

# **(Software) Configuration Management**

Software Technology for  
Mobile and Distributed Applications

Henning Niss (hniss@it-c.dk)  
The IT University of Copenhagen

# Configuration management

---

Babich (1986) “Software configuration management”:

*Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team.*

# Configuration management

---

Babich (1986) “Software configuration management”:

*Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team.*

- identify change;
- control change;
- ensure change properly implemented;
- report change to interested parties.

# Software configuration

---

Software configuration: all items of information produced during the software process.

Example software configuration items:

- design documents;
- program code;
- data.

*These configuration items evolve during the software process and we need to handle the changes.*

# Software configuration, cont'd.

---

Each item exists in a number of **revisions**.

A **software configuration** is a set of (item,revision) pairs.

Configuration management, then, is the art of managing all these revisions.

*Preferably with tool support.*

# Baselines

---

Configuration items in two *states*: before formal review / approval, and after.

Once formally reviewed and approved the configuration item becomes a **baseline**.

Correspondingly, two types of changes

- changes leading to the baselined version;
- changes after the item has been baselined.

# Revision / version control

Need tools to help with making sure changes

- are recorded;
- can be reverted;
- implemented properly;
- ...

Typically, this means that simple, flat files are *not* sufficient.

Revision (version) control systems provide just that.

# Simple revision control

---

Idea: have a central repository keeping track of *all* revisions of configuration items.

Workflow:

- check item out
- edit item
- check item in

# Simple revision control

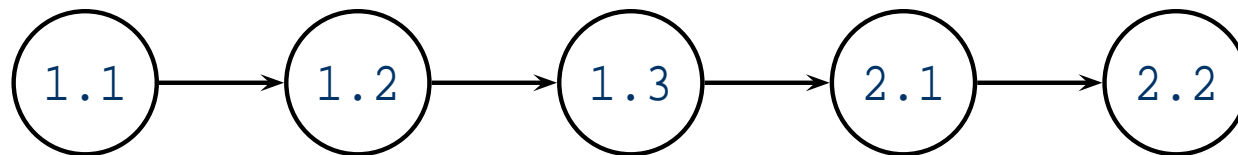
Idea: have a central repository keeping track of *all* revisions of configuration items.

Workflow:	RCS
■ check item out	<code>co file.sml</code>
■ edit item	<code>...</code>
■ check item in	<code>ci file.sml</code>

**RCS** (Revision Control System) by Walter Tichy.

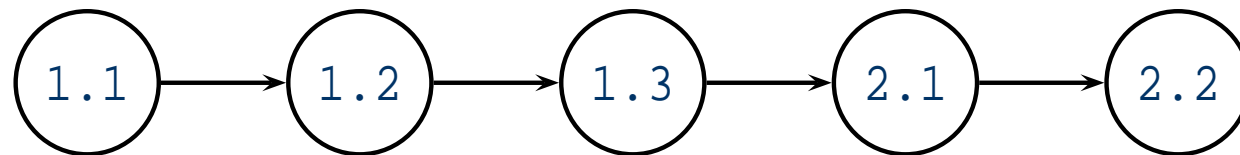
# Version tree

Checking in a file leads to a new revision of the file — a new **version**.



# Version tree

Checking in a file leads to a new revision of the file — a new **version**.

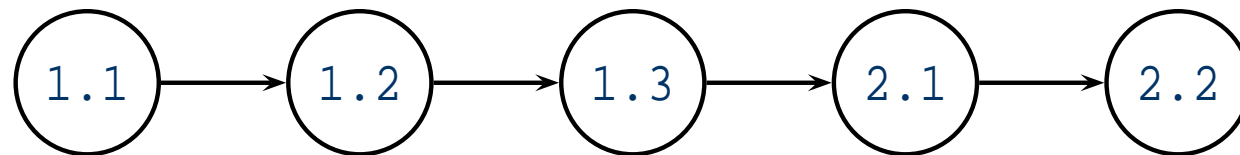


Caveat: The version of a file is *not* (necessarily) the same as the version of the complete system (the configuration).

Example: version 2.0 of the system consists of version 1.2 of `file.sig`, version 1.27 of `file.sml`, and version 1.5 of `main.sml`.

# Version tree

Checking in a file leads to a new revision of the file — a new **version**.



Caveat: The version of a file is *not* (necessarily) the same as the version of the complete system (the configuration).

Example: version 2.0 of the system consists of version 1.2 of `file.sig`, version 1.27 of `file.sml`, and version 1.5 of `main.sml`.

“Get me version 1.3”: `co -r1.3 file.sml`.

# Repository format

RCS: really no repository.

