CVS–Dokumentation

Tino Schwarze, TeeGee

September 2000

Inhaltsverzeichnis

1 Wozu ist CVS da?

CVS ist ein System zur Versionskontrolle. Man kann es benutzen, um die Geschichte seiner Sourcecodes aufzuzeichnen. Dabei arbeitet CVS nur auf der Basis von Textdateien, stellt also keine Anforderungen in Bezug auf die verwendete Programmiersprache oder Ähnliches. Binäre Dateien können auch von CVS verwaltet werden, allerdings verhältnismäßig ineffizient (jede Version wird komplett gespeichert, siehe Abschnitt 10.1, Seite 21).

Wenn eine Software modifiziert wird, schleichen sich Fehler ein, die manchmal erst viel später auftauchen. Mit Hilfe von CVS kann man jetzt genau zurückverfolgen, wann der Fehler eingebaut wurde, da man *jede* Version der betreffenden Datei leicht wiederherstellen kann.

Man könnte natürlich von jeder Version jeder Datei Kopien erzeugen; das würde aber eine Menge Plattenplatz verschwenden, da CVS nur jeweils die Unterschiede zwischen aufeinanderfolgenden Versionen speichert. CVS verwaltet außerdem für jede Source-Datei genau eine große Datei, die alle Informationen und Versionen enthält. Weiterhin ist das manuelle Kopieren der einzelnen Versionen ein nicht zu unterschätzender organisatorischer Aufwand.

Ferner unterstützt CVS die Arbeit mehrerer Leute an einem Projekt, indem es die einzelnen Entwickler voneinander abkoppelt, so daß sie unabhängig voneinander am Projekt arbeiten können, da jeder an seiner eigenen Kopie des Projektes arbeitet. Allerdings müssen die Entwickler nach wie vor miteinander kommunizieren um grundlegende Belange des Projektes zu klären.

2 Wozu ist CVS nicht da?

CVS kann eine Menge für einen tun, aber es kann natürlich nicht alles für jeden erledigen.

- CVS ist kein Build-System.

  Es diktiert nicht, wie man seine Sourcecodes kompiliert. Man muß seine Makefiles nach wie vor selbst schreiben und das kann eine Menge Arbeit sein. Die dazu verwendeten Dateien, Skripte und Tools kann man allerdings wieder unter die Kontrolle von CVS stellen und damit die Versionskontrolle auf diese ausdehnen. (Es wäre zum Beispiel möglich, daß man mit einer neuen Version seiner Software auch ein neues System zum Kompilieren einführt und dann möchte man natürlich ältere Versionen trotzdem noch kompilieren können.)

- CVS ist kein Ersatz für ein ordentliches Management.

  Manager und Projektleiter müssen immer noch mit den Entwicklern reden, um sie über Zeitpläne, „merge points“ (Verschmelzungspunkte), „branch names“ (Namen von Projektzweigen) und Release-Daten zu informieren.

- CVS ist kein Ersatz für die Kommunikation zwischen den Entwicklern.

  CVS kann Konflikte nur auf textueller Basis feststellen, d. h. wenn zwei Entwickler an der gleichen Stelle einer Datei unterschiedliche Änderungen vorgenommen haben. Es kann nicht feststellen, daß jemand ein API geändert hat und jetzt 10 verschiedene Dateien geändert werden müßten. CVS kann *nicht* bei der Auflösung solcher Konflikte helfen. Diese müssen von den Entwicklern vermieden werden, indem man sich auf feste, verbindliche Schnittstellen einigt.

- CVS unterstützt Bug-Tracking nicht direkt.

  Es gibt Möglichkeiten, Bug-Tracking mit CVS zu verheiraten, allerdings muß die Integration selbst hergestellt werden. CVS bietet jedoch einige Schnittstellen, die dafür verwendet werden können.

- CVS ist kein automatisches Testprogramm.

  Automatisches Testen kann in CVS integriert werden, wird aber nicht direkt unterstützt.

3 Ein einfaches Beispiel

Um die Funktionsweise von CVS zu erläutern, betrachten wir ein kleines Beispiel. Zuerst muß man sich vor Augen führen, daß CVS alle Dateien in einem zentralen „Repository“ speichert. Wir gehen hier davon aus, daß dieses Repository schon eingerichtet wurde. (Mehr dazu später in Abschnitt 4, Seite 6.)

Wir nehmen an, daß wir an einem Compiler tc arbeiten. Der Sourcecode besteht aus einigen C-Dateien und einem Makefile.

3.1 Den Sourcecode holen

Zuerst mußsen wir uns eine Arbeitskopie des Sourcecodes holen. Dafür wird das checkout-Kommando verwendet:

```
$ cvs checkout tc
```

Es erzeugt ein Verzeichnis tc und füllt es mit den Sourcedateien. (Die Dateien werden aus dem Repository extrahiert und kopiert.)

```
$ cd tc
$ ls
CVS Makefile backend.c driver.c frontend.c parser.c
```

Das CVS-Verzeichnis wird intern von CVS benutzt. Normalerweise sollte man darin nichts verändern.

Jetzt können wir unseren Lieblingseditor nehmen und z. B. in backend.c herumeditieren. Ein paar Stunden später haben wir den Compiler um Optimierung erweitert.

3.2 „Commit" – Veränderungen einspielen

Nachdem wir überprüft haben, daß sich der Compiler noch kompilieren läßt, können wir uns entscheiden, eine neue Version von backend.c zu erzeugen. Dadurch wird das neue backend.c im Repository gespeichert und ist für alle verfügbar, die das gleiche Repository verwenden.

```
$ cvs commit backend.c
```

CVS ruft jetzt einen Editor auf, damit wir eine Mitteilung für das Log (Protokoll) schreiben können. Wir geben „Optimierung hinzugefügt" ein, speichern die temporäre Datei und verlassen den Editor. (Die Umgebungsvariablen `CVSEDITOR`, `EDITOR` bzw. `VISUAL` werden konsultiert, um den zu verwendenden Editor zu ermitteln.)

Alternativ können wir auch mit

```
$ cvs commit -m "Optimierung hinzugefügt" backend.c
```

die Log-Message direkt angeben.

Wenn man den Editor verwendet, sieht man eine Liste der veränderten Dateien. Es kann manchmal vorkommen, daß CVS Dateien auflistet, die gar nicht verändert wurden, sondern nur einen neuen Timestamp erhalten haben. Das wird im weiteren Verlauf des Commits festgestellt und keine neue Version erzeugt.

3.3 Aufräumen

Bevor wir uns anderen Aufgaben zuwenden, wollen wir wieder aufräumen. Wir könnten einfach das Verzeichnis tc löschen. Eine bessere Möglichkeit ist allerdings, das release-Kommando von CVS zu verwenden:

```
$ cd ..
$ cvs release -d tc
M driver.c
? tc
You have [1] altered files in this repository.
Are you sure you want to release (and delete) directory 'tc': n
** 'release' aborted by user choice
```

Das release-Kommando überprüft, ob alle lokalen Veränderungen in das Repository eingespielt wurden. Mit der Option -d wird die Arbeitskopie gleich nach erfolgreicher Überprüfung gelöscht.

Im obigen Beispiel wurden mehrere Zeilen ausgegeben. „? tc" bedeutet, daß CVS nichts über die Datei tc weiß. Das ist nicht schlimm, denn diese Datei ist der kompilierte Compiler. Er sollte *nicht* unter CVS-Kontrolle gestellt werden.

Die Zeile „M driver.c" hat einen ernsteren Hintergrund: Es bedeutet, daß die Datei driver.c modifiziert wurde nachdem sie ausgecheckt wurde. Das release-Kommando berichtet immer, wie viele modifizierte Dateien in der Arbeitskopie gefunden wurden und fragt nach Bestätigung vor dem Löschen.

3.4 Unterschiede Anzeigen

Wir können uns leider nicht erinnern, driver.c verändert zu haben. Deshalb wollen wir sehen, was damit passiert ist.

```
$ cd tc
$ cvs diff driver.c
```

Dies bewirkt, daß die Arbeitskopie mit dem Repository verglichen wird und man eine Auflistung der Unterschiede erhält. Beim Ansehen der Ausgabe fällt uns ein, daß wir eine Kommandozeilen-Option zum Aktivieren der Optimierung hinzugefügt haben. Also checken wir dies ein und geben dann die Arbeitskopie frei:

```
$ cvs commit -m ”Optimierung hinzugefügt” driver.c
Checking in driver.c:
/usr/local/cvsroot/tc/driver.c,v <- driver.c
new revision: 1.2; previous revision: 1.1
done
$ cd ..
$ cvs release -d tc
? tc You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory ‘tc’: y
```

4 Das Repository

Das CVS „Repository" ist der zentrale Aufbewahrungsort für alle Dateien und Verzeichnisse, die der Versionskontrolle unterliegen. Es stellt quasi die Master-Kopie dar.

