Continuous integration
and
Minimization of dependencies

Tijs van der Storm

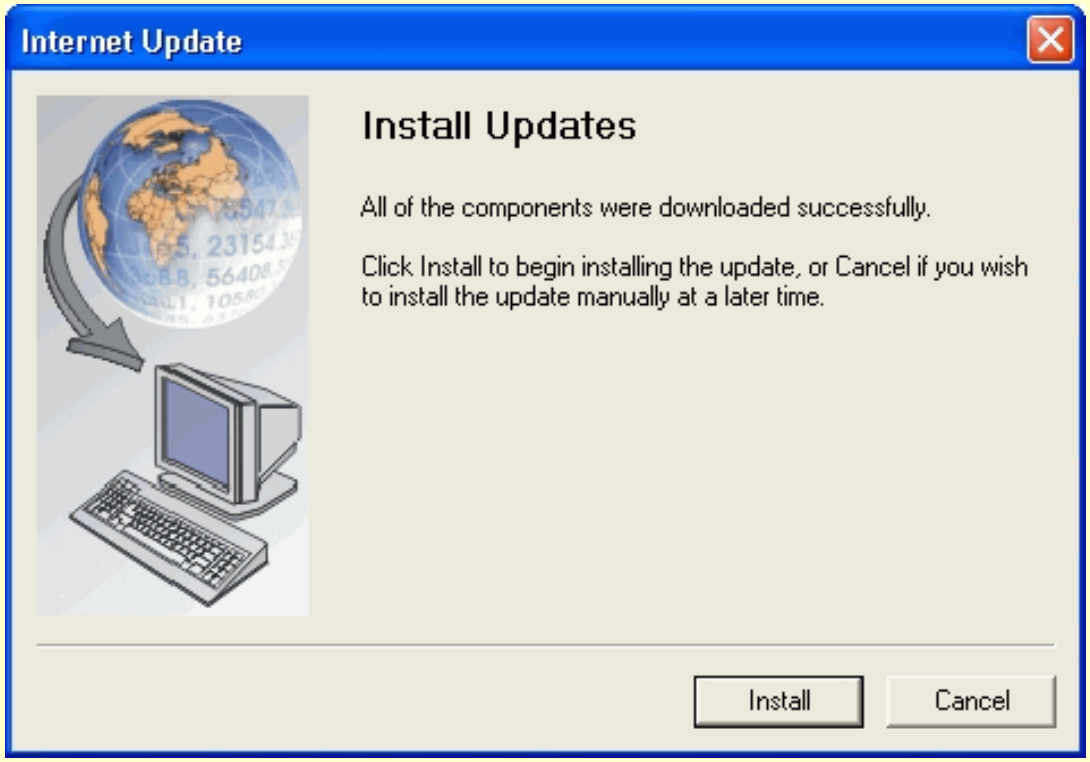
Centrum voor Wiskunde en Informatica

Overview

- Continuous integration
  - Background
  - The Sisyphus system
  - Benefits
- Minimizing compilation dependencies
  - Build architecture of C programs
  - Automatic dependency reduction
  - Qualitative evaluation

Continuous integration

A *Deliver* “Icon”



Updating component-based systems

- Example: ASF+SDF Meta Environment
- Have to maintain knowledge
  - which versions of components “work” together
  - which versions of components the user has
- Lowest level of “it works”: “it builds”
- Continuous integration ⇒ continuous release

