2\_acl\_permission} \\
\hline
current? : PROCESS \\
inode? : INODE \\
mask? : \mathbb{P}\, PERMISSION \\
acl? : ACL \\
result! : ERRNO \\
matched : ACLENTRY \rightarrow ERRNO \\
\hline
matched(entry) = \textbf{if } mask? \subseteq entry.acle\_perms \textbf{ then} \\
\quad \textbf{if } (entry.type = acl\_user\_obj \lor entry.type = acl\_other) \textbf{ then} \\
\quad\quad success \\
\quad \textbf{else if } (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_mask) \textbf{ then} \\
\quad\quad \textbf{if } (mask? \subseteq (\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then} \\
\quad\quad\quad success \\
\quad\quad \textbf{else } deny \\
\quad \textbf{else } success \\
\textbf{else } deny \\
\\
result! = \textbf{if } (current?.fsuid = inode?.i\_uid) \textbf{ then} \\
\quad matched(\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_user\_obj) \\
\textbf{else if } (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_user \land e.tag = current?.fsuid) \textbf{ then} \\
\quad matched(\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_user \land e.tag = current?.fsuid) \\
\textbf{else if } (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in \\
\quad\quad \{current?.fsgid\} \cup current?.groups \lor \\
\quad\quad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups) \textbf{ then} \\
\quad \textbf{if } (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in \\
\quad\quad\quad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor \\
\quad\quad\quad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land \\
\quad\quad\quad mask? \subseteq e.acle\_perms) \textbf{ then} \\
\quad\quad matched(\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in \\
\quad\quad\quad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor \\
\quad\quad\quad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land \\
\quad\quad\quad mask? \subseteq e.acle\_perms) \\
\quad \textbf{else} \\
\quad\quad deny \\
\textbf{else if } (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_other\_obj \land mask? \subseteq e.acle\_perms) \textbf{ then} \\
\quad success \\
\textbf{else} \\
\quad deny \\
\hline
\end{array}
$$

Im nächsten Kapitel werden wir verifizieren, daß eine C Funktion diese Spezifikation erfüllt. Wir versuchen nun schrittweise, die Spezifikation an den Code dieser Funktion anzunähern. Als erstes lösen wir durch Einsetzung die Hilfsfunktion *matched* auf.

$$
\begin{array}{l}
\textit{ext2\_acl\_permission}.1 \\
\hline
\textit{current}? : \textit{PROCESS} \\
\textit{inode}? : \textit{INODE} \\
\textit{mask}? : \mathbb{P}\,\textit{PERMISSION} \\
\textit{result}! : \textit{ERRNO} \\
\hline
\textit{result}! = \textbf{if } (\textit{current}?.\textit{fsuid} = \textit{inode}?.\textit{i\_uid}) \textbf{ then} \\
\quad \textbf{if } \textit{mask}? \subseteq (\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user\_obj}).\textit{acle\_perms} \textbf{ then} \\
\quad\quad \textbf{if } ((\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user\_obj}).\textit{type} = \textit{acl\_user\_obj} \lor \\
\quad\quad\quad (\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user\_obj}).\textit{type} = \textit{acl\_other}) \textbf{ then} \\
\quad\quad\quad \textit{success} \\
\quad\quad \textbf{else if } (\exists\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_mask}) \textbf{ then} \\
\quad\quad\quad \textbf{if } (\textit{mask}? \subseteq (\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_mask}).\textit{acle\_perms}) \textbf{ then} \\
\quad\quad\quad\quad \textit{success} \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textit{deny} \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textit{success} \\
\quad \textbf{else} \\
\quad\quad \textit{deny} \\
\textbf{else if } (\exists\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user} \land e.\textit{tag} = \textit{current}?.\textit{fsuid}) \textbf{ then} \\
\quad \textbf{if } \textit{mask}? \subseteq (\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user} \land \\
\quad\quad e.\textit{tag} = \textit{current}?.\textit{fsuid}).\textit{acle\_perms} \textbf{ then} \\
\quad\quad \textbf{if } ((\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user} \land \\
\quad\quad\quad e.\textit{tag} = \textit{current}?.\textit{fsuid}).\textit{type} = \textit{acl\_user\_obj} \lor \\
\quad\quad\quad (\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_user} \land \\
\quad\quad\quad\quad\quad e.\textit{tag} = \textit{current}?.\textit{fsuid}).\textit{type} = \textit{acl\_other}) \textbf{ then} \\
\quad\quad\quad \textit{success} \\
\quad\quad \textbf{else if } (\exists\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_mask}) \textbf{ then} \\
\quad\quad\quad \textbf{if } (\textit{mask}? \subseteq (\mu\, e : \operatorname{ran} \textit{acl}? \mid e.\textit{type} = \textit{acl\_mask}).\textit{acle\_perms}) \textbf{ then} \\
\quad\quad\quad\quad \textit{success} \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textit{deny} \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textit{success} \\
\quad \textbf{else} \\
\quad\quad \textit{deny}
\end{array}
$$

$\mathbf{else\ if}\ (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in$
$\quad \{current?.fsgid\} \cup current?.groups \lor$
$\quad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups)\ \mathbf{then}$
$\quad \mathbf{if}\ (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in$
$\qquad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor$
$\qquad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land$
$\qquad mask? \subseteq e.acle\_perms)\ \mathbf{then}$
$\qquad \mathbf{if}\ mask? \subseteq (\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in$
$\qquad\quad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor$
$\qquad\quad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land$
$\qquad\quad mask? \subseteq e.acle\_perms).acle\_perms\ \mathbf{then}$
$\qquad\quad \mathbf{if}\ ((\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in$
$\qquad\qquad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor$
$\qquad\qquad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land$
$\qquad\qquad mask? \subseteq e.acle\_perms).type = acl\_user\_obj \lor$
$\qquad\qquad (\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in$
$\qquad\qquad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor$
$\qquad\qquad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land$
$\qquad\qquad mask? \subseteq e.acle\_perms).type = acl\_other)\ \mathbf{then}$
$\qquad\qquad success$
$\qquad\quad \mathbf{else\ if}\ (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_mask)\ \mathbf{then}$
$\qquad\qquad \mathbf{if}\ (mask? \subseteq (\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_mask).acle\_perms)\ \mathbf{then}$
$\qquad\qquad\quad success$
$\qquad\qquad \mathbf{else}\ deny$
$\qquad\quad \mathbf{else}\ success$
$\qquad \mathbf{else}\ deny$
$\quad \mathbf{else}$
$\qquad deny$
$\mathbf{else\ if}\ (\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_other\_obj \land$
$\quad mask? \subseteq e.acle\_perms)\ \mathbf{then}$
$\quad success$
$\mathbf{else}$
$\quad deny$

Die entstehenden Ausdrücke lassen sich stark vereinfachen:

$ext2\_acl\_permission.2$

$current? : PROCESS$
$inode? : INODE$
$mask? : \mathbb{P}\, PERMISSION$
$result! : ERRNO$

$result! = \mathbf{if}\ (current?.fsuid = inode?.i\_uid)\ \mathbf{then}$
$\quad \mathbf{if}\ mask? \subseteq (\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_user\_obj).acle\_perms\ \mathbf{then}$
$\qquad success$
$\quad \mathbf{else}$
$\qquad deny$

$$
\begin{array}{l}
\textbf{else if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land e.tag = current?.fsuid) \textbf{ then} \\
\quad \textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land \\
\qquad e.tag = current?.fsuid).acle\_perms) \textbf{ then} \\
\qquad \textbf{if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_mask) \textbf{ then} \\
\qquad\quad \textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then} \\
\qquad\qquad success \\
\qquad\quad \textbf{else} \\
\qquad\qquad deny \\
\qquad \textbf{else} \\
\qquad\quad success \\
\quad \textbf{else} \\
\qquad deny \\
\textbf{else if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in \\
\qquad \{current?.fsgid\} \cup current?.groups \lor \\
\qquad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups) \textbf{ then} \\
\quad \textbf{if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in \\
\qquad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor \\
\qquad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land \\
\qquad mask? \subseteq e.acle\_perms) \textbf{ then} \\
\qquad \textbf{if } mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in \\
\qquad\quad \{current?.fsgid\} \cup current?.groups \land mask? \subseteq e.perms \lor \\
\qquad\quad e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups \land \\
\qquad\quad mask? \subseteq e.acle\_perms).acle\_perms \textbf{ then} \\
\qquad\quad \textbf{if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_mask) \textbf{ then} \\
\qquad\qquad \textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then} \\
\qquad\qquad\quad success \\
\qquad\qquad \textbf{else } deny \\
\qquad\quad \textbf{else } success \\
\qquad \textbf{else } deny \\
\quad \textbf{else} \\
\qquad deny \\
\textbf{else if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_other\_obj \land \\
\qquad mask? \subseteq e.acle\_perms) \textbf{ then} \\
\quad success \\
\textbf{else} \\
\quad deny
\end{array}
$$

Wir wollen nun noch den Ausdruck in der zweiten Fallunterscheidung umformen, nämlich:

$$
\begin{array}{l}
\textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land \\
\qquad e.tag = current?.fsuid).acle\_perms) \textbf{ then} \\
\quad \textbf{if } (\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_mask) \textbf{ then} \\
\qquad \textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then} \\
\qquad\quad success \\
\qquad \textbf{else} \\
\qquad\quad deny \\
\quad \textbf{else} \\
\qquad success
\end{array}
$$

Für den Fall

$$\exists e : \mathrm{ran}\, acl? \mid e.type = acl\_mask$$

gelangen wir zu $success$ gdw. der Ausdruck

$$(mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$$
$$e.tag = current?.fsuid).acle\_perms) \land$$
$$(mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms)$$

erfüllt ist. Diesen Ausdruck können wir umformen in

$$(mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$$
$$e.tag = current?.fsuid).acle\_perms) \cap$$
$$(\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms).$$

Gilt hingegen

$$\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask$$

so gelangen wir zu $success$ gdw. gilt

$$mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$$
$$e.tag = current?.fsuid).acle\_perms.$$

Ansonsten gelangen wir zu $deny$. Die Menge

$$(\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land e.tag = current?.fsuid).acle\_perms$$

ist per Definition eine Teilmenge von $\{r, w, x\}$. Wir verändern den Ausdruck also nicht, wenn wir die Mengen schneiden. Wir erhalten:

$$mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$$
$$e.tag = current?.fsuid).acle\_perms \cap \{r, w, x\}.$$

Wir können den Abschnitt der Spezifikation also als

$\textbf{if } (\exists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask) \textbf{ then}$
$\quad \textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$
$\qquad e.tag = current?.fsuid).acle\_perms) \cap$
$\qquad (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then}$
$\quad\; sucess$
$\quad \textbf{else}$
$\quad\; deny$
$\textbf{else}$
$\quad \textbf{if } (mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$
$\qquad e.tag = current?.fsuid).acle\_perms \cap \{r, w, x\}) \textbf{ then}$
$\quad\; success$
$\quad \textbf{else}$
$\quad\; deny$

schreiben. Durch umgekehrte Schachtelung erhalten wir einen gleichwertigen Ausdruck:

$\quad \textbf{if } mask? \subseteq (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land$
$\qquad e.tag = current?.fsuid).acle\_perms \cap ($
$\qquad\; \textbf{if } (\exists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask) \textbf{ then}$
$\qquad\;\; (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then}$
$\qquad\; \textbf{else}$
$\qquad\;\; \{r, w, x\})$
$\quad\; success$
$\quad \textbf{else}$
$\quad\; deny$

Auf analoge Weise können wir auch die dritte Fallunterscheidung umschreiben und erhalten insgesamt:

$\mathit{ext2\_acl\_permission.3}$

$$
\begin{array}{l}
\mathit{current?} : \mathit{PROCESS} \\
\mathit{inode?} : \mathit{INODE} \\
\mathit{mask?} : \mathbb{P}\,\mathit{PERMISSION} \\
\mathit{result!} : \mathit{ERRNO}
\end{array}
$$

$$
\begin{array}{l}
\mathit{result!} = \mathbf{if}\ (\mathit{current?.fsuid} = \mathit{inode?.i\_uid})\ \mathbf{then} \\
\quad \mathbf{if}\ \mathit{mask?} \subseteq (\mu\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_user\_obj}).\mathit{acle\_perms}\ \mathbf{then} \\
\qquad \mathit{success} \\
\quad \mathbf{else} \\
\qquad \mathit{deny} \\
\mathbf{else\ if}\ (\exists\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_user} \land e.\mathit{tag} = \mathit{current?.fsuid})\ \mathbf{then} \\
\quad \mathbf{if}\ \mathit{mask?} \subseteq (\mu\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_user} \land \\
\qquad\quad e.\mathit{tag} = \mathit{current?.fsuid}).\mathit{acle\_perms} \cap (\\
\qquad\quad \mathbf{if}\ (\exists\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_mask})\ \mathbf{then} \\
\qquad\qquad (\mu\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_mask}).\mathit{acle\_perms}) \\
\qquad\quad \mathbf{else}\ \{r, w, x\})\ \mathbf{then} \\
\qquad \mathit{success} \\
\quad \mathbf{else} \\
\qquad \mathit{deny} \\
\mathbf{else\ if}\ (\exists\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_group\_obj} \land \mathit{inode?.i\_gid} \in \\
\qquad \{\mathit{current?.fsgid}\} \cup \mathit{current?.groups} \lor \\
\qquad e.\mathit{type} = \mathit{acl\_group} \land e.\mathit{tag} \in \{\mathit{current?.fsgid}\} \cup \mathit{current?.groups})\ \mathbf{then} \\
\quad \mathbf{if}\ (\exists\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_group\_obj} \land \mathit{inode?.i\_gid} \in \\
\qquad\quad \{\mathit{current?.fsgid}\} \cup \mathit{current?.groups} \land \mathit{mask?} \subseteq e.\mathit{perms} \lor \\
\qquad\quad e.\mathit{type} = \mathit{acl\_group} \land e.\mathit{tag} \in \{\mathit{current?.fsgid}\} \cup \mathit{current?.groups} \land \\
\qquad\quad \mathit{mask?} \subseteq e.\mathit{acle\_perms} \cap (\\
\qquad\quad \mathbf{if}\ (\exists\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_mask})\ \mathbf{then} \\
\qquad\qquad (\mu\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_mask}).\mathit{acle\_perms}) \\
\qquad\quad \mathbf{else}\ \{r, w, x\})\ \mathbf{then} \\
\qquad \mathit{success} \\
\quad \mathbf{else} \\
\qquad \mathit{deny} \\
\mathbf{else\ if}\ (\exists\, e : \operatorname{ran} \mathit{acl?} \mid e.\mathit{type} = \mathit{acl\_other\_obj} \land \\
\qquad \mathit{mask?} \subseteq e.\mathit{acle\_perms})\ \mathbf{then} \\
\quad \mathit{success} \\
\mathbf{else} \\
\quad \mathit{deny}
\end{array}
$$

18. Verifikation der Zugriffskontrolle

Wir werden nun die Funktion `ext2_acl_permission` verifizieren, d. h. beweisen, daß sich die Funktion entsprechend der Z-Spezifikation korrekt verhält. Dabei benutzen wir das bei [AO91] vorgestellte Beweissystem für eine prozedurale Sprache. Für die Annotation verwenden wir das mathematische Werkzeug von Z.

Leider läßt sich das Beweissystem bei [AO91] nicht auf C übertragen, vielmehr muß der C Code auf die Sprache des Beweissystem abgebildet werden. Bei dieser Abbildung können wir nicht streng formal vorgehen, da schon die Semantik von C nicht formal definiert ist. Wir können lediglich eine Abbildung finden und validieren, daß diese Abbildung geeignet ist. Dabei bedeutet geeignet, daß Aussagen, die sich für das Bild der C Funktion beweisen lassen, in entsprechender Weise für die C Funktion Geltung haben.

Bei der Validation werden wir uns wiederum Z bedienen. Wir werden Abbildungen von verschiedenen Teilen des C Codes suchen und aufgrund der gewonnenen Ergebnisse schließlich das vollständige Bild der C Funktion zusammensetzen.

Der C Code von `ext2_acl_permission` lautet:

```
static
int ext2_acl_permission(struct inode *inode,
                        struct ext2_acl_entry
                        *acle, int n, int mask)
{
  struct ext2_acl_entry *aclp;
  int in_group = 0;
  int i;
  int acl_mask = S_IRWXO;

  for (aclp = acle, i = 0; aclp != NULL && i < n;
       aclp++, i++)
    switch (le16_to_cpu(aclp->acle_type)) {
    case ACL_USER_OBJ:
      if (current->fsuid == inode->i_uid) {
        if ((le16_to_cpu(aclp->acle_perms) & mask
             & S_IRWXO) == mask)
          return 0;
        else
          return -EACCES;
      }
      break;

    case ACL_MASK:
      acl_mask = le16_to_cpu(aclp->acle_perms);
      break;

    case ACL_USER:
      if (current->fsuid ==
          le32_to_cpu(aclp->acle_tag)) {
        if (
```

```
            (le16_to_cpu(aclp->acle_perms) & mask
             & acl_mask & S_IRWXO) == mask)
          return 0;
        else
          return -EACCES;
      }
      break;

    case ACL_GROUP_OBJ:
      if (in_group_p(inode->i_gid)) {
        in_group = 1;
        if (
            (le16_to_cpu(aclp->acle_perms) & mask
             & acl_mask & S_IRWXO) == mask)
          return 0;
      }
      break;

    case ACL_GROUP:
      if (in_group_p(le32_to_cpu(aclp->acle_tag))) {
        in_group = 1;
        if (
            (le16_to_cpu(aclp->acle_perms) & mask
             & acl_mask & S_IRWXO) == mask)
          return 0;
      }
      break;

    case ACL_OTHER_OBJ:
      if (in_group) {
        return -EACCES;
      }
      if ((le16_to_cpu(aclp->acle_perms) & mask
           & S_IRWXO) == mask)
        return 0;
      else
        return -EACCES;
      break;

    default:
      ext2_error(inode->i_sb,
                 "ext2_acl_permission",
                 "Bad ACL type (%d) for inode %s:%ld"
                 " at position %d",
                 le16_to_cpu(aclp->acle_type),
                 kdevname(inode->i_dev),
                 inode->i_ino, i);
      break;
    }

  return -EACCES;
}
```

18.1 Übersetzung des C Code

18.1.1 Kontrollfluß

Wir beginnen die Abbildung in das Beweissystem bei den C Sprachkonstrukten zur Steuerung des Kontrollflusses. Das Konstrukt `for(X;Y;Z) B` führt einmal `X` aus, dann, solange die Bedingung `Y` erfüllt ist, `B`. Nach jeder Ausführung von `B` wird `Z` ausgeführt. Wir können das Konstrukt also übersetzen in:

X;
while Y **do**
  B;
  Z;
od

Eine `switch` Anweisung hat die Form

```
switch X {
  case A1:
    B1
  case A2:
    B2
  ...
  default:
    B
}
```

Wenn `X` gleich `A1`, wird `B1` ausgeführt. Ansonsten, wenn `X` gleich `A2`, wird `B2` ausgeführt usw. Gleiche Werte für zwei `case` sind verboten. Wenn X zu keinem der `An` gleich ist, wird der `default` Zweig ausgeführt. Ein fehlender `default` Teil ist gleichbedeutend mit einem leeren `B`. Folgendes **if then else** Konstrukt ist also gleichbedeutend zu einer `switch` Anweisung:

if $X = A1$ **then** $B1$ **else if** $X = A2$ **then** $B2 \ldots$ **else** B

`return` springt mit einem Rückgabewert aus der Funktion heraus. Um den Rückgabewert in den Beweis einzubeziehen, ersetzen wir die `return` Anweisung durch eine Zuweisung an eine Hilfsvariable namens $return$ ein. $return$ erhält den initialen Wert $init$, der unterscheidbar sei von allen anderen Werten für $return$. Die (zu beweisende) Bedingung für den Rückgabewert wird im Anschluß an die Zuweisung auf der Hilfsvariablen formuliert. Bei der `return` Anweisung terminiert die Funktion. In dem verwendeten Beweissystem läßt sich eine Termination nur dadurch erzielen, daß die Ausführung das Ende des Programms erreicht. Daher schließen wir alle Anweisungen nach einer `return` Anweisung in Kontrollstrukturen ein, die den Wert von $return$ testen. Die Kontrollstrukturen werden so angelegt, daß sie die Funktion nach der Zuweisung an $return$ terminieren lassen, ohne eine weitere Anweisung auszuführen .

$return := init$;
$\ldots$
$return := success$;
if $return = init$ **then**
  $\ldots$
fi

18.1.2 Bitmasken

In der Implementierung werden Zugriffsrechte als Bitmasken dargestellt. Das Operieren mit Bitmasken würde den Beweis unnötig kompliziert gestalten. Stattdessen übersetzen wir Bitmasken in Mengen und Operationen auf Bitmasken in Mengenoperationen.

Die Rechte r, w, x werden durch die $4, 2, 1$ in dieser Reihenfolge repräsentiert. Diese Zahlen werden durch Variablen vom C Typ +short+ kodiert. Die Menge der gültigen Werte für ein +short+ bezeichnen wir als S.

Die Zahlen für die Rechte sowie ihre kombinierte Bitmaske sind in `stat.h` symbolisch definiert:

$$
\begin{array}{|l}
S\_IRWXO, S\_IROTH, S\_IWOTH, S\_IXOTH : S \\
\hline
S\_IRWXO = 7 \\
S\_IROTH = 4 \\
S\_IWOTH = 2 \\
S\_IXOTH = 1
\end{array}
$$

Die Äquivalenz einer Menge von Zugriffsrechten mit einer Bitmaske wird durch die $\widehat{=}$ Relation ausgedrückt:

$$
\begin{array}{|l}
\_ \widehat{=} \_ : \mathbb{P}\, PERMISSION \leftrightarrow S \\
\hline
\forall\, perm : \mathbb{P}\, PERMISSION;\ mask : S \bullet \\
\quad perm \widehat{=} mask \Leftrightarrow (\\
\quad r \in perm \Leftrightarrow mask \,\&\, S\_IROTH \\
\quad w \in perm \Leftrightarrow mask \,\&\, S\_IWOTH \\
\quad x \in perm \Leftrightarrow mask \,\&\, S\_IXOTH)
\end{array}
$$

Wir versuchen nun, C Ausdrücke so auf Mengenoperationen auf $PERMISSION$ abzubilden, daß folgendes erfüllt ist: Stehen die Parameter für die C Operation und die Mengenoperation in $\widehat{=}$ Relation, so sollten auch die jeweiligen Ergebnisse in $\widehat{=}$ Relation stehen.

Wir verwenden im folgenden eine Anzahl von Parametern, die in $\widehat{=}$ Relation stehen:

$$
\begin{array}{|l}
p1, p2, p3 : \mathbb{P}\, PERMISSION \\
m1, m2, m3 : S \\
\hline
p1 \widehat{=} m1 \\
p2 \widehat{=} m2 \\
p3 \widehat{=} m3
\end{array}
$$

Man kann einen Zusammenhang zwischen der Teilmengenbeziehung auf und dem $\&$ Operator (bitweises Verunden) :

$$
\begin{array}{l}
m1 \,\&\, m2 \,\&\, S\_IRWXO = m1 \Leftrightarrow \\
\quad m1 \,\&\, 4 \Rightarrow m2 \,\&\, 4 \wedge \\
\quad m1 \,\&\, 2 \Rightarrow m2 \,\&\, 2 \wedge \\
\quad m1 \,\&\, 1 \Rightarrow m2 \,\&\, 1 \\
\Leftrightarrow \\
\quad r \in p1 \Rightarrow r \in p2 \wedge \\
\quad w \in p1 \Rightarrow w \in p2 \wedge \\
\quad x \in p1 \Rightarrow x \in p2 \\
\Leftrightarrow \\
\forall x : PERMISSION \bullet x \in p1 \Rightarrow x \in p2 \\
\Leftrightarrow \\
p1 \subseteq p2
\end{array}
$$

Der erste Schritt der läßt sich anhand einer Wertetabelle zeigen. Ebenso kann gefolgert werden, daß eine Beziehung zwischen der C Vergleichsoperation und der Gleichheit auf Mengen besteht:

$$
\begin{array}{l}
m1 \,\&\, S\_IRWXO = m2 \,\&\, S\_IRWXO \\
\Leftrightarrow \\
\quad m1 \,\&\, 4 \Leftrightarrow m2 \,\&\, 4 \wedge \\
\quad m1 \,\&\, 2 \Leftrightarrow m2 \,\&\, 2 \wedge \\
\quad m1 \,\&\, 1 \Leftrightarrow m2 \,\&\, 1 \\
\Leftrightarrow \\
\quad r \in p1 \Leftrightarrow r \in p2 \wedge \\
\quad w \in p1 \Leftrightarrow w \in p2 \wedge \\
\quad x \in p1 \Leftrightarrow x \in p2 \\
\Leftrightarrow \\
p1 = p2
\end{array}
$$

Für dem & Operator gilt:

$$\forall x : S \bullet x\&(m1\&m2) \Leftrightarrow x\&m1 \wedge x\&m2$$

Deswegen kann man folgern:

$$
\begin{array}{l}
\quad r \in p1 \wedge r \in p2 \Leftrightarrow S\_IROTH \,\&\, m1 \wedge S\_IROTH \,\&\, m2 \wedge \\
\quad w \in p1 \wedge w \in p2 \Leftrightarrow S\_IWOTH \,\&\, m1 \wedge S\_IWOTH \,\&\, m2 \wedge \\
\quad x \in p1 \wedge x \in p2 \Leftrightarrow S\_IXOTH \,\&\, m1 \wedge S\_IXOTH \,\&\, m2 \\
\Leftrightarrow \\
\quad r \in (p1 \cap p2) \Leftrightarrow S\_IROTH \,\&\, (m1 \& m2) \wedge \\
\quad w \in (p1 \cap p2) \Leftrightarrow S\_IWOTH \,\&\, (m1 \& m2) \wedge \\
\quad x \in (p1 \cap p2) \Leftrightarrow S\_IXOTH \,\&\, (m1 \& m2) \\
\Leftrightarrow \\
p1 \cap p2 \mathrel{\widehat{=}} m1 \,\&\, m2
\end{array}
$$

Daraus folgt dann:

$$
\begin{array}{l}
(m1 \,\&\, m2 \,\&\, m3 \,\&\, S\_IRWXO) = m1 \Leftrightarrow \\
p1 \subseteq (p2 \cap p3) \Leftrightarrow \\
p1 \subseteq p2 \wedge p1 \subseteq p3
\end{array}
$$

Ebenso zeigen wir, daß der Vereinigung zweier Mengen von $PERMISSION$s die Verknüpfung mit $|$ (bitweises Verodern) entspricht:

$$
\begin{array}{|l}
p1 \mathrel{\widehat{=}} m1 \wedge p2 \mathrel{\widehat{=}} m2 \wedge p = p1 \cup 2p2 \Leftrightarrow \\
\forall x : PERMISSION \bullet x \in p1 \vee x \in p2 \Rightarrow x \in p \Leftrightarrow \\
(\\
\quad r \in p1 \vee r \in p2 \Rightarrow r \in p \wedge \\
\quad w \in p1 \vee w \in p2 \Rightarrow w \in p \wedge \\
\quad x \in p1 \vee x \in p2 \Rightarrow x \in p \wedge \\
) \Leftrightarrow \\
(\\
\quad m1 \,\&\, 4 \vee m2 \,\&\, 4 \Rightarrow m \,\&\, 4 \wedge \\
\quad m1 \,\&\, 2 \vee m2 \,\&\, 2 \Rightarrow m \,\&\, 2 \wedge \\
\quad m1 \,\&\, 1 \vee m2 \,\&\, 1 \Rightarrow m \,\&\, 1 \wedge \\
) \Leftrightarrow \\
m = (m1 \,\&\, S\_IRWXO) \mid (m2 \,\&\, S\_IRWXO)
\end{array}
$$

Entsprechende C Ausdrücke auf als Zahlen kodierte Zugriffsrechte lassen sich also unter Einhaltung der eingangs genannten Bedingung auf entsprechende Mengenoperationen abbilden.

18.1.3 Zeiger

Die ACL wird in der Implementierung als Array übergeben. Ein Array ist ein Speicherbereich, der fortlaufend die Elemente des Array enthält. Der Zugriff erfolgt durch einen Zeiger p, der auf das erste Element zeigt. Den Wert des Elementes, auf den ein Zeiger p zeigt, erhält man mit dem $*$ Operator. Der Zugriff auf das i-te Element erfolgt durch den Zeiger $p + i$. Man schreibt also $*(p+i)$ oder abkürzend $p[i]$. Die Anwendung des $*$ Operators ist nur für $0 \leq i \wedge i < n$ gültig, wobei n die Anzahl der Elemente des Array ist. Dabei ist zu beachten, daß n auch 0 sein kann.

In der Funktion wird auf das Array acl nicht direkt zugegriffen, sondern durch den Zeiger $aclp$. Dieser Zeiger wird schrittweise von Element zu Element gesetzt. Wir werden zeigen, daß die Invariante

$aclp = acl + i$

gilt. Dadurch können wir aclp dann durch den gleichwertigen Ausdruck ersetzen.

Für die folgende Beweisskizze benötigen wir noch die Semantik des Postfix-Inkrement-Operators:

$\{x = y\}\, x{+}{+}\, \{x = y + 1\}$

Wenn also vor dem Inkrement $x = y$ gilt, so gilt nach der Inkrementierung von $x = y + 1$. Hier folgt nun die Beweisskizze für die Schleife. In der Beweisskizze wird der zu verifizierende Code mit der Beweisführung verschränkt. Dabei benutzen wir die vorige Aussage. Wir möchten mit der folgenden Beweisskizze den jederzeit validen Zugriff auf den Speicherbereich nachweisen.

$i := 0;$
$\{i = 0\}$
$aclp := acl;$
$\{aclp = acl + i\}$
while $i < n$ **do**
$\{i < n\}$
$S;$

Da der Rumpf der **for** Schleife, oben durch S bezeichnet, weder acl, noch i, noch $aclp$ verändert, gilt die Invariante $aclp = acl + i$ auch nach S.

$\{aclp = acl + i\}$
$aclp{+}{+};$
$\{aclp = acl + i + 1\}$
$i{+}{+};$
$\{aclp = acl + i\}$
od

Außerdem gilt in S die Invariante $i < n$, so daß die Anwendung des $*$ Operators immer zulässig ist.

18.1.4 Byte-Endian Konvertierung

Zuletzt muß noch die Semantik der Makros `le16_to_cpu` und `le32_to_cpu` erklärt werden. Diese Makros werden auf alle Werte, die vom Dateisystem gelesen werden, angewendet. Das Dateisystem speichert alle Werte mit kleiner Byte-Endian, d.h. mit dem niederwertigsten Byte zuletzt. Wenn die CPU jedoch mit großer Byte-Endian Werten rechnet, müssen diese Werte umgewandelt werden. Das erledigen die entsprechend definierten Makros. In dieser Funktion sind lediglich der Inode und die ACL vom Dateisystem eingelesen worden. Die Werte des Inodes werden bereits beim Einlesen des Inode gewandelt. Wie man leicht sieht, werden alle Werte der ACL gewandelt. Daher beachten wir diese Makros nicht in der weiteren Beweisführung.

18.1.5 Zusammenfassung

Der C Code der Funktion `ext2_acl_permission` kann nun auf das verwendete Beweissystem abgebildet werden. Dazu müssen lediglich die verschiedenen Abbildungen von C Code Fragmenten, die in den vorherigen Abschnitten validiert wurden, gemeinsam auf den C Code angewendet werden. Dann ergibt sich folgende Darstellung von `ext2_acl_permission` im Beweissystem:

$return! := init;$
$in\_group := 0;$
$i := 0;$
$aclmask := \{r, w, x\};$
while $i < n \land return! = init$ **do**
 if $acl?\, i.acle\_type = acl\_user\_obj$ **then**
  if $current?.fsuid = inode?.i\_uid$ **then**
   if $mask? \subseteq acl?\, i.acle\_perms$ **then**
    $return! := success;$
   else
    $return! := deny;$
  fi fi
 else if $acl?\, i.acle\_type = acl\_mask$ **then**
  $aclmask := acl?\, i.acle\_perms;$

else if $acl?\,i.acle\_type = acl\_user$ **then**
if $current?.fsuid = acl?\,i.acle\_tag$ **then**
if $mask? \subseteq (acl?\,i.acle\_perms \cup aclmask)$ **then**
$return! := success;$
else
$return! := deny;$
fi fi
else if $acl?\,i.acle\_type = acl\_group\_obj$ **then**
if $inode?.i\_gid \in current?.groups \cup \{current?.fsgid\}$ **then**
$in\_group = 1;$
if $mask? \subseteq (acl?\,i.acle\_perms \cup aclmask)$ **then**
$return! := success;$
fi fi
else if $acl?\,i.acle\_type = acl\_group$ **then**
if $acl?\,i.acle\_tag \in current?.groups \cup \{current?.fsgid\}$ **then**
$in\_group = 1;$
if $mask? \subseteq (acl?\,i.acle\_perms \cup aclmask)$ **then**
$return! := success;$
fi fi
else if $acl?\,i.acle\_type = acl\_other\_obj$ **then**
if $in\_group \neq 0$ **then**
$return! := deny;$
fi
if $return! = init$ **then**
if $mask? \subseteq acl?\,i.perms$ **then**
$return! := success;$
else
$return! := deny;$
fi fi fi fi fi fi fi
if $return! = init$ **then**
$i{+}{+};$
fi od
if $return! = init$ **then**
$return! := deny;$
fi

18.2 Verifikation

Wir können nun die geforderten Eigenschaften an die Funktion formal fassen. Die folgenden Vorraussetzungen mögen erfüllt sein:

1. Die gleichnamigen Variablen des Z-Schema und der Funktion mögen entsprechende Belegungen besitzen. Dabei wurde bei der Umsetzung der Funktion in das Beweissystem bereits formal definiert, welche Belegungen einander entsprechen.
2. Das Schema $check\_acl$(siehe Abschnitt 17.2.2) möge erfüllt sein, sprich die übergebene ACL möge valide sein. Die Validität der ACL ist eine zentrale Vorraussetzung, die wir im Beweis immer wieder benutzen werden.

Dann soll für alle gültigen Belegungen des Z-Schema $ext2\_acl\_permission$, unter denen das Schema ein definiertes Ergebnis besitzt, das folgende für die gleichnamige Funktion gelten:

1. Die Funktion terminiert.
2. Es gilt bei Termination:

$$ext2\_acl\_permission.result! = return!$$

Um diese Eigenschaften nachzuweisen, prüfen wir, daß die Implementierung für alle Fälle, die die Spezifikation abdeckt, diese Forderungen abdeckt. Dabei treffen wir die folgenden vier Fallunterscheidungen entsprechend den Fallunterscheidungen der Spezifikation:

$ext2\_acl\_permission.3$

$current? : PROCESS$
$inode? : INODE$
$mask? : \mathbb{P}\, PERMISSION$
$result! : ERRNO$

$result! =$ **if** $(current?.fsuid = inode?.i\_uid)$ **then**
Erste Fallunterscheidung
else if $(\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_user \wedge e.tag = current?.fsuid)$ **then**
Zweite Fallunterscheidung
else if $(\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in \{current?.fsgid\} \cup current?.groups \vee e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups)$ **then**
Dritte Fallunterscheidung
else Vierte Fallunterscheidung

Zu jeder der vier Fallunterscheidungen konstruieren wir die Menge minimaler Annahmen, so daß die Fallunterscheidung zutrifft. Diese Annahmen benutzen wir dann, um das korrekte Ergebnis der Funktion nachzuweisen.

18.2.1 Erste Fallunterscheidung

Wir betrachten die erste Fallunterscheidung. Dazu konstruieren wir aus der Bedingung für die Fallunterscheidung die Annahme

$$current?.fsuid = inode?.i\_uid.$$

Man sieht leicht ein, daß diese Annahme für die Fallunterscheidung minimal ist. D. h. es gibt keine weniger restriktive Annahme, so daß die Bedingung der Fallunterscheidung erfüllt ist.

Aus der eingangs geforderten Validität von $acl?$ können wir folgern:

$$\exists_1\, e : \operatorname{ran} acl? \bullet e.type = acl\_user\_obj$$
$$\forall\, i, j : \operatorname{dom} acl? \bullet i \leq j \Rightarrow acl?\, i \leq_{ACL} acl?\, j$$

Durch Umkehrung und Vertauschung von i und j folgt:

$$\forall\, i, j : \operatorname{dom} acl? \bullet acl?\, i <_{ACL} acl?\, j \Rightarrow i < j$$

Da acl_user_obj bzgl. der $\leq_{ACL}$ (siehe Abschnitt 17.2.2) Relation minimal ist, folgt:

$$acl?\, 0.type = acl\_user\_obj$$

Die Anforderung der Spezifikation lautet also

$$mask? \subseteq acl?\,0.perm \Rightarrow return! = success$$
$$mask? \not\subseteq acl?\,0.perm \Rightarrow return! = deny.$$

Wir beweisen, daß der Wert von $return!$ zu Programmende der Anforderung genügt. Der Beweis wird in einer sogenannten Beweisskizze in das Programm eingeflochten. Der Beweis ist auf gesonderten Zeilen, eingeschlossen in geschweiften Klammern ($\{\}$). Die Aussagen des Beweises beziehen sich jeweils auf die Programmausführung bis zu ihrer Position im Programm. Es werden Aussagen über die Belegung der Variablen getroffen und damit implizit über die folgende Programmausführung.

```
return! := init;
in_group := 0;
i := 0;
...
{i = 0}
while i < n ∧ return! = init do
{i = 0 ⇒ acl? i.type = acl_user_obj}
 if acl? i.acle_type = acl_user_obj then
{current?.fsuid = inode?.i_uid}
   if current?.fsuid = inode?.i_uid then
    if mask? ⊆ acl? i.acle_perms then
     return! := success;
    else
     return! := deny;
{mask? ⊆ acl? 0.perm ⇒ return! = success
mask? ⊄ acl? 0.perm ⇒ return! = deny}
   fi fi
...
 fi fi fi fi fi fi fi
{return! ≠ init}  if return! = init then
   i++;
fi od
{return! ≠ init}if return! = init then
 return! := deny;
fi
```

Im ersten Durchlauf wird $return!$ gesetzt. Danach wird, ohne weitere Anweisungen zu durchlaufen, das Ende erreicht. Das Ergebnis ist

$$return! = \textbf{if } mask? \subseteq acl?\,0.perm \textbf{ then } success \textbf{ else } deny$$

entsprechend der Spezifikation.

18.2.2 Zweite Fallunterscheidung

Wir betrachten den nächsten Fall, nehmen also an, daß die erste Fallunterscheidung nicht zutrifft:

$$current?.fsuid \neq inode?.i\_uid$$

Außerdem muß, damit die nächste Fallunterscheidung der Spezifikation zutrifft,

$$\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_user \land e.tag = current?.fsuid$$

mindestens gelten. Dann gilt wegen der Validität der ACL sogar

$$\exists_1\, e : \operatorname{ran} acl? \mid e.type = acl\_user \land e.tag = current?.fsuid.$$

Wie wir bereits gezeigt haben, gilt

$$\forall\, i, j : \operatorname{dom} acl? \bullet acl?\, i <_{ACL} acl?\, j \Rightarrow i < j.$$

Als Spezialfall gilt dann

$$\begin{array}{l} \forall\, i : \operatorname{dom} acl? \mid acl?\, i.type = acl\_user; \\ \ \ j : \operatorname{dom} acl? \mid acl?\, j.type \in \\ \quad \{acl\_group\_obj, acl\_group, acl\_other\_obj\} \bullet \\ \qquad i < j. \end{array}$$

Damit können wir zu obigem verschärfen

$$\begin{array}{l} \exists_1\, i : \operatorname{dom} acl? \mid acl?\, i.type = acl\_user \land acl?\, i.tag = current?.fsuid \bullet \\ \ \ \forall\, j : \operatorname{dom} acl? \mid j.type \in \{acl\_group\_obj, acl\_group, acl\_other\_obj\} \bullet \\ \quad i < j \end{array}$$

Wir bezeichnen dieses eindeutig bestimmte i mit i_{user}. Es hat die Eigenschaft, kleiner als die Indizes der übrigen Einträge mit Ausnahme des acl_user_obj Eintrags zu sein. Damit können wir anstelle des Ausdrucks

$$\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_user \land e.tag = current?.fsuid$$

$acl?\, i_{user}$ benutzen.

Damit können wir nun den Beweis der Korrektheit des Codes für diesen Fall führen:

$$\begin{array}{l}
return! := init; \\
in_group := 0; \\
i := 0; \\
aclmask := \{r, w, x\}; \\
\textbf{while } i < n \land return! = init \textbf{ do} \\
\ \ \textbf{if } acl?\, i.acle_type = acl_user_obj \textbf{ then} \\
\{current?.fsuid \neq inode?.i_uid\} \\
\quad \textbf{if } current?.fsuid = inode?.i_uid \textbf{ then} \\
\ldots \\
\quad \textbf{fi fi} \\
\ \ \textbf{else if } acl?\, i.acle_type = acl_mask \textbf{ then} \\
\quad aclmask := acl?\, i.acle_perms; \\
\{i = i_{user} \Leftrightarrow \\
acl?\, i.acle_type = acl_user \land current?.fsuid = acl?\, i.acle_tag\} \\
\ \ \textbf{else if } acl?\, i.acle_type = acl_user \textbf{ then} \\
\quad \textbf{if } current?.fsuid = acl?\, i.acle_tag \textbf{ then} \\
\qquad \textbf{if } mask? \subseteq (acl?\, i.acle_perms \cap aclmask) \textbf{ then} \\
\qquad\ \ return! := success; \\
\qquad \textbf{else} \\
\qquad\ \ return! := deny; \\
\quad \textbf{fi fi} \\
\{i >= i_{user} \Rightarrow return! \neq init\} \\
\{acl?\, i.acle_type = acl_group_obj \Rightarrow i > i_{user}\} \\
\ \ \textbf{else if } acl?\, i.acle_type = acl_group_obj \textbf{ then} \\
\ldots \\
\quad \textbf{fi fi}
\end{array}$$

$\{acl?\ i.acle\_type = acl\_group \Rightarrow i > i_{user}\}$
else if $acl?\ i.acle\_type = acl\_group$ **then**
...
fi fi
$\{acl?\ i.acle\_type = acl\_other\_obj \Rightarrow i > i_{user}\}$
else if $acl?\ i.acle\_type = acl\_other\_obj$ **then**
...
fi fi fi fi fi fi fi
$\{i < i_{user} \Rightarrow return! = init$
$i >= i_{user} \Rightarrow return! \neq init\}$
if $return! = init$ **then**
$i{+}{+};$
fi od
$\{return! \neq init\}$
if $return! = init$ **then**
$return! := deny;$
fi

Wie gezeigt, kann $return!$ erst gesetzt werden, wenn i den Wert i_{user} erreicht. Da i monoton steigt und mit 0 initialisiert wird, ist dies gesichert: Die Schleife kann nicht vorher terminieren. Nach der Zuweisung wird die Schleife und die Funktion beendet. Die übrigen Zweige können nicht betreten werden, da die daran geknüpften Bedingungen nur für $i > i_{user}$ erfüllt sein können.

Es gilt bei Termination

$return! =$ **if** $(mask? \subseteq (acl?\ i_{user}.acle\_perms \cap aclmask))$ **then**
$success$
else
$deny.$

Wir müssen nun die passende Belegung der Variablen `aclmask` nachweisen: Wenn es einen ACL Eintrag vom Typ acl_mask gibt, so gibt es genau einen solchen Eintrag. Daher gibt es ein eindeutig bestimmtes i_{mask} mit

$acl?\ i_{mask}.type = acl\_mask$

und es gilt innerhalb der **while** Schleife:

$i >= i_{mask} \land \exists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask \Rightarrow$
$aclmask = (\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms)$
$\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask \Rightarrow$
$aclmask = \{r, w, x\},$

Diese Eigenschaft läßt sich an der folgenden Beweisskizze nachvollziehen:

$return! := init;$
$in\_group := 0;$
$i := 0;$
$aclmask := \{r, w, x\};$
$\{aclmask = \{r \mid w \mid x\}\}$

$\textbf{while } i < n \land return! = init \textbf{ do}$
$\ldots$
$\{i = i_{mask} \Rightarrow acl?\ i.acle\_type = acl\_mask\}$
$\{i \neq i_{mask} \Rightarrow acl?\ i.acle\_type \neq acl\_mask\}$
$\{\nexists e : \operatorname{ran} acl? \mid e.type = acl\_mask \Rightarrow acl?\ i.acle\_type \neq acl\_mask\}$
$\quad\textbf{else if } acl?\ i.acle\_type = acl\_mask \textbf{ then}$
$\quad\quad aclmask := acl?\ i.acle\_perms;$
$\{\exists e : \operatorname{ran} acl? \mid e.type = acl\_mask \land i \geq i_{mask} \Rightarrow$
$\quad aclmask = (\mu e : \operatorname{ran} acl? \mid e.type = acl\_mask).acle\_perms)\}$
$\{\nexists e : \operatorname{ran} acl? \mid e.type = acl\_mask \Rightarrow aclmask = \{r \mid w \mid x\}\}$
$\ldots$

Damit haben wir bereits alle Zuweisungen an $aclmask$ betrachtet.

Aus der Sortierung der ACL kann man folgern:

$$\exists i_{mask} : \operatorname{dom} acl? \mid acl?\ i_{mask}.type = acl\_mask \Rightarrow$$
$$i_{user} > i_{mask}.$$

Wenn also i_{mask} existiert, erreichte i diesen Wert bereits und $aclmask$ wurde entsprechend gesetzt. Daraus folgt durch Einsetzung für $acl?\ i_{mask}$ und $aclmask$ sofort entsprechend der Spezifikation:

$return! =$
$\quad\textbf{if } mask? \subseteq (\mu e : \operatorname{ran} acl? \mid e.type = acl\_user \land$
$\quad\quad e.tag = current?.fsuid).acle\_perms \cap ($
$\quad\quad\quad\textbf{if } (\exists e : \operatorname{ran} acl? \mid e.type = acl\_mask) \textbf{ then}$
$\quad\quad\quad\quad(\mu e : \operatorname{ran} acl? \mid e.type = acl\_mask).acle\_perms) \textbf{ then}$
$\quad\quad\quad\textbf{else}$
$\quad\quad\quad\quad\{r, w, x\})$
$\quad\quad success$
$\quad\textbf{else}$
$\quad\quad deny$

18.2.3 Belegung in_group

Vor den nächsten Fallunterscheidungen müssen wir nun die passende Belegung der Variablen `in_group` nachweisen. Wir werden annehmen, daß gilt

$$\exists e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \land inode?.i\_gid \in$$
$$\{current?.fsgid\} \cup current?.groups \lor$$
$$e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups.$$

Aus der Validität von $acl?$ folgt:

$$\exists_1 i_{other\_obj} : \operatorname{dom} acl? \mid acl?\ i_{other\_obj}.type = acl\_other\_obj.$$

Dann folgt aufgrund der Sortierung von $acl?$:

$$\exists i_{group} : \operatorname{dom} acl? \mid i_{group} < i_{other\_obj} \land$$
$$(\textbf{let } e = acl?\ i_{group} \bullet e.type = acl\_group\_obj \land inode?.i\_gid \in$$
$$\{current?.fsgid\} \cup current?.groups \lor$$
$$e.type = acl\_group \land e.tag \in \{current?.fsgid\} \cup current?.groups)$$

Unter diesen Voraussetzungen wird in_group bei i gleich einem gewissen i_{group} auf 1 gesetzt:

$return! := init;$
$in\_group := 0;$
$i := 0;$
$aclmask := \{r, w, x\};$
while $i < n \wedge return! = init$ **do**
$\{\exists\, i_{group} : \mathrm{dom}\, acl? \mid i_{group} < i_{other\_obj} \wedge$
$(\mathbf{let}\ e = acl?\ i_{group} \bullet e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups)\}$
if $acl?\ i.acle\_type = acl\_user\_obj$ **then**
...
fi fi
else if $acl?\ i.acle\_type = acl\_mask$ **then**
$aclmask := acl?\ i.acle\_perms;$
else if $acl?\ i.acle\_type = acl\_user$ **then**
...
else if $acl?\ i.acle\_type = acl\_group\_obj$ **then**
if $inode?.i\_gid \in current?.groups \cup \{current?.fsgid\}$ **then**
$\{\exists\, i : \mathrm{dom}\, acl? \mid i < i_{other\_obj} \wedge$
$(\mathbf{let}\ e = acl?\ i \bullet e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups\}$
$in\_group = 1;$
if $mask? \subseteq (acl?\ i.acle\_perms \cup aclmask)$ **then**
$return! := success;$
fi fi
$\{in\_group = 1\}$
else if $acl?\ i.acle\_type = acl\_group$ **then**
if $acl?\ i.acle\_tag \in current?.groups \cup \{current?.fsgid\}$ **then**
$\{\exists\, i_{group} : \mathrm{dom}\, acl? \mid i_{group} < i_{other\_obj} \wedge$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups)\}$
$in\_group = 1;$
if $mask? \subseteq (acl?\ i.acle\_perms \cup aclmask)$ **then**
$return! := success;$
fi fi
$\{in\_group = 1\}$
else if $acl?\ i.acle\_type = acl\_other\_obj$ **then**
if $in\_group \neq 0$ **then**
$return! := deny;$
fi
if $return! = init$ **then**
if $mask? \subseteq acl?\ i.perms$ **then**
$return! := success;$
else
$return! := deny;$
fi fi fi fi fi fi fi

$\{(\exists\, i_{group} : \mathrm{dom}\, acl? \mid i_{group} < i_{other\_obj} \wedge$
$(\mathbf{let}\ e = acl?\ i_{group} \bullet e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups) \wedge i = i_{group} \Rightarrow$
$in\_group = 1\}$
if $return! = init$ **then**
$i{+}{+};$
fi od
if $return! = init$ **then**
$return! := deny;$
fi

Da in_group nie wieder auf einen anderen Wert gesetzt werden kann, gilt sogar:

$(\exists\, i_{group} : \mathrm{dom}\, acl? \mid i_{group} < i_{other\_obj} \wedge$
$(\mathbf{let}\ e = acl?\ i_{group} \bullet e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups) \wedge i >= i_{group} \Rightarrow$
$in\_group = 1$

Gilt die Eingangsvoraussetzung hingegen nicht, kann in_group nicht auf 1 gesetzt werden, muß also 0 sein:

$i = i_{other\_obj} \Rightarrow$
$in\_group = \mathbf{if}\ \exists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups)$ **then**
1
else
0

Diese letzte Form werden wir im nächsten Schritt verwenden.

18.2.4 Dritte Fallunterscheidung

Wir betrachten nun die dritte Fallunterscheidung. Dazu negieren wir wiederum die Annahmen der ersten und zweiten Fallunterscheidung. Außerdem

$current?.fsuid \neq inode?.i\_uid$

$\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \wedge e.tag = current?.fsuid$

$\exists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups$

Wir betrachten zunächst den Fall, daß die Spezifikation den Zugriff verweigert, d.h. es soll gelten:

$\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \wedge mask? \subseteq e.perms \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups \wedge$
$mask? \subseteq e.acle\_perms \cap ($
if $(\exists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask)$ **then**
$(\mu\, e : \mathrm{ran}\, acl? \mid e.type = acl\_mask).acle\_perms)$
else $\{r, w, x\})$

Wir werden in der folgenden Beweisskizze zeigen, daß die Schleife nicht terminiert, bevor $i = i_{other}$. Aufgrund der bereits nachgewiesenen Belegung von in_group können wir das korrekte Verhalten folgern.

$return! := init;$
$in\_group := 0;$
$i := 0;$
$aclmask := \{r, w, x\};$
while $i < n \land return! = init$ **do**
 if $acl?\ i.acle\_type = acl\_user\_obj$ **then**
$\{current?.fsuid \neq inode?.i\_uid\}$
  if $current?.fsuid = inode?.i\_uid$ **then**
...
  fi fi
 else if $acl?\ i.acle\_type = acl\_mask$ **then**
  $aclmask := acl?\ i.acle\_perms;$
 else if $acl?\ i.acle\_type = acl\_user$ **then**
$\{\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \land e.tag = current?.fsuid\}$
  if $current?.fsuid = acl?\ i.acle\_tag$ **then**
...
  fi fi
 else if $acl?\ i.acle\_type = acl\_group\_obj$ **then**
  if $inode?.i\_gid \in current?.groups \cup \{current?.fsgid\}$ **then**
   $in\_group = 1;$
{Bedingung kann nicht erfüllt sein.}
   if $mask? \subseteq (acl?\ i.acle\_perms \cup aclmask)$ **then**
    $return! := success;$
  fi fi
 else if $acl?\ i.acle\_type = acl\_group$ **then**
  if $acl?\ i.acle\_tag \in current?.groups \cup \{current?.fsgid\}$ **then**
   $in\_group = 1;$
{Bedingung kann nicht erfüllt sein.}
   if $mask? \subseteq (acl?\ i.acle\_perms \cup aclmask)$ **then**
    $return! := success;$
  fi fi
 else if $acl?\ i.acle\_type = acl\_other\_obj$ **then**
$\{acl?\ i.acle\_type = acl\_other\_obj \Rightarrow i = i_{other} \Rightarrow in\_group = 1\}$
  if $in\_group \neq 0$ **then**
   $return! := deny;$
  fi
$\{return! = deny\}$
  if $return! = init$ **then**
   if $mask? \subseteq acl?\ i.perms$ **then**
    $return! := success;$
   else
    $return! := deny;$
 fi fi fi fi fi fi fi
$\{return! = $ **if** $i < i_{other}$ **then** $init$ **else** $deny\}$
 if $return! = init$ **then**
  $i{+}{+};$
fi od

if $return! = init$ **then**
  $return! := deny;$
fi
$\{return! = deny\}$

Nun wollen wir im Gegenteil annehmen, daß gilt

$\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \wedge mask? \subseteq e.perms \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups \wedge$
$mask? \subseteq e.acle\_perms \cap ($
if $(\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_mask)$ **then**
$(\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_mask).acle\_perms)$
else $\{r, w, x\})$

Dann können wir wegen der Sortierung von $acl?$ auch schreiben

$\exists\, i_{group} : \operatorname{dom} acl? \mid i_{group} < i_{other} \wedge$ **let** $e = acl?\ i$ $\bullet$
$(\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \wedge mask? \subseteq e.perms \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups \wedge$
$mask? \subseteq e.acle\_perms \cap ($
if $(\exists\, e : \operatorname{ran} acl? \mid e.type = acl\_mask)$ **then**
$(\mu\, e : \operatorname{ran} acl? \mid e.type = acl\_mask).acle\_perms)$
else $\{r, w, x\})$ **then**

Nun läßt sich auch die Korrektheit für diesen Fall beweisen.

$return! := init;$
$in\_group := 0;$
$i := 0;$
$aclmask := \{r, w, x\};$
while $i < n \wedge return! = init$ **do**
  if $acl?\ i.acle\_type = acl\_user\_obj$ **then**
$\{current?.fsuid \neq inode?.i\_uid\}$
    if $current?.fsuid = inode?.i\_uid$ **then**
...
    fi fi
  else if $acl?\ i.acle\_type = acl\_mask$ **then**
    $aclmask := acl?\ i.acle\_perms;$
  else if $acl?\ i.acle\_type = acl\_user$ **then**
$\{\nexists\, e : \operatorname{ran} acl? \mid e.type = acl\_user \wedge e.tag = current?.fsuid\}$
    if $current?.fsuid = acl?\ i.acle\_tag$ **then**
...
    fi fi
  else if $acl?\ i.acle\_type = acl\_group\_obj$ **then**
    if $inode?.i\_gid \in current?.groups \cup \{current?.fsgid\}$ **then**
      $in\_group = 1;$
{Entweder gibt es so einen $acl?\ i_{group}, i_{group} < i_{other} \ldots$}
      if $mask? \subseteq (acl?\ i.acle\_perms \cup aclmask)$ **then**
        $return! := success;$
    fi fi

else if $acl?\ i.acle\_type = acl\_group$ **then**
if $acl?\ i.acle\_tag \in current?.groups \cup \{current?.fsgid\}$ **then**
$in\_group = 1;$
{... oder so einen $acl?\ i_{group}, i_{group} < i_{other}$}
if $mask? \subseteq (acl?\ i.acle\_perms \cup aclmask)$ **then**
$return! := success;$
fi fi
{Schleife terminiert vorher!}
else if $acl?\ i.acle\_type = acl\_other\_obj$ **then**
if $in\_group \neq 0$ **then**
$return! := deny;$
fi
if $return! = init$ **then**
if $mask? \subseteq acl?\ i.perms$ **then**
$return! := success;$
else
$return! := deny;$
fi fi fi fi fi fi fi
$\{\exists\, i_{group} : \mathrm{dom}\, acl? \mid i < i_{other} \bullet return! = success\}$
if $return! = init$ **then**
$i{+}{+};$
fi od
$\{return! = success\}$
if $return! = init$ **then**
$return! := deny;$
fi
$\{return! = success\}$

Hier wird der letzte Zweig also nicht betreten, da die Schleife schon vorher terminiert, weil *return* bereits auf *success* gesetzt wurde.

18.2.5 Vierte Fallunterscheidung

Es bleibt der letzte Fall zu betrachten. Diese Fallunterscheidung ist selbst an keine Bedingungen geknüpft, außer daß die vorherigen Fallunterscheidungen nicht zutreffen. Daher sind lediglich die entsprechenden Bedingungen zu negieren:

$current?.fsuid \neq inode?.i\_uid$

$\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_user \wedge e.tag = current?.fsuid$

$\nexists\, e : \mathrm{ran}\, acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups$

Die entsprechende Beweisskizze lautet:

$return! := init;$
$in\_group := 0;$
$i := 0;$
$aclmask := \{r, w, x\};$
while $i < n \wedge return! = init$ **do**
if $acl?\ i.acle\_type = acl\_user\_obj$ **then**
$\{current?.fsuid \neq inode?.i\_uid\}$
if $current?.fsuid = inode?.i\_uid$ **then**
...

else if $acl?\ i.acle\_type = acl\_mask$ **then**
$aclmask := acl?\ i.acle\_perms;$
else if $acl?\ i.acle\_type = acl\_user$ **then**
$\{\nexists\, e : \operatorname{ran} acl? \mid e.type = acl\_user \wedge e.tag = current?.fsuid\}$
if $current?.fsuid = acl?\ i.acle\_tag$ **then**
...
else if $acl?\ i.acle\_type = acl\_group\_obj$ **then**
$\{\nexists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups\}$
if $inode?.i\_gid \in current?.groups \cup \{current?.fsgid\}$ **then**
...
else if $acl?\ i.acle\_type = acl\_group$ **then**
$\{\nexists\, e : \operatorname{ran} acl? \mid e.type = acl\_group\_obj \wedge inode?.i\_gid \in$
$\{current?.fsgid\} \cup current?.groups \vee$
$e.type = acl\_group \wedge e.tag \in \{current?.fsgid\} \cup current?.groups\}$
if $acl?\ i.acle\_tag \in current?.groups \cup \{current?.fsgid\}$ **then**
...
else if $acl?\ i.acle\_type = acl\_other\_obj$ **then**
$\{in\_group = 0\}$
if $in\_group \neq 0$ **then**
$return! := deny;$
fi
$\{return = init\}$
if $return! = init$ **then**
if $mask? \subseteq acl?\ i.perms$ **then**
$return! := success;$
else
$return! := deny;$
fi fi fi fi fi fi fi
$\{i = i_{other} \Rightarrow$
$return! =$ **if** $mask? \subseteq acl?\ i_{other}.perms$ **then**
$success$ **else** $deny\}$
if $return! = init$ **then**
$i{+}{+};$
fi od
$\{return \neq init\}$
if $return! = init$ **then**
$return! := deny;$
fi
$\{return! =$ **if** $mask? \subseteq acl?\ i_{other}.perms$ **then**
$success$ **else** $deny\}$

18.3 Zusammenfassung Verifikation

Wir haben für die C Implementierung, die formal nicht faßbar ist, ein, wie wir meinen, äquivalentes Programm in einer formal beschriebenen Sprache gegeben. Da die Ausgangsbasis nun aber nicht formal definiert ist, kann auch die Korrektheit dieses Schritts letztendlich nicht formal bewiesen werden, lediglich plausibel gemacht werden.

Darauf haben wir für alle alle Eingaben, für die die Spezifikation das Verhalten der Implementierung vorschreibt, das korrekte Verhalten der abstrakten Form der Implementierung nachgewiesen. Demnach sind z. B.

die Abfragen

$$\textbf{while}\ i < n \land return! = init\ \textbf{do}$$

und am Schluß

$\textbf{if}\ return! = init\ \textbf{then}$
$\quad return! := deny;$
$\textbf{fi}$

überflüssig. Die Implementierung würde sich auch ohne diese Abfragen korrekt verhalten. Jedoch würde niemand so programmieren. Die Implementierung ist so sehr viel robuster: Schließlich könnte sich die Umgebung ja doch mal falsch verhalten und die Funktion würde dann in eine Endlosschleife gelangen, unzulässige Speicherzugriffe tätigen oder auch bei Termination ein undefiniertes Ergebnis haben.

18.3.1 Termination

Mit $p = n - i$ können wir eine Funktion definieren, deren Wert mit jedem Schleifendurchlauf immer um 1 erniedrigt wird, aber positiv bleibt wegen $i < n$. Folglich ist die Termination der Schleife garantiert und somit die der gesamten Funktion, da keine weiteren Schleifen folgen.

19. Test

Tests benutzten wir in zweierlei Hinsicht. Zunächst verwendeten wir einen manuellen Test zur Fehleraufspürung und -lokalisation. Außerdem testeten wir manuell die Grobfunktionalität unserer Implementierung.

Durch das Verwenden einer automatisierten Testmethode konnten wir eine Teilfunktionalität unserer Implementierung mit einer so hohen Zahl von Testfällen testen, wie sie mit manuellen Tests nicht zu erreichen gewesen wäre. Damit kann mit an Sicherheit grenzender Wahrscheinlichkeit gesagt werden, daß unsere Implementation diese Teilfunktionalität korrekt erfüllt. Aus Aufwandsgründen konnten wir im Rahmen des Projekts nicht die volle Funktionalität mit Hilfe von automatisierten Testverfahren prüfen.

19.1 Manueller Test

Es war uns klar, daß wir nicht die gesamte POSIX-Spezifikation durch einen automatisierten Test würden abdecken können. Letzteres war jedoch das Ziel des ersten manuellen Tests. Dabei benutzten wir nur die dem normalen Benutzer verfügbaren Werkzeuge. Die genaue Testprozedur befindet sich im Anhang A.

Der manuelle Test hatte außerdem das Ziel, ohne großen Aufwand grobe Fehler, die bereits nach wenigen Tests auftreten, zu beseitigen. Auf diese Weise wollten wir vermeiden, daß wir beim automatisierten Test noch allzu oft das System under Test - d.h. unsere Implementierung - hätten ändern müssen. Es wäre sehr zeitraubend gewesen, nach wenigen Tests schon wieder einen Fehler im System under Test zu finden und das Testsystem mit einem neuen Testling neu zu starten. Die Fehler, die wir bei diesen ersten Tests gefunden und behoben haben, sind im Anhang B aufgeführt. Zur Fehlersuche verwendeten wir den Laufzeit-Entwanzer `gdb` mit der grafischen Erweiterung `ddd`. Da sich damit schlecht Kernel Code debuggen läßt, setzten wir hier Kernel Messages ein, um Ausgaben für die Fehleranalyse zu erhalten. Die Kernelfunktionen haben wir auch im White Box Verfahren getestet, um eine Zweigüberdeckung der relevanten Kernelfunktionen zu erhalten. Zweigüberdeckung bedeutet, daß jede Anweisung des zu testenden Codes mindestens einmal ausgeführt wurde.

19.2 Automatisierter Test

Mit dem manuellen Test haben wir die Funktion der Implementierung nur grob getestet. Um bei einem manuell durchgeführten Test eine Abdeckung aller relevanter Testfälle zu erreichen, müßte man für jeden relevanten Testfall einen geeigneten Testaufruf entwickeln, diesen ausführen und die Korrektheit des Ergebnisses prüfen. Angesichts der Vielzahl von verschiedenen Eingabeparametern und Systemzuständen ist dies manuell nicht mehr in vertretbarer Zeit durchzuführen.

Daher haben wir uns entschlossen, einen Teil der Implementierung durch einen automatisierten Test zu prüfen, und zwar das korrekte Setzen einer ACL mit dem Kommando `setfacl`.

Für den automatisierten Test verwendeten wir das Testwerkzeug RT-Tester. Dies ist vornehmlich dazu gedacht, automatisierte Hardware-in-the-loop-Tests und Software-Komponenten-Tests auf Prozeß- oder Thread-Level für eingebettete Echtzeitsysteme durchzuführen. In der Regel wird dazu CSP als Spezifikationssprache verwendet. CSP bietet die Möglichkeit, nicht deterministische Abläufe spezifizieren zu können. Wir konnten so mit CSP das Erzeugen syntaktisch korrekter Kommandozeilen für den Aufruf des Kommandos `setfacl` spezifizieren. Mit Hilfe dieser Spezifikation konnten wir das Test-Werkzeug RT-Tester verwenden, um zufallsgesteuert Kommandozeilen zu erzeugen.

Im üblichen Gebrauch von RT-Tester werden nicht nur die Testwerte – wie in unserem Fall – in CSP spezifiziert, sondern auch die korrekten Antworten des Systems. Theoretisch ließen sich diese auch bei uns mit CSP spezifizieren, praktisch war dies jedoch aus dem folgenden Grund nicht möglich: CSP ist hervorragend geeignet, die Kommunikation zwischen Prozessen zu modellieren. Möchte man jedoch ein Verhalten unterhalb der Prozeßebene, auf Unit-Ebene modellieren und verifizieren, gelangt man schnell zu einem Zustandsraum, der die verfügbaren Werkzeuge überfordert. Auch bei nur etwas komplizierten Datenstrukturen explodiert der Zustandsraum: Die Anzahl der Zustände wächst exponentiell mit der Anzahl der Datenfelder. Das war auch bei unserem Problem der Fall.

Unsere erste Überlegung war, einen Back-To-Back-Test gegen eine existierende ACL-Implementierung durchzuführen. Hier erschien die Solaris Implementierung geeignet. Sie unterscheidet sich lediglich in der Aufruf-Syntax von Draft 13, auf dem die vorliegende Alpha-Implementierung von Remy Card beruht. Draft 17 unterscheidet sich jedoch auch hinsichtlich der Semantik in bezug auf die Gruppenrechte der Solaris-Implementierung. Daher verwarfen wir diese Idee.

Wir benötigten daher eine Spezifikationssprache, mit der es möglich ist, ein Referenzsystem zu erzeugen, gegen das wir unsere Implementierung testen können.

Für diese Anwendung ist ASpecT geeignet. ASpecT ist eine algebraische Spezifikationssprache. Die Spezifikationssprache Z, die wir zur Spezifikation des Algorithmus zur Zugriffskontrolle verwendet haben, hat gegenüber ASpecT den Nachteil, daß es keine Werkzeuge gibt, um aus erstellten Spezifikationen lauffähige Implementierungen zu erzeugen. Auf Grund der Form der Sprache Z wird es solche Werkzeuge wahrscheinlich auch niemals geben. ASpecT wurde von Anfang an als eine Implementierung algebraischer Spezifikationen mit abstrakten Datentypen entwickelt. Das heißt, Spezifikationen können in ein ausführbares Programm übersetzt werden. Auch parametrisierte (generische) Module werden unterstützt.

Wir haben mit Hilfe von ASpecT ein Referenzsystem entwickelt, das die Operationen des Kommandos `setfacl` auf textueller Ebene durchführt. Dieses Referenzsystem erhält eine ACL in Textrepräsentation und eine Kommandozeile zum Aufruf eines `setfacl`-Kommandos als Eingabe und gibt die durch das Kommando entsprechend der Spezifikation modifizierte ACL wieder in einer Textrepräsentation aus.

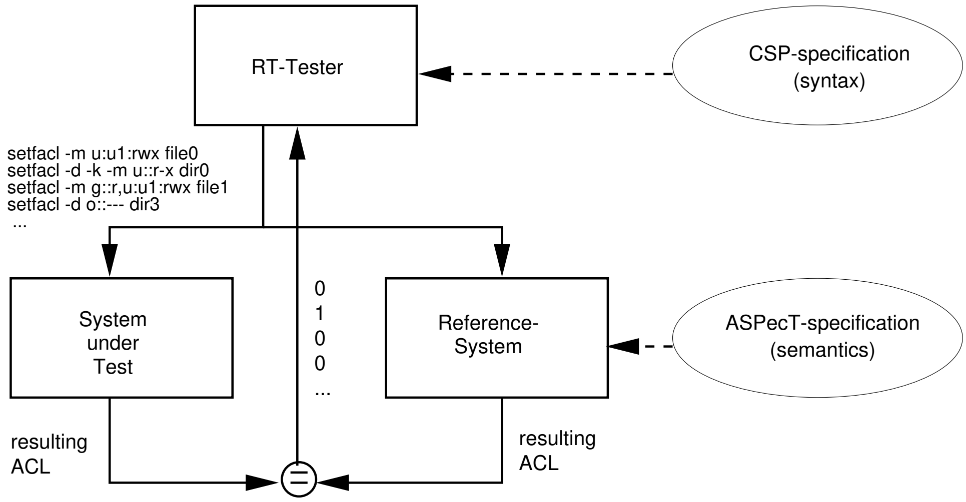


Abbildung 19.1: Architektur des Testsystems

Zusätzlich entwickelten wir ein Testprogramm in Form eines `AWK`-Skripts. Dies erhält eine `setfacl`-Kommandozeile als Eingabe. Zunächst bestimmt das Programm die vorhandenen ACLs der zu modifizierenden Dateien mit Hilfe von `getfacl`. Mit diesen und der Kommandozeile selbst wird das Referenzsystem aufgerufen. Das Testprogramm läßt dann die Kommandozeile durch das zu testenden System ausführen. Schließlich werden die resultierenden ACLs miteinander verglichen. Stimmen sie überein, verhält sich das zu testende System bezüglich der Kommandozeile gemäß der Spezifikation, andernfalls ist ein Fehler aufgetreten. Der Quellcode dieses Testprogramms ist in Anhang D.1 zu finden.

Dieses Testprogramm wird nun vom RT-Tester mit den gemäß der CSP-Spezifikation erzeugten Kommandozeilen aufgerufen. Die Ergebnisse des Testprogramms werden zusammen mit der jeweiligen Kommandozeile fortlaufend protokolliert.

Abbildung 19.1 zeigt die Architektur des gesamten Testsystems.

19.3 RT-Tester-Konfiguration

Neben der Testspezifikation mit CSP sind zur Benutzung des Testwerkzeugs RT-Tester einige weitere Konfigurationen durchzuführen.

Im folgenden werden die wesentlichen Konfigurationsschritte erläutert, die wir durchgeführt haben. Für eine genauere Beschreibung des RT-Testers sei auf den Abschnitt 6.6 verwiesen.

19.3.1 Konfiguration des Abstract Machine Layer (AML)

Zunächst werden dazu die CSP-Spezifikationen mit Hilfe des FDR-Tools in Transitionsgraphen übersetzt. Dies geschieht mit dem Skript `TGgen`, das zu dem Umfang des RT-Tester-Systems gehört. Das Skript erhält als ersten Parameter den Namen der Datei mit der CSP-Spezifikation, wobei die Endung `.csp` weggelassen wird. Als zweiter Parameter wird der Name des CSP-Prozesses angegeben, den die abstrakte Maschine (AM) ausführen soll. Das Skript erzeugt eine Reihe von Dateien, deren Inhalt den Transitionsgraphen beschreiben. Zudem wird eine Alphabet-Datei erzeugt, die zu jedem CSP-Event die interne Repräsentation in Form eines Integer-Wertes und den Eventtyp (`AM_INPUT`, `AM_OUTPUT`, etc.) enthält.

Die Hauptkonfigurationsdatei `$VVTHOME/conf/vvtconfig.txt` beinhaltet für jede abstrakte Maschine eine Reihe von Parametern. Dies sind im wesentlichen:

- Die Namen der aus der CSP-Spezifikation erzeugten Dateien, die zusammen die Informationen über den Transitionsgraphen enthalten. Dies sind die Dateien, die mit Hilfe des Skripts `TGgen` erzeugt wurden.
- Die IDs der CCL-Prozesse von denen `AM_INPUT` Events empfangen werden. Dies sind in unserem Fall die Prozesse `vvtrcvccl` (ID: 12) und auch `vvtsndccl` (ID: 111). Letzterer leitet Events auch direkt von einer AM zu einer anderen weiter, daher muß eine AM auch Events von diesem Prozeß empfangen können.
- Um auch Timing-Bedingungen prüfen zu können, verwendet RT-Tester Timer, die parallel zu den abstrakten Maschinen laufen und mittels Events gesteuert werden. Diese Timer werden hier definiert. Für jede Zeitdauer, die benötigt wird, wird ein eigener Timer definiert. In unserem Fall wird nur ein Timer verwendet, der Timer 9 mit einer Zeit von 2000 ms.

Für unseren Test werden 3 abstrakte Maschinen verwendet. Dies war nötig, damit der Transitionsgraph für eine einzelne abstrakte Maschine nicht zu groß wird. Wie die Aufteilung geschieht, ist unten bei der Beschreibung der CSP-Spezifikation beschrieben. Die vollständige Konfigurationsdatei ist im Anhang C.1 angegeben.

Die abstrakten Maschinen werden in dem Skript `gotest` einzeln gestartet. Dies geschieht mit dem Programm `vvtam`. Dies erhält als ersten Parameter die ID der zu startenden AM. Als optionaler zweiter Parameter wird die Nummer des Startzustandes angegeben. Er muß dann angegeben werden, wenn dieser ungleich 0 ist. Die Aufrufsyntax lautet folgendermaßen:

```
vvtam amid [startstate]
```

Das Skript `TGgen` gibt den Startszustand aus und speichert ihn zudem in der Datei `xxx.startstate`, wobei `xxx` der Name der zugehörigen CSP-Datei (ohne die Endung `.csp`) ist.

19.3.2 Konfiguration des Communication Control Layer (CCL)

Hier ist in erster Linie eine Umwandlung von AM Events in Raw Events und umgekehrt erforderlich. AM Events sind Integerwerte, die von den abstrakten Maschinen beim Interpretieren der CSP-Spezifikationen für jedes CSP-Event erzeugt bzw. gelesen werden. Raw Events sind Repräsentationen der CSP-Events in einer von dem Interface Module verarbeitbaren Datenstruktur. In unserem Fall sind letzteres Zeichenketten. Beispielsweise wird das AM-Event, das in Form des Integer-Wertes 50 das CSP-Event `cmd.setfacl` repräsentiert, vom CCL in die Zeichenkette “`setfacl`” umgewandelt. Im Prozeß `vvtsndccl` wird die Funktion `vvtAm2Raw(...)` zur Umwandlung von AM Events in Raw Events aufgerufen. Umgekehrt wird im Prozeß `vvtrcvccl` die Funktion `vvtRaw2Am(...)` aufgerufen, um Raw Events in AM Events umzuwandeln. Diese Funktionen arbeiten mit internen Tabellen, die durch die Funktion `vvtInitTables(...)` initialisiert werden. Letztere erhält als Parameter die Namen verschiedener Konfigurationsdateien. In unserem Fall ist hier die Datei `ae2raw.txt` wichtig. Hier wird jedem `AM_INPUT`- und `AM_OUTPUT`-Event eine Bytefolge zugewiesen. Die Datei enthält für jedes dieser Events eine Zeile. Jede Zeile ist folgendermaßen aufgebaut:

```
am-addr eventID Name ctp dtp clen byte(hex) ...
                                          ... byte(hex) <ret>
```

Dabei bedeuten:

am-addr	Die ID der abstrakten Maschine, zu der das Event gehört.
eventID	Die interne Repräsentation des Events. Dieser Integer-Wert ist der Alphabet-Datei zu entnehmen.
Name	Der Name des Events, ergibt sich aus der CSP-Spezifikation und ist ebenfalls in der Alphabet-Datei angegeben.
ctp	Command Type: Dieser kann verwendet werden, um eine Zuordnung zu einer bestimmten abstrakten Maschine oder einem bestimmten Interface Module zu realisieren. Dieser Wert ist in unserem Fall unbedeutend und immer 1.
dtp	Data Type: Dieser ist dazu gedacht, das Datenformat des Raw-Event zu spezifizieren. Das kann dazu verwendet werden, Format-Umwandlungen durchzuführen (z.B. Big Byte-Endian - Little Byte-Endian). Dieser Wert ist in unserem Fall unbedeutend und immer 0.
clen	Länge der Bytefolge.
byte(hex)...	Es folgen die Werte der einzelnen Bytes angegeben in textueller Hexadezimaldarstellung und getrennt durch Leerzeichen.

Beispielsweise sieht die Zeile für das CSP-Event `cmd.setfacl` wie folgt aus:

```
1 50 cmd.setfacl 1 0 8 73 65 74 66 61 63 6C 20
```

Der vollständige Inhalt dieser Datei ist im Anhang C.2 angegeben.

Neben dieser Umwandlung ist in unserem Fall auch eine Kommunikation zwischen den einzelnen abstrakten Maschinen nötig. Dazu prüft der Prozeß `vvtsndccl`, nachdem er ein Event von einer AM erhalten hat, mittels der Funktion `vvtAm2Am(...)`, ob dieses Event an eine andere AM gesendet werden soll. Wenn dies der Fall ist, wird es der anderen AM statt dem IFM gesendet. Auch diese Funktion arbeitet mit einer durch die Funktion `vvtInitTables(...)` initialisierten Tabelle. Dazu wird die Datei `ae2mae.txt` gelesen. Diese Datei enthält für jedes Event, das an eine andere AM gesendet werden soll, eine Zeile mit folgendem Aufbau:

```
am-addr eventID am-addr_1 eventID_1 ... am-addr_N eventID_N <ret>
```

Dabei bedeuten die einzelnen Felder:

am-addr	AM-ID der AM, von der das Event stammt
eventID	ID des Events, das zu anderen AMs gesendet werden soll
am-addr_i	AM-ID der i-ten AM, an die das Event gesendet werden soll.
eventID_i	Event-ID, auf die das Event in dieser AM abgebildet werden soll.

Es können mehrere solcher `am_addr`, `eventID`-Paare für ein Event angegeben wird. Dies kann dazu verwendet werden, ein Event einer AM an mehrere andere abstrakten Maschinen zu senden (Multicast). Letztere Funktion haben wir nicht gebraucht. Der Inhalt dieser Datei ist Anhang C.3 angegeben.

Die Funktion der Eventabbildung von AM nach AM ist nicht in der Demo-Version des `vvtsndccl`-Prozeß implementiert. Daher mußten wir das Programm entsprechend erweitern. Das Listing befindet sich im Anhang D.2.

19.3.3 Konfiguration des Interface Module (IFM)

Das Interface Module liest über ein Socket die vom CCL gesendeten Kommandozeilen. Es hat hier lediglich die Aufgabe, das Testprogramm mit der jeweils gelesenen Kommandozeile aufzurufen, den Exit-Status des Testprogramms zu ermitteln und an das CCL zurückzusenden. Dabei protokolliert es Kommandozeile und Rückgabewert in einer Log-Datei. Das Listing dieses Programms befindet sich im Anhang D.3.

19.4 ASpecT-Spezifikation

ASpecT ist eine Implementierung für eine Teilmenge algebraischer Spezifikationen von abstrakten Datentypen. ASpecT wurde von Dr. Richard Seifert und Dr. Jörn von Holten an der Universität Bremen entwickelt. Das System wurde entworfen, so benutzerfreundlich wie möglich zu sein, eingeschlossen der Möglichkeit des Überladen und einem Quellcode-Debugger. Über die Jahre wurden immer mehr Leistungmerkmale hinzugefügt, unter anderem Untersorten, Funktionale und eingeschränkter Polymorphismus. Der ASpecT-Compiler übersetzt den funktionalen Quellcode nach C, mit dem Ergebnis von schnellen und effizienten, ausführbaren Programmen. ASpecT wurde auf viele verschiedene Plattformen portiert, unter anderem Sun3, Sun4, Dec VAX, IBM RS6000, NeXT, Apple A/UX, PC (OS/2, Linux), Amiga und Atari ST/TT. Heute wird der ASpecT-Compiler nicht mehr gepflegt und ist auch nicht mehr öffentlich verfügbar.

Für den automatisierten Test setzen wir Teile der Z-Spezifikation in eine ASpecT Spezifikation um. Dazu gehört:

- Eine ACL und Validität einer ACL.
- Die Zugriffskontrolle durch eine ACL.

Anhand der POSIX-Spezifikation entwickeln wir außerdem eine ASpecT-Spezifikation des Verhaltens von `setfacl`. Bei der Spezifikation berücksichtigen wir bereits die spätere Integration in das Testsystem.

19.4.1 ACL

Wir beginnen mit der Spezifikation einer ACL, zur Z-Spezifikation siehe 17.1.1. ASpecT verfügt über algebraische Datentypen und Listen, um Typen zu konstruieren. Wir spezifizieren ein ACL-Eintrag durch einen algebraischen Datentyp `entry`:

```
entry ::= entry type::string tag::string mode::string.
```

Der Datentyp verfügt über einen einzigen Konstruktor, der ebenfalls mit `entry` benannt ist. Der Datentyp hat drei Felder vom Typ `string`. ASpecT erlaubt es, Felder von Datentypen zu benennen. Dadurch werden jeweils zwei Funktionen des selben Namens definiert. Diese können benutzt werden, um ein Feld auszuwählen oder zu ändern. Die hier gewählte Benennung der Felder entspricht derjenigen der Z-Spezifikation.

Wir spezifizieren eine ACL als eine Liste von `entry`:

```
acl ::= [entry].
```

Die Wahl des Datentyps `string` für die Konstruktion von `entry` vereinfacht die Interaktion mit den Skripten des Testsystems. Wir definieren jedoch eine Operation `type`, die den Typ einer ACL, angegeben als `string`, auf den folgenden Aufzählungstyp abbildet:

```
acl_type ::= acl_user_obj
           | acl_user
           | acl_group_obj
           | acl_group
           | acl_other
           | acl_mask.
```

Die alternativen Konstruktoren für einen Datentyp werden durch `|` getrennt. Ein Aufzählungstyp ist also nichts anderes als ein algebraischer Datentyp, dessen Konstruktoren 0 Felder haben. Wir geben hier lediglich die Signatur der Operation `type` an:

```
OPNS type :: entry -> acl_type.
```

Das Feld `tag` eines ACL-Eintrags diente zur Angabe eines Benutzers oder einer Gruppe. Hier wird offengelassen, ob diese numerisch oder namentlich angegeben werden. Die Angabe wird ohne Änderung verwendet: Es findet z. B. keine Abbildung von Namen von Benutzern auf IDs von Benutzern statt. Die Konsistenz der Art der Angabe muß vom Testsystem gewährleistet werden.

Der `string` für die Zugriffsrechte wird als Liste mit dem entsprechenden Buchstaben für jedes Zugriffsrecht behandelt.

19.4.2 ACL-Validation

Wir spezifizieren nun eine Operation `acl_valid`, die prüft, ob eine ACL valid ist. Die entsprechende Z-Spezifikation findet man in 17.2.2. Die Signatur der Operation lautet:

```
OPNS acl_valid :: acl -> boolean -> boolean.
```

`boolean` ist ein eingebauter Typ zur Darstellung von Wahrheitswerten. Die Operation gibt zurück, ob die ACL valid ist: `true` bzw. `false`.

Die Operation hat zwei Argumente. Das erste Argument ist die ACL, deren Validität geprüft wird. Mit dem zweiten Argument hat es die folgende Bewandnis: Wenn ein Verzeichnis keine Default-ACL hat, so wird dies von der POSIX-konformen Ausgabe von `getfacl` nicht besonders hervorgehoben. Dieser Umstand ist nur daran erkennbar, daß eine leere ACL nicht valid ist. Entsprechend wurde das Verhalten von `setfacl` dahingehend gegenüber POSIX ergänzt, daß es möglich ist, eine leere Default-ACL zu setzen. Die leere ACL wird von `setfacl` jedoch nicht tatsächlich gesetzt. Dies würde der Linux-Kernel als Fehler werten. Stattdessen löscht `setfacl` die Default-ACL. Das Ergebnis stellt sich wiederum als leere ACL dar. Diese

Operation wurde jedoch als zulässig akzeptiert. Wir trugen den Umständen Rechnung und akzeptieren eine leere Default-ACL. Dazu müssen wir aber eine Default-ACL von einer Access-ACL unterscheiden können. Dies gibt das zweite Argument an.

Die Operation ist wie folgt definiert:

```
EQNS acl_valid (Acl::acl) IsDefault =
  (IsDefault && ismt Acl) || (
  single (filter (type;(==) acl_user_obj) Acl) &&
  single (filter (type;(==) acl_group_obj) Acl) &&
  single (filter (type;(==) acl_other) Acl) &&
  (Acl == mkset (equals) Acl) &&
  (not (exist (type;member [acl_user,acl_group]) Acl) ||
   single (filter (type;(==) acl_mask) Acl))).
```

Die Operation `filter` erhält ein Funktional und eine Liste. Die Funktion gibt eine Liste mit sämtlichen Elementen der übergebenen Liste, die einem Auswahlkriterium genügen, zurück. Als Auswahlkriterium dient dabei das Funktional.

Der Operator `;` konkateniert Funktionen. Der Ausdruck

```
(==) acl_user_obj
```

ist eine sogenannte Sektion, ein nicht vollständig angewendeter Operator. Das Funktional

```
(type;(==) acl_user_obj)
```

bildet also einen `entry` auf `acl_type` ab und vergleicht das Ergebnis mit `acl_user_obj`. `single` testet, ob eine Liste genau ein Element enthält. Somit wird also getestet, ob die Basiseinträge genau einmal vorhanden sind.

Die Operation `mkset` entfernt doppelte Elemente anhand eines Tests auf Gleichheit. Dazu wird die Operation `equals` als Test verwendet:

```
OPNS equals :: entry -> entry -> boolean.
EQNS equals A B = (tag A == tag B) && (type A == type B::acl_type).
```

Durch

```
  (Acl == mkset (equals) Acl)
```

wird also getestet, ob sich gleiche Einträge entfernen lassen, d.h. Einträge mit gleichem `type` und `tag`. Dies darf für eine valide ACL nicht der Fall sein. Zuletzt wird getestet ob es sich um eine Extended ACL handelt, sprich eine mit `acl_user`- oder `acl_group`-Eintrag. Wenn ja, muß ein `acl_mask`-Eintrag vorhanden sein. Die Sortierung wird nicht getestet, da ACLs außerhalb des Linux-Kernels beliebig geordnet sein dürfen.

19.4.3 Gleichheit auf ACLs

Beim automatisierten Test werden wir ACLs auf Gleichheit prüfen müssen, um das Ergebnis eines Tests zu ermitteln. Zwei ACLs sind gleich, wenn sie aus den gleichen ACL-Einträgen bestehen. Die Sortierung ist dabei nicht signifikant. Die folgende Operation testet zwei ACLs auf Gleichheit:

```
OPNS equals :: acl -> acl -> boolean.
EQNS equals A B =
  ((--) A B equals_mode == []) &&
  ((--) B A equals_mode == []).
```

```
OPNS equals_mode :: entry -> entry -> boolean.
EQNS equals_mode A B =
  forall (s (member (chars (mode A));(==)) (member (chars (mode B))))
         (['r','w','x']::chars) &&
  equals A B.
```

Die Operation `--` hat die polymorphe Signatur:

```
OPNS (--) :: [a] -> [a] -> (a -> a -> boolean) -> [a].
```

`--` gibt die erste Liste ohne die Elemente, die gleich einem Element der zweiten Liste sind, zurück. Dabei dient zum Vergleich das angegebene Funktional, hier `equals_mode`. `equals_mode` testet, ob zwei ACL-Einträge bzgl. Zugriffsrechte und der Felder `type` und `tag` (in der Operation `equals`) gleich sind.

19.4.4 Spezifikation einer Kommandozeile

Hier spezifizieren wir nun die Kommandozeile von `setfacl`, die vom Testsystem generiert wird. Die Optionen von `setfacl` geben mehrere Möglichkeiten, eine ACL zu manipulieren. Unter anderem lassen sich ACL-Einträge aus Dateien nutzen. Dieses Einlesen halten wir der Übersichtlichkeit halber aus der Spezifikation heraus. Stattdessen liest das Testsystem diese Dateien aus und übergibt die ACL-Einträge direkt der ASpecT-Spezifikation. Die Eingabe der Spezifikation wird aus `options`, einer Liste von `option`, gebildet.

```
options ::= [option].
option ::= option string
         | option string acl.
```

Eine Option kann aus einem `string`, z.B. `"-d"`, gebildet werden, oder einem `string` und einer Liste vom Typ `acl`. Es handelt sich dabei im allgemeinen nicht um eine valide ACL, sondern um eine nicht eingeschränkte Liste von erweiterten ACL-Einträgen.

19.4.5 Auswertung der Kommandozeile

`setfacl` wendet die Optionen auf die übergebene Access- und Default-ACL an. Das Ergebnis sind zwei neue ACLs. Dieses Verhalten spezifizieren wir mit den folgenden Operationen.

```
OPNS process_options ::
  (acl,acl) -> options -> (acl,acl).
EQNS process_options (Access,Default) Options =
  let
    D = member Options (option "-d").
    (Access,Default) =
      foldl (process_option D)
        (Access,Default)
        (remove (member [option "-d",option "-n"]) Options).
  in
    if D && do_recalc_mask Options && not (ismt Default)
    then (Access, recalc_mask Default)
    elsif do_recalc_mask Options
    then (recalc_mask Access, Default)
    else (Access,Default).
```

Die Liste der Optionen wird mit `foldl` von links nach rechts, also in der Reihenfolge ihres Auftretens, mit `process_option` gefaltet. Dabei dient die Access- und die Default-ACL als Akkumulator. Dieser wird entsprechend jeder Option geändert. Die Optionen werden also entsprechend POSIX in der Reihenfolge ihres Auftretens ausgewertet. Die Optionen `-d` und `-n` bilden eine Ausnahme: Einige Optionen können sich sowohl auf die Default-ACL als auch die Access-ACL beziehen. Die Option `-d` steuert, auf welche ACL sich diese Optionen beziehen. Die `-n` Option steuert, ob der `acl_mask`-Eintrag rekalkuliert werden darf. An welcher Stelle diese Optionen gegeben werden, ist jedoch nicht signifikant. Die übrigen Optionen werden von der Operation `process_option` verarbeitet.

```
OPNS process_option :: boolean -> (acl,acl) -> option -> (acl,acl).
EQNS process_option A B (option "-b") = option_b A B.
  process_option A B (option "-k") = option_k A B.
  process_option A B (option "-m" Modify) = option_m Modify A B.
  process_option A B (option "-M" Modify) = option_M Modify A B.
  process_option A B (option "-x" Modify) = option_x Modify A B.
  process_option A B (option "-X" Modify) = option_X Modify A B.
 $process_option _ _ (option C _) = error("Unknown option " ++ C).
```

Wie man sieht, stellt `process_option` lediglich fest, um welche Option es sich handelt. Die eigentliche Arbeit übernehmen die Operationen `option_?`.

```
OPNS option_b,option_k :: boolean -> (acl,acl) -> (acl,acl).
     option_m,option_M,option_x,option_X :: acl -> boolean ->
     (acl,acl) -> (acl,acl).
```

Jede dieser Operationen bekommt als `boolean` durchgereicht, ob die `-d` Option gegeben wurde. Bei den Optionen `-m`, `-M`, `-x`, `-X` wird eine ACL angegeben. Bei den großgeschriebenen Varianten liest `setfacl` die ACL jeweils aus einer Datei. Uns wird hier jedoch bereits der Dateiinhalt übergeben.

Option `-b`

Die Option `-b` löscht alle Einträge außer den Basiseinträgen aus der Access- bzw. Default-ACL:

```
EQNS
  option_b D (Access,Default) =
    if D
    then (Access, filter (is_base) Default)
    else (filter (is_base) Access, Default).
```

Die Basiseinträge sind `acl_user_obj`, `acl_group_obj`, `acl_other`:

```
OPNS is_base :: entry -> boolean.
EQNS is_base = type;member [acl_user_obj,acl_group_obj,acl_other].
```

Option `-k`

Die Option `-k` löscht die Default-ACL. Die Spezifikation löscht alle Einträge aus der Default-ACL. Siehe dazu auch 19.4.2.

```
  option_k _ (Access,_) = (Access,[]).
```

Optionen `-m` und `-M`

Die Optionen `-m` und `-M` suchen zu jedem angegebenen ACL-Eintrag einen Eintrag in der vorhandenen ACL mit gleichem `type` und `tag` und modifizieren diesen Eintrag. Wird ein solcher Eintrag nicht gefunden, wird ein neuer Eintrag hinzugefügt.

```
option_m Modify D (Access,Default) =
  if D
  then (Access, foldl (modify) Default Modify)
  else (foldl (modify) Access Modify, Default).
option_M = option_m.
```

Die ACL-Einträge, die dazu dienen, die Modifikation einer ACL zu spezifizieren, sind gegenüber den normalen ACL-Einträgen erweitert:

1. Absolute Einträge, die wie die normalen ACL-Einträge mit ihren Zugriffsrechten angegeben werden.
2. Relative Einträge, die die angegebenen Zugriffsrechte zu den Zugriffsrechten des vorhandenen Eintrags hinzufügen. Diese haben ein vorgestelltes +. Ist kein passender Eintrag vorhanden, ist das Verhalten wie beim absoluten Eintrag.
3. Relative Einträge, die die angegebenen Zugriffsrechte von den Zugriffsrechten des vorhandenen Eintrags abziehen. Diese haben ein vorgestelltes Dach (`^`). Ist kein passender Eintrag vorhanden, wird ein Eintrag ohne Zugriffsrechte erzeugt.

Die eigentliche Modifikation wird durch die Operation `modify` spezifiziert:

```
OPNS modify :: acl -> entry -> acl.
EQNS modify [H|T] Modify =
  if Modify _equals H
  then
    if member (chars (mode Modify)) '+'
    then [mode(H,string (mkset (chars (mode H) ++
                                chars (mode Modify) --
                                ['+'])) | T]
    elsif member (chars (mode Modify)) '^'
    then [mode(H,string (mkset (chars (mode H) --
                                chars (mode Modify)))) | T]
    else [Modify|T]
  else [H|modify T Modify].

modify [] Modify =
  if member (chars (mode Modify)) '+'
  then [mode(Modify,string (mkset (chars (mode Modify)) -- ['+']))]
  elsif member (chars (mode Modify)) '^'
  then [mode(Modify,"")]
  else [Modify].
```

Optionen `-x` und `-X`

Die Optionen `-x` und `-X` suchen zu jedem angegebenen ACL-Eintrag einen Eintrag in der vorhandenen ACL mit gleichem `type` und `tag` und löschen diesen Eintrag, wenn vorhanden.

```
option_x Modify D (Access,Default) =
  if D
  then (Access,remove (member (equals) Modify) Default)
  else (remove (member (equals) Modify) Access,Default).

option_X = option_x.
```

Rekalkulation der ACL-Maske

Nachdem sämtliche Optionen abgearbeitet worden sind, wird anhand der gegebenen Optionen entschieden, ob der `acl_mask`-Eintrag rekalkuliert werden soll:

```
OPNS do_recalc_mask :: options -> boolean.
EQNS do_recalc_mask Options =
  exist (missing_mask) Options && not (member Options (option "-n")).

OPNS missing_mask :: option -> boolean.
EQNS missing_mask (option "-m" A) =
  not (exist (type;member ["m","mask"]) A).
missing_mask (option "-M" A) =
  not (exist (type;member ["m","mask"]) A).
missing_mask (option "-x" A) =
  not (exist (type;member ["m","mask"]) A).
missing_mask (option "-X" A) =
  not (exist (type;member ["m","mask"]) A).
$missing_mask _ = false.
```

Wurde eine `-m`, `-M`, `-x` oder `-X` Option mit einer ACL ohne `acl_mask`-Eintrag gegeben und nicht die Option `-n` spezifiziert, wird der `acl_mask`-Eintrag rekalkuliert. D. h. es wird ein ggf. neuer `acl_mask`-Eintrag erzeugt, dessen Zugriffsrechte genau die Summe der Zugriffsrechte der Einträge vom Typ `acl_user`, `acl_group` oder `acl_group_obj` der resultierenden ACL sind.

```
OPNS recalc_mask :: acl -> acl.
EQNS recalc_mask Acl =
  let
    Acl = remove (type;(==) acl_mask) Acl.
    GroupClass =
      filter (type;member [acl_user,acl_group,acl_group_obj])
             Acl.
    GroupClassMode =
      string
      (mkset (foldr (mode;chars;(++))
      ([]::chars) GroupClass) -- '-').
  in
    entry "mask" "" GroupClassMode : Acl.
```

19.4.6 ACL-Zugriffskontrolle

Hier spezifizieren wir nun, ob eine ACL einen bestimmten Zugriff zuläßt. Die Spezifikation ist möglichst eng an den Pseudocode (siehe S. 71 in [Sec97a] angelehnt.

Datentypen

Der Zugreifende ist durch den Datentyp `user` modelliert, entsprechend dem Z-Schema *PROCESS* (siehe Abschnitt 17.1.2):

```
user ::= user euid::string egid::string groups::strings.
```

Das Ziel des Zugriffs wird mit `object` modelliert, entsprechend dem Z Schema FILE (siehe Abschnitt 17.1.3):

```
object ::= object uid::string gid::string.
```

Operationen

Die Operation, die den Zugriff kontrolliert, erhält als Parameter

1. den Prozeß, dessen Zugriff kontrolliert wird,
2. das Ziel des Zugriffs,

3. einen String mit den geforderten Zugriffsrechten,
4. eine ACL.

Die Operation gibt zurück, ob die geforderten Zugriffsrechte gewährt werden. Die Operation ist wie folgt spezifiziert:

```
OPNS perm_check ::
  user -> object -> string -> acl -> boolean.
EQNS perm_check User Object Mode Acl =
  if euid User == uid Object
  then matched_entry (mu (type;(==) acl_user_obj) Acl) Mode Acl
  elsif exist (matching_acl_user User) Acl
  then matched_entry (mu (matching_acl_user User) Acl) Mode Acl
  elsif member (egid User : groups User) (gid Object) ||
        exist (matching_acl_group User) Acl
  then
    if exist (granting_group_entry User Object Mode) Acl
    then matched_entry
           (mu (granting_group_entry User Object Mode) Acl)
           Mode
           Acl
    else false
  elsif exist (granting_acl_other Mode) Acl
  then matched_entry (mu (granting_acl_other Mode) Acl) Mode Acl
  else false.
```

Zuerst wird in der Reihenfolge `acl_user_obj`, `acl_user`, `acl_group_obj`, `acl_group`, `acl_other_obj` nach einem passenden Eintrag in der ACL gesucht. Entweder wird ein passender Eintrag gefunden und dann die Operation `matched_entry` auf diesem Eintrag angewendet, um das Ergebnis zu ermitteln, oder der Zugriff wird verweigert.

1. Wenn die effektive UID des Subjektes gleich der UID des Objektes ist, wird ein `acl_user_obj`-Eintrag[1] als passend ausgewählt:

   ```
   if euid User == uid Object
   then matched_entry (mu (type;(==) acl_user_obj) Acl) Mode Acl
   ```

   Die Operation `mu` entspricht dem μ-Operator in Z, mit dem Unterschied, daß die Auswahl deterministisch ist.

2. Ansonsten wird, wenn vorhanden, ein passender `acl_user`-Eintrag verwendet:

   ```
   elsif exist (matching_acl_user User) Acl
   then matched_entry (mu (matching_acl_user User) Acl) Mode Acl
   ```

   D.h. ein Eintrag vom Typ `acl_user` muß als `tag` die effektive UID des Subjektes haben:

   ```
   OPNS matching_acl_user :: user -> entry -> boolean.
   EQNS matching_acl_user User Entry =
     type Entry == acl_user && (tag Entry == euid User).
   ```

3. Darauf wird getestet, ob die Gruppeneinträge, d. h. `acl_group_obj`- oder `acl_group`-Einträge, für das Subjekt gültig sind:

   ```
   elsif member (egid User : groups User) (gid Object) ||
         exist (matching_acl_group User) Acl
   ```

   Dies ist der Fall, wenn die effektive GID des Subjektes oder eine der zusätzlichen Gruppen des Subjektes gleich der GID des Objektes

[1] Diese Auswahl ist bei einer validen ACL eindeutig.

ist oder ein `acl_group`-Eintrag existiert, der die folgende Operation `matching_acl_group` erfüllt:

```
OPNS matching_acl_group :: user -> entry -> boolean.
EQNS matching_acl_group User Entry =
  type Entry == acl_group &&
  member (egid User : groups User) (tag Entry).
```

Die Operation ist erfüllt, wenn ein `acl_group`-Eintrag als `tag` eine der Gruppen des Subjektes hat. Existiert ein Gruppeneintrag, der die Zugriffsrechte gewährt, wird dieser als passend ausgewählt. Andernfalls, wenn ein solcher Eintrag nicht exisiert, wird kein Zugriff gewährt:

```
  then
    if exist (granting_group_entry User Object Mode) Acl
    then matched_entry
           (mu (granting_group_entry User Object Mode) Acl)
           Mode
           Acl
    else false
```

Die folgende Operation ermittelt, ob ein Gruppeneintrag die Zugriffsrechte gewährt:

```
OPNS granting_group_entry ::
  user -> object -> string -> entry -> boolean.
EQNS granting_group_entry User Object Mode Entry =
  (type Entry == acl_group_obj &&
   member (egid User : groups User) (gid Object) &&
   subseteq (mode Entry) Mode) ||
 (type Entry == acl_group &&
  member (egid User : groups User) (tag Entry) &&
  subseteq (mode Entry) Mode).
```

Dabei ist Operation `subseteq` (Teilmenge) wie folgt definiert:

```
OPNS subseteq :: string -> string -> boolean.
EQNS subseteq A B =
  member (chars A) (remove ((==) '-') (chars B)).
```

4. Ist kein Gruppeneintrag für das Subjekt gültig, wird ein `acl_other`-Eintrag, wenn dieser Zugriff gewährt, ausgewählt. Ansonsten wird kein Zugriff gewährt.

```
  elsif exist (granting_acl_other Mode) Acl
  then matched_entry (mu (granting_acl_other Mode) Acl) Mode Acl
  else false.
```

Diese Operation stellt fest, ob ein `acl_other`-Eintrag Zugriff gewährt:

```
OPNS granting_acl_other :: string -> entry -> boolean.
EQNS granting_acl_other Mode Entry =
  type Entry == acl_other && subseteq (mode Entry) Mode.
```

Wurde ein passender Eintrag ausgewählt, wird die folgende Operation `matched_entry` auf den ausgewählten Eintrag angewendet:

```
OPNS matched_entry :: entry -> string -> acl -> boolean.
EQNS matched_entry Entry Mode Acl  =
  if subseteq (mode Entry) Mode
  then
    if (type Entry == acl_user_obj) ||
       (type Entry == acl_other)
    then true
```

```
    elsif
      not (exist (type;(==) acl_mask) Acl) ||
      subseteq
        (mode (mu (type;(==) acl_mask) Acl))
        Mode
    then true
    else false
  else false.
```

Ist der passende Eintrag vom Typ `acl_user_obj` oder `acl_other` oder enthält die ACL keinen `acl_mask`-Eintrag, wird Zugriff gewährt. Ansonsten wird nur Zugriff gewährt, wenn die geforderten Zugriffsrechte auch im `acl_mask`-Eintrag enthalten sind.

19.5 CSP-Spezifikation

CSP bietet die Möglichkeit, nicht-deterministische Abläufe zu spezifizieren. Wir haben dieses Charakteristikum genutzt, um Automaten zu spezifizieren, die zufallsgesteuert Zeichenketten erzeugen, die den gewünschten syntaktischen Regeln entsprechen. Dies verwenden wir, um syntaktisch korrekte Programmaufrufe für das `setfacl`-Kommando zu spezifizieren und mit Hilfe des Testsystems RT-Tester auch zu erzeugen.

Um das Testen von nicht relevanten Testfällen zu vermeiden, haben wir bei dieser Spezifikation nicht alle syntaktisch korrekten Aufrufe zugelassen:

- Es werden nur die Namen vorhandener Dateien als Parameter angegeben.
- Nach Angabe der Optionen `-d` oder `-k` werden nur Verzeichnisnamen als Parameter angegeben.
- Es werden nur ACLs mit gültiger Länge, d.h. mindestens ein Eintrag und maximal 30, erzeugt.
- Benutzernamen und Benutzer-IDs in den ACL-Einträgen sind gültige im System vorhandenen Benutzer.

19.5.1 Spezifikation einer Kommandozeile

Der CSP-Prozeß `CL` spezifiziert eine unendliche Folge von syntaktisch korrekten Kommandozeilen zum Aufruf des `setfacl`-Kommandos. Eine Kommandozeile beginnt mit dem Kommandonamen gefolgt von den Parametern:

```
setfacl [-bdkn] [-m entries] [-M file1]
        [-x entries] [-X file2] [file...]
```

Die Parameter werden durch den CSP-Prozeß `MKPAR` erzeugt. Dieser erhält als Parameter die Menge aller möglicher Optionen `OPTIONS`. Eine Kommandozeile endet mit Carriage Return (`cr`). Anschließend erzeugt das aufgerufene Testprogramm eines der Events `execOk` oder `testError`. Durch das Auftreten des Events `testError` erkennt RT-Tester einen Fehler bei der Testdurchführung. Damit der Test im letzteren Fall weitergeht, wird mit Hilfe der Events `setT.9` ein Timer gestartet. Tritt, bevor der Timer abgelaufen ist, das Event `execOk` auf, wird dieser wieder zurückgesetzt und der Prozeß `CL` rekursiv aufgerufen. Andernfalls erzeugt der Timer das Event `elaT.9` und ruft ebenfalls den `CL`-Prozeß rekursiv auf. Der Timer ist so lang eingestellt, daß er nicht abgelaufen ist, bevor eine Rückmeldung vom Testsystem erfolgt ist. Er dient also nicht dazu, Zeitbedingungen zu testen, sondern dazu, den Test auch nach dem Auftreten eines Fehlers weiter durchzuführen.

```
CL= cmd.setfacl -> MKPAR(OPTIONS); cr -> setT.9 -> (
      (execOK -> resT.9 ->CL)
     []
      (elaT.9 -> CL)
)
```

Der Prozeß `MKPAR` erzeugt die Parameter für den Kommandoaufruf. Zuerst werden ein oder mehrere der möglichen Optionen angegeben. Es wäre zwar syntaktisch korrekt, keine Option anzugeben, dies würde jedoch zu einer Reihe von nicht sinnvollen Testfällen führen. Nach den Optionen folgen dann ein oder mehrere Namen von normalen Dateien oder Verzeichnissen.

```
MKPAR(OPT0) =
  ( OPT0 != {} )&
    (|~| opt: OPT0 @ (
      option.opt ->
      if(member(opt,{x,m})) then (
       MKACL; MKPAR(diff(OPT0,{opt}))
      ) else (
        if(member(opt,{X,M})) then (
          |~| fileid: ACLFILEID @ (
              aclfile.fileid ->
              MKPAR(diff(OPT0,{opt}))
          )
         ) else
          MKPAR(diff(OPT0,{opt}))
      )
     )
    )
  []
  ( OPT0 != OPTIONS )&
    ( if((member(d,OPT0)) and (member(k,OPT0))) then (
        FILENAMES(FILEID)
       ) else (
        DIRNAMES(DIRID)
      )
    )
```

Damit in einer Kommandozeile jede Option maximal einmal angegeben wird, erhält der Prozeß `MKPAR` als Parameter die Menge der möglichen Optionen `OPT0`. Diese enthält initial die Menge aller zugelassenen Optionen `OPTIONS`.

```
OPTIONS= {b,d,k,n,m,M,x,X}
```

Bei jedem rekursiven Aufruf wird die zuletzt ausgewählte Option aus dieser Menge mit Hilfe der Funktion `diff` entfernt:

```
MKPAR(diff(OPT0,{opt}))
```

Der Prozeß `MKPAR` muß mindestens eine Option erzeugen und kann jede der möglichen Optionen maximal einmal erzeugen, bevor er durch Aufruf einer der CSP-Prozesse `FILENAMES` oder `DIRNAMES` anfängt, Datei- bzw. Verzeichnisnamen als weitere Parameter zu erzeugen. Dieses Verhalten wird mittels des Internal-Choice-Operators (`[]`) gesteuert. Damit die eben beschriebenen Bedingungen erfüllt werden, sind die beiden Alternativen mit jeweils einem Guard versehen. Ein Guard ist eine logische Bedingung, die erfüllt sein muß, damit die Alternative, die damit versehen ist, gewählt werden kann.

Die obere Alternative, die zum Erzeugen weiterer Optionen führt, ist mit dem Guard `( OPTO != {} )` versehen. Dieser Zweig kann also nur gewählt werden, wenn die Menge `OPTO` nicht leer ist - also noch mindestens eine Option enthält, die in der bisher erzeugten Kommandozeile noch nicht enthalten ist. Solange dies der Fall ist, können weitere Optionen in die Kommandozeile eingefügt werden. In diesem Fall wird eine Option aus der Menge ausgewählt. Handelt es sich bei der Option um die Option `-x` oder `-m`, erhält diese als Parameter eine ACL. Diese wird durch den später beschriebenen Prozeß `MKACL` erzeugt. Die Optionen `-X` oder `-M` erhalten als Argument keine ACLs sondern stattdessen einen Dateinamen. Dieser bezeichnet eine Datei, die eine ACL in Textform enthält. Wird eine dieser beiden Optionen ausgewählt, erhält sie als Parameter einen der Dateinamen aus der Menge `ACLFILEID`. Diese Dateien haben wir vor der Testdurchführung angelegt, sie enthalten valide ACLs in Textform. In einem erweiterten Test könnte man diese Dateien auch automatisiert erstellen. Alle anderen Optionen sind nicht parametrisiert. Nach Erzeugung der Option und ggf. des Optionsparameters wird der Prozeß `MKPAR` rekursiv aufgerufen, wobei als Argument die Menge der nun noch möglichen Optionen angegeben wird.

Die untere Alternative, die dazu führt, daß nun Datei- bzw. Verzeichnisnamen erzeugt werden, ist mit dem Guard `( OPTO != OPTIONS )` versehen. Diese Bedingung ist dann erfüllt, wenn bereits mindestens eine Option erzeugt wurde. Wenn dies erfüllt ist, kann sich der Prozeß `MKPAR` dazu entscheiden, keine weiteren Optionen mehr zu erzeugen und mit dem Generieren von Datei- bzw. Verzeichnisnamen zu beginnen. Spätestens dann, wenn alle zulässigen Optionen angegeben wurden (`OPTO == {}`), muß sich `MKPAR` für diesen Zweig entscheiden, da dann der obere Zweig nicht mehr zur Verfügung steht.

Wenn eine der Optionen `-k` oder `-d` in der bisherigen Kommandozeile erzeugt wurde, ist es nicht mehr sinnvoll als Parameter Namen von normalen Dateien anzugeben, da sich diese Optionen auf Default ACLs beziehen. Nur Verzeichnisse besitzen aber Default ACLs. Hier wurden die Aufrufe gegen die Menge der syntaktisch korrekten Aufrufe dahingehend eingeschränkt, daß sofern die Option `-k` oder `-d` zuvor erzeugt wurde, nur Verzeichnisnamen angegeben werden. Wenn keine dieser Option angegeben wurde, können Datei- oder Verzeichnis-Namen angegeben werden. Letzteres ist der Fall, wenn beide Optionen noch in der Menge der möglichen Optionen `OPTO` enthalten sind (`member(d,OPTO) and member(k,OPTO)`). Die Datei- bzw. Verzeichnisnamen werden durch die Prozesse `FILENAMES` und `DIRNAMES` generiert:

```
FILENAMES(FNO, DNO) =
         ( FNO != {} )&
            (|~| id: FNO @ (
              spc -> file.id ->
              FILENAMES(diff(FNO,{id}), DNO)
             )
            )
         []
```

```
          ( DNO != {} )&
             (|~| id: DNO @ (
               spc -> dir.id ->
               DIRNAMES(FNO, diff(DNO,{id}))
              )
             )
          []
          ((FNO != FILEID) or (DNO != DIRID ))&
                  (  SKIP)

DIRNAMES(DNO) =
          ( DNO != {} )&
             (|~| id: DNO @ (
               spc -> dir.id ->
               DIRNAMES(diff(DNO,{id}))
              )
             )
          []
          (DNO != DIRID )&
                  (  SKIP)
```

Der Prozeß `FILENAMES` erzeugt einen oder mehrere Dateinamen oder verschiedene Verzeichnisnamen. Er wird mit der Menge der definierten Dateinamen und der Menge der definierten Verzeichnisnamen aufgerufen. Er erzeugt dann nicht deterministisch einen Datei- oder Verzeichnisnamen. Anschließend ruft er sich rekursiv auf, wobei der erzeugte Dateibzw. Verzeichnisname aus der entsprechenden Menge entfernt wird. Sobald mindestens ein Name erzeugt wird und spätestens nachdem alle definierten Datei- und Verzeichnisnamen erzeugt wurden, terminiert der Prozeß.

Der Prozeß `DIRNAMES` arbeitet entsprechend, jedoch erzeugt er ausschließlich Verzeichnisnamen.

```
MKACL= goacl -> acldone -> spc -> SKIP
```

Der Prozeß `MKACL` dient dazu, die für die Optionen `-m` und `-x` nötigen ACLs zu erzeugen. Um den Zustandsraum klein zu halten, werden diese in einer eigenen abstrakten Maschine generiert. Der hier definierte Prozeß dient dazu, den in der anderen AM laufenden CSP-Prozeß zu triggern. Dazu erzeugt er das Event `goacl`. Dieser dient als Eingabeevent für den CSP-Prozeß `ACL` und veranlaßt diesen, eine ACL zu generieren. Ist er damit fertig, generiert er das Event `acldone`, worauf der hier definierte Prozeß wartet. Tritt dieser Event auf, erzeugt der Prozeß noch ein Leerzeichen und terminiert.

19.5.2 Spezifikation einer ACL

Der Prozeß `ACL` wartet zunächst auf das Event `goacl`, das von dem oben beschriebenen Prozeß `MKACL` generiert wird. Wenn dieses Event generiert wurde, wählt der Prozeß nicht-deterministisch eine Länge aus der Menge `ACLLENGTH` aus. Diese enthält alle gültigen Längen einer ACL. Anschließend wird der Prozeß `MKENTRIES` mit dieser Länge als Argument aufgerufen. Außerdem wird dem Prozeß als Parameter ein Flag übergeben, daß anzeigt, ob dies der erste Aufruf ist. Dieser Prozeß erzeugt dann eine ACL mit dieser Länge. Danach wird das Event `acldone` generiert. Dieses signalisiert dem Prozeß der anderen AM, daß die ACL generiert wurde. Dann ruft sich der Prozeß rekursiv wieder selbst auf.

```
ACL= goacl-> ( |~| length: ACLLENGTH @
     MKENTRIES(length,true)); acldone -> ACL
```

Der Prozeß `MKENTRIES` erzeugt eine Liste von ACL-Entries in der folgenden Form:

```
<acl_entry>[,<acl_entry]...
```

Die Anzahl der ACL-Einträge wird dabei mit dem Parameter `length` angegeben. Wenn diese Anzahl 0 ist, terminiert der Prozeß. Wenn der Prozeß zum ersten mal aufgerufen wird, was der Fall ist, wenn der Parameter `first` gleich `true` ist, ruft der Prozeß den Prozeß `MKENTRY` auf. Danach ruft sich `MKENTRIES` rekursiv selbst auf und setzt dabei den Parameter `first` auf `false` und dekrementiert `aclnumber`. Der Prozeß `MKENTRY` erzeugt einen ACL-Eintrag. Ist der Parameter `first` gleich `false`, wird vor dem Aufruf von `MKENTRY` ein Komma erzeugt.

```
MKENTRIES(aclnumber,first) =
  (aclnumber == 0)& (
    SKIP
  )
  []
  ((first == true) and (aclnumber != 0))& (
    MKENTRY; MKENTRIES(aclnumber-1, false)
  )
  []
  ((first == false) and (aclnumber != 0))& (
    comma->
    MKENTRY; MKENTRIES(aclnumber-1, false)
  )
```

Der Prozeß `MKENTRY` erzeugt einen ACL-Eintrag in der folgenden Form:

```
<tag_type>:[qualifier]:permission
```

Der Tag Type ist alternativ ein

- User Tag Type
- Group Tag Type
- Others Tag Type
- Mask Tag Type

Diese werden durch die später beschriebenen Prozesse `MKUTAG`, `MKGTAG`, `MKOTAG` und `MKMTAG` erzeugt. Der Qualifier ist ein Benutzer- oder Gruppenname. Dieser entfällt, wenn der Dateibesitzer oder die besitzende Gruppe spezifiziert werden soll oder der Tag Type vom Typ Others oder Mask ist. Die Rechte werden von einem CSP-Prozeß in einer separaten AM spezifiziert, um den Zustandsraum klein zu halten. Aus dem gleichen Grund wurde hier darauf verzichtet, wahlweise Benutzername oder numerische UID als Qualifier anzugeben. Dies kann auch durch ein entsprechendes Event Mapping erreicht werden.

MKENTRY entscheidet sich zunächst nicht-deterministisch für den Typ von Eintrag, der erzeugt werden soll. Hier gibt es folgende Alternativen:

1. Den Eintrag für den Dateibesitzer.
2. Den Eintrag für die besitzende Gruppe.
3. Den Eintrag für Others.
4. Den Mask-Eintrag.
5. Einen Eintrag für einen bestimmten Benutzer.
6. Einen Eintrag für eine bestimmte Gruppe.

Zunächst wird dann durch Aufruf des entsprechenden Prozesses MKUTAG, MKGTAG, MKOTAG oder MKMTAG der Tag Type spezifiziert. Anschließend wird das Event colon generiert, das dazu führt, daß in der Kommandozeile an dieser Stelle ein : erscheint. In den ersten vier Fällen wird kein Qualifier erzeugt. In den letzten beiden wird nicht-deterministisch ein Benutzer- beziehungsweise ein Gruppenname erzeugt. Bevor mit Hilfe von MKPERM die Rechte erzeugt werden, wird ein zweites colon-Event generiert.

```
MKENTRY=
   (MKUTAG; colon -> colon -> MKPERM
   |~|
   (MKGTAG; colon -> colon -> MKPERM
   |~|
   (MKOTAG; colon -> colon -> MKPERM
   |~|
   (MKMTAG; colon -> colon -> MKPERM
   |~|
   ( |~| newuser: INTERNAL_ID @ (
           MKUTAG;
           colon ->
           utag.uname.newuser ->
           colon ->
           MKPERM)
   )
   |~|
   ( |~| newgroup: INTERNAL_ID @ (
           MKGTAG;
           colon ->
           gtag.gname.newgroup ->
           colon ->
           MKPERM))
```

Die Prozesse MKUTAG, MKGTAG, MKOTAG und MKMTAG erzeugen den Tag Type. Es gibt die folgenden Tag Types:

- u[ser]
- g[roup]
- o[thers]
- m[ask]

Die Tag Types können vollständig oder in Kurzschreibweise angegeben werden. Im letzteren Fall wird nur der Anfangsbuchstabe angegeben. Um dies zu realisieren, wurde für jeden Type eine Menge definiert, die jeweils die Kurz- und die Normal-Schreibweise enthält. Die Menge `UTAGTYPE` ist beispielsweise folgendermaßen definiert:

```
UTAGTYPE= { u,user }
```

`GTAGTYPE`, `OTAGTYPE` und `MTAGTYPE` sind analog definiert. Die jeweiligen Prozesse wählen nicht-deterministisch aus der jeweiligen Menge dann eine dieser Schreibweisen aus und erzeugen das entsprechende Event, bevor sie terminieren.

```
MKUTAG= |~| tag: UTAGTYPE @ (utt.tag -> SKIP)
MKGTAG= |~| tag: GTAGTYPE @ (gtt.tag -> SKIP)
MKOTAG= |~| tag: OTAGTYPE @ (ott.tag -> SKIP)
MKMTAG= |~| tag: MTAGTYPE @ (mtt.tag -> SKIP)
```

Der Prozeß `MKPERM` triggert einen CSP-Prozeß einer anderen AM, der die Rechteangabe erzeugt. Er arbeitet wie der oben beschriebene Prozeß `MKACL`.

```
MKPERM= goperm -> permdone -> SKIP
```

19.5.3 Spezifikation der Rechte

Der Prozeß `PERM` wartet zunächst auf das Event `goperm`, das von dem oben beschriebenen Prozeß `MKPERM` generiert wird. Wenn dieses Event generiert wurde, ruft der Prozeß den lokal definierten Prozeß `MKPERM` auf. Danach wird das Event `permdone` generiert. Dieses signalisiert dem Prozeß der anderen AM, daß die Permissions generiert wurden. Schließlich ruft sich der Prozeß rekursiv wieder selbst auf.

```
PERM= goperm-> MKPERM; permdone -> PERM
```

Der hier definierte Prozeß `MKPERM` ruft nicht-deterministisch entweder den Prozeß `MKABSPERM` oder `MKRELPERM` auf. Der Prozeß `MKABSPERM` erhält als Argumente die Menge der möglichen Rechte und die Anzahl der maximal noch zu vergebenden Rechte. Die Menge der möglichen Rechte besteht aus den Rechten `r`, `w`, `x` und `void`, wobei letzteres für das Zeichen - steht.

```
MKPERM=
        MKABSPERM({r,w,x,void},3)
     |~|
        MKRELPERM
```

Der Prozeß `MKRELPERM` erzeugt nicht-deterministisch das Zeichen `cperm.add (+)` oder das Zeichen `cperm.rem (^)` und ruft anschließend den Prozeß `MKABSPERM` mit den oben beschriebenen. Parametern auf.

```
MKRELPERM=
    (cperm.add  -> MKABSPERM({r,w,x,void},3))
  |~|
    (cperm.rem  -> MKABSPERM({r,w,x,void},3))
```

`MKABSPERM` erzeugt eine mindestens ein und maximal drei Zeichen lange Zeichenkette aus den Zeichen r,w,x und -. Dabei dürfen die Zeichen r,w, und x maximal einmal vorkommen. Dazu erhält der Prozeß die Menge der noch möglichen Zeichen und die Anzahl der noch zu produzierenden Zeichen als Parameter. Sobald die Anzahl dieser Zeichen ungleich drei ist, d.h. es wurde schon ein Zeichen erzeugt, kann sich der Prozeß beenden. Solange diese Anzahl ungleich 0 ist, kann der Prozeß weitere Zeichen erzeugen. Dazu wählt er nicht deterministisch ein Zeichen aus der als Parameter übergebenen Menge `PERMSET` aus und generiert daraus ein Event. Anschließend ruft sich der Prozeß selbst rekursiv auf, wobei die Anzahl der maximal noch zu generierenden Zeichen dekrementiert wird. Außerdem wird das zuletzt generierte Zeichen aus der Menge `PERMSET` entfernt, sofern dieses nicht das Minuszeichen war, denn dieses darf mehrfach vorkommen.

```
MKABSPERM(PERMSET,nofperms)=
    (nofperms != 3)& (
       SKIP
    )
    |~|
    (nofperms != 0)& (
        |~| permchar: PERMSET @ (
           cperm.permchar ->
           if(permchar == void) then (
               MKABSPERM(PERMSET,nofperms-1)
           ) else (
               MKABSPERM(diff(PERMSET,{permchar}),nofperms-1)
           )
        )
    )
```

19.6 Ergebnisse

Rein quantitativ betrachtet wurden die meisten Fehler im manuellen Test gefunden. Beim anschließenden automatisierten Test wurden hingegen nur wenige Fehler gefunden, die jedoch eine andere Qualität aufwiesen. Dies ist schon daran erkennbar, daß sie beim vorhergehenden manuellen Test unentdeckt geblieben sind. Im nachhinein betrachtet, wäre es sehr unwahrscheinlich gewesen, diese Fehler beim manuellen Test zu finden: Zum einen waren die Testfälle, bei denen sich die Fehler auswirkten, eher dünn gestreut im Vergleich zu den Fehlern, die beim manuellen Test entdeckt wurden. Noch mehr hat sich aber ausgewirkt, daß beim manuellen Test das Testergebnis auch manuell ausgewertet werden mußte. Dabei ist es ebenfalls zu Fehlern gekommen. Diese Denkfehler ähneln ironischerweise wahrscheinlich denen, die bei der Implementation begangen worden sind.

Der manuelle Test konnte von seiner Natur her keine quantitativen Angabe über die Güte des Codes, sprich die Wahrscheinlichkeit, daß der Code noch Fehler enthält, liefern. Ebenso konnte anhand des automatisierten Test nur für die getesten Teile des Codes festgestellt werden, daß dieser wahrscheinlich keine Fehler mehr enthält. Daher ist es auch nicht weiter verwunderlich, daß wir nach dem Test nach der Veröffentlichung des Codes Fehlerberichte (engl. bug reports) von Benutzern unseres Codes erhielten. Wir konnten jedoch vermerken, daß in dem automatisiert getesten Code bisher keine Fehler gefunden wurde.

20. Ausblick

20.1 Speichermodell

Im Moment kann eine ACL nicht über mehrere Blöcke verteilt gespeichert werden. Dadurch wird die maximale Größe einer ACL durch die Blockgröße des Dateisystem begrenzt. Wünschenswert wäre das Abspeichern von ACLs in mehreren Blöcken, und dadurch die Aufhebung der Größenbeschränkung von ACLs. Allerdings ist auch abzusehen, daß bei der Zusammenarbeit mit anderen Entwicklern eine völlig andere Speichermethode entstehen wird.

20.2 Portierung

Leider kamen wir mangels Zeit und entsprechender Hardware nicht dazu, den ACL-Patch auf andere Architekturen als x86 zu portieren. Der Aufwand hierfür ist jedoch gering. Es müßte lediglich noch ein entsprechender Systemaufruf hinzugefügt werden.

Wir haben bei der Implementierung darauf geachtet, die Byte-Endian von Werten, die in einer der CPU fremden Byte-Endian, in der des Dateisystems, vorliegen können, ggf. zu konvertieren. Wir konnten dies jedoch nicht testen, weil dazu eine Portierung auf eine Architektur mit einer anderen Byte-Endian erforderlich gewesen wäre.

20.3 NFS-Unterstützung

Damit Solaris-Rechner ACLs auf NFS-Dateisystemen manipulieren können, hat Sun einen neuen RPC-Dienst namens `nfs_acl` eingeführt. Diesen Dienst könnten auch der NFS-Server und -Client von Linux nutzen, so daß ein Linux-Client per NFS ACLs auf einem Linux-Server manipulieren kann. Da die von uns implementierten ACLs und die ACLs von Solaris sich nur semantisch unterscheiden, ist die Interaktion zwischen Solaris- und Linux-Clients und -Servern unproblematisch: Die Auswertung und Validation von ACLs findet immer auf dem Server statt, so daß keine Verständigung darüber notwendig ist, was für ein Client mit welchem Server spricht. Lediglich die Transparenz für den Benutzer ist eingeschränkt. Die gleiche ACL hat auf unterschiedlichen NFS-Servern unterschiedliche Bedeutung. Es wäre zu erwägen, daß der Treiber für das EXT2-Dateisystem optional die Semantik von Solaris-ACLs annimmt, um diese Transparenz wiederherstellen zu können.

Der Linux-NFS-Dateisystemtreiber müßte also den `nfs_acl`-Dienst benutzen, um ACLs zu lesen und zu schreiben. Der Treiber darf selbst die Validität der ACL nicht überprüfen, sondern muß dies dem Server überlassen und das Ergebnis zurückmelden. Umgekehrt müßte man den Linux-NFS-Server um den `nfs_acl`-RPC-Dienst erweitern.

Leider ist der `nfs_acl`-RPC-Dienst von Solaris nicht offiziell standardisiert. D.h. die Dienstschnittstelle könnte sich ohne weiteres ändern. Allerdings wäre dies von Nachteil für Sun, da dann verschiedene Solaris Versionen inkompatibel wären.

21. Fazit

Bei unserer Implementierung stützten wir uns auf die Standardentwürfe POSIX.1e und POSIX.2c Draft 17. Diese Standardentwürfe haben selbstverständlich noch Mängel und sogar Fehler. Außerdem hat unseres Wissens kein anderes Betriebssystem ACLs konform zu Draft 17, wenn überhaupt zu irgendeinem Draft der Standardentwürfe vollständig konform implementiert. Wir hatten darauf gehofft, daß der Standard vollendet und verabschiedet werden würde. Nach derzeitigen Erkenntnissen ist dies jedoch unwahrscheinlich. Genausowenig ist zu erkennen, daß sich mehrere Hersteller von Betriebssystemen auf eine Form von ACLs einigen. Die Zusammenarbeit mit anderen Entwicklern (Alan Cox, Andreas Grünenbacher, Raymond S. Brand) und Benutzern hat uns aber den Eindruck vermittelt, daß die Implementierung dieses Standardentwurfs trotzdem eine durchaus wünschenswerte Lösung ist.

Bei unseren Tests gingen wir von “sinnvollen” Benutzeraktionen aus. Wie die Vergangenheit gezeigt hat, ergeben sich Sicherheitslücken jedoch meist aus ungewöhnlichen Kombinationen, mit denen die Programmierer des Systems nicht gerechnet haben. Dabei sind häufig auch ansonsten sinnlose Benutzeraktionen beteiligt. Derartige Sicherheitslücken zu finden, überfordert jedoch die aktuell verfügbaren formalen Methoden bei weitem: der Aufwand wäre unrealistisch groß. Hier ist es besser, sich auf die unter den Entwicklern von Betriebssystemen einzigartige Dynamik der Linux-Entwicklergemeinde zu verlassen. Findet jemand so eine Sicherheitslücke, kann man sicher sein, daß es binnen kurzer Zeit einen Patch für den Kernel gibt, der das Problem behebt. Durch die „Open Source“-Philosophie von Linux wird dies noch forciert, weil man nicht blind suchen muß: Man muß nicht die Implementation erraten, um nach möglichen Schwächen zu suchen.

Teil IV

FTS

22. Einleitung

22.1 Projektziel

Das Teilprojekt FTS (Fault Tolerant System) beschäftigt sich mit der Konzeption und dem Aufbau eines fehlertoleranten Mailservers, welcher Hardware- wie Software-Fehlertoleranz verwenden soll, der Entwicklung von dazu behilflichen Softwarekomponenten sowie dem Test und der Verifikation des Gesamtsystems mittels formaler Methoden.

