Package-Based Software Development

Merijn de Jonge

http://www.win.tue.nl/~mjonge

Eindhoven Technical University, The Netherlands

October 10, 2003

Component

A component is a binary, independently deployable building block, that can be composed by third parties — Czyperski, 1999.

Thus,

- a component is *immutable* (CBSE is form of *black-box* reuse)
- *not* dependent on other components (only dependent on *interfaces*)
- composition is *not* pre-defined

Component

A component is a binary, independently deployable building block, that can be composed by third parties — Czyperski, 1999.

There are many instances of this component definition

- a component is *immutable* (CBSE is form of *black-box* reuse)
- *not* dependent on other components (only dependent on *interfaces*)
- composition is *not* pre-defined

Levels of Composition

- *module level* — functions, classes, ...
- *execution level* — programs, component architectures (COM), ...
- *code level* — files, directories, libraries, ...
- *design level* — specifications, design documents, requirements, ...

Code-level Components

CBSE is mostly concerned with software reuse between *applications*, i.e., with *execution-level components*.

Question

Given a collection of such execution-level components, how is reuse organized between them, i.e., what are *code-level components*?

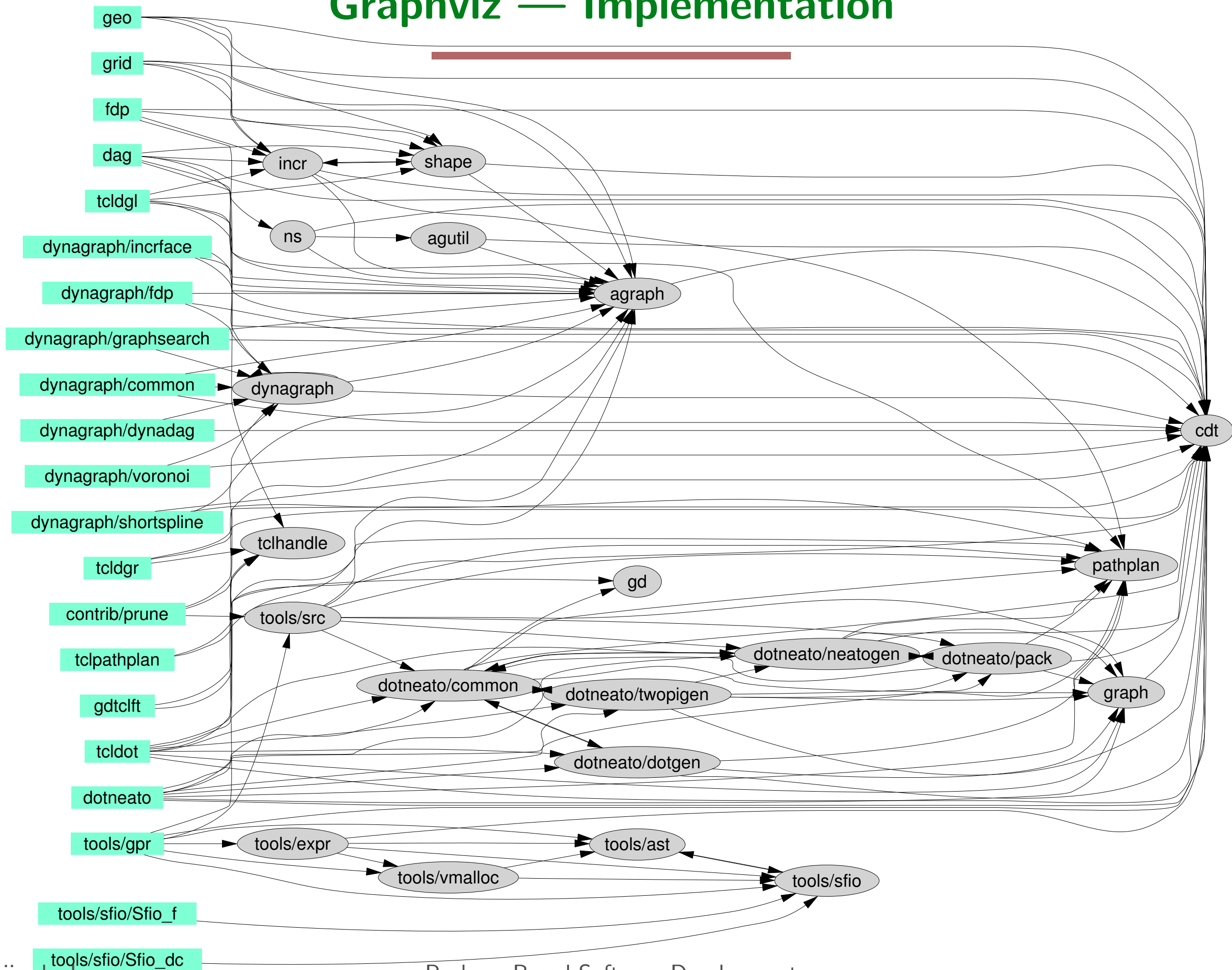
Goal

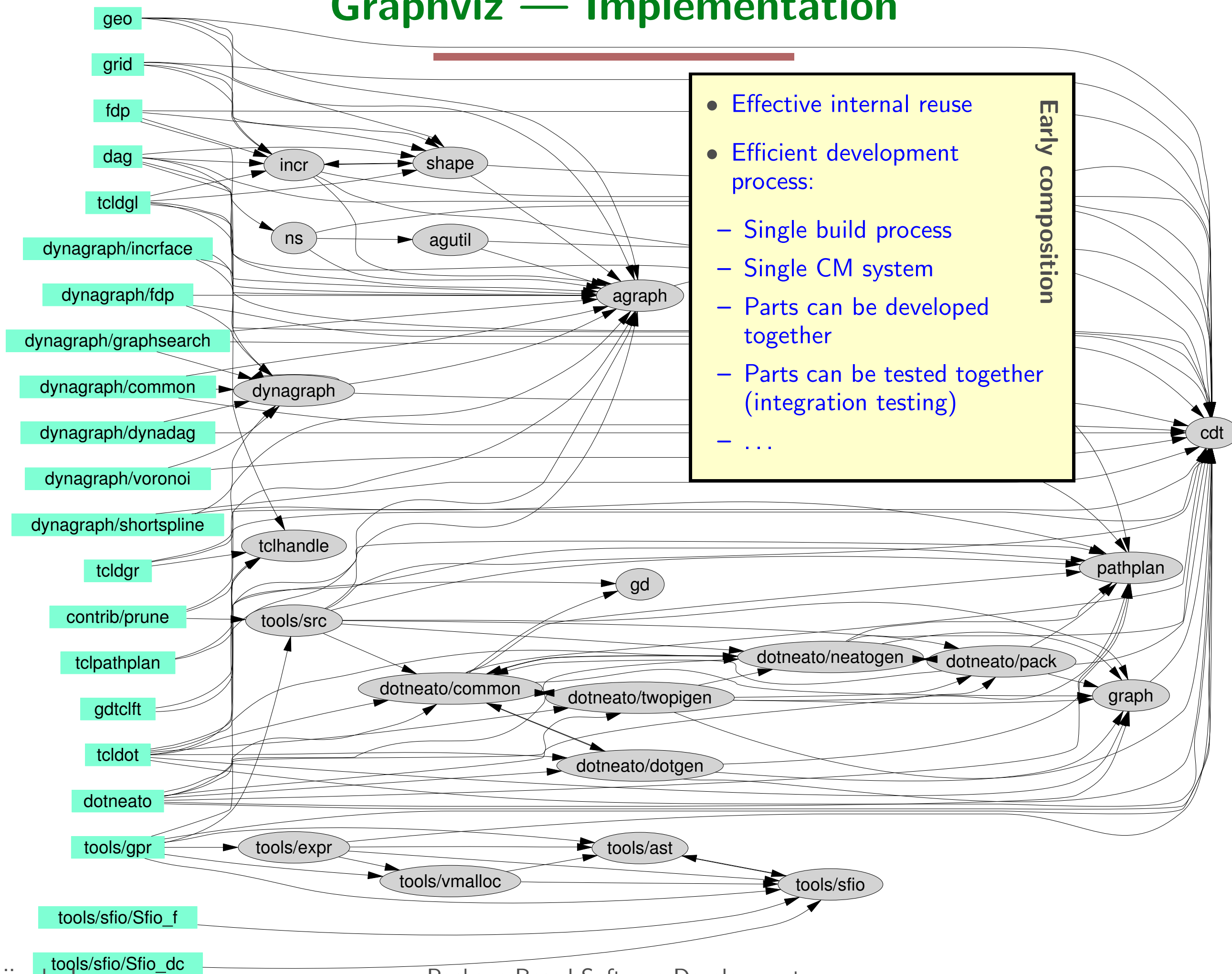
Bring CBSE principles to code-level to improve code-level software reuse

