Spezifikation des Shellskripts `CPYP` in CSP und anschließende Verifikation mit FDR2

Revision : 1.17

Kurs:	**Korrekte Software durch formale Methoden**
VAK:	3-785
Dozent:	Christoph Lüth

Verfasser: **Sönke Jarré** (soenke@informatik.uni-bremen.de)
Raymond Scholz (rscholz@informatik.uni-bremen.de)

Universität Bremen
Fachbereich 3 Informatik
WS 1998/99

Bremen, 27. April 1999

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung entstand im Rahmen der Vorlesung „Korrekte Software durch formale Methoden“, die im Wintersemester 1998/1999 an der Universität Bremen von Christoph Lüth gehalten wurde. Der in der Vorlesung erarbeitete Stoff soll an dieser Stelle angewendet und eine Verifikation eines konkreten Stücks Software durchgeführt werden.

Da wir uns zeitgleich im studentischen Projekt LiVE![1] mit dem Aufbau eines fehlertoleranten Mailservers beschäftigen, erschien es sinnvoll, einen Teil der dort entwickelten und somit schon vorhandenen Software als Verifikationsobjekt zu benutzen. Eine Verifikation ist deswegen sinnvoll, weil es sich hierbei um eine sicherheitskritische Anwendung handelt, wie später noch zu sehen ist.

Im Rahmen der Vorlesung haben wir uns hauptsächlich damit auseinandergesetzt, ein bestimmtes Problem zu lösen, das Problem zu formalisieren, die Lösung zu spezifizieren und anschließend diese mit Hilfe von formalen Methoden in ein Programm zu überführen. In der Praxis sieht es hingegen oft so aus, daß ein Stück Software schon vorhanden ist und dieses im Nachhinein verifiziert werden soll. Dies ist auch hier der Fall. Es existiert ein Stück Software (`CPYP`) und soll versucht werden, dieses auf Korrektheit in bezug auf bestimmte Bedingungen zu überprüfen.

Ein wesentlicher Nachteil der Programmverifikation mittels algebraischer Methoden ist, daß eine direkte Verifikation des Programmcodes nur schwer möglich ist. Dies ist in der Komplexität der verwendeten Sprachen (z.B. C, oder in diesem Fall die Skriptsprache der GNU Bash [2] in Verbindung mit AWK [3]) begründet. Um eine Verifikation von in diesen Sprachen geschriebenen Programmen zu erreichen, müßte zunächst die Semantik aller Sprachkonstrukte formalisiert werden, was gewöhnlich zum Scheitern verurteilt ist.

Deshalb haben wir uns entschieden, die Spezifikation in der Sprache CSP [1] zu erstellen, da es mit ihr möglich ist, die Spezifikation eines Programms nahe an der Implementierung zu halten und dabei die reichhaltige Semantik, die vor allem nebenläufige, kommunizierende Prozesse berücksichtigt, zu benutzen. Auch ist mit FDR2 [4] eine sehr gute Toolunterstützung für automatisches Beweisen durch Model–Checking gegeben.

Die Vorgehensweise ist dabei wie folgt: Indem man die Sicherheitsbedingungen von `CPYP` zugrunde legt, erzeugt man ein abstraktes System, das den un-

[1] Linux Verification Enterprise (WS 97/98 - SS 99)

ter Sicherheitsaspekten möglichen, d.h. erlaubten Ablauf modelliert. Danach wird versucht, ein konkretes System zu erzeugen, das eine direkte Umsetzung des Skript–Codes in die Spezifikationssprache darstellt. Dabei müssen gewisse Abstraktionen vorgenommen werden, die jedoch an der entsprechenden Stelle stets begründet werden. Nun können beide Systeme in CSP umgesetzt werden und diese mit Hilfe des Tools FDR2 in einem Trace–Refinement getestet werden. Hierbei kann man erkennen, ob es noch eine Abfolge von Ereignissen gibt, die nicht den Sicherheitsbedingungen entsprechen. Ebenso kann überprüft werden, ob im System noch Deadlocks vorhanden sind, d.h. daß im Gesamtsystem keine Kommunikation aufgrund gegenseitig aufeinander wartender Prozesse stattfindet.

Ist dieses abgeschlossen, so ist ein sicheres Funktionieren des Skripts gewährleistet, sofern die Spezifikation eine gültige Formalisierung des Skripts `CPYP` darstellt.

2 Das Skript CPYP

2.1 Funktionsweise

Das Shellskript `CPYP` („copy YP“) wird dazu verwendet, um bestimmte, systemweit zentral verwaltete Daten auf den Mailserver zu übertragen und danach diese Dateien für die Einhaltung bzw. Erfüllung der Security– und Safety–Richtlinien des Fachbereichsnetzes [5] zu modifizieren. Der ursprüngliche Zweck des Skripts, nämlich die Erhöhung der Performance des Mailservers und des Fachbereichsnetzes durch die Vermeidung unnötiger Netzzugriffe, ist ebenfalls ein wichtiger Aspekt, der sich jedoch nicht in der Spezifikation und Verifikation niederschlägt.

In größeren Netzen ist es üblich, daß die Verwaltung bestimmter Daten zentral auf einem darauf spezialisierten Rechner erfolgt. Diese Daten sind z.B. die Benutzerdatenbank, die Verwaltung von Benutzergruppen, Informationen über den Zugriff auf systemweit verfügbare Speichermedien (Benutzer–Homeverzeichnisse), die Auflösung von Rechnernamen in IP–Adressen etc. Traditionell werden diese Daten dezentral (d.h. auf allen Rechnern) abgelegt, was jedoch den Verwaltungsaufwand beträchtlich erhöht. Der ursprüngliche Namen für diesen netzweiten Dienst – YP[2] – ist inzwischen von NIS[3] offiziell abgelöst worden, wird jedoch im alltäglichen Gebrauch noch immer verwendet.

Allen Rechnern (Clients) im Netz eines YP–Servers ist es unter der Angabe der YP–Domain (eine Art „Verwaltungshoheit“) möglich, mittels eines Kommandoaufrufs („ypcat“) die von YP–Server verwalteten Daten abzufragen. Dadurch ist es z.B. möglich, daß sich ein Benutzer auf einem beliebigen Client des Netzes einloggen kann, da der Client Identifikation (Benutzername) und Authentifikation (Paßwort) des Benutzers durch eine Anfrage an den YP–Server überprüfen kann und den Benutzer bei Richtigkeit dieser Angaben sich einloggen läßt. Danach hat der Benutzer auch Zugriff auf sein Homeverzeichnis, welches nach einer weiteren Anfrage an den YP–Server auf dem Client gemountet werden kann.

Das vorangegangene Beispiel verdeutlicht, daß schon für den relativ einfachen Vorgang des Einloggens eines Benutzers viele Anfragen an den YP–Server erforderlich sind, die sowohl das Netz als auch den YP–Server selbst belasten. Sind nun für einen bestimmten Vorgang die erforderlichen YP–

[2] YP – Yellow Pages
[3] NIS – Network Information Service

Anfragen bekannt, so ließen sich diese vermeiden, wenn stattdessen traditionell auf lokale Dateien zugegriffen würde, die ein Abbild der Daten des YP–Servers darstellen. Genau diesen Ansatz verfolgt das Skript `CPYP`, indem es die Benutzerdatenbank („passwd“), die Gruppendatenbank („group“) und die Datenbank für den Speicherort der Homeverzeichnisse („auto.home“) in regelmäßigen Abständen (z.Z. alle sechs Stunden) vom YP–Server auf den Mailserver überträgt. Das Skript selbst läuft dabei auf dem Mailserver, der sich dabei wie ein Client des YP–Servers verhält.

Die Auswahl dieser Daten ergibt sich aus der Funktionsweise der zu erbringenden Dienste des Mailservers. Der Mailserver kann Mail von Benutzern oder anderen Rechnern annehmen und weiterleiten[4], Mail zum Abruf bereitstellen[5] oder anderen Rechnern das Mounten des Mail–Spool–Verzeichnisses (hier liegt die Mail für die Benutzer) via NFS[6] erlauben. Für diese Dienste ist das Vorhandensein der Benutzerdatenbank erforderlich. Darüberhinaus soll es möglich sein, Gruppen von Benutzern eine Mail zu schicken. Hierfür wird die Gruppendatenbank benötigt. Einigen Benutzern im System soll es erlaubt sein, sich auf dem Mailserver einzuloggen (z.B. zu Administrationszwecken). Für diesen Fall müssen sie Zugriff auf ihr Homeverzeichnis haben, weshalb die Datenbank für die Homeverzeichnisse vorhanden sein muß. Sie wird auch benötigt, falls der Benutzer beim Empfang von Mail eigene Aktionen definieren möchte, die die Mail z.B. weiterleiten oder filtern („.forward“–Datei im Homeverzeichnis des Benutzers).

Die vom YP–Server gelieferten Daten haben prinzipiell schon das geeignete Format, um als lokale Datenbanken verwendet werden zu können. Allerdings gilt es einige Aspekte zu berücksichtigen, die eine unerwünschte Nutzung (Security) oder eine nicht mehr mögliche Nutzung (Safety) des Mailservers nach sich ziehen könnten.

Damit sich nicht jeder Benutzer, der in der YP–Benutzerdatenbank eingetragen ist, auf dem Mailserver einloggen kann, muß die lokale Version dieser Datenbank nachbearbeitet werden. Konkret heißt das, daß die Login–Shell (also das Programm, das beim Einloggen des Benutzers als Arbeitsumgebung für ihn gestartet wird) durch eine spezielle Login–Shell ersetzt werden muß, die dem Benutzer den Login verweigert. Ein einfaches Löschen aller unerwünschten Benutzer aus der lokalen Datenbank hätte zur Folge, daß diese Benutzer keine Mail vom Mailserver abrufen könnten. Für welche Benutzer das Einloggen auf dem Mailserver weiterhin erlaubt sein soll, legt eine soge-

[4]SMTP – Simple Mail Transport Protocol
[5]POP3 – Post Office Protocol 3
[6]NFS – Network File System

nannte „Allow“–Datei fest, die Login–Namen oder andere eindeutige Kennzeichen von Benutzern oder Benutzergruppen enthält. Alle anderen Benutzer erhalten als Login–Shell die sogenannte „Nologin“–Shell, die den nicht berechtigten Benutzer mit einer Meldung über sein erfolgloses Bemühen informiert und nicht auf dem Mailserver arbeiten läßt.

Die oben erwähnte Tatsache, daß das Skript `CPYP` nur alle sechs Stunden ausgeführt wird, führt dazu, daß die lokale Datenbank bereits kurz nach ihrer Erstellung veraltet ist. So ist es nicht unwahrscheinlich, daß neue Benutzer im System angelegt wurden und bereits in der Datenbank auf dem YP–Server verfügbar sind. Die lokale Benutzerdatenbank ist nun dahingehend modifiziert, daß die Dienste des Mailservers bei einer erfolglosen Anfrage an sie eine Anfrage an den YP–Server nach dem unbekannten Benutzer stellen („Plus–Zeile“). Dabei ist zu beachten, daß diese Benutzer ebenfalls eine „Nologin“–Shell erhalten, damit sie sich nicht auf dem Mailserver einloggen können.

Die Behandlung der Gruppendatenbank und der Datenbank für die Homeverzeichnisse ist ungleich einfacher, da hier keine Modifikationen an den vom YP–Server gelieferten Daten vorgenommen werden müssen.