Es soll dabei auch die Eignung von Linux für den Einsatz in sicherheitskritischen Bereichen geprüft werden. Die Anwendung als Mailserver soll auch auf andere Bereiche übertragbar sein. Es soll erkennbar werden, in welchen Punkten die Stärken von Linux liegen, die den Einsatz zusammen mit entsprechenden Hardwarekomponenten in „Mission–Critical"–Anwendungen befürworten. Genauso sollen die Schwächen von Linux herausgestellt werden und gleichzeitig Alternativen gefunden und Lösungsvorschläge gemacht werden. Gleiches gilt natürlich auch für das von uns erstellte Fehlertoleranzkonzept.

Die Mitglieder dieses Teilprojekts sind Hans–Jürgen Ficker, Matthias Intemann, Klaus Lüttich und Raymond Scholz.

22.2 Projektaufgabe

Der geplante Mailserver soll den bisherigen Mailserver des Fachbereichs 3, Informatik und Mathematik, an der Universität Bremen ersetzen. Es handelt sich bei dem fachbereichsweiten Mailserver um eine zentrale Einrichtung, deren Nichtverfügbarkeit eine eine weitreichende Störung der internen und externen Kommunikation zur Folge hätte. Deshalb wird ein besonderer Wert auf die hohe Verfügbarkeit des Mailservers gelegt. Ebenso ist die Speicherkapazität des alten Mailservers nicht mehr den gestiegenen Ansprüchen gewachsen.

Diese Gründe waren dafür ausschlaggebend, daß die Verantwortlichen des Fachbereichs den Auftrag zur Konzeption und zum Bau eines neuen Mailservers gaben, der den erhöhten Anforderungen entspricht.

Der Einsatz von Linux als Betriebssystem und IBM-kompatiblen PC–Komponenten stellt ein Novum im Fachbereich dar, zumindest was den Einsatz in Servern für zentrale Dienste angeht. Dabei waren die geringen Investitionskosten in die Hard- und Software ein Anreiz zur Verwendung der PC–Technologie und des freien Betriebssystems. Die so eingesparten Mittel sollten in eine erhöhte Hardware–Redundanz und Erlangung von Fehlertoleranz investiert werden.

Der bisherige Mailserver des Fachbereichs hatte einige Ausfälle zu verzeichnen, die auf den Ausfall von Hardwarekomponenten zurückzuführen waren. Andere Ausfälle resultierten aus Fehlkonfigurationen von softwareseitigen Diensten, die die Verfügbarkeit von Ressourcen des Mailservers im Fachbereich verhinderten (siehe Kapitel 24 auf Seite 163).

Die im neuen Mailserver verwendeten PC–Komponenten werden gegenüber im Workstation–Bereich eingesetzten Bauteilen als unzuverlässig eingeschätzt. Durch den Einsatz von Hardware–Redundanz kann hier eine erhöhte Sicherheit erreicht werden.

Ein positiver Nebeneffekt beim Einsatz redundanter Komponenten kann die Erhöhung der Performance sein. Diese, nicht unmittelbar einleuchtende, Tatsache ergibt sich z.B. aus der parallelen Verfügbarkeit eines Dienstes auf zwei Hardwarekomponenten. Beide Dienste können gleichzeitig angesprochen werden, was die Leistungsfähigkeit theoretisch verdoppelt.[1]

Eine wesentliche Aufgabe dieses Teilprojekts stellt die Analyse des bestehenden Mailservers dar, die in einer Konzeption und im Aufbau des neuen Systems mündet. Nach eingehenden manuellen Funktionstests folgt eine automatisierte Test- und Verifikationsphase, die zum einen die Funktionsfähigkeit des Mailservers und die Erbringung der spezifizierten Dienste bestätigen soll, andererseits aber auch das Redundanzkonzept rechtfertigen und testen soll.

22.3 Projektverlauf

In Interviews mit den technischen Verantwortlichen sowie durch eigene Untersuchungen wurden Schwachpunkte des bisherigen Konzepts ausgemacht. Darüberhinaus mußte die Konfiguration der bisher erbrachten Dienste des Mailservers übernommen werden, damit ein möglichst reibungsloser Übergang zum neuen System ermöglicht wird (siehe Kapitel 24 auf Seite 163).

Die Analyse des bestehenden Systems wurde anschließend unter Zuhilfenahme von Methoden erbracht, die Schwachpunkte in einer Konzeption erkennen lassen, die zu schwerwiegenden Problemen führen können. Aus den Ergebnissen dieser sog. Hazard Analysis (siehe Kapitel 26 auf Seite 179) sollte ein Konzept für einen neuen Mailserver gefolgert werden, der eine deutliche, belegbare Verbesserung gegenüber dem alten System darstellt.

Die Analyse legte uns u.a. eine Konzeption des neuen Mailservers als Doppelrechnersystem nahe. Jeder Teilrechner verfügt dabei über die prinzipielle Funktionalität eines Mailservers, jedoch hat nur einer davon Zugriff auf einen externen, redundant ausgelegten Festplattenspeicher und erscheint für die über mehrere Netzverbindungen redundant angebundene Umgebung als der eigentliche, aktive Mailserver. Durch diese Einschränkung war es erforderlich, ein Konzept zum Übergang der kompletten Funktionalität des aktiven Mailservers auf den anderen, passiven Teilrechner zu entwickeln. Der passive Teilrechner kann den eigentlichen Mailserver zusätzlich entlasten. Die genaue Umsetzung findet sich in Kapitel 27 auf Seite 195.

Um die Tauglichkeit des von uns konstruierten, fehlertoleranten Mailservers für den praktischen Einsatz zu prüfen und dabei die Wirksamkeit des Redundanzkonzepts unter Beweis zu stellen, haben wir das System zusätzlich getestet und verifiziert. Dazu wurde im Rahmen eines Hardware-

[1] Die dabei eventuell auftretenden Nebenläufigkeitseigenschaften bleiben in diesem Beispiel unberücksichtigt, werden von uns aber später eingehender untersucht.

In-the-Loop–Tests die spezifizierte Funktionalität des Mailservers gegen die Funktionalität des erstellten Systems getestet. Dabei kam das Produkt RT–Tester von Verified Systems International zum Einsatz (siehe Kapitel 29 auf Seite 213).

22.4 Projektergebnis

In den manuellen Testläufen wurden zunächst einige Probleme gefunden, die auch erfolgreich gelöst werden konnten (siehe Kapitel 29.2 auf Seite 214). Die anschließenden automatisierten Testläufe offenbarten dann allerdings tieferliegende Probleme, die den Einsatz des Mailservers im vorgesehenen Umfeld derzeit nicht ermöglichen.

Dieses zunächst negative Ergbnis der Testläufe ist jedoch auch positiv zu interpretieren: wäre der Mailserver ohne den ausführlichen automatisierten Testlauf in Betrieb gegangen, hätten die von uns gefundenen Probleme früher oder später zu Ausfällen geführt.

Darüberhinaus stellte sich die verwendete Hardware als ungeeignet für die eigentlich angestrebte hohe Verfügbarkeit des Systems heraus. Die Vorteile des Doppelrechner–Konzepts gegenüber der bisherigen Sparc–Plattform unter Sun Solaris können so nicht zu Tragen kommen.

Die genaue Auflistung der momentan ungelösten Probleme sowie Verbesserungsvorschläge bzgl. der verwendeten Hardware finden sich in Kapitel 30 auf Seite 217.

23. Fehlertolerante Systeme

23.1 Grundlagen der Fehlertoleranz

Wie bereits aus dem Namen des Teilprojekts FTS ersichtlich, soll der neue Mailserver als fehlertolerantes System konstruiert sein. Mit **Fehlertoleranz** (engl. fault tolerance) bezeichnet man die Eigenschaft von Systemen, beim internen Auftreten von Fehlern, die Erbringung der Dienste des Systems nach außen aufrecht zu erhalten.

Die Methoden der Fehlertoleranz verfolgen allgemein das Ziel der Vermeidung von Abhängigkeiten zwischen Fehlern in Komponenten und dem Verhalten des Gesamtsystems. Um zwei Beispiele zu nennen: Systeme, die durch den Ausfall einer zentralen Komponente lahmgelegt werden können, sind also genausowenig fehlertolerant wie Systeme, bei denen ein Fehler in einer untergeordneten Komponente weitere Fehler in übergeordneten Komponenten nach sich zieht. Eine grundlegende Einführung in die Aspekte der Fehlertoleranz erfolgt in [Sto96].

23.1.1 Umgang mit Fehlern

Fehlertoleranz ist nur ein Aspekt beim Entwurf eines sicherheitskritischen Systems. Zunächst sollte es Ziel eines Entwurfs sein, Fehler von vornherein zu vermeiden. Dies ist z.B. durch geeignete Methoden im Softwareentwicklungsprozeß zu erreichen. Bezüglich der Hardware ist ihre Auswahl und Zusammenstellung Grundlage von **Fehlervermeidung** (engl. fault avoidance). Um geeignet auf Fehler reagieren zu können, müssen diese zunächst erkannt werden. Die im Methoden der **Fehlererkennung** (engl. fault detection) werden im Abschnitt 23.2.2 behandelt. Als Reaktion auf einen erkannten Fehler ist zunächst eine **Fehlerbeseitigung** (engl. fault removal) wünschenswert, sofern diese möglich ist. Ist die sofortige Beseitigung des Fehlers nicht möglich, so wird zuletzt Fehlertoleranz angestrebt.

23.1.2 Ziele der Fehlertoleranz

All diese Methoden dienen letztendlich einer möglichst hohen **Zuverlässigkeit** (engl. dependability) des Systems. Zuverlässigkeit bedeutet in diesem Zusammenhang, daß Benutzer eines Systems oder einer Komponente voraussetzen können, daß sie ihre spezifizierte Leistung erbringt. Die so definierte Zuverlässigkeit läßt sich in feinere Aspekte unterteilen, deren Erbringung jeweils andere Maßnahmen erfordert. Die Definitionen orientieren sich dabei an [Lap92].

- **Sicherheit** (engl. safety) bezeichnet die Vermeidung von Schaden für die Umgebung eines Systems
- **Verläßlichkeit** (engl. reliability) bezeichnet die kontinuierliche Aufrechterhaltung der spezifizierten Leistung

- **Verfügbarkeit** (engl. availability) bezeichnet die Bereitschaft des Systems, die spezifizierte Leistung zu liefern
- **Schutz** (engl. security) bezeichnet die Vermeidung unberechtigten Zugriffs oder Informationserlangung

Auch wenn die naheliegende Vermutung ist, daß Fehlertoleranz nur zur Unterstützung der ersten drei Aspekte eingesetzt werden kann, ist sie dennoch zur Erfüllung aller vier Punkte eine geeignete Methode. Für den Betrieb eines Mailservers sind alle Aspekte wichtig:

So ist die interne Funktionalität des Mailservers für eine erfolgreiche Verarbeitung von Mails unerläßlich. Daraus folgt die Forderung, daß der Mailserver jederzeit in einem spezifizierten Zustand arbeitet. Von einem direkten materiellen Schaden oder der Gefährdung von Personen ist zwar nicht auszugehen, ein interner Verlust von Mails oder eine Fehlleitung stellen aber einen Schaden für die Benutzer des Mailservers dar.

Die Verläßlichkeit des Mailservers ist ebenfalls essentiell, da aufgrund der Softwarekonfiguration vieler Rechner des Fachbereichs ein Betrieb ohne Mailserver nicht sinnvoll ist (siehe dazu Kapitel 24.2). Daneben wird der Mailserver von externen Mailservern angesprochen. Die externen Mailserver besitzen zwar Mechanismen, die in Kraft treten, falls der angesprochene Mailserver nicht erreichbar ist, dennoch sollte dieser Fall wenn möglich vermieden werden.

Ebenso ist die Verfügbarkeit eine Voraussetzung, da bei einem Mailserver ein fehlerhaftes Verarbeiten, Annehmen etc. von Mails ein wichtiges Kommunikationsmittel im Fachbereich unbrauchbar machen würde.

Es muß zudem der Schutzaspekt gewährleistet sein, daß der Ausfall einer Softwarekomponente auf dem Mailserver nicht den unbeschränkten Zugang zu einer Ressource zur Folge hat. Ein Einloggen aller Benutzer auf dem Mailserver ist dabei ebenso unerwünscht, wie das Überschreiben fremder Mailboxen durch Benutzer. Ebenso würden Fehlzustellungen durch Hard- oder Softwarefehler den Datenschutz aushebeln. Einige dieser Aspekte werden in [JS99] behandelt.

23.1.3 Klassifikation von Fehlern

Für die folgenden Erläuterungen ist zunächst eine Definition notwendig. Die deutsche Sprache bietet leider keine so feine Unterscheidung der verschiedenen Ausprägungen des Begriffs „Fehler“, wie die englische. Daher haben sich (auch in der deutschen Fachliteratur) zur genaueren Unterscheidung der Arten von Fehlern die in [Lap92] definierten folgenden Begriffe durchgesetzt:

- **Failure** für die Abweichung der erbrachten Leistung von der Spezifikation eines Systems
- **Error** für den dafür verantwortlichen fehlerhaften internen Zustand eines Systems
- **Fault** für den tatsächlich auslösenden Grund der oben genannten Effekte
- **Mistake** für menschliche Fehlleistungen, die zu den oben genannten Effekten führen

Faults können sowohl die Hardware betreffen (z.B. der Ausfall einer Festplatte), als auch Software. Software hat nicht typische mechanische Eigenschaften wie Alterung, Abnutzung etc. wie Hardware. Jedoch sind logische Fehler in Programmen, fehlerhaft umgesetzte Spezifikationen, Programmierfehler (Vertipper) usw. fast zwangsläufig in jedem komplexen System enthalten. Fehler im Design eines Systems durch menschliches Versagen werden i.A. als Designfaults bezeichnet. Sie können sowohl Hard- als auch Softwarekomponenten betreffen. Ebenso sind zeitliche Aspekte beim Auftreten eines Faults unterscheidbar (permanente Faults, vorübergehende Faults), sowie die Ausbreitung von Faults im System. Diese Aspekte werden von uns nicht in dem Maße berücksichtigt, wie die oben genannten..

23.1.4 Auswirkungen eines Faults

Ausgehend von einem Fault in einer Komponente kann eine Kettenreaktion von Folgefehlern ausgelöst werden. Der Fault stimuliert zunächst einen nicht spezifizierten internen Zustand einer Komponente (Error), was zu einer Abweichung der erbrachten Leistung der Komponente führen kann (Failure). Die Abschätzung, welchen Effekt ein oder mehrere Faults für das Gesamtsystem haben, ist Thema der sog. Hazard Analysis. Sie wird im Kaptitel 26 abgehandelt.

23.2 Methoden der Fehlertoleranz

Fehlertoleranz wird vornehmlich durch den Einsatz von Redundanz erreicht. Redundanz bezeichnet dabei allgemein das Verwenden von zusätzlichen Komponenten, die für den Betrieb eines fehlerfreien Systems nicht notwendig wären.

Im folgenden sollen nun Formen der Redundanz sowie Methoden zum Erreichen von Fehlertoleranz vorgestellt werden. Dabei wird jeweils kurz auf eine mögliche Anwendung im fehlertoleranten Mailserver eingegangen, sofern dies zum momentanen Zeitpunkt schon absehbar ist.

23.2.1 Formen der Redundanz

Redundanz kann auf verschiedenen Ebenen eines komplexen Systems erreicht werden. Ebenso muß sich Redundanz nicht auf das mehrfache Verwenden von physikalischen oder virtuellen (Software) Komponenten beschränken. Redundanz existiert auch in informationeller wie zeitlicher Ausprägung.

Hardwareredundanz

Redundanz in der Hardware eines Systems bezeichnet das Hinzufügen von Hardwarekomponenten, die bei einer fehlerfreien Funktion des Systems nicht notwendig wären. Diese Redundanzform soll im fehlertoleranten Mailserver besonders ausgeprägt sein. Dies drückt sich bereits durch die Vorgabe aus, ein Doppelrechnersystem zu bauen. Dies bedeutet nichts anderes als die redundante Auslegung der gesamten Mailserver–Hardware. Ausgeschlossen davon soll lediglich das externe Festplattenarray sein, das die Mails der Benutzer vorhalten soll. Der Grund dafür ist, daß diese externe Komponente bereits intern redundant ausgelegt ist und aus mindestens zwei gleichartigen Festplatten besteht, deren Inhalt synchronisiert wird. Ebenso ist es angedacht, die physikalische Außenanbindung des Mailservers redundant auszulegen, d.h. ihm Zugang zu mehr als einem Netz zu bieten.

Softwareredundanz

Analog zu zusätzlich vorhandenen Hardwarekomponenten können auch Softwaremodule hinzugefügt werden. Es gelten dabei die gleichen Regeln wie bei Hardwareredundanz. Diese Redundanzform wird im fehlertoleranten Mailserver weniger stark verwendet werden. Die für den Mailserver zu verwendende Software ist größtenteils vorgegeben (eine Neuimplementation wäre in allen Belangen unrealistisch), ein redundantes Zusammenwirken dieser Komponenten ist nicht vorgesehen. Wo es möglich ist, soll diese Verbindung durch zusätzliche Komponenten hergestellt werden.

Informationsredundanz

Informationsredundanz liegt immer dann vor, wenn mehr Information in einem System abgelegt wird, als eigentlich nötig, um eine bestimmte Leistung zu erbringen. Ein klassisches Beispiel sind hier Paritätsinformationen in Speicherchips. Viele im Mailserver eingesetzte Hardwarebausteine verwenden nativ bereits Informationsredundanz (z.B. Übertragungsprotokolle interner Bussysteme). Als Beispiel für von uns vorgesehene Informationsredundanz ist das Spiegeln von NIS–Datenbanken als lokale Datei angedacht, so daß z.B. bei einer Korrumpierung der lokalen Datei noch auf die Datenbank im Netz zugegriffen werden kann.

zeitliche Redundanz

Zeitliche Redundanz ist dann gegeben, wenn zusätzliche Zeit vorgesehen wird, um eine Leistung zu erbringen. Diese Zeit kann z.B. dazu eingesetzt werden, Berechnungen mehrfach durchzuführen und die Ergebnisse anschließend miteinander zu vergleichen und fehlerhafte Werte auszuschließen. Dieser Form der Redundanz ist im Mailserver allerdings nicht vorgesehen.

23.2.2 Erkennen von Fehlern

Die oben genannten Formen von Redundanz allein genügen nicht, um Fehlertoleranz zu erreichen. Ein Fehler in einer redundanten Komponente muß auch erkannt werden, um in diesem Fall die Komponente abzuschalten oder einfach ihre Ergebnisse zu ignorieren. Es gibt nun verschiedene Methoden, um Fehler in den Komponenten zu erkennen. Einige für den Mailserver relevante Methoden werden im folgenden vorgestellt.

Funktionstest

Falls bei einer Komponente ihr Verhalten im fehlerfreien Zustand bekannt ist, kann durch einen Test das momentane Verhalten der Komponente mit dem spezifizierten Verhalten verglichen werden. Es bietet sich z.B. für den Mailserver an, periodisch Testmails zu senden und zu empfangen und bei einer fehlerhaften Bearbeitung z.B. den Administrator zu benachrichtigen (natürlich *nicht* per Mail).

versch. Konsistenzüberprüfungen

Hardwarekomponenten wie Festplatten müssen ihre Funktionsfähigkeit permanent überprüfen und geben z.B. über das Busprotokoll zu verstehen, daß die Umdrehungszahl unter einen zulässigen Wert gesunken ist. RAM–Bausteine können Paritätsfehler erkennen[1].

„Watchdog"–Timer

Sogenannte „Watchdogs" sind Timer, die unabhängig von einer wie auch immer gearteten Taktung einer Komponente (z.B. Prozessortakt) einen Zähler dekrementieren, der im Normalfall regelmäßig von der Komponente wieder auf den Ausgangswert erhöht wird. Läuft der Zähler ab, ist die Komponente ausgefallen. So ist eine von der Komponente unabhängige Überprüfung ihrer generellen Verfügbarkeit möglich. Der Linux–Kernel bietet z.B. die Unterstützung von PC–Einsteckkarten mit dieser Funktionalität an.

Vergleich von Ergebnissen

Das für einen Vergleich notwendige mehrfache Vorhandensein von Ergebnissen kann durch die in 23.2.1 auf Seite 155 erläuterten Mechanismen erreicht werden. Der Vergleich kann dabei durch eine Hard- oder Softwarekomponente erfolgen. Als Resultat aus diesem Vergleich soll einerseits das korrekte Ergebnis geliefert werden und andererseits bei Abweichungen der Komponenten untereinander geeignete Maßnahmen (Ausschluß der fehlerhaften Komponenten etc.) getroffen werden.

23.2.3 Hardware–Fehlertoleranz

Der oben genannte allgemein beschriebene Vergleich von Ergebnissen läßt das Vergleichsverfahren und die Reaktionen auf falsche Ergebnisse weitgehend offen. Bei der Verwendung von redundanter Hardware kommen verschiedene Methoden zum Einsatz, die für den Mailserver eventuell relevanten werden hier erläutert.

Es wird zwischen **statischer** Redundanz und **dynamischer** Redundanz unterschieden. Der generelle Unterschied besteht darin, wie auf als fehlerhaft erkannte Komponenten reagiert wird. Bei dynamischer Redundanz versucht das redundante System interne Faults zu erkennen (engl. fault detection) und das System so umzukonfigurieren, daß die fehlerhaften Komponenten nicht mehr eingebunden sind und z.B. im laufenden Betrieb ausgetauscht werden können. Statische Redundanz sieht keinen Austausch der fehlerhaften Komponente vor, sondern versucht ihre Fehlfunktion zu maskieren (engl. fault masking). Dies ist besonders bei „fest verdrahteter" Hardware notwendig. In der Kombination von Hard- und Softwaremodulen ist dynamische Redundanz die oft auch kostengünstigere Methode, da weniger redundante Komponenten benötigt werden, um einen Fehler zu erkennen. Daneben gibt es weiterhin die **hybride** Redundanz, die die beiden anderen Konzepte vereint.

[1] Einfache PC–RAM–Bausteine können die Fehler nur erkennen und nicht beheben und halten folglich bei Erkennen eines Paritätsfehlers das System an, um Fehlern in anderen Komponenten vorzubeugen.

statische Redundanz

Die allgemeinste Form der Redundanz stellt die n–**modulare Redundanz** dar. Hier erhalten n Module dieselbe Eingabe und ihre Ergebnisse werden in eine Bewertungskomponente (engl. voter) geführt. Als korrektes Ergebnis wird das Ergebnis gewählt, welches von der Mehrzahl der Module berechnet wurde. Diese Wahl wird gewöhnlich durch ein oder mehrere Hardwaremodule getroffen, die mit einfacher boolescher Logik eine Ergebnis auswählen. Konsequenz aus diesem Vorgehen ist, daß mindestens drei Module vorhanden sein müssen, um eine Mehrheitsentscheidung treffen zu können. Allgemein können maximal $(n-1)/2$ Fehler maskiert werden. Inkorrekte Ergebnisse werden maskiert, ohne daß zwangsläufig das fehlerhafte Modul von zukünftigen Berechnungen ausgeschlossen werden wird. Jedes fehlerhaftes Modul verringert aber die Redundanz des Gesamtsystems.

Als Variante dieser einfachen n–modularen Redundanz ist weitere Redundanz bei der Eingabe denkbar, so daß jedes Modul über eine eigene Eingabeleitung verfügt, deren Daten z.B. wieder von n Sensoren stammen. Ebenso kann die Zahl der Bewertungskomponenten erhöht werden, so daß jede Bewertungskomponente die Ausgaben aller redundanten Module erhält. Dieses Schema läßt sich auch weiter kaskadieren.

All diese Maßnahmen haben die Vermeidung von Failures in einer Komponente zum Ziel, die zu einem Failure des Gesamtsystems führen. Durch eine entsprechende Architektur des Gesamtsystems läßt sich also Fehlertoleranz erreichen, ohne daß das System spezielle Maßnahmen ergreifen muß, um auf fehlerhafte Module zu reagieren. Der Preis für dieses einfache Verhalten sind stark erhöhte Kosten, da mit mindestens drei gleichwertigen Modulen (auf einer hohen Abstraktionsebene) gearbeitet werden muß. Zusätzlich verursachen mehrfach ausgelegte Leitungen zu den Modulen und der/die Voter Kosten.

Für den Mailserver ist deshalb keine statische Redundanz vorgesehen. Auch bei der vorgesehenen Spiegelung der Daten im Festplattenarray handelt es sich nicht um n–Redundanz, solange dabei nur zwei Festplatten verwendet werden.

dynamische Redundanz

Dynamische Redundanz vermeidet den hohen materiellen Aufwand statischer Redundanz, indem statt einer (hardwarelogischen) Ausmaskierung von Faults ein sog. Fault Detector eingesetzt wird, der redundante Module überwacht. Dabei ist er auf fehlerhafte interne Zustände (Errors) angewiesen, die auf einen aufgetretenen Fault schließen lassen. Methoden dazu wurden in Kapitel 23.2.2 auf Seite 156 vorgestellt. Der Fault Detector veranlaßt nach Erkennen eines Errors eine Umkonfiguration des Systems (daher: *dynamische* Redundanz), die das fehlerhafte Modul ausschließt und an seine Stelle eines der redundanten Module setzt. Eine Interpretation der von einem Modul gelieferten Ergebnisse ist optional.

Die am häufigsten verwendete Methode dynamischer Redundanz ist das System der sog. **Standby Spares**. Dafür sind nur zwei Module nötig, von denen das eine aktiv ist und mit den umgebenden Modulen kommuniziert. Seine Ausgabe wird von einem Fault Detecor überwacht. Wird kein Fault erkannt, so wird die Ausgabe über einen Switch, der die Ausgabe der redundanten Module zusammenführt, ausgegeben. Im Fehlerfalle schaltet der Fault Detector das aktive Modul ab und macht das bisher passive (also im Standby befindliche) zum neuen aktiven Modul. Ebenfalls muß der Switch so umkonfiguriert werden, daß die Ausgabe des neuen aktiven Moduls nun nach außen geliefert wird.

Dieser Mechanismus ist die Grundlage für den geplanten Mailserver, der ja aus zwei eigenständigen, redundanten Teilrechnern bestehen soll.

Das Umkonfigurieren des Systems im Fehlerfalle erfordert eine kurze Unterbrechung des Betriebs des Systems. Dies ist der wesentliche Nachteil dynamischer Redundanz gegenüber statischer Redundanz. Es gibt allerdings Methoden, die diese Ausfallzeit minimieren können.

Beim sog. **Hot Standby** arbeitet das redundante passive Modul parallel zum aktiven, erhält also dieselben Eingaben und produziert (solange kein Fehler auftritt) die selben Ausgaben, die allerdings nicht ausgewertet werden. Nachteiling sind bei dieser Methode der höhere Verschleiß (so es sich um Hardwaremodule handelt) durch permanenten Betrieb des passiven Moduls und ein erhöhter Synchronisationsaufwand zwischen den Modulen.

Bei einem **Cold Standby**–System ist das passive Modul solange abgeschaltet, bis der Fault Detector es aktiviert. Danach ist eine Initialisierung nötig, die das ehemals passive Modul mit Informationen versorgt, die für ein Ersetzen des fehlerhaften Moduls mindestens notwendig sind. Dies läßt sich z.B. durch geeignete Kommunikationsprotokolle der Umgebung mit dem Modul erreichen.

Weitere Methoden der dynamischen Redundanz beruhen auf dem Vergleich der Ergebnisse redundanter Module in einer weiteren in Hard- oder Software implementierten Komponente. Diese generiert bei Abweichungen ein Fehlersignal, das zusätzlich zur Ausgabe eines der redundante Module ausgewertet wird. Kombiniert man z.B. zwei Module und eine solche Vergleichskomponente in einem Modul auf höherer Abstraktionsebene, erhält man die Möglichkeit, daß dieses Modul intern eine Abweichung erkennt und sich selbst abschaltet. Diese Methode der **Self Checking Pairs** läßt sich auch beliebig skalieren. Dabei muß allerdings sichergestellt sein, daß die Umgebung des Self Checking Pairs angemessen auf den Wegfall seiner Ausgaben reagiert. Dies ist z.B. durch mehrere Pairs gewährleistet, die wiederum eine Standby Spare–Umgebung bilden.

hybride Redundanz

Eine Kombination von Methoden der statischen und dynamischen Redundanz nennt man hybride Redundanz. Hybride Redundanz kann z.B. eine Kombination aus n–modularer Redundanz mit Standby Spares bedeuten. Der Vorteil liegt hierbei im unterbrechungsfreien Betrieb des Gesamtsystems bei einem Modulfehler. Das Modul wird durch die statische Redundanz ausmaskiert, gleichzeitig wird es durch einen Fault Detector identifiziert und ausgeschlossen. An seine Stelle tritt ein Standbymodul, so daß der Grad der Redundanz (bis auf den kurzen Augenblick des Umschaltens) nicht heruntergestuft wird. Diese Methode findet z.B. bei RAID[2] Anwendung, viele RAID–Festplattencontroller sehen hier den Einsatz von Spare–Festplatten zusätzlich zu einer Festplattenspiegelung vor.

[2]Redundant Array of Independent Disks

23.2.4 Software–Fehlertoleranz

Auch wenn bezüglich des geplanten Mailservers Hardware–Fehlertoleranz eine höhere Priorität genießt, ist eine kurze Erwähnung der Methoden der Software–Toleranz sinnvoll. Sie zeigen deutlich, daß bei Software von völlig anderen Voraussetzungen ausgegangen werden muß, als bei Hardware. n–modulare Redundanz maskiert bei Hardwarekomponenten wirksam Fehler der Module. Der Normalfall ist, daß es sich bei den verwendeten Hardwaremodulen um Komponenten gleicher Bauart handelt. Lediglich in besonders kritischen Umgebungen, die den erhöhten Kostenaufwand rechtfertigen, ist es sinnvoll parallel mehrere Ausführungen eines Hardwaremoduls fertigen zu lassen. So können z.B. bereits in der Entwicklungsphase Spezifikationsfehler oder allgemein Designfehler erkannt werden.

n–modulare Redundanz bei Software macht nur Sinn, wenn die Module nicht nur einfache Kopien sind, sondern unabhängig voneinander auf Basis der gleichen Spezifikation erstellt wurden (engl.: N–Version Programming). Bei gleichen Modulen ist nämlich davon auszugehen, daß sie sich auch im Fehlerfalle gleich verhalten[3], was ein Erkennen des Fehlers z.B. durch einen Voter unmöglich machte. Die Tatsache, daß sie auf der gleichen Spezifikation basieren, ist zwar für die Beschreibung ihres externen Verhaltens unerläßlich, macht das Konzept aber anfällig für generelle Fehler in der Spezifikation.

Die verschiedenen Versionen der Software können nun auf einem Prozessor sequentiell ausgeführt werden (höherer Zeitaufwand) oder aber parallel auf mehreren Prozessoren. So läßt sich n–modulare Redundanz von Hard- und Software kombinieren. Desweiteren existieren Methoden, die in etwa mit dynamischer Redundanz bei Hardware zu vergleichen sind.

Für den Mailserver kommt der Einsatz von Softwarefehlertoleranz im herkömmlichen Sinne nicht in Betracht. Es wäre zwar denkbar z.B. mehrere Implementationen eines Dienstes des Mailservers (z.B. POP3) nebeneinander laufen zu lassen und ihre Ergebnisse zu vergleichen, fehlende Schnittstellen, Performance-Überlegungen und der erhebliche Entwicklungsaufwand lassen diese Möglichkeit jedoch unrealistisch erscheinen.

23.2.5 Folgerungen für den geplanten Mailserver

Angesichts der begrenzten materiellen und zeitlichen Ressourcen kann der geplante Mailserver nur einen Teil der hier vorgestellten Redundanz- bzw. Fehlertoleranzformen umsetzen. Die in den obigen Abschnitten bereits aufgezählten Redundanzformen, die im neuen Mailserver Anwendung finden sollen, werden hier noch einmal kurz zusammengefaßt:

- Hardwareredundanz: Doppelrechner, doppelt ausgelegte Netzanbindung, redundant ausgelegter externer Speicher
- Informationsredundanz: Verwaltungsdaten (NIS) lokal und extern verfügbar

Der geplante Doppelrechner als Gesamtsystem kann als bedingt **dynamisch redundant** aufgefaßt werden, da ein Teilrechner nach außen den

[3] Software unterliegt keinem Alterungsprozeß, identische Module verhalten sich zu jeder Zeit gleich

Mailserver repräsentiert, der andere Teilrechner sich dabei bedingt passiv verhält, im Fehlerfalle aber manuell zum aktiven Teilrechner gemacht werden kann. Der passive Teilrechner ist also ein **Standby Spare**, der jedoch auch den aktiven Teilrechner entlasten kann. Deshalb ist das System nicht eindeutig als Hot Standby- oder Cold Standby–System zu klassifizieren.

24. Analyse der bestehenden Konfiguration

24.1 Vorgehen

Ein wichtiger Vorgehensschritt bei der Konzeption des neuen Mailservers ist die eingehende Analyse des bestehenden Systems. Dies hat mehrere Gründe:

Der Wunsch nach einem neuen Mailserver für den Fachbereich lag in der Unzuverlässigkeit des alten Servers. Die Ursachen für diverse Ausfälle des Mailservers sind sowohl bei der Hardware als auch in der Softwarekonfiguration zu suchen. Bei der Konzeption des neuen Servers sollten eventuelle Schwachstellen und „Flaschenhälse“ bzgl. der Performance vermieden werden. Die Hardware-Konfiguration, sowie die Software-Konfiguration wurde durch ein Interview mit den zuständigen Technikern des Fachbereichs in Erfahrung gebracht. Zusätzlich erhielten wir die Möglichkeit, jederzeit selbst mit erweiterten Zugangsrechten den Mailserver in seiner Konfiguration zu untersuchen.

Um den neuen Mailserver reibungslos in das bestehende Fachbereichs-Rechnernetz zu integrieren, ist es erforderlich, die bisher angebotenen Dienste des alten Mailservers möglichst unverändert zu übernehmen. Eine Neukonfiguration aller Rechner des Fachbereichs durch den Wechsel eines zentralen Servers sollte unter allen Umständen vermieden werden. Obwohl es dabei unerheblich ist, wie ein Dienst intern realisiert wird[1], haben wir dennoch Überlegungen angestellt, ob Software tatsächlich ausgetauscht werden sollte.

Die Analyse des bestehenden Systems hilft bei der Konzeptionierung eines neuen Mailservers. Schwachpunkte können durch Abänderung des Konzepts oder Aufstockung der Resourcen umgangen werden. Wichtig ist also die kombinierte Berücksichtigung der verschiedenen Probleme und Anforderungen.

Hierfür ist es notwendig eine Methodik zu finden, die es ermöglicht Mängel verschiedenster Art zu analysieren. Es sollen bisherige Probleme, bezüglich der Performance und Stabilität, soweit möglich, beseitigt werden. Dabei sollen keine neuen Dienste zur Verfügung gestellt werden. Somit müssen existente Probleme untersucht und in Verbindung gebracht werden.

Folgend soll die Konfiguration des Mailservers näher untersucht und analysiert werden. Das heißt, wir werden zunächst die Dienste, die derzeit zur Verfügung stehen aufzeigen. Dann gehen wir auf das derzeitige Mailsubsystem ein. Folgend wird das Laufzeitverhalten in Verbindung mit der verwendeten Hardware dargestellt.

[1] Black–Box-Modell

24.2 Dienste des Mailservers

Ein Mailserver stellt seinen Nutzern eine Vielzahl von sogenannten Diensten (Services) zur Verfügung. Das bedeutet, daß der Nutzer verschiedene Möglichkeiten hat, mit dem Mailserver zu kommunizieren. Voraussetzung ist, daß ein gemeinsames Format für die Mails gewählt wird. Dieses wird in der RFC 822 [Cro82] beschrieben. Als Dienste bezeichnet man also verschiedene Leistungen des oder verschiedene Kommunikationsmöglichkeiten mit dem Server.

Folgende Dienste werden derzeit zur Verfügung gestellt:

- SMTP (Simple Mail Transfer Protocol)

  Das SMTP ist die Grundlage des heutigen Internet-Mailings. Über dieses Protokoll werden derzeit die meisten Nachrichten versendet. Die Kommunikation zweier im Internet vertretener Mailserver findet auf diese Weise statt. Das Protokoll wird in der RFC 821 [Pos82] beschrieben.

- POP3 (Post Office Protocol)

  POP ist das derzeit wichtigste Protokoll zum Mailpolling, d.h. zum Beziehen von E-Mail von einem Mailserver, also dem Kopieren oder Verschieben auf das lokale System. Näheres in RFC 1939 [MR96].

- NFS (Network File System)

  UNIX-Betriebssysteme sind im Allgemeinen in der Lage, Verzeichnisse von entfernt stehenden Rechnern zu mounten, d.h. einzubinden. Das legt nahe, Mail für jeden Nutzer in eine Datei zu legen, so daß jene einfach über Dateioperationen gelesen werden kann. Dieser Dienst ist aus der heutigen UNIX-Welt nicht wegzudenken, auch wegen des Zusammenhangs mit Mailing. Zu diesem Thema kann in jedem besseren UNIX-Buch etwas gefunden werden. ([Kir95], [Moh97], [HW96])

  Näheres über das von uns verwendete NFS ist in der RFC 1094 [Now89] von Sun Microsystems nachzulesen.

- FAX–Gateway zu einem `HylaFAX`–Server

  Das `HylaFAX`-System bietet eine Schnittstelle, mittels welcher der Fax-Dienst über das Netz genutzt werden kann. Es liegt also nahe, diesen Dienst nicht nur über eigens dafür geschriebene Clients zu nutzen, sondern auch über E-Mail, da so für die Nutzer kein Aufwand der Installation zusätzlicher Software entsteht. Somit ist dieser Dienst auch zentral leichter zu warten. Weitere Information kann über `ftp://ftp.sgi.com/sgi/fax/doc` bezogen werden.

  Derzeit wird dieser Dienst im FB 3 nicht gepflegt, so daß zum Zeitpunkt des der Erstellung dieses Dokuments kein FAX-Gateway zur Verfügung steht.

Existierende, aber nicht im FB 3-Netz umgesetzte Dienste (einige Beispiele):

- IMAP4 (Internet Message Access Protocol)

  IMAP ermöglicht die Ablage von Mails in einem zentralen Mailverzeichnis. Das hat den Vorteil, daß die Sicherung von Mails zentral für alle Nutzer erledigt werden kann. Mit IMAP können auf der Mailpartition des Mailservers Folder (Schubladen, in die Mail sortiert werden kann) verwaltet werden. Ein weiterer Vorteil ist die Möglichkeit, die Maildienste software- und sogar betriebssystemunabhängig zu nutzen, ohne daß die Organisation von Mails verloren geht. (Näheres in RFC 1730 [Cri94].)

Dieser Dienst wird im FB3-Netz nicht genutzt, da IMAP das Netz mehr als die anderen vorgestellten Protokolle belastet. Desweiteren wird die gesamte Mail der Nutzer nicht in deren Home-Verzeichnissen gelagert, sondern auf den Mailpartitionen. Diese würden schnell überlaufen und könnten nur schwer mit einem Quota belegt werden, da sich dieses auf ankommende Mail auswirken könnte. IMAP wird also hauptsächlich aus Platzgründen nicht unterstützt.

- ältere Versionen des POP

  Im FB3-Netz wird keine Software gepflegt, die eine ältere Version des POP unterstützt. Daher ist es nicht von großem Nutzen, eine ältere Version seitens des Mailservers zu unterstützen.

- UUCP - Mailaustausch (Unix2UnixCopy)

  UUCP - Mailaustausch (s. RFC 976 [Hor86]) ist aus der Zeit, zu der Standleitungen eher die Ausnahme waren. Es wurden also über Modemverbindungen Mails ausgetauscht bzw. weitergereicht, bis sie ihr Ziel erreicht hatten. Das so entstandene Netz war langsam (mehrere Tage für eine Mail waren nicht selten) und die Länge der Mails war stark begrenzt. (Damals waren die Modemverbindungen noch recht langsam, d.h. 9600 Baud waren die oberste Grenze.) Desweiteren war das Netz nicht sehr zuverlässig. Über UUCP wurden / werden auch News und andere Dateien ausgetauscht.

  UUCP wird im FB 3 nicht benötigt, da es antiquiert ist und von neueren Protokollen (im Bereich des Mailaustausch) abgelöst worden ist.

- comsat (biff)

  Comsat ist ein Server zu dem Programm `biff`. Es dient dazu, dem Benutzer die ersten Zeilen einer neu eintreffenden Mail auf die Konsole zu schreiben, so dieser die Umgebungsvariable `biff=true` gesetzt hat. (weiteres auf den Manpages zu `comsat` und `biff`)

  Dieser Service ist auf dem Mailserver nicht sinnvoll einzurichten, da dieser kein „Arbeitsplatzrechner“ ist und sich nur wenige, bestimmte User dort einloggen können.

24.3 Mail-Subsystem

Zunächst stellen wir einige an der Zustellung von Mail beteiligte Programme vor und erläutern dann, wie sie zusammenspielen. Abschließend wird die Arbeitsweise der derzeitigen Konfiguration im FB3 vorgestellt.

24.3.1 Sendmail

`Sendmail` ist ein ‘Internetwork Mail Router’. Das bedeutet also, daß die Aufgabe von `sendmail` der Transfer von E-Mail durch verschieden geartete Computernetze ist. E-Mail durch ein heterogenes Netz zu leiten, ist eine sehr komplexe und umfangreiche Aufgabe. `Sendmail` ermöglicht die Kommunikation zwischen den unterschiedlichsten Netzen: das alte und neue ARPANET (RFC 733 [Cro77] bzw. RFC 822 [Cro82]), UUCP (RFC 976 [Hor86]) und durch Erweiterungen viele andere, wie Fax, Fido–Netz, Z–NETZ, etc. Hierbei bietet `sendmail` eine Vielzahl von Konfigurationsmöglichkeiten, die es zu einem der flexibelsten Mail-Transfer Programmen (**MTA – Mail Transfer Agents**) machen.

Die Autoren von `sendmail` an der Berkeley Universität in Kalifornien (USA) hatten folgende Ziele vor Augen, als sie `sendmail` entwickelten:

- Kompatibilität zu existierenden Mail–Programmen: *Bell* Version 6 und 7, *Berkeley, BerkeleyNet, UUCP* und *ARPANET*.
- Verläßlichkeit. Das bedeutete, daß jede Nachricht korrekt zugestellt wurde, oder aber zumindest der Nutzer informiert wurde, daß ein Fehler vorlag. (Gerade hier gab es Probleme mit Fehlermails, die von Mail-Site (Mailserver) zu Site geschickt wurden, und wieder zurück.)
- Bestehende Software sollte soweit wie möglich für die Zustellung als solche benutzt werden.
- Einfache Erweiterbarkeit und geringe Komplexität der Konfiguration sollten gewährleistet werden. Angesichts der verschiedenen Adressierungsarten in den Netzen war diese Zusammenstellung von Erweiterbarkeit und geringer Komplexität nur schwer zu realisieren.
- Die Konfiguration sollte nicht fest in die Programme einkompiliert werden. Ein erzeugtes Programm sollte auf mehreren Plattformen (mit der gleichen Hardwarearchitektur) funktionieren.
- `sendmail` sollte es verschiedenen Gruppen erlauben, ihre eigenen Mailinglisten und Forwardings[2] zu administrieren, ohne das System-Alias-File[3] zu ändern.
- Jedem Nutzer sollte es ermöglicht werden, seinen eigenen Mailer zur lokalen Zustellung der Mail zu verwenden. Dieses ermöglicht die Vorsortierung von Mails, automatische Antworten wie „Ich bin im Urlaub“ oder eine Vielzahl anderer Dinge (s. `procmail` im Kapitel 24.3.3 auf der nächsten Seite).
- Der Verkehr in den Netzen sollte ohne Eingriff der Nutzer auf ein Minimum beschränkt sein (dies bezieht sich vor allem auf das Mailrouting, wie es unter anderem bei UUCP organisiert war).

(Siehe auch [All].)

Derzeit sind die wichtigsten Protokolle im Zusammenhang mit `sendmail` bzw. Internet–Mailing allgemein, SMTP und POP (s. RFC 822 [Cro82] und RFC 1939 [MR96]). Diese sind auch im derzeitigen Internet die Grundlagen für elektronische Post.

Einer der obigen Punkte wurde von den Autoren von `sendmail` nicht erreicht: die Konfigurationsdatei ist sehr gewöhnungsbedürftig und komplex. Studenten der Bremer Universität nannten die „`sendmail.cf`“ kaum

[2] forward = weiterleiten

[3] Datei, in der Zuordnungen von Namen zu E-Mail Adressen beschrieben ist

von einer Binärdatei unterscheidbar. Es wird gesagt, daß man kein wahrer UNIX-Systemadministrator sei, habe man nicht einmal `sendmail` konfiguriert. Hätte man es zweimal gemacht, sei man verrückt. (Olaf Kirch, [Kir95])

24.3.2 Alternativen zu `Sendmail`

Ein weiterer MTA ist `SMail`. Seine Konfiguration ist weniger kryptisch, eher einfach und klar durchschaubar. Es wird Klartext gesprochen. `SMail` bietet bei weitem nicht so viele Möglichkeiten wie `sendmail`, ist dadurch aber auch nicht so kopflastig. Der Ursprung lag darin, eine Alternative zu `sendmail` zu schaffen und für UUCP- und SMTP-Verbindungen leicht konfigurierbar zu sein. Bei größerem Mailaufkommen ist `SMail` jedoch zu langsam.

Für unseren Zweck, die Errichtung eines fehlertoleranten Mailservers, wäre `SMail` sicherlich ausreichend, da der Mailserver des FB3 der Bremer Universität lediglich die Protokolle SMTP und POP, sowie das Bereitstellen von Mail via NFS benötigt und nutzt. Aus Gründen der Performance und da eine schnelle Erweiterung möglich und Konformität zu den vielen anderen Sites vorhanden sein soll, fällt die Wahl auf `sendmail`. Diese Entscheidung wird auch von den Technikern des FB3 getragen, da das derzeitige System ebenfalls auf `sendmail` basiert. Auch ist das geplante Konzept des FTS mit `SMail` nicht realisierbar.

Ähnliche Überlegungen haben wir zu anderen MTAs wie `qmail`, `postfix` und `exim` angestellt.

24.3.3 Mail-Delivery (`procmail`)

`Procmail` ist ein mächtiges Tool zur Verarbeitung von Mail. Genauer gesagt dient es der lokalen Zustellung von Mail, dem Filtern von Mail, dem Weiterleiten von Mail und dem Aufruf externer Programme zur Verarbeitung der Mail.

`Procmail` hat seinen Ursprung in der Mail-Zustellung, als Delivery-Agent. Das heißt, es wird von Mail-Systemen, z.B. `sendmail`, aufgerufen, um einzelne Mails an die entsprechenden Dateien anzuhängen. Dabei gilt es mittlerweile weitere Optionen zu beachten.

Über die „`~/.procmailrc`“ wird es ermöglicht, durch reguläre Ausdrücke Nachrichten nach diversen Eigenschaften zu selektieren und diese dann an andere Applikationen weiterzugeben, in Dateien zu schreiben oder nur zuzustellen.

Dem Nutzer steht die Möglichkeit `procmail` zu nutzen auch dann offen, wenn andere Delivery–Programme genutzt werden. Es ist möglich, über die „`~/.forward`“-Datei andere Programme, z.B. `procmail`, aufzurufen. Dies ist bei `sendmail` und bei `SMail` möglich.

24.3.4 Mail-Delivery (`deliver`)

`Deliver` ist das mit Solaris vertriebene Programm zur Mail-Zustellung. Zusammen mit `sendmail` wird der Mailserver der Stadard Konfiguration realisiert.

`Deliver` hat viele der im Rahmen von `procmail` erwähnten Eigenschaften, kann jedoch nicht über reguläre Ausdrücke spezielle Aktionen tätigen. Es kann lediglich eine „`~/.forward`"-Datei genutzt werden, die von `sendmail` verarbeitet wird, um ein Programm aufzurufen.

24.3.5 Zustellung von Mail

Der Weg einer Mail durch das Netz ist ein beschwerlicher: Das Mail-Tool sendet die Mail über das SMTP[4] zu dem lokalen Mailserver. Dieser akzeptiert die Mail nur von ausgewählten Clients (um Relaying, Kapitel 27.2.1 auf Seite 197, auszuschließen) oder wenn die Mail lokal zugestellt werden kann. Wird sie nicht lokal zugestellt, so wird, im Falle von Internetmailing, über den Domain Name Service (DNS) (s. RFC 1033 [Lot87], RFC 1034 [Moc87a] und RFC 1035 [Moc87b]) der Mailexchanger, der für diese Domain zuständige Mail-Server, ermittelt. An die Domain, in der die Empfängeradresse liegt, wird via SMTP die Mail gesendet. Dort wird dann vom MTA, nach dem Auslesen und Verarbeiten der „`~/.forward`", der Delivery-Agent, z.B. `procmail`, aufgerufen, der dann die Mail zustellt. Ein Mail-Tool kann nun via IMAP4, POP3, NFS oder vieler anderer Protokolle auf diese, zentral liegende Mail zugreifen.

24.4 Mailserver des FB3

Der derzeitige Mailserver „`bettina`" des FB3 benutzt, wie bereits erwähnt, `sendmail`. Da der Rechner unter dem Betriebssystem Solaris operiert, wird für die eigentliche Zustellung der Mail (Mail-Delivery) `deliver` verwendet.

24.4.1 Konfiguration

`Bettina` ist zum aktuellen Zeitpunkt als Mailserver nur ausreichend. Das bedeutet, daß versucht werden muß, viele Aufgaben auf andere Rechner zu verlagern, was andererseits neue Probleme aufwirft.

So werden alle Nutzer, derzeit etwa 1800, alle Gruppen, derzeit etwa 860, und diverse Aliase von dem Rechner „`kinkel`" verwaltet. Das heißt, daß jede dieser Eintragungen auf ein oder mehrere Mail-Spool Dateien verweist. Normalerweise müßte so für jede Mail eine Anfrage an `kinkel` gestartet werden, ob es denn die Adresse gibt, welche Mail-Spool Datei(en) angesprochen werden und eventuell eine Anfrage bezüglich des Homeverzeichnisses des Nutzers. Da dies eine Last für die beteiligten Rechner und das Netz bedeuten würde, wurde dazu übergegangen, periodisch (viertelstündlich) ein Skript aufzurufen, welches all diese Zuordnungen auf `bettina` kopiert und somit lokal verfügbar macht[5].

Es wird von `sendmail` auf die „`~/.forward`"-Datei zugegriffen. Durch Aufruf von Programmen eventuell auch auf andere Homeverzeichnisse. Ein weiteres Problem ist nun also, daß die Homeverzeichnisse auf verschiedenen Servern verteilt gespeichert liegen, was durchaus effizient ist. Allerdings muß so auch jedes einzelne Homeverzeichnis, so der Nutzer Mail erhält, gemountet werden, da die große Anzahl an Verzeichnissen nicht

[4] die hier erwähnten Protokolle wurden in Kapitel 24.2 auf Seite 164 bereits erläutert
[5] eine eigene Umsetzung dieses Programms später unter `cpyp` bei 27.1 auf Seite 196

gleichzeitig gemountet sein kann. Für diesen Vorgang gäbe es Alternativen (später), jedoch ist diese Lösung für den Nutzer die einfachste.

Um dem Nutzer des FB3 Mail via POP3 zur Verfügung zu stellen, wird `QPOP`, ebenfalls Teil von Solaris, genutzt.

Alle „trusted hosts" des Fachbereichs haben Zugriff auf die Mailpartition via NFS, wie es von Solaris vorkonfiguriert ist.

Die legendäre „`sendmail.cf`"[6] wurde, wie üblich ursprünglich über `m4`-Makros (s. 27.4.1 auf Seite 200) erstellt. Aber auch sie wuchs generisch und wurde dann manuell verändert. Somit gab es keine aktuellen Makros, die Konfigurations-Datei zu erstellen. Um sie zu ändern, mußte man die „`sendmail.cf`" verstehen. Im Rahmen des Projektes haben wir jedoch die entsprechenden Makros geschrieben, so daß eine Automatisierung wieder möglich ist.

Abschließend bleibt zu sagen, daß lediglich Rechner aus dem Fachbereichsnetz das Recht haben, `bettina` über den SMTP-Port anzusprechen, um Mail an andere Domains, für die sie selbst nicht zuständig ist, zu schicken. „Relaying" anderer Rechner in fremde Domains wird zurückgewiesen.

24.4.2 Hardware

Der derzeitige Mailserver ist eine Sparcstation 10/411 von Sun Microsystems mit 40 MHz Taktfrequenz. Als Betriebssystem wird Solaris 7 verwendet. `Bettina` ist mit 128 MB Speicher ausgestattet. Sie war mit 4 Netzinterfaces ausgestattet (und fungierte auch als Router), ist derzeit lediglich in nur einem Netz vertreten, um die Auslastung geringer zu halten. Aus dem gleichen Grund können nur ausgewählte Nutzer sich auf `bettina` anmelden.

24.5 Laufzeitverhalten

Eine Analyse des bisherigen Laufzeitverhaltens schien uns sinnvoll zu sein, da dies eine Erkennung der bisherigen Belastung und der Performance-Engpässe ermöglicht. Die daraus gezogenen Schlüsse bedingen die Dimensionierung der Komponenten und die Gesamtkonzeption des neuen Systems.

Unsere Analyse der Auslastung kommt folgendermaßen zustande: wir haben jede Minute einmal den Idle-Wert[7] der Bettina ausgelesen. Diese Werte wurden über acht Tage hinweg gesammelt und dann graphisch verarbeitet. Hierbei läßt sich erkennen, wieviel Rechenzeit `bettina` zu den verschiedenen Tageszeiten verbraucht hat.

Aus der Abbildung 24.1 auf der nächsten Seite können wir nun folgende Auslastungsspitzen erkennen:

Zwischen 2 und 3 Uhr nachts ist täglich ein kurzes Tief in der Auslastungskurve zu sehen. Dies rührt daher, daß um diese Zeit mit `rdist` alle Rechner abgeglichen werden. Das heißt, daß ein Rechner gewartet wird, mit

[6] Konfigurations-Datei zu `sendmail`

[7] ein niedriger Idle-Wert bedeutet hierbei, daß der Rechner wenig freie Rechenzeit hat

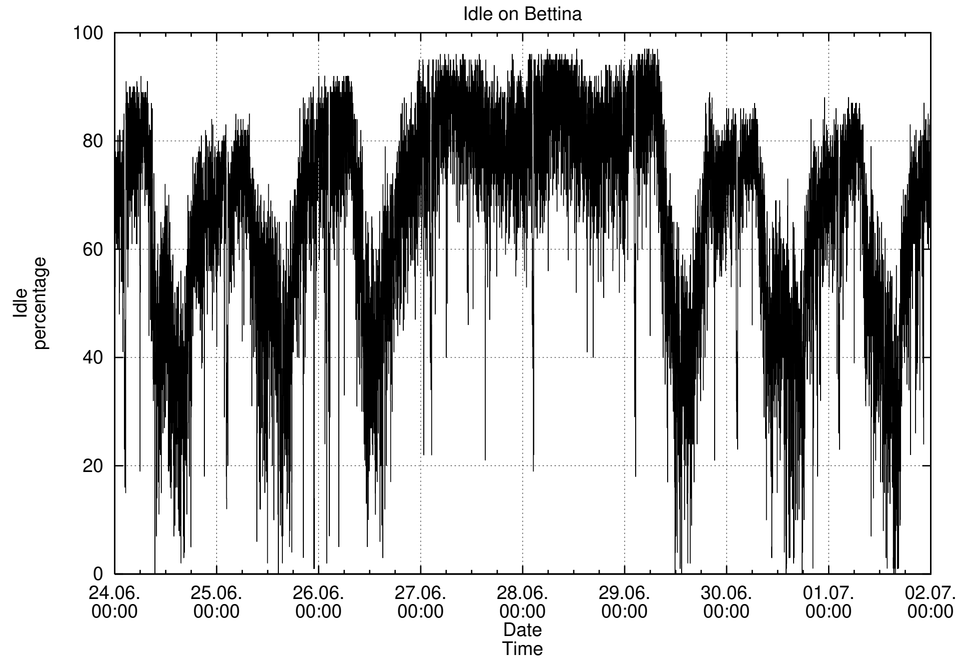


Abbildung 24.1: Idle-Graph für `bettina` vom 24.6. bis 1.7.98

z.B. neuer Software ausgestattet wird, und nachts alle anderen Rechner mit diesem abgeglichen werden. Das gilt auch für `bettina`.

Wochentags sind die meisten Studierenden im FB-Netz eingeloggt. Das bedeutet, daß auch zu dieser Zeit die meisten Mails verschickt werden, bzw. empfangen werden. Auch läßt sich ablesen, daß es viele Morgenmuffel gibt, da nachmittags die Auslastung noch höher ist. Das heißt, daß `bettina` während der Stoßzeiten fast an ihre Grenzen gerät.

An den Wochenenden findet kaum eine Auslastung statt, da hier nur wenige Studierende arbeiten.

Täglich kann man in den Abendstunden eine gesteigerte Auslastung feststellen, was damit zu zusammenhängt, daß abends die Telefonkosten geringer sind und somit viele Nutzer von daheim Mails verschicken, oder, so sie Mitglieder des FB3 sind, abholen.

Es läßt sich nicht erkennen, ob zu gewissen Zeitpunkten Mailinglisten oder Spam–Mails bevorzugt verschickt werden.

25. Hazard Analysis

25.1 Einleitung

Die im vorhergehenden Kapitel aufgestellte Konfiguration des alten Mailservers soll nun einer Analyse unterzogen werden, bei der die Schwachpunkte des Konzepts deutlich gemacht werden sollen und ein verbessertes Konzept für den neuen Mailserver erstellt werden soll.

Da der neue Mailserver die in Kapitel 23.2 auf Seite 155 behandelten Eigenschaften der Fehlertoleranz beinhalten soll, liegt bei der Analyse des alten Mailservers der Schwerpunkt auf diesem Gebiet.

25.2 Definitionen

Eine systematische Analyse der Schwachpunkte eines Systems versucht die Ursachen für mögliches Fehlverhalten zu identifizieren. Falls das von der spezifizierten Leistung abweichende Verhalten eine **potentielle** Gefahr für die Umgebung des Systems darstellt, in dem Menschen, Maschinen o.ä. bedroht sind, spricht man von einem **Hazard**.

Die Wahrscheinlichkeit des Eintretens des Hazards und seine voraussichtlichen Konsequenzen werden als **Risk** bezeichnet. Um die Risiken eines Vorgangs abschätzen zu können, müssen seine Hazards bekannt sein.

Die im Namen dieses Kapitels erwähnte **Hazard Analysis** bezeichnet nun den Vorgang der systematischen Untersuchung eines Systems auf in ihm verborgene Hazards und die Abschätzung ihrer Risiken.

25.3 Methoden

Der Begriff der Hazard Analysis selbst induziert noch nicht die Anwendung einer speziellen Vorgehensweise. Es existieren mehrere Methoden, die im folgenden kurz erläutert werden sollen. Die Anwendbarkeit auf den zu untersuchenden Mailserver soll dabei berücksichtigt werden und schließlich zur Auswahl einer speziellen Methode führen.

25.3.1 Hazard and Operability Studies (HAZOP)

Diese ursprünglich in den Sechziger Jahren in der chemischen Industrie entstandende Methode benutzt bestimmte Schlüsselwörter (engl. guide words), um Abweichungen vom spezifizierten Systemzustand auszuzeichnen. Die Beschreibung ist dabei natürlichsprachlich und so auch von Laien zu verstehen. Die Erstellung der HAZOP allerdings verlangt Expertenwissen und hohen Personalaufwand.

Ausgehend von der Spezifikation eines Systems werden die Folgen von Abweichungen vom normalen Verhalten einer Komponente untersucht. Jede Abweichung wird dahingehend untersucht, ob aus ihr ein Hazard folgen kann und wenn ja, unter welchen Bedingungen. Ebenso werden

Maßnahmen berücksichtigt, die den Hazard verhindern sollen. Die Abhängigkeiten der Komponenten untereinander sind zu ermitteln, sie stellen meist eine Schnittstelle (entity) o.ä. dar, für die Eigenschaften (attributes) seiner Daten aufgestellt werden können.

Zur Klassifikation der Abweichungen der Schnittstellendaten werden Schlüsselwörter benutzt (z.B. „more“, „less“, „before“, „late“), die aber einer Erläuterung im jeweiligen Kontext bedürfen. Alle sinnvollen Klassifikationen von Abweichungen werden nun auf die Schnittstellen angewandt und die Gründe, Folgen und einzuleitenden Maßnahmen tabellarisch aufgelistet. Ausgehend von diesen Tabellen ist eine Gewichtung der Ergebnisse vorzunehmen und aus ihnen Hazards abzuleiten.

25.3.2 Failure Modes and Effect Analysis (FMEA)

Diese Methode versucht ausgehend von einem Failure in einer Komponente seine Folgen aufzuzeigen und die letztendliche Konsequenz zu bestimmen („Bottom-Up“). Die Untersuchung beginnt dabei mit *allen* Arten von Failures (engl. failure modes), die in einer Komponente auftreten können, auch wenn sie keinen Hazard erzeugen können. Die Folgen eines best. Failures werden sowohl für die Komponente als auch bezogen auf das Gesamtsystem betrachtet.

Der umfassende Ansatz dieser Methode macht FMEA besonders bei komplexeren System aufwendig und teuer. Allerdings wird die Methode für Komponenten angewandt, die besonders hohen Anforderungen hinsichtlich der Safety genügen müssen und den Aufwand so rechtfertigen.

FMEA kann für verschiedene Abstraktionsstufen eines Systems angewandt werden. Weiterhin existiert eine Erweiterung (Failure Modes, Effects and Criticality Analysis, FMECA), die eine Klassifikation der Failures nach Schwere, Häufigkeit oder Wahrscheinlichkeit vornimmt und so eine Konzentration auf die wirklichen Schwachpunkte des Systems erlaubt.

25.3.3 Event Tree Analysis (ETA)

Ausgehend von einer Komponente versucht diese Methode das Verhalten abhängiger Komponeten bei normaler Operation und im Falle eines Failures zu bestimmen („Bottom Up“). Für alle Folgekomponenten wird nach dem gleichen Schema verfahren. Die daraus resultierenden Bäume von Ereignissen sind bei komplexeren Systemen schnell sehr groß (sie wachsen exponentiell) und richten die Aufmerksamkeit nicht sofort auf die potentiellen Hazards. Die Blätter eines Baumes zeigen das nach außen sichtbare Verhalten des Gesamtsystems.

25.3.4 Fault Tree Analysis (FTA)

Der offensichtliche Nachteil der ETA wird durch die Fault Tree Analysis behoben. Dabei werden zunächst die möglichen Hazards gesucht und identifiziert, z.B. indem Erfahrungswerte von ähnlichen Systemen in Betracht gezogen werden oder die spezifizierte Leistung des Systems negiert werden. Ausgehend von diesen Hazards werden nun die Ursachen in den Teilkomponenten des Systems gesucht, hilfreich sind dabei logische Verknüpfungen, die Ursachen und Folgen in Beziehung setzen.

Dieses Vorgehen führt zu einfacheren Bäumen als bei der ETA, da die Betonung auf den Hazards liegt. Die Methode ist graphischer Natur, im Gegensatz zur ETA resultieren die Untersuchungen also nicht in komplexen Tabellen sondern in graphisch repräsentierten Bäumen, bei denen die Hazards, Ursachen, Folgen und logischen Verknüpfungen die Knoten bilden.

25.3.5 Wahrscheinlichkeiten

Die o.g. Methoden berücksichtigen zunächst nicht, wie oft oder wie wahrscheinlich ein Failure auftritt. Gerade die große Menge an Informationen, die diese Methoden zunächst liefern, macht eine Konzentration auf die wirklich wichtigen Schwachstellen eines Systems nötig. Dazu existiert eine Erweiterung der Hazard Analysis, die **Probabilistic Hazard Analysis**.

Diese Methode wird vor allem in der FMECA eingesetzt, läßt sich aber auch erfolgreich auf andere Analysemethoden anwenden. Die Verwendung erfolgt intuitiv, indem zwischen dem Auftreten bzw. nicht Nichtauftreten eines Failures gewichtet wird. Diese Gewichtung wird dem jeweiligen Zweig zugeordnet. Geschieht dies für alle Zweige, läßt sich eine Gesamtwahrscheinlichkeit für das Versagen des Systems errechnen, indem für jeden Pfad von der Wurzel zu einem Blatt die Teilwahrscheinlichkeiten der Zweige miteinander multipliziert werden und die so errechneten Wahrscheinlichkeiten für die Blätter (die das Verhalten des Gesamtsystems repräsentieren) addiert werden.

Die so gewonnenen Aussagen für das Gesamtsystem sind jedoch nur so viel wert, wie die Güte der Aussagen über die Komponenten bzgl. ihrer Failure–Wahrscheinlichkeit.

25.3.6 Wahl der geeigneten Methode

Die Wahl der geeigneten Methode für die Hazard Analysis des alten Mailservers und der Überprüfung der noch zu entwickelnden Konzeption des neuen Mailservers fiel auf die Fault Tree Analysis. Dies hatte folgende Gründe:

- Mit dem alten Mailserver existiert ein System, für das bereits Erfahrungswerte bestehen. Die bisherigen Ausfälle und ihre Gründe sind bekannt und dokumentiert. Diese Informationen können als Ausgangspunkt für eine FTA benutzt werden.
- Ein kombiniertes Hard- und Softwaresystem wie ein Mailserver besitzt eine gewisse Komplexität, die bei der Analyse zu Tage tritt. Nicht für alle Komponenten können die internen Vorgänge geklärt werden, die zu einem Fehler führen. Der Ansatz der FTA, ausgehend von den Hazards ihre Ursachen zu finden, führt zu geringerer Komplexität und läßt dem Anwender die Wahl, die Untersuchung auf einem bestimmten Level abzubrechen.
- Die graphische Natur der FTA macht es auch Außenstehenden leicht, die *Zusammenhänge* zu verstehen, die für das Zustandekommen eines Hazards erforderlich sind.
- Die FTA ist nicht auf Hard- oder Software fixiert, sondern eignet sich auch für kombinierte Systeme, wie einen Mailserver.

Im folgenden soll die Methode der Fault Tree Analysis eingehend erläutert werden. Die vorgestellte Syntax und Semantik der FTA wird anschließend zur Analyse des bestehenden Mailservers genutzt.

25.4 Syntax und Semantik der Fault Tree Analysis (FTA)

25.4.1 Terminologie

Wie in 25.3.4 auf Seite 172 erläutert, handelt es sich bei der FTA um eine graphische Methode, die ausgehend von einem Hazard seine Ursachen zu finden versucht. Das Ereignis (engl. event), welches den Hazard direkt auslöst, steht dabei an der Spitze des Baums. Ein Event ist die Folge eines oder mehrerer anderer Events, die durch die logischen Verknüpfungen AND oder OR miteinander verknüpft werden. Dieser Prozeß wird solange durchgeführt, bis man einen oder mehrere Events gefunden hat, die die grundlegende Ursache des Hazards sind.

Die im vorherigen Abschnitt textuelle Beschreibung soll nun durch die entsprechende Terminologie und Symbolik unterstützt werden. Diese ist durch einen internationalen Standard in [Com90] festgelegt. Dieser Standard schlägt zwei Notationsvarianten vor („preferred" und „alternative"), von denen wir jedoch nur die erste verwendet haben. Durch die verwendeten Symbole wird deutlich, daß die Norm aus dem Bereich der Elektrotechnik stammt, was jedoch der Nützlichkeit der Anwendung in der Informatik keinen Abbruch tut.

25.4.2 Symbolik

Ereignisse, die fehlerhafte interne Zustände eines Systems repräsentieren (engl. fault events), werden zusätzlich klassifiziert:

- **Fault Event**, der aus anderen Events resultiert.
- **Basic Event**, der für den Fault Tree als Eingabe dient.
- **Untraced Fault Event**, der für den Fault Tree als Eingabe dient, da der Grund für sein Auftreten nicht weiter verfolgt werden kann.

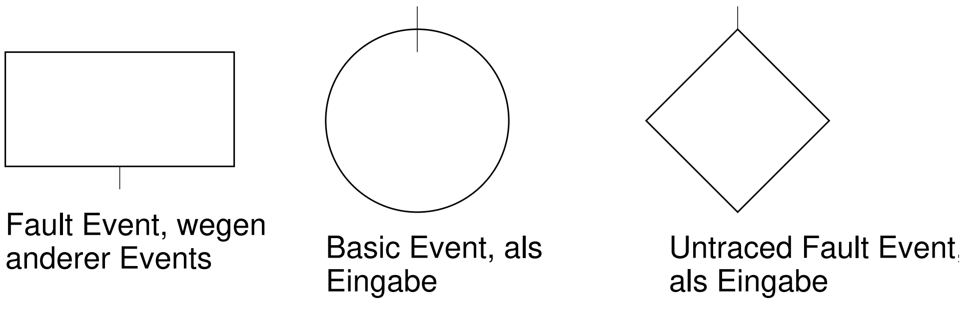


Abbildung 25.1: FTA-Symbole nach IEC 1025 (1)

Der Event, der an der Spitze eines Fault Trees steht, wird auch als **Top Event** bezeichnet. Da er den (Top) Hazard beschreibt, hat er keine weiteren Events zur Folge.

Logische Operatoren dienen dazu, Fault Events miteinander zu verknüpfen. Es gibt lediglich zwei Operatoren:

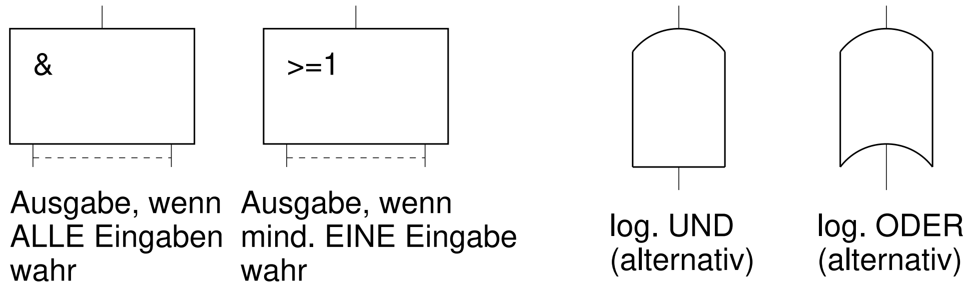


Abbildung 25.2: FTA-Symbole nach IEC 1025 (2)

- **AND**, bei dem der übergeordnete Event nur auftritt, falls alle untergeordneten Events eintreten.
- **OR**, bei dem der übergeordnete Event auftritt, wenn mindestens ein untergeordneter Event eintritt.

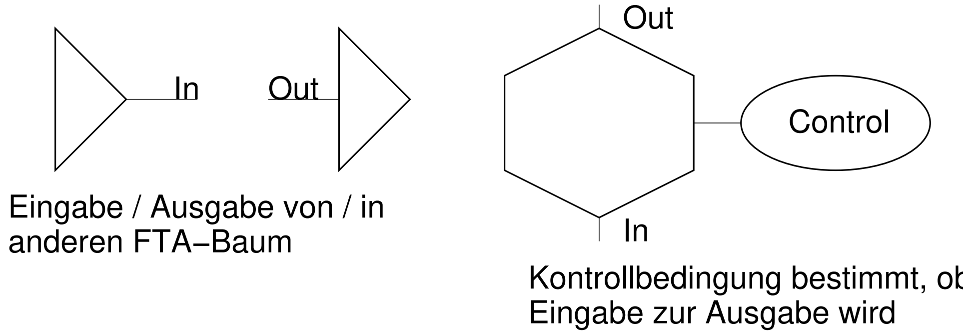


Abbildung 25.3: FTA-Symbole nach IEC 1025 (3)

Da auch Fault Trees in realen Anwendungen eine erhöhte Komplexität zeigen, ist es möglich, Bäume miteinander zu verbinden. Dazu sind Operatoren definiert, die Anschlußstücke an andere Bäume darstellen. Es gibt eine **Out**-Ausgabe, die anstelle eines Top Events stehen kann und als Eingabe für einen anderen Baum steht. Die entgegengesetze Variante ist eine **In**-Eingabe, die einen mit einer Ausgabe abgeschlossenen Fault Tree als Eingabe aufnimmt.

Daneben gibt es noch einen Operator, der eine Kontrollbedingung **Control Condition** definiert, die als Schalter zwischen einem untergeordneten Event und einem übergeordneten fungiert.

Fault Trees in der von uns verwendeten „preferred"-Notation sollten nach dem Standard von links nach rechts (der Top Event steht dabei rechts) notiert werden. Die „alternative"-Notation ist dagegen von oben nach unten aufgebaut (mit dem Top Event oben). Da wir die Fault Trees nicht manuell sondern computerunterstützt kontruiert haben und die verwendeten Programme einige Einschränkungen aufwiesen, haben wir trotz der „preferred"-Notation die Bäume von oben nach unten notiert.

25.4.3 Textuelle Repräsentation

Die oben erwähnte computerunterstützte Generierung von Fault Trees erfordert zunächst ein computerlesbares Eingabeformat, welches die in Kapitel 25.4.2 auf der vorherigen Seite aufgezeigten Symbole in geeigneter Form repräsentiert. Um eine einfache Auswertung zu ermöglichen, verwendeten wir daher eine hierarchische ASCII-Text-Notation, die von einem Scanner bzw. Parser leicht eingelesen und in andere Repräsentationen überführt werden kann.

Die textuelle Repräsentation soll im folgenden kurz aufgeführt werden, die Bedeutung der einzelnen Elemente ist dabei äquivalent zu den in IEC 1025 definierten. Die Control Condition haben wir nicht verwendet, sie ließe sich durch eine geeignete Notation aber nachrüsten.

<table>
<tr><th>IEC 1025</th><th>ASCII-Notation</th></tr>
<tr><td>Fault Event</td><td>[Event Description]</td></tr>
<tr><td>Basic Event</td><td>(Event Description)</td></tr>
<tr><td>Untraced Fault Event</td><td><Event Description></td></tr>
<tr><td>In(put)</td><td>*Event Description*</td></tr>
<tr><td>Out(put)[1]</td><td>*Event Description*</td></tr>
<tr><td>AND</td><td>&&</td></tr>
<tr><td>OR</td><td>||</td></tr>
</table>

Neben den im Standard definierten Syntax-Elementen definieren wir folgende Erweiterungen, die eine bessere Lesbarkeit der ASCII-Notation ermöglichen, sowie die Auswertung und Darstellung in anderen Programmen vereinfachen.

Erweiterung	ASCII-Notation
Titel eines FT	{Title Description}
Kommentar bis Zeilenende	#
Fortsetzung in der nächsten Zeile	\

Zuletzt muss noch festgelegt werden, auf welche Art die hierarchische Struktur eines Baums ausgedrückt wird. Gewöhnlich geschieht dies, indem die Knoten eines Baums (hier also die Events und ihre Verknüpfungen) mit Kanten verbunden werden. Da dies in einer ASCII-Notation der Übersichtlichkeit nicht zuträglich ist, haben wir uns entschieden, die Hierarchien durch Einrückungen vom linken Zeilenrand abzubilden. Die Einrückung erfolgt dabei durch das Tabulator-Zeichen, analog zu den verbreiteten Makefiles. Jeder Tabulator steht dabei für eine Stufe der Hierarchie.

25.4.4 Beispiel der textuellen Repräsentation

Für ein einfaches Beispiel soll nun erläutert werden, wie der zugehörige Fault Tree konstruiert wird. Die Konstruktion erfolgt dabei auf der Ebene der textuellen Repräsentation, diese wird anschließend in eine graphische Repräsentation umgesetzt, die der aus [Com90] nahekommt.

Das Beispiel ist ein Ausschnitt der Fault Tree Analysis für die Stromversorgung eines Computernetzteils. Der Top Hazard steht hier für den Fall, daß das Netzteil keinen Strom erhält. Dies ist der Fall, wenn die Stromzufuhr aus dem Stromnetz gestört ist und (**AND**) die unterbrechungsfreie Stromversorgung (USV) keinen Strom liefert. Die Stromzufuhr aus dem Stromnetz ist gestört, falls das Netzkabel bzw. die Steckdose defekt sind

[1] Aufgrund der eindeutigen Position innerhalb eines Fault Trees kann es trotz der gleichen ASCII-Notation nicht zu Verwechslungen mit In(put) kommen.

oder (**OR**) die Hausverteileranlage defekt ist oder (**OR**) ein externes Problem beim zuständigen Stromversorger vorliegt. Die beiden letzteren Fälle werden dabei nicht näher verfolgt. Die Fehlfunktion der USV wird in einem weiteren Fault Tree behandelt, der hier nicht aufgeführt wird.

Die textuelle Notation für dieses Beispiel sieht nun wie folgt aus:

```
# einfaches Beispiel zur Fault Tree Analysis

{FTA der Stromversorgung eines Computers}

[Netzteil erhält keinen Strom]
        &&
        [Stromversorgung gestört]
                ||
                (Netzkabel oder Steckdose defekt)
                <Hausverteileranlage defekt>
                <externes Problem beim Versorger>
        *USV liefert keinen Strom*
```

Wir glauben mit dieser Notation einen guten Kompromiß aus Maschinen- und Menschenlesbarkeit gefunden zu haben. Es ist denkbar, diese Notation manuell zu erstellen, ebenso könnte sie Ausgabe eines Programms sein, welches den Benutzer die Fault Trees graphisch unterstützt erstellen läßt. In diesem Dokument war es leider erforderlich, in einige textuelle Repräsentationen zusätzliche Zeilenumbrüche einzufügen, damit die zulässige Seitenbreite nicht überschritten wird. Diese sind durch einen Backslash am Zeilenende gekennzeichnet.

Gleichzeitig kann die Notation als Eingabe für andere Programme dienen, die Fault Trees graphisch darstellen oder auf ihnen Berechnungen ausführen können. Letztere Möglichkeit würde eine Hilfe bei der Auswertung von Fault Trees darstellen, welche die in Abschnitt 25.3.5 auf Seite 173 besprochenen Erweiterungen der Hazard Analysis nutzen.

25.4.5 Erstellung der graphischen Repräsentation

Für die graphische Darstellung der Fault Trees haben wir uns für die Graphik–Suite GRAPHVIZ der AT&T und Bell Laboratories entschieden. Die Software ist zwar frei verfügbar, fällt jedoch nicht unter die verbreitete GNU Public License (GPL) sondern unterliegt einer speziellen Lizenz. Die Software selbst, zusammen mit einigen Beispieldateien findet sich unter `http://www.research.att.com/sw/tools/graphviz/`.

GRAPHVIZ besteht aus mehreren Programmen, die zur Konstruktion, Anzeige und Ausgabe gerichteter und ungerichteter Graphen verwendet werden. Die Graphen liegen dabei in einer menschenlesbaren, textuellen Beschreibung vor, die neben den eigentlichen Graphen auch Formatierungsanweisungen für die Ausgabe der Graphen enthält (das sog. `dot`–Format).

Das Programm `dotty` dient dabei zur graphischen Anzeige und Konstruktion der Graphen, die im `dot`–Format gespeichert werden. Mit dem Kommandozeilenprogramm `dot` lassen sich die so erzeugten Dateien in Bildformate wandeln, die von anderen Programmen eingebunden werden können. Wir haben die PostScript–Ausgabe von `dot` benutzt, um die Graphen in diesen Bericht einzubinden.

Der nun noch fehlende Zwischenschritt ist die Konvertierung unserer textuellen Fault Tree–Notation in das von `dot` interpretierbare `dot`–Format. Dies erledigt das von uns erstellte Programm `ft2dot`, welches in C bzw. für den lexikalischen Analysegenerator (Scanner) `flex` geschrieben wurde. Der Quellcode dieses Programms findet sich in Anhang E auf Seite 367. Das Programm scannt die textuelle Repräsentation der Fault Trees und setzt diese in entsprechende von `dot` interpretierbare Graphendeklarationen um. Dabei erledigt es auch die Konvertierung von Umlauten (`dot` versteht leider keine selbigen) und den automatischen Umbruch der Beschriftungen der Symbole.

Der vorgesehene Ablauf ist also folgender:

1. Erstellung des Fault Trees in textueller Repräsentation
2. Konvertierung mit `ft2dot` in `dot`–Format
3. Ausgabe in PostScript–Datei mit `dot`

Alle im Kapitel 26.1 auf der nächsten Seite verwendeten Fault Trees sind also in der von uns erstellten textuellen Repräsentation gehalten, die im Anhang F auf Seite 373 nochmals in graphischer Notation wiedergegeben werden.

Das oben erläuterte Beispiel sieht in graphischer Notation wie folgt aus:

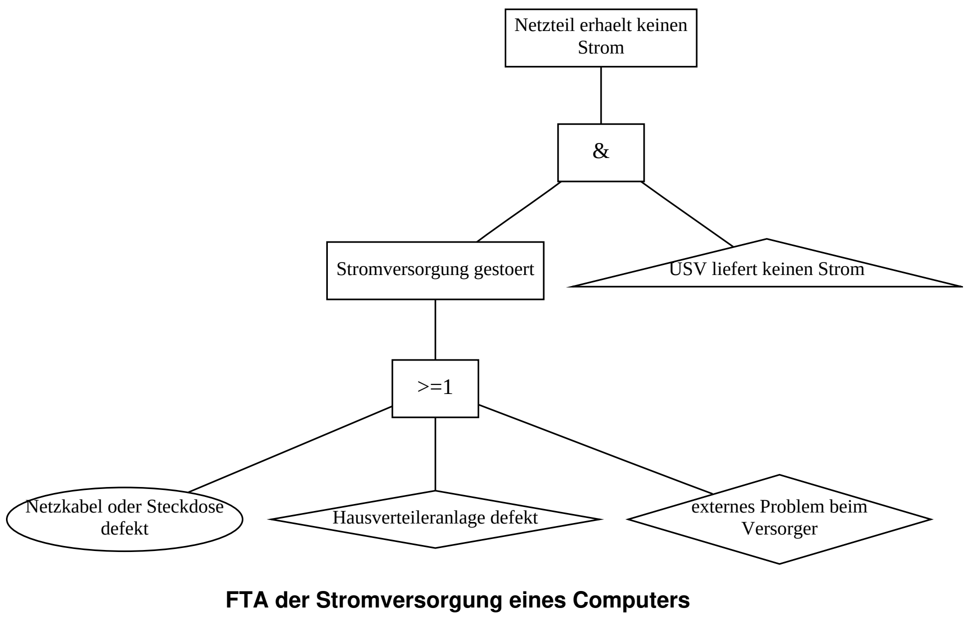


Abbildung 25.4: Beispiel in graphischer Notation als PostScript–Datei

26. Hazard Analysis der alten Konfiguration

26.1 FTA des alten Mailservers

In diesem Kapitel soll nun eine Fault Tree Analysis des alten Mailservers erfolgen, wie sie in Abschnitt 25.3.4 auf Seite 172 eingeführt wurde. Die Fault Trees selbst werden dabei u.a. aus layouttechnischen Gründen in der textuellen Repräsentation verwendet, die in Abschnitt 25.4.3 auf Seite 175 definiert wurde. Alle hier konstruierten Fault Trees sind aber auch in der graphischen Notation (ensprechend der Notation aus Abschnitt 25.4.5 auf Seite 177) im Anhang F auf Seite 373 verfügbar. Aufgrund der begrenzten Seitenbreite mußten in einigen Fault Trees Zeilen umgebrochen werden, dies ist jeweils durch einen Backslash vor dem Zeilenumbruch gekennzeichnet.

Die Komplexität eines realen Systems, wie das eines Mailservers, macht es erforderlich, das Gesamtsystem bei der Analyse in mehrere Teilsysteme aufzubrechen. Das syntaktische Werkzeug hierzu sind die Verbindungsoperatoren **In** und **Out**, die Teilbäume miteinander verbinden.

Wie in Abschnitt 25.4 auf Seite 174 erläutert, beginnt eine Fault Tree Analysis immer mit der Betrachtung des sog. Top Hazards und versucht davon ausgehend die grundlegenden Ursachen für dieses zu vermeidende Ereignis zu finden. Es handelt sich also um einen klassischen Top-Down–Ansatz, folglich beginnen wir mit unserer Analyse des alten Mailservers mit dem Top Hazard und arbeiten uns bis zu seinen möglichen Ursachen vor.

26.1.1 Mailserver (MS) erbringt nicht die spezifizierte Leistung

Ein Top Hazard eines beliebigen Mailservers (MS) könnte sein, daß dieser Mails nicht zustellt, oder das Abholen von Mails nicht zuläßt, diese falsch zustellt usw. Dies zeigt, daß die Aufgaben eines Mailservers vielfältig sind. Wir haben uns daher entschlossen, den Top Hazard allgemeiner zu fassen und stattdessen in untergeordneten Fault Trees die verschiedenen Aufgaben eines Mailservers gesondert zu behandeln.

Diese Aufgaben sind die Erbringung von typischen Diensten eines Mailservers. Im einzelnen sind dies das Bereitstellen der Mailboxen der Benutzer über NFS, über POP3, die Annahme von Mails über SMTP und die interne Verarbeitung dieser Mails. Eine genauere Betrachtung dieser Dienste findet sich in Abschnitt 24.2 auf Seite 164.

Ein Fault Event ist in diesem Fall genau die Nichterbringung eines dieser Dienste. Die Nichterbringung nur eines Dienstes hat zur Folge, daß der gesamte Mailserver nicht mehr die spezifizierte Lesitung erbringt. Daraus bildet sich folgender oberster Fault Tree:

```
[MS erbringt nicht die spezifizierte Leistung]

        ||
        *Spool-Dir nicht über NFS erreichbar*
```

```
*MS kann Mail nicht über POP3 bereitstellen*

*MS kann Mail über SMTP nicht annehmen*

*MS kann Mail nicht korrekt verarbeiten*
```

Die folgenden Abschnitte werden nun die einzelnen Dienste näher betrachten und gegebenfalls weitere Teilbäume einführen, die zur Gesamtanalyse des alten Mailservers notwendig sind.

26.1.2 Spool–Verzeichnis nicht über NFS erreichbar

Das Spool–Verzeichnis des Mailservers enthält für jeden Benutzer eine Datei, die dessen noch nicht abgeholte Mails enthält. Auf dieses Verzeichnis wird vor allem von UNIX-Rechnern (Clients) aus mittels des NFS-Dienstes zugegriffen. Der Mailserver ist somit auch ein NFS-Server. Der Zugriff erfolgt über das Netz, folglich führt eine Überlastung des Netzes dazu, daß das Spool-Verzeichnis nicht über NFS erreichbar ist.

Dieser Aspekt wird in einem weiteren Unterbaum in Abschnitt 26.1.3 auf der nächsten Seite näher untersucht.

```
*Spool-Dir nicht über NFS erreichbar*

        ||
        *MS nicht im Netz erreichbar*
```

Ebenso führt eine Überlastung des Mailservers dazu, daß der NFS-Dienst nicht ordnungsgemäß erbracht werden kann. Dies führt u.a. zu Zeitüberschreitungen bei Antworten auf NFS-Anfragen von NFS-Clients.

Diese Ursachen dieses Fault Events werden in einem weiteren Unterbaum in Abschnitt 26.1.4 auf Seite 183 näher untersucht.

```
*MS überlastet*
```

In der Terminologie von NFS nennt man das Bereitstellen eines Verzeichnisses (in diesem Fall das Spool-Verzeichnis) **Exportieren**. Eine Nichterreichbarkeit des Spool-Verzeichnisses ist somit gegeben, falls dieses vom Mailserver gar nicht erst exportiert wird.

Ein Grund dafür kann sein, daß die Festplatte(n) mit den Mails der Benutzer überhaupt nicht im System eingebunden (gemounted) ist. Grund hierfür könnte z.B. eine Fehlkonfiguration des Mailservers sein, aber auch ein möglicher Plattendefekt.

Eine triviale Ursache, die nicht näher untersucht werden soll, könnte sein, daß der NFS-Dienst überhaupt nicht auf dem Mailserver läuft. Er könnte zu starten vergessen worden sein oder sich ungewollt beendet haben.

Damit nicht jeder beliebige Client auf das exportierte Verzeichnis zugreifen kann, existierte eine Zugriffskontrolle, die in einer Datei (`/etc/exports`) festlegt, welche Rechner mit welchen Rechten auf das exportierte Verzeichnis zugreifen können. Ist es beabsichtigt, daß ein Client schreibenden Zugriff auf das Spool-Verzeichnis erhält, dies ihm unzulässigerweise aber durch die Exports-Datei verweigert wird, handelt es sich also auch um einen Fault Event. Ein Grund hierfür könnten z.B. eine Nachlässigkeit bei der Systemadministration sein.

```
        [MS exportiert Spool-Dir nicht]

                ||
                <Mail-HD nicht gemounted>

                <NFS-Dienst auf MS läuft nicht>

                <Client nicht durch /etc/exports berechtigt>
```

Die eben erläuterten Aspekte der Nichterreichbarkeit bzw. der Überlastung des Mailservers sollen nun in gesonderten Teilbäumen behandelt werden.

26.1.3 Mailserver (MS) nicht im Netz erreichbar

Falls der Mailserver nicht im Netz erreichbar ist, kann er folglich auch nicht das Spool-Verzeichnis exportieren (weitere Folgen werden später noch aufgezählt). Die Nichterreichbarkeit gründet sich auf drei Hauptursachen. Eine fehlerhafte externe Konfiguration, eine fehlerhafte interne Konfiguration und die physikalische Trennung des Mailservers vom Netz. Diese drei Aspekte sollen im folgenden näher aufgegliedert werden.

Eine fehlerhafte externe Konfiguration, also eine Software–Fehleinstellung, die nicht direkt auf dem Mailserver vorliegt, führt zu einer Nichterreichbarkeit des Mailservers. Die Ursache für diese Fehlkonfigurationen ist meist bei der Administration des Mailservers bzw. in fehlerhafter Software zu suchen.

Da der Mailserver im vorliegenden TCP/IP–Netz aus Gründen der Flexibilität nicht über seine IP–Adresse(n) sondern über seinen DNS–Namen angesprochen werden sollte, hat eine fehlerhafte Eintragung des Mailservers auf den DNS–Servern eine Nichterreichbarkeit des Mailservers über seinen Namen (`bettina`) zur Folge.

Ebenso ist es möglich, daß ein weiterer Rechner im Subnetz des Mailservers die gleiche IP–Adresse verwendet, was zu undefinierten Effekten bis zum Zusammenbruch des Subnetzes führen kann.

Um von Rechnern außerhalb des Subnetzes angesprochen zu werden, müssen an den Netzübergangpunkten Informationen über die Lokalisation des Mailservers gespeichert werden. Eine fehlerhafte Eintragung dieser sog. Routing–Informationen kann dazu führen, daß Anfragen den Mailserver nicht erreichen, weil diese in ein anderes Subnetz umgeleitet werden.

```
*MS nicht im Netz erreichbar*

        ||
        [fehlerhafte externe Konfiguration]

                ||
                <DNS-Falscheintrag>
                <IP-Adressen-Doppelbelegungen>
                <falsche Routing-Konfiguration>
```

Auch die interne Software–Konfiguration des Mailservers kann der Grund für eine Nichterreichbarkeit des Mailservers im Netz sein. Grundlage dieser Fehlkonfiguration sind in der Regel eine fehlerhafte Administration oder Software–Fehler.

Analog zu den oben erwähnten doppelten IP–Adressbelegungen kann natürlich die „Schuld“ an einer solchen fehlerhaften Zuweisung auch beim Mailserver liegen.

Daneben sind für eine Erreichbarkeit des Mailservers im Netz einige essentielle Netzdienste zu starten. Laufen diese nicht, so ist eine Erreichbarkeit des Mailservers nicht gegeben.

Auch der Mailserver selbst enthält eine Routing–Information, nämlich den Namen des nächsten Netzübergangpunkts. Ist dieser falsch gesetzt, so kann der Mailserver nicht von externen Subnetzen angesprochen werden.

```
[fehlerhafte interne Konfiguration]

        ||
        <falsche IP-Adressen / Namen>
        <Netz-Dienste laufen nicht>
        <falsch gesetzte Routen durch Routing-SW>
```

Der Mailserver ist einleuchtenderweise nicht im Netz erreichbar, wenn er „nicht online“ ist, d.h. er hat keine physikalische oder elektrische Verbindung zum Netz.

Dies ist z.B. der Fall, wenn der Mailserver nicht mit Strom versorgt. Wird. Der Grund hierfür könnte ein Defekt des lokalen Netzteils sein oder ein lokaler externer Stromausfall, z.B. ein Defekt des Stromkabels oder der Steckdose aber auch der Stromversorgung des Gebäudes.

```
[MS nicht online]

        ||
        [keine Stromversorgung]

                ||
                (Netzteil-Defekt)
                <lokaler externer Stromausfall>
```

Er ist ebenso nicht online, falls er sich in einem Zustand befindet, indem das Betriebsystem nicht läuft bzw. keine Netzverbindung aufbauen kann. Dies ist bei einem System–Stillstand oder einem Reboot–Vorgang der Fall.

Der Grund dafür kann der Wegfall der System–Festplatte aus Sicht des Betriebssystems sein, der sich bei einem Defekt der Systemplatte, einem Defekt des sie ansprechenden SCSI–Controllers sowie einem Mailserver–internen Stromausfall (Kabel defekt, Stecker nicht in Festplatte eingesteckt) ergibt.

Das Betriebsystem verweigert auch jegliche Kommunikation, falls die Systemressourcen erschöpft sind und friert das gesamte System ein. Der Grund hierfür könnte z.B. ein Software–Fehler sein, der zu einem schleichendem Ressourcenverbrauch führt.

Weiterhin ist ein Defekt für den Betrieb des Mailservers essentieller Komponenten, wie dem Mainboard für einen System–Stillstand verantwortlich.

```
        [System-Halt oder temporärer Reboot]

                ||
```

```
        [System-Platte weg]

                ||
                (System-Platten-Defekt)
                (SCSI-Controller-Defekt)
                <interner Stromausfall>

        <Systemressourcen erschöpft>
        <Mainboard-Defekt o. Defekt \
         essentieller Komponenten>
```

Der letzte Komplex behandelt den Fall, daß die Netzanbindung selbst die Ursache der Nichterreichbarkeit ist. Da der bestehende Mailserver bereits über eine redundant ausgelegte Netzverbindung verfügt, ist dies erst der Fall, wenn *beide* (allgemeiner: *alle*) Netzanbindungen ausfallen. Die Gründe dafür sind selbstverständlich die gleichen.

Das Network-Device, also der Adapter, der den Netzzugang ermöglicht, könnte defekt sein. Ebenso verhindert eine Überlastung eines Netzes eine effektive Kommunikation, da hier durch Paketverluste die Antwortzeiten extrem verlangsamt werden können. Außerdem ist eine physikalische externe Störung eines Netzes denkbar, z.B. durch einen Kabeldefekt oder einen Ausfall einer zentralen Komponente, wie einem Hub.

```
[gar keine Netzanbindung]

        &&
        [Netz A defekt]
                ||
                (1. NW-Device defekt)
                <Überlast in Netz A>
                <phys. ext. Störung in Netz A, \
                 z.B. Hub-Ausfall>

        [Netz B defekt]
                ||
                (2. NW-Device defekt)
                <Überlast in Netz B>
                <phys. ext. Störung in Netz B, \
                 z.B. Hub-Ausfall>
```

Als letzte Ursache für eine Nichterreichbarkeit des Mailservers ziehen wir einen generellen externen Stromausfall in Betracht. Damit ist ein Stromausfall gemeint, der außerhalb des Einflußbereichs der Universität liegt, also z.B. ein Problem des Stromversorgers.

```
<genereller externer Stromausfall>
```

26.1.4 Mailserver (MS) überlastet

Im Gegensatz zu einer generellen logischen, physikalischen oder elektrischen Nichterreichbarkeit, wie oben dargestellt, kann auch eine Überlastung des Mailservers bei gleichzeitig gegebener Erreichbarkeit zu Fehlern an den Clients führen. Dies äußert sich zumeist in inakzeptablen oder überschrittenen Antwortzeiten des Servers sowie in der fehlerhaften Erbringung der Dienste auf dem Mailserver.

Die Gründe hierfür sind naturgemäß vielfältig, da auf einem Mailserver viele Prozesse laufen, die das Verhalten im Betrieb negativ beeinflussen können. Wir haben uns bei dieser Untersuchung auf die Betrachtung des Dateisystems und des Prozeß- und Memorymanagements beschränkt, da hier die Ursachen für die bekannt gewordenen Ausfälle des Mailservers lagen.

Eine Überlastung des Filesystems ist gegeben, falls zuviele Dateien in einem Verzeichnis vorliegen. Diese Aussage ist prinzipiell für alle Dateisysteme gültig, die den Inhalt eines Verzeichnisses sequentiell einlesen. Dies führt dazu, daß, je mehr Dateien in einem Verzeichnis enthalten sind, die Einlesezeit linear ansteigt. Konkrete Werte für eine nicht mehr akzeptable Antwortzeit sind sicherlich Anwendungs-, Betriebssystem- und Hardware–abhängig. Da die Mailboxen aller Accounts (momentan ca. 1800) im Spool–Verzeichnis als einzelne Dateien abgelegt werden, führt dies bereits jetzt zu Geschwindigkeitseinbußen. Weiterhin benutzt Sendmail ein Queue–Verzeichnis (`/var/spool/mqueue`), in dem es temporär Dateien anlegt, die noch lokal zuzustellende Mails oder unzustellbare Mails enthalten.

Im Interview mit den technischen Verantwortlichen erfuhren wir, daß z.B. durch eine Fehlkonfiguration eines externen Sendmail–Programms dieses Queue–Verzeichnis mit hunderten von Dateien gefüllt wurde. Ein ähnliches Phänomen wurde durch eine Fehlkonfiguration innerhalb des Fachbereichs erzeugt, welches eine große Anzahl sogenannter Bogus–Dateien[1] im Spool–Verzeichnis erzeugt.

Angesichts der stetig steigenden Benutzerzahlen können auch die Mailbox–Dateien der Benutzer allein zu einem Performance–Problem führen.

Um Mails für einen Benutzer zustellen zu können, muß auf dem Mailserver das Home–Verzeichnis des Benutzers verfügbar sein. Wie auf allen anderen Rechnern im Fachbereich geschieht dies mittels eines sogenannten Automounters, der erst beim Zugriff auf das Home–Verzeichnis dieses per NFS vom zuständigen NFS–Server einbindet. Nach einer definierten Zeitspanne der Nichtbenutzung wird dieses Verzeichnis wieder vom Automounter freigegeben und ist nicht mehr auf dem Mailserver sichtbar. Erhalten nun viele Benutzer gleichzeitig Mails (was bei Spam–Attacken der Fall ist), überträgt sich das oben geschilderte Problem einer hohen Dateianzahl in einem Verzeichnis auf das Verzeichnis, welches die Home–Verzeichnisse der Benutzer enthält.

```
*MS überlastet*

        ||
        [Überlastung des Filesystems]

                ||
                [zuviele Dateien in einem Verzeichnis]

                        ||
                        <fehlerhaft konfigurierte lokale \
                         Sendmails, "Bogus">
                        <zuviele Accounts, \
```

[1] Temporäre Dateien, die bei der Verarbeitung entstehen und bei korrekter Zustellung wieder gelöscht werden.

```
                    "/var/spool/mail/$user">

          <Automounter stark beansprucht>
```

Das Prozeß- und Memory–Management des Mailservers ist überlastet, falls zuviele parallele Sendmail-Prozesse auf dem Mailserver laufen. Da jeder Prozeß eine gewisse Menge Speicher beansprucht und sich erst nach erfolgter Zustellung der Mail beendet, kann dies zu einem übermäßigen Speicherverbrauch führen.

Im Interview mit den technischen Verantwortlichen erfuhren wir von Fällen, indem externe Mailserver fehlerhaft konfiguriert waren, so daß sie keine sogenannte Postmaster–Adresse hatten. Diese Adresse ist dafür zuständig nicht zustellbare Mail anzunehmen, ihre Existenz ist zwingend notwendig. Durch das Nichtvorhandensein dieser Adresse ergab sich ein „Ping–Pong–Effekt“, bei dem Mails mit Fehlermeldungen hinundher geschickt wurden, da sich keine Seite für ihre Annahme zuständig fühlte.

Ebenso kann das oben bereits erwähnte Spamming, also das massenhafte Versenden vom Emails dazu führen, daß viele Sendmail–Prozesse parallel gestartet werden. Dies ist der Fall, wenn von einem externen Rechner (wir gehen davon aus, daß die Angehörigen des Fachbereichs selbst keine Spammer sind) sehr viele Accounts der Informatik adressiert werden.

Es ist aber ebenso denkbar, daß wenige aber dafür große Sendmail-Prozesse den Speicher des Mailservers komplett belegen, da beim Versenden bzw. Empfangen sehr großer Mails diese Mails im Speicher gehalten werden. Aus diesem Grund existiert eine Obergrenze für die Größe von Mails, die vom Mailserver verarbeitet werden. Diese liegt momentan bei 15MB.

```
[Überlastung des Prozeß-/ Memory Managements]

        ||
        [zuviele parallele Sendmails-Prozesse]

                ||
                <"Ping-Pong", z.B. durch nicht \
                 existente externe Postmaster>
                <Spamming>

        <Verschicken / Empfangen sehr großer Mails>
```

26.1.5 Mailserver (MS) kann Mail nicht über POP3 bereitstellen

Nach diesem Exkurs in die Subsysteme des Mailservers kehren wir nun wieder auf eine höhere Abstraktionsebene, die der Dienste des Mailservers, zurück.

Neben der in Abschnitt 26.1.2 auf Seite 180 beschriebenen Möglichkeit, das Spool–Verzeichnis auf UNIX–Clients zu mounten, gibt es die Möglichkeit, Mails über das POP3–Protokoll abzurufen. Dazu läuft auf dem Mailserver sein sogenannter POP–Server, der über das POP3–Protokoll mit Clients kommuniziert, deren Mail–Programme es zum Abrufen von Mails einsetzen. POP3 kann keine Mails annehmen, es ist ein reines Abrufverfahren, welches eine Authentifikation des Abrufers erfordert.

Ein Abruf von Mails über POP3 ist nicht möglich, wenn der Mailserver nicht im Netz erreichbar oder überlastet ist. Eine Diskussion dieser Aspekte findet sich in Abschnitt 26.1.3 auf Seite 181 bzw. 26.1.4 auf Seite 183.

Eine triviale Fehlerursache könnte auch sein, daß der POP3–Dienst, also der POP–Server nicht auf dem Mailserver läuft. Grund könnte ein Softwarefehler oder ein Fehler der Administration sein.

```
*MS kann Mail nicht über POP3 bereitstellen*

        ||
        *MS nicht im Netz erreichbar*

        *MS überlastet*

        <POP3-Dienst auf MS läuft nicht>
```

Eine eingehendere Betrachtung erfordert der Fall, daß über einen Client ein Benutzer seine Mails abrufen will, der Server jedoch dies nicht zuläßt. Damit ist natürlich nur der Fall gemeint, daß dieser Benutzer auch prinzipiell berechtigt ist, der Mailserver ihm diese Berechtigung aber fälschlicherweise nicht erteilt. Eine eingehende Behandlung der Security–Aspekte des Mailservers, die u.a. den unberechtigten Zugriff auf den Mailserver zum Thema hat, findet sich in [JS99].

Der erste Schritt beim Abrufen von Mails mittels POP3 ist die Angabe des Benutzernamens (Identifikation). Die Benutzernamen samt der verschlüsselten Paßwörter finden sich auf dem Mailserver in der lokalen Benutzerdatenbank (`/etc/passwd`). Diese Datei wird aus der systemweiten Benutzerdatenbank generiert, die gleichzeitig über den NIS–Dienst verteilt wird. Es existieren also zwei Möglichkeiten zur Identifikation bzw. Authentifizierung.

Findet sich der Benutzer nicht in der lokalen Benutzerdatenbank, so kann dies an einem Fehler bei der Erstellung selbiger liegen. Die zugrundeliegende NIS–Map mit den Benutzerdaten könnte unvollständig übertragen worden sein. Ebenso ist ein Softwarefehler im Skript, welches die lokale Datenbank erstellt, denkbar.

Ist die lokale Benutzerdatenbank korrekt erstellt worden, bleibt die Möglichkeit eines menschlichen Versagens durch die Administration, d.h. einem Benutzer wurden nicht die Rechte gewährt, die ihm eigentlich zuständen.

```
[Benutzer wird nicht identifiziert / authentifiziert]

        &&
        [Benutzer nicht in /etc/passwd geführt]

                ||
                [Erstellen der /etc/passwd \
                 fehlgeschlagen]

                        ||
                        <NIS-Map nicht vollst. übertragen>
                        <Fehler im Script zum \
                         Erstellen der Map>
```

```
<Benutzer nicht berechtigt>
```

Wird der Benutzer nicht in der lokalen Benutzerdatenbank gefunden, so konsultiert der Mailserver die NIS–Datenbank. Findet sich der berechtigte Benutzer auch hier nicht, so kann dies an einer fehlerhaften Konfiguration des NIS–Clients (also in diesem Fall des Mailservers) liegen. Der Mailserver könnte z.B. den falschen NIS–Server nach dem gesuchten Benutzer befragen.

Ebenso ist es möglich, daß zwar der korrekte NIS–Server angesprochen wird, dieser jedoch nicht erreichbar ist und so keine Informationen über den Benutzer liefern kann.

Zuletzt bleibt die Möglichkeit, daß die NIS–Datenbank des NIS–Servers selbst defekt ist und so eine Abfrage verhindert oder nur unvollständige Antworten zurückgibt.

```
[Benutzer nicht in NIS-Database zu finden]

           ||
           <NIS-Client falsch konfiguriert>
           <NIS-Server nicht erreichbar>
           <NIS-Database defekt>
```

Schließlich kann eine Bereitstellung der Mails über POP3 nicht erfolgen, falls auf diese überhaupt nicht zugegriffen werden kann. Dies ist der Fall, wenn die Festplatte(n) mit den Mails der Benutzer überhaupt nicht eingebunden ist. Ein Hardwaredefekt könnte genauso dafür verantwortlich sein wie eine fehlerhafte Konfiguration seitens der Administration.

```
<Mail-HD nicht gemounted>
```

26.1.6 Mailserver (MS) kann Mail über SMTP nicht annehmen

Um Mails von anderen Mailservern entgegenzunehmen, bzw. Mails von Benutzern zur Weiterleitung anzunehmen, läuft auf dem Mailserver ein SMTP–Server, in unserem Fall ist dies Sendmail. Der Mailserver kommuniziert dabei sowohl mit externen Mailservern außerhalb des Einflußbereichs des Fachbereichs sowie mit lokalen Clients innerhalb des Fachbereichs. Deshalb ist ein Ausfall dieses Dienstes als besonders kritisch zu betrachten.

Die Annahme der Mails ist trivialerweise nicht möglich, wenn der Mailserver nicht im Netz erreichbar ist oder wegen Überlastung nicht auf diesbezügliche Anfragen reagieren kann. Die Diskussion dieser Aspekte findet sich in den Abschnitten 26.1.3 auf Seite 181 bzw. 26.1.4 auf Seite 183.

Zur Zuordnung einer Emailadresse zu einem Mailserver wird ein Dienst des DNS–Servers verwendet, der sogenannte MX–Eintrag (Mail Exchanger). Dieser Eintrag ordnet dem Domainpart einer Emailadresse (rechts des „Klammeraffen") einen zuständigen Mailserver zu. Diese Eintragung erfolgt durch die lokale Administration und ist weltweit gültig.

Verweist nun dieser MX–Eintrag nicht auf den zuständigen Mailserver, können externe Mailserver Mails an Benutzer des Fachbereichs nicht zustellen. Der Grund dafür könnte in einem Fehler der lokalen Administration liegen, die eventuell einen falschen Rechnernamen als Mailserver eingetragen hat.

Ebenso ist es möglich, daß der externe Mailserver keine Anfrage bzgl. des MX–Eintrags an die zuständigen DNS–Server machen kann. Dieser Fault liegt zwar normalerweise[2] außerhalb des Einflußbereichs der lokalen Administration, sollte aber auch in dieser Untersuchung gewürdigt werden, da er die Zustellung von Mails verhindert. Im Normalfall hat ein Rechner aus Gründen der Ausfallsicherheit Zugriff auf mehrere DNS–Server, der Ausfall eines DNS–Servers führt also nicht zu einem Fault.

```
*MS kann Mail über SMTP nicht annehmen*

        ||
        *MS nicht im Netz erreichbar*

        *MS überlastet*

        [MX-Eintrag verweist nicht auf MS]

                ||
                <fehlerhafte DNS-MX-Einträge>
                [DNS-Anfrage-Fehler]

                        &&
                        <1. DNS-Server-Fehler>
                        <2. DNS-Server-Fehler>
```

Wie bei allen anderen Diensten des Mailservers ist es denkbar, daß das zuständige Programm (hier: Sendmail) nicht auf dem Mailserver läuft. In diesem Fall kann kein anderer Rechner eine SMTP–Verbindung zum Mailserver aufbauen, folglich kann dieser keine Mails annehmen. Die möglichen Ursachen für diesen Fault sind die gleichen wie für die analogen Faults der anderen Dienste.

Damit der Mailserver nicht von fachbereichsfremden Rechnern als Ausgangspunkt für Spamming–Attacken genutzt werden kann, ist er so konfiguriert, daß nur Mails von Rechnern innerhalb des Fachbereichs angenommen werden können. Sendmail interpretiert dafür geeignete Konfigurationsdateien, die eine Aussage darüber zulassen, ob einem Client–Rechner Zugriff gewährt werden soll. Ist diese Konfiguration z.B. durch einen Fehler der Administration fehlerhaft, ist es möglich, daß unberechtigterweise die Annahme von Mails eigentlich zu autorisierender Rechner abgewiesen werden können.

Nach der Annahme von Mails legt Sendmail diese zunächst in einem Queue–Verzeichnis ab (siehe Abschnitt 26.1.4 auf Seite 183), um sie dann später weiterzuleiten. Ebenso werden dort Mails abgelegt, die zwischenzeitlich unzustellbar sind. Ist das Ablegen der Mails in diesem Verzeichnis nicht möglich, können Mails auch nicht angenommen werden.

Der Grund für die Nichterreichbarkeit dieses Verzeichnisses könnte sein, daß die Festplatte, auf der das Verzeichnis abgelegt ist, nicht im System eingebunden (gemounted) ist. Der Grund könnte ein Hardware–Defekt oder ein Fehler bei der Administration des Mailservers sein.

Ein Ablegen der angenommenen Mails ist ebenfalls nicht möglich, falls das Dateisystem auf dieser Festplatte defekt ist und ein Schreiben darauf scheitert. Der Grund hierfür ist fast immer ein Hardware–Defekt der Festplatte.

[2] Es ist denkbar, daß vergessen wurde, den Mailserver im DNS einzutragen.

Das Schreiben könnte ebenso scheitern, wenn der Speicherplatz auf der Festplatte erschöpft ist. Hier könnte der Grund in den oben beschriebenen lokalen oder externen Fehlkonfigurationen liegen, der ein Zustellen von Mails verhindert. Nicht zustellbare Mails gehen dabei nicht verloren, sondern werden im Queue–Verzeichnis zwischengespeichert, welches dann überlaufen kann.

```
        <SMTP-Dienst auf MS läuft nicht>

        <SMTP-Dienst auf MS Fehl-Konfiguration>

        [Mail kann nicht gequeued werden]

                ||
                [Mail-Queue-Verzeichnis nicht erreichbar]

                        ||
                        <Mail-HD nicht gemounted>

                <Mail-Queue Filesystem defekt>
                <Mail-Queue Partition voll>
```

26.1.7 Mailserver (MS) kann Mail nicht korrekt verarbeiten

Hat der Mailserver eine Mail erfolgreich annehmen können, gibt es mehrere Möglichkeiten, was mit der Mail geschehen kann. Ist die Mail an einen lokalen Benutzer adressiert, muß der Mailserver ermitteln, wie diese Mail dem Benutzer zugestellt werden soll. Ist die Mail an einen externen Benutzer adressiert, muß sie an den zuständigen Mailserver weitergeleitet werden.

Wie im vorigen Abschnitt beschrieben, landen Mails der Annahme durch den Mailserver zunächst im Queue–Verzeichnis. Nachdem sie dort abgelegt wurden, werden sie von Sendmail weiterverarbeitet.

Im einfachsten Fall werden die Mails aus der Queue in die Mailbox (auf der Mail–Festplatte) der entsprechenden Benutzer geschrieben. Dieses Leeren des Queue–Verzeichnisses kann fehlschlagen, falls die Mail–Festplatte nicht im System eingebunden ist. Der Grund hierfür könnte ein Hardwaredefekt oder ein Administrationsfehler sein.

Ebenso ist es möglich, daß die Mails nicht auf der Mail–Festplatte abgelegt werden können, da ihr Speicherplatz erschöpft ist. Dies kann z.B. der Fall sein, wenn Benutzer ihre Mails nicht regelmäßig abholen oder den Speicherplatz auf der Mail–Festplatte für andere Zwecke mißbrauchen.

Wie für das Queue–Verzeichnis besteht auch für das Mail–Verzeichnis die Gefahr, daß das Dateisystem korrumpiert wird (z.B. durch einen Hardware–Fehler) und so die Mails nicht in den Mailboxen der Benutzer abgelegt werden können.

```
*MS kann Mail nicht korrekt verarbeiten*

        ||
        [Mail-Queue des MS kann nicht geleert werden]

                ||
                <Mail-HD nicht gemounted>
```

```
<Mail-HD / -Filesystem voll>
<Mail-Filesystem defekt>
```

Benutzer im Fachbereich haben die Möglichkeit, die Zustellung von Mails an sie zu beeinflussen. Sendmail sieht dafür die sogenannte Forward–Datei (`.forward`) vor, die im Home–Verzeichnis des Benutzers liegt. In dieser Datei kann der Benutzer z.B. festlegen, daß alle Mails an seine Adresse im Fachbereich eine andere Email–Adresse umgeleitet werden. Ebenso kann er seine Mails beim Eintreffen sortieren lassen, was über den Aufruf eines externen Programms (z.B. Procmail mit der Konfigurationsdatei `.procmailrc` im Home–Verzeichnis des Benutzers) aus der Forward–Datei geschieht. Prinzipiell lassen sich über die Forward–Datei beliebige Aktionen definieren, die beim Eintreffen einer Mail ausgeführt werden sollen.

Können nun diese Forward–Datei oder die Konfigurationsdatei eines der externen Programme nicht gelesen werden, so besteht die Gefahr, daß die Mail lokal zugestellt wird, obwohl der Benutzer eine Zustellung an eine andere Adresse gewünscht hat oder eine Sortierung der Mails erwartet. Benutzer, die diese optionalen Dateien nicht verwenden, sind davon nicht betroffen.

Da diese Konfigurationsdateien im Home–Verzeichnis des jeweiligen Benutzers liegen, können sie nicht gelesen werden, wenn das Home–Verzeichnis nicht auf dem Mailserver eingebunden ist. Dies geschieht über NFS (siehe dazu auch Abschnitt 26.1.4 auf Seite 183).

Das Einbinden des Home–Verzeichnisses schlägt fehl, wenn der Mailserver nicht im Netz erreichbar ist, bzw. überlastet ist (Abschnitte 26.1.3 auf Seite 181 bzw. nochmals 26.1.4 auf Seite 183).

```
[.forward / .procmailrc des Users können nicht \
 gelesen werden]

          ||
          [Home-Dir nicht gemounted]

                    ||
                    *MS nicht im Netz erreichbar*

                    *MS überlastet*
```

Wie auf allen UNIX–Rechnern des Fachbereichs erfolgt das Einbinden der Home–Verzeichnisse der Benutzer mittels eines Automounters. Der Automounter wird über Konfigurationsdateien (`/etc/auto.*`) gesteuert, die festlegen, von welchem Rechner und aus welchem Verzeichnis das Homeverzeichnis des jeweiligen Benutzers eingebunden werden muß. Diese Dateien werden analog zur Benutzerdatenbank (siehe dazu Abschnitt 26.1.5 auf Seite 185) über den NIS–Dienst verteilt und in bestimmten Zeitabständen lokal auf dem Mailserver abgelegt. Eine nähere Betrachtung der Security–Aspekte dieses Skripts findet sich in [JS99].

Der Automounter sucht zunächst in der lokalen Kopie der Konfigurationsdatei. Falls er dort das keinen zum Home–Verzeichnis des Benutzers zugehörigen Eintrag findet, konsultiert er die über NIS verteilte Konfiguration. Ein Fault liegt also nur vor, wenn beide Möglichkeiten versagen.

Ist das Home–Verzeichnis nicht in der lokalen Konfigurationsdatei eingetragen, könnte der Grund sein, daß beim Übertragen der Konfigurationsdatei aus dem NIS ein Fehler aufgetreten ist und die Datei nur unvollständig übetragen wurde. Ebenso ist ein Fehler im Skript denkbar, welches die NIS–Konfiguration in die lokale Datei umsetzt.

Falls sich das Home–Verzeichnis auch nicht in der Konfiguration in der NIS–Datenbank findet, ist ein Einbinden des Home–Verzeichnisses nicht mehr möglich.

Eine Ursache für diesen Fault könnte die Angabe eines falschen NIS–Servers auf dem Mailserver sein, so daß keine oder eine falsche NIS–Datenbank angesprochen wird. Der Grund hierfür dürfte in der lokalen Administration des Mailservers liegen.

Ist der korrekte NIS–Server eingetragen, besteht trotzdem die Möglichkeit, daß er nicht erreichbar ist und so die Datenbank nicht zur Verfügung stellen kann. Ebenso ist es möglich, daß die Datenbank auf dem NIS–Server defekt ist (z.B. unvollständig oder nicht lesbar). Der rund hierfür könnte ein Hardware–Defekt auf dem NIS–Server sein.

```
# Dieser Auszug wurde aus layouttechnischen Gründen
# um eine Ebene nach links verschoben.

                [automount-Fehler]

                        &&
                        [Home-Dir nicht in /etc/auto.* geführt]

                                ||
                                <NIS-Map nicht vollst. übertragen>
                                <Fehler im Script zum \
                                 Erstellen der Map>

                        [Home-Dir nicht in NIS-Database zu finden]

                                ||
                                <NIS-Client falsch konfiguriert>
                                <NIS-Server nicht erreichbar>
                                <NIS-Database defekt>
```

Um letztendlich das Home–Verzeichnis eines Benutzers einbinden zu können, muß der Mailserver den in der Konfigurationsdatei angegebenen Namen für den Rechner, der das Home–Verzeichnis des Benutzers enthält, in eine IP–Adresse auflösen. Dies geschieht über DNS, ein Fault bei der Anfrage tritt dabei analog zu den o.g. Fällen von DNS–Abfragen auf.

Schließlich ist es möglich, daß der Server, der das Home–Verzeichnis des Benutzers enthält selbst einen nicht näher spezifizierten Fehler aufweist, der das Einbinden des Verzeichnisses verhindert. Möglich sind hier Hardware–Defekte genauso wie Software–Fehlkonfigurationen.

```
                        [DNS-Anfrage-Fehler]

                                &&
                                <1. DNS-Server-Fehler>
                                <2. DNS-Server-Fehler>

                        <ext. NFS-Server-Fehler>
```

Stellt Sendmail fest, daß eine Mail nicht für die lokale Zustellung bestimmt ist, muß die Mail an einen externen Mailserver weitergeleitet werden. Diese Weiterleitung schlägt prinzipiell fehl, falls der Mailserver nicht im Netz erreichbar ist bzw. überlastet ist (siehe dazu die Abschnitte 26.1.3 auf Seite 181 bzw. 26.1.4 auf Seite 183).

Um eine Mail an eine externe Adresse zuzustellen, muß zunächst der zugehörige Mailserver bestimmt werden. Dies geschieht über das Ermitteln des MX–Eintrags (siehe dazu Abschnitt 26.1.6 auf Seite 187). Da dies den Zugriff mittels DNS erfordert, gelten auch hier die bekannten Bedingungen für einen Fault bei einer DNS–Anfrage.

```
[Weiterleitung von Mails schlägt fehl]

        ||
        *MS nicht im Netz erreichbar*

        *MS überlastet*

        [DNS-Anfrage-Fehler]

                &&
                <1. DNS-Server-Fehler>
                <2. DNS-Server-Fehler>
```

26.1.8 Fazit der Analyse

Die von uns erstellte Fault Tree Analysis macht deutlich, daß der alte Mailserver in seiner derzeitigen Konzeption nicht den Anforderungen an ein sicherheitskritisches System entspricht.

Eine Nichterreichbarkeit des Spool–Verzeichnisses wie in Abschnitt 26.1.2 auf Seite 180 erläutert, führt zu einer Blockade der Rechner im Fachbereich, die dieses Verzeichnis einbinden. Ebenso ist der Abruf und die Annahme von Mails nicht möglich (Abschnitte 26.1.5 auf Seite 185, 26.1.6 auf Seite 187 und 26.1.7 auf Seite 189).

Der Grund für eine Nichterreichbarkeit des Verzeichnisses kann in der Netzanbindung liegen (Abschnitt 26.1.3 auf Seite 181). Daher wurde auch bisher die Netzanbindung redundant ausgelegt, was eine gesteigerte Ausfallsicherheit nach sich zieht. Ebenso sind wichtige Dienste wie DNS und die Benutzerdatenbank (lokal und über NIS) redundant ausgelegt.

Ein weiterer neuralgischer Punkt ist die Stromversorgung des Mailservers. Durch eine (mehrfach) abgesicherte Stromversorgung durch eine unterbrechungsfreie Stromversorgung (USV) und externe Maßnahmen (z.B. Dieselaggregate) ließe sich hier eine weitgehende Sicherheit erreichen.

Ebenso kann eine Nichterreichbarkeit durch Überlastung des Mailservers gegeben sein (Abschnitt 26.1.4 auf Seite 183). Neben einer Aufstockung der Rechenleistung und Speicherkapazität bieten sich hier Überwachungskomponenten an, die einen übermäßigen Verbrauch von Ressourcen anzeigen und verhindern.

Ausfälle interner wie externer Hardware des Mailservers gehen meist einher mit der Nichtverfügbarkeit des gesamten Mailservers. Hierbei ist noch zu unterscheiden, ob der Ausfall oder die Zerstörung einer Komponente einen Datenverlust nach sich zieht oder die Komponente einfach durch

Austausch zu ersetzen ist. Dem ersten Fall (z.B. ein Defekt der Spool–Platte) ist durch geeignete Datensicherungsmaßnahmen vorzubeugen, welche bereits angewandt werden. Um die Ausfallzeit dennoch gering zu halten, bietet es sich an, die Spool-Platte aus dem Mailserver auszulagern und redundant auszulegen.

Als weiterer Grund für eine Nichtverfügbarkeit einzelner Dienste bzw. des Mailservers im allgemeinen ist menschliches Versagen zu nennen, welches Fehlkonfigurationen auf dem Mailserver bzw. auf den Clients zur Folge hat. Technische Maßnahmen gegen die Fehlerursache „Mensch“ können nur begrenzt wirksam sein, da der menschlichen Phantasie bei der Fehlererzeugung keine Grenzen gesetzt sind. Gewissenhafte Systemadministration vorausgesetzt, sollten sich Fehler durch menschliches Versagen jedoch in Grenzen halten.

Wie eingangs erwähnt, soll es sich bei dem neuen fehlertoleranten Mailserver um ein Doppelrechner–System handeln. Diese Konzeption bietet einen guten Schutz gegen Ausfälle der Hardware sowie Folgen menschlichen Versagens. Sofern durch eine Fehlkonfiguration nur ein Teilrechner betroffen ist[3] und ein Ausfall einer Hardwarekomponente nicht weitergehende Zerstörungen nach sich zieht, wirken sich diese Ausfälle nur auf einen Teilrechner aus. Dabei muß der zweite Teilrechner die Funktion des ersten automatisch oder durch manuellen Eingriff übernehmen.

Im folgenden Kapitel werden wir nun die hier angerissenen Punkte konkretisieren und die Architektur des fehlertoleranten Mailservers sowie seine Konfiguration erläutern.

[3]Die Administration eines Teilrechners darf sich *nicht* unmittelbar auf den anderen auswirken.

27. Entwurf eines fehlertoleranten Mailservers

27.1 Konzept

Wie schon vorher erwähnt, soll unser Mailserver–System aus zwei Rechnern bestehen. Dabei sollen beide Rechner unter verschiedenen Adressen im Netz laufen, einer als aktiver, der andere als passiver Mailserver. Dies geschieht aus mehreren Gründen:

- Beide Rechner haben jeweils einen eigenen MX–Eintrag[1], der passive dabei den mit der geringeren Priorität. Dadurch bleibt der SMTP–Dienst auch beim Ausfall oder bei einer Überlastung des aktiven Mailservers erreichbar, da nach der RFC 974 [Pat86] Mailclients versuchen, alle Rechner, die einen entsprechenden MX–Eintrag haben, in der Reihenfolge der Priorität zu erreichen, und die Mail entsprechend zustellen. Somit kann der zweite Mailserver den ersten sogar noch entlasten.
- Die Administration wird einfacher, da beide Rechner eigene IP–Adressen haben und somit auch auf dem jeweils passiven Rechner die Software– und Konfigurations–Updates leicht vorgenommen werden können.

Sowohl die Mailspool–Dateien der einzelnen Nutzer, als auch die Mailqueue werden auf einem RAID–Array gelagert. Dadurch können die Mails nicht verloren gehen, falls der aktive Mailserver einen Defekt hat[2]. Die Mailspooldateien und auch die Mailqueue können vom neuen aktiven Mailserver verarbeitet werden.

Da auch der passive Mailserver E-Mails annehmen kann, muß von Zeit zu Zeit dafür gesorgt werden, daß diese E-Mails auf den aktiven Mailserver gelangen. Dies geschieht durch den Aufruf von `sendmail -q` auf dem passiven Mailserver. Damit dies nicht gerade dann passiert, wenn der aktive Mailserver sowieso überlastet ist, sollte dies nur geschehen, wenn hier der Load–Wert[3] unterhalb eines bestimmten Wertes liegt. Dieser Wert war noch empirisch zu ermitteln, wir haben für unseren Aufbau den Wert 5 gewählt.

Auch nach dem Wechseln der Identitäten, dem sogenannten „Switching“ (zum Beispiel, weil der primäre Mailserver einen Hardware–Defekt hatte) befinden sich die E-Mails, die der alte sekundäre und jetzt primäre Mailserver noch als passiver Mailserver angenommen hatte und noch nicht an den aktiven Mailserver weitergeben konnte, auf einer eigenen Partition auf der System–Festplatte. Da dies nicht das Verzeichnis ist, in dem `sendmail` seine Mailqueue erwartet, muß dieses Verzeichnis noch als Mailqueue abgearbeitet werden.

[1] MX == **M**ail e**X**changer

[2] natürlich nur, solange das RAID nicht komplett defekt ist

[3] Der Load–Wert gibt an, wieviele Prozesse auf dem System gleichzeitig lauffähig (das heißt, nicht blockiert) sind. Unter Linux läßt er sich aus `/proc/loadavg` auslesen

Damit allerdings der passive Mailserver schnellstmöglich zum aktiven gemacht werden kann, falls der aktive ausfällt, müssen auf beiden Rechnern sowohl die Konfiguration für den passiven als auch für den aktiven Mailserver vorhanden sein. Dies betrifft insbesondere die IP-Adressen, als auch die Konfigurationsdateien für Sendmail, einige Startup–Scripts und einige Cron–Jobs. Glücklicherweise befinden sich auf einem sinnvoll konfiguriertem Linux–System alle Konfigurationsdateien im `/etc`–Verzeichnis, und so braucht man nur zwei „`/etc`–Verzeichnisse“, nämlich `/etc.primary` und `/etc.secondary`, und einen symbolischen Link `/etc`, der die gerade aktuelle Konfiguration anzeigt.

Ganz am Anfang des Boot–Prozesses muß jetzt festgelegt werden, ob der jeweilige Rechner jetzt primärer (das heißt aktiver) oder sekundärer (also passiver) Mailserver ist, um festzulegen, ob der Symbolische Link `/etc` auf `/etc.primary` oder auf `/etc.secondary` zeigen soll. Anfangs haben wir das noch über eine Eingabe über Tastatur gelöst, später sind wir allerdings dazu übergegangen, abzufragen, ob das RAID–Array am Rechner hängt, denn das kann nur am aktiven Mailserver hängen.

Damit der Mailserver nicht absolut abhängig von der stetigen Erreichbarkeit des NIS–Servers ist, ist es sinnvoll, alle Nutzerdaten, wie E-Mail–Adressen und Paßwörter, lokal auf dem Mailserver zu halten. Dies reduziert auch den Netzverkehr zwischen NIS–Server und Mailserver erheblich. Da diese Daten jedoch immer auf dem neuesten Stand gehalten werden müssen, werden sie regelmäßig auf den Mailserver kopiert. Dies geschieht über das Skript `cpyp`. Die Korrektheit dieses Skripts wurde in [JS99] untersucht.

Linux wurde als zugrundeliegendes Betriebsystem gewählt, da es neben der Kostenersparnis noch weitere Vorteile gegenüber kommerziellen UNIX–Derivaten bietet. So ist der Source–Code des Betriebssystemkerns sowie der Serverdienste frei zugänglich und in einem gewissen Maße übersichtlich und gut zu verstehen. Die Behebung von Bugs dauert aufgrund der internationalen, engagierten Entwicklergemeinde meist nur wenige Tage (oder Stunden). Gerade zentrale Server sind häufig Ziele externer sowie netzinterner Attacken, die eine Blockierung von Diensten des Servers (DoS, Denial of Service) oder ein Eindringen in den Server mit weitreichenden Folgen für Daten und Nutzer zum Ziel haben. Eine freie Verfügbarkeit des Source–Codes ermöglicht das rasche Schließen oder sogar vorzeitige Erkennen von Sicherheitslücken.

27.2 Dienste

27.2.1 Unterstützte Dienste

Simple Mail Transfer Protocol (SMTP)

Wie schon in Kapitel 24.2 auf Seite 164 erwähnt, ist das SMTP, das in der RFC 821 [Pos82] beschrieben ist, die Grundlage des heutigen Internet-Mailings. Aus diesem Grund muß es auch in dem von uns konzipierten Mailserver unterstützt werden.

Der Mailserver muß dabei Mails von externen Mailservern annehmen, die an E-Mail–Adressen innerhalb des FB3–Netzes adressiert sind. Weiterhin ist `bettina` zur Zeit als *Smarthost* für die meisten Rechner des Fachbereichs eingetragen, was bedeutet, daß alle E-Mails zunächst hierher geschickt werden, und von hier aus weiter an die jeweiligen externen Mailserver. Auch dieses *Relaying* soll weiter so bestehen bleiben, weil dadurch nur der zentrale Mailserver gewartet werden muß, und nicht durch weitere „inoffizielle Mailserver“ zusätzliche, nur schwer zu erkennende Sicherheitslücken entstehen. Dies bedeutet also, daß der neue Mailserver auch alle E-Mails aus dem FB3–Netz annehmen muß.

