Subversion: A True Community Modeling Paradigm

Jose-Henrique Alves

The WAVEWATCH III Team + friends
Marine Modeling and Analysis Branch
NOAA / NWS / NCEP / EMC

NCEP.list.WAVEWATCH@NOAA.gov
NCEP.list.waves@NOAA.gov

Overview

- Developers' Best Practices
- Model development paradigms: why use Subversion
- How Subversion works
- How we use Subversion

Best practices

- For those who want to modify / contribute to WAVEWATCH III, a best practices guide is available.
- Note that as a part of the license, additions made to the model have to be offered to NCEP for inclusion in future model distributions.
- Best practices cover :
  - Programming style
  - Adding to the model.
  - Manual and documentation.
  - Subversion repository.
  - Regression testing.

Best practices guide

Best practices

Programming style:

- Use WAVEWATCH III documentation style (see templates).
- Use coding style:
  - Free format but layout as in old fixed format.
  - Upper case for permanent code, lower case for temporarily code. Latter can be included as permanent testing using `!/Tn` switches.
- Maintain update log at header for documentation.
- Embed all subroutines in modules or main programs, using naming convention outlined before.
- Follow FORTRAN 90 standard, with best practices as outlined in section 2 of the guide.
- Provide appropriate documentation in LaTeX format for inclusion in the manual.

Best practices

Adding to the model (with NCEP subversion access)

- Same rules apply as for those without svn access with following exceptions:
  - NCEP code managers will assign switches to new sources and propagation scheme to be used instead of the 'X' switches.
  - Developers will be responsible for integration in the data structure:
    - Do this only after rigorous testing of self-contained system.
  - NCEP code managers will add new code to the `TRUNK` of the repository. Changes to be provided relative to most recent `TRUNK`, not to most recent distributed version

Best practices

Manual and documentation.

- Provide full LaTeX documentation for inclusion in the manual:
  - NCEP svn users have access to manual, and are expected to add to it directly.
    - NCEP will provide editing.
  - Others provide separate files.
    - NCEP will integrate.
  - Use BibTEX.
  - Use dynamic references to equations, figures and tables only.

Best practices

Testing

- Regression testing are based on previous WAVEWATCH III tests and new materials and tools provided by Erick Rogers and Tim Campbell from NRL Stennis.
  - nrltest, will replace current test directory in near future

Model Development Framework

Developer's Universe

- Development work involves including new features to the wave model, upgrading existing features, as well as bug fixing, clean up etc
- Modifying the code will lead to changes in the whole system
- Developers should be aware of what all the components are and what needs to be changed
- Approach changes as a system, looking at all components:
  - .ftn files
  - Pre-compiler (*ww3_make* and associated programs)
  - FORTRAN compiler specifics (*comp* and *link* scripts)
  - Regression tests (make new friends, but keep the old)
  - Integrating a lot of developers, developing a complex system: communication + centralized repository = SVN!

