

“push button, send product”

- Configuration, Version, and Release Management

Sommerville, Chapter 29

Instructor: Peter Baumann

email: p.baumann@jacobs-university.de

tel: -3178

office: room 88, Research 1

*"Good, Fast, Cheap:
Pick any two (you can't have all three)."*
-- from RFC 1925

Symptoms of the Change Chaos

■ Problems of **identification and tracking**:

- “This program worked yesterday. What happened?”
- “I fixed this error last week. Why is it back?”
- “Where are all my changes from last week?”
- “This seems like an obvious fix. Has it been tried before?”
- “Who is responsible for this change?”

■ Problems of **version selection**:

- “Has everything been compiled? Tested?”
- “How do I configure for test, with my updates and no others?”
- “How do I exclude this incomplete/faulty change?”
- “I can’t reproduce the error in this copy!”
- “Exactly which fixes went into this configuration?”
- “Oh shoot. I need to merge 250 files!”

More Chaos

- Software **delivery problems**:
 - “Which configuration does this customer have?”
 - “Did we deliver a consistent configuration?”
 - “Did the customer modify the code?”
 - “The customer skipped the previous two releases. What happens if we send him the new one?”

*...familiar to anyone
who has worked in software development.*

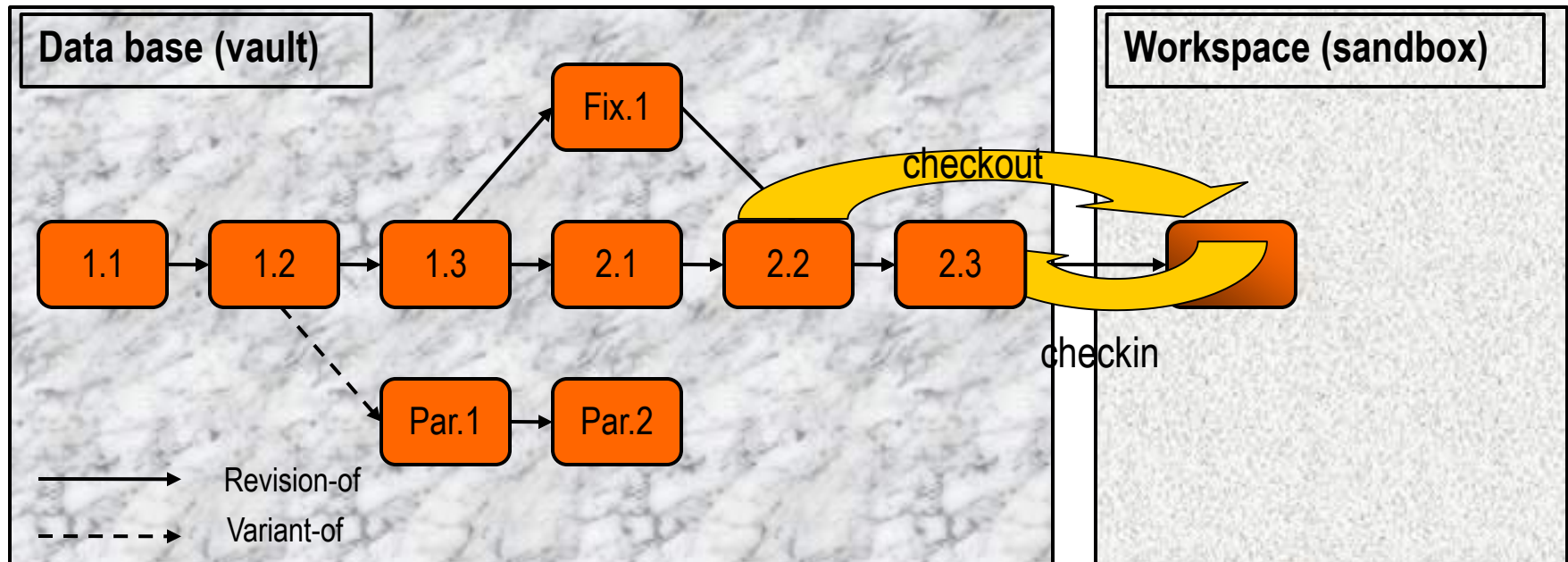
Definition

- **Software Configuration Management (SCM)**
= the discipline of controlling the evolution of software systems
- SCM indispensable for large, long-lived software!

