RCS Revision Control System

Un problème majeur dans le développement et la maintenance des programmes est le contrôle de version. C'est-à-dire la conservation bien organisée de tous les changements effectués sur les programmes développés. Au final, un programme est constitué de différentes versions.

L'un des programmes standard pour faire du contrôle de version est **GNU's RCS**, qui signifie *Revision Control System*.

RCS est un ensemble de commandes effectuant ce travail. Il automatise le stockage, la récupération, la tenue d'un journal et l'identification des différentes révisions de fichiers de différents types (texte de n'importe quel format, et même binaire si les outils associés, tel que `diff`, peuvent gérer ce type de fichiers).

RCS ne conserve pas une copie entière de chaque nouvelle version. Il stocke des *deltas*, c'est à dire les différences entre les révisions successives. Pour cela, les changements au fichier *filename* sont conservés dans le fichier *filename,v*.

Parmi les principales caractéristiques de RCS, on peut noter qu'il permet d'extraire une version antérieure des fichiers, de conserver des journaux des modifications apportées, de conserver l'identification des personnes ayant fait les modifications. RCS permet également de comparer deux versions et fournit un mécanisme pour fusionner deux branches de développement différentes d'un fichier. RCS permet également le verrouillage (*lock*) d'un fichier, de telle façon qu'une seule personne puisse apporter des changements (les autres personnes peuvent toujours utiliser le fichier, par exemple pour le compiler).

• Principe général :

La fonction principale de RCS est de gérer des ***groupes de révisions***. On peut définir une révision comme un ensemble de textes appelés révisions, qui évoluent les uns après les autres.

Une nouvelle révision est créée en éditant une révision actuelle. La révision initiale est la racine de l'arbre des révisions. En effet, RCS organise les révisions en un arbre ancestral. La révision initiale de l'arbre (la racine) est normalement numérotée 1.1 et les révisions successives sont numérotées 1.2, 1.3, 1.4 ... Le premier champ du numéro de révision est appelé le ***release number***, et le second champ est appelé le ***level number***.

RCS assigne un nouveau numéro de révision en incrémentant le *level number* de la révision précédente. Le *release number* doit être incrémenté explicitement (uniquement dans le cas d'une transition majeure dans le développement, par exemple lorsqu'une nouvelle release du produit a été complétée).

• Commandes de base :

L'interface de RCS est relativement simple et seulement deux commandes sont suffisantes dans la plupart des cas : **`ci`** et **`co`**.

Nous supposons l'existence d'un fichier nommé `foo.c`. La source du fichier est le suivant :

```
#include <stdio.h>
main()
{
printf("Hello world\n");
```



```
}
```

Nous allons créer un nouveau groupe de révisions avec `foo.c` comme révision initiale (1.1). Le groupe sera stocké dans le fichier nommé `foo.c,v`. Par défaut, le fichier `foo.c` sera effacé.

Pour cela, nous utilisons la commande de ***check-in***:

```
ci foo.c
```

Cette commande demande également une description pour le groupe.

```
foo.c,v <-- foo.c enter description, terminated with single '.' or end of
file:
NOTE: This is NOT the log message!
>>
```

Par exemple, nous tapons :

```
>> programme hello world.
>> .
```

Le texte décrit ce que le programme fait, et `ci` rappelle que ce n'est pas une entrée dans le journal. Les commandes `ci` ultérieures demanderont une entrée du journal, laquelle résumera les modifications apportées. Ces messages doivent être brefs, décrivant les changements apportés.

Pour extraire la dernière révision dans un groupe, on utilise la commande de ***check-out*** :

```
co foo.c
```

Pour pouvoir modifier le fichier `foo.c`, nous devons utiliser l'option `-l` :

```
co -l foo.c
```

Cela permet d'extraire le fichier `foo.c` mais en posant un verrou sur le fichier. Cela signifie que vous, et vous seul, avez la permission de faire un *check-in* sur une nouvelle révision du fichier.

