Is /main/LATEST too dynamic?

Software Integration Strategies Using ClearCase

What is Software Integration?

- The act of producing a buildable and functionally consistent set of software from changes which have occurred in parallel.
- Software integration occurs at many levels.
- The best approach is dependent on your organization and your development processes.

Levels of Software Integration

- Integrating multiple developer's changes for release 1.0 of a product
- Integrating bug fixes made to release 1.1 of a product with new development made for release 2.0 of a product
- Integrating concurrently developed features A, B, and C into release 3.0 of a product
- Integrating concurrently developed subsystems into a new release of a product.

Parallel Development

- This talk focuses on a team of software engineers working on a common task.
- The scope of this task could be a major project release, a minor project release, a set of bug fixes, an experimental version, etc.
- How and how often do engineers integrate with each others changes?

Integrate now or later, that is the question?

- Early integration helps point out design deficiencies and reduces the amount of redesign and code rework.
- Each organization must try to strike a balance between early integration and the need for isolation.
- Integrate as early and regularly as possible.

Integration Strategy Requirements

- Should maintain dynamic behavior (e.g. the LATEST label)
- User should be able to chose the settings once and the updates occur automatically.
- Add to system complexity only when required

/main/LATEST

- First level of isolation (isolation between views)
- No additional tools or procedures required
- On-going integration of changes at checkin time
- clearmake does not include check-ins which occur during a build

/main/LATEST

- Typically a team of one to eight engineers can work with a /main/LATEST rule. Maybe more if...
- Communication lines between team members are good
- Members are physically located close together
- Care is taken to ensure compilability and consistency when checking in changes.

Top four signs that /main/LATEST is too dynamic

- You can’t get a clean build because there is always some file that does not compile.
- You can’t get a clean build because someone always forgets to checkin a header file change along with the source file change.
- You have difficulty debugging because you can’t tell whether the problem is in your code or someone elses.
- You reminisce about the “good-ole-days” of SCCS.

time and -time

- Second level of isolation
- Minimal tooling required
- Checkin times still used to determine visibility
- Engineer must still take care to ensure a buildable consistent change at checkin time.

Simple variations on time

- time now
- time 9:00
- time Friday
- time 10-Jul-1995.19:00
- All (time or -time) statements require a ‘cleartool setcs -current’ to update.

User change visibility after checkin

```
time 12-Oct-1995.1900
element * CHECKEDOUT
element * /main/{created_by(myuid) &&
                  created_since(12-Oct-1995.1900)}
element * /main/LATEST
```

Note: 3rd rule must be contained on one line

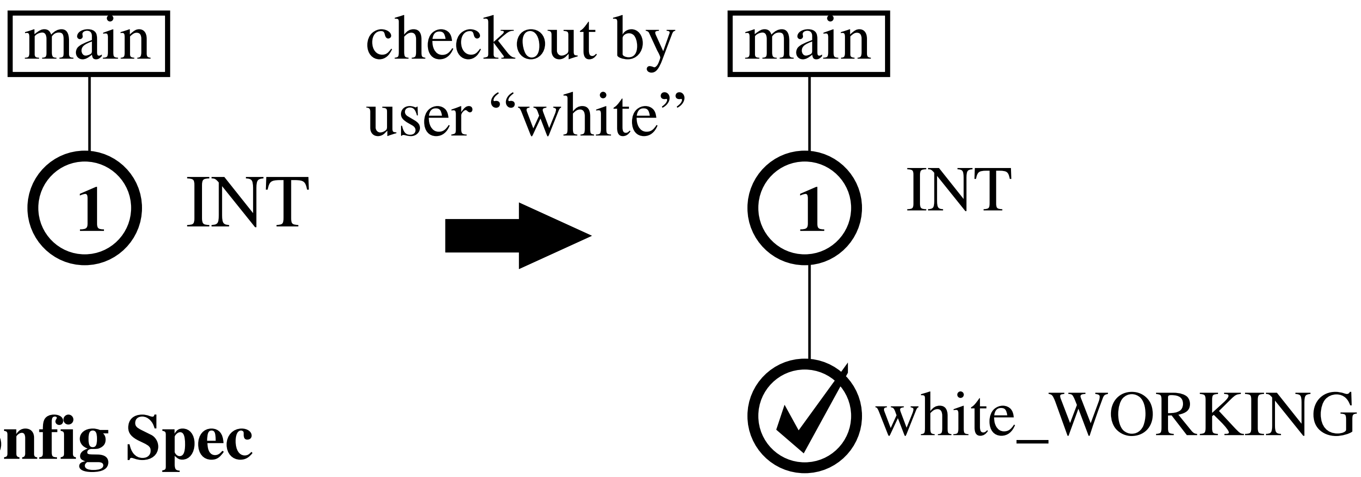
Merge avoidance using LABELS

- Allows for controlled increase in visibility of changes
- Does not require branching or merging
- Requires an integrator role to test and promote (make visible) engineering changes
- ClearCase process facilities are required to implement this approach
- Typically needed once build problems become frequent

Manually LATEST label INTEGRATED

- A label is used to identify the "approved" latest version vs. the /main/LATEST version.
- Two labels per engineer are used to identify their "latest" version
- Label manipulation is automated via triggers
- Labels are of the "one per branch" type.