Version information for all revisions of `file.sml` stored in `file.sml,v` (the **RCS file**).

Last revision stored in full — previous revisions available via **backwards deltas**.

Delta: edit script detailing how to edit the file in order to go from one revision to another.

In RCS a delta is the output of `diff` (that is, line based).

Example: “get me version 1.3” is realized as getting the latest version (i.e., 2.2), applying backwards delta  $2.2 \rightarrow 2.1$  then  $2.1 \rightarrow 1.3$ .

# Controlling updates

**But** multiple developers may work on the same file.

Developer A

```
co file.sml
```

```
...
```

```
ci file.sml
```

```
...
```

Developer B

```
co file.sml
```

```
...
```

```
...
```

```
ci file.sml
```

# Controlling updates

**But** multiple developers may work on the same file.

Developer A	Developer B
co file.sml	co file.sml
...	...
ci file.sml	...
...	ci file.sml

Solution: serialize updates by **locking** files.

Workflow:	RCS	} file.sml locked
■ check item out	co file.sml	
■ edit item	...	
■ check item in	ci file.sml	

# Concurrent development

---

Since only locked files can be changed, a single developer may block further progress.

**CVS** (Concurrent Version System) rectifies that.

**Synchronization instead of locking.**

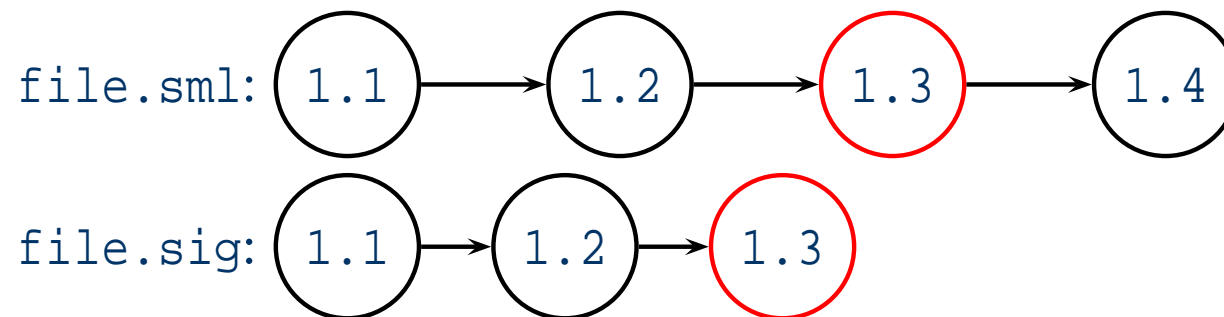
*The developer explicitly says when to do synchronization and when to publish changes.*

# Working copy and repository

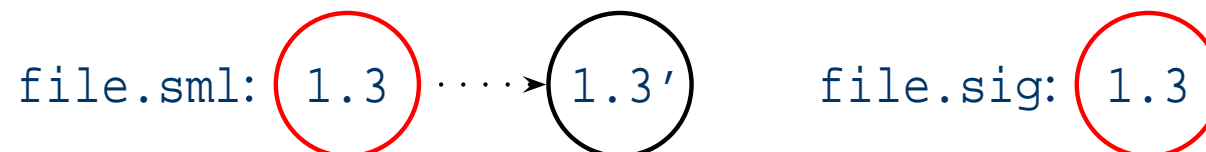
Each developer has a local working directory and is free to edit any file at all.

Trick: changes only visible **locally**!

Repository:



Working copy:

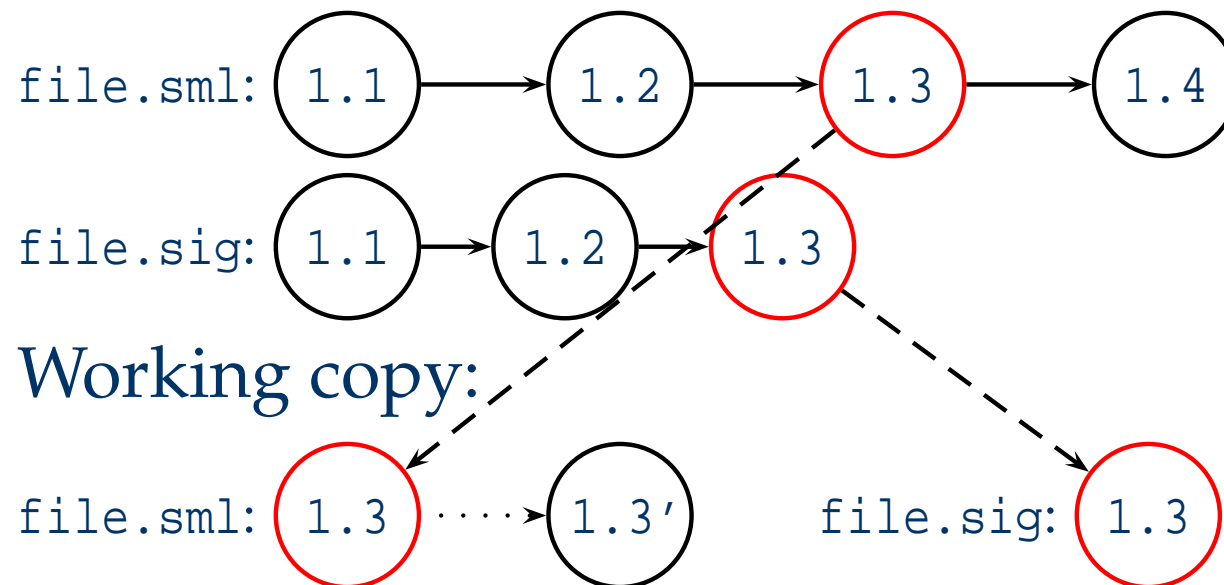


# Working copy and repository

Each developer has a local working directory and is free to edit any file at all.

Trick: changes only visible **locally**!

Repository:

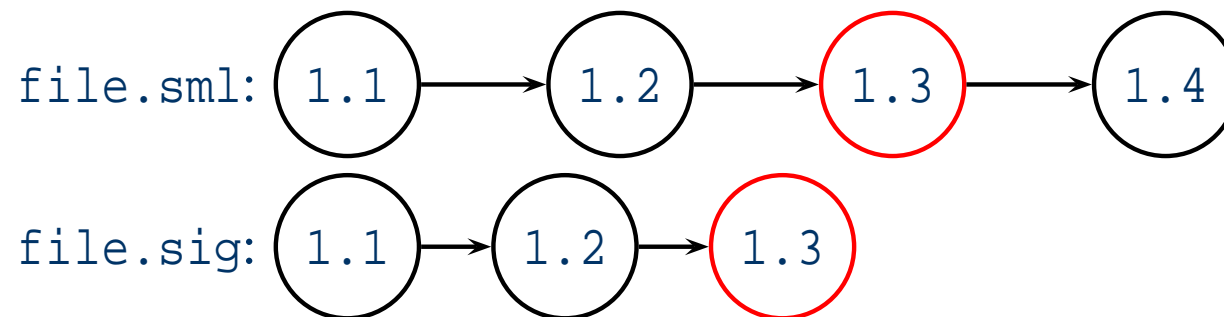


# Working copy and repository

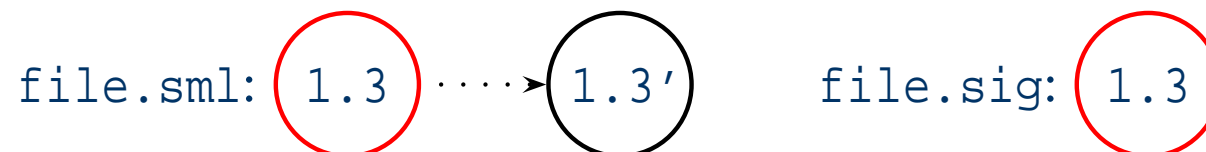
Each developer has a local working directory and is free to edit any file at all.

Trick: changes only visible **locally**!

Repository:



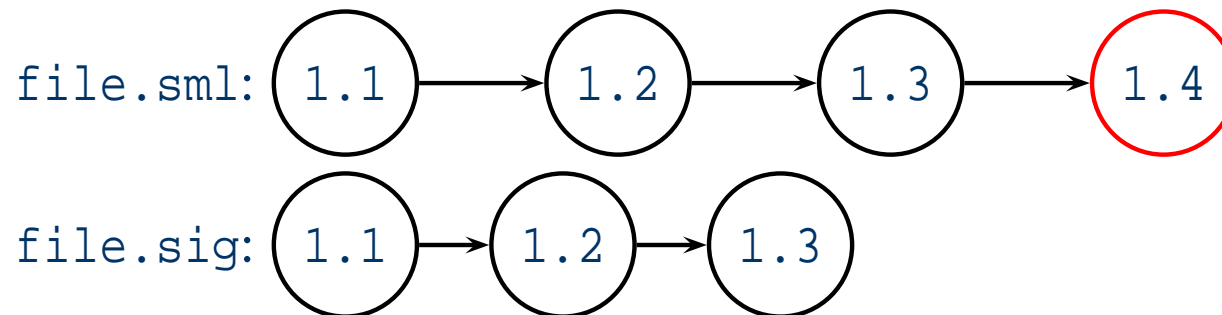
Working copy:



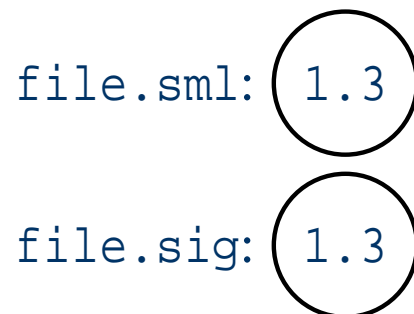
# Getting newest version

cvs update brings the working directory up-to-date with respect to the repository.

Repository:



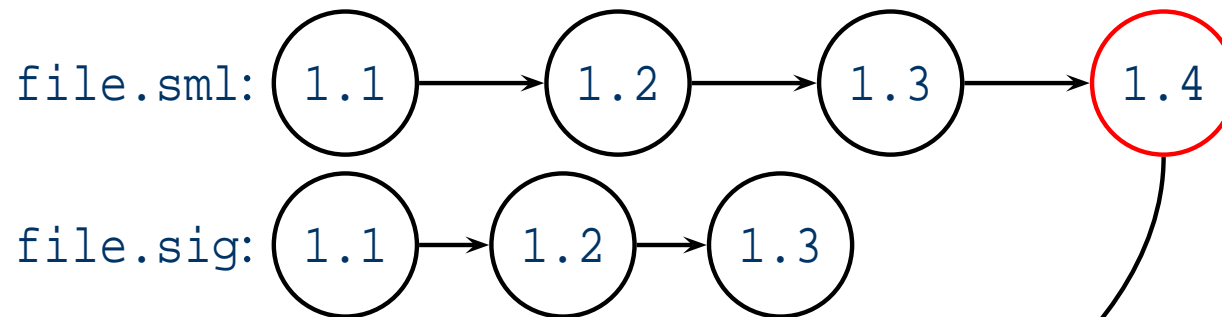
Working copy:



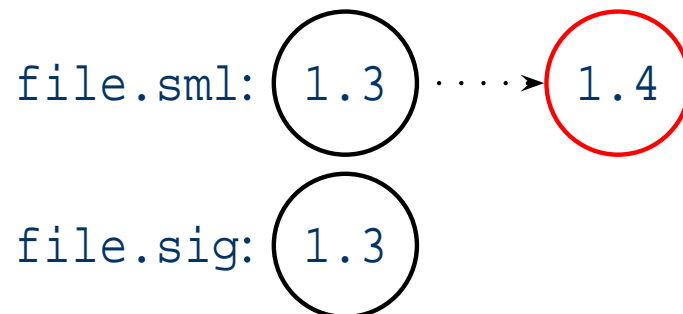
# Getting newest version

cvs update brings the working directory up-to-date with respect to the repository.

Repository:

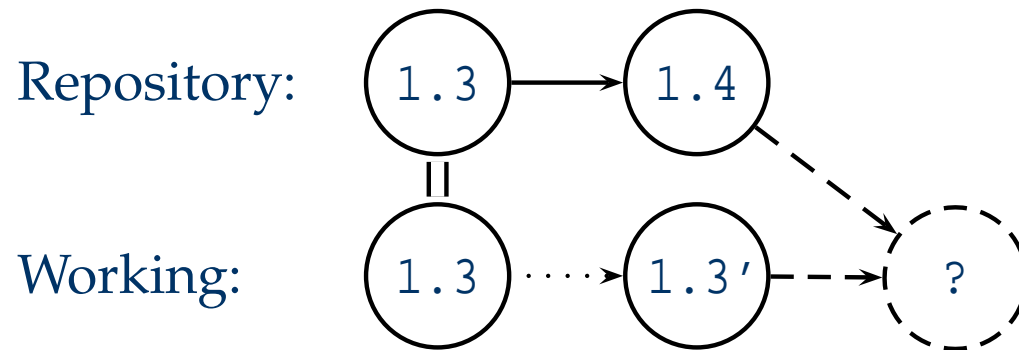


Working copy:



# Synchronization problem

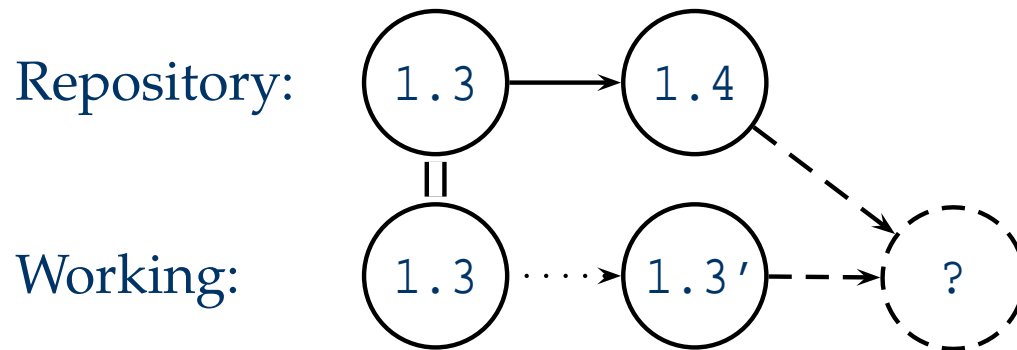
Changes to working copy *and* changes in repository.



We want ? to represent both the changes made in the repository and the changes to the working copy.

# Synchronization problem

Changes to working copy *and* changes in repository.



We want ? to represent both the changes made in the repository and the changes to the working copy.

Note that there is a common ancestor (namely 1.3)—we can thus use `diff3`.

Intuitively, `diff3 mine older yours` “subtracts” older from yours and “adds” that to mine.

# Automatic synchronization

`cv`s `update` (`diff3`) handles most synchronization issues automatically and *often* correctly.

Will detect some conflicts and let the user handle them manually.

Issues:

- line based;
- conflict region (2 lines?);
- only for text files;
- ...

# Synchronization failure

Example (original):

```
fun myFunction (x, y, z) =  
    let val res = (* complex computation  
                  involving x and y *)  
        (* more stuff *)  
    in ... res ...  
    end
```

# Synchronization failure

Example (modified by first developer):

```
fun myFunction (x, y, z) =  
  let val res = (* complex computation  
                involving x and y *)  
      (* more stuff *)  
  in ... res ...  
  end
```

# Synchronization failure

Example (modified by second developer):

```
fun myFunction (x, y, z) =  
  let val res = (* complex computation  
                involving x and y *)  
        (* more stuff *)  
  in ... res ... z ...  
  end
```

# Synchronization failure

Example:

```
fun myFunction (x, y, z) =  
  let val res = (* complex computation  
                involving x and y *)  
        (* more stuff *)  
  in ... res ... z ...  
  end
```

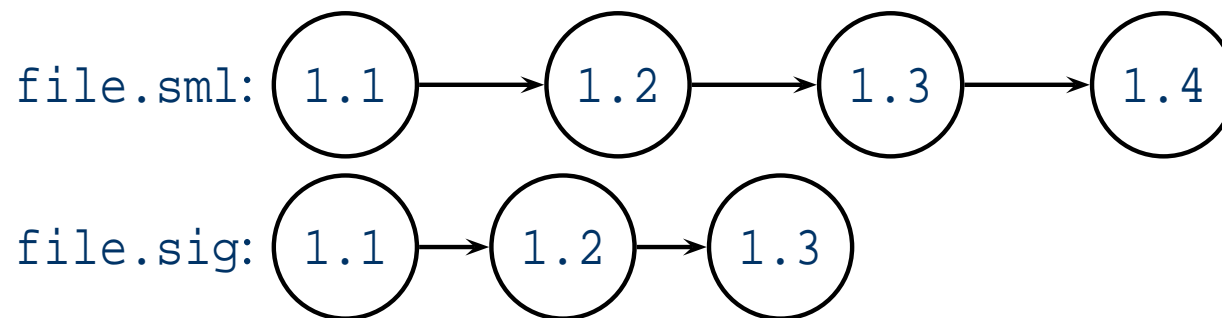
Synchronization result:

```
fun myFunction (x, y) =  
  let val res = (* complex computation  
                involving x and y *)  
        (* more stuff *)  
  in ... res ... z ...  
  end
```

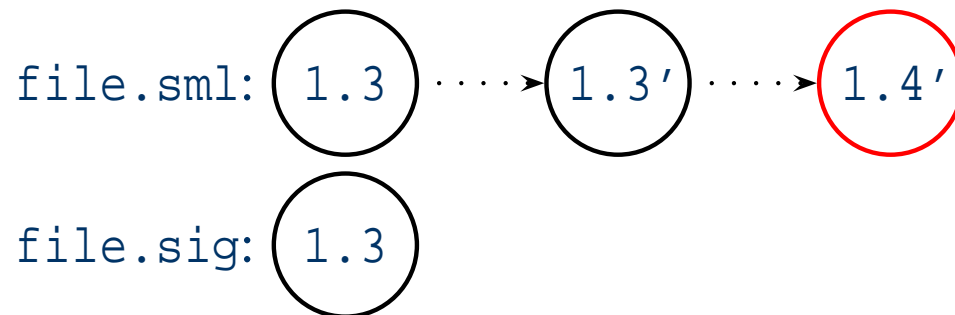
# Committing changes

Once satisfied with the current changes, execute `cvsc` `commit` to propagate them to the repository.