Le système peut être configuré avec la caractéristique du 'strict locking'. Tous les fichiers RCS sont initialisés de telle sorte que les opérations de *check-in* nécessitent un verrou sur la révision précédente.

Nous pouvons alors éditer le fichier `foo.c`, et une fois les modifications effectuées, en faire une nouvelle révision par la commande:

```
ci foo.c
```

La commande nous demande alors le message du journal :

```
foo.c,v <-- foo.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Affichage 10 fois du message
>> .
```

La nouvelle révision est la 1.2.

• Identification automatique :

Avec RCS, vous pouvez marquer vos fichiers sources et objets par des chaînes d'identification spéciales. Cette chaîne décrit le nom du fichier, la révision, la date et l'heure du *check-out*, l'auteur, l'état (par exemple, *Exp* pour expérimental) et celui qui verrouille le fichier. La commande `co` remplacera ce mot-clé avec une chaîne de la forme : `$Id filename revisionnumber date time author state locker$`. D'où, cette chaîne sera mise à jour automatiquement à chaque fois.

Cette possibilité est très importante pour la maintenance des programmes. Ainsi, la commande `ident` extrait ces mots-clés à partir de n'importe quel fichier. Cela permet de retrouver quelles révisions de quels modules ont été utilisées pour construire le programme.

Un mot-clé important est `$Log: nom_fichier.*,v.` Son fonctionnement est un peu différent, il permet d'accumuler les messages du journal saisis lors du processus de *check-in*. On utilise ce mot-clé dans un commentaire en début de fichier source. Ainsi le fichier source est auto-documenté.

- **Verrouillage (locking) :**

Le problème posé est le suivant : au moins deux personnes veulent déposer des modifications d'une même révision. Si nous supposons deux programmeurs apportant des modifications à une même révision (par exemple la 2.5). Le programmeur A fait un `ci` sur sa révision avant le programmeur B. Le programmeur B n'a pas vu les modifications de A, donc l'effet est que les changements de A sont couverts par les modifications de B.

RCS prévient ce conflit par le verrouillage. Lorsque quelqu'un veut éditer une révision pour la modifier, la révision doit être extraite et verrouillée, en utilisant l'option `-l` de `co`. La prochaine opération de *check-in* effacera le verrou. Au plus un programmeur à la fois peut verrouiller une révision, et seulement ce programmeur peut la déverrouiller.

De plus, chaque fichier RCS possède également une liste d'accès, qui spécifie quels utilisateurs peuvent effectuer des opérations de mise à jour.

- **L'arbre de révision :**

Nous avons vu que les numéros de révisions démarraient à 1.1 et, qu'ensuite, la commande `ci` incrémentait uniquement le *level number*. Pour incrémenter le *release number* il faut le faire explicitement avec l'option `-r` de `ci`, par exemple :

`ci -r2.1 foo.c` assigne le numéro 2.1 à la nouvelle révision. Un nouveau *check-in* donnera ensuite 2.2.

Un arbre de révision est constitué normalement d'une branche unique appelée tronc. On peut néanmoins créer des branches latérales. Par exemple, supposons l'arbre de révision suivant : **1.1 -> 1.2 -> 1.3 -> 1.4 -> 2.1 -> 2.2 -> 2.3 ...**
Cet arbre a 7 révisions groupées en 2 *releases*. La release 1.4 est en activité sur un site client, tandis que la *release* 2 est en développement.

Imaginons maintenant que le client demande une correction dans la révision 1.4. Nous allons alors créer une branche à la révision 1.4, et insérer les corrections sur cette branche. La première branche démarrant à 1.4 est numérotée 1.4.1 et les révisions sur cette branche sont numérotées 1.4.1.1, 1.4.1.2, 1.4.1.3 ... Cette notation permet de créer d'autres branches à partir de 1.4.
Les étapes nécessaires sont les suivantes :

```
co -r1.4 foo.c
```

Editer foo.c pour apporter les corrections

```
ci -r1.4.1 foo.c
```


Nous obtenons alors l'arbre suivant :

```
1.1 -> 1.2 -> 1.3 -> 1.4 -> 2.1 -> 2.2 -> 2.3 ...
                \
                 \
                  \> 1.4.1.1 -> ...
```

Il peut être nécessaire d'incorporer les différences entre 1.4 et 1.4.1.1 dans une révision de la release 2. Pour cela, il faut utiliser la commande *rcsmerge* qui automatise le processus.