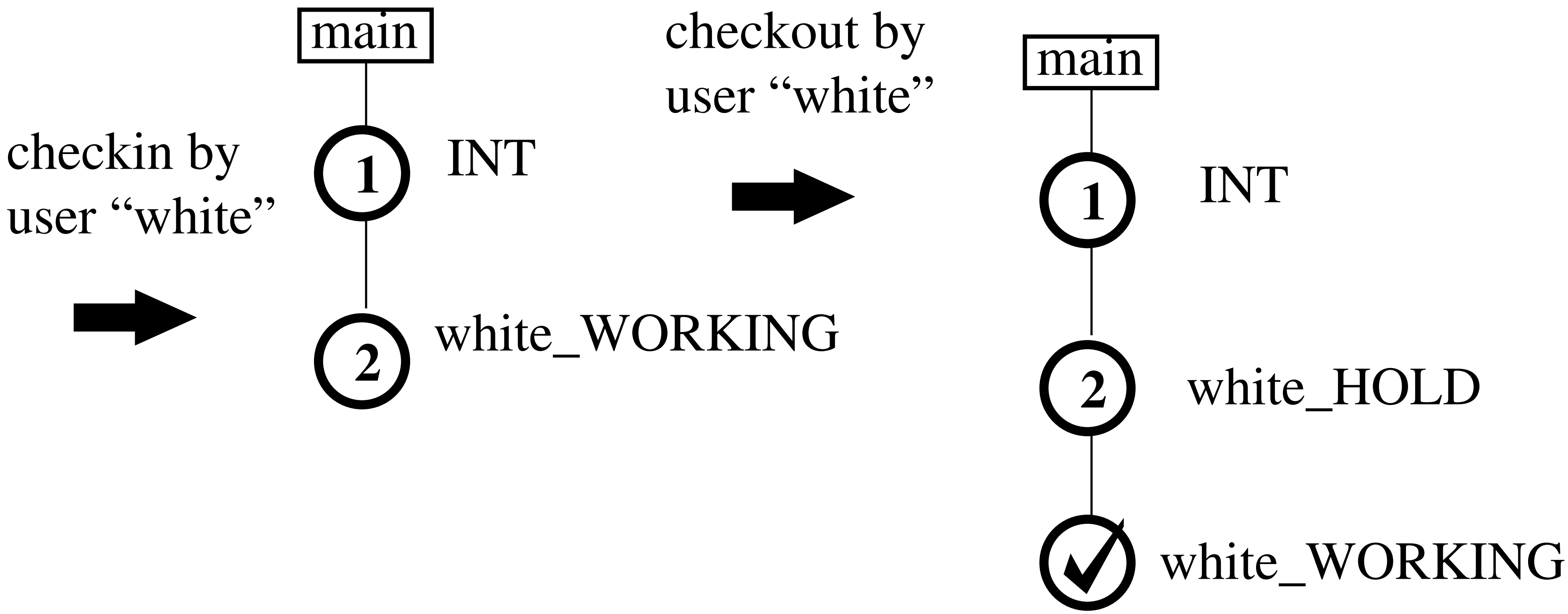
An Example - Checkout



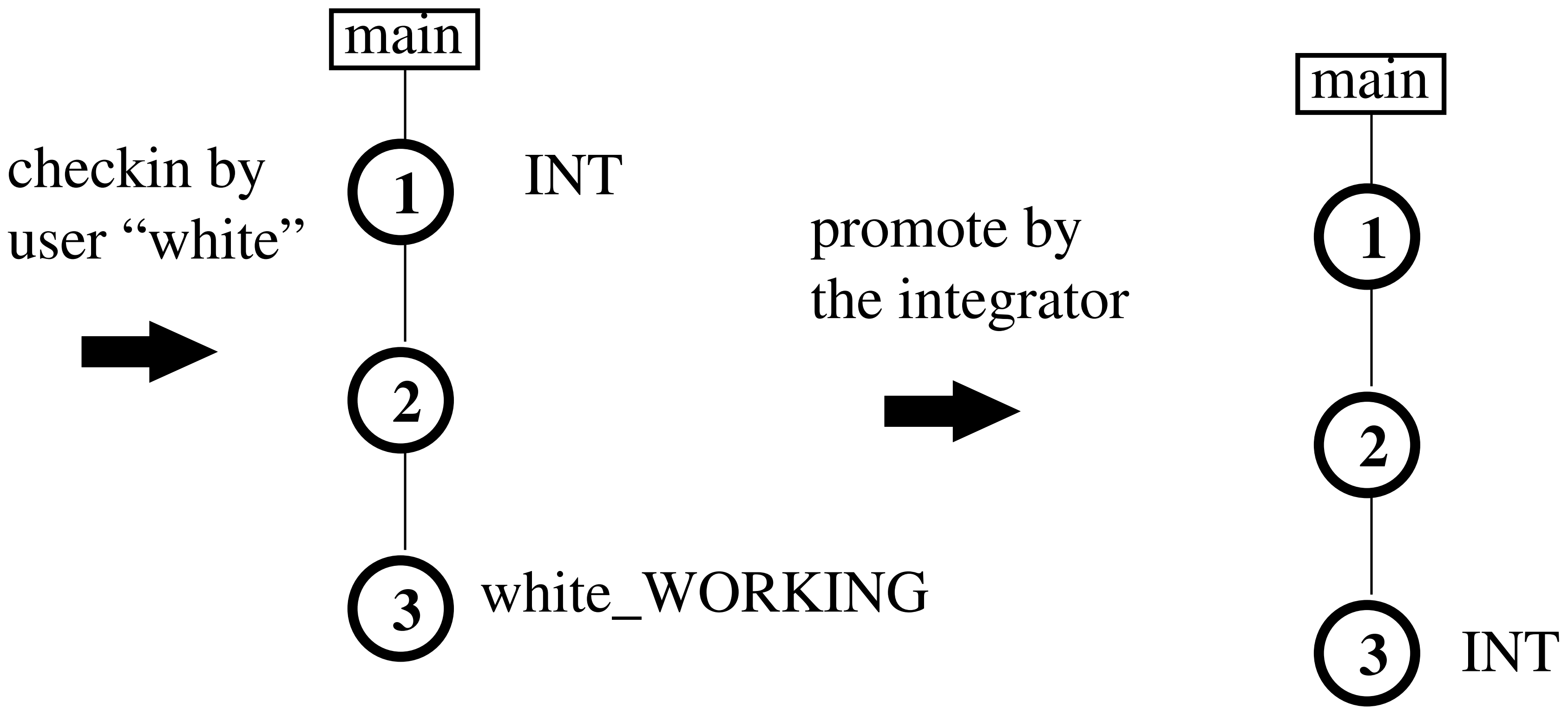
Development Config Spec

element * CHECKEDOUT
element * /main/white_WORKING
element * /main/white_HOLD
element * /main/INT
element * /main/LATEST

