CVS – Ein System zur Versionsverwaltung

Eine praktische Einführung aus Anwendersicht

Jörg Lehmann

3. November 1999

1 Einleitung

Bei dem *Concurrent Versions System (CVS)* [1] handelt es sich um ein weitverbreitetes *Versionskontrollsystem*. Als solches unterstützt es den Anwender bei der Verwaltung von verschiedenen Programmversionen, indem es die Änderungen an den Quelltexten[1] eines (Software-)projekts mitprotokolliert. Eine Besonderheit von CVS, die es von dem vor allem früher häufig verwendeten RCS unterscheidet, besteht darin, daß explizit die gleichzeitige Bearbeitung ein und derselben Programmdatei durch verschiedene Personen erlaubt ist. CVS geht dabei von der (optimistischen) Annahme aus, daß nicht gleichzeitig Änderungen an nahe beieinanderliegenden Stellen innerhalb einer Datei durchgeführt werden. Sollte dies tatsächlich nicht der Fall sein, kann CVS die verschiedenen Modifikationen automatisch zusammenfügen (*mergen*). Ist dies einmal (ausnahmsweise) nicht möglich, d.h. treten *Konflikte* zwischen verschiedenen Änderungen auf, so ist ein manueller Eingriff des Benutzers notwendig. In der Praxis hat sich dieses Vorgehen bis hin zu großen Software-Projekten, wie z.B. KDE, als günstig herausgestellt.
Dieser Vortrag wendet sich an denjenigen, der mit einem bereits bestehenden CVS-Repository (s.u.) arbeiten will. Er legt also den Schwerpunkt auf die Anwendersicht und geht nicht auf die zur Einrichtung und Verwaltung eines CVS-Repositories notwendigen Kommandos ein. Wer sich für den letzteren Aspekt interessiert, sei auf die umfangreiche CVS-Dokumentation [2, 3] verwiesen.

2 Einführung in die Benutzung von CVS

2.1 Allgemeines

Das CVS speichert die jeweils aktuellste Version der Projektdateien zusammen mit den Informationen zu allen an diesen bisher erfolgten Änderungen im sogenannten *CVS-Repository*. Zur Bearbeitung der Dateien muß man von diesen zunächst eine lokale Kopie der entsprechenden Dateien des Repositories bei sich anlegen (*auschecken*). Diese lokalen

[1] Unter Quelltexten verstehen wir hier und im folgenden nicht nur die Quelltexte von Programmen, sondern jegliche Art von Dateien, die unter die Versionskontrolle gestellt werden sollen. Vorbeugend sei an dieser Stelle jedoch darauf hingewiesen, daß beim Einchecken von Binärcode in das CVS-Repository besondere Vorsichtsmaßnahmen ergriffen werden müssen (vgl. Abschnitt 2.6)

Kopien sollte man dann von Zeit zu Zeit durch die aktuellsten Versionen aus dem Repository ersetzen (*updaten*). Außerdem kann man natürlich die eigenen Modifikationen in das Repository übertragen (*einchecken* bzw. *commiten*). Sollten dabei Konflikte auftreten, d.h. hat bereits ein anderer Benutzer Modifikationen an der/den entsprechenden Datei(en) vorgenommen, die vom CVS nicht automatisch aufgelöst werden können, wird man aufgefordert diese zu beseitigen und muß die Veränderungen erneut einchecken.