Une autre raison de créer une branche est liée au problème posé par un programmeur ayant verrouillé une révision pour la modifier et qui n'a pas encore effectué le dépôt de ses modifications. Si un autre programmeur désire modifier cette révision, il ne peut pas le faire tant que le verrou n'est pas supprimé. La solution est de créer une branche, et ensuite, une fois que le premier programmeur a effectué son *check-in*, utiliser la commande **`rcsmerge`** pour incorporer les deux révisions.

- **Autres commandes utiles :**

La commande **`rcsdiff`** permet de comparer la version actuelle avec la plus récente version qui est sauvegardée :

```
rcsdiff -u foo.c
```

L'option `-u` de `rcsdiff` permet d'utiliser un diff unifié (*unified diff*). En fait, `rcsdiff` accepte toutes les options que la commande `diff` accepte. Par exemple, pour générer un *context diff*, on utilisera l'option `-c`.
Par exemple, nous extrayons la révision 1.2 de foo.c, nous effectuons une petite modification (correction d'un bug), et `rcsdiff foo.c` nous donne:

```
===================================================================
RCS file: foo.c,v
retrieving revision 1.2
diff -u -r1.2 foo.c
--- foo.c       2005/04/10 20:33:36     1.2
+++ foo.c       2005/04/11 20:36:20
@@ -24,7 +24,7 @@
main()
{
int i = 0;
-    for(i = 0; i <= 10; i++)
+    for(i = 0; i < 10; i++)
printf("Hello world\n");

}
===================================================================
```

Les lignes qui n'ont pas été modifiées sont affichées avec une simple espace en début de ligne. Les lignes effacées dans la version la plus récente ont un `-` en début de ligne, et les lignes qui ont été ajoutées ont un `+` en début de ligne. Comme on peut le remarquer, les lignes modifiées sont considérées comme effacées dans l'ancienne version, et les changements sont considérés comme ajoutés dans la nouvelle.

On peut également comparer deux versions :

```
rcsdiff -u -r1.2 -r1.4 foo.c
```

Une utilisation de `rcsdiff` est la génération de mises à jour appliquées par des *patch*. Par exemple, une fois un programme terminé, effectuez un *check-in* sur tous les fichiers avec

un nouveau numéro de release, par exemple 2.0. Puis, pour la release suivante (par exemple la 3.0), lancez `rcsdiff` sur la révision 2.0 pour tous les fichiers :

```
rcsdiff -c -r2.0 RCS/* > monprog-3.0.patch 2>&1
```

Vous obtiendrez ainsi un fichier *patch* que vous pourrez distribuer aux personnes possédant la version 2.0.

Une autre possibilité intéressante est de consulter le journal des modifications effectuées. Pour cela on utilise la commande **`rlog`**, qui donne l'historique des changements effectués.

Le plus intéressant ce sont les commentaires saisis lors d'un `ci`. Par exemple, on peut noter que pour la révision 1.3, le commentaire est *Correction d'un petit bug*.

A noter que les dates et heures sont en UTC, et non pas dans la zone locale. Cela permet à des développeurs de différentes zones géographiques de pouvoir collaborer.

La commande **`rcs`** est utilisée pour modifier l'état des fichiers RCS. Par exemple, pour bloquer un fichier qui ne l'est pas ou pour casser un verrou.

Bibliographie :

http://fr.wikipedia.org/wiki/Contrôle_de_version.htm
http://www.april.org/groupes/doc/rcs/rcs.html