Checkin and Checkout Again

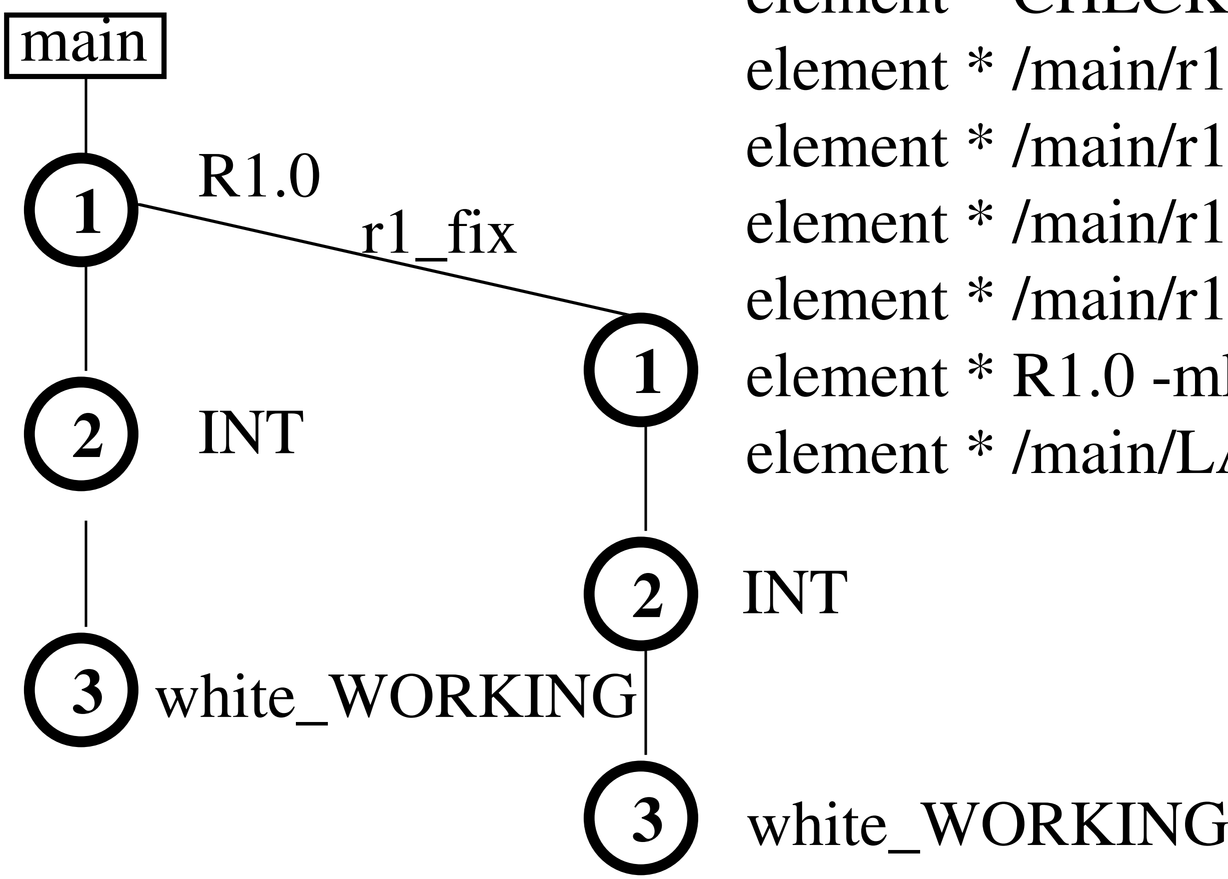


Promotion


main
checkin by
user “white”
1
INT
2
3
white_WORKING
promote by
the integrator
main
1
2
3
INT

Parallel Development using a label based approach

```
element * CHECKOUT
element * /main/r1_fix/white_WORKING
element * /main/r1_fix/white_HOLD
element * /main/r1_fix/INT
element * /main/r1_fix/LATEST
element * R1.0 -mkbranch r1_fix
element * /main/LATEST -mkbranch r1_fix
```



The Setup Tools

- mkuserlbs - A preevent on co which creates the user labels
- install_triggers - A script which install all the triggers and creates the integration label.
- remove_triggers - A script which deinstalls the triggers