Three Classic CM Problems

- Configuration management must address at least the following key problems:
 - **The double maintenance problem**
Prevent the occurrence of multiple copies of the same file that must be independently updated.
 - **The shared data problem**
Allow two or more developers with access to the same file/data.
 - **The simultaneous update problem**
Prevent file clobbering from two developers updating the same file at the same time.

Versions



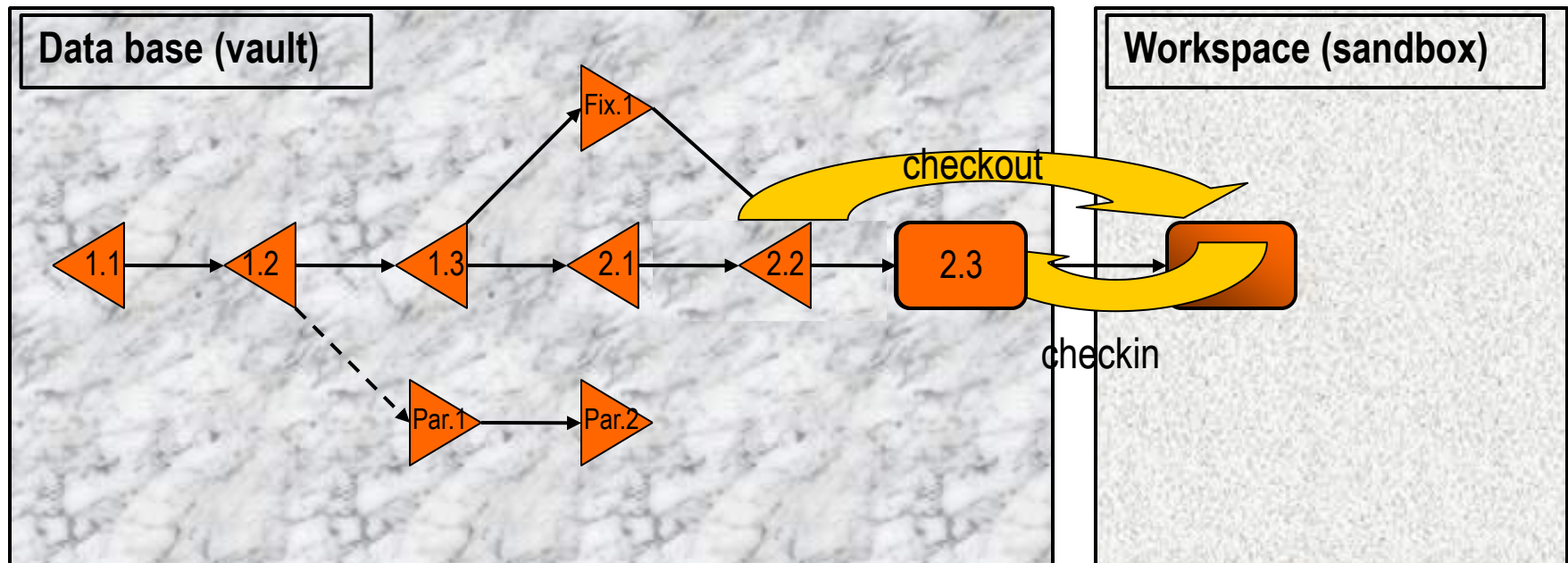
Change cycle:



1. checkout
2. modify copy in workspace
3. checkin

Versions: Terminology

- **Version** ::= revision | variant
- **Revision:**
A software object that was created by modifying an existing one
- **Variant:**
Two software objects sharing an important property, differing in others
 - user interface, platform, algorithms, data structures, user groups, ...