Normalerweise wird auf keine Datei im Repository direkt zugegriffen. Stattdessen werden CVS-Kommandos verwendet, um eine eigene Kopie der Dateien in ein Arbeitsverzeichnis zu bekommen und dann wird mit dieser Kopie gearbeitet. Wenn man mit einer Anzahl von Veränderungen fertig ist, werden diese wieder in das Repository eingecheckt (committed). Das Repository enthält dann die Veränderungen, wann diese gemacht wurden, was verändert wurde und einige Informationen mehr. Das Repository ist dabei unabhängig vom Arbeitsverzeichnis und sollte es auch sein.

Das Repository kann sich auf dem lokalen Computer oder auf entfernten Rechnern befinden. Wenn man ein Projekt das erste Mal auscheckt, muß man CVS mitteilen, wo sich das Repository befindet. Dies geschieht mit der Umgebungsvariable `CVSROOT` oder der Option `-d`. In jedem Verzeichnis, das Dateien unter Versionskontrolle enthält, legt CVS ein `CVS`-Unterverzeichnis an und merkt sich dort u. a. die Position des Repository in der Datei `CVS/Root`, so daß man es später nicht noch einmal angeben muß. Der Ausdruck `$CVSROOT` soll im Weiteren als Platzhalter für die Position des Repository verwendet werden.

Man sollte nicht vergessen, daß sich in einem CVS-Repository viele verschiedene (unabhängige) Projekte befinden können.

Nähere Informationen zum Repository finden sich in Abschnitt 11, Seite 23.

5 Revisionen

In den meisten Fällen braucht man sich um die Revisionsnummern keine Gedanken zu machen. CVS weist den Dateien im Repository Revisionsnummern wie 1.1, 1.2 usw. zu.

Wenn man den Überblick über die Revisionen von mehr als einer Datei behalten möchte, verwendet man „Tags“ (siehe 5.3). Das sind symbolische Revisionen, die den numerischen Revisionen jeder Datei zugeordnet werden.

5.1 Revisionsnummern

Jede Version einer Datei hat eine eindeutige Revisionsnummer wie 1.1, 1.2, 1.3.2.2 oder sogar 1.3.2.2.4.5. Sie besteht immer aus einer geraden Anzahl von ganzen Zahlen, die durch Punkte getrennt sind.

Standardmäßig ist Revision 1.1 die erste einer Datei. Jede nachfolgende Revision wird erzeugt, indem die am weitesten rechte Zahl um eins erhöht wird.

Zu Revisionen mit mehr als einem Punkt siehe Abschnitt 9.

5.2 Versionen, Revisionen und Releases

Um Verwirrungen zu vermeiden, wird hier nicht von Versionen gesprochen, sondern von Revisionen (Version einer Datei) und Releases (Version eines Softwarepaketes).

5.3 Tags

Die Revisionsnummern sind unabhängig von den Releasenummern der Software, die im CVS entwickelt wird.

Das tag-Kommando (sprich: „Täg“) kann verwendet werden, um einer bestimmten Revision einer Datei einen symbolischen Namen zu geben. Mit der Option -v des status-Kommandos kann man sich alle Tags einer Datei anzeigen lassen.

Tag-Namen müssen mit einem Groß- oder Kleinbuchstaben beginnen und bestehen aus Buchstaben, Zahlen, „-“ und „_“. Die Namen BASE und HEAD sind reserviert.

Man sollte eine einheitliche Konvention für die Benennung von Tags einführen, z. B. für eine Programmrelease „Name-Version“ wie in „cvs-1_9“, damit man nicht dauernd raten muß, ob es jetzt „cvs1-9“, „cvs-1_9“ oder „cvs_1-9“ heißt.

Normalerweise wird nicht nur eine einzelne Datei „getaggt“, sondern meist werden alle Dateien, die zu einem bestimmten Teil der Software (wie z. B. einem Modul) gehören, mit einem gemeinsamen Tag versehen, z. B. wenn eine neue Softwarerelease herausgegeben wird.

Mit

```
$ cd tc
$ cvs tag tc-1_0
```

erzeugt man einen Tag „tc-1_0“ und kann später unter Verwendung dieses Namens auf genau diese Revision zugreifen. Indem man die Option -c angibt, veranlasst man CVS, zu prüfen, ob sich im Arbeitsverzeichnis die aktuelle Version befindet, d. h. ob nicht im Repository oder an der lokalen Kopie Veränderungen vorgenommen wurden. Manchmal möchte man die neueste Version „taggen“, oft jedoch genau die Version, die sich im Arbeitsverzeichnis befindet; wurde diese verändert, so muß sie zuerst eingecheckt werden.

Das checkout-Kommando hat eine Option -r, mit der eine bestimmte Revision ausgecheckt werden kann:

```
$ cvs checkout -r tc-1_0 tc
```

Somit ist es möglich, eine ältere Version einer Software zu erhalten, zum Beispiel wenn dort ein Bug enthalten ist, der mittlerweile behoben wurde.

Abbildung 1 illustriert, wie man sich einen symbolischen Tag „tc-1_0“ vorstellen kann.

Tags sollten normalerweise nicht gelöscht oder verschoben werden. Es gibt aber Situationen, in denen dies erwünscht ist – zum Beispiel wenn ein Tag versehentlich angelegt wurde oder wenn man immer einen Tag für die letzte stabile Version einer Software haben möchte. Um Tags zu löschen oder zu verschieben, gibt es die Optionen -d und -F:

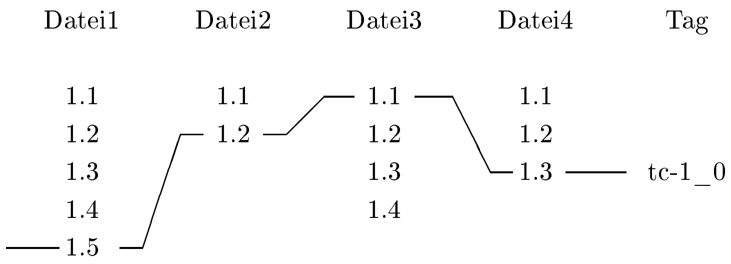

Abbildung 1: Ein Tag auf mehreren Dateien

```
$ cd tc
Wir haben uns geirrt - der Tag sollte nicht tc-1_0 heissen.
$ cvs tag -d tc-1_0
$ cvs tag release-1-0
$ cvs tag latest-stable-release
Einige Wochen später wird Release 1.1 herausgegeben.
$ cvs tag release-1-1
$ cvs tag -F latest-stable-release
```

6 Ein neues Projekt beginnen

Da das Verschieben von Dateien im CVS etwas kompliziert und manchmal verwirrend ist, sollte man, bevor man ein neues Projekt in die Hände von CVS gibt, dessen Verzeichnisstruktur neu überdenken und evtl. reorganisieren.

6.1 Ein neues Repository erzeugen

Um ein Repository zu erzeugen, sucht man sich zuerst einen Rechner und genügend Speicherplatz. Die Anforderungen an die Rechenleistung und den Speicherplatz sind nicht hoch - heutzutage kann jeder normale Rechner ein CVS-Repository aufnehmen. Man muß nur genügend Platz einplanen – eine Faustregel besagt drei mal so viel Platz wie der Code voraussichtlich beanspruchen wird. Für kleinere Projekte kann man also mit einigen hundert Kilobyte bis zu mehreren Megabyte rechnen.

Dann erzeugt man das Repository mit dem Kommando cvs init:

```
$ cvs -d /pfad/zum/repository/ init
```

Dabei ist CVS vorsichtig und überschreibt keine Dateien. Außerdem wird „History Logging“ aktiviert, indem eine Datei CVSROOT/history im Repository angelegt wird. (Siehe dazu Abschnitt 8.2, Seite 17)

6.2 Repository mit vorhandenen Dateien füllen

Wenn man bereits ein Projekt hat, daß man jetzt unter CVS-Kontrolle stellen möchte, kann man das import-Kommando von CVS benutzen.