- rmuserlb - A script which removes all the user labels

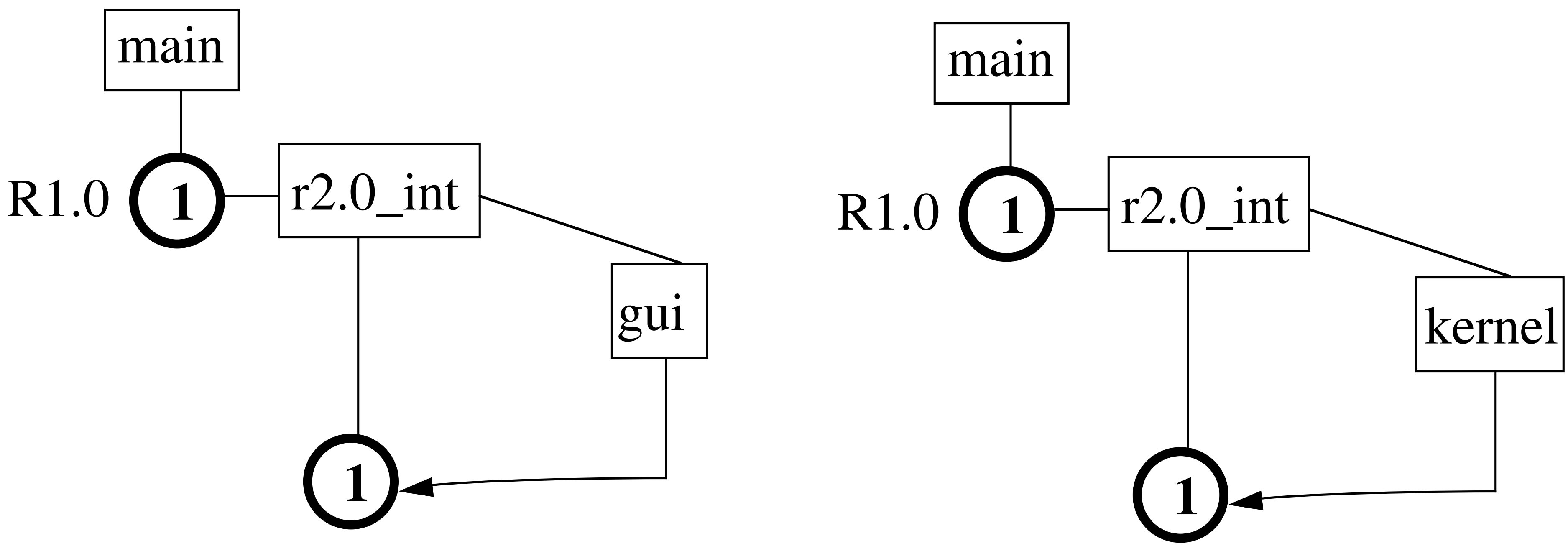
The Operational Tools

- mkworking - A post-event checkout trigger which attaches the working label to the checkedout version and attaches a hold label to the previous version if it had a working label prior to the checkout.
- rmholdci - A post-event checkin trigger which removes the hold label from the checkedin versions predecessor.
- rmhold - A post-event uncheckout trigger which replaces the hold label with working, effectively restoring the labels states prior to checkin.
- promote - Not a trigger at all, this tool is used by the integrator to promote the INTEGRATED label and remove any user labels on intervening versions.