Ein weiterer Safety–Aspekt von `CPYP` ist die Fehlertoleranz bezüglich der Kommunikation mit dem YP–Server. Damit der YP–Server die Datenbanken netzweit zur Verfügung stellen kann, muß der YP–Dienst auf ihm ausgeführt werden. Falls dieser, z.B. durch einen Programmfehler oder Überlastung, nicht angesprochen werden kann, erhält der Client (in diesem Fall der Mailserver) keine Daten. Diesen Fall kann das Skript erkennen und auf einen zweiten Weg ausweichen. Dazu loggt sich das Skript auf dem YP–Server ein und kopiert die Daten („rcp“ bzw. „rsh cat“) auf den Mailserver. Sollte auch diese Methode fehlschlagen, bricht das Skript mit einer Fehlermeldung ab, ohne die lokalen Daten zu verändern. So kann zumindest noch mit den veralteten Daten weitergearbeitet werden, was immer noch besser ist, als gar keine oder eine fehlerhafte lokale Datenbasis mehr zu haben.

2.2 Erläuterungen zu `CPYP`

Die Implementation des in Kapitel 2.1 beschriebenen Vorgehens soll kurz erläutert werden. Es handelt sich um ein Shellskript, das zusätzlich AWK benutzt. Das Shellskript ist im Anhang A zu finden. Die in Klammern gesetzten Ziffern korrespondieren mit den entsprechenden Ziffern im Shellskript.

Zu Beginn (1) werden einige Variablen gesetzt, die z.B. YP–Anfragen an den YP–Server unterbinden (somit nutzt `CPYP` nur eine Methode zur Datenerlangung), den Namen des YP–Servers, die Namen der Datenbanken und ihren Speicherort festlegen. Darüberhinaus werden eindeutige Namen für einige temporäre Dateien erzeugt.

Das Skript definiert eine Funktion (2), die für alle drei Datenbanken je einmal aufgerufen wird (9). Zu Beginn der Funktion werden eventuell vorhandene lokale Daten der neuen Datenbank hinzugefügt (z.B. die Daten des Root–Benutzers) (3). Anschließend werden, falls der YP–Dienst benutzt werden soll, die Daten mittels einer YP–Anfrage kopiert (die „auto.home"–Datei erfordert eine Sonderbehandlung).

Falls nun (4) die YP–Anfrage fehlgeschlagen ist oder gar keine YP–Anfrage erwünscht war, werden die Daten auf dem alternativen Weg übertragen. Schlägt auch dies fehl (5), bricht das Skript mit einer Fehlermeldung sauber ab.

Falls die gerade bearbeitete Datenbank die Benutzerdatenbank ist (6), wird sie der in Kapitel 2.1 beschriebenen Modifikation unterzogen. Es wird ein temporäres AWK–Skript erzeugt, welches für alle Benutzer(-gruppen), die nicht in der „Allow"–Datei verzeichnet sind, die „Nologin"–Shell als neue Login–Shell einträgt. Danach wird die beschriebene „Plus–Zeile" angefügt (7) und die so erzeugte neue lokale Datenbank an ihren Platz kopiert und mit den richtigen Benutzerrechten versehen (8).

3 Abstraktes System

3.1 Sicherheitsbedingungen von CPYP

Ziel der Verifikation von CPYP soll es sein, bestimmte Sicherheitsbedingungen nachzuweisen, die eine korrekte Implementation von CPYP erfüllen muß. Diese ergeben sich aus den in Kapitel 2.1 genannten Punkten und sollen hier noch einmal formalisiert werden.

Für diese Betrachtung werden die verschiedenen Mail–Dienste des Servers (POP3, SMTP, NFS–Export) zu einem einzigen, abstrakten „Mail–Dienst“ zusammengefaßt. Die anderen betrachteten Dienste sind die Möglichkeit eines Logins auf dem Mailserver, um dort unter einer Login–Shell zu arbeiten und die Bereitstellung des Homeverzeichnisses des Benutzers.

Eine weitere Funktion von CPYP ist es, die Gruppendatenbank vom YP–Server auf den Mailserver zu übertragen, so daß es möglich ist, an alle im Rechnerverbund registrierten Benutzer–Gruppen Mails zu schicken. Diese Funktionalität wird nicht verifiziert, da sie einerseits eine erhebliche Erweiterung des konkreten Systems (und damit eine Vergrößerung des Zustandsraums) zur Folge hätte und andererseits — im Skript ersichtlich — keine aufwendige Operationen erfordert.

Die in Kapitel 2.1 beschriebene Klassifikation von Benutzern formalisieren wir folgendermaßen:

- Die Benutzermenge MAIL_OK darf den Mail–Dienst benutzen und umfaßt alle Benutzer, die in der lokalen Paßwortdatei mit einem korrekten Paßwort eingetragen sind oder, falls sie nicht in der lokalen Datei eingetragen sind, in der YP–Paßwortdatenbank mit korrektem Paßwort verzeichnet sind.

- Die Benutzermenge LOGIN_OK darf den Login–Dienst benutzen und beinhaltet die Benutzer, die in der lokalen Paßwortdatei mit einem korrekten Paßwort und gleichzeitig von der „Allow“–Datei abgedeckt sind, was dazu führt, daß sie in der lokalen Paßwortdatei ihre angestammte Login–Shell behalten.

- Die Benutzermenge MAIL_NOK darf den Mail–Dienst nicht benutzen und ist die Menge aller Benutzer, die nicht in der Menge MAIL_OK enthalten sind.

- Die Benutzermenge LOGIN_NOK ist nicht berechtigt, den Login–Dienst zu benutzen und ist die Menge aller Benutzer, welche nicht in der Menge LOGIN_OK enthalten sind.

- Es gilt: LOGIN_OK $\subseteq$ MAIL_OK, da für beide Mengen die Voraussetzung eines korrekten Paßworts in der lokalen Paßwortdatei gegeben sein muß.

- Die Benutzermenge HOME_OK darf den Homeverzeichnis–Dienst benutzen und beinhaltet die Benutzer, die in der lokalen Datei oder der YP–Datenbank mit den Speicherorten der Homeverzeichnisse aufgeführt sind.

- Die Benutzermenge HOME_NOK darf den Homeverzeichnis–Dienst nicht benutzen und ist die Menge aller Benutzer, die nicht in der Menge HOME_OK enthalten sind.

- Es muß dabei gelten:
  $user \in MAIL\_OK \vee user \in LOGIN\_OK \Rightarrow user \in HOME\_OK$,
  d.h. falls ein Benutzer sich einloggen kann oder Mail lesen / schreiben darf, muß er Zugriff auf sein Home–Verzeichnis haben.

3.2 Spezifikation des sicheren Systems

Die im vorhergehenden Kapitel definierten Mengen, die eine Klassifikation der Benutzer vornehmen, werden nun in Events umgesetzt, die die (Nicht-) Berechtigung eines Benutzers bzgl. der Anwendung eines Dienstes darstellen. Anschließend erfolgt ein Event, das die erwartete Antwort eines sicheren Systems auf den Versuch einen Dienst zu benutzen repräsentiert. Die Menge HOME_OK bzw. HOME_NOK wird hier nicht explizit berücksichtigt, da im konkreten System die Benutzung des Mail– bzw. des Login–Dienstes nur erfolgreich sein kann, falls der Home–Dienst ebenfalls verfügbar war (letzte Sicherheitsbedingung in Kapitel 3.1). Die Überprüfung dieser Bedingung erfolgt also implizit.

In CSP würde ein sicheres System, das die definierten Sicherheitsbedingungen umsetzt, folgendermaßen aussehen:

$$
\begin{array}{l}
SAFE\_SYS = \\
(user\_mail\_nok \rightarrow user\_login\_nok \rightarrow \\
\quad (\\
\quad\quad (user\_wants\_mail \rightarrow user\_no\_mail \rightarrow SAFE\_SYS) \\
\quad\quad \sqcap \\
\quad\quad (user\_wants\_login \rightarrow user\_no\_login \rightarrow SAFE\_SYS) \\
\quad) \\
) \\
\sqcap \\
(user\_mail\_ok \rightarrow user\_login\_nok \rightarrow \\
\quad (\\
\quad\quad (user\_wants\_mail \rightarrow user\_reads\_mail \rightarrow SAFE\_SYS) \\
\quad\quad \sqcap \\
\quad\quad (user\_wants\_login \rightarrow user\_no\_login \rightarrow SAFE\_SYS) \\
\quad) \\
) \\
\sqcap \\
(user\_mail\_ok \rightarrow user\_login\_ok \rightarrow \\
\quad (\\
\quad\quad (user\_wants\_mail \rightarrow user\_reads\_mail \rightarrow SAFE\_SYS) \\
\quad\quad \sqcap \\
\quad\quad (user\_wants\_login \rightarrow user\_logs\_in \rightarrow SAFE\_SYS) \\
\quad) \\
)
\end{array}
$$

Bei der Implementierung von `CPYP` standen diese Sicherheitsbedingungen im Mittelpunkt der Konzeption. Es soll nun im folgenden verifiziert werden, ob die tatsächliche Implementierung von `CPYP` diese Bedingungen erfüllt.

4 Spezifikation des konkreten Systems

4.1 Gesamtsystem

Um einen gewissen Überblick über die im Gesamtsystem stattfindende Kommunikation und die Gliederung desselben darzustellen, zunächst eine Skizze des Gesamtsystems. Die Definition der Kommunikationskanäle erfolgt später in diesem Kapitel.

Das Gesamtsystem läßt sich in zwei Teile gliedern: die Datenbanken mit den Informationen über die Berechtigungen der Benutzer auf der einen Seite, sowie die darauf zugreifenden Dienste, Benutzer und `CPYP` auf der anderen. Die Benutzer wiederum greifen auf die Dienste des Mailservers zurück. Zusätzlich existiert ein Semaphor, dessen Verwendung in Kapitel 4.3 erläutert wird. Die Kommunikation erfolgt über mehrere Kanäle, die nach funktionalen Kriterien eingeteilt wurden und später erläutert werden.

$$\begin{aligned}
&SYS = \\
&(\ (USER \underset{channels2}{\|} SERVICES) \;|||\; CPYP(0, USENIS)\ ) \underset{channels1}{\|} \\
&\qquad\qquad (\ YPSERVER \;|||\; FILES \;|||\; PLUS \;|||\; SEM(1)\ )
\end{aligned}$$

Der *USER*–Prozeß (Kapitel 4.3) hat dabei die Aufgabe, Benutzer aus einer Menge von Benutzer–IDs zu erzeugen und sie im anschließenden *USERINIT*–Prozeß mit allen möglichen Kombinationen aus Berechtigungen zu versehen. Diese Berechtigungen werden in die lokalen Dateien (Prozesse *FILES* und *PLUS*, Kapitel 4.5.1) sowie in den YP–Server (Prozeß *YPSERVER*, Kapitel 4.4) eingetragen. Danach wird für jeden Dienst durch ein Event angezeigt, ob der Benutzer für ihn eine Benutzungsberechtigung besitzt. Diese Events sind später für die Refinement–Untersuchungen wichtig. Der *USERINIT*–Prozeß geht in den *USERACTION*–Prozeß über, der für einen Benutzer einen zufällig ausgewählten Dienst in Anspruch nimmt. Die Dienste sind im *SERVICES*–Prozeß zusammengefaßt (Kapitel 4.5.2), der eine externe Auswahl über den *LOGIN*-, den *MAIL*- und den *HOME*–Prozeß ausführt. Der *USERACTION*–Prozeß erhält eine Rückmeldung über den Erfolg oder Nichterfolg der Benutzung eines Dienstes und generiert daraufhin ein ensprechendes Event, welches für die Refinement–Untersuchungen wichtig ist. Die Kommunikationskanäle zwischen den Benutzer–Prozessen und den Diensten sind dabei in der Menge *channels*2 zusammengefaßt.