2.2 Befehlssyntax

Der Zugriff auf alle CVS-Befehle erfolgt über das Kommando `cvs`. Dessen Parameter gliedern sich dabei wie folgt

> `cvs` [*CVS-Optionen*] *CVS-Kommando* [*Kommando-Optionen*]
> [*Kommando-Argumente*]

Dabei haben die Parameter folgende Bedeutung:

CVS-Optionen:	Für alle CVS-Kommandos gleichermaßen relevanten Optionen (z.B. `-d /home/cvsroot`)
CVS-Kommando:	Der Name des auszuführenden CVS-Kommandos (z.B. `co`, `ci`, `update`, `login`, `logout`)
Kommando-Optionen:	Optionen, die sich auf das angegebene CVS-Kommando beziehen (z.B. `-m "Kommentar zur neuen Version"`)
Kommando-Argumente:	Eigentliche Argumente für das CVS-Kommando (z.B. Datei- und/oder Verzeichnisnamen)

Hinweis: Da einige der CVS- und Kommando-Optionen sehr häufig verwendet werden, lassen sich für diese in der Datei `~/.cvsrc` Standardwerte festlegen. Das Format ist dabei selbsterklärend, es sei deshalb lediglich ein Beispiel angegeben, wobei die Bedeutung der einzelnen Parameter später klar werden wird:

```
# Minmalversion ~/.cvsrc
cvs -z3
update -P -d
checkout -P
tag -c
```

Die mit `cvs` beginnende Zeile legt dabei die standardmäßig verwendeten globalen CVS-Optionen fest.

2.3 Angabe des CVS-Repositories

Zunächst einmal muß man CVS natürlich mitteilen, auf welches Repository man zugreifen will. Dies geschieht mit Hilfe der Variablen `CVSROOT`.[2] Dabei existieren zwei wichtige Fälle:

[2] Alternativ kann man auch die CVS-Option `-d` verwenden, also z.B. `cvs -d /home/cvsroot ...` Allerdings ist dann unter Umständen eine mehrfache Eingabe des Wurzelpfades notwendig. Hierbei hat die CVS-Option `-d` eine höhere Priorität als der Inhalt der Variablen `CVSROOT`.

- Zum Zugriff auf ein lokales Repository wird einfach dessen Wurzelpfad angeben, also z.B. (`sh`-artige Shells)

  ```
  $ export CVSROOT=/home/cvsroot
  ```

- Liegt das Repository auf einem anderen Rechner und wird, wie meistens üblich, eine Authentifizierung mittels eines Paßwortes durchführt, so verwendet man stattdessen beispielsweise

  ```
  $ export CVSROOT=:pserver:gast@www.luga.de:/home/cvsroot
  ```

  und meldet sich anschließend bei dem entsprechenden Server an:

  ```
  $ cvs login
  CVS password:
  ```

  Nach der Eingabe des Benutzerpaßwortes kann man dann so lange auf das Repository zugreifen, bis man via

  ```
  $ cvs logout
  ```

  die Verbindung wieder trennt.

  Achtung: Zwischen dem Ausführen von `cvs login` und `cvs logout` wird das Passwort standardmäßig in der Datei `~/.cvspass` gespeichert und dabei nur trivial verschlüsselt. Diese „Verschlüsselung“ dient aber einzig und allein dem Zweck, daß bei einem versehentlichem Anzeigen dieser Datei wenigstens nicht unmittelbar das Paßwort erscheint. **Ansonsten stellt dies aber nicht den geringsten Schutz für das Paßwort dar!**

2.4 Das Auschecken der Quelldateien

Da nun die Lage des CVS-Repositories festgelegt ist, kann man sich die aktuelle Version der Quelldateien von dort besorgen. Dazu dient der Befehl `cvs checkout` oder kurz `cvs co`. Wollen wir beispielsweise die Quelldateien des Projekts `test` bearbeiten, so verwenden wir

```
$ cvs checkout test
```

Dadurch werden im aktuellen Verzeichnis ein neues Verzeichnis `test` angelegt und dort die entsprechenden Quelldateien (samt eventuellen Unterverzeichnissen) abgelegt. Hinzu kommen jeweils in jedem Unterverzeichnis (auch in dem Verzeichnis, in dem `test` selbst liegt) die CVS-Verwaltungsunterverzeichnisse mit dem Namen `CVS`. Man erhält also z.B. folgendes Resultat:

```
$ ls test
CVS         Makefile    test.c      test.h
```