Branching method

- Allows for high-level integration management using branches
- Chance of concurrent changes on a single branch reduced
- Integrator may need to resolve merge conflicts
- Management of branches key to success
- Typically needed when concurrent changes prior to integration become a problem

Subsystem Branching (unique file set)



Subsystem Development Configuration Specification

```
element * CHECKEDOUT
element * .../r2.0_int/kernel/LATEST
element * .../r2.0_int/LATEST -mkbranch kernel
element * REL1 -mkbranch r2.0_int
element * /main/LATEST -mkbranch r2.0_int
```

Subsystem Integration

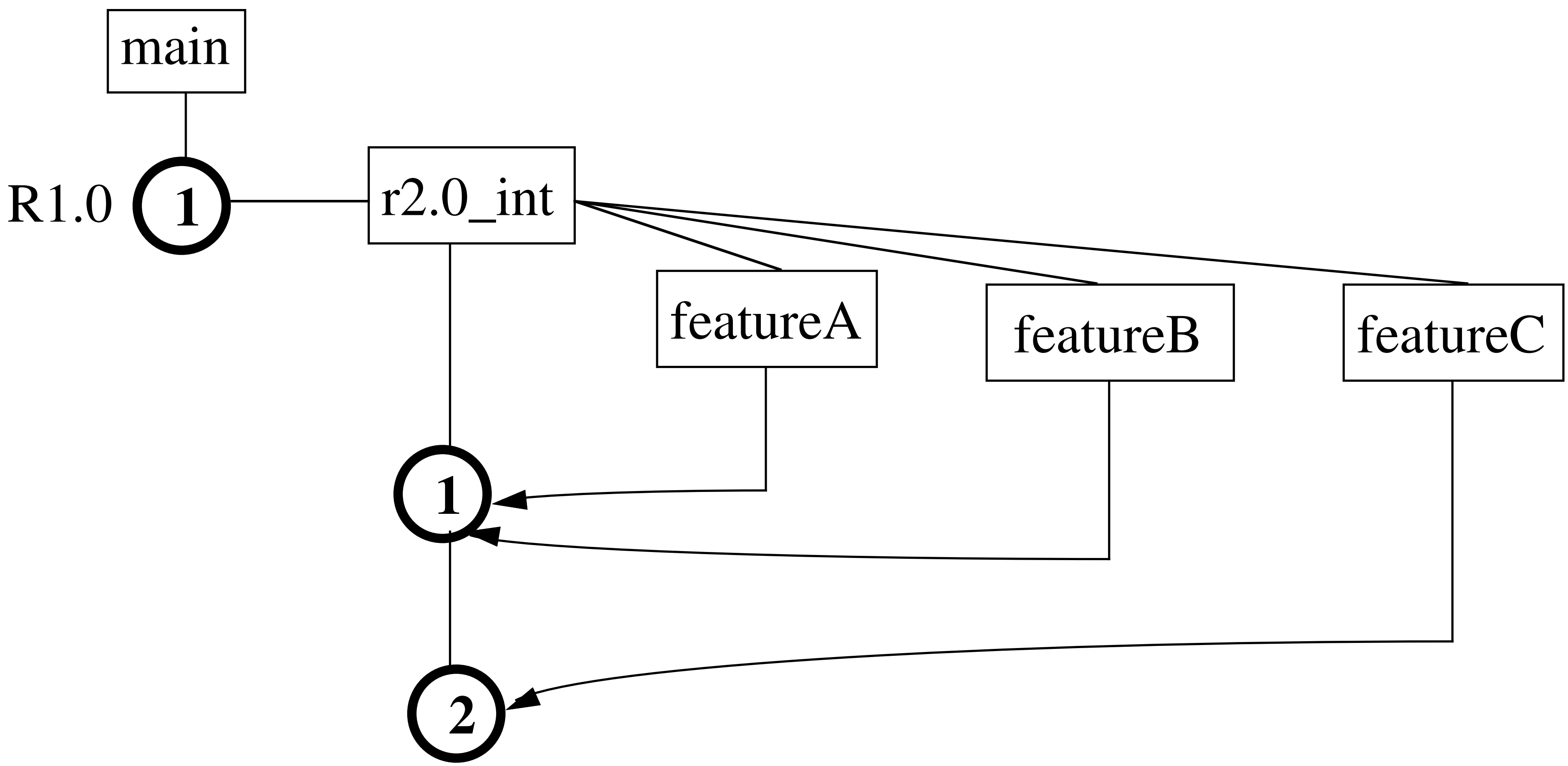
element * CHECKEDOUT
element * .../r2.0_int/LATEST
element * REL1 -nocheckout
element * /main/LATEST -nocheckout

ct findmerge . -type d -fver .../r2.0_int/kernel/LATEST -merge

ct findmerge . -fver .../r2.0_int/kernel/LATEST -merge -xmerge

Checkin signifies approval for use by other subsystems!