Wir nehmen an, daß sich die Dateien, die wir ins CVS einspielen möchten, im Verzeichnis WDIR befinden. Wir möchten sie im Repository unter TeeGee/WDIR einordnen. Die Umgebungsvariable `CVSROOT` sollte auf das vorbereitete Repository zeigen, z. B. /snoopy/CVSROOT. Jetzt können die Dateien in das Repository aufgenommen werden:

```
$ cd WDIR
$ cvs import -m "Jetzt im CVS" TeeGee/WDIR TeeGee start
```

Die Zeichenkette „TeeGee“ ist ein sogenannter „Vendor Tag“ und „start“ ist ein „Release Tag“. Sie sind hier von keiner speziellen Bedeutung, werden aber von CVS verlangt.

Jetzt überprüfen wir, ob alles geklappt hat:

```
$ cd ..
$ mv WDIR WDIR.orig
$ cvs checkout TeeGee/WDIR
$ diff -r WDIR.orig WDIR
$ rm -r WDIR.orig
```

Dem checkout-Kommando kann man entweder einen Modulnamen oder einen Pfad relativ zu `CVSROOT` angeben (wie oben geschehen).

Falls einige Dateien binär sind, muß man noch spezielle Vorkehrungen vor dem Import treffen (siehe dazu Abschnitt 13.3, Seite 31).

6.3 Ein Modul definieren

Module sind nicht unbedingt notwendig, um CVS zu verwenden, sie sind allerdings sehr nützlich und erleichtern die Übersicht.

Um die Moduldefinition zu ändern, holt man sich zuerst mit

```
$ cvs checkout CVSROOT/modules
```

eine aktuelle Kopie der Datei modules.

Dann editiert man diese und fügt z. B. eine Zeile

```
project_w TeeGee/WDIR
```

hinzu. Damit hat man ein Modul project_w definiert, das man direkt auschecken kann.

Jetzt checkt man die neue modules-Datei ein:

```
$ cvs commit -m "project_w hinzugefügt" modules
```

und gibt die Datei wieder frei:

```
$ cd ..
$ cvs release -d CVSROOT
```

Nun kann man das Modul direkt auschecken:

```
$ cvs checkout project_w
```

und erhält ein Verzeichnis project_w, daß die Dateien aus dem Repository-Unterverzeichnis TeeGee/WDIR enthält.

6.4 Die Datei modules

Diese Datei liegt im CVS-Repository im Verzeichnis CVSROOT und enthält alle Moduldefinitionen des Repositories.

Die Datei ist zeilenorientiert und ermöglicht es, auf drei Arten ein Modul zu definieren:

1. Alias Module

   Syntax: `MODULNAME -a ALIAS [ALIAS[...]]`

   Dies ist der einfachste Weg, ein Modul zu definieren. Man vergibt für ein Verzeichnis einfach einen Aliasnamen.

   Einzelne Verzeichnisse können explizit ausgeschlossen werden, indem man sie mit `!VERZEICHNIS` vor die Liste der Aliases schreibt.

2. „normale“ Module

   Syntax: `MODULNAME [Optionen] VERZEICHNIS [Dateien...]`

   Die einfachste Form dieser Variante ist „`MODULNAME VERZEICHNIS`“. Man definiert ein Modul, daß aus dem angegebenen Verzeichnis (relativ zum `CVSROOT`) besteht. Wenn man also

   ```
   einmodul einverzeichnis
   ```

   angibt, führt ein

   ```
   $ cvs checkout einmodul
   ```

   dazu, daß ein Verzeichnis einmodul angelegt wird, daß die Dateien (und Verzeichnisse) aus $CVSROOT/einverzeichnis enthält.

   Gibt man zusätzlich Dateien an, werden nur diese ausgecheckt.

3. Ampersand-Module

   Syntax: `MODULNAME [Optionen] &MODULNAME [&MODULNAME...]`

   Damit kann man andere Module zu einem Modul hinzufügen. Die Angabe von

   ```
   grossesmodul &modul1 &modul2
   ```

   bewirkt, daß beim Checkout von grossesmodul die Verzeichnisse grossesmodul/modul1 und grossesmodul/modul2 angelegt und jeweils mit den Dateien aus Modul 1 und Modul 2 gefüllt werden.

Normalen und Alias-Modulen kann man mit der Option -d noch das Verzeichnis angeben, daß beim Checkout anstatt des Modulnamens erzeugt werden soll. Weitere Optionen ermöglichen es, eigene Programme zu starten, wenn ein Modul ausgecheckt, geupdated etc. wird (siehe dazu die CVS-Dokumentation).

7 Dateien hinzufügen und entfernen

Im Verlauf eines Projektes muß man natürlich Dateien hinzufügen oder entfernen. Das Gleiche gilt für Verzeichnisse. Generell sollte man im Hinterkopf behalten, daß man keine irreversible Veränderung vornehmen möchte, sondern daß CVS aufzeichnen soll, daß eine Veränderung stattgefunden hat (in der gleichen Weise als würde man eine Datei modifizieren).

Das genaue Vorgehen mit CVS variiert je nach Situation.

7.1 Dateien hinzufügen

Um eine neue Datei zu einem Verzeichnis hinzuzufügen, geht man wie folgt vor:

1. Man benötigt eine Arbeitskopie des entsprechenden Verzeichnisses. (siehe 3.1, Seite 4)

2. Man legt die neue Datei im Arbeitsverzeichnis an.

3. Mit

   ```
   $ cvs add <DATEINAME>
   ```

   teilt man CVS mit, daß man die Datei unter Versionskontrolle stellen möchte. Falls es sich um eine Datei mit binären Daten (also nicht reinen Text) handelt, fügt man die Option -kb hinzu (siehe auch 10.1, Seite 21).

4. Mit

   ```
   $ cvs commit <DATEINAME>
   ```

wird die Datei in das Repository eingecheckt. Bis zu diesem Schritt können andere Entwickler die Datei noch nicht sehen (cvs add markiert die Datei nur für das Hinzufügen durch commit).

Die Dateien oder Verzeichnisse, die mit add hinzugefügt werden sollen, müssen bereits existieren. Um eine komplette Verzeichnishierarchie zum Repository hinzuzufügen, verwendet man besser das import-Kommando (siehe 6.2, Seite 10).

Mit der Option -m kann man eine Beschreibung der Datei angeben. Falls die Option fehlt, wird man *nicht* nach einer Beschreibung gefragt.

Falls man mit verschiedenen Branches arbeitet, wird die Datei zum aktuellen Branch hinzugefügt (siehe Abschnitt 9, Seite 17).

7.2 Dateien entfernen

Mit der Zeit verändern sich die Verzeichnisse. Neue Dateien werden hinzugefügt, alte werden überflüssig und gelöscht. Man möchte aber nach wie vor in der Lage sein, eine alte Version der Software komplett wiederherzustellen.

Um eine Datei im CVS zu löschen, geht man wie folgt vor:

1. Man muß sicherstellen, daß die lokale Datei nicht modifiziert wurde, z. B. mit dem status-Kommando. Entfernt man die Datei, ohne die Modifikationen vorher in das Repository einzuspielen, gehen diese unwiederbringlich verloren.
2. Die Datei aus dem Arbeitsverzeichnis entfernen, z. B. mit rm.
3. Mit

   ```
   $ cvs remove <DATEINAME>
   ```

   wird CVS mitgeteilt, daß man die Datei wirklich löschen möchte. (Tut man dies nicht, wird sie beim nächsten update aus dem Repository wiederhergestellt.)
4. Mit

   ```
   $ cvs commit <DATEINAME>
   ```

   wird die Datei tatsächlich im Repository gelöscht.

Wenn man das Entfernen einer Datei eincheckt, merkt sich CVS die Tatsache, daß die Datei nicht länger existiert. Es ist möglich, daß eine Datei nur in einigen Branches existiert und in anderen nicht. Man kann eine Datei auch später noch einmal hinzufügen.

Man kann das Löschen der Datei im Arbeitsverzeichnis auch CVS antragen, indem man die Option -f verwendet. Dann braucht man die Datei nicht von Hand zu löschen.

Wenn man eine Datei versehentlich gelöscht, aber nicht mit remove markiert hat, kann man sie mittels cvs update wiederherstellen.

Wenn man eine Datei gelöscht und mit remove markiert hat, aber die Änderung noch nicht in das Repository eingecheckt hat, kann man die Datei mittels cvs add wiederherstellen.

Wie beim Hinzufügen wirkt sich das Löschen einer Datei nur auf den aktuellen Branch aus.