Heutzutage ist *Spamming* , also das massenhafte Versenden von (Werbe–)E-Mails, zu einem Problem geworden. Die Versender dieser E-Mails wollen dabei anonym bleiben und versuchen deshalb den Weg der E-Mail möglichst zu verschleiern. Dazu machen sie sich die oben beschriebene Relaying–Funktion von MTAs zunutze. Um also einen Mißbrauch des SMTP–Dienstes zu verhindern, muß er insofern eingeschränkt werden, daß eine E-Mail, die von außen kommt, aber nicht an eine Adresse im FB3–Netz adressiert ist, abgewiesen wird.

Network File System (NFS)

Der neue Mailserver soll, wie auch der jetzige, das Mailspool–Verzeichnis `/var/spool/mail` über NFS [Now89] bereitstellen. Dieser Dienst darf aber aus *Security*–Gründen nicht allen Rechnern im Fachbereichsnetz (und erst recht nicht außerhalb) zur Verfügung gestellt werden, da ein Zugriffsschutz nur durch die normalen UNIX–Rechte gegeben ist. Wer also `root`–Rechte hat, kann die E-Mails sämtlicher Nutzer lesen und verändern. Dagegen schützt auch die „`rootsquash`“–Option von NFS nichts, die Zugriffe von `root` als Nutzer `nobody` ausführt, da `root` die Identität beliebiger Nutzer annehmen kann. Daher darf das Mailspool–Verzeichnis nur an vertrauenswürdige Rechner exportiert werden, das heißt an Rechner, bei denen nur die normalen Administratoren `root`–Rechte haben.

Weiterhin wird über NFS auch das Verzeichnis `/home/alias` zur Verfügung gestellt, in dem Benutzer Mailverteiler selbst verwalten können. Auch dieses Verzeichnis darf aus oben genannten Gründen nur an vertrauenswürdige Rechner exportiert werden.

Es stand weiterhin noch zur Diskussion, ob auch die Funktionalität, die jetzt die `.forward`–Datei im Homeverzeichnis der einzelnen Benutzer zur Verfügung stellt, durch ein Verzeichnis auf dem Mailserver erreicht werden kann. Damit wäre der Mailserver weitgehend unabhängig von den Homeverzeichnis–Servern. Dieses Verzeichnis wäre dann auch über NFS an alle vertrauenswürdigen Rechner exportiert worden. Dies haben wir allerdings nicht in unseren Entwurf aufgenommen, da sich dies zu sehr vom heutigen Verfahren und auch vom üblichen UNIX–Verfahren unterscheidet. Abgesehen davon müßte das Homeverzeichnis trotzdem gemountet werden, wenn die Mail beispielsweise durch `procmail` vorsortiert würde, so daß dann dieser Vorteil wieder verloren ginge.

Post Office Protocol, Version 3 (POP3)

Da es Rechner gibt, die die Mail nicht über NFS lesen können, entweder, weil sie nicht vertrauenswürdig sind, oder aber, weil ihr Betriebssystem kein NFS unterstützt[4], muß die E-Mail noch über einen weiteren Dienst bereitgestellt werden. Hier bietet sich POP3 [MR96] an, zum einen, weil

[4]Windows, oder MacOS

das Protokoll sehr weit verbreitet ist und von fast allen Mail–Tools unterstützt wird, vor allem aber, weil es vom alten Mailserver auch bereitgestellt wird. Somit müßten von den Nutzern keine Umstellungen vorgenommen werden.

27.2.2 Nicht unterstützte Dienste

Post Office Protocol, ältere Versionen

Da es die Version 3 des Post Office Protocols schon seit 1988 gibt und die älteren Versionen auch vom alten Mailserver nicht unterstützt werden, macht es keinen großen Sinn, sie im neuen Mailserver zu verwenden.

Internet Message Access Protocol, Version 4 (IMAP4)

Wie schon im Kapitel 24.2 auf Seite 164 erwähnt, wird IMAP4 momentan im FB3–Netz nicht genutzt, da es zum einen das Netz sehr stark belastet. Zum anderen würde die gesammte Mail auf der Mailpartition gelagert, und dadurch würde der Platz darauf sehr schnell verbraucht. Eine Quota kann man auf dieser Partition nur sehr schlecht anlegen, da sich dies nur auf die ankommende Mail auswirken würde.

Aus diesen Gründen, und da IMAP4 auch (noch) nicht so weit verbreitet ist wie POP3, haben wir es auch in unseren Entwurf nicht mit aufgenommen.

HylaFAX

Da das Verschicken und Empfangen von Faxen nicht zu den eigentlichen Diensten eines Mailservers gehört und es auch in der momentanen Konfiguration nicht mehr unterstützt wird, haben wir den HylaFAX–Dienst nicht in unseren Entwurf aufgenommen.

27.3 Hardware

Der Grundgedanke bei der Auslegung der neuen Hardware war nicht allein der, unbedingt dem Projekt entsprechend Linux zu verwenden. Dennoch liegt diese Überlegung nahe, wenn man das Konzept verfolgt, nicht eine etwa 16.000 DM teure Workstation oder gar einen entsprechenden Server zu kaufen, sondern für etwa 9.000 DM ein redundant ausgelegtes System, allerdings auf IBM–PC Basis.

Die zu verwendende Hardware war zum Teil durch die unserem Projekt zur Verfügung gestellten PCs bestimmt. Ein solcher fügt sich zusammen aus einem AMD K6 Prozessor, mit 200 MHz getaktet. Der Hauptspeicher umfaßt 64MB. Eine IBM DCAS–Festplatte mit etwa 4GB Kapazität wird für das System genutzt. Als Netzinterface wird eine 3Com Boomerang eingesetzt.

Um das System für unsere Bedürfnisse zu erweitern, haben wir zunächst für jeden der zwei von uns als Mailserver ausgelegten Rechner eine zusätzliche Netzkarte, ebenfalls des Typs Boomerang, hinzugefügt. Sollte nun also ein Subnetz ausfallen oder überlastet sein, so kann der Mailserver über das zweite Netz ohne lange Ausfälle erreicht werden. Sobald der `routed`[5] die Routing–Tabelle[6] des Rechners neu aufbaut, wird das defekte Netz umgangen.

Außerdem sind beide Rechner mit einem SCSI–RAID–Controller von ICP–Vortex, mit 64MB RAM, ausgerüstet. Dieser unterstützt die RAID–Level 0, 1, 4, 5 und 10. Leider ist der Controller nicht Cluster–fähig, was bedeutet, daß nur ein Controller zur Zeit verwendet werden kann, um die Festplatten anzusprechen. Alternativ könnte man beide Rechner zugleich an die Platten anschließen, so daß ein manuelles Wechseln der externen Platten, im Falle einer Fehlfunktion im primären Mailserver, nicht notwendig wäre. Somit können beide Rechner als primärer Mailserver eingesetzt werden, was ein vollständig redundantes System darstellt. Leider ist dieses Vorgehen mit den uns zur Verfügung gestellten ICP–Vortex Controllern nicht möglich.

Als Mailplatten haben wir zwei etwa 4GB große, externe Festplatten verwendet. Die Verwendung von drei oder mehr Festplatten währe möglich, um die Gefahr des Datenverlusts geringen zu halten, waren zu Testzwecken aber nicht notwendig. Wir also zumindest sicher, den Ausfall einer Festplatte überstehen zu können.

Es sind also zwei komplett gleiche Rechner vorhanden, die so jederzeit den anderen ablösen können. Jeder Rechner für sich verfügt außerdem über Mechanismen, Ausfälle zum Teil zu überbrücken. Dies wird sowohl durch die Hardware (doppelt ausgelegte Netzinterfaces) als auch über die Konfiguration der Software (im Falle von Überlast) umgesetzt. Die externen Festplatten sind, für sich, auch doppelt ausgelegt, so daß die Fehlfunktion einer Platte auch abgefangen werden kann. Dies wird jedoch vom RAID-Controller verwaltet. Sobald eine der Festplatten ausfällt, ertönt ein *angenehmes* Piepen des Controllers.

Das größte Problem bei der Hardware liegt darin, daß seitens der externen Festplatten der Ausfall einiger Elemente einen Ausfall der gesamten Mail-Platten bedeuten würde. Dennoch könnte der Mailserver Mail annehmen, was aus unserer Sicht der wichtigste Dienst ist, aber keine Mail zur Verfügung stellen. Solche Schwachpunkte sind die Verbindungskabel zwischen Controller und Platten und zwischen den Platten selbst. Auch der Terminator kann fehlerhaft sein. Auch müssen die externen Festplattengehäuse beide einwandfrei funktionieren.

Aber dadurch, daß der Rechner doppelt ausgelegt ist, ist es möglich, solche Fehler schneller als üblich zu lokalisieren und dann auch relativ schnell zu beheben.

27.4 Sendmail

Bei der Konfiguration von `sendmail` für den neuen Mailserver haben wir uns an der bestehenden Konfiguration orientiert, da dies von den Technikern, die den alten Mailserver betreuen bzw. den neuen betreuen werden, so gewünscht wurde. Deshalb werden wir zuerst auf die Besonderheiten eingehen, die die schon bestehende Konfiguration enthält, sowie unser Umgehen mit diesen. Darauf folgen dann die Neuerungen bzgl.

[5] Programm, welches hilft, die Routing–Tabelle zu erstellen und aktuell zu halten
[6] hier wird gespeichert, über welche Wege andere Netze erreicht werden

Redundanz, die wir aufgrund der Analyse des bestehenden Mailservers entwickelt haben.

27.4.1 Umsetzen der bestehenden Konfiguration

Die `sendmail`-Konfiguration(`sendmail.cf`) des bestehenden Mailservers hat eine Besonderheit, die sich nicht mit Hilfe der üblichen Konfigurationsmethoden realisieren läßt. So werden alle Sender-Adressen so umgewandelt, daß sie die Form `login@domainname` erhalten. D.h. sowohl `@localhost` als auch `@hostname.domainname` werden in die eben genannte Form verwandelt. Dieses Umschreiben der Mailadressen geschieht mit Hilfe von Regel-Mengen, die sehr *einfache* Regeln[7] enthalten, die nach dem Prinzip funktionieren: Falls die Mailadresse der linken Seite einer Regel entspricht, wird sie mit Hilfe der rechten Seite umgeschrieben. Die Regeln einer Menge werden dabei in der Reihenfolge angewendet, in der sie definiert werden.

Um das oben genannte Umschreiben der Mailadressen bei der Mailversendung zu erreichen, gibt es bis jetzt noch keine offiziellen FEATUREs oder HACKs in den Konfigurationsdateien, die mit den `sendmail`-Quellen mitgeliefert werden. Deshalb befindet sich auf dem jetzigen Mailserver im Konfigurationsverzeichnis eine Datei, die `manuell.zu.aendern` heißt. Es ist eine Ausgabe von `diff`, aus der zu entnehmen ist, in welchen Regeln Änderungen vorzunehmen sind. Da uns diese Art der Nachbehandlung einer `sendmail`-Konfigurationsdatei mißfiel, haben wir einige neue `m4`-Dateien erzeugt, die jetzt die „automatisierte" Erstellung der Konfigurationsdateien ermöglichen.

Exkurs zu m4 `m4` ist ein Makroprozessor. Dieses Programm macht nichts weiter als seine Standardeingabe in die Standardausgabe zu kopieren und dabei bestimmte Makros zu entfalten. Es gibt viele vordefinierte Makros, aber der Benutzer kann ebenfalls Makros definieren, die beliebig viele Argumente haben können. Außerdem ist es möglich andere Dateien einzubinden, sowie andere UNIX-Programme auszuführen und vieles andere mehr.

Da die Konfigurationsdatei von `sendmail`, wie schon in Abschnitt 24.3.1 auf Seite 166 beschrieben, ein schwer lesbares Format hat, wird sie normalerweise mit Hilfe einer ganzen Menge von `m4`-Makrodateien erstellt. Daher auch die oben genannten Begriffe FEATURE und HACK. Beides sind Konfigurationsmöglichkeiten für die Makrodateien, die die Vorlage für eine `sendmail.cf`-Datei bilden. Da sich die oben beschriebene Art, Mailadressen zu verändern, auf viele Regeln in der `sendmail.cf` bezieht, war es nicht möglich, dieses neue „FEATURE" in einer Datei zu formulieren, die sich dann auf die anderen Makrodateien auswirkt. Stattdessen haben wir die `proto.m4` verändert und die geänderte Datei `proto-no-hostname.m4` genannt. In dieser Datei stehen für alle Regeln, mit denen `sendmail` die Adressen umschreibt, vorgefertigte Versionen, die durch FEATURES bzw. Hacks abgeändert werden. Uns war es zu gefährlich, die `proto.m4` direkt zu ändern, so daß wir uns dazu entschieden, eine neue Datei einzuführen. Dabei wird das Einbinden

[7] Die Regeln sind denen einer Chomsky-Grammatik ähnlich. Wenn sie danach klassifiziert werden sollten, wären sie wohl eher kontext-sensitiv, aber in Ermangelung von Nicht-Terminalen läßt sich eine Einordnung nicht korrekt vornehmen.

der `proto-no-hostname.m4` über ein neues FEATURE(„no-hostname“) erreicht.

Eine weitere Besonderheit des alten Mailservers ist, daß er nicht nur für eine Domain zuständig ist. D.h., daß jeder Nutzer des FB3-Netzes nicht nur Mails an die Domain `informatik.uni-bremen.de` erhalten kann, sondern auch an z.B. `tzi.de`. Dabei gibt es keine Überprüfung, ob der jeweilige Nutzer überhaupt etwas mit der Domain zu tun hat. Sofern er ein Account im FB3 hat, hat er auch die Möglichkeit für alle Domains unter seinem *login* Mail zu erhalten. Dasselbe ist natürlich auch beim Versenden möglich. Wenn eine Nutzerin z.B. `ginke@tzi.de` als Absenderadresse nutzt, soll diese nicht in die Standard-Domain (`informatik.uni-bremen.de`) übersetzt werden.

Um das Relaying[8] einzuschränken, wurden in der alten `sendmail.cf` die Dateien `/etc/mail/RelayTo` und `/etc/mail/LocalIP` eingebunden. Die erste Datei erlaubt das Relaying von bestimmten Domains der Universität Bremen, die zweite Datei gibt IP-Adressen[9] der Rechner an, die als lokal gelten sollen, d.h. von denen Mail immer akzeptiert wird.

27.4.2 Änderungen gegenüber der bestehenden Konfiguration

Um das Hinzufügen von neuen Domains, für die der Mailserver zuständig ist, zu erleichtern, haben wir das FEATURE „use-cw-file“ genutzt. Die neuen Domains werden dann in die Datei `/etc/mail/sendmail.cw`[10] eingetragen. Auf dem alten Mailserver wurden die neuen Domains einfach nachträglich in die `sendmail.cf` eingetragen. Das Ändern der generierten Datei „von Hand“ wollen wir jedoch möglichst verhindern.

Unter Sun Solaris ist, wie in Abschitt 24.3.4 auf Seite 167 schon erwähnt, das Programm `deliver` für die lokale Mailzustellung zuständig. Unter Linux hingegen wird häufig `procmail` verwendet. Das hat auch den Vorteil, daß eine Nutzerin, die `procmail` zum Sortieren ihrer Mail nutzt, keine `~/.forward`-Datei mehr benötigt, sondern lediglich eine `~/.procmailrc` anlegen muß.

27.4.3 Konfiguration des secondary-Mailservers

Da unser Doppelrechner-System vorsieht, daß auch der Standby-Rechner (*secondary*) Mail annehmen kann, braucht er eine spezielle `sendmail.cf`. Auf ihn zeigt, wie oben in Abschnitt 27.1 auf Seite 195 beschrieben, ein MX-Eintrag mit niedrigerer Priorität als auf den *primary*-Mailserver. Damit braucht der *secondary* also eine spezielle `sendmail`-Konfiguration, die sich sowohl von der des *primary* als auch von der der „normalen“ Arbeitsplatzrechner unterscheidet.

Da nur der *primary* direkten Zugriff auf die RAID-Platten hat und dort die Mailqueue abgelegt ist, benötigt der *secondary* eine eigene Mailqueue, die mit auf der Systemplatte liegt. Sie muß natürlich einen eigenen Namen bekommen, der sich von dem auf dem *primary* unterscheidet, damit nach einem „Switching“ der neue *primary* trotzdem noch auf die

[8] zur Definition von Relaying siehe Abschnitt 27.2.1 auf Seite 197

[9] Es müssen nicht alle IP-Adressen explizit aufgeführt werden: Mit einer unvollständigen IP-Adresse, wie z.B. „134.102.“ werden alle IP-Adressen akzeptiert, die so beginnen

[10] Mit `define(‘confCW_FILE’, ‘/etc/mail/sendmail.cw’)` muß natürlich noch festgelegt werden, daß die Datei in `/etc/mail` liegen soll.

zuvor von ihm als *secondary* „gequeueten“ Mails Zugriff hat. Dazu wurde `define('QUEUE_DIR', '/var/spool/mqueue.secondary')` in die Makrokonfigurationsdatei für die Erzeugung der `sendmail.cf` des *secondary* eingetragen.

Der *secondary* bekommt durch seinen niedriger priorisierten MX-Eintrag nur dann Mail, wenn der *primary* überlastet oder ausgefallen ist. Deshalb darf er nicht von sich aus versuchen, Mails für lokale Nutzer an der *primary* weiterzuleiten. Mit `define('confDELIVERY_MODE', 'queued')` wurde eingestellt, daß alle Mails erstmal in der Queue abgelegt werden und dort verbleiben, es sei denn, daß sie für einen externen Mailserver bestimmt sind.

Damit der *secondary* auch weiß, welcher Rechner der *primary* ist, der die Mail lokal zustellen kann, muß dies mit `define('MAIL_HUB', 'smtp:primary-mail')`[11] angegeben werden.

Die Größe der Mails ist genauso wie auf dem *primary* beschränkt. Mit `define('confMAX_MESSAGE_SIZE', '10000000')` ist sie auf etwa 10MByte[12] gesetzt.

27.5 Übernahme der alten Daten / Konfiguration

Um den Entwurf abzurunden, möchten wir zuletzt ein Konzept zur Übernahme der Daten des alten Mailservers auf den neuen vorschlagen. Beide Rechner sollten soweit konfiguriert sein, daß *nur* noch auf dem RAID die Partitionen für `/var/spool/mail/`, `/var/spool/mqueue/` und `/home/alias/` angelegt und mit den richtigen Daten gefüllt werden müssen. Ausserdem sollte es schon beide MX-Einträge für den primary-Mailserver (*primary*) und secondary-Mailserver (*secondary*) auf allen DNS-Servern geben.

Zuerst kann einer der Rechner des neuen Mailserver-Systems ohne RAID hochgefahren werden. Dieser wird damit zum *secondary* und nimmt im Falle des Ausfalls jetzt Mail an. Danach bringt man den alten Mailserver in den *Single User Mode*[13], damit gefahrlos der Rechnername in einen neuen temporären Namen umbenannt werden kann. Außerdem wird jetzt das Exportieren der Verzeichnisse `/var/spool/mail/` und `/home/alias/` via NFS abgeschaltet, sowie das Öffnen des SMTP-Ports verhindert. Ebenso wird der POP3-Daemon stillgelegt. Danach wird der alte Mailserver mit seinem neuen Namen neu gebootet.

Der zweite Rechner des Mailserver-Systems wird jetzt mit RAID ebenfalls im *Single User Mode* gebootet. Dann wird das RAID konfiguriert und partioniert (s.o.). Als Rechnername wird erstmal ein weiterer temporärer Name genommen, damit gar nicht erst versucht werden kann, diesen Rechner während der Übernahme der Daten als Mailserver anzusprechen. Um jetzt aber nicht ein leeres `/var/spool/mail/` und ein leeres `/home/alias/` zu exportieren, muß NFS erstmal deaktiviert werden. Ebenso darf dieser Rechner noch keine Mail annehmen bzw. bereitstellen, also darf der SMTP- und der POP3-Port nach dem nächsten Booten nicht geöffnet werden. Jetzt wird dieser Rechner unter seinem temporären Namen gebootet.

[11] Natürlich muß „primary-mail“ durch den Hostname des *primary* ersetzt werden.

[12] 10MB sind 10486760Byte

[13] Damit wird der Netzzugang abgeschaltet und er ist nicht mehr von anderen Rechnern erreichbar, d.h. also auch nicht mehr als Mailserver

Da jetzt sowohl der alte Mailserver als auch der Rechner des Mailserver-Systems mit dem RAID unter temporären Namen einen Netzzugang haben, können die Dateien in `/var/spool/mail/` und die Verzeichnisse samt Dateien in `/home/alias/` vom alten Mailserver auf das RAID des neuen Mailserver-Systems via `scp`[14] übertragen werden. Mit demselben Mechanismus werden noch die Dateien `/etc/shadow` und `/etc/passwd` auf den neuen Mailserver übertragen.

Jetzt kann der alte Mailserver heruntergefahren werden. Der Rechner mit dem RAID wird jetzt im *Single User Mode* neu gestartet. Dann werden die Rechnernamen und Adressen des alten Mailservers übernommen, damit auf den Arbeitsplatzrechnern nichts umkonfiguriert werden muß. Der NFS-Export der Verzeichnisse `/var/spool/mail/` und `/home/alias/` wird wieder aktiviert. Und `sendmail` so konfiguriert, daß nach dem nächsten Booten ein SMTP-Port geöffnet wird. Außerdem wird nach dem Booten der POP3-Daemon gestartet. Jetzt wird der neue Mailserver neu gebootet und sollte wieder dieselbe Funktionalität bereitstellen wie der alte Mailserver. Zuletzt noch ein `sendmail -q` auf dem *secondary* ausführen, damit die Mails, die sich während des Mailserver-Ausfalls angesammelt haben, jetzt zugestellt werden.

[14] Mit `scp` lassen sich Dateien verschlüsselt von einem Rechner auf einen anderen übertragen; außerdem läßt sich noch eine Kompression aktivieren, um die Übertragung der Dateien zu beschleunigen.

28. Hazard Analysis der neuen Konfiguration

28.1 FTA des neuen Mailservers

Der im vorigen Kapitel entworfene, fehlertolerante Mailserver soll nun einer eingehenden Analyse unterworfen werden. Dazu verwenden wir, wie schon für den alten Mailserver, die Methode der Fault Tree Analysis, wie sie im Abschnitt 25.3.4 auf Seite 172 eingeführt wurde.

Durch die Verwendung derselben Analysemethode sind die Unterschiede in der Konzeption zwischen dem alten Mailserver und dem neuen, fehlertoleranten Mailserver einfach auszumachen. Aus diesem Grund beschränken wir uns in diesem Abschnitt auch auf die Erläuterung der voneinander abweichenden Bereiche der Fault Tree Analysis. Für alle anderen, nicht explizit behandelten Bereiche gelten daher weiterhin die Erkenntnisse der Fault Tree Analysis des alten Mailservers aus Kapitel 26 auf Seite 179.

Die grobe Untergliederung der Fault Tree Analysis sowie die Reihenfolge der Unterbäume bei der Erläuterung wurde dabei in diesem Kapitel beibehalten, so daß die Orientierung leicht fallen sollte.

Die vollständigen Fault Trees in graphischer Notation finden sich in Anhang G auf Seite 381.

28.1.1 Mailserver (MS) erbringt nicht die spezifizierte Leistung

Durch die Konzeption des neuen Mailservers als Doppelrechnersystem ändert sich auch auf oberster Ebene die Voraussetzung für das Eintreten des Top Hazards. Der fehlertolerante Mailserver kann einige der spezifizierten Dienste auch noch erbringen, wenn einer der Teilrechner ausgefallen ist.

Im Detail ist dies der SMTP–Dienst, über den Mails von Clients aus dem Fachbereich bzw. von externen Rechnern angenommen werden. Da die MX–Einträge (siehe Abschnitt 27.1 auf Seite 195) entsprechend gesetzt sind, wird beim Ausfall des SMTP–Dienstes auf dem primären Mailserver der sekundäre Mailserver angesprochen, der ebenfalls Mails annehmen kann. Ein kompletter Ausfall des SMTP–Dienstes ist damit erst gegeben, wenn auf beiden Teilrechnern der SMTP–Dienst ausfällt.

```
[MS erbringt nicht die spezifizierte Leistung]

        ||
        *Spool-Dir nicht über NFS erreichbar*

        *MS kann Mail nicht über POP3 bereitstellen*

        [MS kann Mail über SMTP nicht annehmen]

                &&
                *1. MS kann Mail über SMTP nicht annehmen*
                *2. MS kann Mail über SMTP nicht annehmen*
```

```
*1. MS kann Mail nicht korrekt verarbeiten*
```

Da sowohl für das Bereitstellen des Spool–Verzeichnisses als auch für die Bereitstellung der Mails via POP3 der Zugriff auf die Mail–Festplatte[1] von Nöten ist, kann für diese Dienste keine Verbesserung der Verfügbarkeit erreicht werden. Erst durch einen manuellen Eingriff, nämlich dem Umhängen der Mail–Platten, kann der sekundäre Mailserver zum primären werden. Die so verringerte Ausfallzeit hat jedoch keinen Einfluß auf die Erkenntnisse dieser Fault Tree Analysis.

28.1.2 Spool–Verzeichnis nicht über NFS erreichbar

Wie schon erläutert, wirkt sich das Doppelrechner–Konzept nicht verbessernd auf die Verfügbarkeit des Spool–Verzeichnisses über NFS aus, da die Mail–Festplatte immer nur von einem Teilrechner zur Zeit angesprochen werden kann[2].

Um nun trotzdem eine Verbesserung der Situation zu erreichen, ist die Mail–Festplatte in ein redundantes Festplatten–Array ausgelagert worden. Der für den alten Mailserver als nicht weiter verfolgter Event angenommene Fall, daß die Mail–Festplatte nicht eingebunden werden kann, muß nun näher untersucht werden. Dies geschieht in einem zusätzlichen Teilbaum.

```
*Mail-HD nicht gemounted*
```

28.1.3 Mail-HD nicht gemounted

Der verwendete RAID–Controller ist so konfiguriert, daß die beiden gespiegelten Festplatten nach außen (also für das Betriebssystem) als einfache Festplatte erscheinen. Diese virtuelle Festplatte, auch Host–Drive genannt, ist Voraussetzung dafür, daß die Mail–Festplatte eingebunden werden kann.

Durch einen physikalischen Defekt des RAID–Controllers würde das Host–Drive nicht mehr zur Verfügung stehen. Ebenso kann ein Fehler in der RAID–Firmware, also der auf dem RAID–Controller vorhandenen Software zur Ansteuerung und Verwaltung der Festplatten, ein Grund für eine Nichtverfügbarkeit sein[3].

```
*Mail-HD nicht gemounted*

        ||
        [RAID-Host-Drive nicht sichtbar]

                ||
                (RAID-Controller phys. defekt)
                (Fehler in RAID-Firmware)
```

Die RAID–Firmware sowie gleichwertige Verwaltungsprogramme für den RAID–Controller, die unter dem eingesetzten Betriebssystem lauffähig sind, ermöglichen eine komplexe Konfiguration. Neben einer Spiegelung

[1] Diese ist am primären Mailserver angeschlossen.

[2] Dieser Teilrechner wird dadurch automatisch zum aktivem Teilrechner bzw. primären Mailserver.

[3] Dieser Fall trat bei einem der Controller im Testbetrieb auf.

(RAID–Level 1) sind auch komplexere Redundanzmechanismen (RAID–Level 4 und 5) einstellbar. Durch die relativ unübersichtliche Konfiguration des von uns verwendeten Controllers kann es zu Fehleinstellungen kommen, die in einer Nichtverfügbarkeit des Host–Drives resultieren.

Zuletzt ist noch ein möglicher Hardware–Defekt zu betrachten. Da die Festplatten, die das Host–Drive bilden, in einzelnen, externen Gehäusen untergebracht sind, besteht die Gefahr, daß die SCSI–Bus–Verbindung vom aktiven Teilrechner zu den externen Festplatten unterbrochen wird[4].

```
<Fehler in RAID-Konfiguration>
(Rechner-nach-RAID-SCSI-Kabel defekt)
```

Durch die entsprechende Konfiguration des RAID–Controllers ist das Host–Drive als Spiegelplatte ausgelegt. Ein Ausfall dieser virtuellen Festplatte ist also dann gegeben, falls keine der beiden als Spiegel eingesetzten physikalischen Festplatten mehr ansprechbar ist. Die Bedingungen für den Ausfall jeder Platte sind dabei identisch, weshalb hier nur die erste Platte betrachtet wird.

Eine physikalische Festplatte ist aus Sicht des RAID–Controller naturgemäß nicht mehr ansprechbar, falls diese nicht mit Strom versorgt wird. Der Grund hierfür könnte ein Ausfall des Gehäusenetzteils sein, welches die jeweilige Festplatte mit Strom versorgt. Ebenso könnte die externe lokale Stromversorgung ausgefallen sein, beispielsweise durch einen Defekt des Stromkabels zum externen Gehäuse. Der naheliegendste Grund ist jedoch der physikalische Defekt der Festplatte selbst.

Die analogen Events für die zweite physikalische Festplatte sind hier nicht gesondert aufgeführt.

```
[keine physikalischen Drives ansprechbar]

        &&
        [1. phys. Drive nicht ansprechbar]

                ||
                [keine Stromversorgung]

                        ||
                        (externes Gehäusenetzteil \
                         defekt)
                        <externer lokaler \
                         Stromausfall>

                (1. phys. Drive defekt)

        [2. phys. Drive nicht ansprechbar]
```

Da es sich bei dem RAID–Controller um ein SCSI–Controller handelt, sind alle Geräte an einem Bus angeschlossen. Wird dieser Bus unterbrochen, ist durch die fehlende Termination keine sichere Übertragung gewährleistet. Deshalb muß ebenfalls das Verbindungskabel zwischen den beiden externen Festplatten berücksichtigt werden. Ein Defekt dieses Kabels hat damit eine Nichtverfügbarkeit aller physikalischen Festplatten zur Folge.

```
(RAID-nach-RAID-SCSI-Kabel defekt)
```

[4] Auch dieser Defekt trat in der Testphase auf, ein Pin des Steckers war umgeknickt.

Die Einbindung des Host–Drives auf dem primären Mailserver erfolgt beim Booten des Rechners durch ein Boot–Skript. Da die Erkennung, ob die Mail–Festplatte am jeweiligen Teilrechner angeschlossen ist, nicht trivial ist, sind hier Fehler nicht auszuschließen. Ebenso ist es denkbar, daß das Verzeichnis, unter welchem die Mail–Festplatte eingebunden werden soll, bereits belegt ist. Dies ist z.B. der Fall, wenn unabsichtlich das lokale Mail–Verzeichnis, welches nur für den sekundären Mailserver eine Rolle spielt, eingebunden wurde.

Darüberhinaus können sonstige Fehler in der Software–Konfiguration der Grund für eine Nichteinbindung der Mail–Festplatte sein, z.B. das simple Vergessen eines Eintrags in der Datei für die einzubindenden Partitionen (`/etc/fstab`).

```
[lokaler Konfigurations-Fehler]

        ||
        (Fehler im FTS-Bootscript)
        (Mount-Point schon belegt)
        <sonst. SW-Config-Fehler, z.B. in /etc/fstab>
```

28.1.4 Teil-Mailserver (MS) nicht im Netz erreichbar

Die Gründe für die Nichterreichbarkeit eines Teilrechners entsprechen weitgehend denen aus Abschnitt 26.1.3 auf Seite 181 für einen einzelnen Rechner. Die Tatsache, daß nun zwei Rechner im System vorhanden sind, hat naturgemäß keinen Einfluß auf das Verhalten eines einzelnen Rechners.

Durch die veränderte Hardware gegenüber dem alten Mailserver ergeben sich Änderungen bzw. Ergänzungen in diesem Teilbaum.

Eine Nichterreichbarkeit eines Teilrechners könnte auch „planmäßig“ durch einen Watchdog ausgelöst sein, der einen Reboot ausgelöst hat, um ein nicht anderweitig lösbares (Ressourcen-)Problem zu beheben. Trotz des absichtlichen Auslösens ist dieser Zustand weiterhin als Fehlerzustand zu betrachten.

Daneben ist durch die Verwendung eines SCSI–Controllers auf der Hauptplatine („onboard“) statt eines separaten Controllers für die internen Festplatten des Teilrechners eine veränderte Fehlerquelle zu vermerken.

```
[System-Halt oder temporärer Reboot z.B. durch \
 Watchdog]

        ||
        [System-Platte weg]

                ||
                (System-Platten-Defekt)
                (onboard-SCSI-Defekt)
                <interner Stromausfall>
```

28.1.5 Teil-Mailserver (MS) überlastet

Analog zu Abschnitt 26.1.4 auf Seite 183.

28.1.6 Mailserver (MS) kann Mail nicht über POP3 bereitstellen

Analog zu Abschnitt 26.1.5 auf Seite 185, dabei wird der Fall, daß die Mail–Festplatte nicht eingebunden werden kann, wiederum in einem separaten Teilbaum (Abschnitt 28.1.3 auf Seite 206) behandelt (ähnlich wie in Abschnitt 28.1.2 auf Seite 206).

28.1.7 1. Mailserver (MS) kann Mail über SMTP nicht annehmen

Eine wesentliche Verbesserung des fehlertoleranten Mailservers stellt die neue Konzeption zur Annahme von Mail dar. Es ist beiden Teilrechnern möglich, Mails über das SMTP–Protokoll anzunehmen, wobei nur der primäre Mailserver diese Mails auch sofort an andere Mailserver weiterleiten kann. Der sekundäre Mailserver hingegen nimmt zwar Mails an, leitet diese jedoch, sofern möglich, an den aktiven Mailserver weiter. Das genaue Konzept wird in Abschnitt 27.1 auf Seite 195 erläutert.

Durch diese unterschiedliche Konfiguration der Teilrechner ist auch für die Fault Tree Analysis eine getrennte Betrachtung notwendig. Der primäre Mailserver verhält sich dabei analog zum alten Mailserver, weshalb dieser Teilbaum dem alten Teilbaum aus Abschnitt 26.1.6 auf Seite 187 entspricht. Dabei ist wiederum die Einbindung der Mail–Festplatte in einem separaten Teilbaum behandelt (Abschnitt 28.1.3 auf Seite 206).

28.1.8 2. Mailserver (MS) kann Mail über SMTP nicht annehmen

Wie oben erläutert, muß der sekundäre Mailserver Mails an den primären Mailserver weiterleiten. Da er als sekundärer Mailserver keinen Zugriff auf die Mail–Festplatte hat, aber trotzdem nach außen einen vollwertigen SMTP–Dienst anbietet, unterscheidet sich seine Konfiguration von der des ersten Mailservers.

Die von ihm angenommenen Mails werden auf der internen Festplatte gespeichert, die als Mail–Queue fungiert. Mails, die an Personen außerhalb des Fachbereichs adressiert sind, können an die entsprechenden externen Mailserver weitergeleitet werden. Mails, die an Mitglieder des Fachbereichs adressiert sind, also auf die Mail–Festplatte geschrieben werden sollen, werden zwischengespeichert. In periodischen Abständen wird versucht, diese an den primären Mailserver weiterzuleiten.

Für den sekundären Mailserver muß also der Teilbaum um die Untersuchung der Einbindung der internen Mail–Queue erweitert werden.

Die Ursachen für ein nicht mögliches Ablegen von Mails in der Mail–Queue sind dabei dieselben, wie im alten Mailserver.

Ein Ablegen der Mails ist nicht möglich, falls die Mail–Queue nicht im System eingebunden (gemounted) ist. Da es sich bei der Festplatte, auf der die Mail–Queue residiert, um eine interne Festplatte des Systems handelt, sind die Gründe für eine Nichterreichbarkeit dieser Platte analog zu den Gründen, die eine Nichterreichbarkeit des Teilrechners durch des Ausfall der Systemplatte nach sich ziehen (Abschnitt 28.1.4 auf der vorherigen Seite).

```
            [Mail kann nicht gequeued werden]

                        ||
```

```
[Mail-Queue-Verzeichnis nicht gemounted]

        ||
        [Mail-Queue-Partition nicht erreichbar]

                ||
                <interner Stromausfall>
                (onboard-SCSI-Defekt)
                (Festplatten-Defekt)

        (Mount-Point schon belegt)
        <sonst. SW-Config-Fehler, z.B. in \
         /etc/fstab>
```

Ebenso verhindert ein defektes bzw. gefülltes Dateisystem ein Ablegen der Mails in der Mail–Queue (analog zum alten Mailserver, Abschnitt 26.1.6 auf Seite 187).

```
<Mail-Queue Filesystem defekt>
<Mail-Queue Partition voll>
```

28.1.9 1. Mailserver (MS) kann Mail nicht korrekt verarbeiten

Im Gegensatz zur Annahme von Mails, erfolgt die weitere Verarbeitung von Mails (Filterung, lokale Zustellung) immer auf dem primären Mailserver.

Neu gegenüber der Betrachtung in Abschnitt 26.1.7 auf Seite 189 ist die Behandlung von Mails, die vom sekundären Mailserver angenommen wurden. Diese Queue soll periodisch geleert werden und die enthaltenen Mails an den primären Mailserver weitergeleitet werden. Dieser Vorgang soll auch beim Systemstart des primären Mailservers geschehen. Dies ist dann sinnvoll, wenn der sekundäre Mailserver zum primären wird und die bisher in der Queue liegenden Mails verarbeitet werden sollen.

Die Leerung dieser Mail–Queue kann nun scheitern, falls der primäre Mailserver zu diesem Zeitpunkt überlastet ist. Dieser Aspekt wird in Abschnitt 28.1.5 auf Seite 208) ausführlich erläutert.

Daneben kann auch ein Fehler im Bootskript, welches ermittelt, welcher Teilrechner der aktive und welcher der passive wird, verhindern, daß die Mail–Queue geleert wird. In diesem Fall würde die Mail–Queue auf der Mail–Festplatte die Mail–Queue auf der Systemplatte überlagern und die abgelegten Mails nicht sichtbar sein. Wir hoffen dennoch, diesen Fall ausgeschlossen zu haben.

```
[Mail-Queue des 2. MS kann nicht geleert werden]

        ||
        *1. MS überlastet*

        (Fehler im FTS-Bootscript)
```

28.1.10 Fazit der Analyse

Wie schon am relativ geringen Umfang der Änderungen gegenüber der Fault Tree Analysis des alten Mailservers zu erkennen ist, sind die Verbesserungen des Konzeptes eher allgemeiner Natur und wirken sich nicht auf jede einzelne Komponente des Mailservers aus.

Durch die Verwendung eines Doppelrechner–Systems wird bereits auf der obersten Ebene des Fault Trees eine Verbesserung deutlich (Abschnitt 28.1.1 auf Seite 205). Im Gegensatz zum alten System ist die Verfügbarkeit bei der Annahme von Mail (SMTP) nun erheblich verbessert worden.

Da eine Fault Tree Analysis über keine zeitbehaftete Dimension verfügt, sind in ihr einige Vorteile des neuen Konzepts nicht sichtbar. Durch die Verfügbarkeit zweier Teilrechner, die als primärer Mailserver fungieren können, reduziert sich die Ausfallzeit bei einem Hardware–Defekt des aktiven Teilrechners erheblich. Eine wesentliche Motivation für den Einsatz des Doppelrechner–Systems wird dadurch nur unzureichend gewürdigt.

Eine weitere Verbesserung des Konzepts ist im Abschnitt 28.1.3 auf Seite 206 zu erkennen. Durch die redundante Auslegung der Mail–Festplatte ist eine höhere Verfügbarkeit und Datensicherheit der Mails bzw. der Mail–Queue gegeben.

Bedingt durch die verwendete Hardware ist leider keine Verbesserung bzgl. des Abholens von Mail (POP3 und NFS) analog zur Annahme von Mails (SMTP) möglich. Die Diskussion dieser Einschränkung finde sich in Abschnitt 30.2 auf Seite 217.

Durch die Natur eines Mailservers bzw. eines Rechners im allgemeinen ist natürlich die grundlegende Konzeption nicht vollständig zu verändern. Trotzdem hätten wir uns eine weitreichendere Verbesserung gewünscht.

29. Verifikation der Komponenten

29.1 Testaufbau

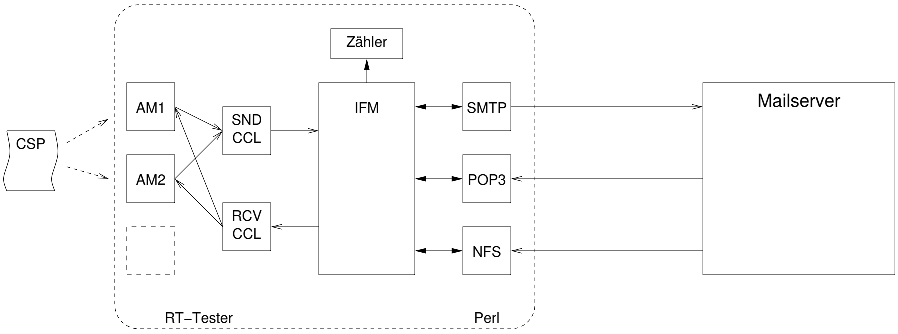


Abbildung 29.1: Testaufbau

Die Verifikation des Gesamtsystems wurde mit einem Test mit Hilfe des Programms RT-Tester durchgeführt. Dabei wurde der Mailserver als Black Box betrachtet, der nur die speziellen Dienste SMTP, POP3 und NFS zur Verfügung stellt. Über diese Schnittstellen wurde der Mailserver während des Tests angesprochen.

Abbildung 29.1 veranschaulicht unseren Testaufbau: Durch eine CSP–Spezifikation wurden das Verhalten von mehreren *Abstract Machines* (AM) spezifiziert. Diese Abstract Machines steuern nun über das *Communication Control Layer* (CCL) , bestehend aus `rcvccl` zum Empfangen, und `sndccl` zum Senden, das *Interface Module* (IFM) an. Dieses Interface Module muß nicht auf dem selben Rechner laufen wie die Abstract Machines und das Communication Control Layer, da es über TCP/IP mit dem CCL kommuniziert, bei unserem Testaufbau ist das aber der Fall.

Das Interface Module erledigt nun die ganze „grobe Arbeit“. Es interpretiert die Signale, die vom RT–Tester kommen, und sorgt dafür, daß die zum IFM gehörigen Perl–Programme `send_test_mail.pl` (SMTP), `receive_pop.pl` (POP3) und `receive_nfs.pl` (NFS) die Kommunikation mit dem Mail–Server aufnehmen.

Das Versenden und Empfangen wurde über Perl-Programme realisiert, da Perl durch seine String-Operationen besonders zum Vergleichen der empfangenen EMails geeignet ist.

Es wurden während des Tests gleichzeitig verschiedene Mails aus einer Mail-Bibliothek über SMTP verschickt, und es wurden gleichzeitig die Mails über POP3 und das über NFS exportierte Mail-Verzeichnis `/var/spool/mail/` die empfangenen EMails ausgelesen. Diese konnten mit den verschickten EMails verglichen werden.

Später wurde noch ein Zähler angesteuert, damit die Anzahl der empfangenen Mails mit der Anzahl der verschickten verglichen werden kann. Die Ansteuerung erfolgt über Message–Queues, zum einen, weil die Message–Queues im Teilprojekt IPC verifiziert wurden, vor allem aber, weil die Kommunikation damit am einfachsten zu realisieren war.

29.2 Gefundene Probleme

29.2.1 POP3

Beim Testen fiel auf, daß manchmal eine Mailbox korrumpiert wurde. Dieses Problem trat dann auf, wenn gleichzeitig über POP3 und über einen NFS-Client versucht wurde, auf die Mail zuzugreifen.

Als Fehlerquelle konnten wir den von RedHat verwendeten POP–Daemon aus dem IMAP–Paket ermitteln. Dieser POP–Daemon schützt die Mailbox nur während des Scans und während der UPDATE-Phase vor einem Zugriff durch andere Programme. Während der gesammten Transaktionsphase ist die Mailbox jedoch ungeschützt.

Im Allgemeinen werden die Mailspool–Dateien in `/var/spool/mail` durch sogenannte Lockdateien geschützt. Diese Dateien haben den Namen der Spooldatei mit angelångtem „`.lock`“. Die Existenz einer solchen Datei weist darauf hin, daß die geschützte Datei (also die ohne „`.lock`“) gerade von einem Programm verändert wird. Sie sollte also gerade nicht von einem anderem Programm geöffnet werden. Alle üblichen Mailprogramme, und auch `procmail` (für die Mail–Zustellung), halten sich an diese Konvention.

Wenn man annimmt, daß außer dem POP–Daemon nur noch `procmail` auf die Mailbox zugreift, dann ist dieses Verhalten unkritisch, weil `procmail` nur weitere Daten anhängt, und der POP-Daemon nur die ersten Daten liest. Es ist auch tatsächlich sehr unwahrscheinlich, daß jemand gleichzeitig mit zwei verschiedenen Methoden auf seine Email zugreifen will.

Trotzdem ist dieses Verhalten nicht konform zur RFC 1939 [MR96], die besagt, daß der POP3–Server nach der Authorisierungsphase bis zur UPDATE–Phase exklusiven Zugriff auf die Mailbox haben muß.

Es erschien uns nicht praktikabel, diesen POP3–Daemon zu patchen, damit er sich konform zur RFC verhält, da der Source–Code sehr unübersichtlich ist. Wir haben daher einen anderen POP3–Daemon eingesetzt, den IDS–POP3–Daemon, der mittlerweile in GNU–POP3–Daemon[1] umbenannt wurde. Dieser wurde mit dem Gedanken geschrieben, möglichst klein, schnell, effizient und konform zu den RFCs zu sein.

Doch auch an diesem POP3–Daemon haben wir Fehler gefunden. Zum einen konnten die Lock–Dateien in der Konfiguration unseres Mail–Servers nicht gelöscht werden, da wir das Sticky–Bit für das Mail–Verzeichnis benutzen, aber die Lock–Dateien nicht dem Benutzer gehören. Zum anderen haben wir eine mögliche Race–Condition beim Test und Erzeugen der Lock–Dateien gefunden. Diese haben wir allerdings nicht durch Testen, sondern durch Analyse des Source–Codes gefunden.

Diese Änderungen kann man in einer Patch-Datei im Anhang K auf Seite 447 finden.

Dadurch, daß die Mailbox jetzt RFC–konform während der gesammten Transaktions–Phase gelockt bleibt, aber `procmail` nach 1024 Sekunden beziehungsweise nach 17 Minuten und 4 Sekunden eine Lock–Datei für

[1] URL: `http://www.nodomainname.net/software/gnu-pop3d.shtml`

ungültig erklärt, müssen wir die Verbindungszeit begrenzen. Dies ist laut RFC auch möglich.

29.2.2 Hängende Mount–Prozesse

Bei längeren Tests kam es vor, daß einzelne Mount–Prozesse hängen blieben. Leider konnten wir dieses Problem nicht näher lokalisieren. Das Problem trat sowohl mit einem Linux–NFS–Server, als auch mit einem Solaris–NFS–Server auf.

Im Linux–Kernel 2.2.14 wurde eine Verklemmung von autofs und NFS behoben (Quelle: `http://www.linuxhq.com/changes/2.2.html#2_2_14`). Eventuell war das der Fehler, auf den auch wir gestoßen sind. Allerdings konnten wir das nicht mehr prüfen.

30. Ungelöste Probleme und mögliche Verbesserungen

30.1 Probleme

Wir haben gute Arbeit geleistet. Dennoch sind am Ende des Projekts ungelöste Probleme geblieben. Dies lag überwiegend daran, daß diese von Faktoren abhängen, auf die wir im Verlauf des Projektes keinen Einfluß nehmen konnten. Einige Probleme haben sich mittlerweile sicherlich „wie von selbst“ gelöst.

Unsere größte Sorge kam während des Testens auf: hängende Mount–Prozesse (siehe Kapitel 29.2.2). Dieses Problem konnte nicht mit voller Sicherheit lokalisiert werden. Wir vermuten jedoch, daß dieses mit einer späteren Kernel–Version behoben wurde.

Ein weiteres Problem ist die benutzerunfreundliche Software von ICP–Vortex zur Konfiguration des RAID–Controllers. Auch haben wir keinen Weg gefunden, eine befriedigende Automatisierung einzuführen. Das bedeutet, daß bei jedem Ausfall einer Platte ein umständliches Tool aufgerufen werden muß, um allein das Alarm–Geräusch zu stoppen. Das Einbinden einer neuen Festplatte in das RAID–System ist aufgrund der komplexen Software eine sehr ansruchsvolle Aufgabe. Neben der Komplexität sind Menüstruktur und Betitelung der einzelnen Punkte eine Hürde. Oft ist man sich nicht ganz sicher, ob nun nicht vielleicht doch die Festplatten überschrieben wurden und alle Mail gelöscht wurde. Trotz aller Schwierigkeiten sollte im laufenden Betrieb die Lösung von Problemen am RAID, also das Austauschen einzelner Festplatten, möglich sein.

Ein weiteres Problem ist der Support von ICP–Vortex. Defekte Controller werden mit etwas Wartezeit bearbeitet, jedoch ist die Organisation innerhalb der Firma offensichtlich nicht die beste. Soll heißen: wir bekamen einen Controller mit nur halb so viel RAM wieder, als wir eingeschickt hatten.

30.2 Verbesserungsvorschläge

Wir haben natürlich weitergedacht. Die Verbesserungsvorschläge beziehen sich also nicht nur auf die ungelösten Probleme.

Zunächst einmal die offensichtlichen Verbesserungsvorschläge. Es wäre schön, wenn die Mount–Prozesse nicht hängenbleiben würden.

Die Konfiguration des RAIDs sollte wesentlich einfacher handhabbar sein. Die ursprüngliche Idee war, daß es von uns automatisierbar sein sollte. Dies war mangels Interface zum Controller nicht möglich.

Desweiteren war es angedacht, die Rechner (halb)automatisch wechseln zu lassen. Das ist aber nicht möglich, da die Controller nicht clusterfähig sind. So müssen die Festplatten manuell „umgehängt“ werden. Es wird dennoch automatisch erkannt, welche Rolle der Rechner innerhalb des Mailsystems hat.

Auch war eine Überlegung, daß ein Handbuch sowie ein Handzettel zu dem gesamten System sehr sinnvoll sind, da unser komplexer Aufbau nicht ohne weiteres zu verstehen ist.

Der wichtigste Punkt jedoch ist eine Oberfläche für die Administratoren, so daß der Status des Mailservers jederzeit zu überwachen ist und bei Fehlfunktion die Administratoren umgehend benachrichtigt werden, ohne eine Mail zu verschicken, was unter Umständen auch nicht die beste Idee wäre. Es wäre auch schön gewesen, über diese Oberfläche das Wechseln der Rollen der Rechner anstoßen zu können. Weitere Administrationsmöglichkeiten lassen sich beliebig einbauen.

30.3 Sicherheit

Nachdem wir uns immer nur mit der Ausfallsicherheit des Mailservers beschäftigt haben, möchten wir jetzt kurz darauf eingehen, daß E-Mail in Bezug auf den Schutz des Inhalts noch lange nicht soweit ist wie die Briefpost. Es ist zwar mit vielen Mailprogrammen möglich, Mails mit *PGP*[1] zu verschlüsseln und auch wieder zu entschlüsseln. Aber trotzdem ist es nicht schwierig, diese verschlüsselten Mails unbemerkt „abzuhören“; denn Absender und Empfänger, sowie das „Subject“ bleiben unverschlüsselt. Jetzt kann also die unbemerkt abgefangene E-Mail mit viel Rechenpower doch noch entschlüsselt werden.

Im Vergleich dazu: Beim Briefverkehr ist nur dem Postboten und dem Sortiersystem bekannt, wer wem einen Brief sendet. Doch die unterliegen bestimmten Gesetzen, so daß sie diese Informationen nicht weitergeben dürfen. Zu dem kann jeder Absender selbst entscheiden, ob er seine Adresse außen auf den Brief schreibt oder nicht.

Um also den E-Mail-Verkehr wirklich sicherer zu gestalten, müsste auch die Kommunikation zwischen zwei Mailservern verschlüsselt werden. Und es müsste auch für Laien möglich sein, die eigene Mailadresse nur dem Empfänger mitzuteilen. Das Verbergen der Absenderadresse könnten aber auch wieder mehr *Spamming* (siehe Abschnitt 27.2.1 auf Seite 197) erzeugen, so daß dies wohl keine große Verbreitung finden würde. Die Verbindung zwischen zwei Mailservern könnte mit Hilfe eines zu `ssh`[2] ähnlichen Verfahrens erreicht werden.

Beim Abholen der Mails via POP3 ergeben sich zwei Sicherheitsprobleme: Zum einen wird das Paßwort, das zur Authentifizierung genutzt wird, unverschlüsselt zum Mailserver übertragen, zum anderen werden die Mails unverschlüsselt zum Empfänger übertragen. In [MR96] wird zwar ein Befehl `APOP` definert, der die verschlüsselte Übertragung des Paßwortes ermöglicht, dieser wird jedoch von so gut wie keinem POP3-Server implementiert. Deshalb würde hier ebenfalls ein zu `ssh` ähnliches Verschlüsselungsverfahren helfen.

Auch über NFS (Network File System) das Verzeichnis `/var/spool/mail/` bereitzustellen, ist nicht sehr sicher, da sich ein Rechner als vertrauenswürdig ausgeben kann, indem er eine vertrauenswürdige IP-Adresse ver-

[1] Pretty Good Privacy (PGP) ist ein asymetrisches Verschüsselungsverfahren, das auf einem geheimen und einem öffentlichen Schlüssel basiert.

[2] `ssh` (Secure Shell) ist eine Programm mit dem sich *remote login* verschlüsselt abwickeln lässt.

wendet. Außerdem sind die Daten, die via NFS übertragen werden nicht verschlüsselt. (vgl. hierzu auch Abschitt 27.2.1 auf Seite 197)

Abschliessend läßt sich der heutige E-Mail-Verkehr mehr mit der Kommunikation über „Buschtrommeln“ vergleichen, als mit dem Briefverkehr über die Post in Bezug auf die Vertrauenswürdigkeit des Inhalts. Es ließen sich also noch viele Verbesserungen finden.

31. Fazit

Das Projekt war, aus unserer Sicht, erfolgreich. Wir haben ein System nach unseren Vorstellungen geschaffen und verifiziert. Natürlich ist ein solches Projekt nie wirklich fertig, da man immer etwas verbessern kann oder neue beziehungsweise bessere Versionen der verwendeten Software installieren kann. In diesem Rahmen haben wir uns in verschiedene Gebiete der Informatik eingearbeitet und auch bewegt.

Obwohl wir uns nicht in den Linux–Kernel, wie die anderen Teilprojekte, eingearbeitet haben, ist unser Teilprojekt dennoch Linux–bezogen. Wir haben Linux in eine von uns erdachte Umgebung eingebettet und in diesem Rahmen getestet. Wir habe also gezeigt, daß es möglich ist, eine ausfallsichere Umgebung als redundanten Mailserver zu schaffen. Dabei haben wir auch festgestellt, daß Software, ob Open Source oder kommerziell, nicht immer den Spezifikationen entspricht. Dies erschwert natürlich die Aufgabe, ein System zu schaffen, welches sich den eigenen Spezifikationen entsprechend verhält.

Sicherlich war die Arbeit zum Teil etwas schleppend, dennoch zu jedem Zeitpunkt zielgerichtet. Die Kommunikation innerhalb des Teilprojektes hat meist ohne Probleme funktioniert und war befriedigend.

Das einzige, was uns negativ erscheint, ist die Kommunikation mit den anderen Teilprojekten und die Tatsache, daß gegen Ende des Projekts die Anzahl der Betreuer rapide gesunken ist.

Teil V

IPC

Das Teilprojekt IPC setzte sich von Anfang an nur aus zwei Personen zusammen: Hans-Jürgen Ficker und Sönke Jarré. Im Laufe der vier Semester hat sich die Personenanzahl auch nicht weiter vergrößert, eher das Gegenteil war der Fall. Auf mehr (Hans-Jürgen) oder weniger (Sönke) dauerhafte Weise haben wir in einem anderen Teilprojekt (FTS) mitgearbeitet/ausgeholfen. Dieses hat aber unserer Arbeit nicht geschadet, sondern sie durch neue Erfahrungen im Hinblick auf Spezifikations- und Verifikationsarten eher noch gefördert.

In unserem Teilprojekt wollten wir einen Teil des Inter–Process–Communication–Packages, oder kurz des IPC–Packages, spezifizieren und verifizieren. Dieses IPC–Package nach System V besteht aus den Message–Queues, Shared Memory und Semaphoren. Es ging uns darum, Spezifikations- und Verifikationsmöglichkeiten kennenzulernen und an einem realen Fall ihre Tauglichkeit auszuprobieren. Aus dem IPC-Package haben wir uns die Message-Queues herausgesucht. Die genauen Gründe für diese Auswahl sind weiter unten aufgeführt.

Im Endeffekt haben wir die Message-Queues in Z spezifiziert und mit dem Tool `Fuzz` ein Typechecking durchgeführt. Anschließend haben wir einen Teil dieser Spezifikation durch Data-Refinement verifiziert.

32. Ziele des Teilprojektes

32.1 Einleitung

Computer spielen in unserem Leben eine immer wichtigere Rolle. Nicht nur, daß inzwischen fast jeder einen eigenen PC besitzt, vielmehr werden in immer mehr Bereichen Computer verschiedenster Form eingesetzt. Sie werden immer schneller, können immer mehr Daten verarbeiten und somit weitere Aufgaben übernehmen.

Doch allein die schnellere und bessere Aufnahmefähigkeit und Verarbeitung von Daten in Computern nützt nichts. Auch die Korrektheit der Daten ist entscheidend. Dies ist vor allem in sicherheitskritischen Bereichen wie der Atomkraft und der Luft- und Raumfahrt der Fall.

Es muß daher von vornherein festgelegt sein, was ein Programm leisten soll. Demnach muß im Vorfeld eine Spezifikation des Programmes erstellt werden, die sein gewünschtes Verhalten beschreibt. Anschließend muß diese Spezifikation in Programmcode überführt werden oder das existierende Programm auf seine Korrektheit bezüglich der Spezifikation getestet werden.

Sämtliche Programme, die heutzutage auf Computern laufen, zu verifizieren ist momentan noch utopisch. Jedoch existiert auf jedem Computer ein Betriebssystem, auf dem die weiteren Programme basieren. Wäre dieses Betriebssystem fehlerhaft, würden auch absolut korrekte Programme nicht mehr fehlerfrei laufen.

Da sich unser Projekt LiVE! mit Linux beschäftigt, gab es nichts naheliegenderes, als dieses für unsere Zwecke zu benutzen.

Den gesamten Linux-Kern zu spezifizieren und verifizieren ist für zwei Personen und dem bißchen Zeit, das uns zur Verfügung stand, nicht möglich. Dies war uns von vornherein klar, also mußten wir uns auf einen kleinen Teil beschränken.

Dieser Teil mußte zudem schon ziemlich stabil sein, da die Erstellung einer Spezifikation und Verifikation schon eine gewisse Zeit umfaßt. Und wenn in dieser Zeit der zugrundeliegende Teil von Linux sich schon mehrfach verändert und gewandelt hätte, wäre unsere Arbeit sinnlos gewesen. Die Arbeit am Linux-Kern geht momentan sehr zügig voran.

Seit der Einführung von Unix System V von AT&T 1983 übernimmt das IPC-Package die Verwaltung der Kommunikation zwischen den Prozessen. Aber auch dieses Package ist noch zu groß für uns, um es zu spezifizieren. Daher haben wir uns für eines dieser Kommunikationsmittel entschieden, die Message–Queues.

Da dies ein sehr wichtiger und häufig benutzter Teil des Betriebssystems ist, der sicher funktionieren muß, ist er in den frühen Jahren von Linux, nämlich 1992, sehr gründlich entwickelt worden. Er ist im November 1997 das letzte mal verändert worden.

Message-Queues sind im Grunde genommen eine Datenstruktur im Linux-Kern. Dabei kann ein Prozeß eine Nachricht hineineinschreiben, ein anderer kann diese auslesen. Die vollständige Spezifikation kann im Anhang eingesehen werden.

Nach unserer Auswahl war es nun endlich möglich, uns in diesen Bereich von Linux einzuarbeiten. Anschließend konnten wir uns in den Bereich der Spezifikation einarbeiten, diese anwenden und uns dann der Verifikation zuwenden.

32.2 Bisheriger Stand der Dinge

Bei Linux verhält es sich so wie bei vielen Programmen, es hat nie die Phasen eines idealen Softwareentwicklungsprozesses durchlaufen. Insbesondere gab es nie eine Anforderungsdefinition, auf der eine spätere Implementierung basiert.

Die Anforderung bei Linux war stets: „Es gibt ein Problem, programmier mal ein bißchen rum, um dieses zu beheben."

Da Linux ein Open-Source-Projekt ist und sich ziemlich viele Programmierer an der Entwicklung beteiligen, ist dies auch kein Problem. Wird später ein Fehler gefunden oder funktioniert etwas nicht richtig, so wird dies schnell von irgend jemandem behoben. Solange dies rechtzeitig, oder überhaupt, bemerkt wird, ist es akzeptabel.

Dabei hat dann auch die Dokumentation gelitten. Es ist nicht niedergeschrieben, welche Teile was genau können sollen. Lediglich im Source Code gibt es teilweise entsprechende Kommentare. Und die erstellte Online–Dokumentation ist leider nicht umfassend und eindeutig.

Mit einer Verifikation hat sich noch niemand beschäftigt, da die Entwicklung von Linux nicht abgeschlossen ist, und es auch wohl nie sein wird.

Obwohl noch nicht fertig gestellt, aber schon benutzbar und stabiler als die meisten Betriebssysteme, ist Linux in den letzten Jahren immer populärer geworden. Immer mehr Computernutzer verwenden Linux. Dies ist eine natürlich sehr positive Entwicklung, allerdings stellen viele Nutzer gewisse Anforderungen an das System. Demnach sollte also dafür gesorgt werden, daß es wirklich stabil wird.

Bei den Message-Queues sieht es dabei folgendermaßen aus: Sie sind 1992 von Krishna Balasubramanian implementiert worden. Zuletzt sind sie im November 1997 von Alan Cox und Andi Kleen geändert worden, wobei hauptsächlich die nicht mehr benötigten `kerneld messages` eleminiert wurden.

Auch hier gibt es keine Spezifikation, die Message-Queues funktionieren halt so, wie sie funktionieren.

Als Dokumentation sind online die Man-Pages vorhanden. Diese sind allerdings nicht *wirklich* vollständig.

32.3 Ziele

Die Ziele für unser Teilprojekt sind schnell aufgezählt.

Zuerst wollen wir eine vollständige Spezifikation der Message-Queues erstellen.

Anschließend wollen wir die Message-Queues verifizieren. Stellen wir dabei Fehler in der Implementierung fest, so wollen wir diese verbessern und bekanntgeben.

Aber so ganz uneigennützig soll es sich ja nicht gestalten. Für uns selbst wollen wir uns zuerst einmal mit dem IPC-Package besser auskennen.

Weiterhin wollen wir uns in die Welt der formalen Spezifikation einarbeiten und in diesem Bereich verschiedene Spezifikationssprachen kennenlernen und eine davon anwenden.

Zuletzt geht es uns darum, die Verifikation kennenzulernen. Wir wollen aus verschiedenen Verifikationsarten eine besonders geeignete heraussuchen und diese anwenden.

So ist es uns also hoffentlich möglich, etwas zu lernen, ein Ergebnis zu erzielen und dieses dann an die Linux-Gemeinde weiterzugeben, in der Hoffnung, daß es ihr auch etwas nützt.

33. Spezifikation

Zuerst haben wir uns mit der Spezifikation allgemein beschäftigt, was das ist und wie sie aussieht. Somit sind wir schnell zu verschiedenen Spezifikationssprachen gekommen, von denen wir uns dann für unsere Spezifikation die Sprache Z ausgesucht haben.

Dann haben wir die Message–Queues in Z spezifiziert und unsere Spezifikation anschließend durch den Typechecker `fuzz` prüfen lassen.

Anschließend haben wir noch geprüft, ob unsere Spezifikation denn überhaupt vollständig ist.

Das einzige Element, das durchgehend auftrat, waren die Probleme, die sich uns in den Weg stellten. Diese beschreiben wir erst gemeinsam am Ende des Kapitels.

33.1 Spezifikation allgemein

Bei einem normalen Softwareentwicklungsprozeß sollte zuerst eine Anforderungsdefinition geschrieben werden. In dieser sollte festgelegt sein, was das zu erstellende System können soll.

Auf welche Art die Anforderungsdefinition erstellt wird, ist nicht vorgeschrieben. Sie kann umgangssprachlich gehalten sein, bis hin zu einer formalen Spezifikation.

Mit einer umgangssprachlichen informellen Spezifikation kann man das System sehr gut beschreiben. Doch leicht werden Details vergessen, oder es treten Widersprüche auf.

Bei der formalen Spezifikation ist ein erheblich größerer Aufwand zu betreiben. Sie ist meist umfangreicher als eine informelle Spezifikation, dafür aber auch vollständig und eindeutig.

Auf welche Art die Spezifikation geschieht, ist stets situationsabhängig. Häufig werden Mischformen benutzt, in denen einige wichtige Teile des Systems formal beschrieben werden, die nicht ganz so entscheidenden Teile aber nur informell.

Anschließend wird die Software entsprechend den Vorgaben implementiert. Aus formalen Spezifikationen läßt sich meist sehr leicht der Aufbau der Implementierung herleiten. So kann der größere zeitliche Aufwand der Spezifikation teilweise wiedergutgemacht werden. Eventuell ist es sogar möglich, die formale Spezifikation in den Code zu überführen.

In der Verifikationsphase wird die Sicherheit und Korrektheit der Implementierung gegenüber der Spezifikation geprüft.

Soviel zur Theorie über den Ablauf eines Softwareentwicklungsprozesses. Bei uns ist das natürlich alles ganz anders. Es ist eine Implementierung vorhanden, die auch schon über einen längeren Zeitraum stabil läuft.

Um diese zu verifizieren, benötigen wir eine genaue Spezifikation. Leider ist solch eine Spezifikation bisher noch nicht erstellt worden, die Message-Queues sind einfach „eingehackt“ worden.

Nun ist es unsere Aufgabe, zuerst eine solche Spezifikation zu erstellen. Hierfür können wir auf verschiedene Dokumentationen zurückgreifen.

Zuerst einmal gibt es den Code der fertigen Implementierung. Diesen haben wir im Anhang mit angegeben (siehe Anhang N auf Seite 463).

Doch bringt eine Spezifikation an Hand dieses Codes für die Verifikation nichts. Schließlich wollen wir Unterschiede zwischen dem gewünschten und tatsächlichem Verhalten aufzeigen.

Demnach haben wir uns an die bei der Implementierung erstellte Online–Dokumentation gehalten. Dieses sind speziell die Man–Pages, die wir ebenfalls dem Anhang beigefügt haben (siehe Anhang M auf Seite 457).

Diese Man–Pages sind als eine Art informelle Spezifikation anzusehen, obwohl sie bei genauer Betrachtung eigentlich nur das tatsächliche Verhalten der Message-Queues beschreiben. Allerdings erscheint es uns so, daß sie einer vorab erstellten Anforderungsdefinition noch am nächsten kommen.

Da die Man–Pages jedoch informell, oder semi–formell sind, entstehen an einigen Stellen Ungenauigkeiten. Daher mußten wir an der einen oder anderen Stelle doch den Code zu Rate ziehen, beziehungsweise kleine Programme schreiben und das tatsächliche Verhalten austesten, um eine vollständige Spezifikation erstellen zu können. Dieses Testen fand jedoch nicht nur auf der Linux–Plattform statt, sondern zusätzlich noch auf einem Solaris–System.

Die anschließende Verifikation beschreiben wir im Kapitel 34 auf Seite 249.

33.2 Wahl der Spezifikationssprache

Die informelle Spezifikation wird in natürlicher Sprache notiert, entsprechend dem, wie man es spricht. Für die formale Spezifikation hingegen werden spezielle Sprachen verwendet, in denen man die Spezifikation notiert.

Diese Sprachen unterliegen bestimmten syntaktischen Regeln. Jede hat ihre Besonderheiten und ist dadurch für einige Fälle besser, für andere Fälle schlechter geeignet. Man kann die Spezifikationssprachen in einige Kategorien aufteilen.

Da gibt es zunächst einmal die Spezifikationssprachen, die hauptsächlich die Architektur eines Systems wiedergeben. Dies wären z.B. OSM–Modelle.

Dann gibt es noch die Spezifikationssprachen, die den zeitlichen Verlauf berücksichtigen.

Weiterhin gibt es die Spezifikationssprachen, die die Parallelität von Ereignissen besonders gut darstellen können. Da wäre vor allem CSP zu nennen, das auch in anderer Stelle im Projekt verwendet wird. Hier ist der Vorteil, daß es eine hervorragende Toolunterstützung gibt. Mit FDR kann man die syntaktische Korrektheit überprüfen und untersuchen, ob Deadlocks entstehen. Mit RT-Tester kann man Implementationen auf Korrektheit gegen die CSP–Spezifikation testen.

Und abschließend seien noch die algebraischen Spezifikationssprachen erwähnt, mit denen sich vor allem Datenstrukturen sehr schön modellieren lassen. Ein typischer Vertreter wäre Z, aber auch ASpecT wurde bei uns im Projekt verwendet. Einige dieser Sprachen lassen sich sehr leicht in funktionale Sprachen überführen, die dann ausführbar sind.

Und wie sieht es nun bei uns aus? Message-Queues sind, vereinfacht gesagt, Datenstrukturen im Kernel. Es kommt also hauptsächlich auf die korrekte Spezifikation dieser Strukturen an.

Demnach haben wir uns entschieden, mit der Spezifikationssprache Z zu arbeiten.

Z ist eine algebraische Spezifikationssprache und dementsprechend sehr mengenbasiert. Es gibt zuerst einmal einen Zustandsraum, die hier aufgeführten Bedingungen müssen stets gelten.

Die häufigste Struktur in Z sind Schemata. Ein Schema teilt sich in zwei Teile auf, die durch einen waagerechten Strich getrennt sind. Im ersten Teil stehen die Ein- und Ausgaben sowie eventuelle Einbindungen anderer Schemata, speziell des Zustandraumes. Die dort geltenden Bedingungen werden hierdurch importiert.

Im zweiten Teil eines Schemas werden die Vorbedingungen, die für den Eintritt des Schemas gelten müssen, die Veränderungen auf dem Zustandsraum und die Nachbedingungen, die nach der Benutzung des Schemas gelten, angeführt.

Nun ist es möglich, mehrere solcher Schemata miteinander zu kombinieren. Hierdurch kann man so ein ganzes System darstellen.

Z hat dabei den Nachteil, daß sämtlichen Operationen in Nullzeit ablaufen. Demnach kann man einen zeitlichen Verlauf in Z nicht darstellen. Auch kann man Parallelität nicht darstellen, und es kann auch Nichtdeterminismus auftreten.

Dies ist nicht die richtige Stelle, um eine vollständige Beschreibung von Z zu geben. Wer dieses möchte, der sei auf die Bücher „Using Z – Specification, Refinement, and Proof“ von Jim Woodcock and Jim Davies [WD96] und „The Z Notation: A Reference Manual“ von J. M. Spivey [Spi92] verwiesen.

Um aber die folgenden Abschnitte auch für den Nicht–Z–Experten verständlich zu halten, findet sich im Anhang L auf Seite 451 eine Auflistung der wichtigsten Symbole aus Z.

33.3 Spezifikation der Message-Queues

Bei unserem Versuch, die Message-Queues zu spezifizieren, haben wir festgestellt, wie komplex ein Beispiel aus der Realität wird. Zuerst haben wir ganz einfach angefangen und versucht die grundlegenden Verhaltensweisen zu spezifizieren. Schnell wurde es jedoch immer umfangreicher und komplexer, zusätzliche Funktionalität mußte mit eingebaut werden und der Zustandsraum wurde immer größer.

Wir hatten dabei zwischenzeitlich Fehler eingebaut, die uns erst später auffielen. Meist waren sie klein und schnell zu beheben, aber es gab auch größere Fehler, die uns nötigten, die gesamte Spezifikation zu überarbeiten.

Die gesamte Spezifikation in diesem Bericht wiederzugeben, wäre nicht sinnvoll. Sie umfaßt mittlerweile 67 Seiten. Daher werden wir einen Auszug vorstellen, an Hand dessen das Vorgehen bei der Spezifikation offensichtlich wird.

Für diejenigen, die hierdurch auf den Geschmack gekommen sind und die ganze Spezifikation anschauen wollen, haben wir die vollständige Spezifikation in den Anhang O auf Seite 473 gestellt.

33.3.1 Datenstrukturen

In diesem Kapitel widmen wir uns zuerst dem Grundlegenden: den Datenstrukturen, mit denen wir später arbeiten.

Sowohl in diesem, als auch in den folgenden Kapiteln, werden wir versuchen, für die bessere Verständlichkeit möglichst aussagekräftige Namen zu verwenden. Hierbei haben meist die im Source-Code verwendeten Namen als Vorlage gedient. Allerdings geht an einigen Stellen diese bessere Verständlichkeit zu Lasten der Lesbarkeit, da sich lange Wörter mit mathematischen Symbolen gemischt nur schwer lesen lassen. Wir hoffen aber einen guten Kompromiß gefunden zu haben.

Unsere formale Spezifikation soll möglichst abstrakt gehalten werden, auch wenn wir den Source-Code teilweise bei der Erstellung hinzugezogen haben. Wir wollen schließlich eine Beschreibung des nach Außen sichtbaren Verhaltens erstellen und keine Wiedergabe interner Abläufe.

Wie das übliche Vorgehen bei einer Z-Spezifikation ist, können wir nicht genau sagen. Die kleinen Beispiele aus den Z-Lehrbüchern konnten dies nicht ganz wiedergeben.

Wir haben bei der Spezifikation mit dem grundlegenden Teil der Message–Queues angefangen, entsprechende Datenstrukturen erdacht, auf diesen Funktionen definiert und betrachtet, ob damit die Message–Queue–Operationen dargestellt waren. Anschließend haben wir alles verfeinert, mehr Features der Message-Queues hinzugenommen und dabei teilweise die Datenstrukturen abwandeln oder ergänzen müssen.

Entsprechend abstrakt sind einige Datenstrukturen und Funktionen, vor allem solche, die Systemdienste oder Systemstrukturen darstellen. Diese könnten jedoch in anderen Teilen des Betriebssystems, in denen sie benutzt oder bereitgestellt werden, genauer definiert werden, es ist aber in dieser Spezifikation nicht nötig.

Dieses Kapitel beginnt mit den Basistypen, die grundlegend vorhanden sein müssen und mit denen, oder auf denen, die Message-Queues arbeiten.

Anschließend werden die Konstanten eingeführt, die systemweit feststehen sollen.

Daran schließen sich die Basisfunktionen an, die im Prinzip jedem Prozeß zur Verfügung stehende Systemdienste darstellen.

Basistypen

Bei den Basistypen kann man zwei verschieden Arten unterscheiden.

Mit der einen Art wird nicht weiter gerechnet, die einzelnen Elemente müssen nur einem bestimmten Typen entsprechen. Also kann man sie wunderschön abstrakt halten.

Bei der anderen Art sieht es schon schwieriger aus, mit ihnen soll auch gerechnet werden. Dementsprechend muß eine Struktur mitspezifiziert werden, eventuell sogar noch entsprechende Rechenoperationen auf dieser Struktur. Da es sich aber bei der Struktur häufig um Zahlen handelt, sind diese Rechenoperationen schon implizit mitgegeben.

In einem System gibt es zuerst einmal mehrere Benutzer. Diese haben eine eindeutige Identifikation, die User-ID aus der Menge UID.

$[UID]$

Da wir uns in UNIX-artigen Systemen aufhalten, gibt es für jeden User auch eine zugehörige Group-ID, diesmal aus der Menge GID.

$[GID]$

Ein Benutzer (user), eine Gruppe (group), oder „die ganze Welt" (other) können das Recht bekommen, in eine Message-Queue zu schreiben, beziehungsweise aus ihr zu lesen. Diese Rechte sind ähnlich den UNIX-Dateirechten, nur das execute-Bit hat keine Auswirkungen und wurde deshalb auch weggelassen. Man kann es im realen System sogar setzen, nur hat es keinerlei Auswirkungen.

$PERM ::= u\_write \mid u\_read \mid g\_write \mid g\_read \mid o\_write \mid o\_read$

Ein Benutzer startet Prozesse. Diese müssen teilweise untereinander einander oder mit anderen Prozessen kommunizieren. Jeder Prozeß hat daher systemweit eine eindeutige Identifikation, den Prozeßidentifikator.

$[PID]$

Für die Kommunikation zwischen den Prozessen können Message-Queues verwendet werden. Für diese Kommunikation müssen verschiedene Prozesse dieselbe Message-Queue adressieren können. Dies geschieht über den KEY.

$KEY : \mathbb{P}(\mathbb{N} \cup \{0\})$

Für die einzelnen Operationen wird die Message-Queue jedoch über einen Identifikator adressiert.

$ID : \mathbb{P}(\mathbb{N} \cup \{0\})$

In einer Message-Queue werden Nachrichten gesendet. Diese bestehen aus einer Folge von Bytes und einem Typ für die Nachricht.

Da die Struktur der Folgen von Bytes nicht weiter relevant ist, nehmen wir eine Folge von Bytes als einen abstrakten Datentypen. Dies ist ein schönes Beispiel für die Abstraktion, denn eigentlich hätte man hier einzelne

Bytes als Datentypen nehmen müssen, die sich wiederum aus Bits zusammensetzen.

$[BYTES]$

Der Typ einer Nachricht wird in Form einer ganzen Zahl angegeben. Zu beachten ist, daß negative Zahlen zwar nicht als Typ einer Nachricht erlaubt sind, sie aber bei der Message-Queue Operation $msgrcv$ eine spezielle Bedeutung haben.

$$TYPE : \mathbb{P}\,\mathbb{Z}$$

Mit einer Message-Queue können verschiedene Operationen ausgeführt werden, wie sie in den folgenden Kapiteln noch ausführlich beschrieben werden.

Bei den Operationen $msgrcv$ und $msgget$ können dabei einige Flags als Option mitgegeben werden.

$$\begin{aligned} FLAG ::= {} & MSG\_EXCEPT \mid IPC\_NOWAIT \mid MSG\_NOERROR \mid \\ & IPC\_CREAT \mid IPC\_EXCL \end{aligned}$$

Der Operation $msgctl$ können bestimmte Kommandos als Argumente mitgegeben werden.

$$CMD ::= IPC\_STAT \mid IPC\_SET \mid IPC\_RMID$$

Message–Queue–Operationen können eine Anzahl von Fehlern verursachen:

$$\begin{aligned} ERROR ::= {} & EACCES \mid EAGAIN \mid EEXIST \mid EINVAL \mid ENOENT \mid \\ & ENOMSG \mid E2BIG \mid ENOSPC \mid EPERM \mid NOERROR \end{aligned}$$

Und als Abschluß kann einem Prozeß als Returnwert mitgeteilt werden, ob ein Fehler vorliegt (Returnwert -1) oder es eine erfolgreiche Operation war (Returnwert 0 oder eine positive ganze Zahl).

$$RETURN : \mathbb{P}(\mathbb{N} \cup \{0, -1\})$$

Konstanten

Es gibt eine Reihe von Konstanten, die einmalig systemweit festgelegt sind. Diese sind insoweit konstant, als daß Änderungen nur durch eine Neukompilierung des Kernels vorgenommen werden können.

Die wirklich reale Größe, also die Definition der einzelnen Konstanten, ist an dieser Stelle nicht wichtig und sollte an anderer Stelle im System erfolgen.

Um eine mißbräuchliche Nutzung des Systems zu verhindern, beziehungsweise um an keine Systemgrenzen zu stoßen, gibt es gewisse Begrenzungen für die Message-Queues.

In einem System sollte es eine Obergrenze für die Anzahl der Message-Queues geben. Daher gibt es eine Konstante , die diese maximale Anzahl angibt:

$MSGMNI : \mathbb{N}$

Damit eine Message-Queue nicht beliebig voll geschrieben werden kann, gibt es eine maximale Länge einer Message-Queue:

$MSGMNB : \mathbb{N}$

Auch Nachrichten sollten eine gewisse Länge nicht überschreiten.

$MSGMAX : \mathbb{N}$

Zudem gibt es einige spezielle Elemente aus den im vorherigen Abschnitt deklarierten Basistypen, die für einige Operationen benannt sein müssen.

So wird für $msgget$ ein spezielles Element aus KEY benötigt, um eine neue Message–Queue ohne KEY zu erzeugen. eine solche Message–Queue kann nur vom Prozess, der sie erzeugt hat, und von seinen Kindprozessen angesprochen werden.

$IPC\_PRIVATE : KEY$

Auch wird ein spezielles Element aus UID benötigt, das den Superuser (root) kennzeichnet:

$SUPERUSER : UID$

Basisfunktionen

Im folgenden werden einige Basisfunktionen eingeführt, die Systemdienste beschreiben, die den einzelnen Prozessen jeweils zur Verfügung stehen.

Einige dieser Systemdienste kommen aus anderen Bereichen des Betriebsystems, wie zum Beispiel dem *virtual file system*, und werden von uns nicht näher spezifiziert. (Dieses müßte bei der Spezifikation dieser Bereiche geschehen.)

Es werden zuerst einmal einige Funktionen benötigt, die sich um die Rechtevergabe kümmern.

So gibt es eine Funktion, die zu einer UID alle GIDs der Gruppen zuordnet, in denen der jeweilige User ist:

$getallgroup : UID \rightarrow \mathbb{P}\, GID$

Auch gibt es Funktionen, die einer PID die UID bzw. die GID des Prozesses zuordnet. Dieses sind partielle Funktionen, da nur verwendeten PID ein Wert zugeordnet ist.

$$getuid : PID \rightarrow\!\!\!\!\!\!+\; UID$$
$$getgid : PID \rightarrow\!\!\!\!\!\!+\; GID$$

Zum Lesen und Schreiben auf einer Message-Queue werden ebenfalls Funktionen benötigt.

So ermittelt eine allgemeine Funktion, ob ein User auf eine Message-Queue *schreiben* darf, wobei diese Message-Queue bestimmte Rechte, eine bestimmte UID und eine bestimmte GID hat:

$$
\begin{array}{l}
\_ \text{ maywrite } \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID) \\
\hline
\forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet \\
u1 \text{ maywrite } (p, u2, g) \Leftrightarrow (u1 = u2 \land u\_write \in p) \lor \\
\qquad (g \in getallgroup(u1) \land g\_write \in p) \lor \\
\qquad o\_write \in p
\end{array}
$$

Eine andere allgemeine Funktion ermittelt, ob ein User von einer Message-Queue *lesen* darf, welche bestimmte Rechte, eine bestimmte UID und eine bestimmte GID hat:

$$
\begin{array}{l}
\_ \text{ mayread } \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID) \\
\hline
\forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet \\
u1 \text{ mayread } (p, u2, g) \Leftrightarrow (u1 = u2 \land u\_read \in p) \lor \\
\qquad (g \in getallgroup(u1) \land g\_read \in p) \lor \\
\qquad o\_read \in p
\end{array}
$$

Betrachten wir eine einzelne Nachricht, so benötigen wir noch eine Funktion, die nur das Anfangsstück der Länge n der $BYTES$ zurückgibt. (Da wir $BYTES$ sehr abstrakt gehalten haben können wir an dieser Stelle diese Funktion nicht näher spezifizieren.)

$$truncate : BYTES \times \mathbb{N} \rightarrow BYTES$$

33.3.2 Message-Queue Zustandsraum

Nach unserer Sichtweise sind Message-Queues Kernel-Datenstrukturen. In diesem Abschnitt geht es nun darum, den Zustandsraum, in dem sich diese Datenstrukturen befinden, zu beschreiben.

Hierfür gibt es einige Objekte, die den Zustand der Menge der Message-Queues charakterisieren. Für diese gelten gewisse Konsistenzbedingungen.

Um nicht zu sehr von der gegebenen Implementierung zu abstrahieren, haben wir die im Source-Code vorhandenen Strukturen teilweise übernommen.

Insgesamt denken wir, mit folgenden zehn Objekten den Zustandsraum hinreichend charakterisieren zu können:

- $getkey$ gibt zu jeder ID einen eindeutigen KEY zurück.
- $msgq$ weist einer ID eine Message-Queue zu.
- $creator$ gibt den Erzeuger einer Message-Queue an.
- $owner$ gibt den Besitzer einer Message-Queue an.
- $group$ gibt die Gruppe einer Message-Queue an.
- $perm$ zeigt die Rechte an, die ein Benutzer für die Message-Queue besitzt.

- *qbytes* liefert die maximale Länge einer Message-Queue.
- *cbytes* gibt die Länge einer Message-Queue wieder.
- *msg_ts* gibt für jede Nachricht aus einer Message-Queue die Länge dieser Nachricht an.
- *errno* gibt die Art des Fehlers aus.

Diese zehn Objekte, ihr aktueller Zustand und ihre Veränderungen, sind von nun an bei allen Schemata von Bedeutung.

Die Abstraktion des Zustandraumes für die Kommunikation mit Hilfe von Message-Queues sieht nun folgendermaßen aus:

$$
\begin{array}{l}
\textit{MsgQStateSpace} \\
\hline
getkey : ID \mapsto KEY \\
msgq : ID \mapsto \operatorname{seq}(TYPE \times BYTES) \\
creator : ID \mapsto UID \\
owner : ID \mapsto UID \\
group : ID \mapsto GID \\
perm : ID \mapsto \mathbb{P}\, PERM \\
qbytes : ID \mapsto \mathbb{N} \\
cbytes : ID \mapsto \mathbb{N} \\
msg\_ts : ID \mapsto \operatorname{seq} \mathbb{N} \\
errno : ERROR \\
\hline
\# \operatorname{dom} getkey \leq MSGMNI \qquad (1) \\
\operatorname{dom} msgq = \operatorname{dom} getkey \qquad (2) \\
\operatorname{dom} msgq = \operatorname{dom} creator \qquad (3) \\
\operatorname{dom} msgq = \operatorname{dom} owner \qquad (4) \\
\operatorname{dom} msgq = \operatorname{dom} group \qquad (5) \\
\operatorname{dom} msgq = \operatorname{dom} perm \qquad (6) \\
\operatorname{dom} msgq = \operatorname{dom} qbytes \qquad (7) \\
\operatorname{dom} msgq = \operatorname{dom} cbytes \qquad (8) \\
\operatorname{dom} msgq = \operatorname{dom} msg\_ts \qquad (9) \\
\forall\, i : ID \mid i \in \operatorname{dom} msgq \bullet \operatorname{dom}(msgq(i)) = \operatorname{dom}(msg\_ts(i)) \qquad (10) \\
\forall\, i1, i2 : ID \mid i1 \in \operatorname{dom} msgq \wedge i2 \in \operatorname{dom} msgq \bullet \qquad (11) \\
\qquad getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \vee getkey(i1) = IPC\_PRIVATE
\end{array}
$$

Dabei haben die in der unteren Hälfte aufgestellten Konsistenzbedingungen folgende Bedeutung:

- (1): Die Anzahl der Message-Queues im System darf nicht größer sein, als im System insgesamt mit *MSGMNI* erlaubt ist.
- (2) bis (9): Wenn es für eine bestimmte *ID* eine Message-Queue gibt, dann gibt es für diese *ID* auch *creator*, *owner*, *group*, *perm*, *qbytes*, *cbytes* und *msg_ts*.
- (10): Jeder Nachricht in einer Message-Queue wird auch eine Länge zugeordnet. Dies bedeutet, daß wenn einer *ID* eine Message-Queue zugeordnet ist, *msg_ts* diese *ID* auf eine Sequenz von natürlichen Zahlen abbildet, die jeweils die Länge der einzelnen Nachrichten beschreiben.
- (11): Für alle Elemente aus *KEY*, außer für *IPC_PRIVATE*, gilt, daß sie immer eindeutig sind. Dies bedeutet, daß niemals zwei Message-

Queues über ein und denselben KEY angesprochen werden können, mit der einzigen Ausnahme, $IPC\_PRIVATE$.

33.3.3 Systeminitialisierung

Nachdem wir nun den Zustandsraum für die Message-Queues festgelegt haben, geht es in diesem Abschnitt um die Startbedingungen, sprich die Systeminitialisierung.

Beim Start des Betriebssystems gibt es noch keine Prozesse, diese müssen erst noch erzeugt werden. Daher existieren auch noch keine Message-Queues, diese werden erst später initialisiert.

Entsprechend einfach stellt sich somit die Initialisierung dar: Da noch keine Message-Queues existent sind, ist der Domain, das heißt, „die linke Seite“, der benutzten Funktionen leer.

$Init$

$MsgQStateSpace'$

$$
\begin{array}{l}
\mathrm{dom}\, getkey' = \varnothing \\
\mathrm{dom}\, msgq' = \varnothing \\
\mathrm{dom}\, creator' = \varnothing \\
\mathrm{dom}\, owner' = \varnothing \\
\mathrm{dom}\, group' = \varnothing \\
\mathrm{dom}\, perm' = \varnothing \\
\mathrm{dom}\, qbytes' = \varnothing \\
\mathrm{dom}\, cbytes' = \varnothing \\
\mathrm{dom}\, msg\_ts' = \varnothing \\
errno' = NOERROR
\end{array}
$$

Für die mit Z noch nicht so bewanderten Leser:

Dieser Kasten ist ein Schema, auf der oberen Linie steht der Name des Schemas, in diesem Falle $Init$. In der oberen Hälfte des Schemas, also über dem Querstrich, werden sämtliche Ein- und Ausgabewerte deklariert sowie andere Schemata eingebunden. Mit einem Quote (’) wird ein Zustand, der nach dem Schema gelten muß, gekennzeichnet.

Anschließend werden sämtliche Objekte des Zustandsraumes und ihre eventuellen Veränderungen aufgezeigt.

33.3.4 Message-Queue Operation msgsnd()

Die Operation $msgsnd()$ wird von den Prozessen dazu benutzt, um in einer vorhandenen und ihnen bekannten Message-Queue Nachrichten zu senden. Eine Nachricht besteht dabei aus einem Typ und einer Anzahl von Bytes.

Es gibt hier eine Aufteilung, und zwar in einen erfolgreichen Fall und in mehrere Fehlerfälle. Dadurch wird die Spezifikation lesbarer und auch leichter Verständlich.

Im erfolgreichen Fall muß der Prozeß eine bestehende Message–Queue angeben, in die er seine Nachricht senden will. Dazu muß er natürlich Schreibrechte für diese Message–Queue besitzen. Von der Nachricht selbst muß er den Typ und die Länge der Nachricht wissen.

Die hier genannten Fehlerfälle sind keine Fehler im Sinne eines Systemfehlers, sondern das gewünschte Verhalten tritt für den Benutzer nicht ein.

Der erste solche Fehlerfall ist der Versuch in eine Message-Queue zu senden, die schon voll ist.

Weiterhin kann es passieren, daß der Prozeß für die Message-Queue keine Schreibrechte besitzt.

Und es ist noch möglich, daß es eine falsche Eingabe gibt. Solch eine Eingabe könnte eine ID sein, die überhaupt nicht existiert, oder ein ungültiger Typ einer Nachricht, oder aber die Nachricht ist zu lang.

Es gibt auch einige Fehlerfälle, die wir in unserer Spezifikation nicht anführen. Hierfür gibt es zweierlei Gründe.

Zum einen werden einige dieser Fehlerfälle in anderen Teilen des Betriebsystems abgefangen. Hierunter fallen die Situationen:

- ein Pointer zeigt auf eine nicht existierende Adresse
- die Message-Queue wurde inzwischen gelöscht
- das System hat nicht genügend Speicher, um den gesamten Inhalt der Message-Queue zu speichern

Zum anderen existiert in der Spezifikationssprache Z keine Zeitkomponente, ein zeitlicher Verlauf und insbesondere das Warten auf ein Ereignis kann nicht dargestellt werden. Daher kann in unserer Spezifikation das Warten auf eine Message-Queue und gleichzeitige Erhalten eines Interruptes nicht angeführt werden.

Für den Benutzer ist dies wieder völlig unwichtig, er benutzt $msgsnd()$ und erwartet ein Ergebnis. Um dieses abstrakt darzustellen, erstellen wir ein User-Interface, in dem wir das nach außen sichtbare Verhalten spezifizieren.

In diesem Teil fangen wir mit dem Erfolgsfall an, beschreiben anschließend die verschiedenen Fehlerfälle gehen näher auf die nicht spezifizierten Fehlerfälle ein, und erstellen zum Schluß das User-Interface.

Somit erhalten wir abschließend eine abstrakte formale Spezifikation des Verhaltens der Message-Queue Operation $msgsnd()$.

Erfolgsfall

Die erste Hälfte aller Schemata für $msgsnd()$ sieht gleich aus, um bei der Erstellung des User-Interfaces sämtliche Schemata miteinander kombinieren zu können, beziehungsweise weil die Aufrufsyntax der Operation stets gleich ist.

Zuerst wird stets der Zustandsraum eingebunden, auf dem Veränderungen stattfinden können.

Dann gibt es als Eingaben die Prozeßidentifikation PID sowie die Identifikation einer Message-Queue ID. Um eine Nachricht zu senden, benötigt man eine Nachricht, insbesondere ihren Typen und eine Anzahl von $BYTES$. Auch die Länge der Nachricht muß angegeben werden. Eventuell können auch Flags vorgegeben werden.

Als Rückgabe erhält man einen Returnwert, bei den Fehlerfällen wird dieser stets auf -1, beim Erfolgsfall auf 0 gesetzt.

Der Erfolgsfall ist das erfolgreiche Senden einer Nachricht in einer Message-Queue. Es muß natürlich eine existierende Message-Queue adressiert sein, und der Prozeß muß für diese Message-Queue auch Schreibrechte besitzen.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\mathit{Msgsnd}} \\
\Delta \mathit{MsgQStateSpace} & \\
t? : \mathit{TYPE} & \\
b? : \mathit{BYTES} & \\
\mathit{msgsz}? : \mathbb{N} & \\
p? : \mathit{PID} & \\
i? : \mathit{ID} & \\
f? : \mathbb{P}\,\mathit{FLAG} & \\
r! : \mathit{RETURN} & \\
\hline
i? \in \operatorname{dom} \mathit{msgq} & (1) \\
\mathit{getuid}(p?) \text{ maywrite } (\mathit{perm}(i?), \mathit{owner}(i?), \mathit{group}(i?)) & (2) \\
\mathit{cbytes}(i?) + \mathit{msgsz}? \leq \mathit{qbytes}(i?) & (3) \\
\mathit{msgsz}? \leq \mathit{MSGMAX} & (4) \\
\mathit{msgsz}? \geq 0 & (5) \\
t? > 0 & (6) \\
r! = 0 & (7) \\
 & \\
\mathit{getkey}' = \mathit{getkey} & \\
\mathit{msgq}' = \mathit{msgq} \oplus \{i? \mapsto \mathit{msgq}(i?) \frown \langle (t?, b?) \rangle\} & \\
\mathit{creator}' = \mathit{creator} & \\
\mathit{owner}' = \mathit{owner} & \\
\mathit{group}' = \mathit{group} & \\
\mathit{perm}' = \mathit{perm} & \\
\mathit{qbytes}' = \mathit{qbytes} & \\
\mathit{cbytes}' = \mathit{cbytes} \oplus \{i? \mapsto \mathit{cbytes}(i?) + \mathit{msgsz}?\} & \\
\mathit{msg\_ts}' = \mathit{msg\_ts} \oplus \{i? \mapsto \mathit{msg\_ts}(i?) \frown \langle \mathit{msgsz}? \rangle\} & \\
\mathit{errno}' = \mathit{errno} & \\
\hline
\end{array}
$$

In der zweiten Hälfte des Schema sind die Vor- und Nachbedingungen angegeben:

- (1): Die *ID* muß eine schon existierende Message-Queue adressieren.
- (2): Der Prozeß muß auf der durch die *ID* bezeichnete Message-Queue Schreibrechte besitzen.
- (3): Durch die zu sendende Nachricht darf die Länge der Message-Queue die maximale Länge von Message-Queues nicht überschreiten.
- (4): Die Länge der Nachricht darf nicht die maximale Länge einer Nachricht überschreiten.
- (5): Genauso darf die Länge einer Nachricht nicht einem negativen Wert entsprechen.
- (6): Der Typ der Nachricht muß positiv sein.

- (7): Daraufhin gibt es als Returnwert die 0 als Zeichen des erfolgreichen Sendens der Nachricht.

Auch die Objekte des Zustandraumes unterliegen gewissen Änderungen, wobei die Konsistenzbedingungen des Zustandraumes nicht verletzt werden.

Da wäre zuerst $msgq$, bei der an die existierende Message-Queue die Nachricht mit Angabe des Typs hinten angehängt wird.

Bei $cbytes$ wird die Länge der Message-Queue um die Länge der gesendeten Nachricht erhöht.

Und bei msg_ts wird noch die Länge für die letzte Nachricht eingefügt.

Die restlichen Objekte bleiben vom Senden einer Nachricht unberührt und somit unverändert bestehen.

Fehlerfall

Einige der möglichen Fehlerfälle sind in unserer Spezifikation nicht enthalten. Dieses hat natürlich seine Gründe. Entweder wird der Fehlerfall nicht in dem von uns spezifizierten Teil des Betriebssystems, den Message-Queues, abgefangen oder er ist in Z nicht darstellbar.

Gehört ein Fehlerfall nicht in den Bereich der Message-Queues, so ist unsere Spezifikation natürlich weiterhin vollständig. Dieser Fall würde bei der Spezifikation des entsprechenden Teils des Betriebsystems mit zu erfassen sein.

Leider gibt es auch den Fall, daß die Sprache Z nicht mächtig genug ist, insbesondere durch das Fehlen einer zeitlichen Komponente. Diesen Fall müßten wir eigentlich mit spezifizieren, können dies aber nicht. Unsere Spezifikation ist an dieser Stelle also nicht mehr vollständig; treten entsprechende Vorbedingungen auf, so geht sie in einen unspezifizierten Zustand über.

Dieses läßt sich leider nicht vermeiden, aber in anderen Spezifikationssprachen hätten wir größere Probleme bekommen.

Hier sind die von uns nicht mitspezifizierten Fehlerfälle:

- EFAULT: Ein Pointer zeigt auf eine nicht existierende Adresse. Hierfür ist der Kernel zuständig
- EIDRM: Die Message-Queue ist gelöscht worden, während der Prozess blockiert war. Dies können wir aufgrund der fehlenden zeitlichen Komponente von Z nicht spezifizieren.
- EINTR: Der Prozeß wartet auf eine Message-Queue und erhält einen Interrupt. Dies kann wegen der fehlenden Zeitkomponente in Z nicht dargestellt werden.
- ENOMEM: Das System hat nicht genügend Speicher, um den gesamten Inhalt der Message-Queue zu speichern. Hierfür ist der Kernel zuständig.

Es gibt bei allen Fehlerfällen Gemeinsamkeiten, die wir hier anführen wollen, um sie nicht stets ansprechen zu müssen.

Die erste Hälfte des Schemas entspricht immer den schon beschriebenen üblichen Ein- und Ausgaben bei $msgsnd()$.

Der Returnwert $r!$ wird auf den für Fehlerfälle üblichen Wert von -1 gesetzt.

Für den Zustandsraum gilt allgemein, daß $errno$ auf den Namen des Fehlers gesetzt wird, also den Namen des Schemas ohne das vorgefügte $msgsnd$.

Die restlichen Objekte des Zustandraumes unterliegen keinerlei Veränderungen, werden deshalb hier auch nicht aufgeführt. Durch das Einbinden des Zustandsraumes ist dieses schon implizit vorgegeben.

Nun kann der Fall eintreten, daß das Senden einer Nachricht in eine Message-Queue erfolglos ist, da die Message-Queue schon voll ist. Gleichzeitig muß das Flag IPC_NOWAIT gesetzt sein.

$msgsndEAGAIN$

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \qquad (1) \\
cbytes(i?) + msgsz? > qbytes(i?) \qquad (2) \\
IPC\_NOWAIT \in f? \qquad (3) \\
r! = -1 \\
errno' = EAGAIN
\end{array}
$$

Damit dieses Schema ausgeführt werden kann, muß eine ID vorgegeben werden, für die eine Message-Queue existiert (1). Auch muß die Länge der Message-Queue mit der Länge der zu sendenden Nachricht die maximale Länge dieser Message-Queue überschreiten (2). Zudem muß das Flag IPC_NOWAIT gesetzt sein (3).

Weiter geht es mit dem Fall, daß das Senden einer Nachricht erfolglos ist, da der Prozeß für die angegebene Message-Queue keine Schreibrechte besitzt.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad msgsndEACCES \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \qquad (1) \\
\neg\, (getuid(p?) \text{ maywrite } (perm(i?), owner(i?), group(i?))) \qquad (2) \\
r! = -1 \\
errno' = EACCES \\
\hline
\end{array}
$$

Für den Eintritt dieses Schema muß wieder ein gültige *ID* angegeben sein (1). Der Prozeß darf aber keine Schreibrechte für diese Message-Queue besitzen (2).

Der nächste Fall tritt bei verschiedenen Möglichkeiten ein, stets ist jedoch eine falsche Eingabe vorgegeben.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad msgsndEINVAL \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \notin \operatorname{dom} msgq \lor \qquad (1) \\
\quad t? \leq 0 \lor \qquad (2) \\
\quad msgsz? < 0 \lor \qquad (3) \\
\quad msgsz? > MSGMAX \qquad (4) \\
r! = -1 \\
errno' = EINVAL \\
\hline
\end{array}
$$

Entweder wird hier eine falsche Message-Queue-ID vorgegeben (1), oder der Typ einer Nachricht ist nicht richtig (2), oder aber die Länge der Nachricht ist zu klein (3) oder zu groß (4).

User-Interface

Die Aufteilung in die verschiedenen Fälle ergibt sich für den Benutzer nicht, er sieht nur das Verhalten einer einzelnen Operation und erhält entsprechend seiner Vorgaben Rückgaben und Veränderungen. Um dieses nach außen sichtbare Verhalten wiederzuspiegeln, fügen wir alle erstellten Schemata zusammen. Dies geschieht mit einer Schemadisjunktion.

Da hier der seltene Fall eintritt, daß wir den Erfolgsfall nicht weiter unterteilen mußten, brauchen wir für ihn auch keine Schemadisjunktion durchzuführen.

Tritt ein für den Benutzer nicht gewünschtes Ereignis ein, so haben wir hierfür die Fehlerfälle. Weiter vorne haben wir einige Fälle angeführt, die wir in unserer Spezifikation nicht darstellen.

Ansonsten ergibt sich die Gesamtzahl der Fehler aus der Schemadisjunktion:

$$Msgsndfault \mathrel{\widehat=} msgsndEAGAIN \lor msgsndEACCES \lor msgsndEINVAL$$

Für den Benutzer ergibt sich die Unterteilung in Erfolgs- und Fehlerfall nicht, er ruft nur eine einzige Operation auf. Daher führen wir auch noch diese beiden Fälle zusammen:

$$msgsnd \mathrel{\widehat=} Msgsnd \lor Msgsndfault$$

Somit entspricht das formal spezifizierte, abstrakte $msgsnd$ dem nach außen sichtbaren Verhalten der Message-Queue Operation $msgsnd()$.

33.4 Typechecking

Die Spezifikation sollte nun auf ihre syntaktische Korrektheit geprüft werden. Hierfür haben wir ein Tool benutzt, den `fuzz`.

`fuzz` arbeitet auf dem LaTeX–Source und meldet syntaktische Fehler. Am Anfang erschienen uns die Fehlermeldungen jedoch gewöhnungsbedürftig und nicht sehr aussagekräftig.

Zusätzlich zu dem Syntaxcheck vollführt `fuzz` noch einen Typecheck. Dabei werden die verwendeten Typen bei den Operationen überprüft.

Nach einigen Fehlerbehebungen läuft unsere Spezifikation inzwischen anstandslos durch den `fuzz`.

Es werden momentan noch weitere Tools entwickelt, um eine anschließende Verifikation, mehr oder weniger automatisch, zu unterstützen. Doch sind sie noch nicht weit genug, um uns bei unserem Problem helfen zu können. Demnach ist die Verifikation „von Hand“ zu erledigen.

33.5 Vollständigkeit der Spezifikation

Da wir natürlich nicht nur eine korrekte, sondern auch eine vollständige und eindeutige Spezifikation erstellen wollen, ist nun dieses noch zu überprüfen. Auch dies wird noch nicht durch Tools unterstützt.

In Z wird mit Mengen gearbeitet und bei bestimmten Zuständen treten bestimmte Schemata in Kraft. Demnach müssen wir also mit den Vorbedingungen arbeiten, die vor dem Eintritt eines Schemas gelten.

Für die Vollständigkeit der Spezifikation müssen sämtliche möglichen Vorbedingungen abgedeckt sein. Für die Eindeutigkeit der Spezifikation dürfen bei zwei verschiedenen Schemata nicht dieselben Vorbedingungen vorhanden sein.

Für das Beispiel $msgsnd$ sehen die Vorbedingungen der verschiedenen Schemata wie in Abbildung 33.1 auf der nächsten Seite aus.

Mit Hilfe dieser Tabelle kann man jetzt feststellen, daß $msgsnd$ vollständig und eindeutig ist.

Operation	ID	msgsz	qbytes	TYPE	WRITE/READ
Msgsnd	vorhanden	$(msgsz \geq 0) \wedge (msgsz \leq MSGMAX)$	$msgsz + cbytes \leq qbytes$	> 0	maywrite
msgsndEAGAIN	vorhanden		$msgsz + cbytes > qbytes$		
msgsndEACCESS	vorhanden				$\neg$ maywrite
msgsdEINVAL	nicht vorhanden				
msgsdEINVAL	vorhanden	$(msgsz < 0) \vee (msgsz > MSGMAX)$			
msgsdEINVAL	vorhanden			≤ 0	

Abbildung 33.1: Vorbedingungen der Schemata von $msgsnd$

Jedoch sollten wir hier noch einmal darauf aufmerksam machen, daß wir einige Fälle aus den Message-Queues in Z nicht darstellen konnten. Daraufhin haben wir diese in der Spezifikation nicht weiter beachtet, genauso wenig die für diese Fälle gesetzten Flags. Somit mußten wir, um trotzdem eine vollständige Spezifikation zu erhalten, diese Flags in sämtlichen Fällen unberücksichtigt lassen.

33.6 Probleme und Ergebnisse der Spezifikation

Bei der Spezifikation traten natürlich einige Probleme auf. Daß sie insgesamt wesentlich umfangreicher und komplexer als gedacht wurde, sei nur am Rande erwähnt.

In den vorherigen Abschnitten ist es schon des öfteren angeklungen: in

Z gibt es keine zeitliche Komponente. Somit können wir nicht den vollen Umfang der Message-Queues spezifizieren.

Als ein kleines Beispiel, was damit gemeint ist, sei hier bei den Fehlerfällen der Operation $msgsnd$ der Fall $msgsndEINTR$ angeführt.

Ein Prozeß versucht eine Nachricht in einer Message–Queue zu senden. Diese ist allerdings schon voll, und das Flag IPC_NOWAIT ist nicht gesetzt. Daraufhin würde der Prozeß so lange warten, bis in der Queue genügend Platz für seine Nachricht frei würde. Während dieser Wartezeit erhält er einen Interrupt.

Da man das Warten, was ja ein zeitlicher Verlauf ist, nicht darstellen kann, können wir diesen Fall in der Spezifikation nicht weiter berücksichtigen. Wohl aber den Fall, wenn IPC_NOWAIT gesetzt ist.

Aber nicht immer war es so einfach zu entscheiden. Es gab Situationen, da wurde es aus der Dokumentation nicht eindeutig klar, wie denn das gewünschte Verhalten sein solle.

In diesen Situationen blieb uns nichts anderes übrig, als ein kleines Testprogramm zu schreiben und das tatsächliche Verhalten der Message-Queues auszuprobieren.

Doch unser zwischenzeitlich größtes Problem entstand, da wir die Dokumentation an einer Stelle nicht richtig interpretierten.

Message-Queues erhalten einen systemweit eindeutigen KEY. Nach unserer Interpretation wurden die Message-Queues auch stets über diesen KEY angesprochen. Doch dem ist leider nicht so, sie werden über ihre ID angesprochen.

Daraufhin mußten wir die gesamte Spezifikation überarbeiten und die Relationen aus dem Zustandsraum entsprechend abändern.

Trotz dieser Probleme und der immer komplexeren Situation haben wir eine vollständige und eindeutige Spezifikation erstellt.

Zudem haben wir viel über Spezifikationen allgemein und den Ablauf einer Spezifikation gelernt. Und wir haben die Spezifikationssprache Z mit ihren Vor- und Nachteilen kennengelernt.

34. Verifikation

In diesem Kapitel widmen wir uns nun der Verifikation eines Teilbereiches der Message-Queues, der Operation $msgsnd()$. Die vollständige Verifikation haben wir in den Anhang gestellt, an dieser Stelle wird ein Einblick in unsere Arbeit gegeben, der diese möglichst genau wiederspiegeln soll (siehe Anhang P auf Seite 549).

Entsprechend fangen wir auch mit einer allgemeinen Beschreibung von Verifikation an, wann und wofür sie überhaupt benötigt wird, wie sie aussehen sollte. Hier beschreiben wir auch kurz verschiedene Verifikationsarten.

Etwas ausführlicher beschreiben wir dann die von uns verwendete Verifikationsart des *Data Refinement*.

Daran schließt sich ein Ausschnitt aus unserer Verifikation an, der einen möglichst guten Einblick in unsere Arbeit geben soll. Hierbei verwenden wir die im vorherigen Kapitel beschriebene und vollständig im Anhang vorhandene Spezifikation von Message-Queues (siehe Anhang O auf Seite 473).

Zum Abschluß äußern wir uns noch zu Problemen, die uns bei diesem Verfahren aufgefallen sind, sowie zu Problemen, die sich bei diesem speziellem Fall ergaben.

34.1 Verifikation allgemein

Verifikation könnte man grob gesagt als das Beweisen, daß ein Programm wunschgemäß läuft, beschreiben.

Dies ist natürlich immer wünschenswert, doch gibt es einige Bereiche, in denen Fehler sich sehr fatal auswirken, sogar Menschenleben gefährden. In diesen sicherheitskritischen Bereichen ist das ordnungsgemäße Funktionieren von Programmen Voraussetzung. Zu diesen Bereichen gehören z.B. die Luft- und Raumfahrt, die Eisenbahn und Atomkraftwerke.

Beim heimischen PC ist dies nicht ganz so tragisch, zwar kommen Systemabstürze vor oder es werden falsche Daten angezeigt, doch ist dies jedenfalls selten lebensbedrohend.

Es gibt verschiedenste Arten der Verifikation. Sie fangen an bei „einmal anschauen und sagen, daß schon alles stimmen wird“ über Testen des korrekten Funktionierens bis hin zu seitenlangen formalen mathematischen Beweisen.

Die informellen Verifikationen sind weder vollständig noch korrekt. Fehler werden leicht übersehen und bestimmte Situationen erst gar nicht beachtet. Daher eignet sich diese Art der Verifikation nicht für unsere Zwecke.

Beim Testen gibt es wiederum mehrere Möglichkeiten. Läuft das Programm, so kann man per Hand versuchen, möglichst viele und interessante Testfälle abzudecken. Hierbei findet man im Normalfall schon die ersten Fehler, jedoch ist auch hier keine Vollständigkeit gegeben.

Man kann auch das Programm an möglichst viele Anwender geben, auf die Rückmeldungen warten und dann entsprechend nachbessern. Auch diese Möglichkeit kommt für uns nicht in Frage, obwohl dies meist die Vorgehensweise bei Open-Source-Projekten ist.

Oder man kann durch Toolunterstützung versuchen, alle relevanten Testfälle zu erzeugen und das Programm gegen eine gewünschte Spezifikation laufen lassen. Der Vorteil hierbei ist die Vollständigkeit, Nachteil ist der enorme Aufwand. Auch lassen sich auf heutigen Rechnern mit den momentan zur Verfügung stehenden Tools keine allzugroßen Programme testen. Daher kam auch diese Variante für uns nicht in Frage.

Bleibt für uns nur noch die formale Verifikation. Hierbei hat man eine formale Spezifikation des Programmes und versucht, über Beweise darzulegen, daß diese Spezifikation mit der Implementierung übereinstimmt.

Diese Beweise können dabei z.B. über die Vor- und Nachbedingungen gehen, die vor und nach einer Operation herrschen müssen, oder aber aus mathematische „Beweisschritte“ bestehen, die die Überführung von der Spezifikation in die Implementierung darlegen.

34.2 Data Refinement

Aus den oben beschriebenen Verifikationsarten haben wir uns für das Data Refinement entschieden.

Hierfür sprach, daß ein Data Refinement vollständig und korrekt ist. Auch hatten wir uns bei der Spezifikation für die mengenbasierte Spezifikationssprache Z entschieden, da lag eine entsprechende Beweisführung schon nahe.

Normalerweise verhält es sich bei einem Data Refinement so, daß man eine formale Spezifikation erstellt hat. Diese versucht man durch mathematische Beweise immer konkreter umzuformen, bis man schließlich aus einer konkreten Spezifikation direkt die Implementierung erstellt.

Dies ist bei uns so nicht möglich, die Implementierung ist schon vorhanden. Daher erstellen wir aus der Implementierung eine konkrete Spezifikation. Anschließend versuchen wir, die formale Spezifikation in die konkrete Spezifikation zu überführen. Treten dabei Differenzen auf oder ist dies nicht möglich, so muß ein Fehler vorliegen, entweder in einer unserer Spezifikationen oder in der Implementierung.

34.3 Verifikation von msgsnd()

Wie weiter oben schon beschrieben ist dieses Kapitel größtenteils ein Auszug unserer Verifikation, die sich auch im Anhang dieses Projektberichts befindet (Anhang P auf Seite 549). Dort ist auch der vollständige Source-Code vorhanden.

Ebenfalls verwenden wir unsere Spezifikation der Message-Queues, die sich im Anhang O auf Seite 473 befindet. Allerdings verwenden wir hieraus nur Teile, die auch im vorherigen Kapitel schon beschrieben wurden. Somit ist niemand genötigt, die gesamte Spezifikation und Verifikation zum Verständnis dieses Teils durchzuarbeiten.

Die verwendeten Datenstrukturen entsprechen denen der Spezifikation, die wir schon im Abschnitt 33.3.1 auf Seite 234 angeführt haben.

34.3.1 Message-Queue Zustandsraum

Nach unserer Sichtweise sind Message–Queues Kernel–Datenstrukturen. Hier geht es nun darum, den Zustandsraum, in dem sich diese Datenstrukturen befinden, zu beschreiben.

Hierfür gibt es einige Objekte, die den Zustand der Menge der Message–Queues charakterisieren. Für diese gelten gewisse Konsistenzbedingungen.

Der abstrakte Zustandraum ist aus der Spezifikation übernommen und enthält die Sichtweise, wie es anhand der Dokumentation aussehen sollte.

Im konkreten Zustandsraum haben wir versucht, den Source-Code direkt in Z-Notation zu übernehmen.

Weiterhin existiert noch eine Relation, die eine Verbindung zwischen abstrakter Sicht und konkreter Sicht darstellt.

Der abstrakte Zustandsraum

Die Abstraktion des Zustandraumes für die Kommunikation mit Hilfe von Message-Queues befindet sich im Abschnitt 33.3.2 auf Seite 239.

Der konkrete Zustandsraum

Der konkrete Zustandsraum basiert auf dem Source-Code. Allerdings haben wir nur einen Teil konkretisiert, um an diesem den Ablauf exemplarisch zu verdeutlichen.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{CMsgQStateSpace}} \\
\hline
msgque : (0 \mathrel{..} MSGMNI - 1) \rightarrow SPECIALS & (1) \\
msg\_perm\_key : (0 \mathrel{..} MSGMNI - 1) \nrightarrow KEY & (2) \\
msg\_perm\_seq : (0 \mathrel{..} MSGMNI - 1) \nrightarrow (\mathbb{N} \cup \{0\}) & (3) \\
msgq : ID \nrightarrow \mathrm{seq}(TYPE \times BYTES) & \\
creator : ID \nrightarrow UID & \\
owner : ID \nrightarrow UID & \\
group : ID \nrightarrow GID & \\
perm : ID \nrightarrow \mathbb{P}\, PERM & \\
qbytes : ID \nrightarrow \mathbb{N} & \\
cbytes : ID \nrightarrow \mathbb{N} & \\
msg\_ts : ID \nrightarrow \mathrm{seq}\, \mathbb{N} & \\
errno : ERROR & \\
\hline
\forall id : (0 \mathrel{..} MSGMNI - 1) \bullet msgque(id) = DEFINED & (4) \\
\qquad \Leftrightarrow id \in \mathrm{dom}\, msg\_perm\_key & \\
\forall id : (0 \mathrel{..} MSGMNI - 1) \bullet msgque(id) = DEFINED & (5) \\
\qquad \Leftrightarrow id \in \mathrm{dom}\, msg\_perm\_seq & \\
\forall i : ID \bullet i \in \mathrm{dom}\, msgq \Leftrightarrow (msgque(i \bmod MSGMNI) = DEFINED \land & (6) \\
\quad msg\_perm\_seq(i \bmod MSGMNI) * MSGMNI + i \bmod MSGMNI = i) & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, creator & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, owner & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, group & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, perm & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, qbytes & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, cbytes & \\
\mathrm{dom}\, msgq = \mathrm{dom}\, msg\_ts & \\
\forall i : ID \mid i \in \mathrm{dom}\, msgq \bullet \mathrm{dom}(msgq(i)) = \mathrm{dom}(msg\_ts(i)) & \\
\forall i1, i2 : ID \mid i1 \in \mathrm{dom}\, msgq \land i2 \in \mathrm{dom}\, msgq \bullet & (7) \\
\quad msg\_perm\_key(i1 \bmod MSGMNI) = msg\_perm\_key(i2 \bmod MSGMNI) & \\
\qquad \Rightarrow i1 = i2 \lor msg\_perm\_key(i1 \bmod MSGMNI) = IPC\_PRIVATE & \\
\hline
\end{array}
$$

Hierbei haben wir gegenüber dem abstrakten Zustandraum folgendes konkretisiert:

(1) Mit $msgque$ wird für jede Zahl zwischen 0 und $MSGMNI - 1$ angegeben, ob ihr eine Message-Queue zugeordnet ist. Ansonsten nimmt sie einen speziellen Wert an.
Gleichzeitig wird hiermit ausgesagt, daß die Größe des Domain maximal die Größe von $MSGMNI$ annimmt. Somit ist die mit (1) bezeichnete Zeile aus dem abstrakten $MsgQStateSpace$ hier enthalten.

(2) Mit msg_perm_key wird einer Message-Queue ein eindeutiger KEY zugeordnet.

(3) Mit msg_perm_seq wird einer Message-Queue eine Zahl zugeordnet. In der Implementierung wird diese Zahl so gewählt, daß sie mit $MSGMNI$ multipliziert noch in eine vorzeichenbehaftete 32–Bit–Zahl paßt.

(4) Wenn eine Message-Queue vorhanden ist, dann ist auch ein KEY für sie vergeben. Und umgekehrt.

(5) Wenn eine Message-Queue vorhanden ist, dann ist ihr auch eine Zahl zugeordnet. Und umgekehrt.

(6) Diese Gleichung beschreibt, wie die ID aus msg_perm_seq berechnet wird. Dies stellt sicher, daß nicht versehentlich mit einer falschen ID auf die Message-Queue zugegriffen wird, weil beispielsweise die vorhandene Message-Queue seit dem letzten Zugriff gelöscht wurde und mittlerweile eine neue Message-Queue mit derselben ID erzeugt wurde.

(7) Entspricht der letzten Konsistenzbedingung aus dem abstrakten Zustandsraum, nur mit konkreteren Werten und Berechnungen.

Abstraktionsfunktion

Die Abstraktionsfunktion ist eine Refinement–Relation, die eine Gleichwertigkeit der abstrakten und der konkreten Message-Queues herstellen soll.

Da wir bisher nur versucht haben, $getkey$ zu konkretisieren, muß auch nur dieses in der Relation berücksichtigt werden.

$$
\begin{array}{l}
\textit{Rel} \\
\hline
\textit{MsgQStateSpace} \\
\textit{CMsgQStateSpace} \\
\hline
\forall\, i : (0 \,..\, MSGMNI - 1) \mid msgque(i) = DEFINED \bullet \\
\quad i \in \operatorname{dom} msg\_perm\_seq \wedge \\
\quad i \in \operatorname{dom} msg\_perm\_key \wedge \\
\quad (i + msg\_perm\_seq(i) * MSGMNI) \in \operatorname{dom} getkey \wedge \\
\quad msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI) \\
\forall\, i : ID \mid i \in \operatorname{dom} getkey \bullet \\
\quad msgque(i \bmod MSGMNI) = DEFINED \\
\hline
\end{array}
$$

Durch diese Refinement–Relation entsteht eine Verbindung zwischen der abstrakten und der konkreten Sicht. Insbesondere muß für jede Operation im Konkreten Zustandsraum gelten, daß sie auch für die korrospondierenden Zustände im abstrakten Zustandsraum möglich sein muß. Diese kann man anschaulich im Diagramm 34.1 sehen.

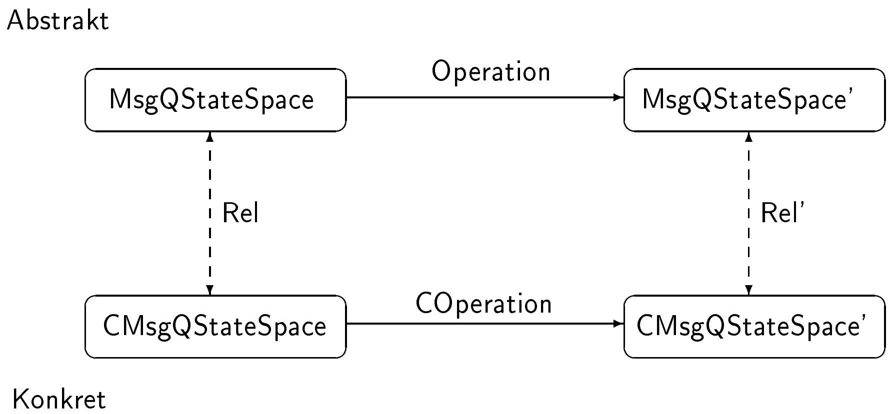


Abbildung 34.1: Beziehung zwischen abstraktem und konkretem Zustandsraum

Wenn die Vorbedingungen für ein abstraktes Schema einer Message-Queue-Operation vorliegen, müssen auch die Vorbedingungen für das konkrete Schema der Message-Queue-Operation vorliegen. Die

Nachbedingungen dieser beiden Schemata müssen durch die Relation übereinstimmen.

Da eine entsprechende Verbindung nun existieren muß, kann man dieses sehr gut für die Verifikation benutzen. Da wir ein Data-Refinement durchführen wollen, haben wir zu zeigen, daß die konkrete Sicht eine Verfeinerung der abstrakten Sicht ist. Somit haben wir in den folgenden Abschnitten zwei Beweisverpflichtungen.

Zuerst haben wir die *safety condition*, die Sicherheitsbedingung. Wenn die Vorbedingungen des abstrakten Schemas erfüllt sind und gleichzeitig die Relation vorausgesetzt wird, müssen auch die Vorbedingungen des konkreten Schemas erfüllt sein. Also immer wenn es ein abstraktes Schema gibt, muß es auch ein entsprechendes konkretes Schema geben.

Anschließend gibt es noch die *liveness condition*. Wenn die Vorbedingungen für ein abstraktes Schema vorliegen, die Relation und das konkrete Schema vorausgesetzt werden, dann muß sich hieraus ein abstrakter Zustandsraum folgern lassen, der dem abstrakten Schema und der Relation genügt. Oder anders gesagt: Betrachtet man obiges Bild, so ist der Weg von $MsgQStateSpace$ über Rel nach $CMsgQStateSpace$ und über $COperation$ nach $CMsgQStateSpace'$ gegeben. Aus diesem muß sich dann auch der andere Weg verfolgen lassen.

34.3.2 Systeminitialisierung

Nachdem nun der Zustandsraum für die Message-Queues festgelegt ist, müssen wir die Startbedingungen für das System vorgeben.

Beim Start des Betriebssystems gibt es noch keine Prozesse, diese müssen erst noch erzeugt werden. Daher existieren auch noch keine Message-Queues, diese werden erst später initialisiert.

Die abstrakte Systeminitialisierung

Dieser Teil befindet sich im Abschnitt 33.3.3 auf Seite 240.

Die konkrete Systeminitialisierung

Bei der konkreten Systeminitialisierung sind dieselben Bedingungen wie bei der abstrakten Systeminitialisierung vorhanden.

CInit

$CMsgQStateSpace'$

$$
\begin{array}{l}
\forall\, i : (0 \,..\, MSGMNI - 1) \bullet msgque'(i) = IPC\_UNUSED \\
\operatorname{dom} msg\_perm\_key' = \emptyset \\
\operatorname{dom} msg\_perm\_seq' = \emptyset \\
\operatorname{dom} msgq' = \emptyset \\
\operatorname{dom} creator' = \emptyset \\
\operatorname{dom} owner' = \emptyset \\
\operatorname{dom} group' = \emptyset \\
\operatorname{dom} perm' = \emptyset \\
\operatorname{dom} qbytes' = \emptyset \\
\operatorname{dom} cbytes' = \emptyset \\
\operatorname{dom} msg\_ts' = \emptyset \\
errno' = NOERROR
\end{array}
$$

Auch hier wird entsprechend der abstrakten Systeminitialisierung der Domain der benutzten Funktionen auf die leere Menge, bzw. den Defaultwert gesetzt. Dies geschieht, da noch keine Message-Queues im System existieren.

Refinement

An dieser Stelle beginnt nun die eigentliche Verifikation. Es muß gezeigt werden, daß die konkrete Systeminitialisierung eine Verfeinerung der abstrakten Systeminitialisierung ist.

Da es sich hier nicht um eine Message–Queue Operation handelt, haben wir nicht die weiter oben beschriebenen Beweisverpflichtungen. Stattdessen haben wir zwei andere Refinement-Bedingungen.

Erste Bedingung Als erstes muß gezeigt werden, daß es stets einen konkreten Systeminitialisierungszustand gibt und dieser konsistent ist. Dies bedeutet in Z:

$$\vdash \exists\, CMsgQStateSpace' \bullet CInit$$

Da alle Werte der Funktion $msgque$ auf IPC_UNUSED gesetzt wurden, und sie Domains der Funktionen $msg_p erm_k ey$ und $msg_p erm_s eq$ auf die leere Menge gesetzt wurden, Diese Funktionen also für alle Werte undefiniert sind, sind die Bedingungen (4) und (5) von CMsgQStateSpace erfüllt. Da die Domain vom msgq auf die leere Menge gesetzt wurde, ist auch Gleichungen (6) und (7) erfüllt. Da auch die Domains der restlichen Funktionen auf die leere Menge gesetzt wurden, sind auch die anderen anderen Bedingungen erfüllt. Somit haben wir einen Konsistenten Zustand.

Zweite Bedingung Es muß für jeden konkreten Initialisierungszustand gezeigt werden, daß er einem abstrakten Initialisierungszustand entspricht.

Zu zeigen ist also, daß gilt:

$$CInit \vdash (\exists\, MsgQStateSpace' \bullet (Init \wedge Rel'))$$

Für den Beweis sind nur die Prädikate relevant, die vom abstrakten Zustandsraum zum konkreten Zustandsraum verändert wurden.

Wir führen also alle relevanten Prädikate der Hypothese an:

(1)	$\forall\, i : (0 \mathrel{..} MSGMNI - 1) \bullet$ $msgque'(i) = IPC\_UNUSED$	aus $CInit$
(2)	$\mathrm{dom}\, msg\_perm\_key' = \emptyset$	aus $CInit$
(3)	$\mathrm{dom}\, msg\_perm\_seq' = \emptyset$	aus $CInit$

Da wir nur die Veränderungen vom abstrakten zum konkreten Zustandsraum betrachten, müssen wir jetzt zeigen, daß es ein $getkey'$ gibt, für das folgendes gilt:

$\exists\, getkey' \bullet$

(4) $(\mathrm{dom}\, msgq' = \mathrm{dom}\, getkey'$ aus $MsgQStateSpace'$

(5) $\wedge\ \mathrm{dom}\, getkey' = \emptyset$ aus $Init$

(6) $\wedge\ \forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus Rel'
$msgque'(i) = DEFINED \bullet$
$i \in \mathrm{dom}\, msg\_perm\_seq' \ \wedge$
$i \in \mathrm{dom}\, msg\_perm\_key' \ \wedge$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\mathrm{dom}\, getkey'$
$\wedge\ msg\_perm\_key'(i) = getkey'(i +$
$msg\_perm\_seq'(i) * MSGMNI))$

(7) $\forall\, i : ID \mid i \in \mathrm{dom}\, getkey' \bullet$ aus Rel'
$msgque'(i \bmod MSGMNI)$
$= DEFINED$

Behauptung: $getkey' = \emptyset$ erfüllt (4)–(7).

Mit $getkey' = \emptyset$ ist die Gleichung (4) äquivalent zu $\mathrm{dom}\, msgq' = \mathrm{dom}\ \emptyset = \emptyset$. Dies ist auch in $CInit$ so definiert.

Weiterhin ist damit die Gleichung (5) äquivalent zu $\mathrm{dom}\ \emptyset = \emptyset$, was trivialerweise eine wahre Aussage ist.

Gleichung (6) ist eine wahre Aussage, da es kein i gibt, so daß $msgque'(i) = DEFINED$, denn in $CInit$ wurde für alle $i \in \{1..MSGMNI - 1\}$ $msgque'(i) = IPC\_UNUSED$ gesetzt.

Gleichung (7) ist mit $getkey' = \emptyset$ eine wahre Aussage, da der Domain von $getkey'$ damit auch $\emptyset$ ist, und eine Allaussage über eine leere Menge immer wahr ist.

Somit ist die *liveness condition* erfüllt.

Ergebnis Da wir zeigen konnten, daß es stets einen konsistenten konkreten Systeminitialisierungszustand gibt und jeder konkrete Initialisierungszustand auch einem abstrakten Initialisierungszustand entspricht, haben wir die Systeminitialisierung verifizieren können.

34.3.3 Message-Queue Operation msgsnd()

Die Operation $msgsnd()$ wird von den Prozessen dazu benutzt, um in einer vorhandenen und ihnen bekannten Message-Queue Nachrichten zu senden. Eine Nachricht besteht dabei aus einem Typ und einer Anzahl von Bytes.

Es gibt eine Aufteilung in den erfolgreichen Fall des Sendens und in mehrere Fehlerfälle. Hierdurch ist eine bessere Lesbarkeit gegeben, und auch das Verständnis wird vereinfacht. Auch werden im Source-Code an diesen Stellen Fallunterscheidungen vorgenommen, und somit ist ein Aufsplitten quasi vorgegeben.