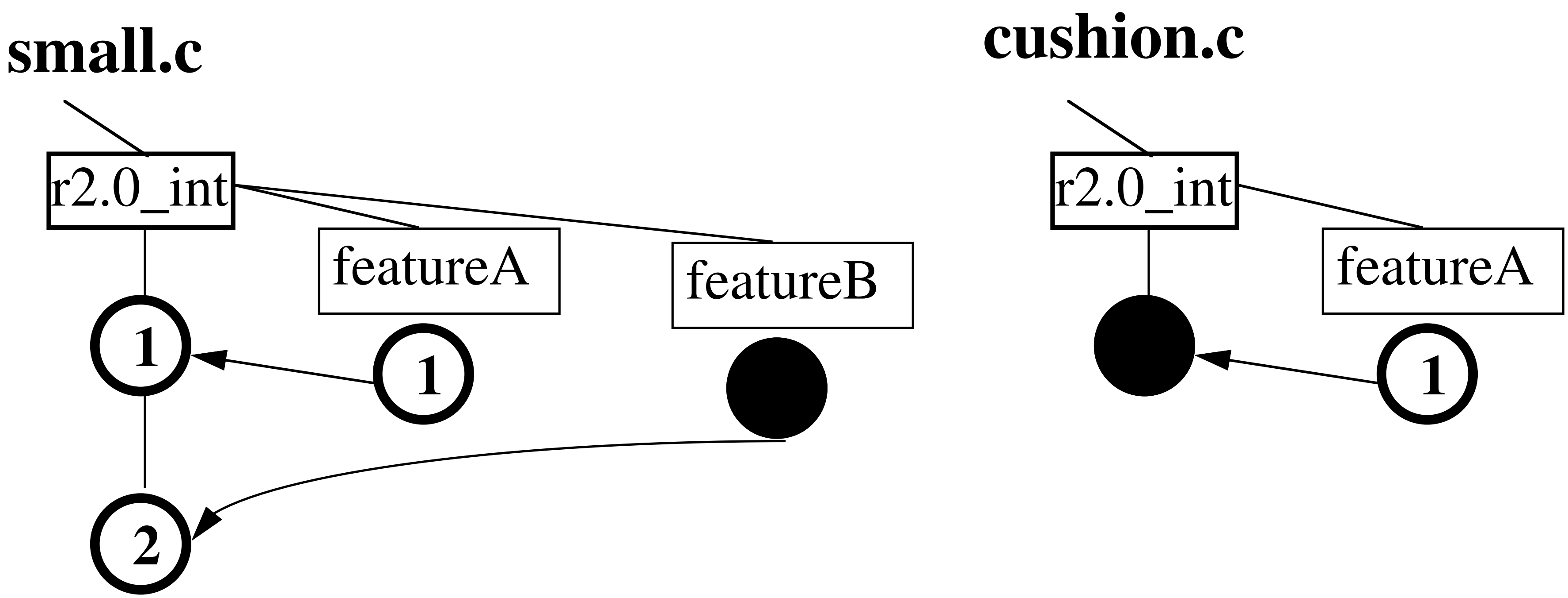
Feature Branching (common file set)



Feature Development Configuration Specification

```
element * CHECKEDOUT
element * .../r2.0_int/featureA/LATEST
element * .../r2.0_int/0 -mkbranch featureA
element * REL1 -mkbranch r2.0_int
element * /main/LATEST -mkbranch r2.0_int
```

Why not /main/r2.0_int/ LATEST?



Feature B development would select the dark versions, potentially dependent feature A changes!

Feature Integration

- Merging technically the same as previous model
- Testing of integrated changes can only occur on the r2.0_int branch. Feature branches do not “see” other branch changes.
- Feature development with this approach allows for easy migration of features to other releases.