Graphviz — Example of Reusable Components

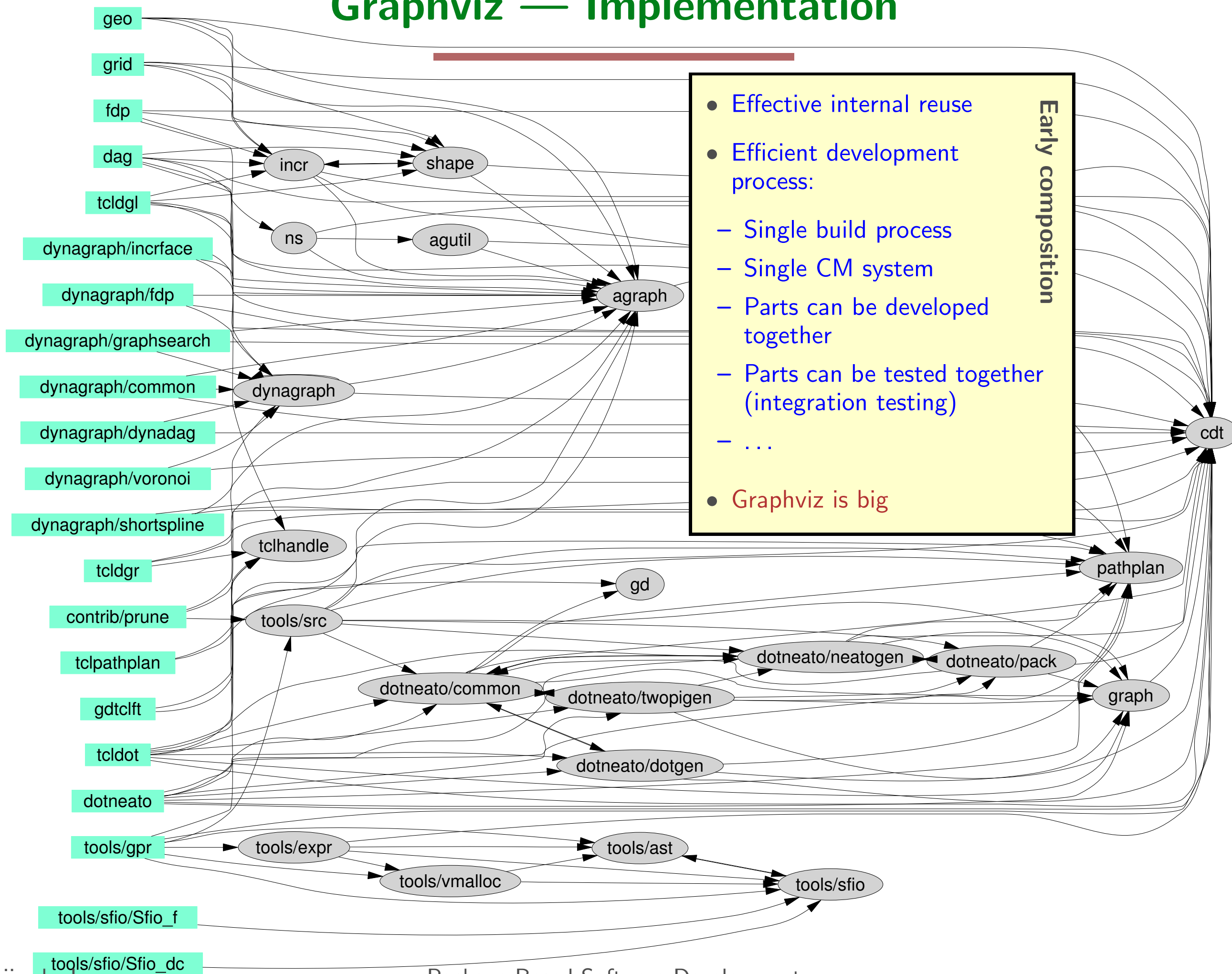
- geo
- grid
- fdp
- dag
- tcldgl
- dynagraph/incrface
- dynagraph/fdp
- dynagraph/graphsearch
- dynagraph/common
- dynagraph/dynadag
- dynagraph/voronoi
- dynagraph/shortspline
- tcldgr
- contrib/prune
- tclpathplan
- gdtclft
- tcldot
- dotneato
- tools/gpr
- tools/sfio/Sfio_f
- tools/sfio/Sfio_dc

Graphviz — Implementation




Graphviz — Implementation
Effective internal reuse
Efficient development process:
– Single build process
– Single CM system
– Parts can be developed together
– Parts can be tested together (integration testing)
– . . .
Early composition
geo
grid
fdp
dag
tcldgl
dynagraph/incrface
dynagraph/fdp
dynagraph/graphsearch
dynagraph/common
dynagraph/dynadag
dynagraph/voronoi
dynagraph/shortspline
tcldgr
contrib/prune
tclpathplan
gdtclft
tcldot
dotneato
tools/gpr
tools/sfio/Sfio_f
tools/sfio/Sfio_dc
incr
shape
ns
agutil
agraph
dynagraph
tclhandle
gd
tools/src
dotneato/common
dotneato/neatogen
dotneato/twopigen
dotneato/pack
dotneato/dotgen
tools/expr
tools/vmalloc
tools/ast
tools/sfio
cdt
pathplan
graph