Model Evolution

Communication: the big leap forward

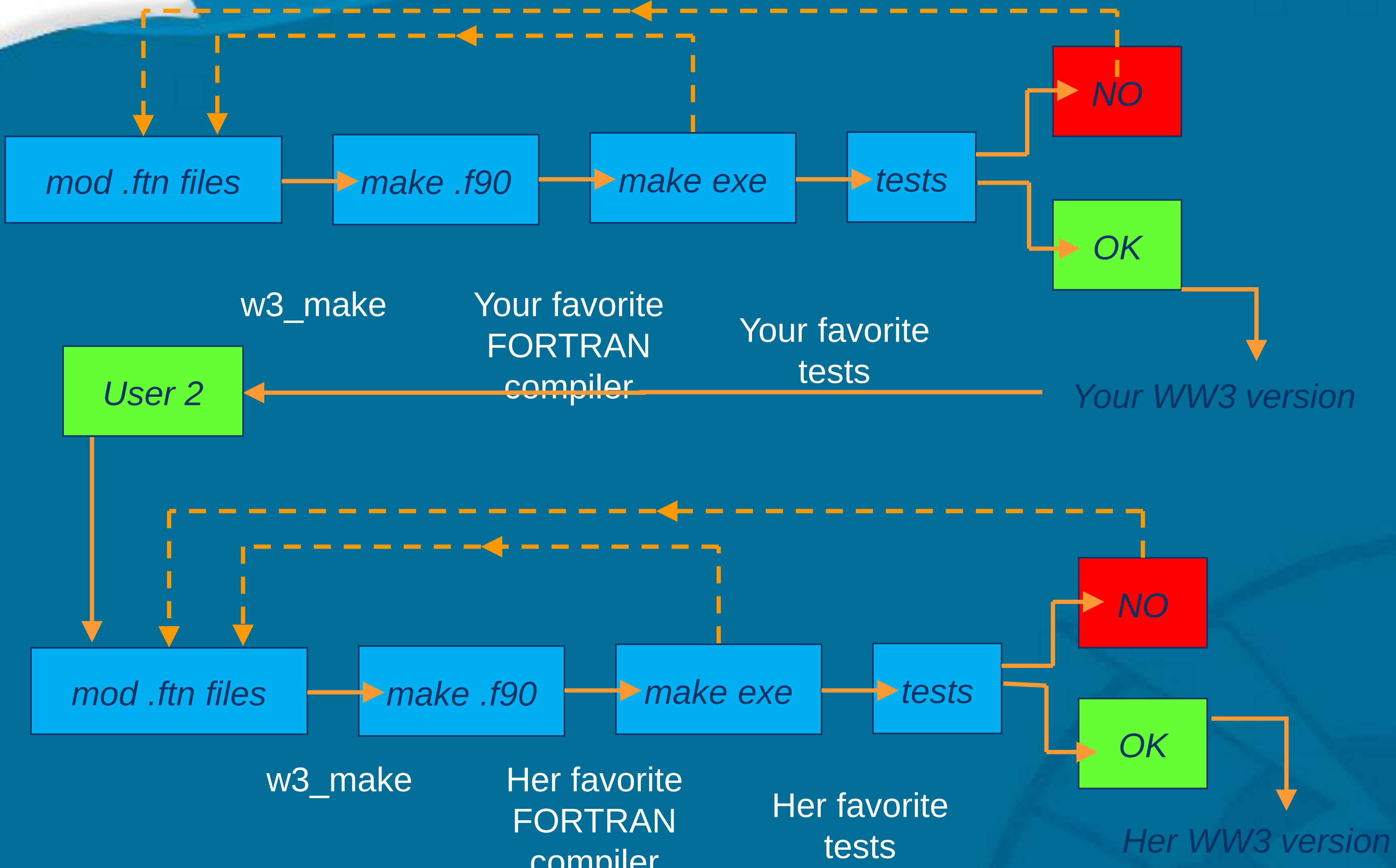
The “Usual” Development Model

A Self Disgruntling Model

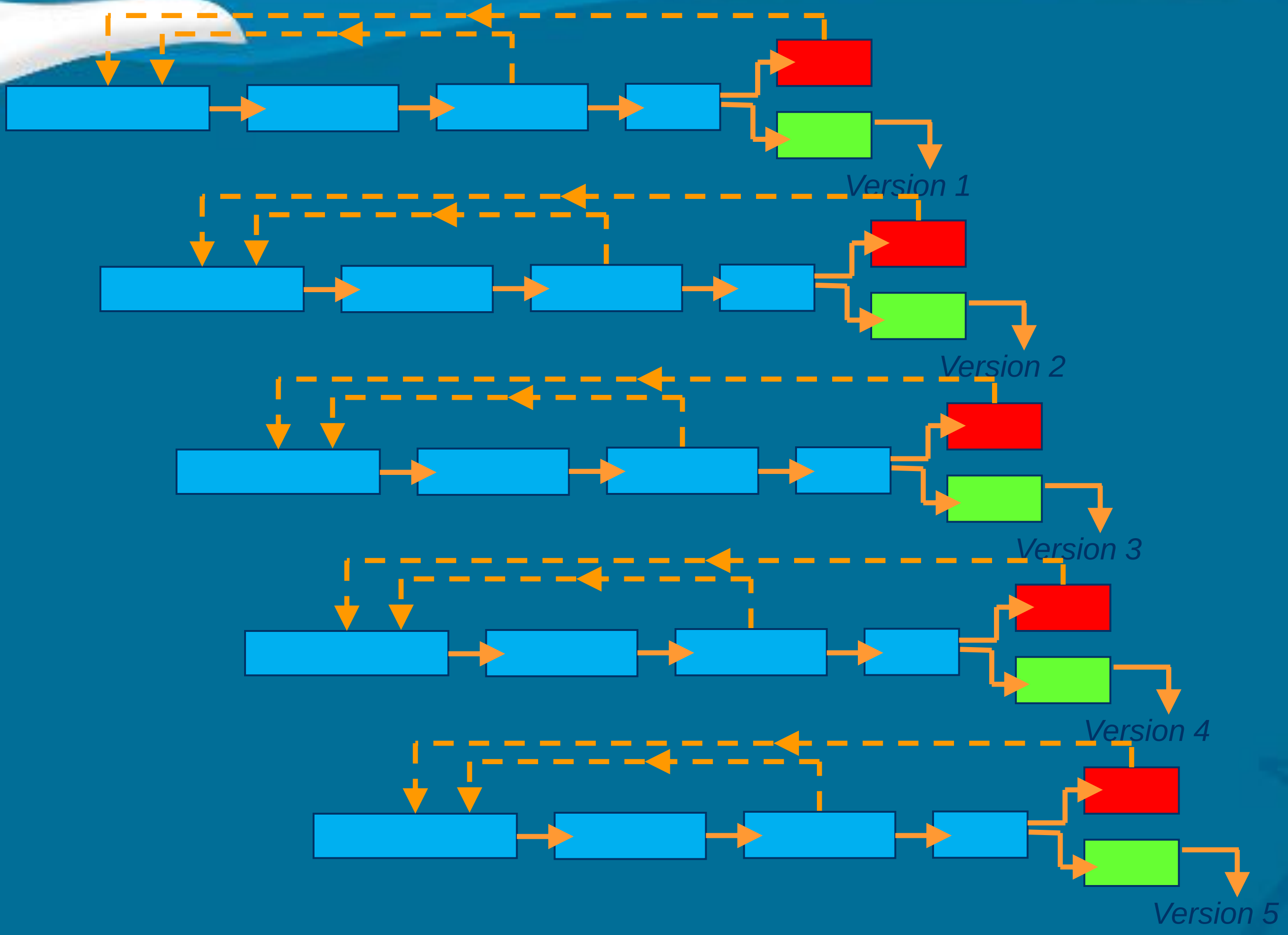
- Individual development
  - Each one comes up with a different solution
  - Each one has his/her own code writing standards
- End up with a set of many versions, diverging = diversions