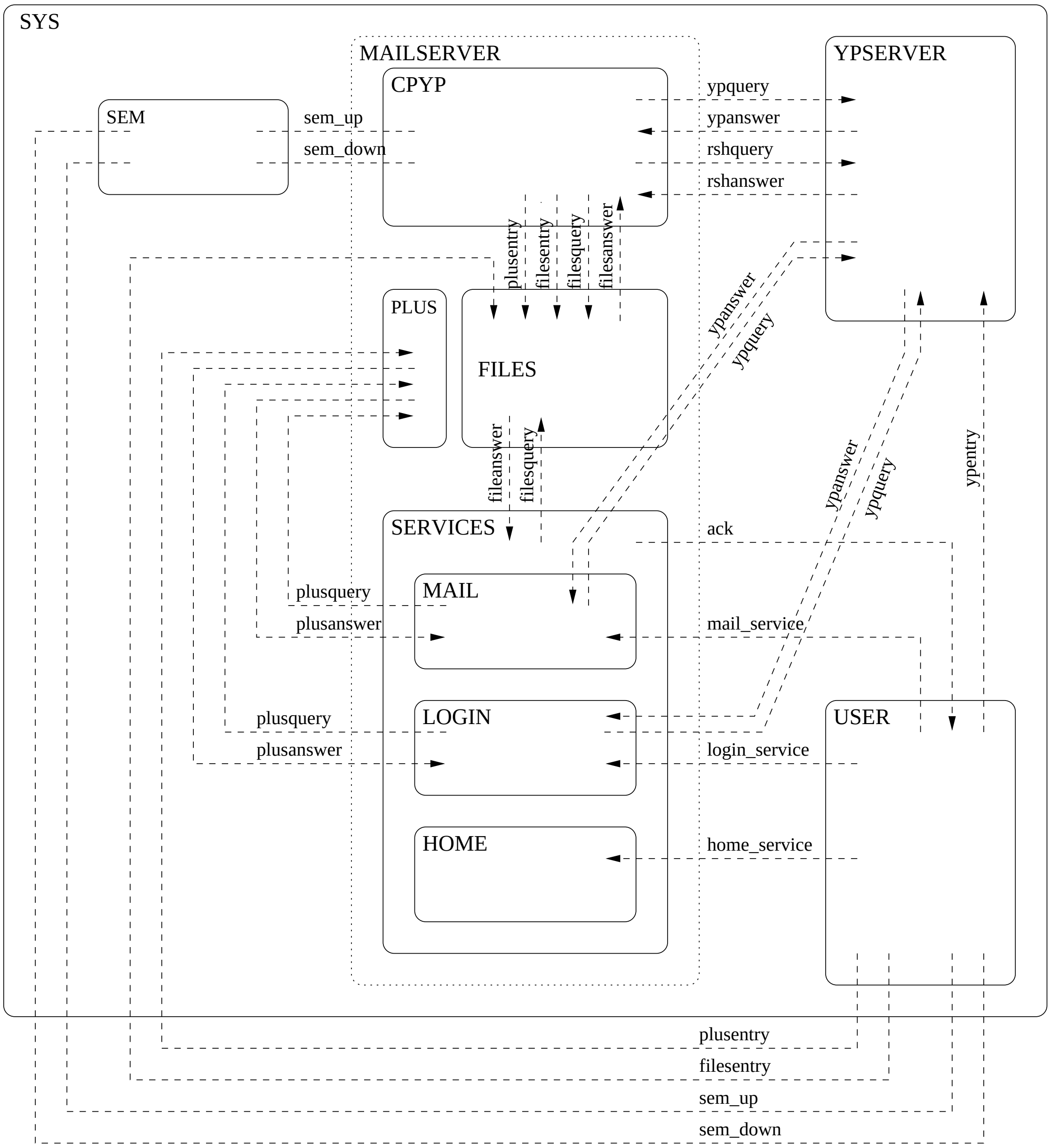

Abbildung 1: Struktrierte Darstellung des Gesamtsystems

Unsynchronisiert parallel dazu läuft der *CPYP*–Prozeß (Kapitel 4.5.3), der das zu untersuchende Shell–Skript repräsentiert. Er erhält als Parameter die User–ID, von der an das Kopieren der YP–Daten beginnen soll und eine Konstante, die die Benutzung der YP–Kommunikationskanäle steuert. Um Konflikte beim Schreiben der lokalen Dateien zu verhindern, mußte ein Semaphor eingeführt werden (Prozeß *SEM*), dessen Verwendung in Kapitel 4.3 erläutert wird.

Der *CPYP*–Prozeß kommuniziert dabei ähnlich den Benutzer–Prozessen mit den lokalen Dateien bzw. dem YP–Server und dem Semaphor über Kommunikationskanäle, die in der Menge $channels1$ zusammengefaßt wurden. Ebenso werden Kanäle aus dieser Menge von den Diensten benutzt, um die Berechtigungen der Benutzer zu erfahren.

4.2 Verwendete Kanäle und Datentypen

Die Benutzer–IDs stammen aus einer Menge *UID*. Die maximale Anzahl der Benutzer im System ist begrenzt. Dies hat praktische Gründe, die in Kapitel 5.2 erläutert werden.

Es ist statisch festgelegt, welchen Wert die *USENIS*–Variable von `CPYP` haben soll (vgl. Anhang A (1)).

Der Datentyp *PERM* beschreibt die drei Zustände, die die Berechtigung eines Benutzers für einen bestimmten Dienst ausdrücken („noentry“ bedeutet dabei, daß der Benutzer nicht berechtigt ist, da er nicht in der entsprechenden Datenbank eingetragen ist).

Außerdem wird ein Datentyp *DATABASE* definiert, der die Variablen für die verschiedenen zu übertragenden Dateien repräsentiert (vgl. Anhang A (1)). Darüberhinaus wird ein Datentyp *SERVICE* für die vom Mailserver angebotenen Dienste definiert.

Die Erfolgs- bzw. Nichterfolgsmeldung bei der Benutzung eines Dienstes wird in einem weiteren Datentyp *RESULT* ausgedrückt.

$$MAXUID = 1$$

$$UID = \{0..MAXUID\}$$

$$USENIS = 1$$

$$PERM = ok \mid nok \mid noentry$$

$$RSH\_PERM = PERM$$
$$YP\_PERM = PERM$$
$$PLUS\_PERM = PERM$$

$$DATABASE = passwd \mid group \mid autohome$$

$$SERVICE = mail \mid login \mid home \mid allow \mid plus$$

$$RESULT = res\_ok \mid res\_nok$$

Dabei gibt es folgende Kanäle über die die oben definierten Datentypen ausgetauscht werden:

$$channel\ ypentry : UID.SERVICE.YP\_PERM$$
$$channel\ ypquery : UID.SERVICE$$
$$channel\ ypanswer : UID.SERVICE.YP\_PERM$$

$$channel\ rshquery : UID.SERVICE$$
$$channel\ rshanswer : UID.SERVICE.RSH\_PERM$$

$$channel\ filesentry : UID.SERVICE.PERM$$
$$channel\ filesquery : UID.SERVICE$$
$$channel\ filesanswer : UID.SERVICE.PERM$$

$$channel\ plusentry : PLUS\_PERM$$
$$channel\ plusquery$$
$$channel\ plusanswer : PLUS\_PERM$$

$$channel\ mail\_service : UID$$
$$channel\ login\_service : UID$$
$$channel\ home\_service : UID$$

$$channel\ ack : RESULT$$

$$channel\ sem\_down, sem\_up$$

$$channels1 = \{ypquery, ypanswer, rshquery, rshanswer, filesquery, filesanswer, filesentry, plusquery, plusanswer, plusentry, ypentry, sem\_up, sem\_down\}$$

$$channels2 = \{login\_service, mail\_service, home\_service, ack\}$$

$$channel\ user\_mail\_ok, user\_mail\_nok, user\_login\_ok, user\_login\_nok$$

$$channel\ user\_wants\_mail, user\_reads\_mail, user\_no\_mail, user\_wants\_login, user\_logs\_in, user\_no\_login$$

Für die Kommunikation mit dem YP–Server existiert ein Kanal zum Eintragen neuer Informationen (*ypentry*), einer zur YP–Abfrage von Informationen (*ypquery*) sowie ein weiterer zur Beantwortung einer solchen Abfrage (*ypanswer*). Für die beiden letztgenannten Kanäle existiert ein Äquivalent, das den alternativen Zugang zum YP–Server („rsh“) benutzt (*rshquery*, *rshanswer*). Analog zu den YP–Kanälen sind die Kanäle zu den lokalen Dateien definiert (*filesentry*, *filesquery*, *filesanswer*).

Für das Eintragen (*plusentry*), Abfragen (*plusquery*) und Beantworten (*plusanswer*) der Abfrage der „Plus–Zeile“ in der Paßwortdatei existieren eigene Kanäle, da diese Information nicht für jeden Benutzer einzeln existiert, sondern nur einmal in der Paßwortdatei definiert ist.

Die Dienste des Mailservers erhalten eigene Kanäle (*mail_service*, *login_service*, *home_service*), über die Benutzer mit ihm kommunizieren können. Antwort erhalten sie über einen Acknowledgement–Kanal (*ack*).

Zusätzlich werden zwei Kanäle für die Benutzung eines Semaphors eingeführt (*sem_down*, *sem_up*), dessen Verwendung im nächsten Kapitel erklärt wird.

Die Kanäle werden in Mengen zusammengefaßt, um die Beschreibung paralleler Prozesse, die über diese Kanäle synchronisiert werden, zu vereinfachen. Außerdem werden die Mengen beim Hiding von Events des konkreten Systems verwendet.

Zuletzt werden die Kanäle definiert, die sowohl im abstrakten wie auch im konkreten System verwendet werden und ein Trace–Refinement ermöglichen. Die Benutzung erfolgt dabei analog zu Definition der Mengen in Kapitel 3.1.

4.3 Benutzer

Die Benutzer des Systems sind folgendermaßen modelliert: sie unterscheiden sich in Benutzer, die unerlaubterweise und erlaubterweise den Mail–Dienst und analog den Login–Dienst benutzen wollen. Vereinfacht wird davon ausgegangen, daß immer nur ein Benutzer auf dem System aktiv sein kann. Es ist aber möglich, verschiedene Benutzer zu erzeugen und zu initialisieren. Da mögliche Probleme durch nebenläufige Zugriffe auf die Dienste des Mailservers jedoch nicht Gegenstand des Skripts sind, sondern durch das zugrundeliegende Betriebssystem behandelt werden und die Funktionsweise des Skripts davon nicht berührt wird, ist diese Vereinfachung zulässig.