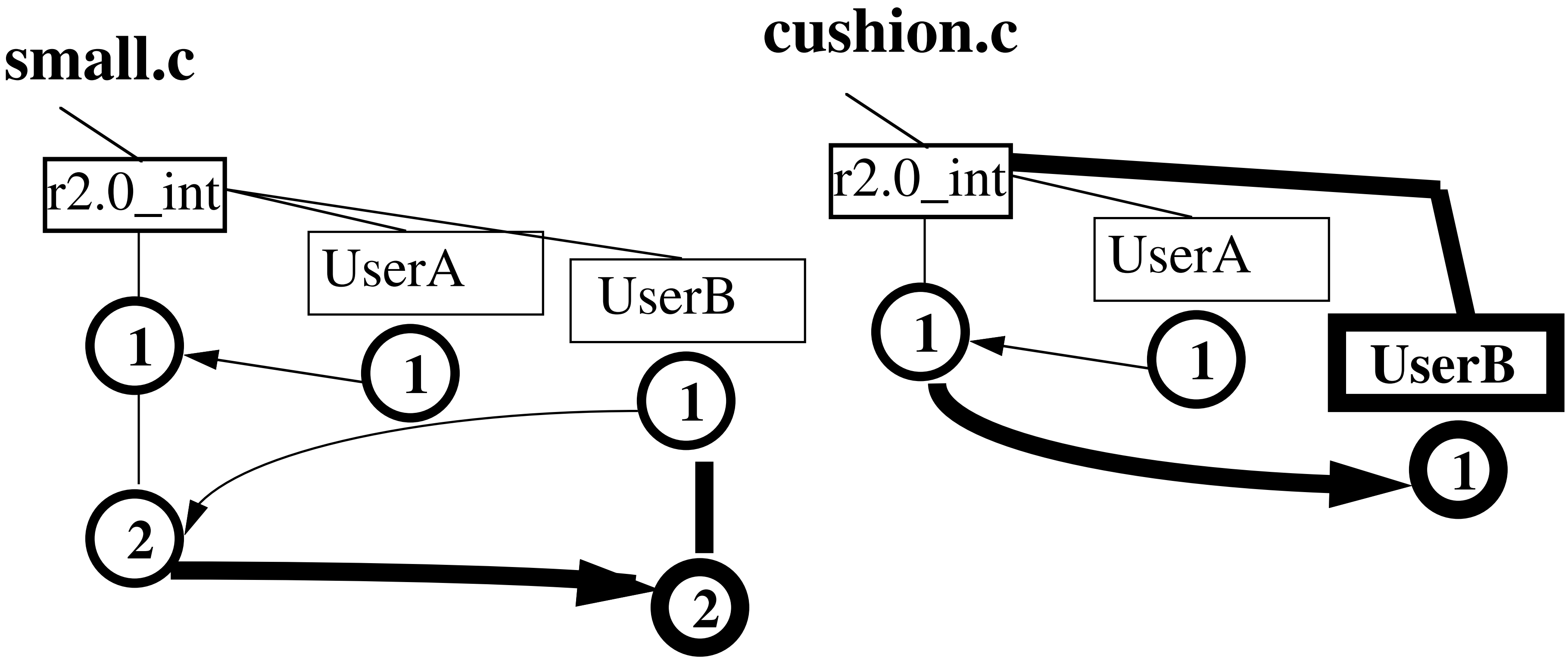
Change Sets

- Smaller granularity, but the same approach
- Branch management become critical to a successful implementation
- Change set dependencies must be tracked
- Integration branching scheme becomes more complex
- Building scheme must allow for inclusion/ exclusion of change sets

User branches Vs Task branches

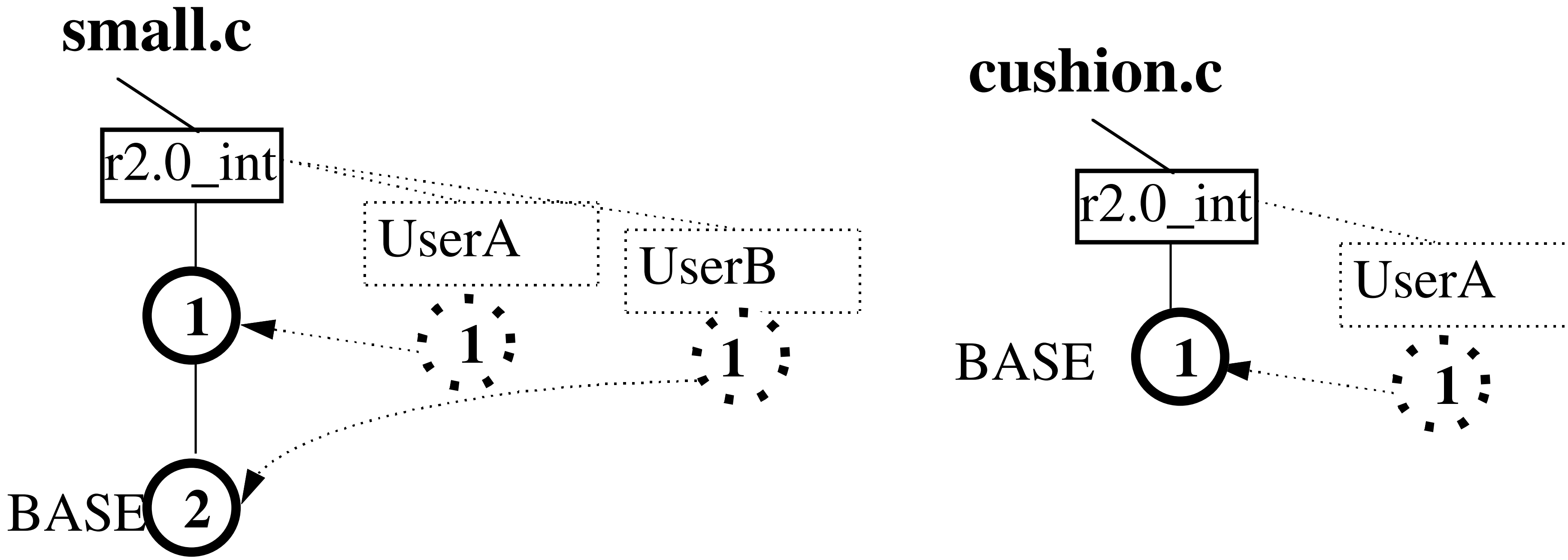
- In general, user based branches are not a good idea.
- Users should be isolated by views rather than by branch
- Synchronization problems common with copy based systems will reoccur.
- Does not meet the dynamic update requirement.