Für den Projektbericht beschränken wir uns auf den erfolgreichen Fall des Sendens. Alles weitere kann wie schon erwähnt, im Anhang eingesehen werden.

Im erfolgreichen Fall, der in diesem Kapitel behandelt wird, muß der Prozeß eine bestehende Message-Queue angeben, in die er seine Nachricht senden will und für die er natürlich Schreibrechte besitzen muß. Von der Nachricht selbst muß er den Typ und die Länge der Nachricht wissen.

Zuerst führen wir einen Hinweis auf das abstrakte Schema aus der formalen Spezifikation an. Anschließend erstellen wir aus dem Source-Code eine konkrete Spezifikation, die wir dann versuchen, per Data Refinement zu beweisen.

Abstraktes Senden einer Nachricht

In diesem Schema wird aus abstrakter Sicht das Senden einer Nachricht in einer Message-Queue spezifiziert. Es muß natürlich eine existierende Message-Queue adressiert sein und der Prozeß muß für diese Message-Queue auch Schreibrechte besitzen. Eine Nachricht besteht aus einem Typ und einer Anzahl von Bytes.

Dieses ist nachzulesen in Kapitel 33.3.4 auf Seite 242.

Konkretes Senden einer Nachricht

Und nun die Spezifikation des konkreten Sendens einer Nachricht in einer Message-Queue.

$$
\begin{array}{|l}
\hline
\textit{CMsgsnd} \\
\Delta \textit{CMsgQStateSpace} \\
t? : \textit{TYPE} \\
b? : \textit{BYTES} \\
\textit{msgsz}? : \mathbb{N} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
f? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{getuid}(p?) \ \mathrm{maywrite}\ (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?)) \\
\textit{cbytes}(i?) + \textit{msgsz}? \leq \textit{qbytes}(i?) \\
\textit{msgsz}? \leq \textit{MSGMAX} \\
\textit{msgsz}? \geq 0 \\
t? > 0 \\
r! = 0 \\
\\
\textit{msgque}' = \textit{msgque} \qquad (1) \\
\textit{msg\_perm\_key}' = \textit{msg\_perm\_key} \qquad (2) \\
\textit{msg\_perm\_seq}' = \textit{msg\_perm\_seq} \qquad (3) \\
\textit{msgq}' = \textit{msgq} \oplus \{i? \mapsto \textit{msgq}(i?) \frown \langle (t?, b?) \rangle\} \\
\textit{creator}' = \textit{creator} \\
\textit{owner}' = \textit{owner} \\
\textit{group}' = \textit{group} \\
\textit{perm}' = \textit{perm} \\
\textit{qbytes}' = \textit{qbytes} \\
\textit{cbytes}' = \textit{cbytes} \oplus \{i? \mapsto \textit{cbytes}(i?) + \textit{msgsz}?\} \\
\textit{msg\_ts}' = \textit{msg\_ts} \oplus \{i? \mapsto \textit{msg\_ts}(i?) \frown \langle \textit{msgsz}? \rangle\} \\
\textit{errno}' = \textit{errno} \\
\hline
\end{array}
$$

Das konkrete Schema ist unterschiedlich zum abstrakten Schema:

Anstelle eines unveränderten $getkey$ des Zustandraumes werden die drei konkreteren Objekte des Zustandraumes verwendet und ebenfalls nicht verändert (1 bis 3).

Refinement

Nun muß auch hier gezeigt werden, daß das konkrete Senden in einer Message-Queue eine Verfeinerung des abstrakten Sendens ist. Das Vorgehen entspricht hierbei der in Kapitel 34.3.1 beschriebenen Vorgehensweise.

Erste Bedingung (safety condition) Zu zeigen ist hier, daß, falls abstrakte Vorbedingungen erfüllt werden, auch im konkreten Fall die Vorbedingungen erfüllt werden.

$$(\text{pre } Msgsnd) \land Rel \vdash \text{pre } CMsgsnd$$

Die Vorbedingung zum abstrakten $Msgsnd$ und die Relation Rel sehen folgendermaßen aus. Da bei $Msgsnd$ der Zustandsraum eingebunden wird, haben wir auch die dortigen Konsistenzbedingungen zu beachten. Allerdings übernehmen wir nur die Konsistenzbedingungen, für die sich etwas ändert. Für die anderen ist nachher bei der Verifikation die Herleitung trivial, da sie unverändert auf beiden Seiten stehen.

(1)	$i? \in \mathrm{dom}\, msgq$	aus pre $Msgsnd$
(2)	$getuid(p?)$ maywrite $(perm(i?), owner(i?), group(i?))$	aus pre $Msgsnd$
(3)	$cbytes(i?) + msgsz? \leq qbytes(i?)$	aus pre $Msgsnd$
(4)	$msgsz? \leq MSGMAX$	aus pre $Msgsnd$
(5)	$msgsz? \geq 0$	aus pre $Msgsnd$
(6)	$t? > 0$	aus pre $Msgsnd$
(7)	$\# \mathrm{dom}\, getkey \leq MSGMNI$	aus pre $Msgsnd$
(8)	$\mathrm{dom}\, msgq = \mathrm{dom}\, getkey$	aus pre $Msgsnd$
(9)	$\forall\, i1, i2 : ID \mid i1 \in \mathrm{dom}\, msgq \wedge i2 \in \mathrm{dom}\, msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \vee getkey(i1) = IPC\_PRIVATE$	aus pre $Msgsnd$
(10)	$\forall\, i : (0 \mathinner{.\,.} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \mathrm{dom}\, msg\_perm\_seq \wedge i \in \mathrm{dom}\, msg\_perm\_key \wedge (i + msg\_perm\_seq(i) * MSGMNI) \in \mathrm{dom}\, getkey \wedge msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(11)	$\forall\, i : ID \mid i \in \mathrm{dom}\, getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$	aus Rel

Die Vorbedingungen zum konkreten $CMsgsnd$ sehen folgendermaßen aus:

(12)	$i? \in \mathrm{dom}\, msgq$	aus pre $CMsgsnd$
(13)	$getuid(p?)$ maywrite $(perm(i?), owner(i?), group(i?))$	aus pre $CMsgsnd$
(14)	$cbytes(i?) + msgsz? \leq qbytes(i?)$	aus pre $CMsgsnd$
(15)	$msgsz? \leq MSGMAX$	aus pre $CMsgsnd$
(16)	$msgsz? \geq 0$	aus pre $CMsgsnd$
(17)	$t? > 0$	aus pre $CMsgsnd$
(18)	$\forall\, id : (0 \mathinner{.\,.} MSGMNI - 1) \bullet msgque(id) = DEFINED \Leftrightarrow id \in \mathrm{dom}\, msg\_perm\_key$	aus pre $CMsgsnd$

(19) $\forall id : (0 \mathinner{..} MSGMNI - 1) \bullet$ aus pre $CMsgsnd$
$msgque(id) = DEFINED \Leftrightarrow$
$id \in \mathrm{dom}\, msg\_perm\_seq$

(20) $\forall i : ID \bullet i \in \mathrm{dom}\, msgq \Leftrightarrow$ aus pre $CMsgsnd$
$(msgque(i \,\mathsf{mod}\, MSGMNI) = DEFINED \wedge$
$msg\_perm\_seq(i \;\mathsf{mod}\; MSGMNI) *$
$MSGMNI + i \;\mathsf{mod}\; MSGMNI = i)$

(21) $\forall i1, i2 : ID \mid$ aus pre $CMsgsnd$
$i1 \in \mathrm{dom}\, msgq \wedge i2 \in \mathrm{dom}\, msgq \bullet$
$msg\_perm\_key(i1 \;\mathsf{mod}\; MSGMNI) =$
$msg\_perm\_key(i2 \;\mathsf{mod}\; MSGMNI)$
$\Rightarrow i1 = i2 \vee$
$msg\_perm\_key(i1 \;\mathsf{mod}\; MSGMNI) =$
$IPC\_PRIVATE$

Wie nun leicht zu sehen ist, ergeben sich einige Folgerungen von allein:

(12) folgt aus (1)
(13) folgt aus (2)
(14) folgt aus (3)
(15) folgt aus (4)
(16) folgt aus (5)
(17) folgt aus (6)

Ähnlich einfach läßt sich direkt aus der Relation Rel folgern:

(18) folgt aus (10)
(19) folgt aus (10)

Etwas komplizierter wird es mit den restlichen Vorbedingungen. Der Beweis für (20):

Aus (10) können wir ersehen, daß für ein

$j \in \{0..MSGMNI - 1\}$ und $msgque(j) = DEFINED$

gilt:

$$j + msg\_perm\_seq(j) * MSGMNI \in \mathrm{dom}\, getkey$$

Sei $k = j + msg\_perm\_seq(j) * MSGMNI$
Nehmen wir diese Gleichung modulo $MSGMNI$, so erhalten wir:

$$\begin{aligned} k \;\mathsf{mod}\; MSGMNI &= (j + msg\_perm\_seq(j) * MSGMNI) \;\mathsf{mod}\; MSGMNI \\ &= (j \;\mathsf{mod}\; MSGMNI + (msg\_perm\_seq(j) * MSGMNI \\ &\qquad \mathsf{mod} MSGMNI)) \;\mathsf{mod}\; MSGMNI \\ &= (j \;\mathsf{mod}\; MSGMNI + 0) \;\mathsf{mod}\; MSGMNI \\ &= j \end{aligned}$$

Letzteres gilt, da $j \in \{0..MSGMNI - 1\}$

Somit kann man folgende Gleichung ableiten:

$$msgque(i \bmod MSGMNI) = DEFINED \Rightarrow$$
$$msg\_perm\_seq(i \bmod MSGMNI) * MSGMNI + i \bmod MSGMNI = i$$

Nun ist für den Beweis von (20) zu zeigen, daß die Äquivalenz zwischen $i \in \mathrm{dom}\, msgq$ und $msgque(i \bmod MSGMNI) = DEFINED$ gilt.

Für die eine Richtung kann man dies aus (11) und (8) folgern.

Für die andere Richtung kann man aus (10) folgern, daß, wenn für ein $j \in \mathrm{dom}\, msgque = DEFINED$ ist, $j + msg\_perm\_seq(j) * MSGMNI) \in \mathrm{dom}\, getkey$ gilt. Wenn wir jetzt $i = j + msg\_perm\_seq(j) * MSGMNI)$ setzen, sind beide Gleichungen auf der rechten Seite von (20) erfüllt, und mit (8) kann man daraus dann die linke Seite folgern.

Dagegen ist der Beweis für (21) wieder überschaubar:
Mit (9) und der oben erstellten Gleichung, daß $k \bmod MSGMNI = j$ ist, kann man folgern, daß $getkey(i) = msg\_perm\_key(i \bmod MSGMNI$ gilt. Damit ergibt sich dann (21).

Zweite Bedingung (liveness condition) Wir müssen zeigen, daß es bei jedem konkreten $CMsgsnd$ einen abstrakten $MsgQStateSpace$ gibt, so daß ein konkretes $Msgsnd$ möglich und die Relation Rel erfüllt ist.

$$(\mathrm{pre}\ Msgsnd) \land Rel \land CMsgsnd \vdash \exists\, MsgQStateSpace' \bullet (Msgsnd \land Rel')$$

Zuerst führen wir alle Prädikate der Hypothese an, und wieder nur die relevanten Teile aus dem eingebundenen Zustandsraum:

(1)	$i? \in \mathrm{dom}\, msgq$	aus $\mathrm{pre}\ Msgsnd$
(2)	$getuid(p?)\ \mathrm{maywrite}$ $(perm(i?), owner(i?), group(i?))$	aus $\mathrm{pre}\ Msgsnd$
(3)	$cbytes(i?) + msgsz? \leq qbytes(i?)$	aus $\mathrm{pre}\ Msgsnd$
(4)	$msgsz? \leq MSGMAX$	aus $\mathrm{pre}\ Msgsnd$
(5)	$msgsz? \geq 0$	aus $\mathrm{pre}\ Msgsnd$
(6)	$t? > 0$	aus $\mathrm{pre}\ Msgsnd$
(7)	$\#\, \mathrm{dom}\, getkey \leq MSGMNI$	aus $\mathrm{pre}\ Msgsnd$
(8)	$\mathrm{dom}\, msgq = \mathrm{dom}\, getkey$	aus $\mathrm{pre}\ Msgsnd$
(9)	$\forall\, i1, i2 : ID \mid$ $i1 \in \mathrm{dom}\, msgq \land i2 \in \mathrm{dom}\, msgq \bullet$ $getkey(i1) = getkey(i2) \Rightarrow$ $i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$	aus $\mathrm{pre}\ Msgsnd$

(10) $\forall i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus Rel
$msgque(i) = DEFINED \bullet$
$i \in \mathrm{dom}\, msg\_perm\_seq \land$
$i \in \mathrm{dom}\, msg\_perm\_key \land$
$(i + msg\_perm\_seq(i) * MSGMNI) \in$
$\mathrm{dom}\, getkey \land$
$msg\_perm\_key(i) = getkey(i +$
$msg\_perm\_seq(i) * MSGMNI)$

(11) $\forall i : ID \mid i \in \mathrm{dom}\, getkey \bullet$ aus Rel
$msgque(i \mathbin{\mathsf{mod}} MSGMNI) = DEFINED$

(12) $msgque' = msgque$ aus $CMsgsnd$

(13) $msg\_perm\_key' = msg\_perm\_key$ aus $CMsgsnd$

(14) $msg\_perm\_seq' = msg\_perm\_seq$ aus $CMsgsnd$

Nun expandieren wir die Folgerung, die eine Existenz-Quantifizierung ist, und beschränken uns dabei auf einen kleinen Teil, der nur beim konkreten Part verändert wurde.

$\exists getkey' : ID \pfun KEY \bullet$

(15) $(getkey' = getkey$ aus $Msgsnd$

(16) $\land \forall i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus Rel'
$msgque'(i) = DEFINED \bullet$
$i \in \mathrm{dom}\, msg\_perm\_seq' \land$
$i \in \mathrm{dom}\, msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\mathrm{dom}\, getkey' \land$
$msg\_perm\_key'(i) = getkey'(i +$
$msg\_perm\_seq'(i) * MSGMNI))$

(17) $\forall i : ID \mid i \in \mathrm{dom}\, getkey' \bullet$ aus Rel'
$msgque'(i \mathbin{\mathsf{mod}} MSGMNI) = DEFINED$

Mit (15) finden wir ein geeignetes $getkey'$, und setzen es in (16) und (17) ein.

(18) $\land \forall i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus (15) und (16)
$msgque'(i) = DEFINED \bullet$
$i \in \mathrm{dom}\, msg\_perm\_seq' \land$
$i \in \mathrm{dom}\, msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\mathrm{dom}\, getkey \land$
$msg\_perm\_key'(i) = getkey(i +$
$msg\_perm\_seq'(i) * MSGMNI))$

(19) $\forall i : ID \mid i \in \mathrm{dom}\, getkey \bullet$ aus (15) und (17)
$msgque'(i \mathbin{\mathsf{mod}} MSGMNI) = DEFINED$

Wie nun offensichtlich, ist (18) herzuleiten aus (10), (12), (13) und (14). Genauso offensichtlich ist (19) herzuleiten aus (11), (12), (13) und (14).

Ergebnis Wir haben sowohl die *safety condition* als auch die *liveness condition* bewiesen. Demnach ist die konkrete Spezifikation eine Verfeinerung der abstrakten Spezifikation von $Msgsnd$.

34.4 Probleme bei der Verifikation

Da wir die Spezifikationssprache Z inzwischen wesentlich besser verstehen und uns bei der Spezifikation in den Source-Code hineingearbeitet haben, gab es bei der Verifikation keinerlei Probleme mit dem Verständnis, wie noch bei der Spezifikation. Trotzdem erwies sie sich als äußerst langwierig und nicht sehr einfach zu beweisen.

Leider gibt es für diese Art des Beweisens noch keine Toolunterstützung, die einen Teil der Arbeit hätte ersparen können.

Entsprechend ist es uns zeitlich nicht mehr möglich, die Message-Queues vollständig zu verifizieren.

Insgesamt erscheint uns dieser sehr hohe Aufwand für eine Verifikation nur für sicherheitskritische Bereiche sinnvoll zu sein.

35. Was noch zu tun ist

Wie in den vorherigen Abschnitten schon beschrieben, reichte unsere Zeit leider nicht aus, um die Message-Queues vollständig zu verifizieren. Dies war uns nur für eine Message-Queue Operation möglich.

Demnach wäre eine mögliche Aufgabe für die Zukunft, noch die restlichen drei Message-Queue Operationen zu verifizieren. Hierfür haben wir mit der erstellten Spezifikation schon eine gute Grundlage geschaffen.

Es wäre sicherlich eine nützliche Sache, wenn man für diese Art der Verifikation eine gewisse Toolunterstützung hätte und somit die Beweise automatisieren könnte. Ob und wie dieses möglich ist, ist uns noch nicht ganz klar, jedoch wäre es sicher eine interessante und lohnenswerte Forschungsaufgabe.

36. Ergebnis

Zuerst mal die ganz persönlichen Ergebnisse.

Inzwischen kennen wir uns mit den Message-Queues und den verschiedenen Message-Queue Operationen, insbesondere $msgsnd()$, sehr gut aus.

Wir haben eine Menge über Spezifikation gelernt, insbesondere über die Spezifikationssprache Z, und eine umfangreiche Spezifikation durchgeführt.

Dasselbe gilt für die Verifikation, die wir intensiv kennengelernt haben, indem wir selbst ein Data Refinement durchgeführt haben.

Aber auch für die Allgemeinheit und speziell die Linux-Gemeinschaft haben wir einige Resultate vorzuweisen.

Wir haben eine formale Spezifikation der Message-Queues durchgeführt, auf die man sich bei späteren Unklarheiten immer berufen kann.

Und schließlich haben wir noch einen Teil der Message-Queues verifiziert. Dabei haben wir, für uns leider etwas unbefriedigend, für die Allgemeinheit glücklicherweise, keinen Fehler mehr gefunden.

Das heißt, man kann nach unseren Erkenntnissen die Message-Queues mit der Gewissheit des richtigen Funktionierens benutzen.

Teil VI

SMP

Vorspann

Das Teilprojekt SMP wird von Klaas-Henning Zweck, Harald Wagener, Mark Hapke und Ewgeni Gisbrecht gebildet.

Unser Ziel ist es, die Unterstützung von Mehrprozessorsystemen zu optimieren. Da wir nicht der Auffassung sind, daß die bereits vorhandenen Mechanismen ausgetauscht werden sollen, haben wir unseren Schwerpunkt auf die besondere Behandlung von I/O–Systemaufrufen gelegt.

37. Design–Optionen

37.1 Wahl des Kernels

Zu Beginn dieses Projektteils fiel relativ schnell und eindeutig die Entscheidung, unsere Modifikationen am Standard–Kernel auf der i386–Architektur vorzunehmen. Dafür sprechen mehrere Gründe:

- Die SMP–Unterstützung ist hier am weitesten fortgeschritten.
- Die Hardware ist am billigsten.
- Wir wissen, wo man die Informationen herbekommt.
- Wir wollen nicht auch noch SPARC/Amiga/PPC-Assembler lernen.

37.2 Einschränkung des Arbeitsbereichs

Alle Kernelfunktionen, die mit symmetrischen Multiprocessing zusammenhängen zu verifizieren wäre ein Unterfangen, das den Rahmen des Projektes bei weitem sprengen würde.

Deswegen haben wir uns dafür entschieden, uns auf einen relativ speziellen Bereich zu beschränken, der Übersichtlich genug erscheint, um ihn im gegebenen Rahmen fertigzustellen und zusätzlich dazu dem Standardkernel hinzugefügt werden kann, ohne andere Funktionalität zu verändern oder zu verlieren.

38. Parvo in Multum – Kleine Helfer in der Kommunikation

Unser **PiM**–Scheduler–AddOn soll die Kommunikation zwischen Prozessen und I/O–Schnittstellen über im Scheduler besonders behandelte Systemaufrufe in Hinsicht auf die Möglichkeiten mit Mehrprozessorsystemen optimieren.

Unsere Implementierung sieht vor, für jede unterstützte I/O–Schnittstelle einen besonderen Serverprozeß zu starten, der die I/O–Aufrufe an Clients weitergibt, die die tatsächliche Verarbeitung der Aufrufe erledigen.

Zunächst müssen wir in der *IDT* um einen Eintrag erweitern, damit feststellbar ist, ob der jeweilige Systemaufruf vom **PiM**–Scheduler–AddOn unterstützt wird. Ist dies der Fall, wird der I/O–Systemaufruf nicht, wie sonst, an ein normales `dev_write()` übergeben, sondern an `pim_write()`.

Die **PiM**–Prozesse haben sowohl für das Scheduling des Kernels als auch für die Berechtigung, auf Schnittstellen zuzugreifen, eine besondere Priorität. Ihnen wird das *Code Selector Segment* **1** zugewiesen, damit sie die Schnittstellen ansprechen dürfen; im Kernel-Scheduler bekommen die Server–Prozesse eine etwas höhere Priorität als alle anderen Prozesse haben können, die **PiM**–Clients haben eine Priorität etwas unter dem Maximum.

Eine weitere Abweichung ist, daß es für jeden Prozessor einen Server pro Schnittstelle und eine bestimmte Anzahl Clients pro Server gibt. Diese sollen nicht wie andere Prozesse beliebig die CPU wechseln, sondern fortwährend einer CPU zugewiesen bleiben, damit die Systemaufrufe keinen CPU–Switch verursachen.

Weiterhin müssen wir den Scheduler für um zwei Warteschlangen für **PiM**–Server und –Clients erweitern.

39. Einleitung

39.1 Was ist SMP?

Die heutige Form des Arbeitens in der EDV stellt sich im Allgemeinen so dar, daß leistungsfähige Server mit im Vergleich zu diesen, relativ leistungsarmen Arbeitsplatzrechnern vernetzt sind. Hauptanwendungen wie etwa Datenbanken oder rechenintensive Prozesse wie solche zur Echtzeitverarbeitung, z.B. Echtzeitverschlüsselung, laufen auf dem Server und der Arbeitsplatzrechner steht Rede und Antwort.

Server mit nur einer CPU des heutigen Stands der Technik wären zumeist nicht in der Lage die anfallenden Aufgaben in angemessener Zeit zu Bearbeiten. Damit sie dies können hat sich die Idee der Paralellverarbeitung auf Hardwareseite etabliert. Moderne Betriebssysteme wie Linux bieten zwar kernelseitige Multiuser- und Multitaskingfähigkeiten, bilden diese aber auf nur eine CPU ab.

Bei hardwaregestützter Parallelarchitektur werden Prozesse entweder auf mehrere Rechner, wie bei Clusterarchitekturen üblich, oder auf mehrere CPUs in einem Rechner, wie bei SMP-Systemen, verteilt. SMP ist ein Akronym und steht für ***S*ymmetric *M*ulti *P*rocessing**. So werden eben solche hardwareseitigen Parallelsysteme bezeichnet, in denen sich mehrere CPUs die restlichen Ressourcen des Rechners teilen.

Aus dieser Situation entsteht sowohl der Hauptvorteil von SMP Systemen, als auch deren Hauptnachteil gegenüber anderen Parallelarchitekturen. Der Vorteil besteht in der Preisgünstigkeit, da Kosten nur bei der Hauptplatine und je zusätzlicher CPU anfallen, der Nachteil entsteht jedoch genau hieraus, da die nun parallelen Prozesse sich nicht jede im Rechner vorhandene Ressource exakt zur gleichen Zeit teilen können. In solchen Situationen können SMP Systeme das theoretische Optimum von einer hunderprozentigen Parallelität nicht gewährleisten. SMP Systeme und Clusterarchitekturen sind gleichermaßen stark von der vom Betriebssystem gewährleisteten Unterstützung paralleler Archtekturen abhängig.
SMP Systeme sind jedoch, was z.B. Gesamtzahl der CPUs angeht, im Gegensatz zu Clusterarchitekturen sehr stark an die Vorgaben des jeweiligen Prozessorherstellers gebunden.

Das kommt daher, daß SMP-Systeme genau solche sind, bei denen mehrere CPUs über einen vom CPU Hersteller spezifizierten Systembus auf eine gemeinsame Harware zugreifen und für die Konsistenz des Systems entscheidend ist, wie sie das tun. Das *wie* wird in der Regel durch den CPU-Hersteller vorgegeben.

Wir haben uns aus Kostengründen und des großen Verbreitungsgrades für ein System mit zwei Intel Pentium II Prozessoren entschieden. Somit ist für den Entwurf und die Implementierung von SMP-Optimierungen im Linux Kern die Intel SMP-Spezifikation die maßgebende Grundlage.Die Intel SMP Spezifikation liegt Mittlerweile in der Version 1.4 vor und ist auf der CD enthalten.

40. SMP bei Intel

40.1 Die Hardwareseite eines SMP Systems

Ein-Prozessor-Systeme verschiedener Hardwarehersteller unterscheiden sich aus Programmierersicht im Prinzip nur durch die unterschiedliche Anzahl und Benamung ihrer Register, sowie ihrer Maschinensprache bzw. ihres Microcodes. Ansonsten sind sie 'die guten alten' von Neumann Eingabe-Verarbeitung-Ausgabe[1] Rechner [2].

Bei SMP-Systemen treten nun folgende Fragen auf, die stark hardwareabhängig sind:

- Wie ändert sich der Bootvorgang/Shutdownvorgang im Gegensatz zum Einprozessorsystem ?
- Wie groß ist die maximale Anzahl der CPUs ?
- Können die CPUs gleichzeitig Betriebssystemcode ausführen ?
- Können die CPUs gleichzeitig die selbe Speicherstelle ansprechen ?
- Wie wird auf Hardware-Interrupts reagiert ?
- Wie wird der Cache behandelt ?

In der obigen Liste wird mehrmals der Begriff *gleichzeitig* verwendet. Obwohl die Frage, wie ein gleichzeitiger Zugriff mehrerer CPUs auf ein und dieselbe (physikalische) Speicherstelle für ein konsistentes System von Bedeutung scheint, ist der Begriff der Gleichzeitigkeit mit Vorsicht zu geniessen.

Da CPUs wie diese Massenware sind und selbst CPUs mit demselben Takt immer marginale Abweichungen zeigen werden, ist es allein aus diesem Grund schon unwahrscheinlich, daß Gleichzeitigkeit erreicht wird.

[1] engl. fetch and execute
[2] Siehe auch: http://www.tu-clausthal.de/r̃zppk/wscript/node4.html

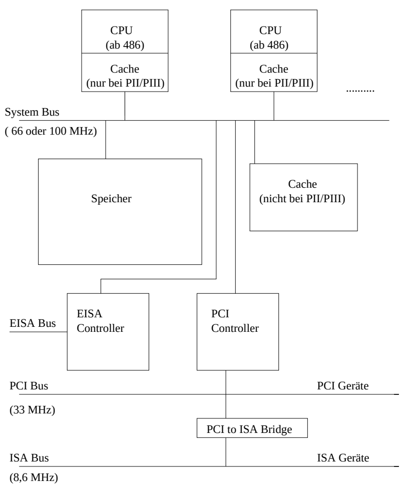


Abbildung 40.1: Aufbau eines SMP-Systems (PC)

Desweiteren sind diese CPUs mit sogenannten Pipelines ausgestattet, so daß *Fetch-And-Execute* im ursprünglichen Sinne nicht mehr anwendbar ist. Gleichzeitigkeit mit Pipelines setzt identischen Code, zumindest in Länge der Pipeline, in den CPUs voraus. Ansonsten könnte es zu Pipeline-Auffrischungen kommen, oder zu Blockierung einer der beiden Pipelines im Pentium II. Das Resultat wäre eine Verschiebung der Ausführung in den CPUs um mindestens Pipelinelänge Takte.

Alles in allem ist es also logisch, daß absolute Gleichzeitigkeit mehrerer CPUs bei Pentium II Systemen nur durch Warten, also aktive Synchronisierung erzwungen werden kann und dann auch nur nach einem ausgeführten Befehl gilt und nicht während dessen.

40.1.1 Hardwareseitige Schutztmechanismen

Da jedoch nicht absolut auszuschließen ist, daß zwei Programme die absolut nicht gleichzeitig laufen sollen, mit einem Zugriff auf eine physikalische Speicherstelle beginnen, existieren bei Intel-Prozessoren drei unabhängige Hauptverfahren, die die Systemkorrektheit gewährleisten sollen:

- Garantiert atomare Operationen
- Befehle zum Sperren des Systembusses
- Ein Protokoll zur Cache-Konsistenz

Die atomaren Operationen sind im wesentlichen *read*- und *write*-Befehle, die im Speicher auf *byte* bis *doubleword* operieren. Dazu kommen die Befehle zum Sperren des Systembusses sind ein explizites *LOCK* als Befehl, oder ein von der CPU ausgelöstes, implizietes *LOCK* bei bestimmten Operationen.

Für das Caching wird abstrahiert vom Caching-Verfahren[3], daß das *M.E.S.I.* Protokoll[4] verwendet, um systemweite Cachekonsistenz zu gewährleisten. Zusätzlich existieren noch weitere Verfahren, die sich um spezielle Eigenschaften des Systems im Multiprozessorbetrieb kümmern.

Dazu gehören :

- Programmierbare Interruptcontroller auf dem Chip
- Methoden zum delegieren von Interrupts an bestimmte CPUs
- Möglichkeiten zur Interprozessorkommunikation

Da die Pentium II und Pentium III CPUs den 1st- und 2nd-level-Cache jeweils auf dem Chip und nicht gemeinsam auf dem Motherboard haben sind inkonsistente Caches bei diesen CPUs unwahrscheinlicher geworden. Ein entscheidender Faktor im Betrieb eines Multiprozessorsystems ist die Frage der Software- und Hardware-Interruptbehandlung.

Bei Intel läßt sich diese schematisch so darstellen:

Der lokale APIC [5] einer CPU kann Interrupts von lokalen Geräten, vom I/O APIC, Softwareinterrupts und Interrupts vom lokalen APIC einer

[3] Write Through, Write Back

[4] *M*odified, *E*xclusiv, *S*hared, *I*nvalid

[5] *A*dvanced-*P*rogrammable-*I*nterrupt-*C*ontroller

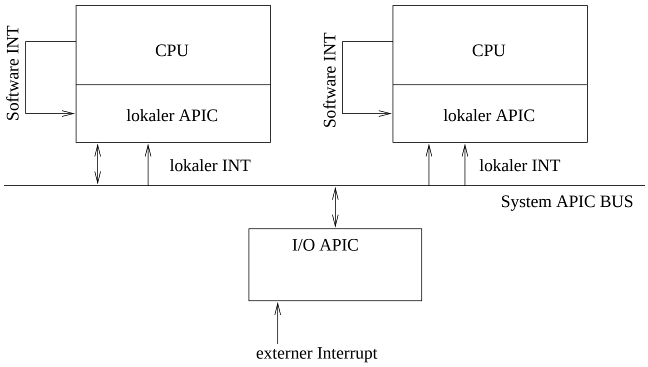


Abbildung 40.2: Intel SMP Interrupt Handling

anderen CPU, als sogenannter Inter-Prozessor-Interrupt, empfangen. Traditionell kann der externe IO-APIC 15 Interrupts versenden. Der Interrupt Nummer 2 ist durch die Kaskadierung zweier 8-Bit Interrupt-Controller schon vergeben. Es sind insgesamt 240 weiter Interrupts belegbar (16..255), wobei deren Priorität durch: Interruptnummer Modulo 16 ermittelt wird und somit immer im klassischen Rahmen von 0..15 bleibt[6].

Programmierbare Interruptcontroller und die Möglichkeit zur Inter-Prozessor-Kommunikation über ein Protokoll sollen es dem Betriebssystem und dem Systemprogrammierer ermöglichen, eine optimale Lastverteilung zwischen den CPUs zu erreichen.

[6] 0 hat die höchste Priorität, 15 die geringste

41. SMP bei Linux

41.1 Die Softwareseite eines Linux-SMP-Systems

Bei Linux begann die Möglichkeit, SMP-Systeme zu betreiben bereits ab Kernel 1.3.x. Im Kernel 1.3.68 waren bereits die Kernel-Dateien smp.c, smp.h vorhanden und main.c auf das mögliche Booten eines SMP-Systems abgeändert.

Die SMP Unterstützung war einer der sich am meisten weiterentwickelnden Bereiche der letzten veröffentlichten Linux-Kernel Versionen. In den Kerneln der 2.0.x-Serie, die beim Start unseres Projektes aktuell waren, war ein Hauptproblem von Linux-SMP noch die hohe Anzahl von Systemaufrufen, die den gesamten Kernel blockierten. Der Grund dafür liegt nicht in den Anwendungen, sondern im Linux Kernel selbst.

Führt eine Anwendung Systemaufrufe aus, ist die Architektur des Betriebssystems ausschlaggebend für die Performance der Anwendungen. Moderne Betriebssysteme wie Linux verwenden ein mehrstufiges Prioritätensystem, mit dem sie Anwendungen und Betriebssytem in verschiedene Schichten einordnen und so die Befugnisse auf die Hardware beschränken.

Linux verwendet von den vier Schichten, die die Intel-CPU hardwareseitig unterstützt, effektiv nur zwei, *NULL* für das Betriebssystem (die höchste Priorität) und *VIER* für Anwendungen (die niedrigste Priorität). Kernelmodule werden ebenfalls in die Schicht *NULL* eingeordnet.

Anwendungen einer Schicht können nur Code ihrer eigenen oder einer niedriger priorisierten Schicht ausführen. Jeder direkte Aufruf einer Systemfunktion oder das Laden einer Speicheradresse des Betriebssystems führt über die Memory-Management-Unit(MMU) zu einem *Pagefault*, und landet sofort in einer entsprechenden Routine des Betriebssystems.

Im Gegensatz zum Aufruf einer anwendungseigenen Funktion kann ein Programm also nur über einen, vom Betriebssystem klar definierten Weg Funktionen des Linux-Kerns aufrufen.

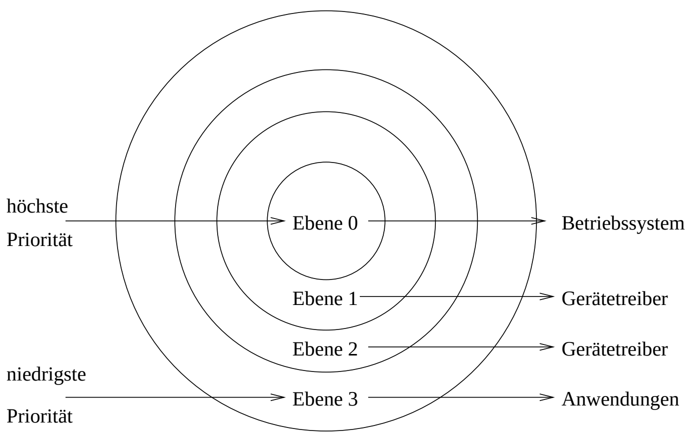


Abbildung 41.1: Ringmodell der CPU-Prioritäten

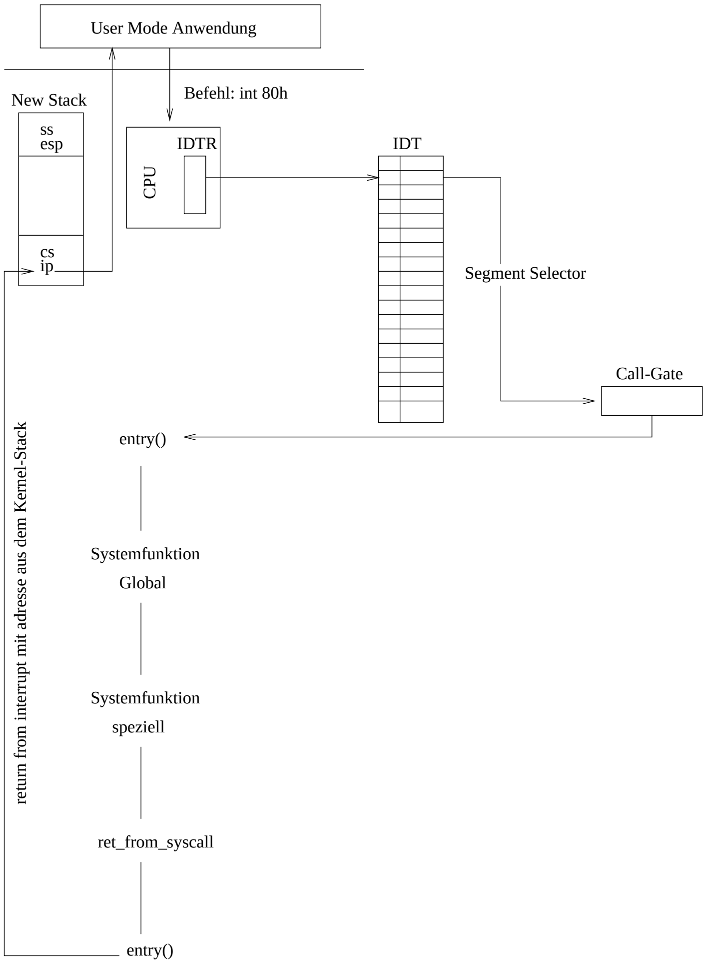


Abbildung 41.2: Schematischer Ablauf eines Systemaufrufs unter Linux

Ein Systemaufruf bei Linux sieht schematisch wie folgt aus:

Die Linux-SMP Unterstützung in den 2.0.x Kerneln war so implementiert, daß eine sehr hohe Anzahl von Systemaufrufen den gesamten Linux-Kern bis zum Verlassen des Systemaufrufs blockiert. Dies war insbesondere deshalb sinnvoll, da das Augenmerk der Entwickler zunächst auf der Unterstützung der gängigen PC-Hardware [1] lag. Somit wurde sichergestellt, daß ein überschneidendes Aufrufen der Systemfunktionen von verschiedenen CPUs aus und ein eventuelles Auftreten von Problemen mit konkurrierenden Zugriffen [2] vermieden werden konnte.

Die Art der Problemvermeidung war jedoch gleichzeitig die mit dem größten Geschwindigkeitsverlust behaftete.
Somit war die Ausrichtung der Linux-SMP-Entwickler auf eine reentrante Implementierung der Systemaufrufe das nächste Ziel, nachdem die gängigste Hardware unterstützt wurde (Kernel 1.3.x - 2.1.x). Seit der Linux-Kernel-Version 2.2.x hat das *SMP*-Projekt den Status des Experimentellen verlassen und wird zumindest für die Intel x86 Plattform als stabiler Bestandteil bezeichnet. Die blockierenden Systemaufrufe sind in aktuellen Linux-Kernen der Versionen 2.3.x oder sogar 2.4.x (aktueller Hackerkernel) fast völlig verschwunden.

[1] diverse CPU Versionen, Motherboards ..
[2] dead-lock, life-lock, race-conditions ...

Die weiteren Beschreibungen der Linux-SMP Unterstützung und Eigenschaften beziehen sich, wenn nicht anders erwähnt, ausschliesslich auf die Intel x86 Plattform.

41.1.1 Allgemeines zur Linux-SMP-Nutzung

Um SMP unter Linux nutzen zu können, sollte man einen Kernel der Version 2.0.X oder höher zur Verfügung haben. Der Kernel muß dann nur noch mit aktivierter SMP-Unterstützung und Wahl des korrekten Prozessortyps neuübersetzt und gebootet werden. Mit LILO [3] ist es sehr komfortabel einen testweise erstellten SMP-Kernel zu booten.

41.1.2 SMP Bootvorgang

Beim Booten des System mittels LILO ist es nicht ganz so deutlich, daß PC's auch heute noch im sog. Real-Mode der CPU booten. In diesem Modus gibt es keine hardwareseitigen Schutzmechanismen, und nur 1 MB adressierbaren Speicher. Daher sorgt auch im Multiprozessorsystem zunächst nur eine CPU für das Ausführen der initialisierenden Bootsequenz. Diese besteht im Wesentlichen aus dem Laden des Bootsektors[4] und Ausführen des in ihm enthaltenen Codes. Dadurch wird der Kernel in den Speicher geladen und die hineinkompilierten Funktionen aus : */usr/src/linux/arch/i386/boot/* aufgerufen. Diese sind für das Umschalten in den Protected-Mode, der alle im Real-Mode fehlenden Vorzüge für ein modernes Betriebssystem, wie Linux, bietet, und die notwendigen vorherigen Initialisierungen zuständig.

Es werden die GDT, IDT[5] angelegt und von der Boot-CPU [6] die anderen CPUs initialisiert. Das Initialisieren der weiteren CPUs besteht im wesentlichen aus dem aktualisieren der Register auf den Stand der Boot-CPU. Danach wird, wie auch im Ein-Prozessor-System, der Init-Prozess gestartet. Die eigentliche *main()*-Funktion heißt bei Linux *start_kernel()* und befindet sich in: */usr/src/linux/init/main.c* . Der weitere Ablauf richtet sich nach den in */etc/* enthaltenen Dateien zur Systemkonfiguration.

Das Verhalten zur Laufzeit Wie oben erwähnt hat man im Prinzip vollständiges Multiprocessing unter Linux, kann also davon ausgehen, daß Prozesse über alle im System befindlichen CPUs verteilt werden.
Dies ist in der Realität leider *noch* nicht ganz so. Auf Intel basierten Plattformen ist die Anzahl der unterstützten CPUs auf 32[7] beschränkt, was allerdings ob der überwältigenden Mehrheit von Dual-Prozessor-Systemen im PC-SMP Bereich kein wirklicher Nachteil ist.

41.1.3 SMP Scheduling

Das Scheduling ist unter Linux nicht an SMP angepaßt worden. Der Scheduler ist im Ein- und Mehrprozessorbetrieb bei den zur Verfügung ste-

[3] Der Linux Loader, näheres unter: http://www.linuxdoc.org/HOWTO/mini/LILO.html
[4] Dies ist der von LILO angegebene Bootsektor, muss nicht der Master-Boot-Sektor sein
[5] Globale-Descriptor-Tabelle, Interrupt-Descriptor-Tabelle... Näheres in den Intel Beschreibungen von Pentium, Pentium II und Pentium Pro.
[6] Anm.: Die CPUs werden bei der Linux Bootmessage von 0 an gezählt
[7] Seit den Kerneln der Version 2.2.x oder höher

henden Prioritäten und verwendeten Algorithmen identisch.
Einige Anpassungen mussten natürlich stattfinden. Die Implemtierung des Linux-Schedulers ist in der Datei *sched.c* zu finden. So wird, im Unterschied zum Ein-Prozessor-System, in der Funktion *static inline void reschedule_idle_slow(struct task_struct * p)* ein Prozess nach einem *wake_up* auf eine CPU verteilt, die sich im *idle*-Zustand befindet. Dies beinhaltet die mögliche Ausführung eines Prozesses auf einer anderen CPU als bisher, wenn er aufgeweckt wird.
Hier lag auch ein Ansatz des *Pim*-Systems zur möglichen Verbesserung innerhalb der bei unserem System involvierten Prozesse (Siehe Ideen zur Verbesserung des *Pim*-Systems).

Das eigentliche Scheduling wird in der Funktion *asmlinkage void schedule(void)* durchgeführt. Hierbei wird zunächst der aktuelle Task (*current*) zum vorherigen, den der vor dem schedulen dran war, gemacht und dann die CPU des neuen Task anhand der CPU des *current*-Task festgelegt.

```
asmlinkage void schedule(void){

    /* SNIP */
    prev = current;
    this_cpu = prev->processor;

    /* SNIP */
#ifdef __SMP__
    next->has_cpu = 1;
    next->processor = this_cpu;
#endif
    /* SNIP */
}
```

Eine weitere Anpassung an SMP im Linux-Scheduler ist die Verwaltung der durchschnittlich verbrauchten Zeiteinheiten, die pro Prozess gezählt werden. Dies wird im Ein-Prozessor-Linux gar nicht gemacht.

41.1.4 Threads

Es gibt unter Linux keine direkt vom Kernel unterstützten Möglichkeiten zur Thread-Programmierung. Hierzu ist es notwendig, die Linux-Thread-Library [8] zu verwenden. Diese POSIX 1003.1c kompatibele Bibliothek läuft jedoch selbst als User-Level-Prozess, wenn sie zu einem Programm hinzugelinkt wird.
Hieraus entsteht der Nachteil, daß alle Threads innnerhalb der Thread-Library selbst gescheduled werden ,und somit nicht vom Linux-Kern auf mehrere CPUs verteilt werden können. Dies könnte nur im Scheduler des Linux-Kerns selbst geschehen, der als Prozess aber nur das gesamte Programm inlusive aller Threads, nicht jedoch einzelne Threads selbst kennt.

41.1.5 Clone(s) und Cloning

Um dieses Manko gegenüber anderen Unix-Systemen einzuschränken, bietet Linux ein thread-ähnliches Konstrukt an. Man kann mittels des Sy-

[8] Siehe auch: http://pauillac.inria.fr/ xleroy/linuxthreads/

stemaufrufs *clone()* einen, dem System bekannten, Prozeß erzeugen der mit seinem erzeugenden Prozeß verschiedene Ressourcen teilt. Da hierzu auch Speicherbereiche des Datenbereichs zählen können reicht ein solcher “geclonter” Prozeß schon recht gut an einen Thread heran.
Die durch *clone()* erzeugten Prozesse sind mit eigener Prozeß-Id im System bekannt und können somit auf verschiedenen CPUs gescheduled und ausgeführt werden.

Bei Linux werden alle Systemaufrufe zum Erzeugen neuer Prozesse (vfork, execX, clone) über die in der Datei */usr/src/linux/arch/kernel/process.c* definierten Systemaufrufe an die kernelinterne Funktion *do_fork()* weitergeleitet. Diese ist wiederum in */usr/src/linux/kernel/fork.c* implementiert. Auf sie wird an späterer Stelle noch eingegangen.

```
/* Auszug aus: /usr/src/linux/arch/i386/kernel/process.c */

    /* SNIP*/
    asmlinkage int sys_fork(struct pt_regs regs)
    {
->          return do_fork(SIGCHLD, regs.esp, &regs);
    }

    asmlinkage int sys_clone(struct pt_regs regs)
    {
         unsigned long clone_flags;
         unsigned long newsp;

         clone_flags = regs.ebx;
         newsp = regs.ecx;
         if (!newsp)
                 newsp = regs.esp;
->       return do_fork(clone_flags, newsp, &regs);
    }

    asmlinkage int sys_vfork(struct pt_regs regs)
    {
->       return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp,
&regs);
    }

    /* SNIP */
```

Die Funktion *do_fork* selbst ist kein Sytemaufruf, sondern bietet über die erwähnten Systemaufrufe wohldefinierte Schnittstellen. Auf die *clone()* Funktion wird im Kapitel *Ideen zur Verbesserung von Pim* näher eingegangen.

42. Das Teilprojekt PiM

42.1 Einleitung

Unser Teilprojekt *PiM* sah seine Aufgabe nicht darin, an der Verfeinerung des Blockierens der Systemaufrufe zu partizipieren. Dies lag unter anderem auch an der durch die Rasanz der Entwicklung innerhalb dieses Gebietes vernachlässigten Dokumentation, die eigentlich nur innerhalb der Source-Codes zu finden war. Es existiert zwar eine Internetseite zu Linux-SMP[1]., jedoch ist diese auf einem hoffnungslos veraltetem Stand Hätten wir Änderungen am Locking vorgenommen, so hätten sich diese mit an Sicherheit grenzender Wahrscheinlichkeit mit denen der SMP-Hauptentwickler[2] überschnitten, oder sie hätten an Stellen stattgefunden, die bereits geändert aber noch nicht veröffentlicht worden sind.
Da es nicht Ziel sein konnte, ein von vornherein obsoletes Projekt zu beginnen, haben wir uns einem anderen, wie wir meinen, SMP-relevanten Teil eines Betriebssystems zugewandt.
Einige Ressourcen in einem Computer sind von ihrer Art im Prinzip nicht zwischen mehreren laufenden Prozessen aufzuteilen. Speziell gehören in diese Kategorie die zeichenorientierten Geräte, insbesondere die serielle Schnittstelle. Die Möglichkeit zur sinnvollen gemeinsamen Nutzung, gerade innnerhalb von SMP-Systemen, besteht hier nur in der Nutzung der Linux-Thread-Library.
So könnten sich mehrere Threads eines Prozesses die serielle Schnittstelle teilen. Da die Prozesse eines Systems nicht aus nur einer Quelle stammen, wäre es bereits für den oder die Entwickler eines anderen Prozesses nahezu unmöglich, sich an dieser Ressource zu beteiligen.
Der oder die Entwickler müßten alle anderen Prozesse (multithreadded oder nicht), die außer Ihrem eigenen auf die Schnittstelle zugreifen:

- 1. Im Quellcode zur Verfügung haben
- 2. An das Verhalten ihres eigenen Prozesses anpassen
- 3. Diese Anpassungen auch vornehmen dürfen

Die Haupteinsatzgebiete , wo die Verfügbarkeit der zeichenorientierten Geräte verbreitet, und die Aufteilung auf mehrere Prozesse wünschenswert, wenn nicht sogar notwendig ist, sind nahezu alle Prozesse der Meßwerterfassung und Steuerung. Da in diesem Bereich die Software sowohl kommerziell genutzt als auch entwickelt wird, ist die Wahrscheinlichkeit, daß die Punkte 1.-3. erfüllt sind, äußerst gering. Somit ist das Gerät zwar unter mehreren Threads, weiterhin aber nicht unter mehreren Prozessen teilbar.

Die Idee von *Pim* ist es, die serielle Schnittstelle über Anpassungen am Linux-Kern mehreren Prozessen zur Verügung zu stellen, ohne das diese voneinander Kenntnis haben müssen.
Die Informationen, die Prozesse beim lesenden Zugriff erhalten, sind

[1] www.uruk.org/ erich/mps-linux-status.html
[2] Alan Cox und Erich Boleyn

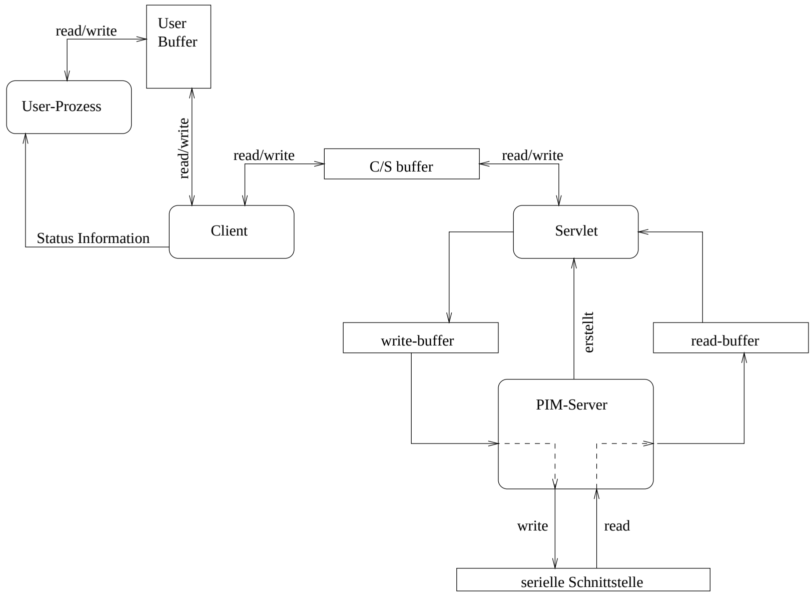


Abbildung 42.1: Schematische Darstellung des PiM-Systems

hierbei zunächst für alle beteiligten Prozesse gleich.

Pim besteht aus einem *Pim-Server*, n-Clients und n-Servlets, bei n auf die serielle Schnittstelle zugreifenden Prozessen.
Der Zugriff auf die Ressource geschieht hierbei noch nicht völlig transparent, das heißt insbesondere User-Level Prozesse müssen sich an die *Pim* konventionen (siehe *Die Pim-Konventionen*) halten. Auf der Verwaltungsebene der *Pim-Server*, *Pim-Clients* und *Pim-Servlets* stellt sich ein *Pim*-System mit einem User-Level-Prozess, der die serielle Schnittstelle nutzt, schematisch so dar:

Für jeden read- oder write-Auftrag des User-Prozesses wird ein *Pim*-Client mittels des Systemaufrufs *clone* gestartet. Aus User-Prozess-Sicht kommt jeder read- oder write-Auftrag, egal ob synchron oder asynchron sofort zurück. Die Verweildauer des User-Prozesses im Kern bzw. in Kernfunktionen ist also minimal.
Der *Pim* Client übernimmt in der Folge die Abarbeitung des Auftrags im Hintergrund. Er benötigt zur Abarbeitung eines Auftrags Zugriff auf das Datensegment des User-Prozesses, um entweder gelesene Daten an eine vom User-Prozess definierte Speicherstelle zu kopieren oder von dieser Speicherstelle Daten auf die serielle Schnittstelle zu schreiben. Dazu kommuniziert der Client über Message Queues mit dem Server und überlässt diesem das reale Schreiben oder Lesen der seriellen Schnittstelle.
Der Server startet je User-Prozess-Auftrag ein sogenanntes *Pim*-Servlet, das die Datenkonsistent im *Pim* System bewahrt.

Der User-Prozess bekommt bei einem asynchronen Auftrag Statusinformationen vom *Pim* Client übermittelt. Diese enthalten auch die Anzahl der Bytes, die bereits verarbeitet wurden, um dem User-Prozess die Möglichkeit zu geben, Daten die z.B. ansynchron gelesen werden bis zum letzten aktuell gelesenen Byte zu bearbeiten.
Bei den *Pim* Demo-Programmen[3] wird diese Statusinformation genutzt, um bei asynchron gelesenen Bilddaten parallel zu deren Anzeige und

[3] Siehe auch Kapitel Implementierung -> Demoapplikation

Empfang ein Histogramm zu berechnen.
Nach Beendigung des Auftrags werden sowohl der Client als auch das Servlet vom Server beendet, und der User-Prozess erhält als letzte Information die Prozess ID des Client Prozesses. Beim *Pim* System kann eine beliebige Anzahl völlig unabhängiger Prozesse die serielle Schnittstelle in der beschriebene Weise benutzen, ohne über interne Protokolle die Datenkonsistenz selbst erhalten und überwachen zu müssen. Die von *Pim* gebotenene Möglichkeiten zum Zugriff sind:

- 1. Öffnen der Schnittstelle
- 2. Synchrones Lesen von der Schnittstelle
- 3. Asynchrones Lesen von der Schnittstelle
- 4. Schreiben auf die Schnittstelle
- 5. Schliessen der Schnittstelle

Das *Pim* System lässt sich auch benutzen, wenn man die serielle Schnittstelle weiterhin wie bisher, exclusiv für einen Prozess, nutzen möchte. Hierbei entsteht nur ein erhöhter Speicherbedarf von max. 300Kb durch die *Pim* Erweiterungen Server, Servlet und Client.

42.1.1 Erforderliche Voraussetzungen zur Benutzung des Pim-Systems

- Ein Linux Kernel der Version 2.2.x oder höher
- Ein C Compiler (z.B.: gcc oder egcs) mit glibc2 oder libc5
- Die Source-Codes eines Linux-Kernels der Version 2.2.x oder höher
- Neucompilieren des Kernels mit den *Pim* Erweiterungen
- Starten des *Pim* Servers beim Systemstart

42.1.2 Konventionen zur Benutzung von Pim im User-Level

Das *Pim* System soll eine Möglichkeit zur Aufteilung einer ansonsten exklusiven Ressource auf mehrere unabhängige Prozesse in einem SMP-System bieten. Für das *Pim* System wurde die serielle Schnittstelle ausgewählt. Um das *Pim*-System im User-Level nutzen zu können muss ein Prozess noch einige Konventionen einhalten. Ein kurzer Beispiel User-Prozess könnte wie folgt aussehen:

Dies ist ein Demo User-Prozess, der asyncron 1000 Bytes von der seriellen Schnittstelle liest.

```
/*  Einbinden der Pim Header-Dateien */
\index{all}{PiM}{PiM-Headerdateien}
#include "/INCLUDE_DIR/pim.h"
#include "/INCLUDE_DIR/pimclient.h"
```

Vorwärtsdeklaration der in den Header-Dateien Variable *pim_client_-read_data*, in der die Anzahl der aktuell verarbeiteten Bytes fuer den User-Prozess enthalten ist.

```
extern long pim_client_read_data;
```

Vorwärtsdeklaration der in den Header-Dateien definierten Funktion zum starten von Pim

```
extern int start_thread(void (*fn)(), void* );
```

Hauptprogramm des User-Prozesses

```
int main( int argc, char** argv )
{
```

Benötigte Variablen :

```
    struct pim_args* str_p;
    int pid;
```

Ein Pointer auf die Struktur pim_args, in der der Auftrag definiert wird und ein Integer, der später die Prozess ID des Clients zugewiesen bekommt.

Allokieren des benötigten Speicherplatztes Hier wird der Pim-Client die Daten später ablegen.

```
    temp_buffer = (char *)malloc(1000);
```

Initialisierung der pim_args struct. Immer in der Form:

1. Lesen oder schreiben (TASK_READ | TASK_WRITE)
2. Angabe der Speicheradresse von der gelesen oder in die geschrieben werden soll
3. Anzahl der zu lesenden oder zu schreibenden Bytes
4. Synchron oder asynchron (HOW_SYNC | HOW_ASYNC)

```
    PIM_ARGS.task = TASK_READ;
    PIM_ARGS.buffer = temp_buffer;
    PIM_ARGS.count = 1000;
    PIM_ARGS.how = HOW_SYNC;
```

Ausrichten des Pointers auf die erzeugte PIM_ARGS Struktur.

```
    str_p = &PIM_ARGS;
```

Erteilen des Auftrags mit *start_thread()*. Der Name *start_thread()* ist von uns frei gewählt und hat nichts mit der Linux-Thread-Library gemeinsam, noch werden Teile dieser benutzt.

```
    pid = start_thread( pim_client, (void*) str_p );
```

Ab hier ist der Auftrag an das Pim-System wie in PIM_ARGS definiert erteilt.

```
    while( !waitpid
           ( pid, PIM_CLIENT_RET_VAL, WNOHANG ))
    {
```

Hier könnte ein Programm mit neuen Daten arbeiten, wenn diese empfangen wurden oder andere Aufgabe wahrnehmen. Beispiele hierzu sind im Kapitel *Implementation* und *Demoapplikation* aufgeführt.

```
    }

    free((void *)temp_buffer);
    exit(0);
}
```

Der oben als Beispiel zur Benutzung angegebene User-Prozeß ist die minimale Implementierung zur Benutzung des *Pim*-Systems.
Er macht nichts mit den Daten und führt auch parallel zu deren Empfang, der über den *Pim*-Client-Prozess im Hintergrund läuft auch keine anderen Operationen aus. Selbst dieser Prozeß hat jedoch den Vorteil, daß er die Ressource nutzen kann, ohne zu wissen wie andere Prozesse diese zur Zeit benutzen. Der Datenempfang ist durch *Pim* in jedem Fall sichergestellt.

43. Ideen zu Verbesserung des *Pim*-Systems

Unsere Idee war nicht von Beginn an die oben beschriebene. Wir wollten ursprünglich einen eigenen Systemaufruf zum Zugriff auf die serielle Schnittstelle bieten, der nach außen wie der alte Aufruf aussehen sollte.
Der Client-Prozess sollte vollständig ohne Wissen des User-Prozesses erzeugt werden, den Auftrag erledigen und sich auch ohne Wissen des User-Prozesses beenden. Dies konnte in der aktuellen Version nicht erreicht werden.
Es gibt noch weitere nicht oder nicht vollständig umgesetzte Verbesserungsideen, für die jedoch Teilweise schon Vorbereitungen implementiert sind.
So ist bereits darüber diskutiert worden, daß Linux beim Scheduling von SMP Prozessen keinen besonderen Algorithmus zur Verfügung stellt. Desweiteren wurde beschrieben, daß Prozesse auf andere CPUs verlagert werden können.
Für das *Pim*-System wäre es ideal, wenn im Falle des Verlagerns eines User-Prozesses oder eines Clients auf eine andere CPU, der jeweilig dazugehörende Prozess auch verlagert würde.
Dies ist aus Performancegründen insofern interessant, da der Client in jedem Fall auf Datenbereiche des User-Prozesses zugreift und es von Vorteil ist, wenn beide auf dem Datencache *derselben* CPU arbeiten.
Zusätzlich wäre es sinnvoll, daß die *Pim*-Prozesse im Scheduler in einer eigenen Routine behandelt werden, um ihnen nötigenfalls Vorrang zu verleihen, oder sie zu unterbrechen, falls sehr rechenintensive Prozesse die CPU besser auslasten würden.
Hierzu ist es nötig, daß bestimmte Prozesse als benutzende oder bedienende Teile des *Pim*-Systems identifiziert werden können.
Ausgehend vom Verzeichnis

```
/usr/src/linux
```

werden im folgenden die nötigen Änderungen und Erweiterungen in den Linux-Kernel Dateien aufgeführt, die diese Verbeserungen als Voraussetzung benötigen oder die Implementierung dieser vereinfachen.
Die Kernelversion der besprochenen Dateien ist 2.2.14.

43.1 Vorbereitungen für spätere Erweiterungen

- Datei: sched.h

Clone-Flags Die für das Linux-Task-Management benötigten Strukturen, Variablen, Funktionen und Makros befinden sich einheitlich in der Datei:

```
/include/linux/sched.h
```

Sehr wichtig für eine Erweiterung des Scheduling im *Pim*-Projekt ist die Definition der Flags des Systemaufrufs *clone()*. Dieser erzeugt (wie in SMP bei Linux beschrieben) einen Thread-ähnlichen Prozeß, der sich mit seinem erzeugenden Prozeß bestimmte Ressourcen teilen kann. Diese Prozeßresourcen werden vor der Erzeugung über Flags definiert. Die Flags können bei Bedarf durch ein logisches *oder* kombiniert werden.

```
/* SNIP */
\index{all}{PiM!System!CLONE_FLAGS}
/* cloning flags: */

#define CSIGNAL         0x000000ff
#define CLONE_VM        0x00000100
#define CLONE_FS        0x00000200
#define CLONE_FILES     0x00000400
#define CLONE_SIGHAND   0x00000800
#define CLONE_PID       0x00001000
#define CLONE_PTRACE    0x00002000
#define CLONE_VFORK     0x00004000
```

Um festzustellen, ob der erzeugte Prozess Teil des *Pim*-Systems ist, wurden die Clone-Flags erweitert. CLONE_PIM bedeutet nur, daß der Prozeß Teil des Systems ist. CLONE_CLI, CLONE_USR und CLONE_SVL geben an, ob es der User-, Client- oder Servlet-Prozess ist.

```
#define CLONE_PIM       0x00005000
#define CLONE_CLI       0x00006000
#define CLONE_USR       0x00007000
#define CLONE_SVL       0x00008000
```

Zur speziellen Identifikation der Funktion des Prozesses im *Pim*-System (Client/Server/Servlet) haben wir der *task_struct* einige Parameter hinzugefügt.

In der Task-struct sind bei Linux alle prozeßrelevanten Parameter festgehalten. Für jeden Prozess wird eine Struktur erzeugt und in der verketteten Liste: *task[NR]* eingetragen.

```
struct task_struct {
        /* SNIP */
        int pim_is_pim;
```

Dieser Wert wird bei allen erzeugten Prozessen auf 0 gesetzt. Nur solche, bei denen CLONE_PIM als Parameter des *clone()* Aufrufs angegeben wurde, erhalten hier den Wert 1.

```
        int pim_is_what;
```

Diese Variable Kann drei verschiedene Werte annehmen, je nach Prozessart im *Pim*-System. Hierbei steht der Wert 0 für den User-Prozess, 1 für den Client und 2 für das Servlet.

```
        int pim_pid;
}
```

Sollte es sich um ein Servlet handeln, ist dieser Wert 0. Für einen Client hat dieser Wert die *PID* des User-Prozesses, den er bedient, und für den User-Prozess ist dies die *PID* des ihn bedienenden Client-Prozesses.

- Datei: fork.c

Hier müssen die neuen Parameter der *task_struct* belegt werden. Da dies wie beschrieben immer in der Funktion *do_fork()* geschieht, ist hier der ideale Ort um unsere Prozesse zu identifizieren und die Parameter entsprechend zu belegen.

```
\index{all}{PiM!Systemaufrufe!do_fork()}
int do_fork(unsigned long clone_flags,
            unsigned long usp,
            struct pt_regs *regs){

    /* SNIP */

#ifdef __SMP__
                int i;
                p->has_cpu = 0;
                /* das ist so okay f\"ur uns */
                p->processor = current->processor;
------>         if ( clone_flags & CLONE_PIM ){
                    p->pim_is_pim = 1;
                    if ( clone_flags & CLONE_USR )
                        p->pim_is_what = 0;
                        current->pid = p->pid;
                    if ( clone_flags & CLONE_CLI )
                        p->pim_is_what = 1;
                        p->pim_pid = current->pid;
                        p->processor = current->processor;
                    if ( clone_flags & CLONE_SVL )
                        p->pim_is_what = 2;
------>         }

                for(i = 0; i < smp_num_cpus; i++)
                        p->per_cpu_utime[i] =
                                p->per_cpu_stime[i] = 0;
                spin_lock_init(&p->sigmask_lock);
        }
#endif
/* SNIP */
```

Mit diesen Erweiterungen sollte es möglich sein, *Pim*-Prozesse mit einer eigenen Funktion, die in *sched.c* hinzugefügt werden müßte, seperat von anderen Prozessen im System zu schedulen.

Insbesondere sollte festgestellt werden, ob der User- oder Client-Prozess die CPU wechseln, um den jeweiligen Partnerprozess auch auf die neue CPU zu wechseln.Dies würde gerade bei Pentium/II und Pentium/III Systemen eine Leistungsverbesserung zur Folge haben, da die auzutauschenden Daten im CPU-internen Cache sind.
Weiterhin könnten im Scheduler die Cache-Treffer und Cache-Fehler der *Pim*-Prozesse gemessen werden, und User-/Client-/Servlet-Prozesse, die eine Grenzanzahl an Cache-Fehlern aufweisen, abgescheduled werden.
Die Cache-Fehler bilden, gerade bei Systemen mit Caches innerhalb der CPU selbst, einen enormen Anteil der Leistungsverschlechterung, da im Fehlerfall der Cache mit Daten aus dem Hauptspeicher aktualisiert

werden muss, und dieser, im Gegensatz zum CPU-internen Cache,[1] mit maximal 133 MHz betrieben wird. Dies stellt bei modernen CPUs mit mehr als 400 MHz Taktfrequenz einen typischen Flaschenhals da.
Eine weitere mögliche Verbesserung bestünde in der Spezifikation und Implementierung eines geeigneten Kommunikationsprotokolls vom Pim-Server-Programm auf dem Client-Rechner und Pim-Server-Programm auf dem Server-Rechner.
Dies würde es, im Gegensatz zum aktuellen Stand ermöglichen, daß Daten, die über die serielle Schnittstelle übertragen werden, an ganz bestimmte User-Prozesse versandt werden können. So könnte ein Client-Rechner Meßwerte von verschiedenen Stellen erfaßen und an spezielle Prozesse senden, die diese Daten unterschiedlich aber parallel verarbeiten können.

[1] in der Regel mit halbem CPU-Takt oder schneller betrieben

44. Spezifikation

Für die Spezifikation erwies es sich als günstig, daß wir kein vorgegebenes System verifizieren wollten. Ganz entsprechend der eigentlichen Idee von Spezifikation als Vorstufe der Entwicklung konnten wir verschiedenen Alternativen auf der Spezifikationsebene untersuchen. Neben der Konkretisierung unserer Idee entstanden so auch schriftliche Annäherungen an eine Spezifikation der einzelenen Komponenten, aus denen unser System besteht.

44.1 Systemkomponenten

Ein Meilenstein vor dieser Notation war die Erstellung eines Diagramms, das die Beziehungen zwischen den einzelnen Systemkomponenten beschreibt. Der abschließende Schritt war die Überführung in die CSP-Notation, wodurch es uns möglich war, Tests mit dem Tool *FDR* durchzuführen.

44.2 FDR und Tests

FDR ist ein Tool, das erlaubt auf Basis von CSP durch Verfeinerung des Systems, bestimmte Eigenschaften dieses Systems zu beweisen. Mit dem Tool ist es auch möglich, das System auf Dead- und Livelocks und Determinismus zu prüfen. *FDR* unterstützt drei verschiedene Verfeinerungsmodelle: Trace Refinement, Failures Refinement und Failures-Divergences Refinement. Bei der Verifizierung unseres Systems haben wir das Traces Refinement Modell benutzt. Ein Trace eines Prozesses ist eine Folge von Events, die der Prozeß durchführen kann. Der Prozeß Q ist eine Verfeinerung des Prozesses P im Trace Model, wenn alle mögliche Traces von Q auch Traces von P sind. Wenn wir P als Spezifikation betrachten, welche sichere Zustände von unserem System beschreibt, dann ist Q die sichere Implementierung im Sinne von Verfeinerung im Trace Modell: keine falschen Events sind erlaubt.

44.3 CSP-Spezifikation

Es folgt nun die Beschreibung der Komponenten, sowohl funktional, als auch als Umsetzung nach CSP. Dabei sieht man auch gut, daß wir die CSP-Spezifikation dahingehend optimieren mußten, daß FDR sie auch verarbeiten kann. Deshalb wird hier nur die an FDR angepaßte Version beschrieben, die einige starke Vereinfachungen beinhaltet bzw. wo bestimmte Komponenten entfernt wurden. Im konkreten System sind sie aber nachwievor vorhanden. Beispiele dafür sind der APIC, TIMER, CPU, ENTRY. Die wichtigen Bestandteile des PiM-_Systems sind natürlich vertreten (USER-Prozeß, SERVER, CLIENT, MsQ etc.). Unsere Sicht der Dinge beginnt beim User-Prozeß (U). Wenn dieser nicht im Moment Berechnungen durchführt, oder andere System-Calls ausführt, ruft er einen

System-Call auf, der einen unserer Clients startet (*trigger_client*). Nach dessen Aufruf wird die Bearbeitung fortgeführt.

```
U(i) =  something -> U(i)
        |~|
        sys_call -> trigger_client -> U(i)
```

Diese Zeile startet mehrere (*user_count*) USER-Prozesse parallel.

```
USER = (|||i:{0..(user_count-1)} @ U(i))
```

Der Event *trigger_client* veranlaßt die Verarbeitung des Clients ((*INIT_CLIENT* und (*CLIENT*). Die erste Aufgabe des Clients ist, den MessageQueue-Key des Servers aus der Datei zu lesen, was durch einen einfachen CSP-Prozeß und einen Event simuliert wird. Im Folgenden wurde die Menge der zu bearbeitenden Daten ermittelt. Hier wird ein zufälliger Wert durch *BYTE_AMOUNT_GENERATOR* erzeugt und dem Client über den Event *rd_bag* mitgeteilt. Im tatsächlichen System ist dieser Wert Teil des Auftrages, den der User-Prozeß an den Client abgibt, d.h. er ist nicht zufällig erzeugt.

```
BYTE_AMOUNT_GENERATOR = (|~| i:{0..(byte_to_read_max-1)} @
                        B_A_GENERATOR(i))

B_A_GENERATOR(x) = rd_bag!x -> B_A_GENERATOR(x)
                   []
                   wr_bag?y -> B_A_GENERATOR(y)
```

Auch die Art der Verarbeitung (asynchron, oder synchron) wird im tatsächlichen System nicht zufällig bestimmt, sondern ist durch den User-Prozeß bestimmt.

```
INIT_CLIENT = trigger_client -> get_msq_key?client_key ->
              rd_bag?amount ->
              CLIENT(client_key,amount,type_of_read.asynchron)
              |~|
              CLIENT(client_key,amount,type_of_read.synchron))
```

In der weiteren Verarbeitung des Clients wird der Auftrag an den Server übermittelt. So wird dem Server der MessageQueue-Key des Clients mitgeteilt, der für die weitere Verarbeitung von Bedeutung ist. Der Server antwortet mit einem Wert. Dieser Wert ist entweder positiv, dann ist das ein legaler SharedMemory-Key, auf den der Client zugreifen kann, oder es ist ein negativer Wert (-1), was bedeutet, daß der Server die Anfrage des Clients in Moment nicht bearbeiten kann. In diesem letzten Fall muß der Client noch einmal auf eine Message von Server warten. Der Wert der in Key gespeichert ist, dient in unserer CSP-Spezifikation lediglich dazu, den richtigen zur Anfrage passenden Client zu identifizieren. Die parallel laufenden Clients kommunizieren über die Kanäle *order_msq* und *request_service* mit dem Server, und so könnte es zu Verwirrungen kommen. Die Clients haben lokal ihren eigenen MessageQueue-Key gespeichert und vergleichen die Antwort des Servers (in key) mit diesem. Sind diese identisch, war die Antwort des Servers für sie bestimmt. Der READ-Prozeß, der hier gestartet wird, ist nur eine geeignete Repräsentation des Lesens im tatsächlichen System und wird deswegen hier nicht weiter erläutert.

```
CLIENT(client_key,amount,how) =
                       start_client -> order_msq!client_key ->
                       request_service?key.x ->
                       if (x == true and key==client_key) then
                         (read ->READ(amount,amount,how);
                         END_CLIENT(client_key))
                       else
                         CLIENT(client_key,amount,how)
```

Der Server überprueft bei einer Anfrage seitens der Clients zunächst die Anzahl der Aufträge, die bereits berarbeitet werden (*out_client_count1* und *out_client_count2*).

```
SERVER = (order_msq?client_key -> out_client_count1?x ->
          out_client_count2?y ->
```

Wenn es keine solche gibt, dann wird ein Prozeß gestartet, der die Daten bytesweise vom Device liest und in einen “shared memory”-Bereich schreibt (*READ_FROM_DEVICE*). Die Clients können auf diesen Bereich zugreifen.

```
          if((x==0) and (y==0))
          then
             start_read_from_device
          else
             something ->
```

Danach wird abgefragt, ob die Anzahl der momentan laufenden Clients schon maximal ist. Wenn ja, dann wird der Client in die Waitqueue eingefügt. Und ihm wird über den Channel *request_service* mitgeteilt, daß er warten muß. Wenn die Waitqueue voll ist, wird die Anfrage verworfen.

```
         if ( (x+y) ==( run_clients-1) )
         then
            (ADD_TO_WAITQUEUE(0, client_key);
            request_service!client_key.false -> SERVER)
```

Wenn aber die maximale Zahl noch nicht erreicht ist, wird der Client in die Runqueue eingefügt. Es wird ihm die CPU zugeordnet, auf welcher zur Zeit weniger Clients ausgeführt werden. Danach wird ihm über den Channel *request_service* mitgeteilt, daß er mit seinem Auftrag beginnen kann.

```
        else
           if (x <= y)
           then
              (ADD_TO_RUNQUEUE(0,client_key,1);
               inc_client_count1 ->
               request_service!client_key.true ->
               SERVER)
           else
              (ADD_TO_RUNQUEUE(0,client_key,2);
               inc_client_count2 ->
               request_service!client_key.true ->
               SERVER))
```

Parallel wartet der Server auf dem Channel (*end_client_msq*) auf die Nachrichten von Clients. Falls eine Nachricht von irgendeinem Client ankommt, dies bedeutet, daß der Client seinen Auftrag erledigt hat. Daher wird er aus der Runqueue gelöscht.

```
[]
(end_client_msq?end_client_key ->
 DEL_FROM_RUNQUEUE(0,end_client_key);
```

Der nächste Client wird aus der Waitqueue selektiert und genau wie im oberen Teil in die Runqueue eingefügt.

```
SELECT_FROM_WAITQUEUE(0); rd_select?client_key ->
out_client_count1?x -> out_client_count2?y ->
if ( client_key != 0 )
then
   if (x <= y)
   then
      (ADD_TO_RUNQUEUE(0,client_key,1);
       inc_client_count1 ->
       request_service!client_key.true ->
       SERVER)
   else
      (ADD_TO_RUNQUEUE(0,client_key,2);
       inc_client_count2 ->
       request_service!client_key.true ->
       SERVER)
```

Wenn es keinen laufenden Client mehr gibt, wird der Prozeß *READ_FROM_DEVICE* beendet.

```
else
   if(x==0 and y==0)
   then
      (end_read_from_device -> SERVER)
   else
      SERVER)
```

44.4 Verifikation

Die oben beschriebene CSP-Prozesse haben wir zu einem *Sys*-Prozeß zusammengefaßt. *Sys* beschreibt damit das Verhalten unseres Systems, wie die Prozesse miteinander kommunizieren und die Art dieser Kommunikation (parallel, parallel synchronisiert). Anschließend haben wir *Sys* mit *FDR* auf Deadlocks*FDR* und Lifelocks getestet und festgestellt, daß Sys deadlock- und lifelockfrei ist.

```
Sys= ((((USER [|{|trigger_client|}|] (INIT_CLIENT
                [|{|rd_bag,wr_bag|}|]
                BYTE_AMOUNT_GENERATOR))[|{|get_msq_key|}|]
                MSQ_QUEUES_KEYS(1) ||| CLIENT_COUNT1(0) |||
                CLIENT_COUNT2(0) ||| RUN_QUEUE |||
                WAIT_QUEUE |||
                SELECTED_CLIENT ||| DATA_SEGMENT )
                [|{|request_service,order_msq,end_client_msq,
                inc_client_count1,
                inc_client_count2,dec_client_count1,
                dec_client_count2,out_client_count1,
                out_client_count2,
                rd_select,wr_select,rd_rq,wr_rq,rd_wq,wr_wq,
                rd_ds,wr_ds|}|]
                SERVER) [|{|shm,start_read_from_device,
                end_read_from_device|}|]
                (DEVICE [|{|byte_of_data|}|]
                START_READ_FROM_DEVICE)
```

Für die Verifizierung unseres Systems haben wir das Trace Refinement Modell gewählt. Da wir nur das gewünschte Verhalten unseres Systems spezifiziert und keine Ausnahmesituationen betrachtet haben, wie zum Beispiel den Absturz des Server-Prozesses, das Trace Refinement Modell war in unserem Fall ausreichend. Wir haben folgende drei Bedingungen formuliert, die das richtige Verhalten vom *Sys* beschreiben:

1. Der Client startet, erledigt seinen Leseauftrag, beendet sich (*ABS_CL*). Der Prozeß *ABS_CLIENT* definiert dabei mehrere abstrakte Clients, die parallel laufen.

```
Q1 = ABS_CLIENT ||| CHAOS_CLIENT
ABS_CLIENT = ||| i:{1..run_clients}@ ABS_CL(i)
ABS_CL(i) = start_client -> read -> end_client -> ABS_CL(i)
```

2. In *Sys* kann nur ein Prozeß *READ_FROM_DEVICE* zu einem Zeitpunkt laufen. Irgendwann wird er vom Server beendet, was bedeutet, daß alle Clients ihre Leseaufträge erledigt haben.

```
Q2 = ABS_READ_FROM_DEVICE ||| CHAOS_READ_FROM_DEVICE
```

```
ABS_READ_FROM_DEVICE =start_read_device -> end_read_device ->
ABS_READ_FROM_DEVICE
```

3. Der User erhält die gelesenen Daten des von ihm gestarteten Client zurück. (byteweise beim synchronen Read, oder das letzten Byte beim asynchronen Read, was bedeutet, daß der Client seine Aufgabe richtig erfüllt hat).

```
Q3 = ABS_USER ||| CHAOS_USER
ABS_USER = byte_to_user -> ABS_USER
           |~|
           last_byte_to_user -> ABS_USER
```

Die Events des abstrakten Prozesses sind bei den Prozessen von *Sys* an den entsprechenden Stellen eingebaut. Wir haben diese drei Verifizierungsbedingungen mit *FDR* auf Trace Refinement mit jeweils positivem Ergebnis geprüft (bei unterschiedlicher Anzahl von Clients und User-Prozessen).

```
assert Q1[T=Sys
assert Q2[T=Sys
assert Q3[T=Sys
```

Hier ist die Ergebnistabelle (Bedingung1) :

```
Refine checked 14448 states
With 47720 transitions
Stopped timer

Resource   Start            End              Elapsed
Wall time    194011           194022               11
CPU (self)        4               13                9
CPU (sys)         0                0                0
(inc children)

CPU (self)        0                0                0
CPU (sys)         0                0                0
```

45. Implementation

In diesem Kapitel wollen wir den Prozeß der Implementierung, sowie die dabei entstandenen Probleme beschreiben. Die Wahl der Progammiersprache fiel aus nachvollziehbaren Gründen auf C.

In der Endversion ist es geplant, daß der Kernel bei einem zu behandelnden System-Call das Clonen des User-Prozesses übernimmt, wonach die Verarbeitung wie in den vorherigen Kapiteln vonstatten gehen soll. Die Technik des Clonens von seiten des Kernels war aber nicht vollständig ermittelbar, so daß wir in der Zwischenzeit in Zeitdruck gerieten, der uns veranlaßte, von unserem Plan abzurücken und einige Eingeständnisse zu machen.

Zunächst lösten wir uns von der Vorstellung, der Kernel könnte das Clonen ohne das Wissen des User-Prozesses übernehmen. In der nun von uns implementierten Zwischenlösung macht dies der User-Prozeß selbst, was dazu führt, daß wir die gewünschte Transparenz bezüglich der bisherigen Programmierweise aufgeben mußten. Der Programmierer ist also nun selbst für das Clonen zuständig.

Es ist aber immernoch so einfach geblieben, daß dafür nur eine Funktion *start_thread* aufgerufen werden muß, die zwei Argumente erhält: einen Funktionspointer auf eine ebenfalls bereits fertige Funktion, sowie eine Struktur, die wiederum Argumente für die Funktion hinter dem Funktionspointer enthält. Als einzige Hürde muß der Programmierer lediglich noch zwei Headerdateien mittels #include-Statement einfügen.

```
#include <pim.h>
#include <pimclient.h>
```

In einer dieser Dateien ist außerdem eine Variable deklariert, welche vom gestarteten Client mit dem jeweils aktuellen Lese- bzw. Schreibstand beschrieben wird; so kann sie genutzt werden, um z.B. auf gelesenen Daten zu arbeiten, obwohl der Lesevorgang als Ganzes noch nicht beendet wurde. Die genaue Technik hierzu wird aber in einem späteren Teil beschrieben. Für den User-Prozeß steht hier nun mehr keine weitere nötige Arbeit an, um den Lese-/Schreibvorgang zu betreiben.

Alles weitere wird von den Clients, dem Server, den Servlets, sowie dem Lese- und, Schreibprozeß erledigt. Die Kommunikation erfolgt durch die Nutzung von Standardkomponenten. Funktionsbezogene Mitteilungen werden mit Message Queues übermittelt, Daten über Shared Memory. Wir haben dabei nicht auf Laufsicherheit (Safety), und auch nicht auf Zugriffssicherheit (Security) geachtet; dies war ein Punkt, der uns zur Fertigstellung des Systems am unwichtigsten erschien.

Bezogen auf die Laufsicherheit konnten wir dennoch einigen Problemen im Vorraus dadurch entgehen, daß wir unser System auf Basis der getesteten CSP-Spezifikation implementierten. Dort waren einige Algorithmen schon soweit ausgebildet, daß es nur noch geringe Mühe machte, diese zu portieren. Andere wiederum waren in der Spezifikation zu unwesentlich, als daß wir sie konkretisieren wollten. Das Lesen von der seriellen Schnittstelle z.B. wird durch einen einfachen Event *start_read_from_device* unter CSP repräsentiert.

```
...
SERVER = (order_msq?client_key -> out_client_count1?x ->
         out_client_count2?y ->
         (if((x==0) and (y==0)) then start_read_from_device
         else something) ->         ^^^^^^^^^^^^^^^^^^^^^^
...
```

Dieser Event veranlaßt nichts weiter; das tatsächliche Lesen wurde also nicht in CSP umgesetzt. Das wiederum brachte uns auch einige Probleme, die aber später näher erleutert werden.

45.1 Die Programme

45.1.1 User- und Clientprogramm

Im Programm des User-Prozesses müssen zunächst die Header-Dateien *pim.h*, sowie *pimclient.h* eingebunden werden.

```
#include <pim.h>
```

Wie bereits erwähnt, enthält diese Datei grundlegende Konstanten, Struktur-Definitionen, und die Definitionen der Funktionen, die die Server- und Servletfunktionalität zur Verfügung stellen. Hier befindet sich auch die Funktion *start_thread*, die das spätere Clonen des User-Prozesses ausführt. Der Grund für diesen Umweg wurde schon vorher beschrieben.

```
#include <pimclient.h>
```

Hier sind die Funktionen definiert, die für die Funktion der Clients zuständig sind. In dieser Datei ist auch die globale Variable *pim_client_read_data* deklariert; sie dient dem synchronisierten Zugriff auf die Daten beim asynchronen Lesen von der Schnittstelle.

Der Programmierer muß sich zuerst überlegen, welche Programmteile er wie parallelisieren will, d.h. wo er einen Nutzen daraus ziehen kann, daß er noch während des Lesevorganges auf die bereits erhaltenen Daten Zugriff hat, er mehrere Leseaufträge erteilt, oder etwas auf die Schnittstelle schreiben will, ohne das Ende des Schreibens abwarten zu wollen. An der Stelle im Programmtext angekommen, an der der Vorgang angestoßen werden soll, wird die Funktion *start_thread* aufgerufen.

```
pid = start_thread( pim_client, (void*) str_p );
```

Das erste Argument ist ein Funktionspointer auf die Funktion *pim_client*, die sich in der Datei *pimclient.h* befindet; dies ist die Hauptfunktion des Clients, der den Auftrag an den Server weiterleitet. Dieser Auftrag wird in Form eines Zeigers auf eine Struktur vom Typ *pim_args* als zweites Argument übergeben.

```
/*
  Argumente, die clients von
  user-prozessen uebergeben bekommen
```

```
*/
struct pim_args {
    int task;
    char *buffer;
    long count;
    int how;
};
```

Das erste Element definiert den Auftrag, also Lesen (*TASK_READ*) oder Schreiben (*TASK_WRITE*), das zweite den Speicherbereich, in den gelesen oder aus dem geschrieben werden soll, das dritte gibt die Anzahl der zu übertragenen Bytes an, und das vierte definiert die Art und Weise der Übertragung, ob synchron (*HOW_SYNC*) oder asynchron (*HOW_ASYNC*). Der Rückgabewert ist die PID des Clones.

Hiernach wird der Auftrag bearbeitet, und im User-Programm kann mit anderen Dingen fortgefahren werden oder es wird auf den bereits empfangenen Daten (beim Lesen) gearbeitet. Das User-Programm kann den Stand der Übermittlung anhand des Wertes der Variablen *pim_client_read_data* überprüfen und z.B. als Grenzwert in einer for-Schleife verwenden.

```
while(!waitpid(pid, PIM_CLIENT_RET_VAL, WNOHANG)){
/* hier koennte er rechnen by async read */
   if ((already_printed  < pim_client_read_data) &&
       !already_painting){
       ...
       }
   }
```

Die Funktion *pim_client*, welche nun im Clone des User-Prozesses ausgeführt wird, legt zunächst eine lokale Kopie der Auftragsstruktur an. Durch den folgenden Aufruf der Funktion *init_client* wird der Key der Message Queue des Servers aus der Datei */var/pim/server.key* gelesen und eine eigene Queue für den Client angelegt. Im Rest der *pim_client*-Funktion wird zwischen den grundsätzlichen Betriebsmodi *Lesen* und *Schreiben* unterschieden.

```
    /* Falls das Lesen die Aufgabe war... */
    if(my_args->task == TASK_READ)
    {
        ...
    } /* TASK_READ zuende */

    /*  ...oder das Schreiben. */
    else if(my_args->task == TASK_WRITE)
    {
        ...
    }  /* TASK_WRITE zuende */
```

Im Lesezweig wird zuerst der Auftrag durch die Funktion *request_to_server* an den Server übermittelt.

```
request_to_server(my_args->task,
                  my_args->count,
                  my_args->how, 0);
```

Die Argumente sind die Aufgabe, die Anzahl der zu verarbeitenden Bytes, die Art der Verarbeitung und ein Key eines Shared Memory Bereiches, der aber nur beim Schreiben benutzt wird. Innerhalb dieser Funktion werden die benötigten Daten in die entsprechenden Felder einer *request_service*-Struktur kopiert.

```
struct request_service* req;
req = (struct request_service *)
      malloc(sizeof(struct request_service));

...
req->type = TYPE_REQUEST;
req->client_msq_key = (key_t)client_key;
req->task = task;
req->bytes = count;
req->write_buffer_shm_key = shm_key;
req->how = my_how;
```

Anschließend wird dieser *Request* an den Server gesendet.

```
msgsnd(server_key, req,
       (sizeof(struct request_service) - sizeof(long)), 0);
```

Nach der Abgabe des Auftrages an den Server über eine Message Queue, wartet der Client auf eine Antwort seitens des Servers.

```
wait_for_server_reply(my_args);
```

In dieser Funktion wird beim Empfang einer Nachricht eine Struktur vom Typ *answer_from_server* ausgefüllt. Es gibt hier verschiedene Typen einer Antwort. Eine Antwort *TYPE_INTERIMS_ANSWER* signalisiert, daß ein Auftrag vom Servlet mit der Art *HOW_ASYNC*, dies ist asynchron, bearbeitet und das Servlet über den Server einen Zwischenbericht an den Client liefert. Dieser erhält so den Stand der Verarbeitung und kann gegebenfalls die schon gelesenen Daten in den Adressraum des User-Prozesses kopieren.

```
memcpy(((void *)my_args->buffer + offset),
       ((const void *)shm_buf + offset), (size_t)amount_bytes);
```

Der Wert von *pim_client_read_data* wird dementsprechend erhöht. Als letztes wird hier auf weitere Messages gewartet. Dieser Typ einer Antwort ist der einzige, der häufiger auftreten kann.

Erhält der Client eine Antwort vom Typ *TYPE_ANSWER*, bedeutet dies, sofern das Feld *shm_key* weder den Wert *ERROR_RESSOURCE*, noch den Wert *ERROR_SERVER* hat, daß der Auftrag problemlos bearbeitet und die Bearbeitung abgeschlossen wurde. *ERROR_RESSOURCE* bedeutet, daß der Server den Auftrag zurückgestellt hat, da bereits ein Maximum von Clients bedient wird. An dieser Stelle wartet der Client auf eine weitere Antwort. Ist die nächste vom Typ *ERROR_RESSOURCE* oder *ERROR_SERVER*, führt das zum Beenden des Clients mit vorherigem Abmelden vom Server und Aufräumen. Der User-Prozeß, zu dem dieser Client gehört, wird im Moment noch nicht vom fehlerhaften Verlauf benachrichtigt. Die Meldung vom Typ *ERROR_SERVER* wird vom Server versendet und deutet auf nicht weiter spezifizierten Fehler.

Nachfolgend ist die Struktur der zuletzt beschriebenen Funktion dargestellt.

```
void wait_for_server_reply(struct pim_args *my_args)
{
    ...
    answer = (struct answer_from_server *)
             malloc(sizeof(struct answer_from_server));
    msgrcv(client_key, answer,
           (sizeof(long) + sizeof(struct answer_from_server)),
           0, 0);

    switch(answer->type)
    {
    case TYPE_INTERIMS_ANSWER:
    {
        ...
        while(answer->type == TYPE_INTERIMS_ANSWER)
        {
            ...
            memcpy(((void *)my_args->buffer + offset),
                   ((const void *)shm_buf + offset),
                   (size_t)amount_bytes);

            ...
            msgrcv(client_key, answer,
                   (sizeof(long) +
                    sizeof(struct answer_from_server)), 0, 0);
        }
        ...
    }

    case TYPE_ANSWER:
    {
        if(answer->shm_key == ERROR_RESSOURCE)
        {
            /** Fehler!
             ** Warte bis Du dran bist...
             **/

            ...
            msgrcv(client_key, answer,
                   (sizeof(long) +
                    sizeof(struct answer_from_server)),
                    0, 0);

            if(answer->shm_key == ERROR_RESSOURCE){
                /** schon wieder nicht => ENDE **/
                ...
            }
            else if(answer->shm_key == ERROR_SERVER)
            {
                ...
            }
```

```
        }
        else if(answer->shm_key == ERROR_SERVER)
        {
            ...
        }
    }
    default:
    {
        break;
    }

    }
}
```

45.1.2 Serverprogramm

Das Serverprogramm besteht im wesentlichen aus einer *while(1)*-Endlosschleife. Vor dieser Schleife wird lediglich eine Message Queue allokiert, und dessen Key in der Datei */var/pim/server.key* zur weiteren Benutzung durch die Clients vermerkt.

Bei jedem Schleifendurchlauf wird zunächst auf eine Nachricht gewartet; Quelle dieser können zum einen die Clients sein, die eine Anfrage an den Server stellen, oder sich nach verrichteter Arbeit ordnungsgemäß abmelden, zum anderen kann es auch eine Nachricht vom Kontrollprogramm sein, welches dem Server mitteilen will, daß es sich beenden möchte. Die Nachricht wird in einer Kopie der Struktur *request_service* gesichert und anhand des Elementes *type* wird entschieden, wie nun weiter zu verfahren ist.

```
msgrcv(own_key, msg_buffer,
       (sizeof(long) + sizeof(struct request_service)), 0, 0);

while(1)
{
        /* main switch-statement */
        switch(msg_buffer->type)
        {
                case TYPE_REQUEST :
                {
                        ...
                }
                case TYPE_UNQUEUE :
                {
                        ...
                }
                case TYPE_RWREADY:
                {
                        ...
                }
                case TYPE_READ_INTERIM:
                {
                        ...
                }
```

```
            case TYPE_CONTROL:
            {
                    ...
            }
            case TYPE_READER_READY:
            {
                    ...
            }
            default:
                break;
        }

        free((void *)answer);
        free((void *)msg_buffer);

        if(waitpid(0, servlet_exit_status, WNOHANG));

}
```

Wie man hier sieht, sind sechs verschiedene Nachrichtentypen zu verarbeiten:

TYPE_REQUEST Dieser Typ besagt, daß ein Client eine neue Anfrage stellt. Dabei ist beim Server zu prüfen, ob die maximale Anzahl an bedienten Clients erreicht ist; dies geschieht durch Abfrage und Setzen der Variablen *full_runqueue*. Hat diese den Wert *PIM_CLIENT_MAX* erreicht, bedeutet dies, der Server bedient genau soviele Clients, alle weiteren Anfragen werden in eine Warteschlange *wql* gestellt.
wql ist vom Typ *struct wait_queue_list* und bietet zusammen mit der Struktur *wait_queue_node* das Grundgerüst für eine doppelt verkettete Liste. Diese Art der Verwaltung ist in dieser Phase der Programmentwicklung ausreichend.
Daraufhin erhält der die Anfrage stellende Client eine Nachricht vom Typ *TYPE_ANSWER* und dem Wert -1 im Element *shm_key*; das weitere Verfahren des Clients wurde bereits oben erwähnt. Sollte *PIM_CLIENT_MAX* noch nicht erreicht sein, wird die Funktion *start_task* aufgerufen, mit der aktuellen Anfrage als Argument (diese ist vom Typ *request_service*).

TYPE_UNQUEUE Mit dieser Nachricht möchte sich ein Client regulär nach Beendigung seiner Aufgabe vom Server abmelden. Nach Abschicken dieser Nachricht beendet sich der Client; der Server löscht dessen Eintrag in der RunQueue.
Die RunQueue – im Programmtext als *service_queue* bezeichnet – ist ein Array vom Typ *struct request_service* und hat PIM_CLIENT_MAX Elemente. Ein neu bearbeiteter Auftrag wird an die erste freie Stelle, von 0 an gezählt, gestellt. Zum Löschen eines Auftrages aus der RunQueue wird in diesem Array nach einem Eintrag gesucht, dessen Message Queue-Key mit dem des Clients der diese Nachricht sendet übereinstimmt. Bei Auffinden dieses Eintrages wird der dazugehörige Shared Memory Bereich gelöscht und danach der Eintrag im Array. Anschließend wird geprüft, ob es einen Eintrag in der Warteschlange gibt und dieser gegebenenfalls abgearbeitet.

TYPE_RWREADY Diese Nachricht erhält der Server vom Servlet. Sie bedeutet, daß das

Servlet seinen Auftrag abgearbeitet hat; diese Nachricht wird an den entsprechenden Client weitergereicht.

TYPE_INTERIMS_ANSWER Diese Nachricht wird ebenfalls an den dazugehörigen Client weitergeleitet. Sie wird von einem Servlet gesendet, das einen Leseauftrag von der Art *HOW_ASYNC* ausführt. Sie bedeutet, daß ein weiteres Datenpaket komplett empfangen wurde.

TYPE_CONTROL Der Empfang dieser Nachricht veranlaßt den Server, sich nach Freigabe aller Ressourcen zu beenden.

TYPE_READER_READY Der Prozeß, der das tatsächliche Lesen von der Schnittstelle bearbeitet, und dabei die empfangenen Daten in einen Shared Memory Bereich schreibt, auf den die Servlets zugreifen können, sendet diese Nachricht, um anzuzeigen, daß das letzte Byte gesendet und empfangen wurde. Dieser Prozeß beendet sich dann.

45.1.3 Demoapplikation

In der Demoapplikation wird von einem Rechner ein Bild über die serielle Schnittstelle auf einen anderen Rechner übertragen.

Auf dem emfangenden Rechner laufen sowohl der PiM-Server, als auch die Clients, welche die Daten erhalten und verarbeiten, und der sendende Rechner schreibt einfach die Bilddaten auf die serielle Schnittstelle.

Der Server wird mittels des *start_server.sh*-Scriptes gestartet, selbstverständlich vor jeglicher Benutzung des PiM-Systems. Zu Beginn der Übertragung werden zuerst die Clients auf dem Empfänger mit dem *start_clients.sh*-Script gestartet; beide Shell-Scripte befinden sich in dem Verzeichnis
pim/bmp_demo/qt. Es werden vier Clients gestartet, wo je einer den roten, grünen und blauen Anteil des Bildes beachtet. Ein weiterer Client interpretiert die Daten als Graustufenbild. Die Clients laufen vollkommen unabhängig voneinander ab, so daß es keinerlei Kommunikation oder Datenaustausch zwischen ihnen gibt. Die Quelldateien zu den Clients befinden sich in den entsprechenden Verzeichnissen unterhalb von *pim/bmp_demo/qt*.

Alle Clients zeigen ihre Interpretation der empfangenen Daten in einem Fenster. Das Bild wird auf die vorher beschriebene Weise noch, während die Daten empfangen werden, erstellt. Neben der Visualisierung kann man sich aber noch andere Anwendung vorstellen, wie zum Beispiel die Berechnung eines Histogramms, oder sonstige Algorithmen, die man auf derartige Daten anwenden möchte.

Die Client-Programme starten eine Instanz der Klasse *PimWidget*, welche in ihrem Konstruktor neben einigen Variableninitialisierungen die Methode *getbytes()* aufruft, die das eigentliche PiM-System in Gang setzt. Die weitere Verarbeitung geht wie bereits beschrieben vonstatten. In der hier enthaltenen *while()*-Schleife wird die Variable *pim_client_read_data* dazu benutzt, dem Widget mitzuteilen, wieviele Pixel bereits dargestellt werden dürfen.

```
while( !waitpid ( pid, PIM_CLIENT_RET_VAL, WNOHANG )){
/* hier koennte er rechnen by async read */

    if ((already_printed  < pim_client_read_data) &&
```

```
          !already_painting){
        to_draw = pim_client_read_data;
        if (pim_client_read_data > 184000) completed = 1;
        repaint((bool)FALSE);
    }
}
```

45.2 Probleme

Probleme traten sowohl bei der Umsetzung der Spezifikation in Quellcode auf, als auch bei der Erstellung der Demoapplikation.

45.2.1 Umsetzung der Spezifikation

Wie weiter oben erwähnt, ergaben sich Probleme dadurch, daß wir für das eigentliche System unwichtige Teile nicht spezifizierten, sondern an dieser Stelle starke Vereinfachungen benutzten.

Das Lesen vom Gerät sollte in der Demonstration von der seriellen Schnittstelle geschehen. Um die Funktionsweise der Schnittstelle, sowie deren Programmierung kennenzulernen, schrieben wir kleine Programme, die lediglich Daten von Rechner *A* nach Rechner *B* transprotierten. Zu Anfang waren die Daten Zahlen von *1* bis *n*. Diese sollten beim Empfänger ankommen und außerdem in der selben Reihenfolge, wie sie versendet wurden.

Dies war in der ersten Version nicht der Fall, was sich aber anhand von Handbüchern korrigieren ließ. Das eigentliche Problem, auf das wir stießen, war die Tatsache, daß der Sender die Einstellungsparameter der seriellen Schnittstelle willkürlich zurücksetzte.

Willkürlich war dabei der Zeitpunkt, zu dem dieses geschah; er war nicht vorhersagbar. Da uns die Zeit davon lief, begnügten wir uns viel später mit der Lösung, die Einstellungen vor jeder Sendung mit Hilfe des Linux-Kommandos *setserial* zu setzen. Unter anderem mußte regelmäßig die Geschwindigkeit auf *spd_vhi* gesetzt werden, was einer Übertragungsgeschwindigkeit von 115200bps entspricht. Selbstverständlich müssen die Schnittstellen der beteiligten Rechner identische Einstellungen aufweisen.

Man sollte dabei auch beachten, daß die maximalen Kabellängen bei entsprechenden Geschwindigkeiten nicht überschrittten werden. Die maximale Länge für 115200 bps ist 2 Meter (bei 57600 bps 5 Meter, bei 38400 bps 10 Meter). Überschreitungen können die Übertragungsqualität beeinträchtigen.

45.2.2 Erstellung der Demoapplikation

Die Schwierigkeit in diesem Bereich war die Wahl des X-Toolkits, mit dem wir die übertragenen Bilder auf der Benutzeroberfläche darstellen wollten. In die nähere Auswahl gelangten zunächst *Tcl/TK*, dann auch *GTk+*. Die Wahl fiel auf *Qt* aus dem einfachen Grunde, daß uns die Zeit fehlte, uns in die anderen Umgebungen einzuarbeiten, und wir hier bereits Vorkenntnisse hatten.

46. Ausblick

Natürlich bleibt noch einiges zu tun. Um unsere Arbeit am PiM–Projekt beenden zu können, sind zwei größere Arbeitspakete notwendig:

46.1 Die Scheduler–Erweiterung

Das Standard–Verfahren zum Scheduling von Prozessen unter Linux ist für unsere Zwecke nicht ausreichend, weil die Server/Servlet–Prozesse oft genug einfach nichts zu tun haben und trotzdem Ressourcen zugeteilt bekommen. Insbesondere ist unser Ziel, solche Servlets besonders zu behandeln, die kurzfristig abweichende Ressourcenanforderungen stellen. Ein Verfahren, dass uns geeignet erscheint, faires Scheduling im PiM–System zu ermöglichen, stützt sich auf dem *Lottery–Scheduling* ab.

46.1.1 Das Lottery–Scheduling im allgemeinen

Die Grundlagenarbeit über Lottery–Scheduling ist [WW94][1]; mittlerweile gibt es neben der Referenz–Implementierung für den MACH–Kernel eine Implementierung dieses Scheduling–Verfahrens für freeBSD[PMG99], die zeigt, daß das Verfahren nicht nur theoretische Relevanz hat, sondern auch im echten Leben eingesetzt werden kann.

Das *Lottery–Scheduling* hat ein einfaches Prinzip: jedem Prozess wird, je nach Priorität, eine Anzahl Lose zugeteilt. Beim Umverteilen der verfügbaren Ressourcen wird ein Los gezogen – der Prozess, dessen Los gezogen wurde, bekommt die nächste Zeitscheibe.

Dabei sind einige dynamische Eingenschaften erwähnenswert, die übliche Schedulingverfahren mit Priorisierung nicht haben:

Tickets können immer dann umverteilt werden, wenn ein Client blockiert ist, weil er auf eine Ressource warten muß. Das ist entweder der Fall, wenn ein anderer Client eine Ressource blockiert, weil es sich um einen geschützten kritischen Abschnitt handelt, oder wenn ein Server des Clients eine Anfrage noch nicht vollständig abgearbeitet hat.

Das Lottery–Scheduling ermöglicht also eine inhärente dynamische Priorisierung von Prozessgruppen, die eng aufeinander angewiesen sind.

46.1.2 Das Lottery–Scheduling im PiM–System

Da wir den eigentlichen Linux–Kern sowenig wie möglich verändern wollen, müssen wir den herkömmlichen Scheduler in seiner Funktionalität bewahren. Deswegen ist unsere Absicht, den Scheduler um ein weiteres Verfahren zu erweitern, dass aber nur unseren PiM–Prozessen zur Verfügung stehen soll.

[1] erweitert in [Wal95]

Der erste Schritt auf diesem Weg ist, den Scheduler um eine neue Prioritätsklasse `SCHED_LOTTERY` zu erweitern, und dann natürlich, die entsprechenden tools zur Verwaltung von Tickets zu schreiben. Dabei könnte uns die Implementierung des Verfahrens für FreeBSD[2]helfen .

46.2 PiM–Prozesse mit clone() aus dem Kernel starten

Während der Entwicklung des PiM–Systems waren wir natürlich davon ausgegangen, dass wir die PiM–Prozesse einfach mit `clone()` starten können. Unglücklicherweise ergibt sich dabei aber die Schwierigkeit, daß wir nicht herausgefunden haben, wie die Rücksprungadresse zum gerade laufenden Prozeß zu ermitteln ist. Also haben wir kurzerhand `clone()` als User Space–Aufruf nachgebaut.

Das ist für Testzwecke zwar erst einmal ausreichend, aber damit verspielen wir natürlich unseren Zeitgewinn. Die Nutzung vom systeminternen `clone()` zu ermöglichen ist also mit hoher Priorität voranzutreiben.

46.3 Weitere Arbeitsschritte

Auf der Arbeit anderer Entwickler aufsetzend könnte man versuchen, zur Laufzeit eine Lastanalyse zu machen und dynamische Prozeßverteilungsverfahren einzusetzen. Es gibt auch schon entsprechende Patches für den Linux–Kern, die Cache Misses und andere wichtige Kenngrößen messen[3] und so die Grundlage für solche Erweiterungen bilden. Das ist allerdings sehr ferne Zukunftsmusik; vielleicht findet sich aber noch jemand, der Interesse an dieser Aufgabe hat.

46.4 Fazit

Im Laufe des Projektes haben wir das Vorgehen zum Entwickeln von Kernel-Erweiterungen kennengelernt. Kurz gesagt: Man geht einen steinigen Weg.

Mit dem Schwerpunkt auf Symmetrischem Multiprocessing haben wir uns ein Gebiet ausgesucht, für das es keine kochrezeptartigen Musterlösungen gibt. Vielmehr kann und muß hier noch jede Menge Grundlagenarbeit geleistet werden, bis dieses Thema in die "allgemeine Kernel-Lehre" übergeht.

Besondere Schwierigkeiten hatten wir damit, Dokumentation bezüglich der softwareseitigen Implementierung zu finden – Linux war zur Zeit des Projektes in dieser Hinsicht so etwas wie ein "Moving Target", und die Entwickler waren zu beschäftigt, die Kern–Struktur zu renovieren, als daß sie uns Neulinge unterstützen konnten.

Gegen Ende des dritten Projektquartals mußten wir dann den Entschluß fassen, unsere Zielsetzung zu schrumpfen und mit einer spezialisierten Lösung zu beginnen, weil wir sonst unsere eigentlichen Ideen nie umgesetzt hätten. Gleichzeitig haben wir natürlich versucht, weiterhin eine allgemein erweiterbare Konzeption aufzustellen.

[2] `http://www.cs.cmu.edu/~dpetrou/papers/freebsd_lottery_writeup98.ps`, bzw `http://www.cs.cmu.edu/~dpetrou/code/freebsd_lottery_code.tar.gz`

[3] Allerdings nur auf i386–Systemen

Die Arbeit mit CSP und FDR hat die Entwicklung entscheidend geprägt. Zwar war es zeitaufwendig, die Ideen formal aufzuschreiben und zu testen, aber letztendlich konnten wir so sehen, daß wir hinsichtlich der geprüften Problematiken keine Denkfehler gemacht hatten.

Einem Folgeprojekt würden wir empfehlen, die Ideen und Vorhaben so klein wie möglich anzusetzen. Erweitern oder vergrößern läßt sich ein Aufgabengebiet immer noch, und an Arbeit wird es beim Testen und Verifizieren “lebendiger” Software-Stücke niemals mangeln.

Auch sollte man sich frühzeitig auf eine Codebasis festlegen, auf der gearbeitet wird. Oft ist es einfacher, ein fertiges Stück Erweiterung auf eine neue Version des Kerns zu übertragen, als die Entwicklungsgrundlage unter einem unfertigen Projekt auszutauschen[4].

Die Erweiterung des Kerns oder auch die Reimplementierung von Teilen des Kerns sind anspruchsvolle Programmieraufgaben, die auch neue Strategien bei der Implementierung fordern. Selten genug hat man die Möglichkeit, einen laufenden Kernel zu debuggen, obwohl auch hier die Enwicklung von kdb auf der einen und virtuellen Computern auf der anderen Seite bald neue Wege öffnen werden, bzw. können.

[4] das ist nämlich so gut wie unmöglich!

Teil VII

Fazit

Es gab viel zu tun.

Wie schon im Vorwort der Betreuenden erwähnt, zeichnete sich das Projekt durch lockeres Projekt–Management und hohe Motivation aller Teilnehmenden – vor allem kurz vor Deadlines – aus.

Natürlich gibt es auch weiterhin viel zu tun, aber die Arbeit wird sich jetzt nur noch vereinzelt in Diplomarbeiten und den Köpfen der einzelnen Projektteilnehmer weiterentwickeln.

Ein paar Worte zum Projektnamen noch einmal, ähnlich der Einleitung:

Linux ist mittlerweile zu einer Informatiker–Mode–Erscheinung geworden und auch nichts wirklich neues mehr.

Verifikation und Test sind aber immer noch spannende Forschungsgebiete. Nachfolgenden Projekten raten wir, mit der echten Arbeit eher anzufangen, denn die fünf Prozent Restarbeit nehmen immer noch 95 Prozent der Zeit in Anspruch…

Enterprise Unsere Aufgabe, unser Abenteuer ist nominell schon seit langer Zeit beendet. Aber man könnte fast denken, daß die lange Zeit bis zur Fertigstellung des Projektberichts in einer Art Nostalgie, am Festhalten an Prinzipien, die aus der Projektarbeit entstanden sind, begründet ist.

Für viele von uns endet die Zeit als Studenten an der Uni bald, einige haben ihre Karriere schon in die freie Wirtschaft verlegt. Bleibt zu hoffen, daß das Projekt nicht nur eine persönliche Bereicherung war, sondern auch inhaltlich zum Werdegang unser aller beiträgt.

Only future can tell…

Anhang A. ACL: Testprozedur

A.1 Allgemein

Jeder Test, der eine ACL verwendet, ist mit einer ACL aus einem Satz von validen ACLs durchzuführen. Jeder Test, der einen *Mode* verwendet, ist mit allen Kombinationen, die einen *Mode* ergeben, durchzuführen. Alle Ziel Dateien müssen in dem zu testenden Dateisystem befinden.

Aus der Spezifikation der Korrektheit geht hervor, ob der Test erfolgreich ist.

Folgende Werkzeuge werden zum Testen benutzt:

- `ls` zum Lesen des *Mode* einer Datei.
- `getfacl` (`-d`) zum Lesen der Access- (Default-) ACL.
- `setfacl` (`-d`) zum Setzen der Access- (Default-) ACL.
- `chmod` zum Setzen des *Mode* einer Datei.

A.1.1 POSIX.1e Draft 17

> Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API), Amendment e: Protection, Audit and Control Interfaces [C Language], Draft 17

Dieser Standard definiert sicherheitsrelevante Programm Schnittstelle von offenen Systemen, insbesondere die Schnittstelle zur Manipulation von ACLs.

A.1.2 Abschnitt 2, Zeile 238-243

If the process possesses appropriate privilege:

- *If read, write or directory search permission is requested, access is granted.*
- *If execute permission is requested, access is granted if execute permission is specified in at least one ACL entry; otherwise, access is denied.*

Test Spezifikation:

Ein Prozess mit UID 0 versucht, aus einer Datei mit ACL zu lesen.

Korrektheit:

Erfolg mit allen Test ACLs.

Test Spezifikation:

Ein Prozess mit UID 0 versucht, in eine Datei mit ACL zu schreiben.

Korrektheit:

Erfolg mit allen Test ACLs.

Test Spezifikation:

Ein Prozess mit UID 0 versucht, eine Datei mit ACL auszuführen.

Korrektheit:

Die Datei muß als auführbar erkannt werden, wenn mindestens ein ACL Eintrag das x permission bit gesetzt hat.

A.1.3 Abschnitt 3, Zeile 4-6

If `_POSIX_ACL` *is defined, the child process shall have its own copy of any ACL pointers and ACL entry descriptors in the parent, and any ACL working storage to which they refer.*

Test Spezifikation:

Ein Prozeß liest die ACL einer Datei und kreiert darauf einen Kindprozeß. Der Kindprozeß kopiert den Speicher und signalisiert dem Vaterprozeß. Daraufhin überschreibt der Vaterporzeß den Speicher mit der ACL und signalisiert dann seinem Kindprozeß. Der Kindprozeß vergleicht darauf die Kopie der ACL und die Original ACL.

Korrektheit:

Kopie und Original ACL sind identisch.

A.1.4 Abschnitt 5, Zeile 2 ff.

Open a File: If `{_POSIX_ACL}` *is defined and* `{_POSIX_ACL_EXTENDED}` *is in effect for the directory in which the file is being created (the "containing directory") and said directory has a default ACL, the following actions shall be performed:*

1. *The default ACL of the containing directory is copied to the access ACL of the new file.*
2. *Both the* `ACL_USER_OBJ` *ACL entry permission bits and the file owner class permission bits of the access ACL are set to the intersection of the default ACL's* `ACL_USER_OBJ` *permission bits and the file owner class permission bits in mode. The action taken for any implementation-defined permissions that may be in the* `ACL_USER_OBJ` *entry shall be implementation-defined.*
3. *If the default ACL does not contain an* `ACL_MASK` *entry, both the* `ACL_GROUP_OBJ` *ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the default ACL's* `ACL_GROUP_OBJ` *permission bits and the file group class permission bits in mode. The action taken for any implementation-defined permissions that may be in the* `ACL_GROUP_OBJ` *entry shall be implementation-defined.*

4. *If the default ACL contains an* `ACL_MASK` *entry, both the* `ACL_MASK` *ACL entry permission bits and the file group class per mission bits of the access ACL are set to the intersection of the default ACL's* `ACL_MASK` *permission bits and the file group class permission bits in mode. The action taken for any implementation-defined permissions that may be in the* `ACL_MASK` *entry shall be implementation-defined.*
5. *Both the* `ACL_OTHER` *ACL entry permission bits and the file other class permission bits of the access ACL are set to the intersection of the default ACL's* `ACL_OTHER` *permission bits and the file other class permission bits in mode. The action taken for any implementation-defined permissions that may be in the* `ACL_OTHER` *entry shall be implementation-defined.*

Test Spezifikation:

Eine reguläre Datei wird in einem Verzeichnis mit Default-ACL unter Angabe eines Mode angelegt.

Korrektheit:

Mit Ausnahme von ACL_USER_OBJ, ACL_MASK, ACL_GROUP_OBJ und ACL_OTHER_OBJ muß die Default-ACL als Access-ACL übernommen werden, die anderen Einträge werden als Durchschnitte mit dem Mode übernommen.

A.1.5 Abschnitt 5, Zeile 136 ff.

Make a Directory:

Test Spezifikation:

Verzeichnis in einem Verzeichnis mit Default-ACL und Angabe eines Mode anlegen.

Korrektheit:

Mit Ausnahme von ACL_USER_OBJ, ACL_MASK, ACL_GROUP_OBJ und ACL_OTHER_OBJ muß die Default-ACL als Access-ACL übernommen werden, die anderen Einträge werden ale Durchschnitte mit dem Mode übernommen. Die Default-ACL muß identisch als Default-ACL übernommen werden.

A.1.6 Abschnitt 5, Zeile 143 ff.

Make a FIFO Special File:
If `{_POSIX_ACL}` *is defined and* `{_POSIX_ACL_EXTENDED}` *is in effect for the directory in which the FIFO is being created (the c̈ontaining directory)̈ and said directory has a default ACL, the following actions shall be performed: The default ACL of the containing directory is copied to the access ACL of the new FIFO.*

Test Spezifikation:

FIFO Datei in einem Verzeichnis mit Default-ACL und Angabe eines Mode anlegen.

Korrektheit:

Mit Ausnahme von ACL_USER_OBJ, ACL_MASK, ACL_GROUP_OBJ und ACL_OTHER_OBJ muß die Default-ACL als Access-ACL übernommen werden, die anderen Einträge werden ale Durschnitte mit dem Mode übernommen.

A.1.7 Abschnitt 5, Zeile 226 ff.

Get File Status: If `{_POSIX_ACL}` *is defined, and* `{_POSIX_ACL_EXTENDED}` *is in effect for the pathname, and the access ACL contains an* `ACL_MASK` *entry, then the file group class permission bits represent the* `ACL_MASK` *access ACL entry file permission bits. If* `{_POSIX_ACL}` *is defined, and* `{_POSIX_ACL_EXTENDED}` *is in effect for the pathname, and the access ACL does not contain an* `ACL_MASK` *entry, then the file group class permission bits represent the* `ACL_GROUP_OBJ` *access ACL entry file permission bits.*

Test Spezifikation:

Setzen einer Access-ACL einer Datei.

Korrektheit:

Der Mode der Datei muß den Einträgen der ACL entsprechen: user class muß dem ACL_USER_OBJ Eintrag entsprechen, group class muß dem ACL_MASK Eintrag entsprechen, bzw. bei dessen Fehlen dem ACL_GROUP_OBJ Eintrag.

A.1.8 Abschnitt 5, Zeile 252 ff.

Change File Modes: If `{_POSIX_ACL}` *is defined and* `{_POSIX_ACL_EXTENDED}` *is in effect for the pathname, then the following actions shall be performed.*

1. *The* `ACL_USER_OBJ` *access ACL entry permission bits shall be set equal to the file owner class permission bits.*
2. *If an* `ACL_MASK` *entry is not present in the access ACL, then the* `ACL_GROUP_OBJ` *access ACL entry permission bits shall be set equal to the file group class permission bits. Otherwise, the* `ACL_MASK` *access ACL entry permission bits shall be set equal to the file group class per mission bits, and the* `ACL_GROUP_OBJ` *access ACL entry permission bits shall remain unchanged.*
3. *The* `ACL_OTHER` *access ACL entry permission bits shall be set equal to the file other class permission bits.*

Test Spezifikation:

Setzen eines Mode mit chmod.

Korrektheit:

Der Mode der Datei muß den Einträgen der ACL entsprechen: user class muß dem ACL_USER_OBJ Eintrag entsprechen, group class muß dem ACL_MASK Eintrag entsprechen, bzw. bei dessen Fehlen dem ACL_GROUP_OBJ Eintrag.

A.1.9 POSIX.2c Draft 17

> Portable Operating System Interface (POSIX), Part 2: Shell and Utilities, Amendment c: Protection and Control Interfaces, Draft 17

Dieser Standard definiert sicherheitsrelvante System Programme von offenen Systemen, insbesondere die Programme zur Manipulation von ACLs.

A.1.10 Abschnitt 4, Zeile 8 ff.

cp – Copy files: If `{_POSIX_ACL}` *is defined and* `{_POSIX_ACL_EXTENDED}` *is in effect for the destination file, the ACLs. If this fails for any reason, cp shall write a diagnostic message to the standard error, do nothing more with the current* `source_file` *and go on with any remaining files.*

Test Spezifikation:

Eine reguläre Datei wird mit `cp -p` kopieren.

Korrektheit:

Die Access-ACL muß erhalten bleiben.

Test Spezifikation:

Ein Verzeichnis wird mit `cp -p` kopieren.

Korrektheit:

Die Access-ACL und die Default-ACL muß erhalten bleiben.

A.1.11 Abschnitt 4, Zeile 14 ff.

ls – List directory contents:
If `_POSIX_ACL` *is defined and* `_POSIX_ACL_EXTENDED` *is in effect for the file, then the <optional alternate access method flag> shall be a plus sign ("+").*

Test Spezifikation:

Eine Datei mit (ohne) extended ACL mit `ls` anzeigen.

Korrektheit:

Ein (Kein) + für die ACL muß beim *Mode* angezeigt werden.

A.1.12 Abschnitt 4, Zeile 19 ff.

mv – Move files: If `_POSIX_ACL` *is defined and* `{_POSIX_ACL_EXTENDED}` *is in effect for* `dest_file`*, then the ACLs associated with the* `dest_file` *shall reflect the ACLs associated with the* `source_file`*. If this fails for any reason, mv shall write a diagnostic message to the standard error, do nothing more with the current* `source_file` *and go on with any remaining files.*

Test Spezifikation:

Eine reguläre Datei wird mit `mv` verschoben.

Korrektheit:

Die Access-ACL muß erhalten bleiben.

Test Spezifikation:

Ein Verzeichnis wird mit `mv` verschoben.

Korrektheit:

Die Access-ACL und die Default-ACL muß erhalten bleiben.

Anhang B. ACL: Fehler im Prototyp

Hier folgt nun eine Auflistung der Fehler, die wir im Prototypen fanden.

1. - **Beschreibung** `getacl` und `setacl` brechen immer mit der Fehlermeldung `Invalid argument` ab:

     ```
     prj@hast:/home/prj/test > getacl /acl/dat1
     /acl/dat1: Invalid argument
     prj@hast:/home/prj/test > setacl -u u:prj:rwx /acl/dat1
     ...
     acl_get_file: /acl/dat1: Invalid argument
     ```

   - **Ursache** `sys_acl_ctl()` wird nicht richtig aufgerufen. In der Datei `./aclutils-0.1/lib/ctl.c` wird `acl_ctl()` mit der Systemcall Nr. 172 aufgerufen. Dies ist nicht die Aufrufnummer in dem aktuellen Kernel:

     ```
     #include <syscall.h>

     #ifndef SYS_acl_ctl
     #define SYS_acl_ctl 172
     #endif

     _syscall4 (int, acl_ctl, int, operation, const void *,
               file, int, acl_type, void *, data)
     ```

   - **Lösung** Das Problem wurde gelöst indem die Headerdatei `unistd.h` des aktuellen Kernel eingebunden wurde und dort ein entsprechender Eintrag hinzugefügt wurde.
     Änderung in `./aclutils-0.1/lib/ctl.c`:

     ```
     #include <syscall.h>
     #include <linux/unistd.h>

     _syscall4 (int, acl_ctl, int, operation, const void *,
               file, int, acl_type, void *, data)
     ```

     Änderung in `./include/include/asm-i386/unistd.h`

     ```
     #define __NR_acl_ctl            184
     ```

     Hierbei richtet sich die Aufrufnummer nach der entsprechenden Kernelversion und muß mit der Position des Eintrags für `sys_acl_ctl` in `./arch/i386/kernel/entry.S` konsistent sein.

2. - **Beschreibung** Für Gruppeneinträge wird immer Gruppe `root` eingetragen.

     ```
     prj@hast:/home/prj/test > setacl -u g:g1:rwx /acl/dat1
     prj@hast:/home/prj/test > getacl -l /acl/dat1
     # file: /acl/dat1
     # owner: prj
     # group: users
     user::rw-
     group::r--
     group:root:rwx
     other::r--
     ```

- **Ursache** Die Funktion `add_entry(...)`, definiert in der Datei `./aclutils-0.1/lib/from_text.c`, dient dazu ACL-Entrys von textueller Form in den Datentyp `acl_entry_t` umzuwandeln und in eine ACL einzufügen. Dort wurde das Qualifier-Feld des acl-entries folgendermaßen gesetzt:

```
if (type == ACL_USER)
  if (acl_set_qualifier (acle, (const void *) &uid) != 0)
    return -1;
else if (type == ACL_GROUP)
  if (acl_set_qualifier (acle, (const void *) &gid) != 0)
    return -1;
```

  Auch wenn die Einrückung des Codes es einen glauben läßt, kann der Qualifier für einen Eintrag vom Typ `ACL_GROUP` nie gesetzt werden, da sich das `else`-Statement auf das `if`-Statement unmittelbar davor bezieht.

- **Lösung**

```
if (type == ACL_USER) {
  if (acl_set_qualifier (acle, (const void *) &uid) != 0)
    return -1;
} else if (type == ACL_GROUP)
  if (acl_set_qualifier (acle, (const void *) &gid) != 0)
    return -1;
```

3. - **Beschreibung** Wenn ein Gruppen- oder Benutzername länger als drei Zeichen ist, wird wird das Programm mit der Fehlermeldung `Math result not representable` beendet.

   - **Ursache** Die Funktion `decode_entry()` erhält ein char array als Parameter. Um zu vermeiden, daß über den maximalen Index dieses arrays geschrieben wird, wird mit dem `sizeof` Operator die Größe dieser arrays getestet, bevor hineingeschrieben wird. Das array wird jedoch als Zeiger der Funktion übergeben. Daher ergibt `sizeof` die Größe des Zeigers und nicht des arrays.

   - **Lösung** Es wird eine maximale Stringlänge definiert, mit der all diese arrays deklariert werden. Diese wird dann anstatt `sizeof` verwendet.

4. - **Beschreibung** Es läßt sich immer nur jeweils ein Gruppen- bzw. User-Eintrag erzeugen:

```
> setacl -bu g:g1:rwx,g:g2:r-- /acl/dat1
> getacl -l /acl/dat1
# file: /acl/dat1
# owner: prj
# group: users
user::rw-
group::r--
group:g1:r--
other::r--
```

   - **Ursache** Eine ACL wird ergänzt, indem zunächst die für die entsprechende Datei bestehende ACL gelesen wird (sie besteht zumindest aus den drei Einträge für ACL_USER_OBJ, ACL_GROUP_OBJ und ACL_OTHER_OBJ, die ggf. aus den File Permission Bits der Datei erzeugt werden) und in diese werden die Einträge der neuen ACL sortiert eingefügt. Befindet sich

in der einzufügenden ACL ein Eintrag mit dem gleichem Typ und Qualifier, so ersetzt er den bestehenden Eintrag. Dieser Algorithmus findet sich in der Funktion `update_entries(...)` (Datei
`aclutils-0.1/progs/setacl.c`). In der vorliegenden Version wurde das Qualifier-Feld des einzufügenden ACL-Entries mit seinem eigenen Qualifier Feld verglichen. Folglich wurde nur ein Eintrag pro ACL-Entry-Typ erzeugt und dieser dann jedesmal durch die folgenden Einträge ersetzt.

- **Lösung**

```
...
  uidp1 = (gid_t *)acl_get_qualifier (acle1);
...
  gidp1 = (gid_t *)acl_get_qualifier (acle1);
...
switch (tag2) {
  case ACL_USER:
-   uidp2 = (uid_t *)acl_get_qualifier (acle1);
+   uidp2 = (uid_t *)acl_get_qualifier (acle2);
...
    if (*uidp1 == *uidp2) {
...
  case ACL_GROUP:
-   gidp2 = (gid_t *)acl_get_qualifier (acle1);
+   gidp2 = (gid_t *)acl_get_qualifier (acle2);
...
    if (*gidp1 == *gidp2) {
```

5. - **Beschreibung** Es lassen sich keine einzelnen ACL-Entries löschen:

```
 prj@hast:/home/prj > getacl -l /acl/dat1
# file: /acl/dat1
# owner: prj
# group: users
user::rw-
group::r--
group:g1:rwx
group:g2:r--
other::r--
prj@hast:/home/prj > setacl -x g:g2:r-- /acl/dat1
prj@hast:/home/prj > getacl -l /acl/dat1
# file: /acl/dat1
# owner: prj
# group: users
user::rw-
group::r--
other::r--
prj@hast:/home/prj >
```

- **Ursache** ACL-Entries werden gelöscht, indem zunächst die für die entsprechende Datei bestehende ACL gelesen wird. Die Einträge dieser ACL werden mit den zu löschenden Einträgen in der gleichen Art verglichen wie es auch beim Einfügen geschieht, also auch mit dem gleichen Fehler. Dieser Algorithmus findet sich in der Funktion `remove_entries(...)`
(Datei `aclutils-0.1/progs/setacl.c`).

- **Lösung** Wie beim Einfügen von ACL-Entries.

6. - **Beschreibung** Es wird ein Segmentation-Fault erzeugt, wenn die bei `setacl` und `getacl` angegebene Datei nicht existiert.

   - **Ursache** In der Datei `./fs/acl.c` in der Funktion `sys_acl_ctl()` wird unter Verwendung der Funktion `dentry(...)` aus dem Dateinamen ein Zeiger auf eine Struktur vom Typ `dentry` erzeugt. Existiert die Datei nicht, wird dieses durch einen ungültigen Zeiger angezeigt. Dies kann mit dem Makro `IS_ERR()` festgestellt werden kann. Dieser Zeiger ist nicht der Null Zeiger. Am Ende der Funktion wird diese Struktur wieder zurückgeschrieben. Zuvor wird abgefragt, ob der Zeiger auf die Struktur ungleich Null ist. Wenn ein Name nicht existiert ist der Zeiger jedoch ungleich Null aber trotzdem ungültig. Dadurch entsteht ein Segmentation-Fault, da `dput` versucht den ungültigen Zeiger zurückzuschreiben.

```
...
  dentry = namei (filename);
  if (IS_ERR(dentry))
    error = PTR_ERR(dentry);
  else
  inode = dentry->d_inode;
...
  if (dentry)
    dput (dentry);
```

   - **Lösung**

```
 if (dentry && !IS_ERR(dentry))
    dput (dentry);
```

7. `check_descriptor` meldet falsche Werte für `freecount`, `max` und `min`.