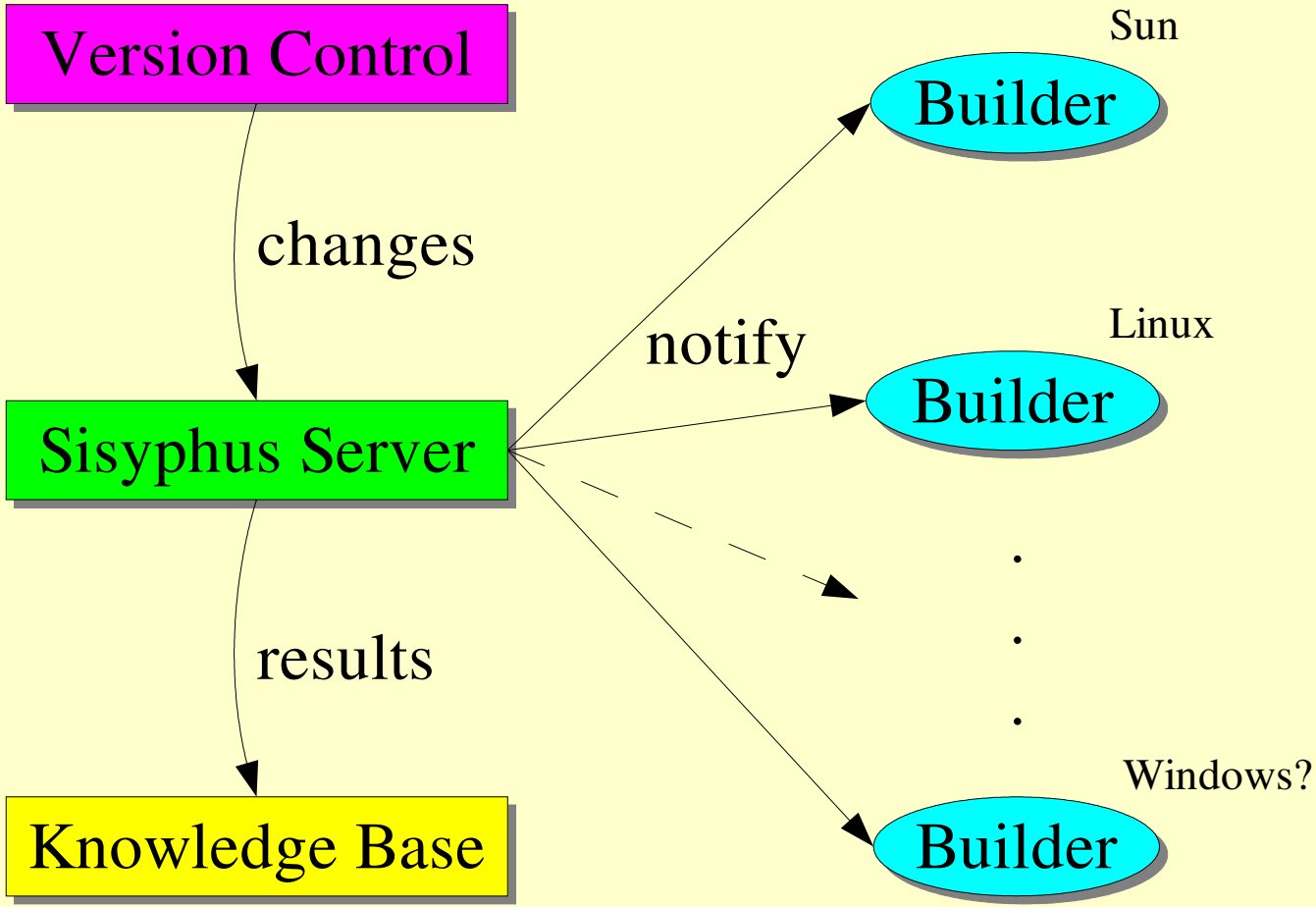
Continuous Integration (CI)

- "Extreme" version of daily builds
  - Frequent checkins
  - Frequent builds
  - Testing (unit, integration, acceptance, ...)
- Key features
  - Unified build script for automation
  - Centralized source location (version control)