Hinweis: Die `CVS`-Verzeichnisse enthalten unter anderem auch Informationen über die Lage des CVS-Repositories. Deshalb ist es nicht möglich zwei Projekte aus verschiedenen Repositories ausgehend von einem gemeinsamen Verzeichnis auszuchecken. Andererseits ist

es aber auch nach dem ersten Auschecken nicht mehr notwendig, die Variable `CVSROOT` zu setzen!
Die sicherlich wichtigste Kommando-Option von `cvs co` stellt `-P` dar. Sie verhindert das Auschecken von leeren Verzeichnissen. Da das CVS-Repository alle jemals existenten Verzeichnisse enthält, auch wenn diese vielleicht inzwischen nicht mehr benötigt werden oder umbenannt wurden, und diese standardmäßig auch auscheckt, wird man diese Option wohl schnell zu schätzen wissen. Sie gehört deshalb eigentlich auch in jede `~/.cvsrc`-Datei.
Eine wesentliche Funktionalität der Versionsverwaltung wird durch die Option `-r` zur Verfügung gestellt, was übrigens automatisch die Option `-P` aktiviert, um ein Verzeichniswirrwar zur verhindern. Diese erlaubt das Auschecken von bestimmten Versionen der entsprechenden Dateien, z.B.

```
$ cvs checkout -r rel-1-0-patches test
```

Damit erhält man die mit dem *Tag* (deutsch: Etikett) `rel-1-0-patches` versehenen Versionen der entsprechenden Dateien aus dem Projekt `test`. Wie man Dateien mit einem Tag versieht wird später noch erläutert. Oftmals muß sich der Anwender darum aber auch gar nicht kümmern, denn er will vielleicht nur die Dateien in einem bestimmten *Entwicklungszweig* bearbeiten (auch dazu später mehr). Dies wird dadurch ermöglicht, daß die einen Entwicklungszweig bezeichnenden Tags *sticky* (deutsch: klebrig) sind, d.h. das CVS merkt sich, daß die Dateien unter der Angabe eines solchen Tags ausgecheckt wurden und ordnet sie beim späteren Einchecken wieder in den entsprechenden Zweig ein. Ist dieses Verhalten nicht erwünscht, da man sich z.B. nur kurz eine ältere Version ansehen will, hilft die zusätzliche Angabe von `-p`, die zu einer Ausgabe der Dateien auf `stdout` führt. Außerdem ist ein Löschen der Sticky Tags durch die Angabe von `-A` möglich.

2.5 Aktualisieren der lokalen Quelldateien

Nach dem ersten Auschecken ist es ab und zu angebracht, die lokalen Quelldateien mit den neuesten Versionen im Repository abzugleichen, da ja evtl. andere Projektmitarbeiter inzwischen Änderungen vorgenommen haben. Dazu dient das CVS-Kommando `cvs update`. Aus der Ausgabe dieses Kommandos läßt sich entnehmen, welche Dateien im Repository verändert wurden (diese werden durch ein `U` oder `P` gekennzeichnet), bzw. welche Dateien man selbst verändert, aber noch nicht eingecheckt hat (Kennzeichnung durch ein `M`). Außerdem wird eine Meldung ausgegeben, wenn beide Fälle bei einer Datei gleichzeitig auftreten, d.h. wenn man gleichzeitig mit anderen Entwicklern Modifikationen an der selben Datei vorgenommen hat. Treten dabei auch noch die oben beschriebenen Konflikte auf, die das CVS nicht automatisch auflösen kann, wird man auch darauf hingewiesen und muß diese dann selbst beseitigen. Die entsprechenden Quelltextabschnitte werden dabei durch Zeilen der Form <<<<<< und >>>>>> eingefaßt.