   - **Beschreibung** Am Anfang jedes Blocks, der zum Speichern von ACLs verwendet wird, steht eine Struktur vom Typ `ext2_acl_desc`:

```
struct ext2_acl_desc /* Descriptor of blocks in ACL inodes */
{
  __u16  acld_max_acl;   /* Size of largest free fragment */
  __u16  acld_min_acl;   /* Size of smallest free fragment */
  __u16  acld_free_acl;  /* Count of free acl */
....
};
```

   Die Funktion `check_descriptor` dient dazu, die Einträge dieser Struktur auf Ihre Gültigkeit hin zu überprüfen. Treten dabei Inkonsistenzen auf, werden Fehlermeldungen erzeugt, die vom `syslogd` in der Datei `/var/log/warn` gespeichert werden:

```
EXT2-fs error (device 02:00): check_acl_descriptor: Inode 3,
block 0, free count wrong: stored=29, counted=435
EXT2-fs error (device 02:00): check_acl_descriptor: Inode 3,
block 0, smallest fragment count wrong: stored=30, counted=0
EXT2-fs error (device 02:00): check_acl_descriptor: Inode 3,
block 0, smallest fragment count wrong: stored=30, counted=0
EXT2-fs error (device 02:00): check_acl_descriptor: Inode 4,
block 0, free count wrong: stored=26, counted=351
EXT2-fs error (device 02:00): check_acl_descriptor: Inode 4,
block 0, smallest fragment count wrong: stored=25, counted=0
```

- **Ursache** Neben der Verwendung falscher Variablennamen war auch der Algorithmus zur Ermittelung der Werte von `acld_max_acl`, `acld_min_acl` und `acld_free_acl` fehlerhaft:

```
for (i = 0; i < EXT2_ACLE_PER_BLOCK(sb); i++)
  if (!test_bit (i, bitmap)) {
    free_count++;
    for (ok = 1, j = 1;
         i + j < EXT2_ACLE_PER_BLOCK(sb) && ok; j++)
      if (test_bit (i + j, bitmap))
        ok = 0;
      else
        free_count++;
      if (j - 1 > max)
        max = j - 1;
      if (j - 1 < min)
        min = j - 1;
   }
```

  - `free_count` ist zu hoch, da nach durchlaufen der inneren Schleife `i` nicht um j erhöht wird, wodurch ein freies Bit mehrmals gezählt wird.
  - Nach Durchlaufen der inneren Schleife ist j nicht eins größer als die Anzahl der leeren Bits deren Index größer oder gleich i ist.
    Wenn $(i+j) = EXT2\_ACLE\_PER\_BLOCK(sb)$, dann ist j eins zu niedrig.

  Dieser Algorithmus wird in ähnlicher Form in den Funktionen `update_descriptor(...)` und `allocate_acl_entries(...)` verwendet.

- **Lösung**
  - In der Funktion `check_acl_descriptor`:

```
for (i = 0; i < EXT2_ACLE_PER_BLOCK(sb); i++)
  if (!test_bit (i, bitmap)) {
    free_count++;
    for (j = 1; i + j < EXT2_ACLE_PER_BLOCK(sb) &&
          !test_bit(i + j, bitmap); j++) {
      free_count++;
    }
    if (j  > max)
      max = j;
    if (j  < min)
      min = j;
    i+= j ;
  }
```

  - In der Funktion `update_descriptor ()`

```
for (i = 0; i < EXT2_ACLE_PER_BLOCK(sb); i++)
  if (!test_bit (i, bitmap)) {
    for (j = 1; i + j < EXT2_ACLE_PER_BLOCK(sb) &&
          !test_bit (i + j, bitmap); j++);
      if (j  > max)
        max = j;
      if (j  < min)
        min = j;
      i+= j;
  }
```

– In der Funktion `allocate_acl_entries ()`

```
int cont = 0;
...
for(k = 2; k < EXT2_ACLE_PER_BLOCK(sb) &&
           cont < entry_count; k++) {
  if(test_bit(k, bitmap)) {
    cont = 0;
   } else {
    if(cont == 0)
      j = k;
    cont++;
   }
}
ok = (cont == entry_count);
...
```

8. Eine ACL läßt sich nicht löschen, Fehlermeldung beim Löschen.

   - **Beschreibung** Wenn die Access ACL einer Datei mit der Option `-k`, `-x` oder `-X` gelöscht wird, wird eine Fehlermeldung ausgegeben und die alte ACL bleibt bestehen.

   - **Ursache** Eine ACL wird gelöscht, indem eine neue ACL mit den drei notwendigen Einträgen gespeichert wird. Dies geschieht mit Hilfe der Funktion `set_acl()`. Die Funktion `set_acl()` löscht zunächst eine eventuell bestehende ACL zu einer Datei. Eine ACL kann von mehreren Dateien referenziert werden. Das geschieht dann, wenn eine Datei eine Default ACL verwendet. Beim Erzeugen einer Datei in einem Verzeichnis, für das eine Default ACL definiert ist, verwendet diese dieselbe ACL. Erst wenn die ACL der Datei oder die Default ACL verändert werden, wird die veränderte ACL separat gespeichert. Um dies zu realisieren, wird zu jeder ACL ein Referenzzähler unterhalten. Jede Referenz erhöht diesen, jedes Löschen erniedrigt ihn. Erst wenn dieser 0 ist, wird die ACL tatsächlich gelöscht. Dieser Referenzzähler wird in der Funktion `allocate_acl_header` fälschlicherweise mit 0 initialisiert. Die Funktion `remove_acl()` erniedrigt den Zähler und überprüft dann, ob er 0 ist. Ist er dies nicht, bricht die Funktion erfolgreich ab, da ja noch andere Dateien diese ACL benutzen. Durch die falsche Initialisierung ist der Zähler anfangs Null und somit nach dem Decrementieren ungleich 0. Daher wird die ACL nicht gelöscht. Wenn der Referenzzähler ungleich 0 ist bleibt auch der Verweis auf die ACL im Inode bestehen.

     Wenn nun die Bestehende ACL gelöscht wurde, testet `set_acl()`, ob die neue ACL lediglich aus den drei notwendigen Einträgen besteht. Ist dies der Fall, wird lediglich unter Verwendung der Funktion `update_mode_from_acl()` der Mode des Inodes entsprechend der ACL eingestellt. Es wird jedoch keine ACL gespeichert. Diese Funktion gibt grundsätzlich 1 statt 0 zurück, was zu einer Fehlermeldung führt.

   - **Lösung**

     – `allocate_acl_header`: Referenzzähler richtig initialisieren.

- `update_mode_from_acl()`: Returnwert von 0 auf 1 abändern.
- `remove_acl()`: Referenz auf ACL im Inode löschen.

9.
- **Beschreibung** Wird eine Datei in einem Verzeichnis mit einer Default ACL erzeugt, erhält die Datei zwar diese ACL als Access ACL, jedoch wird der Datei-Mode nicht entsprechend der ACL eingestellt.
- **Ursache** In der Funktion `ext2_inherit_acl()` fehlt der Code, um die Dateirechte entsprechend der geerbten ACL einzustellen. Die Funktion `update_mode_from_acl()` funktioniert nur mit einer ACL die lediglich die drei benötigten ACL-Entries beinhaltet.
- **Lösung** Vermutlich `update_mode_from_acl()` erweitern und dann von `ext2_inherit_acl()` aus aufrufen.

10.
- **Beschreibung** Das Execute-Recht wird für normale Dateien nur dann gewährt, wenn es für wenigsten einen Benutzer durch die Dateirechte gewährt wird.
- **Ursache** In der Funktion `prepare_binprm()` erfolgt vor der eigentlichen Berechtigungsüberprüfung eine Abfrage ob mindestens ein execute bit im Dateimodus gesetzt ist. Dies dient dazu, daß auch priveligierten Prozessen verwehrt wird, Dateien auszuführen, für die kein Execute Bit gesetzt ist. Ansonsten könnte root alle Dateien ausführen.

  Draft 17 definiert, daß es ausreicht, wenn das Execute Bit durch mindestens einem ACL-Entry gewährt wird.
- **Lösung** Ein nicht benutztes Bit in den File Permission Bits wird benutzt, um zu markieren, daß in einer für den Inode vorhandenen Access ACL für mindestens einen Eintrag das Execute-Recht gewährt wird. Dieses Bit wird zusätzlich von der Funktion `prepare_binprm()` abgefragt.

11.
- **Beschreibung** Versuch des Abspeicherns einer zu großen ACL führt zum Löschen der bestehenden ACL.
- **Ursache** In der bestehenden Implementierung kann eine ACL nicht fragmentiert über mehrere Datenblöcke gespeichert werden. Damit ist die Größe einer ACL bei einer Blockgröße von 1 kb auf 30 Einträge beschränkt. Die Größe der ACL wird durch `check_acl()` nicht überprüft und `set_acl()` löscht die bestehende ACL, bevor die neue ACL gespeichert wird. Erst beim Speichern der neuen ACL tritt dann der Fehler auf.
- **Lösung** `check_acl()` wird im VFS aufgerufen und kann somit tendenziell auch für andere Dateisysteme verwendet werden. Da die Größeneinschränkung EXT2-spezifisch ist, gehört ein Test hierauf nicht an diese Stelle. Daher wird die Abfrage zunächst in `set_acl()` eingebaut.

12.
- **Beschreibung** `e2fsck` erkennt die für die ACLs verwendeten Blöcke nicht als benutzt und will sie daher freigeben.

- **Ursache** Die Funktion `check_blocks()` wird für jeden Inode aufgerufen und registriert die von den Inodes referenzierten Blöcke als benutzt.
  Am Anfang der Funktion wird zuerst mit Hilfe der Funktion `ext2fs_inode_has_valid_blocks()` geprüft, ob der Inode überhaupt gültige Blöcke referenzieren kann. Diese Funktion gibt für die ACL-Inodes false zurück.
- **Lösung** Statt die Funktion `ext2fs_inode_has_valid_blocks()` zu ändern, wurde die Funktion `check_blocks()` so verändert, daß explizit nachgefragt wird, ob es sich um einen ACL-Inode handelt. Ist dies der Fall, dann wird nicht abgebrochen.

Anhang C. ACL: RT Tester Konfigurationsdateien

C.1 vvtconfig.txt

```
# Configuration File for VVT-RT
# IDs 601 (RX) und 602 (TX) are reserved for CCL processes
# CVMID and AMID numbers must not be the same (unique IDs)

GLOBAL                          # Global definitions for all abstract machines
  CTRLSHMID     301             # Shared memory ID of the control structure
  CVSHMID       302             # Shared memory ID of the cv block
END

AM 1                                    # shell abstract machine
  AMSHMID       1                       # Shared memory ID
  CVMID         301                     # CCL-ID of the continuous value part
  ALPHABETFILE  /conf/shell/shell.t     # Name of the alphabet file incl. path
                                        # from VVTHOME
  TRANSFILE     /conf/shell/shell.trans # Name of the transition file
  STATEFILE     /conf/shell/shell.state # Name of the state file
  REFFILE       /conf/shell/shell.ref   # Name of the refusals file
  EVENTFILE     /conf/shell/shell.event # Name of the event file
  TRACEFILE     /conf/shell/shell.trace # Name of the trace file that is
                                        # generated by VVT-RT
  CASESFILE     /conf/shell/shell.tst   # Name of the cases file that is
                                        # generated by VVT-RT
  CASESLTXFILE  /conf/shell/shell.ltx   # Name of the LaTeX cases file
                                        # that is generated by VVT-RT
  RX_ADDR       12                      # Internal ID of the CCL-RX-Process
  RX_ADDR       111                     # Internal ID of the CCL-RX-Process
  SEED          42                      # random seed for this AM
  STOPONIOERROR NO                      # do not stop if an unexpected output
                                        # occurs
  VERBOSE       YES                     # display unexpected outputs

  # Timer definitions: No:LowerBound:UpperBound
  TIMER 0:1000:3000
        1:1000:3000
        9:2000:-
END

AM 2                                    # shell abstract machine
  AMSHMID       2                       # Shared memory ID
  CVMID         301                     # CCL-ID of the continuous value part
  ALPHABETFILE  /conf/shell/acl.t       # Name of the alphabet file incl. path
                                        # from VVTHOME
  TRANSFILE     /conf/shell/acl.trans   # Name of the transition file
  STATEFILE     /conf/shell/acl.state   # Name of the state file
  REFFILE       /conf/shell/acl.ref     # Name of the refusals file
  EVENTFILE     /conf/shell/acl.event   # Name of the event file
  TRACEFILE     /conf/shell/acl.trace   # Name of the trace file that is
                                        # generated by VVT-RT
  CASESFILE     /conf/shell/acl.tst     # Name of the cases file that is
                                        # generated by VVT-RT
  CASESLTXFILE  /conf/shell/acl.ltx     # Name of the LaTeX cases file
                                        # that is generated by VVT-RT
  RX_ADDR       12                      # Internal ID of the CCL-RX-Process
  RX_ADDR       111                     # Internal ID of the CCL-RX-Process
  SEED          42                      # random seed for this AM
  STOPONIOERROR NO                      # do not stop if an unexpected output
                                        # occurs
  VERBOSE       YES                     # display unexpected outputs
  TIMER 0:1000:3000
        1:1000:3000
        9:2000:-
END
```

```
AM 3                                    # shell abstract machine
  AMSHMID       3                       # Shared memory ID
  CVMID         301                     # CCL-ID of the continuous value part
  ALPHABETFILE  /conf/shell/perm.t      # Name of the alphabet file incl. path
                                        # from VVTHOME
  TRANSFILE     /conf/shell/perm.trans  # Name of the transition file
  STATEFILE     /conf/shell/perm.state  # Name of the state file
  REFFILE       /conf/shell/perm.ref    # Name of the refusals file
  EVENTFILE     /conf/shell/perm.event  # Name of the event file
  TRACEFILE     /conf/shell/perm.trace  # Name of the trace file that is
                                        # generated by VVT-RT
  CASESFILE     /conf/shell/perm.tst    # Name of the cases file that is
                                        # generated by VVT-RT
  CASESLTXFILE  /conf/shell/perm.ltx    # Name of the LaTeX cases file
                                        # that is generated by VVT-RT
  RX_ADDR       12                      # Internal ID of the CCL-RX-Process
  RX_ADDR       111                     # Internal ID of the CCL-RX-Process
  SEED          42                      # random seed for this AM
  STOPONIOERROR NO                      # do not stop if an unexpected output
                                        # occurs
  VERBOSE       YES                     # display unexpected outputs
  TIMER 0:1000:3000
        1:1000:3000
        9:2000:-
END
```

C.2 ae2raw.txt

```
1 50 cmd.setfacl 1 0 8 73 65 74 66 61 63 6C 20
1 51 cmd.getfacl 1 0 8 67 65 74 66 61 63 6C 20
1 52 option.b 1 0 3 2D 62 20
1 53 option.d 1 0 3 2D 64 20
1 54 option.k 1 0 3 2D 6B 20
1 55 option.n 1 0 3 2D 6E 20
1 56 option.m 1 0 3 2D 6D 20
1 57 option.M 1 0 3 2D 4D 20
1 58 option.x 1 0 3 2D 78 20
1 59 option.X 1 0 3 2D 58 20
1 60 file.0 1 0 11 20 2F 61 63 6C 2F 66 69 6C 65 30
1 61 file.1 1 0 11 20 2F 61 63 6C 2F 66 69 6C 65 31
1 62 file.2 1 0 11 20 2F 61 63 6C 2F 66 69 6C 65 32
1 63 file.3 1 0 11 20 2F 61 63 6C 2F 66 69 6C 65 33
1 64 dir.0 1 0 10 20 2F 61 63 6C 2F 64 69 72 30
1 65 dir.1 1 0 10 20 2F 61 63 6C 2F 64 69 72 31
1 66 dir.2 1 0 10 20 2F 61 63 6C 2F 64 69 72 32
1 67 dir.3 1 0 10 20 2F 61 63 6C 2F 64 69 72 33
1 68 aclfile.0 1 0 9 61 63 6C 66 69 6C 65 30 20
1 69 aclfile.1 1 0 9 61 63 6C 66 69 6C 65 30 20
1 70 aclfile.2 1 0 9 61 63 6C 66 69 6C 65 30 20
1 71 aclfile.3 1 0 9 61 63 6C 66 69 6C 65 30 20
1 72 spc 1 0 1 20
1 73 cr 1 0 1 A
1 74 goacl 1 0 0 0
1 75 acldone 1 0 0
1 76 execOK 1 0 1 30
1 77 testError 1 0 1 31
1 78 wrongSyntax 1 0 1 32
1 79 undefData 1 0 0
2 50 utt.u 1 0 1 75
2 51 utt.user 1 0 4 75 73 65 72
2 52 gtt.g 1 0 1 67
2 53 gtt.group 1 0 5  67 72 6F 75 70
2 54 ott.o 1 0 1 6F
2 55 ott.other 1 0 5 6F 74 68 65 72
2 56 mtt.m 1 0 1 6D
2 57 mtt.mask 1 0 4 6D 61 73 6B
2 58 utag.uname.0 1 0 2 75 30
2 59 utag.uname.1 1 0 2 75 31
2 60 utag.uname.2 1 0 2 75 32
2 61 utag.uname.3 1 0 2 75 33
2 62 utag.uname.4 1 0 2 75 34
2 63 utag.uname.5 1 0 2 75 35
```

```
2 64 utag.uname.6 1 0 2 75 36
2 65 utag.uname.7 1 0 2 75 37
2 66 utag.uname.8 1 0 2 75 38
2 67 utag.uname.9 1 0 2 75 39
2 68 utag.uname.10 1 0 3 75 31 30
2 69 utag.uname.11 1 0 3 75 31 31
2 70 utag.uname.12 1 0 3 75 31 32
2 71 utag.uname.13 1 0 3 75 31 33
2 72 utag.uname.14 1 0 3 75 31 34
2 73 utag.uname.15 1 0 3 75 31 35
2 74 utag.uname.16 1 0 3 75 31 36
2 75 utag.uname.17 1 0 3 75 31 37
2 76 utag.uname.18 1 0 3 75 31 38
2 77 utag.uname.19 1 0 3 75 31 39
2 78 utag.uname.20 1 0 3 75 32 30
2 79 utag.uname.21 1 0 3 75 32 31
2 80 utag.uname.22 1 0 3 75 32 32
2 81 utag.uname.23 1 0 3 75 32 33
2 82 utag.uname.24 1 0 3 75 32 34
2 83 utag.uname.25 1 0 3 75 32 35
2 84 utag.uname.26 1 0 3 75 32 36
2 85 utag.uname.27 1 0 3 75 32 37
2 86 utag.uname.28 1 0 3 75 32 38
2 87 utag.uname.29 1 0 3 75 32 39
2 88 utag.uid.0 1 0 3 35 30 30
2 89 utag.uid.1 1 0 3 35 30 31
2 90 utag.uid.2 1 0 3 35 30 32
2 91 utag.uid.3 1 0 3 35 30 33
2 92 utag.uid.4 1 0 3 35 30 34
2 93 utag.uid.5 1 0 3 35 30 35
2 94 utag.uid.6 1 0 3 35 30 36
2 95 utag.uid.7 1 0 3 35 30 37
2 96 utag.uid.8 1 0 3 35 30 38
2 97 utag.uid.9 1 0 3 35 30 39
2 98 utag.uid.10 1 0 3 35 31 30
2 99 utag.uid.11 1 0 3 35 31 31
2 100 utag.uid.12 1 0 3 35 31 32
2 101 utag.uid.13 1 0 3 35 31 33
2 102 utag.uid.14 1 0 3 35 31 34
2 103 utag.uid.15 1 0 3 35 31 35
2 104 utag.uid.16 1 0 3 35 31 36
2 105 utag.uid.17 1 0 3 35 31 37
2 106 utag.uid.18 1 0 3 35 31 38
2 107 utag.uid.19 1 0 3 35 31 39
2 108 utag.uid.20 1 0 3 35 32 30
2 109 utag.uid.21 1 0 3 35 32 31
2 110 utag.uid.22 1 0 3 35 32 32
2 111 utag.uid.23 1 0 3 35 32 33
2 112 utag.uid.24 1 0 3 35 32 34
2 113 utag.uid.25 1 0 3 35 32 35
2 114 utag.uid.26 1 0 3 35 32 36
2 115 utag.uid.27 1 0 3 35 32 37
2 116 utag.uid.28 1 0 3 35 32 38
2 117 utag.uid.29 1 0 3 35 32 39
2 118 gtag.gname.0 1 0 2 67 30
2 119 gtag.gname.1 1 0 2 67 31
2 120 gtag.gname.2 1 0 2 67 32
2 121 gtag.gname.3 1 0 2 67 33
2 122 gtag.gname.4 1 0 2 67 34
2 123 gtag.gname.5 1 0 2 67 35
2 124 gtag.gname.6 1 0 2 67 36
2 125 gtag.gname.7 1 0 2 67 37
2 126 gtag.gname.8 1 0 2 67 38
2 127 gtag.gname.9 1 0 2 67 39
2 128 gtag.gname.10 1 0 3 67 31 30
2 129 gtag.gname.11 1 0 3 67 31 31
2 130 gtag.gname.12 1 0 3 67 31 32
2 131 gtag.gname.13 1 0 3 67 31 33
2 132 gtag.gname.14 1 0 3 67 31 34
2 133 gtag.gname.15 1 0 3 67 31 35
2 134 gtag.gname.16 1 0 3 67 31 36
2 135 gtag.gname.17 1 0 3 67 31 37
2 136 gtag.gname.18 1 0 3 67 31 38
2 137 gtag.gname.19 1 0 3 67 31 39
2 138 gtag.gname.20 1 0 3 67 32 30
2 139 gtag.gname.21 1 0 3 67 32 31
```

```
2 140 gtag.gname.22 1 0 3 67 32 32
2 141 gtag.gname.23 1 0 3 67 32 33
2 142 gtag.gname.24 1 0 3 67 32 34
2 143 gtag.gname.25 1 0 3 67 32 35
2 144 gtag.gname.26 1 0 3 67 32 36
2 145 gtag.gname.27 1 0 3 67 32 37
2 146 gtag.gname.28 1 0 3 67 32 38
2 147 gtag.gname.29 1 0 3 67 32 39
2 148 gtag.gid.0 1 0 3 31 30 30
2 149 gtag.gid.1 1 0 3 31 30 31
2 150 gtag.gid.2 1 0 3 31 30 32
2 151 gtag.gid.3 1 0 3 31 30 33
2 152 gtag.gid.4 1 0 3 31 30 34
2 153 gtag.gid.5 1 0 3 31 30 35
2 154 gtag.gid.6 1 0 3 31 30 36
2 155 gtag.gid.7 1 0 3 31 30 37
2 156 gtag.gid.8 1 0 3 31 30 38
2 157 gtag.gid.9 1 0 3 31 30 39
2 158 gtag.gid.10 1 0 3 31 31 30
2 159 gtag.gid.11 1 0 3 31 31 31
2 160 gtag.gid.12 1 0 3 31 31 32
2 161 gtag.gid.13 1 0 3 31 31 33
2 162 gtag.gid.14 1 0 3 31 31 34
2 163 gtag.gid.15 1 0 3 31 31 35
2 164 gtag.gid.16 1 0 3 31 31 36
2 165 gtag.gid.17 1 0 3 31 31 37
2 166 gtag.gid.18 1 0 3 31 31 38
2 167 gtag.gid.19 1 0 3 31 31 39
2 168 gtag.gid.20 1 0 3 31 32 30
2 169 gtag.gid.21 1 0 3 31 32 31
2 170 gtag.gid.22 1 0 3 31 32 32
2 171 gtag.gid.23 1 0 3 31 32 33
2 172 gtag.gid.24 1 0 3 31 32 34
2 173 gtag.gid.25 1 0 3 31 32 35
2 174 gtag.gid.26 1 0 3 31 32 36
2 175 gtag.gid.27 1 0 3 31 32 37
2 176 gtag.gid.28 1 0 3 31 32 38
2 177 gtag.gid.29 1 0 3 31 32 39
2 178 colon 1 0 1 3A
2 179 spc 1 0 1 20
2 180 comma 1 0 1 2C
2 181 acldone 1 0 0
2 182 goperm 1 0 0
2 183 goacl 1 0 0
2 184 permdone 1 0 0
2 185 undefData 1 0 0
3 50 cperm.r 1 0 1 72
3 51 cperm.w 1 0 1 77
3 52 cperm.x 1 0 1 78
3 53 cperm.void 1 0 1 2D
3 54 cperm.add 1 0 1 2B
3 55 cperm.rem 1 0 1 5E
3 56 operm.0 1 0 1 30
3 57 operm.1 1 0 1 31
3 58 operm.2 1 0 1 32
3 59 operm.3 1 0 1 33
3 60 operm.4 1 0 1 34
3 61 operm.5 1 0 1 35
3 62 operm.6 1 0 1 36
3 63 operm.7 1 0 1 37
3 64 permdone 1 0 0
3 65 goperm  1 0 0
3 66 undefData 1 0 0
```

C.3 ae2mae.txt

```
1 74 2 183
2 181 1 75
2 182 3 57
3 56 2 184
```

Anhang D. ACL: Quellcode für den Test

D.1 Quellcode von setfacl.awk

```
#!/usr/bin/awk -f

function getfacl(target,cmd,   acl)
{
  old = FS;
  FS = ":";
  acl = "[";
  cmd = "getfacl " cmd " " target;
  /* printf("cmd: %s NF %d\n",cmd,NF); */
  while ((cmd | getline) > 0){
    if ($0 !~ /^[[:blank:]]*$/ && $0 !~ /^#.*$/){
      acl=acl "entry(\"" $1 "\",\"" $2 "\",\"" $3 "\"),"
    }
  }
  acl=acl "]";
  close(cmd);
  FS = old;
  return acl;
}

/* process setfacl commandline */
function toaspect(){
  options = "[\n";
  for(i = 2; i < NF; i++){
    if ($i == "-m" || $i == "-x"){
      options = options "  option(\"" $i "\",[\n";
      i++;
      split($i,acl,",");
      for (j in acl){
        split(acl[j], tags, ":");
        options = options "    entry(\"" tags[1] "\",\"" tags[2] "\",\""
                                                   tags[3] "\"),\n";
      }
      options = options "  ]),\n";
    } else if ($i == "-M" || $i == "-X"){
      options = options "  option(\"" $i "\",[\n";
      i++;
      while ((getline line < $i) > 0){
        if (line !~ /^[[:blank:]]*$/ && line !~ /^#.*$/){
          split(line, tags, ":");
          options = options "    entry(\"" tags[1] "\",\"" tags[2] "\",\""
                                                     tags[3] "\"),\n";
        }
      }
      close($i);
      options = options "  ]),\n";
    } else if( $i == "-b" || $i == "-d" || $i == "-k" || $i == "-n" ){
      options = options "  option(\"" $i "\"),\n";
    } else {

      break;
    }
  }
  options = options "]";
  optind= i;
  print options;
  return options;
}

function aclvalid(acl, result){
  print acl > "tmp"
  close(tmp);
  /*printf("aclvalid: ");*/
```

```
  result = !system("./aclvalid < tmp");
  close("aclvalid < tmp");
  /*print result;*/
  return result;
}

function aclequals(a,b, result){
  print a > tmp
  print b >> tmp
  close(tmp);
  printf("aclequals: ");
  result = !system("./aclequals < tmp");
  close("aclequals < tmp");
  print result;
  return result;
}

function specsetfacl(access,default,cmdline, oldFS){
  print access > tmp
  print default >> tmp
  print cmdline >> tmp
  close(tmp);
  printf("predefault: \n%s\n", default);
  printf("specsetfacl");
  ("./specsetfacl < tmp") | getline access3
  ("./specsetfacl < tmp") | getline default3
  close("./specsetfacl < tmp");
  printf(" done.\n");
  printf("postdefault: \n%s\n", default3);
}

function valid(cmdline,isdir){
  if(isdir){
    return 1;
  }
  for(i = 2; i < NF; i++){
    if ($i == "-d" || $i == "-k"){
      return 0;
    }
  }
  return 1
}

BEGIN { optind= 0 }

{
  tmp = "tmp";
  target = $NF;
  cmdline = $0;
  aspect = toaspect();

 printf("optind= %d NF= %d\n", optind, NF);
  cmdline=""
  for(oidx=1;oidx < optind; oidx++) {
    cl= cl " " $oidx;
  }
  printf("cl= %s\n", cl);
  tidx= 1;
  for(idx=optind;idx<= NF;idx++) {
    targets[tidx]= $idx;
    tidx++;
  }
  printf("tidx= %d\n",tidx);

  for(idx=1;idx < tidx;idx++) {
    cmdline= cl " " targets[idx];
    target= targets[idx];
    isdir = !system("test -d " target);
    /*printf("cmdline: %s\n", cmdline);*/
    /*printf("t: %s\n", target);*/

    /* pre op state */
    access1 = getfacl(target);

    if(isdir) default1 = getfacl(target,"-d");
```

```
    /* paranoia check */
    /* print access1; */
    if(!aclvalid(access1) || isdir && !aclvalid(default1)){
      printf("Paranoia check 1 failed!\n");
      exit 1;
    }

    /* do it */
    printf("%s\n",cmdline);
    system(cmdline);
/*    printf(" done.\n");*/

    /* post op state */
    access2 = getfacl(target);
    if(isdir) default2 = getfacl(target,"-d");

    /* paranoia check */
    if(!aclvalid(access2) || isdir && !aclvalid(default2)){
      printf("Paranoia check 2 failed!\n");
      exit 1;
    }

    /* check it */
    if (valid(cmdline,isdir)){
      if (isdir){
        specsetfacl(access1, default1, aspect);
      } else {
        specsetfacl(access1, "[]", aspect);
      }
      /* paranoia check */
      if(!aclvalid(access2) || isdir && !aclvalid(default2)){
        printf("Paranoia check 3 failed!\n");
        exit 1;
      }
      if(aclequals(access2,access3)) {
        if (!isdir) {
          printf("%s access ok\n",target);
        } else if(aclequals(default2,default3)){
          printf("%s access and default ok\n",target);
        } else {
          printf("%s default not like spec exit 1\n",target);
          printf("system: \n%s\n", default2);
          printf("spec: \n%s\n", default3);
          exit 1;
        }
      } else {
        print "access not like spec exit: 1";
        exit 1;
      }
    } else {
      if(aclequals(access1,access2) && aclequals(default1,default2)){
        printf("%s ok\n",target);
      } else {
        print "invalid cmdl changed acl ! exit: 1";
        exit 1;
      }
    }
  }
  print "all done exit: 0";
  exit(0);
}
```

D.2 Quellcode von vvtsndccl.c

*/**
\begin{verbatim}
//
//
// Verified Systems International GmbH
// Prof. Dr. Jan Peleska
// Parkstrasse 78
// D-28209 Bremen
// Germany

```
// Tel. : +49 421 344576
// Fax : +49 421 344586
//
// e-mail: jp@informatik.uni-bremen.de
//
//-----------------------------------------------------------------
//
// (C) Copyright Verified Systems International GmbH
// $Date: 1999/08/23 08:03:29 $
//
//-----------------------------------------------------------------
//
// Produkt: VVT-RT - DEMO SYSTEM
//
//-----------------------------------------------------------------
//
// File Identification: vvtsndccl.c
//
//
// First edition by: Cornelia Zahlten
// Last update by $Author: steenbo $
//
//-----------------------------------------------------------------
//
// Description: CCL send process for demo system.
//
//
-----------------------------------------------------------------*/

/* Standard - Headerfiles */
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include <unistd.h>
        #include <sys/time.h>
#include <signal.h>

/* Windows - Headerfiles */
#ifdef _WINDOWS
        /* Do not forget to link winsock */
        #include <winsock.h>
        #include <io.h>
        #include <string.h>
        #define WSA_Version 0x0101
#endif
#ifdef _WIN32
        /* Do not forget to link wsock32.lib */
        #include <winsock.h>
        #include <io.h>
        #define WSA_Version 0x0101
#endif

/* LINUX - Headerfiles */
#ifdef _LINUX
        #include <sys/types.h>
        #include <sys/socket.h>
/* #include <linux/in.h>
 */
#endif

/* Solaris - Headerfiles */
```

```
#ifdef _SOL
        /* Do not forget to link lsocket */
        #include <sys/types.h>
        #include <sys/socket.h>
        #include <netinet/in.h>
#endif


/*
#ifdef _HPUX
        #include <sys/types.h>
        #include <netinet/in.h>
        #include <sys/socket.h>
#endif
*/

#include "vvtiflib.h"
#include "vvtemlib.h"
#include "vvtshmlib.h"
#include "vvtoslib.h"

#define MY_ID 111
#define SERV_INETADDR "127.0.0.1"
#define SERV_TCP_PORT 6543
#define MSGLEN 100
#define B_ARRAYLEN 4000
#define LEN_FILE_NAME 500
int sfd, sfdx;                                   /* socketdescriptors */


void sighandler() {
  char buf[100];
  shutdown(sfdx,2);
  shutdown(sfd,2);
  close(sfdx);
  close(sfd);
  exit(0);
}


void main(void) {

  int clilen;                                    /* length of address */
  struct sockaddr_in saddr, cliaddr;             /* socketaddresses of
                                                  server and client */
  char byteArray[B_ARRAYLEN];                    /* buffer to send/rcv messages*/

  t_vvtReturncodes retcode;

  t_vvtAddress am;
  t_vvtAEvent event;
  t_vvtCtrlShm *cPtr;

  t_vvtCmdType ctp;
  t_vvtDataType dtp;
  INT32 clen;
  char *cmd;
  t_vvtAddress addr;

  t_vvtMaeStruct mae;
```

```
    char ctptargetStr[LEN_FILE_NAME];
    char ctpifmStr[LEN_FILE_NAME];
    char amcclStr[LEN_FILE_NAME];
    char aerawStr[LEN_FILE_NAME];
    char aemaeStr[LEN_FILE_NAME];

    INT64 thisTimeStamp;
    INT64 duration = CCLSND_RESOLUTION;



#ifdef _WINDOWS
    WSADATA wsadata;
    WSAStartup (WSA_Version, &wsadata);
    clilen = 0;
#endif

#ifdef _WIN32
    WSADATA wsadata;
    WSAStartup (WSA_Version, &wsadata);
    clilen = 0;
#endif

#ifdef _LINUX
    bzero((char *)&saddr,sizeof(saddr));
#endif

#ifdef _SOL
    bzero((char *)&saddr,sizeof(saddr));
#endif

    /*--- install signal handler ---*/
    signal(15,&sighandler);
    /*--- set my socket address ---*/

    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(SERV_TCP_PORT);

    if ( -1 == (sfd = socket( AF_INET, SOCK_STREAM, 0 )) ) {
        fprintf(stderr, "cclsnd: Error when creating socket\n");
        fflush(stderr);
    }
    else
        printf("cclsnd: Socket: %i\n",sfd);

    if ( -1 == bind(sfd,(struct sockaddr *)&saddr,sizeof(saddr)) ) {
        fprintf(stderr, "cclsnd: Error on bind\n");
        fflush(stderr);
    }

    listen(sfd,5);

#ifdef _WINDOWS
    sfdx = accept(sfd, (struct sockaddr *) &cliaddr, (int FAR*) clilen);
#endif
#ifdef _WIN32
    sfdx = accept(sfd, (struct sockaddr *) &cliaddr, (int FAR*) clilen);
#endif
#ifndef _WINDOWS
#ifndef _WIN32
```

```
    sfdx = accept(sfd, (struct sockaddr *) &cliaddr, &clilen);
#endif
#endif



    if (vvtOk != (retcode = vvtGetShm(NULL,0))) {
       fprintf(stderr,
               "cclsnd: vvtGetShm returns code %i \n", retcode);
       fflush(stderr);
    }

    if (vvtOk != (retcode = vvtConnectShm(0))) {
       fprintf(stderr,
               "cclsnd: vvtConnectShm returns code %i \n", retcode);
       fflush(stderr);
    }

    cPtr = vvtGetCtrlShm();

    /*--- open event mapping files ---*/
    strncpy(ctptargetStr,getenv("VVTEVENTMAPS"),LEN_FILE_NAME);
    strncpy(ctpifmStr,getenv("VVTEVENTMAPS"),LEN_FILE_NAME);
    strncpy(amcclStr,getenv("VVTEVENTMAPS"),LEN_FILE_NAME);
    strncpy(aerawStr,getenv("VVTEVENTMAPS"),LEN_FILE_NAME);
    strncpy(aemaeStr,getenv("VVTEVENTMAPS"),LEN_FILE_NAME);
    strcat(ctptargetStr,"/ctp2target.txt");
    strcat(ctpifmStr,"/ctp2ifm.txt");
    strcat(amcclStr,"/am2ccl.txt");
    strcat(aerawStr,"/ae2raw.txt");
    strcat(aemaeStr,"/ae2mae.txt");


    retcode = vvtInitTables(ctptargetStr,
                            ctpifmStr,
                            amcclStr,
                            aerawStr,
                            aemaeStr,
                            (int)cPtr->amDefs[1].firstUnspec);

    if (retcode != vvtOk) {
       fprintf(stderr,
               "cclsnd: vvtInitTables returns code %i \n", retcode);
       fflush(stderr);
    }


    /*$$!!--- lock process into memory ---*/

    /*$$!!--- get best priority of all VVT-RT processes ---*/

    /*--- set start-of-test time stamp ---*/
    vvtInitTimeMilliSec();

    do {

       vvtWaitMilliSec(duration);

       /*--- get my wakeup time ---*/
       thisTimeStamp = vvtGetTimeMilliSec();
```

```
if ( vvtOk == vvtGetSndBufShm( MY_ID,
                               &am,
                               &event ) ) {

   /*-- check if event is mapped to other AM ---*/
   if( (vvtOk == vvtAm2Am(am,
                          event,
                          &mae)) &&
       (mae.noAm != 0)) {
      /*--- send event to other AM ---*/
      if (cPtr->testRunning) {
         retcode = vvtSetOccShm( mae.ae[0].amId,
                                 MY_ID,
                                 mae.ae[0].evId);
         if (retcode != vvtOk) {
            fprintf(stderr,
                    "cclrcv, %011lli: vvtSetOccShm returns code %i
\n",
                    cPtr->testTimeTic, retcode);
            fflush(stderr);
         }

      }
   } else {                         /*--- if no am2am map present ---*/
      /*--- init byteArray for msg transmission ---*/
      retcode = vvtResetByteArray(B_ARRAYLEN, byteArray);
      if (retcode != vvtOk) {
         fprintf(stderr,
                 "cclsnd, %011lli: vvtResetByteArray returns code %i
\n",
                 cPtr->testTimeTic, retcode);
         fflush(stderr);
      }

      printf("event: %d am: %d\n", event, am);
      retcode = vvtAm2Raw(am,
                          event,
                          &ctp,
                          &dtp,
                          &clen,
                          &cmd,
                          &addr);
      printf("ctp :%d dtp: %d clen: %d\n", ctp, dtp, clen);
      if (retcode != vvtOk) {
         fprintf(stderr,
                 "cclsnd, %011lli: vvtAm2Raw returns code %i \n",
                 cPtr->testTimeTic, retcode);
         fflush(stderr);
      }

      /*--- insert msg to vvt-lib buffer ---*/
      retcode = vvtPutCmd (ctp,dtp,cmd,clen, byteArray);
      if (retcode != vvtOk) {
         fprintf(stderr,
                 "cclsnd, %011lli: vvtPutCmd returns code %i \n",
                 cPtr->testTimeTic, retcode);
         fflush(stderr);
      }

      retcode = vvtWriteToSocket(sfdx,byteArray);
      if (retcode != vvtOk) {
```

```
            fprintf(stderr,
                    "cclsnd, %011lli: vvtWriteToSocket returns code %i
\n",
                    cPtr->testTimeTic, retcode);
            fflush(stderr);
          }

#ifdef _VERBOSE
          printf("cclsnd: Bytes written. Event %i\n",event);
#endif

        }
      }

      /*--- calculate next sleep time ---*/
      duration = TIMER_RESOLUTION
                    - (vvtGetTimeMilliSec() - thisTimeStamp);
      if ( duration < 0 ) duration = CCLSND_RESOLUTION;


   } while (1);


}

/*////////////////////////////////////////////////////////////////
//
// Change History:
//
// $Log: AppSrcTest.tex,v $
// Revision 1.1 1999/08/23 08:03:29 steenbo
// Test
//
// Revision 1.6 1999/03/15 18:03:05 jp
// New version of vvtInitTables().
//
// Revision 1.5 1999/02/22 10:48:12 jp
// .
//
// Revision 1.4 1999/01/15 09:04:13 jp
// .
//
// Revision 1.3 1998/12/02 10:20:54 jp
// - testTimeTic is now INT64
// - Event Mapping has now additional aemae.txt file ->
// call to vvtInitTables() changed
//
// Revision 1.2 1998/11/29 18:05:31 jp
// - adaptions for new vvtoslib
// - adaptions for new timer working with constant resolution
//
// Revision 1.1 1998/10/28 10:51:23 jp
// new demo target system
//
//
////////////////////////////////////////////////////////////////
\end{verbatim}
*/
```

D.3 Quelltext des Interface Moduls

```
/*
\begin{verbatim}
///////////////////////////////////////////////////////////////////////
//
// Verified Systems International GmbH
// Prof. Dr. Jan Peleska
// Parkstrasse 78
// D-28209 Bremen
// Germany
// Tel. : +49 421 344576
// Fax : +49 421 344586
// e-mail: jp@informatik.uni-bremen.de
//
//------------------------------------------------------------------
//
// (C) Copyright Verified Systems International GmbH $Date: 1999/08/23 08:03:29
$
//
//------------------------------------------------------------------
//
// Produkt: VVT-RT - DEMO APPLICATION
//
//------------------------------------------------------------------
//
// File Identification: vvtdemo.c
//
// $Header: /var/lib/cvs/live/bericht/acl/AppSrcTest.tex,v 1.1 1999/08/23 08:03:29
steenbo Exp $
//
// $Revision: 1.1 $
//
// First edition by: Jan Peleska
// Last update by $Author: steenbo $
//
//------------------------------------------------------------------
//
// Description: Source file implementing the
// demo target process acting as System under Test
//
------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "vvtiflib.h"
#include <string.h>
#include <signal.h>

#define MY_ADDR 100
#define BUFFLEN 4000

FILE *fd;
extern int sfds;
extern int sfdr;

void sighandler() {
   fflush(fd);
   fclose(fd);
   exit(0);
}
```

```
void main(void) {

  t_vvtReturncodes retcode;
  char byteArray[BUFFLEN];
  char buf[BUFFLEN];
  INT32 len;
  t_vvtCmdType ctp;
  t_vvtDataType dtp;
  INT32 num;
  INT32 timeBomb = 0;
  int i= 0;
  int cr= 0;
  int ret, idx;
  char logfile[255];
  char cmdline[255];


  strncpy(logfile,getenv("VVTDEMO"),255);
  strcat(logfile,"/testdata/ifmlog.txt");
  if((fd= fopen(logfile,"w")) == NULL) {
    fprintf(stderr,"error opening logfile\n");
    gets(buf);
    exit(1);
  }
  signal(15,sighandler);
  retcode = vvtConnect(MY_ADDR,"config.txt");
  fprintf(fd,"sfdr: %d\n",sfdr);
  fprintf(fd,"sfds: %d\n",sfds);
  strcpy(cmdline,"dosetfacl ");
  idx= strlen(cmdline);
  do {

    /*--- get data from test engine ---*/
    retcode = vvtResetByteArray(BUFFLEN,byteArray);
    vvtReceiveFromTestDriver(vvtWait,byteArray);
    len = BUFFLEN;
    while ( vvtOk == vvtGetNextCmd(&ctp,&dtp,(void *)buf,&len,byteArray) ) {
      int j;
      printf("len: %d %2i: ", len, i++);
      for (j=0; j<len; j++) {
        printf("%2.2X ", buf[j]);
        if(buf[j] != '\n') {
          cmdline[idx++]= buf[j];
        } else {
          cr= 1;
          cmdline[idx++]= 0;
        }
        // fprintf(fd,"%c",buf[j]);
      }
      printf("\n");
      buf[len] = '\0';
      printf("RECEIVED : len: %d CTP %i DTP %i DATA %s \n",
              len,ctp,dtp,buf);
      len = BUFFLEN;
    }

    if(cr==1) {
      cr= 0;

      fprintf(fd,"%s\t",cmdline);
```

```
        ret= system(cmdline);
        ret »= 8;
        fprintf(fd,"returnvalue: %d\n",ret);
        if(ret!=0) {
            ret= 1;
        }
        printf("RETURN DATA: %i\n",ret);
        sprintf(buf,"%d",ret);
        strcpy(cmdline,"dosetfacl ");
        idx= strlen(cmdline);
        retcode = vvtResetByteArray(BUFFLEN,byteArray);
        retcode = vvtPutCmd(ctp,dtp,buf,strlen(buf),byteArray);
        vvtSendToTestDriver(byteArray);
    }


  } while (1);

}
/*//////////////////////////////////////////////////////////////
//
// Change History:
//
// $Log: AppSrcTest.tex,v $
// Revision 1.1 1999/08/23 08:03:29 steenbo
// Test
//
// Revision 1.2 1998/12/02 10:26:05 jp
// .
//
// Revision 1.1 1998/10/28 10:51:23 jp
// new demo target system
//
//
//
//////////////////////////////////////////////////////////////
\end{verbatim}
*/
```

D.4 Acl.AS

```
SPEC Acl =
  Boolean+
  String+
  ListOps+
  Error+
  Chars+
  Strings+
  Combinator+

SORTS
  acl ::= [entry].
  entry ::= entry
            type::string
            tag::string
            mode::string.

  booleans ::= [boolean].

  acl_types ::= [acl_type].
  acl_type ::= acl_user_obj
             | acl_user
             | acl_group_obj
             | acl_group
             | acl_other
             | acl_mask.

OPNS equals :: entry -> entry -> boolean.
EQNS equals A B =
  (tag A == tag B) &&
  (type A == type B::acl_type).

OPNS equals :: acl -> acl -> boolean.
EQNS equals A B =
  ((--) A B equals_mode == []) &&
  ((--) B A equals_mode == []).

OPNS equals_mode :: entry -> entry -> boolean.
EQNS equals_mode A B =
  forall (s (member (chars (mode A));(==)) (member (chars (mode B))))
       (['r','w','x']::chars) &&
  equals A B.

OPNS type :: entry -> acl_type.
EQNS type Entry =
  if member ["u","user"] (type Entry::string) &&
     ismt (tag Entry)
  then acl_user_obj
  elsif member ["u","user"] (type Entry::string) &&
        not (ismt (tag Entry))
  then acl_user
  elsif member ["g","group"] (type Entry::string) &&
        ismt (tag Entry)
  then acl_group_obj
  elsif member ["g","group"] (type Entry::string) &&
        not (ismt (tag Entry))
```

```
  then acl_group
  elsif member ["m","mask"] (type Entry::string)
  then acl_mask
  elsif member ["o","other"] (type Entry::string)
  then acl_other
  else error(
    "invalid acl entry " ++
    type Entry ++ ":" ++
    tag Entry ++ ":" ++
    mode Entry).

END.
```

D.5 AclEquals.AS

```
SPEC AclEquals =
  Acl+
  SysShell+
  Boolean+
  ListOps+
  Combinator+
  StreamIO+
  StreamRW+
  Error+
  Chars+

OPNS goal :: system -> system.
EQNS goal System =
  let
    (Ok,In,System) = stdin System.
    (Ok1,A,In) = read([]::acl,In).
    (Ok2,B,In) = read([]::acl,In).
  in
    if not (forall (id) [Ok,Ok1,Ok2])
    then error("Cannot read stdin.")
    elsif A _equals B
    then set_returnvalue 0 System
    else set_returnvalue 1 System.

END.
```

D.6 AclSort.AS

```
SPEC AclSort =
  Acl+

LOCAL
  Combinator+
  StreamIO+
  StreamRW+
  ListOps+
  Error+
  SysShell+
  Strings+

OPNS goal :: system -> system.
EQNS goal System =
  let
    (Ok1,In,System) = stdin System.
    (Ok2,Acl,_) = read([]::acl,In).

    Acl = acl_sort Acl.

    (Ok3,Out,System) = stdout System.
    (Ok4,_) = write([]::acl,Out).
  in
    if not (forall (id) [Ok1,Ok2])
    then error("Cannot read stdin.")
    elsif not (forall (id) [Ok3,Ok4])
    then error("Cannot write to stdout.")
    else System.

OPNS acl_sort :: acl -> acl.
EQNS acl_sort Acl = sort acl_leq Acl.

OPNS acl_leq :: entry -> entry -> boolean.
EQNS acl_leq A B = index acl_order A <= index acl_order B.

OPNS index :: tests -> entry -> integer.
EQNS index []     _ = error("Not found in acl_order.").
     index (A:X) D = if A D then 0 else (1 + index X D).

SORTS tests ::= [entry -> boolean].
OPNS acl_order :: tests.
EQNS acl_order = [
    s (type;(==) "user";(&&)) (tag;ismt),
    s (type;(==) "u";(&&)) (tag;ismt),
    s (type;(==) "user";(&&)) (tag;ismt;not),
    s (type;(==) "u";(&&)) (tag;ismt;not),
    type;(==) "mask",
    type;(==) "m",
    s (type;(==) "group";(&&)) (tag;ismt),
    s (type;(==) "g";(&&)) (tag;ismt),
    s (type;(==) "group";(&&)) (tag;ismt;not),
    s (type;(==) "g";(&&)) (tag;ismt;not),
    type;(==) "other",
    type;(==) "o"
  ].
```

```
END.
```

D.7 AclValid.AS

```
SPEC AclValid =
LOCAL
  Acl+
  Combinator+
  StreamIO+
  StreamRW+
  ListOps+
  Error+
  SysShell+
  Strings+
  Chars+
  ListFold+
  ListMap+

OPNS goal :: system -> system.
EQNS goal System =
  let
    (Ok,In,System) = stdin System.
    (Ok1,Acl,In) = read([]::acl,In).
    (Ok2,IsDefault,_) = read(true::boolean,In).
  in
    if not (forall (id) [Ok,Ok1,Ok2])
    then error("Cannot read stdin.")
    elsif acl_valid Acl IsDefault
    then set_returnvalue 0 System
    else set_returnvalue 1 System.

OPNS acl_valid :: acl -> boolean -> boolean.
EQNS acl_valid (Acl::acl) IsDefault =
  (IsDefault && ismt Acl) || (
  single (filter (type;(==) acl_user_obj) Acl) &&
  single (filter (type;(==) acl_group_obj) Acl) &&
  single (filter (type;(==) acl_other) Acl) &&
  (Acl == mkset (equals) Acl) &&
  (not (exist (type;member [acl_user,acl_group]) Acl) ||
   single (filter (type;(==) acl_mask) Acl))).

OPNS acl_check_syntax :: acl -> boolean.
EQNS acl_check_syntax Acl =
  forall
    (type;
     member ["u","user","g","group","o","other","m","mask"])
    Acl &&
  forall
    (mode;chars;forall (member (['r','w','x','-']::chars)))
    Acl &&
  forall
    (s (type;member ["m","mask"];not;(||)) (tag;ismt))
    Acl &&
  forall
    (s (type;member ["o","other"];not;(||)) (tag;ismt))
    Acl.

END.
```

D.8 Mu.AS

```
SPEC Mu =
FORMAL
SORTS data.
      list ::= [data].

GLOBAL
  Boolean+
  ListOps+
  Combinator+

OPNS mu :: (data -> boolean) -> list -> data.
EQNS mu F L = hd (filter F L).

END.
```

D.9 PermCheck.AS

```
SPEC PermCheck =
  Acl+
  Mu+

  String+
  Strings+
  Integer+
  StreamIO+
  StreamRW+
  SysShell+

SORTS
  user ::= user
             euid::string
             egid::string
             groups::strings.

  object ::= object
               uid::string
               gid::string.

OPNS goal :: system -> system.
EQNS goal System =
  let
    (Ok,In,System) = stdin System.
    (Ok1,User,In) = read(user "" "" [],In).
    (Ok2,Object,In) = read(object "" "",In).
    (Ok2,Mode,In) = read("",In).
    (Ok3,Acl,In) = read([]::acl,In).
  in
    if not (forall (id) [Ok,Ok1,Ok2,Ok3])
    then error("Cannot read stdin.")
    elsif perm_check User Object Mode Acl
    then set_returnvalue 0 System
    else set_returnvalue 1 System.

OPNS perm_check ::
  user -> object -> string -> acl -> boolean.
EQNS perm_check User Object Mode Acl =
  if euid User == uid Object
  then matched_entry
    (mu (type;(==) acl_user_obj) Acl)
    Mode
    Acl

  elsif exist (matching_acl_user User) Acl
  then matched_entry
    (mu (matching_acl_user User) Acl)
    Mode
    Acl

  elsif member (egid User : groups User) (gid Object) ||
        exist (matching_acl_group User) Acl
  then
    if exist (granting_group_entry User Object Mode) Acl
```

```
    then matched_entry
      (mu (granting_group_entry User Object Mode) Acl)
      Mode
      Acl
    else false

  elsif exist (granting_acl_other Mode) Acl
  then matched_entry
    (mu (granting_acl_other Mode) Acl)
    Mode
    Acl

  else false.

OPNS matched_entry :: entry -> string -> acl -> boolean.
EQNS matched_entry Entry Mode Acl  =
  if subseteq (mode Entry) Mode
  then
    if (type Entry == acl_user_obj) ||
       (type Entry == acl_other)
    then true
    elsif
      not (exist (type;(==) acl_mask) Acl) ||
      subseteq
        (mode (mu (type;(==) acl_mask) Acl))
        Mode
    then true
    else false
  else false.

OPNS subseteq :: string -> string -> boolean.
EQNS subseteq A B =
  member (chars A) (remove ((==) '-') (chars B)).

OPNS matching_acl_user :: user -> entry -> boolean.
EQNS matching_acl_user User Entry =
  type Entry == acl_user &&
  (tag Entry == euid User).

OPNS matching_acl_group :: user -> entry -> boolean.
EQNS matching_acl_group User Entry =
  type Entry == acl_group &&
  member (egid User : groups User) (tag Entry).

OPNS granting_group_entry ::
  user -> object -> string -> entry -> boolean.
EQNS granting_group_entry User Object Mode Entry =
  (type Entry == acl_group_obj &&
   member (egid User : groups User) (gid Object) &&
   subseteq (mode Entry) Mode) ||
 (type Entry == acl_group &&
  member (egid User : groups User) (tag Entry) &&
  subseteq (mode Entry) Mode).

OPNS granting_acl_other :: string -> entry -> boolean.
```

```
EQNS granting_acl_other Mode Entry =
  type Entry == acl_other &&
  subseteq (mode Entry) Mode.

END.
```

D.10 SpecSetfacl.AS

```
SPEC SpecSetfacl =
OPNS
      goal:: system -> system.

LOCAL
  Combinator+
  StreamIO+
  StreamRW+
  Error+
  ListOps+
  ListFold+
  System+

  Acl+

SORTS
  options ::= [option].
  option ::= option string
           | option string acl.

EQNS goal System =
  let
    (Ok,In,System) = stdin System.
    (Ok1,Access,In) = read([]::acl,In).
    (Ok2,Default,In) = read([]::acl,In).
    (Ok3,Options,In) = read([]::options,In).
    (Access_,Default_) = process_options (Access,Default) Options.
    (Ok10,Out,System) = stdout System.
    (Ok11,Out) = write(Access_,Out).
    System = System + "\n" >> Out.
    (Ok12,Out) = write(Default_,Out) >> System.
    System = System + "\n".
  in
    if not (forall (id) [Ok,Ok1,Ok2,Ok3])
    then error("Cannot read stdin.")
    elsif not (forall (id) [Ok10,Ok11,Ok12])
    then error("Cannot write stdout.")
    else System.

OPNS process_options ::
  (acl,acl) -> options -> (acl,acl).
EQNS process_options (Access,Default) Options =
  let
    D = member Options (option "-d").
    (Access,Default) =
      foldl (process_option D)
        (Access,Default)
        (remove (member [option "-d",option "-n"]) Options).
  in
    if D && do_recalc_mask Options && not (ismt Default)
    then (Access, recalc_mask Default)
    elsif do_recalc_mask Options
    then (recalc_mask Access, Default)
    else (Access,Default).
```

```
OPNS process_option ::
  boolean -> (acl,acl) -> option -> (acl,acl).
EQNS
  process_option A B (option "-b") = option_b A B.
  process_option A B (option "-k") = option_k A B.
  process_option A B (option "-m" Modify) = option_m Modify A B.
  process_option A B (option "-M" Modify) = option_M Modify A B.
  process_option A B (option "-x" Modify) = option_x Modify A B.
  process_option A B (option "-X" Modify) = option_X Modify A B.
 $process_option _ _ (option C _) = error("Unknown option " ++ C).
 $process_option _ _ (option C) = error("Unknown option " ++ C).

OPNS
  option_b,option_k :: boolean -> (acl,acl) -> (acl,acl).
  option_m,option_M,option_x,option_X ::
    acl -> boolean -> (acl,acl) -> (acl,acl).

EQNS
  option_b D (Access,Default) =
    if D
    then (Access, filter (is_base) Default)
    else (filter (is_base) Access, Default).

  option_k _ (Access,_) = (Access,[]).

  option_m Modify D (Access,Default) =
    if D
    then (Access, foldl (modify) Default Modify)
    else (foldl (modify) Access Modify, Default).

  option_M = option_m.

  option_x Modify D (Access,Default) =
    if D
    then (Access,remove (member (equals) Modify) Default)
    else (remove (member (equals) Modify) Access,Default).

  option_X = option_x.

OPNS is_base :: entry -> boolean.
EQNS is_base =
  type;member [acl_user_obj,acl_group_obj,acl_other].

OPNS modify :: acl -> entry -> acl.
EQNS modify [H|T] Modify =
  if Modify _equals H
  then
    if member (chars (mode Modify)) '+'
    then [mode(H,string (mkset (chars (mode H) ++ chars (mode Modify) -- ['+'])))|T]
    elsif member (chars (mode Modify)) '^'
    then [mode(H,string (mkset (chars (mode H) -- chars (mode Modify))))|T]
    else [Modify|T]
  else [H|modify T Modify].
    modify [] Modify =
  if member (chars (mode Modify)) '+'
```

```
  then [mode(Modify,string (mkset (chars (mode Modify)) -- ['+']))]
  elsif member (chars (mode Modify)) '^'
  then [mode(Modify,"")]
  else [Modify].

OPNS do_recalc_mask :: options -> boolean.
EQNS do_recalc_mask Options =
  exist (missing_mask) Options && not (member Options (option "-n")).

OPNS missing_mask :: option -> boolean.
EQNS
  missing_mask (option "-m" A) =
    not (exist (type;member ["m","mask"]) A).
  missing_mask (option "-M" A) =
    not (exist (type;member ["m","mask"]) A).
  missing_mask (option "-x" A) =
    not (exist (type;member ["m","mask"]) A).
  missing_mask (option "-X" A) =
    not (exist (type;member ["m","mask"]) A).
 $missing_mask _ = false.

OPNS recalc_mask :: acl -> acl.
EQNS recalc_mask Acl =
  let
    Acl = remove (type;(==) acl_mask) Acl.
    GroupClass = filter (type;member [acl_user,acl_group,acl_group_obj]) Acl.
    GroupClassMode = string (mkset (foldr (mode;chars;(++)) ([]::chars) GroupClass) -- '-').
  in
    entry "mask" "" GroupClassMode : Acl.

END.
```

Anhang E. FTS: Sourcecode von `ft2dot`

```
/* Emacs, das ist -*- C -*-, und nicht Lisp
 *
 * ft2dot
 * Liest eine ft-Datei (Fault-Tree in ASCII-Darstellung) von stdin
 * und schreibt eine dot-Datei für graph-viz auf stdout
 *
 * Optionen:
 * -l <zahl> Länge, nach denen in den Labels umgebrochen wird
 * -t <font> Font für den Titel
 * -n <font> Font für die Knoten
 *
 * Bugs:
 * - Das Programm setzt eine syntaktisch korrekte ft-Datei
 * vorraus, ansonsten kann alles mögliche passieren
 * - Der Source ist extrem unverständlich
 *
 * $Id: ft2dot.l,v 1.2 2000/07/24 15:05:39 hjficker Exp $
 */

%{
#include <unistd.h>
#include <stdio.h>
    typedef struct _stack{
        int nodename;
        char *data;
        char *shape;
        struct _stack *next;
    } charstack;
    charstack* event_stack = 0;
    int akt = 0;
    void charstack_push(int, char *, char *);
    char *charstack_pop();
    void knoten(char*, char*);
    char *format_text(char *, int);

    int einrueckung=0, einrueckung1=0;
    int max_len = 20;
    char *title_font = "Helvetica-Bold";
    int title_font_size = 14;
    char *node_font = "Times-Roman";
    int node_font_size = 12;
%}

%%
#.*          /* Kommentar, nichts */
^\*[^*]*\*   { /* Top event */
               printf("\t\"0\" [label = \"\", shape=triangle]\n"
                      "\t\"0\" - \"%d\"\n", akt+1);
               knoten(yytext, "box"); }
\t           { einrueckung++; }
\n           { einrueckung = 0; }
\{[^}]*\}    { /* Titel */
               printf("\tlabel = \"\\r%s\\r\"\n", format_text(yytext, 0)); }
\[[^]]*\]    { knoten(yytext, "box"); }
```

```
\<[^>]*\>     { knoten(yytext, "diamond"); }
\([^)]*\)     { knoten(yytext, "ellipse"); }
\*[^*]*\*     { knoten(yytext, "triangle"); }
\|\|          { printf("\t\"%d\" [label = \">=1\", shape=\"box\"]\n", ++akt);
                printf("\t\"%d\" - \"%d\"\n",
                       event_stack->nodename, akt);
                charstack_pop();
                charstack_push(akt, ">=1", "box"); }
&&            { printf("\t\"%d\" [label = \"&\", shape=\"box\"]\n", ++akt);
                printf("\t\"%d\" - \"%d\"\n", event_stack->nodename, akt);
                charstack_pop();
                charstack_push(akt, "&", "box"); }


%%


int main(int argc, char **argv) {
    int option;
    while ((option = getopt(argc, argv, "l:t:n:")) != EOF) {;
        switch (option) {
        case ':':
            fprintf(stderr, "Fehlendes Argument\n");
            exit(1);
        case '?':
            fprintf(stderr, "Unbekannte Option\n");
            exit(1);
        case 'l':
            max_len=atoi(optarg);
            break;
        case 't':
            title_font = optarg;
            break;
        case 'n':
            node_font = optarg;
            break;
        default:
            fprintf(stderr, "Hä?\nOption = %d\n", option);
            exit(1);
        }
    }
    printf("graph G {\n"
           "\tgraph [\n"
           "\t\tfontname = \"%s\"\n"
           "\t\tfontsize = \"%d\"\n"
           "\t]\n", title_font, title_font_size);
    yylex();
    printf("}\n");
    return 0;
}


/*
 * Operationen push und pop auf dem Stack
 */
void charstack_push(int node, char *text, char *shape) {
    charstack *tmp_stack;

    tmp_stack = malloc(sizeof(charstack));
    tmp_stack->nodename = node;
    tmp_stack->data = text;
```

```
        tmp_stack->shape = shape;
        tmp_stack->next = event_stack;
        event_stack = tmp_stack;
}


char *charstack_pop() {
        char *tmp_text;
        charstack *tmp_stack;
        if (!event_stack) return 0;
        tmp_stack = event_stack;
        event_stack = event_stack->next;
        tmp_text = tmp_stack->data;
        free(tmp_stack);
        return tmp_text;
}


/*
 * Formatiert den Text um.
 * - Konvertiert Umlaute (ü -> ue usw.)
 * - fügt durch "\" verbundene Zeilen zusammen
 * - Fügt bei Bedarf Zeilenumbrüche ein
 */
char *format_text(char *name, int max_len) {
        char *cpy;
        int i;

        /* Für jeden Zeilenumbruch brauchen wir ein zusätzliches
           Zeichen. Für mögliche Umlaute auch noch etwas Platz
           (hoffenlich kommen nicht auf mehr als 20 Umlaute ;-) */
        if (max_len)
                cpy = malloc(strlen(name) + strlen(name)/max_len + 21);
        else
                cpy = malloc(strlen(name) + 21);

        /* Das erste und letzte Zeichen wollen wir nicht abspeichern */
        strncpy(cpy, name + 1, strlen(name) - 2);
        cpy[strlen(name)-2] = 0;

        /* Als erstes Umlaute konvertieren (dotty versteht zwar Umlaute,
           trägt sie aber im Postscript nicht richtig ein */
        for (i = 0; i < strlen(cpy); i++) {
                switch (cpy[i]) {
                case 'ü':
                        memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
                        cpy[i] = 'u';
                        cpy[++i] = 'e';
                        break;
                case 'Ü':
                        memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
                        cpy[i] = 'U';
                        cpy[++i] = 'e';
                        break;
                case 'ö':
                        memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
                        cpy[i] = 'o';
                        cpy[++i] = 'e';
                        break;
                case 'Ö':
                        memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
```

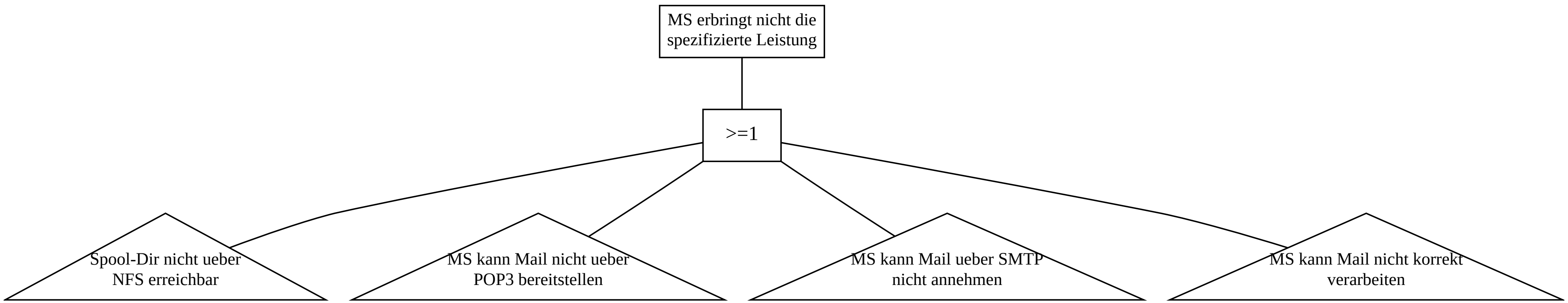
```
            cpy[i] = 'O';
            cpy[++i] = 'e';
            break;
        case 'ä':
            memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
            cpy[i] = 'a';
            cpy[++i] = 'e';
            break;
        case 'Ä':
            memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
            cpy[i] = 'A';
            cpy[++i] = 'e';
            break;
        case 'ß':
            memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
            cpy[i] = 's';
            cpy[++i] = 's';
            break;
        case '"':
            /* Ist zwar kein Umlaut, aber das Prinzip ist ähnlich,
               Wir müssen Anführungzeichen escapen */
            memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);
            cpy[i] = '\\';
            cpy[++i] = '"';
            break;
        case '\\':
            /* Behandlung von \ am Zeilenende. Paßt vom Konzept
               her (Umlaute erweitern) nicht genau hierhin, aber
               diese Stelle bietet sich an, weil wir sowieso durch
               den String durchgehen, und dabei verändern */
            if (cpy[i+1] == '\n') {
                /* Alle Tabs/Leerzeichen überspringen */
                int j = i + 2;
                while (cpy[j] == '\t' || cpy[j] == ' ') j++;
                memmove(cpy+i, cpy+j, strlen(cpy)-j+1);
            }
        }
}

/* max_len = 0 -> Kein Umbruch */
if (max_len == 0)
    return cpy;

/* Jetzt passende Umbrüche einfügen */
i = max_len;
while (i < strlen(cpy)) {
    /* Nächstes Leerzeichen suchen */
    while (cpy[i] != 0 && cpy[i] != ' ') i++;

    if (cpy[i] == 0) break;

    /* Mehr Platz machen */
    memmove(cpy+i+1, cpy+i, strlen(cpy)-i+1);

    /* Umbruch einfügen */
    cpy[i] = '\\';
    cpy[++i] = 'n';

    /* Weiter in 20 Zeichen */
    i += max_len;
}
```

```
        return cpy;
}


/*
 * Schreibt einen neuen Knoten, und die Verbindung zum
 * übergeordneten Knoten
 */
void knoten(char *name, char *shape) {

        name = format_text(name, max_len);

        /* Alle Daten auf dem Stack mit größerer oder gleicher
           Einrückung wegwerfen (brauchen wir nicht mehr) */
        while (einrueckung1 >= einrueckung) {
                einrueckung1 --;
                charstack_pop();
        }
        if (einrueckung == einrueckung1 + 1) {
                akt++;
                if (event_stack) {
                        /* Kante zur höheren Ebene */
                        printf("\t\"%d\" - \"%d\"\n",
                                event_stack->nodename,
                                akt);
                }

                /* Den Knoten mit passenden Attributen einfügen */
                printf("\t\"%d\" [\n"
                        "\t\tlabel = \"%s\"\n"
                        "\t\tshape=\"%s\"\n"
                        "\t\tfontname=\"%s\"\n"
                        "\t\tfontsize=\"%d\"\n"
                        "\t]\n",
                        akt, name, shape, node_font, node_font_size);
                einrueckung1 = einrueckung;

                /* Und euen Knoten auf den Stack schreiben */
                charstack_push(akt, name, shape);
        }
}

int yywrap() {
        /* Wir wollen nur eine Datei lesen, keine zweite */
        return 1;
}

/*
 * Changelog:
 * $Log: ft2dot.l,v $
 * Revision 1.2 2000/07/24 15:05:39 hjficker
 * Behandlung von \ am Zeilenende verbessert
 *
 * Revision 1.1 2000/03/09 20:04:29 rscholz
 * FTA-Automatismen für FTS, bitte Mail lesen!
 *
 * Revision 1.2 1999/07/01 11:35:42 hjficker
 * - Returnwert 0 im Erfolgsfall (wg Make)
 * - Escaping von Anführungszeichen
 *
 * Revision 1.1 1999/06/28 13:51:42 hjficker
```

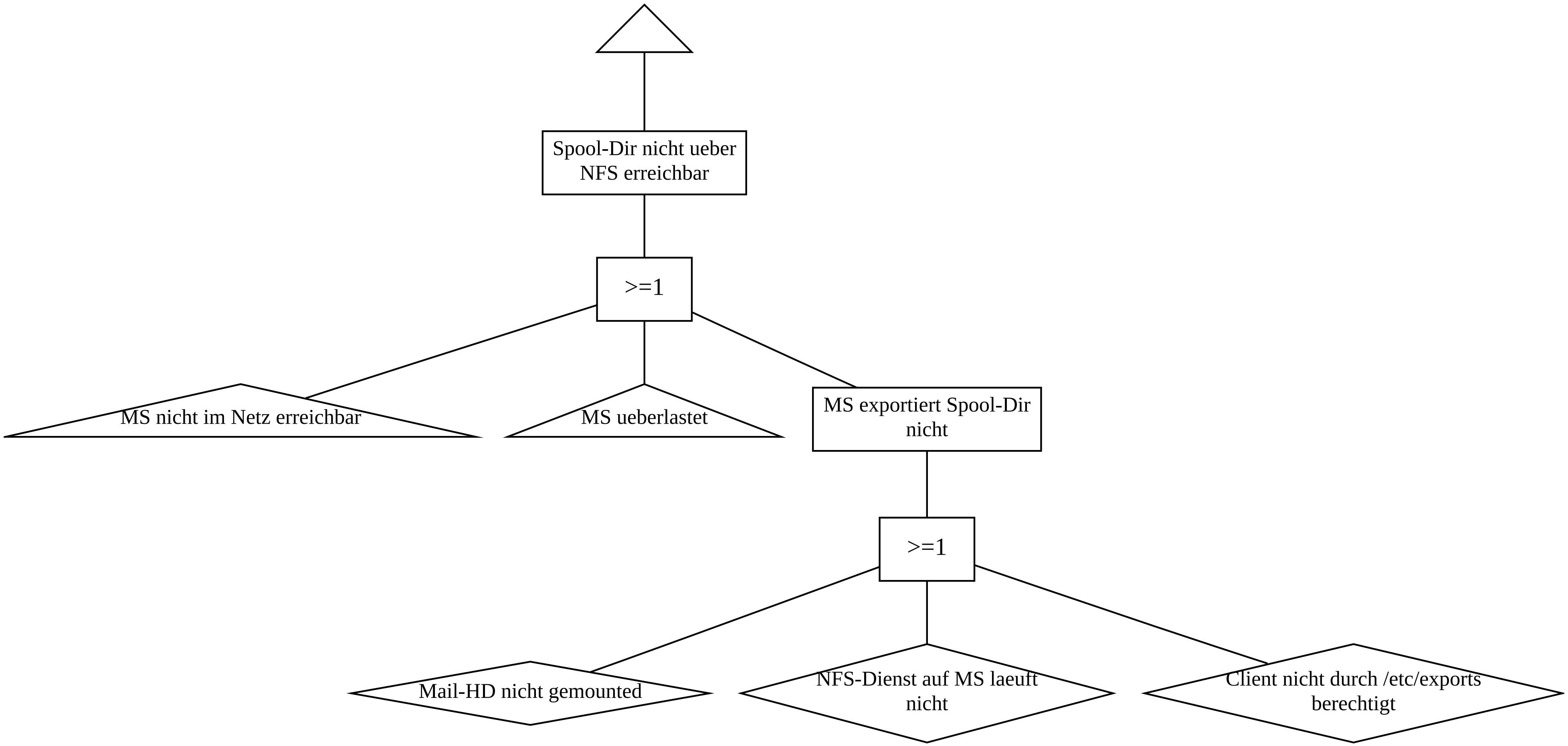
∗ *Nach langer Entwicklung endlich ins CVS eingecheckt*
∗
∗/

Anhang F. FTS: Fault Trees des alten Mailservers

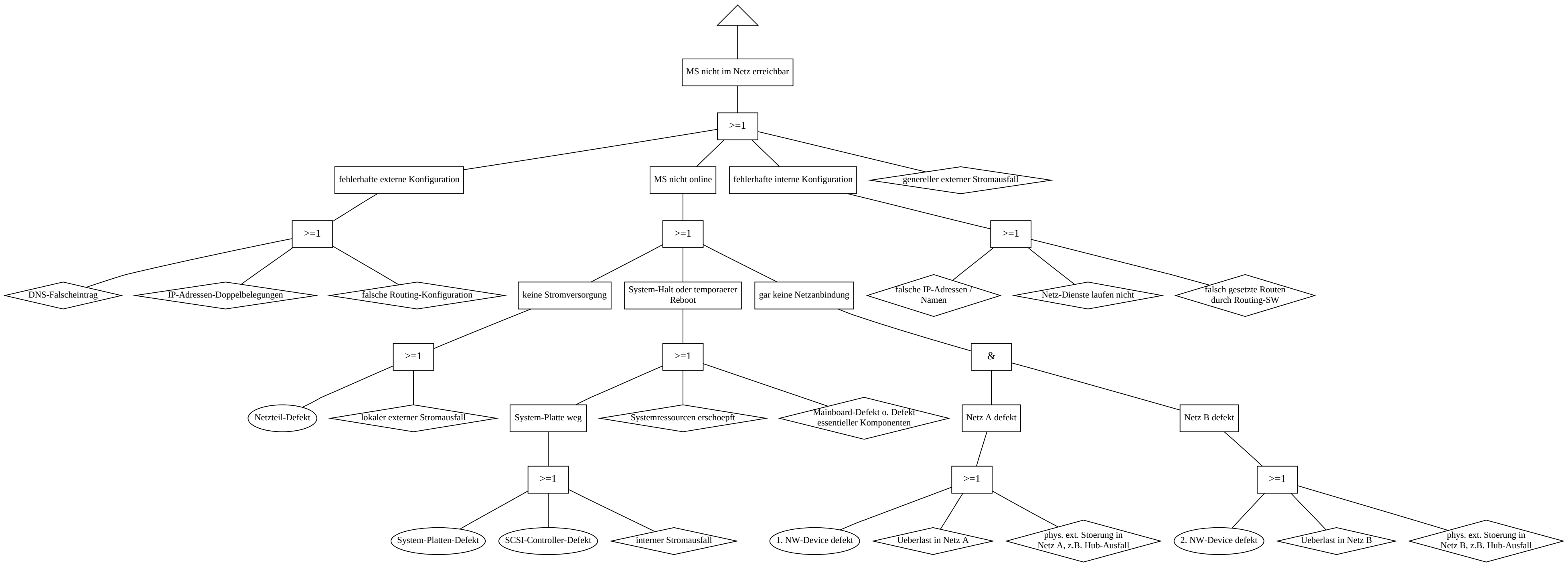
Aufgrund layouttechnischer Beschränkungen sind einige der nun folgenden Fault Trees nur stark verkleinert wiedergegeben.



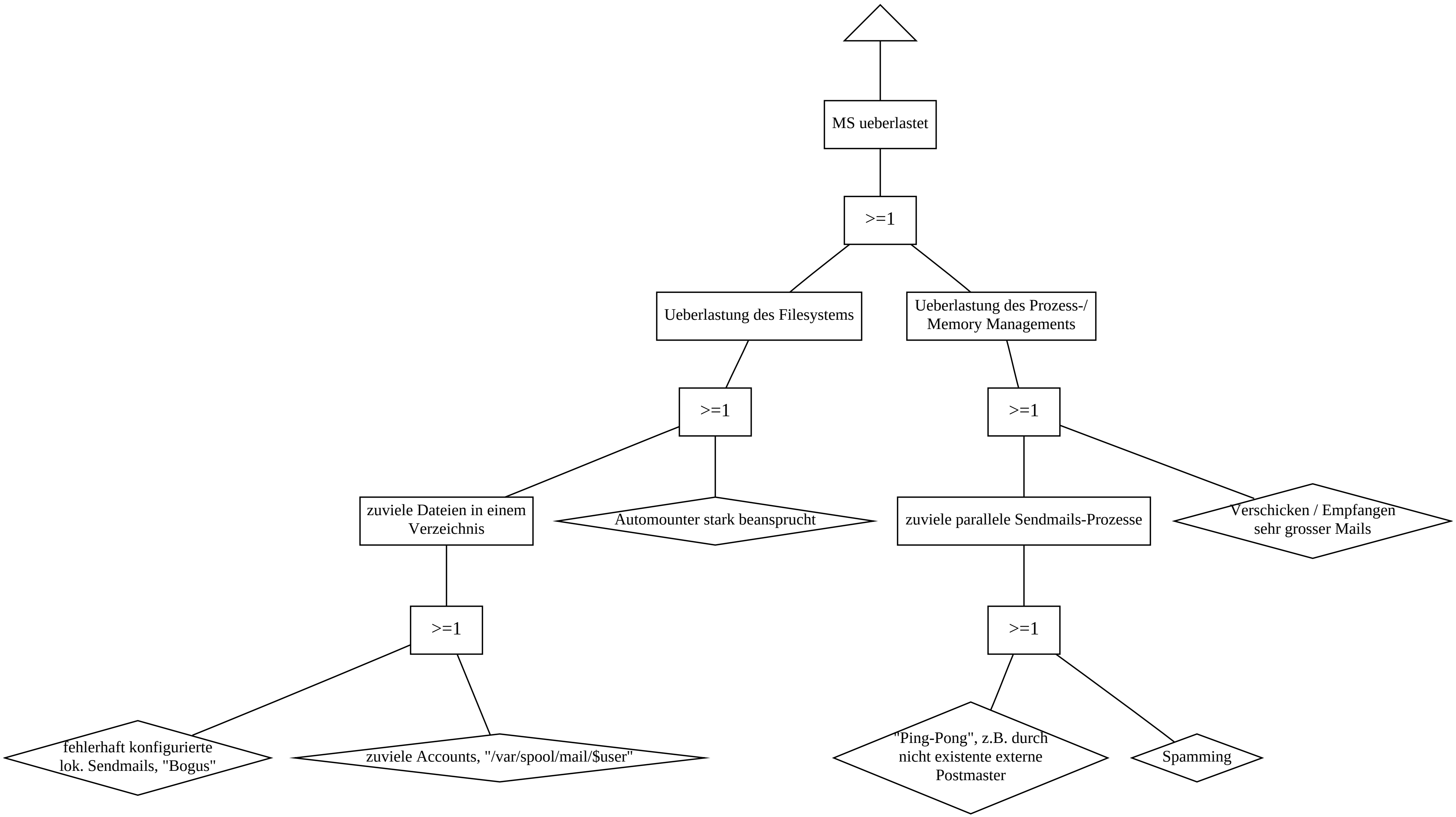
alter Mailserver: MS erbringt nicht die spezifizierte Leistung



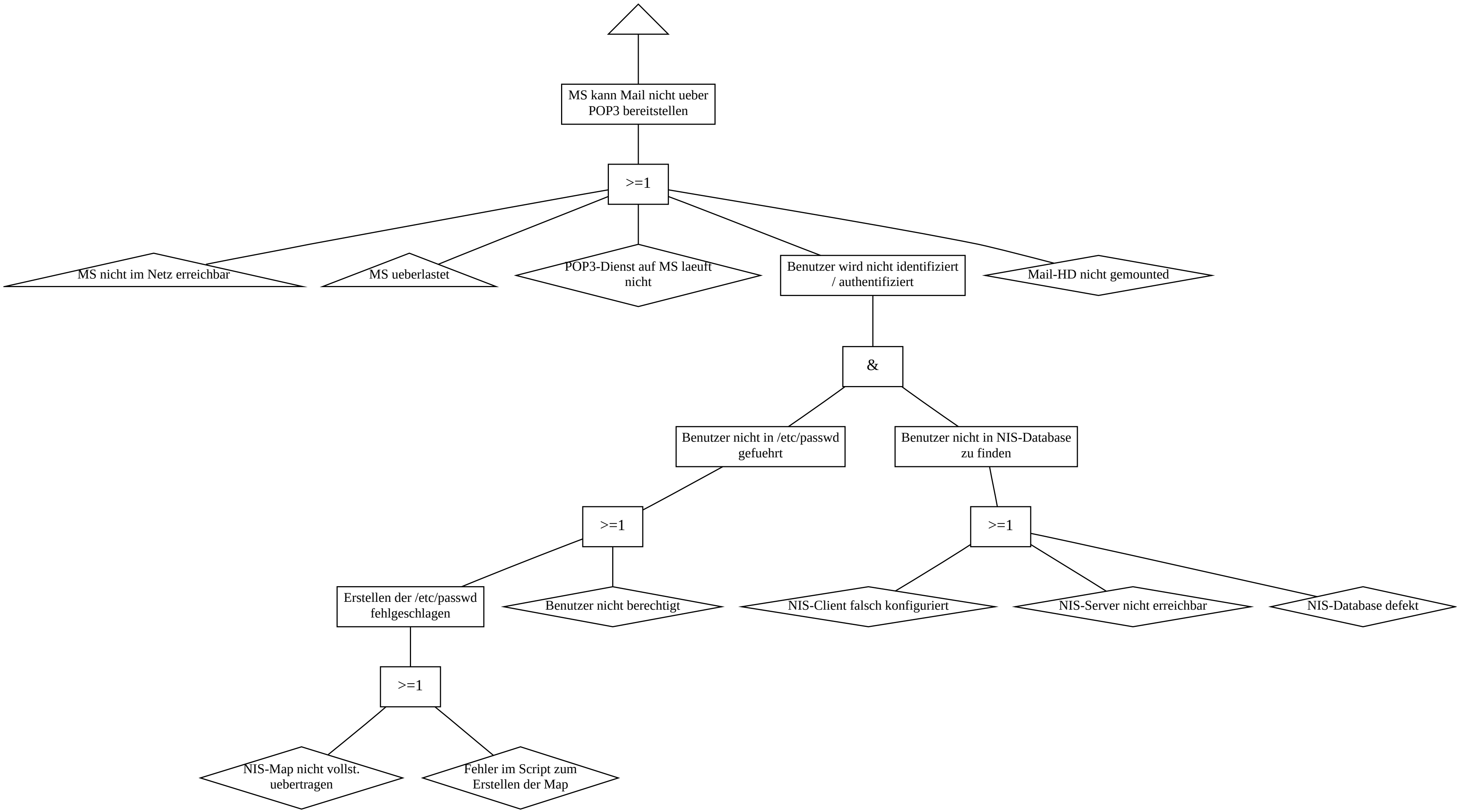
alter Mailserver: Spool-Dir nicht ueber NFS erreichbar



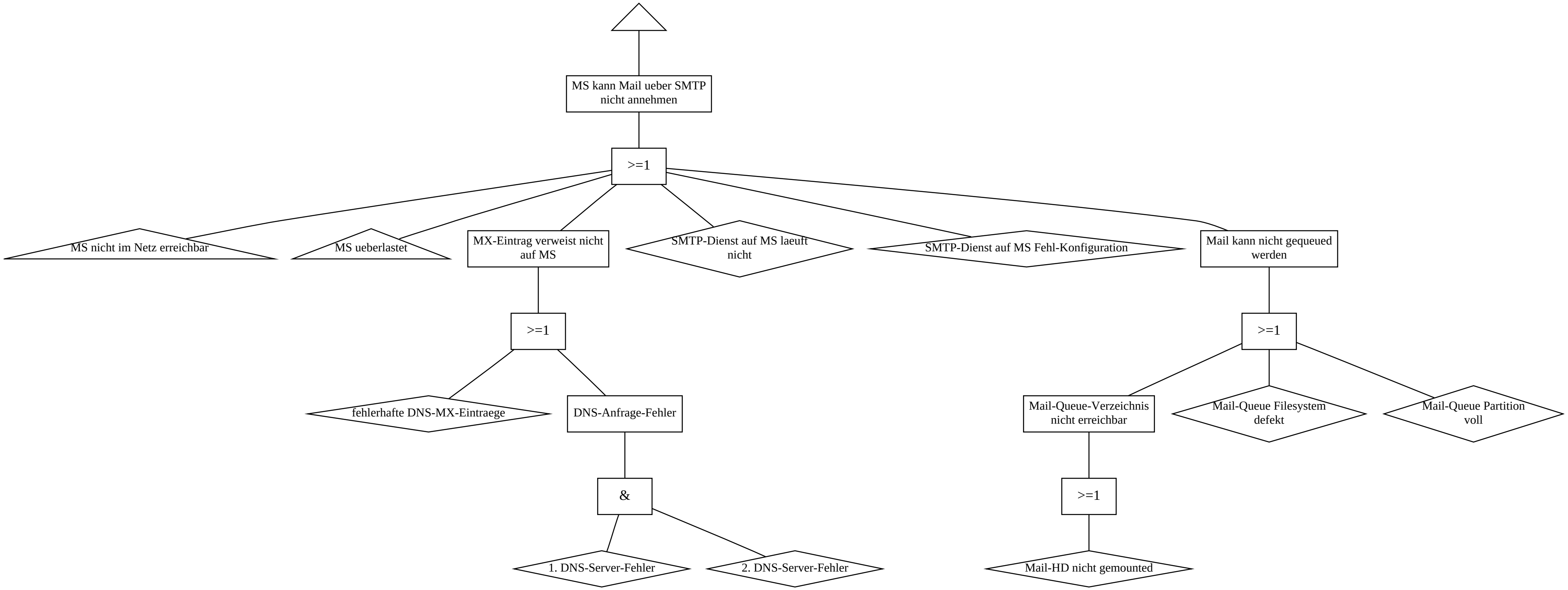
alter Mailserver: MS nicht im Netz erreichbar



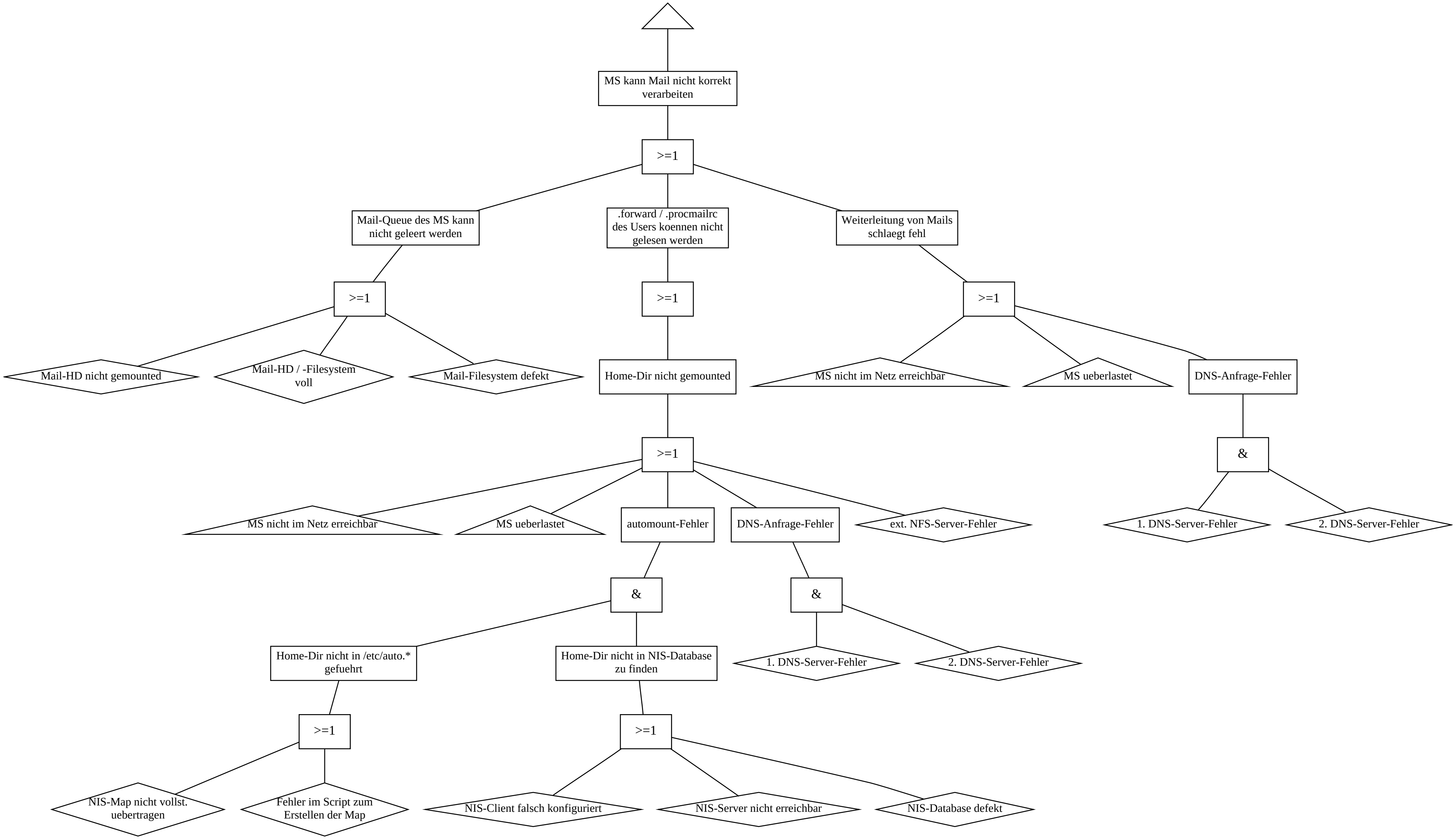
alter Mailserver: MS ueberlastet



alter Mailserver: MS kann Mail nicht ueber POP3 bereitstellen

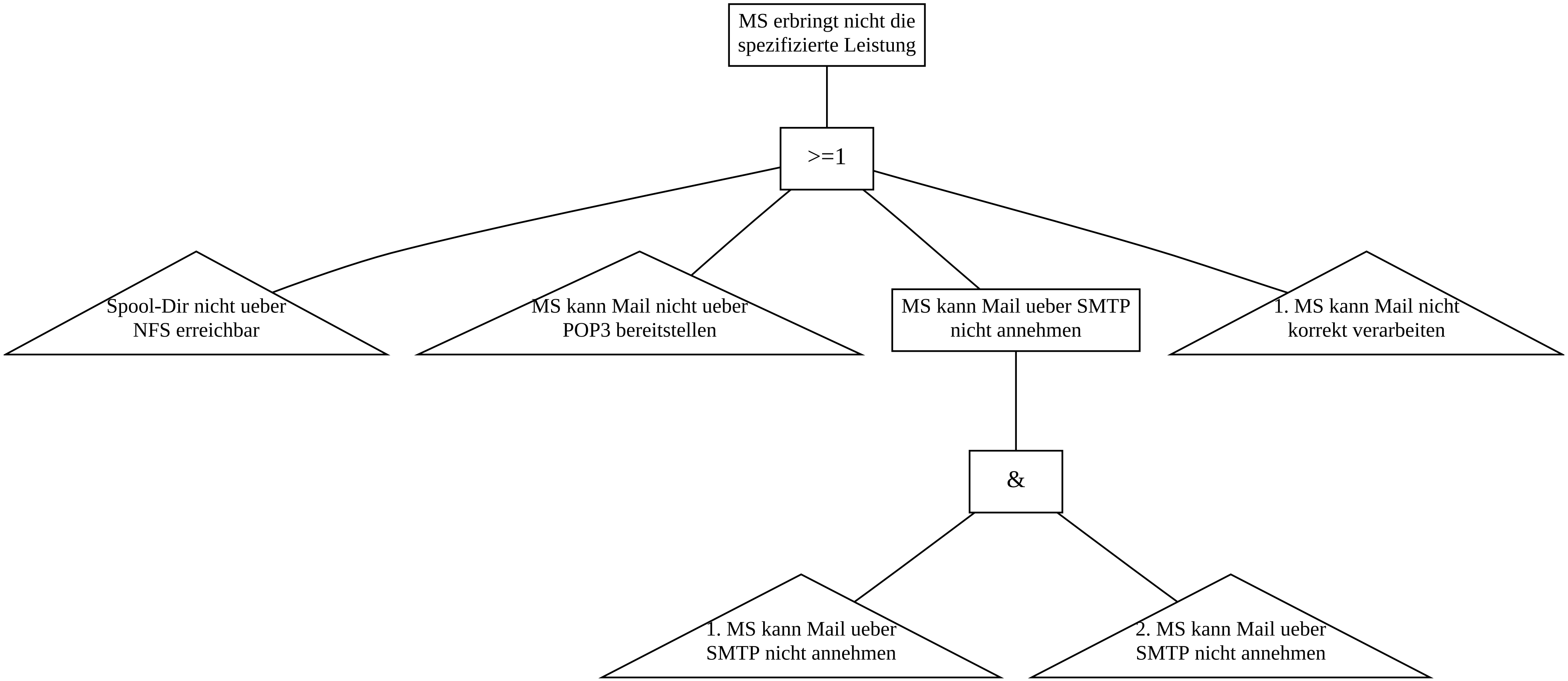


alter Mailserver: MS kann Mail ueber SMTP nicht annehmen

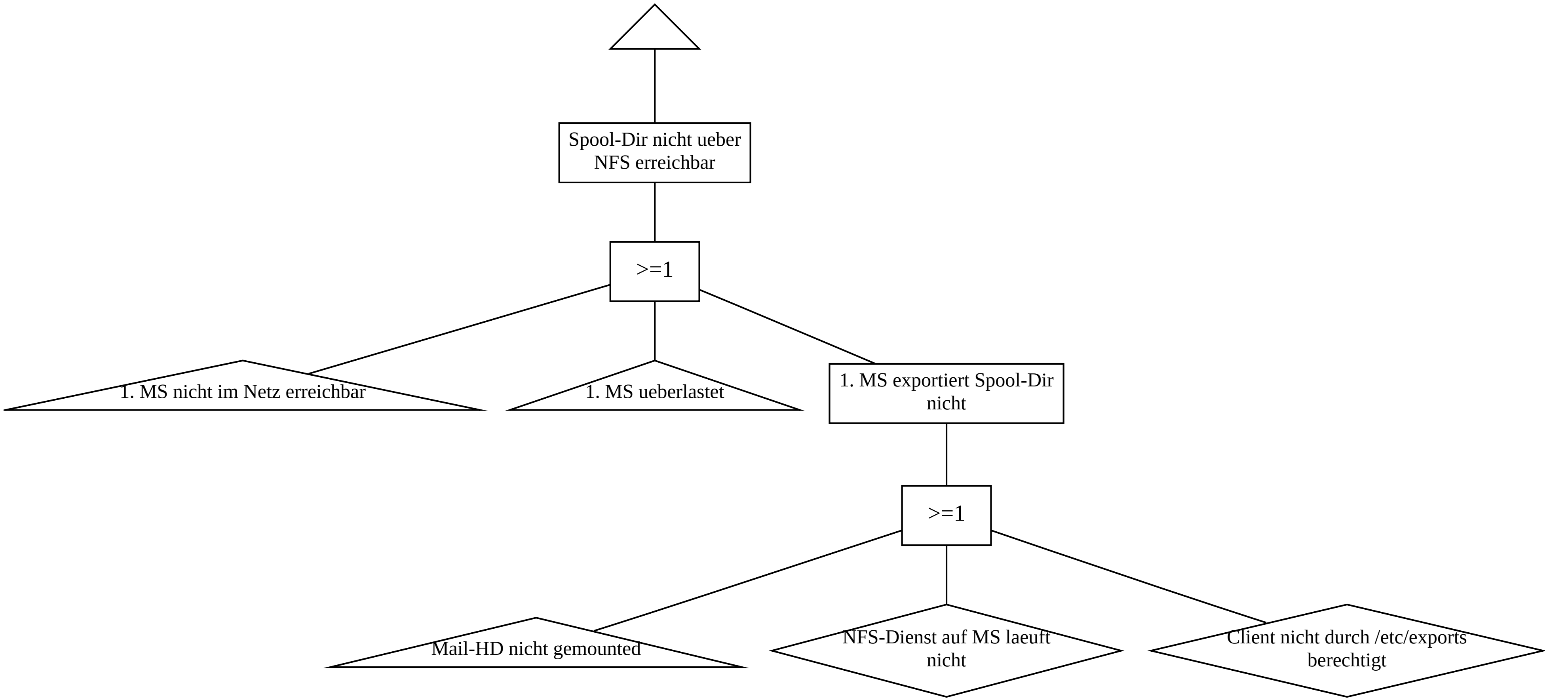


alter Mailserver: MS kann Mail nicht korrekt verarbeiten

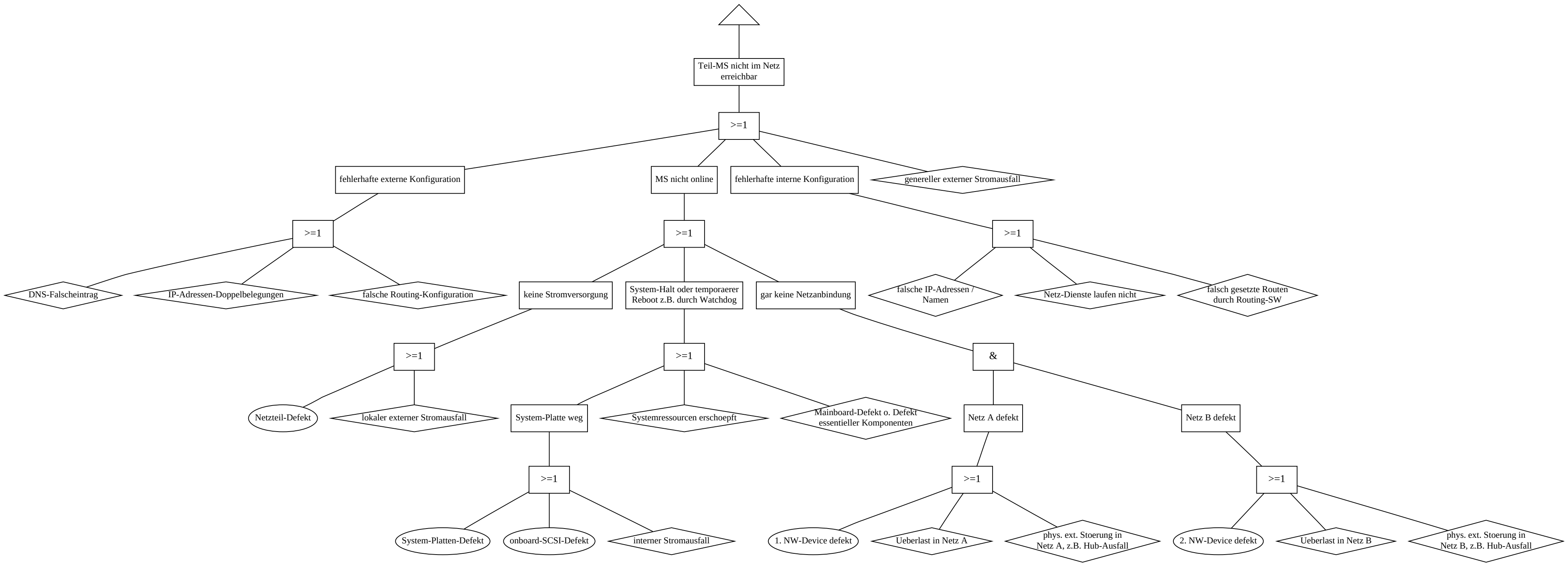
Anhang G. FTS: Fault Trees des neuen Mailservers



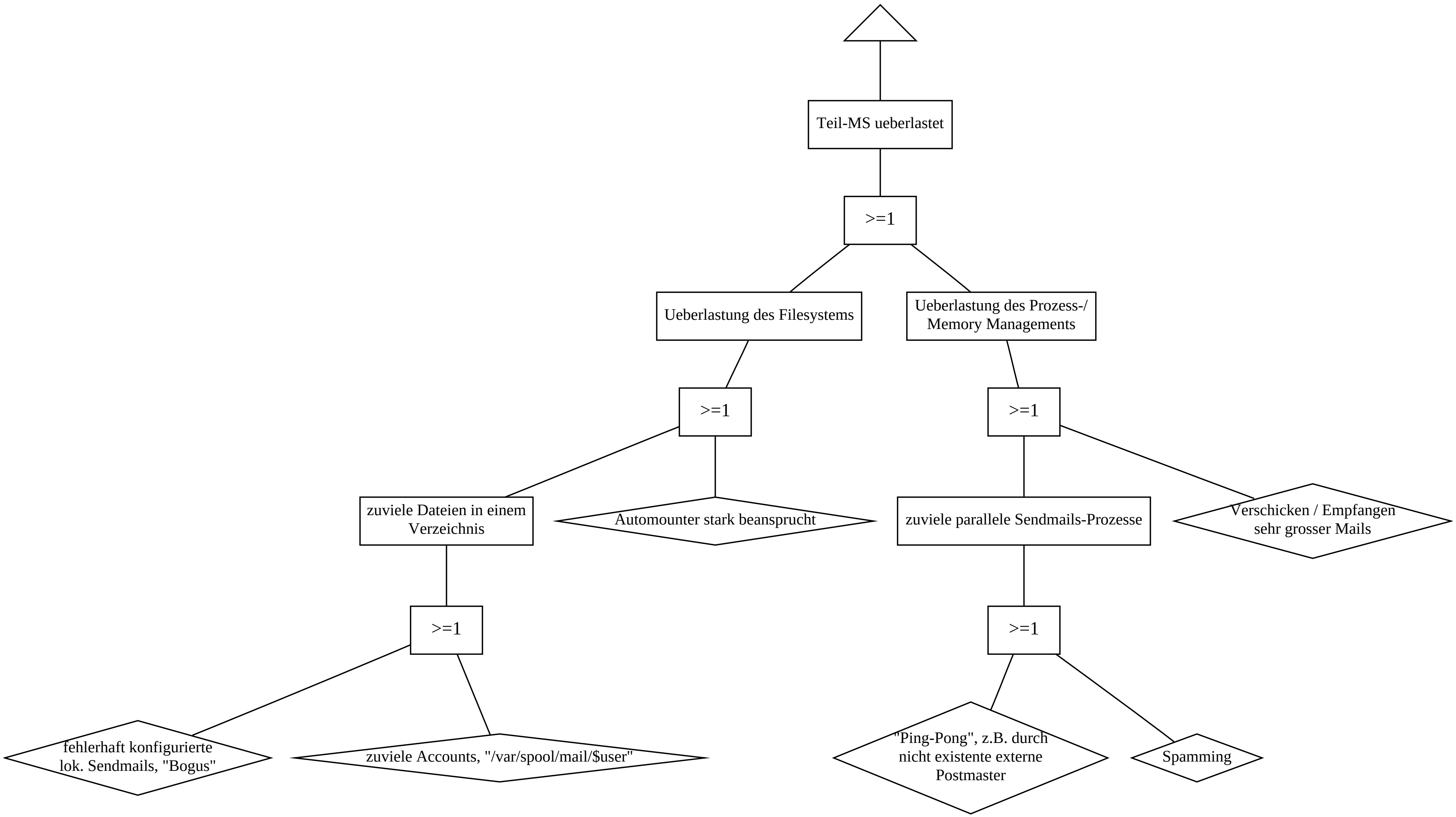
FTS-Mailserver: MS erbringt nicht die spezifizierte Leistung



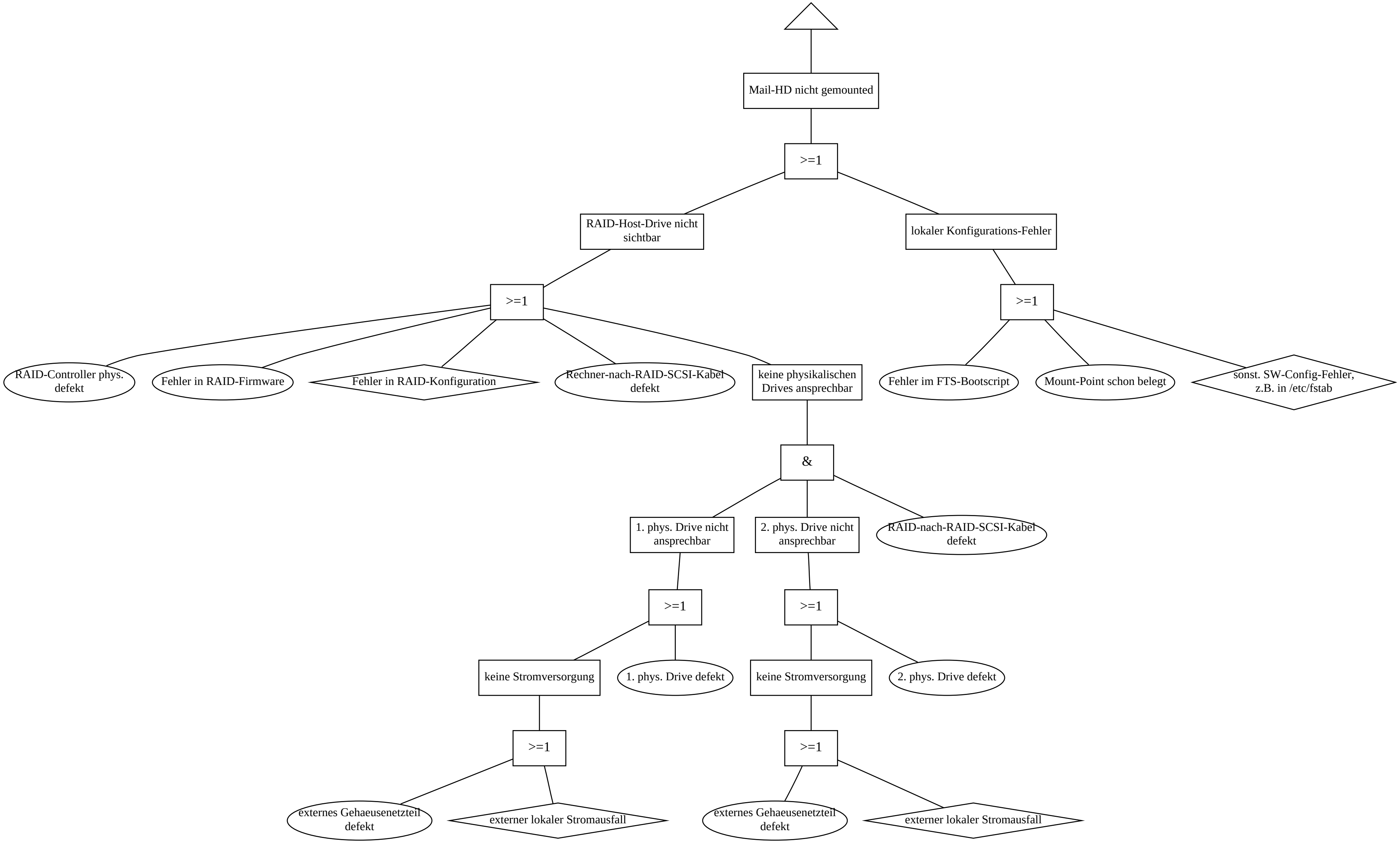
FTS-Mailserver: Spool-Dir nicht ueber NFS erreichbar



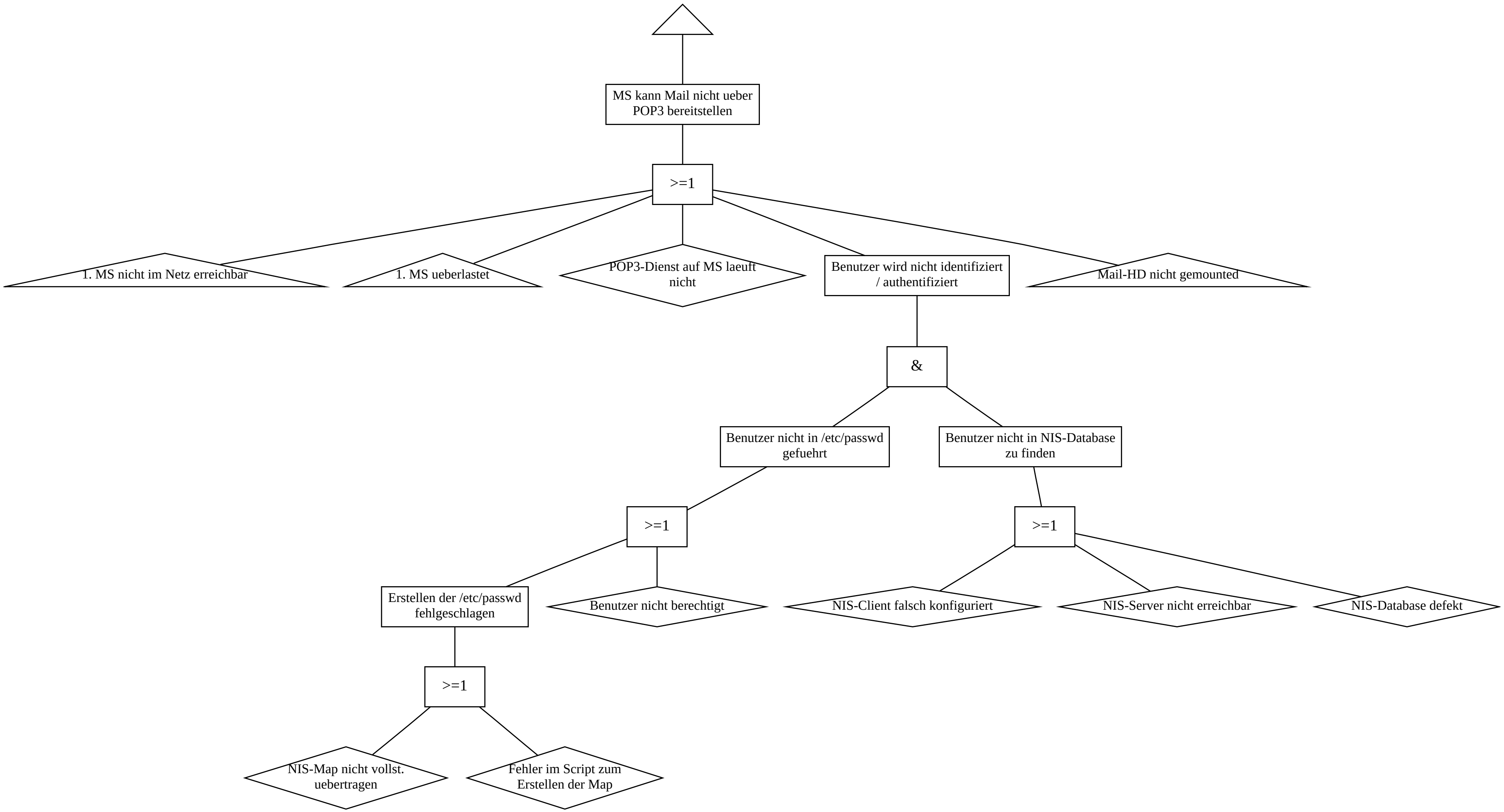
FTS-Mailserver: Teil-MS nicht im Netz erreichbar



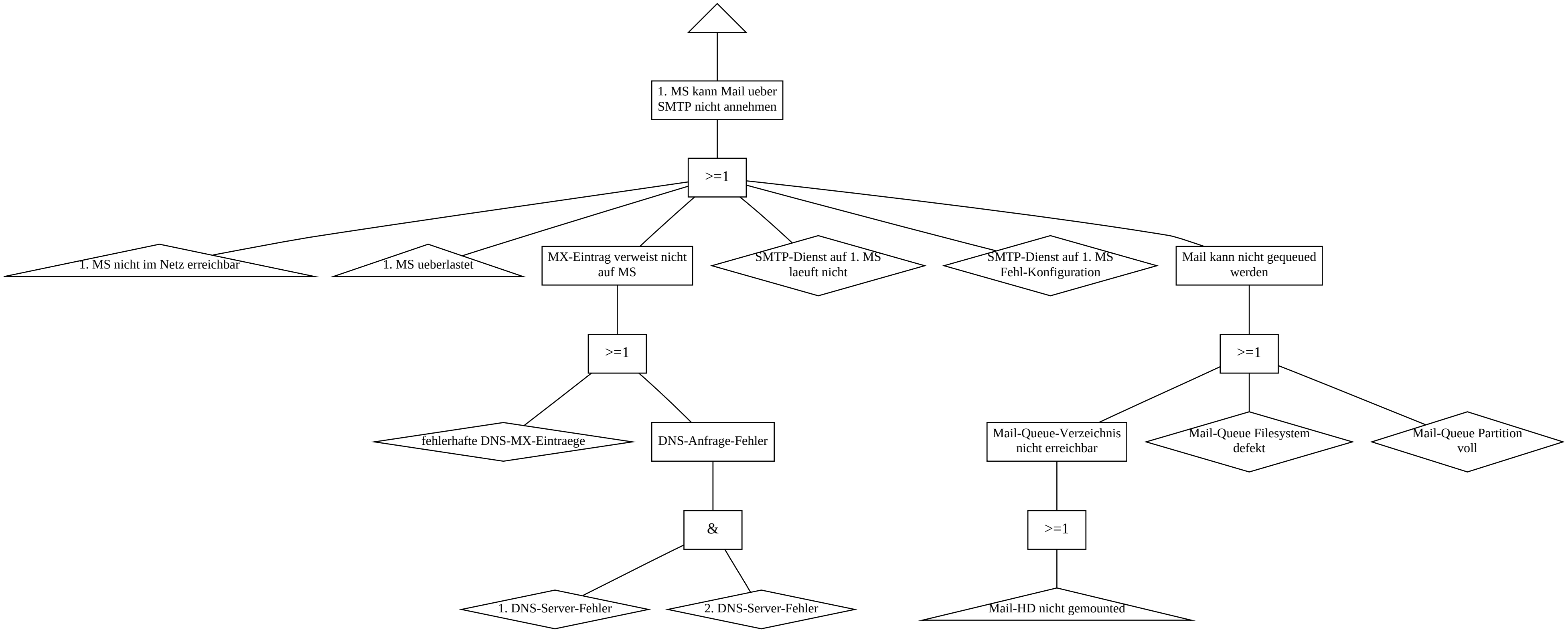
FTS-Mailserver: Teil-MS ueberlastet



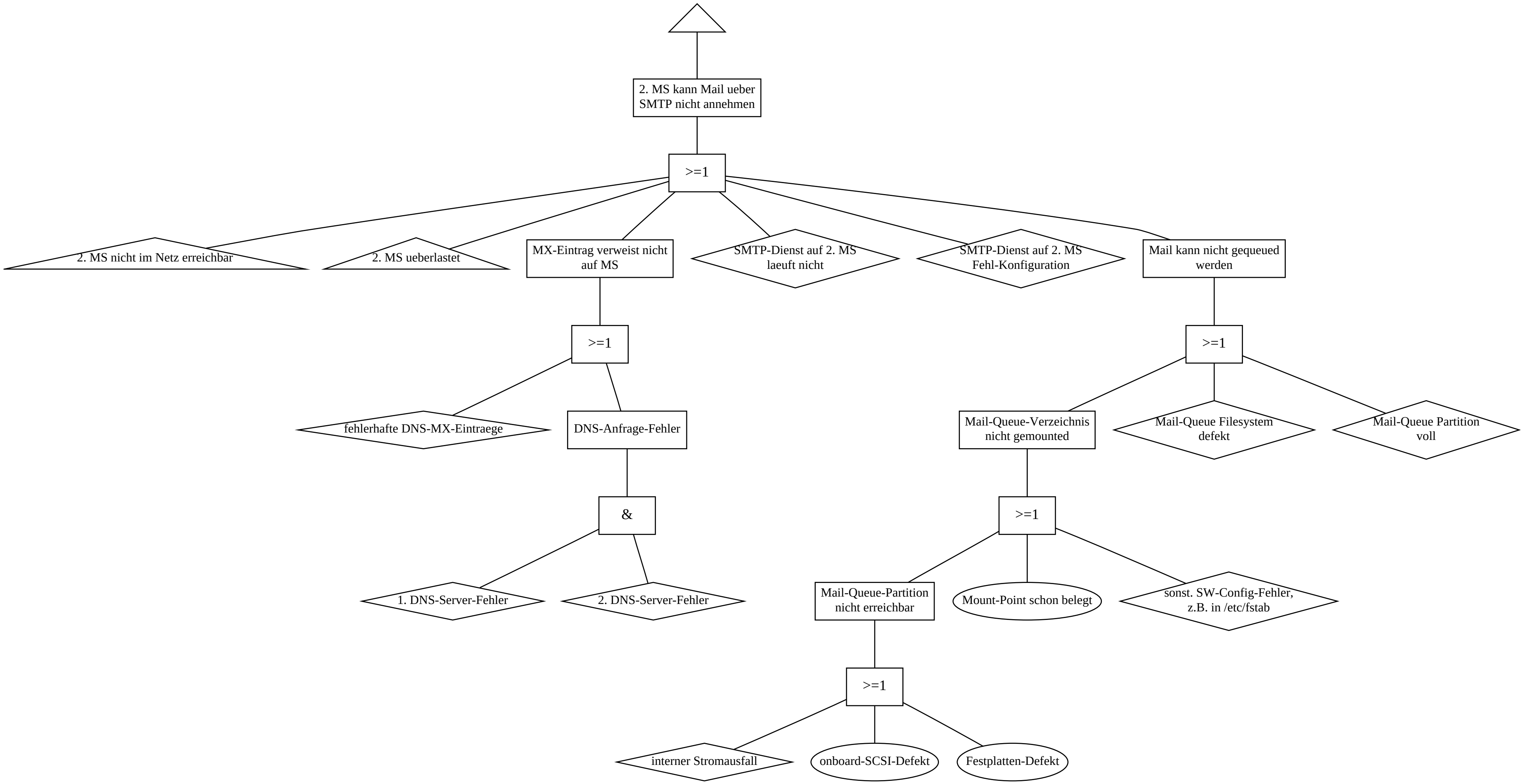
FTS-Mailserver: Mail-HD nicht gemounted



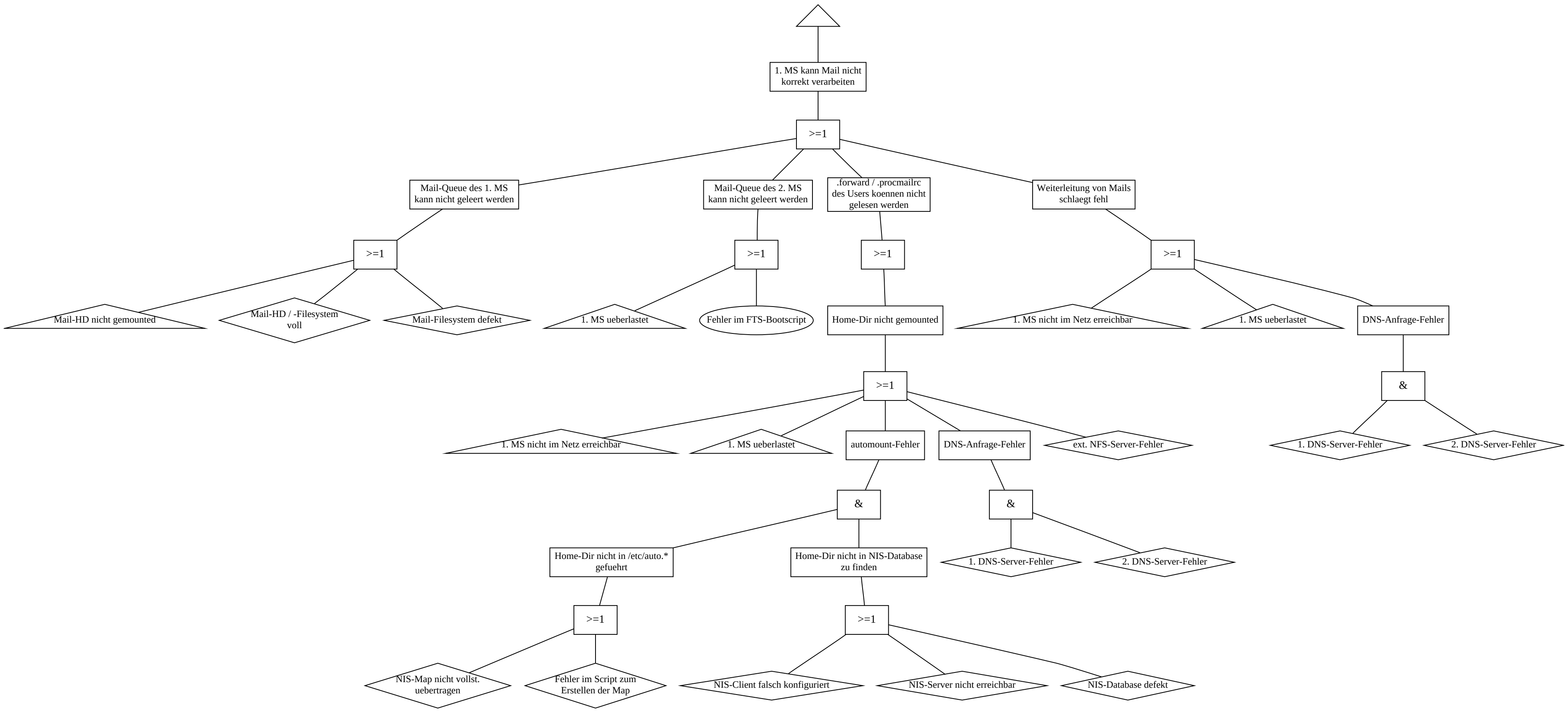
FTS-Mailserver: MS kann Mail nicht ueber POP3 bereitstellen



FTS-Mailserver: 1. MS kann Mail ueber SMTP nicht annehmen



FTS-Mailserver: 2. MS kann Mail ueber SMTP nicht annehmen



FTS-Mailserver: 1. MS kann Mail nicht korrekt verarbeiten

Anhang H. FTS: Konfiguration des RT–Testers

H.1 CSP–Spezifikation

```
-----------------------------------------------------------
-- FTS-MS.CSP
--
-- $Id: fts-ms.csp,v 1.1 2000/07/31 14:30:49 hjficker Exp $
--
-- Fault Tolerant System - Mail Server
--
-- Test-SPEC for RT-Tester
-----------------------------------------------------------

----------------
-- KONSTANTEN
----------------

-- symbolische Namen fuer Mailserver
PRI_MS = 0
SEC_MS = 1

------------------------------------
-- Datentypen
------------------------------------

-- Rueckgabewerte der Methoden
datatype result = ok | nok

-- Benutzer
user_snd = {0,1,2,3,4,5,6,7,8,9,10,11,12}
user_rcv = {0,1,2,3,4,5,6,7,8,9,10}

-- user_snd = {0}
-- user_rcv = {0}

-- Server
what_ms = {0,1}

-- Timer
timer_id = {0,1,2,3}

-- Warnings
datatype war_id = war_smtp_timeout_pri | war_smtp_timeout_sec |
                  war_pop | war_pop_timeout | war_nfs |
                  war_nfs_timeout | war_noms
-- Errors
datatype err_id = err_smtp_pri | err_smtp_sec

------------------
-- Timer dieser AM
------------------

--$$AM_SET_TIMER
channel setT : timer_id
--$$AM_ELAPSED_TIMER
channel elaT : timer_id
--$$AM_RESET_TIMER
channel resT : timer_id
```

```
--------------------
-- Errors of this AM
--------------------

--$$AM_ERROR
channel err : err_id

---------------------
-- Warnings dieser AM
---------------------

--$$AM_WARNING
channel war : war_id

--------------------
-- Outputs dieser AM
--------------------

--$$AM_OUTPUT
channel smtp_write : what_ms.user_rcv.user_snd
channel pop_read   : user_rcv
channel nfs_read   : user_rcv

------------------
-- Input dieser AM
------------------

--$$AM_INPUT
channel smtp_ack : result
channel pop_ack  : result
channel nfs_ack  : result

channel whatms : what_ms

------------------
-- Undefined Data
------------------

--$$AM_UNDEF
channel undefData

-----------------------------
-- internal events of this AM
-----------------------------

--$$AM_INTERNAL

channel tellms : what_ms

---------------------------
-- Prozesse
---------------------------

GETMS(ms) =

(

  whatms?ms -> GETMS(ms)

[]

  tellms!ms -> GETMS(ms)

)
```

```
-------------
-- USERACTION
-------------


USERACTION =

(


SMTP ; USERACTION

|~|

POP  ; USERACTION

|~|

NFS ; USERACTION

)


-------
-- SMTP
-------

SMTP =

( tellms?ms ->

 |~| u_from: user_rcv, u_to: user_snd @
        smtp_write!ms.u_from.u_to -> setT.1 ->

 (
  smtp_ack.ok -> resT.1 -> SKIP
  []
  smtp_ack.nok -> resT.1 -> err!err_smtp_pri -> SKIP
  []
  elaT.1 -> war!war_smtp_timeout_pri -> SKIP
  )

)

------
-- POP
------

POP =

(

 |~| u_from: user_rcv @

 pop_read!u_from -> setT.2 ->

 (
  pop_ack.ok -> resT.2 -> SKIP
  []
  pop_ack.nok -> resT.2 -> war!war_pop -> SKIP
  []
  elaT.2 -> war!war_pop_timeout -> SKIP
 )
)

------
```

```
-- NFS
------

NFS =

(

 |~| u_from: user_rcv @

 nfs_read!u_from -> setT.3 ->

 (
  nfs_ack.ok -> resT.3 -> SKIP
  []
  nfs_ack.nok -> resT.3 -> war!war_nfs -> SKIP
  []
  elaT.3 -> war!war_nfs_timeout -> SKIP
 )
)


FTS = whatms?ms -> (USERACTION [| {| tellms |} |] GETMS(ms) \ {| tellms |})
```

H.2 Erzeugung von Konfigurationsdateien

H.2.1 Makefile

```
#
# Config-Makefile
#
# Erzeugt die Config-Dateien fuer den Testlauf, die von
# den Alphabet-Dateien abhaengen und manuell zu aufwendig
# zu erzeugen sind.
#
# $Id: Makefile_conf,v 1.1 2000/08/02 11:42:12 hjficker Exp $
#

NUM_FTS=10
FTS_OFFSET=2


all:	config trans
	@echo "Done!"


trans:
	TGgen fts-ms FTS
	TGgen fts-env ENV


config:
	makevvtc $(NUM_FTS) $(FTS_OFFSET) > vvtconfig.txt
	cat fts-ms.t | makeamccl > amccl.txt
	cat fts-ms.t | makeaeraw $(NUM_FTS) $(FTS_OFFSET) > aeraw.txt
	cat fts-ms.t | makeabs $(NUM_FTS) $(FTS_OFFSET) > abseventmaps.txt
	AM_confs.sh vvtconfig.txt abseventmaps.txt aemae.txt
```

H.2.2 makevvtc

```
#!/bin/sh
#
# makevvtc
#
# Dieses Skript erzeugt nach Angabe der Anzahl
# der gewuenschten AMs und dem Offset der FTS-AMs
# die Datei vvtconfig.txt
#
# $Id: makevvtc,v 1.2 2000/08/03 14:15:47 hjficker Exp $
#

test $# != 2 && {
    echo "Aufruf: makevvtc number_of_ams am_offset > vvtconfig.txt";
    exit 2;
  } || { NUM_FTS=$1; FTS_OFFSET=2; }

cat vvtconfig.head

TMP=$FTS_OFFSET

while [ $TMP -lt $(($NUM_FTS+$FTS_OFFSET+1)) ] ; do

cat <<EOF
AM $TMP                                # fts-ms abstract machine
  AMSHMID       $TMP                   # Shared memory ID
  CVMID         111                    # CCL-ID of the continuous value part
  ALPHABETFILE  /conf/fts-conf/fts-ms.t # Name of the alphabet file incl.
                                       # path from VVTHOME
  TRANSFILE     /conf/fts-conf/fts-ms.trans  # Name of the transition file
  STATEFILE     /conf/fts-conf/fts-ms.state  # Name of the state file
  REFFILE       /conf/fts-conf/fts-ms.ref    # Name of the refusals file
  EVENTFILE     /conf/fts-conf/fts-ms.event  # Name of the event file
  TRACEFILE     /conf/fts-conf/fts-ms.trace  # Name of the trace file that
                                             # is generated by VVT-RT
  CASESFILE     /conf/fts-conf/fts-ms.tst    # Name of the cases file that
                                             # is generated by VVT-RT
  CASESLTXFILE  /conf/fts-conf/fts-ms.ltx    # Name of the LaTeX cases file
                                             # that is generated by VVT-RT
  RX_ADDR       111                    # Internal ID of the CCL-RX-Process
  RX_ADDR       112                    # Internal ID of the CCL-RX-Process
  TX_ADDR       111

  SEED          $((40+$TMP))                   # random seed for this AM
  STOPONIOERROR NO                     # do not stop if an unexpected
                                       # output occurs
  VERBOSE       YES                    # display unexpected outputs

  # Timer definitions: No:LowerBound:UpperBound
  TIMER 0:10000:-                      # Wait for answer of whatms
        1:120000:-                     # Wait for answer SMTP
        2:120000:-                     # Wait for answer POP
        3:120000:-                     # Wait for answer NFS


END

EOF

    TMP=$(($TMP+1))
done
```

```
#
# Changelog:
# $Log: makevvtc,v $
# Revision 1.2  2000/08/03 14:15:47  hjficker
# Anhänge korrigiert. Overfull hboxes u.s.w.
#
# Revision 1.1  2000/08/02 11:42:12  hjficker
# Änderungen im Anhang (pretty-printing, übern Rand geschrieben,
# Teilprojekt-Namen dazugeschrieben)
#
# Revision 1.2  1999/09/10 12:53:28  rscholz
# neue Timer fuer SMTP etc., Generieren der Transitionsgraphen im Makefile
#
# Revision 1.1  1999/09/10 09:34:44  rscholz
# dynam. vvtconfig.txt
#
```