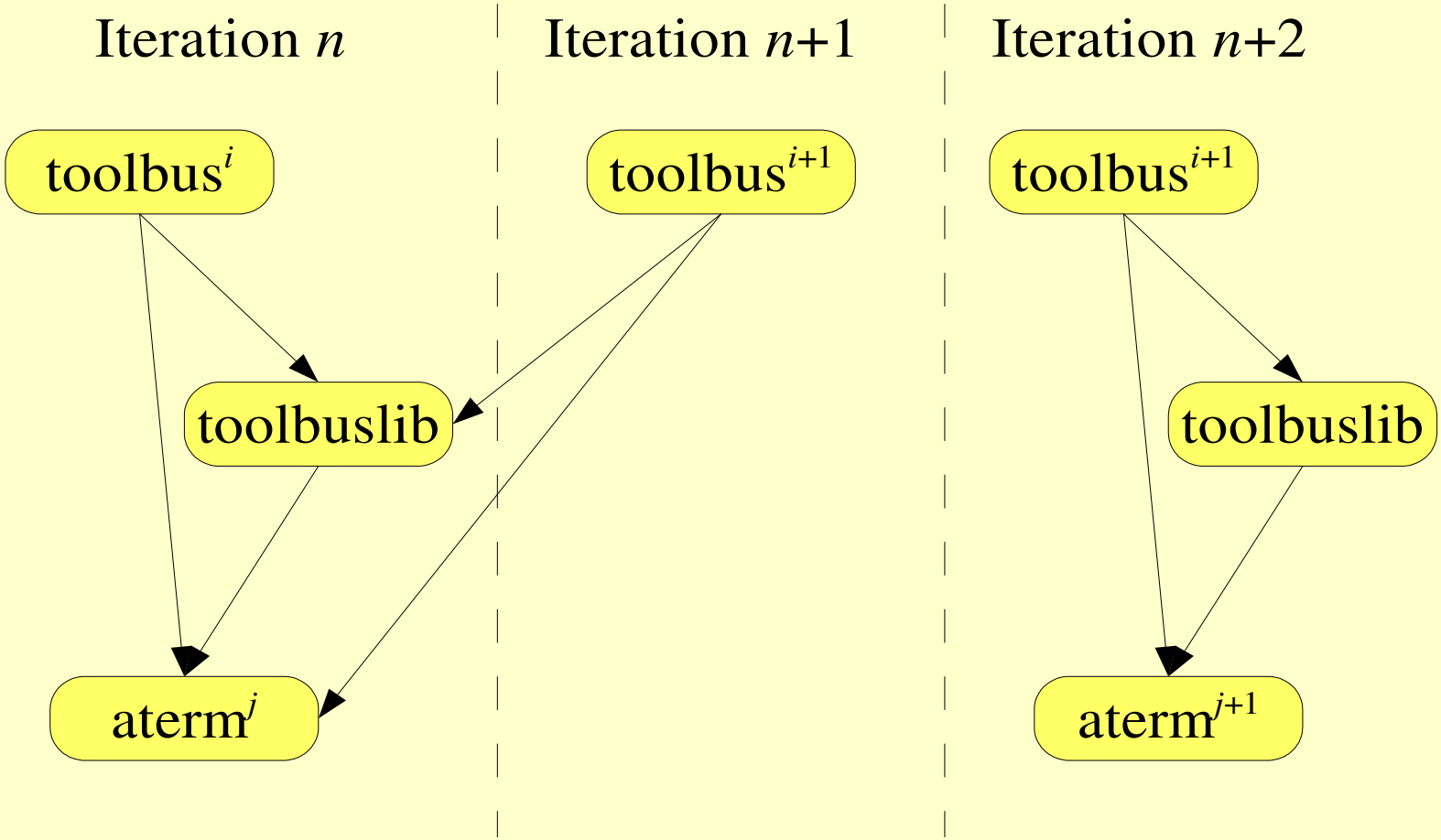
Sisyphus

- Component-based CI
- Language & platform agnostic
- High configurability
- Accurate version tracking
- Update generation
- Work in progress