2.6 Einchecken der Änderungen an den Quelldateien

Hat man nun die Quelldateien geändert und getestet, so will man normalerweise die neuen Versionen den anderen Entwicklern zur Verfügung stellen. Nehmen wir also in Fortsetzung unseres oberen Beispiels an, wir hätten die Datei `test.c` modifiziert. Das Einchecken erfolgt dann mittels des Kommandos `cvs commit` oder kurz `cvs ci` unter Angabe des Pfades der entsprechenden Datei, also z.B. aus dem `test`-Verzeichnis heraus via

```
$ cvs commit test.c
```

Bei der Angabe von Verzeichnisnamen werden alle darin befindlichen Dateien auf Modifikationen untersucht und dementsprechend eingecheckt. Dabei werden rekursiv auch alle in Unterverzeichnissen befindlichen Dateien berücksichtigt. Ohne Angabe eines Datei- oder Verzeichnisnamens werden die Veränderungen in den Dateien des aktuellen Verzeichnisses (samt denen in evtl. Unterverzeichnissen) eingecheckt.
Bevor die Dateien wirklich in das Repository eingecheckt werden wird der in der Umgebungsvariablen `$CVSEDITOR` oder, falls diese nicht existiert, der in `$EDITOR` eingestellte Texteditor gestartet, und man sollte kurz die durchgeführten Änderungen beschreiben. Kurze Kommentare kann man praktischerweise auch mit der Kommando-Option `-m` vor dem Dateinamen angeben, also z.B.

```
$ cvs ci -m "Fehler in Funktion test() behoben" test.c
```

Wie beim Updaten gilt auch hier, daß evtl. auftretende Konflikte mit Versionen von anderen Entwicklern bei Bedarf manuell beseitigt werden müssen, bevor das Einchecken möglich ist.

2.7 Hinzufügen und Löschen von Dateien

Von Zeit zu Zeit ist es natürlich notwendig, neue Quelltextdateien in das Projekt aufzunehmen. Dazu genügt es nicht, einfach in der lokalen Kopie des Repositories die entsprechende Datei zu erzeugen bzw. zu löschen; CVS wüßte ja auch gar nicht, ob es nicht um eine, z.B. durch einen Kompilationslauf erzeugte, Binärdatei handelt, bzw., ob man eine Datei vielleicht nur aus Mangel an vorhandenem Plattenplatz gelöscht hat. Es ist also ein manueller Eingriff des Benutzers erforderlich, der durch die CVS-Kommandos `cvs add` bzw. `cvs remove` ermöglicht wird.
Das Hinzufügen ist mit `cvs add` unter der Angabe des entsprechenden Datei- oder Verzeichnisnamens möglich. Also beispielsweise:

```
$ ls
CVS        Makefile   test.c     test.h     neu.c
$ cvs add neu.c
cvs add: scheduling file `neu.c' for addition
cvs add: use 'cvs commit' to add this file permanently
```

Wie man der letzten Meldung entnehmen kann, wird das Hinzufügen der Datei erst beim nächsten Einchecken wirksam. Ergo:

```
cvs ci -m "Erste Version von neu.c" neu.c
```

Hinweis: Beim Hinzufügen von Binärdateien ist stets die Kommando-Option `-kb` zu verwenden, also z.B.

```
$ cvs add -kb logo.jpg
```

Dadurch wird die standardmäßig durchgeführte Konvertierung von Zeilenendezeichen sowie die sogenannte *Keyword-Substitution*[3] abgeschaltet. Sollte man dies vergessen, kann es durchaus zu unliebsamen Änderungen in der jeweiligen Datei kommen! Ein nachträgliches Hinzufügen des `-kb`-Flags erlaubt das `cvs admin`-Kommando, daß in der CVS-Dokumentation näher beschrieben ist.
Benötigt man eine Quelldatei oder ein Unterverzeichnis nicht mehr, kann man diese mit `cvs remove` löschen. Dabei sollte man allerdings vorher alle Änderungen an der entsprechenden Datei einchecken, so daß diese nicht für eine eventuelle spätere Verwendung verloren sind. Außerdem muß normalerweise vor dem Entfernen aus dem CVS die entsprechende Datei gelöscht werden. Konkret sieht das Ganze also wie folgt aus:

```
$ rm neu.c
$ cvs remove neu.c
cvs remove: scheduling `neu.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs ci -m "neu.c geloescht" neu.c
```