7.3 Verzeichnisse entfernen

Im Prinzip ist das Entfernen von Verzeichnissen vergleichbar mit dem Entfernen von Dateien – man möchte das Verzeichnis nicht mehr im aktuellen Arbeitsverzeichnis aber es muß durchaus in älteren Versionen wieder auftauchen.

Um ein Verzeichnis zu entfernen, löscht man zuerst alle Dateien darin. Das Verzeichnis selbst kann nicht entfernt werden! Stattdessen gibt man die Option -P bei den CVS-Kommandos update, checkout und export an, um leere Verzeichnisse aus den Arbeitsverzeichnissen zu entfernen. Am Besten gibt man immer -P an. Wenn man jetzt explizit ein leeres Verzeichnis behalten möchte, legt man einfach eine Dummy-Datei an, so daß das Verzeichnis nicht leer ist.

Anmerkung: Die Optionen -r und -D der CVS-Kommandos checkout und export implizieren -P.

7.4 Dateien verschieben oder umbenennen

In den folgenden Abschnitten gehen wir davon aus, daß wir eine Datei ALT in eine Datei NEU umbenennen wollen.

Der normale Weg zum Umbenennen

```
$ mv ALT NEU
$ cvs remove ALT
$ cvs add NEU
$ cvs commit -m ”ALT nach NEU umbenannt” ALT NEU
```

Dies ist der einfachste Weg, eine Datei umzubenennen. Einzig die Datei-Historie ist jetzt etwas verstreut, da man, um die komplette Historie zu erfahren, sowohl ALT als auch NEU betrachten muß.

Da die Datei NEU jetzt eine neue Datei ist, fängt Ihre Revisionsnummer bei 1.1 an. Mit der Option -r des commit-Kommandos kann man explizit eine andere Revision vergeben.

Direkt im Repository umbenennen Diese Methode ist etwas gefährlicher, da Dateien direkt im Repository umbenannt werden! Also Vorsicht!

```
$ cd $CVSROOT/VERZEICHNIS
$ mv ALT,v NEU,v
```

Vorteile:

- Das Protokoll der Veränderungen („ChangeLog“) bleibt intakt.
- Die Revisionsnummern werden nicht verändert.

Nachteile:

- Ältere Releases können nicht problemlos aus dem Repository erhalten werden. (Die Datei erscheint als NEU auch in Revisionen vor der Umbenennung!)
- Es gibt keinen Protokolleintrag für die Umbenennung.
- Schlimme Dinge können passieren, wenn jemand während der Umbenennung auf die Dateien im Repository zugreift. Man sollte sicherstellen, daß in dieser Zeit niemand irgendwelche CVS-Kommandos verwendet.

Im Repository eine Kopie anfertigen Diese Variante beinhaltet auch direkte Modifikationen im Repository. Sie ist allerdings etwas sicherer, aber nicht ohne Nachteile.

```
Zuerst die RCS-Datei im Repository kopieren
$ cd $CVSROOT/VERZEICHNIS
$ cp ALT,v NEU,v
Dann die alte Datei löschen (im Arbeitsverzeichnis)
$ cd ~/VERZEICHNIS
$ rm ALT
$ cvs remove ALT
$ cvs commit ALT
Datei NEU in das Arbeitsverzeichnis übernehmen
$ cvs update NEU
Hier alle Tags merken, die nicht zu Branches gehören
$ cvs log NEU
Jetzt alle Tags löschen, die nicht zu Branches gehören
$ cvs tag -d <TAG1>
$ cvs tag -d <TAG2>
$ ...
```

Indem man die Tags löscht, kann man ältere Revisionen korrekt auschecken (die Datei NEU wird dann nicht mit ausgecheckt).

Vorteile:

- Das Auschecken älterer Revisionen funktioniert korrekt, solange man die Option -r<TAG> und nicht -D<DATUM> verwendet.
- Das Veränderungsprotokoll bleibt intakt.
- Die Revisionsnummern werden nicht verändert.

Nachteil:

- Die Umbenennung der Datei ist komplizierter und somit fehleranfälliger.

Verzeichnisse Verschieben oder Umbenennen Der normale Weg um ein Verzeichnis zu Verschieben oder Umzubenennen ist, jede Datei im Verzeichnis ganz normal zu verschieben (siehe 7.4, Seite 14). Danach checkt man mit der Option -P aus und das Verzeichnis sollte nicht mehr auftauchen, da es jetzt leer ist (siehe 7.3, Seite 14).

8 Untersuchen der Dateigeschichte (History)

Bei der Benutzung von CVS entsteht eine „Geschichte“ des Projektes – welche Dateien wurden wann geändert, wie und von wem. Es gibt einige Möglichkeiten, um sich die Geschichte anzusehen.

8.1 Protokollmitteilungen (Log Messages)

Wann immer man eine Datei in das Repository mittels commit eincheckt, wird man nach einer Mitteilung für das Protokoll gefragt. Diese Mitteilung wird zusammen mit der veränderten Datei zu Dokumentationszwecken aufbewahrt. Sie sollte kurz dokumentieren, was und warum man etwas geändert hat.

Mit dem CVS-Kommando log kann man sich die Log Messages einer oder mehrerer Dateien anzeigen lassen. Die Liste aller Log Messages einer Datei wird auch als „ChangeLog“ bezeichnet.

Das Kommando hat verschiedene Optionen:

-l

Nur das aktuelle Verzeichnis bearbeiten, keine Unterverzeichnisse.

-N

Keine Tags anzeigen.

-b

Nur Revisionen im Standard-Zweig auflisten.

-w<logins>

Nur Revisionen anzeigen, die von den angegebenen Nutzern eingecheckt wurden.

-r<revisionen>

Nur angegebene Revisionen anzeigen.

8.2 Die Geschichtsdatenbank

Wenn man ein Repository mit dem init-Kommando erzeugt hat, wird automatisch das sogenannte „History Logging“ aktiviert, indem im Repository im Verzeichnis CVSROOT eine Datei history angelegt wird.

Wenn diese Datei existiert, zeichnet CVS darin jedes checkout, commit, rtag, update und release Kommando auf. Mit dem CVS-Kommando history läßt sich dann auf vielfältige Art und Weise auf diese Informationen zugreifen.

9 Branches – verschiedene Zweige

CVS erlaubt es, Veränderungen in verschiedenen Entwicklungszweigen (oder -richtungen) zu isolieren. Wenn man einen Zweig modifiziert, werden die Änderungen nicht in anderen Zweigen sichtbar.

Später werden die Veränderungen dann in andere Zweige übernommen, indem man mit cvs update -j die Änderungen in das Arbeitsverzeichnis einspielt, sie danach in das Repository eincheckt und somit effektiv die Änderungen in einen anderen Branch übernimmt.

9.1 Motivation für Branches

Nehmen wir an, daß von unserem Compiler tc eine Release 1.0 gemacht wurde. Wir setzen die Entwicklung von tc fort und wollen in einigen Monaten eine Version 1.1 herausbringen.

Nach einer Weile beschweren sich die Kunden über einen fatalen Fehler. Wir checken die Version 1.0 wieder aus (siehe 5.3, Seite 8) und finden den Bug, dessen Korrektur sich als trivial erweist. Leider unterliegen die aktuellen Revisionen des Sourcecodes ständigen Veränderungen und werden voraussichtlich nicht vor einem Monat stabil werden. Deshalb kann auf der Basis der neuesten Sourcecodes kein Bugfix herausgegeben werden.

In dieser Situation erzeugt man einen Branch (Zweig) in den Versionsbäumen aller Dateien, die die Version 1.0 vom tc ausmachen; nennen wir ihn „release-1-0-patches“. Dann können wir Modifikationen an diesem Zweig vornehmen, ohne den Hauptzweig der Entwicklung zu stören. Wenn die Veränderungen abgeschlossen sind, kann man auswählen, ob man sie in den Hauptzweig aufnimmt oder sie im Zweig läßt. Dies ist in Abbildung 2 illustriert.