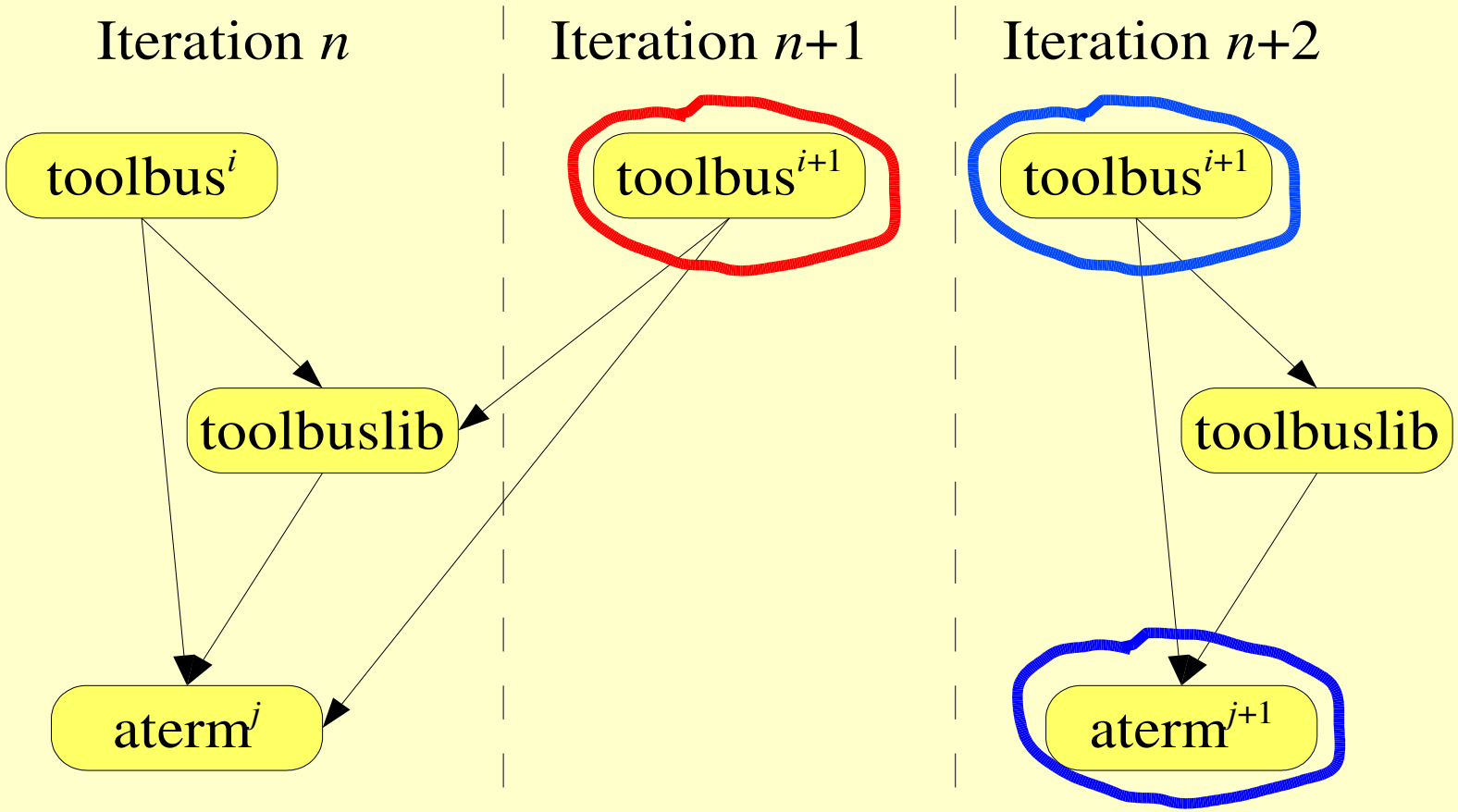
Sisyphus architecture



Selective recompilation



Flexible updates



Benefits of continuous integration

- Integration bugs detected early
  - "*the effort of integration is exponentially proportional to the amount of time between integrations*" [Fowler]
- Frequent releases
  - always a "working" executable available
- The more often the better
- Fast builds are a good thing

Minimizing dependencies

Architecture influences build process

- Currently: dependencies between packages
  - components are built from scratch
- Possible: dependencies between files
  - dictates single language, e.g., C
- Optimization opportunities
  - Reduce number of files read from disk
  - Increase incrementality

Some facts...

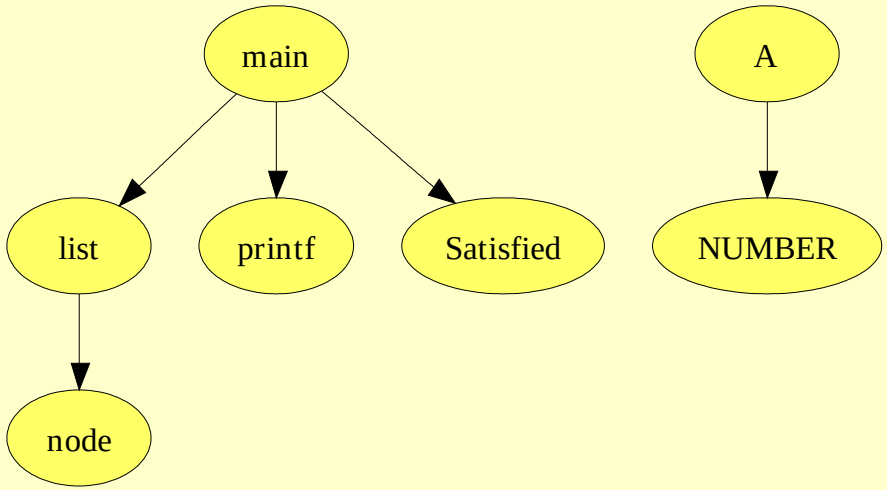
- Some facts about a large industrial C program:
  - 80 to 90% of unused but included program units
  - on average, each file includes 60% of headers multiple times
  - average size program file: 37Kb
  - average size preprocessed image: 1.96 Mb
- Courtesy of Yu et. al.