Repository:



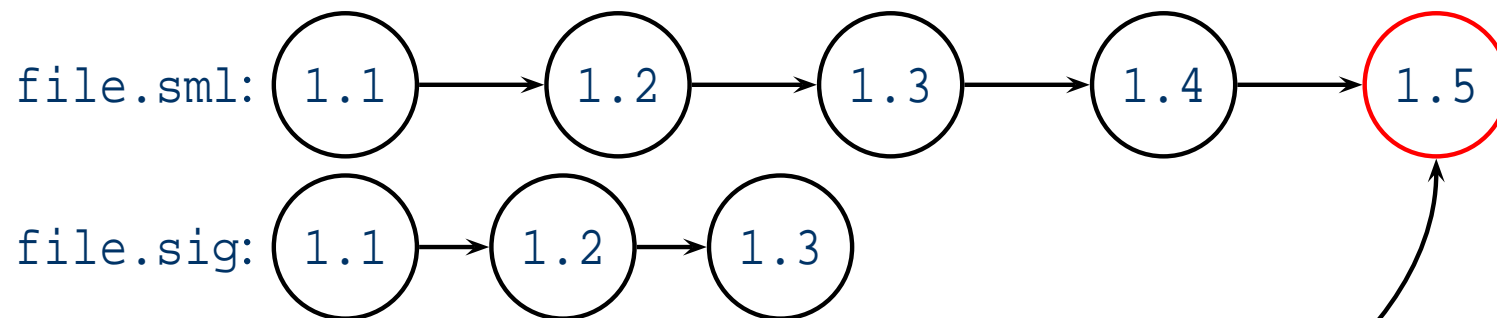
Working copy:



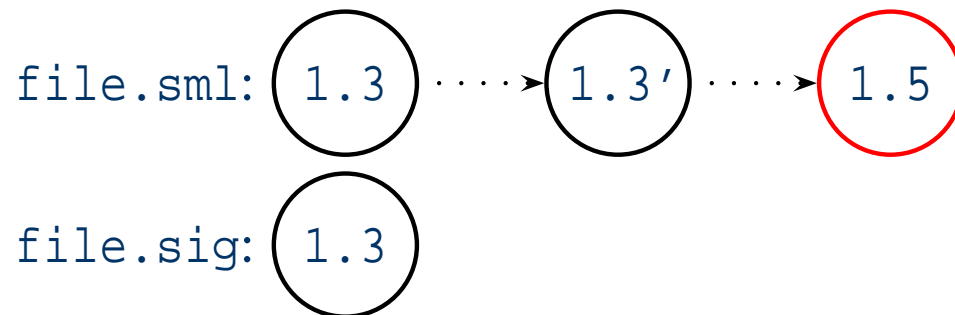
# Committing changes

Once satisfied with the current changes, execute `cv`s `commit` to propagate them to the repository.

Repository:



Working copy:



*Committing is only allowed if the file is up-to-date.*

# CVS vs RCS

---

- centralized repository;
- local working directory;
- synchronization instead of locking;
- directory hierarchies;
- ...

# Tagging

Files are identified by versions, a configuration is identified by a list of (file,version) pairs.

1.1	1.1	1.1	1.1
1.2	<u>1.2</u>	1.2	1.2
1.3		<u>1.3</u>	1.3
1.4			<u>1.4</u>
<u>1.5</u>			

# Tagging

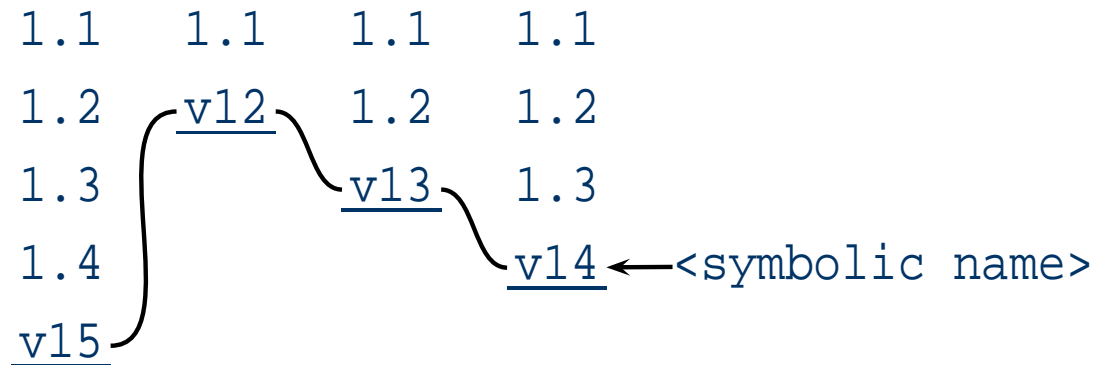
Files are identified by versions, a configuration is identified by a list of (file,version) pairs.