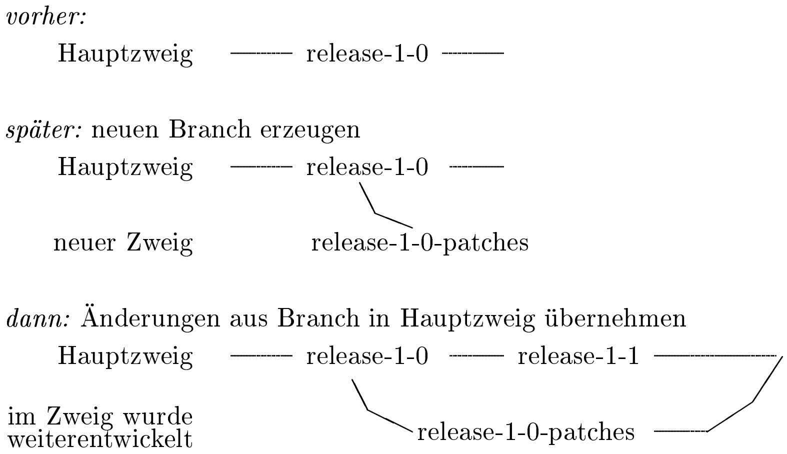

Abbildung 2: Anwendung von Branches

9.2 Einen Branch erzeugen

Einen Branch erzeugt man mit dem CVS-Kommando `tag -b`, zum Beispiel (im Arbeitsverzeichnis):

```
$ cvs tag -b release-1-0
```

Dies spaltet basierend auf den aktuellen Revisionen in der Arbeitskopie einen Zweig (Branch) ab und gibt diesem den Namen „release-1-0“.

Man kann auch einen Branch erzeugen, der (unabhängig von der Arbeitskopie) an einer anderen Revision beginnt:

```
$ cvs rtag -b -r release-1-0 release-1-0-patches tc
```

erzeugt einen Branch im Modul `tc`, der bei den Revisionen beginnt, die den „release-1-0“-Tag haben.

9.3 Zugriff auf Branches

Einen Branch kann man auf zwei Arten erhalten: Indem man ihn frisch aus dem Repository auscheckt oder indem man eine Arbeitskopie in den Branch wechselt.

Um einen Branch aus dem Repository auszuchecken, verwendet man die Option `-r` für `checkout`:

```
$ cvs checkout -r release-1-0-patches tc
```

Oder, wenn man bereits eine Arbeitskopie hat:

```
$ cd tc
$ cvs update -r release-1-0-patches
```

Dadurch wird die Arbeitskopie in den Branch gewechselt - es spielt keine Rolle, in welchem Branch sie vorher war, nach der Ausführung des Kommandos hat man eine Kopie des angegebenen Branch im Arbeitsverzeichnis.

9.4 Branches und Revisionen

Normalerweise sind die Revisionen einer Datei eine Folge linear ansteigender Nummern. CVS ist allerdings nicht auf lineare Entwicklung beschränkt. Der „Revisionsbaum“ kann in Zweige (Branches) geteilt werden, wobei jeder Branch eine eigenständige Entwicklungslinie ist. Die Änderungen in einem Branch können recht einfach wieder in den Hauptzweig übernommen werden. (siehe Abbildung 3)

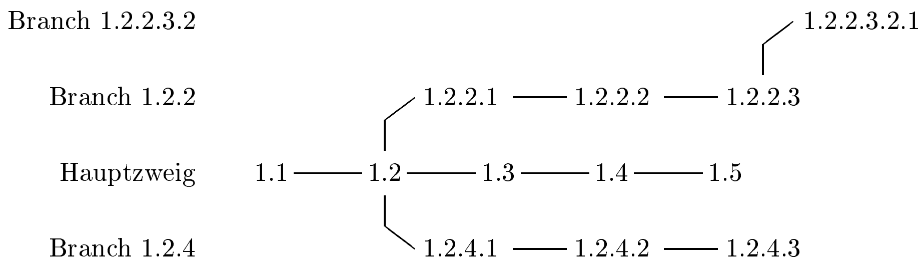

Abbildung 3: Branches

Anmerkung: Die Revisionsnummern der Branches werden für jede Datei einzeln verwaltet.

9.5 Branches verschmelzen

Man kann die Änderungen, die in einem anderen Branch gemacht wurden, in seine Arbeitskopie einspielen, indem man die Option -j des CVS-Kommandos `update` verwendet.

Wir nehmen an, daß wir den Revisionsbaum aus Abbildung 4 vorliegen haben.

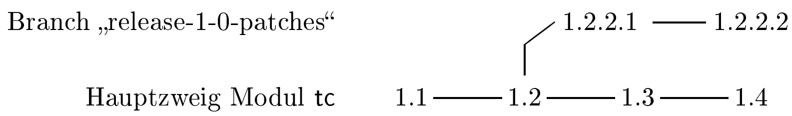

Abbildung 4: Vereinigung von Branches vorher

Mit den folgenden Kommandos können wir jetzt die Änderungen aus dem Branch „release-1-0-patches“ in den Hauptzweig übernehmen.

```
$ cvs checkout tc
$ cvs update -j release-1-0-patches
$ cvs commit -m ”Patches von Release 1.0 integriert”
```

Damit erzeugen wir jetzt eine Version 1.5 der betreffenden Datei im Hauptzweig.

Das CVS-Kommando `checkout` bietet auch eine Option `-j`, die analog funktioniert:

```
$ cvs checkout -j release-1-0-patches tc
$ cvs commit -m ”Patches von Release 1.0 integriert”
```

bewirkt das Selbe wie oben.

9.6 Branches mehrmals verschmelzen

Der Revisionsbaum sieht jetzt wie in Abbildung 5 aus (um bei unserem Beispiel zu bleiben):

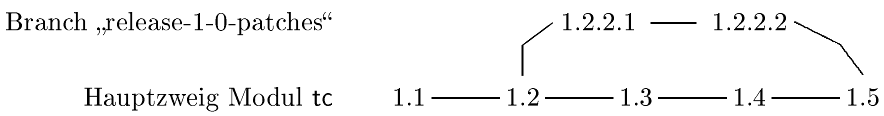

Abbildung 5: Vereinigung von Branches nachher

Wenn nun die Entwicklung im Branch „release-1-0-patches“ fortgesetzt wird, etwa weil weitere Probleme entdeckt werden, möchte man die folgenden Änderungen natürlich auch in den Hauptzweig übernehmen.

Zu diesem Zweck kann man dem `update`-Kommando zweimal die Option `-j` geben – die erste mit einer Angabe, bis wohin bereits zusammengeführt wurde und die zweite mit dem Branchnamen:

```
$ cvs update -j 1.2.2.2 -j release-1-0-patches
```

Um die Versionsnummer 1.2.2.2 nicht von Hand angeben zu müssen, sollte man direkt vor oder nach jedem Zusammenführen den Branch mit einem Tag versehen.

10 Anwendung von CVS

In diesem Abschnitt sollen einige Probleme bei der realen Nutzung von CVS behandelt werden.

10.1 Binäre Dateien

Normalerweise werden im CVS nur Textdateien aufbewahrt. CVS kann verschiedene Revisionen verschmelzen und Unterschiede zwischen Revisionen lesbar anzeigen. Wenn man diese Möglichkeiten aufgibt, kann man auch binäre Dateien im CVS aufbewahren. Zum Beispiel könnte man eine komplette Webseite unter CVS-Kontrolle stellen und möchte dabei natürlich auch die Bilder einschließen.

Eine Grundfunktion der Versionskontrolle ist es, die Unterschiede zwischen zwei Revisionen anzuzeigen. Für Textdateien kann man das CVS-Kommando `diff` verwenden, um anzuzeigen, was geändert wurde. Bei binären Dateien muß man beide Revisionen extrahieren und mit einem externen Tool vergleichen (falls es ein solches gibt). Es ist daher sinnvoll und notwendig, daß die Entwickler aussagekräftige und korrekte Vermerke über die Veränderungen im Protokoll hinterlassen.

Eine weitere Fähigkeit der Versionskontrolle ist, zwei verschiedene Revisionen zu vereinigen. Dies ist zum Beispiel notwendig, wenn mehrere Entwickler in Ihren Arbeitskopien Änderungen an der gleichen Datei vornehmen. Nachdem der erste seine Änderungen eingecheckt hat und der zweite dies mit seinen Änderungen tun möchte, wird er von CVS informiert, daß die Datei mittlerweile geändert wurde. Mittels `cvs update` bringt er sein Arbeitsverzeichnis auf den aktuellen Stand und dabei müssen die Änderungen aus dem Repository in seine Arbeitskopie eingespielt werden (die wiederum Änderungen enthalten kann!).

Im Fall von Textdateien kann CVS die Veränderungen vereinigen sofern sie unabhängig voneinander sind (verschiedene Bereiche der Datei betreffen); kollidieren sie, kann CVS den Nutzer davon in Kenntnis setzen und die entsprechenden Stellen in der Datei markieren. Handelt es sich jedoch um Binärdateien, kann CVS nur beide Revisionen extrahieren und muß dem Nutzer überlassen, die Unterschiede zu verschmelzen oder eine der Revisionen zur weiteren Verwendung zu wählen. Dieses Vorgehen ist natürlich fehleranfällig – der Nutzer könnte zum Beispiel Änderungen übersehen oder für unnötig halten.