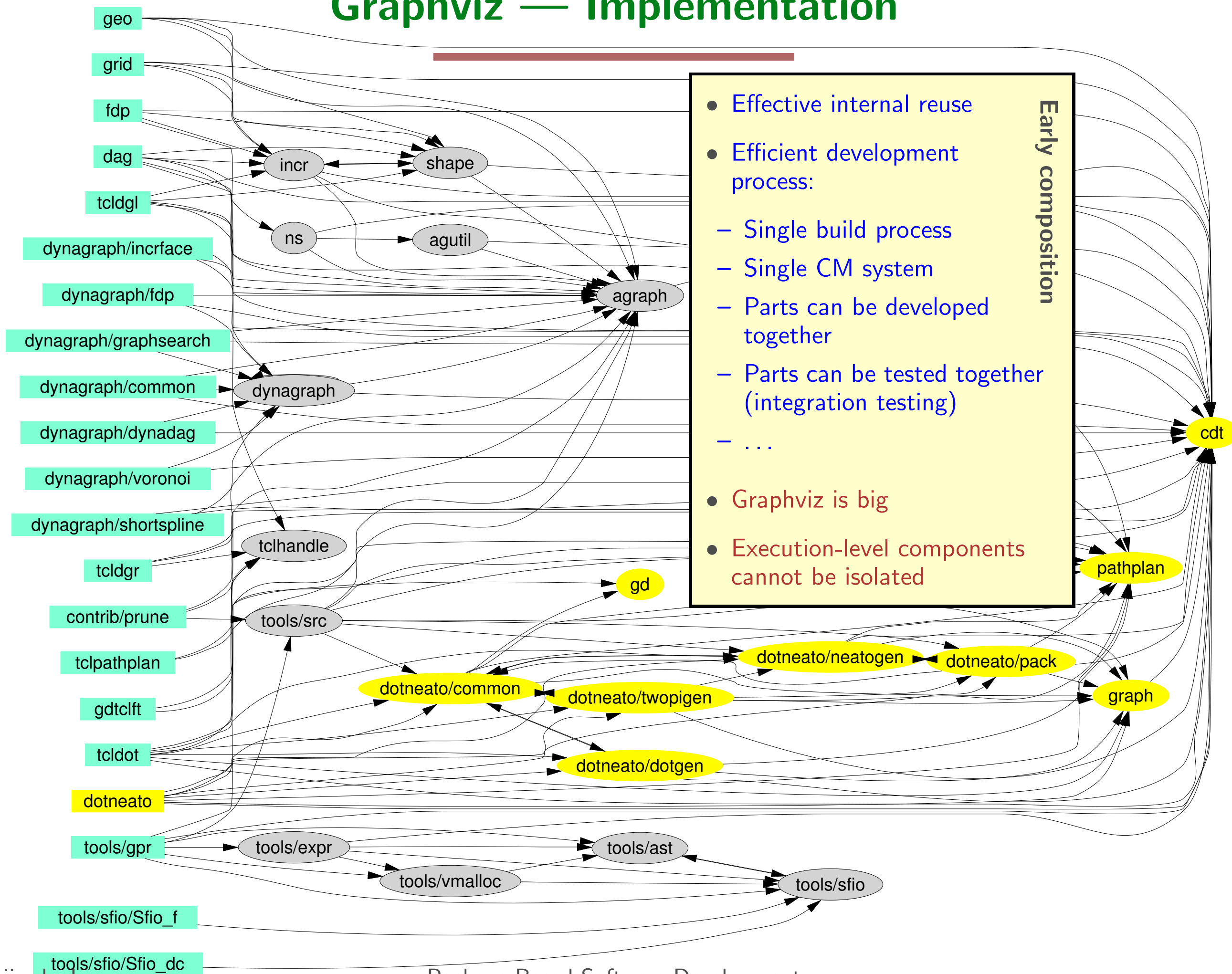
Graphviz — Implementation



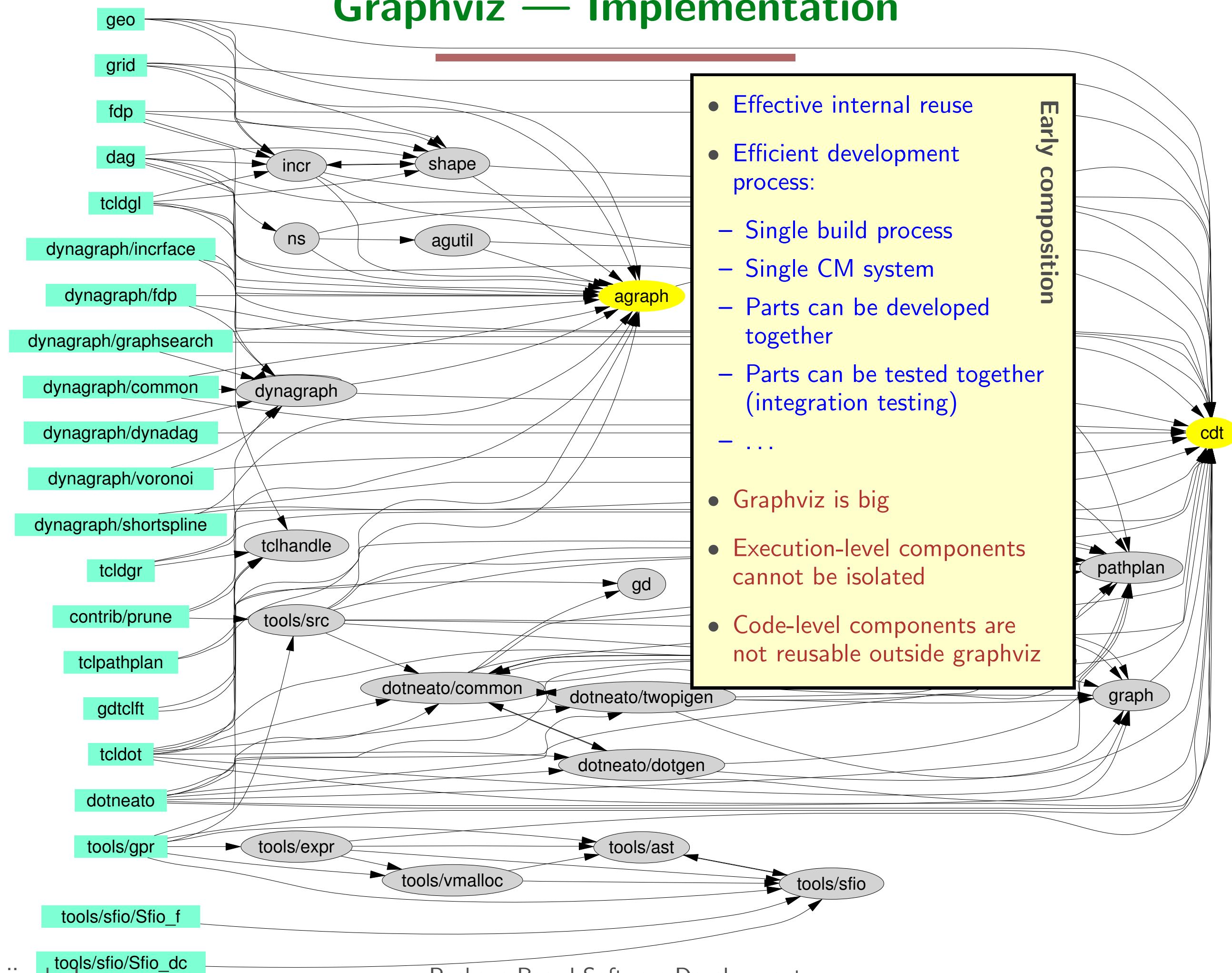
Early composition

- Effective internal reuse
- Efficient development process:
  - Single build process
  - Single CM system
  - Parts can be developed together
  - Parts can be tested together (integration testing)
  - . . .
- Graphviz is big