Precompilation

- Not my work:
  - "Removing false code dependencies to speedup software build processes" [Yu et. al. CASCON03]
- Build optimization in three steps:
  - extract fine-grained dependency graph
  - remove unused program entities from it
  - generate restructured headers
- Is *just-in-time* and *lazy*

Program unit dependency graph

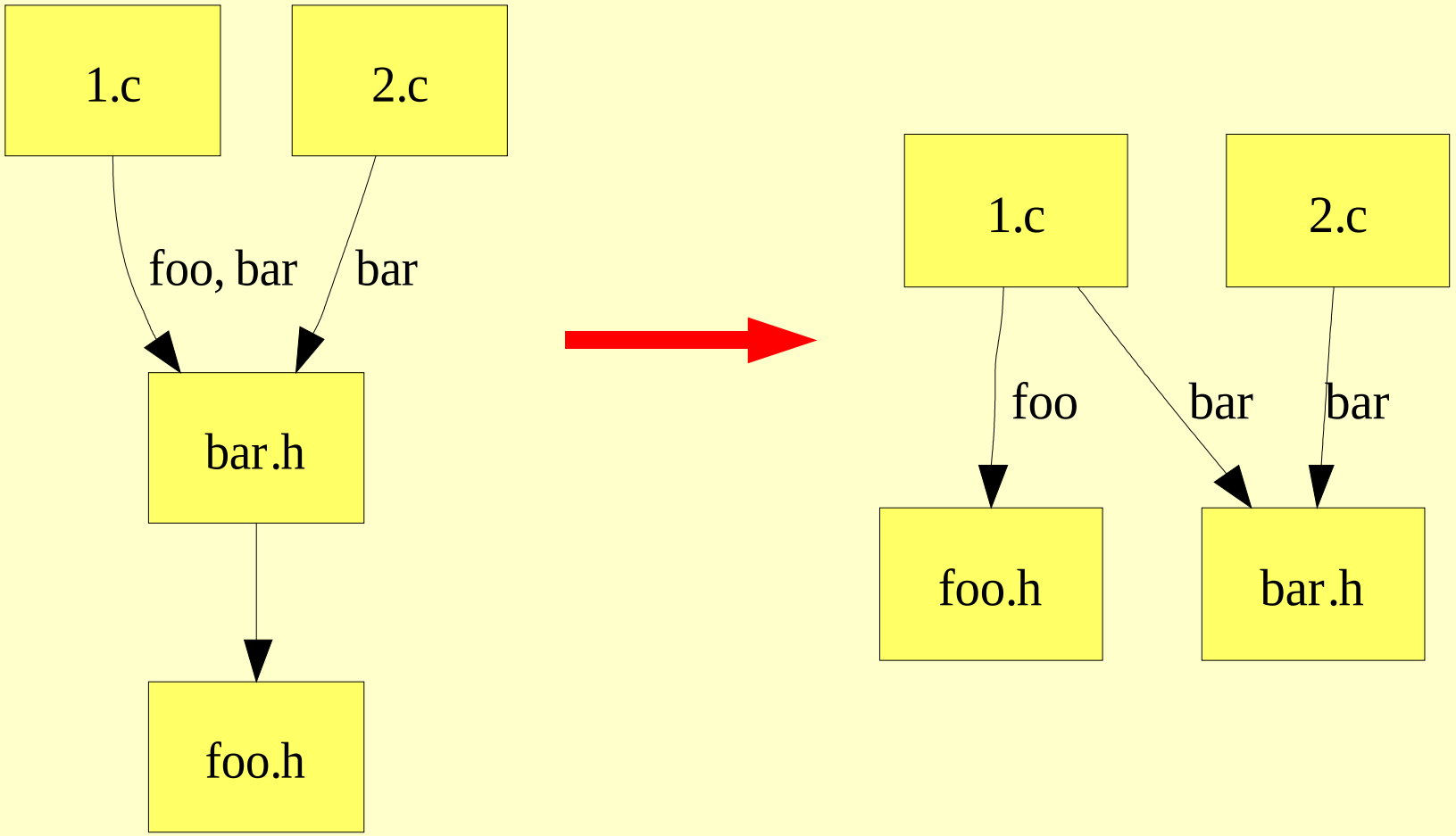
```
typedef int NUMBER;
struct node;
typedef struct node {
    float value;
    struct node* next;
} *list;
struct A {
    union {
        NUMBER value;
    } u;
};
extern int printf(char
*format,...);
enum { Satisfied, Denied };
int main(void) {
    list l, n;
    for (n = l; n; n = n->next)
        printf("%f", n->value);
    return (int)Satisfied;
}
```