Bei der Umsetzung des Systems in CSP ergab sich ein Problem, daß zur Einführung eines Semaphors führte (Kapitel 5.2). Dieser Semaphor wird vom Benutzerinitialisierungsprozeß sowie von `CPYP` benutzt und verhindert, daß während der Initialisierung eines neuen Benutzers inkonsistente Daten von `CPYP` verwendet werden.

$$
\begin{array}{l}
\mathit{SEM}(n) = \\
(n == 1)\ \&\ \mathit{sem\_down} \rightarrow \mathit{SEM}(0) \\
\Box \\
(n == 0)\ \&\ \mathit{sem\_up} \rightarrow \mathit{SEM}(1)
\end{array}
$$

Bei der Initialisierung des Benutzers werden die Benutzerrechte immer in den YP–Server eingetragen und eventuell auch in die lokale Paßwortdatei, sofern der Benutzer im System existiert. Dies abstrahiert den Umstand, daß die Aktualisierung der lokalen Paßwortdatei durch `CPYP` nur alle 6 Stunden geschieht.

Dabei ergibt die Kombination aus der Berechtigung (*PERM*), dem Dienst und dem Speicherort (lokal oder im YP–Server) folgende Bedeutung in der „realen“ Umgebung.

	mail_service	login_service	home_service
lokal ok	richtiges Paßwort	richtiges Paßwort und Eintrag in „Allow“–Datei	eingetragen
lokal nok	falsches Paßwort	keine Login–Shell	nicht eingetragen
lokal noentry	nicht eingetragen	nicht eingetragen	nicht eingetragen
YP ok	richtiges Paßwort	—	eingetragen
YP nok	falsches Paßwort	keine Login–Shell	—
YP noentry	nicht eingetragen	nicht eingetragen	nicht eingetragen

Es mag seltsam erscheinen, daß bereits vor dem Versuch der Benutzung eines der Dienste festgelegt sein soll, ob ein Benutzer die Erlaubnis dazu besitzt oder nicht. Dies ist jedoch Voraussetzung für ein Trace–Refinement des konkreten Systems gegen die Sicherheitsbedingungen, da hier mit abstrakten Events und nicht einer konkreten Datenbasis aus Login–Namen, Paßwörtern etc. gearbeitet wird.

$USER = \bigsqcap u : UID \bullet USERINIT(u)$

$USERINIT(u) =$
$sem\_down \rightarrow$
$($
$\quad (ypentry.u!mail.noentry \rightarrow ypentry.u!login.noentry \rightarrow$
$\quad filesentry.u!mail.noentry \rightarrow filesentry.u!login.noentry \rightarrow$
$\quad filesentry.u!allow.noentry \rightarrow ypentry.u!home.noentry \rightarrow$
$\quad filesentry.u!home.noentry \rightarrow user\_mail\_nok \rightarrow$
$\quad user\_login\_nok \rightarrow SKIP$
$\quad)$
$\quad \sqcap$
$\quad (ypentry.u!mail.nok \rightarrow ypentry.u!login.nok \rightarrow$
$\quad\quad ($
$\quad\quad\quad (filesentry.u!mail.nok \rightarrow$
$\quad\quad\quad filesentry.u!login.nok \rightarrow$
$\quad\quad\quad filesentry.u!allow.noentry \rightarrow SKIP$
$\quad\quad\quad)$
$\quad\quad\quad \sqcap$
$\quad\quad\quad (filesentry.u!mail.noentry \rightarrow$
$\quad\quad\quad filesentry.u!login.noentry \rightarrow$
$\quad\quad\quad filesentry.u!allow.noentry \rightarrow SKIP$
$\quad\quad\quad)$
$\quad\quad);\ user\_mail\_nok \rightarrow user\_login\_nok \rightarrow$
$\quad\quad ypentry.u!home.ok \rightarrow$
$\quad\quad ($
$\quad\quad\quad (filesentry.u!home.noentry \rightarrow SKIP)$
$\quad\quad\quad \sqcap$
$\quad\quad\quad (filesentry.u!home.ok \rightarrow SKIP)$
$\quad\quad)$
$\quad)$

(weiter nächste Seite...)

$\sqcap$

$(ypentry.u!mail.ok \rightarrow user\_mail\_ok \rightarrow ypentry.u!login.nok \rightarrow$

$($

$(filesentry.u!mail.ok \rightarrow$

$($

$(filesentry.u!login.nok \rightarrow$

$filesentry.u!allow.noentry \rightarrow$

$user\_login\_nok \rightarrow SKIP$

$)$

$\sqcap$

$(filesentry.u!login.ok \rightarrow filesentry.u!allow.ok \rightarrow$

$user\_login\_ok \rightarrow SKIP$

$)$

$)$

$)$

$\sqcap$

$(filesentry.u!mail.noentry \rightarrow$

$filesentry.u!login.noentry \rightarrow$

$filesentry.u!allow.noentry \rightarrow user\_login\_nok \rightarrow SKIP$

$)$

$);$

$(ypentry.u!home.ok \rightarrow filesentry.u!home.ok \rightarrow SKIP)$

$)$

$);\ sem\_up \rightarrow USERACTION(u)$

Anschließend versucht der soeben generierte Benutzer, einen zufällig ausgewählten Dienst (Login oder Mail) zu benutzen und erhält eine Rückmeldung über einen speziellen Kanal. Entscheidend für das Trace–Refinement ist die Generierung von Events, die den Versuch des Benutzers melden und ob er erfolgreich war.

$USERACTION(u) =$
$($
  $mail\_service!u \rightarrow user\_wants\_mail \rightarrow ack?result \rightarrow$
  if $(result == res\_ok)$
  then (
    $home\_service!u \rightarrow ack?result \rightarrow$
    if $(result == res\_ok)$
    then $(user\_reads\_mail \rightarrow USER)$
    else $user\_no\_mail \rightarrow USER$
)
  else $(user\_no\_mail \rightarrow USER)$
$)$
$\sqcap$
$($
  $login\_service!u \rightarrow user\_wants\_login \rightarrow ack?result \rightarrow$
  if $(result == res\_ok)$
  then (
    $home\_service!u \rightarrow ack?result \rightarrow$
    if $(result == res\_ok)$
    then $(user\_logs\_in \rightarrow USER)$
    else $user\_no\_login \rightarrow USER$
)
  else $(user\_no\_login \rightarrow USER)$
$)$

4.4 YP–Server

Der YP–Server stellt `CPYP` die Datenbasis zur Verfügung, die Grundlage für die zu erstellenden Dateien auf dem Mailserver sind. Es werden drei Datenbanken unterschieden (Mail, Login und Home). Es können Eintragungen bei der Initialisierung der Benutzer vorgenommen werden, sowie der aktuelle Status eines Eintrags abgefragt werden. Dabei können über zwei Kommunikationswege („ypcat“ und „rsh“) mit anderen Prozessen Daten ausgetauscht werden. Für jeden Benutzer existiert ein eigener Subprozeß, der seine Benutzerrechte speichert.

$YPSERVER =$
$\vert\vert\vert\ u : \{0..MAXUID\} \bullet YPSERVER2(u, noentry, noentry, noentry)$

$$YPSERVER2(u, mailstatus, loginstatus, homestatus) =$$

$$\begin{array}{l}
ypquery.u?login \rightarrow ypanswer.u!login.loginstatus \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homestatus) \\
\Box \\
ypquery.u?mail \rightarrow ypanswer.u!mail.mailstatus \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homestatus) \\
\Box \\
ypquery.u?home \rightarrow ypanswer.u!home.homestatus \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homestatus) \\
\Box \\
rshquery.u?login \rightarrow rshanswer.u!login.loginstatus \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homestatus) \\
\Box \\
rshquery.u?mail \rightarrow rshanswer.u!mail.mailstatus \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homestatus) \\
\Box \\
rshquery.u?home \rightarrow rshanswer.u!home.homestatus \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homestatus) \\
\Box \\
ypentry.u?mail.mailperm \rightarrow \\
YPSERVER2(u, mailperm, loginstatus, homestatus) \\
\Box \\
ypentry.u?login.loginperm \rightarrow \\
YPSERVER2(u, mailstatus, loginperm, homestatus) \\
\Box \\
ypentry.u?home.homeperm \rightarrow \\
YPSERVER2(u, mailstatus, loginstatus, homeperm)
\end{array}$$

4.5 Mailserver

Aus der Zusammensetzung des konkreten Systems in Kapitel 4.1 ist nicht direkt ersichtlich, welche Komponenten der Spezifikation physikalisch (lokale Dateien) oder virtuell (Dienste) dem Mailserver angehören. Trotzdem wurde aus Gründen der Übersichtlichkeit diese Gliederung in der Beschreibung der Spezifikation angewendet. Es folgen nun also Prozesse, die aus „nicht–CSP–Sicht“ dem Mailserver zugeordnet würden.

4.5.1 Lokale Dateien

Die Files repräsentieren die lokalen Dateien auf dem Mailserver. Die Spezifikation ist dabei analog zum YP–Server, es kommen lediglich zwei Varianten hinzu. Die „Allow–Datei“ enthält die Merkmale von Benutzern bzw. Gruppen von Benutzern, die sich auf dem Mailserver einloggen dürfen. Der Plus–Eintrag bestimmt, ob ein Dienst neben den lokalen Dateien auch eine Anfrage an die YP–Datenbank stellen darf und wird in einem eigenen Prozeß modelliert. Für jeden Benutzer existiert ein eigener Subprozeß, der seine Benutzerrechte speichert.

$$
\begin{aligned}
&\mathit{FILES} = \\
&|||\; u : \{0..\mathit{MAXUID}\} \bullet \mathit{FILES}2(u, \mathit{noentry}, \mathit{noentry}, \mathit{noentry}, \mathit{noentry})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homestatus}) = \\
&\mathit{filesquery}.u?\mathit{mail} \rightarrow \mathit{filesanswer}.u!\mathit{mail}.\mathit{mailstatus} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesquery}.u?\mathit{login} \rightarrow \mathit{filesanswer}.u!\mathit{login}.\mathit{loginstatus} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesquery}.u?\mathit{allow} \rightarrow \mathit{filesanswer}.u!\mathit{allow}.\mathit{allowstatus} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesquery}.u?\mathit{home} \rightarrow \mathit{filesanswer}.u!\mathit{home}.\mathit{homestatus} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesentry}.u?\mathit{mail}.\mathit{mailperm} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailperm}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesentry}.u?\mathit{login}.\mathit{loginperm} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginperm}, \mathit{allowstatus}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesentry}.u?\mathit{allow}.\mathit{allowperm} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowperm}, \mathit{homestatus}) \\
&\Box \\
&\mathit{filesentry}.u?\mathit{home}.\mathit{homeperm} \rightarrow \\
&\mathit{FILES}2(u, \mathit{mailstatus}, \mathit{loginstatus}, \mathit{allowstatus}, \mathit{homeperm})
\end{aligned}
$$

Der Plus–Eintrag in der lokalen Paßwortdatei, wird hier in einem separaten Prozeß behandelt, da er nicht für jeden Benutzer einzeln vorhanden ist. Der Initialzustand

(„ok“) kann sich in der Spezifikation niemals ändern, da der Prozeß `CPYP` (Kapitel 4.5.3) der einzige ist, der schreibend auf ihn zugreift und ebenfalls nur „ok“ schreiben kann. Dies entspricht dem realen Ablauf von `CPYP`, bei dem zu keinem Zeitpunkt eine unvollständige (d.h. keine „Plus–Zeile“) Paßwortdatei im System existiert[7].