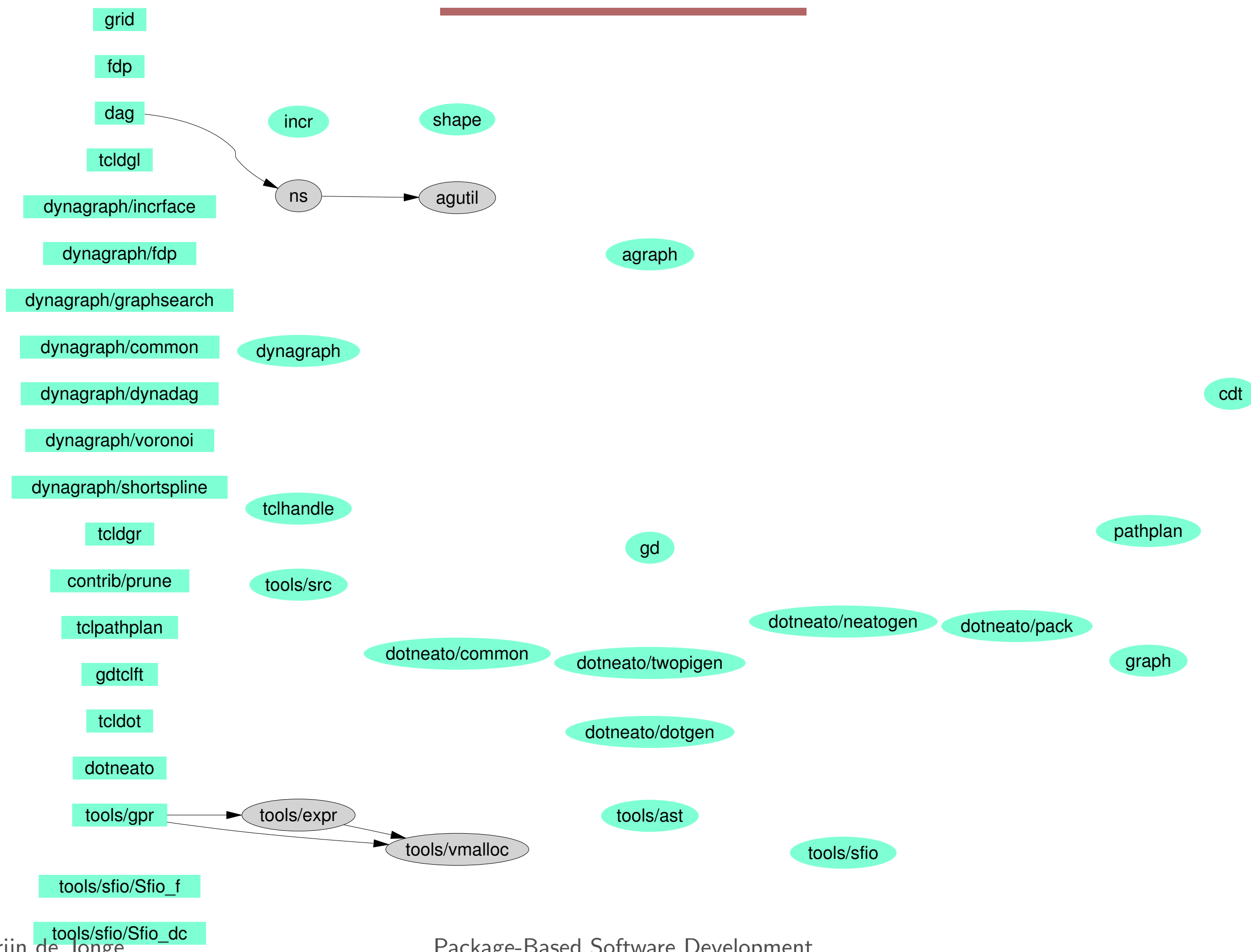
Graphviz — Implementation



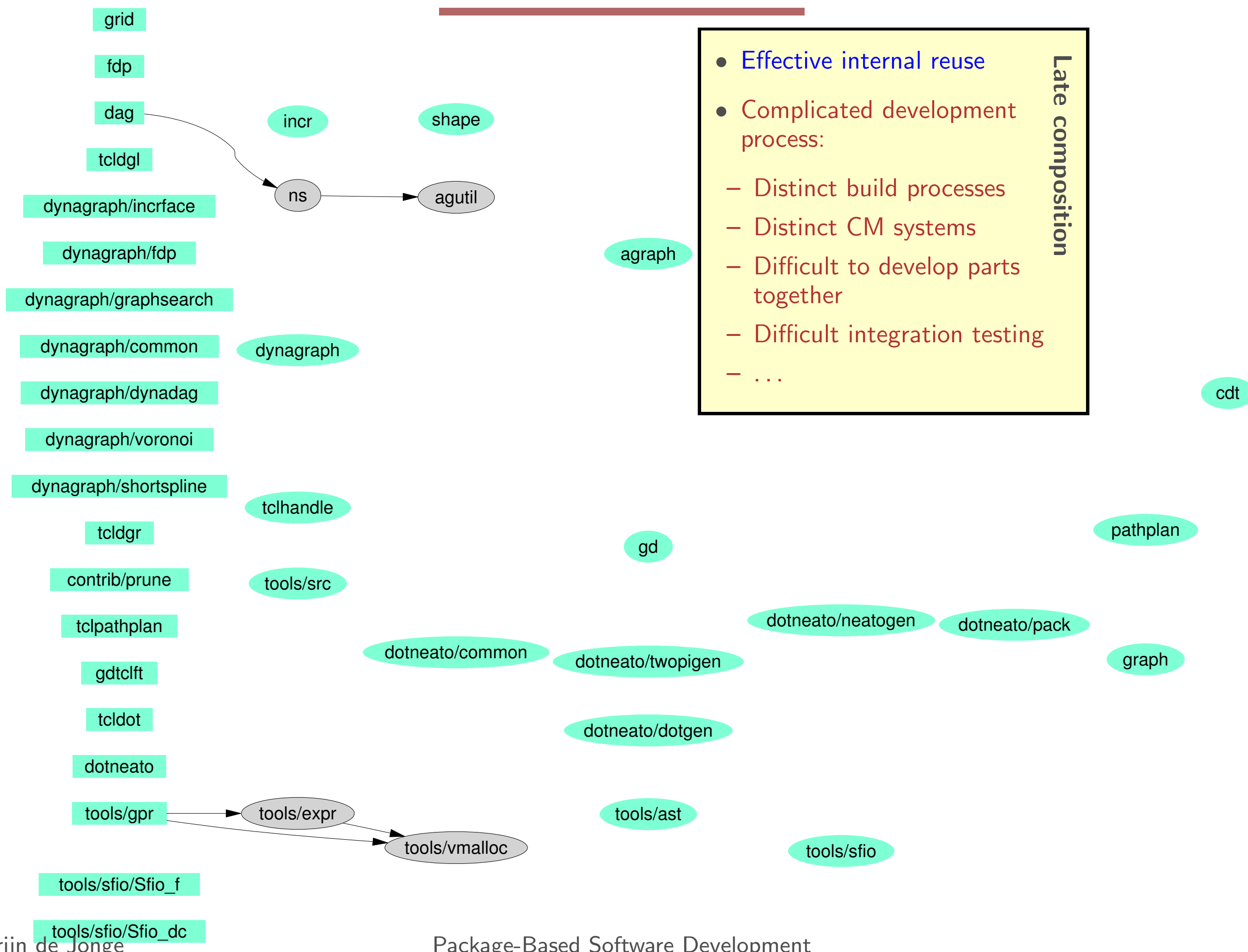
Graphviz — Implementation



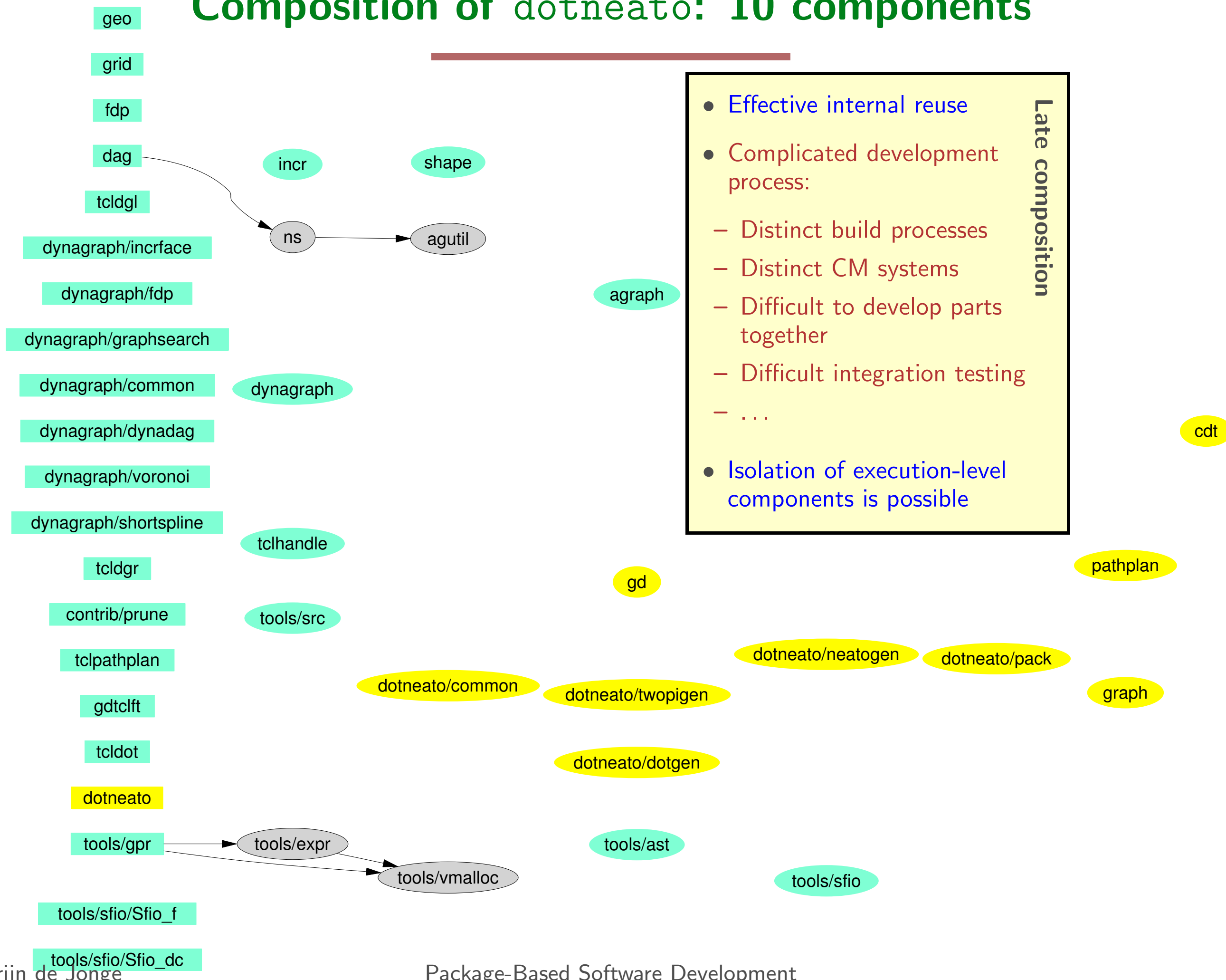
Splitting-up To Improve Reusability: 37 Components



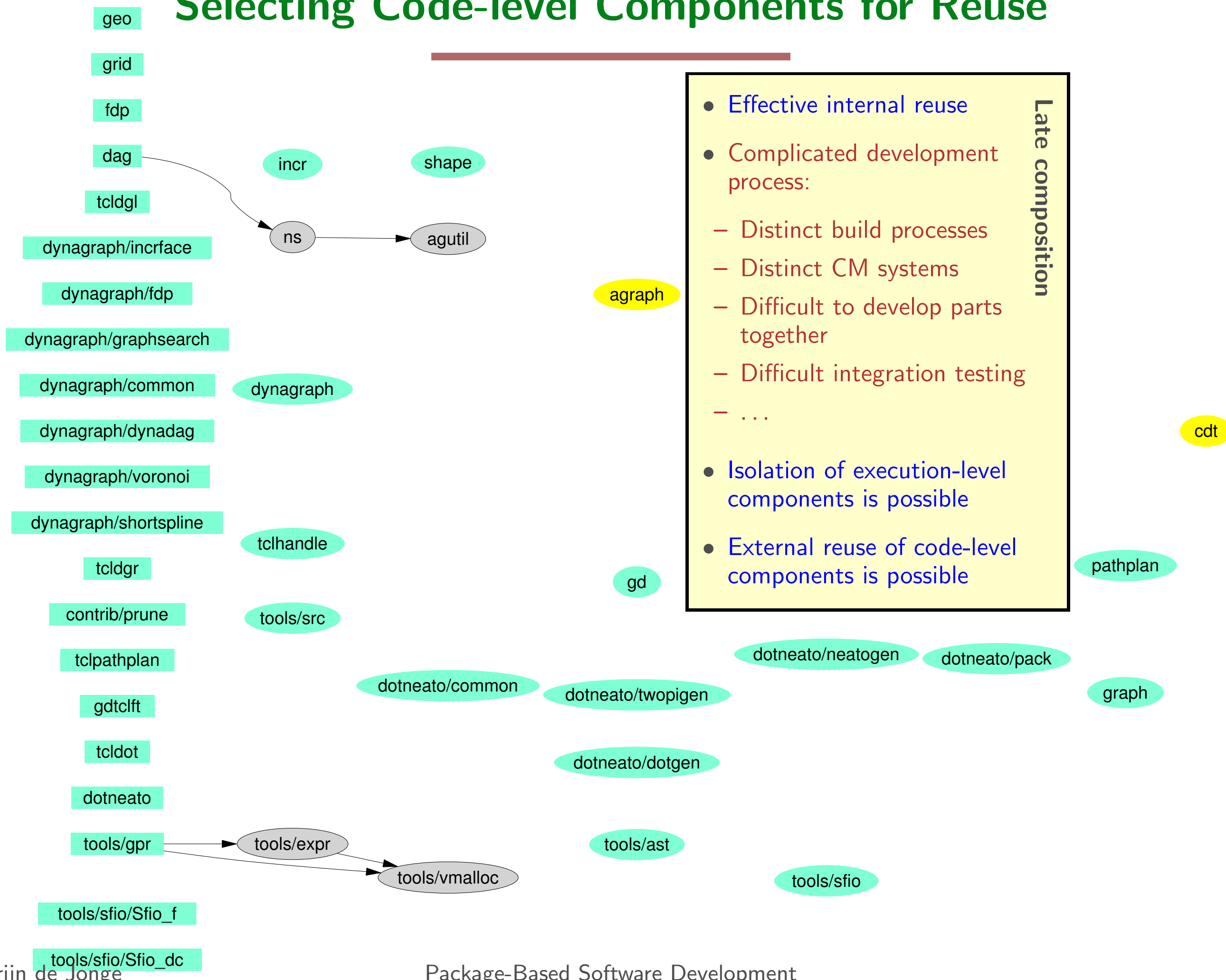
Splitting-up To Improve Reusability: 37 Components