Using Deltas to Reduce Version Space

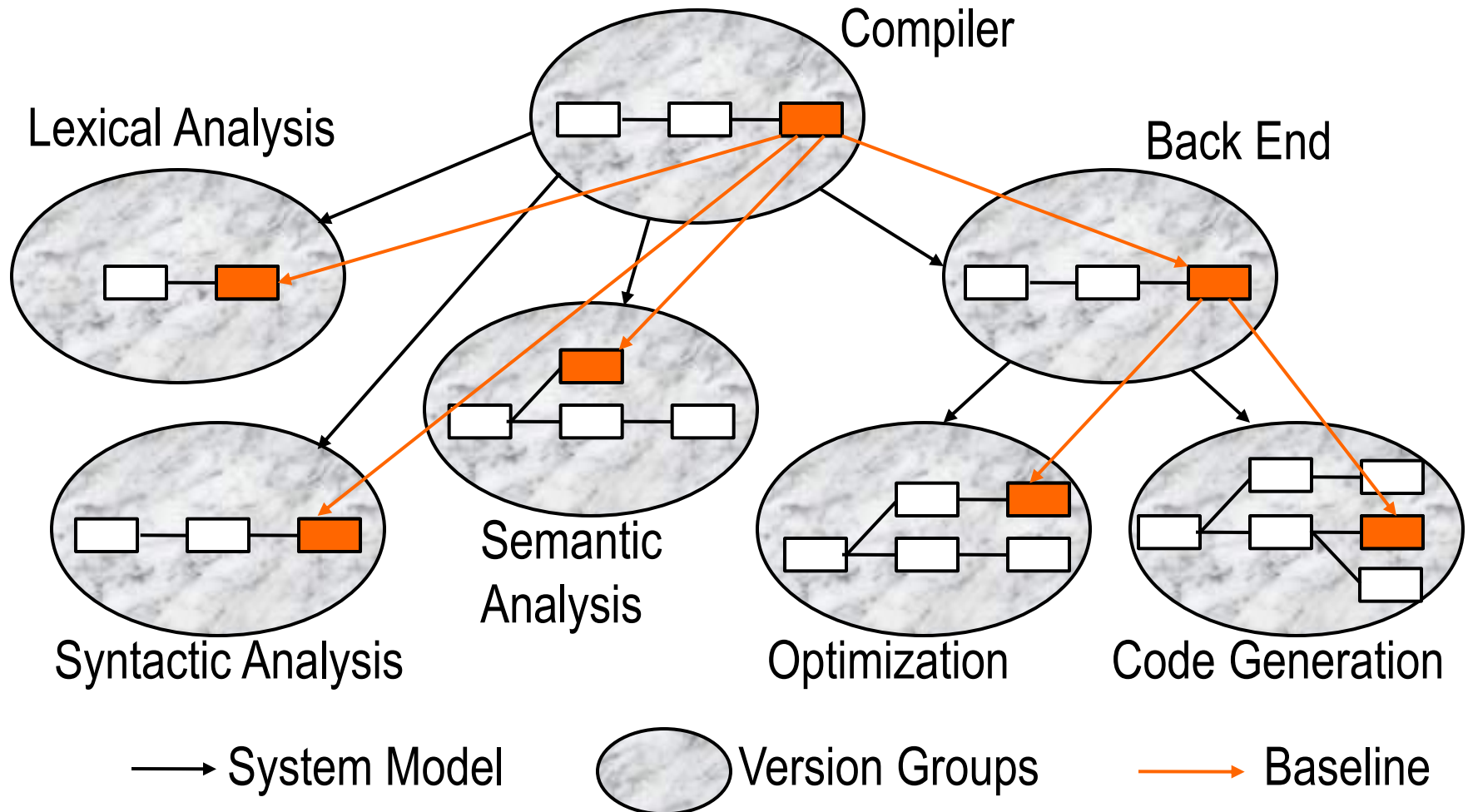


A delta is a difference between two revisions. Can be computed in forward  or backward  direction on checkin. Used to regenerate one revision from another.