- Clunky communication
- Open loops: difficult to debug, reconcile
- Time consuming

The Usual Development Model



The Usual Development Model



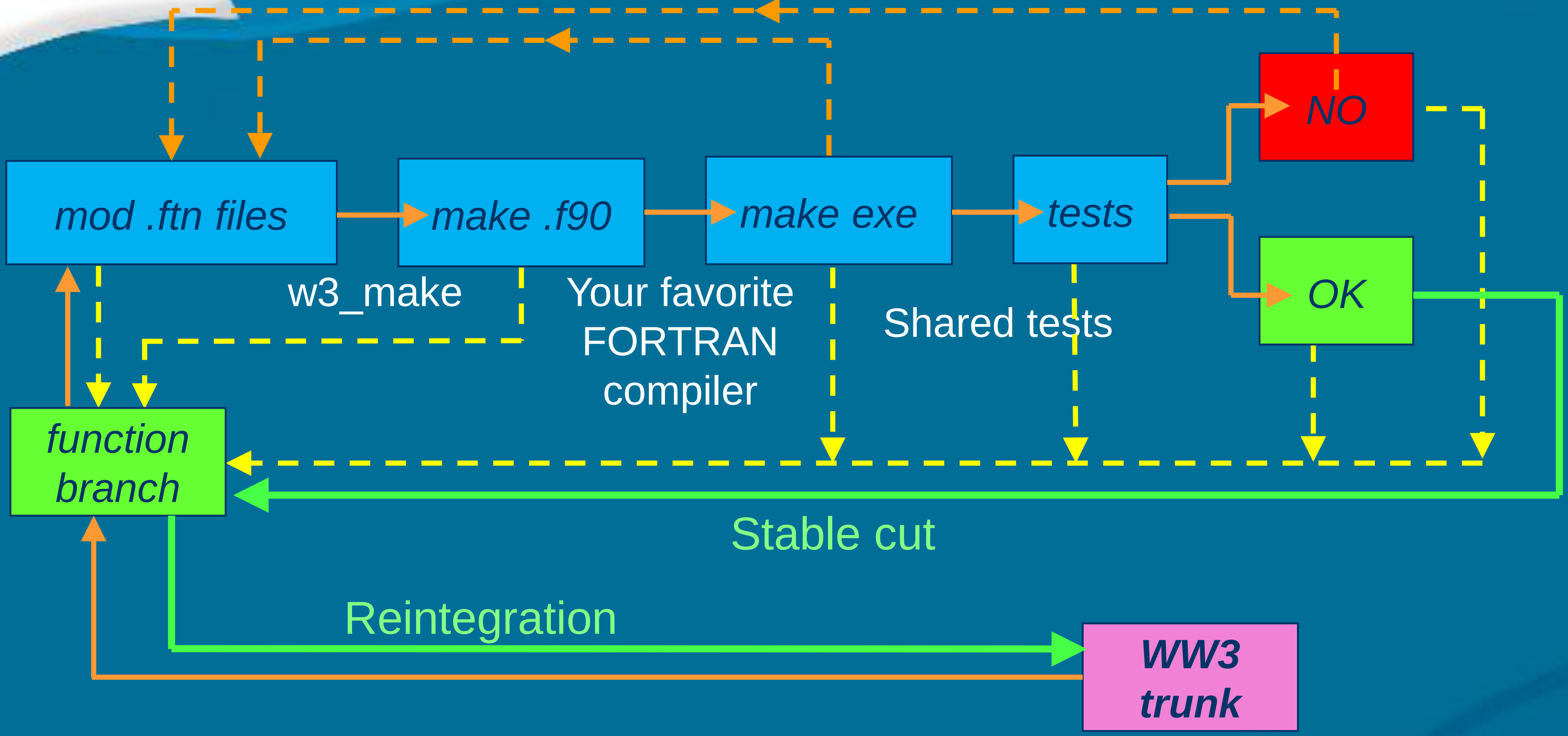
The Community Development Model

A Self Sufficient Model

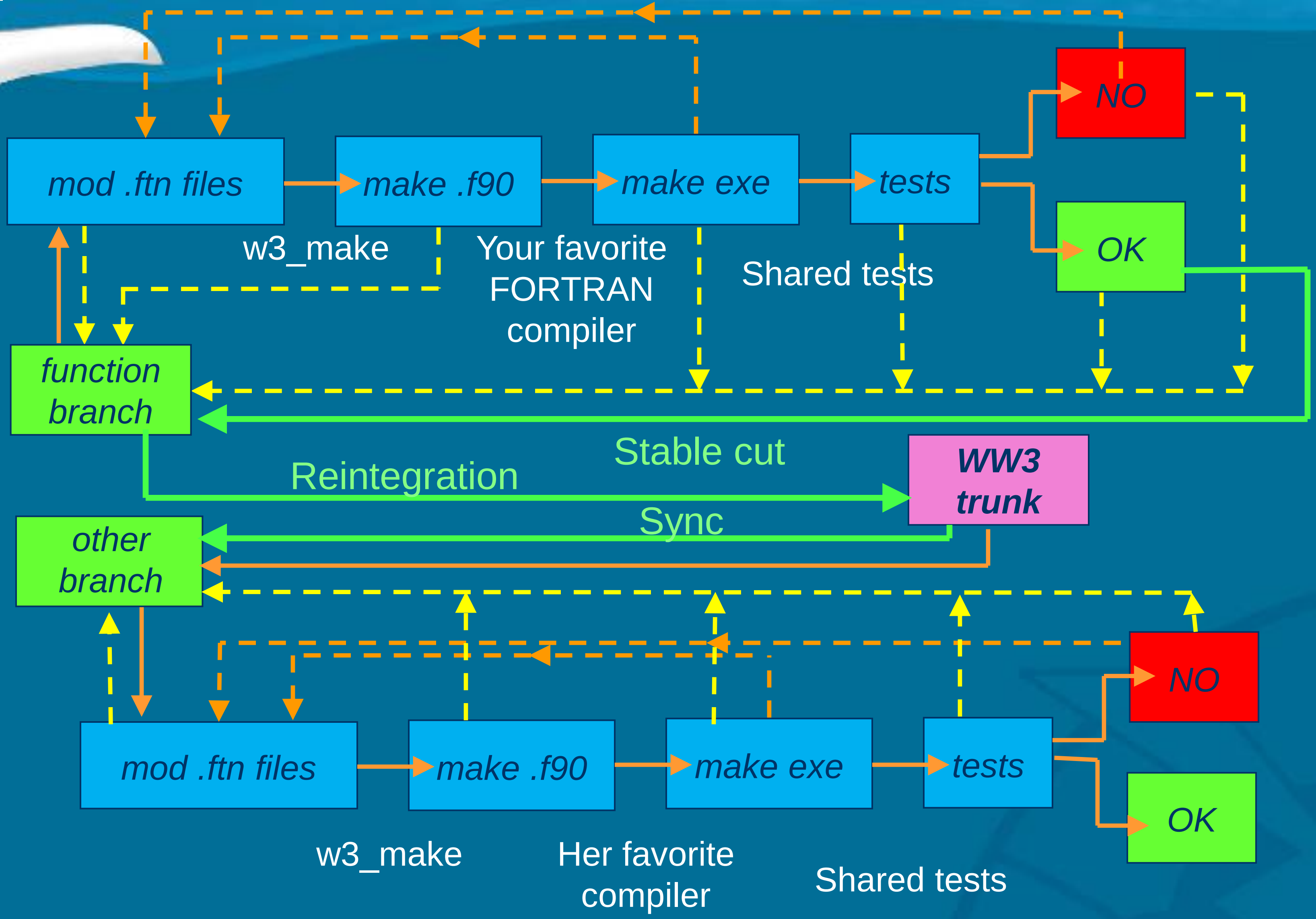
- Collective, simultaneous development
  - Each one comes up with a different solution
  - Solutions are integrated and reconciled
  - All should follow same code writing standards
- Subversion: a single version, with many options = convergence

- Streamlined communication
- Consistent codes, easy to debug and reconcile
- Time saving

The Community Development Model



The Community Development Model



Context: What is Subversion?

- Subversion is an open source *centralised* version control system (i.e. accesses a central repository) that allows one or more users to easily share and maintain collections of files.

- IOW: it's a database that keeps track of *changes* made to files

What it is not.

- Magic.
- It is not a substitute for management.
- **It is not a substitute for developer communication.**

There is nothing inherently special about subversion. Many other revision control systems exist.

- Git, Mercurial, CVS, Bazaar, darcs, etc

Reference: **http://svnbook.red-bean.com**

EMC and NCO Subversion Servers

NCEP repository

- URL: `https://svnemc.ncep.noaa.gov/projects`/ww3

Repository organisation (1)

Project ww3 has four main directories:

trunk

- The main line of development.
- Typically in an "almost ready for release" state.