Binärdateien speichern Zwei Dinge sind beim Speichern von Binärdateien im CVS zu beachten:

1. CVS konvertiert standardmäßig die Zeilentrenner (LF im Repository, sonst abhängig vom Betriebssystem)
2. Eine Binärdatei könnte zufällig Wörter enthalten, die von der Keyword Expansion behandelt werden. Dadurch würde die Binärdatei in den meisten Fällen zerstört, deshalb muß Keyword Expansion ausgeschaltet werden.

Die Option `-kb` einiger CVS Kommandos (z. B. `add`) verhindert sowohl die Konvertierung der Zeilentrenner als auch die Keyword Expansion (siehe dazu Abschnitt 13, Seite 29 und 13.3, Seite 31).

10.2 Mehrere Entwickler

Wenn mehrere Personen am gleichen Projekt arbeiten, werden die Dinge oft etwas komplizierter. Oftmals versuchen zwei Leute, die gleiche Datei zu editieren. Eine als „File Locking" oder „Reserved Checkouts" bekannte Lösung ist, immer nur jeweils eine Person eine Datei editieren zu lassen. Dieser Ansatz ist allerdings sehr restriktiv und bringt einige Probleme mit sich.

Das Standardmodell von CVS wird „Unreserved Checkouts" genannt. Das heißt, daß die Entwickler jeweils ihre eigene Arbeitskopie einer Datei simultan editieren können. Die erste Person, die Ihre Änderungen eincheckt, hat keine Möglichkeit, herauszufinden, daß jemand anderes ebenfalls dem Editieren der Datei begonnen hat. Die anderen erhalten dann Fehlermeldungen, wenn Sie versuchen, Ihre Änderungen einzuchecken. Sie müssen dann CVS-Kommandos verwenden, um Ihre eigene Kopie auf den neuesten Stand zu bringen (siehe Abbildung 6, Seite 22). Dies erfolgt fast automatisch.

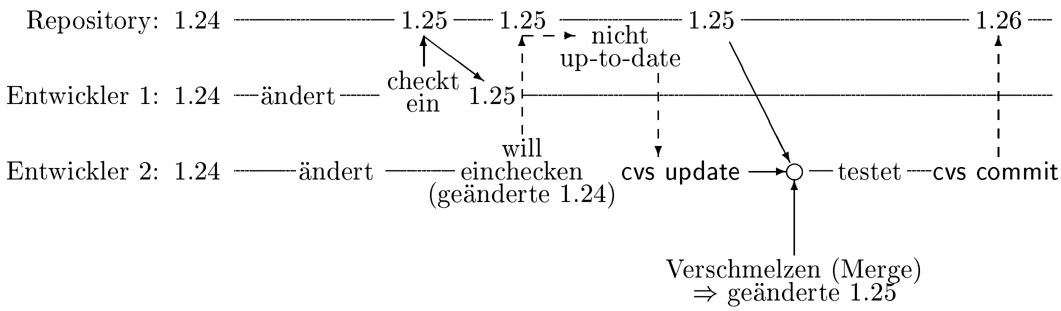

Abbildung 6: Vereinigung zweier Revisionen einer Datei

Im weiteren werden einige Aspekte der Arbeit mehrerer Entwickler an einem CVS-verwalteten Projekt besprochen.

Dateistatii

Je nachdem, was man mit seiner Arbeitskopie gemacht hat und was andere in der Zwischenzeit mit dem Repository gemacht haben, kann eine Datei verschiedene Statii haben. Den Status einer Datei fragt man mit `cvs status` ab.

Up-to-date Die Arbeitskopie ist identisch mit der neuesten Revision im Repository.

Locally Modified Die Arbeitskopie wurde verändert und die Änderungen noch nicht in das Repository eingespielt.

Locally Added Die Datei wurde lokal hinzugefügt, aber noch nicht eingecheckt.

Locally Removed Analog.

Needs Checkout Die Datei wurde im Repository hinzugefügt oder verändert. Dieser Status müßte eigentlich „Needs Update“ heißen, da man die aktuelle Revision mittels `cvs update` erhält.

Needs Patch Ähnlich wie „Needs Checkout“, aber die lokale Datei kann mittels eines Patch auf den Stand der neuesten Revision gebracht werden.

Needs Merge Jemand hat eine neuere Version in das Repository eingespielt, an der Arbeitskopie wurden jedoch auch Veränderungen vorgenommen. Diese Veränderungen müssen vor dem einchecken mittels `cvs update` vereinigt werden (falls nötig mit Handarbeit).

File had conflicts on merge Beim Verschmelzen von Veränderungen ist ein Konflikt aufgetreten. Wenn nicht bereits geschehen, muß die Datei von Hand bearbeitet werden, um den Konflikt zu beheben.

Unknown CVS ist nichts über diese Datei bekannt. Die Datei steht nicht unter Versionskontrolle.

Eine Datei auf den neuesten Stand bringen

Wenn man eine Datei auf den neuesten Stand bringen möchte, verwendet man das `update`-Kommando.

Dabei gehen die Veränderungen an einer Datei niemals verloren! Wenn die Arbeitskopie der Datei verändert wurde, werden nur die Änderungen integriert.

Wenn man also die Revision 1.4 einer Datei auscheckt und editiert, kann mittlerweile jemand eine Revision 1.5 und 1.6 der Datei erzeugen (indem er eigene Änderungen in das Repository einspielt) und man kann mit `cvs update` seine eigene Kopie auf den aktuellen Stand der Revision 1.6 bringen, ohne die eigenen Änderungen zu verlieren.

Umgekehrt kann man eigene Änderungen, die noch nicht eingecheckt sind, verwerfen, indem man die betreffende Datei einfach im Arbeitsverzeichnis löscht und mit `cvs update` wieder aus dem Repository einspielt.

11 Das Repository im Detail

Wie in Abschnitt 4, Seite 6 bereits erläutert, speichert CVS sämtliche Informationen über ein Projekt an zentraler Stelle im sogenannten Repository.

11.1 Die Struktur

Die Struktur des Repository entspricht der Verzeichnisstruktur im Arbeitsverzeichnis (sofern keine Module verwendet werden). Angenommen, das Repository befindet sich im

```
/
└── cvsroot/
    ├── CVSROOT/
    ├── TeeGee/
    │   └── WDIR/
    ├── tc/
    ┆
```

Abbildung 7: Verzeichnisstruktur im Repository

Verzeichnis /cvsroot. Ausgehend von den hier verwendeten Beispielen könnte die Verzeichnisstruktur dann aussehen wie in Abbildung 7.

In den einzelnen Verzeichnissen sind die sogenannten „History Files“ (auf deutsch etwa „Geschichtsdateien“) für jede Datei, die sich unter Versionskontrolle befindet. Der Dateiname des „History File“ setzt sich zusammen aus der korrespondierenden Datei und einem angehängten „,v“. In Abbildung 8 sind die Dateien im Repository für unseren tc-Compiler zu sehen.

```
/
└── cvsroot/
    ├── CVSROOT/
    ├── tc/
    │   ├── Makefile,v
    ┆   ├── backend.c,v
        ├── driver.c,v
        ├── frontend.c,v
        └── parser.c,v
```

Abbildung 8: Dateien im Repository

In jedem „History File“ sind unter anderem genügend Informationen gespeichert, um jede Revision der entsprechenden Datei zu erzeugen. Auch die Protokollmitteilungen und der Name des Nutzers, der die Änderungen eingespielt hat, sind in der Datei vermerkt. Die „History Files“ sind auch als „RCS Files“ (RCS Dateien) bekannt, da dieses Dateiformat zuerst vom Versionskontrollsystem RCS verwendet wurde.

11.2 Dateizugriffsrechte

Alle Dateien, die auf „,v“ enden, werden nur-lesbar angelegt und diese Zugriffsrechte sollten nicht geändert werden. Die Verzeichnisse im Repository sollten schreibbar sein für alle Personen, die die Dateien im Verzeichnis verändern dürfen. Das bedeutet normalerweise, daß man eine Unix-Gruppe erzeugen muß, die alle Nutzer enthält, die Schreibzugriff auf das Repository erhalten sollen und daß das Verzeichnis im Repository dieser Gruppe gehören muß.