Merging of Versions

- line based produces too many false conflicts
- Full semantic merging not practical
- Structural merges (smart merges)
 - Ex: developer A changes identifiers, developer B changes code using the old identifiers. Now what?
 - research stage

Version Selection For Configuration



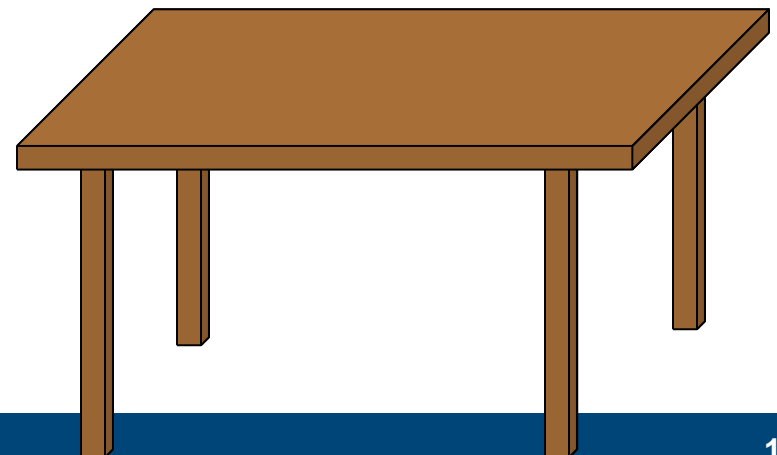
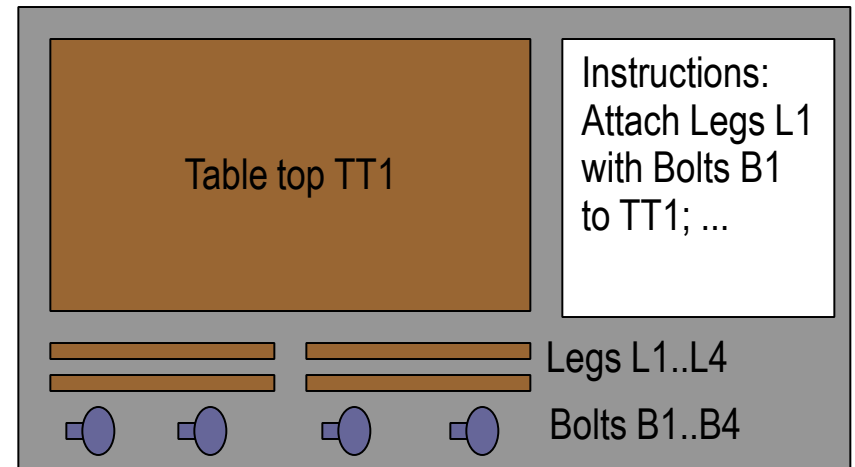
Configuration: Terminology

- **Release:**
a version that has been made **available** to user / client
- **Configuration:**
combination of components into a system according to case-**specific** criteria
- **Baseline:**
a static **reference point** for any configurable items in your project
 - does not imply that baseline is "perfect". It is only stating, in an example of software development, that the current version of the code is locked down so it can be verified and validated (tested)
 - Beware: used with various connotations; see <http://projectmanagement.ittoolbox.com/documents/popular-q-and-a/meaning-of-baseline-in-the-software-project-1480> for a discussion

- Three major models:
- **Composition Model**
 - Configuration = set of **software objects** *-- module oriented aspect*
- **Change Set Model**
 - Configuration = bundle of **changes** *-- dynamic aspect*
- **Long Transaction Model**
 - Configuration = all changes are isolated into transactions *-- collaborative aspect*
- *Common understanding:*
configuration = selection (according to some individual criteria) of components from repository which together make up a release

Ex: Furniture Configuration

- Available product parts are combined into the variety of products offered
- Sample configuration:
 - Configuration name
"Table MySweetHome"
 - Enumeration of elements,
plus their pertaining versions
(components, user documentation, ...)
 - Process description:
how to generate & package
 - Responsible employee
 - ...



Configurations in rasdaman

- Operating systems:
 - Linux, Solaris, HP-UX, DEC Tru64, AIX, ...
- Compilers
 - (gcc 2.95) / gcc 3.3 / gcc 4 / ...
- Database systems:
 - Oracle, DB2, Informix, PostgreSQL, MySQL
- With or without debug (internal test version)
- Release timeline:
..., 3.6, 5.0, 5.1pre, 5.1, 5.2, 6.0, 6.1; 7.0; 8.0, 8.1, 8.5, 9.0

Rule-Based Configuration Building

(sample roles)

[versions | configurations]

1. Baseline (aka "default"):

no versions, no selection needed

2. Developer:

select all version checked out by me,
+ the newest revision on main branch
for others.