branches

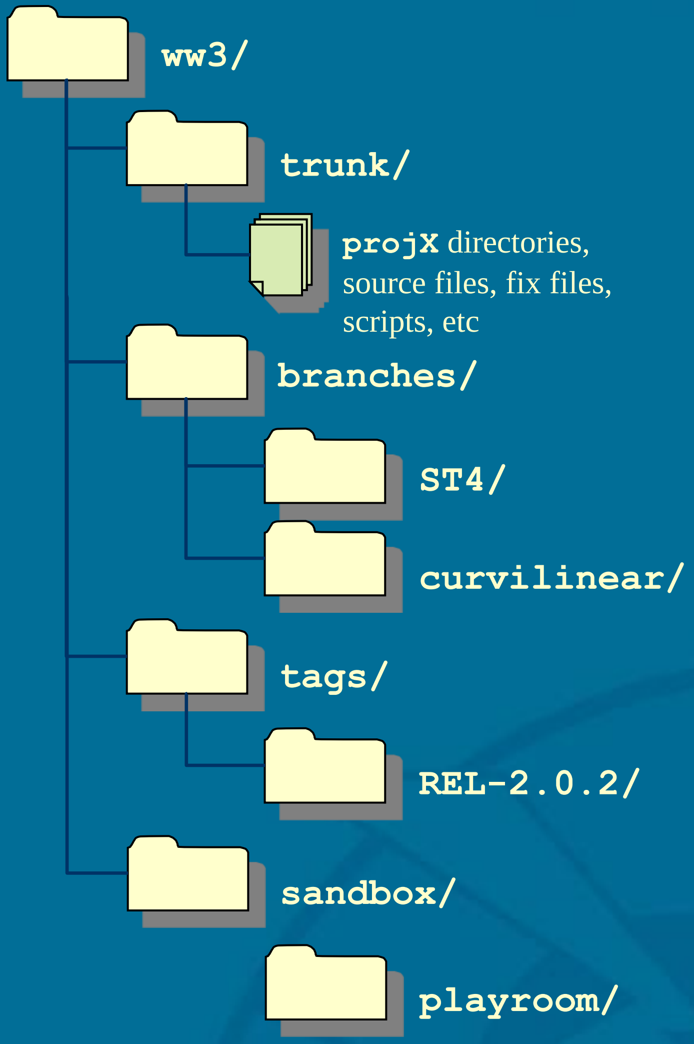
- Where development is done.
- Experimental development branches

sandbox

- Are available for developers to store branches that are kept out of sync on purpose

tags

- Snapshots and releases
- No development



Repository organisation (2)

`trunk/`, **`branches/`** , **`tags/`** , and **`sandbox/`** are just directories. Subversion has no concept of a "branch" or "tag". When the trunk is copied, it is a branch or tag *only* because we attach that meaning to it.

Repository copies (i.e. branches and tags) are "cheap". That is, you are not actually copying all of the source/data - you are just creating a reference to a particular revision.

Subversion stores only the diff content when changes are made, still referencing to the originating files.

Repository revision numbering

When subversion repo is created, starts at revision 0.

Each subsequent commit to the repository increments the revision number by 1.

The revision number is **repository-wide** so *any* commit in *any* project increments the revision number

NCEP, currently has 69 registered projects, all share revision numbers

- Typically, don't need to worry about a revision number value
- But need to be aware of the numbers related to your codes
- The trac source browser provides easily navigable information about what was done at which revision.

Checking out files

Two ways of getting repository content:

1. Grabbing an unversioned copy: svn export
2. Checking out a versioned copy: create local workcopy

```
$ cd $HOME/workcopy
```

```
$ svn checkout URL[@rev] [wcPATH]
```

The repository URL of the project you want to checkout

The revision number of the project you want to checkout. Defaults to **HEAD**, i.e. the latest version.

The path where you want to create your working copy. Defaults to the *basename* of URL.

Editing existing files

Once you have created a working copy of your project(s), edit, compile, debug, and test as usual

The files are the same as when they were unversioned

To avoid creating junk (compiled code, output etc) in your svn folder, a good practice is to create symbolic links

```
$ cd $HOME/workcopy
$ mkdir svn; cd svn
$ svn checkout URL[@rev][wcPATH]
$ cd ..; ln -s ./svn/* .
```

Changes made to linked files are propagated to svn area.

Compiled codes and outputs etc are kept out of svn area.

Adding new files

When you create a new file – for svn means creating from scratch or copying from outside the versioned workcopy – it remains local until you commit.

```
$ touch newfile.txt
$ cp outsider.txt
$ svn status
?       newfile.txt
?       outsider.txt
$ svn add newfile.txt outsider.txt
A       newfile.txt
A       outsider.txt
$ svn commit -m "1st commit" newfile.txt outsider.txt
Adding              new_script.sh
Transmitting file data .......
Committed revision 12083.
```

If adding an entire directory, this will work recursively on all its contents.

Deleting files

Deleting using local command has no effect to versioning.

```
$ rm old_script.sh
$ svn status
!         old_script.sh
$ svn delete old_script.sh
D         old_script.sh
```

Versioning requires svn command, which also deletes the file from your working copy.

Local delete does not affect repository: commit required!

```
$ svn commit –m “Removed file”
old_script.sh
Deleting       old_script.sh
Transmitting file data .......
Committed revision 12084.
```

But I didn't want to delete it!

If you **svn deleted** now want it back to have it available in future revisions/checkouts, need to undo the commit.

This can be done using **svn merge** but specifying the revision numbers in reverse order (reverse merge).

```
$ svn merge -r12084:12083 .
--- Reverse-merging r12084 into '.':
A    old_script.sh
$ svn status
A  +     old_script.sh
$ svn commit -m "undoing deletion in r12084"
Adding           old_script.sh
Transmitting file data ....
Committed revision r12087.
```

Undoing local changes

If you haven’t committed changes: **`svn revert`**

```
$ svn status changed_script.sh
added_script.sh
A           added_script.sh
M           changed_script.sh
$ svn revert changed_script.sh
added_script.sh
Reverted ‘changed_script.sh’
Reverted ‘added_script.sh’
```

If there are no targets, **`svn revert`** will do nothing.