Das bedeutet, daß Zugriffskontrolle nur auf Verzeichnisebene möglich ist!

Es ist außerdem zu beachten, daß die Nutzer Schreibrechte benötigen, um Dateien auszuchecken, da CVS Lock Files anlegen muß, um sicheren Betrieb mit mehreren Benutzern zu gewährleisten.

Jede RCS-Datei gehört dem Nutzer, der sie zuletzt eingecheckt hat. Dies hat kaum Bedeutung – in Wirklichkeit zählt nur der Eigentümer des Verzeichnisses.

CVS versucht, die Zugriffsrechte angemessen zu setzen, wenn neue Verzeichnisse im Baum hinzugefügt werden. Man muß also von Hand nacharbeiten, wenn das neue Verzeichnis andere Zugriffsrechte als das Elternverzeichnis haben soll.

11.3 Die „Bodenkammer“ (Attic)

Manchmal bemerkt man, daß CVS eine RCS-Datei in einem Unterverzeichnis Attic speichert, also wie z. B. in /cvsroot/tc/Attic/backend.c,v. Aus Nutzersicht sollte es keinen Unterschied machen, ob eine Datei in der „Bodenkammer“ liegt oder nicht. CVS merkt sich dies und schaut dort nach, wenn es notwendig ist.

Der Vollständigkeit halber sei erwähnt, daß eine Datei im Verzeichnis Attic gespeichert wird, wenn sie im Hauptentwicklungszweig nicht oder nicht mehr existiert. Sie könnte zum Beispiel in einem anderen Branch hinzugefügt worden sein oder sie wurde mittlerweile gelöscht. In jedem Fall ist es notwendig, die Datei aufzubewahren, um ältere Revisionen wiederherstellen zu können.

11.4 Ein Verzeichnis CVS im Repository

Für einige interne Zwecke legt CVS auch im Repository ein CVS-Verzeichnis an, wenn dies notwendig ist.

11.5 Dateien im CVSROOT-Verzeichnis

Das Verzeichnis $CVSROOT/CVSROOT enthält einige Dateien zu administrativen Zwecken. In gewisser Weise ist dieses Verzeichnis wie jedes andere im Repository: Es enthält RCS-Dateien, die auf „,v“ enden und viele der CVS-Kommandos funktionieren in bekannter Weise – mit ein paar Unterschieden.

Zu jeder RCS-Datei gibt es eine ausgecheckte Kopie; zur Datei loginfo,v gibt es z. B. eine Datei loginfo. Wenn man jetzt eine der Administrationsdateien eincheckt, sollte CVS eine Meldung ausgeben:

```
$ cvs commit loginfo
cvs commit: Rebuilding administrative file database
```

Und es sollte die ausgecheckten Kopien aktualisieren. Man kann auch eigene Dateien hinzufügen und muß diese dann in der Datei checkoutlist eintragen, damit sie mit beachtet werden.

Genaue Informationen zu den einzelnen Dateien finden sich im nächsten Abschnitt.

12 Dateien zur Administration

Im Unterverzeichnis CVSROOT des Repository befinden sich einige Dateien, die man nutzen kann, um sich das Leben zu erleichtern. Wie man diese Dateien editiert, steht in Abschnitt 11.5.

12.1 modules

In dieser Datei werden die Module definiert, die zum Auschecken verwendet werden können. Diese Datei wurde bereits in 6.3, Seite 10 beschrieben.

12.2 cvswrappers

Diese Datei kontrolliert ein CVS-Feature, das es erlaubt, einige Einstellungen in Abhängigkeit vom Dateinamen der betreffenden Datei zu ändern.

Die Datei ist zeilenorientiert, jede Zeile hat folgendes Format:

```
wildcard [option 'value'] [option 'value']...
```

Die verfügbaren Optionen sind:

-k

Ändern des Modus für Keyword Expansion

-m

Ändern der Update-Methode

Für die verschiedenen Modi der Keyword Expansions siehe Abschnitt 13.3, Seite 31. Die Update-Methode spielt in dem Augenblick eine Rolle, wo Änderungen im Repository in eine lokal veränderte Datei eingespielt werden müssen (meist mittels cvs update angefordert).

Die Standardmethode (Schlüsselwort „MERGE“) für Textdateien versucht, die Änderungen mittels eines Patch in die lokale Datei zu übernehmen, d.h. die lokalen mit den

Änderungen aus dem Repository zu vereinigen. Ist dies für bestimmte Dateitypen nicht erwünscht, so kann man alternativ die Update-Methode „COPY“ angeben, die (seit CVS-Version 1.10) die Änderungen nicht einspielt, sondern eine Kopie der Datei anlegt, so daß der Nutzer die Änderungen selbst zusammenführen muß.

Die Option -m beeinflußt nur das Verhalten beim Updaten einer Datei, nicht die Art der Speicherung. Für Binärdateien muß die Option -m nicht extra angegeben werden, sie wird durch -kb impliziert.

Die Angaben in dieser Datei können auch einigen CVS-Kommandos auf der Kommandozeile übergeben werden – dazu dient die Option -W, die z. B. wie folgt angewendet werden kann:

```
$ cvs import -W “*.exe -k ’b’“ -W “*.c -k ’o’“ ...
```

Dies entspricht zwei Zeilen in der Datei cvswrappers:

```
*.exe -k ’b’
*.c -k ’o’
```

12.3 Dateien für commit

Es gibt einige Dateien, die es ermöglichen, einen commit abhängig vom betroffenen Verzeichnis im Repository bestimmte Aktionen ausführen zu lassen.

Die Dateien sind wiederum zeilenorientiert und haben einen gemeinsamen Syntax:

```
<regexp> Kommando
```

Es wird jeweils das erste Kommando ausgeführt, dessen Regular Expression auf das derzeitige Verzeichnis im Repository passt.

Die einzelnen Dateien spezifizieren Programme für folgende Zwecke:

commitinfo

Das Progamm überprüft, ob das Einchecken zulässig ist. Kehrt es mit einem Rückgabewert ungleich Null zurück, wird der commit abgebrochen.

verifymsg

Das angegebene Programm ist für die Überprüfung der eingegebenen Protokollmittleiung zuständig. Dies ist besonders nützlich im Zusammenspiel mit der Datei rcsinfo (siehe 12.4).

editinfo

(Veraltet für verifymsg.)

loginfo

Das Programm wird aufgerufen, wenn das Einchecken abgeschlossen ist. Es erhält die Log Message und einige zusätzliche Informationen und kann diese in einer Datei speichern, jemandem zumailen, in einer lokalen Newsgroup posten, usw..

12.4 rcsinfo

Die Datei rcsinfo kann verwendet werden, um ein Formular zu spezifizieren, daß beim Angeben der Log Message editiert wird. Der Syntax ähnelt dem der Dateien commitinfo & Co. (siehe 12.3) allerdings ist der zweite Teil *kein* auszuführendes Kommando, sondern sollte ein kompletter Pfad zu einer Datei sein, die eine Schablone für die Log Message enthält.

Wenn der Verzeichnisname im Repository auf keine der Regular Expressions passt, wird die Zeile „DEFAULT" verwendet, falls sie existiert.

Gibt man allerdings mit der Option -m explizit eine Mitteilung für das Protokoll an, wird die Schablone nicht verwendet.

Findet CVS in der rcsinfo-Datei keinen passenden Eintrag, wird ein leeres Formular verwendet.

12.5 cvsignore

In dieser Datei kann man Wildcards für Dateien angeben, die CVS komplett ignorieren soll. Zum Beispiel entstehen beim Compilieren von C-Programmen „*.o"-Dateien, die nicht mit unter Versionskontrolle gestellt werden sollten, da sie

1. sich aus den C-Quelltexten erzeugen lassen
2. binäre Dateien sind
3. sich je nach verwendetem Compiler unterscheiden können, obwohl sie aus dem selben Sourcecode entstanden sind
4. ⇒ nur Platz verschwenden.

Weiterhin gibt es u. U. temporäre Dateien, die von den verwendeten Tools erzeugt werden. Fertig kompilierte Programme sollten auch maximal zu Dokumentationszwecken im CVS gelagert werden.

Ohne spezielle Vorkehrungen gibt CVS bei einem update für jede unbekannte Datei eine Zeile aus, um darauf hinzuweisen, daß diese Datei nicht unter Versionskontrolle steht. Dies läßt sich vermeiden, indem man eine Liste von Dateien angibt, die CVS bei update, import und release ignorieren soll.