Composition of `dotneato`: 10 components



Selecting Code-level Components for Reuse



- Effective internal reuse
- Complicated development process:
  - Distinct build processes
  - Distinct CM systems
  - Difficult to develop parts together
  - Difficult integration testing
  - ...
- Isolation of execution-level components is possible
- External reuse of code-level components is possible

Observation

For software software development:

- Composition comes first

For software reuse:

- Composition comes last

Observation

For software software development:

- Composition comes first

For software reuse:

- Composition comes last

Question: How to combine both goals?

Goals

Software development:

- Component development in context of applications
- Development tasks on component compositions rather than on components in isolation

Software reuse:

- Fine-grained software reuse
- Crossing component, application, group, and institute boundaries

How?

Applying principles of CBSE to the code-level

- Making code-level artifacts *available* for reuse by decomposing software systems into code-level components

- Making code-level components *usable* by assembling them into software systems using automatic composition techniques

Software Building Via Standardized Build Interface

Automake provides a build interface consisting of a standard set of build actions:

- all
- install
- uninstall
- check
- dist
- ...

It is always obvious how to compile/install a program

All Compile-time Variability Via Configuration Interface

Autoconf provides a standard syntax for compile-time switches and it provides a set of predefined switches

```
configure --prefix=/usr
configure --with-aterm=/usr/lib
configure --enable-optimization
configure --disable-debug
configure --help
```

It is always obvious how to specify an installation directory, or how to obtain a list of configuration switches ...

No Coupling Via SCM System

Automake supports generation of versioned software distributions

```
configure.in:
   AM_INIT_AUTOMAKE(aterm, 1.6.3)
```

```
make dist
```

- We call a versioned distribution a *package*
- Reuse based on packages prevents coupling via SCM systems

Code-level Design Rules

Patterns or rules to improve code-level software reuse

- Files with strong cohesion belong in the same directory
- Reuse/deployment with directory granularity
- Circular dependencies should be minimized
- Late binding of build-time dependencies
- Software building via standardized build interface
- All compile-time variability via configuration interface
- Independent build process definition per source-level component
- Independent configuration interface per source-level component
- No composition in build process definition or configuration process
- No coupling via SCM system
- Automatic component composition

Code-level Design Rules

Patterns or rules to improve code-level software reuse

- Files with strong cohesion belong in the same directory
- Reuse/deployment with directory granularity
- Circular dependencies should be minimized
- Late binding of build-time dependencies
- So
- Al
- Independent build process definition per source-level component
- Independent configuration interface per source-level component
- No composition in build process definition or configuration process
- No coupling via SCM system
- Automatic component composition

If applied, then decomposition in code-level components becomes possible and composition can be automated.

Source Code Components

Rules 1–10 turn code-level artifacts into *true* components

- Independently deployable
- Composable by third-parties
- Immutable (?)

From now on, we call such components *Source Code Components*

Source Tree Composition

With rules 1–10, we can turn a system (e.g., graphviz) into source code components

Now we want to construct a strongly coupled source tree from them. I.e., we want to make a composition of:

- All needed implementation files
- All build processes
- All configuration processes

Such that the result can be managed as a unit. I.e., that building, testing, distributing etc. can be performed for the composite system as a whole.

Compositionality

Rule 11 states that source tree composition should be *automated*

Observe:

- Build processes are standardized
- All dependencies are bound via configuration interfaces
- All configuration parameters are bound via configuration interfaces

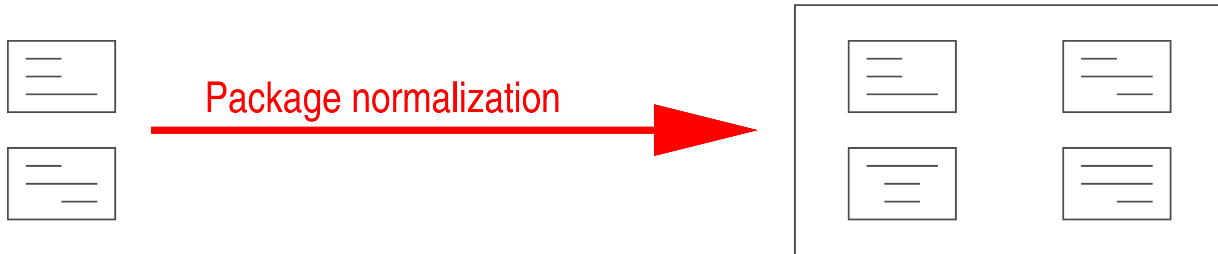
If we formalize dependencies and configuration parameters, source tree composition can be automated

Package Definitions

Dependencies, parameters a.o., are formalized in *package definitions*.

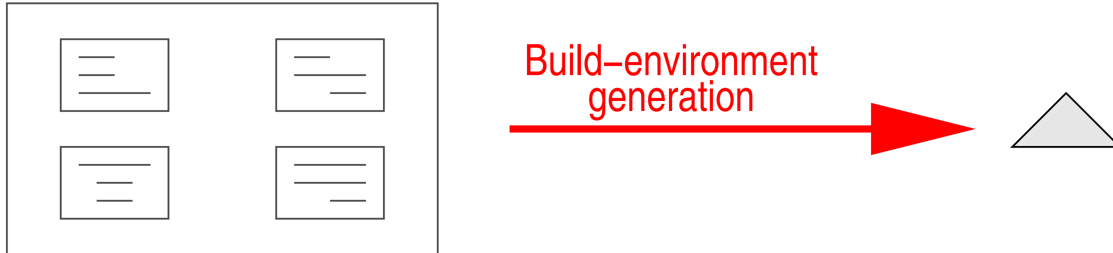
package
identification
name=asf
version=1.2
location=http://www.cwi.nl/projects/MetaEnv/asf
info=http://www.cwi.nl/projects/MetaEnv/asf
description='Implementation of the rewriting language ASF.'
keywords=asf**,** compiler**,** library
configuration interface
traversals 'evaluator implements traveral functions'
requires
sdf-support 0.6
pt-support 0.7
aterm 1.6.5
toolbus 0.21.1 **with** java=on

From Package Selection to Bundle Definition



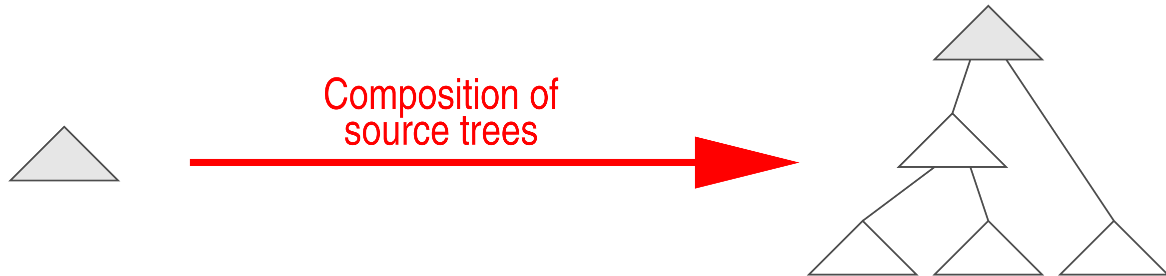
- Obtained through *package normalization*
- Defines ingredients of a bundle: the selected packages and those that are required by them
- Defines (partial) configuration of packages