IOW: revert will make you *lose* any local changes you have made.

Checking file status

`svn status` check the status of a file in your working copy:

- Check if you have made any local changes.
- Check for new changes in the repository (-u flag)

Local-changes case:

```
$ svn status
?         SensorInfo
!         source.comment
M         osrf__load_viirs.pro
D         viirs-i_npp-S1-04.inp
A         viirs-iS1
A   +     viirs-iS1/viirs-i_npp-04.inp
```

The "**?**" indicates this is an *unversioned* file in your working copy.

The "**!**" indicates this is a versioned file that is missing from your working copy.

The "**M**" indicates this file has been *locally* modified.

The "**D**" and "**A**" indicate these are versioned files that have been scheduled for deletion and addition respectively.

The "**A**" with the "**+**" in the 4th column indicates this file has been scheduled for addition *with history*.

Checking file status

Use **svn status** to check for changes in the repository:

```
$ svn status --show-updates #or -u
M           10564 CRTM_Fastem4.f90
?                 Reflection_Correction.f90
       *     9675 SensorInfo/SensorInfo
       *     9675 MW_SensorData/MW_SensorData_Define.f90
       *          Create_SpcCoeff/Create_SpcCoeff.f90
       *          Create_SpcCoeff/Makefile
       *          Create_SpcCoeff/make.dependencies
Status against revision: 10738
```

With a trailing revision number, the 8th column ‘*’ indicates the local file is outdated (there’s a newer version in the repository).

With no trailing revision number, the ‘*’ in 8th column indicates a file was added to the repository, but never existed in the working copy.

Updating files

After changes are done, you want to commit them to the repository, but what if another user has changed and committed the same file(s)?

- Communication between developers is important
- Use of **`svn status`** helps the process

Subversion handles this potential "overlap" by requiring you to update your working copy to the current repository version before you can commit.

```
$ svn update run_modelX.sh
U   run_modelX.sh
Updated to revision 12034
```

This merges changes (assuming no conflicts) found in the repository (only run_modelX.sh) into your working copy.

Updating files: solving conflicts

What if your **`svn update`** subcommand produces the following:

```
$ svn update run_modelX.sh
C   run_modelX.sh
Updated to revision 12034
```

Look at the **`svn status`** output:

```
$ svn status
?       run_modelX.sh.mine
?       run_modelX.sh.r11987
?       run_modelX.sh.r12034
C       run_modelX.sh
```

The file as it existed in your working copy prior to the update

This is the file that was the **BASE** revision before the update. That is, the file that was checked out before the latest edits.

This is the file that the Subversion client just received from the server due to the update. This file corresponds to the **HEAD** revision of the repository.

The merged file containing the conflict between conflict markers:

>>>>>

=====

<<<<<

Updating files

What if there *is* a conflict?

To resolve conflicts we use **`svn resolve`** with **`--accept`**

1. To keep the version that you last checked out before your edits (in example the **`.r11987`** file), use **`base`** argument

```
$ svn resolve --accept base run_modelX.sh
Resolved conflicted state of run_modelX.sh
```

2. To keep the version that contains *only* your edits (the **`.mine`** file), use the **`mine-full`** argument

```
$ svn resolve --accept mine-full run_modelX.sh
```

3. To keep version update pulled from the server (**`.r12034`** file, *discards all your edits)*, use **`theirs-full`**

```
$ svn resolve --accept theirs-full run_modelX.sh
```

4. To edit the conflicted text "by hand" (follow conflict markers) and use **`working`**

```
$ svn resolve --accept working run_modelX.sh
```

Updating files

What if there *is* a conflict?

Another option is to throw out your changes and start your edits over again, run **`svn revert`**

```
$ svn revert run_modelX.sh
Reverted 'run_modelX.sh'
```

If you choose to revert to avoid the conflict, you do not have to run the **`svn resolve`** subcommand.

⚠ Remember though: Using **`svn revert`** will discard all your local edits!

Committing files

When working copy is up to date, commit changes to the repository using **`svn commit`**

```
$ svn commit run_modelX.sh
      <enter log message>
Sending             run_modelX.sh
Transmitting file data ......
Committed revision 12078.
```

Subversion will start an editor allowing you to enter a log message that describes the change.

- Briefly describe changes and *why* they were made.
- Shared format allows logs to become ChangeLog files.

Exiting the editor, Subversion will commit changes to the repository, where they will become the latest version of the shared code, visible to all users.

Trunk to branch merges: Why?

Best Practice: Frequent updates on branches minimize the likelihood of conflicts (the branch will be merged back into the trunk someday, right?)

- Usually referred to as "sync" [with the trunk]
- Also makes available to the branch any updates or bug-fixes that have been implemented in the trunk (or merged into the trunk from a different branch.)
- Each development team needs to determine the "best" frequency of regular trunk→branch updates. Once a week?

First merges may be a painful experience, but persistence pays off for all!

- Next: Zen & the Art of Merging (or, How is it done, now?!)