Will man sich das vorherige Löschen der Datei ersparen, so ermöglicht dies die Kommando-Option `-f`. Damit läßt sich obige Befehlskombination auch etwas kürzer schreiben:

```
$ cvs remove -f neu.c
cvs remove: scheduling `neu.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs ci -m "neu.c geloescht" neu.c
```

Grundsätzlich beachte man, daß auch eine gelöschte Datei weiterhin im Repository vorhanden ist; man will ja später eventuell eine alte Version der Datei wiederherstellen. Bei Verzeichnissen gilt dasselbe, wobei hier ein explizites Löschen mittels `cvs remove` nicht möglich und auch nicht notwendig ist. Stattdessen entfernt der nächste `cvs update`- oder `cvs checkout`-Aufruf mit der Kommando-Option `-P` das leere Verzeichnis.
Zum Umbenennen einer Datei verwendet man zuerst den Standardbefehl `mv`, löscht dann mittels `cvs remove` die alte Datei und fügt die neue mit `cvs add` wieder hinzu. Abschließend darf man den `cvs ci`-Aufruf natürlich wieder nicht vergessen.

3 Diverse Kommandos für Fortgeschrittene

3.1 „Historisches“

Ein Hauptzweck eines Versionskontrollsystems besteht ja darin, daß man jederzeit nachvollziehen kann, wer wann welche Veränderungen an den Quelltexten durchgeführt hat. Diese Funktionalität wird durch verschiedene CVS-Kommandos zur Verfügung gestellt, von denen im folgenden einige kurz erwähnt werden sollen.
Mit Hilfe von `cvs log` (gefolgt von einem/mehreren Dateinamen) lassen sich die beim Einchecken angegeben Kurzbeschreibungen der vorgenommenen Änderungen anzeigen. Zusätzlich wird jeweils angegeben, wann und von wem die entsprechende Modifikation vorgenommen wurde.

[3] Die Keyword-Substitution ersetzt Vorkommen von Ausdrücken wie `$Header$` oder `$Id$` durch die entsprechenden Informationen zu der jeweiligen Datei (Name, Versionsnummer, etc.)

```
$ cvs log test.c
```

Wer einen Überblick über alle an dem Projekt vorgenommenen Versionsänderungen erhalten will, kann dafür `cvs history` verwenden. Die Optionen hierfür entnimmt man am besten der CVS-Dokumentation [2]. Als Beispiel sei hier nur das Kommando

```
$ cvs history -a -e
```

angegeben, mit dem sich eine Liste aller Änderungen (`-e`) aller Benutzer (`-a`) anzeigen läßt. Interessant ist manchmal auch das Kommando `cvs annotate` (gefolgt von einem/mehreren Dateinamen), das die aktuellste Version der entsprechenden Datei(en) ausgibt, wobei jede Zeile durch Informationen über die letzte Änderung versehen wird:

```
$ cvs annotate test.c
```

Will man nur die Unterschiede zwischen der eigenen lokalen Version und der aktuellsten Version im Repository anzeigen, d.h. sich über die selber durchgeführten Änderungen informieren, so ermöglicht dies das `cvs diff`-Kommando, wobei wie immer der/die entsprechende(n) Dateiname(n) anzugeben ist/sind.

```
$ cvs diff -u test.c
```

Die Kommando-Option `-u` gibt hierbei an, daß man die Ausgabe im Unified-diff-Stil wünscht.