H.2.3 vvtconfig.head

```
# Configuration File for VVT-RT
#
# $Id: vvtconfig.head,v 1.3 2000/08/03 14:15:47 hjficker Exp $
#
# IDs 111 (RX) und 112 (RX,TX) are reserved for CCL processes
# CVMID and AMID numbers must not be the same (unique IDs)

GLOBAL                     # Global definitions for all abstract machines
  CTRLSHMID     301        # Shared memory ID of the control structure
  CVSHMID       302        # Shared memory ID of the cv block
END


AM 1                                 # fts-env abstract machine
  AMSHMID       1                    # Shared memory ID
  CVMID         111                  # CCL-ID of the continuous value part
  ALPHABETFILE  /conf/fts-conf/fts-env.t     # Name of the alphabet file
                                             # incl. path from VVTHOME
  TRANSFILE     /conf/fts-conf/fts-env.trans # Name of the transition file
  STATEFILE     /conf/fts-conf/fts-env.state # Name of the state file
  REFFILE       /conf/fts-conf/fts-env.ref   # Name of the refusals file
  EVENTFILE     /conf/fts-conf/fts-env.event # Name of the event file
  TRACEFILE     /conf/fts-conf/fts-env.trace # Name of the trace file that
                                             # is generated by VVT-RT
  CASESFILE     /conf/fts-conf/fts-env.tst   # Name of the cases file that
                                             # is generated by VVT-RT
  CASESLTXFILE  /conf/fts-conf/fts-env.ltx   # Name of the LaTeX cases file
                                             # that is generated by VVT-RT
  RX_ADDR       111                  # Internal ID of the CCL-RX-Process
  RX_ADDR       112                  # Internal ID of the CCL-RX-Process
  TX_ADDR       111

  SEED          42                   # random seed for this AM
  STOPONIOERROR NO                   # do not stop if an unexpected
                                     # output occurs
  VERBOSE       YES                  # display unexpected outputs

  # Timer definitions: No:LowerBound:UpperBound
  TIMER 0:10000:20000                # Usage of primary mail server
        1:3000:6000                  # Usage of secondary mail server
        2:10000:15000                # Downtime of a network device
        3:1000:-                     # Downtime of raid power supply
        4:40000:50000                # Gap between occurence of two
                                     # hazards
```

```
END
```

H.2.4 makeaeraw

```
#! /usr/bin/awk -f
#
# makeaeraw
#
# Dieses Skript kann aus der Alphabeth-Datei die
# Datei aeraw erzeugen, wie wir sie brauchen
#
# $Id: makeaeraw,v 1.1 2000/08/02 11:42:12 hjficker Exp $
#

BEGIN {

    if (ARGC != 3) {
        printf "Aufruf: cat alphabetfile.t | ";
        printf "makeaeraw number_of_ams am_offset > aeraw.txt\n";
        exit;
    } else {
        num_am = ARGV[1];
        am_offset = ARGV[2];
        ARGC -= 2;
    }
}

$4 ~ /^AM_(OUT|IN)PUT$/ {
    # printf "1 %d %s ", $1, $2;
    a_len = split($2, a, ".");

    # Erzeugen fuer alle Mailserver-AMs
    for (i = num_am+am_offset-1; i>=am_offset; i--) {

    # Jetzt noch "raw data" schreiben, abhängig von Channel
      if ($2 ~ /^smtp_write/) {
        printf "%d %d %s %d 0 5 01 %02X %02X %02X %02X\n",
          i, $1, $2, i, a[2], a[3], a[4], i;
      }
      else if ($2 ~ /^pop_read/) {
        printf "%d %d %s %d 0 3 02 %02X %02X\n", i, $1, $2, i, a[2], i;
      }
      else if ($2 ~ /^nfs_read/) {
        printf "%d %d %s %d 0 3 03 %02X %02X\n", i, $1, $2, i, a[2], i;
      }
      else if ($2 ~ /^smtp_ack/) {
        printf "%d %d %s %d 0 3 04 %02X %02X\n",
               i, $1, $2, i, a[2] == "ok" ? 1 : 0, i;
      }
      else if ($2 ~ /^pop_ack/) {
        printf "%d %d %s %d 0 3 05 %02X %02X\n",
               i, $1, $2, i, a[2] == "ok" ? 1 : 0, i;
      }
      else if ($2 ~ /^nfs_ack/) {
        printf "%d %d %s %d 0 3 06 %02X %02X\n",
               i, $1, $2, i, a[2] == "ok" ? 1 : 0, i;
      }

    }
}

#
# Changelog:
```

```
# $Log: makeaeraw,v $
# Revision 1.1  2000/08/02 11:42:12  hjficker
# Änderungen im Anhang (pretty-printing, übern Rand geschrieben,
# Teilprojekt-Namen dazugeschrieben)
#
# Revision 1.7  1999/09/02 13:48:11  rscholz
# Umstellung auf neues VVT-RT mit Mapping zw. AMs (Teil 1)
#
# Revision 1.6  1999/07/13 16:14:45  hjficker
# Die Auswahl der Mailtypen geschieht jetzt im ifm-Programm, und
# nicht mehr über Events. Daher muß Dies auch nicht mehr mit-
# geschickt werden
#
# Revision 1.5  1999/06/26 16:33:33  hjficker
# - awk liegt in /usr/bin
# - Bedeutung von ctp geändert
#
# Revision 1.4  1999/06/24 11:57:45  hjficker
# ack werden pro user zurückgegeben.
#
# Revision 1.3  1999/06/15 08:42:13  hjficker
# Wenn man awk könnte... ;-)
# Bugs im Skript gefixt
#
# Revision 1.2  1999/06/14 19:03:57  hjficker
# Komplett umgeschrieben in awk
#
# Revision 1.1  1999/06/10 15:41:52  hjficker
# Erste Version von makeaeraw
#
#
```

H.2.5 makeamccl

```
#! /usr/bin/awk -f
#
# makeamccl
#
# Dieses Skript kann aus der Alphabeth-Datei die Datei amccl
# so erzeugen wie wir sie brauchen
#
# $Id: makeamccl,v 1.1 2000/08/02 11:42:12 hjficker Exp $
#


# Was wir brauchen:
#  - AM_ERROR
#  - AM_WARNING
#  - AM_INPUT
#  - AM_OUTPUT
#
#  Warum am Ende allerdings 11 steht, ist mir auch nicht klar...


$4 ~ /AM_(ERROR|WARNING|(IN|OUT)PUT)/ {
    # Nur diese Zeilen ausgeben
    a_len = split($2, a, ".");
    # Bei jedem sollte Action Number am Ende stehen...
    # Wir erinnern uns: Action Nummer = Abstract Machine
    #  Warum am Ende allerdings 11 steht, ist mir auch nicht klar...
    printf "%d %d 111\n", a[a_len], $1;
}

#
# Changelog:
```

```
# $Log: makeamccl,v $
# Revision 1.1  2000/08/02 11:42:12  hjficker
# Änderungen im Anhang (pretty-printing, übern Rand geschrieben,
# Teilprojekt-Namen dazugeschrieben)
#
# Revision 1.5  1999/09/06 13:38:15  rscholz
# bis zu 10 AMs moeglich, automatisierte Erstellung der Config-Files
#
# Revision 1.4  1999/09/02 10:46:01  hjficker
# - Altes Shell-Skript entsorgt
# - Kommentare geupgedatet
#
# Revision 1.3  1999/07/13 16:15:32  hjficker
# Es werden nicht alle Events gebraucht (z.B. Timerevents nicht)
#
# Revision 1.2  1999/06/26 16:34:51  hjficker
# - Ist jetzt awk-Script
# - mehrere AMs möglich
#
# Revision 1.1  1999/06/10 15:46:13  hjficker
# Erste Version
#
#
```

H.2.6 makeabs

```
#! /usr/bin/awk -f
#
# makeabs
#
# Dieses Skript kann aus der Alphabeth-Datei die
# Datei abseventmaps.txt erzeugen, wie wir sie brauchen
#
# Copy & Paste-Programmierung von makeaeraw...
#
# $Id: makeabs,v 1.1 2000/08/02 11:42:12 hjficker Exp $
#

BEGIN {

    if (ARGC != 3) {
        printf "Aufruf: cat alphabetfile.t | ";
printf "makeabs number_of_ams am_offset > abseventmaps.txt\n";
        exit;
    } else {
        num_am = ARGV[1];
        am_offset = ARGV[2];
        ARGC -= 2;
    }
}

$4 ~ /^AM_(OUT|IN)PUT$/ {

    a_len = split($2, a, ".");

    # Erzeugen fuer alle Mailserver-AMs
    for (i = num_am+am_offset-1; i>=am_offset; i--) {

    # Jetzt noch "raw data" schreiben, abhängig von Channel
#      if ($2 ~ /^smtp_write/) {
#       printf "AM %d %s IFM 120 sw.%d.%d.%d.%d\n",
#         i, $2, a[2], a[3], a[4], i;
#      }
#      else if ($2 ~ /^pop_read/) {
```

```
#       printf "AM %d %s IFM 120 pr.%d.%d\n", i, $2, a[2], i;
#      }
#      else if ($2 ~ /^nfs_read/) {
#       printf "AM %d %s IFM 120 nr.%d.%d\n", i, $2, a[2], i;
#      }
#      else if ($2 ~ /^smtp_ack/) {
#       printf "IFM 120 sa.%s.%d AM %d smtp_ack.%s\n", a[2], i, i, a[2];
#      }
#      else if ($2 ~ /^pop_ack/) {
#       printf "IFM 120 pa.%s.%d AM %d pop_ack.%s\n", a[2], i, i, a[2];
#      }
#      else if ($2 ~ /^nfs_ack/) {
#       printf "IFM 120 na.%s.%d AM %d nfs_ack.%s\n", a[2], i, i, a[2];
#      }
    }
}

END {

    printf "AM 1 whatms.0 ";
    for (i = num_am+am_offset-1; i>=am_offset; i--) {
      printf "AM %d whatms.0 ", i;
    }
    printf "\n";

    printf "AM 1 whatms.1 ";
    for (i = num_am+am_offset-1; i>=am_offset; i--) {
      printf "AM %d whatms.1 ", i;
    }
    printf "\n";
}

#
# Changelog:
# $Log: makeabs,v $
# Revision 1.1  2000/08/02 11:42:12  hjficker
# Änderungen im Anhang (pretty-printing, übern Rand geschrieben,
# Teilprojekt-Namen dazugeschrieben)
#
# Revision 1.2  1999/09/06 13:38:15  rscholz
# bis zu 10 AMs moeglich, automatisierte Erstellung der Config-Files
#
# Revision 1.1  1999/09/02 14:29:33  rscholz
# neue Skripte zum autom. Erstellen von Config-Files
#
#
```

Anhang I. FTS: Source–Code des IFM

I.1 ifm.c

```
/*
 * ifm.c
 *
 * $Id: ifm.c,v 1.13 1999/09/10 13:34:22 hjficker Exp $
 */

/* Standard Header */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* VVT-RT Header */
#include "vvtiflib.h"

/* Eigene Header */
#include "ifm.h"


#define MY_ADDR 100
#define BUFFLEN 4000

char *SMTP_COMMAND = "ifm/smtp/send_test_mail.pl";
char *POP3_COMMAND = "ifm/pop3/receive_pop.pl";
char *NFS_COMMAND = "ifm/nfs/receive_nfs.pl";
char *RCV_FWD_COMMAND = "ifm/forward/receive_forw.pl";
char *LOAD_COMMAND = "ifm/load/setload";
char *NETCTRL_COMMAND = "ifm/network/netctrl";

char *host_name[] = {"ariane5",
                     "wac",
                     /* IP-Adressen */
                     "134.102.204.204",        /* ariane5 */
                     "134.102.218.202",        /* ariane5, 218er-Netz */
                     "134.102.204.203",        /* wac */
                     "134.102.218.201"         /* wac, 218er-Netz */
};


char *vvtfts;
char *vvthome;
int debug;

/* Merken, wieviele Mails jeweils gesendet und abgeholt wurden */
```

```
int **mail_send;
int **mail_received;

/* Realisiert über Message_queues */
int msgq_id;

void initialize() {
    char *env_debug;
    char *env_seed;
    int i,j;

    if ( ! (vvtfts = getenv("VVTFTS"))) {
        fprintf(stderr, "VVTFTS nicht gesetzt\n");
        exit(255);
    }

    if ( ! (vvthome = getenv("VVTHOME"))) {
        fprintf(stderr, "VVTHOME nicht gesetzt\n");
        exit(255);
    }

    if ((env_debug = getenv("DEBUG")) != NULL) {
        debug = atoi(env_debug);
        if (debug) {
            printf("Debugging enabled\n");
        }
    } else {
        debug = 0;
    }

    if ((env_seed = getenv("SEED")) != NULL) {
        srandom(atoi(env_seed));
        if (debug) {
            printf("Random Seed = %d\n", atoi(env_seed));
        }
    }

    if (getuid()) {
        /* Not root */
        fprintf(stderr, "WARNUNG: Funktioniert nur als root!");
    }

    msgq_id = msgget(IPC_PRIVATE, 0666);
    if (msgq_id == -1) {
        perror("msgget");
        exit(1);
    }

    mail_send = malloc((MAX_USER + 1) * sizeof(int*));
    mail_received = malloc((MAX_USER + 1)*sizeof(int*));

    for (i = 0; i < MAX_USER + 1; i++) {
        mail_send[i] = malloc((MAX_MAILTYPE + 1) * sizeof(int));
        mail_received[i] = malloc((MAX_MAILTYPE + 1) * sizeof(int));
        for (j = 0; j < MAX_MAILTYPE + 1; j++) {
            mail_send[i][j] = 0;
            mail_received[i][j] = 0;
        }
    }
}
```

```
/*
 * Diese Unterfunktion sammelt in einer Endlosschleife alle Messages
 * auf, und trägt sie in die Arrays ein. Diese Funktion MUSS in einem
 * extra Prozess aufgerufen werden, da sie nie nicht zurückkehren
 * werden tut.
 */
void mail_counter() {
    struct mail_ack msgbuf;
    for (;;) {
        if (msgrcv(msgq_id, &msgbuf, sizeof(msgbuf), 0, 0) == -1)
            perror("msgrcv");

        if (msgbuf.snd_rcv == MAIL_SND) {
            mail_send[msgbuf.mail_rcpt][msgbuf.mail_type] +=
                msgbuf.no_mails;
        }
        else if (msgbuf.snd_rcv == MAIL_RCV) {
            mail_received[msgbuf.mail_rcpt][msgbuf.mail_type] +=
                msgbuf.no_mails;
        }
        else if (msgbuf.snd_rcv == PRINT_STATUS) {
            /* Wir benutzen es zwar noch nicht, aber hier könnte man
               den Status abfragen */
            int i,j;
            printf("Mails geschickt an:\n ");
            for (j = 1; j < MAX_MAILTYPE + 1; j++) {
                printf("%6d", j);
            }
            printf("\n");
            for (i = 0; i < MAX_USER + 1; i++) {
                printf("fts%03d ", i);
                for (j = 1; j < MAX_MAILTYPE + 1; j++) {
                    printf("%6d", mail_send[i][j]);
                }
                printf("\n");
            }
            printf("Mails empfangen:\n ");
            for (j = 1; j < MAX_MAILTYPE + 1; j++) {
                printf("%6d", j);
            }
            printf("\n");
            for (i = 0; i < MAX_USER + 1; i++) {
                printf("fts%03d ", i);
                for (j = 1; j < MAX_MAILTYPE + 1; j++) {
                    printf("%6d", mail_received[i][j]);
                }
                printf("\n");
            }
        }
        fflush(NULL);
    }
}


int main(int argc, char **argv) {
    t_vvtReturncodes retcode;
    char byteArray[BUFFLEN];
    char buf[BUFFLEN];
    INT32 len;
    t_vvtCmdType ctp;
    t_vvtDataType dtp;
```

```
int i;
int pid;
int ret;

initialize();
fflush(NULL);


retcode = vvtConnect(MY_ADDR,"config.txt");
if ((pid = fork()) == -1) {
      perror("fork");
      exit(1);
}
else if (pid == 0) {
      /* Kind-Prozess, zuständig für Messagequeue */
      mail_counter();
}

do {
      int mail_type;

      /*— get data from test engine —*/
      retcode = vvtResetByteArray(BUFFLEN,byteArray);
      vvtReceiveFromTestDriver(vvtWait,byteArray);
      len = BUFFLEN;
      while (vvtOk == vvtGetNextCmd(&ctp,&dtp,(void *)buf,
                                    &len,byteArray)) {
            if (debug & 1) {
                  printf("RECEIVED: CTP %i DTP %i DATA",ctp,dtp);
                  for (i = 0; i < len; i++) {
                        printf(" %02X", buf[i]);
                  }
                  printf("\n");
            }
            len = BUFFLEN;
      }
      /* Mailtyp beim Schicken (muß VOR dem fork passieren,
         weil sonst immer dieselbe Zahl gewählt wird */
      mail_type = 1 + (random() % MAX_MAILTYPE);

      fflush(NULL);
      /* Hier verschiedene Aufrufe für die verschiedenen Interfaces
         machen, in mehreren Prozessen, damit weitere Events
         akzeptiert werden */
      if ((pid = fork()) < 0) {
            perror("fork");
      }
      else if (pid == 0) {
            switch (buf[0]) {
            case 1:                                    /* smtp_write */
                  /* Mail_nummer wird intern geregelt, und nicht
                     Über event */
                  ret = do_smtp(buf[1], /* Mailserver */
                              buf[2], /* Mail from */
                              buf[3], /* rcpt to */
                              mail_type); /* mail type */
                  buf[0] = 4; /* smtp_ack */
                  buf[1] = ret; /* ret=1 -> OK ret != 0 -> NOK */
                  buf[2] = buf[4]; /* Action Number */
                  len=3;
                  break;
```

```
                case 2:                                         /* pop_read */
                    ret = do_pop3(buf[1]);
                    buf[0] = 5; /* pop_ack */
                    buf[1] = ret; /* return ok/nok */
                    buf[2] = buf[2]; /* Action Number */
                    len=3;
                    break;
                case 3:                                         /* nfs_read */
                    ret = do_nfs(buf[1]);
                    buf[0] = 6; /* nfs_ack */
                    buf[1] = ret; /* return ok/nok */
                    buf[2] = buf[2]; /* Action Number */
                    len=3;
                    break;
                default:                     /* something really strange happend */
                    buf[0] = 42; /* invalid */
                    buf[1] = 0; /* return nok */
                    len=1;
                    break;
                }

                /* Im Fertigen Programm sollte das Return von
                   den einzelnen Interfaces kommen, und nicht zentral */
                if (debug & 1) {
                    printf("RETURN: CTP %i DTP %i DATA:", ctp, dtp);
                    for (i = 0; i < len; i++) {
                        printf(" %02X", buf[i]);
                    }
                    printf("\n");
                }

                retcode = vvtResetByteArray(BUFFLEN,byteArray);
                retcode = vvtPutCmd(ctp,dtp,buf,len,byteArray);
                vvtSendToTestDriver(byteArray);
                fflush(NULL);
                exit(0); /* Exit Child */
            } else {
                /* pid > 0 -> parent */
                /* Collect Children */
                while (waitpid(-1, NULL, WNOHANG) > 0);
            }
        } while (1);
}

/*
 * Changelog:
 *
 * $Log: ifm.c,v $
 * Revision 1.13 1999/09/10 13:34:22 hjficker
 * Überflüssige und vor allem falsche Kommentare
 * entfernt
 *
 * Revision 1.12 1999/07/28 13:21:58 rscholz
 * Umstellung auf neue Pfadnamen
 *
 * Revision 1.11 1999/07/14 17:12:50 hjficker
 * Debuging-Meldung rausgenommen
 *
 * Revision 1.10 1999/07/13 15:29:27 hjficker
 * - Ausgehende/eingehende Mails werden gezählt
 * - Mailtyp wird intern entschieden, nicht per Event
```

```
 *
 * Revision 1.9 1999/06/26 16:24:57 hjficker
 * - Für mehrere Abstract Machines umgeschrieben
 * - Weniger DEBUG-Ausgaben als default
 *
 * Revision 1.8 1999/06/23 12:41:56 hjficker
 * Output-Events bezogen auf Benutzer
 *
 * Revision 1.7 1999/06/14 19:02:45 hjficker
 * Den Aufruf der einzelnen Interface-Submodule implementiert
 *
 * Revision 1.6 1999/06/11 14:22:24 hjficker
 * "Multithreading" bzw. Verteilung auf mehrere Prozesse
 * eingebaut, damit mehrer Sachen gleichzeitig bearbeitet
 * werden können
 *
 * Revision 1.5 1999/06/10 16:12:07 hjficker
 * Sachen für die Verbindung zu VVT-RT eingebaut
 * Ist momentan nur ein Fake (ruft keine weiteren Programme auf,
 * gibt nur OK zurück)
 *
 * Revision 1.4 1999/06/02 16:01:09 hjficker
 * %id% und %log% nachgetragen
 *
 * Revision 1.3 1999/06/02 13:27:04 hjficker
 * setload- und netctrl-Aufrufe implementiert
 *
 * Revision 1.2 1999/05/19 12:59:46 hjficker
 * Zwei Makros geschrieben, die die Fehlerbehandlung von Library-
 * Calls übernehmen
 *
 * Revision 1.1 1999/05/18 14:12:06 hjficker
 * Erster Anfang der Interface-Module
 *
 */
```

I.2 ifm.h

```
/*
 * ifm.h
 *
 * Includedatei für alle Teile des Interface-Module
 *
 * $Id: ifm.h,v 1.7 1999/07/13 15:34:36 hjficker Exp $
 */


#define MAX_MAILTYPE 10
#define MAX_USER 10


/*
 * Einige wichtige Ausnahmen
 */
#define BIG_MAIL 8          /* Darf nicht an GMX_USER geschickt werden */

#define FWD_USER 7
#define GMX_USER 11
#define ALIAS_USER 12
```

```
/*
 * "Befehle", die über die Messagequeue geschickt werden
 */
#define MAIL_SND 1
#define MAIL_RCV 2
#define PRINT_STATUS 3



extern char *SMTP_COMMAND;
extern char *POP3_COMMAND;
extern char *NFS_COMMAND;
extern char *RCV_FWD_COMMAND;
extern char *LOAD_COMMAND;
extern char *NETCTRL_COMMAND;

extern char *host_name[];

extern char *vvtfts;
extern char *vvthome;

extern int debug;                               /* DEBUG-Ausgaben
                          1 -> normales debug
                          2 -> smtp log an
                          4 -> smtp debug an
                          8 -> pop3/nfs log an
                         16 -> pop3/nfs debug an */

extern int msgq_id;

/*
 * Mit TEST_COMMAND kann ein Library-Call ausgeführt werden,
 * und bei einem Rückgabewert -1 wird automatisch ein
 * return -1 (bzw abort() bei TEST_COMMAND_ABORT) durchgeführt.
 * Dadurch wird der Source etwas lesbarer
 */
#define TEST_COMMAND(command) \
    {\
        if ((command) == -1) {\
            perror(#command);\
            return -1;\
        }\
    }

#define TEST_COMMAND_ABORT(command) \
    {\
        if ((command) == -1) {\
            perror(#command);\
            abort();\
        }\
    }



struct mail_ack {
    long mtype;                        /* Wichtig für MessageQueue */

    int snd_rcv;                       /* MAIL_SND / MAIL_RCV */

    int mail_type;
```

```
    int mail_rcpt;

    int no_mails;          /* Klaus' tool schreibt evtl. mehrere Mails gleichzeitig */
};


void initialize();
void calculate_user(int user, char *name);
char *read_line(int fd, char *buffer, int len, int *offset, int *eof);

int do_smtp(int host, int sender, int rcpt, int mailno);
int do_pop3(int user);
int do_nfs(int user);
int readmail(char *command, int user);
int do_setload(int mailserver, int newload);
int do_netctrl(int mailserver, int device, int on);

/*
 * Changelog:
 *
 * $Log: ifm.h,v $
 * Revision 1.7 1999/07/13 15:34:36 hjficker
 * - Typen zum Zählen von Mails zugefügt
 * - Einige Konstanten (GMX_USER, usw.) definiert
 *
 * Revision 1.6 1999/06/23 12:42:48 hjficker
 * Reihenfolge der Argumente bei do_smtp geändert
 *
 * Revision 1.5 1999/06/14 19:01:09 hjficker
 * NFS-Client hinzugefügt
 *
 * Revision 1.4 1999/06/02 16:26:24 hjficker
 * %id% und %log% (und einige Kommentare) nachgetragen
 *
 * Revision 1.3 1999/06/02 16:01:43 hjficker
 * setload- und netctrl-Aufrufe implementiert
 *
 * Revision 1.2 1999/05/19 12:59:47 hjficker
 * Zwei Makros geschrieben, die die Fehlerbehandlung von Library-
 * Calls übernehmen
 *
 * Revision 1.1 1999/05/18 14:12:06 hjficker
 * Erster Anfang der Interface-Module
 *
 */
```

I.3 smtp.c

```
/*
 * smtp.c
 *
 * do_smtp startet den den smtpclient
 *
 * $Id: smtp.c,v 1.6 1999/07/13 16:01:04 hjficker Exp $
 */


#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "ifm.h"

#define BUFFERSIZE 8192

void calculate_user(int user, char *name) {
    /* Kleiner Hack, damit gmx-Accounf genutzt werden kann */
    switch (user) {
    case GMX_USER:
        strcpy(name, "ftschumann@gmx.de");
        break;
    case ALIAS_USER:
        user = 10;
    default:
        sprintf(name, "fts%03d@Live.Informatik.Uni-Bremen.DE", user);
    }
}


/*
 * Eine Tabelle, aus der man sehen kann, wo eine Mail
 * im Endeffekt hingeht
 * Jeder Eintrag besteht aus einer Liste von Empfängern, und
 * wird mit -1 terminiert (0 ist gültiger Nutzer)
 */
int usermapping[13][5] =
{
    { 0, -1 },
    { 1, -1 },
    { 2, -1 },
    { 3, -1 },
    { 4, -1 },
    { 5, -1 },                                  /*** Normale Nutzer */
    { 6, -1 },
    { 7, -1 },
    { 8, -1 },
    { 9, -1 },
    { 10, -1 },

    { 10, -1 },                                 /* GMX_USER */
    { 10, -1 }                                  /* ALIAS_USER */
};


/*
 * Hier wird die Zeile line geparst. Wichtig ist nur die erste Zahl...
 */
void smtp_parse_output(char *line, int rcpt, int mailtype) {
    struct mail_ack msgbuf;
    int no_of_mails;
    int i;
```

```
    if (sscanf(line, "\"%d\" Mail(s) wurde(n) gesendet.", &no_of_mails) !=
1) {
        fprintf(stderr, "Warnung: konnte Zeile (%s) nicht parsen\n", line);
        return;
    }
    msgbuf.mtype=1;
    msgbuf.snd_rcv = MAIL_SND;
    msgbuf.mail_type = mailtype;
    msgbuf.no_mails = no_of_mails;
    i = 0;
    while (usermapping[rcpt][i] != -1) {
        msgbuf.mail_rcpt = usermapping[rcpt][i];
        if (msgsnd(msgq_id, &msgbuf, sizeof(msgbuf), 0) == -1) {
            perror("msgsnd");
        }
        i++;
    }
}


int do_smtp(int host, int sender, int rcpt, int mailno) {
    int pid;
    int retstatus;
    int exitstatus;
    char sender_name[60];
    char rcpt_name[60];

    int stdoutpipe[2];
    int stderrpipe[2];

    char stdout_buffer[BUFFERSIZE];
    char stderr_buffer[BUFFERSIZE];

    int stdout_offset;
    int stderr_offset;

    int eof;
    fd_set readfds;
    int maxfd;
    int retval;


    calculate_user(sender, sender_name);
    calculate_user(rcpt, rcpt_name);

    /* BIG_MAIL nicht an GMX_USER schicken */
    if (rcpt == GMX_USER && mailno == BIG_MAIL) return 1;

    printf("Schicke Mail %d von %s nach %s...\n",
           mailno, sender_name, rcpt_name);

    TEST_COMMAND(pipe(stdoutpipe));
    TEST_COMMAND(pipe(stderrpipe));
    fflush(NULL);

    TEST_COMMAND(pid = fork());

    if (pid == 0) {
        /* Kindprozess */
        char *filename;
        char mailfile[10];
```

```
        char *args[20];                             /* Argumente für exec */
        int i;

        TEST_COMMAND_ABORT(close(stdoutpipe[0]));
        TEST_COMMAND_ABORT(close(stderrpipe[0]));
        TEST_COMMAND_ABORT(dup2(stdoutpipe[1], 1));
        TEST_COMMAND_ABORT(dup2(stderrpipe[1], 2));

        filename = malloc(strlen(vvtfts) + strlen(SMTP_COMMAND) + 2);
        /* filename wird nicht wieder freigegeben, aber
           es wird execv ausgeführt */
        strcpy(filename, vvtfts);
        strcat(filename, "/");
        strcat(filename, SMTP_COMMAND);

        sprintf(mailfile, "%d", mailno);

        args[0] = filename;
        args[1] = "-s"; args[2] = sender_name;
        args[3] = "-r"; args[4] = rcpt_name;
        args[5] = "-h"; args[6] = host_name[host];
        args[7] = "-m"; args[8] = mailfile;
        i=9;
        if (debug & 2) {
            args[i++] = "-l";
        }
        if (debug & 4) {
            args[i++] = "-d";
        }
        args[i] = NULL;
        execv(args[0], args);
        /* sollte nicht zurückkehren, wenn doch... */
        perror("execv");
        abort();
    }
    /* Sonst pid > 0 => Vaterprozess */

    /* Lesen von stdout und stderr des Kindprozesses */
    TEST_COMMAND(close(stdoutpipe[1]));
    TEST_COMMAND(close(stderrpipe[1]));

    TEST_COMMAND(fcntl(stdoutpipe[0], F_SETFL, O_NONBLOCK));
    TEST_COMMAND(fcntl(stderrpipe[0], F_SETFL, O_NONBLOCK));

    stdout_offset = 0;
    stderr_offset = 0;

    stdout_buffer[0] = 0;
    stderr_buffer[0] = 0;

    do {
        FD_ZERO(&readfds);
        FD_SET(stdoutpipe[0], &readfds);
        FD_SET(stderrpipe[0], &readfds);
        maxfd = stdoutpipe[0] < stderrpipe[0] ? stderrpipe[0] : stdoutpipe[0];

        retval = select(maxfd+1, &readfds, NULL, NULL, NULL);

        if (retval < 0) {
            perror("select");
            close(stdoutpipe[0]);
```

```
                close(stderrpipe[0]);
                return -1;
            }
            if (FD_ISSET(stdoutpipe[0], &readfds)) {
                char *line;
                while ((line = read_line(stdoutpipe[0],
                                         stdout_buffer,
                                         BUFFERSIZE,
                                         &stdout_offset,
                                         &eof)) != NULL) {
                    if (debug & 1)
                        printf("<%s stdout> %s\n", SMTP_COMMAND, line);
                    smtp_parse_output(line, rcpt, mailno);
                }
            }

            if (FD_ISSET(stderrpipe[0], &readfds)) {
                char *line;
                while ((line = read_line(stderrpipe[0],
                                         stderr_buffer,
                                         BUFFERSIZE,
                                         &stderr_offset,
                                         &eof)) != NULL) {
                    if (debug & 1)
                        printf("<%s stderr> %s\n", SMTP_COMMAND, line);
                }
            }
        } while (!eof);


        /* warten auf Rückgabe */
        TEST_COMMAND(waitpid(pid, &retstatus, 0));

        if (! WIFEXITED(retstatus)) {
            if (WIFSIGNALED(retstatus)) {
                fprintf(stderr, "<%s> %d beendet mit %s\n", SMTP_COMMAND,
                        pid, sys_siglist[WTERMSIG(retstatus)]);
                return -1;
            }
            /* Kann eigentlich nicht vorkommen */
            fprintf(stderr, "Häh?\n");
            return -1;
        }

        exitstatus = WEXITSTATUS(retstatus);
        if (exitstatus == 0) {
            printf("Mail %d von %s nach %s gschickt.\n",
                   mailno, sender_name, rcpt_name);
        } else {
            printf("Fehler beim schicken von Mail %d von %s nach %s\n",
                   mailno, sender_name, rcpt_name);
        }
        return !exitstatus;          /* In C ist true = 1, nicht 0 wie im Shell-script */
}

/*
 * Changelog:
 * $Log: smtp.c,v $
 * Revision 1.6 1999/07/13 16:01:04 hjficker
 * - Zählen der abgesendeten Mails, dazu
 * - Parsen der Ausgaben
```

```
 *
 * Revision 1.5 1999/06/26 16:29:57 hjficker
 * Bessere Ausgaben, was das Programm gerade tut.
 *
 * Revision 1.4 1999/06/23 12:46:32 hjficker
 * Feineres Debug möglich
 * Reihenfolge der Argumente von do_smtp geändert (böser Bug)
 *
 * Revision 1.3 1999/06/14 18:56:47 hjficker
 * gmx-Account als "User 11" eingebaut
 *
 * Revision 1.2 1999/05/19 12:59:47 hjficker
 * Zwei Makros geschrieben, die die Fehlerbehandlung von Library-
 * Calls übernehmen
 *
 * Revision 1.1 1999/05/18 14:12:06 hjficker
 * Erster Anfang der Interface-Module
 *
 */
```

I.4 pop3.c

```
/*
 * pop3.c
 *
 * do_pop3 startet den pop3client
 *
 * $Id: pop3.c,v 1.5 1999/07/13 15:40:49 hjficker Exp $
 */

#include <stdio.h>

#include "ifm.h"


int do_pop3(int user) {
    printf("Hole Mail für fts%03d über POP...\n", user);
    return readmail(POP3_COMMAND, user);
}

/*
 * Changelog:
 *
 * $Log: pop3.c,v $
 * Revision 1.5 1999/07/13 15:40:49 hjficker
 * Weitere Include-Datei (<stdio>, verhindert Warnungen beim Compilieren)
 *
 * Revision 1.4 1999/06/26 16:26:07 hjficker
 * Bessere Meldungen
 *
 * Revision 1.3 1999/06/14 18:58:03 hjficker
 * Die meiste Funktionalität nach readmail.c übertragen
 *
 * Revision 1.2 1999/05/19 12:59:47 hjficker
 * Zwei Makros geschrieben, die die Fehlerbehandlung von Library-
 * Calls übernehmen
 *
 * Revision 1.1 1999/05/18 14:12:06 hjficker
 * Erster Anfang der Interface-Module
 *
```

```
 */
```

I.5 nfs.c

```
/*
 * pop3.c
 *
 * do_nfs startet den nfsclient
 *
 * $Id: nfs.c,v 1.4 1999/07/13 15:39:11 hjficker Exp $
 */

#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
#include <errno.h>
#include <grp.h>
#include <sys/types.h>
#include "ifm.h"

int do_nfs(int user) {
    struct passwd *pwd;
    char username[9];

    printf("Hole Mail für fts%03d über NFS...\n", user);
    sprintf(username, "fts%03d", user);
    pwd = getpwnam(username);
    if (pwd == NULL) {
        if (errno) {
            perror("pwd = getpwnam(username)");
        } else {
            fprintf(stderr, "No user %s ?!?\n", username);
        }
        exit(1);
    }
    if (debug & 1)
        printf("New user: %s\n", username);
    TEST_COMMAND(initgroups(username, pwd->pw_gid));
    TEST_COMMAND(setgid(pwd->pw_gid));
    TEST_COMMAND(setuid(pwd->pw_uid));
    if (user == FWD_USER)
        return readmail(RCV_FWD_COMMAND, user);
    else
        return readmail(NFS_COMMAND, user);
}


/*
 * Changelog:
 *
 * $Log: nfs.c,v $
 * Revision 1.4 1999/07/13 15:39:11 hjficker
 * - seteuid durch setuid ersetzt (Ist besser, wenn auch die
 * reale UID gesetzt wird)
 * - Hack, damit für FWD_USER ein anderes Programm gestartet wird
 *
 * Revision 1.3 1999/06/26 16:25:42 hjficker
 * - Gruppen werden gesetzt
 * - Bessere Meldungen
 *
```

```
 * Revision 1.2 1999/06/23 12:43:38 hjficker
 * setuid auf User.
 *
 * Revision 1.1 1999/06/14 19:00:26 hjficker
 * NFS-Client, benutzt die selben Funktionen wie der
 * pop3-Client
 *
 */
```

I.6 readmail.c

```
/*
 * readmail.c
 *
 * readmail ruft entweder den pop3client oder den nfsclient auf.
 * Da beide das selbe Interface besitzen, reicht eine Funktion.
 * Diese Funktion wird von do_pop3 und von do_nfs aufgerufen
 *
 * $Id: readmail.c,v 1.4 1999/07/13 15:48:48 hjficker Exp $
 */


#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/time.h>
#include <sys/wait.h>

#include "ifm.h"

/* Puffer zum lesen */
#define BUFFERLENGTH 8192

#define NO_MAIL "Keine"

int line_no;
int no_of_mails;


/*
 * Hier möchte ich eine Zeile parsen
 * Rückgabewert: 1 -> OK
 * 0 -> nicht OK
 * Weiterhin wird über MessageQueue zurückgeliefert, welche Mail
 * erhalten wurde.
 */
int readmail_parse_output(char *line, int rcpt) {
    int i;
    int ret;
    int mail_no;
    char mail_type[20];
    char sender[20];
```

```
        char status[20];
        struct mail_ack msgbuf;

        if (line_no == 1) {
                /* Erste Zeile -> Statusmeldung */
                if (strncmp(NO_MAIL, line, strlen(NO_MAIL)) == 0) {
                        /* Keine Mail */
                        no_of_mails = 0;
                } else {
                        /* Anzahl der Mails steht ganz vorne */
                        no_of_mails = atoi(line);
                }
                line_no++;
                return 1;
        }
        /* lineno > 1 -> Ausgabe einer Mail */
        /* Ich möchte die Zeile auftrennen */
        ret = 1;
        line_no++;
        i = sscanf(line,
                   "%d.: Mail \"%[^\"]\" from \"%[^\"]\" : \"%[^\"]\".",
                   &mail_no, mail_type, sender, status);
        if (i != 4) {
                fprintf(stderr, "Konnte nicht lesen. Format geändert?\n"
                        "Return-Wert %d\n", i);
                return 0;
        }

        if (mail_no != line_no - 2) {                          /* Hack */
                fprintf(stderr, "Falsche Mailnummer?\n");
                return 0;
        }

        /* Sender interessiert uns momentan nicht. */

        if (strcmp(status, "valid") != 0) {
                /* Kaputte Mail! */
                return 0;
        }

        /* Mailtyp per MessageQueue zurüchliefern */
        msgbuf.mtype=1;
        msgbuf.snd_rcv = MAIL_RCV;
        msgbuf.mail_type = atoi(mail_type);
        msgbuf.mail_rcpt = rcpt;
        msgbuf.no_mails = 1;
        if (msgsnd(msgq_id, &msgbuf, sizeof(msgbuf), 0) == -1) {
                perror("msgsnd");
        }

        return 1;
}


/*
 * Ließt eine Zeile vom File-Descriptor fd, und benutzt dabei
 * den Puffer buffer mit der Länge len. Offset zeigt den Anfang
 * der nächsten (evtl unvollständigen) Zeile, und brauch den
 * Aufrufenden nicht zu interessieren, außer daß er möglichst
 * nicht verändert werden braucht.
 * Rückgabewert: nächste Zeile. Ein NULL-Pointer zeigt nicht
```

```
 * unbedingt ein EOF an, sondern nur, daß momentan nichts gelesen
 * werden konnte. Für eof ist eof zuständig (genau!).
 */
char *read_line(int fd, char *buffer, int len, int *offset, int *eof) {
    int l;
    int rd;
    char *next;                                    /* Adresse des nächsten '\n' */
    char *ret;

    *eof = 0;
    if (*offset == 0) {
        /* Hier muß ich erstmal lesen */
        l = strlen(buffer);
        rd = read(fd, buffer + l, len - l - 1);
        if (rd == 0) {
            if (errno != EAGAIN) {
                *eof = 1;
            }
            return NULL;
        }
        buffer[l + rd] = 0;
    }

    if (buffer[*offset] == 0) {
        /* Nichts gelesen => Nächste Runde weiter */
        *offset = 0;
        return NULL;
    }

    next = index(buffer + *offset, '\n');
    if (next == NULL) {
        /* Kein weiters '\n' => In der nächsten Runde Rest lesen */
        memmove(buffer, buffer + *offset, strlen(buffer + *offset) + 1);
        *offset = 0;
        return 0;
    }

    *next = 0;
    ret = buffer + *offset;
    *offset = (next - buffer) + 1;
    return ret;
}


/*
 * readmail ließt die Mail mit dem Kommando command.
 * stdout wird geparst, und anhand dessen ein Fehlerwert
 * oder nicht zurückgegeben
 */
int readmail(char *command, int user) {
    int pid;
    int stdoutpipe[2];
    int stderrpipe[2];

    fd_set readfds;
    int retval;
    int maxfd;
    int eof;
    int retstatus;
    int exitstatus;
    int mails_ok;
```

```
int stdout_offset;
int stderr_offset;

char stdout_buffer[BUFFERLENGTH];
char stderr_buffer[BUFFERLENGTH];

TEST_COMMAND(pipe(stdoutpipe));
TEST_COMMAND(pipe(stderrpipe));
fflush(NULL);
TEST_COMMAND(pid = fork());
if (pid == 0) {
    /* Kindprozess */
    char username[20];
    char *filename;
    char *args[20];                         /* Argumente für execv */
    int i;

    /* Ausgabeumleitung richtig machen */
    TEST_COMMAND_ABORT(close(stdoutpipe[0]));
    TEST_COMMAND_ABORT(close(stderrpipe[0]));
    TEST_COMMAND_ABORT(dup2(stdoutpipe[1], 1));
    TEST_COMMAND_ABORT(dup2(stderrpipe[1], 2));

    filename = malloc(strlen(vvtfts) + strlen(command) + 2);
    /* filename wird nicht wieder freigegeben, aber
       es wird execv ausgeführt */
    strcpy(filename, vvtfts);
    strcat(filename, "/");
    strcat(filename, command);

    sprintf(username, "fts%03d", user);

    args[0] = filename;
    args[1] = "-u"; args[2] = username;
    i = 3;
    if (debug & 8) {
        args[i++] = "-l";
    }
    if (debug & 16) {
        args[i++] = "-d";
    }
    args[i] = NULL;

    execv(args[0], args);
    /* sollte nicht zurückkehren, wenn doch... */
    fprintf(stderr, "execv: %s\n", strerror(errno));
    fflush(NULL);
    usleep(200000);
    abort();
}
/* Sonst pid > 0 => Vaterprozess */

TEST_COMMAND(close(stdoutpipe[1]));
TEST_COMMAND(close(stderrpipe[1]));

TEST_COMMAND(fcntl(stdoutpipe[0], F_SETFL, O_NONBLOCK));
TEST_COMMAND(fcntl(stderrpipe[0], F_SETFL, O_NONBLOCK));

line_no = 1;
mails_ok = 1;
```

```
stdout_offset = 0;
stderr_offset = 0;

stdout_buffer[0] = 0;
stderr_buffer[0] = 0;

/* von stdin und stderr muß gleichzeitig gelesen werden */
do {
      FD_ZERO(&readfds);
      FD_SET(stdoutpipe[0], &readfds);
      FD_SET(stderrpipe[0], &readfds);
      maxfd = stdoutpipe[0] < stderrpipe[0] ? stderrpipe[0] : stdoutpipe[0];

      retval = select(maxfd+1, &readfds, NULL, NULL, NULL);

      if (retval < 0) {
            perror("select");
            close(stdoutpipe[0]);
            close(stderrpipe[0]);
            return -1;
      }
      if (FD_ISSET(stdoutpipe[0], &readfds)) {
            char *line;
            while ((line = read_line(stdoutpipe[0],
                                    stdout_buffer,
                                    BUFFERLENGTH,
                                    &stdout_offset,
                                    &eof)) != NULL) {
                  if (debug & 1)
                        printf("<%s stdout> %s\n", command, line);
                  mails_ok &= readmail_parse_output(line, user);
            }
      }

      if (FD_ISSET(stderrpipe[0], &readfds)) {
            char *line;
            while ((line = read_line(stderrpipe[0],
                                    stderr_buffer,
                                    BUFFERLENGTH,
                                    &stderr_offset,
                                    &eof)) != NULL) {
                  if (debug & 1)
                        printf("<%s stderr> %s\n", command, line);
            }
      }
} while (!eof);

TEST_COMMAND(close(stdoutpipe[0]));
TEST_COMMAND(close(stderrpipe[0]));

/* warten auf Rückgabe */
TEST_COMMAND(waitpid(pid, &retstatus, 0));

if (! WIFEXITED(retstatus)) {
      if (WIFSIGNALED(retstatus)) {
            fprintf(stderr, "<%s> %d beendet mit %s\n", command,
                        pid, sys_siglist[WTERMSIG(retstatus)]);
            return 0;
      }
      /* Kann eigentlich nicht vorkommen */
      fprintf(stderr, "Häh?\n");
```

```
        return 0;
    }
    exitstatus = WEXITSTATUS(retstatus);
    if (exitstatus == 0) {
        printf("Mail für fts%03d abgeholt.\n", user);
    } else {
        fprintf(stderr,
                "WARNUNG: %s kehrte fur fts%03d mit %d zurück\n"
                "Eventuell lock nicht erhalten?\n",
                command, user, exitstatus);
    }

    return mails_ok;
}


/*
 * Changelog
 *
 * $Log: readmail.c,v $
 * Revision 1.4 1999/07/13 15:48:48 hjficker
 * - Zählen der empfangenen Mails (beziehungweise Schicken einer
 * Nachricht an den "Zähl-Prozeß"
 * - Kommentare geschrieben
 *
 * Revision 1.3 1999/06/26 16:29:14 hjficker
 * - Besseres Scannen von stdout und stderr
 * - Endlich auch Auswertung von stdout
 * - Fehler jetzt nur noch "invalid" Mails, nicht Abbruch des
 * Programms (z.B. wegen lock)
 *
 * Revision 1.2 1999/06/23 12:44:54 hjficker
 * Feineres Debuging möglich
 * Bessere Ausgabe
 *
 * Revision 1.1 1999/06/14 18:59:34 hjficker
 * Die Funktionalität aus pop3.c hierher übertragen, damit der
 * NFS-Client auch was davon hat.
 *
 */
```

I.7 Makefile

```
#
# Makefile für das Interfacemodul
#
# $Id: Makefile,v 1.5 1999/06/14 19:01:29 hjficker Exp $
#



OBJECTS     = ifm.o smtp.o pop3.o nfs.o readmail.o
SOURCES     = $(OBJECTS:.o=.c)
TARGET      = ifm

CFLAGS      = -Wall -pedantic -g3 -D_VERBOSE -D_$(VVTOS)
CC          = gcc

INCLUDES    = -I$(VVTHOME)/inc \
              -I$(VVTHOME)/vvtemlib/inc \
```

```
                -I$(VVTHOME)/vvtshmlib/inc \
                -I$(VVTHOME)/vvtiflib/inc \
                -I$(VVTHOME)/vvtoslib/inc \
                -I$(VVTHOME)/vvtconflib/inc \
                -I$(VVTHOME)/targets/demo/inc

LDFLAGS       = -L$(VVTHOME)/vvtemlib/lib \
                -L$(VVTHOME)/vvtshmlib/lib \
                -L$(VVTHOME)/vvtiflib/lib \
                -L$(VVTHOME)/vvtoslib/lib \
                -L$(VVTHOME)/vvtconflib/lib

LIBS          = -lvvtemlib -lvvtshmlib -lvvtiflib -lvvtconflib


all: $(TARGET)


$(TARGET): $(OBJECTS)
              $(CC) $(LNFLAGS) -o $@ $(OBJECTS) $(LDFLAGS) $(LIBS)


%.o: %.c
              $(CC) -c $(CFLAGS) $(INCLUDES) -MMD -o $@ $<


clean:
              rm -f $(OBJECTS)
              rm -f $(OBJECTS:.o=.d)


realclean: clean
              rm -f $(TARGET)

-include $(wildcard *.d)
```

Anhang J. FTS: Perl-Skripte für das IFM

J.1 Versenden von Mails (send_test_mail.pl)

```
#!/usr/bin/perl
# Dieses Programm verbindet sich mit einem als Parameter angegebenem Rechner
# auf dem SMTP-Port (25) und sendet die angegebene Mail an die angegebenen
# Nutzer! (Angegeben bedeutet in diesem Fall als Parameter übergeben!!)
# Falls der Switch -n <Anzahl> angegeben wird, wird die Mail nacheinander
# <Anzahl>mal geschickt.
# Falls der Switch -f <Anzahl> angegeben wird, wird die Mail von
# <Anzahl> Prozessen gleichzeitig geschickt.
# Usage:
# send_test_mail -s <Sender> -r <Empfaengerliste> -h <Rechnername> \
#                -m <Dateiname> [ -n <Anzahl>] [ -f <Anzahl> ]

use lib ($ENV{"VVTHOME"}.'/targets/fts-target/ifm/perl-share');
use strict;
use Getopt::Std;
use IO::File;
use sock;
use log_io;

# Pfad der Mail-Bibliothek:
my $mail_bib = $ENV{"VVTHOME"}."/targets/fts-target/ifm/smtp/mail-lib/";

# Name der LOG-Datei:
# (sollte im Home des Senders liegen)
# es wird die Kommunikation von Sendmail-Meldungen, sowie der Rechnername,
# der Dateiname der Mail und die Liste der Empfänger gespeichert!
my $log_datei = ".send_test_mail.log";

# Variable, die den Erfolg mit größer 0 anzeigt;
# am Anfang sind wir erfolgreich ;-)
my $erfolg = 42;

# Kommandozeilenswitches bzgl. Logging und Debuging und username parsen
getopts("lds:r:m:h:n:f:");
# Öffnen der Log-Datei im Appendmode und als Standard Ausgabedatei auswählen
if ($Getopt::Std::opt_d) {
    #open(LOG, ">-"); # for debug-option debugging only
    open(LOG, ">&STDERR");
} else {
    if ($Getopt::Std::opt_l) {
        open(LOG, ">>".$ENV{"HOME"}."/".$log_datei);
    } else {
        open(LOG, ">>/dev/null");
    }
}
select LOG;

# Domainname für helo
my $domainname = "live.informatik.uni-bremen.de";

#  diese Daten werden in die LOG-Datei geschrieben
my ($sender, $rechnername, $mail_name, $prozesse, $mehrfach, @empf);
my @rueck = &argumente_parsen;
$sender = shift(@rueck);
$rechnername = shift(@rueck);
$mail_name = shift(@rueck);
```

```
$prozesse = shift(@rueck);
$mehrfach = shift(@rueck);
@empf = @rueck;

# ermitteln ob die zu sendende Mail existiert und lesbar ist
if (!(-r $mail_bib.$mail_name)) {
    print LOG "Die Mail existiert nicht oder kann nicht gelesen werden!\n";
    close(LOG);
    exit(10);
}

# Dieser Zähler zählt die tatsächlich gesendeten Mails
my $mail_zaehler = 0;

# Array, daß die PIDs der geforkten Prozesse hält, $kind ist klar,
# $prz Nummer des Prozesses fürs logging
my (@PIDs, $prz, $fork_ausg, @handles,
    $write_handle, $socket);
$fork_ausg = "";
$prz = 1;
# Forken falls notwendig
if ($prozesse != 1) {
    $prz = $prozesse;
    while($prz > 1) {
        &new_child && ($prz--);
    }
}
# Bin Vater bzw. Einzelprozess
if ($prz == 1) {
    print LOG $fork_ausg;
    &sende_mail;
}

sub new_child {
    my ($read, $write) = (new IO::File, new IO::File);
    pipe $read, $write;
    my $pid = fork;
    if ($pid == 0) {
        close($read);
        # autoflush für den Writehandle einschalten!
        select($write);
        $| = 1;
        select(LOG);
        $write_handle = $write;
        &sende_mail;
        exit(244);
    } elsif ($pid == -1) {
        # Fehler beim Forken: weiterversuchen!
        close(READ);
        close(WRITE);
        return 0;
    } else {
        $fork_ausg .= $prz.":".$pid.": forked.\n";
        unshift @PIDs, $pid;
        unshift @handles, $read;
        #close(READ);
        #open ($handles{$pid}, "<&READ");
        close($write);
        return 1;
    }
}

sub sende_mail {
    # Setzen der Prozessnummer falls nötig:
    if ($prozesse != 1) {
        &log_io::setze_proz_nummer($prz);
```

```
    }
    # Verbinden mit dem Socket
    $socket = &sock::socket_verbinden(\*SOCK, $rechnername,
                                       25, &log_io::_pn);

    # Kommunikation über den Socket
    $erfolg = &test_antwort(&log_io::read_line($socket, \*LOG)) ||
        &ende_des_prog;
    # Mit Helo anmelden am Server
    &log_io::lprint ($socket, "HELO ".$domainname."\n", \*LOG);
    $erfolg = &test_antwort(&log_io::read_line($socket, \*LOG)) ||
        &ende_des_prog;
    # Mail $mehrfach versenden
    while ($mehrfach) {
        # Sender der Mail festlegen
        &log_io::lprint($socket, "MAIL From: ".$sender."\n", \*LOG);
        ((($erfolg =
           &test_antwort(&log_io::read_line($socket, \*LOG))) == 38)
         && goto wait_reset);
        ($erfolg) || &ende_des_prog;
        # Alle Empfänger der Mail angeben
        foreach my $elem (@empf) {
            &log_io::lprint($socket, "RCPT To: ".$elem."\n", \*LOG);
            ((($erfolg =
               &test_antwort(&log_io::read_line($socket, \*LOG))) == 38)
             && goto wait_reset);
            ($erfolg) || &ende_des_prog;
        }
        # Mailbody auf den Filehandle schreiben
        &log_io::lprint($socket, "DATA\n", \*LOG);
        ((($erfolg =
           &test_antwort(&log_io::read_line($socket, \*LOG))) == 38)
         && goto wait_reset);
        ($erfolg) || &ende_des_prog;
        &print_mail($socket, $mail_name);
        print(LOG &log_io::_pn."Mail \"".$mail_bib.$mail_name.
              "\" geschickt.\n");
        &log_io::lprint($socket, ".\n", \*LOG);
        ($erfolg = &test_antwort(&log_io::read_line($socket, \*LOG)) ||
         &ende_des_prog) &&
             ($mail_zaehler++);
        $mehrfach--;
      wait_reset:
        &log_io::lprint($socket, "RSET\n", \*LOG);
        $erfolg = &test_antwort(&log_io::read_line($socket, \*LOG)) ||
            &ende_des_prog;
    }
    &ende_des_prog;
}

# Sauberes Beenden des Programms
sub ende_des_prog {
    &log_io::lprint($socket, "QUIT\n", \*LOG);
    my $erf = &test_antwort(&log_io::read_line($socket, \*LOG));
    $erfolg = $erfolg ? $erf : $erfolg;
    # Schließen des Socket
    close($socket);
    #Schließen der Log-Datei
    close(LOG);
    # Bin ich Kind oder Elternprozess und hatte ich Kinder
    if(($prz == 1) && ($prozesse>1) ) {
        # Daten der Kinder lesen
        while (scalar @handles) {
            my $handle = shift(@handles);
            my $line = <$handle>;
            $line =~ m/^\"(\d+)\" Mail\(s\)/ ||
```

```
                die("Kind hatte eine seltsame Ausgabe auf".
                    " der Pipe: $line");
            $mail_zaehler += $1;
        }
        # Kindprozesse aufsammeln
        my (@wPIDs);
        my $pid = 0;
        while ($pid != -1) {
            $pid = wait;
            if ($pid != -1) {
                print LOG $pid.": ended.\n";
                if($erfolg) { $erfolg = $? ? 0 : 42};
                foreach my $ele (@PIDs) {
                    if ($pid == $ele) {
                        unshift @wPIDs,$pid;
                    }
                }
            }
        }
        if (scalar @wPIDs != scalar @PIDs) {
            die "Wrong Number of Childs\n";
        }
    }
    # Kind gibt Anzahl der Mails nach WRITE
    ($prz > 1) &&
        (&gib_anzahl_aus($write_handle, $mail_zaehler));
    # Anzahl der Mails nach STDOUT ausgeben, wenn Vater oder Einzelprozess
    ($prz == 1) &&
        (&gib_anzahl_aus(\*STDOUT, $mail_zaehler));
    # Rückmeldung, ob Mail erfolgreich angenommen wurde
    # print $erfolg,"\n";
    if ($erfolg) {
        exit(0);
    } else {
        exit(42);
    }
}
############
#Subroutinen, die die Drecksarbeit erledigen
############
sub copy_handle(*) {
    my $handle = shift;
    open INTERN, "&$handle";
    return \*INTERN;
}
            #open ($handles{$pid}, "<&READ");

sub gib_anzahl_aus(*$) {
    my ($handle, $anz) = @_;
    print$handle "\"$anz\" Mail(s) wurde(n) gesendet.\n";
}

# Argumente parsen:
# Hiernach liegen die Empfänger im Array @empf,
# der Rechnername in $rechnername und der Dateiname der Mail in $mail_name
# außerdem wurden diese Daten in die LOG-Datei geschrieben
sub argumente_parsen {
    #$rechnername = $opt_h;
    my @empf = split(/,/,$Getopt::Std::opt_r);
    #$mail_name = $opt_m;
    if (!defined(@empf) ||
        !defined($Getopt::Std::opt_h) ||
        !defined($Getopt::Std::opt_m) ||
        !defined($Getopt::Std::opt_s)) {&usage;};
    # Testen, ob die Mail mehrfach nacheinander verschickt werden soll
    my $mehrfach = 1;
```

```
    if(defined($Getopt::Std::opt_n)) {
        if($Getopt::Std::opt_n =~ /^\d*$/) {
            $mehrfach = $Getopt::Std::opt_n;
        } else { &usage;}
    }
    my $prozesse = 1;
    if(defined($Getopt::Std::opt_f)) {
        if($Getopt::Std::opt_f =~ /^\d*$/) {
            $prozesse = $Getopt::Std::opt_f;
        } else { &usage;}
    }
    my @rueckgabe;
    unshift @rueckgabe, $Getopt::Std::opt_s, $Getopt::Std::opt_h,
            $Getopt::Std::opt_m, $prozesse, $mehrfach, @empf;
    my $ausg = join(",", @empf);
    print "######\n";
    print "#> Die Mail \"".$mail_bib.$Getopt::Std::opt_m;
    print "\"\n#> wird von \"".$Getopt::Std::opt_s."\" an \"";
    print $ausg."\"\n#> auf dem Rechner \"".$Getopt::Std::opt_h;
    print "\" \n#> " . $mehrfach . "mal von " . $prozesse .
          " Prozessen geschickt:\n";
    return @rueckgabe;
}

sub usage {
    select STDERR;
    print "Usage: send_test_mail.pl -s <Sender> -r <Empfaengerliste> ".
        "-h <Rechnername> -m <Dateiname>\n";
    print "send_test_mail.pl -s <sender> -r <list of rcpt> ".
        "-h <hostname> -m <filename>\n";
    print "-l schaltet Logging in eine Datei ein\n";
    print "-d schaltet Debbugging ein\n";
    print "-n <Anzahl>   Die Mail wird Anzahl mal nacheinander geschickt\n";
    print "-f <Anzahl>   Mail wird von Anzahl Prozessen verschickt\n";
    exit(3);
}

# Schreibt eine Mail aus der Bibliothek auf den angegebenen Filehandle
sub print_mail(*$) {
    my ($handle, $file, $line, $mail_name);
    $handle = shift;
    $mail_name = shift;
    $file = $mail_bib.$mail_name;
    open LOCAL, "<".$file;
    $line = <LOCAL>;
    $line =~ s/>><</>>$mail_name<</;
    print $handle $line;
    while ($line = <LOCAL>) {
        print $handle $line;
    }
    close(LOCAL);
}

# Testet eine Rückmeldung von Sendmail auf Fehler
# bricht Programm mit Fehlercode ab
sub test_antwort($) {
    my $antwort = shift(@_);
    my ($char, $erf);
  SWITCH: for($antwort) {
      /^[23]\d\d(.)/ && do {
          $erf = 42;
          $char = $1;
          last SWITCH;};
      /^5\d\d(.)/ && do {
          $erf = 0;
          $char = $1;
```

```
            last SWITCH;};
        /^4\d\d(.)/ && do {
            $erf = 38; # steht für warten
            $char = $1;
            last SWITCH;};
        }
    if ($char eq "-") {
            &test_antwort(&read_line($socket, "LOG"));
    }
    return $erf;
}
```

J.2 Empfangen von Mails via POP3 (`receive_pop.pl`)

```
#!/usr/bin/perl
# Dieses Programm liest alle Mails ueber das POP3 Protokoll von ariane5 und
# löscht diese!!
# Dabei wird die Vollstaendigkeit der Mails ueberprueft und in einer Datei
# mit Sender abgelegt.
# USAGE: receive_pop.pl -u <user>

use lib ($ENV{"VVTHOME"}.'/targets/fts-target/ifm/perl-share');
use strict;
use Getopt::Std;
# eigene Module
use pfade;
use sock;
use log_io;
use pass;
use mail_test;
use tmp_io;

# Pfad der Mail-Bibliothek:
my $mail_bib = $ENV{"VVTHOME"}."/targets/fts-target/ifm/smtp/mail-lib/";
# Name der LOG-Datei:
# (sollte im Home des Senders liegen)
# es wird die Kommunikation von Sendmail-Meldungen, sowie der Rechnername,
# der Dateiname der Mail und die Liste der Empfänger gespeichert!
my $log_datei = ".receive_pop.log";

# Variable, die den Erfolg mit größer 0 anzeigt;
# am Anfang sind wir erfolgreich ;-)
my $erfolg = 42;

# Kommandozeilenswitches bzgl. Logging und Debuging und username parsen
getopts("ldw:u:");
# Öffnen der Log-Datei im Appendmode und als Standard Ausgabedatei auswählen
if ($Getopt::Std::opt_d) {
    # open(LOG, ">-"); # for debugging
    open(LOG, ">&STDERR");
} else {
    if ($Getopt::Std::opt_l) {
        open(LOG, ">>".$ENV{"HOME"}."/".$log_datei);
    } else {
        open(LOG, ">>/dev/null");
    }
}
select LOG;

#  diese Daten werden in die LOG-Datei geschrieben
my $rechnername = "ariane5.informatik.uni-bremen.de";
my @rueck = &argument_parsen;
my $user = shift(@rueck);
my $user_id = shift(@rueck);
my $warten = shift(@rueck);
```

```perl
# Verbinden mit dem Socket
my $socket =
    &sock::socket_verbinden(\*SOCK, $rechnername, 110, &log_io::_pn);

# Kommunikation über den Socket
&test_antwort(&log_io::read_line($socket, \*LOG));
# Username dessen Mail gelesen werden soll
&log_io::lprint($socket, "USER ".$user."\n", \*LOG);
&test_antwort(&log_io::read_line($socket, \*LOG));
# Passwort des Users, Achtung kein logging
&log_io::lprint($socket, "PASS ".&pass::passwt($user_id)."\n");
print LOG "PASS >geheim<\n";
&test_antwort(&log_io::read_line($socket, \*LOG));
# Testen ob Mail zu lesen ist
&log_io::lprint($socket, "STAT\n", \*LOG);
&test_antwort(my $stat = &log_io::read_line($socket, \*LOG));
if ($stat =~ /^\+OK (\d+) (\d+)/) {
    # Anzahl der Mails besorgen
    my $mail_anz = $1;
    if ($mail_anz > 0) {
        print STDOUT "$mail_anz Mail(s) wurde(n) empfangen,".
        " Status der Mail(s) folgt:\n";
    } else {
        print STDOUT "Keine Mails im Maildrop.\n";
        &log_io::lprint($socket, "QUIT\n", \*LOG);
        &test_antwort(&log_io::read_line($socket, \*LOG));
        &ende_des_prog;
    }
} else {
    print STDERR "Die Implementierung von \"STAT\" des POP3-Deamon ist " .
                 "nicht RFC 1939 konform\n";
    $erfolg = 0;
    &log_io::lprint($socket, "QUIT\n", \*LOG);
    &test_antwort(&log_io::read_line($socket, \*LOG));
    &ende_des_prog;
}
# Warten falls gewünscht!
if (defined($warten)) {
    sleep $warten;
}
# Auflisten der Mails im Maildrop
&log_io::lprint($socket, "LIST\n", \*LOG);
&test_antwort(&log_io::read_line($socket, \*LOG));
# es liegen Messages vor, da Prog vorher nicht abgebrochen wurde
my @msg_ids = &list_parsen;
my (@msg_names);
foreach my $msg_id (@msg_ids) {
    # Mail lesen
    my $tmp_msgname = &get_msg($msg_id);
    push @msg_names, $tmp_msgname;
    # Mail Löschen
    &log_io::lprint ($socket, "DELE ".$msg_id."\n", \*LOG);
    &test_antwort(&log_io::read_line($socket, \*LOG));
}
&log_io::lprint($socket, "QUIT\n", \*LOG);
&test_antwort(&log_io::read_line($socket, \*LOG));

# Mail mit der Mailbiliothek vergleichen
my $ausg = "";
foreach my $tmp_msg_name (@msg_names) {
    my $msg_id = shift @msg_ids;
    $ausg .= "$msg_id.: " .
      &mail_test::test_msg($tmp_msg_name, $mail_bib) . "\n";
}
print STDOUT $ausg;
```

```
print LOG $ausg;
&ende_des_prog;

# Sauberes Beenden des Programms
sub ende_des_prog {
    # Schließen des Socket
    close($socket);
    #Schließen der Log-Datei
    close(LOG);

    # Rückmeldung, ob Mail gelesen wurde
    # print $erfolg,"\n";
    if ($erfolg) {
        exit(0);
    } else {
        exit(42);
    }
}
############
#Subroutinen, die die Drecksarbeit erledigen
############
# Argumente parsen:
# user und user_id (nicht Systemuserid!) werden ermittelt und
# außerdem wurden diese Daten in die LOG-Datei geschrieben
sub argument_parsen {
    if (!defined($Getopt::Std::opt_u)){ &usage};
    my $user    = $Getopt::Std::opt_u;
    $user =~ /^fts(\d\d\d)$/ || &usage;
    my $user_id = $1;
    my $warten = $Getopt::Std::opt_w;
    if (defined($warten)) {
        $warten =~ /^\d+$/ || &usage;
    }
    my $alter_handle = select;
    select LOG;
    print "######\n";
    print "#> Die Mail von \"".$user."\" auf $rechnername wird gelesen:\n";
    select $alter_handle;
    my @rueck;
    unshift @rueck, $user, $user_id, $warten;
    return @rueck;
}

sub usage {
    select STDERR;
    print "Usage: receive_pop.pl -u <Benutzer> \n";
    print "receive_pop.pl -u <user>\n";
    print "-l schaltet Logging in eine Datei ein\n";
    print "-d schaltet Debbugging ein\n";
    print "<Benutzer>/<user> ::= fts###\n";
    exit(3);
}

# Liest die Mail mit der Id vom Socket und übergibt sie in einem Array
sub get_msg($) {
    my ($msg_id, $line, $tmp_file);
    $msg_id = shift;
    $tmp_file = &tmp_io::tmp_filename("rec_pop", $msg_id);
    open(TMP, ">$tmp_file");
    &log_io::lprint($socket, "RETR ".$msg_id."\n", \*LOG);
    &test_antwort(&log_io::read_line($socket, \*LOG));
    $line = "default";
    while ($line) {
        $line = <$socket>;
        # Zeilen die mit einem Punkt beginnen, werden mit einem
        # zusätzlichen führenden Punkt übertragen, dieser wird
```

```
        # hier entfernt
        if ($line =~ /^\.\./) {$line =~ s/^\.//; };
        $line =~ tr/\r//d; # da die empfangenen Zeilen ein \r enthalten,
        # daß die Orginale nicht enthalten, wird dieses gleich gefiltert!
        if ($line !~ /^\.\n$/) {
            print TMP $line;
        } else {
            # letzte Zeile erreicht
            undef $line;
        }
    }
    close(TMP);
    return $tmp_file;
}

# Liest die message_ids vom $socket und gibt sie in einem Array zurück
sub list_parsen {
    my (@rueck, $line, $test);
    $test = 1;
    loop: while ($test) {
        $line = <$socket>;
      SWITCH2: for($line) {
          /^(\d*)\s/ && do {
              push @rueck, $1;
              last SWITCH2;};
          /^\.\r\n$/ && do {
              $test = 0;
              last loop;};
      }
#       print $line;
    }
    return @rueck;
}

# Testet eine Rückmeldung von POP3 auf Fehler
# bricht Programm mit Fehlercode ab
sub test_antwort($) {
    my $antwort = shift;
  SWITCH: for($antwort) {
      /^\+OK/ && do {
          $erfolg = 42;
          last SWITCH;};
      /^-ERR/ && do {
          $erfolg = 0;
          last SWITCH;};
      }
    if ($erfolg) {
        return 1;
    } else {
        &log_io::lprint($socket, "QUIT\n", \*LOG);
        &log_io::read_line($socket, \*LOG);
        &ende_des_prog;
    }
}
```

J.3 Empfangen von Mails via NFS (`receive_nfs.pl`)

```
#!/usr/local/bin/perl
# Dieses Programm liest alle Mails ueber NFS und löscht diese!!
# Dabei wird die Vollständigkeit der Mails überprueft und in einer Datei
# mit Sender abgelegt.
# USAGE: receive_nfs.pl -u <user>

use lib ($ENV{"VVTHOME"}.'/targets/fts-target/ifm/perl-share');
use strict;
```

```
use Fcntl qw(:flock);
use Getopt::Std;
# eigene Module
use mail_test;
use tmp_io;

# Wir wollen später auf diese zugreifen... auch im Sig-Handler
my (@msg_names);
my (@msg_ids);

# Pfad der Mail-Bibliothek:
my $mail_bib = $ENV{"VVTHOME"}."/targets/fts-target/ifm/smtp/mail-lib/";

# Mail-Pfad
my $mail_pfad = "/var/spool/mail/";

use sigtrap qw(die normal-signals);

# Name der LOG-Datei:
# (sollte im Home des Senders liegen)
# es wird die Kommunikation von Sendmail-Meldungen, sowie der Rechnername,
# der Dateiname der Mail und die Liste der Empfänger gespeichert!
my $log_datei = ".receive_nfs.log";

# Variable, die den Erfolg mit größer 0 anzeigt;
# am Anfang sind wir erfolgreich ;-)
my $erfolg = 42;

# Kommandozeilenswitches bzgl. Logging und Debuging und username parsen
getopts("ldu:");
# Öffnen der Log-Datei im Appendmode und als Standard Ausgabedatei auswählen
if ($Getopt::Std::opt_d) {
    # open(LOG, ">-"); # for debugging
    open(LOG, ">&STDERR");
} else {
    if ($Getopt::Std::opt_l) {
        open(LOG, ">>".$ENV{"HOME"}."/".$log_datei);
    } else {
        open(LOG, ">>/dev/null");
    }
}
select LOG;

# diese Daten werden in die LOG-Datei geschrieben
my @rueck = &argument_parsen;
my $user = shift(@rueck);

# Öffnen der Maildatei zum Lesen und Schreiben, gelockt, falls Mail da.

#locken...
if (system("lockfile","-6","-r 10",
           $mail_pfad.$user.".lock")) {
    print STDOUT "Timeout beim Filelocking...\n";
    print LOG "Timeout beim Filelocking...\n";
    $erfolg = 0;
} else {
    if (-s $mail_pfad.$user) {
# überprüfen der Schreib- und Leserechte, rootsquash nicht beachtend
        if (!(-r $mail_pfad.$user && -w $mail_pfad.$user)) {
            print STDOUT "Keine Rechte auf Maildrop!\n";
            print LOG"Keine Rechte auf Maildrop!\n";
            $erfolg = 0;
            unlink ($mail_pfad.$user.".lock");
        } else {
            my ($line, $prev_line, $tmp_file);
            my $id = 1;
```

```
            open (INMAIL, $mail_pfad.$user);
            do {
                $line = "";
                $tmp_file = &tmp_io::tmp_filename("rec_nfs", $id);
                push @msg_names, $tmp_file;
                push @msg_ids, $id;
                $id += 1;
                open(TMP, ">$tmp_file");
                do {
                    print TMP $line;
                    $prev_line = $line;
                    $line = <INMAIL>;
                } until ((eof INMAIL)||
                         (($line =~ /^From/)&&($prev_line eq "\n")));
                if (eof INMAIL) {
                    print TMP $line;
                }
                close(TMP);
            } until (eof INMAIL);
            close (INMAIL);
            if (!(-s $msg_names[0])) {
                print STDOUT "Lesefehler... (Root-Squash?)\n";
                print LOG "Lesefehler... (Root-Squash?)\n";
                $erfolg = 0;
                unlink ($mail_pfad.$user.".lock");
                unlink (shift (@msg_names));
                shift (@msg_ids);
            } else {
                open (INMAIL, ">".$mail_pfad.$user);
# Derzeit öffnen und Schließen -> leeren...

#               open (MAIL, "<".$msg_names[0]);
#               my @intdatmail = <MAIL>;
#               close (MAIL);
### Hack:
#               if ($intdatmail[3] eq
#                   "Subject: DON'T DELETE THIS MESSAGE -- ".
#                   "FOLDER INTERNAL DATA\n"){
#                   print STDOUT "Folder Internal Data erkannt.\n";
#                   print INMAIL @intdatmail;
#                   unlink (shift (@msg_names));
#                   shift (@msg_ids);
#               }

                close (INMAIL);
# unlocken der Maildatei
                unlink($mail_pfad.$user.".lock");
# Mail mit der Mailbiliothek vergleichen
                if (scalar @msg_names > 0) {
                    print STDOUT scalar (@msg_names)." Mail(s) wurde(n)",
                    " empfangen, Status der Mail(s) folgt:\n";
                    my $ausg = "";
                    foreach my $tmp_msg_name (@msg_names) {
                        my $msg_id = shift @msg_ids;
                        $ausg .= "$msg_id.: ";
                        $ausg .= &mail_test::test_msg($tmp_msg_name,
                                                      $mail_bib)."\n";
                    }
                    print STDOUT $ausg;
                    print LOG $ausg;
                } else {
                    print STDOUT "Keine Mails im Maildrop.\n";
                    print LOG "Keine Mails im Maildrop.\n";
                }
            }
        }
```

```
    } else {
        print STDOUT "Keine Mails im Maildrop. ",
        "(Es existiert keine bzw. keine gefüllte Maildatei.)\n";
        print LOG "Keine Mails im Maildrop. ",
        "(Es existiert keine bzw. keine gefüllte Maildatei.)\n";
    }
}
# Schließen der Log-Datei
close(LOG);

# Rückmeldung, ob Mail gelesen wurde
# print $erfolg,"\n";
if ($erfolg) {
    exit(0);
} else {
    exit(42);
}


############
#Subroutinen, die die Drecksarbeit erledigen
############
# Argumente parsen:
# user und user_id (nicht Systemuserid!) werden ermittelt und
# außerdem wurden diese Daten in die LOG-Datei geschrieben
sub argument_parsen {
    if (!defined($Getopt::Std::opt_u)){ &usage};
    my $user    = $Getopt::Std::opt_u;
    $user =~ /^fts(\d\d\d)$/ || &usage;
    my $user_id = $1;
    my $alter_handle = select;
    select LOG;
    print "######\n";
    print "#> Die Mail von \"".$user."\" wird gelesen:\n";
    select $alter_handle;
    my @rueck;
    unshift @rueck, $user, $user_id;
    return @rueck;
}

sub usage {
    select STDERR;
    print "Usage: receive_nfs.pl -u <Benutzer> \n";
    print "receive_nfs.pl -u <user>\n";
    print "-l schaltet Logging in eine Datei ein\n";
    print "-d schaltet Debbugging ein\n";
    print "<Benutzer>/<user> ::= fts###\n";
    exit(3);
}

# Abfangen der SIGs zum Beenden des Progs und Löschen der lock-Datei

sub END {
    unlink ($mail_pfad.$user.".lock");
#     if (scalar (@msg_names) > 0) {
#       foreach my $tmp_msg_name (@msg_names) {
#           if (-e($tmp_msg_name)){
#               unlink ($tmp_msg_name);
#           }
#       }
#     }
 }
```

J.4 Empfangen von Mails via ~/.forward-Datei (receive_pop.pl)

```
#!/usr/bin/perl
# Dieses Programm liest alle Mails ueber das POP3 Protokoll von ariane5 und
# löscht diese!!
# Dabei wird die Vollstaendigkeit der Mails ueberprueft und in einer Datei
# mit Sender abgelegt.
# USAGE: receive_pop.pl -u <user>

use lib ($ENV{"VVTHOME"}.'/targets/fts-target/ifm/perl-share');
use strict;
use Getopt::Std;
# eigene Module
use log_io;
use mail_test;
use tmp_io;
use lock;
use status; # braucht: &dienice
use sigtrap qw(die normal-signals);

# Wiederholversuche zum locken
my $MAX_RETRIES = 4;

# Pfad der Mail-Bibliothek:
my $mail_bib = $ENV{"VVTHOME"}."/targets/fts-target/ifm/smtp/mail-lib/";
# Mailverzeichnis
my $mail_verz = (getpwuid($<))[7]."/Mailtest";
# Name der Datei für Status-Infos:
my $status = ".mailstatus";
# Name der LOG-Datei:
# (sollte im Home des Senders liegen)
# es wird die Kommunikation von Sendmail-Meldungen, sowie der Rechnername,
# der Dateiname der Mail und die Liste der Empfänger gespeichert!
my $log_datei = ".receive_forw.log";

# Kommandozeilenswitches bzgl. Logging und Debuging und username parsen
getopts("ldu:");
# Öffnen der Log-Datei im Appendmode und als Standard Ausgabedatei auswählen
if ($Getopt::Std::opt_d) {
    # open(LOG, ">-"); # for debugging
    open(LOG, ">&STDERR");
} else {
    if ($Getopt::Std::opt_l) {
        open(LOG, ">>".$ENV{"HOME"}."/".$log_datei);
    } else {
        open(LOG, ">>/dev/null");
    }
}
select LOG;

#  diese Daten werden in die LOG-Datei geschrieben
&argument_parsen;

# existiert das Mailverzeichnis?
&status::test_verz_or_init($mail_verz);
# Wechseln in das Mailverzeichnis
chdir $mail_verz;
# Versuche zu locken...
my $locked = 0;
my $retries = $MAX_RETRIES;
print LOG "Versuche Statusdatei zu sperren...\n";
do {
    $locked = &lock::lock($status);
    $retries--;
    if(!$locked) {
        print LOG ($MAX_RETRIES - $retries).". Versuch fehlgeschlagen\n";
```

```
        sleep(5);
    }
} while(!$locked && ($retries));
# Locking geklappt?
if (!$locked) {
    exit(0);
}
print "Statusdatei gelockt.\n";
#status-file vorhanden und initialisiert ? sonst initialisieren und anlegen
&status::test_or_init_status($status, $mail_verz) ||
    exit(50);
# Schauen ob Mail zu lesen bis keine mehr "da" ist:
my (@msgnames, @tmp_msgnames, $msgname, $mail_anz);
$mail_anz = 0;
do {
    ($msgname = &status::mail_nr_ermitteln("read", $status)) && do {
        # Mail linken falls $msg_name > 0
        my $tmp_msgname = (getpwuid($<))[7].
            &tmp_io::tmp_filename("rec_forw", $msgname);
        if (-e $msgname) { # nicht vorhandene $msgname werden übersprungen
            link($msgname, $tmp_msgname);
            (stat($msgname))[3] == (stat($tmp_msgname))[3] ||
                die("Seltsames Verhalten von link($msgname, $tmp_msgname)");
            push @tmp_msgnames, $tmp_msgname;
            push @msgnames, $msgname;
            $mail_anz++;
        }
    };
} until ($msgname == 0);
$locked = &lock::unlock($status);
print "Statusdatei freigegeben.\n";
($mail_anz > 0) &&
    (print STDOUT "$mail_anz Mail(s) wurde(n) empfangen,".
     " Status der Mail(s) folgt:\n");
# Mail mit der Mailbiliothek vergleichen
my ($ausg, $msg_id) = ("", 1);
foreach my $tmp_msgname (@tmp_msgnames) {
    my $msgname = shift @msgnames;
    #print "FILE: $msg_id: MSG=$msgname, TMP=$tmp_msgname, BIB=$mail_bib\n";
    my $line = $msg_id++.".: ".
        &mail_test::test_msg($tmp_msgname, $mail_bib)."\n";
    $line =~ / Mail \"8\"/ && (unlink($msgname));
    $ausg .= $line;
}
$ausg eq "" && ($ausg = "Keine Mail(s) gelesen.\n");
print STDOUT $ausg;
print LOG $ausg;
exit(0);

# Sauberes Beenden des Programms
sub END {
    if ($locked) {&lock::unlock($status)}
    #Schließen der Log-Datei
    close(LOG);
}
############
#Subroutinen, die die Drecksarbeit erledigen
############
# Argumente parsen:
# user und user_id (nicht Systemuserid!) werden ermittelt und
# außerdem wurden diese Daten in die LOG-Datei geschrieben
sub argument_parsen {
    if (!defined($Getopt::Std::opt_u)){ &usage};
    my $user    = $Getopt::Std::opt_u;
    $user =~ /^(fts\d\d\d|luettich)$/ || &usage;
    select LOG;
```

```
    print "######\n";
    print "#> Die Mail von \"".$user."\" wird gelesen:\n";
    return $user;
}

sub usage {
    select STDERR;
    print "Usage: receive_pop.pl -u <Benutzer> \n";
    print "receive_pop.pl -u <user>\n";
    print "-l schaltet Logging in eine Datei ein\n";
    print "-d schaltet Debbugging ein\n";
    print "<Benutzer>/<user> ::= fts###\n";
    exit(3);
}

sub dienice($) {
    print LOG shift()."\n";
    exit(40);
}
```

Das folgende Skript wurde in der ~/.forward des Test-Nutzers aufgerufen:

```
#!/usr/bin/perl
# Dieses Programm liest alle Mails ueber das POP3 Protokoll von ariane5 und
# löscht diese!!
# Dabei wird die Vollstaendigkeit der Mails ueberprueft und in einer Datei
# mit Sender abgelegt.
# USAGE: get_mail_pipe.pl -u <user>

use lib ($ENV{"VVTHOME"}.'/targets/fts-target/ifm/perl-share');
use strict;
use Getopt::Std;
use Fcntl;
# eigene Module
use log_io;
use status;
use lock;
use sigtrap qw(die normal-signals);

# Wiederholversuche zum locken
my $MAX_RETRIES = 6;

# Mailverzeichnis
my $mail_verz = (getpwuid($<))[7]."/Mailtest";
# Name der Datei für Status-Infos:
my $status = ".mailstatus";
# Name der LOG-Datei:
# (sollte im Home des Senders liegen)
# es wird nur Kommunikation von Sendmail-Meldungen, sowie der Rechnername,
# der Dateiname der Mail und die Liste der Empfänger gespeichert!
my $log_datei = ".get_mail_pipe.log";

# Kommandozeilenswitches bzgl. Logging und Debuging und username parsen
getopts("ldu:");
# Öffnen der Log-Datei im Appendmode und als Standard Ausgabedatei auswählen
if ($Getopt::Std::opt_d) {
    # open(LOG, ">-"); # for debugging
    open(LOG, ">&STDERR");
} else {
    if ($Getopt::Std::opt_l) {
        open(LOG, ">>".(getpwuid($<))[7]."/".$log_datei);
    } else {
        open(LOG, ">>/dev/null");
    }
}
select LOG;
# diese Daten werden in die LOG-Datei geschrieben
```

```
my $user = &argument_parsen;
# existiert das Mailverzeichnis?
&status::test_verz_or_init($mail_verz);
# Wechseln in das Mailverzeichnis
chdir $mail_verz;
# Versuche zu locken...
my $locked = 0;
my $retries = $MAX_RETRIES;
print LOG "Versuche Statusdatei zu sperren...\n";
do {
    $locked = &lock::lock($status);
    $retries--;
    if(!$locked) {
        print LOG ($MAX_RETRIES - $retries).". Versuch fehlgeschlagen\n";
        sleep(5);
    }
} while(!$locked && ($retries));
# Locking geklappt?
if (!$locked) {
    exit(75);
}
print "Statusdatei gelockt.\n";
#status-file vorhanden und initialisiert ? sonst initialisieren und anlegen
&status::test_or_init_status($status, $mail_verz) ||
    exit(50);
# Datei-Nummer für die Mail besorgen:
my $mail_nr = &status::mail_nr_ermitteln("new", $status);
$locked = &lock::unlock($status);
print "Statusdatei freigegeben.\n";
# Mail in die Maildatei speichern
&put_msg($mail_nr);
print LOG "# Die Mail $mail_nr wurde abgelegt.\n";
exit(0);

# Sauberes Beenden des Programms
sub END {
    if ($locked) {&unlock($status)}
    #Schließen der Log-Datei
    close(LOG);
}

############
#Subroutinen, die die Drecksarbeit erledigen
############
# Argumente parsen:
# user und user_id (nicht Systemuserid!) werden ermittelt und
# außerdem wurden diese Daten in die LOG-Datei geschrieben
sub argument_parsen {
    if (!defined($Getopt::Std::opt_u)){ &usage};
    my $user    = $Getopt::Std::opt_u;
    $user =~ /^(fts(\d\d\d)|luettich)$/ || &usage;
    select LOG;
    print "######\n";
    print "# Die Mail von \"".$user."\" wird von Stdin gelesen:\n";
    return $user;
}

sub usage {
    select STDERR;
    print "Usage: get_mail_pipe.pl -u <Benutzer> \n";
    print "get_mail_pipe.pl -u <user>\n";
    print "-l schaltet Logging in eine Datei ein\n";
    print "-d schaltet Debbugging ein\n";
    print "<Benutzer>/<user> ::= fts###\n";
    exit(3);
}
```

```
# Liest eine Mail von Standard-Input und schreibt sie in eine Datei
# mit übergebenem Namen
sub put_msg($) {
    my ($line, $mail);
    $mail = shift;
    open(MAIL, ">$mail");
    $line = "default";
    while ($line = <STDIN>) {
        print MAIL $line;
    }
    close(MAIL);
}

sub dienice($) {
    print LOG shift()."\n";
    &ende_des_prog(40);
}
```

J.5 Diverse Perl-Module

Die nun folgenden Module wurden von oben angegebenen Skripten gemeinsam genutzt.

J.5.1 `lock.pm`

```
# Dieses Modul sperrt eine Datei gegen gleichzeitigen Zugriff
package lock;

use strict;

sub lock_filename($) {
    return shift(@_).".lock";
}

sub lock($) {
    my $filename = &lock_filename(shift);
    my $zeit = (time % 300);
    my $tmp_filename = "tmp".$$."a$zeit.lock";
    open(TMP_LOCK, ">$tmp_filename") || print "OPENshhit\n";
    close (TMP_LOCK);
    link($tmp_filename, $filename) || print "LINKshit\n";
    my @stat = stat $tmp_filename;
    my $rueck;
    if (defined($stat[3])) {
        if ($stat[3] != 2) { # locking fehlgeschlagen
            $rueck = 0;
        } else { # locking hat geklappt
            $rueck = 1;
        }
    } else {
        die("stat: $tmp_filename nicht gültig");
    }
    unlink ($tmp_filename);
    return $rueck;
}

sub unlock($) {
    my $filename = &lock_filename(shift);
    unlink($filename);
    return 0;
}

1;
```

J.5.2 log_io.pm

```
# diese Funktionen ermöglichen ein mitgeloggtes Schreiben und
# Lesen auf einem file-handle!
package log_io;

use strict;

# Bei Prozessen, die sich forken wichtig:
my $prozess_nummer = -42;

sub setze_proz_nummer($) {
    $prozess_nummer = shift;
}

# liest eine Zeile vom handle, der als zweites Argument anzugeben ist,
# und entfernt "\n" nicht. Falls ein zweites Argument angegeben ist, wird
# auf diesen Handle geloggt!
sub read_line(**) {
    my ($handle, $r_value, $log);
    $handle = shift;
    $log = shift;
    $r_value = <$handle>;
    if (defined($log)) {print $log &_pn.$r_value};
    return $r_value;
}

# schreibt auf den als erstes Argument übergebenen Filehandle und Loggt
# dasselbe auf dem Filehandle LOG mit!!
sub lprint(*$*) {
    my ($handle, $arg, $log);
    $handle = shift;
    $arg = shift;
    $log = shift;
    print $handle $arg;
    if (defined($log)) {print $log &_pn.$arg};
}

sub _pn {
    if ($prozess_nummer != -42) {
        return $prozess_nummer.":";
    } else {
        return "";
    }
}

1;
```

J.5.3 mail_test.pm

```
# Dieses Modul berechnet testet Mails auf Konsitens
# ACHTUNG: Die Variable $mail_bib muß im main-Programm auf den richtigen
# Pfad gesetzt werden, mit "/" am Ende!!!
package mail_test;

use strict;
my ($sender, $tmp_filename, $mail_bib);

# Vergleicht die empfangene Mail (Argument) mit der Mailbiliothek
sub test_msg($) {
    my ($subj_line, $file, @r_head, @o_head,
        $o_line, $r_line, $mail_id);
    $tmp_filename = shift;
    $mail_bib = shift;
    open(TMP, "<$tmp_filename") ||
        die("tmp: ".$file." kann nicht geöffnet werden");
```

```
    # Header der empfangenen Mail parsen ($sender wird gesetzt!)
    @r_head = &__header_parsen(\*TMP);
    # Mail_id aus der Subject-Zeile parsen
    $subj_line = shift @r_head;
    if ($subj_line =~ />>(\d*)<</) {
        $mail_id = $1;
        $subj_line =~ s/>>\d*<</>><</;
    } else {
        $mail_id = -42; # bedeutet nicht in der Mailbibliothek
        &__tidy_up;
        return &__test_ausg($mail_id, "valid");
    }
    unshift @r_head, $subj_line;
    # Originaldatei öffnen
    $file = $mail_bib.$mail_id;
    open(LOCAL, "<$file") ||
        die("original: ".$file." kann nicht geöffnet werden");
    # Headerzeilen der Originalmail parsen
    @o_head = &__header_parsen(\*LOCAL);
    my $i = 0; # Zaehler der Zeilen für Fehlermeldung
    # Header vergleichen
    while ((scalar(@o_head) > 0) && (scalar(@r_head) > 0)) {
        $i++;
        $r_line = shift @r_head;
        $o_line = shift @o_head;
#       my $mo_line = $o_line;
#       my $mr_line = $r_line;
#       $mo_line =~ tr/\r \t/§~+/;
#       $mr_line =~ tr/\r \t/§~+/;
#       print  ":mr=> $mr_line";
#       print  ":mo=> $mo_line";
#       print  ":r => $r_line";
#       print  ":o => $o_line";
        if (!&__zeilen_vgl($r_line, $o_line, $i)) {
            &__tidy_up;
            return &__test_ausg($mail_id, "invalid");
        }
    }
    if ((scalar(@o_head) > 0) || (scalar(@r_head) > 0)) {
        print STDERR "Header war unterschiedlich lang!\n";
        &__tidy_up;
        return &__test_ausg($mail_id, "invalid");
    }
    # Body der Mails vergleichen
    while (($o_line = <LOCAL>) && ($r_line = <TMP>)) {
        $i++;
        if (!&__zeilen_vgl($r_line, $o_line, $i)) {
            print "HLLO $i\n";
            &__tidy_up;
             return &__test_ausg($mail_id, "invalid");
        }
    }
    &__tidy_up;
    return &__test_ausg($mail_id, "valid");
}

# vergleicht zwei Zeilen miteinander, dritter Parameter gibt Zeilennummer an
sub __zeilen_vgl($$$) {
    my $r_line = shift;
    my $o_line = shift;
    my $z = shift;
    if ($o_line ne $r_line) {
        print STDERR "$z:r => $r_line";
        print STDERR "$z:o => $o_line";
        &__tidy_up;
        return 0;
```

```
    } else {
        return 1;
    }
}

# aufraeumen
sub __tidy_up {
    close (LOCAL);
    close(TMP);
    unlink $tmp_filename;
}

# holt alle Informationen aus dem Header
sub __header_parsen(*) {
    my ($handle, $subj_line, $cont_line, $r_line, @rueck);
    $handle = shift;
    do {
        $r_line = <$handle>;
      SWITCH3: for ($r_line) {
          /^[Ff]rom:\s([\w\s]*\s<([\w]*(@[\w]*)?)>|fts\d\d\d)/ && do {
              # sender bestimmen
              my $outer = $1;
              my $inner = $2;
              if (defined($inner)) {
                  $sender = $inner;
              } else {
                  $sender = $outer;
              }
              last SWITCH3;};
          /^Subject:/ && do {
              # Subject-Zeile speichern
              $subj_line = $r_line;
              last SWITCH3;};
          /^Content-Type:/ && do {
              # Content-Type speichern
              $cont_line = $r_line;
              #print "C => $r_line";
              last SWITCH3;};
      }
    } while ($r_line !~ /^\n/);
    if (defined($cont_line)) {
        unshift @rueck, $cont_line;
    }
    unshift @rueck, $subj_line;
    return @rueck;
}

# Macht Ausgabe von Test fertig:
sub __test_ausg($$) {
    my $mail_id = shift;
    my $status = shift;
    return "Mail \"".$mail_id."\" from \"".$sender."\" : \"".$status."\".";
}

1;
```

J.5.4 pass.pm

```
# Dieses Modul berechnet Passwoerter
package pass;

use strict;
sub passwt($) {
    my $id = shift;
    return ((($id * 101)^ 42) % 97);
```

```
}

1;
```

J.5.5 sock.pm

```
# Dieses Modul kann eine socket-Verbindung oeffnen
package sock;
use strict;
use Socket;
use IO::Handle;

sub socket_verbinden(*$$*) {
    my ($handle, $remote,$port, $iaddr, $paddr, $proto, $line);
    $handle = shift;
    $remote  = shift;
    $port    = shift;
    my $prozess_nummer = shift; #optional falls erwünscht
    if ($port =~ /\D/) { $port = getservbyname($port, "tcp") }
    die "No port" unless $port;
    $iaddr   = inet_aton($remote)               || die "no host: $remote";
    $paddr   = sockaddr_in($port, $iaddr);

    $proto   = getprotobyname("tcp");
    socket($handle, PF_INET, SOCK_STREAM, $proto)  || die "socket: $!";
    connect($handle, $paddr)    || die "$prozess_nummer connect: $!";

    # autoflush fuer den socket
    my $oldhandle = select;
    select $handle;
    $| = 1;
    select $oldhandle;
    return $handle;
}

1;
```

J.5.6 status.pm

```
# Dieses Modul stellt Funktionen bereit, um eine Statusdatei zu
# verwalten!
package status;

use strict;
use log_io;
sub test_verz_or_init($) {
    my $mail_verz = shift;
    if (!(-e $mail_verz)) {
        # sonst anlegen!
        mkdir ($mail_verz, 0700);
        if(-e $mail_verz) {
            print "$mail_verz wurde angelegt\n";
        } else {
            die("$!:$mail_verz kann nicht angelegt werden");
        }
    } else {
        #print "VORHANDEN\n";
    }
    return 1;
}

sub test_or_init_status($$) {
    my $status = shift;
    my $mail_verz = shift;
    if (!(-e $status) || -z $status) {
```

```
        # initialisieren
        opendir VERZ, $mail_verz ||
            die("init: $mail_verz kann".
                " nicht geöffnet werden");
        my (@files, $file_name);
        while ($file_name = readdir VERZ) {
            #print LOG "file:".$file_name."\n";
            if ($file_name =~ /^\d+$/) {
                push @files, $file_name;
            }
        }
        closedir VERZ;
        my @mails = sort {$a <=> $b} @files;
        my $last = pop @mails;
        my $first = shift @mails;
        my $out = (defined($first) ? $first."-".$last :
                   (defined($last) ? $last : ""));
        open STATUS, ">$status" ||
            die("init: $status kann".
                " nicht geschrieben werden");
        print LOG "# .mailstatus wird initialisiert\n";
        &log_io::lprint(\*STATUS, "unseen: ".$out."\n", \*main::LOG);
        &log_io::lprint(\*STATUS, "seen:\n", \*main::LOG);
        close STATUS;
        return &test_or_init_status($status, $mail_verz);
    } else {
        # alles okay
        return 1;
    }
}

sub mail_nr_ermitteln ($$) {
    my $which = shift;
    my $status = shift;
    my $nr = 1;
    # Status-datei auslesen
    open (STATUS, "<$status") ||
        die("mail_nr: $status kann nicht gelesen werden");
    my @data = <STATUS>;
    close (STATUS);
    my ($ulast, $unseen, $slast, $ufirst);
    # relevante Zeilen parsen
    foreach my $line (@data) {
        $line =~ /^(un)?seen:\s?(\d+-\d+|\d+)?\n$/ && do {
            $unseen = $1;
            my $tmp = $2;
            if (defined($unseen) && $unseen =~ /^un$/) {
                $ulast  = &parse_last($tmp);
                $ufirst = &parse_first($tmp);
            } else { # seen
                $slast = &parse_last($tmp);

            }
        };
    }
    undef($unseen);
    # Nr. der (neuen) Maildatei bestimmen
    $which eq "new" &&
        ((defined($ulast) && ($nr = (++$ulast))) ||
         (defined($slast) && ($nr = ($ulast = ($slast + 1)))));
    $which eq "read" &&
        ((defined($ufirst) && ($nr = $ufirst++) && (1)) ||
         (return 0));
    # Datei mit neuen Daten schreiben
    print LOG "# .mailstatus wird aktualisiert\n";
    open (STATUS, ">$status") ||
```

```
        die("mail_nr: $status kann nicht geschrieben werden");
    foreach my $line (@data) {
        $which eq "new" && $line =~ /^unseen:\s?(\d+-\d+|\d+)?\n$/ && do {
            my $tmp = $1;
            if (defined($tmp)) {
                $line =~ /(\d+-)(\d+)/ && $line =~ s//$1$nr/ &&
                    undef($tmp);
                defined($tmp) &&
                    $line =~ /(\d+)/ && $line =~ s//$1-$nr/;
            } else {
                $line = "unseen: ".$nr."\n";
            }
        };
        $which eq "read" && $line =~ /^(un)?seen:\s?(\d+-\d+|\d+)?\n$/
        && do {
            $unseen = $1;
            my $tmp = $2;
            if (defined($unseen) && $unseen =~ /^un$/) {
                if (defined($tmp)) {
                    $line =~ /(\d+)(-\d+)/ &&
                        (($ufirst == $ulast && ($line =~ s//$ufirst/)) ||
                         ($line =~ s//$ufirst$2/)) &&
                             undef($tmp);
                    defined($tmp) &&
                        $line =~ /(\d+)/ && $line =~ s///;
                } else {
                    # Hier sollte nichts passieren, da es keine Mail zu
                    # lesen gibt und wir hier nie landen, oder?
                    print "nie\n\n";
                }
            } else { # seen
                if (defined($tmp)) {
                    $line =~ m/(\d+-)(\d+)/ && $line =~ s//$1$nr/ &&
                        undef($tmp);
                    defined($tmp) &&
                        $line =~ m/(\d+)/ && $line =~ s//$1-$nr/;
                } else {
                    $line = "seen: ".$nr."\n";
                }
            }
        };
        &log_io::lprint(\*STATUS, $line, \*main::LOG);
    }
    close (STATUS);
    return $nr;
}

sub parse_last($) {
    my $eing = shift;
    my $tmp2;
    $eing =~ /\d+-(\d+)/ && ($tmp2 = $1);
    defined($tmp2) || ($eing =~ /(\d+)/) && ($tmp2 = $1);
    return $tmp2;
}

sub parse_first($) {
    my $eing = shift;
    my $tmp2;
    $eing =~ /(\d+)-\d+/ && ($tmp2 = $1);
    defined($tmp2) || ($eing =~ /(\d+)/) && ($tmp2 = $1);
    return $tmp2;
}

1;
```

J.5.7 tmp_io.pm

```
# Dieses Modul erzeugt einen Dateinamen, der bzgl. Zeit in Sekunden, PID,
# $prefix und übergebener $mail_id eindeutig ist
package tmp_io;

use strict;
sub tmp_filename($$) {
    my $prefix = shift;
    my $id = shift;
    #$id = (((($id * 101) ^ 42) % 97) * 4);
    my $zeit = substr(time, -4);
    return "/tmp/$prefix.$id.$$.$zeit.zt";
}

1;
```

Anhang K. FTS: Veränderungen am IDS–Pop3–Daemon

Die Veränderungen, die am IDS–Pop3–Daemon vorgenommen wurden, haben wir hier als Patch–Datei angegeben, weil man daran am besten unsere Verbesserungen erkennen kann. Das Format ist das bei Patches übliche *unified diff* Format. Die neu hinzugekommenen Zeilen sind mit „+“ und die entfernten Zeilen mit „–“ am Anfang der jeweiligen Zeile gekennzeichnet.

```
diff -u -r ids-pop3d-0.9.6/apop.c ids-pop3d-0.9.6-fts/apop.c
--- ids-pop3d-0.9.6/apop.c      Tue Jun 15 18:39:35 1999
+++ ids-pop3d-0.9.6-fts/apop.c  Wed Jun 23 10:49:22 1999
@@ -202,6 +202,13 @@
        return ERR_MBOX_LOCK;
      }

+  if (pw->pw_uid > 0)
+    {
+      /* Change the owner of the lockfile, so we can delete it */
+      if (lockfile != NULL)
+       chown(lockfile, pw->pw_uid, -1);
+    }
+
   if ((pw->pw_uid < 1)
 #ifdef MAILSPOOLHOME
   /* Drop mail group for home dirs */
diff -u -r ids-pop3d-0.9.6/extra.c ids-pop3d-0.9.6-fts/extra.c
--- ids-pop3d-0.9.6/extra.c     Tue Jun 15 18:39:35 1999
+++ ids-pop3d-0.9.6-fts/extra.c Wed Jun 23 13:41:38 1999
@@ -134,28 +134,85 @@
 }

 /* This locks the mailbox file, works with procmail locking */
+/* It uses a locking algorithm as described in man page open(2)
+   to avoid race conditions on NFS. */

 int
 pop3_lock (void)
 {
   struct flock fl;
+  struct stat status;
+  char *uniquefile;
+  char *number;
+  char *hostname;
+  int lock;
   fl.l_type = F_RDLCK;
   fl.l_whence = SEEK_CUR;
   fl.l_start = 0;
   fl.l_len = 0;
+
+/* We have to get a unique file
+   the filename contains the hostname and the pid, so it
+   should be unique */
+  hostname = malloc(MAXHOSTNAMELEN+1);
+  if (hostname == NULL)
+    pop3_abquit (ERR_NO_MEM);
+  gethostname(hostname, MAXHOSTNAMELEN);
+
+  number = malloc(10);
+  sprintf (number, "%d", getpid());
+
```

```
+  uniquefile = malloc (strlen (mailbox) + 1 +
+                      strlen (hostname) + 1 +
+                      strlen (number) +
+                      strlen (".lock") + 1);
+  if (uniquefile == NULL)
+    pop3_abquit (ERR_NO_MEM);
+
+  strcpy (uniquefile, mailbox);
+  strcat (uniquefile, ".");
+  strcat (uniquefile, hostname);
+  strcat (uniquefile, ".");
+  strcat (uniquefile, number);
+  strcat (uniquefile, ".lock");
+
   lockfile = malloc (strlen (mailbox) + strlen (".lock") + 1);
   if (lockfile == NULL)
     pop3_abquit (ERR_NO_MEM);
   strcpy (lockfile, mailbox);
   strcat (lockfile, ".lock");
-  lock = fopen (lockfile, "r");
-  if (lock != NULL)
+
+  lock = open(uniquefile, O_CREAT | O_EXCL | O_WRONLY, 0666);
+  if (lock == -1)
+    {
+      /* This shouldn't happen */
+      syslog(LOG_ERR, "%s\n", strerror (errno));
+      return ERR_FILE;
+    }
+
+  link(uniquefile, lockfile);
+      /* Do not use the return value of the link() call (man open(2)) */
+  close(lock);
+  stat(uniquefile, &status);
+
+  if (status.st_nlink != 2)
     {
       free (lockfile);
       lockfile = NULL;
+      unlink (uniquefile);
+      free (uniquefile);
+      free (number);
+      free (hostname);
       return ERR_MBOX_LOCK;
     }
-  lock = fopen (lockfile, "w");
+
+  /* No we have locked the file, so we dont need the unique
+     file any more */
+  unlink (uniquefile);
+  free (uniquefile);
+  free (number);
+  free (hostname);
+
   if (fcntl (fileno (mbox), F_SETLK, &fl) == -1)
     {
       syslog (LOG_ERR, "%s\n", strerror (errno));
@@ -174,7 +231,6 @@
   fl.l_whence = SEEK_CUR;
   if (lockfile != NULL)
     {
-      fclose (lock);
       if (unlink (lockfile) == -1)
        {
          syslog (LOG_ERR, "%s\n", strerror (errno));
diff -u -r ids-pop3d-0.9.6/pop3d.h ids-pop3d-0.9.6-fts/pop3d.h
```

```
--- ids-pop3d-0.9.6/pop3d.h     Tue Jun 15 18:39:35 1999
+++ ids-pop3d-0.9.6-fts/pop3d.h Wed Jun 23 10:23:46 1999
@@ -153,7 +153,6 @@
 char *lockfile;
 char *username;
 FILE *mbox;
-FILE *lock;
 int ifile;
 FILE *ofile;
 time_t curr_time;
diff -u -r ids-pop3d-0.9.6/user.c ids-pop3d-0.9.6-fts/user.c
--- ids-pop3d-0.9.6/user.c      Tue Jun 15 18:39:35 1999
+++ ids-pop3d-0.9.6-fts/user.c  Wed Jun 23 11:04:44 1999
@@ -205,9 +205,14 @@
        pop3_abquit (ERR_NO_MEM);
       state = TRANSACTION;

-      if (pw != NULL && pw->pw_uid > 1)
-       setuid (pw->pw_uid);
-
+      if (pw != NULL && pw->pw_uid > 0)
+       {
+         /* Change the owner of the lockfile, so we can delete it */
+         if (lockfile != NULL)
+           chown(lockfile, pw->pw_uid, -1);
+         setuid (pw->pw_uid);
+       }
+
       messages = NULL;
       pop3_getsizes ();
       fprintf (ofile, "+OK opened mailbox for %s\r\n", username);
```

Anhang L. IPC: Z–Notation

L.1 Names

a, b	identifiers
d, e	declarations (e.g., $a : A;\ b, \ldots : B \ldots$)
f, g	functions
m, n	numbers
p, q	predicates
s, t	sequences
x, y	expressions
A, B	sets
C, D	bags
Q, R	relations
S, T	Schemas
X	schema text (e.g., d, $d \mid p$ or S)

L.2 Definitions

$a == b$	Abbreviation definition
$a ::= b \mid \ldots$	Free type definition (or $a ::= b\langle\langle x \rangle\rangle \mid \ldots$)
$[a]$	Introduction of a given set (or $[a, \ldots]$)
$a\_$	Prefix operator
$\_a$	Postfix operator
$\_a\_$	Infix operator

L.3 Logic

$true$	Logical true constant
$false$	Logical false constant
$\neg\ p$	Logical negation
$p \land q$	Logical conjunction
$p \lor q$	Logical disjunction
$p \Rightarrow q$	Logical implication ($\neg\ p \lor q$)
$p \Leftrightarrow q$	Logical equivalence ($p \Rightarrow q \land q \Rightarrow p$)
$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existention quantification
$\exists_1 X \bullet q$	Unique existatial quantification
let $a == x;\ \ldots \bullet p$	Local definition

L.4 Sets and expressions

$x = y$	Equality of expressions
$x \neq y$	Inequality $(\neg\ (x = y))$
$x \in A$	Set membership
$x \notin A$	Non-membership$(\neg\ (x \in A))$
$\emptyset$	Empty set
$A \subseteq B$	Set inclusion
$A \subset B$	Strict set inclusion$(A \subseteq B \wedge A \neq B)$
$\{x, y, \ldots\}$	Set of elements
$\{X \bullet x\}$	Set comprehension
$\lambda\, X \bullet x$	Lambda-expression — function
$\mu\, X \bullet x$	Mu-expression — unique value
$\mathbf{let}\ a == x;\ \ldots \bullet y$	Local definition
$\mathbf{if}\ p\ \mathbf{then}\ x\ \mathbf{else}\ y$	Conditional expression
$(x, y, \ldots)$	Ordered tuple
$A \times B \times \ldots$	Cartesian product
$\mathbb{P}\, A$	Power set (set of subsets)
$\mathbb{P}_1\, A$	Non-empty power set
$\mathbb{F}\, A$	Set of finite subsets
$\mathbb{F}_1\, A$	Non-empty set of finite subsets
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$\bigcup A$	Generalized union of a set of sets
$\bigcap A$	Generalized intersection of a set of sets
$\mathit{first}\ x$	First element of an ordered pair
$\mathit{second}\ x$	Second element of an ordered pair
$\# A$	Size of a finite set

L.5 Relations

$A \leftrightarrow B$	Relation $(\mathbb{P}(A \times B))$
$a \mapsto b$	Maplet $((a, b))$
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$\mathrm{id}\, A$	Identity relation
$Q \mathbin{;} R$	Forward relational composition
$Q \circ R$	Backward relational composition $(R \mathbin{;} Q)$
$A \lhd R$	Domain restriction
$A \mathbin{⩤} R$	Domain anti-restriction
$R \rhd A$	Range restriction
$R \mathbin{⩥} A$	Range anti-restriction
$R(\!\vert\ A\ \vert\!)$	Relational image
$\mathit{iter}\ n\ R$	Relation composed n times
R^n	Same as $\mathit{iter}\ n\ R$
$R^\sim$	Inverse of relation (R^{-1})
R^*	Reflexive-transitiv closure
R^+	Irreflexive-transitive closure
$Q \oplus R$	Relational overriding $((\mathit{dom} R \mathbin{⩤} Q) \cup R)$
$a \underline{R} b$	Infix relation

L.6 Functions

$A \rightarrowtail\!\!\!\!\mapsto B$	Partial functions
$A \to B$	Total functions
$A \rightarrowtail\!\!\!\mapsto B$	Partial injections
$A \rightarrowtail B$	Total injections
$A \twoheadrightarrow\!\!\!\!\mapsto B$	Partial surjections
$A \twoheadrightarrow B$	Total surjections
$A \rightarrowtail\!\!\!\!\twoheadrightarrow B$	Bijective functions
$A \mapsto\!\!\!\!\!\mapsto B$	Finite partial functions
$A \rightarrowtail\!\!\!\!\mapsto\!\!\!\!\mapsto B$	Finite partial injections
$f\ x$	Function application (or $f(x)$)

L.7 Numbers

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers $\{1, 2, \ldots\}$
$\mathbb{N}_1$	Set of non-zero natural numbers ($\mathbb{N} \setminus \{0\}$)
$m + n$	Addition
$m - n$	Substraction
$m \operatorname{div} n$	Division
$m \operatorname{mod} n$	Modulo arithmetic
$m \leq n$	Less than or equal
$m < n$	Less than
$m \geq n$	Greater than or equal
$m > n$	Greater than
$succ\ n$	Successor function $\{0 \mapsto 1, 1 \mapsto 2, \ldots\}$
$m \mathinner{..} n$	Number range
$min\ A$	Minimum of a set of numbers
$max\ A$	Maximum of a set of numbers

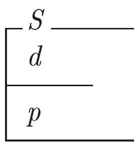
L.8 Sequences

$\mathrm{seq}\ A$	Set of finite sequences
$\mathrm{seq}_1\ A$	Set of non-empty finite sequences
$\mathrm{iseq}\ A$	Set of finite injective sequences
$\langle\rangle$	Empty sequence
$\langle x, y, \ldots\rangle$	Sequence $\{1 \mapsto x, 2 \mapsto y, \ldots\}$
$s \frown t$	Sequence concatenation
cat/s	Distributed sequence concatenation
$head\ s$	First element of sequence ($s(1)$)
$tail\ s$	All but the head element of a sequence
$last\ s$	Last element of sequence ($s(\#s)$)
$front\ s$	All but the last element of a sequence
$rev\ s$	Reverse a sequence
$squash\ f$	Compact a function to a sequence
$A \upharpoonleft s$	Sequence extraction ($squash(A \lhd s)$)
$s \upharpoonright A$	Sequence filtering ($squash(s \rhd A)$)
$s \mathrel{\mathsf{prefix}} t$	Sequence prefix relation ($s \frown v = t$)
$s \mathrel{\mathsf{suffix}} t$	Sequence suffix relation ($u \frown s = t$)
$s \mathrel{\mathsf{in}} t$	Sequence segmentat relation ($u \frown s \frown v = t$)
$\mathsf{disjoint}\ A$	Disjointness of an indexed family of sets
$A \mathrel{\mathsf{partition}} B$	Partition an indexed family of sets

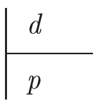
L.9 Bags

bag A	Set of bags or multisets $(A \twoheadrightarrow \mathbb{N}_1)$
$[\![\,]\!]$	Empty bag
$[\![x, y, \ldots]\!]$	Bag $\{x \mapsto 1, y \mapsto 1, \ldots\}$
$count\ C\ x$	Multiplicity of an element in a bag
$C \sharp x$	Same as $count\ C\ x$
$n \otimes C$	Bag scaling of multiplicity
x in C	Bag membership
$C \sqsubseteq D$	Sub-bag relation
$C \uplus D$	Bag union
$C \cup\!\!\!\!- D$	Bag difference
$items\ s$	Bag of elements in a sequence

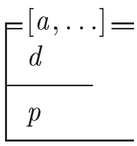
L.10 Schema notation



Vertical schema
New lines denote ';' and '∧'. The schema name and predicate part are optional. The schema may subsequently be refered by name in the document.



Axiomatic definition
The definitions may be non-unique. The predicate part is optional. The definitions apply globally in the document.



Generic definition
The generic parameters are optional. The definitions must be unique. The definitions apply globally in the document.

$S \mathrel{\widehat{=}} [X]$	Horizontal schema
$[T;\ \ldots \mid \ldots]$	Schema inclusion
$z.a$	Component selection (given $z : S$)
θS	Tuple of components
$\neg\ S$	Schema negation
pre S	Schema precondition
$S \land T$	Schema conjunction
$S \lor T$	Schema disjunction
$S \Rightarrow T$	Schema implication
$S \Leftrightarrow T$	Schema equivalence
$S \setminus (a, \ldots)$	Hiding of component(s)
$S \upharpoonright T$	Projection of components
$S \fatsemi T$	Schema composition (S then T)
$S >> T$	Schema piping (S outputs to T inputs)
$S[a/b, \ldots]$	Schema component renaming (b becomes a, etc.)
$\forall X \bullet S$	Schema universal quantification
$\exists X \bullet S$	Schema existential quantification
$\exists_1 X \bullet S$	Schema unique existential quantification

L.11 Conventions

$a?$	Input to an operation
$a!$	Output from an operation
a	State component before an operation
a'	State component after an operation
S	State schema before an operation
S'	State schema after an operation
ΔS	Change of state (normally $S \wedge S'$)
ΞS	No change of state (normally $[S \wedge S' \mid \theta S = \theta S']$)

Anhang M. IPC: Manualpages

M.1 Manpage von msgget

Name

msgget - get a message queue identifier

Synopsis

include <sys/types.h>

include <sys/ipc.h>

include <sys/msg.h>

int msgget (key_t *key*, **int** *msgflg* **)**

Description

The function returns the message queue identifier associated to the value of the *key* argument. A new message queue is created if *key* has value **IPC_PRIVATE** or *key* isn't **IPC_PRIVATE**, no existing message queue is associated to *key*, and **IPC_CREAT** is asserted in *msgflg* (i.e. *msgflg***&***IPC_CREAT* is nonzero). The presence in *msgflg* of the fields **IPC_CREAT** and **IPC_EXCL** plays the same role, with respect to the existence of the message queue, as the presence of **O_CREAT** and **O_EXCL** in the mode argument of the `open`**(2)** system call: i.e. the **msgget** function fails if *msgflg* asserts both **IPC_CREAT** and **IPC_EXCL** and a message queue already exists for *key*.

Upon creation, the lower 9 bits of the argument *msgflg* define the access permissions of the message queue. These permission bits have the same format and semantics as the access permissions parameter in `open`**(2)** or `creat`**(2)** system calls. (The execute permissions are not used.)

Furthermore, while creating, the system call initializes the system message queue data structure **msqid_ds** as follows:

- **msg_perm.cuid** and **msg_perm.uid** are set to the effective user-ID of the calling process.
- **msg_perm.cgid** and **msg_perm.gid** are set to the effective group-ID of the calling process.
- The lowest order 9 bits of **msg_perm.mode** are set to the lowest order 9 bit of *msgflg*.
- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime** and **msg_rtime** are set to 0.
- **msg_ctime** is set to the current time.
- **msg_qbytes** is set to the system limit **MSGMNB**.

If the message queue already exists the access permissions are verified, and a check is made to see if it is marked for destruction.

Return Value

If successful, the return value will be the message queue identifier (a nonnegative integer), otherwise **-1** with **errno** indicating the error.

Errors

For a failing return, **errno** will be set to one among the following values:

- **EACCES** A message queue exists for *key*, but the calling process has no access permissions to the queue.
- **EEXIST** A message queue exists for **key** and *msgflg* was asserting both **IPC_CREAT** and **IPC_EXCL**.
- **EIDRM** The message queue is marked for removal.
- **ENOENT** No message queue exists for *key* and *msgflg* wasn't asserting **IPC_CREAT**.
- **ENOMEM** A message queue has to be created but the system has not enough memory for the new data structure.
- **ENOSPC** A message queue has to be created but the system limit for the maximum number of message queues (**MSGMNI**) would be exceeded.

Notes

IPC_PRIVATE isn't a flag field but a **key_t** type. If this special value is used for *key*, the system call ignores everything but the lowest order 9 bits of *msgflg* and creates a new message queue (on success).

The following is a system limit on message queue resources affecting a **msgget** call:

- **MSGMNI** System wide maximum number of message queues: policy dependent.

Bugs

Use of **IPC_PRIVATE** does not actually prohibit other processes from getting access to the allocated message queue.

As for the files, there is currently no intrinsic way for a process to ensure exclusive access to a message queue. Asserting both **IPC_CREAT** and **IPC_EXCL** in *msgflg* only ensures (on success) that a new message queue will be created, it doesn't imply exclusive access to the message queue.

Conforming to

SVr4, SVID. SVr4 does not document the EIDRM error code.

See Also

ftok**(3)**, ipc**(5)**, msgctl**(2)**, msgsnd**(2)**, msgrcv**(2)**.

M.2 Manpage von msgop

Name

msgop - message operations

Synopsis

include <sys/types.h>

include <sys/ipc.h>

include <sys/msg.h>

int msgsnd (int *msqid*, **struct msgbuf** **msgp*, **int** *msgsz*, **int** *msgflg* **)**

int msgrcv (int *msqid*, **struct msgbuf** **msgp*, **int** *msgsz*, **long** *msgtyp*, **int** *msgflg* **)**

Description

To send or receive a message, the calling process allocates a structure that looks like the following

```
struct msgbuf {
    long    mtype;    /* message type, must be > 0 */
    char    mtext[1];    /* message data */
};
```

but with an array **mtext** of size *msgsz*, a non-negative integer value. The structure member **mtype** must have a strictly positive integer value that can be used by the receiving process for message selection (see the section about **msgrcv**).

The calling process must have write access permissions to send and read access permissions to receive a message on the queue.

The **msgsnd** system call enqueue a copy of the message pointed to by the *msgp* argument on the message queue whose identifier is specified by the value of the *msqid* argument.

The argument *msgflg* specifies the system call behaviour if enqueuing the new message will require more than **msg_qbytes** in the queue. Asserting **IPC_NOWAIT** the message will not be sent and the system call fails returning with **errno** set to **EAGAIN**. Otherwise the process is suspended until the condition for the suspension no longer exists (in which case the message is sent and the system call succeeds), or the queue is removed (in which case the system call fails with **errno** set to **EIDRM**), or the process receives a signal that has to be caught (in which case the system call fails with **errno** set to **EINTR**).

Upon successful completion the message queue data structure is updated as follows:

- **msg_lspid** is set to the process-ID of the calling process.
- **msg_qnum** is incremented by 1.
- **msg_stime** is set to the current time.

The system call **msgrcv** reads a message from the message queue specified by *msqid* into the **msgbuf** pointed to by the *msgp* argument removing from the queue, on success, the read message.

The argument *msgsz* specifies the maximum size in bytes for the member **mtext** of the structure pointed to by the *msgp* argument. If the message text has length greater than *msgsz*, then if the *msgflg* argument asserts **MSG_NOERROR**, the message text will be truncated (and the truncated part will be lost), otherwise the message isn't removed from the queue and the system call fails returning with **errno** set to **E2BIG**.

The argument *msgtyp* specifies the type of message requested as follows:

- If *msgtyp* is **0**, then the message on the queue's front is read.
- If *msgtyp* is greater than **0**, then the first message on the queue of type *msgtyp* is read if **MSG_EXCEPT** isn't asserted by the *msgflg* argument, otherwise the first message on the queue of type not equal to *msgtyp* will be read.
- If *msgtyp* is less than **0**, then the first message on the queue with the lowest type less than or equal to the absolute value of *msgtyp* will be read.

The *msgflg* argument asserts none, one or more (or-ing them) among the following flags:

- **IPC_NOWAIT** For immediate return if no message of the requested type is on the queue. The system call fails with errno set to **ENOMSG**.
- **MSG_EXCEPT** Used with *msgtyp* greater than **0** to read the first message on the queue with message type that differs from *msgtyp*.
- **MSG_NOERROR** To truncate the message text if longer than *msgsz* bytes.

If no message of the requested type is available and **IPC_NOWAIT** isn't asserted in *msgflg*, the calling process is blocked until one of the following conditions occurs:

- A message of the desired type is placed on the queue.

- The message queue is removed from the system. In such a case the system call fails with **errno** set to **EIDRM**.
- The calling process receives a signal that has to be caught. In such a case the system call fails with **errno** set to **EINTR**.

Upon successful completion the message queue data structure is updated as follows:

- **msg_lrpid** is set to the process-ID of the calling process.
- **msg_qnum** is decremented by 1.
- **msg_rtime** is set to the current time.

Return Value

On a failure both functions return **-1** with **errno** indicating the error, otherwise **msgsnd** returns **0** and **msgrvc** returns the number of bytes actually copied into the **mtext** array.

Errors

When **msgsnd** fails, at return **errno** will be set to one among the following values:

- **EAGAIN** The message can't be sent due to the **msg_qbytes** limit for the queue and **IPC_NOWAIT** was asserted in *mgsflg*.
- **EACCES** The calling process has no write access permissions on the message queue.
- **EFAULT** The address pointed to by *msgp* isn't accessible.
- **EIDRM** The message queue was removed.
- **EINTR** Sleeping on a full message queue condition, the process received a signal that had to be caught.
- **EINVAL** Invalid *msqid* value, or nonpositive *mtype* value, or invalid *msgsz* value (less than 0 or greater than the system value **MSGMAX**).
- **ENOMEM** The system has not enough memory to make a copy of the supplied **msgbuf**.

When **msgrcv** fails, at return **errno** will be set to one among the following values:

- **E2BIG** The message text length is greater than *msgsz* and **MSG_NOERROR** isn't asserted in *msgflg*.
- **EACCES** The calling process has no read access permissions on the message queue.
- **EFAULT** The address pointed to by *msgp* isn't accessible.
- **EIDRM** While the process was sleeping to receive a message, the message queue was removed.
- **EINTR** While the process was sleeping to receive a message, the process received a signal that had to be caught.
- **EINVAL** Illegal *msgqid* value, or *msgsz* less than **0**.
- **ENOMSG** **IPC_NOWAIT** was asserted in *msgflg* and no message of the requested type existed on the message queue.

Notes

The followings are system limits affecting a **msgsnd** system call:

- **MSGMAX** Maximum size for a message text: the implementation set this value to 4080 bytes.
- **MSGMNB** Default maximum size in bytes of a message queue: policy dependent. The super-user can increase the size of a message queue beyond **MSGMNB** by a **msgctl** system call.

The implementation has no intrinsic limits for the system wide maximum number of message headers (**MSGTQL**) and for the system wide maximum size in bytes of the message pool (**MSGPOOL**).

Conforming to

SVr4, SVID.

See Also

ipc(**5**), msgctl(**2**), msgget(**2**), msgrcv(**2**), msgsnd(**2**).

M.3 Manpage von msgctl

Name

msgctl - message control operations

Synopsis

include <sys/types.h>

include <sys/ipc.h>

include <sys/msg.h>

int msgctl (int *msqid*, **int** *cmd*, **struct msqid_ds** **buf* **)**

Description

The function performs the control operation specified by *cmd* on the message queue with identifier *msqid*. Legal values for *cmd* are:

- **IPC_STAT** Copy info from the message queue data structure into the structure pointed to by *buf*. The user must have read access privileges on the message queue.
- **IPC_SET** Write the values of some members of the **msqid_ds** structure pointed to by *buf* to the message queue data structure, updating also its **msg_ctime** member. Considered members from the user supplied **struct msqid_ds** pointed to by *buf* are

  msg_perm.uid
  msg_perm.gid
  msg_perm.mode *\/* only lowest 9-bits *\/*
  msg_qbytes

  The calling process effective user-ID must be one among super-user, creator or owner of the message queue. Only the super-user can raise the **msg_qbytes** value beyond the system parameter **MSGMNB**.
- **IPC_RMID** Remove immediately the message queue and its data structures awakening all waiting reader and writer processes (with an error return and **errno** set to **EIDRM**). The calling process effective user-ID must be one among super-user, creator or owner of the message queue.

Return Value

If successful, the return value will be **0**, otherwise **-1** with **errno** indicating the error.

Errors

For a failing return, **errno** will be set to one among the following values:

- **EACCES** The argument *cmd* is equal to **IPC_STAT** but the calling process has no read access permissions on the message queue *msqid*.
- **EFAULT** The argument *cmd* has value **IPC_SET** or **IPC_STAT** but the address pointed to by *buf* isn't accessible.
- **EIDRM** The message queue was removed.
- **EINVAL** Invalid value for *cmd* or *msqid*.

- **EPERM** The argument *cmd* has value **IPC_SET** or **IPC_RMID** but the calling process effective user-ID has insufficient privileges to execute the command. Note this is also the case of a non super-user process trying to increase the **msg_qbytes** value beyond the value specified by the system parameter **MSGMNB**.

Notes

The **IPC_INFO**, **MSG_STAT** and **MSG_INFO** control calls are used by the ipcs**(8)** program to provide information on allocated resources. In the future these can be modified as needed or moved to a proc file system interface.

Conforming to

SVr4, SVID. SVID dies not document the EIDRM error condition.

See Also

ipc**(5)**, msgget**(2)**, msgsnd**(2)**, msgrcv**(2)**.