What is merging? (1)

From the Subversion manual:

The main source of confusion is the *name* of the command. The term “merge” somehow denotes that branches are combined together, or that there's some sort of mysterious blending of data going on. That's not the case. A better name for the command might have been

svn diff-and-apply

because that's all that happens: two repository trees are compared, and the diffs are applied to a working copy.

- Subversion’s differencing algorithms work on text (including non-English, non-Roman alphabet languages apparently) as well as binary files.
- Even though subversion’s diff subcommand only provides output for textual differences, internally subversion can handle binary file differences efficiently between versions.

Merge command syntax (v1.4)

Examples of **svn merge**:

- Specifying all three arguments explicitly

```
$ svn merge \
      http://svnemc.ncep.noaa.gov/projects/modelX/trunk@100 \
      http://svnemc.ncep.noaa.gov/projects/modelX/trunk@200 \
      my-branch-working-copy
```

- Shorthand (comparing two different revisions of same URL)

```
$ svn merge -r 100:200 \
      http://svnemc.ncep.noaa.gov/projects/trunk \
      my-branch-working-copy
```

- Working-copy argument optional (defaults to current directory)

```
$ svn merge -r 100:200 \
      http://svnemc.ncep.noaa.gov/projects/trunk
```

We' ll be discussing this form for the trunk-to-branch merging

Merging the first time (1)

1. Determine the trunk revision from which the branch was created; let's say it was `1000`.
2. Determine the current trunk revision in repository; say it is `1050` (`svn` also recognises `HEAD` for this case).
3. Test the `merge` subcommand with `--dry-run` switch,

```
$ svn merge --dry-run \
           -r 1001:1050 \
           https://.../projects/modelX/trunk .
```

This will list all the changes that *will* occur, *without actually doing anything to your branch working copy*, so you can see if there are any conflicts.

Merging the first time (2)

4. If there are no conflicts, or their number is reasonable (more later), reissue `merge` *without* the `--dry-run` switch to perform the merge in your branch working copy.

5. Deal with any files in conflict and resolve them.

6. Commit the merge changes with a useful log message.

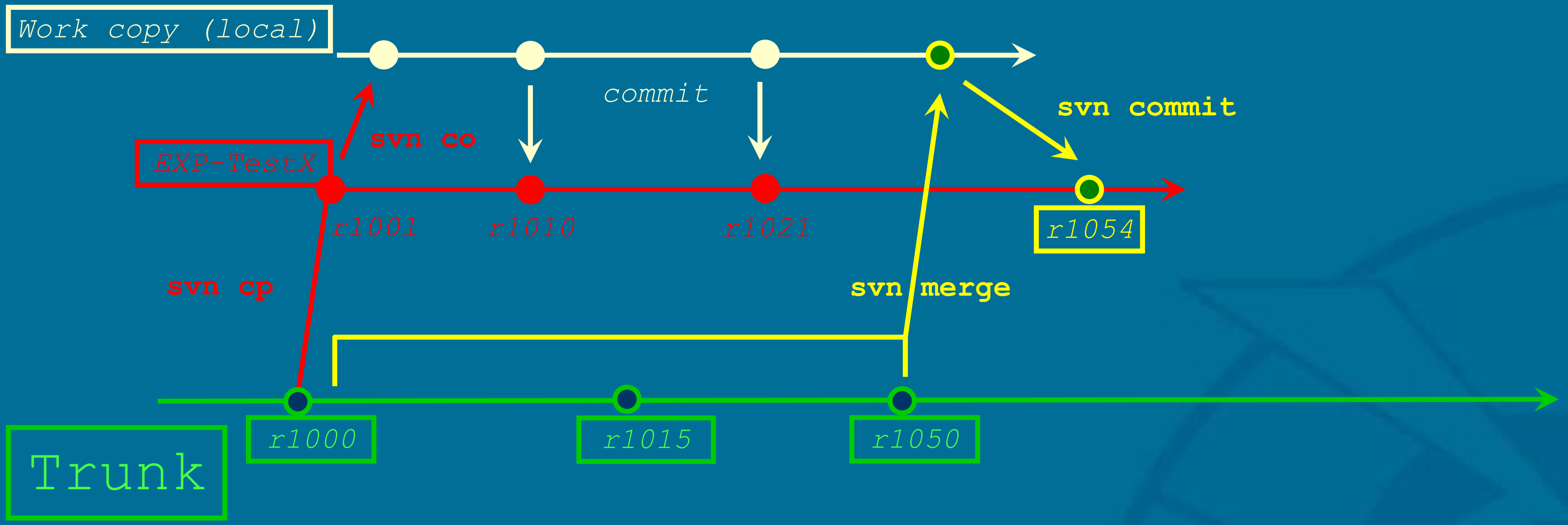
```
$ svn commit -m \
"Synced EXP-TestX branch with latest trunk r1001:1050"
```

Merging the first time (3)

The Art of Merging (or, how was that done, now?!)

```
$ svn merge -r 1000:1050 https://.../trunk .
```

```
$ svn commit -m "EXP-TestX branch. Merged trunk r1001:1050"
```



Trunk to branch merges: Next time

1. Determine the end revision of the last trunk merge into the branch by looking at the log message for the branch,

```
$ svn log | more
```

   In our example that was `1050`.

