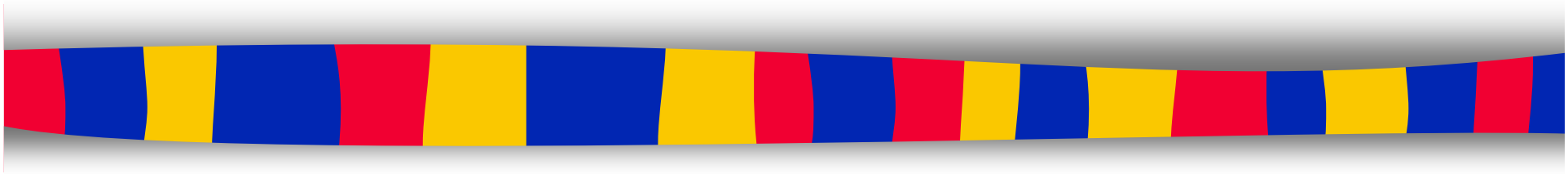
CISC 322

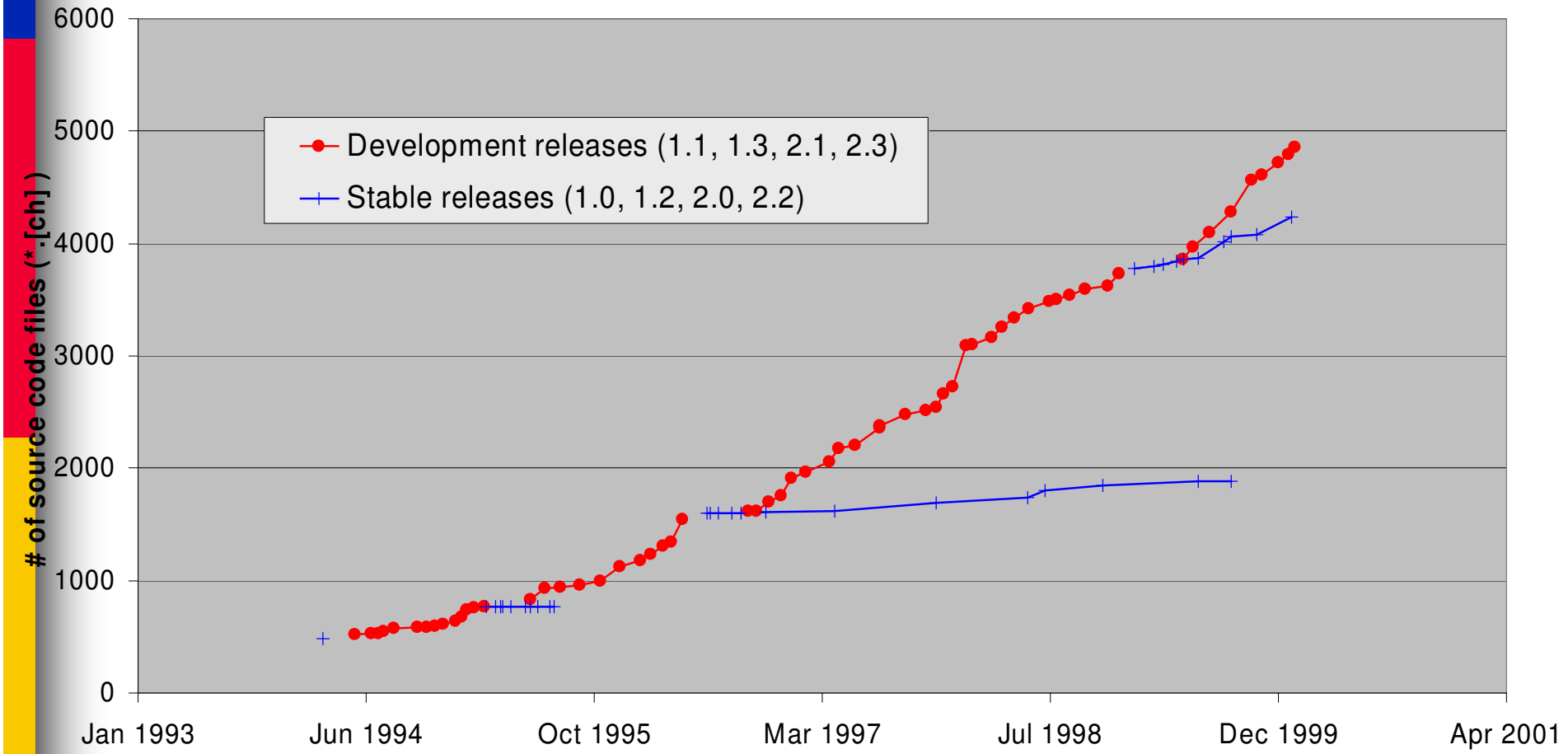