$$PLUS = PLUS2(ok)$$

$$\begin{array}{l} PLUS2(plusstatus) = \\ plusquery \rightarrow plusanswer!plusstatus \rightarrow PLUS2(plusstatus) \\ \Box \\ plusentry?plusperm \rightarrow PLUS2(plusstatus) \end{array}$$

4.5.2 Services

Für die Betrachtung des Skriptes `CPYP` genügt es, die Dienste Mail, Login und Home zu betrachten, von denen im konkreten System immer nur einer zur Zeit aktiv sein kann. Der Grund dafür wird in Kapitel 5.2 erläutert.

$$SERVICES = MAIL \;\Box\; LOGIN \;\Box\; HOME$$

Der Mail–Dienst hat in etwa folgendes Ablaufschema (auch hier wird stark vereinfacht, was der Verifikation von `CPYP` jedoch nicht abkömmlich ist). Falls der Benutzer in der lokalen Paßwortdatei eingetragen ist (und sein Paßwort korrekt war), erhält er eine positive Antwort. Falls er nicht in der lokalen Datei eingetragen ist, wird eine Anfrage an den YP–Server gestartet, falls die „Plus–Zeile“ in der lokalen Paßwortdatei existiert. Ist der Benutzer in der YP–Datenbank mit korrektem Paßwort eingetragen, erhält er ebenfalls eine positive Antwort. Die Entscheidung, ob der Benutzer in der lokalen Datei eingetragen ist oder über YP–Anfrage herausgefunden werden muß, ist nicht beeinflußbar (Internal Choice bei der Benutzerinitialisierung) und hängt mit den Zeitpunkten der Benutzereintragung in den YP–Server und den Ausführungzeitpunkten von `CPYP` zusammen.

Die genaue Bedeutung der Einträge im YP–Server sowie in den lokalen Dateien ist der Tabelle in Kapitel 4.3 zu entnehmen.

[7] temporäre Dateien sind dagegen sehr wohl unvollständig

$$
\begin{array}{l}
\mathit{MAIL} = \\
\mathit{mail\_service}?u \rightarrow \mathit{filesquery}.u!\mathit{mail} \rightarrow \mathit{filesanswer}.u?\mathit{mail}.\mathit{status}1 \rightarrow \\
\text{if } (\mathit{status}1 == \mathit{ok}) \\
\text{then } (\mathit{ack}!\mathit{res\_ok} \rightarrow \mathit{SERVICES}) \\
\text{else } (\\
\quad \mathit{plusquery} \rightarrow \mathit{plusanswer}?\mathit{status}2 \rightarrow \\
\quad \text{if } (\mathit{status}2 == \mathit{ok}) \\
\quad \text{then } (\\
\quad\quad \mathit{ypquery}.u!\mathit{mail} \rightarrow \mathit{ypanswer}.u?\mathit{mail}.\mathit{status}3 \rightarrow \\
\quad\quad \text{if } (\mathit{status}3 == \mathit{ok}) \\
\quad\quad \text{then } (\mathit{ack}!\mathit{res\_ok} \rightarrow \mathit{SERVICES}) \\
\quad\quad \text{else } (\mathit{ack}!\mathit{res\_nok} \rightarrow \mathit{SERVICES}) \\
\quad\quad) \\
\quad \text{else } (\mathit{ack}!\mathit{res\_nok} \rightarrow \mathit{SERVICES}) \\
\quad)
\end{array}
$$

Der Login–Dienst ist dem Mail–Dienst ähnlich. Die YP–Anfrage, nachdem festgestellt wurde, daß der Benutzer nicht in der lokalen Paßwortdatei eingetragen ist, ist zwar unnötig (vgl. Tabelle in Kapitel 4.3), wird aber in der Realität ebenfalls ausgeführt.

$$
\begin{array}{l}
\mathit{LOGIN} = \\
\mathit{login\_service}?u \rightarrow \mathit{filesquery}.u!\mathit{login} \rightarrow \mathit{filesanswer}.u?\mathit{login}.\mathit{status}1 \rightarrow \\
\text{if } (\mathit{status}1 == \mathit{ok}) \\
\text{then } (\mathit{ack}!\mathit{res\_ok} \rightarrow \mathit{SERVICES}) \\
\text{else } (\\
\quad \text{if } (\mathit{status}1 == \mathit{nok}) \\
\quad \text{then } (\mathit{ack}!\mathit{res\_nok} \rightarrow \mathit{SERVICES}) \\
\quad \text{else } (\\
\quad\quad \mathit{plusquery} \rightarrow \mathit{plusanswer}?\mathit{status}2 \rightarrow \\
\quad\quad \text{if } (\mathit{status}2 == \mathit{ok}) \\
\quad\quad \text{then } (\\
\quad\quad\quad \mathit{ypquery}.u!\mathit{login} \rightarrow \mathit{ypanswer}.u?\mathit{login}.\mathit{status}3 \rightarrow \\
\quad\quad\quad \mathit{ack}!\mathit{res\_nok} \rightarrow \mathit{SERVICES} \\
\quad\quad\quad) \\
\quad\quad \text{else } (\mathit{ack}!\mathit{res\_nok} \rightarrow \mathit{SERVICES}) \\
\quad\quad) \\
\quad)
\end{array}
$$

Der Home–Dienst ist einfacher aufgebaut, da hier bei einer erfolglosen Suche in der lokalen Datei keine YP–Anfrage gemacht wird (zumindest ist dies die momentane Konfiguration des Mailservers).

$HOME =$
$home\_service?u \rightarrow filesquery.u!home \rightarrow filesanswer.u?home.status \rightarrow$
if $(status == ok)$
then $(ack!res\_ok \rightarrow SERVICES)$
else $(ack!res\_nok \rightarrow SERVICES)$

4.5.3 CPYP

Das Skript `CPYP` wird auf dem Mailserver ausgeführt und erstellt dort die lokalen Dateien, die von den Diensten des Mailservers benutzt werden, um festzustellen, ob ein Benutzer bzgl. eines Dienstes berechtigt ist.

`CPYP` kopiert dabei die Daten aller Benutzer, was in der Spezifikation über einen rekursiven Aufruf mit iteriertem Parameter erledigt wird. Um eine Dateninkonsistenz zu vermeiden, wird der in Kapitel 4.3 beschriebene Semaphor verwendet. Falls die Daten aller Benutzer kopiert wurden, beginnt `CPYP` die Prozedur von neuem. In der Realität würde hier eine Pause von ca. 6 Stunden eingelegt — die Verwendung von Timed–CSP schien dies allerdings nicht zu rechtfertigen, da es sich um keine Timeout–Bedingung handelt.

Die Datenakquisition erfolgt fehlertolerant, falls die USENIS–Variable auf „1“ gesetzt wurde. Der External–Choice–Operator stellt dabei sicher, daß, falls ein Zweig sich nicht synchronisieren kann (z.B. weil der YP–Dienst nicht verfügbar ist), der andere ausgewählt wird. Falls beide Zweige sich nicht mit dem YP–Server synchronisieren können, bleibt der spezifizierte Prozeß `CPYP` stehen. In der Realität würde hier eine Fehlermeldung generiert werden. Der Effekt ist jedoch derselbe: es werden keine fehlerhaften lokalen Dateien erzeugt. Das Lesen eventuell vorhandener lokaler Rümpfe von lokalen Dateien wurde abstrahiert („.local“–Dateien, siehe Shellskript in Kapitel A).

Die anschließenden Aktionen entsprechen dem Vorgehen im konkreten Skript, die eingeklammerten Ziffern beziehen sich auf die jeweiligen Stellen im Skript `CPYP` in Anhang A.

$CPYP(u, usenis) =$
if $(u \leq MAXUID)$
then (
  $sem\_down \rightarrow CPYP2(u, usenis, passwd);\ CPYP2(u, usenis, group);$
  $CPYP2(u, usenis, autohome);$ (9)
  $sem\_up \rightarrow CPYP(u+1, usenis)$
)
else $CPYP(0, usenis)$

$CPYP2(u, usenis, database) =$ (2)
if $(usenis == 1)$
then (
$CPYP\_YP(u, database) \square CPYP\_RSH(u, database)$
)
else $CPYP\_RSH(u, database)$ (4)

$CPYP\_YP(u, database) =$
if $(database == passwd)$ (6)
then (
$ypquery.u!mail \rightarrow ypanswer.u?mail.status1 \rightarrow$
$ypquery.u!login \rightarrow ypanswer.u?login.status2 \rightarrow$
$filesquery.u!allow \rightarrow filesanswer.u?allow.status3 \rightarrow$
(
if $(status3 == ok \wedge status2 == nok)$
then $(filesentry.u!login.ok \rightarrow filesentry.u!mail.ok \rightarrow SKIP)$
else $(filesentry.u!login.nok \rightarrow filesentry.u!mail.status1 \rightarrow SKIP)$
); $plusentry!ok \rightarrow SKIP$ (7)
)
else if $(database == group)$
then (
$SKIP$
)
else if $(database == autohome)$
then (
$ypquery.u!home \rightarrow ypanswer.u?home.status4 \rightarrow$
$filesentry.u!home.status4 \rightarrow SKIP$
)
else $STOP$

$CPYP\_RSH(u, database) =$
if $(database == passwd)$ (6)
then (
$rshquery.u!mail \rightarrow rshanswer.u?mail.status1 \rightarrow$
$rshquery.u!login \rightarrow rshanswer.u?login.status2 \rightarrow$
$filesquery.u!allow \rightarrow filesanswer.u?allow.status3 \rightarrow$
(
if $(status3 == ok \wedge status2 == nok)$
then $(filesentry.u!login.ok \rightarrow filesentry.u!mail.ok \rightarrow SKIP)$
else $(filesentry.u!login.nok \rightarrow filesentry.u!mail.status1 \rightarrow SKIP)$
); $plusentry!ok \rightarrow SKIP$ (7)
)
else if $(database == group)$
then (
$SKIP$
)
else if $(database == autohome)$
then (
$rshquery.u!home \rightarrow rshanswer.u?home.status4 \rightarrow$
$filesentry.u!home.status4 \rightarrow SKIP$
)
else $STOP$

5 Verifikation mit FDR2

5.1 Refinement

Wie schon in der Einleitung erläutert, sollte das Ziel der Verifikation von `CPYP` der Beweis der Einhaltung der Sicherheitsbedingungen sein, die durch das sichere, abstrakte System definiert werden. Für diese Zwecke bietet sich das Trace–Refinement an, bei dem festgestellt wird, ob ein konkreter Prozeß eine Verfeinerung eines abstrakten im Trace–Modell darstellt. Eine Trace wiederum beschreibt eine mögliche von einem Prozeß erzeugte Folge von Events (in FDR2 sind dies immer Kommunikationen über Channels). Ein Prozeß ist eine Verfeinerung eines anderen, wenn alle seine Traces Teilmenge der Traces des anderen Prozesses sind. Dies heißt, daß der konkrete (verfeinerte) Prozeß keine Traces erzeugen kann, die der abstrakte nicht auch erzeugen könnte. Ein solches Modell eignet sich hervorragend für eine Safety–Untersuchung, wie sie durchgeführt wurde, da ein abstraktes, sicheres System alle erlaubten Traces definieren kann und damit das gewünschte Systemverhalten modelliert.