2. Follow the same procedure as for the first time.

Without manually tracking the merged revisions in the commit log message, there is *no simple way* to determine which revisions from the trunk have been merged!

If you remerge previously merged revisions, you will typically get many, many conflicts. If this happens, it's a clue that the merge revision range is probably incorrect.

Handling merge conflicts (1)

What if your **svn merge** produces the following:

```
$ svn merge -r 1001:1050 \
                 https://.../projects/modelX/trunk .
C     run_modelX.sh
```

Look at the **svn status** output:

```
$ svn status
?      run_modelX.sh.working
?      run_modelX.sh.merge-left.r1001
?      run_modelX.sh.merge-right.r1050
C      run_modelX.sh
```

The file as it existed in your working copy prior to the merge.

This is the initial, or “left”, side of the double tree comparison.

This is the final, or “right”, side of the double tree comparison.

Merged file with the conflict between conflict markers:

>>>>>
=====
<<<<<

Handling merge conflicts (2)

To resolve the conflict `svn resolve` with `--accept`

1. To keep the version that exists in your branch, use the `mine-full` or `mine-conflict` argument

```
$ svn resolve --accept mine-full run_modelX.sh
```

2. To keep the version that was pulled from the trunk, use `theirs-full` or `theirs-conflict`

```
$ svn resolve --accept theirs-full run_modelX.sh
```

3. To edit the conflicted text "by hand" (follow conflict markers) and use `working`

```
$ svn resolve --accept working run_modelX.sh
```

4. Typically, you never want to use the `base` argument in this case.

The TRAC Wiki

- The TRAC wiki: using your new friend to help out understanding what’s going on
  - Directly linked with svn repository change log
  - Real-time availability of information about repository dynamics
- Major features
  - General information page (wiki)
  - Timeline
  - Roadmap: milestones, model versions
  - Browse source
    - View repository structure
    - View change sets and logs
  - Tickets
    - Viewing
    - Opening new

Best practices

- For those who want to modify / contribute to WAVEWATCH III, a best practices guide is available.
  - Link via the wiki page of the SVN TRAC
- Note that as a part of the license, additions made to the model have to be offered to NCEP for inclusion in future model distributions.
- Best practices cover :
  - Programming style
  - Adding to the model.
  - Manual and documentation.
  - Subversion repository.
  - Regression testing.

Best practices guide

Best practices

Common SVN Server Practice

- Server will follow the structure:
  - trunk – will contain the “developed” WW3 version
  - branches – contain the under-development codes
  - tags – contain relevant static copies, snapshots of the trunk (before branch reintegrations etc)
  - sandbox – area for development that may diverge from trunk
  - released – area for tags pointing to release versions

Best practices

Common SVN Server Practice

- Branch philosophy
  - Active development areas
  - Should remain in sync with the trunk
    - Regular sync
    - Sync after new trunk is announced
  - Should only be created by admins
  - Will be full copies of the trunk
  - Contain preferably only one development item (ie, ST4, BT4, nonlinear interactions, multigrid, esmf, nrltest etc)
    - Branch names will reflect such “function”
    - If changes require new switch: request to admins
  - Recommend: local copies should be full branch copies
    - Use the install_ww3_svn_trunk script

Best practices

Stable version complete: back to trunk

- Before reintegration
  - Branch is fully tested
  - Branch is in sync with the latest trunk
- Communicate to admins that a branch reintegration is needed/imminent with some notice (at least a few days) Branch reintegration can only be done by svn admins
- After reintegration
- Any conflicts should be solved by the admins
  - But these will not exist since you will have tested, right?
  - The clean new trunk on a local admin copy will be regression tested
  - All good, the new local trunk is committed (we go out for beers and one ginger ale)
- All branches are synced with the new trunk

Best practices

Common SVN Server Practice

- Bugfixes
  - Special case of changes to code
  - Developers should communicate bugs that are common to other parts of the code
  - A Ticket will be created
  - A branch will be created for fixing that bug
  - The branch will be reintegrated to trunk
  - The bugfix will be propagated to all branches via regular synchronizations

Best practices

Programming style:

- Use WAVEWATCH III documentation style (see templates).
- Developers, responsible for integration in the data structure: do only after rigorous testing of self-contained system.
- Use coding style:
  - Free format but layout as in old fixed format.
  - Upper case for permanent code, lower case for temporarily code.
- Maintain update log at header for documentation.
- Embed all subroutines in modules or main programs, using naming convention outlined before.
- Follow FORTRAN 90 standard, with best practices as outlined in section 2 of the guide.

Best practices

Manual and documentation.

- Provide full LaTeX documentation for inclusion in the manual:
  - NCEP svn users have access to manual, and are expected to add to it directly.
    - NCEP will provide editing.
  - Others provide separate files.
    - NCEP will integrate.
  - Use BibTEX.
  - Use dynamic references to equations, figures and tables only.

Best practices

Testing

- Regression testing are based on previous WAVEWATCH III tests and new materials and tools provided by Erick Rogers and Tim Campbell from NRL Stennis.
  - nrltest, will replace current test directory in near future

The end

End of lecture