3. Cautious Developer:

select last baseline
+ all version checked out by me
+ all versions checked in by me

4. Reconfiguration:

select according to ...
+ variants by attribute (e.g. platform=Unix)

5. Time machine:

select according to ...
+ ignore everything after a cutoff date.

6. New Release:

select according to ...
+ the newest, stable versions that are
associated with a given configuration of
change requests

Tools: svn & friends

- SVN (subversion) ← CVS ← RCS [Tichy, 80s] ← SCCS
 - GUI frontends: RapidSVN, TortoiseSVN
- Some svn commands:
 - `svn import http://yourhost/svn/repos myproject myproject`
 - *Don't forget the last argument, or else all of the files in myproject will be dumped into the main SVN repository directory... at the top of the hierarchy*
 - `svn checkout http://yourhost/svn/repos/myproject/trunk -d myproject`
 - `svn status myfile.cc`
 - `svn update myfile.cc`
 - `svn diff myfile.cc`
 - `svn commit myfile.cc`
- diff, tkdiff

Tools: make

- "make": controls generation of derivatives from their prerequisites
- Configuration rule base: (i) builtin, (ii) Makefile, makefile
- Rule format:
 - **target: prerequisites**
 - **how-to-generate**
- Macro: aka variable
 - `SRC = myclass1.cc myclass2.cc`
 - `gcc -o $(OBJECTS) ...`

`$$` Full name of current target

`$$?` current dependencies out-of-date

`$$<` source file of current dependency

Multi-Step Generation

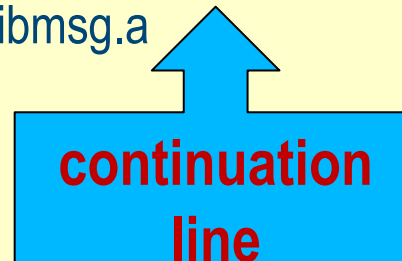
- Makefile can contain any number of rules
- Advantage: update only when needed

```
server : server.o socket.o database.o /home/xyz1234/Messages/libmsg.a
        cc -o server server.o socket.o database.o \
        /home/xyz1234/Messages/libmsg.a

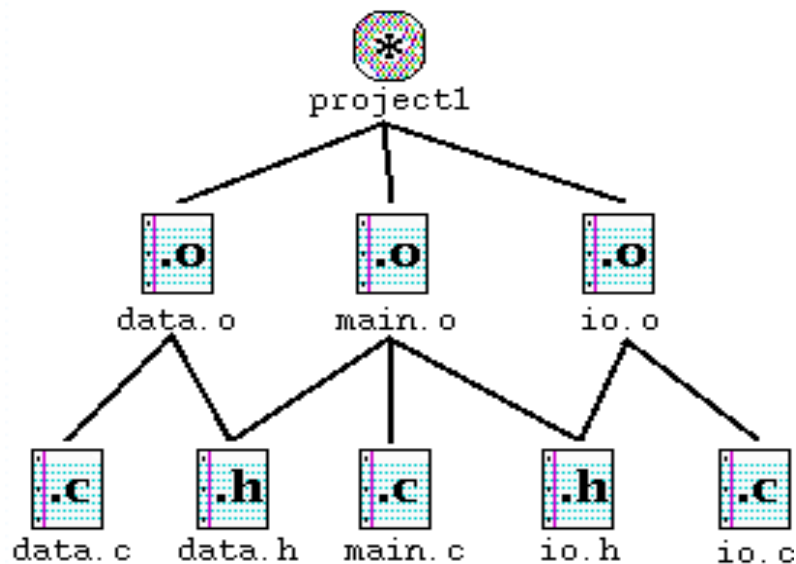
server.o : server.c
        cc -c server.c

socket.o : socket.c
        cc -c socket.c

database.o : database.c
        cc -c database.c
```



How Make Sees Your Project



Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

- Configuration Management
for controlling change through life cycle
 - Version/revision/variant, release/configuration
 - svn, git
 - diff
- Rule-based code generation
 - make
 - also used for configuration