Eine weitergehende Verfeinerungsrelation ist das Failures–Refinement, bei dem das Verhalten der Prozesse mittels Tupeln aus einer Trace und der nach dieser Trace möglichen verweigerten Events definiert ist. Ein Prozeß ist im Failures–Modell eine Verfeinerung eines anderen, wenn seine gesamten Failures Teilmenge der Failures des anderen Prozesses sind. Damit ist es möglich, festzulegen daß ein Prozeß bestimmte Events annehmen muß, um einen anderen zu verfeinern. Dies ist beim Trace–Modell nicht unbedingt gegeben, wie später noch gezeigt wird. Failures–Refinement ist also auch geeignet, Deadlocks zwischen Prozessen offenzulegen.

Die dritte Verfeinerungsrelation, die mit FDR2 überprüft werden kann, ist das Failures–Divergence–Refinement. Es kombiniert das Failures–Refinement mit der Untersuchung von Traces nach Divergenzen. Eine Divergenz ist dabei ein Auftreten unendlich vieler interner (z.B. durch Hiding nach außen nicht sichtbarer) Events nach einer bestimmten Trace.

Für die Verifikation von `CPYP` wurden alle Refinementrelationen betrachtet. Um das ursprüngliche Ziel der Verifikation zu erfüllen, würde, wie erläutert, schon der Beweis einer gültigen Verfeinerung im Trace–Modell genügen.

5.2 Probleme bei der Erstellung der CSP–Spezifikation

Die Spezifikation des Systems wurde zunächst in LaTeX erstellt, um bei der Entwicklung jederzeit eine druckreife und vor allem gut lesbare Spezifikation zu haben. Diese Vorteile wiegen den Nachteil des fehlenden Syntax–Checks, den man bei der Erstellung in FDR–Syntax hätte, wieder auf.

Eine erste, vollständig erscheinende Spezifikation wurde dann in FDR–Syntax umgesetzt und zunächst syntaktische Fehler behoben. Dabei handelte es sich vornehmlich um ungültig beschriebene Prozesse in einer sequentiellen Komposition von Prozessen, die durch das Einfügen diverser „SKIP"–Aufrufe behoben werden konnte.

Die nächste Hürde war die Kommunikation der Prozesse untereinander. Hier mußten der Hauptprozeß *SYS* mehrmals umformuliert werden. Der Grund lag dabei in den komplexen Kommunikationsbeziehungen, die nicht nur zweiseitig, sondern teilweise auch mit drei Partnern definiert sind (vgl. Abbildung 1). Dadurch kam es zu Deadlocks bzw. zunächst zu gar keiner Kommunikation.

Nachdem FDR2 die Spezifikation syntaktisch annahm und semantisch so interpretierte, wie ursprünglich intendiert, wurde ein großer Nachteil eines Model–Checkers deutlich: exponentiell wachsender Zustandsraum. Der Grund darin lag in der Verwendung eines Datentyps, der 15 Zustände annehmen konnte und dem *FILES*–Prozeß als vierfacher Parameter und dem *YPSERVER*–Prozeß als dreifacher Parameter diente. Die Anzahl der möglichen Benutzer war bereits auf einen zurückgeschraubt worden. Zur Lösung des Problems wurden der Datentyp *PERM* verallgemeinert und die übrige Parametrisierungsinformation auf die Kanäle abgewälzt.

Aus dem gleichen Grund mußten im *SERVICES*–Prozeß unsynchronisiert nebenläufige Subprozesse in Prozesse gewandelt werden, die über eine externe Auswahl angesprungen werden. Dies stellt jedoch keine Einschränkung des spezifizierten Systems dar, weil ohnehin immer nur ein Benutzer zur Zeit auf die Services zugreifen kann und der Gegenstand der Verifikation das Skript `CPYP` und nicht die Dienste des Mailservers war.

Eine weitere Möglichkeit, die Anzahl der Zustände zu verringern, ist das Umformulieren von „if – then – else"–Konstrukten in externe Auswahlen, was ebenfalls beim *YPSERVER*– und beim *FILES*–Prozeß angewandt wurde.

Es war nun möglich, auch mehrere Benutzer zu erzeugen (auf einem Pentium II 400 MHz mit 128 MB RAM und 256 MB Auslagerunsspeicher unter Linux bis zu 4 Benutzer), um die generelle Eignung für mehrere Benutzer zu zeigen, sollten aber schon zwei genügen. Das System hat dann 161556 Zustände und 372960 Zustandsübergänge. Bei drei Benutzern werden dann schon 3132150 Zustände und 7295763 Zustandsübergänge benötigt.

5.3 Refinement von `CPYP`

Um ein Refinement von `CPYP` zu erreichen, muß zunächst eine Variante des konkreten Systems geschaffen werden, die alle Kanäle nach außen hin verbirgt. Nach außen sichtbar sind lediglich Events, die auch im sicheren System Verwendung finden.

$$SYS2 = SYS \setminus (channels1 \cup channels2)$$

FDR2 bietet die Möglichkeit, zu beweisende Verfeinerungen bereits in der CSP–Datei anzugeben (assert), hier sind dies alle drei Arten des Refinements vom sicheren abstrakten Systems zum modifizierten konkreten System. Es handelt sich dabei legiglich um eine Arbeitserleichterung, die keine weitere Sematik zu der CSP–Spezifikation hinzufügt. Die Bedeutung der Refinementrelationen ist im Kapitel 5.1 erklärt.

$$SAFE\_SYS \sqsubseteq_T SYS2$$

$$SAFE\_SYS \sqsubseteq_F SYS2$$

$$SAFE\_SYS \sqsubseteq_{FD} SYS2$$

5.3.1 Gefundene Trace–Verletzungen

Da die leere Trace Teilmenge jeder anderen Trace ist, kann es passieren, daß das konkrete System — ohne ein Event zu produzieren — eine Verfeinerung des abstrakten Prozesses im Trace–Modell darstellt. Dies wurde durch das absichtliche Einfügen von Trace–Verletzungen im konkreten System an geeigneter Stelle überprüft, da nach der Erkennung dieser ein Debuggen der ausgeführten Trace möglich ist. Hier zeigte sich, daß das System wie erwünscht auch längere Traces als die des „STOP"–Prozesses ausführt. Diese Überprüfung ließe sich eleganter mit dem Failures–Refinement lösen. Zum Zeitpunkt dieses Tests stand uns leider noch keine Spezifikation zur Verfügung, die ein sinnvolles Failures–Refinement ermöglichte.

Das Trace–Refinement brachte zunächst einen Design–Fehler des realen Mailservers an den Tag. Der Homeverzeichnis–Dienst macht keine YP–Anfrage, falls der Benutzer nicht in der lokalen „auto.home"–Datei eingetragen ist. Somit können Benutzer, die neu im System angemeldet sind, im schlimmsten Fall 6 Stunden lang den Mail–Dienst nicht benutzen. Diese „Panne" soll jedoch schnellstmöglich behoben werden, in der Spezifikation wird das Problem umgangen, indem bei der Benutzerinitialisierung immer auch ein Eintrag in die lokale Homeverzeichnis–Datei vorgenommen wird (siehe dazu auch den Kommentar in Anhang B an entsprechender Stelle).

Die zweite Traceverletzung, auf diesem Wege gefunden wurde, resultierte aus einem konkurrierendem Zugriff von `CPYP` und des *USERINIT*–Prozesses auf die YP–Datenbank bzw. die lokalen Dateien. Das konkrete Problem war, daß `CPYP` bzgl. eines Benutzers eine YP–Anfrage stellt und die Antworten zwischenspeichert. Danach wurde dieser Benutzer neu initialisiert und erhielt schwächere Rechte. Darauf schrieb `CPYP` die alten zwischengespeicherten Werte in die neu erzeugten lokalen

Dateien, wonach der eigentlich nicht berechtigte Benutzer sich auf dem Mailserver einloggen durfte.

Dieses Beispiel dürfte in der Realität kaum vorkommen, da die Zeitspanne vom Auslesen der YP–Daten und dem Schreiben der lokalen Datei in `CPYP` unter einer Sekunde liegt, es zeigt aber sehr schön den Nutzen eines Trace–Refinements.

Um den konkurrierenden Zugriff auszuschließen, wurde ein Semaphor definiert (Kapitel 4.3), der sicherstellt, daß entweder die Benutzerinitialisierung oder `CPYP` auf die YP–Datenbank und die lokalen Dateien zugreift. Die Dienste des Mailservers, die ja ebenfalls lesend auf die YP–Datenbank zugreifen, benötigen übrigens keinen Semaphor, da sie nicht parallel zur Benutzerinitialisierung laufen können.

5.4 Erfolgreiche und nicht erfolgreiche Refinements

Nach den oben genannten Korrekturen war das Trace–Refinement erfolgreich, das konkrete System (und damit auch das Shellskript `CPYP` selbst) erfüllt also die geforderten Sicherheitsbedingungen.

Das Failures–Refinement war ebenfalls erfolgreich, was die Deadlock–Freiheit des Gesamtsystems beweist.

Das Failures–Divergence–Refinement war dagegen nicht erfolgreich. Der Grund dafür liegt interessanterweise in der Einführung des Semaphors. Ein Debuggen der erzeugten Trace zeigt, daß sich das System in einem Livelock befinden kann, bei dem `CPYP` permanent Daten kopiert, ohne eine Benutzerinitialisierung zuzulassen. `CPYP` kann den Semaphor immer wieder neu benutzen, der Semaphor verhält sich also „nicht fair“. In der Realität würde der Scheduler des zugrundeliegenden Betriebssystems diese Unfairness verhindern.

Als wenig aussagekräftiger Beweis für die Refinement–Checks ein Screenshot von FDR2:

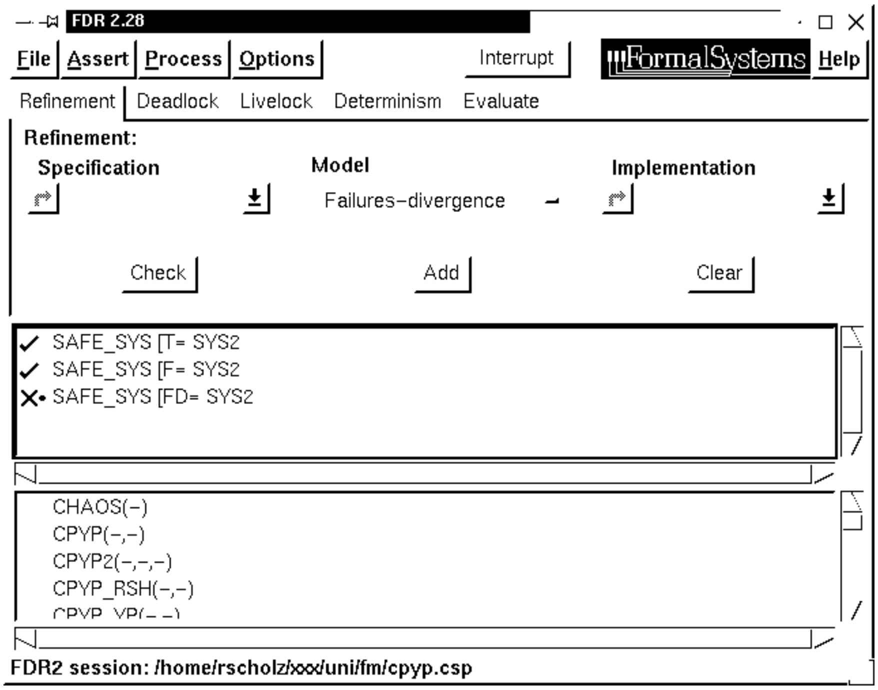

Abbildung 2: FDR2 nach Absolvierung aller Asserts

6 Fazit

Wir hatten beide in vorhergehenden Veranstaltungen die Spezifikationssprache CSP kennengelernt und waren durch die Arbeit im studentischen Projekt mit der Materie „Mailserver" vertraut. Die Einarbeitungszeit in diese Themen war also entsprechend kurz.