Relational formalization

- Program units
  - $Unit = Decl \cup Def$
  - $RequirementsOf \subseteq Unit \times Decl$ (= PUDG)
  - $UsersOf = RequirementsOf^{-1}$
- Files
  - $File = Hfile \cup Cfile$
  - $UnitsOf \subseteq File \times Unit$
  - $FilesOf = UnitsOf^{-1}$

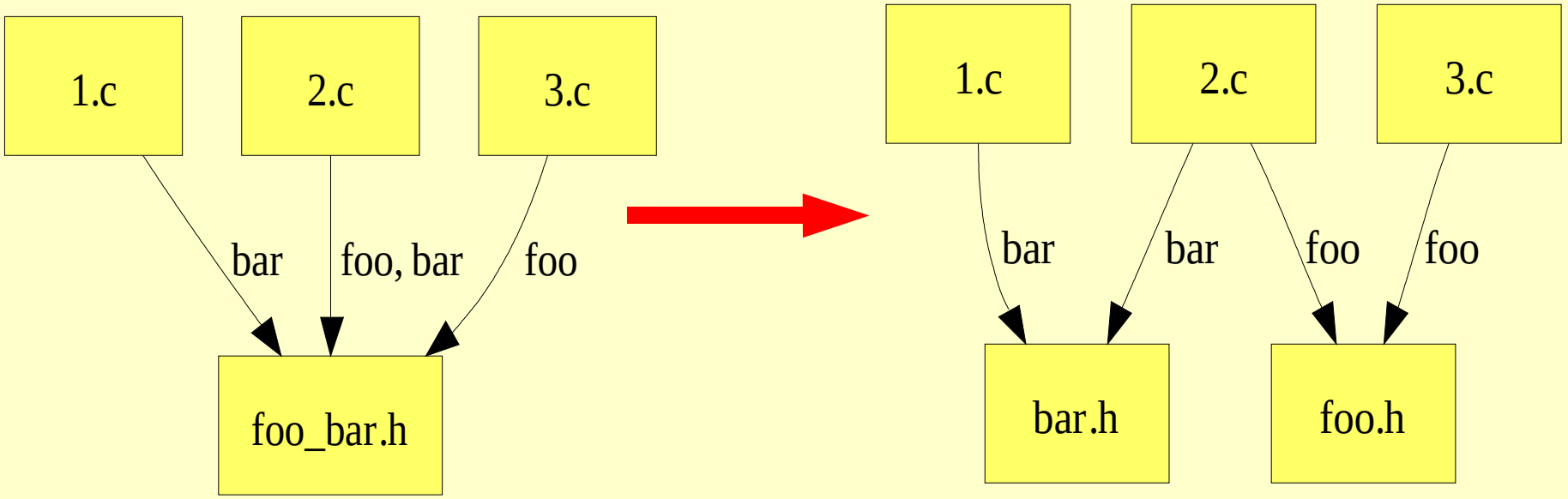
Reorganizing includes (example)



Reorganizing includes

- Determine the definitions of a C file $c \in Cfile$
  - $U = UnitsOf[c]$
- Determine the required declarations
  - $R = RequirementsOf^{+}[U]$
- Determine the header files that contain them
  - $I = FilesOf[R]$
- Let c include I

Restructuring headers (example)



Restructuring headers

- Start with an initial partitioning:
  - put each declaration in its own header
  - $H = \{ \{d\} \mid d \in Decl \}$
- Do a fixpoint computation over H:
  - if any A, B in H with $UsersOf^{+}[A] = UsersOf^{+}[B]$
  - merge declarations: $H = H \cup \{A \cup B\} \setminus \{A, B\}$
- Determine file dependencies as before

Evaluation

- Reported performance improvements:
  - build size
  - fresh build time
  - incremental build time
- Nice features:
  - independent of compiler
  - orthogonal w.r.t. `make -j`, `distcc`, `ccache`
- Useful technique to scale continuous integration

Summary

- Continuous integration system Sisyphus
  - distributed architecture
  - selective recompilation
- Dependency minimization
  - build optimization
  - header restructuring

Questions?

References

- Yu et. al., "*Reducing build time through precompilations for evolving large software*", Tech. rep. CSRG-504, Univ. of Toronto, 2005.
- Yu et. al., "*Removing false code dependencies to speedup software build processes*", CASCON, 2003.
- Dayani-Fard et. al., "*Improving the build architecture of legacy C/C++ software systems*", submitted to FASE, 2005.