1.1	1.1	1.1	1.1
1.2	<u>1.2</u>	1.2	1.2
1.3	1.3	<u>1.3</u>	1.3
1.4			<u>1.4</u>
<u>1.5</u>			1.5

# Tagging

Files are identified by versions, a configuration is identified by a list of (file,version) pairs.

```
1.1  1.1  1.1  1.1
1.2  v12 1.2  1.2
1.3  v13 1.3
1.4  v14 ←<symbolic name>
v15
```



Attach a symbolic name to the files.

In CVS: `cvs tag <symbolic name>`.

# Tagging

Files are identified by versions, a configuration is identified by a list of (file,version) pairs.

1.1	1.1	1.1	1.1
1.2	<u>v12</u>	1.2	1.2
1.3	1.3	<u>v13</u>	1.3
1.4		<u>v14</u>	←<symbolic name>
<u>v15</u>		1.4	

Attach a symbolic name to the files.

In CVS: `cvs tag <symbolic name>`.

“Get me the configuration <symbolic name>”: `cvs checkout -r <symbolic name>`.

*Remember to always tag releases!*

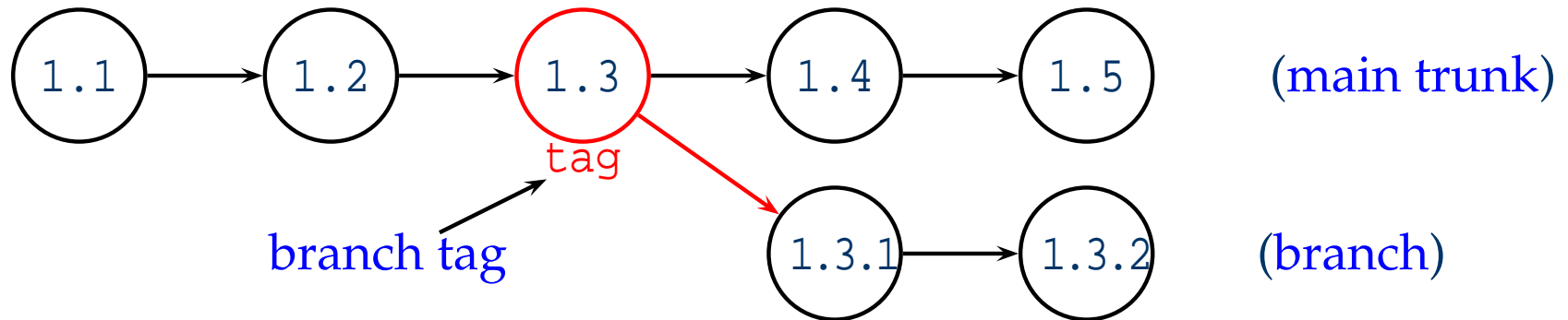
# Referring to revisions

---

- `cv update file.sml;`
- `cv update -r 1.3 file.sml;`
- `cv update -r mytag file.sml;`
- `cv update -D <date spec> file.sml`

# Branches

Instead of having a linear sequence of revisions, allow **branches** in the repository.