Da es für uns trotzdem die erste größere Verifikationsaufgabe war, die zudem noch auf einem realen Programm basierte, war es sehr schwierig, im voraus den zu bewältigenden Aufwand abzuschätzen. So stellte es sich schnell heraus, daß für die Verifikation des eigentlich zu untersuchenden Objekts (das Shellskript `CPYP`) die Umgebung, in dem es arbeitet, modelliert werden mußte. Ein Blick auf die erstellte CSP–Spezifikation zeigt, daß weit über die Hälfte der Spezifikation dafür aufgewendet wurde. Als Nebeneffekt kann dieses modellierte Environment im Projekt bei der Verifikation des gesamten Mailserver–Konzeptes verwendet werden.

Trotz der Erfahrung mit CSP kam es zu den in Kapitel 5.2 geschilderten Problemen, die jedoch glücklicherweise alle recht schnell gelöst werden konnten.

Die gefundenen Trace–Verletzungen in der Spezifikation zeigen deutlich, welchen Sinn eine Verifikation gerade in Safety- oder Security-relevanten Bereichen machen kann. Da hier meist schon eine Implementation in einer Programmiersprache vorliegt, ist CSP das geeignete Mittel, weil es eine direktere Umsetzung in eine Spezifikation ermöglicht und mittels Model–Checking eine Verifikation möglich ist.

Daß Model–Checking aber auch nur einen beschränkten Anwendungsbereich (und Zustandsbereich...) haben kann, wurde bei dieser Verifikationsaufgabe ebenfalls deutlich.

A Shellskript CPYP

```
#!/bin/sh
#
# Uebernahme der passwd, group und auto.home von NIS-Server
# auf FTS-Mailserver
# 14.01.1999 live@informatik.uni-bremen.de
#
# crontab :
# 21 3,9,15,21 * * * /fts/bin/cpyp 2>&1
#
############

# (1)
# Auskommentieren, falls NIS nicht benutzt werden soll. Bei
# Nichtbenutzung von NIS (ypcat) werden die Dateien vom
# NIS-Server $NIS vom Speicherort $NISPREFIX mittels
# "rsh $NIS cat $NISPREFIX/<file>" kopiert.
USENIS=1

# Name des NIS-Servers, nur noetig, falls $USENIS
# nicht definiert
NIS="aerobee"

# Speicherort der NIS-Dateien, nur noetig, falls
# $USENIS nicht definiert
NISPREFIX="/etc"

# "Shell", die No-Permission-Meldung ausgibt.
SHELL="/fts/bin/nologin"

PASSWD="passwd"
GROUP="group"
AUTOHOME="auto.home"

# temp. Dateien
YPCATTMP=`mktemp /tmp/cpyp.XXXXXX`
CPYPTMP=`mktemp /tmp/cpyp.XXXXXX`
AWKPRG=`mktemp /tmp/cpyp.XXXXXX`

# (2)
getnis () {
```

```
# (3)
    # lokale Dateien beruecksichtigen, falls vorhanden
    test -s /etc/$1.local && cat /etc/$1.local >> $CPYPTMP

    if [ $1 = $AUTOHOME ] ; then
        # Falls NIS benutzt werden soll, $AUTOHOME
        # braucht -k -Parameter
        test ! -z $USENIS && (COMMAND="ypcat -k $1" ; \
          $COMMAND >> $YPCATTMP)
    else
        # Falls NIS benutzt werden soll
        test ! -z $USENIS && (COMMAND="ypcat $1" ; \
          $COMMAND >> $YPCATTMP)
    fi

# (4)
    # Falls NIS nicht benutzt werden soll ODER #
    # NIS fehlgeschlagen
    test ! -s $YPCATTMP && \
      (COMMAND="rsh $NIS cat $NISPREFIX/$1" ; $COMMAND >> $YPCATTMP)

# (5)
    # Datei immer noch leer, also genereller Fehler
    if [ ! -s $YPCATTMP ] ; then
        echo "$0: Fehler beim Kopieren von $1"
        rm -f $YPCATTMP
        rm -f $CPYPTMP
    else

# (6)
        if [ $1 = $PASSWD ] ; then

            # Erzeugt awk-Skript, das bei best. Namen,
            # Gruppen, ... die urspruengliche Shell
            # beibehaelt und bei allen anderen die
            # NOLOGIN-Shell einsetzt.

            echo -e "(‘cat /etc/$PASSWD.allow‘) {print \$0} \n \
              !(‘cat /etc/$PASSWD.allow‘) {print \$1\":\"\$2\" \
              :\"\$3\":\"\$4\":\"\$5\":\"\$6\":$SHELL\"}" > $AWKPRG

            cat $YPCATTMP | awk -F":" -f $AWKPRG >> $CPYPTMP
```

```
# (7)
            echo -e "+::0:0:::$SHELL" >> $CPYPTMP
            rm $AWKPRG
        else
# (8)
            cat $YPCATTMP >> $CPYPTMP
        fi
        test -w $YPCATTMP && rm $YPCATTMP
        cp $CPYPTMP /etc.primaerer-mailserver/$1
        mv $CPYPTMP /etc.sekundaerer-mailserver/$1
        chmod 644 /etc.primaerer-mailserver/$1
        chmod 644 /etc.sekundaerer-mailserver/$1
    fi
}

# (9)
getnis $PASSWD
getnis $GROUP
getnis $AUTOHOME
```

A.1 Allow–Datei

Ein mögliches Beispiel für eine „Allow–Datei“, die dem Benutzer „wichtig“ sowie allen Mitliedern der Gruppe 111 einen Login erlaubt.

```
$1 ~ /^wichtig$/ || $3 ~ /^111$/
```

$1 steht dabei für das erste Feld der Paßwortdatei (User Name), $3 ist das dritte Feld, das die Group ID enthält. Die Felder in der Paßwortdatei sind bekanntermaßen mit einem „:“ getrennt.