Anhang N. IPC: Source–Code

N.1 /usr/src/linux/include/linux/ipc.h

```
#ifndef _LINUX_IPC_H
#define _LINUX_IPC_H

#include <linux/types.h>

#define IPC_PRIVATE ((__kernel_key_t) 0)

struct ipc_perm
{
        __kernel_key_t key;
        __kernel_uid_t uid;
        __kernel_gid_t gid;
        __kernel_uid_t cuid;
        __kernel_gid_t cgid;
        __kernel_mode_t mode;
        unsigned short seq;
};

/* resource get request flags */
#define IPC_CREAT 00001000              /* create if key is nonexistent */
#define IPC_EXCL 00002000               /* fail if key exists */
#define IPC_NOWAIT 00004000             /* return error on wait */

/* these fields are used by the DIPC package so the kernel as standard
   should avoid using them if possible */

#define IPC_DIPC 00010000               /* make it distributed */
#define IPC_OWN 00020000                /* this machine is the DIPC owner */

/*
 * Control commands used with semctl, msgctl and shmctl
 * see also specific commands in sem.h, msg.h and shm.h
 */
#define IPC_RMID 0                      /* remove resource */
#define IPC_SET 1                       /* set ipc_perm options */
#define IPC_STAT 2                      /* get ipc_perm options */
#define IPC_INFO 3                      /* see ipcs */

#ifdef __KERNEL__

/* special shmsegs[id], msgque[id] or semary[id]  values */
#define IPC_UNUSED ((void *) -1)
#define IPC_NOID ((void *) -2)          /* being allocated/destroyed */

#endif                                  /* __KERNEL__ */

#endif                                  /* _LINUX_IPC_H */
```

N.2 /usr/src/linux/include/linux/msg.h

```
#ifndef _LINUX_MSG_H
#define _LINUX_MSG_H
```

```c
#include <linux/ipc.h>

/* ipcs ctl commands */
#define MSG_STAT 11
#define MSG_INFO 12

/* msgrcv options */
#define MSG_NOERROR 010000           /* no error if message is too big */
#define MSG_EXCEPT 020000        /* recv any msg except of specified type.*/

/* one msqid structure for each queue on the system */
struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg *msg_first;                  /* first message on queue */
        struct msg *msg_last;                   /* last message in queue */
        __kernel_time_t msg_stime;                  /* last msgsnd time */
        __kernel_time_t msg_rtime;                  /* last msgrcv time */
        __kernel_time_t msg_ctime;                  /* last change time */
        struct wait_queue *wwait;
        struct wait_queue *rwait;
        unsigned short msg_cbytes;       /* current number of bytes on queue */
        unsigned short msg_qnum;           /* number of messages in queue */
        unsigned short msg_qbytes;       /* max number of bytes on queue */
        __kernel_ipc_pid_t msg_lspid;               /* pid of last msgsnd */
        __kernel_ipc_pid_t msg_lrpid;                  /* last receive pid */
};

/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
        long mtype;                                  /* type of message */
        char mtext[1];                                  /* message text */
};

/* buffer for msgctl calls IPC_INFO, MSG_INFO */
struct msginfo {
        int msgpool;
        int msgmap;
        int msgmax;
        int msgmnb;
        int msgmni;
        int msgssz;
        int msgtql;
        unsigned short msgseg;
};

#define MSGMNI 128      /* <= 1K */     /* max # of msg queue identifiers */
#define MSGMAX 4056     /* <= 4056 */    /* max size of message (bytes) */
#define MSGMNB 16384    /* ? */      /* default max size of a message queue */

/* unused */
#define MSGPOOL (MSGMNI*MSGMNB/1024) /* size in kilobytes of message pool */
#define MSGTQL MSGMNB                /* number of system message headers */
#define MSGMAP MSGMNB                /* number of entries in message map */
#define MSGSSZ 16                              /* message segment size */
#define __MSGSEG ((MSGPOOL*1024)/ MSGSSZ)          /* max no. of segments */
#define MSGSEG (__MSGSEG <= 0xffff ? __MSGSEG : 0xffff)

#ifdef __KERNEL__

/* one msg structure for each message */
struct msg {
        struct msg *msg_next;                   /* next message on queue */
        long msg_type;
        char *msg_spot;                         /* message text address */
        time_t msg_stime;                                 /* msgsnd time */
```

```
        short msg_ts;                                   /* message text size */
};

asmlinkage int sys_msgget (key_t key, int msgflg);
asmlinkage int sys_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
asmlinkage int sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp,
                           int msgflg);
asmlinkage int sys_msgctl (int msqid, int cmd, struct msqid_ds *buf);

#endif                                                  /* __KERNEL__ */

#endif                                                  /* _LINUX_MSG_H */
```

N.3 /usr/src/linux/ipc/msg.c

```
/*
 * linux/ipc/msg.c
 * Copyright (C) 1992 Krishna Balasubramanian
 *
 * Removed all the remaining kerneld mess
 * Catch the -EFAULT stuff properly
 * Use GFP_KERNEL for messages as in 1.2
 * Fixed up the unchecked user space derefs
 * Copyright (C) 1998 Alan Cox & Andi Kleen
 *
 */

#include <linux/malloc.h>
#include <linux/msg.h>
#include <linux/interrupt.h>
#include <linux/smp_lock.h>
#include <linux/init.h>

#include <asm/uaccess.h>

extern int ipcperms (struct ipc_perm *ipcp, short msgflg);

static void freeque (int id);
static int newque (key_t key, int msgflg);
static int findkey (key_t key);

static struct msqid_ds *msgque[MSGMNI];
static int msgbytes = 0;
static int msghdrs = 0;
static unsigned short msg_seq = 0;
static int used_queues = 0;
static int max_msqid = 0;
static struct wait_queue *msg_lock = NULL;

void __init msg_init (void)
{
        int id;

        for (id = 0; id < MSGMNI; id++)
                msgque[id] = (struct msqid_ds *) IPC_UNUSED;
        msgbytes = msghdrs = msg_seq = max_msqid = used_queues = 0;
        msg_lock = NULL;
        return;
}

static int real_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
{
        int id;
        struct msqid_ds *msq;
        struct ipc_perm *ipcp;
```

```
        struct msg *msgh;
        long mtype;

        if (msgsz > MSGMAX || (long) msgsz < 0 || msqid < 0)
                return -EINVAL;
        if (get_user(mtype, &msgp->mtype))
                return -EFAULT;
        if (mtype < 1)
                return -EINVAL;
        id = (unsigned int) msqid % MSGMNI;
        msq = msgque [id];
        if (msq == IPC_UNUSED || msq == IPC_NOID)
                return -EINVAL;
        ipcp = &msq->msg_perm;

slept:
        if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI)
                return -EIDRM;

        if (ipcperms(ipcp, S_IWUGO))
                return -EACCES;

        if (msgsz + msq->msg_cbytes > msq->msg_qbytes) {
                if (msgsz + msq->msg_cbytes > msq->msg_qbytes) {
                        /* still no space in queue */
                        if (msgflg & IPC_NOWAIT)
                                return -EAGAIN;
                        if (signal_pending(current))
                                return -EINTR;
                        interruptible_sleep_on (&msq->wwait);
                        goto slept;
                }
        }

        /* allocate message header and text space*/
        msgh = (struct msg *) kmalloc (sizeof(*msgh) + msgsz, GFP_KERNEL);
        if (!msgh)
                return -ENOMEM;
        msgh->msg_spot = (char *) (msgh + 1);

        if (copy_from_user(msgh->msg_spot, msgp->mtext, msgsz))
        {
                kfree(msgh);
                return -EFAULT;
        }

        if (msgque[id] == IPC_UNUSED || msgque[id] == IPC_NOID
                || msq->msg_perm.seq != (unsigned int) msqid / MSGMNI) {
                kfree(msgh);
                return -EIDRM;
        }

        msgh->msg_next = NULL;
        msgh->msg_ts = msgsz;
        msgh->msg_type = mtype;
        msgh->msg_stime = CURRENT_TIME;

        if (!msq->msg_first)
                msq->msg_first = msq->msg_last = msgh;
        else {
                msq->msg_last->msg_next = msgh;
                msq->msg_last = msgh;
        }
        msq->msg_cbytes += msgsz;
        msgbytes += msgsz;
        msghdrs++;
```

```
        msq->msg_qnum++;
        msq->msg_lspid = current->pid;
        msq->msg_stime = CURRENT_TIME;
        wake_up (&msq->rwait);
        return 0;
}

static int real_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int
msgflg)
{
        struct msqid_ds *msq;
        struct ipc_perm *ipcp;
        struct msg *tmsg, *leastp = NULL;
        struct msg *nmsg = NULL;
        int id;

        if (msqid < 0 || (long) msgsz < 0)
                return -EINVAL;

        id = (unsigned int) msqid % MSGMNI;
        msq = msgque [id];
        if (msq == IPC_NOID || msq == IPC_UNUSED)
                return -EINVAL;
        ipcp = &msq->msg_perm;

        /*
         *  find message of correct type.
         *  msgtyp = 0 => get first.
         *  msgtyp > 0 => get first message of matching type.
         *  msgtyp < 0 => get message with least type must be < abs(msgtype).
         */
        while (!nmsg) {
                if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI) {
                        return -EIDRM;
                }
                if (ipcperms (ipcp, S_IRUGO)) {
                        return -EACCES;
                }

                if (msgtyp == 0)
                        nmsg = msq->msg_first;
                else if (msgtyp > 0) {
                        if (msgflg & MSG_EXCEPT) {
                                for (tmsg = msq->msg_first; tmsg;
                                     tmsg = tmsg->msg_next)
                                        if (tmsg->msg_type != msgtyp)
                                                break;
                                nmsg = tmsg;
                        } else {
                                for (tmsg = msq->msg_first; tmsg;
                                     tmsg = tmsg->msg_next)
                                        if (tmsg->msg_type == msgtyp)
                                                break;
                                nmsg = tmsg;
                        }
                } else {
                        for (leastp = tmsg = msq->msg_first; tmsg;
                             tmsg = tmsg->msg_next)
                                if (tmsg->msg_type < leastp->msg_type)
                                        leastp = tmsg;
                        if (leastp && leastp->msg_type <= - msgtyp)
                                nmsg = leastp;
                }

                if (nmsg) {                     /* done finding a message */
                        if ((msgsz < nmsg->msg_ts) && !(msgflg & MSG_NOERROR)) {
```

```
                                return -E2BIG;
                        }
                        msgsz = (msgsz > nmsg->msg_ts)? nmsg->msg_ts : msgsz;
                        if (nmsg == msq->msg_first)
                                msq->msg_first = nmsg->msg_next;
                        else {
                                for (tmsg = msq->msg_first; tmsg;
                                     tmsg = tmsg->msg_next)
                                        if (tmsg->msg_next == nmsg)
                                                break;
                                tmsg->msg_next = nmsg->msg_next;
                                if (nmsg == msq->msg_last)
                                        msq->msg_last = tmsg;
                        }
                        if (!(--msq->msg_qnum))
                                msq->msg_last = msq->msg_first = NULL;

                        msq->msg_rtime = CURRENT_TIME;
                        msq->msg_lrpid = current->pid;
                        msgbytes -= nmsg->msg_ts;
                        msghdrs--;
                        msq->msg_cbytes -= nmsg->msg_ts;
                        wake_up (&msq->wwait);
                        if (put_user (nmsg->msg_type, &msgp->mtype) ||
                            copy_to_user (msgp->mtext, nmsg->msg_spot, msgsz))
                                msgsz = -EFAULT;
                        kfree(nmsg);
                        return msgsz;
                } else {                                /* did not find a message */
                        if (msgflg & IPC_NOWAIT) {
                                return -ENOMSG;
                        }
                        if (signal_pending(current)) {
                                return -EINTR;
                        }
                        interruptible_sleep_on (&msq->rwait);
                }
        }  /* end while */
        return -1;
}

asmlinkage int sys_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
{
        int ret;

        lock_kernel();
        ret = real_msgsnd(msqid, msgp, msgsz, msgflg);
        unlock_kernel();
        return ret;
}

asmlinkage int sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,
        long msgtyp, int msgflg)
{
        int ret;

        lock_kernel();
        ret = real_msgrcv (msqid, msgp, msgsz, msgtyp, msgflg);
        unlock_kernel();
        return ret;
}

static int findkey (key_t key)
{
        int id;
        struct msqid_ds *msq;
```

```
        for (id = 0; id <= max_msqid; id++) {
                while ((msq = msgque[id]) == IPC_NOID)
                        interruptible_sleep_on (&msg_lock);
                if (msq == IPC_UNUSED)
                        continue;
                if (key == msq->msg_perm.key)
                        return id;
        }
        return -1;
}

static int newque (key_t key, int msgflg)
{
        int id;
        struct msqid_ds *msq;
        struct ipc_perm *ipcp;

        for (id = 0; id < MSGMNI; id++)
                if (msgque[id] == IPC_UNUSED) {
                        msgque[id] = (struct msqid_ds *) IPC_NOID;
                        goto found;
                }
        return -ENOSPC;

found:
        msq = (struct msqid_ds *) kmalloc (sizeof (*msq), GFP_KERNEL);
        if (!msq) {
                msgque[id] = (struct msqid_ds *) IPC_UNUSED;
                wake_up (&msg_lock);
                return -ENOMEM;
        }
        ipcp = &msq->msg_perm;
        ipcp->mode = (msgflg & S_IRWXUGO);
        ipcp->key = key;
        ipcp->cuid = ipcp->uid = current->euid;
        ipcp->gid = ipcp->cgid = current->egid;
        msq->msg_perm.seq = msg_seq;
        msq->msg_first = msq->msg_last = NULL;
        msq->rwait = msq->wwait = NULL;
        msq->msg_cbytes = msq->msg_qnum = 0;
        msq->msg_lspid = msq->msg_lrpid = 0;
        msq->msg_stime = msq->msg_rtime = 0;
        msq->msg_qbytes = MSGMNB;
        msq->msg_ctime = CURRENT_TIME;
        if (id > max_msqid)
                max_msqid = id;
        msgque[id] = msq;
        used_queues++;
        wake_up (&msg_lock);
        return (unsigned int) msq->msg_perm.seq * MSGMNI + id;
}

asmlinkage int sys_msgget (key_t key, int msgflg)
{
        int id, ret = -EPERM;
        struct msqid_ds *msq;

        lock_kernel();
        if (key == IPC_PRIVATE)
                ret = newque(key, msgflg);
        else if ((id = findkey (key)) == -1) {                  /* key not used */
                if (!(msgflg & IPC_CREAT))
                        ret = -ENOENT;
                else
                        ret = newque(key, msgflg);
```

```
        } else if (msgflg & IPC_CREAT && msgflg & IPC_EXCL) {
                ret = -EEXIST;
        } else {
                msq = msgque[id];
                if (msq == IPC_UNUSED || msq == IPC_NOID)
                        ret = -EIDRM;
                else if (ipcperms(&msq->msg_perm, msgflg))
                        ret = -EACCES;
                else
                        ret = (unsigned int) msq->msg_perm.seq * MSGMNI + id;
        }
        unlock_kernel();
        return ret;
}

static void freeque (int id)
{
        struct msqid_ds *msq = msgque[id];
        struct msg *msgp, *msgh;

        msq->msg_perm.seq++;
        msg_seq = (msg_seq+1) % ((unsigned)(1<<31)/MSGMNI);
/* increment, but avoid overflow */
        msgbytes -= msq->msg_cbytes;
        if (id == max_msqid)
                while (max_msqid && (msgque[--max_msqid] == IPC_UNUSED));
        msgque[id] = (struct msqid_ds *) IPC_UNUSED;
        used_queues--;
        while (waitqueue_active(&msq->rwait) || waitqueue_active(&msq->wwait)) {
                wake_up (&msq->rwait);
                wake_up (&msq->wwait);
                schedule();
        }
        for (msgp = msq->msg_first; msgp; msgp = msgh ) {
                msgh = msgp->msg_next;
                msghdrs--;
                kfree(msgp);
        }
        kfree(msq);
}

asmlinkage int sys_msgctl (int msqid, int cmd, struct msqid_ds *buf)
{
        int id, err = -EINVAL;
        struct msqid_ds *msq;
        struct msqid_ds tbuf;
        struct ipc_perm *ipcp;

        lock_kernel();
        if (msqid < 0 || cmd < 0)
                goto out;
        err = -EFAULT;
        switch (cmd) {
        case IPC_INFO:
        case MSG_INFO:
                if (!buf)
                        goto out;
        {
                struct msginfo msginfo;
                msginfo.msgmni = MSGMNI;
                msginfo.msgmax = MSGMAX;
                msginfo.msgmnb = MSGMNB;
                msginfo.msgmap = MSGMAP;
                msginfo.msgpool = MSGPOOL;
                msginfo.msgtql = MSGTQL;
                msginfo.msgssz = MSGSSZ;
```

```
        msginfo.msgseg = MSGSEG;
        if (cmd == MSG_INFO) {
                msginfo.msgpool = used_queues;
                msginfo.msgmap = msghdrs;
                msginfo.msgtql = msgbytes;
        }

        err = -EFAULT;
        if (copy_to_user (buf, &msginfo, sizeof(struct msginfo)))
                goto out;
        err = max_msqid;
        goto out;
}
case MSG_STAT:
        if (!buf)
                goto out;
        err = -EINVAL;
        if (msqid > max_msqid)
                goto out;
        msq = msgque[msqid];
        if (msq == IPC_UNUSED || msq == IPC_NOID)
                goto out;
        err = -EACCES;
        if (ipcperms (&msq->msg_perm, S_IRUGO))
                goto out;
        id = (unsigned int) msq->msg_perm.seq * MSGMNI + msqid;
        tbuf.msg_perm = msq->msg_perm;
        tbuf.msg_stime = msq->msg_stime;
        tbuf.msg_rtime = msq->msg_rtime;
        tbuf.msg_ctime = msq->msg_ctime;
        tbuf.msg_cbytes = msq->msg_cbytes;
        tbuf.msg_qnum = msq->msg_qnum;
        tbuf.msg_qbytes = msq->msg_qbytes;
        tbuf.msg_lspid = msq->msg_lspid;
        tbuf.msg_lrpid = msq->msg_lrpid;
        err = -EFAULT;
        if (copy_to_user (buf, &tbuf, sizeof(*buf)))
                goto out;
        err = id;
        goto out;
case IPC_SET:
        if (!buf)
                goto out;
        err = -EFAULT;
        if (!copy_from_user (&tbuf, buf, sizeof (*buf)))
                err = 0;
        break;
case IPC_STAT:
        if (!buf)
                goto out;
        break;
}

id = (unsigned int) msqid % MSGMNI;
msq = msgque [id];
err = -EINVAL;
if (msq == IPC_UNUSED || msq == IPC_NOID)
        goto out;
err = -EIDRM;
if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI)
        goto out;
ipcp = &msq->msg_perm;

switch (cmd) {
case IPC_STAT:
        err = -EACCES;
```

```
			if (ipcperms (ipcp, S_IRUGO))
				goto out;
			tbuf.msg_perm = msq->msg_perm;
			tbuf.msg_stime = msq->msg_stime;
			tbuf.msg_rtime = msq->msg_rtime;
			tbuf.msg_ctime = msq->msg_ctime;
			tbuf.msg_cbytes = msq->msg_cbytes;
			tbuf.msg_qnum = msq->msg_qnum;
			tbuf.msg_qbytes = msq->msg_qbytes;
			tbuf.msg_lspid = msq->msg_lspid;
			tbuf.msg_lrpid = msq->msg_lrpid;
			err = -EFAULT;
			if (!copy_to_user (buf, &tbuf, sizeof (*buf)))
				err = 0;
			goto out;
		case IPC_SET:
			err = -EPERM;
			if (current->euid != ipcp->cuid &&
			    current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN))
			    /* We _could_ check for CAP_CHOWN above, but we don't */
				goto out;
			if (tbuf.msg_qbytes > MSGMNB && !capable(CAP_SYS_RESOURCE))
				goto out;
			msq->msg_qbytes = tbuf.msg_qbytes;
			ipcp->uid = tbuf.msg_perm.uid;
			ipcp->gid = tbuf.msg_perm.gid;
			ipcp->mode = (ipcp->mode & ~S_IRWXUGO) |
				(S_IRWXUGO & tbuf.msg_perm.mode);
			msq->msg_ctime = CURRENT_TIME;
			err = 0;
			goto out;
		case IPC_RMID:
			err = -EPERM;
			if (current->euid != ipcp->cuid &&
			    current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN))
				goto out;

			freeque (id);
			err = 0;
			goto out;
		default:
			err = -EINVAL;
			goto out;
		}
out:
	unlock_kernel();
	return err;
}
```

Anhang O. IPC: Spezifikation der Message-Queues

O.1 Einleitung

Die vorliegende Spezifikation entstand im Rahmen des studentischen Projektes LiVE! , das vom Wintersemester 1997/1998 bis Sommersemester 1999 an der Universität Bremen durchgeführt wurde. Der Sinn und Inhalt des Projektes ist am besten unter `http://aerobee.informatik.uni-bremen.de/start.html` nachzulesen. Da dieses Dokument im Rahmen des Projektes LiVE! entstand, befindet es sich auch im Anhang an den Projektbericht.

In unserer Teilgruppe von LiVE! , nach dem Inter-Process-Communication Paket IPC genannt, beschäftigen wir uns mit der Spezifikation und Verifikation der Message-Queues unter Linux.

Zuerst sollten wir kurz erklären, warum wir uns überhaupt mit Linux und dem IPC-Package beschäftigen, ohne zu sehr auf LiVE! einzugehen. Linux ist ein modernes Betriebssystem, das zudem immer mehr in Mode kommt. Es hat den großen Vorteil, daß nicht nur der Source-Code vorliegt sondern es unter der Open-Source-Lizens steht und somit der Source-Code auch beliebig abwandelbar ist. Der immer größeren Anhängerschar von Linux wollen wir uns anschließen, ebenfalls am Code herumbasteln und einen sinnvollen Beitrag für die Linux-Gemeinde leisten.

Entsprechend den Zielen des Projektes LiVE! wollten wir in unserem Teilprojekt einen Teil von Linux verifizieren. Hierfür benötigten wir einen Abschnitt aus Linux, der zum einen nicht zu lang ist, schon über einen längeren Zeitraum nicht mehr verändert worden ist, zudem weiterhin verwendet werden wird und auch nicht mehr großen Veränderungen unterliegt.

Hier sind wir im IPC-Package mit den Message-Queues fündig geworden. Der Message-Queue Source-Code ist 494 Zeilen lang (entspricht 11598 Bytes), und ist zuletzt am 20.11.1998 verändert worden. Die Kommunikation zwischen den einzelnen Prozessen wird auch in Zukunft sicher benötigt, zudem arbeiten viele Programme mit den Message-Queues und gehen von einem korrekten Funktionieren aus.

Im normalen Software-Entwicklungsprozeß sollte erst eine Spezifikation erstellt werden und anschließend entsprechend implementiert werden. Bei vielen Open-Source-Projekten verläuft dies jedoch anders, es wird implementiert und bestehende Fehler werden durch ausprobieren gefunden und behoben, eine Spezifikation findet niemals statt.

Die Message-Queues sind bisher nicht formal spezifiziert. Es gibt den Source-Code, in dem einige spärliche Kommentare zum besseren Verständnis stehen, die Online-Hilfe, vor allem die *man pages*, die eine informelle Beschreibung versuchen, sowie einige Bücher, die eine teilweise Beschreibung der Funktionsweise der Message-Queues geben.

Entgegen dem normalen Software-Entwicklungsprozeß verwenden wir zur Erstellung der formalen Spezifikation eine informelle Spezifikation und in Zweifelsfällen den Source-Code.

An verschiedenen Stellen waren wir uns über das tatsächliche Systemverhalten nicht ganz im klaren. Um das Verhalten der Message-Queues auszutesten schrieben wir kleine Testprogramme, die sowohl unter Linux als auch unter Solaris angewendet wurden.

Für die formale Spezifikation haben wir als Spezifikationssprache Z ausgewählt. Message-Queues könnte man als Kernel-Datenstrukturen bezeichnen. Um solche Datenstrukturen sowie deren Veränderungen am besten darzustellen erschien uns eine mengenbasierte Spezifikationssprache, wie Z eine ist, am besten geeignet zu sein.

Wir wollen mit unserer Spezifikation keine Einführung in die Sprache Z geben. Diejenigen Leser, die sich so etwas noch wünschen, seien auf die die Bücher „Using Z – Specification, Refinement, and Proof" von Jim Woodcock and Jim Davies [WD96] und „The Z Notation: A Reference Manual" von J. M. Spivey [Spi92] verwiesen.

Als wir mit der Spezifikation anfingen, beschränkten wir uns vorerst auf einige wesentliche Teile. Trotzdem fabrizierten wir auch hier schon Fehler, die erst ersichtlich wurden, als wir die restlichen Teile nach und nach ergänzten. Diese Fehler ließen sich teilweise leicht entfernen, bei einigen mußten wir aber auch die gesamte Spezifikation überarbeiten und umschreiben.

Insgesamt wurde die Spezifikation viel aufwendiger und langwieriger, als zuerst gedacht. Und natürlich ist sie inzwischen wesentlich komplexer als die kleinen konstruierten Beispiele aus den Lehrbüchern.

Unsere Spezifikation findet sich wie folgt in diesem Dokument wieder:

Im Kapitel O.2 führen wir die für die Spezifikation benötigten Datenstrukturen ein. Wir widmen uns erst den benötigten Basistypen, anschließend wenden wir uns den Konstanten zu bevor wir die verwendeten Basisfunktionen einführen.

Anschließend, im Kapitel O.3, widmen wir uns dem Zustandsraum, auf denen die Message-Queues arbeiten. Wir erläutern die dort verwendeten Objekte und erklären auch die Konsistenzbedingungen, die immer gelten müssen.

Das Kapitel O.4, mit der Systeminitialisierung, ist ziemlich kurz. Hier versuchen wir auch den mit der Spezifikationssprache Z nicht so bewandetem Leser einige grundlegende Dinge kurz näher zu bringen.

Es gibt insgesamt vier verschiedene Message-Queue-Operationen, denen wir jeweils ein eigenes Kapitel widmen (Kapitel O.5 bis Kapitel O.8). Es wird jeweils zuerst der vom Benutzer gewünschte Erfolgsfall der Operation beschrieben, dann werden sämtliche Fehlerfälle aufgeführt. Zum Abschluß wird jeweils das User-Interface erstellt, das der Benutzer zu sehen bekommt.

Im letzten Abschnitt (Kapitel O.9) werden wir dann die Ergebnisse vorstellen sowie die Erfahrungen schildern, die wir im Laufe der Zeit erlebt haben.

Inzwischen sind wir zu der Ansicht gelangt, daß die hier vorliegende aktuelle Spezifikation vollständig und korrekt ist. Teile von ihr haben wir auch verifiziert (siehe Projektbericht des Projektes LiVE! oder das Dokument

“IPC Message-Queue, Verifikation“). Sollten noch Unklarheiten auftreten, so bitten wir um Benachrichtigung.

O.2 Datenstrukturen

In diesem Kapitel widmen wir uns zuerst dem Grundlegenden: die Datenstrukturen, mit denen wir später arbeiten.

Sowohl in diesem, als auch in den folgenden Kapiteln, werden wir versuchen für die bessere Verständlichkeit möglichst aussagekräftige Namen zu verwenden. Hierbei haben meist die im Source-Code verwendeten Namen als Vorlage gedient. Allerdings geht an einigen Stellen diese besser Verständlichkeit zu Lasten der Lesbarkeit, da sich lange Wörter mit mathematischen Zeichen gemischt nur schwer lesen lassen. Wir hoffen aber einen guten Kompromiß gefunden zu haben.

Unsere formale Spezifikation soll möglichst abstrakt gehalten werden, auch wenn wir den Source-Code teilweise bei der Erstellung hinzugezogen haben. Wir wollen schließlich eine Beschreibung des nach außen sichtbaren Verhaltens erstellen und keine Wiedergabe interner Abläufe.

Wie das übliche Vorgehen bei einer Z-Spezifikation ist, können wir nicht genau sagen. Die kleinen Beispiele aus den Z-Lehrbüchern konnten dies nicht ganz wiedergeben.

Wir haben bei der Spezifikation mit den grundlegenden Teil der Message-Queues angefangen, uns entsprechende Datenstrukturen erdacht, auf diesen Funktionen definiert und betrachtet, ob somit die Message-Queue-Operationen dargestellt waren. Anschließend haben wir alles verfeinert, mehr Features der Message-Queues hinzugenommen und dabei teilweise die Datenstrukturen abwandeln oder ergänzen müssen.

Entsprechend abstrakt sind einige Datenstrukturen und Funktionen, vor allem solche, die Systemdienste oder Systemstrukturen darstellen. Diese könnten jedoch in anderen Teilen des Betriebssystems, in denen sie benutzt oder bereitgestellt werden, genauer definiert werden, es ist in dieser Spezifikation aber nicht nötig.

Dieses Kapitel beginnt mit den Basistypen, die grundlegend vorhanden sein müssen und mit denen, oder auf denen, die Message-Queues arbeiten.

Sodann werden die Konstanten eingeführt, die systemweit feststehen sollen.

Daran schließen sich die Basisfunktionen an, die im Prinzip jedem Prozeß zur Verfügung stehende Systemdienste darstellen.

O.2.1 Basistypen

Bei den Basistypen kann man zwei verschieden Arten unterscheiden.

Mit der einen Art wird nicht weiter gerechnet, die einzelnen Elemente müssen nur einem bestimmten Typen entsprechen. Also kann man sie wunderschön abstrakt halten.

Bei der anderen Art sieht es schon schwieriger aus, mit ihnen soll auch gerechnet werden. Dementsprechend muß eine Struktur mit spezifiziert werden, eventuell sogar noch entsprechende Rechenoperationen auf dieser Struktur. Da es sich aber bei der Struktur häufig um Zahlen handelt sind diese Rechenoperationen schon implizit mitgegeben.

In einem System gibt es zuerst einmal mehrere Benutzer. Diese haben eine eindeutige Identifikation, die User-ID aus der Menge UID:

$$[UID]$$

Da wir uns in UNIX-artigen Systemen aufhalten gibt es für jeden User auch eine zugehörige Group-ID, diesmal aus der Menge GID:

$$[GID]$$

Ein Benutzer (user), eine Gruppe (group), oder „die ganze Welt“ (other) können das Recht bekommen, in eine Message-Queue zu schreiben, beziehungsweise aus ihr zu lesen. Diese Rechte sind ähnlich den UNIX-Dateirechten, nur das execute-Bit hat keine Auswirkungen, und wurde deshalb auch weggelassen. Man kann es im realen System sogar setzen, nur hat es keinerlei Auswirkungen.

$$PERM ::= u\_write \mid u\_read \mid g\_write \mid g\_read \mid o\_write \mid o\_read$$

Ein Benutzer startet Prozesse. Diese müssen teilweise miteinander oder mit anderen Prozessen kommunizieren. Jeder Prozeß hat daher systemweit eine eindeutige Identifikation, den Prozeßidentifikator:

$$[PID]$$

Für die Kommunikation zwischen den Prozessen können Message-Queues verwendet werden. Für diese Kommunikation müssen verschiedene Prozesse dieselbe Message-Queue adressieren können. Dies geschieht über den KEY:

$$KEY : \mathbb{P}(\mathbb{N} \cup \{0\})$$

Für die einzelnen Operationen wird die Message-Queue jedoch über einen Identifikator adressiert:

$$ID : \mathbb{P}(\mathbb{N} \cup \{0\})$$

In einer Message-Queue werden Nachrichten gesendet. Diese bestehen aus einer Folge von Bytes und einem Typ für die Nachricht.

Da die Struktur der Folgen von Bytes nicht weiter relevant ist, nehmen wir eine Folge von Bytes als einen abstrakten Datentypen. Dies ist ein schönes Beispiel für die Abstraktion, denn eigentlich hätte man hier einzelne

Bytes als Datentypen nehmen müssen, die sich wiederum aus Bits zusammensetzen.

$[BYTES]$

Der Typ einer Nachricht wird in Form einer ganzen Zahl angegeben. Zu beachten ist, daß negative Zahlen zwar nicht als Typ einer Nachricht erlaubt sind, sie aber bei der Message-Queue Operation $msgrcv$ eine spezielle Bedeutung haben.

$$TYPE : \mathbb{P}\,\mathbb{Z}$$

Mit einer Message-Queue können verschiedene Operationen ausgeführt werden, wie sie in den folgenden Kapiteln noch ausführlich beschrieben werden.

Bei den Operationen $msgrcv$ und $msgget$ können dabei einige Flags als Option mitgegeben werden:

$$\begin{aligned} FLAG ::= {} & MSG\_EXCEPT \mid IPC\_NOWAIT \mid MSG\_NOERROR \mid \\ & IPC\_CREAT \mid IPC\_EXCL \end{aligned}$$

Der Operation $msgctl$ können dagegen bestimmte Kommandos als Argumente mitgegeben werden:

$$CMD ::= IPC\_STAT \mid IPC\_SET \mid IPC\_RMID$$

Message-Queue Operationen können eine Anzahl von Fehlern verursachen:

$$\begin{aligned} ERROR ::= {} & EACCES \mid EAGAIN \mid EEXIST \mid EINVAL \mid ENOENT \mid \\ & ENOMSG \mid E2BIG \mid ENOSPC \mid EPERM \mid NOERROR \end{aligned}$$

Und als Abschluß kann einem Prozeß als Returnwert mitgeteilt werden, ob ein Fehler vorliegt (Returnwert -1) oder es eine erfolgreiche Operation war (Returnwert 0 oder eine positive ganze Zahl):

$$RETURN : \mathbb{P}(\mathbb{N} \cup \{0, -1\})$$

O.2.2 Konstanten

Es gibt eine Reihe von Konstanten, die einmalig systemweit festgelegt sind. Diese sind insoweit konstant, als daß Änderungen nur durch Neukompilieren des Linux–Kernels möglich sind.

Die wirklich reale Größe, also die Definition der einzelnen Konstanten, ist an dieser Stelle nicht wichtig und sollte an anderer Stelle im System erfolgen.

Um eine mißbräuchliche Nutzung des Systems zu verhindern bzw. um an keine Systemgrenzen zu stoßen gibt es gewisse Begrenzungen für die Message-Queues.

In einem System sollte es eine Obergrenze für die Anzahl der Message-Queues geben. Die Konstante $MSGMNI$ gibt diese maximale Anzahl von Message-Queues an:

$$MSGMNI : \mathbb{N}$$

Damit eine Message-Queue nicht beliebig voll geschrieben werden kann, gibt es eine maximale Länge einer Message-Queue:

$$MSGMNB : \mathbb{N}$$

Auch Nachrichten sollten eine gewisse Länge nicht überschreiten.

$$MSGMAX : \mathbb{N}$$

Zudem gibt es einige spezielle Elemente aus den im vorherigen Abschnitt deklarierten Basistypen die für einige Operationen benannt sein müssen.

So wird für $msgget$ ein spezielles Element aus KEY benötigt, um eine neue Message-Queue mit einem beliebigen KEY zu erzeugen.

$$IPC\_PRIVATE : KEY$$

Auch wird ein spezielles Element aus UID benötigt, daß den Superuser (root) kennzeichnet:

$$SUPERUSER : UID$$

O.2.3 Basisfunktionen

Im Folgendem werden einige Basisfunktionen eingeführt, die Systemdienste beschreiben, die den einzelnen Prozessen jeweils zur Verfügung stehen.

Einige dieser Systemdienste kommen aus anderen Bereichen des Betriebssystems und werden von uns nicht näher spezifiziert. Dieses müßte bei der Spezifikation dieser Bereiche geschehen.

Es werden zuerst einmal einige Funktionen benötigt, die sich um die Rechtevergabe kümmern.

So gibt es eine Funktion, die zu einer UID alle GIDs der Gruppen zuordnet, in denen der jeweilige User ist:

$$getallgroup : UID \rightarrow \mathbb{P}\, GID$$

Auch gibt es Funktionen, die einer PID die UID bzw. die GID des Prozesses zuordnet. Dieses sind partielle Funktionen, da nur verwendeten PID ein Wert zugeordnet ist.

$$getuid : PID \nrightarrow UID$$
$$getgid : PID \nrightarrow GID$$

Zum Lesen und Schreiben auf einer Message-Queue werden ebenfalls Funktionen benötigt.

So ermittelt eine allgemeinen Funktion, ob ein User auf eine Message-Queue schreiben darf, wobei diese Message-Queue bestimmte Rechte, eine bestimmte UID und eine bestimmte GID hat.

$$\_\ \mathrm{maywrite}\ \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID)$$

$$\begin{aligned}
&\forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet\\
&u1\ \mathrm{maywrite}\ (p, u2, g) \Leftrightarrow (u1 = u2 \wedge u_write \in p) \vee\\
&\qquad (g \in getallgroup(u1) \wedge g_write \in p) \vee\\
&\qquad o_write \in p
\end{aligned}$$

Eine andere allgemeinen Funktion ermittelt, ob ein User von einer Message-Queue lesen darf, welche bestimmte Rechte, eine bestimmte UID und eine bestimmte GID hat.

$$\_\ \mathrm{mayread}\ \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID)$$

$$\begin{aligned}
&\forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet\\
&u1\ \mathrm{mayread}\ (p, u2, g) \Leftrightarrow (u1 = u2 \wedge u_read \in p) \vee\\
&\qquad (g \in getallgroup(u1) \wedge g_read \in p) \vee\\
&\qquad o_read \in p
\end{aligned}$$

Betrachten wir eine einzelne Nachricht, so benötigen wir noch eine Funktion, die nur das Anfangsstück der Länge n der $BYTES$ zurückgibt. Da wir $BYTES$ sehr abstrakt gehalten haben können wir an dieser Stelle diese Funktion nicht näher spezifizieren.

$$truncate : BYTES \times \mathbb{N} \rightarrow BYTES$$

O.3 Message-Queue Zustandsraum

Nach unserer Sichtweise sind Message-Queues Kernel-Datenstrukturen. In diesem Kapitel geht es nun darum, den Zustandsraum, in dem sich diese Datenstrukturen befinden, zu beschreiben.

Hierfür gibt es einige Objekte, die den Zustand der Menge der Message-Queues charakterisieren. Für diese gelten gewisse Konsistenzbedingungen.

Um nicht zu sehr von der gegebenen Implementierung zu abstrahieren haben wir die im Source-Code vorhandenen Strukturen teilweise übernommen.

Insgesamt denken wir mit folgenden zehn Objekten den Zustandsraum hinreichend charakterisieren zu können:

- $getkey$ gibt zu jeder ID einen eindeutigen KEY zurück.
- $msgq$ weist einer ID eine Message-Queue zu.
- $creator$ gibt den Erzeuger einer Message-Queue an.
- $owner$ gibt den Besitzer einer Message-Queue an.
- $group$ gibt die Gruppe einer Message-Queue an.
- $perm$ zeigt die Rechte an, die ein Benutzer für die Message-Queue besitzt.
- $qbytes$ liefert die maximale Länge einer Message-Queue.
- $cbytes$ gibt die Länge einer Message-Queue wieder.
- msg_ts gibt für jede Nachricht aus einer Message-Queue die Länge dieser Nachricht an.
- $errno$ gibt die Art des Fehlers aus.

Diese zehn Objekte, ihr aktuelle Zustand und ihr Veränderungen, sind von nun an bei allen Schemata von Bedeutung.

Die Abstraktion des Zustandraumes für die Kommunikation mit Hilfe von Message-Queues sieht nun folgendermaßen aus:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, MsgQStateSpace\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
getkey : ID \pfun KEY \\
msgq : ID \pfun \mathrm{seq}(TYPE \times BYTES) \\
creator : ID \pfun UID \\
owner : ID \pfun UID \\
group : ID \pfun GID \\
perm : ID \pfun \mathbb{P}\, PERM \\
qbytes : ID \pfun \mathbb{N} \\
cbytes : ID \pfun \mathbb{N} \\
msg\_ts : ID \pfun \mathrm{seq}\, \mathbb{N} \\
errno : ERROR \\
\hline
\# \mathrm{dom}\, getkey \leq MSGMNI \qquad (1) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, getkey \qquad (2) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, creator \qquad (3) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, owner \qquad (4) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, group \qquad (5) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, perm \qquad (6) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, qbytes \qquad (7) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, cbytes \qquad (8) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, msg\_ts \qquad (9) \\
\forall\, i : ID \mid i \in \mathrm{dom}\, msgq \bullet \mathrm{dom}(msgq(i)) = \mathrm{dom}(msg\_ts(i)) \qquad (10) \\
\forall\, i1, i2 : ID \mid i1 \in \mathrm{dom}\, msgq \land i2 \in \mathrm{dom}\, msgq \bullet \qquad (11) \\
\qquad getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \lor getkey(i1) = IPC\_PRIVATE \\
\hline
\end{array}
$$

Dabei haben die in der unteren Hälfte aufgestellten Konsistenzbedingungen folgende Bedeutung:

- (1): Die Anzahl der Message-Queues im System darf nicht größer sein, als im System insgesamt mit $MSGMNI$ erlaubt ist.
- (2) bis (9): Wenn es für eine bestimmte ID eine Message-Queue gibt, dann gibt es für diese ID auch $creator$, $owner$, $group$, $perm$, $qbytes$, $cbytes$ und msg_ts.
- (10): Jeder Nachricht in einer Message-Queue wird auch eine Länge zugeordnet. Dies bedeutet, daß wenn einer ID eine Message-Queue zugeordnet ist, msg_ts diese ID auf eine Sequenz von natürlichen Zahlen abbildet, die jeweils die Länge der einzelnen Nachrichten beschreiben.
- (11): Für alle Elemente aus KEY, außer für $IPC\_PRIVATE$, gilt, daß sie immer eindeutig sind. Dies bedeutet, daß niemals zwei Message-Queues über ein und denselben KEY angesprochen werden können, mit der einzigen Ausnahme, $IPC\_PRIVATE$.

O.4 Systeminitialisierung

Nachdem wir nun den Zustandsraum für die Message-Queues festgelegt haben, geht es in diesem Kapitel nun um die Startbedingungen, sprich die Systeminitialisierung.

Beim Start des Betriebssystems gibt es noch keine Prozesse, diese müssen erst noch erzeugt werden. Daher existieren auch noch keine Message-Queues, diese werden erst später initialisiert.

Entsprechend einfach stellt sich somit die Initialisierung dar: Da noch keine Message-Queues existent sind, ist der Domain der benutzten Funktionen leer. Nur die Funktion, die auf bestimmte Werte abgebildet wird, ist auf den Default-Wert gesetzt.

$Init$

$MsgQStateSpace'$

$$
\begin{array}{l}
\mathrm{dom}\, getkey' = \emptyset \\
\mathrm{dom}\, msgq' = \emptyset \\
\mathrm{dom}\, creator' = \emptyset \\
\mathrm{dom}\, owner' = \emptyset \\
\mathrm{dom}\, group' = \emptyset \\
\mathrm{dom}\, perm' = \emptyset \\
\mathrm{dom}\, qbytes' = \emptyset \\
\mathrm{dom}\, cbytes' = \emptyset \\
\mathrm{dom}\, msg\_ts' = \emptyset \\
errno' = NOERROR
\end{array}
$$

Für die mit Z noch nicht so bewanderten Leser:

Dieser Kasten ist ein Schema, auf der oberen Linie steht der Name des Schemas, in diesem Falle $Init$. In der oberen Hälfte des Schemas, also über dem Querstrich, werden sämtliche Ein- und Ausgabewerte deklariert sowie andere Schemata eingebunden. Mit einem Quote (') wird ein Zustand, der nach dem Schema gelten muß, gekennzeichnet.

Die untere Hälfte haben wir noch einmal aufgeteilt, zuerst stellen wir immer die Vorbedingungen und Nachbedingungen, die für dieses Schema gelten, dar. In diesem Falle sind keine vorhanden.

Anschließend werden sämtliche Objekte des Zustandsraum und ihre eventuellen Veränderungen aufgezeigt.

O.5 Message-Queue Operation $msgget()$

Die Operation $msgget()$ ermöglicht es einem Prozeß eine Message-Queue für sich zugänglich zu machen.

Dabei Teilen wir die Operation in verschiedene Fälle auf. Dieses hat mehrere Gründe.

Zum einen gibt es die natürliche Aufteilung in den erfolgreichen und den nicht erfolgreichen Fall. Diese beiden Fälle werden aber noch jeweils in mehrere Einzelfälle aufgeteilt.

Hierdurch ist eine bessere Lesbarkeit gegeben und auch das Verständnis wird vereinfacht. Auch werden im Source-Code an diesen Stellen Fallunterscheidungen getroffen und somit ein Aufsplitten quasi vorgegeben.

Es gibt verschiedene Möglichkeiten bei $msgget()$ erfolgreich zu sein, je nachdem welche Vorbedingungen vorliegen und welche Nachbedingungen man erreichen will.

Zuerst kann man natürlich eine Message-Queue mit einem bestimmten KEY einrichten, wobei der KEY bisher noch nicht existieren darf.

Dann ist es möglich, sich bei einer Message-Queue, die schon existiert und deren KEY einem bekannt ist, „einzuklinken“.

Und zuletzt ist es möglich definitiv eine neue Message-Queue einzurichten, indem man den besonderen KEY $IPC\_PRIVATE$ vorgiebt.

Natürlich ist es auch möglich, daß ein Fehler auftritt. Dies bedeutet nicht, daß im System etwas fehlerhaft implementiert ist, sondern daß der vom Benutzer gewünschte Erfolgsfall nicht möglich ist.

So kann es vorkommen, daß der Benutzer für die Message-Queue keine Zugriffsrechte hat, daß ein $FLAG$ falsch oder nicht gesetzt ist oder daß die Systemgrenzen überschritten würden.

In allen diesen Fällen wird keine Message-Queue für den Benutzer zur Verfügung gestellt und er erhält statt dessen eine entsprechende Fehlermeldung.

Insgesamt gibt es zwei Fehlerfälle, die wir nicht in unserer Spezifikation darstellen.

Da wäre die Möglichkeit, daß die Message-Queue zwischenzeitlich gelöscht wurde. Die andere Möglichkeit ist, daß nicht mehr genügend Speicher für den Inhalt der Message-Queue vorhanden ist.

Diese beiden Fälle müssen in anderen Teilen des Betriebsystems abgefangen werden, unsere Spezifikation der Message-Queues ist daher immer noch vollständig.

Für den Benutzer existieren diese Unterscheidungen in verschiedene Fehler- und Erfolgsfälle nicht, sie sind für ihn auch nicht weiter relevant.

Daher gibt es das User-Interface, in dem alle Fälle zusammengeführt werden und das nach Aussen sichtbare Verhalten von $msgget()$ simuliert wird.

In diesem Kapitel führen wir zuerst sämtliche Erfolgsfälle auf. Hier versuchen wir die Unterschiede besonders hervorzuheben.

Anschließend stellen wir die Fehlerfälle vor und erklären jeden möglichst ausführlich.

Im letzten Abschnitt erstellen wir das User-Interface, indem wir die einzelnen Schema miteinander kombinieren.

Insgesamt erhalten wir somit eine formale abstrakte Beschreibung der Message-Queue Operation $msgget()$ durch das Schema $msgget$

O.5.1 Erfolgsfall

Alle Erfolgsfällen, aber auch den Fehlerfällen, sind in der ersten Hälfte des Schemas, mit Ausnahme ihres Namens, identisch.

Zuerst werden der Zustandsraum in das Schema eingebunden, auf dem auch Veränderungen aufgeführt werden können.

Dann werden die Ein- und Ausgabewerte aufgeführt, die aus der Aufrufsyntax von $msgget()$ und den Rückgabewerten dieser Operation resultieren. Da diese Werte stets diegleichen sind und für eine spätere Schema-Komposition verwenden wir in allen Schemata dieselben Ein- und Ausgabewerte.

Im folgendem betrachten wir daher nur noch die zweite Hälfte der Schemata.

Fangen wir mit dem erfolgreichem Einrichten einer Message-Queue, die noch nicht existiert, an. Dabei wird ein KEY vorgegeben, der noch nicht an eine andere Message-Queue vergeben ist.

$Msgget1$

$$
\begin{array}{ll}
\Delta MsgQStateSpace & \\
k? : KEY & \\
p? : PID & \\
i! : ID & \\
perm? : \mathbb{P}\, PERM & \\
flag? : \mathbb{P}\, FLAG & \\
r! : RETURN & \\
\hline
k? \notin \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE & (1) \\
i! \notin \operatorname{dom} msgq & (2) \\
\#(\operatorname{dom} msgq) < MSGMNI & (3) \\
IPC\_CREAT \in flag? & (4) \\
r! = i! & (5) \\
 & \\
getkey' = getkey \oplus \{i! \mapsto k?\} & \\
msgq' = msgq \cup \{i! \mapsto \langle\rangle\} & \\
creator' = creator \oplus \{i! \mapsto getuid(p?)\} & \\
owner' = owner \oplus \{i! \mapsto getuid(p?)\} & \\
group' = group \oplus \{i! \mapsto getgid(p?)\} & \\
perm' = perm \oplus \{i! \mapsto perm?\} & \\
qbytes' = qbytes \oplus \{i! \mapsto MSGMNB\} & \\
cbytes' = cbytes \oplus \{i! \mapsto 0\} & \\
msg\_ts' = msg\_ts \oplus \{i! \mapsto \langle\rangle\} & \\
errno' = errno &
\end{array}
$$

Interessant wird es bei den Vor- und Nachbedingungen, da sich hier die Schemata der einzelnen Fälle unterscheiden.

- (1): Bei $Msgget1$ wird ein KEY als Eingabe verlangt, der bisher keiner Message-Queue zugeordnet ist und der auch nicht $IPC\_PRIVATE$ entspricht.
- (2): Die ID, der diese Message-Queue zugewiesen wird, darf noch nicht vorhanden sein.

- (3): Auch darf die Höchstgrenze der maximal im System zulässigen Message-Queues nicht überschritten werden.
- (4): Als Flag muß IPC_CREAT gesetzt sein.
- (5): Zurückgegeben wird als Returnwert die ID.

Auch die Objekte des Zustandsraumes unterliegen gewissen Änderungen, wobei die Konsistenzbedingungen des Zustandraumes natürlich weiterhin gelten müssen.

$errno$ verändert sich nicht. Bei allen anderen Objekten wird die durch die neue ID gekennzeichnete Message-Queue eingefügt.

Weiter geht es mit dem erfolgreichem „Einklinken“ in eine schon existierende Message-Queue. Hierfür muß dem Prozeß der KEY, mit dem die Message-Queue angesprochen wird, bekannt sein.

$$
\begin{array}{l}
\textbf{Msgget2} \\
\hline
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? \in \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE \qquad (1) \\
i! = (\mu\, x : ID \mid getkey(x) = k?) \qquad (2) \\
(getuid(p?) \text{ maywrite } (perm(i!), owner(i!), group(i!)) \lor \\
\quad getuid(p?) \text{ mayread } (perm(i!), owner(i!), group(i!))) \qquad (3) \\
\neg\, (IPC\_CREAT \in flag? \land IPC\_EXCL \in flag?) \qquad (4) \\
r! = i! \qquad (5) \\
\\
getkey' = getkey \oplus \{i! \mapsto k?\} \\
msgq' = msgq \\
creator' = creator \\
owner' = owner \\
group' = group \\
perm' = perm \\
qbytes' = qbytes \\
cbytes' = cbytes \\
msg\_ts' = msg\_ts \\
errno' = errno \\
\hline
\end{array}
$$

Für die Vor- und Nachbedingungen ergibt sich bei $Msgget2$ folgendes:

- (1): Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entsprechen darf.
- (2): Als ID wird dann diejenige ID zurückgegeben, die die Message-Queue für den Prozeß adressiert.
- (3): Der Prozeß muß das Recht haben, Nachrichten in diese Message-Queue zu schreibe oder Nachrichten aus dieser Message-Queue zu lesen.
- (4): Zusätzlich dürfen nicht gleichzeitig die Flags IPC_CREAT und IPC_EXCL gesetzt sein.
- (5): Der zurückgegebene Returnwert entspricht der ID.

Gleichzeitig wirkt sich dieses Schema einzig und allein auf $getkey$ aus dem Zustandsraum aus. Alle anderen Objekte des Zustandraumes bleiben unverändert.

Der hauptsächliche Unterschied zum vorherigen Schema ist, daß der KEY dem Prozeß schon bekannt ist. Dies bedeutet, es wird keine neue Message-Queue erzeugt, daher ist auch die Überprüfung, ob die maximale Anzahl der Message-Queues im System überschritten wird, nicht notwendig. Statt dessen muß für die schon bestehende Message-Queue ein Lese- oder Schreibrecht vorhanden sein.

Somit ändert sich der Zustandsraum nicht allzu sehr, es gibt lediglich eine weitere ID, der der KEY zugeordnet ist.

Nun kommt noch das erfolgreiche Erzeugen einer neuen Message-Queue, wobei man durch Vorgabe von $IPC\_PRIVATE$ ganz sicher geht eine neue Message-Queue zu erzeugen.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\mathit{Msgget3}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? = IPC\_PRIVATE \qquad (1) \\
i! \notin \operatorname{dom} msgq \qquad (2) \\
\#(\operatorname{dom} msgq) < MSGMNI \qquad (3) \\
r! = i! \qquad (4) \\
\\
getkey' = getkey \oplus \{i! \mapsto k?\} \\
msgq' = msgq \oplus \{i! \mapsto \langle\rangle\} \\
creator' = creator \oplus \{i! \mapsto getuid(p?)\} \\
owner' = owner \oplus \{i! \mapsto getuid(p?)\} \\
group' = group \oplus \{i! \mapsto getgid(p?)\} \\
perm' = perm \oplus \{i! \mapsto perm?\} \\
qbytes' = qbytes \oplus \{i! \mapsto MSGMNB\} \\
cbytes' = cbytes \oplus \{i! \mapsto 0\} \\
msg\_ts' = msg\_ts \oplus \{i! \mapsto \langle\rangle\} \\
errno' = errno \\
\hline
\end{array}
$$

Hierbei müssen wiederum gewisse Vor- und Nachbedingungen vorherschen:

- (1): Der KEY muß auf $IPC\_PRIVATE$ gesetzt sein.
- (2): Die ID, der diese Message-Queue zugewiesen wird, darf noch nicht vorhanden sein.
- (3): Auch darf die Höchstgrenze der maximal im System zulässigen Message-Queues nicht überschritten werden.
- (4): Es wird als Returnwert die ID zurückgegeben

Bei den Objekten des Zustandraumes verändert sich $errno$ nicht. Bei allen anderen Objekten wird die durch die neue ID gekennzeichnete Message-Queue eingefügt.

Dieses Schema unterscheidet sich nicht sehr von $Msgget1$ und man hätte sie tatsächlich zusammenfassen können. Dies geschah aber aus zwei Gründen nicht.

Zum einen, da auch im Source-Code eine entsprechende Fallunterscheidung gegeben ist. Zum anderen, um das bessere Verständnis zu erleichtern und eine gewisse Trennung dieser beiden Fälle zu verdeutlichen. Bei $Msgget1$ wird ein KEY vorgegeben, von dem man nicht sicher weiß, ob er schon im System existiert. Daher muß auch das Flag IPC_CREAT gesetzt sein, um sich nicht versehentlich in eine andere Message-Queue mit einzuklinken. Bei $Msgget3$ ist man sich hingegen sicher, stets eine neue

Message-Queue zu erzeugen. Die Folge ist allerdings, daß im System mehrere Message-Queues mit demselben KEY existieren können und in dem Fall eine korrekte Identifikation nur über die ID möglich ist.

O.5.2 Fehlerfall

Für den Fall, daß ein vom Benutzer nicht erwünschtes Ergebnis auftritt, ergibt sich einer der Fehlerfälle. Hierbei ist zu beachten, daß es einige Fälle gibt, die wir in unserer Spezifikation leider nicht mitmodellieren können. Dies wären:

- EIDRM: Die Message-Queue ist zwischenzeitlich zum Löschen markiert worden, aber wird noch von Prozessen benutzt. Dies können wir aufgrund der fehlenden zeitlichen Komponente nicht spezifizieren.
- ENOMEM: Das Betriebssystem hat nicht genügend Speicher, um den gesamten Inhalt der Message-Queue zu speichern. Hierfür ist der Kernel zuständig.

Bei den spezifizierten Fehlerfälle gibt es einige Gemeinsamkeiten.

Die erste Hälfte dieser Schemata sind alle gleich, da sie die selben Ein- und Ausgaben haben. Dies hat die bei den Erfolgsfällen aufgeführten Gründe.

Der Returnwert $r!$ wird stets auf den für Fehlerfälle vorgesehene Wert -1 gesetzt.

Im Zustandsraum wird $errno$ stets auf den Namen des Fehlers gesetzt. Dies ist auch der Name des Schema ohne das vorgesetzte msgget.

Für die restlichen Objekte des Zustandraumes ergeben sich keinerlei Veränderungen. Daher werden sie nicht explizit aufgeführt, da dieses durch das Einbinden des Zustandraumes implizit mit vorgegeben ist.

Im weiteren Text wenden wir uns daher nur den restlichen Vor- und Nachbedingungen zu.

Der erste hier beschriebene Fehlerfall ist der erfolglose Versuch sich in eine Message-Queue einzuklinken, da der Prozeß keine entsprechenden Zugriffsrechte hat.

$$
\begin{array}{|l}
\textit{msggetEACCES} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
k? : \mathit{KEY} \\
p? : \mathit{PID} \\
i! : \mathit{ID} \\
\mathit{perm}? : \mathbb{P}\, \mathit{PERM} \\
\mathit{flag}? : \mathbb{P}\, \mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
k? \in \operatorname{ran} \mathit{getkey} \land k? \neq \mathit{IPC\_PRIVATE} \qquad (1) \\
(\mathbf{let}\ i == (\mu\, x : \mathit{ID} \mid \mathit{getkey}(x) = k?) \bullet \\
\quad \neg\, (\mathit{getuid}(p?)\ \mathrm{maywrite}\ (\mathit{perm}(i), \mathit{owner}(i), \mathit{group}(i))) \land \\
\quad \neg\, (\mathit{getuid}(p?)\ \mathrm{mayread}\ (\mathit{perm}(i), \mathit{owner}(i), \mathit{group}(i)))) \qquad (2) \\
r! = -1 \\
\\
\mathit{errno}' = \mathit{EACCES} \\
\hline
\end{array}
$$

Die hierbei vorherschenden Vor- und Nachbedingungen sehen wie folgt aus:

- (1): Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entspricht.
- (2): Für diese Message-Queue darf der Prozeß weder Schreibrechte noch Leserechte besitzen.

Der zweite Fehlerfall ist der erfolglose Versuch sich in eine Message-Queue einzuklinken, da die beiden Flags IPC_CREAT und IPC_EXCL gesetzt sind.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{msggetEEXIST}}} \\
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? \in \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE \qquad (1) \\
IPC\_CREAT \in flag? \qquad (2) \\
IPC\_EXCL \in flag? \qquad (3) \\
r! = -1 \\
errno' = EEXIST \\
\hline
\end{array}
$$

Wie aus der informellen Beschreibung des Schemas schon fast zu ersehen ist, ergeben sich folgende Vor- und Nachbedingungen:

- (1): Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entspricht.
- (2): Das Flag IPC_CREAT muß gesetzt sein.
- (3): Ebenso muß das Flag IPC_EXCL gesetzt sein.

Bei diesem Fehlerfall gibt es zwei Sichtweisen. Entweder wird versucht sich in eine bestehende Message-Queue einzuklinken, es existiert aber für den angegebenen KEY gar keine Message-Queue, oder man versucht eine neue Message-Queue zu erzeugen, vergißt aber das Flag IPC_CREAT zu setzen.

$$
\begin{array}{|l}
\hline
\textit{msggetENOENT} \\
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? \notin \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE \qquad (1) \\
IPC\_CREAT \notin flag? \qquad (2) \\
r! = -1 \\
errno' = ENOENT \\
\hline
\end{array}
$$

Die Vor- und Nachbedingungen dieses Schemas ergeben sich wieder von selbst:

- (1): Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entspricht.
- (2): Das Flag IPC_CREAT ist nicht gesetzt.

Als letzten Fehlerfall führen wir den Versuch zur Erzeugung einer neuen Message-Queue an, der erfolglos bleibt, da die Systemgrenze $MSGMNI$ der maximalen Anzahl von Message-Queues in einem System überschritten würde.

$$
\begin{array}{|l}
\hline
\textit{msggetENOSPC} \\
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? = IPC\_PRIVATE \lor \\
(IPC\_CREAT \in flag? \land k? \notin \operatorname{ran} getkey) \qquad (1) \\
\#(\operatorname{dom} msgq) \geq MSGMNI \qquad (2) \\
r! = -1 \\
\\
errno' = ENOSPC \\
\hline
\end{array}
$$

Bei diesen Vor- und Nachbedingungen wird es noch einmal interessanter:

- (1): Es wird gefordert, daß entweder der KEY $IPC\_PRIVATE$ entspricht, oder der KEY noch nicht vorhanden ist und gleichzeitig das Flag IPC_CREAT gesetzt ist.
- (2): Die Anzahl der vorhandenen Message-Queues müssen schon größer oder gleich der maximalen Anzahl von Message-Queues, $MSGMNI$, sein.

O.5.3 User-Interface

Für die Sichtweise des Benutzers stellen wir die verschiedene Fälle zu einem einzigen Schema zusammen, da ein Benutzer nicht zwischen diesen Fällen unterscheidet.

Hierfür benutzen wir die Schema-Komposition, bei der das Schema durch die nichtdeterministische Verknüpfung anderer Schemata entsteht. Um diesen Nichtdeterminismus aufzulösen haben wir die Vorbedingungen der einzelnen Schemata stets unterschiedlich gehalten, so daß immer eindeutig ist, welches Schema zum tragen kommt.

So kann man erst einmal sämtliche Erfolgsfälle zusammenfassen. Tritt einer dieser Fälle ein, so erhält der Benutzer ein Ergebnis, wie er es sich erwünscht hat.

$$Msgget \mathrel{\widehat{=}} Msgget1 \lor Msgget2 \lor Msgget3$$

Für den Fehlerfall ergibt sich die Gesamtzahl der Fehler aus der Schemadisjunktion der vier Fehler-Schemata:

$$\begin{aligned} Msggetfault \mathrel{\widehat{=}}\ & msggetEACCES \lor msggetEEXIST \lor msggetENOENT \\ & \lor msggetENOSPC \end{aligned}$$

Für den Benutzer besteht die Aufteilung zwischen Erfolgs- und Fehlerfall nicht. Er benutzt das System und erwartet eine entsprechende Rückmeldung. Somit sollten der Erfolgs- und der Fehlerfall zusammengefaßt werden:

$$msgget \mathrel{\widehat{=}} Msgget \lor Msggetfault$$

Somit ist $msgget$ eine abstrakte formale Spezifizierung des nach Aussen sichtbaren Verhalten der Message-Queue Operation $msgget()$.

O.6 Message-Queue Operation $msgsnd()$

Die Operation $msgsnd()$ wird von den Prozessen dazu benutzt, um in einer vorhandenen und ihnen bekannten Message-Queue Nachrichten zu senden. Eine Nachricht besteht dabei aus einem Typ und einer Anzahl von Bytes.

Auch hier gibt es wieder eine Aufteilung, diesmal in einen erfolgreichen Fall und in mehrere Fehlerfälle. Die Begründung ist immer noch dieselbe wie bei $msgget()$.

Im erfolgreichen Fall muß der Prozeß eine bestehende Message-Queue angeben, in die er seine Nachricht senden will, und für die er natürlich Schreibrechte besitzen muß. Von der Nachricht selbst muß er den Typ und die Länge der Nachricht wissen.

Die hier genannten Fehlerfälle sind keine Fehler im Sinne eines Systemfehlers, sondern das gewünschte Verhalten tritt für den Benutzer nicht ein.

Der erste solche Fehlerfall ist der Versuch in eine Message-Queue zu senden, die schon voll ist.

Sodann kann es passieren, daß der Prozeß für die Message-Queue keine Schreibrechte besitzt.

Und es ist noch möglich, daß es eine falsche Eingabe gibt. Solch eine Eingabe könnte eine *ID* sein, die überhaupt nicht existiert, oder der Typ einer Nachricht, der kein gültiger Wert ist, oder aber die Länge der Nachricht, die nicht einem korrektem Wert entspricht.

Es gibt auch wieder einige Fehlerfälle, die wir in unserer Spezifikation nicht anführen. Hierfür gibt es zweierlei Gründe.

Zum einen werden einige dieser Fehlerfälle in anderen Teilen des Betriebsystems abgefangen. Hierunter fallen die Situationen, daß ein Pointer auf eine nicht existierende Adresse zeigt, die Message-Queue inzwischen gelöscht wurde und das das System nicht genügend Speicher hat, um den gesamten Inhalt der Message-Queue zu speichern.

Zum anderen existiert in der Spezifikationssprache Z keine Zeitkomponente, ein zeitlicher Verlauf und insbesondere das Warten auf ein Ereignis kann daher nicht dargestellt werden. Und daher kann in unserer Spezifikation das Warten auf eine Message-Queue und gleichzeitige Erhalten eines Interruptes nicht angeführt werden.

Für den Benutzer ist dies wieder völlig unwichtig, er benutzt $msgsnd()$ und erwartet ein Ergebnis. Um dieses abstrakt darzustellen erstellen wir ein User-Interface, in dem wir das nach Außen sichtbare Verhalten spezifizieren.

In diesem Kapitel fangen wir mit dem Erfolgsfall an, beschreiben anschließend die verschiedenen Fehlerfälle und erstellen zum Schluß das User-Interface. In dem letzten Abschnitt gehen wir auch noch mal näher auf die nicht spezifizierten Fehlerfälle ein.

Somit erhalten wir abschließend eine abstrakte formale Spezifikation des Verhaltens der Message-Queue Operation $msgsnd()$.

O.6.1 Erfolgsfall

Die erste Hälfte aller Schemata für $msgsnd()$ sieht gleich aus, um bei der Erstellung des User-Interfaces sämtliche Schemata miteinander kombinieren zu können, bzw. weil die Aufrufsyntax der Operation stets gleich ist.

Zuerst wird stets der Zustandsraum eingebunden, auf dem Veränderungen stattfinden können.

Dann gibt es als Eingaben die Prozeßidentifikation PID sowie die Identifikation einer Message-Queue ID. Um eine Nachricht zu Senden benötigt man eine Nachricht, insbesondere ihren Typen und eine Anzahl von $BYTES$, die dabei eine bestimmte Länge besitzt. Eventuell können auch Flags vorgegeben werden.

Als Rückgabe erhält man einen Returnwert, bei den Fehlerfällen wird dieser stets auf -1, beim Erfolgsfall auf 0 gesetzt.

Der Erfolgsfall ist das erfolgreiche Senden einer Nachricht in einer Message-Queue. Es muß natürlich eine existierende Message-Queue adressiert sein und der Prozeß muß für diese Message-Queue auch Schreibrechte besitzen.

$$
\begin{array}{|l}
\textit{Msgsnd} \\
\hline
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \qquad (1) \\
getuid(p?)\ \mathrm{maywrite}\ (perm(i?), owner(i?), group(i?)) \qquad (2) \\
cbytes(i?) + msgsz? \leq qbytes(i?) \qquad (3) \\
msgsz? \leq MSGMAX \qquad (4) \\
msgsz? \geq 0 \qquad (5) \\
t? > 0 \qquad (6) \\
r! = 0 \qquad (7) \\
\\
getkey' = getkey \\
msgq' = msgq \oplus \{i? \mapsto msgq(i?) \frown \langle (t?, b?) \rangle \} \\
creator' = creator \\
owner' = owner \\
group' = group \\
perm' = perm \\
qbytes' = qbytes \\
cbytes' = cbytes \oplus \{i? \mapsto cbytes(i?) + msgsz?\} \\
msg\_ts' = msg\_ts \oplus \{i? \mapsto msg\_ts(i?) \frown \langle msgsz? \rangle \} \\
errno' = errno \\
\hline
\end{array}
$$

In der zweiten Hälfte des Schema sind die Vor- und Nachbedingungen angegeben:

- (1): Die ID muß eine schon existierende Message-Queue adressieren.
- (2): Der Prozeß muß auf der durch die ID bezeichneten Message-Queue Schreibrechte besitzen.
- (3): Durch die zu sendende Nachricht darf die Länge der Message-Queue die maximale Länge der Message-Queue nicht überschreiten.
- (4): Die Länge der Nachricht darf nicht die maximale Länge einer Nachricht überschreiten.
- (5): Genauso darf die Länge einer Nachricht nicht einem negativem Wert entsprechen.
- (6): Der Typ der Nachricht muß positiv sein.
- (7): Daraufhin gibt es als Returnwert die 0 als Zeichen des erfolgreichen Sendens der Nachricht.

Auch die Objekte des Zustandraumes unterliegen gewissen Änderungen, wobei die Konsistenzbedingungen des Zustandraumes nicht verletzt werden.

Da wäre zuerst $msgq$, bei der an die existierende Message-Queue die Nachricht mit Angabe des Typen hinten angehängt wird.

Bei $cbytes$ wird die Länge der Message-Queue um die Länge der gesendeten Nachricht erhöht.

Und bei msg_ts wird noch die Länge für die letzte Nachricht eingefügt.

Die restlichen Objekte bleiben vom Senden einer Nachricht unberührt und somit unverändert bestehen.

O.6.2 Fehlerfall

Einige der möglichen Fehlerfälle sind in unserer Spezifikation nicht enthalten. Dieses hat natürlich sein Gründe. Entweder wird der Fehlerfall nicht in dem von uns spezifizierten Teil des Betriebssystems, den Message-Queues, abgefangen, oder er ist in Z nicht darstellbar.

Gehört ein Fehlerfall nicht in den Bereich der Message-Queues, so ist unsere Spezifikation natürlich weiterhin vollständig. Dieser Fall würde bei der Spezifikation des entsprechenden Teils des Betriebsystem mit zu erfassen sein.

Leider gibt es auch den Fall, daß die Sprache Z nicht mächtig genug ist, insbesondere durch das Fehlen einer zeitlichen Komponente. Diesen Fall müßten wir eigentlich mit spezifizieren, können dies aber nicht. Unsere Spezifikation ist an dieser Stelle also nicht mehr vollständig, treten entsprechende Vorbedingungen auf, so geht sie in einen unspezifizierten Zustand über.

Dieses läßt sich leider nicht vermeiden, aber in anderen Spezifikationssprachen hätten wir größere Probleme bekommen.

Hier sind die von uns nicht mitspezifizierten Fehlerfälle:

- EFAULT: Ein Pointer zeigt auf eine nicht existierende Adresse. Hierfür ist der Kernel zuständig
- EIDRM: Die Message-Queue ist gelöscht worden, während der Prozess blockiert war. Dies können wir aufgrund der fehlenden zeitlichen Komponente von Z nicht spezifizieren.
- EINTR: Der Prozeß wartet auf eine Message-Queue und erhält einen Interrupt. Dies kann wegen der fehlenden Zeitkomponente in Z nicht dargestellt werden.
- ENOMEM: Das System hat nicht genügend Speicher um den gesamten Inhalt der Message-Queue zu speichern. Hierfür ist der Kernel zuständig.

Es gibt bei allen Fehlerfällen Gemeinsamkeiten, die wir hier anführen wollen, um sie nicht stets ansprechen zu müssen.

Die erste Hälfte des Schemas entspricht immer den schon beschriebenen üblichen Ein- und Ausgaben bei $msgsnd()$.

Der Returnwert $r!$ wird auf den für Fehlerfälle üblichen Wert von -1 gesetzt.

Für den Zustandsraum gilt allgemein, daß $errno$ auf den Namen des Fehlers gesetzt wird, also den Namen des Schema ohne das vorgefügte msgsnd.

Die restlichen Objekte des Zustandraumes unterliegen keinerlei Veränderungen, werden deshalb hier auch nicht aufgeführt. Durch das Einbinden des Zustandsraumes ist dieses schon implizit mit vorgegeben.

Nun kann der Fall eintreten, daß das Senden einer Nachricht in einer Message-Queue erfolglos ist, da die Message-Queue schon voll ist. Gleichzeitig muß das Flag IPC_NOWAIT gesetzt sein.

$$
\begin{array}{|l}
\hline
\textit{msgsndEAGAIN} \\
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \qquad (1) \\
cbytes(i?) + msgsz? > qbytes(i?) \qquad (2) \\
IPC\_NOWAIT \in f? \qquad (3) \\
r! = -1 \\
errno' = EAGAIN \\
\hline
\end{array}
$$

Damit dieses Schema ausgeführt werden kann, muß eine ID vorgegeben werden, für die eine Message-Queue existiert (1). Auch muß die Länge der Message-Queue mit der Länge der zu sendenden Nachricht die maximale Länge dieser Message-Queue überschreiten (2). Zudem muß das Flag IPC_NOWAIT gesetzt sein (3).

Weiter geht es mit dem Fall, daß das Senden einer Nachricht erfolglos ist, da der Prozeß für die angegebene Message-Queue keine Schreibrechte besitzt.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{msgsndEACCES}} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
t? : \mathit{TYPE} \\
b? : \mathit{BYTES} \\
\mathit{msgsz}? : \mathbb{N} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
f? : \mathbb{P}\, \mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} \mathit{msgq} \qquad (1) \\
\neg\ (\mathit{getuid}(p?)\ \mathrm{maywrite}\ (\mathit{perm}(i?), \mathit{owner}(i?), \mathit{group}(i?))) \qquad (2) \\
r! = -1 \\
\mathit{errno}' = \mathit{EACCES} \\
\hline
\end{array}
$$

Für den Eintritt dieses Schema muß wieder eine gültige *ID* angegeben sein (1). Der Prozeß darf aber keine Schreibrechte für diese Message-Queue besitzen (2).

Der nächste Fall tritt bei verschiedenen Möglichkeiten ein, stets ist jedoch eine falsche Eingabe vorgegeben.

$$
\begin{array}{|l}
\hline
\textit{msgsndEINVAL} \\
\Delta \textit{MsgQStateSpace} \\
t? : \textit{TYPE} \\
b? : \textit{BYTES} \\
\textit{msgsz}? : \mathbb{N} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
f? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
i? \notin \operatorname{dom} \textit{msgq} \lor \qquad (1) \\
\quad t? \leq 0 \lor \qquad (2) \\
\quad \textit{msgsz}? < 0 \lor \qquad (3) \\
\quad \textit{msgsz}? > \textit{MSGMAX} \qquad (4) \\
r! = -1 \\
\textit{errno}' = \textit{EINVAL} \\
\hline
\end{array}
$$

Entweder wird hier eine falsche Message-Queue-ID vorgegeben (1), oder der Typ einer Nachricht ist nicht richtig (2), oder aber die Länge der Nachricht ist zu klein (3) oder zu groß (4).

O.6.3 User-Interface

Die Aufteilung in die verschiedenen Fälle ergibt sich für den Benutzer nicht, er sieht nur das Verhalten einer einzelnen Operation und erhält entsprechend seinen Vorgaben Rückgaben und Veränderungen. Um dieses nach außen sichtbare Verhalten wiederzuspiegeln, fügen wir alle erstellten Schemata zusammen.

Dies geschieht mit einer Schemadisjunktion, die ermöglicht wird, da wir bei allen Schemata dieselben Ein- und Ausgaben benutzen.

Da hier der seltene Fall eintritt, daß wir den Erfolgsfall nicht weiter unterteilen mußten, brauchen wir für ihn auch keine Schemadisjunktion durchführen.

Tritt ein für den Benutzer nicht gewünschtes Ereignis ein, so haben wir hierfür die Fehlerfälle. Weiter vorne haben wir einige Fälle angeführt, die wir in unserer Spezifikation nicht darstellen.

Ansonsten ergibt sich die Gesamtzahl der Fehler aus der Schemadisjunktion:

$$Msgsndfault \mathrel{\widehat{=}} msgsndEAGAIN \lor msgsndEACCES \lor msgsndEINVAL$$

Für den Benutzer ergibt sich die Unterteilung in Erfolgs- und Fehlerfall nicht, er ruft nur eine einzige Operation auf. Daher führen wir auch noch diese beiden Fälle zusammen:

$$msgsnd \mathrel{\widehat{=}} Msgsnd \lor Msgsndfault$$

Somit entspricht das formal spezifizierte abstrakte $msgsnd$ dem nach Aussen sichtbaren Verhalten der Message-Queue Operation $msgsnd()$.

O.7 Message-Queue Operation $msgrcv()$

Bei der Operation $msgrcv()$ kann ein Prozeß aus einer ihm bekannten Message-Queue eine Nachricht auslesen.

Dieses ist die am kompliziertesten darzustellende Operation. Daher müssen wir die meisten Fallunterscheidungen aller Message-Queue Operationen treffen, um die Verständlichkeit und Lesbarkeit noch halbwegs zu gewährleisten.

Im erfolgreichen Fall ist natürlich stets vorausgesetzt, daß eine korrekte Message-Queue adressiert ist und der Prozeß auch Leserechte besitzt. Nun gibt es allerdings noch vier verschiedene Möglichkeiten, eine Nachricht aus dieser Message-Queue zu lesen.

Zuerst wäre dies das Lesen der ersten Nachricht, die in der Message-Queue existiert. Dabei ist es egal, welchen Typ diese besitzt.

Dann wäre noch das Lesen der ersten Nachricht, die einem bestimmten Typ entspricht.

Weiter geht es mit der Möglichkeit, die erste Nachricht zu lesen, die nicht einem bestimmten Typ besitzt.

Und als letztes existiert noch die Möglichkeit die erste Nachricht des kleinsten Typen der Message-Queue zu lesen, wobei diese allerdings kleiner oder gleich einem vorgegebenen Typ sein muß.

Durch den Aufbau der Erfolgsfälle ist es aber leider so, daß einige Fehlerfälle nicht allgemein für die Operation spezifiziert werden können, da sie vom Typ der Nachricht abhängen.

Dies ist dann gleich der Fall, wenn die Länge einer Nachricht größer als erlaubt ist, und das Flag $IPC\_NOERROR$ nicht gesetzt ist. Hierbei ergeben sich also insgesamt vier Fehlerfälle.

Hat der Prozeß hingegen für die Message-Queue keine Leserechte, so ist dies ein allgemeines Problem, und es gibt einen weiteren Fehlerfall.

Genauso allgemein ist es, wenn ein fehlerhafter Wert angegeben wird. Dies kann eine nicht existierende ID oder eine negative Nachrichtenlänge sein.

Und wieder vom Typ der Nachricht abhängig ist der Fall, daß keine Nachricht durch den angegebenen Typen gekennzeichnet wird. Also entstehen bedingt durch die Anzahl der Erfolgsfällen vier Fehlerfälle.

Aus den schon weiter vorne angeführten Gründen gibt es wieder einige Fehlerfälle, die wir nicht spezifizieren.

Diese Aufteilung in verschiedene Fälle entstanden wie schon erwähnt zum besseren Verständnis und besseren Lesbarkeit der Spezifikation. Für den Benutzer eines Systems geht es aber nur um das Resultat seiner Eingaben. Daher erstellen wir für ihn ein User-Interface.

Doch zuerst beschreiben wir die verschiedenen Erfolgsfälle, anschließend die Fehlerfälle und erstellen dann das User-Interface.

Somit erhalten wir abschließend eine abstrakte formale Spezifikation des Verhaltens der Message-Queue Operation $msgrcv()$.

O.7.1 Erfolgsfall

Aus den schon beschriebenen Gründen haben wieder alle Schemata dieselben Ein- und Ausgaben.

Es wird der Zustandsraum eingebunden, auf dem Veränderungen ausgeführt werden können. Mit der ID wird eine Message-Queue bezeichnet und mit der PID herausgefunden, ob ein Leserecht vorliegt. Es wird natürlich ein Typ für die Nachricht vorgegeben, der aber verschiedene Bedeutung haben kann. Die zu lesende Nachricht darf eine gewisse Länge nicht überschreiten. Und es können Flags gesetzt werden.

Zurückgegeben wird der Typ einer Nachricht, der von dem vorgegbenen Typen je nach dem entsprechenden Fall, unterschiedlich sein kann. Die Nachricht selbst wird natürlich auch zurückgegeben, genauso wie ein Returnwert, der im Erfolgsfall immer der Länge der gelesenen Nachricht entspricht.

Um das Verständnis zu erleichtern und die einzelnen Erfolgsfälle besser miteinander vergleichen zu können, haben wir sie alle gleich aufgebaut. Vor allem der erste Erfolgsfall $Msgrcv1$ ließe sich wesentlich einfacher darstellen, allerdings wäre ein Vergleich mit den anderen Erfolgsfällen nicht mehr so einfach. Zudem muß man nur einmal den Aufbau verstehen, und schon hat man die vier Erfolgsfälle verstanden.

Die Vorbedingungen gleichen sich auch teilweise. Die mit der ID angegebene Message-Queue muß existieren, diese darf nicht leer sein, sondern muß mindestens eine Nachricht enthalten, und der Prozeß muß für diese Message-Queue Leserechte besitzen.

Die Schemata unterscheiden sich bei dem vorgegebenen Typ für die Nachricht und den zu setzenden Flags.

Auch der Zustandsraum unterliegt bei allen Schemata denselben Veränderungen. Dieses ähnelt zudem sehr $msgsnd()$, nur wird diesmal eine Nachricht aus der Message-Queue gelöscht.

Bei $msgq$ wird in der durch die ID vorgegebenen Message-Queue die durch den Typ und die Flags angegebene Nachricht gelöscht und die verbleibenden Nachrichten in der Message-Queue in die richtige Reihenfolge gebracht.

Um die Länge dieser Nachricht muß auch die Länge der Message-Queue in $cbytes$ geändert werden.

Da nun eine Nachricht weniger in der Message-Queue existiert muß auch die Länge dieser Nachricht in msg_ts gelöscht werden.

Alle anderen Objekte des Zustandraumes bleiben unverändert.

Doch nun zu den vier Erfolgsfällen, wobei wir versuchen werden die Unterschiede deutlich darzustellen. Beim ersten Erfolgsfall wollen wir dann auch probieren unsere Spezifikation noch etwas näher zu erläutern.

Das erfolgreiche Lesen der ersten Nachricht in einer Message-Queue.

Msgrcv1

$$
\begin{array}{ll}
\Delta MsgQStateSpace & \\
p? : PID & \\
i? : ID & \\
t? : TYPE & \\
msgsz? : \mathbb{N} & \\
f? : \mathbb{P}\, FLAG & \\
t! : TYPE & \\
b! : BYTES & \\
r! : RETURN & \\
\hline
i? \in \operatorname{dom} msgq & \\
t? = 0 & (1) \\
getuid(p?) \text{ mayread } (perm(i?), owner(i?), group(i?)) & \\
msgq(i?) \neq \langle\rangle \land & \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \bullet (t1, b1)\} \bullet & (2) \\
\quad msgq(i?) \upharpoonright j \neq \langle\rangle \land & (3) \\
\quad (MSG\_NOERROR \in f? \lor & (4) \\
\qquad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) \leq msgsz?) \land & (5) \\
\quad (t!, b!) = (\mu\, t2 : TYPE;\ b2 : BYTES \mid (t2, b2) = (msgq(i?) \upharpoonright j)(1) \bullet & \\
\qquad\quad (t2, truncate(b2, msgsz?))) \land & (6) \\
\quad r! = min\{msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))), msgsz?\} \land & (7) \\
& \\
\quad getkey' = getkey \land & \\
\quad msgq' = msgq \oplus \{i? \mapsto & \\
\qquad squash(\{min(\operatorname{dom}(msgq(i?) \rhd j))\} \lhd msgq(i?))\} \land & \\
\quad creator' = creator \land & \\
\quad owner' = owner \land & \\
\quad group' = group \land & \\
\quad perm' = perm \land & \\
\quad qbytes' = qbytes \land & \\
\quad cbytes' = cbytes \oplus & \\
\qquad \{i? \mapsto cbytes(i?) - msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j)))\} \land & \\
\quad msg\_ts' = msg\_ts \oplus & \\
\qquad \{i? \mapsto squash(\{min(\operatorname{dom}(msgq(i?) \rhd j))\} \lhd msg\_ts(i?))\} \land & \\
\quad errno' = errno) &
\end{array}
$$

Die Gemeinsamkeiten mit den anderen Schemata sind oben schon beschrieben worden. Nun wenden wir uns den Unterschieden zu.

Um das Lesen der ersten Nachricht einer Message-Queue zu erreichen wird der Typ 0 vorgegeben (1).

Hierbei ist es nun unwichtig, ob das Flag MSG_EXCEPT gesetzt ist oder nicht. Daher braucht es in diesem Schema nicht aufgeführt zu werden.

Sodann werden zwei lokale Variablen deklariert, die der zu lesenden Nachricht und ihrem Typ entsprechen (2). Diese Variablen müssen einigen Anforderungen genügen.

Für die Variable des Typen kann es vergleichende Bedingungen mit dem vorgegebenen Typen $t?$ geben. Dies ist in diesem Fall nicht gegeben.

Die Message-Queue muß eine Nachricht dieses Typs besitzen (3).

Sodann muß entweder das Flag $MSG\_NOERROR$ gesetzt sein (4) oder die Länge der zu lesenden Nachricht muß kleiner oder gleich der Länge sein, die als maximale Länge für die Nachricht angegeben wurde (5).

Da immer noch mehrere Nachrichten vielleicht sogar unterschiedlichen Typs (hier sieht man die Verallgemeinerung, da man in diesem speziellen Fall diesen Teil vereinfachen könnte) diesen Anforderungen gerecht werden können, muß für die Ausgabe eine spezielle Nachricht aus dieser Menge genommen werden (6).

Da es eine geordnete Menge ist, können wir die erste entsprechende Nachricht nehmen und deren Typ zurückgeben sowie die Bytes, wobei eventuell nur die ersten Bytes bis zu der gewünschten Länge der Nachricht genommen werden (6).

Als Returnwert erhält man die Länge der gelesenen Nachricht, also die Anzahl der gelesenen Bytes. Hierfür müssen in der Spezifikation einige Verrenkungen vorgenommen werden (7).

$msgq(i?)$ entspricht einer Sequenz von Tupeln aus Typen und Bytes. $msgq(i?) \rhd j)$ ergibt eine Sequenz, bei der nur noch Typen und Bytes, die den lokalen Variablen entsprechen, vorhanden sind. $\mathrm{dom}(msgq(i?) \rhd j)$ ergibt den Domain dieser Sequenz, wobei eine Sequenz eine Abbildung der natürlichen Zahlen in eine Menge entspricht. Mit $min(\mathrm{dom}(msgq(i?) \rhd j))$ wird das kleinste Element dieser Menge gefunden. Diese Zahl entspricht der Stellen, an der die zu lesende Nachricht in der Message-Queue positioniert ist.

Im vorliegendem Falle ist dies immer die erste Stelle, man hätte es sich also wesentlich einfacher machen können.

Für den Returnwert erhält man also die Länge der Nachricht, die an dieser Stelle in der Message-Queue eingetragen ist oder die vorgegebene maximale Länge einer Nachricht, je nachdem, welcher Wert kleiner ist.

Für die Objekte des Zustandraumes arbeitet man ebenfalls mit diesem Wert.

Bei $msgq$ wird die Nachricht, die an dieser Stelle steht, per Domain Substraktion gelöscht und die Message-Queue mit $squash$ neu durchnummeriert.

Bei $cbytes$ wird die Länge der Nachricht, die an der ausgerechneten Stelle sich befindet, von der bisherigen Gesamtlänge abgezogen.

Und bei msg_ts wird die Länge, die an der ausgerechneten Stelle verzeichnet ist, ebenfalls per Domain-Substraktion gelöscht und die Sequenz mit $squash$ neu geordnet.

Bei diesem Fall geht es um das erfolgreiche Lesen der ersten Nachricht, die einen bestimmten Typen besitzt.

Msgrcv2

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN
\end{array}
$$

$$
\begin{array}{ll}
i? \in \mathrm{dom}(msgq) & \\
t? > 0 & (1) \\
MSG\_EXCEPT \notin f? & (2) \\
getuid(p?)\ \mathrm{mayread}\ (perm(i?), owner(i?), group(i?)) & \\
msgq(i?) \neq \langle\rangle \land & \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 = t? \bullet & (3) \\
\quad (t1, b1)\} \bullet & \\
\qquad msgq(i?) \upharpoonright j \neq \langle\rangle \land & \\
\qquad (MSG\_NOERROR \in f? \lor & \\
\qquad\quad msg\_ts(i?)(min(\mathrm{dom}(msgq(i?) \rhd j))) \leq msgsz?) \land & \\
\qquad (t!, b!) = (\mu\, t2 : TYPE;\ b2 : BYTES \mid (t2, b2) = (msgq(i?) \upharpoonright j)(1) \bullet & \\
\qquad\quad (t2, truncate(b2, msgsz?))) \land & (4) \\
\qquad r! = min\{msg\_ts(i?)(min(\mathrm{dom}(msgq(i?) \rhd j))), msgsz?\} \land & \\
\\
\qquad getkey' = getkey \land & \\
\qquad msgq' = msgq \oplus \{i? \mapsto & \\
\qquad\quad squash(\{min(\mathrm{dom}(msgq(i?) \rhd j))\} \ntriangleleft msgq(i?))\} \land & \\
\qquad creator' = creator \land & \\
\qquad owner' = owner \land & \\
\qquad group' = group \land & \\
\qquad perm' = perm \land & \\
\qquad qbytes' = qbytes \land & \\
\qquad cbytes' = cbytes \oplus \{i? \mapsto cbytes(i?) - & \\
\qquad\quad msg\_ts(i?)(min(\mathrm{dom}(msgq(i?) \rhd j)))\} \land & \\
\qquad msg\_ts' = msg\_ts \oplus \{i? \mapsto & \\
\qquad\quad squash(\{min(\mathrm{dom}(msgq(i?) \rhd j))\} \ntriangleleft msg\_ts(i?))\} \land & \\
\qquad errno' = errno) &
\end{array}
$$

In diesem Schema ist der vorgegebene Typ größer als 0 (1).

Gleichzeitig darf das Flag MSG_EXCEPT nicht gesetzt (2).

Und als Bedingung für die lokalen Variablen ergibt sich, daß der Typ gleich dem vorgegebenen Typ sein muß (3).

Alle restlichen Gegebenheiten entsprechen den schon beschriebenen allgemeinen Bedingungen für die Erfolgsfälle und den Beschreibungen der Notation.

Der Unterschied bei $Msgrcv2$ zu $Msgrcv1$ besteht also darin, daß hier ein konkreter Typ für die zu lesende Nachricht vorgegeben wird. Daraufhin wird auch die Menge der lokalen Variablen, die der Menge aller möglichen passenden Nachrichten entspricht, eingeschränkt (4).

Nun kann es auch noch zum erfolgreichen Lesen der ersten Nachricht einer Message-Queue kommen, die nicht einem bestimmten Typ entspricht.

$$
\begin{array}{l}
\textit{Msgrcv3} \\
\hline
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
t? > 0 \hfill (1) \\
\textit{MSG\_EXCEPT} \in f? \hfill (2) \\
\textit{getuid}(p?)\ \text{mayread}\ (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?)) \\
\textit{msgq}(i?) \neq \langle\rangle \land \\
(\textbf{let}\ j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \mid t1 \neq t? \bullet \hfill (3) \\
\quad (t1, b1)\} \bullet \\
\quad\quad \textit{msgq}(i?) \restriction j \neq \langle\rangle \land \\
\quad\quad (\textit{MSG\_NOERROR} \in f? \lor \\
\quad\quad\quad \textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))) \leq \textit{msgsz}?) \land \\
\quad\quad (t!, b!) = (\mu\, t2 : \textit{TYPE};\ b2 : \textit{BYTES} \mid (t2, b2) = (\textit{msgq}(i?) \restriction j)(1) \bullet \\
\quad\quad\quad (t2, \textit{truncate}(b2, \textit{msgsz}?))) \land \\
\quad\quad r! = \textit{min}\{\textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))), \textit{msgsz}?\} \land \\
\\
\quad\quad \textit{getkey}' = \textit{getkey} \land \\
\quad\quad \textit{msgq}' = \textit{msgq} \oplus \{i? \mapsto \\
\quad\quad\quad \textit{squash}(\{\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))\} \ntriangleleft \textit{msgq}(i?))\} \land \\
\quad\quad \textit{creator}' = \textit{creator} \land \\
\quad\quad \textit{owner}' = \textit{owner} \land \\
\quad\quad \textit{group}' = \textit{group} \land \\
\quad\quad \textit{perm}' = \textit{perm} \land \\
\quad\quad \textit{qbytes}' = \textit{qbytes} \land \\
\quad\quad \textit{cbytes}' = \textit{cbytes} \oplus \{i? \mapsto \textit{cbytes}(i?) - \\
\quad\quad\quad \textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j)))\} \land \\
\quad\quad \textit{msg\_ts}' = \textit{msg\_ts} \oplus \{i? \mapsto \\
\quad\quad\quad \textit{squash}(\{\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))\} \ntriangleleft \textit{msg\_ts}(i?))\} \land \\
\quad\quad \textit{errno}' = \textit{errno})
\end{array}
$$

Auch in diesem Fall ist der vorgegebene Typ größer als 0 (1).

Allerdings muß das Flag MSG_EXCEPT unbedingt gesetzt sein (2).

Die zusätzliche Bedingung für die lokale Variable des Types ist diesmal, daß sie ungleich dem vorgegebenen Typ zu sein hat (3).

Der Unterschied zu $Msgrcv2$ besteht also darin, daß durch Setzen des Flags MSG_EXCEPT der Typ der Nachricht ungleich dem vorgegebenen Typ sein muß.

Beim letzten Erfolgsfall wird die erste Nachricht des kleinsten Typen aus einer Message-Queue gelesen, die kleiner oder gleich einem vorgegebenen Typ ist.

$$
\begin{array}{l}
\textit{Msgrcv4} \\
\hline
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
t? < 0 \hfill (1) \\
\textit{getuid}(p?) \text{ mayread } (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?)) \\
\textit{msgq}(i?) \neq \langle\rangle \land \\
(\textbf{let } j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \mid t1 \leq -t? \land \hfill (2) \\
\quad t1 = \textit{min}(\{t2 : \textit{TYPE};\ b2 : \textit{BYTES} \mid (t2, b2) \in \operatorname{ran}(\textit{msgq}(i?)) \bullet t2\}) \bullet \\
\qquad (t1, b1)\} \bullet \hfill (3) \\
\qquad\quad \textit{msgq}(i?) \upharpoonright j \neq \langle\rangle \land \\
\qquad (\textit{MSG\_NOERROR} \in f? \lor \\
\qquad\quad \textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))) \leq \textit{msgsz}?) \land \\
\qquad (t!, b!) = (\mu\, t3 : \textit{TYPE};\ b3 : \textit{BYTES} \mid (t3, b3) = (\textit{msgq}(i?) \upharpoonright j)(1) \bullet \\
\qquad\quad (t3, \textit{truncate}(b3, \textit{msgsz}?))) \land \\
\qquad\quad r! = \textit{min}\{\textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))), \textit{msgsz}?\} \land \\
\\
\qquad\quad \textit{getkey}' = \textit{getkey} \land \\
\qquad\quad \textit{msgq}' = \textit{msgq} \oplus \{i? \mapsto \\
\qquad\qquad \textit{squash}(\{\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))\} \ndres \textit{msgq}(i?))\} \land \\
\qquad\quad \textit{creator}' = \textit{creator} \land \\
\qquad\quad \textit{owner}' = \textit{owner} \land \\
\qquad\quad \textit{group}' = \textit{group} \land \\
\qquad\quad \textit{perm}' = \textit{perm} \land \\
\qquad\quad \textit{qbytes}' = \textit{qbytes} \land \\
\qquad\quad \textit{cbytes}' = \textit{cbytes} \oplus \{i? \mapsto \textit{cbytes}(i?) - \\
\qquad\qquad \textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j)))\} \land \\
\qquad\quad \textit{msg\_ts}' = \textit{msg\_ts} \oplus \{i? \mapsto \\
\qquad\qquad \textit{squash}(\{\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))\} \ndres \textit{msg\_ts}(i?))\} \land \\
\qquad\quad \textit{errno}' = \textit{errno})
\end{array}
$$

Dieses mal wird ein Typ erwartet, der kleiner als 0 ist, also ein negativer Wert (1).

Als Bedingung für die lokale Variable ergibt sich diesmal, daß der Typ kleiner oder gleich dem negativem des vorgegebenen Typen ist. Da der vorgegebene Typ ein negativer Wert ist, muß der Typ also kleiner oder gleich dem Absolutwert des vorgegebenen Typen sein (2).

Und nun ergibt sich in diesem speziellen Fall doch noch einmal eine Abweichung vom üblichen Aufbau der Erfolgsschemata. Da nun mehrere verschiedene Typen möglich sind müssen zuerst die Nachrichten mit dem kleinsten Typen genommen werden. Dies ist also eine zusätzliche Bedingung an die lokale Variable (3).

Durch die Vorgabe eines negativen Wertes für den Typ unterscheidet sich dieser Fall von den restlichen drei Erfolgsfällen. Diesmal muß zuerst noch der Typ gefunden werden, bevor man die erste Nachricht aus der Menge betrachten kann.

Als abschließende Bemerkung wollen wir hier noch einmal erwähnen, daß die ähnliche Darstellung der verschiedenen Fälle für die Verständlichkeit und Vergleichbarkeit wohl doch positiv ist. Im ersten Moment erscheinen die Schemata sehr kompliziert und ähnlich, unterscheiden tun sie sich tatsächlich nur in drei Punkten. Doch hierdurch werden diese Unterschiede wesentlich besser erfaßt.

O.7.2 Fehlerfall

Auch bei dieser Message-Queue Operation ist es wieder so, daß es einige Fehlerfälle gibt, die nicht in unserer Spezifikation enthalten sind.

Diese sind in den vorherigen Kapiteln schon erklärt worden, seien hier aber noch einmal aufgeführt.

- EFAULT: ein Pointer zeigt auf eine nicht exitstierende Adresse. Für die Behandlung ist der Kernel zuständig.
- EIDRM: Die Message-Queue ist gelöscht worden, während der Prozess darauf wartete, aus dieser Message-Queue zu lesen. Dies läßt sich aufgrund der fehlenden zeitlichen Komponente von Z nicht spezifizieren.
- EINTR: Der Prozeß wartet auf eine Message-Queue und erhält einen Interrupt. Dies kann in Z wegen der fehlenden Zeitkomponente nicht dargestellt werden.

In diesem Abschnitt werden die spezifizierbaren Fehlerfälle aufgeführt. Es gibt insgesamt vier solcher Fälle, von denen allerdings zwei von der Vorgabe des Typen abhängig sind. Diese beiden Fehlerfälle müssen noch einmal aufgeteilt werden in dieselben vier Unterscheidungen, die wir beim Erfolgsfall schon getroffen haben. Ansonsten würden diese Schemata zu unübersichtlich werden.

Prinzipiell gemeinsam ist dem Aufbau der Schemata, daß dieselben Ein- und Ausgaben wie bei den Erfolgsfällen vorhanden sind. Ebenso gibt es stets mit -1 den für Fehlerfälle typischen Rückgabewert. Auch bleibt der Zustandsraum unverändert mit Ausnahme von des Objektes $errno$, welches auf den Namen des Fehlers gesetzt wird.

Der erste Fehlerfall ist die Möglichkeit, daß die Länge einer Nachricht größer ist, als die vorgegebene maximale Länge einer Nachricht, und gleichzeitig das Flag $MSG\_NOERROR$ nicht gesetzt ist.

Da dieser Fall von der Nachricht abhängig ist, man diese also erst finden muß, müssen dieselben Fallunterscheidungen wie bei dem Erfolgsfall getroffen werden.

Zuerst also der Fehler $E2BIG$ für den Fall, daß die erste Nachricht einer Message-Queue gelesen werden sollte und diese länger als erlaubt ist.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{msgrcv1E2BIG} \\
\Delta MsgQStateSpace & \\
p? : PID & \\
i? : ID & \\
t? : TYPE & \\
f? : \mathbb{P}\, FLAG & \\
msgsz? : \mathbb{N} & \\
t! : TYPE & \\
b! : BYTES & \\
r! : RETURN & \\
\hline
i? \in \operatorname{dom} msgq & (1) \\
t? = 0 & (2) \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \bullet (t1, b1)\} \bullet & (3) \\
\quad msgq(i?) \upharpoonright j \neq \langle\rangle \land & (4) \\
\quad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) > msgsz?) & (5) \\
MSG\_NOERROR \notin f? & (6) \\
r! = -1 & \\
errno' = E2BIG &
\end{array}
$$

Für die Schemata des Fehlers $E2BIG$ sind einige Vorbedingungen immer gleich. So muß stets eine korrekte Message-Queue adressiert sein (1) und das Flag $MSG\_NOERROR$ ist nicht gesetzt (6). Die Message-Queue hat mindestens eine Nachricht des gewünschten Typens zu enthalten (4) und die erste Nachricht, die dem gewünschten Typen entspricht muß größer sein, als die vorgegebene maximale Länge einer Nachricht (5).

Die Schemata unterscheiden sich auch stets in denselben Punkten. So muß in diesem Fall der vorgegebene Typ 0 sein (2) und es findet für die Menge der möglichen Typen keine Einschränkung statt (3). Dies entspricht den Vorgaben beim Erfolgsfall $Msgrcv1$.

Wird jedoch das Lesen der ersten Nachricht eines bestimmten Typs aus einer Message-Queue gewünscht, und diese Nachricht ist länger, als die vorgegebene maximale Länge einer Nachricht, so tritt das zweite Schema des Fehlers $E2BIG$ ein.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\underline{\; msgrcv2E2BIG \;}} \\
\Delta MsgQStateSpace & \\
p? : PID & \\
i? : ID & \\
t? : TYPE & \\
f? : \mathbb{P}\, FLAG & \\
msgsz? : \mathbb{N} & \\
t! : TYPE & \\
b! : BYTES & \\
r! : RETURN & \\
\hline
i? \in \operatorname{dom} msgq & \\
t? > 0 & (1) \\
MSG\_EXCEPT \notin f? & (2) \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 = t? \bullet (t1, b1)\} \bullet & (3) \\
\quad msgq(i?) \upharpoonright j \neq \langle\rangle \land & \\
\quad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) > msgsz?) & \\
MSG\_NOERROR \notin f? & \\
r! = -1 & \\
 & \\
errno' = E2BIG & \\
\hline
\end{array}
$$

Bei diesem Schema entsprechen die Vorgaben dem Erfolgsfall $Msgrcv2$. Es wird ein Typ größer 0 verlangt (1) und gleichzeitig darf das Flag MSG_EXCEPT nicht gesetzt sein (2). Die Menge der passenden Nachrichten wird durch die Einschränkung auf den vorgegebenen Typen definiert (3).

Tritt der Fehler einer zu langen Nachricht auf, wenn die erste Nachricht, die nicht einem bestimmten Typ entspricht, ausgelesen werden soll, so ist dieses Schema relevant.

$$
\begin{array}{|l}
\hline
\textit{msgrcv3E2BIG} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
t? > 0 \hfill (1) \\
MSG\_EXCEPT \in f? \hfill (2) \\
(\textbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 \neq t? \bullet (t1, b1)\} \bullet \hfill (3) \\
\quad msgq(i?) \upharpoonright j \neq \langle\rangle \land \\
\quad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) > msgsz?) \\
MSG\_NOERROR \notin f? \\
r! = -1 \\
errno' = E2BIG \\
\hline
\end{array}
$$

Entsprechend dem Erfolgsfall $Msgrcv3$ wird hier ein Typ vorausgesetzt, der größer als 0 ist (1). Dabei muß das Flag MSG_EXCEPT gesetzt sein (2), was der Unterschied zum vorherigen Fall darstellt. Demzufolge wird die Menge der passenden Nachrichten dadurch eingeschränkt, daß sie nicht dem vorgegebenen Typen entsprechen (3).

Ist die Nachricht nun länger als erlaubt bei den Vorgaben, die $Msgrcv4$ entsprechen, so tritt dieser Fall ein.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{msgrcv4E2BIG}} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
t? < 0 \hfill (1) \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 \leq -t? \land \hfill (2) \\
\quad t1 = min(\{t2 : TYPE;\ b2 : BYTES \mid \\
\qquad (t2, b2) \in \operatorname{ran}(msgq(i?)) \bullet t2\}) \bullet \hfill (3) \\
\qquad\quad (t1, b1)\} \bullet \\
\qquad\qquad msgq(i?) \upharpoonright j \neq \langle\rangle \land \\
\qquad\qquad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) > msgsz?) \\
MSG\_NOERROR \notin f? \\
r! = -1 \\
errno' = E2BIG \\
\hline
\end{array}
$$

In diesem Fall wird also ein Typ vorausgesetzt, der kleiner als 0 zu sein hat (1). Die Menge der passenden Nachrichten wird diesmal dadurch eingeschränkt, daß der Typ kleiner als der Absolutbetrag des vorgegebenen Typen zu sein hat (2).

Auch bei diesem Schema wird ebenso wie bei $Msgrcv4$ die Menge der passenden Nachrichten noch weiter eingeschränkt, indem sie auf die Nachrichten mit dem kleinsten Typen beschränkt wird (3).

Der zweite Fehlerfall ist das erfolglose Lesen einer Nachricht, da der Prozeß keine Leserechte für die vorgegebene Message-Queue besitzt.

Hierbei ist der vorgegebene Typ der Nachricht nicht weiter von Bedeutung, also muß auch keine weitere Fallunterscheidung gemäß den Erfolgsfällen getroffen werden.

$msgrcvEACCES$

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \qquad (1) \\
\neg\ (getuid(p?)\ \text{mayread}\ (perm(i?), owner(i?), group(i?))) \qquad (2) \\
r! = -1 \\
errno' = EACCES
\end{array}
$$

Auch bei diesem Fall ist stets Voraussetzung, daß eine korrekte Message-Queue angegeben wurde (1). Daraufhin darf der Prozeß jedoch keine Leserechte für die Message-Queue besitzen (2).

Auch der dritte Fehlerfall ist von den vorhergehenden Message-Queue Operationen her bekannt. Er tritt ein, wenn ein falscher Wert vorgegeben wurde, sei es für die ID oder für die Länge einer Nachricht.

$$
\begin{array}{|l}
\hline
\textit{msgrcvEINVAL} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \notin \operatorname{dom} \textit{msgq} \lor \qquad (1) \\
\quad \textit{msgsz}? < 0 \qquad (2) \\
r! = -1 \\
\textit{errno}' = \textit{EINVAL} \\
\hline
\end{array}
$$

Die Vorbedingungen sind diesmal ganz einfach. Entweder ist ein Message-Queue Identifier angegeben worden, der nicht vorhanden ist (1), oder es wurde als maximale Nachrichtenlänge ein negativer Wert angegeben (2).

Der letzte spezifizierte Fehlerfall ist wieder von der zu lesenden Nachricht abhängig, dementsprechend müssen wieder dieselben Fallunterscheidung wie beim Erfolgsfall getroffen werden.

Auch dieser Fehler ist von den anderen Message-Queue Operationen her bekannt. $NOMSG$ kennzeichnet, daß keine Nachricht des vorgegebenen Typen in der angegebenen Message-Queue vorhanden ist.

Fangen wir mit dem ersten Fall an, bei dem die erste Nachricht aus dem Schema ausgelesen werden sollte. Damit dieser Fehler eintritt muß die Message-Queue in diesem Falle leer sein.

$$
\begin{array}{l}
\underline{\quad msgrcv1NOMSG \quad} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \hfill (1) \\
t? = 0 \hfill (2) \\
IPC\_NOWAIT \in f? \land \hfill (3) \\
\quad (msgq(i?) = \langle\rangle \lor \hfill (4) \\
\quad (\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \bullet (t1, b1)\} \bullet \hfill (5) \\
\qquad msgq(i?) \restriction j = \langle\rangle)) \hfill (6) \\
r! = -1 \\
\\
errno' = ENOMSG
\end{array}
$$

Bei diesem Fehler gibt es wieder einige Vorbedingungen, die bei allen Fällen vorzuliegen haben. Dies wäre zuerst die Angabe einer korrekten ID (1). Dann muß das Flag IPC_NOWAIT gesetzt sein (2), wäre es nicht gesetzt träte übriegens der nicht spezifizierte Fall des Wartens ein. Die Message-Queue ist entweder leer (4) oder sie darf keine Nachricht des vorgegebenen Typens enthalten (5 und 6).

Zusätzlich gibt es für die Fallunterscheidung wieder die üblichen unterschiedlichen Vorbedingungen. In diesem Fall muß als Typ die 0 vorgegeben sein, eine weiter Einschränkung der Menge der passenden Nachrichten kann daraufhin entfallen (2).

Tritt nun dieser Fehler ein, wenn man die erste Nachricht eines bestimmten Typen lesen möchte, ist dieses Schema entscheidene.

$$
\begin{array}{|l}
\multicolumn{1}{l}{msgrcv2NOMSG} \\
\hline
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \\
t? > 0 \qquad (1) \\
MSG\_EXCEPT \notin f? \qquad (2) \\
IPC\_NOWAIT \in f? \land \\
\quad (msgq(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 = t? \bullet (t1, b1)\} \bullet \qquad (3) \\
\qquad msgq(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
\\
errno' = ENOMSG \\
\hline
\end{array}
$$

Die unterschiedlichen Vorbedingungen gegenüber den anderen Schemate dieses Fehlers entsprechen wieder dem Erfolgsfall $Msgrcv2$, wobei ein Typ vorgegeben wird, der größer als 0 sein muß (1) und gleichzeitig darf das Flag MSG_EXCEPT nicht gesetzt sein (2). Daraufhin wird die Menge der passenden Nachrichten durch das Testen auf Gleichheit mit dem vorgegebenen Typ beschränkt (3).

Der dritten Fall, entsprechend $Msgrcv3$, ist der Fall, wenn die erste Nachricht, die nicht einem bestimmten Typ entspricht, gelesen werden soll, und keine entsprechende Nachricht existiert.

$$
\begin{array}{|l}
\textit{msgrcv3NOMSG} \\
\hline
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \\
t? > 0 \qquad (1) \\
MSG\_EXCEPT \in f? \qquad (2) \\
IPC\_NOWAIT \in f? \land \\
\quad (msgq(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 \neq t? \bullet (t1, b1)\} \bullet \qquad (3) \\
\qquad msgq(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
errno' = ENOMSG \\
\hline
\end{array}
$$

Der Unterschied zum vorherigen Fall ist wieder, daß ein Typ größer 0 vorausgesetzt wird (1) und das Flag MSG_EXCEPT gesetzt sein muß (2). Entsprechend wird die Menge der passenden Nachrichten durch das Testen auf Ungleichheit beschränkt (3).

Zum Schluß noch der Versuch eine Nachricht entsprechend dem Fall $Msgrcv4$ zu lesen, wobei keine solche Nachricht vorhanden ist.

$$
\begin{array}{|l}
\hline
\textit{msgrcv4NOMSG} \\
\Delta \mathit{MsgQStateSpace} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
t? : \mathit{TYPE} \\
f? : \mathbb{P}\,\mathit{FLAG} \\
\mathit{msgsz}? : \mathbb{N} \\
t! : \mathit{TYPE} \\
b! : \mathit{BYTES} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} \mathit{msgq} \\
t? < 0 \qquad (1) \\
\mathit{IPC\_NOWAIT} \in f? \land \\
\quad (\mathit{msgq}(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : \mathit{BYTES};\ t1 : \mathit{TYPE} \mid t1 \leq -t? \land \qquad (2) \\
\qquad t1 = \mathit{min}(\{t2 : \mathit{TYPE};\ b2 : \mathit{BYTES} \mid \\
\qquad\quad (t2, b2) \in \operatorname{ran}(\mathit{msgq}(i?)) \bullet t2\}) \bullet \qquad (3) \\
\qquad\qquad (t1, b1)\} \bullet \\
\qquad\qquad\quad \mathit{msgq}(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
\mathit{errno}' = \mathit{ENOMSG} \\
\hline
\end{array}
$$

Im Gegensatz zu den vorherigen Fällen wird hier ein Typ vorausgesetzt, der kleiner als 0 ist (1). Die Menge der passenden Nachrichten wird zuerst auf die eingeschränkt, die kleiner als der Absolutbetrag des vorgegebenen Typens sind (2) und anschließend auf die Nachrichten mit dem kleinsten Typen dieser Menge (3).

O.7.3 User-Interface

Die von uns vorgenommene Aufteilung in verschiedene Fälle ist "willkürlich" und diente wie schon Beschrieben der besseren Lesbarkeit und Verständlichkeit und der Nähe zum Source-Code. Nun müssen wir diese Fälle wieder zusammenführen, um das Verhalten der Message-Queue Operation $msgrcv()$ zu beschreiben.

Fangen wir mit dem Erfolgsfall an und führen zuerst diese Fälle zusammen. Die Gesamtwirkung der Operation $Msgrcv$, die sämtliche Erfolgsfälle beinhaltet, ergibt sich aus folgender Schemadisjunktion:

$$Msgrcv \mathrel{\widehat{=}} Msgrcv1 \lor Msgrcv2 \lor Msgrcv3 \lor Msgrcv4$$

Sodann ist es sinnvoll alle Fälle zusammenzuführen, die nicht das vom Benutzer gewünschte Ergebnis darstellen. Dieses ergibt sich aus folgender Schemadisjunktion:

$$\begin{aligned} Msgrcvfault \mathrel{\widehat{=}} & msgrcv1E2BIG \lor msgrcv2E2BIG \lor msgrcv3E2BIG \\ & \lor msgrcv4E2BIG \lor msgrcvEACCES \lor msgrcvEINVAL \\ & \lor msgrcv1NOMSG \lor msgrcv2NOMSG \lor msgrcv3NOMSG \\ & \lor msgrcv4NOMSG \end{aligned}$$

Da in $msgrcv()$ sowohl der Erfolgsfall als auch der Fehlerfall enthalten ist, müssen wir auch diese beiden Fälle zusammenfügen. Auch hier ist wieder eine Schemadisjunktion möglich, da sämtliche unterschiedlichen Schemata dieselben Ein- und Ausgaben besitzen.

$$msgrcv \mathrel{\widehat{=}} Msgrcv \lor Msgrcvfault$$

Somit entspricht das abstrakt formal spezifizierte $msgrcv$ dem nach Aussen sichtbaren Verhalten der Message-Queue Operation $msgrcv()$.

O.8 Message-Queue Operation $msgctl()$

Mit der Message-Queue Operation $msgctl()$ können die Daten über eine Message-Queue manipuliert werden.

Auch bei dieser Operation haben wir uns entschlossen, zur besseren Lesbarkeit und zur einfacheren Verständlichkeit diese Operation in den Erfolgsfall und den Fehlerfall aufzuteilen, und diese jeweils noch weiter zu unterteilen.

Im Erfolgsfall ist es zum einen denkbar eine Message-Queue zu löschen. Desweiteren können Informationen über die Message-Queue ausgegeben werden oder diese Informationen können sogar verändert werden.

Auch die Fehlerfälle haben wir weiter unterteilt. Dabei entstanden drei Fehlerfälle, die aus den anderen Operationen schon bekannt sein dürften.

Zuerst $EACCESS$, der Prozeß hat keine Leserechte für die angegebenen Message-Queue. Weiter geht es mit $EINVAL$, es gab eine falsche Angabe für ein Kommando oder eine falsche Angabe beim Message-Queue Identifier. Und als drittes gibt es noch $EPERM$, der Prozeß hat nicht die entsprechenden Rechte zur Ausführung der Aktion.

Desweiteren gibt es wieder zwei Fehlerfälle, die wir nicht mitspezifiziert haben. Auch diese Fälle sind aus vorherigen Kapiteln schon bekannt und die Begründung für das Nichtspezifizieren ist dort schon mehrfach genannt.

- EFAULT: ein Pointer zeigt auf eine nicht exitstierende Adresse. Für dies Behandlung ist der Kernel zuständig.
- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden, und im selben „Slot" $(id\%MSGMNI)$ gibt es wieder eine neue Messagequeue. Dies ist allerdings so nah an der Implementierung, daß wir das in der Spezifikation nicht von EINVAL unterscheiden, insbesondere, da dieser Fehlerfall nicht ausreichend dokumentiert ist. Die Manual–Page von $msgctl$ sagt dazu: „*SVID dies not document the EIDRM error condition.*".

Nachdem wir wieder so viele Fallunterscheidungen getroffen haben ist es anschließend notwendig die verschiedenen Fälle zusammenzuführen um das Verhalten von $msgctl()$ darzustellen.

Dieses Kapitel beginnt also wieder mit den Erfolgsfällen, die diesmal nicht sehr viele Gemeinsamkeiten aufweisen, woraufhin die Beschreibungen etwas länger werden. Weiter geht es mit den Fehlerfällen, bei denen wir den letzten Fehlerfall der Übersichtlichkeit halber in zwei Schemata aufgeteilt haben. Abschließend wird das User-Interface erstellt.

Insgesamt erhalten wir in diesem Kapitel eine abstrakte formale Spezifikation der Message-Queue Operation $msgctl()$.

O.8.1 Erfolgsfall

Um eine spätere Schemadisjunktion zu ermöglichen haben wieder alle Erfolgsfälle dieselben Eingaben.

Zuerst wird wie üblich der Zustandsraum eingebunden, der Veränderungen unterliegen kann. Die PID wird bei der Rechtevergabe benötigt, die ID um die Message-Queue zu kennzeichnen.

Mit dem CMD wird ein Kommando vorgegeben, das angibt, welcher der drei Erfolgsfälle eintritt.

Desweiteren können die Informationen über die Message-Queue entweder als Eingaben oder als Ausgaben benutzt werden. Hierdurch werden die Schemata auch so lang, aber nicht kompliziert.

Voraussetzung für den Erfolgsfall ist stets, daß mit der ID eine existierende Message-Queue angegeben ist.

Zurückgegeben wird stets ein Returnwert, der bei dieser Operation im Erfolgsfalle stets 0 ist.

Die Vor- und Nachbedingungen und die Veränderungen des Zustandraumes sind derart unterschiedlich, daß diese erst bei den speziellen Schemata erklärt werden.

Der erste Fall ist das erfolgreiche Löschen einer Message-Queue, entsprechende Rechte werden natürlich vorausgesetzt.

Die Folge dieses Schemas ist demnach, das eine Message-Queue mit sämtlichen noch enthaltenen Nachrichten sowie sämtliche Informationen über diese Message-Queue, wie Zugriffsrechte oder Angabe des Erzeugers oder Besitzers, gelöscht werden.

$Msgctl1$

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
cmd? : CMD \\
msg\_uid? : UID \\
msg\_gid? : GID \\
msg\_perm? : \mathbb{P}\, PERM \\
msg\_qbytes? : \mathbb{N} \\
msg\_uid! : UID \\
msg\_gid! : GID \\
msg\_perm! : \mathbb{P}\, PERM \\
msg\_qbytes! : \mathbb{N} \\
r! : RETURN
\end{array}
$$

$$
\begin{array}{ll}
i? \in \operatorname{dom} msgq & \\
cmd? = IPC\_RMID & (1) \\
getuid(p?) = SUPERUSER \lor & (2) \\
\qquad getuid(p?) = owner(i?) \lor & (3) \\
\qquad getuid(p?) = creator(i?) & (4) \\
r! = 0 & \\
 & \\
getkey' = getkey \setminus \{i? \mapsto getkey(i?)\} \land & \\
msgq' = msgq \setminus \{i? \mapsto msgq(i?)\} \land & \\
creator' = creator \setminus \{i? \mapsto creator(i?)\} \land & \\
owner' = owner \setminus \{i? \mapsto owner(i?)\} \land & \\
group' = group \setminus \{i? \mapsto group(i?)\} \land & \\
perm' = perm \setminus \{i? \mapsto perm(i?)\} \land & \\
qbytes' = qbytes \setminus \{i? \mapsto qbytes(i?)\} \land & \\
cbytes' = cbytes \setminus \{i? \mapsto cbytes(i?)\} \land & \\
msg\_ts' = msg\_ts \setminus \{i? \mapsto msg\_ts(i?)\} \land & \\
errno' = errno &
\end{array}
$$

Um auf diesen Fall zu kommen, muß das Kommando IPC_RMID angegeben sein (1).

Zudem muß der Prozeß entweder Superuser sein (2), oder der Eigentümer der Message-Queue (3) oder aber der Erzeuger der Message-Queue(4).

Daraufhin müssen bei sämtlichen Objekten des Zustandraumes, außer bei $errno$, die Abbildungen mit der ID als Ursprungswert gelöscht werden. Da die ID systemweit einmalig vergeben ist, wird nur die gewünschte Message-Queue gelöscht.

Beim zweiten Fall geht es um die erfolgreiche Ausgabe von Informationen der Message-Queues.

Falls man ein Leserecht für die Message-Queue besitzt kann man sich den Besitzer, die Gruppe, die Zugriffsrechte sowie die maximale Länge der Message-Queue anzeigen lassen.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\mathit{Msgctl2}} \\
\hline
\Xi \mathit{MsgQStateSpace} & (1) \\
p? : \mathit{PID} & \\
i? : \mathit{ID} & \\
cmd? : \mathit{CMD} & \\
msg\_uid? : \mathit{UID} & \\
msg\_gid? : \mathit{GID} & \\
msg\_perm? : \mathbb{P}\,\mathit{PERM} & \\
msg\_qbytes? : \mathbb{N} & \\
msg\_uid! : \mathit{UID} & \\
msg\_gid! : \mathit{GID} & \\
msg\_perm! : \mathbb{P}\,\mathit{PERM} & \\
msg\_qbytes! : \mathbb{N} & \\
r! : \mathit{RETURN} & \\
\hline
i? \in \operatorname{dom} msgq & \\
cmd? = \mathit{IPC\_STAT} & (2) \\
getuid(p?)\ \mathrm{mayread}\ (perm(i?), owner(i?), group(i?)) & (3) \\
r! = 0 & \\
 & \\
msg\_uid! = owner(i?) & \\
msg\_gid! = group(i?) & \\
msg\_perm! = perm(i?) & \\
msg\_qbytes! = qbytes(i?) & \\
\hline
\end{array}
$$

Für diesen Fall ist das Kommando $\mathit{IPC\_STAT}$ notwendig (2).

Genauso wird vorausgesetzt, daß der Prozeß Leserecht für die Message-Queue besitzt (3).

Bei diesem Schema ist es nicht möglich, und auch nicht nötig, die Objekte des Zustandraumes zu verändern, schließlich sollen nur Informationen ausgegeben werden. Somit wird der Zustandsraum entsprechend eingebunden (1) und die Objekte brauchen an dieser Stelle nicht aufgeführt zu werden.

Dieser Fall unterscheidet sich von $\mathit{Msgctl1}$ dadurch, daß die Objekte des Zustandraumes nicht verändert werden und statt dessen einige Werte über die Message-Queue zurückgegeben werden.

Der dritte Erfolgsfall ist das Verändern der Informationen über die Message-Queue.

Um solche Veränderungen zu bewirken müssen natürlich entsprechende Rechte vorliegen.

$$
\begin{array}{|l}
\hline
\textit{Msgctl3} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
\textit{cmd}? : \textit{CMD} \\
\textit{msg\_uid}? : \textit{UID} \\
\textit{msg\_gid}? : \textit{GID} \\
\textit{msg\_perm}? : \mathbb{P}\,\textit{PERM} \\
\textit{msg\_qbytes}? : \mathbb{N} \\
\textit{msg\_uid}! : \textit{UID} \\
\textit{msg\_gid}! : \textit{GID} \\
\textit{msg\_perm}! : \mathbb{P}\,\textit{PERM} \\
\textit{msg\_qbytes}! : \mathbb{N} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{cmd}? = \textit{IPC\_SET} \hfill (1) \\
\textit{getuid}(p?) = \textit{SUPERUSER} \lor \hfill (2) \\
\qquad\qquad \textit{getuid}(p?) = \textit{owner}(i?) \lor \hfill (3) \\
\qquad\qquad \textit{getuid}(p?) = \textit{creator}(i?) \hfill (4) \\
\textit{msg\_qbytes}? \leq \textit{MSGMNB} \lor \textit{getuid}(p?) = \textit{SUPERUSER} \hfill (5) \\
r! = 0 \\
\\
\textit{getkey}' = \textit{getkey} \\
\textit{msgq}' = \textit{msgq} \\
\textit{creator}' = \textit{creator} \\
\textit{owner}' = \textit{owner} \oplus \{i? \mapsto \textit{msg\_uid}?\} \hfill (6) \\
\textit{group}' = \textit{group} \oplus \{i? \mapsto \textit{msg\_gid}?\} \hfill (7) \\
\textit{perm}' = \textit{perm} \oplus \{i? \mapsto \textit{msg\_perm}?\} \hfill (8) \\
\textit{qbytes}' = \textit{qbytes} \oplus \{i? \mapsto \textit{msg\_qbytes}?\} \hfill (9) \\
\textit{cbytes}' = \textit{cbytes} \\
\textit{msg\_ts}' = \textit{msg\_ts} \\
\textit{errno}' = \textit{errno} \\
\hline
\end{array}
$$

Für diesen Fall muß als Kommando IPC_SET vorgegeben sein (1).

Auch muß die UID des Prozesses der Superuser (2), der Besitzer der Message-Queue (3) oder der Erzeuger der Message-Queue sein (4).

Möchte man dabei die maximale Länge der Message-Queue verändern, so muß diese entweder kleiner als die vom System vorgegebene maximale Länge sein oder die UID des Prozesses muß Superuser sein (5).

Je nachdem, welche Eingaben existieren, werden die Objekte des Zustandraumes verändert.

Soll der Besitzer der Message-Queue verändert werden, so wird $owner$ beeinflußt (6), soll die Gruppe der Message-Queue verändert werden, wird bei $group$ der entsprechende Wert durch die Vorgabe ersetzt (7). Bei einer Veränderung der Zugriffsrechte wird $perm$ beeinflußt (8), bei einer Veränderung der maximalen Länge der Message-Queue hingegen $qbytes$ (9).

Die restlichen Objekte bleiben stets unverändert.

Bei diesem Schema geht es im Unterschied zu $Msgctl2$ darum Werte zu verändern, demnach werden die Objekte des Zustandraumes mehr oder weniger verändert, es werden keine Informationen ausgegeben.

O.8.2 Fehlerfall

Auch bei dieser Message-Queue Operation gibt es wieder einige Fehlerfälle, die wir nicht spezifiziert haben. Die Begründung wurde in den vorherigen Kapiteln mehrfach genannt, daher hier nur ein kurzes Aufführen der ebenfalls schon aus den vorherigen Kapiteln bekannten nicht spezifizierten Fehler:

- EFAULT: ein Pointer zeigt auf eine nicht exitstierende Adresse. Für dies Behandlung ist der Kernel zuständig.
- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden, und im selben „Slot“ ($id\%MSGMNI$) gibt es wieder eine neue Messagequeue. Dies ist allerdings so nah an der Implementierung, daß wir das in der Spezifikation nicht von EINVAL unterscheiden, insbesondere, da dieser Fehlerfall nicht ausreichend dokumentiert ist. Die Manual–Page von $msgctl$ sagt dazu: „*SVID dies not document the EIDRM error condition.*“.

Die spezifizierten Fehlerfälle erscheinen ziemlich lang, was daran liegt, daß in allen Fällen dieselben Ein- und Ausgaben wie bei den Erfolgsfällen verwendet werden. Dies ermöglicht eine spätere Schemadisjunktion.

Allen Fehlerfällen ist wieder gleich, daß sie als Returnwert die -1 besitzen. Genauso wird aus dem Zustandsraum einzig $errno$ auf den Namen des Fehlers gesetzt, alle anderen Objekte des Zustandraumes bleiben unverändert.

Beim Fehler $EACCESS$ ist das Kommando IPC_STAT gesetzt, der Prozeß möchte also die Informationen über die Message-Queue ausgegeben bekommen. Leider besitzt er keine Leserechte für die Message-Queue.

$$
\begin{array}{|l}
\hline
\textit{msgctlEACCES} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
cmd? : CMD \\
msg\_uid? : UID \\
msg\_gid? : GID \\
msg\_perm? : \mathbb{P}\, PERM \\
msg\_qbytes? : \mathbb{N} \\
msg\_uid! : UID \\
msg\_gid! : GID \\
msg\_perm! : \mathbb{P}\, PERM \\
msg\_qbytes! : \mathbb{N} \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \qquad (1) \\
cmd? = IPC\_STAT \qquad (2) \\
\neg\, (getuid(p?)\ \mathrm{mayread}\ (perm(i?), owner(i?), group(i?))) \qquad (3) \\
r! = -1 \\
errno' = EACCES \\
\hline
\end{array}
$$

Voraussetzung ist, daß eine korrekte Message-Queue adressiert ist (1) und das Kommando IPC_STAT gesetzt ist (2).

Desweiteren darf der Prozeß keine Leserechte für die Message-Queue besitzen (3).

$EINVAL$ ist der schon bekannte Fall, daß eine falsche Angabe gemacht wird, entweder existiert die Message-Queue nicht oder das Kommando exisitiert nicht. Für diesen Fall ist dieses Schema entsprechend.

$$
\begin{array}{|l}
\hline
\textit{msgctlEINVAL} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
cmd? : CMD \\
msg\_uid? : UID \\
msg\_gid? : GID \\
msg\_perm? : \mathbb{P}\, PERM \\
msg\_qbytes? : \mathbb{N} \\
msg\_uid! : UID \\
msg\_gid! : GID \\
msg\_perm! : \mathbb{P}\, PERM \\
msg\_qbytes! : \mathbb{N} \\
r! : RETURN \\
\hline
i? \notin \operatorname{dom} msgq \lor \qquad (1) \\
cmd? \notin CMD \qquad (2) \\
r! = -1 \\
errno' = EINVAL \\
\hline
\end{array}
$$

Als Vorbedingung für dieses Schema ist entweder die angegebenen ID nicht vorhanden (1) oder das angegebene Kommando ist nicht korrekt (2).

Für die bessere Verständlichkeit haben wir uns entschieden den Fehler $EPERM$ in zwei Schemata aufzuteilen. Er beschreibt die Situation, das ein Prozeß nicht die notwendigen Rechte für die Ausführung der Aktion besitzt.

Ein weiterer Grund für diese Aufteilung ist der Source-Code, in dem ebenfalls zwei Fallunterscheidungen für diesen Fehler vorhanden sind.

Dieser Fall tritt ein, wenn bei $Msgctl1$ oder $Msgctl3$ der Prozeß nicht die notwendigen Rechte besitzt, er also weder Superuser noch Eigentümer noch Erzeuger der Message-Queue ist.

$msgctl1EPERM$

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
cmd? : CMD \\
msg\_uid? : UID \\
msg\_gid? : GID \\
msg\_perm? : \mathbb{P}\, PERM \\
msg\_qbytes? : \mathbb{N} \\
msg\_uid! : UID \\
msg\_gid! : GID \\
msg\_perm! : \mathbb{P}\, PERM \\
msg\_qbytes! : \mathbb{N} \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
cmd? = IPC\_SET \lor cmd? = IPC\_RMID \qquad (1) \\
getuid(p?) \neq SUPERUSER \land \qquad (2) \\
\quad getuid(p?) \neq owner(i?) \land \qquad (3) \\
\quad getuid(p?) \neq creator(i?) \qquad (4) \\
r! = -1 \\
\\
errno' = EPERM
\end{array}
$$

Entsprechend dem einführendem Text muß das Kommando IPC_SET oder IPC_RMID gesetzt sein (1).

Und der Prozeß darf keine Rechte als Superuser (2), Message-Queue Eigentümer (3) oder Message-Queue Erzeuger besitzen (4).

Ist hingegen das Kommando IPC_SET gesetzt, so müßte der Prozeß Superuser-Rechte besitzen, um die Systemgrenzen der maximalen Message-Queue-Länge zu überschreiten. Ist dies nicht der Fall, so tritt folgendes Schema in Kraft:

$$
\begin{array}{|ll}
\multicolumn{2}{l}{msgctl2EPERM} \\
\Delta MsgQStateSpace & \\
p? : PID & \\
i? : ID & \\
cmd? : CMD & \\
msg\_uid? : UID & \\
msg\_gid? : GID & \\
msg\_perm? : \mathbb{P}\, PERM & \\
msg\_qbytes? : \mathbb{N} & \\
msg\_uid! : UID & \\
msg\_gid! : GID & \\
msg\_perm! : \mathbb{P}\, PERM & \\
msg\_qbytes! : \mathbb{N} & \\
r! : RETURN & \\
\hline
i? \in \operatorname{dom} msgq & \\
cmd? = IPC\_SET & (1) \\
getuid(p?) \neq SUPERUSER & (1) \\
msg\_qbytes? > MSGMNB & (1) \\
r! = -1 & \\
errno' = EPERM & \\
\hline
\end{array}
$$

Als Kommando wird IPC_SET vorausgesetzt (1), der Prozeß hat keine Superuser-Rechte (2) und die gewünschte maximale Länge der Message-Queue ist größer als die vom System vorgegebene maximale Länge (3).

Dieses unterscheidet sich in soweit vom vorherigem Schema, als daß es um ein anderes Kommando geht und der Eigentümer und der Erzeuger der Message-Queue dieses Recht nicht besitzen.

O.8.3 User-Interface

In diesem Abschnitt setzen wir die Fallunterscheidungen wieder zu dem einen Fall zusammen. Dies wird durch die Verwendung derselben Ein- und Ausgaben ermöglicht.

Mit $Msgctl$ wollen wir wieder den Erfolgsfall der Message-Queue Operation $msgctl()$ bezeichnen, der sich aus der Schemadisjunktion der einzelnen Erfolgsfälle ergibt:

$$Msgctl \mathrel{\widehat=} Msgctl1 \lor Msgctl2 \lor Msgctl3$$

Die Gesamtzahl der Fehler ergibt sich aus der Schemadisjunktion:

$$Msgctlfault \mathrel{\widehat=} msgctlEACCES \lor msgctlEINVAL \lor msgctl1EPERM \lor msgctl2EPERM$$

Für den Benutzer des Systems stellt sich selbst diese Aufteilung in Erfolgs- und Fehlerfall nicht. Demnach führen wir diese auch noch zusammen:

$$msgctl \mathrel{\widehat=} Msgctl \lor Msgctlfault$$

Somit entspricht das abstrakte formal spezifizierte $msgctl$ dem nach Aussen sichtbaren Verhalten der Message-Queue Operation $msgctl()$.

O.9 Ergebnis

Wir haben während der Spezifikation sehr viel gelernt über die Spezifikationssprache Z und Spezifikation im allgemeinen. Und wir haben die üblichen Probleme einer formalen Spezifikation kennengelernt.

Es entsteht eigentlich immer ein zeitliches Problem, die formale Spezifikation wird immer wesentlich aufwendiger, als man es sich am Anfang gedacht hat.

Auch sind die Vorgaben nicht immer eindeutig. Obwohl es bei uns anders war, als die Erstellung einer Spezifikation normalerweise abläuft, so waren doch einige Aussagen in der Online-Dokumentation nicht eindeutig und wir mußten den Source-Code zu Rate ziehen bzw. das Systemverhalten testen.

Wir hatten $msgq$ als eine partielle Abbildung von der Menge der KEY in die Menge der Message-Queues spezifiziert. Dabei hatten wir angenommen, daß bei einem $msgget$ mit dem KEY $IPC\_PRIVATE$ ein neuer, noch nicht vorhandener KEY gewählt wird. In der Implementierung von Linux und Solaris ist dies jedoch nicht so, sondern es wird als Key $IPC\_PRIVATE$ bzw. 0 gesetzt. Dieses haben wir über das Testen mit kleinen Programmen herausgefunden.