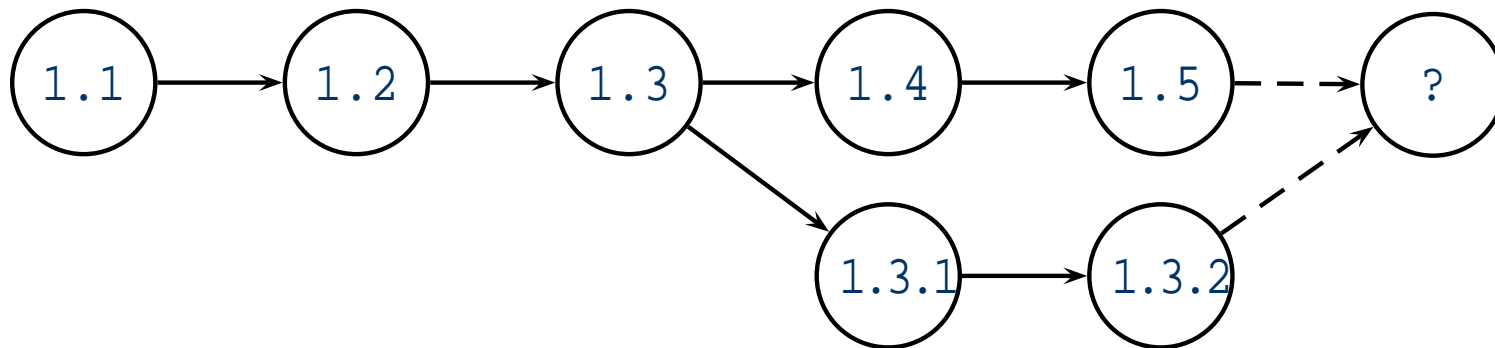
Branches are often useful, for example

- fixes to previously released versions;
- experimental changes;
- ...

CVS: `cvs tag -b <branch-tag>`.

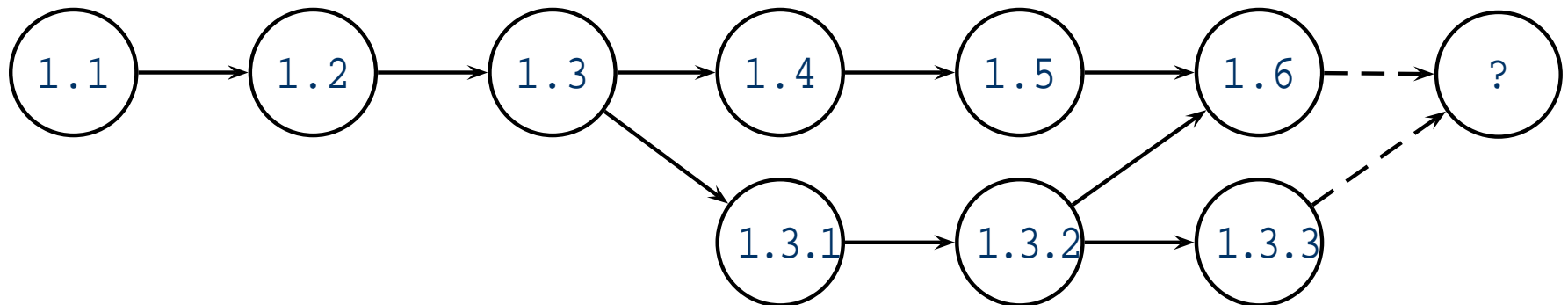
# Merging branches

Merging the main trunk and a branch



can be accomplished using `diff3`.

In CVS (on main trunk): `cvs update -j <tag>`.



# Branch support in CVS

In practice branches are not used very often.

CVS support is lacking and thus people have come up with various advice/policies/commandments/....

- always tag before you branch  
(`cvs tag <branch>-ROOT`);
- then branch (`cvs tag -b <branch>`);
- always include `-ROOT` tag for merges  
(`cvs update -j <branch>-ROOT -j branch`);
- remember where you last merged  
(`cvs -F -r <branch> <branch>-ROOT`);
- have fun.

# CVS

---

- Available everywhere (Unix, Windows, VMS, Crays, ...);
- Web page: <http://www.cvshome.org/>;
- FAQ-O-Matic:  
<http://www.loria.fr/~molli/fom-serve/cache/1.ht>

# Subversion

Subversion is the CVS that the CVS users *wish* they had.

- versioning of file system metadata (directories, file renamings and removals, etc);
- atomic commits (files don't have their own version numbers);
- better binary file handling;
- constant time and space tagging and branching.

Version 1.0 not yet released (milestones leading to the release scheduled).

# (Other) CM systems

- SCCS (Bell Labs, 1975);
- RCS (Tichy, 1982);
- CVS (Grune, 1986; Berliner, 1990);
- Subversion (<http://subversion.tigris.org/>);
- Arch (<http://regexps.srparish.net/www/>);
- OpenCM (<http://www.opencm.org/>).
- Visual SourceSafe  
(<http://msdn.microsoft.com/ssafe/>);
- ...

# Getting started with CVS

- Init repository `cv`s -d <path> init export  
CVSR00T=<path>
- Begin a module `cd` source  
`cv`s import <dir> <name> start
- Add a file `touch` file.sml  
`cv`s add file.sml
- Commit file `cv`s commit [file.sml]
- Get updates `cv`s update