B Spezifikation in FDR–Syntax

```
----------------------------------------------------------------
--
-- Korrekte Software durch formale Methoden
-- WS 98/99 Uni Bremen, FB 3 Informatik
--
--   Semesteraufgabe
--
--   Spezifikation in CSP und Verifikation mit FDR2
--   des Shell-Skripts CPYP (Projekt LiVE!)
--
-- Soenke Jarre, Raymond Scholz
--
-- $Id: cpyp.tex,v 1.17 1999/04/20 10:35:59 rscholz Exp $
--
----------------------------------------------------------------

-- Maximale User ID, bei 128 RAM + 256 Swap max. 9
MAXUID = 1

-- Menge der gueltigen User IDs
UID = {0..MAXUID}

-- Parameter fuer CPYP
USENIS = 1

-- Erlaubnis, Nichterlaubnis und Nichtvorhandensein
-- einer Aussage bzgl. eines Dienstes
datatype PERM = ok | nok | noentry

-- Aliases
RSH_PERM = PERM
YP_PERM = PERM
PLUS_PERM = PERM

-- die von CPYP uebertragenen Datenbanken
datatype DATABASE = passwd | group | autohome

-- moegliche Dienste des Mailservers
datatype SERVICE = mail | login | home | allow | plus

-- Rueckgabe der Dienste
```

```
datatype RESULT =  res_ok | res_nok

-- Kommunikation mit dem YP-Server
channel ypentry : UID.SERVICE.YP_PERM
channel ypquery : UID.SERVICE
channel ypanswer: UID.SERVICE.YP_PERM

-- Kommunikation mit dem YP-Server (alternativ)
channel rshquery :  UID.SERVICE
channel rshanswer : UID.SERVICE.RSH_PERM

-- Kommunikation mit den lokalen Dateien
channel filesentry : UID.SERVICE.PERM
channel filesquery : UID.SERVICE
channel filesanswer : UID.SERVICE.PERM

-- Kommunikation um "Plus-Zeile" einzutragen/abzufragen
channel plusentry : PLUS_PERM
channel plusquery
channel plusanswer : PLUS_PERM

-- die vom Mailserver angebotenen Dienste
channel mail_service : UID
channel login_service : UID
channel home_service : UID

-- Rueckmeldung der Dienste
channel ack : RESULT

-- Semaphor
channel sem_down, sem_up

-- Mengen, die spaeter zur Synchronisation und Hiding genutzt werden
channels1 = {| ypquery, ypanswer, rshquery, rshanswer, filesquery,
               filesanswer, filesentry, plusquery, plusanswer,
               plusentry, ypentry, sem_up, sem_down |}

channels2 = {| login_service, mail_service, home_service, ack |}

-- abstrahierte Berechtigungen der Benutzer
channel user_mail_ok, user_mail_nok,
        user_login_ok, user_login_nok
```

```
-- abstrahierte Aktionen / Antworten
channel user_wants_mail, user_reads_mail, user_no_mail,
        user_wants_login, user_logs_in, user_no_login

-- das sichere System
SAFE_SYS =

( user_mail_nok -> user_login_nok ->
  ( ( user_wants_mail -> user_no_mail -> SAFE_SYS )
    |~|
    ( user_wants_login -> user_no_login -> SAFE_SYS )
  )
)

|~|

( user_mail_ok -> user_login_nok ->
  ( ( user_wants_mail -> user_reads_mail -> SAFE_SYS )
    |~|
    ( user_wants_login -> user_no_login  -> SAFE_SYS )
  )
)

|~|

( user_mail_ok -> user_login_ok ->
  ( ( user_wants_mail -> user_reads_mail -> SAFE_SYS )
    |~|
    ( user_wants_login -> user_logs_in  -> SAFE_SYS )
  )
)


-- das konkrete System (siehe Zeichnung)
-- (CPYP erhaelt eine initiale UID als Parameter,
-- USENIS wie im Shell-Skript)
SYS =

(
  ( USER [| channels2 |] SERVICES) ||| CPYP(0,USENIS)
)

[| channels1 |]
```

```
(
  YPSERVER ||| FILES ||| PLUS ||| SEM(1)
)


-- einfacher Semaphor (benutzt von USERINIT und CPYP, verhindert
-- gleichzeitigen Zugriff auf YP--Datenbank bzw. lokale Dateien)
SEM(n) =

(n == 1)&sem_down -> SEM(0)

[]

(n == 0)&sem_up -> SEM(1)


-- die Benutzer, in zufaelliger Reihenfolge generiert
-- aus der Menge der UIDs
USER =

|~| u:UID @ USERINIT(u)


-- Initialisierung des erzeugten Benutzers
USERINIT(u) =

sem_down ->

(
  (ypentry.u!mail.noentry -> ypentry.u!login.noentry ->
   filesentry.u!mail.noentry ->
   filesentry.u!login.noentry ->
   filesentry.u!allow.noentry ->
   ypentry.u!home.noentry ->
   filesentry.u!home.noentry -> user_mail_nok ->
   user_login_nok -> SKIP
  )

  |~|

  (ypentry.u!mail.nok -> ypentry.u!login.nok ->
    (
```

```
    (filesentry.u!mail.nok ->
     filesentry.u!login.nok ->
     filesentry.u!allow.noentry -> SKIP
    )

    |~|

    (filesentry.u!mail.noentry ->
     filesentry.u!login.noentry ->
     filesentry.u!allow.noentry -> SKIP
    )
  ); user_mail_nok -> user_login_nok ->  ypentry.u!home.ok ->
  (
    (filesentry.u!home.noentry -> SKIP)
    |~|
    (filesentry.u!home.ok -> SKIP)
  )
)

|~|

(ypentry.u!mail.ok -> user_mail_ok -> ypentry.u!login.nok ->
  (
    (filesentry.u!mail.ok ->
      (
        (filesentry.u!login.nok ->
         filesentry.u!allow.noentry ->
         user_login_nok -> SKIP
        )
        |~|

        (filesentry.u!login.ok -> filesentry.u!allow.ok ->
         user_login_ok -> SKIP
        )
      )
    )

    |~|

    (filesentry.u!mail.noentry ->
     filesentry.u!login.noentry ->
     filesentry.u!allow.noentry -> user_login_nok -> SKIP
```

```
      )
    ) ;

-- so sollte ein guter Mailserver eigentlich arbeiten,
-- unserer tut es nicht...    *** vielleicht doch ???

--    ( ypentry.u!home.ok ->
--      (
--        (filesentry.u!home.noentry -> SKIP)
--        |~|
--        (filesentry.u!home.ok -> SKIP)
--      )
--    )

    ( ypentry.u!home.ok -> filesentry.u!home.ok -> SKIP  )
  )

); sem_up -> USERACTION(u)


-- moegliche Aktionen des Benutzers
USERACTION(u) =

(
  mail_service!u -> user_wants_mail -> ack?result ->
  if ( result == res_ok )
  then ( home_service!u -> ack?result ->
         if ( result == res_ok  )
         then ( user_reads_mail -> USER )
         else user_no_mail -> USER
       )
  else ( user_no_mail -> USER )
)

|~|

(
  login_service!u -> user_wants_login -> ack?result ->
  if ( result == res_ok  )
  then ( home_service!u -> ack?result ->
         if ( result == res_ok  )
         then ( user_logs_in -> USER )
         else user_no_login -> USER
```

```
       )
  else ( user_no_login -> USER )
)


-- der YP-Server, zunaechst Initialisierung
YPSERVER  =

||| u:{ 0..MAXUID } @ YPSERVER2(u,noentry,noentry,noentry)

-- der wirkliche YP-Server
YPSERVER2(u, mailstatus, loginstatus, homestatus) =

ypquery.u?login -> ypanswer.u!login.loginstatus ->
YPSERVER2(u,mailstatus,loginstatus,homestatus)

[]

ypquery.u?mail -> ypanswer.u!mail.mailstatus ->
YPSERVER2(u,mailstatus,loginstatus,homestatus)

[]

ypquery.u?home -> ypanswer.u!home.homestatus ->
YPSERVER2(u,mailstatus,loginstatus,homestatus)

[]

rshquery.u?login -> rshanswer.u!login.loginstatus ->
YPSERVER2(u,mailstatus,loginstatus,homestatus)

[]

rshquery.u?mail -> rshanswer.u!mail.mailstatus ->
YPSERVER2(u,mailstatus,loginstatus,homestatus)

[]

rshquery.u?home -> rshanswer.u!home.homestatus ->
YPSERVER2(u,mailstatus,loginstatus,homestatus)

[]
```

```
-- Eintrag eines neuen Benutzers

ypentry.u?mail.mailperm ->
YPSERVER2(u, mailperm, loginstatus, homestatus)

[]

ypentry.u?login.loginperm ->
YPSERVER2(u, mailstatus, loginperm, homestatus)

[]

ypentry.u?home.homeperm ->
YPSERVER2(u, mailstatus, loginstatus, homeperm)


-- die lokalen Dateien,  zunaechst Initialisierung
FILES =

||| u:{ 0..MAXUID } @ FILES2(u,noentry,noentry,noentry,noentry)

-- die wirklichen Files
FILES2(u, mailstatus, loginstatus, allowstatus, homestatus) =

filesquery.u?mail -> filesanswer.u!mail.mailstatus ->
FILES2(u,mailstatus,loginstatus,allowstatus,homestatus)

[]

filesquery.u?login -> filesanswer.u!login.loginstatus ->
FILES2(u,mailstatus,loginstatus,allowstatus,homestatus)

[]

filesquery.u?allow -> filesanswer.u!allow.allowstatus ->
FILES2(u,mailstatus,loginstatus,allowstatus,homestatus)

[]

filesquery.u?home -> filesanswer.u!home.homestatus ->
FILES2(u,mailstatus,loginstatus,allowstatus,homestatus)

[]
```

```
-- Eintrag eines neuen Benutzers

filesentry.u?mail.mailperm ->
FILES2(u, mailperm, loginstatus, allowstatus, homestatus)

[]

filesentry.u?login.loginperm ->
FILES2(u, mailstatus, loginperm, allowstatus, homestatus)

[]

filesentry.u?allow.allowperm ->
FILES2(u, mailstatus, loginstatus, allowperm, homestatus)

[]

filesentry.u?home.homeperm ->
FILES2(u, mailstatus, loginstatus, allowstatus, homeperm)


-- Die Plus--Zeile in der Paßwort-Datei (einzelner Prozeß, weil
-- nicht von einem Usereintrag abhaengig
PLUS =

PLUS2(ok) -- initial ist ein Plus in der Paßwortdatei vorhanden

PLUS2(plusstatus) =

plusquery -> plusanswer!plusstatus ->
PLUS2(plusstatus)

[]

plusentry?plusperm ->  -- von cpyp
PLUS2(plusperm)


-- die Services des Mailservers, nur einer zur Zeit (state space...)
SERVICES = MAIL [] LOGIN [] HOME
```

```
-- der Mailservice
MAIL =

mail_service?u -> filesquery.u!mail -> filesanswer.u?mail.status1 ->
if ( status1 == ok  )
then ( ack!res_ok -> SERVICES  )
else (
       plusquery -> plusanswer?status2 ->
       if ( status2 == ok )
       then (
              ypquery.u!mail -> ypanswer.u?mail.status3 ->
              if ( status3 == ok  )
              then ( ack!res_ok -> SERVICES )
              else ( ack!res_nok -> SERVICES )
            )
       else ( ack!res_nok -> SERVICES )
     )


-- der Loginservice
LOGIN  =

login_service?u -> filesquery.u!login ->
filesanswer.u?login.status1 ->
if ( status1 == ok  )
then ( ack!res_ok -> SERVICES  )
else (
       if ( status1 == nok  )
       then ( ack!res_nok -> SERVICES  )
       else ( plusquery -> plusanswer?status2 ->
              if ( status2 == ok )
              then (
                     ypquery.u!login -> ypanswer.u?login.status3 ->
-- seltsam, siehe Erlaeuterung im Text
                     ack!res_nok -> SERVICES
                   )
              else ( ack!res_nok -> SERVICES )
             )
     )


-- der Homeverzeichnis-Service, momentan keine YP-Anfragen, da der
-- reale Mailserver dies auch nicht tut
```

```
HOME =

home_service?u -> filesquery.u!home -> filesanswer.u?home.status ->
if ( status == ok  )
then ( ack!res_ok -> SERVICES )
else ( ack!res_nok -> SERVICES )


-- Umsetzung des Skripts CPYP in CSP
CPYP(u,usenis) =
if (u <= MAXUID)
then (
       sem_down ->
       CPYP2(u,usenis,passwd); CPYP2(u,usenis,group);
       CPYP2(u,usenis,autohome);
-- bel. Wartezeit, hier nicht modelliert
       sem_up -> CPYP(u+1,usenis)
     )
else CPYP(0,usenis)

-- Datenaquisition, fehlertolerant
CPYP2(u, usenis, database) =

-- lesen der lokalen Dateien, hier nicht modelliert
if (usenis == 1)
then (
       CPYP_YP(u, database) [] CPYP_RSH(u, database)
     )
else CPYP_RSH(u,database) -- wenn ypcat nicht geht, dann rsh


-- entspricht der Funktion cpyp () im Skript wenn YP benutzt wird
CPYP_YP(u,database) =

if (database == passwd)
then (
       ypquery.u!mail -> ypanswer.u?mail.status1 ->
       ypquery.u!login -> ypanswer.u?login.status2 ->
       filesquery.u!allow -> filesanswer.u?allow.status3 ->

       (
         if (status3 == ok and status2 == nok)  -- noentries abfangen
         then (
```

```
                filesentry.u!login.ok ->
                filesentry.u!mail.ok -> SKIP
              )
         else (
                filesentry.u!login.nok ->
                filesentry.u!mail.status1 -> SKIP
              )
       ); plusentry!ok -> SKIP
     )
else if (database == group)
     then ( -- nicht modelliert, siehe Text
            SKIP
          )
     else if (database == autohome)
          then (
                 ypquery.u!home -> ypanswer.u?home.status4 ->
                 filesentry.u!home.status4 -> SKIP
               )
          else STOP --hoppla


-- entspricht der Funktion cpyp () im Skript wenn RSH benutzt wird
CPYP_RSH(u,database) =

if (database == passwd)
then (
       rshquery.u!mail -> rshanswer.u?mail.status1 ->
       rshquery.u!login -> rshanswer.u?login.status2 ->
       filesquery.u!allow -> filesanswer.u?allow.status3 ->

       (
         if (status3 == ok and status2 == nok)  -- noentries abfangen
         then (
                filesentry.u!login.ok ->
                filesentry.u!mail.ok -> SKIP
              )
         else (
                filesentry.u!login.nok ->
                filesentry.u!mail.status1 -> SKIP
              )
       ); plusentry!ok -> SKIP
     )
else if (database == group)
```

```
    then ( -- nicht modelliert, siehe Text
           SKIP
         )
    else if (database == autohome)
         then (
                rshquery.u!home -> rshanswer.u?home.status4 ->
                filesentry.u!home.status4 -> SKIP
              )
         else STOP --hoppla


-- das konkrete System ohne die internen Channel-Kommunikationen
SYS2 = SYS \ union(channels1, channels2)

-- Trace-Refinement
assert  SAFE_SYS [T= SYS2

-- Failures-Refinement
assert  SAFE_SYS [F= SYS2

-- Failures-Divergence-Refinement, klappt nicht, weil SYS nicht fair
-- (wg. Semaphor)
assert  SAFE_SYS [FD= SYS2
```

Literatur

[1] C.A.R. Hoare. Communicating Sequential Processes. Prentice–Hall International, Englewood Cliffs, New Jersey, 1985.

[2] Man–Page: GNU Bourne–Again Shell. Free Software Foundation, 1989-1996.

[3] Man–Page: AWK Pattern Scanning and Processing Language. Free Software Foundation, 1996

[4] FDR2 Online–Help und Manual. FDR2 2.28. Formal Systems Europe (Ltd.), 1992-1997.

[5] Günter Feldmann et. al. Vereinbarung zur Informationstechnik–Infrastruktur im Fachbereich 3. Universität Bremen, 1997