Software Architecture

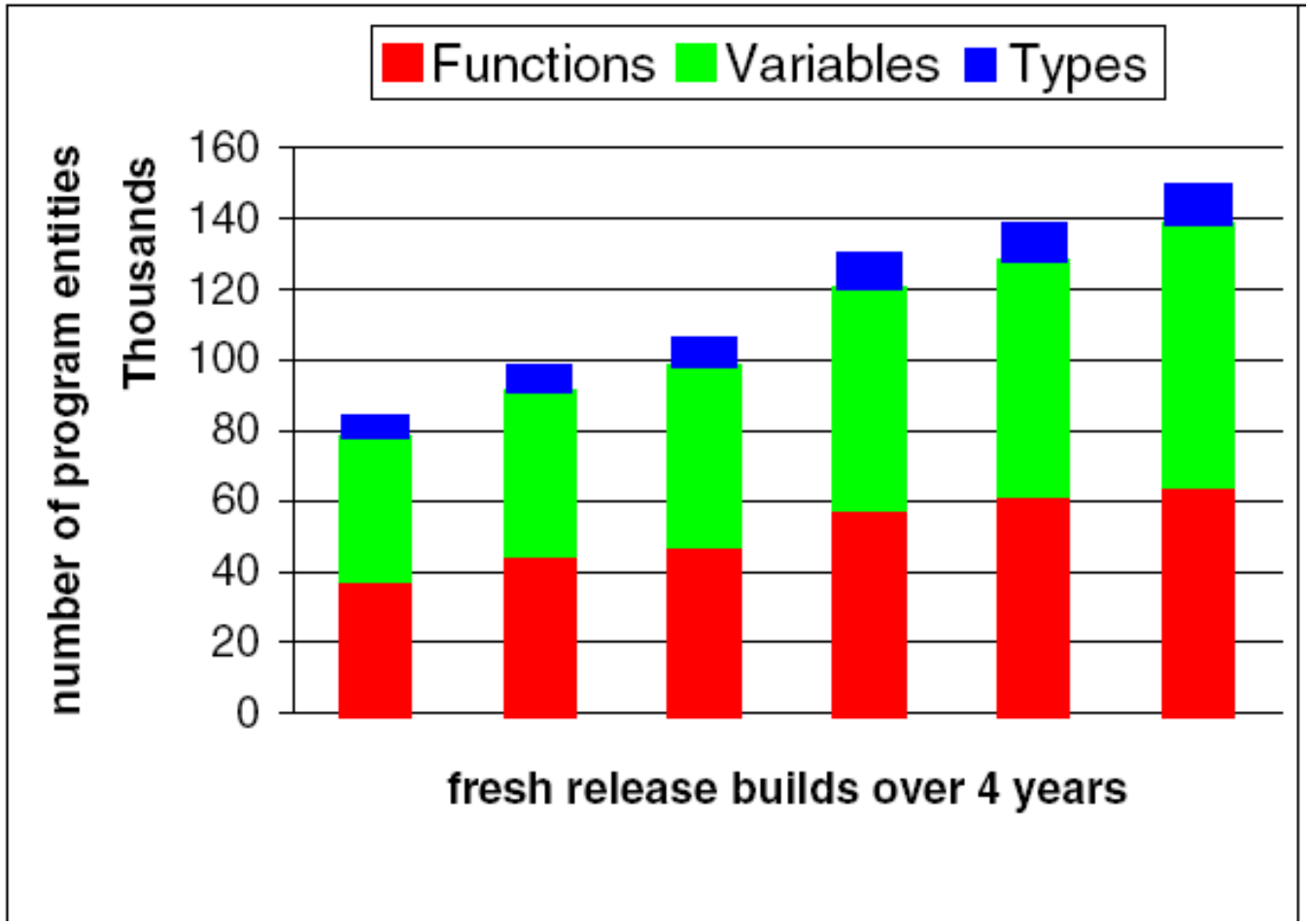


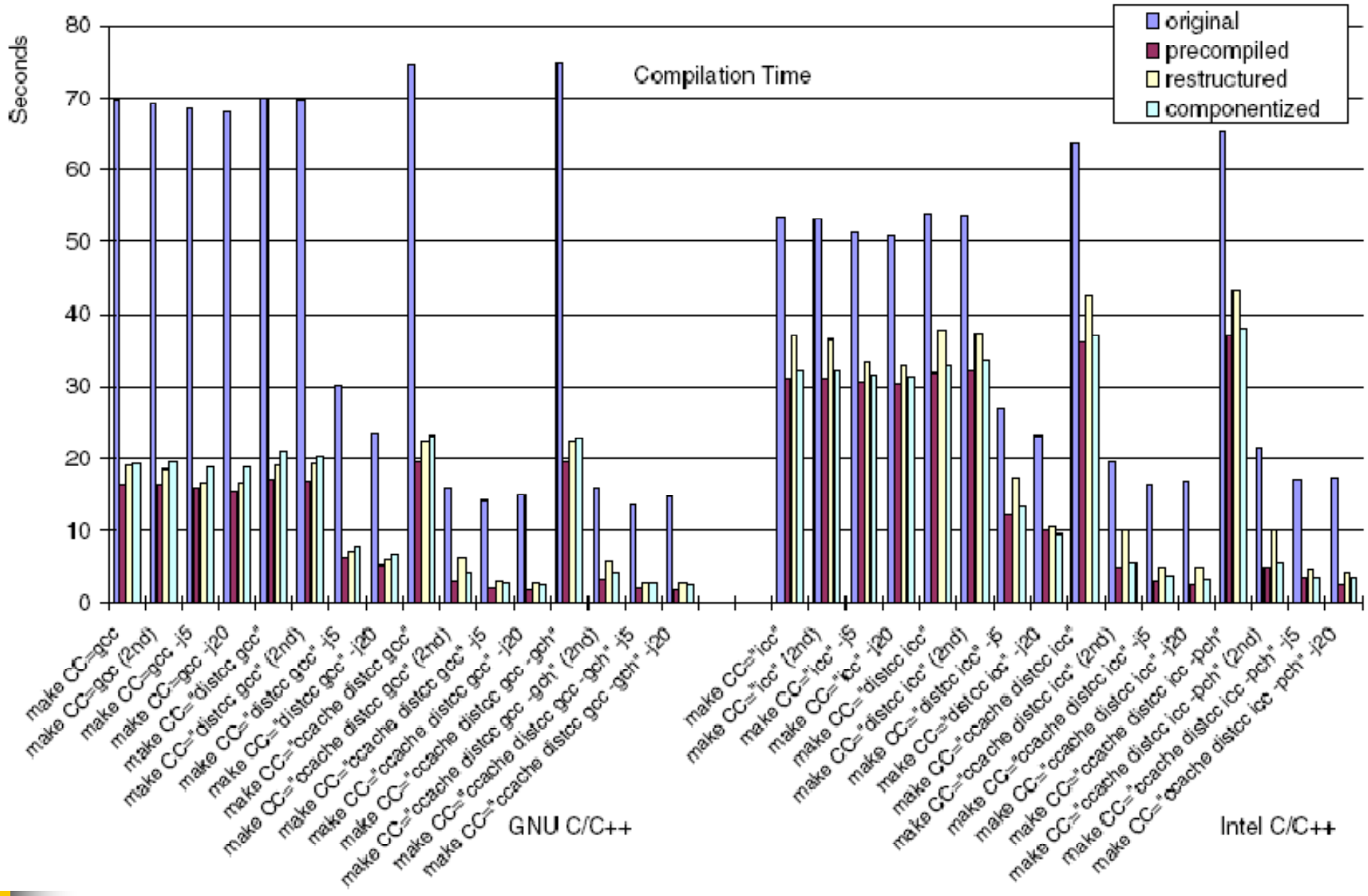
Extending the 4+1 Views:
Build Time View BTV
[based on slides by Michael W. Godfrey at UW]