Synchronizing to establish a new baseline



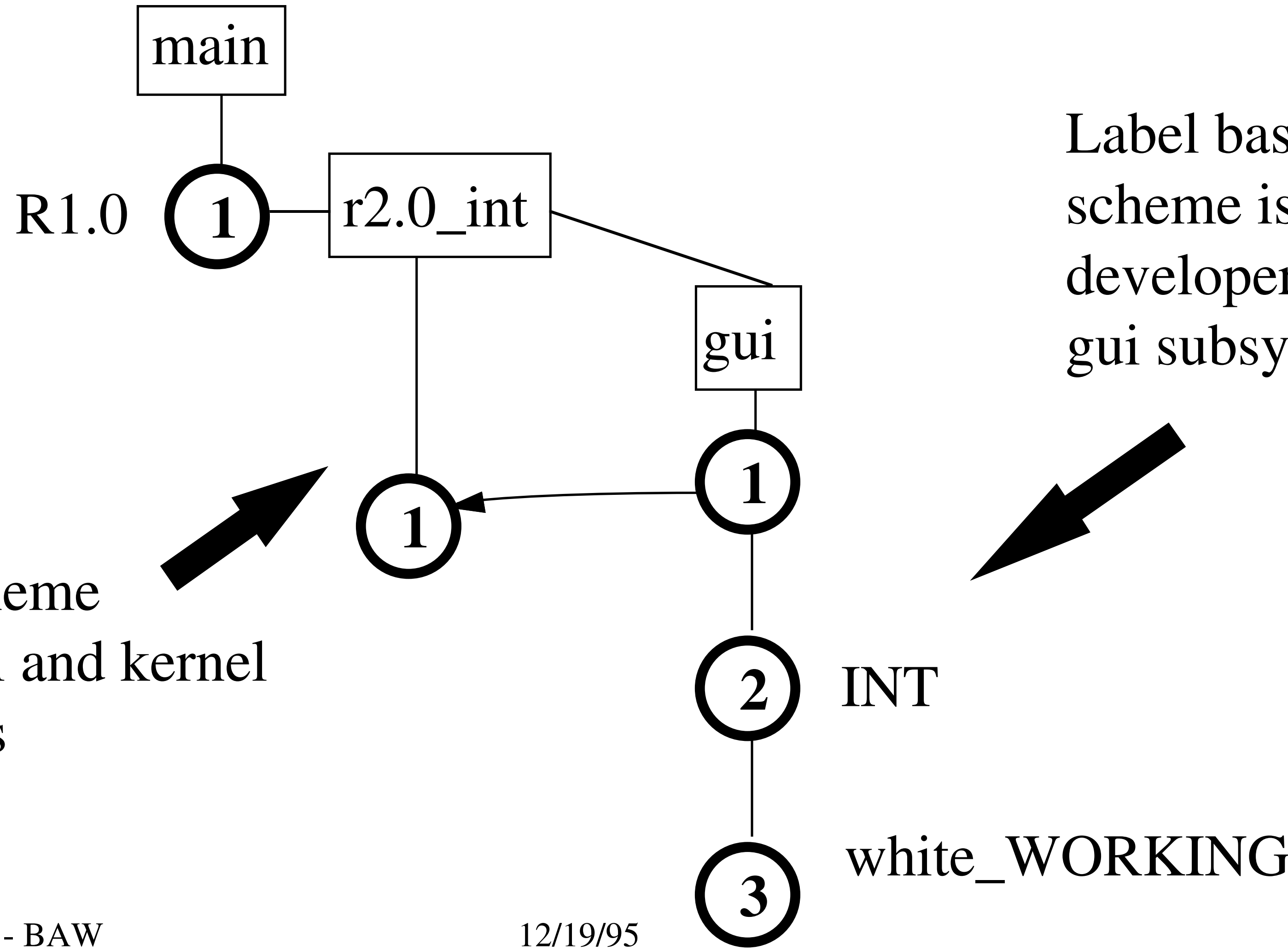
NOTE: UserA has not yet synchronized

Removing branches to establish a new baseline



Note: Need to add BASE label to avoid a LATEST rule.

Combinations



Label based scheme isolates developers on gui subsystem

Branch scheme isolates gui and kernel subsystems

Conclusions

- ClearCase allows for a wide variety of software integration strategies.
- Early integration is desirable to reducing rework.
- However, it is necessary to strike an appropriate balance between early integration and the need for isolation.
- Every organization and project is slightly different. Chose the approach carefully to match your integration needs.