Dies war nur ein Beispiel für einen Fehler, der zu kleinen Änderungen führte. Allerdings hatten wir auch einen Fehler mit eingebaut, für dessen Behebung die ganze Spezifikation umgeschrieben werden mußte.

Wir waren davon ausgegangen, daß die ID, mit denen man in $msgsnd$, $msgrcv$ und $msgctl$ die Message-Queue adressiert, jeweils für einen Prozeß vergeben wird. Dies ist nicht so, sondern die ID für eine Message-Queue wird systemweit einheitlich vergeben. Somit kann ein Prozeß sogar auf eine Message-Queue zugreifen, wenn er gar kein $msgget$ vorher aufgerufen hat, ihm muß nur die ID bekannt sein und er muß die nötigen Rechte besitzen.

Demnach mußten wir viele Objekte des Zustandraumes abändern, denn bisher waren wir davon ausgegangen, daß Message-Queues stets über ihren KEY adressiert würden. Tatsächlich wird die Message-Queue aber über die systemweit einheitlich vergebene ID angesprochen. Dies hatte natürlich Auswirkung auf die gesamte Spezifikation und diese mußte vollständig überarbeitet werden.

Diese und weitere Fehler sind inzwischen vollständig eleminiert. Natürlich haben wir dann auch überprüft, ob unsere Spezifikation korrekt und vollständig ist.

Zuerst haben wir unsere Spezifikation auf syntaktische Korrektheit geprüft. Hierfür gibt es Hilfsmittel, wir haben das Tool `fuzz` benutzt. Dieses ist ein Typchecker, der auf dem LaTeX-Source eines Dokumentes arbeitet und die in bestimmten Umgebungen gesetzten Z-Spezifikationen prüft.

Nach einem Durchlauf der hier vorliegenden Spezifikation teilt uns Fuzz mit, daß die Spezifikation syntaktisch korrekt ist.

Fuzz ist aber auch ein Typechecker, er prüft also zusätzlich, ob an sämtlichen Stellen die Typen korrekt verwendet werden. Auch dieses ist der Fall.

Nun könnte es natürlich noch sein, daß ein Fall existiert, den wir nicht spezifiziert haben und somit unsere Spezifikation nicht vollständig wäre. Leider gibt es hierfür keine Toolunterstützung, somit haben wir selbst Tabellen erstellt, die die Vorbedingungen der einzelnen Schemata darstellen. Bei einer Überprüfung haben wir erkannt, daß lediglich die Fälle, die wir in Z wegen der fehlenden Zeitkomponente nicht zu spezifizieren waren, fehlen. Diese sind in den einzelnen Kapiteln jeweils aufgeführt, insbesondere handelt es sich um den Fall, wenn das Flag IPC_NOWAIT nicht gesetzt ist.

Diese Tabellen wollen wir natürlich nicht vorenthalten und stellen sie hier vor.

Es folgt zuerst die Tabelle mit den Vorbedingungen für die verschiedenen Msgget-Operationen, dann geht es mit den Msgsnd-Operationen weiter bevor wir die Vorbedingungen für die Msgrcv-Operationen darlegen und wir schließen mit der Tabelle über die Msgctl-Operationen.

Operation	KEY	FLAGS
Msgget1	nicht vorhanden $\wedge \neq$ IPC_PRIVATE	IPC_CREATE
Msgget2	$\neq$ IPC_PRIVATE	$\neg$ (IPC_CREATE $\wedge$ IPC_EXCL)
Msgget3	IPC_PRIVATE	
msggetEACCESS	$\neq$ IPC_PRIVATE	
msggetEEXIST	$\neq$ IPC_PRIVATE	IPC_CREATE $\wedge$ IPC_EXCL
msggetENOENT	nicht vorhanden $\wedge \neq$ IPC_PRIVATE	$\neg$ IPC_CREATE
msggetENOSPC msggetENOSPC	IPC_PRIVATE nicht vorhanden $\wedge \neq$ IPC_PRIVATE	 IPC_CREATE

Operation	SIZE	WRITE/READ
Msgget1	$<$ MSGMNI	
Msgget2		maywrite $\vee$ mayread
Msgget3	$<$ MSGMNI	
msggetEACCESS		$\neg$ maywrite $\wedge$ $\neg$ mayread
msggetEEXIST		
msggetENOENT		
msggetENOSPC msggetENOSPC	 $\geq$ MSGMNI	

Operation	ID	msgsz	qbytes
Msgsnd	vorhanden	$(msgsz \geq 0) \wedge (msgsz \leq MSGMAX)$	$msgsz + cbytes \leq qbytes$
msgsndEAGAIN	vorhanden		$msgsz + cbytes > qbytes$
msgsndEACCESS	vorhanden		
msgsdEINVAL	nicht vorhanden		
msgsdEINVAL	vorhanden	$(msgsz < 0) \vee (msgsz > MSGMAX)$	
msgsdEINVAL	vorhanden		

Operation	TYPE	WRITE/READ
Msgsnd	$TYPE > 0$	maywrite
msgsndEAGAIN		
msgsndEACCESS		$\neg$ maywrite
msgsdEINVAL		
msgsdEINVAL		
msgsdEINVAL	$TYPE \leq 0$	

Operation	ID	FLAGS	SIZE
Msgrcv1	vorhanden		MSG_NOERROR $\vee$ length msg $\leq$ msgsz
Msgrcv2	vorhanden	$\neg$ MSG_EXCEPT	MSG_NOERROR $\vee$ length msg $\leq$ msgsz
Msgrcv3	vorhanden	MSG_EXCEPT	MSG_NOERROR $\vee$ length msg $\leq$ msgsz
Msgrcv4	vorhanden		MSG_NOERROR $\vee$ length msg $\leq$ msgsz
msgrcv1E2BIG	vorhanden		$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcv2E2BIG	vorhanden	$\neg$ MSG_EXCEPT	$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcv3E2BIG	vorhanden	MSG_EXCEPT	$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcv4E2BIG	vorhanden		$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcvEACCESS	vorhanden		
msgrcvEINVAL	nicht vorhanden		
msgrcvEINVAL	vorhanden		msgsz $<$ 0
msgrcv1NOMSG	vorhanden	IPC_NOWAIT	
msgrcv2NOMSG	vorhanden	IPC_NOWAIT $\wedge$ $\neg$ MSG_EXCEPT	
msgrcv3NOMSG	vorhanden	IPC_NOWAIT $\wedge$ MSG_EXCEPT	
msgrcv4NOMSG	vorhanden	IPC_NOWAIT	

Operation	TYPE	msgq(i)	msgq $\restriction$ j	WRITE/READ
Msgrcv1	TYPE = 0	$\neq \{\}$	$\neq \{\}$	mayread
Msgrcv2	TYPE $>$ 0	$\neq \{\}$	$\neq \{\}$	mayread
Msgrcv3	TYPE $>$ 0	$\neq \{\}$	$\neq \{\}$	mayread
Msgrcv4	TYPE $<$ 0	$\neq \{\}$	$\neq \{\}$	mayread
msgrcv1E2BIG	t = 0	$\neq \{\}$		
msgrcv2E2BIG	t $>$ 0	$\neq \{\}$		
msgrcv3E2BIG	t $>$ 0	$\neq \{\}$		
msgrcv4E2BIG	t $<$ 0	$\neq \{\}$		
msgrcvEACCESS				$\neg$ mayread
msgrcvEINVAL msgrcvEINVAL				
msgrcv1NOMSG	t = 0	= {}	= {}	
msgrcv2NOMSG	t $>$ 0	= {}	= {}	
msgrcv3NOMSG	t $>$ 0	= {}	= {}	
msgrcv4NOMSG	t $<$ 0	= {}	= {}	

Operation	ID	CMD	UID
Msgctl1	vorhanden	IPC_RMID	Superuser $\vee$ owner $\vee$ creator
Msgctl2	vorhanden	IPC_STAT	
Msgctl3	vorhanden	IPC_SET	Superuser $\vee$ owner $\vee$ creator
msgctlEACCESS	vorhanden	IPC_STAT	
msgctlEINVAL msgctlEINVAL	nicht vorhanden	 nicht vorhanden	
msgctl1EPERM	vorhanden	IPC_RMID $\vee$ IPC_SET	$\neg$ (Superuser $\vee$ owner $\vee$ creator)
msgctl2EPERM	vorhanden	IPC_SET	

Operation	MSGMNB	WRITE/READ
Msgctl1		
Msgctl2		mayread
Msgctl3	(msgq_qbytes $\leq$ MSGMNB) $\vee$ Superuser	
msgctlEACCESS		$\neg$ mayread
msgctlEINVAL msgctlEINVAL		
msgctl1EPERM		
msgctl2EPERM	(msgq_qbytes $>$ MSGMNB) $\wedge$ $\neg$ Superuser	

Eine Überprüfung dieser manuell erstellten Tabellen ist auch nur manuell möglich, aber wie schon Beschrieben, es werden sämtliche möglichen Fälle abgedeckt.

Ein Fazit für uns ist, daß wir sehr viel gelernt haben über Spezifikation im Allgemeinen und speziell über die Spezifikationssprache Z, dabei sind uns weder die negativen noch die positiven Seiten verborgen geblieben. Natürlich haben wir auch sehr viel über das IPC-Package und speziell die Message-Queues gelernt.

Als Ergebnis für die Allgemeinheit bleibt ein vielleicht sinnvoller Beitrag für die Linux-Gemeinde.

Und natürlich eine vollständige und korrekte abstrakte formale Spezifikation der Message-Queues.

Anhang P. IPC: Verifikation der Message-Queues

P.1 Einleitung

Die vorliegende Verifikation entstand im Rahmen des studentischen Projektes LiVE! , das vom Wintersemester 1997/1998 bis Sommersemester 1999 an der Universität Bremen durchgeführt wurde. Der Sinn und Inhalt dieses Projektes ist am besten nachzulesen unter: `http://aerobee.informatik.uni-bremen.de/start.html`, beziehungsweise im Projektbericht.

In unserer Teilgruppe von LiVE! , nach dem Inter-Process-Communication Paket IPC genannt, beschäftigen wir uns mit der Spezifikation und Verifikation der Message-Queues unter Linux.

Wir haben die Message-Queues bereits formal spezifiziert, daher setzen wir auf dem Dokument “Message-Queue Abstraktion in Z“ auf. Dieses ist ein eigenständiges Dokument, das sich aber auch im Anhang des oben angeführten Projektberichts wiederfindet.

In diesem Dokument werden wir die Verifikation von Message-Queues behandeln. Auch dieses Dokument soll in sich geschlossen und vollständig sein, so daß Interessierte mit der Verifikation arbeiten können. Es wird aber auch in den Anhang des Projektberichtes gestellt.

Um die Verifikation übersichtlicher zu gestalten, haben wir aus der oben bereits erwähnten Spezifikation die relevanten Teile übernommen. Wer also beide Dokumente durchliest, wird einige Redundanz feststellen, dies geschah jedoch, um die Dokumente jeweils in sich abgeschlossen und lesbar zu halten.

Da wir nunmal die Message-Queues in der formalen Spezifikationssprache Z spezifiziert haben, werden wir diese auch weiterhin verwenden. Auch in diesem Dokument setzen wir Kenntnisse dieser Spezifikationssprache voraus, es sei auf die entsprechende Literatur verwiesen, wie zum Beispiel „Using Z – Specification, Refinement, and Proof“ von Jim Woodcock and Jim Davies [WD96], oder „The Z Notation: A Reference Manual“ von J. M. Spivey [Spi92].

Bei der Verifikation werden wir uns nur auf die Verifikation einer Message-Queue-Operation beschränken. In diesem Falle haben wir willkürlich $msgsnd$ gewählt. Diese Einschränkung geschah größtenteils aus Zeitgründen, da es uns nicht mehr möglich erscheint, im Rahmen unseres Projektes eine vollständige Verifikation durchzuführen. Auch dieses ist eine leidvolle Erfahrung, auf die wir im letzten Kapitel dieses Dokumentes oder im Projektbericht noch kurz eingehen werden.

Es gibt verschiedene Arten zu verifizieren/testen, je nachdem welche Anforderungen man hat, was zu verifizieren ist und wie man spezifiziert hat.

Da wir uns bei der Spezifikation für die Sprache Z entschieden haben, stehen uns nicht mehr alle Verifikationsmethoden zur Verfügung. Es gibt als Tool-Unterstützung nur reine Syntax- und Typechecker, wie wir mit `fuzz` auch einen benutzt haben. Aber weitere Tool-Unterstützung, um bestimmte Eigenschaften der Sprache zu prüfen, wie sie für die Spezifikationssprache CSP mit FDR vorhanden ist, oder die Sprache sogar ausführen zu können, ist in Z leider nicht gegeben.

Auch wären solche Tools und die heutigen Rechner wahrscheinlich mit diesem kleinen Beispiel aus der Realität eines Betriebssystems hoffnungslos überfordert. Selbst Testsysteme kommen mit komplexen Bereichen eines Betriebssystems nicht zurecht.

Daher haben wir uns auf eine in Z übliche formale Verifikationsmethode beschränkt, dem Data Refinement. Eine andere Verifikationsmethode wäre noch das Operation Refinement, aber die konnten wir wieder aus Zeitgründen nicht durchführen.

Beim Data Refinement versucht man, aus den Daten einer abstrakt gehaltenen Spezifikation konkretere Daten zu entwickeln, die man dann in einer Programmiersprache wie zum Beispiel C umsetzen kann. Dabei muß man bei jedem Schritt nicht nur darauf achten, sondern auch beweisen, daß die Umformung korrekt ist.

Bei unserem Projekt hatten wir die Besonderheit, daß die Implementation schon vorhanden war, und daß wir prüfen mußten, ob diese Implementation unserer Spezifikation entspricht, die wir anhand der Dokumentation erstellt haben.

Die abstrakte Spezifikation haben wir schon erstellt. Nun haben wir den Source-Code, den wir versuchen in Z-Notation zu überführen, damit erhalten wir eine konkrete Spezifikation. Anschließend überführen wir die abstrakte in die konkrete Spezifikation, wobei wir jeden einzelnen Schritt beweisen müssen. Sollte dies nicht möglich sein oder Differenzen auftreten, so hätten wir einen Fehler gefunden, entweder in einer unserer Spezifikationen, im Source-Code oder in der Dokumentation.

Um es vorweg schon zu sagen, bei dieser Verifikation sind wir lediglich auf Probleme mit der Spezifikationssprache und der Art zu beweisen gestoßen. Wir haben keine Fehler in der Implementation ausmachen können. Dies ist zum einen sehr beruhigend, da alles so funktioniert, wie es zu funktionieren hat. Zum anderen aber nicht sehr befriedigend, wenn man nach dem großen Aufwand keine Verbesserungen erzielen kann.

Unsere Verifikation findet sich wie folgt in diesem Dokument wieder:

Im Kapitel P.2 führen wir die für die Verifikation benötigten Datenstrukturen ein. Wir widmen uns erst den benötigten Basistypen, anschließend wenden wir uns den Konstanten zu bevor wir die verwendeten Basisfunktionen einführen.

Im Kapitel P.3 geben wir den abstrakten und konkreten Zustandsraum an, sowie eine Überführungsrelation.

In den Kapiteln P.4 bis P.10 findet die eigentliche Verifikation statt. Es werden die Schemata aus der Spezifikation übernommen und einzeln behandelt. Entsprechend gibt es die Aufteilung in einen Erfolgsfall und mehrere Fehlerfälle, die einem vom Benutzer nicht gewünschtem Systemverhalten entsprechen. Zuerst geben wir immer das abstrakte Schema aus der Spezifikation an. Dann stellen wir die konkrete Spezifikation vor. Und zum Abschluß verifizieren wir diese Schemata durch Data Refinement.

Im Kapitel P.11 stehen dann die Erfolge, Erfahrungen und Erlebnisse, die uns im Zusammenhang mit der Verifikation wiederfahren sind.

P.2 Datenstrukturen

In diesem Kapitel widmen wir uns zuerst dem Grundlegenden: die Datenstrukturen, mit denen wir später arbeiten.

Sowohl in diesem, als auch in den folgenden Kapiteln, werden wir versuchen für die bessere Verständlichkeit möglichst aussagekräftige Namen zu verwenden. Hierbei haben meist die im Source-Code verwendeten Namen als Vorlage gedient, bzw. die Namen aus der Spezifikation. Allerdings geht an einigen Stellen diese bessere Verständlichkeit zu Lasten der Lesbarkeit, da sich lange Wörter mit mathematischen Zeichen gemischt nur schwer lesen lassen. Wir hoffen aber einen guten Kompromis gefunden zu haben.

Da wir ein reales System betrachten, ist es ziemlich komplex. Entsprechend viele Datenstrukturen gibt es. Diese Menge an verschiedenen Datenstrukturen sollte aber nicht weiter schrecken, da sie größtenteils sehr einfache und in UNIX-Systemen weitverbreitete Strukturen darstellt. Auch werden diese Strukturen teilweise sehr abstrakt übernommen, in anderen Betriebssystemteilen als dem IPC-Package müßte man sie teilweise wesentlich feiner aufgliedern.

Wir haben die Datenstrukturen aus der formalen Spezifikation übernommen. Wer die Spezifikation schon gelesen hat, kann dieses Kapitel also getrost überspringen.

Dieses Kapitel beginnt mit den Basistypen, die grundlegend vorhanden sein müssen und mit denen, oder auf denen, die Message-Queues arbeiten. Sodann werden die Konstanten eingeführt, die systemweit feststehen sollen. Daran schließen sich die Basisfunktionen an, die im Prinzip jedem Prozeß zur Verfügung stehende Systemdienste darstellen.

P.2.1 Basistypen

Bei den Basistypen kann man zwei verschieden Arten unterscheiden.

Mit der einen Art wird nicht weiter gerechnet, die einzelnen Elemente müssen nur einem bestimmten Typen entsprechen. Also kann man sie wunderschön abstrakt halten.

Bei der anderen Art sieht es schon schwieriger aus, mit ihnen soll auch gerechnet werden. Dementsprechend muß eine Struktur mit spezifiziert werden, eventuell sogar noch entsprechende Rechenoperationen auf dieser Struktur. Da es sich aber bei der Struktur häufig um Zahlen handelt, sind diese Rechenoperationen schon implizit mitgegeben.

In einem System gibt es zuerst einmal mehrere Benutzer. Diese haben eine eindeutige Identifikation, die User-ID aus der Menge UID:

$[UID]$

Da wir uns in UNIX-artigen Systemen aufhalten gibt es für jeden User auch eine zugehörige Group-ID, diesmal aus der Menge GID:

$[GID]$

Ein Benutzer (user), eine Gruppe (group), oder „die ganze Welt" (other) können das Recht bekommen, in eine Message-Queue zu schreiben, beziehungsweise aus ihr zu lesen. Diese Rechte sind ähnlich den UNIX-Dateirechten, nur das execute-Bit hat keine Auswirkungen, und wurde deshalb auch weggelassen. Man kann es im realen System sogar setzen, nur hat es keinerlei Auswirkungen.

$PERM ::= u\_write \mid u\_read \mid g\_write \mid g\_read \mid o\_write \mid o\_read$

Ein Benutzer startet Prozesse. Diese müssen teilweise miteinander oder mit anderen Prozessen kommunizieren. Jeder Prozeß hat daher systemweit eine eindeutige Identifikation, den Prozeßidentifikator:

$[PID]$

Für die Kommunikation zwischen den Prozessen können Message-Queues verwendet werden. Für diese Kommunikation müssen verschiedene Prozesse dieselbe Message-Queue adressieren können. Dies geschieht über den KEY:

$KEY : \mathbb{P}(\mathbb{N} \cup \{0\})$

Für die einzelnen Operationen wird die Message-Queue jedoch über einen Identifikator adressiert:

$ID : \mathbb{P}(\mathbb{N} \cup \{0\})$

In einer Message-Queue werden Nachrichten gesendet. Diese bestehen aus einer Folge von Bytes und einem Typ für die Nachricht.

Da die Struktur der Folgen von Bytes nicht weiter relevant ist, nehmen wir eine Folge von Bytes als einen abstrakten Datentypen. Dies ist ein schönes Beispiel für die Abstraktion, denn eigentlich hätte man hier einzelne

Bytes als Datentypen nehmen müssen, die sich wiederum aus Bits zusammensetzen.

$[BYTES]$

Der Typ einer Nachricht wird in Form einer ganzen Zahl angegeben. Zu beachten ist, daß negative Zahlen zwar nicht als Typ einer Nachricht erlaubt sind, sie aber bei der Message-Queue Operation $msgrcv$ eine spezielle Bedeutung haben.

$$TYPE : \mathbb{P}\,\mathbb{Z}$$

Mit einer Message-Queue können verschiedene Operationen ausgeführt werden, wie sie in den folgenden Kapiteln noch ausführlich beschrieben werden.

Bei den Operationen $msgrcv$ und $msgget$ können dabei einige Flags als Option mitgegeben werden:

$$\begin{aligned} FLAG ::= {} & MSG\_EXCEPT \mid IPC\_NOWAIT \mid MSG\_NOERROR \mid \\ & IPC\_CREAT \mid IPC\_EXCL \end{aligned}$$

Der Operation $msgctl$ können dagegen bestimmte Kommandos als Argumente mitgegeben werden:

$$CMD ::= IPC\_STAT \mid IPC\_SET \mid IPC\_RMID$$

In der konkreten Implementierung gibt es spezielle Werte, die $msgque[i]$ haben kann, falls die Message-Queue nicht vorhanden ist, oder angibt, ob sie vorhanden ist. Dabei sind IPC_UNUSED und IPC_NOID im C-Source definiert, und $DEFINED$ repräsentiert einen gültigen Pointer.

$$SPECIALS ::= IPC\_NOID \mid IPC\_UNUSED \mid DEFINED$$

Message-Queue Operationen können eine Anzahl von Fehlern verursachen:

$$\begin{aligned} ERROR ::= {} & EACCES \mid EAGAIN \mid EEXIST \mid EINVAL \mid ENOENT \mid \\ & ENOMSG \mid E2BIG \mid ENOSPC \mid EPERM \mid NOERROR \end{aligned}$$

Und als Abschluß kann einem Prozeß als Returnwert mitgeteilt werden, ob ein Fehler vorliegt (Returnwert -1) oder es eine erfolgreiche Operation war (Returnwert 0 oder eine positive ganze Zahl).

$$RETURN : \mathbb{P}(\mathbb{N} \cup \{0, -1\})$$

P.2.2 Konstanten

Es gibt eine Reihe von Konstanten, die einmalig systemweit festgelegt sind. Diese sind insoweit konstant, als daß Änderungen nur durch Neukompilieren des Linux–Kernels möglich sind.

Um eine mißbräuchliche Nutzung des Systems zu verhindern bzw. um an keine Systemgrenzen zu stoßen, gibt es gewisse Begrenzungen für die Message-Queues.

In einem System sollte es eine Obergrenze für die Anzahl der Message-Queues geben. Die Konstante $MSGMNI$ gibt diese maximale Anzahl von Message-Queues an:

$$MSGMNI : \mathbb{N}$$

Damit eine Message-Queue nicht beliebig voll geschrieben werden kann, gibt es eine maximale Länge einer Message-Queue:

$$MSGMNB : \mathbb{N}$$

Auch Nachrichten sollten eine gewisse Länge nicht überschreiten.

$$MSGMAX : \mathbb{N}$$

Zudem gibt es einige spezielle Elemente aus den im vorherigen Abschnitt deklarierten Basistypen, die für einige Operationen benannt sein müssen.

So wird für $msgget$ ein spezielles Element aus KEY benötigt, um eine neue Message-Queue mit einem beliebigen KEY zu erzeugen.

$$IPC\_PRIVATE : KEY$$

Weil wir KEY schon recht konkret als natürliche Zahl (inklusive Null) definiert haben, setzen wir $IPC\_PRIVATE$ auf den konkreten Wert der Implementierung.

$$IPC\_PRIVATE = 0$$

Auch wird ein spezielles Element aus UID benötigt, daß den Superuser (root) kennzeichnet:

$$SUPERUSER : UID$$

P.2.3 Basisfunktionen

Im Folgendem werden einige Basisfunktionen eingeführt, die Systemdienste beschreiben, die den einzelnen Prozessen jeweils zur Verfügung stehen. Einige dieser Systemdienste kommen aus anderen Bereichen des Betriebsystems und werden von uns nicht näher spezifiziert. Dieses müßte bei der Spezifikation dieser Bereiche geschehen.

Es werden zuerst einmal einige Funktionen benötigt, die sich um die Rechtevergabe kümmern. So gibt es eine Funktion, die zu einer UID alle GIDs der Gruppen zuordnet, in denen der jeweilige User ist:

$$getallgroup : UID \rightarrow \mathbb{P}\, GID$$

Auch gibt es Funktionen, die einer PID die UID bzw. die GID des Prozesses zuordnet. Dieses sind partielle Funktionen, da nur verwendeten PIDs ein Wert zugeordnet ist.

$$\begin{array}{l} getuid : PID \nrightarrow UID \\ getgid : PID \nrightarrow GID \end{array}$$

Zum Lesen und Schreiben auf einer Message-Queue werden ebenfalls Funktionen benötigt.

So ermittelt eine allgemeinen Funktion, ob ein User auf eine Message-Queue schreiben darf, wobei diese Message-Queue bestimmte Rechte, eine bestimmte UID und eine bestimmte GID hat.

$$\begin{array}{l} \_\ \mathrm{maywrite}\ \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID) \\ \hline \forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet \\ u1\ \mathrm{maywrite}\ (p, u2, g) \Leftrightarrow (u1 = u2 \land u\_write \in p) \lor \\ \qquad (g \in getallgroup(u1) \land g\_write \in p) \lor \\ \qquad o\_write \in p \end{array}$$

Eine andere allgemeinen Funktion ermittelt, ob ein User von einer Message-Queue lesen darf, welche bestimmte Rechte, eine bestimmte UID und eine bestimmte GID hat.

$$\begin{array}{l} \_\ \mathrm{mayread}\ \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID) \\ \hline \forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet \\ u1\ \mathrm{mayread}\ (p, u2, g) \Leftrightarrow (u1 = u2 \land u\_read \in p) \lor \\ \qquad (g \in getallgroup(u1) \land g\_read \in p) \lor \\ \qquad o\_read \in p \end{array}$$

Betrachten wir eine einzelne Nachricht, so benötigen wir noch eine Funktion, die nur das Anfangsstück der Länge n der $BYTES$ zurückgibt. Da wir $BYTES$ sehr abstrakt gehalten haben können wir an dieser Stelle diese Funktion nicht näher spezifizieren.

$$truncate : BYTES \times \mathbb{N} \rightarrow BYTES$$

P.3 Message-Queue Zustandsraum

Nach unserer Sichtweise sind Message-Queues Kernel-Datenstrukturen. Hier geht es nun darum, den Zustandsraum, in dem sich diese Datenstrukturen befinden, zu beschreiben. Hierfür gibt es einige Objekte, die den Zustand der Menge der Message-Queues charakterisieren. Für diese gelten gewisse Konsistenzbedingungen.

Der abstrakte Zustandraum ist aus der Spezifikation übernommen und enthält die Sichtweise, wie es anhand der Dokumentation aussehen sollte. Im konkreten Zustandsraum haben wir den Source-Code versucht direkt in Z-Notation zu übernehmen. Weiterhin existiert noch eine Relation, die eine Verbindung zwischen abstrakter Sicht und konkreter Sicht darstellt.

P.3.1 Der abstrakte Zustandsraum

Hierbei handelt es sich um die Abstraktion des Zustandraumes für die Kommunikation mit Hilfe von Message-Queues. Dieser Part ist aus der Spezifikation übernommen.

Die hier benutzten Objekte haben folgende Bedeutungen:

- $getkey$ gibt zu jeder ID einen eindeutigen KEY zurück.
- $msgq$ weist einer ID eine Message-Queue zu.
- $creator$ gibt den Erzeuger einer Message-Queue an.
- $owner$ gibt den Besitzer einer Message-Queue an.
- $group$ gibt die Gruppe einer Message-Queue an.
- $perm$ zeigt die Rechte an, die ein Benutzer für die Message-Queue besitzt.
- $qbytes$ liefert die maximale Länge einer Message-Queue.
- $cbytes$ gibt die Länge einer Message-Queue wieder.
- msg_ts gibt für jede Nachricht aus einer Message-Queue die Länge dieser Nachricht an.
- $errno$ gibt die Art des Fehlers aus.

Diese zehn Objekte, ihr aktueller Zustand und ihre Veränderungen, sind von nun an bei allen Schemata von Bedeutung.

Die Abstraktion des Zustandsraumes für die Kommunikation mit Hilfe von Message-Queues sieht nun folgendermaßen aus:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MsgQStateSpace}} \\
\hline
getkey : ID \nrightarrow KEY \\
msgq : ID \nrightarrow \mathrm{seq}(TYPE \times BYTES) \\
creator : ID \nrightarrow UID \\
owner : ID \nrightarrow UID \\
group : ID \nrightarrow GID \\
perm : ID \nrightarrow \mathbb{P}\, PERM \\
qbytes : ID \nrightarrow \mathbb{N} \\
cbytes : ID \nrightarrow \mathbb{N} \\
msg\_ts : ID \nrightarrow \mathrm{seq}\, \mathbb{N} \\
errno : ERROR \\
\hline
\#\, \mathrm{dom}\, getkey \leq MSGMNI \qquad (1) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, getkey \qquad (2) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, creator \qquad (3) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, owner \qquad (4) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, group \qquad (5) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, perm \qquad (6) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, qbytes \qquad (7) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, cbytes \qquad (8) \\
\mathrm{dom}\, msgq = \mathrm{dom}\, msg\_ts \qquad (9) \\
\forall\, i : ID \mid i \in \mathrm{dom}\, msgq \bullet \mathrm{dom}(msgq(i)) = \mathrm{dom}(msg\_ts(i)) \qquad (10) \\
\forall\, i1, i2 : ID \mid i1 \in \mathrm{dom}\, msgq \wedge i2 \in \mathrm{dom}\, msgq \bullet \qquad (11) \\
\qquad getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \vee getkey(i1) = IPC\_PRIVATE \\
\hline
\end{array}
$$

Dabei haben die in der unteren Hälfte aufgestellten Konsistenzbedingungen folgende Bedeutung:

- (1): Die Anzahl der Message-Queues im System darf nicht größer sein, als im System insgesamt mit $MSGMNI$ erlaubt ist.
- (2) bis (9): Wenn es für eine bestimmte ID eine Message-Queue gibt, dann gibt es für diese ID auch $creator$, $owner$, $group$, $perm$, $qbytes$, $cbytes$ und msg_ts.
- (10): Jeder Nachricht in einer Message-Queue wird auch eine Länge zugeordnet. Dies bedeutet, daß wenn einer ID eine Message-Queue zugeordnet ist, msg_ts diese ID auf eine Sequenz von natürlichen Zahlen abbildet, die jeweils die Länge der einzelnen Nachrichten beschreiben.
- (11): Für jedes Element aus KEY, außer für $IPC\_PRIVATE$, gilt, daß es immer eindeutig ist. Dies bedeutet, daß niemals zwei Message-Queues über ein und denselben KEY angesprochen werden können, mit der einzigen Ausnahme, $IPC\_PRIVATE$.

P.3.2 Der konkrete Zustandsraum

Der konkrete Zustandsraum basiert auf dem Source-Code. Allerdings haben wir nur einen Teil konkretisiert, um an diesem den Ablauf exemplarisch zu verdeutlichen.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\mathit{CMsgQStateSpace}} \\
\mathit{msgque} : (0 \mathrel{..} \mathit{MSGMNI} - 1) \rightarrow \mathit{SPECIALS} & (1) \\
\mathit{msg\_perm\_key} : (0 \mathrel{..} \mathit{MSGMNI} - 1) \nrightarrow \mathit{KEY} & (2) \\
\mathit{msg\_perm\_seq} : (0 \mathrel{..} \mathit{MSGMNI} - 1) \nrightarrow (\mathbb{N} \cup \{0\}) & (3) \\
\mathit{msgq} : \mathit{ID} \nrightarrow \mathrm{seq}(\mathit{TYPE} \times \mathit{BYTES}) & \\
\mathit{creator} : \mathit{ID} \nrightarrow \mathit{UID} & \\
\mathit{owner} : \mathit{ID} \nrightarrow \mathit{UID} & \\
\mathit{group} : \mathit{ID} \nrightarrow \mathit{GID} & \\
\mathit{perm} : \mathit{ID} \nrightarrow \mathbb{P}\,\mathit{PERM} & \\
\mathit{qbytes} : \mathit{ID} \nrightarrow \mathbb{N} & \\
\mathit{cbytes} : \mathit{ID} \nrightarrow \mathbb{N} & \\
\mathit{msg\_ts} : \mathit{ID} \nrightarrow \mathrm{seq}\,\mathbb{N} & \\
\mathit{errno} : \mathit{ERROR} & \\
\hline
\forall\, \mathit{id} : (0 \mathrel{..} \mathit{MSGMNI} - 1) \bullet \mathit{msgque}(\mathit{id}) = \mathit{DEFINED} & (4) \\
\qquad\qquad\qquad \Leftrightarrow \mathit{id} \in \mathrm{dom}\,\mathit{msg\_perm\_key} & \\
\forall\, \mathit{id} : (0 \mathrel{..} \mathit{MSGMNI} - 1) \bullet \mathit{msgque}(\mathit{id}) = \mathit{DEFINED} & (5) \\
\qquad\qquad\qquad \Leftrightarrow \mathit{id} \in \mathrm{dom}\,\mathit{msg\_perm\_seq} & \\
\forall\, i : \mathit{ID} \bullet i \in \mathrm{dom}\,\mathit{msgq} \Leftrightarrow (\mathit{msgque}(i \bmod \mathit{MSGMNI}) = \mathit{DEFINED} \land & (6) \\
\quad \mathit{msg\_perm\_seq}(i \bmod \mathit{MSGMNI}) * \mathit{MSGMNI} + i \bmod \mathit{MSGMNI} = i) & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{creator} & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{owner} & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{group} & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{perm} & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{qbytes} & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{cbytes} & \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{msg\_ts} & \\
\forall\, i : \mathit{ID} \mid i \in \mathrm{dom}\,\mathit{msgq} \bullet \mathrm{dom}(\mathit{msgq}(i)) = \mathrm{dom}(\mathit{msg\_ts}(i)) & \\
\forall\, i1, i2 : \mathit{ID} \mid i1 \in \mathrm{dom}\,\mathit{msgq} \land i2 \in \mathrm{dom}\,\mathit{msgq} \bullet & (7) \\
\quad \mathit{msg\_perm\_key}(i1 \bmod \mathit{MSGMNI}) = \mathit{msg\_perm\_key}(i2 \bmod \mathit{MSGMNI}) & \\
\quad\quad \Rightarrow i1 = i2 \lor \mathit{msg\_perm\_key}(i1 \bmod \mathit{MSGMNI}) = \mathit{IPC\_PRIVATE} & \\
\hline
\end{array}
$$

Hierbei haben wir gegenüber dem abstrakten Zustandraum folgendes konkretisiert:

(1) Mit $msgque$ wird für jede Zahl zwischen 0 und $MSGMNI - 1$ angegeben, ob ihr eine Message-Queue zugeordnet ist. Ansonsten nimmt sie einen speziellen Wert an.
Gleichzeitig wird hiermit ausgesagt, daß die Größe des Domain maximal die Größe von $MSGMNI$ annimmt. Somit ist die mit (1) bezeichnete Zeile aus dem abstrakten $MsgQStateSpace$ hier enthalten.

(2) Mit msg_perm_key wird einer Message-Queue ein eindeutiger KEY zugeordnet

(3) Mit msg_perm_seq wird einer Message-Queue eine Zahl zugeordnet. In der Implementierung wird diese Zahl so gewählt, daß sie mit $MSGMNI$ multipliziert noch in eine Vorzeichenbehaftete 32 Bit Zahl paßt. Mit dieser Zahl wird die ID berechnet. Siehe dazu auch den Punkt (6).

(4) und (5) Diese beiden Gleichungen stellen sicher, daß genau dann, wenn eine Messagequeue vorhanden ist, und damit $msgque(id) = DEFINED$ ist, $msg\_perm\_key(id)$ und $msg\_perm\_seq(id)$ definiert sind.

(6) Diese Gleichung beschreibt, wie die ID aus msg_perm_seq berechnet wird. Dies stellt sicher, daß nicht versehentlich mit einer falschen ID auf die Message-Queue zugegriffen wird, weil beispielsweise die vorhandene Message-Queue seit dem letzten Zugriff gelöscht wurde, und mittlerweile eine neue Message-Queue mit derselben ID erzeugt wurde.

(7) Entspricht der letzten Konsistenzbedingung aus dem abstrakten Zustandsraum, nur mit konkreteren Werten und Berechnungen.

P.3.3 Abstraktionsfunktion

Dies ist eine Refinement–Relation, die eine Gleichwertigkeit der abstrakten und der konkreten Message-Queues herstellen soll.

Da wir bisher nur $getkey$ konkretisiert haben, muß auch nur dieses in der Relation berücksichtigt werden.

Rel

$$MsgQStateSpace$$
$$CMsgQStateSpace$$

$$\begin{array}{l}
\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet \\
\quad i \in \operatorname{dom} msg_perm_seq \land \\
\quad i \in \operatorname{dom} msg_perm_key \land \\
\quad (i + msg_perm_seq(i) * MSGMNI) \in \operatorname{dom} getkey \land \\
\quad msg_perm_key(i) = getkey(i + msg_perm_seq(i) * MSGMNI) \\
\forall\, i : ID \mid i \in \operatorname{dom} getkey \bullet \\
\quad msgque(i \mathbin{\mathsf{mod}} MSGMNI) = DEFINED
\end{array}$$

Durch diese Refinement–Relation entsteht eine Verbindung zwischen der abstrakten und der konkreten Sicht. Insbesondere muß für jede Operation im Konkreten Zustandsraum gelten, daß sie auch für die korrospondierenden Zustände im abstrakten Zustandsraum möglich sein muß. Diese kann man anschaulich im Diagramm P.1 sehen.

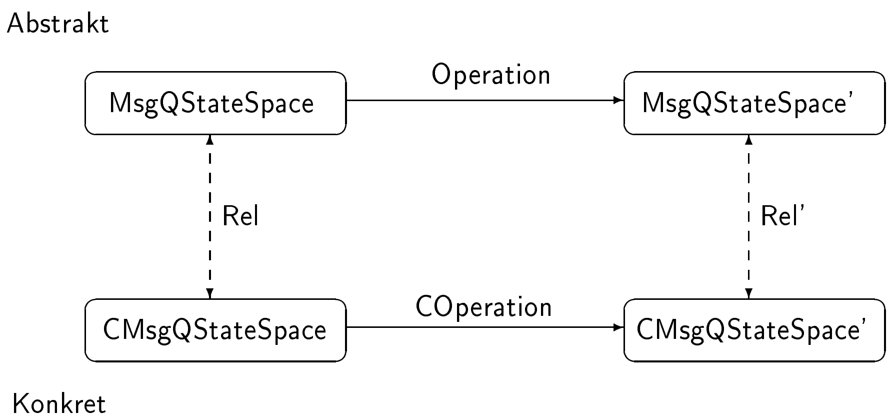


Abbildung P.1: Beziehung zwischen abstraktem und konkretem Zustandsraum

Wenn die Vorbedingungen für ein abstraktes Schema einer Message-Queue-Operation vorliegen, müssen auch die Vorbedingungen für das konkrete Schema der Message-Queue-Operation vorliegen. Die Nachbedingungen dieser beiden Schemata müssen durch die Relation übereinstimmen.

Da eine entsprechende Verbindung nun existieren muß, kann man dieses sehr gut für die Verifikation benutzen. Da wir ein Data-Refinement durchführen wollen, haben wir zu zeigen, daß die konkrete Sicht eine Verfeinerung der abstrakten Sicht ist. Somit haben wir in den folgenden Abschnitten zwei Beweisverpflichtungen.

Zuerst haben wir die *safety condition*, die Sicherheitsbedingung. Wenn die Vorbedingungen des abstrakten Schemas erfüllt sind und gleichzeitig die

Relation vorausgesetzt wird, müssen auch die Vorbedingungen des konkreten Schemas erfüllt sein. Also immer wenn es ein abstraktes Schema gibt, muß es auch ein entsprechendes konkretes Schema geben.

Anschließend gibt es noch die *liveness condition*. Wenn die Vorbedingungen für ein abstraktes Schema vorliegen, die Relation und das konkrete Schema vorausgesetzt werden, dann muß sich hieraus ein abstrakter Zustandsraum folgern lassen, der dem abstrakten Schema und der Relation genügt. Oder anders gesagt: Betrachtet man obiges Bild, so ist der Weg von $MsgQStateSpace$ über Rel nach $CMsgQStateSpace$ und über $COperation$ nach $CMsgQStateSpace'$ gegeben. Aus diesem muß sich dann auch der andere Weg verfolgen lassen.

P.4 Systeminitialisierung

Nachdem nun der Zustandsraum für die Message-Queues festgelegt ist, müssen wir die Startbedingungen für das System vorgeben.

Beim Start des Betriebssystems gibt es noch keine Prozesse, diese müssen erst noch erzeugt werden. Daher existieren auch noch keine Message-Queues, diese werden erst später initialisiert.

P.4.1 Die abstrakte Systeminitialisierung

Entsprechend einfach stellt sich somit die Initialisierung dar: Da noch keine Message-Queues existent sind, ist der Domain der benutzten Funktionen leer. Nur die Funktion, die auf bestimmte Werte abgebildet wird, ist auf den Default-Wert gesetzt.

$Init$

$$
\begin{array}{l}
MsgQStateSpace' \\
\hline
\mathrm{dom}\ getkey' = \emptyset \\
\mathrm{dom}\ msgq' = \emptyset \\
\mathrm{dom}\ creator' = \emptyset \\
\mathrm{dom}\ owner' = \emptyset \\
\mathrm{dom}\ group' = \emptyset \\
\mathrm{dom}\ perm' = \emptyset \\
\mathrm{dom}\ qbytes' = \emptyset \\
\mathrm{dom}\ cbytes' = \emptyset \\
\mathrm{dom}\ msg\_ts' = \emptyset \\
errno' = NOERROR
\end{array}
$$

Für die mit Z noch nicht so bewanderten Leser:

Dieser Kasten ist ein Schema, auf der oberen Linie steht der Name des Schemas, in diesem Falle $Init$. In der oberen Hälfte des Schemas, also über dem Querstrich, werden sämtliche Ein- und Ausgabewerte deklariert sowie andere Schemata eingebunden. Mit einem Quote (') wird ein Zustand, der nach dem Schema gelten muß, gekennzeichnet.

Die untere Hälfte haben wir noch einmal aufgeteilt, zuerst stellen wir immer die Vorbedingungen und Nachbedingungen, die für dieses Schema gelten, dar. In diesem Falle sind keine vorhanden.

Anschließend werden sämtliche Objekte des Zustandsraum und ihre eventuellen Veränderungen aufgezeigt.

P.4.2 Die konkrete Systeminitialisierung

Bei der konkreten Systeminitialisierung sind dieselben Bedingungen wie bei der abstrakten Systeminitialisierung vorhanden.

$$
\begin{array}{|l}
\hline
\textit{CInit} \\
\textit{CMsgQStateSpace}' \\
\hline
\forall i : (0 \mathrel{..} \textit{MSGMNI} - 1) \bullet \textit{msgque}'(i) = \textit{IPC\_UNUSED} \\
\operatorname{dom} \textit{msg\_perm\_key}' = \emptyset \\
\operatorname{dom} \textit{msg\_perm\_seq}' = \emptyset \\
\operatorname{dom} \textit{msgq}' = \emptyset \\
\operatorname{dom} \textit{creator}' = \emptyset \\
\operatorname{dom} \textit{owner}' = \emptyset \\
\operatorname{dom} \textit{group}' = \emptyset \\
\operatorname{dom} \textit{perm}' = \emptyset \\
\operatorname{dom} \textit{qbytes}' = \emptyset \\
\operatorname{dom} \textit{cbytes}' = \emptyset \\
\operatorname{dom} \textit{msg\_ts}' = \emptyset \\
\textit{errno}' = \textit{NOERROR} \\
\hline
\end{array}
$$

Auch hier wird entsprechend der abstrakten Systeminitialisierung der Domain der benutzten Funktionen auf die leere Menge, bzw. den Defaultwert, gesetzt. Dies geschieht, da noch keine Message-Queues im System existent sind.

P.4.3 Refinement

An dieser Stelle beginnt nun die eigentliche Verifikation. Es muß gezeigt werden, daß die konkrete Systeminitialisierung eine Verfeinerung der abstrakten Systeminitialisierung ist.

Da es sich hier nicht um eine Message-Queue Operation handelt, haben wir nicht die weiter oben beschriebenen Beweisverpflichtungen. Statt dessen haben wir zwei andere Refinement-Bedingungen.

Erste Bedingung

Als erste Refinement-Bedingung muß gezeigt werden, daß es stets einen konkreten Systeminitialisierungszustand gibt und dieser konsistent ist. Dies bedeutet:

$$\vdash \exists\, CMsgQStateSpace' \bullet CInit$$

Da alle Werte der Funktion $msgque$ auf IPC_UNUSED gesetzt wurden, und sie Domains der Funktionen $msg_p erm_k ey$ und $msg_p erm_s eq$ auf die leere Menge gesetzt wurden, Diese Funktionen also für alle Werte undefiniert sind, sind die Bedingungen (4) und (5) von CMsgQStateSpace erfüllt. Da die Domain vom msgq auf die leere Menge gesetzt wurde, ist auch Gleichungen (6) und (7) erfüllt. Da auch die Domains der restlichen Funktionen auf die leere Menge gesetzt wurden, sind auch die anderen anderen Bedingungen erfüllt. Somit haben wir einen Konsistenten Zustand.

Zweite Bedingung

Es muß für jeden konkreten Initialisierungszustand gezeigt werden, daß er einem abstrakten Initialisierungszustand entspricht.

Zu zeigen ist also, daß gilt:

$$CInit \vdash (\exists\, MsgQStateSpace' \bullet (Init \land Rel'))$$

Für den Beweis sind nur die Prädikate relevant, die vom abstrakten Zustandsraum zum konkreten Zustandsraum verändert wurden.

Wir führen also alle relevanten Prädikate der Hypothese an:

(1)	$\forall\, i : (0 \mathinner{..} MSGMNI - 1) \bullet$ $msgque'(i) = IPC\_UNUSED$	aus $CInit$
(2)	$\operatorname{dom} msg\_perm\_key' = \emptyset$	aus $CInit$
(3)	$\operatorname{dom} msg\_perm\_seq' = \emptyset$	aus $CInit$

Da wir nur die Veränderungen vom abstrakten zum konkreten Zustandsraum betrachten, müssen wir jetzt zeigen, daß es ein $getkey'$ gibt für das folgendes gilt:

	$\exists\, getkey' \bullet$	
(4)	$(\operatorname{dom} msgq' = \operatorname{dom} getkey'$	aus $MsgQStateSpace'$
(5)	$\land \operatorname{dom} getkey' = \emptyset$	aus $Init$

(6) $\land \forall i : (0 \mathinner{..} MSGMNI - 1) \mid$ aus Rel'

$msgque'(i) = DEFINED \bullet$
$i \in \operatorname{dom} msg\_perm\_seq' \land$
$i \in \operatorname{dom} msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\operatorname{dom} getkey'$
$\land\ msg\_perm\_key'(i) = getkey'(i +$
$msg\_perm\_seq'(i) * MSGMNI))$

(7) $\forall i : ID \mid i \in \operatorname{dom} getkey' \bullet$ aus Rel'

$msgque'(i \bmod MSGMNI)$
$= DEFINED$

Behauptung: $getkey' = \varnothing$ erfüllt (4)–(7).

Mit $getkey' = \varnothing$ ist die Gleichung (4) äquivalent zu $\operatorname{dom} msgq' = \operatorname{dom} \varnothing = \varnothing$. Dies ist auch in $CInit$ so definiert.

Weiterhin ist damit die Gleichung (5) äquivalent zu $\operatorname{dom} \varnothing = \varnothing$, was trivialerweise eine wahre Aussage ist.

Gleichung (6) ist eine wahre Aussage, da es kein i gibt, so daß $msgque'(i) = DEFINED$, denn in $CInit$ wurde für alle $i \in \{1..MSGMNI - 1\}$ $msgque'(i) = IPC\_UNUSED$ gesetzt.

Gleichung (7) ist mit $getkey' = \varnothing$ eine wahre Aussage, da der Domain von $getkey'$ damit auch $\varnothing$ ist, und eine Allaussage über eine leere Menge immer wahr ist.

Somit ist die *liveness condition* erfüllt.

Ergebnis

Da wir zeigen konnten, daß es stets einen konsistenten konkreten Systeminitialisierungszustand gibt und jeder konkreten Initialisierungszustand auch einem abstrakten Initialisierungszustand entspricht, haben wir die Systeminitialisierung verifizieren können.

P.5 Message-Queue Operation msgsnd()

Die Operation $msgsnd()$ wird von den Prozessen dazu benutzt, um in einer vorhandenen und ihnen bekannten Message-Queue Nachrichten zu senden. Eine Nachricht besteht dabei aus einem Typ und einer Anzahl von Bytes.

Es gibt eine Aufteilung in den erfolgreichen Fall des Sendens und in mehrere Fehlerfälle. Hierdurch ist eine bessere Lesbarkeit gegeben und auch das Verständnis wird vereinfacht. Auch werden im Source-Code an diesen Stellen Fallunterscheidungen getroffen, und somit ist ein Aufsplitten praktisch vorgegeben.

Im erfolgreichen Fall, der in diesem Kapitel behandelt wird, muß der Prozeß eine bestehende Message-Queue angeben, in die er seine Nachricht senden will, und für die er natürlich Schreibrechte besitzen muß. Von der Nachricht selbst muß er den Typ und die Länge der Nachricht wissen.

Zuerst führen wir das abstrakte Schema aus der formalen Spezifikation an. Anschließend erstellen wir aus dem Source-Code eine konkrete Spezifikation, die wir anschließend versuchen, per Data Refinement zu beweisen.

P.5.1 Abstraktes Senden einer Nachricht

In diesem Schema wird aus abstrakter Sicht das Senden einer Nachricht in einer Message-Queue spezifiziert. Es muß natürlich eine existierende Message-Queue adressiert sein und der Prozeß muß für diese Message-Queue auch Schreibrechte besitzen. Eine Nachricht besteht aus einem Typ und einer Anzahl von Bytes.

Msgsnd

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN
\end{array}
$$

$$
\begin{array}{lr}
i? \in \operatorname{dom} msgq & (1) \\
getuid(p?) \text{ maywrite } (perm(i?), owner(i?), group(i?)) & (2) \\
cbytes(i?) + msgsz? \leq qbytes(i?) & (3) \\
msgsz? \leq MSGMAX & (4) \\
msgsz? \geq 0 & (5) \\
t? > 0 & (6) \\
r! = 0 & (7)
\end{array}
$$

$$
\begin{array}{l}
getkey' = getkey \\
msgq' = msgq \oplus \{i? \mapsto msgq(i?) \frown \langle (t?, b?) \rangle \} \\
creator' = creator \\
owner' = owner \\
group' = group \\
perm' = perm \\
qbytes' = qbytes \\
cbytes' = cbytes \oplus \{i? \mapsto cbytes(i?) + msgsz?\} \\
msg\_ts' = msg\_ts \oplus \{i? \mapsto msg\_ts(i?) \frown \langle msgsz? \rangle \} \\
errno' = errno
\end{array}
$$

Für den Erfolgsfall und die Fehlerfälle in den folgenden Kapiteln gilt allgemein:

Zuerst wird der Zustandsraum eingebunden, auf dem Veränderungen stattfinden können.

Dann gibt es als Eingaben die Prozeßidentifikation *PID* sowie die Identifikation einer Message-Queue *ID*. Um eine Nachricht zu Senden benötigt man eine Nachricht, insbesondere ihren Typen und eine Anzahl von *BYTES*, die dabei eine bestimmte Länge besitzt. Eventuell können auch Flags vorgegeben werden.

Als Rückgabe erhält man einen Returnwert, bei den Fehlerfällen wird dieser stets auf -1, beim Erfolgsfall auf 0 gesetzt.

In der zweiten Hälfte des Schema sind die Vor- und Nachbedingungen angegeben, die speziell für diesen Fall zutreffen. Dies sind:

- (1): Die *ID* muß eine schon existierende Message-Queue adressieren.

- (2): Der Prozeß muß auf der durch die ID bezeichneten Message-Queue Schreibrechte besitzen.
- (3): Durch die zu sendende Nachricht darf die Länge der Message-Queue die maximale Länge von Message-Queues nicht überschreiten.
- (4): Die Länge der Nachricht darf nicht die maximale Länge einer Nachricht überschreiten.
- (5): Genauso darf die Länge einer Nachricht nicht einem negativen Wert entsprechen.
- (6): Der Typ der Nachricht muß positiv sein.
- (7): Daraufhin gibt es als Returnwert die 0 als Zeichen des erfolgreichen Sendens der Nachricht.

Auch die Objekte des Zustandraumes unterliegen gewissen Änderungen, wobei die Konsistenzbedingungen des Zustandsraumes nicht verletzt werden.

Da wäre zuerst $msgq$, bei der an die existierende Message-Queue die Nachricht mit Angabe des Typs hinten angehängt wird.

Bei $cbytes$ wird die Länge der Message-Queue um die Länge der gesendeten Nachricht erhöht.

Und bei msg_ts wird noch die Länge für die letzte Nachricht eingefügt.

Die restlichen Objekte bleiben vom Senden einer Nachricht unberührt und somit unverändert bestehen.

P.5.2 Konkretes Senden einer Nachricht

Und nun die Spezifikation des konkreten Sendens einer Nachricht in einer Message-Queue.

$$
\begin{array}{l}
\textbf{CMsgsnd} \\
\hline
\Delta \mathit{CMsgQStateSpace} \\
t? : \mathit{TYPE} \\
b? : \mathit{BYTES} \\
\mathit{msgsz}? : \mathbb{N} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
f? : \mathbb{P}\,\mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} \mathit{msgq} \\
\mathit{getuid}(p?) \text{ maywrite } (\mathit{perm}(i?), \mathit{owner}(i?), \mathit{group}(i?)) \\
\mathit{cbytes}(i?) + \mathit{msgsz}? \leq \mathit{qbytes}(i?) \\
\mathit{msgsz}? \leq \mathit{MSGMAX} \\
\mathit{msgsz}? \geq 0 \\
t? > 0 \\
r! = 0 \\
\\
\mathit{msgque}' = \mathit{msgque} \qquad (1) \\
\mathit{msg\_perm\_key}' = \mathit{msg\_perm\_key} \qquad (2) \\
\mathit{msg\_perm\_seq}' = \mathit{msg\_perm\_seq} \qquad (3) \\
\mathit{msgq}' = \mathit{msgq} \oplus \{i? \mapsto \mathit{msgq}(i?) \frown \langle (t?, b?) \rangle\} \\
\mathit{creator}' = \mathit{creator} \\
\mathit{owner}' = \mathit{owner} \\
\mathit{group}' = \mathit{group} \\
\mathit{perm}' = \mathit{perm} \\
\mathit{qbytes}' = \mathit{qbytes} \\
\mathit{cbytes}' = \mathit{cbytes} \oplus \{i? \mapsto \mathit{cbytes}(i?) + \mathit{msgsz}?\} \\
\mathit{msg\_ts}' = \mathit{msg\_ts} \oplus \{i? \mapsto \mathit{msg\_ts}(i?) \frown \langle \mathit{msgsz}? \rangle\} \\
\mathit{errno}' = \mathit{errno} \\
\hline
\end{array}
$$

Das konkrete Schema ist unterschiedlich zum abstrakten Schema:

Anstelle eines unveränderten getkey des Zustandsraumes werden die drei konkreteren Objekte des Zustandsraumes verwendet und ebenfalls nicht verändert (1 bis 3).

P.5.3 Refinement

Nun muß auch hier gezeigt werden, daß das konkrete Senden in einer Message-Queue eine Verfeinerung des abstrakten Sendens ist. Das Vorgehen entspricht hierbei der in Kapitel P.3.3 beschriebenen Vorgehensweise.

Erste Bedingung (safety condition)

Zu zeigen ist hier, daß, falls abstrakte Vorbedingungen erfüllt werden, auch im konkreten Fall die Vorbedingungen erfüllt werden.

$$(\mathrm{pre}\ Msgsnd) \land Rel \vdash \mathrm{pre}\ CMsgsnd$$

Die Vorbedingung zum abstrakten $Msgsnd$ und die Relation Rel sehen folgendermaßen aus. Da bei $Msgsnd$ der Zustandsraum mit eingebunden wird, haben wir auch die dortigen Konsistenzbedingungen mit zu beachten. Allerdings übernehmen wir nur die Konsistenzbedingungen, für die sich etwas ändert. Für die anderen ist nachher bei der Verifikation die Herleitung trivial, da sie unverändert auf beiden Seiten stehen.

(1) $i? \in \mathrm{dom}\ msgq$ — aus pre $Msgsnd$

(2) $getuid(p?)\ \mathrm{maywrite}\ (perm(i?), owner(i?), group(i?))$ — aus pre $Msgsnd$

(3) $cbytes(i?) + msgsz? \leq qbytes(i?)$ — aus pre $Msgsnd$

(4) $msgsz? \leq MSGMAX$ — aus pre $Msgsnd$

(5) $msgsz? \geq 0$ — aus pre $Msgsnd$

(6) $t? > 0$ — aus pre $Msgsnd$

(7) $\#\ \mathrm{dom}\ getkey \leq MSGMNI$ — aus pre $Msgsnd$

(8) $\mathrm{dom}\ msgq = \mathrm{dom}\ getkey$ — aus pre $Msgsnd$

(9) $\forall\, i1, i2 : ID \mid i1 \in \mathrm{dom}\ msgq \land i2 \in \mathrm{dom}\ msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$ — aus pre $Msgsnd$

(10) $\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \mathrm{dom}\ msg\_perm\_seq \land i \in \mathrm{dom}\ msg\_perm\_key \land (i + msg\_perm\_seq(i) * MSGMNI) \in \mathrm{dom}\ getkey \land msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$ — aus Rel

(11) $\forall\, i : ID \mid i \in \mathrm{dom}\ getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$ — aus Rel

Die Vorbedingungen zum konkreten $CMsgsnd$ sehen folgendermaßen aus:

(12) $i? \in \mathrm{dom}\, msgq$ aus pre $CMsgsnd$

(13) $getuid(p?)$ maywrite $(perm(i?), owner(i?), group(i?))$ aus pre $CMsgsnd$

(14) $cbytes(i?) + msgsz? \leq qbytes(i?)$ aus pre $CMsgsnd$

(15) $msgsz? \leq MSGMAX$ aus pre $CMsgsnd$

(16) $msgsz? \geq 0$ aus pre $CMsgsnd$

(17) $t? > 0$ aus pre $CMsgsnd$

(18) $\forall id : (0 \mathrel{..} MSGMNI - 1) \bullet msgque(id) = DEFINED \Leftrightarrow id \in \mathrm{dom}\, msg\_perm\_key$ aus pre $CMsgsnd$

(19) $\forall id : (0 \mathrel{..} MSGMNI - 1) \bullet msgque(id) = DEFINED \Leftrightarrow id \in \mathrm{dom}\, msg\_perm\_seq$ aus pre $CMsgsnd$

(20) $\forall i : ID \bullet i \in \mathrm{dom}\, msgq \Leftrightarrow (msgque(i \,\mathsf{mod}\, MSGMNI) = DEFINED \land msg\_perm\_seq(i \,\mathsf{mod}\, MSGMNI) * MSGMNI + i \,\mathsf{mod}\, MSGMNI = i)$ aus pre $CMsgsnd$

(21) $\forall i1, i2 : ID \mid i1 \in \mathrm{dom}\, msgq \land i2 \in \mathrm{dom}\, msgq \bullet msg\_perm\_key(i1 \,\mathsf{mod}\, MSGMNI) = msg\_perm\_key(i2 \,\mathsf{mod}\, MSGMNI) \Rightarrow i1 = i2 \lor msg\_perm\_key(i1 \,\mathsf{mod}\, MSGMNI) = IPC\_PRIVATE$ aus pre $CMsgsnd$

Wie nun leicht zu ersehen ist, ergeben sich einige Folgerungen von alleine:

(12) folgt aus (1)
(13) folgt aus (2)
(14) folgt aus (3)
(15) folgt aus (4)
(16) folgt aus (5)
(17) folgt aus (6)

Ähnlich einfach läßt sich direkt aus der Relation Rel folgern:

(18) folgt aus (10)
(19) folgt aus (10)

Etwas komplizierter wird es mit den restlichen Vorbedingungen. Der Beweis für (20):

Aus (10) können wir ersehen, daß für ein

$j \in \{0..MSGMNI - 1\}$ und $msgque(j) = DEFINED$

gilt:

$$j + msg\_perm\_seq(j) * MSGMNI \in \mathrm{dom}\, getkey$$

Sei $k = j + msg\_perm\_seq(j) * MSGMNI$
Nehmen wir diese Gleichung Modulo $MSGMNI$, so erhalten wir:

$$\begin{aligned} k \bmod MSGMNI &= (j + msg\_perm\_seq(j) * MSGMNI) \bmod MSGMNI \\ &= (j \bmod MSGMNI + (msg\_perm\_seq(j) * MSGMNI \\ &\qquad \bmod MSGMNI)) \bmod MSGMNI \\ &= (j \bmod MSGMNI + 0) \bmod MSGMNI \\ &= j \end{aligned}$$

Letzteres gilt, da $j \in \{0..MSGMNI - 1\}$

Somit kann man folgende Gleichung folgern:

$$\begin{aligned} &msgque(i \bmod MSGMNI) = DEFINED \Rightarrow \\ &\quad msg\_perm\_seq(i \bmod MSGMNI) * MSGMNI + i \bmod MSGMNI = i \end{aligned}$$

Nun ist für den Beweis von (20) zu zeigen, daß die Äquivalenz zwischen $i \in \mathrm{dom}\, msgq$ und $msgque(i \bmod MSGMNI) = DEFINED$ gilt.

Für die eine Richtung kann man dies aus (11) und (8) folgern.

Für die andere Richtung kann man aus (10) folgern, daß, wenn für ein $j \in \mathrm{dom}\, msgque = DEFINED$ ist, $j + msg\_perm\_seq(j) * MSGMNI \in \mathrm{dom}\, getkey$ gilt. Wenn wir jetzt $i = j + msg\_perm\_seq(j) * MSGMNI$ setzen, sind beide Gleichungen auf der rechten Seite von (20) erfüllt, und mit (8) kann man daraus dann die linke Seite folgern.

Dagegen ist der Beweis für (21) wieder überschaubar:
Mit (9) und der oben erstellten Gleichung, daß $k \bmod MSGMNI = j$ ist, kann man folgern, daß $getkey(i) = msg\_perm\_key(i \bmod MSGMNI)$ gilt. Damit ergibt sich dann (21).

Zweite Bedingung (liveness condition)

Wir müssen zeigen, daß es bei jedem konkreten $CMsgsnd$ einen abstrakten $MsgQStateSpace$ gibt, so daß ein konkretes $Msgsnd$ möglich ist, und die Relation Rel erfüllt ist.

$$(\text{pre } Msgsnd) \wedge Rel \wedge CMsgsnd \vdash \exists\, MsgQStateSpace' \bullet (Msgsnd \wedge Rel')$$

Zuerst führen wir alle Prädikate der Hypothese an, und erneut nur die relevanten Teile aus dem eingebundenen Zustandsraum:

(1)	$i? \in \text{dom}\, msgq$	aus pre $Msgsnd$
(2)	$getuid(p?)$ maywrite $(perm(i?), owner(i?), group(i?))$	aus pre $Msgsnd$
(3)	$cbytes(i?) + msgsz? \leq qbytes(i?)$	aus pre $Msgsnd$
(4)	$msgsz? \leq MSGMAX$	aus pre $Msgsnd$
(5)	$msgsz? \geq 0$	aus pre $Msgsnd$
(6)	$t? > 0$	aus pre $Msgsnd$
(7)	$\#\, \text{dom}\, getkey \leq MSGMNI$	aus pre $Msgsnd$
(8)	$\text{dom}\, msgq = \text{dom}\, getkey$	aus pre $Msgsnd$
(9)	$\forall\, i1, i2 : ID \mid i1 \in \text{dom}\, msgq \wedge i2 \in \text{dom}\, msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \vee getkey(i1) = IPC\_PRIVATE$	aus pre $Msgsnd$
(10)	$\forall\, i : (0 \mathinner{.\,.} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \text{dom}\, msg\_perm\_seq \wedge i \in \text{dom}\, msg\_perm\_key \wedge (i + msg\_perm\_seq(i) * MSGMNI) \in \text{dom}\, getkey \wedge msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(11)	$\forall\, i : ID \mid i \in \text{dom}\, getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$	aus Rel
(12)	$msgque' = msgque$	aus $CMsgsnd$
(13)	$msg\_perm\_key' = msg\_perm\_key$	aus $CMsgsnd$
(14)	$msg\_perm\_seq' = msg\_perm\_seq$	aus $CMsgsnd$

Nun expandieren wir die Folgerung, die eine Existenz-Quantifizierung ist, und beschränken uns dabei auf einen kleinen Teil, der nur beim konkreten Part verändert wurde.

$\exists\, getkey' : ID \pfun KEY \bullet$

(15) $(getkey' = getkey$ aus $Msgsnd$

(16) $\land \forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus Rel'
$msgque'(i) = DEFINED \bullet$
$i \in \operatorname{dom} msg\_perm\_seq' \land$
$i \in \operatorname{dom} msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\operatorname{dom} getkey' \land$
$msg\_perm\_key'(i) = getkey'(i +$
$msg\_perm\_seq'(i) * MSGMNI))$

(17) $\forall\, i : ID \mid i \in \operatorname{dom} getkey' \bullet$ aus Rel'
$msgque'(i \mathbin{\mathsf{mod}} MSGMNI) = DEFINED$

Mit (15) finden wir ein geeignetes $getkey'$, und setzen es in (16) und (17) ein.

(18) $\land \forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus (15) und (16)
$msgque'(i) = DEFINED \bullet$
$i \in \operatorname{dom} msg\_perm\_seq' \land$
$i \in \operatorname{dom} msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\operatorname{dom} getkey \land$
$msg\_perm\_key'(i) = getkey(i +$
$msg\_perm\_seq'(i) * MSGMNI))$

(19) $\forall\, i : ID \mid i \in \operatorname{dom} getkey \bullet$ aus (15) und (17)
$msgque'(i \mathbin{\mathsf{mod}} MSGMNI) = DEFINED$

Wie nun offensichtlich, ist (18) herzuleiten aus (10), (12), (13) und (14). Genauso offensichtlich ist (19) herzuleiten aus (11), (12), (13) und (14).

Ergebnis

Wir haben sowohl die *s*afety condition als auch die *l*iveness condition bewiesen. Demnach ist die konkrete Spezifikation eine Verfeinerung der abstrakten Spezifikation von $Msgsnd$.

P.6 Fehlerfall EAGAIN

In diesem und den folgenden Kapiteln werden die möglichen Fehlerfälle die entsprechend der formalen Spezifikation auftreten können, angeführt. Fehlerfälle heißt wiederum nicht, daß ein unerlaubter Systemzustand erreicht wird, sondern lediglich, daß ein vom Benutzer nicht erwünschtes Verhalten zustande kommt.

Es kann der Fall eintreten, daß eine Message-Queue schon voll ist und der Prozeß trotzdem versucht, eine Nachricht zu senden. Wenn er dabei nicht warten soll, so erhält er eine entsprechende Fehlermeldung und kann, so wie er möchte, weiterarbeiten.

P.6.1 Abstrakt EAGAIN

Dieses ist der Fall aus abstrakter Sicht betrachtet.

$$
\begin{array}{|l}
\hline
\textit{msgsndEAGAIN} \\
\Delta MsgQStateSpace \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \qquad (1) \\
cbytes(i?) + msgsz? > qbytes(i?) \qquad (2) \\
IPC\_NOWAIT \in f? \qquad (3) \\
r! = -1 \\
errno' = EAGAIN \\
\hline
\end{array}
$$

Es gibt bei allen abstrakten Fehlerfällen Gemeinsamkeiten, die wir hier anführen wollen, um sie nicht stets ansprechen zu müssen.

Die erste Hälfte des Schemas entspricht immer den schon beschriebenen üblichen Ein- und Ausgaben bei $msgsnd()$.

Der Returnwert $r!$ wird auf den für Fehlerfälle üblichen Wert von -1 gesetzt.

Für den Zustandsraum gilt allgemein, daß $errno$ auf den Namen des Fehlers gesetzt wird, also den Namen des Schema ohne das vorgefügte $msgsnd$.

Die restlichen Objekte des Zustandsraumes unterliegen keinerlei Veränderungen, werden deshalb hier auch nicht aufgeführt. Durch das Einbinden des Zustandsraumes ist dieses schon implizit mit vorgegeben.

Damit dieses Schema ausgeführt werden kann, muß eine ID vorgegeben werden, für die eine Message-Queue existiert (1). Auch muß die Länge der Message-Queue mit der Länge der zu sendenden Nachricht die maximale Länge dieser Message-Queue überschreiten (2). Zudem muß das Flag IPC_NOWAIT gesetzt sein (3).

P.6.2 Konkret EAGAIN

Und dies ist der Fall aus der konkreteren Sicht betrachtet.

$$
\begin{array}{l}
\textit{CmsgsndEAGAIN} \\
\hline
\Delta \mathit{CMsgQStateSpace} \qquad (1) \\
t? : \mathit{TYPE} \\
b? : \mathit{BYTES} \\
\mathit{msgsz}? : \mathbb{N} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
f? : \mathbb{P}\,\mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} \mathit{msgq} \\
\mathit{cbytes}(i?) + \mathit{msgsz}? > \mathit{qbytes}(i?) \\
\mathit{IPC\_NOWAIT} \in f? \\
r! = -1 \\
\\
\mathit{errno}' = \mathit{EAGAIN} \\
\hline
\end{array}
$$

Auch die konkreten Schemata sind genauso aufgebaut wie die abstrakten und unterscheiden sich lediglich bei den Vor- und Nachbedingungen voneinander.

Wie man sieht unterscheidet sich das konkrete Schema vom abstrakten Schema nur in einem Punkt, es wird der konkrete $\mathit{CMsgQStateSpace}$ eingebunden (1).

Alle anderen Punkte sind von der Konkretisierung nicht betroffen.

P.6.3 Refinement

Für die Verifikation muß erneut gezeigt werden, daß der konkrete Fall eine Verfeinerung des abstrakten Falls ist.

Erste Bedingung (safety condition)

Dabei ist zuerst einmal zu zeigen, daß, wenn alle abstrakten Vorbedingungen erfüllt werden, auch im konkreten Fall die Vorbedingungen erfüllt werden.

$$(\mathrm{pre}\ msgsndEAGAIN) \land Rel \vdash \mathrm{pre}\ CmsgsndEAGAIN$$

Die Vorbedingungen zum abstrakten $msgsndEAGAIN$ und die Relation Rel sehen folgendermaßen aus:

(1)	$i? \in \mathrm{dom}\ msgq$	aus $\mathrm{pre}\ msgsndEAGAIN$
(2)	$cbytes(i?) + msgsz? > qbytes(i?)$	aus $\mathrm{pre}\ msgsndEAGAIN$
(3)	$IPC\_NOWAIT \in f?$	aus $\mathrm{pre}\ msgsndEAGAIN$
(4)	$\#\,\mathrm{dom}\ getkey \leq MSGMNI$	aus $\mathrm{pre}\ msgsndEAGAIN$
(5)	$\mathrm{dom}\ msgq = \mathrm{dom}\ getkey$	aus $\mathrm{pre}\ msgsndEAGAIN$
(6)	$\forall\, i1, i2 : ID \mid i1 \in \mathrm{dom}\ msgq \land i2 \in \mathrm{dom}\ msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$	aus $\mathrm{pre}\ msgsndEAGAIN$
(7)	$\forall\, i : (0 \mathinner{..} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \mathrm{dom}\ msg\_perm\_seq \land i \in \mathrm{dom}\ msg\_perm\_key \land (i + msg\_perm\_seq(i) * MSGMNI) \in \mathrm{dom}\ getkey \land msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(8)	$\forall\, i : ID \mid i \in \mathrm{dom}\ getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$	aus Rel

Die Vorbedingungen für das konkrete $CmsgsndEAGAIN$ sehen folgendermaßen aus:

(9)	$i? \in \mathrm{dom}\ msgq$	aus $\mathrm{pre}\ CmsgsndEAGAIN$
(10)	$cbytes(i?) + msgsz? > qbytes(i?)$	aus $\mathrm{pre}\ CmsgsndEAGAIN$
(11)	$IPC\_NOWAIT \in f?$	aus $\mathrm{pre}\ CmsgsndEAGAIN$
(12)	$\forall\, id : (0 \mathinner{..} MSGMNI - 1) \bullet msgque(id) = DEFINED \Leftrightarrow id \in \mathrm{dom}\ msg\_perm\_key$	aus $\mathrm{pre}\ CmsgsndEAGAIN$

(13) $\forall id : (0 \mathinner{..} MSGMNI - 1) \bullet$ aus
$msgque(id) = DEFINED \Leftrightarrow$ pre $CmsgsndEAGAIN$
$id \in \mathrm{dom}\, msg\_perm\_seq$

(14) $\forall i : ID \bullet i \in \mathrm{dom}\, msgq \Leftrightarrow$ aus
$(msgque(i \,\mathsf{mod}\, MSGMNI) = DEFINED \land$ pre $CmsgsndEAGAIN$
$msg\_perm\_seq(i \,\mathsf{mod}\, MSGMNI) *$
$MSGMNI + i \,\mathsf{mod}\, MSGMNI = i)$

(15) $\forall i1, i2 : ID \mid$ aus
$i1 \in \mathrm{dom}\, msgq \land i2 \in \mathrm{dom}\, msgq \bullet$ pre $CmsgsndEAGAIN$
$msg\_perm\_key(i1 \,\mathsf{mod}\, MSGMNI) =$
$msg\_perm\_key(i2 \,\mathsf{mod}\, MSGMNI)$
$\Rightarrow i1 = i2 \lor$
$msg\_perm\_key(i1 \,\mathsf{mod}\, MSGMNI) =$
$IPC\_PRIVATE$

Nun sind folgende Vorbedingungen einfach herzuleiten:

(9) aus (1)
(10) aus (2)
(11) aus (3)

Für die Vorbedingungen (12) bis (15) haben wir dieses im Kapitel P.5.3 schon bewiesen. Die für den Beweis dort verwendeten Vorbedingungen sind hier ebenfalls gegeben.

Zweite Bedingung (liveness condition)

Wir müssen nun noch zeigen, daß es bei jedem $CmsgsndEAGAIN$ einen abstrakten $MsgQStateSpace'$ gibt, so daß ein konkretes $CmsgsndEAGAIN$ möglich ist, und die Relation Rel erfüllt ist.

$$(\text{pre } msgsndEAGAIN) \land Rel \land CmsgsndEAGAIN \vdash \\ \exists\, MsgQStateSpace' \bullet (msgsndEAGAIN \land Rel')$$

Zuerst führen wir alle Prädikate der Hypothese an:

(1) $i? \in \text{dom}\, msgq$ aus pre $msgsndEAGAIN$

(2) $cbytes(i?) + msgsz? > qbytes(i?)$ aus pre $msgsndEAGAIN$

(3) $IPC\_NOWAIT \in f?$ aus pre $msgsndEAGAIN$

(4) $\#\,\text{dom}\, getkey \leq MSGMNI$ aus pre $msgsndEAGAIN$

(5) $\text{dom}\, msgq = \text{dom}\, getkey$ aus pre $msgsndEAGAIN$

(6) $\forall\, i1, i2 : ID \mid i1 \in \text{dom}\, msgq \land i2 \in \text{dom}\, msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$ aus pre $msgsndEAGAIN$

(7) $\forall\, i : (0 \,..\, MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \text{dom}\, msg\_perm\_seq \land i \in \text{dom}\, msg\_perm\_key \land (i + msg\_perm\_seq(i) * MSGMNI) \in \text{dom}\, getkey \land msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$ aus Rel

(8) $\forall\, i : ID \mid i \in \text{dom}\, getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$ aus Rel

(9) $r! = -1$
aus $CmsgsndEAGAIN$

(10) $errno' = EAGAIN$ aus $CmsgsndEAGAIN$

Da $msgque'$, msg_perm_key' und msg_perm_seq' im Schema nicht explizit verändert werden, gilt implizit:

(11) $msgque' = msgque$ aus Text

(12) $msg\_perm\_key' = msg\_perm\_key$ aus Text

(13) $msg\_perm\_seq' = msg\_perm\_seq$ aus Text

Nun expandieren wir die Folgerung, die eine Existenz-Quantifizierung ist und beschränken uns dabei nur auf die beim Konkreten veränderten Teile:

$\exists\, getkey' : ID \rightarrowtail KEY \bullet$

(14) $(r! = -1$ aus $msgsndEAGAIN$

(15) $errno' = EAGAIN$ aus $msgsndEAGAIN$

(16) $\forall\, i : (0\,..\,MSGMNI - 1) \mid$ aus Rel'
$msgque'(i) = DEFINED \bullet$
$i \in \mathrm{dom}\, msg\_perm\_seq' \land$
$i \in \mathrm{dom}\, msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\mathrm{dom}\, getkey' \land$
$msg\_perm\_key'(i) = getkey'(i +$
$msg\_perm\_seq'(i) * MSGMNI)$

(17) $\forall\, i : ID \mid i \in \mathrm{dom}\, getkey' \bullet$ aus Rel'
$msgque'(i \bmod MSGMNI) = DEFINED$

Nehmen wir nun ein bestimmtes Element, und zwar setzen wir für $getkey'$ $getkey$ ein. Somit erhalten wir:

(18) $\forall\, i : (0\,..\,MSGMNI - 1) \mid$ aus (16) und Text
$msgque'(i) = DEFINED \bullet$
$i \in \mathrm{dom}\, msg\_perm\_seq' \land$
$i \in \mathrm{dom}\, msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\mathrm{dom}\, getkey \land$
$msg\_perm\_key'(i) = getkey(i +$
$msg\_perm\_seq'(i) * MSGMNI)$

(19) $\forall\, i : ID \mid i \in \mathrm{dom}\, getkey \bullet$ aus (17) und Text
$msgque'(i \bmod MSGMNI) = DEFINED$

Wie nun offensichtlich ist, ist (18) herzuleiten aus (7), (11), (12) und (13). Genauso ist (19) herzuleiten aus (8), (11), (12) und (13).

Ergebnis

Bei diesem ersten Fehlerfall läßt sich zeigen, daß die konkrete Spezifikation eine Verfeinerung der abstrakten Spezifikation ist.

P.7 Fehlerfall EACCES

In dem Fall, daß ein Prozeß keine Schreibrechte für eine Message-Queue besitzt und trotzdem versucht, eine Nachricht in dieser Message-Queue zu senden, tritt dieser Fehlerfall ein. Der Prozeß erhält eine entsprechende Rückmeldung und kann so weiterarbeiten, wie er möchte.

P.7.1 Abstrakt EACCES

Zuerst wieder die abstrakte Sichtweise auf diesen Fall.

$$
\begin{array}{|l}
\hline
\textit{msgsndEACCES} \\
\Delta \textit{MsgQStateSpace} \\
t? : \textit{TYPE} \\
b? : \textit{BYTES} \\
\textit{msgsz}? : \mathbb{N} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
f? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \qquad (1) \\
\neg\,(\textit{getuid}(p?)\ \mathrm{maywrite}\ (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?))) \qquad (2) \\
r! = -1 \\
\\
\textit{errno}' = \textit{EACCES} \\
\hline
\end{array}
$$

Für den Eintritt dieses Schema muß wieder ein gültige *ID* angegeben sein (1). Der Prozeß darf aber keine Schreibrechte für diese Message-Queue besitzen (2).

P.7.2 Konkret EACCES

Und nun die konkrete Sichtweise.

$$
\begin{array}{|l}
\hline
\textit{CmsgsndEACCES} \\
\Delta\, CMsgQStateSpace \qquad (1) \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
\neg\ (getuid(p?)\ \mathrm{maywrite}\ (perm(i?), owner(i?), group(i?))) \\
r! = -1 \\
\\
errno' = EACCES \\
\hline
\end{array}
$$

Wie man sieht unterscheidet sich das konkrete Schema vom abstraktem Schema nur in einem Punkt, es wird der konkrete $CMsgQStateSpace$ eingebunden (1).

Alle anderen Punkte sind von der Konkretisierung nicht betroffen.

P.7.3 Refinement

Auch hier muß für die Verifikation gezeigt werden, daß der konkrete Fall eine Verfeinerung des abstrakten Falls ist.

Erste Bedingung (safety condition)

Für die safety condition ist wieder zu zeigen, daß man aus den abstrakten Vorbedingungen und der Relation die konkreten Vorbedingungen folgern kann.

$$(\mathrm{pre}\ msgsndEACCES) \land Rel \vdash \mathrm{pre}\ CmsgsndEACCES$$

Die Vorbedingungen für das abstrakte $msgsndEACCES$ und die Relation sehen folgendermaßen aus:

(1)	$i? \in \mathrm{dom}\ msgq$	aus $\mathrm{pre}\ msgsndEACCES$
(2)	$\neg\ (getuid(p?)\ \mathrm{maywrite}$ $(perm(i?), owner(i?), group(i?)))$	aus $\mathrm{pre}\ msgsndEACCES$
(3)	$\#\ \mathrm{dom}\ getkey \leq MSGMNI$	aus $\mathrm{pre}\ msgsndEACCES$
(4)	$\mathrm{dom}\ msgq = \mathrm{dom}\ getkey$	aus $\mathrm{pre}\ msgsndEACCES$
(5)	$\forall\, i1, i2 : ID \mid i1 \in \mathrm{dom}\ msgq \land i2 \in \mathrm{dom}\ msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$	aus $\mathrm{pre}\ msgsndEACCES$
(6)	$\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \mathrm{dom}\ msg\_perm\_seq \land i \in \mathrm{dom}\ msg\_perm\_key \land (i + msg\_perm\_seq(i) * MSGMNI) \in \mathrm{dom}\ getkey \land msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(7)	$\forall\, i : ID \mid i \in \mathrm{dom}\ getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$	aus Rel

Dagegen sind die Vorbedingungen des konkreten $CmsgsndEACCES$ folgende:

(8)	$i? \in \mathrm{dom}\ msgq$	aus $\mathrm{pre}\ CmsgsndEACCES$
(9)	$\neg\ (getuid(p?)\ \mathrm{maywrite}$ $(perm(i?), owner(i?), group(i?)))$	aus $\mathrm{pre}\ CmsgsndEACCES$
(10)	$\forall\, id : (0 \mathrel{..} MSGMNI - 1) \bullet msgque(id) = DEFINED \Leftrightarrow id \in \mathrm{dom}\ msg\_perm\_key$	aus $\mathrm{pre}\ CmsgsndEACCES$
(11)	$\forall\, id : (0 \mathrel{..} MSGMNI - 1) \bullet msgque(id) = DEFINED \Leftrightarrow id \in \mathrm{dom}\ msg\_perm\_seq$	aus $\mathrm{pre}\ CmsgsndEACCES$

(12) $\forall\, i : ID \bullet i \in \operatorname{dom} msgq \Leftrightarrow$ aus
$(msgque(i \operatorname{mod} MSGMNI) = DEFINED \wedge$ pre $CmsgsndEACCES$
$msg\_perm\_seq(i \bmod MSGMNI) *$
$MSGMNI + i \bmod MSGMNI = i)$

(13) $\forall\, i1, i2 : ID \mid$ aus
$i1 \in \operatorname{dom} msgq \wedge i2 \in \operatorname{dom} msgq \bullet$ pre $CmsgsndEACCES$
$msg\_perm\_key(i1 \bmod MSGMNI) =$
$msg\_perm\_key(i2 \bmod MSGMNI)$
$\Rightarrow i1 = i2 \vee$
$msg\_perm\_key(i1 \bmod MSGMNI) =$
$IPC\_PRIVATE$

Nun sind folgende Vorbedingungen einfach herzuleiten:

(8) aus (1)
(9) aus (2)

Für die Vorbedingungen (10) bis (13) haben wir dieses im Kapitel P.5.3 schon bewiesen. Die für den Beweis dort verwendeten Vorbedingungen sind hier ebenfalls gegeben.

Zweite Bedingung (liveness condition)

Für die liveness condition müssen wir an dieser Stelle zeigen, daß man aus den Vorbedingungen des abstrakten $msgsndEACCES$, der Relation Rel und dem konkreten $CmsgsndEACCES$ einen abstrakten $MsgQStateSpace'$ folgern kann, der bestimmte Bedingungen erfüllt.

$$\begin{array}{l}(\text{pre } msgsndEACCES) \land Rel \land CmsgsndEACCES \vdash \\ \quad \exists\, MsgQStateSpace' \bullet (msgsndEACCES \land Rel')\end{array}$$

Zuerst führen wir alle Prädikate der Hypothese an.

(1)	$i? \in \text{dom}\, msgq$	aus pre $msgsndEACCES$
(2)	$\neg\, (getuid(p?) \text{ maywrite } (perm(i?), owner(i?), group(i?)))$	aus pre $msgsndEACCES$
(3)	$\#\, \text{dom}\, getkey \leq MSGMNI$	aus pre $msgsndEACCES$
(4)	$\text{dom}\, msgq = \text{dom}\, getkey$	aus pre $msgsndEACCES$
(5)	$\forall\, i1, i2 : ID \mid i1 \in \text{dom}\, msgq \land i2 \in \text{dom}\, msgq \bullet getkey(i1) = getkey(i2) \Rightarrow i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$	aus pre $msgsndEACCES$
(6)	$\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid msgque(i) = DEFINED \bullet i \in \text{dom}\, msg\_perm\_seq \land i \in \text{dom}\, msg\_perm\_key \land (i + msg\_perm\_seq(i) * MSGMNI) \in \text{dom}\, getkey \land msg\_perm\_key(i) = getkey(i + msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(7)	$\forall\, i : ID \mid i \in \text{dom}\, getkey \bullet msgque(i \bmod MSGMNI) = DEFINED$	aus Rel
(8)	$r! = -1$	aus $CmsgsndEACCES$
(9)	$errno' = EACCES$	aus $CmsgsndEACCES$

Da $msgque'$, msg_perm_key' und msg_perm_seq' im Schema nicht explizit verändert werden, gilt implizit:

(10)	$msgque' = msgque$	aus Text
(11)	$msg\_perm\_key' = msg\_perm\_key$	aus Text
(12)	$msg\_perm\_seq' = msg\_perm\_seq$	aus Text

Nun expandieren wir die Folgerung, die eine Existenz-Quantifizierung ist und beschränken uns dabei nur auf die beim Konkreten veränderten Teile.

$\exists\, getkey' : ID \rightarrowtail KEY \bullet$

(13) $r! = -1$ aus $msgsndEACCES$

(14) $errno' = EACCES$ aus $msgsndEACCES$

(15) $\forall\, i : (0 \,..\, MSGMNI - 1) \mid msgque'(i) = DEFINED \bullet i \in \operatorname{dom} msg\_perm\_seq' \land i \in \operatorname{dom} msg\_perm\_key' \land (i + msg\_perm\_seq'(i) * MSGMNI) \in \operatorname{dom} getkey' \land msg\_perm\_key'(i) = getkey'(i + msg\_perm\_seq'(i) * MSGMNI)$ aus Rel'

(16) $\forall\, i : ID \mid i \in \operatorname{dom} getkey' \bullet msgque'(i \bmod MSGMNI) = DEFINED$ aus Rel'

Nun nehmen wir ein spezielles $getkey'$, nämlich $getkey$, und erhalten:

(17) $\forall\, i : (0 \,..\, MSGMNI - 1) \mid msgque'(i) = DEFINED \bullet i \in \operatorname{dom} msg\_perm\_seq' \land i \in \operatorname{dom} msg\_perm\_key' \land (i + msg\_perm\_seq'(i) * MSGMNI) \in \operatorname{dom} getkey \land msg\_perm\_key'(i) = getkey(i + msg\_perm\_seq'(i) * MSGMNI)$ aus (15) und Text

(18) $\forall\, i : ID \mid i \in \operatorname{dom} getkey \bullet msgque'(i \bmod MSGMNI) = DEFINED$ aus (16) und Text

Wie nun offensichtlich ist, ist (17) herzuleiten aus (6), (10), (11) und (12). Genauso ist (18) herzuleiten aus (7), (10), (11) und (12).

Ergebnis

Die abstrakte Spezifikation $msgsndEACCES$ wird durch die konkrete Spezifikation $msgsndEACCES$ verfeinert.

P.8 Fehlerfall EINVAL

Dieser Fall tritt bei verschiedenen Möglichkeiten ein, stets ist es jedoch eine falsche Eingabe. Entweder wird eine falsche Message-Queue-ID eingegeben, oder der Typ einer Message ist nicht richtig, oder aber die Länge der Nachricht ist zu klein oder zu groß.

P.8.1 Abstrakt EINVAL

Auch hier erst einmal die abstrakte Sichtweise.

$$
\begin{array}{|l}
\hline
\textit{msgsndEINVAL} \\
\Delta \textit{MsgQStateSpace} \\
t? : \textit{TYPE} \\
b? : \textit{BYTES} \\
\textit{msgsz}? : \mathbb{N} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
f? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
i? \notin \operatorname{dom} \textit{msgq} \lor \qquad (1) \\
\quad t? \leq 0 \lor \qquad (2) \\
\quad \textit{msgsz}? < 0 \lor \qquad (3) \\
\quad \textit{msgsz}? > \textit{MSGMAX} \qquad (4) \\
r! = -1 \\
\textit{errno}' = \textit{EINVAL} \\
\hline
\end{array}
$$

Entweder wird hier eine falsche Message-Queue-ID vorgegeben (1), oder der Typ einer Nachricht ist nicht richtig (2), oder aber die Länge der Nachricht ist zu klein (3) oder zu groß (4).

P.8.2 Konkret EINVAL

Und nun die konkrete Sichtweise.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ CmsgsndEINVAL\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta CMsgQStateSpace \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (1) \\
t? : TYPE \\
b? : BYTES \\
msgsz? : \mathbb{N} \\
p? : PID \\
i? : ID \\
f? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
i? \notin \operatorname{dom} msgq \lor \\
\quad t? \leq 0 \lor \\
\quad msgsz? < 0 \lor \\
\quad msgsz? > MSGMAX \\
r! = -1 \\
errno' = EINVAL \\
\hline
\end{array}
$$

Wie man sieht unterscheidet sich das konkrete Schema vom abstraktem Schema nur in einem Punkt, es wird der konkrete $CMsgQStateSpace$ eingebunden (1).

Alle anderen Punkte sind von der Konkretisierung nicht betroffen.

P.8.3 Refinement

In diesem Fall muß ebenfalls für die Verifikation die *s*afety condition und die *l*iveness condition gezeigt werden.

Erste Bedingung (safety condition)

Nun ist wieder zu zeigen, daß man aus den Vorbedingungen des abstrakten $msgsndEINVAL$ und der Relation Rel das konkrete $CmsgsndEINVAL$ folgern kann.

$$(\mathrm{pre}\ msgsndEINVAL) \wedge Rel \vdash \mathrm{pre}\ CmsgsndEINVAL$$

Die Vorbedingungen für den abstrakten Fall und die Relation sehen folgendermaßen aus:

(1)	$i? \notin \mathrm{dom}\, msgq \vee$ $t? \leq 0 \vee$ $msgsz? < 0 \vee$ $msgsz? > MSGMAX$	aus $\mathrm{pre}\ msgsndEINVAL$
(2)	$\#\,\mathrm{dom}\, getkey \leq MSGMNI$	aus $\mathrm{pre}\ msgsndEINVAL$
(3)	$\mathrm{dom}\, msgq = \mathrm{dom}\, getkey$	aus $\mathrm{pre}\ msgsndEINVAL$
(4)	$\forall\, i1, i2 : ID \mid$ $i1 \in \mathrm{dom}\, msgq \wedge i2 \in \mathrm{dom}\, msgq \bullet$ $getkey(i1) = getkey(i2) \Rightarrow$ $i1 = i2 \vee getkey(i1) = IPC\_PRIVATE$	aus $\mathrm{pre}\ msgsndEINVAL$
(5)	$\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ $msgque(i) = DEFINED \bullet$ $i \in \mathrm{dom}\, msg\_perm\_seq \wedge$ $i \in \mathrm{dom}\, msg\_perm\_key \wedge$ $(i + msg\_perm\_seq(i) * MSGMNI) \in$ $\mathrm{dom}\, getkey \wedge$ $msg\_perm\_key(i) = getkey(i +$ $msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(6)	$\forall\, i : ID \mid i \in \mathrm{dom}\, getkey \bullet$ $msgque(i \bmod MSGMNI) = DEFINED$	aus Rel

Dagegen sind die Vorbedingungen des konkreten $CmsgsndEINVAL$ folgende:

(7)	$i? \notin \mathrm{dom}\, msgq \vee$ $t? \leq 0 \vee$ $msgsz? < 0 \vee$ $msgsz? > MSGMAX$	aus $\mathrm{pre}\ CmsgsndEINVAL$
(8)	$\forall\, id : (0 \mathrel{..} MSGMNI - 1) \bullet$ $msgque(id) = DEFINED \Leftrightarrow$ $id \in \mathrm{dom}\, msg\_perm\_key$	aus $\mathrm{pre}\ CmsgsndEINVAL$
(9)	$\forall\, id : (0 \mathrel{..} MSGMNI - 1) \bullet$ $msgque(id) = DEFINED \Leftrightarrow$ $id \in \mathrm{dom}\, msg\_perm\_seq$	aus $\mathrm{pre}\ CmsgsndEINVAL$

(10) $\forall\, i : ID \bullet i \in \operatorname{dom} msgq \Leftrightarrow$ aus pre $CmsgsndEINVAL$

$$(msgque(i \,\mathsf{mod}\, MSGMNI) = DEFINED \land msg\_perm\_seq(i \;\mathsf{mod}\; MSGMNI) * MSGMNI + i \;\mathsf{mod}\; MSGMNI = i)$$

(11) $\forall\, i1, i2 : ID \mid$ aus pre $CmsgsndEINVAL$

$$i1 \in \operatorname{dom} msgq \land i2 \in \operatorname{dom} msgq \bullet msg\_perm\_key(i1 \;\mathsf{mod}\; MSGMNI) = msg\_perm\_key(i2 \;\mathsf{mod}\; MSGMNI) \Rightarrow i1 = i2 \lor msg\_perm\_key(i1 \;\mathsf{mod}\; MSGMNI) = IPC\_PRIVATE$$

Nun ist folgende Vorbedingung einfach herzuleiten:
(7) aus (1)

Für die Vorbedingungen (8) bis (11) haben wir dieses im Kapitel P.5.3 schon bewiesen. Die für den Beweis dort verwendeten Vorbedingungen sind hier ebenfalls gegeben.

Zweite Bedingung (liveness condition)

Nun wenden wir uns wieder der liveness condition zu. Aus den Vorbedingungen von $msgsndEINVAL$, Rel und $CmsgsndEINVAL$ muß man einen abstrakten $MsgQStateSpace'$ folgern können.

$$\begin{array}{l}(\mathrm{pre}\ msgsndEINVAL) \land Rel \land CmsgsndEINVAL \vdash \\ \quad \exists\, MsgQStateSpace' \bullet (msgsndEINVAL \land Rel')\end{array}$$

Zuerst führen wir alle Prädikate der Hypothese an.

(1)	$i? \notin \mathrm{dom}\, msgq \lor$ $t? \leq 0 \lor$ $msgsz? < 0 \lor$ $msgsz? > MSGMAX$	aus $\mathrm{pre}\ msgsndEINVAL$
(2)	$\#\, \mathrm{dom}\, getkey \leq MSGMNI$	aus $\mathrm{pre}\ msgsndEINVAL$
(3)	$\mathrm{dom}\, msgq = \mathrm{dom}\, getkey$	aus $\mathrm{pre}\ msgsndEINVAL$
(4)	$\forall\, i1, i2 : ID \mid$ $i1 \in \mathrm{dom}\, msgq \land i2 \in \mathrm{dom}\, msgq \bullet$ $getkey(i1) = getkey(i2) \Rightarrow$ $i1 = i2 \lor getkey(i1) = IPC\_PRIVATE$	aus $\mathrm{pre}\ msgsndEINVAL$
(5)	$\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ $msgque(i) = DEFINED \bullet$ $i \in \mathrm{dom}\, msg\_perm\_seq \land$ $i \in \mathrm{dom}\, msg\_perm\_key \land$ $(i + msg\_perm\_seq(i) * MSGMNI) \in$ $\mathrm{dom}\, getkey \land$ $msg\_perm\_key(i) = getkey(i +$ $msg\_perm\_seq(i) * MSGMNI)$	aus Rel
(6)	$\forall\, i : ID \mid i \in \mathrm{dom}\, getkey \bullet$ $msgque(i \bmod MSGMNI) = DEFINED$	aus Rel
(7)	$r! = -1$	aus $CmsgsndEINVAL$
(8)	$errno' = EINVAL$	aus $CmsgsndEINVAL$

Da $msgque'$, msg_perm_key' und msg_perm_seq' im Schema nicht explizit verändert werden, gilt implizit:

(9)	$msgque' = msgque$	aus Text
(10)	$msg\_perm\_key' = msg\_perm\_key$	aus Text
(11)	$msg\_perm\_seq' = msg\_perm\_seq$	aus Text

Nun expandieren wir die Folgerung und beschränken uns hierbei wieder auf die beim Konkreten veränderten Teile

$\exists\, getkey' : ID \nrightarrow KEY \bullet$

(12) $r! = -1$ aus $msgsndEINVAL$

(13) $errno' = EINVAL$ aus $msgsndEINVAL$

(14) $\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus Rel'
$msgque'(i) = DEFINED \bullet$
$i \in \operatorname{dom} msg\_perm\_seq' \land$
$i \in \operatorname{dom} msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\operatorname{dom} getkey' \land$
$msg\_perm\_key'(i) = getkey'(i +$
$msg\_perm\_seq'(i) * MSGMNI)$

(15) $\forall\, i : ID \mid i \in \operatorname{dom} getkey' \bullet$ aus Rel'
$msgque'(i \bmod MSGMNI) = DEFINED$

Wenn wir für $getkey'$ $getkey$ einsetzen, erhalten wir:

(16) $\forall\, i : (0 \mathrel{..} MSGMNI - 1) \mid$ aus (14) und Text
$msgque'(i) = DEFINED \bullet$
$i \in \operatorname{dom} msg\_perm\_seq' \land$
$i \in \operatorname{dom} msg\_perm\_key' \land$
$(i + msg\_perm\_seq'(i) * MSGMNI) \in$
$\operatorname{dom} getkey \land$
$msg\_perm\_key'(i) = getkey(i +$
$msg\_perm\_seq'(i) * MSGMNI)$

(17) $\forall\, i : ID \mid i \in \operatorname{dom} getkey \bullet$ aus (15) und Text
$msgque'(i \bmod MSGMNI) = DEFINED$

Wie nun offensichtlich ist, ist (16) herzuleiten aus (5), (9), (10) und (11). Genauso ist (17) herzuleiten aus (6), (9), (10) und (11).

Ergebnis

Auch bei diesem Fehlerfall ist die konkrete Spezifikation eine Verfeinerung der abstrakten Spezifikation.

P.9 Fehler Gesamt

Die in den vorherigen Abschnitten beschriebenen Fehler sind keine Fehler in dem Sinne, daß etwas falsch gelaufen ist und das System nicht in Ordnung ist. Sondern es ist gemeint, daß Eingaben getätigt werden oder Systembedingungen vorhanden sind, die ein erfolgreiches Senden der Nachricht nicht ermöglichen.

Diese Fehlerfälle sollten alle zusammengefaßt werden. Entweder es tritt ein Erfolg ein oder ein Fehler. Die dritte Alternative, daß das System in einen unbestimmten Zustand übergeht, sollte ausgeschlossen werden, wie wir dies in durch die Spezifikation auch getan haben.

Einige Fehlerfälle können wir in Z jedoch nicht modellieren, da es uns zum Beispiel nicht möglich ist, einen zeitlichen Verlauf zu beschreiben. Somit sind die im System vorhandenen Fälle:

- EFAULT: Ein Pointer zeigt auf eine nicht existierende Adresse. Hierfür ist der Kernel zuständig.
- EIDRM: Die Message-Queue ist gelöscht worden, während der Prozess blockiert war. Dies können wir aufgrund der fehlenden zeitlichen Komponente von Z nicht spezifizieren.
- EINTR: Der Prozeß wartet auf eine Message-Queue und erhält einen Interrupt. Dies kann wegen der fehlenden Zeitkomponente in Z nicht dargestellt werden.
- ENOMEM: Das System hat nicht genügend Speicher um den gesamten Inhalt der Message-Queue zu speichern. Hierfür ist der Kernel zuständig.

in keinem der verschiedenen Fehlerfälle berücksichtigt. Trotzdem ist die Spezifikation so gehalten, daß sie konsistent ist.

P.9.1 Abstrakte Fehlerfälle

Hier sind sämtliche abstrakten Fehlerfälle vereinigt. Die Gesamtzahl der Fehler ergibt sich aus der Schemadisjunktion:

$$Msgsndfault \mathrel{\widehat=} msgsndEAGAIN \lor msgsndEACCES \lor msgsndEINVAL$$

P.9.2 Konkrete Fehlerfälle

Die Gesamtzahl der konkreten Fehler ergibt sich aus der Schemadisjunktion:

$$\begin{aligned} CMsgsndfault \mathrel{\widehat=} {} & CmsgsndEAGAIN \lor CmsgsndEACCES \\ & \lor CmsgsndEINVAL \end{aligned}$$

P.9.3 Refinement

Auch hier ist es für die Verifikation entscheidend zu beweisen, daß die konkreten Fehlerfälle eine Verfeinerung der abstrakten Fehlerfälle sind.

Bei einer Schemadisjunktion ist das konkrete eine Verfeinerung des abstrakten, wenn für jedes einzelne Disjunktionsglied die konkrete Form eine Verfeinerung der abstrakten Form ist.

$$\begin{aligned} & ((msgsndEAGAIN \vdash CmsgsndEAGAIN) \land \\ & (msgsndEACCES \vdash CmsgsndEACCES) \land \\ & (msgsndEINVAL \vdash CmsgsndEINVAL)) \\ & \Rightarrow Msgsndfault \vdash CMsgsndfault \end{aligned}$$

Dies haben wir in den vorherigen Abschnitten bewiesen, somit ist dies Aussage gegeben.

P.10 User-Interface

Für den Benutzer besteht die bisherige Aufteilung zwischen Fehlerfall und Erfolgsfall nicht. Er benutzt das System und erwartet eine Rückmeldung. Dafür besteht das User-Interface, bei dem der Benutzer lediglich die Message-Queue-Operation $msgsnd$ aufrufen kann.

P.10.1 Abstraktes User-Interface

Die abstrakte Version des User-Interfaces:

$$msgsnd \mathrel{\widehat{=}} Msgsnd \lor Msgsndfault$$

P.10.2 Konkretes User-Interface

Die konkrete Version des User-Interfaces:

$$Cmsgsnd \mathrel{\widehat{=}} CMsgsnd \lor CMsgsndfault$$

P.10.3 Refinement

Und wieder ist es für die Verifikation entscheidend auch dieses zu beweisen. Aber es sieht genauso gut aus wie im vorherigen Kapitel.

Bei einer Schemadisjunktion ist das Konkrete eine Verfeinerung des Abstrakten, wenn für jedes einzelne Disjunktionsglied die konkrete Form eine Verfeinerung der abstrakten Form ist.

$$\begin{array}{l} ((Msgsnd \vdash CMsgsnd) \land \\ (Msgsndfault \vdash CMsgsndfault)) \\ \Rightarrow msgsnd \vdash Cmsgsnd \end{array}$$

Auch dieses haben wir in den vorherigen Abschnitten schon bewiesen, somit ist die Verifikation schon durchgeführt. Das abstrakte $msgsnd$ wird durch das konkrete $Cmsgsnd$ verfeinert.

P.11 Ergebnis

Zuerst einmal das wichtigste Ergebnis vorweg: wir haben während unserer Verifikation keinen Fehler in den Message-Queues finden können. Daher sollte man davon ausgehen, daß sie korrekt funktionieren und keine ungewöhnlichen Effekte auftreten.

Allerdings war der Weg bis zu dieser Erkenntnis sehr lang. Den größten Teil dieses Weges kann man auch gar nicht mehr erkennen. Denn auch bei einer noch so guten Vorbereitung und gründlichen Recherche für die Spezifikation mußten wir dann erkennen, daß wir einige Irrwege eingeschlagen hatten.

So mußten wir zwischenzeitlich die gesamte Spezifikation überarbeiten, da wir uns in die Idee verrannt hatten, daß Message-Queues grundsätzlich über ihren KEY angesprochen würden. Nur leider ist es so implementiert, daß eine Message-Queue zwar einen systemweiten eindeutigen KEY besitzt, sie aber trotzdem über die ID angesprochen wird.

Sämtliche kleineren Irrungen und Wirrungen hier zu notieren wäre sehr aufwendig und lang, es sollte genügen hier zu erwähnen, daß es eine Menge davon gab.

Wir hatten Z als die am meisten geeignete Spezifikationssprache für die Darstellung von Message-Queues angesehen, da es darum ging, interne Datenstrukturen darzustellen. Da unser Beispiel jedoch aus der Realität gegriffen ist, ist es wesentlich komplexer. Somit stießen wir an der einen oder anderen Stelle an die Grenzen von Z. So ist es, wie schon mehrfach im Text erwähnt, nicht möglich einen zeitlichen Verlauf darzustellen.

Nachdem wir solche Probleme aus dem Weg geräumt hatten wandten wir uns mehr dem verifizieren zu. Auch hier mußten wir uns erst in die Materie einarbeiten und versuchen die konstruierten Spielbeispiele aus den Fachbüchern auf ein konkretes Beispiel aus der Realität zu übertragen. Es sah alles wesentlich weniger elegant aus und war auch nicht so einfach zu durchschauen wie bei den Beispielen aus den Büchern, aber es funktioniert. Jedenfalls nach einigen Fehlversuchen.

Durch die Verwendung der Spezifikationssprache Z wird die Verifikation auf die formale mathematische Verifikation eingeschränkt. Das Data Refinement ist eine sehr elegante Methode, bei Beispielen aus der Realität werden die Beweise jedoch sehr schnell lang und unübersichtlich. Auch ist es häufig sehr schwer den richtigen Beweisweg zu finden, man muß erst einmal auf die richtige Idee kommen, dann geht es jedoch meist recht zügig.

Zum Schluß hin trat bei uns das übliche Problem bei solchen Projekten auf: die Zeit lief uns davon. So sind wir leider nicht mehr dazu gekommen die Message-Queue Operation $msgsnd$ vollständig formal zu verifizieren. Dies gelang uns nur für einen Teil, dieser ist allerdings nach unseren Erkenntnissen vollständig korrekt. Für den restlichen Teil sind wir nach unserer Beschäftigung mit dem Source-Code und dem Vergleich mit unserer Spezifikation ebenfalls der Ansicht, dass korrekt implementiert wurde.

Insgesamt war es sehr interessant, mit Z zu arbeiten und festzustellen, wie umfangreich eine formale Verifikation wird. Auch wenn es an einigen Stellen Probleme mit Z gab, so war dies doch die geeignete Sprache für unsere Spezifikation und Verifikation.

Jedenfalls, um den enormen Aufwand einer Verifikation zu rechtfertigen, bedarf es schon so wichtiger Argumente wie das sichere Funktionieren eines Betriebsystems.

Anhang Q. SMP: CSP–Spezifikation

```
-- Konstante

--Die Anzahl der laufenden Clients pro CPU.run_clients_cpu=1
byte_to_read_max=50
--Die gesamte Anzahl der laufenden Clients
run_clients=run_clients_cpu*2
--Die maximale Anzahl der wartenden Clients
wait_clients=run_clients*2
--Die Anzahl der User-Prozesse
user_count=2
--Der maximal moegleiche Key von Clients-MSQ
max_key=run_clients+wait_clients


-- Datatypes

datatype type_of_read = asynchron | synchron
datatype byte_of_data_datatype = data | null

-- Channels vom implementierenden System

-- Channel trigger_client wird fuer die Synchronisation
-- zwischen dem User-Prozess und dem Client verwendet.

channel something,sys_call,trigger_client,
channel copy_data_segment_to_user
channel start_read_from_device,end_read_from_device

-- Channel fuer den DATA_SEGMENT-Prozess(Array)

channel rd_ds,wr_ds:{0..byte_to_read_max}.
                    byte_of_data_datatype

-- Channel fuer BYTE_AMOUNT_GENERATOR.

channel rd_bag,wr_bag:{0..byte_to_read_max}

-- Ueber den Channel request_service erhaelt der Client
-- vom Serverdie Nachricht, ob er anfangen kann oder
-- warten muss.

channel request_service:{1..max_key}.{true,false}

-- Ueber die Channels order_msq und end_client schickt
-- der Client an den Server seinen msq_key.Was im ersten
-- Fall bedeutet, dass er gestartet ist und mitseinem
-- Auftrag  beginnen moechte und im zweiten Fall,dass der
```

```
-- Auftrag erledigt ist und der Client sich beendet hat.
-- Ueber den Channel get_msq_key erhaelt der Client seinen
-- msq_key, der spaeter als seine id verwendet wird.

channel order_msq,end_client_msq,get_msq_key:{1..max_key}

-- Diese Channels werden fuer die Prozesse (Variablen)
-- CLIENT_COUNT1, CLIENT_COUNT2, die die Anzahl der
-- laufenden pro CPU Clients speichern, verwendet.

channel out_client_count1:{0..(run_clients-1)}
channel out_client_count2:{0..(run_clients-1)}
channel inc_client_count1,inc_client_count2
channel dec_client_count1,dec_client_count2

-- Diese Channels werden fuer die Run- und Waitqueues
-- verwendet.

channel rd_select,wr_select:{0..max_key}
channel rd_rq, wr_rq:{0..(run_clients-1)}.
                    {0..max_key}.{0..2}
channel rd_wq, wr_wq:{0..(wait_clients-1)}.
                    {0..max_key}

-- Channels fuer die Daten.

channel byte_of_data,shm:byte_of_data_datatype

-- Channels fuer die abstrakte Prozesse:

channel start_client,end_client,read,start_read_device,
channel end_read_device,byte_to_user,last_byte_to_user

--CSP-Prozesse:

-- Variablen in welchen die Anzahl der laufenden pro
-- CPU Clients  gespeichert wird.

CLIENT_COUNT1(n) = (n > 0 ) & dec_client_count1 ->
                   CLIENT_COUNT1(n-1)
                   []
                   (n < (run_clients_cpu-1)) &
                   inc_client_count1 ->
                   CLIENT_COUNT1(n+1)
                   []
                   out_client_count1!n ->
                   CLIENT_COUNT1(n)

CLIENT_COUNT2(n) = (n > 0 ) & dec_client_count2 ->
                   CLIENT_COUNT2(n-1)
                   []
                   (n < (run_clients_cpu-1)) &
                   inc_client_count2 ->
                   CLIENT_COUNT2(n+1)
```

```
                  []
                  out_client_count2!n ->
                  CLIENT_COUNT2(n)

-- User-Prozesse.

U(i) =  something -> U(i)
        |~|
        sys_call -> trigger_client -> U(i)


USER = (||| i:{0..(user_count-1)} @ U(i))

-- INIT_CLIENT holt den msq_key fuer den Client und
-- startet ihn asynchron oder synchron

INIT_CLIENT = trigger_client -> get_msq_key?client_key ->
              rd_bag?amount -> (
              CLIENT(client_key,amount,type_of_read.asynchron)
              |~|
              CLIENT(client_key,amount,type_of_read.synchron))

-- BYTE_AMOUNT_GENERATOR erzeugt zufaellig die Anzahl
-- zu lesenden Bytes.

BYTE_AMOUNT_GENERATOR = (|~| i:{0..(byte_to_read_max-1)} @
                        B_A_GENERATOR(i))

B_A_GENERATOR(x) = rd_bag!x -> B_A_GENERATOR(x)
                   []
                   wr_bag?y -> B_A_GENERATOR(y)


-- Server-Prozess besteht aus zwei Zweigen.
-- 1) Wenn der msq_key von irgendeinem Client ankommt,
-- dann holt er die Anzahl bereits laufender Clients
-- pro CPU. Wenn beide gleich 0 sind (laeuft momentan
-- kein Client), dann wird der Prozess READ_FROM_DEVICE
-- gestartet. Danach wird geprueft, ob die Runqueue voll
-- ist. Wenn ja, dann wird der Client in die Waitqueue
-- eingefuegt, bzw, wenn die auch voll ist, wird die
-- Anfrage verworfen. Sonst wirdder Client in die Runqueue
-- eingefuegt und ihm wird mitgeteilt,dasser mit Lesen
-- beginnen darf. Die Clients werden gleichmaessig auf
-- Cpu's verteilt.
-- 2) Wenn ein Client sich beendet, dann wird er aus der
-- Runqueue geloescht. Danach wird versucht einen neuen
-- Client aus der Waitqueue zu selektieren. Im Erfolgsfall
-- wird der selektierte Client in die Runqueue eingefuegt
-- und kann laufen. Sonst wird der Prozess READ_FROM_DEVICE
-- beendet, da es keine Clients mehr gibt.

SERVER = (order_msq?client_key -> out_client_count1?x ->
         out_client_count2?y ->
```

```
          (if ((x==0) and (y==0)) then
             start_read_from_device
          else
             something) ->
          if ((x+y) ==(run_clients-1)) then (
            ADD_TO_WAITQUEUE(0, client_key);
            request_service!client_key.false ->
            SERVER)
          else (
               if (x <= y) then (
                 ADD_TO_RUNQUEUE(0,client_key,1);
                 inc_client_count1 ->
                 request_service!client_key.true ->
                 SERVER)
               else (
                    ADD_TO_RUNQUEUE(0,client_key,2);
                    inc_client_count2->
                    request_service!client_key.true ->
                    SERVER)))
          []
          (end_client_msq?end_client_key ->
          DEL_FROM_RUNQUEUE(0,end_client_key);
          SELECT_FROM_WAITQUEUE(0); rd_select?client_key ->
          out_client_count1?x -> out_client_count2?y ->
          if ( client_key != 0 ) then (
             if (x <= y) then (
                ADD_TO_RUNQUEUE(0,client_key,1);
                inc_client_count1 ->
                request_service!client_key.true -> SERVER)
             else (
                ADD_TO_RUNQUEUE(0,client_key,2);
                inc_client_count2 ->
                request_service!client_key.true -> SERVER))
          else (
              if((x==0) and (y==0)) then (
                  end_read_from_device -> SERVER)
              else
                 SERVER))

-- Erzeugt msq_keys fuer die Clients.

MSQ_QUEUES_KEYS(n) = (n != max_key  ) & get_msq_key!n ->
                     MSQ_QUEUES_KEYS(n + 1)
                     []
                     (n == max_key ) & MSQ_QUEUES_KEYS(1)


-- CLIENT hat folgende Parameter:
-- client_key- der msq_key (id)
-- amount- die Anzahl zu lesenden bytes
-- how- synchron oder asynchron

-- Am Anfang schickt der Client seinen msq_key
-- an den Server und wartet bis er mit dem Lesen
```

```
-- beginnen kann.Dann liest er und beendet sich.

CLIENT(client_key,amount,how) =
                  start_client -> order_msq!client_key ->
                  request_service?key.x ->
                  if (x == true and key == client_key) then
                    (read -> READ(amount,amount,how);
                    END_CLIENT(client_key))
                  else
                    CLIENT(client_key,amount,how)

-- READ Liest aus dem Shared memory Bereich in den
-- Speicherbereich des User-Prozesses. Das Lesen kann
-- asynchron oder synchron erfolgen. Im zweiten Fall
-- werden die Daten in einem DATA_SEGMENT-Array
-- gespeichert. Und dann alle auf ein mal beim Userkopiert
-- (COPY_TO_USER).Da wir nicht weiter spezifiziert haben,
-- wie die Daten benutzt werden,verwenden wir nur einen
-- einzigen event copy_data_segment_to_user,um das zu
-- beschreiben.Im unteren Teil vom diesen Prozess wird
-- geprueft ,ob das letzte Feld im DATA_SEGMENT-Array
-- beschrieben ist.(Fuer die Verifikation)

READ(amount,amount_copy,how) =
                   ((amount == 0) and (how == asynchron)) &
                   SKIP
                   []
                   ((amount != 0) and (how == asynchron)) &
                   shm?d -> write_byte_to_user!d ->
                   byte_to_user -> READ(amount-1,how)
                   []
                   ((amount == 0) and (how == synchron)) &
                   COPY_TO_USER(amount_copy);SKIP
                   []
                   ((amount != 0) and (how == synchron)) &
                     shm?d -> wr_ds.amount!d ->
                     READ(amount-1,how)

DATA_SEGMENT =
         (||| i:{0..(byte_to_read_max-1)} @ D_SEGMENT(i,null))

D_SEGMENT(i,x) = rd_ds.i!x -> D_SEGMENT(i,x)
                 []
                 wr_ds.i?y -> D_SEGMENT(i,y)

COPY_TO_USER(amount) = copy_data_segment_to_user ->
               rd_ds.(amount-1)?x ->
               if(x == data) then (
                  last_byte_to_user -> SKIP)
               else
                  SKIP

-- Beenden eines Clients.
```

```
END_CLIENT(client_key) = end_client_msq!client_key ->
                         end_client -> SKIP

-- Run- und Waitqueues.

RUN_QUEUE = (||| i:{0..(run_clients-1)} @ R_QUEUE(i,0,0))

R_QUEUE(i,key,cpu) = rd_rq.i!key.cpu -> R_QUEUE(i,key,cpu)
                     []
                     wr_rq.i?new_key.new_cpu ->
                     R_QUEUE(i,new_key,new_cpu)

ADD_TO_RUNQUEUE(n,client_key,cpu) =
                     (n != run_clients) & rd_rq.n?y.x ->
                      if (y == 0) then
                        (wr_rq.n!client_key.cpu -> SKIP)
                      else
                        ADD_TO_RUNQUEUE(n + 1,
                                        client_key,cpu)
                     []
                     (n == run_clients) & SKIP

DEL_FROM_RUNQUEUE(n, client_key) =
                     (n != run_clients) & rd_rq.n?y.x ->
                      if (client_key == y) then
                        (wr_rq.n!0.0 ->
                         if (x==1) then
                           (dec_client_count1 -> SKIP)
                         else
                           (dec_client_count2 -> SKIP))
                      else
                        DEL_FROM_RUNQUEUE(n + 1,
                                          client_key)
                     []
                     (n == run_clients) & SKIP


WAIT_QUEUE = (||| i:{0..(wait_clients-1)} @ W_QUEUE(i,0))

W_QUEUE(i, x) = rd_wq.i!x -> W_QUEUE(i, x)
                   []
                   wr_wq.i?y -> W_QUEUE(i, y)

ADD_TO_WAITQUEUE(n, client_key) =
                 (n != wait_clients) & rd_wq.n?y ->
                  if (y == 0) then
                    (wr_wq.n!client_key -> SKIP)
                  else
                    ADD_TO_WAITQUEUE(n + 1, client_key)
                 []
                 (n == wait_clients) & SKIP

DEL_FROM_WAITQUEUE(n, client_key) =
                 (n != wait_clients) & rd_wq.n?y ->
```

```
                 if (client_key == y) then
                   (wr_wq.n!0 -> SKIP)
                 else
                   DEL_FROM_WAITQUEUE(n + 1,client_key)
                []
                (n == wait_clients) & SKIP


SELECT_FROM_WAITQUEUE(n) =(
                (n != wait_clients) & rd_wq.n?client_key ->
                 if (client_key == 0) then
                   SELECT_FROM_WAITQUEUE(n + 1)
                 else
                   (wr_select!client_key ->
                    wr_wq.n!0 -> SKIP))
                []
                ((n == wait_clients) & wr_select!0 -> SKIP)

SELECTED_CLIENT=S_C(0)

S_C(n) = rd_select!n -> S_C(n)
         []
         wr_select?x -> S_C(x)

-- Startet READ_FROM_DEVICE-Prozess.

START_READ_FROM_DEVICE = start_read_from_device ->
               start_read_device -> READ_FROM_DEVICE

READ_FROM_DEVICE = byte_of_data?d -> shm!d ->
                   READ_FROM_DEVICE
                   []
                   end_read_from_device ->
                   end_read_device -> SKIP

-- DEVICE_Prozess.

DEVICE = byte_of_data!data -> DEVICE

-- Abstrakte Prozesse

ABS_CLIENT = ||| i:{1..run_clients}@ ABS_CL(i)

ABS_CL(i) = start_client -> read -> end_client -> ABS_CL(i)

ABS_READ_FROM_DEVICE = start_read_device ->
                 end_read_device -> ABS_READ_FROM_DEVICE

ABS_USER = byte_to_user -> ABS_USER
           |~|
           last_byte_to_user -> ABS_USER

-- Das konkrete System.
```

```
Sys=((((USER [|{|trigger_client|}|] (INIT_CLIENT
        [|{|rd_bag,wr_bag|}|]
        BYTE_AMOUNT_GENERATOR))[|{|get_msq_key|}|]
        MSQ_QUEUES_KEYS(1))
        ||| CLIENT_COUNT1(0) ||| CLIENT_COUNT2(0) |||
        RUN_QUEUE ||| WAIT_QUEUE |||
        SELECTED_CLIENT ||| DATA_SEGMENT )
        [|{|request_service,order_msq,end_client_msq,
         inc_client_count1,inc_client_count2,
         dec_client_count1,dec_client_count2,
         out_client_count1,out_client_count2,
         rd_select,wr_select,rd_rq,wr_rq,rd_wq,
         wr_wq,rd_ds,wr_ds|}|]
        SERVER)
        [|{|shm,start_read_from_device,
         end_read_from_device|}|]
        (DEVICE [|{|byte_of_data|}|]
        START_READ_FROM_DEVICE)


CHAOS_CLIENT = CHAOS({|something,sys_call,
                 request_service,
                 order_msq,trigger_client,get_msq_key,
                 inc_client_count1,dec_client_count1,
                 out_client_count1,inc_client_count2,
                 dec_client_count2,out_client_count2,
                 rd_select,wr_select,end_client_msq,
                 rd_rq,wr_rq,rd_wq,wr_wq,
                 byte_of_data,shm,
                 copy_data_segment_to_user,
                 start_read_from_device,
                 end_read_from_device,
                 start_read_device,end_read_device,
                 rd_ds,wr_ds,byte_to_user,
                 last_byte_to_user,rd_bag,wr_bag|})

CHAOS_READ_FROM_DEVICE = CHAOS({|something,sys_call,
                 order_msq,trigger_client, get_msq_key,
                 inc_client_count1,dec_client_count1,
                 out_client_count1,inc_client_count2,
                 dec_client_count2,out_client_count2,
                 rd_select,wr_select,end_client_msq,
                 rd_rq,wr_rq,rd_wq,wr_wq,byte_of_data,
                 shm,copy_data_segment_to_user,
                 start_read_from_device,
                 end_read_from_device,
                 start_client,end_client,read,rd_ds,
                 wr_ds,byte_to_user,last_byte_to_user,
                 rd_bag,wr_bag,request_service|})

CHAOS_USER = CHAOS({|something,sys_call,
                 request_service,
                 trigger_client,get_msq_key,
                 dec_client_count1,out_client_count1,
```

```
                 inc_client_count2,dec_client_count2,
                 out_client_count2,rd_select,
                 wr_select,end_client_msq,
                 rd_rq,wr_rq,rd_wq,
                 wr_wq,byte_of_data,shm,
                 copy_data_segment_to_user,
                 start_read_from_device,
                 end_read_from_device,
                 start_client,end_client,
                 read,rd_ds,wr_ds,rd_bag,
                 start_read_device,end_read_device,
                 wr_bag,order_msq,inc_client_count1|})

-- Verifikation Bedingungen.

Q1 = ABS_CLIENT ||| CHAOS_CLIENT

Q2 = ABS_READ_FROM_DEVICE ||| CHAOS_READ_FROM_DEVICE

Q3 = ABS_USER ||| CHAOS_USER

assert Q1[T=Sys

assert Q2[T=Sys

assert Q3[T=Sys
```

Literaturverzeichnis

[All] Allman, Eric: *SENDMAIL – An Internetwork Mail Router*. University of California, Berkeley, Mammoth Project

[AO91] Apt, Krzysztof R. ; Olderog, Ernst-Rüdiger: *Verification of Sequential and Concurrent Programs*. Springer Verlag, 1991. – ISBN 0–387–97532–2 ; 3–540–97532–2

[BBD 97] Beck ; Böhme ; Dziadzka [u. a.] : *Linux-Kernel-Programmierung*. 4., aktualisierte und erweiterte Auflage. Addison-Wesley, 1997. – ISBN 3–8273–1144–6

[CAR94] Costales ; Allman ; Rickert: *sendmail*. O'Reilly & Associates, Inc., September 1994. – ISBN 1–56592–056–2

[Com90] Commission, International E.: International Standard IEC 1025: Fault Tree Analysis (FTA) / International Electrotechnical Commission, Genf. 1990. – Forschungsbericht

[Cri94] Crispin, M.: RFC 1730: Internet Message Access Protocol – Version 4 / University of Washington. 1994. – Forschungsbericht

[Cro77] Crocker, David H.: RFC 733: Standard For The Format Of ARPA Network Text Messages / The Rand Corporation. 1977. – Forschungsbericht

[Cro82] Crocker, David H.: RFC 822: Standard For The Format Of Arpa Internet Text Messages / University of Delaware, Newark. 1982. – Forschungsbericht

[Dig96] Digital Equipment Corporation. Using ACLs. http://www.unix.digital.com/faqs/publications/base_doc /DOCUMENTATION/HTML/AA-Q0R2D-TET1_html /sec.c27.html#no_id_22. 1996

[Hew96a] Hewlett-Packard Company: *access(2) HP-UX 10.20 manual page*. 1996

[Hew96b] Hewlett-Packard Company: *setacl(2) HP-UX 10.20 manual page*. 1996

[Hew96c] Hewlett-Packard Company: *setaclentry(3c) HP-UX 10.20 manual page*. 1996

[Hor86] Horton, Mark R.: RFC 976: UUCP Mail Interchange Format Standard / Bell Laboratories. 1986. – Forschungsbericht

[HW96] Hein, Mathias ; Weihrich, Thomas: *UNIX-Rechnernetze in Theorie und Praxis*. International THomson Publishing, 1996

[JS99] Jarré, Sönke ; Scholz, Raymond. Spezifikation des Shellskripts `cpyp` in CSP und anschließende Verifikation mit FDR2. http://aerobee.informatik.uni-bremen.de/~rscholz/ps/cpyp.ps. 1999

[Kir95] KIRCH, Olaf: *Linux Network Administrator's Guide.* O'Reilly & Associates, 1995

[Lap92] LAPRIE, J.C. et a.: *Dependability: Basic Concepts and Terminology.* Springer Verlag, Berlin, Heidelberg, New York, 1992

[Lot87] LOTTOR, M.: RFC 1033: Domain Administrators Operations Guide / SRI Interantional. 1987. – Forschungsbericht

[Moc87a] MOCKAPETRIS, P.: RFC 1034: Domain Names – Concepts and Facilities / ISI. 1987. – Forschungsbericht

[Moc87b] MOCKAPETRIS, P.: RFC 1035: Domain Names – Implentation and Specification / ISI. 1987. – Forschungsbericht

[Moh97] MOHR, James: *Linux User's Resource.* Prentice Hall, 1997

[MR96] MYERS, J. ; ROSE, M.: RFC 1939: Post Office Protocol – Version 3 / Carnegie Mellon; Dover Beach Consulting, Inc. 1996. – Forschungsbericht

[Now89] NOWICKI, B.: RFC 1094: NFS: Network File System (Version 2) / Sun Microsystems, Inc. 1989. – Forschungsbericht

[Pat86] PATRIDGE, Craig: RFC 974: Mail Routing And The Domain System / CSNET CIC BBN Laboratories Inc. 1986. – Forschungsbericht

[PMG99] PETROU, David ; MILFORD, John W. ; GIBSON, Garth A.: Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. (1999), S. 14

[Pos82] POSTEL, Jonathan B.: RFC 821: Simple Mail Transfer Protocol / University of Southern California. 1982. – Forschungsbericht

[Sec97a] SECURITY WORKING GROUP: Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API), Amendment e: Protection, Audit and Control Interfaces [C Language], Draft 17 / IEEE. 345 East 47th Street, New York, NY 10017, USA, Oktober 1997 (1003.1e Draft 17). – Draft Standard

[Sec97b] SECURITY WORKING GROUP: Portable Operating System Interface (POSIX), Part 2: Shell and Utilities, Amendment c: Protection and Control Interfaces, Draft 17 / IEEE. 345 East 47th Street, New York, NY 10017, USA, Oktober 1997 (1003.2c Draft 17). – Draft Standard

[Spi92] SPIVEY, J. M.: *The Z Notation: A Reference Manual.* 2nd. Prentice Hall International Series in Computer Science, 1992. – Dieses Buch wird nicht mehr aufgelegt. Es ist aber zur Zeit unter `http://spivey.oriel.ox.ac.uk/~mike/zrm/` als Postscript verfügbar. – ISBN 013–978529–9

[Sto96] STOREY, Neil: *Safety-Critical Computer Systems.* Addison-Wesley, 1996

[Sun94a] Sun Microsystems, Inc.: *aclcheck(3) Solaris manual page.* 1994

[Sun94b] Sun Microsystems, Inc.: *aclsort(3) Solaris manual page.* 1994

[Sun94c] Sun Microsystems, Inc.: *acltomode(3) Solaris manual page.* 1994

[Sun94d] Sun Microsystems, Inc.: *acltopbits(3) Solaris manual page.* 1994

[Sun94e] Sun Microsystems, Inc.: *acltotext(3) Solaris manual page.* 1994

[Sun94f] Sun Microsystems, Inc.: *getfacl(1) Solaris manual page.* 1994

[Sun96] Sun Microsystems, Inc.: *acl(2) Solaris manual page.* 1996

[Sun97] Sun Microsystems, Inc.: *setfacl(1) Solaris manual page.* 1997

[Wal95] WALDSPURGER, Carl A.: Lottery and Stride Scheduling: Flexible Proportional–Share Resource Management. (1995)

[WD96] WOODCOCK, Jim ; DAVIES, Jim: *Using Z - Specification, Refinement, and Proof*. London : Prentice Hall International Series in Computer Science, 1996. – ISBN 0–13–948472–8

[WW94] WALDSPURGER, Carl A. ; WEIHL, William E.: Lottery Scheduling: Flexible Proportional–Share Resource Management. (1994)

Index

Index