Ahmed E. Hassan

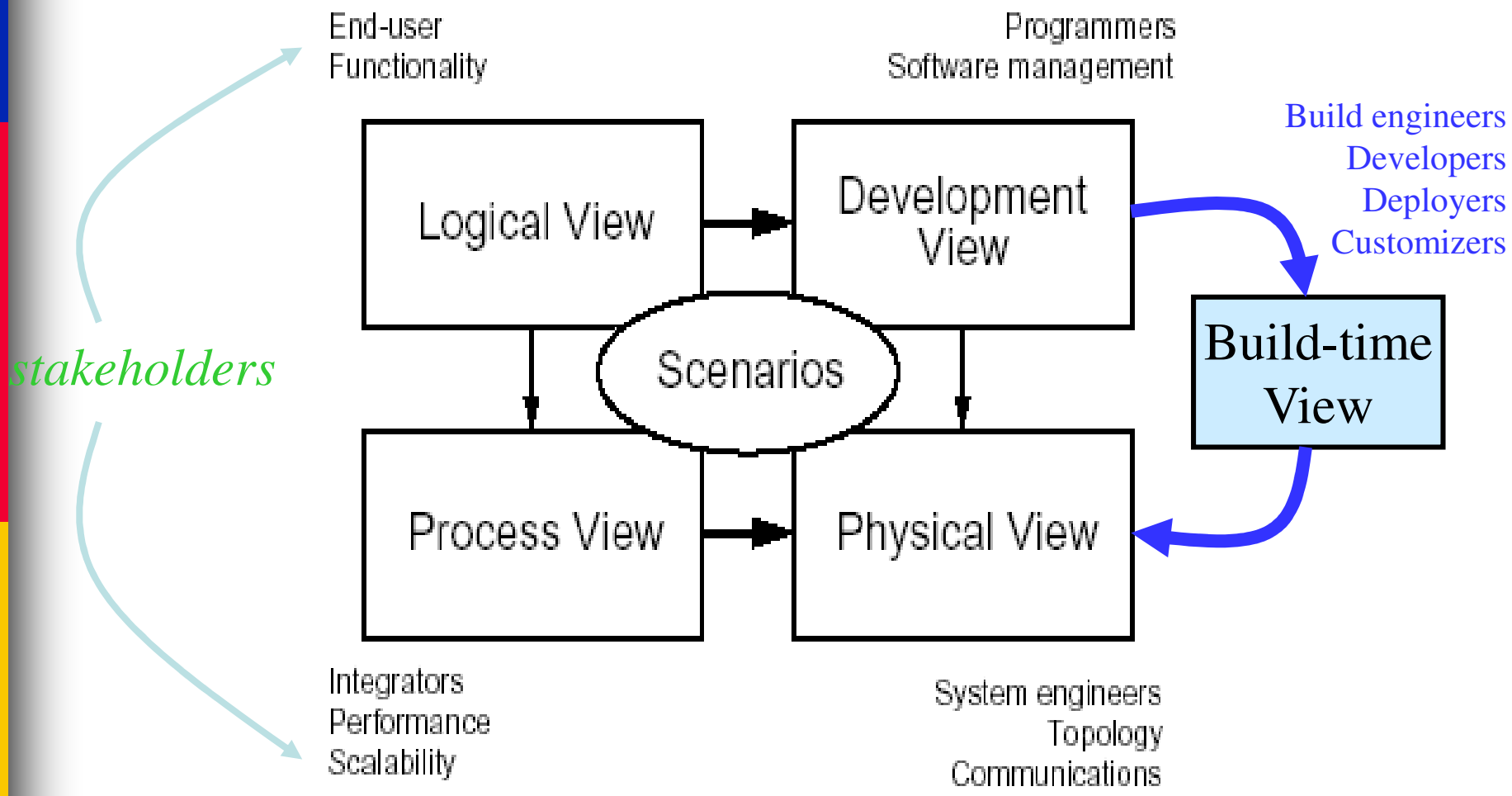
Growth of # of source files




Functions
Variables
Types
number of program entities
Thousands
160
140
120
100
80
60
40
20
0
fresh release builds over 4 years