CVS bringt bereits eine eingebaute Liste mit, die u. a. „*~", „*.o" und „*.exe" enthält. Dazu kommen noch die Einträge aus $CVSROOT/CVSROOT/cvsignore, .cvsignore im Arbeitsverzeichnis (diese Datei kann und sollte unter Versionskontrolle gestellt werden!), der Inhalt der Umgebungsvariable CVSIGNORE, jede Option -I zum CVS-Kommando, und der Inhalt der Dateien .cvsignore während ein Verzeichnisbaum durchmustert wird.

12.6 checkoutlist

Manchmal ist es nützlich, CVS zu verwenden, um eigene Dateien im Verzeichnis CVS-ROOT aufzubewahren. Zum Beispiel könnte man die Programme, die von commitinfo & Co. aufgerufen werden dort ablegen. Indem man diese Dateien dann in die Datei checkoutlist einträgt (eine Datei pro Zeile, keine Pfadangaben), werden sie genau wie die CVS-eigenen administrativen Dateien behandelt (siehe dazu am Anfang dieses Abschnitts, Seite 26).

12.7 history

In dieser Datei werden einige CVS-Aktionen aufgezeichnet, falls sie existiert. Siehe Abschnitt 8.2, Seite 17. Im Unterschied zu den anderen administrativen Dateien gibt es zu dieser Datei keine RCS-Datei, da an die Datei immer nur angehängt wird.

12.8 Variablen in administrativen Dateien

Manchmal benötigt man innerhalb der administrativen Dateien einige Informationen. Das Home-Verzeichnis des ausführenden Nutzers läßt sich Unix-üblich mit „`~`" bzw. „`~user`" ermitteln. Weiterhin kann man mittels „`${VARIABLE}`" Werte von Variablen ermitteln:

CVSROOT
: Spezifiziert das aktuell verwendete Repository.

CVSEDITOR, VISUAL, EDITOR
: Enthalten alle den selben Wert, nämlich den Editor, den CVS verwendet.

USER
: Der Nutzer, der CVS gerade verwendet (auf dem Server).

13 Keyword Expansion

So lange man seine Sourcecodes in einem Arbeitsverzeichnis editiert, das aus einem Repository stammt, kann man mit `cvs log` und `cvs status` jederzeit den Status der einzelnen Dateien ermitteln.

Doch sobald man die Dateien in eine andere Umgebung exportiert (z. B. indem man eine Softwarerelease mit Sourcecode macht), wird es schwieriger, herauszufinden, um welche Revisionen es sich handelt.

CVS bietet einen Mechanismus, der „Keyword Expansion" bzw. „Keyword Substitution" (auf Deutsch etwa „Schlüsselwortersetzung") genannt wird und es ermöglicht, die Dateien zu identifizieren.

Der Mechanismus basiert darauf, daß Zeichenketten der Form `$KEYWORD$` und `$KEYWORD:...$` immer beim Erhalten einer neuen Revision durch Zeichenketten der Form `$KEYWORD:WERT$` ersetzt werden.

13.1 verfügbare Schlüsselwörter

CVS kennt folgende Schlüsselwörter (Keywords):

`$Author$`

Der Loginname des Nutzers, der diese Revision eingecheckt hat.

`$Date$`

Das Datum und die Zeit in UTC als die Revision eingecheckt wurde.

`$Header$`

Ein Standard-Kopf, der den vollen Pfad der RCS-Datei, die Revisionsnummer, das Datum (UTC), den Autor und den Status der Datei enthält.

`$Id$`

Ähnelt **`$Header$`**, nur daß die RCS-Datei ohne Pfad angegeben wird.

`$Name$`

Der Tag-Name, der zum Auschecken der Datei verwendet wurde. Wird nur ersetzt, wenn beim Auschecken explizit ein Tag angegeben wurde.

`$Log$`

Die Protokollmitteilungen, die beim Einchecken angegeben wurden. Dabei wird nicht das gesamte Protokoll neu ersetzt, sondern nur die neu hinzugekommenen Meldungen eingefügt.

Die Verwendung dieses Schlüsselwortes ist umstritten. So lange man innerhalb des Entwicklungssystems arbeitet, sind die Informationen leicht verfügbar, auch ohne daß man dieses Schlüsselwort verwendet. Und sobald man die Datei exportiert könnten die Informationen sowieso nutzlos sein.

Ernster sieht die Situation aus, wenn Branches zusammengeführt werden sollen – es werden oft Konflikte auftreten, da sich die Protokolle unterscheiden.

Weiterhin tendieren die Leute dazu, die Protokolleinträge zu korrigieren (Rechtschreibefehler oder logische Fehler). Deswegen können die Informationen in der Datei von den Informationen, die `cvs log` ausgibt, abweichen.

Es wird empfohlen, das **`$Log$`**-Schlüsselwort *am Ende* der Datei einzufügen und nicht im Dateikopf, da so sehr lange Protokolle nicht bei der täglichen Arbeit behindern.

`$RCSfile$`

Der Name der RCS-Datei, ohne Pfad.

`$Revision$`

Die Revisionsnummer.

`$Source$`

Der komplette Pfad der RCS-Datei.

`$State$`

Der Status, der dieser Revision zugeordnet wurde.

Den Status einer Datei kann man mit `cvs admin -s<STATUS>` zuordnen. Dabei kann man einen beliebigen Status angeben. Geläufig sind „Exp“ für experimentell (diesen Status erhält eine neu erzeugte Datei), „Stab“ für stabil und „Rel“ für released. Der Status „dead“ wird von CVS intern verwendet.

13.2 Keywords verwenden

Um ein Schlüsselwort zu verwenden, schreibt man einfach die entsprechende Zeichenkette (z. B. `$Id$`) in die Datei und checkt die Datei ein. Dieses Dokument steht unter Versionskontrolle. Die folgende Zeile wurde erzeugt, indem `$Id$` an die entsprechende Stelle geschrieben wurde – CVS hat dies dann entsprechend expandiert:

```
$Id: cvs_intro.tex,v 1.13 2000/08/28 12:19:39 tisc Exp $
```

Um die Expansion von Schlüsselwörtern zu vermeiden, genügt es, zwischen dem „$“ und dem Schlüsselwort beliebige Zeichen unterzubringen, also kann man z. B. nach dem „$“ je nach dem Typ der bearbeiteten Datei einen Leerbefehl einfügen, so daß CVS das Schlüsselwort nicht mehr erkennt. In diesem LaTeX-Dokument erreicht man das z. B. mit `$\mbox{}Id$` – `\mbox{}` erzeugt ein leeres Zeichen. In einem C-Programm könnte man eine Zeichenkette `$Id$` erzeugen mit `char *zk = "$""Id$"`.

13.3 Ersetzungsmodi

Die Ersetzung von Schlüsselwörtern läßt sich noch beeinflussen. Jeder RCS-Datei ist ein Standard-Ersetzungsmodus zugeordnet, jeder Arbeitskopie ebenso. Der erstere wird durch den Parameter `-k` des CVS-Kommandos `add` bestimmt, der letztere durch die Option `-k` der CVS-Kommandos `update` und `checkout`. `cvs diff` kennt auch eine Option `-k`.

Es gibt die folgenden Modi:

-kkv

Dies ist der Standardmodus. Er sorgt für die Ersetzung der Schlüsselwörter wie oben beschrieben.

-kk

Die Werte für die Keywords werden nicht eingesetzt, sondern allein das Schlüsselwort im Text belassen. Dieser Modus ist nützlich für das Vergleichen verschiedener Revisionen einer Datei, da dann sofort die Unterschiede durch verschiedene Keyword Expansion wegfallen.

-ko

Die Keywords werden nicht ersetzt, sondern so im Text belassen, wie sie beim Einchecken waren. D. h. wenn man eine Datei eincheckt, die die Zeichenkette

`$Revision: 1.2 $` enthält, so wird diese Zeichenkette nicht verändert, sondern bleibt auch in der Revision 1.7 der Datei erhalten.

-kb

Genau wie `-ko`, aber verhindert zusätzliche die Konvertierung der Zeilentrenner.

-kv

Ersetzt die Schlüsselwörter allein durch die Werte. Eine Zeichenkette `$Revision$` wird also z. B. zu `1.7`, nicht zu `$Revision: 1.7 $`. Dadurch kann aber später keine weitere Keyword Expansion stattfinden – das Schlüsselwort ist nicht mehr vorhanden.

Diese Option ist nützlich für das Exportieren von Sourcecodes aus dem CVS; binäre Dateien werden dann allerdings nicht mehr korrekt behandelt.

Index