From Bundle Definition to Source Tree (1)



- Generation of an “empty” source tree from a bundle definition
- Contains global build process
- Contains global configuration process
- Knows how to collect/integrate source trees of constituent packages

From Bundle Definition to Source Tree (2)



- Source trees are obtained and integrated
- Result is a strongly coupled source tree with integrated build and configuration processes
- One action to build the complete composite system
- Composite system can be distributed as a *self-contained* software bundle

Package Bases

Online Package base - Mozilla

File Edit View Go Bookmarks Tools Window Help

This is the *Online Package Base*: a collection of reusable Free Software packages developed at different institutes.

bundle | dependency graph | clear | Package search: | search

apigen — 1.5 | 1.3 | exclude

Generate typed tree-like data-structures in C and Java. The implementation is based o
keywords: *meta-environment, code generation, C, java, aterm, maximal sharing*

asc-support — 1.6 | 1.5 | 1.4 | 1.3 | 1.2 | 1.1 | 1.0

The library libasc-support is a runtime library which is required to build stand-alone executables from C files generated by the ASF+SDF compiler.
keywords: *meta-environment, asf, compiler, run-time library*

asf — 1.6 | 1.5 | 1.4 | 1.3 | 1.2 | 1.1 | 1.0

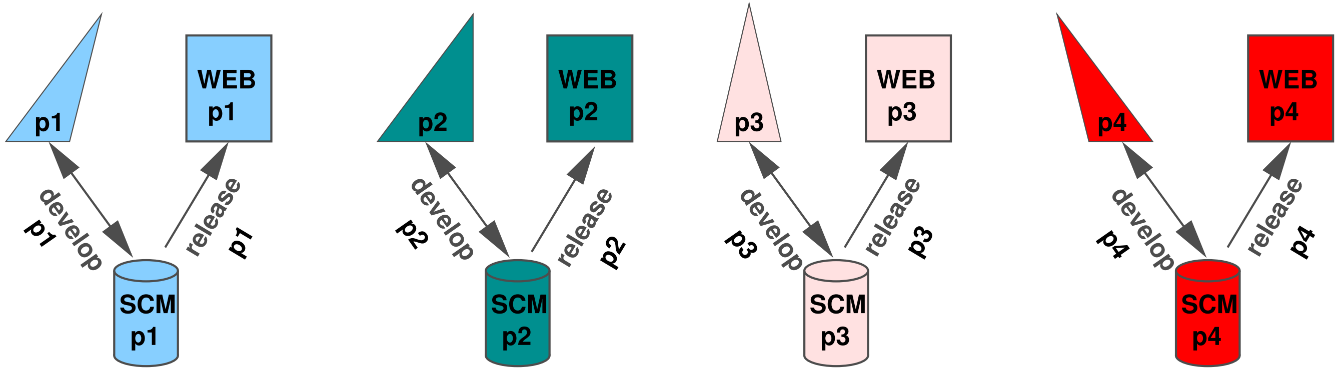
This package contains the implementation of the rewriting language ...yntax. It can be used to define ...nming languages. It can also ...nsformations in a very concise manner.
keywords: *meta-environment, asf, compiler, interpreter, term rewriting, source code transformation, program transformation, list matching, traversal functions, compiler generation*

asf-library — 0.3 | 0.2 | 0.0 | exclude

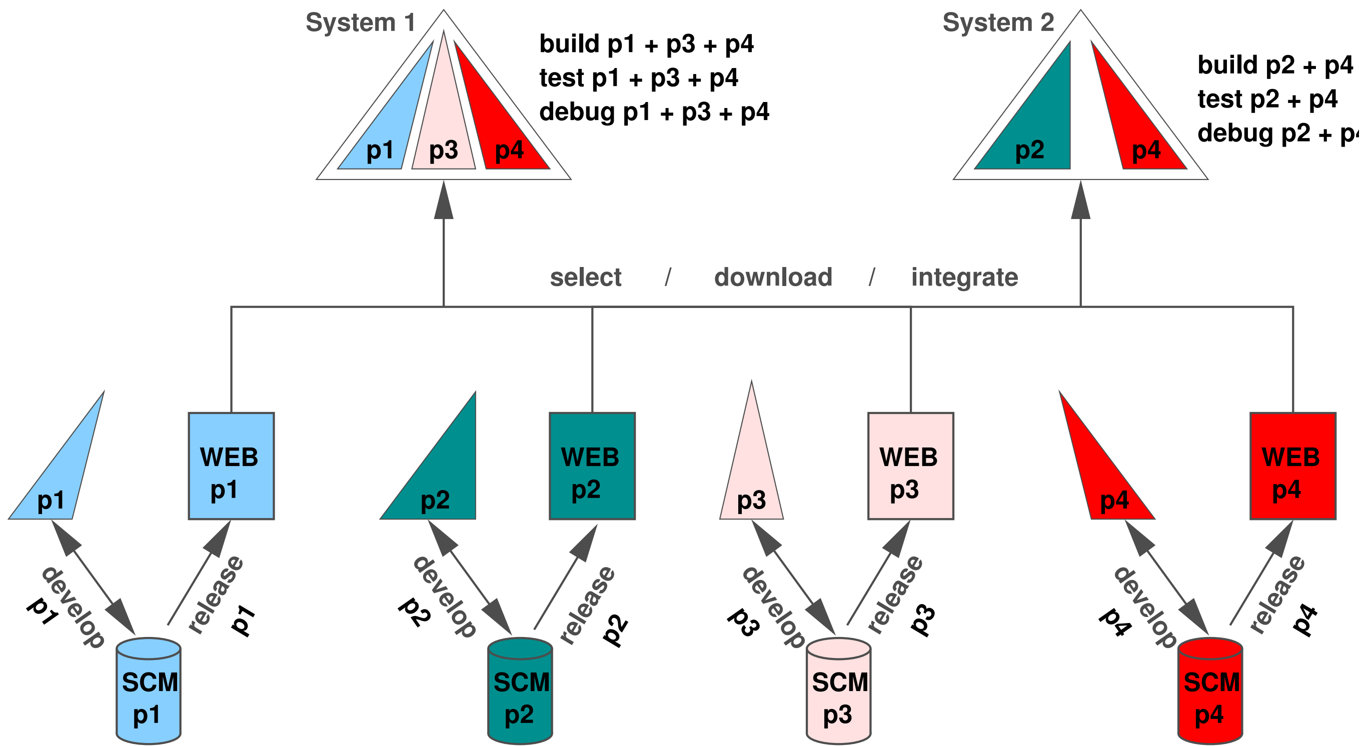
A library of ASF+SDF modules. Also implementations of builtins for ASF
keywords: *meta-environment, asf, utility, standard library, builtins*

`www.program-transformation.org/package-base`

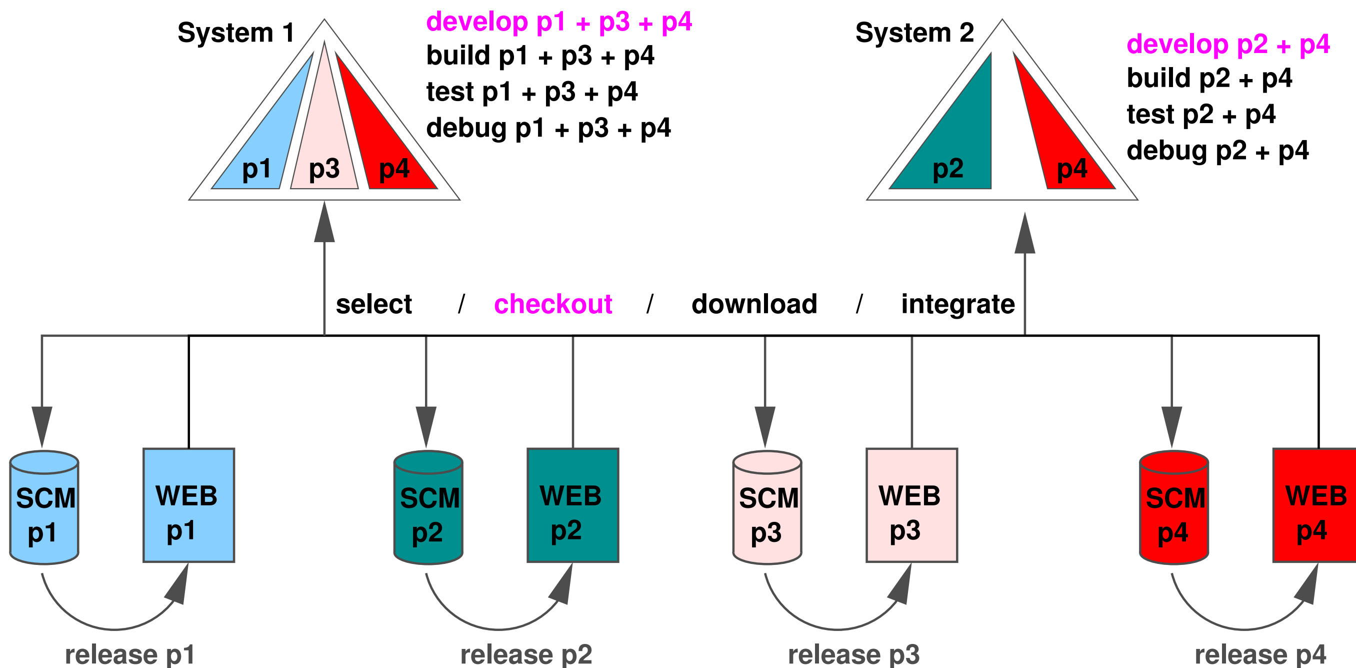
Package Development Cycles (1)