3.2 Revisionsnamen und Entwicklungszweige

Oftmals ist es sinnvoll, den momentanen Stand aller Projektdateien besonders zu kennzeichnen, z.B. wenn eine bestimmte Phase des Entwicklungsprozesses erreicht wird (neuer Release o.ä.). Dies geschieht mit Hilfe des `cvs tag`-Befehls. Als Kommando-Argument ist dabei zunächst der gewählte Versionsname (z.B. `rel-2-2`) anzugeben. Gibt man zusätzlich Datei- bzw. Verzeichnisnamen an, so werden nur die entsprechenden Dateien mit dem entsprechenden Versionsnamen versehen.
Das Standardvorgehen besteht darin, den jeweils aktuellsten Versionen aller Projektdateien den entsprechenden Namen zu geben. Dazu begibt man sich einfach in das Wurzelverzeichnis des entsprechenden Projekts (im bereits häufiger verwendeten Beispiel das Verzeichnis `test`) und führt beispielsweise

```
$ cvs tag -c rel-2-2
```

aus. Die Kommando-Option `-c` führt dabei dazu, daß CVS zunächst überprüft, ob noch nicht eingecheckte Änderungen vorhanden sind. Ist dies der Fall, wird der Tag-Prozeß gar nicht erst gestartet und man erhält einen entsprechenden Hinweis. Da die Namen nämlich immer den entsprechenden Dateien im CVS-Repository gegeben werden (und eben nicht den lokalen Kopien, weshalb übrigens nach dem Ausführen von `cvs tag` auch kein `cvs ci`-Aufruf notwendig ist), wird so verhindert, daß die evtl. falsche Dateiversion benannt wird.

Oftmals ist es im Laufe eines Entwicklungsprozesses notwendig, an Release-Versionen Änderungen (z.B. Bugfixes) durchzuführen. Dies wird durch das Erzeugen von *Entwicklungszweigen* ermöglicht. Alle Änderungen, die an solch einem Zweig durchgeführt werden, laufen vollkommen unbeeinflußt von den Änderungen an anderen Zweigen, also insbesondere am „Entwicklungsstamm“ (main trunk), d.h. dem Hauptentwicklungszweig, an dem normalerweise gearbeitet wird.
Die sicherste Methode, einen neuen Entwicklungszweig zu erstellen, besteht darin, diesen ausgehend von den gekennzeichneten Release-Versionen abzuspalten. Diese geschieht mit Hilfe des Kommandos `cvs rtag`. Exemplarisch sei dazu folgender Befehl angegeben:

```
$ cvs rtag -b -r rel-2-2 rel-2-2-patches test
```

Die Option `-b` weist dabei den `rtag`-Befehl an, einen neuen Entwicklungszweig (branch) abzuspalten. Durch die Angabe `-r` gefolgt von dem vorher vergebenen Versionsnamen (hier: `rel-2-2`) wird genau festgelegt, von welchen Versionen der neue Zweig (mit dem Namen `rel-2-2-patches`) abgetrennt wird. Zuletzt benötigt das `rtag`-Kommando noch den Projektnamen (d.h. den Namen, den man auch beim Auschecken verwenden würde), da ja in einem Repository durchaus mehrere Entwicklungsprojekte untergebracht sein können. In unserem Fall geben wir also `test` an.
Auf den neuen Entwicklungszweig kann man jetzt, wie in Abschnitt 2.4 beschrieben, ganz einfach unter der Angabe seines Namens zugreifen. Also entweder man erzeugt eine neue Arbeitskopie aus dem Repository mit Hilfe von

```
$ cvs co -r rel-2-2-patches test
```

oder man hat bereits die Dateien eines Zweiges (z.B. des Hauptzweiges) ausgecheckt und kann dann mittels

```
$ cvs update -r rel-2-2-patches test
```

in den neuen Zweig wechseln. In jeden Fall werden alle eingecheckten Änderungen nur in dem Zweig `rel-2-2-patches` durchgeführt, und zwar solange, bis man die auf diesen Zweig verweisenden Sticky Tags löscht (vgl. Abschnitt 2.4).

Literatur & Links

[1] `http://www.cyclic.com/`

[2] Texinfo-Dateien zum CVS (`info cvs`)

[3] `http://www.loria.fr/~molli/cvs-index.html`