Package Development Cycles (2)



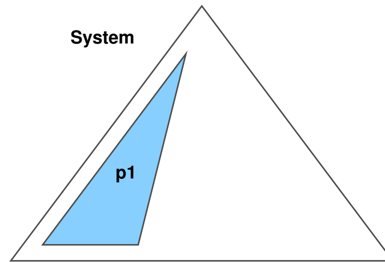
Integrated Package Development Cycles



Implementation

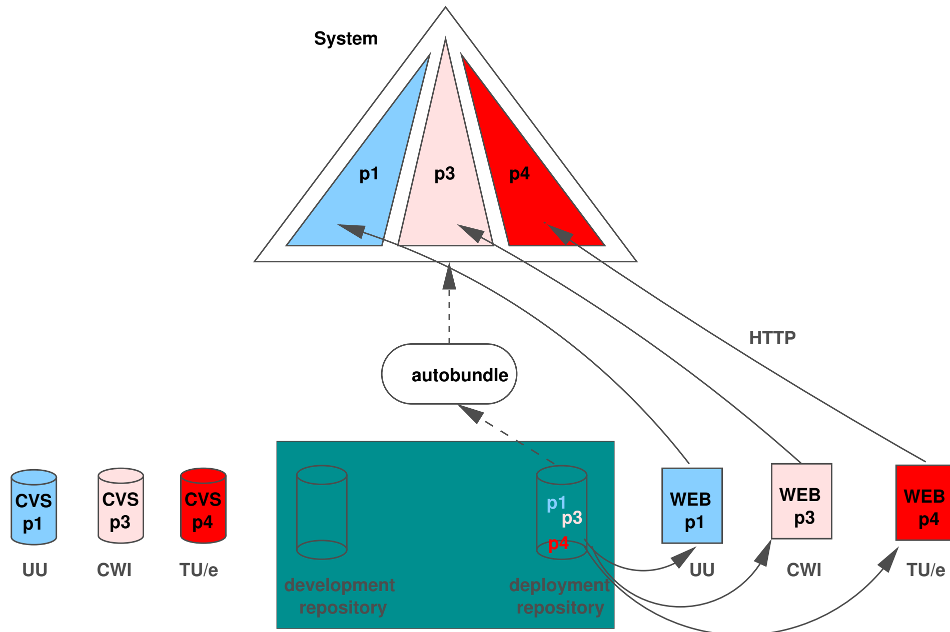
- *Separate* development/deployment package bases
- Tool '`autobundle`' for automated *source tree composition*
- Tool '`edit`' to put component in *development mode*
- Tool '`release`' to switch back to *deployment mode*

Example (deployment)



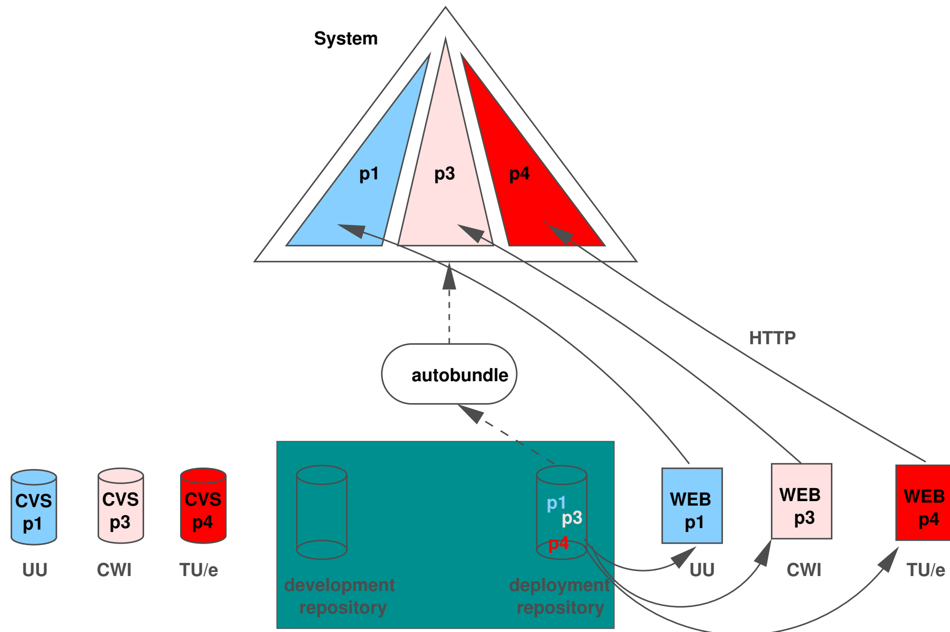

```
> autobundle -p p1
```

Example (deployment)



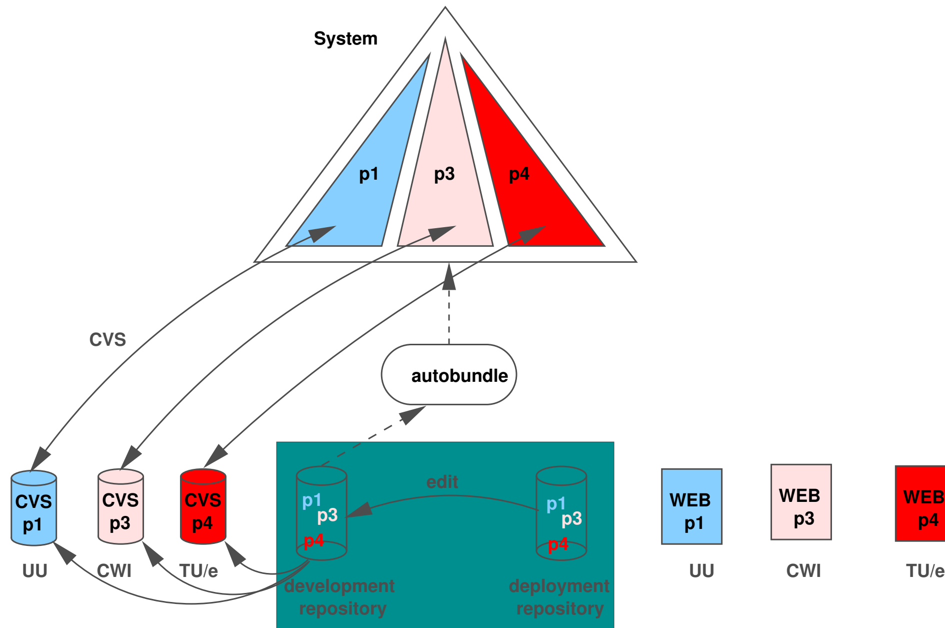

```
> autobundle -p p1
Obtaining a distribution of "p1" via HTTP
Obtaining a distribution of "p3" via HTTP
Obtaining a distribution of "p4" via HTTP
```

Example (development)



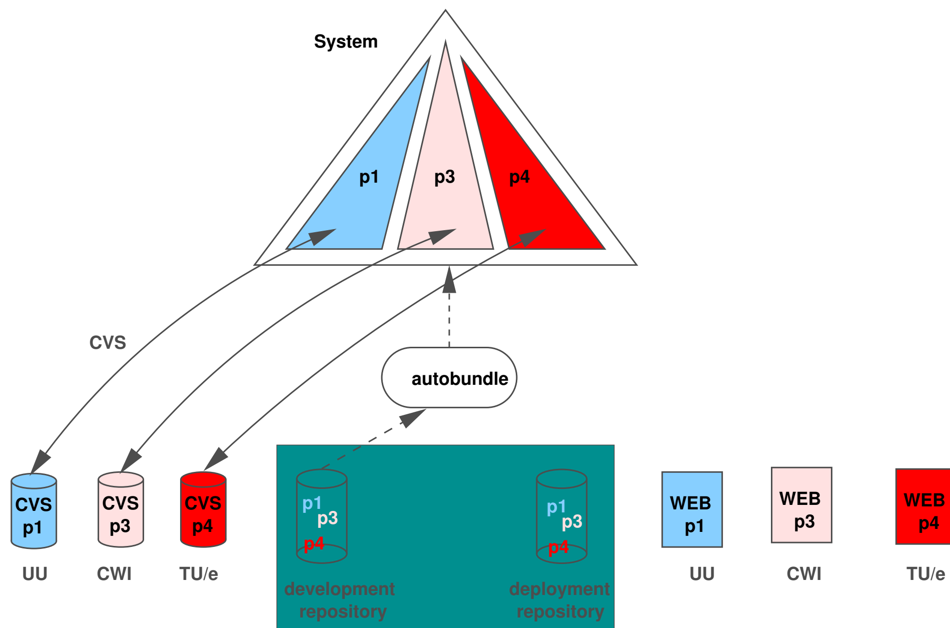

```
> edit p1 p3 p4
Obtaining p1 from CVS
Obtaining p3 from CVS
Obtaining p4 from CVS
```

Example (development)




```
> edit p1 p3 p4
Obtaining p1 from CVS
Obtaining p3 from CVS
Obtaining p4 from CVS
```

Example (development)

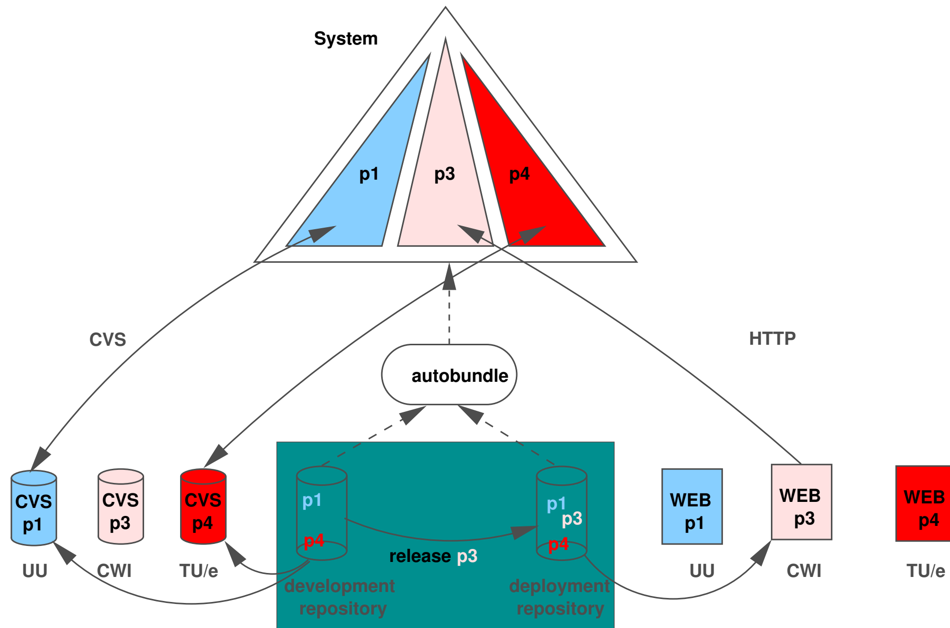



```
> cvs update
cvs update: Updating p1
cvs update: Updating p3
U p3/f.c
cvs update: Updating p4

> emacs p3/f.c
```

```
> cvs commit
cvs commit: Examining p1
cvs commit: Examining p3
cvs commit: Examining p4
Checking in  p3/f.c
```

Example (release)




```
> release p3
Obtaining a distribution of "p3" from HTTP
```

(Semi-)Automatic Decomposition (Graphviz)

process

- Code-level analysis
  - analyzing directory structure
  - analyzing Makefiles
  - Fine-tuning
- Code-level transformation
  - Creating package definitions
  - Creating packages
  - Fine-tuning
- Importing in SCM system
- Creating package-base

Code-level Analysis

Analysis

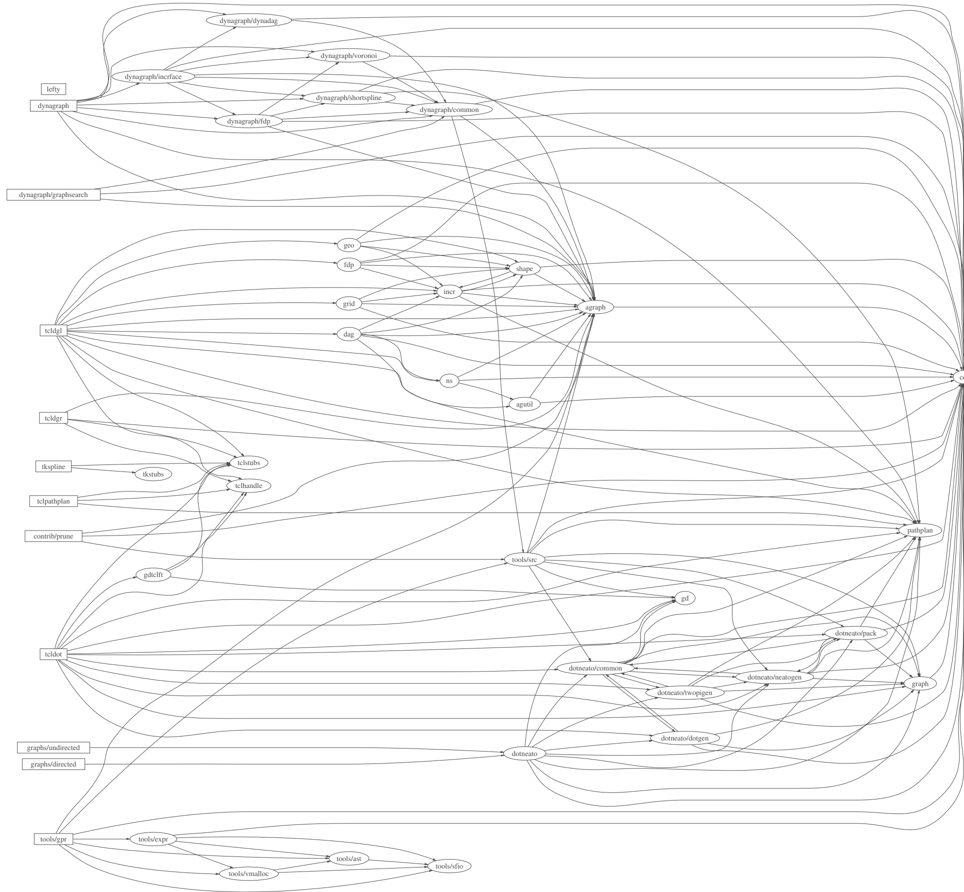
- Directory analysis to determine components (i.e., in principle each directory constitutes a component)
- Makefile analysis to find reuse relations (i.e., each directory reference indicates a reuse relation)

Fine-tune

- Add missing edges
- Remove (specific) cyscles
- Remove sub-graphs for strongly coupled directories

Result is a dependency graph in Dot, displaying components as nodes and reuse relations as edges

Directory Structure



Code-level Transformation

Transformation:

- Create package definitions for each component
- Create packages
  - Create configure.in from top-level configure.in by replacing directory references by dependency switches
  - Create/patch Makefile by replacing directory references by component references and local targets by package-specific targets
  - Fix directory structure by removing sub directories of separate components

Fine-tune:

- Fix circular dependencies
- Restructure some files

Package Structure (46 packages)

lefty-1.0
graphs_undirected-1.0
graphs_directed-1.0
tkstubs-1.0
tkspline-1.0
dotneato-1.0
dotneato_twopigen-1.0
dotneato_neatogen-1.0
dotneato_pack-1.0
graph-1.0
dotneato_common-1.0
tcldot-1.0
dotneato_dotgen-1.0
gd-1.0
gdtclft-1.0
tclstubs-1.0
tclhandle-1.0
tclpathplan-1.0
pathplan-1.0
tcldgr-1.0
tools_src-1.0
contrib_prune-1.0
dynagraph_shortspline-1.0
dynagraph_dynadag-1.0
dynagraph-1.0
dynagraph_common-1.0
cdt-1.0
dynagraph_fdp-1.0
dynagraph_incrface-1.0
dynagraph_voronoi-1.0
dynagraph_graphsearch-1.0
agraph-1.0
dag-1.0
ns-1.0
agutil-1.0
tcldgl-1.0
geo-1.0
incr-1.0
fdp-1.0
shape-1.0
grid-1.0
tools_gpr-1.0
tools_expr-1.0
tools_sfio-1.0
tools_ast-1.0
tools_vmalloc-1.0

Summary

Pure Component-based software engineering (with late composition)

- *Improves* software reuse but *complicates* software development process

In practice, large-scale components (early composition)

- *Improves* software development process but *limits* software reuse

Package-based software development

- Combines advantages of late *and* early composition
- By integrating component *deployment* and component *development*