Compilation Time
Seconds
original
precompiled
restructured
componentized
0
10
20
30
40
50
60
70
80
make CC=gcc
make CC=gcc (2nd)
make CC=gcc -j5
make CC=gcc -j20
make CC="distcc gcc"
make CC="distcc gcc" (2nd)
make CC="distcc gcc" -j5
make CC="distcc gcc" -j20
make CC="ccache distcc gcc"
make CC="ccache distcc gcc" (2nd)
make CC="ccache distcc gcc" -j5
make CC="ccache distcc gcc" -j20
make CC="ccache distcc gcc -gch"
make CC="ccache distcc gcc -gch" (2nd)
make CC="ccache distcc gcc -gch" -j5
make CC="ccache distcc gcc -gch" -j20
make CC="icc"
make CC="icc" (2nd)
make CC="icc" -j5
make CC="icc" -j20
make CC="distcc icc"
make CC="distcc icc" (2nd)
make CC="distcc icc" -j5
make CC="distcc icc" -j20
make CC="ccache distcc icc"
make CC="ccache distcc icc" (2nd)
make CC="ccache distcc icc" -j5
make CC="ccache distcc icc" -j20
make CC="ccache distcc icc -pch"
make CC="ccache distcc icc -pch" (2nd)
make CC="ccache distcc icc -pch" -j5
make CC="ccache distcc icc -pch" -j20
GNU C/C++
Intel C/C++

The (4+1)++ model


End-user
Functionality
Programmers
Software management
Build engineers
Developers
Deployers
Customizers
Logical View
Development View
Scenarios
Build-time View
Process View
Physical View
stakeholders
Integrators
Performance
Scalability
System engineers
Topology
Communications

Overview

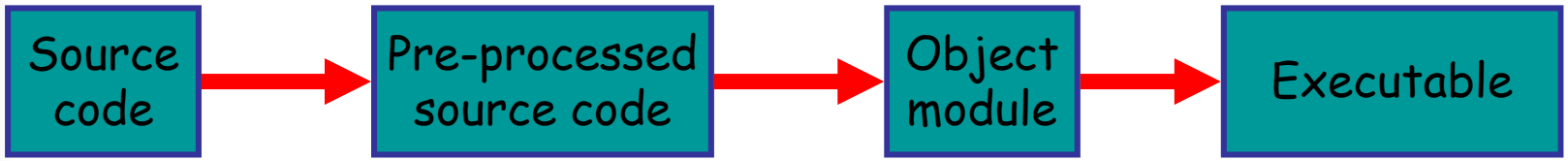
- The build / comprehend pipelines
  - Software architecture views
- The build-time software architecture view
  - What and why
  - Examples: **`GCC, Perl, JNI`**
  - The “code robot” architectural style
  - Representing build-time views in UML

The build / comprehend pipelines

- *"Use the source, Luke"*
  - Typical program comprehension tool:
    - based on static analysis of source code,
      [with maybe a little run-time profiling]
  - … but developers often use knowledge of the build process and other underlying technologies to encode aspects of a system's design.
    e.g., lookup ordering of libraries
    e.g., file boundaries and **`#include`** implement modules/imports
      - This info is lost/ignored during program understanding

The build / comprehend pipelines

- *The comprehension process should mimic the build process!*
  - So create tools that can interact with design artifacts at different stages of the build pipeline.
  - Create comprehension bridges/filters that can span stages.



Software architecture: What and why

- **What:**
  - Consists of descriptions of:
    - components, connectors, rationale/constraints, …
  - Shows *high-level structure*
    - Composition and decomposition, horizontal layers and vertical slices
  - Reflects *major design decisions*
    - Rationale for why one approach taken, what impact it has
- **Why:**
  - Promotes *shared mental model* among developers and other stakeholders

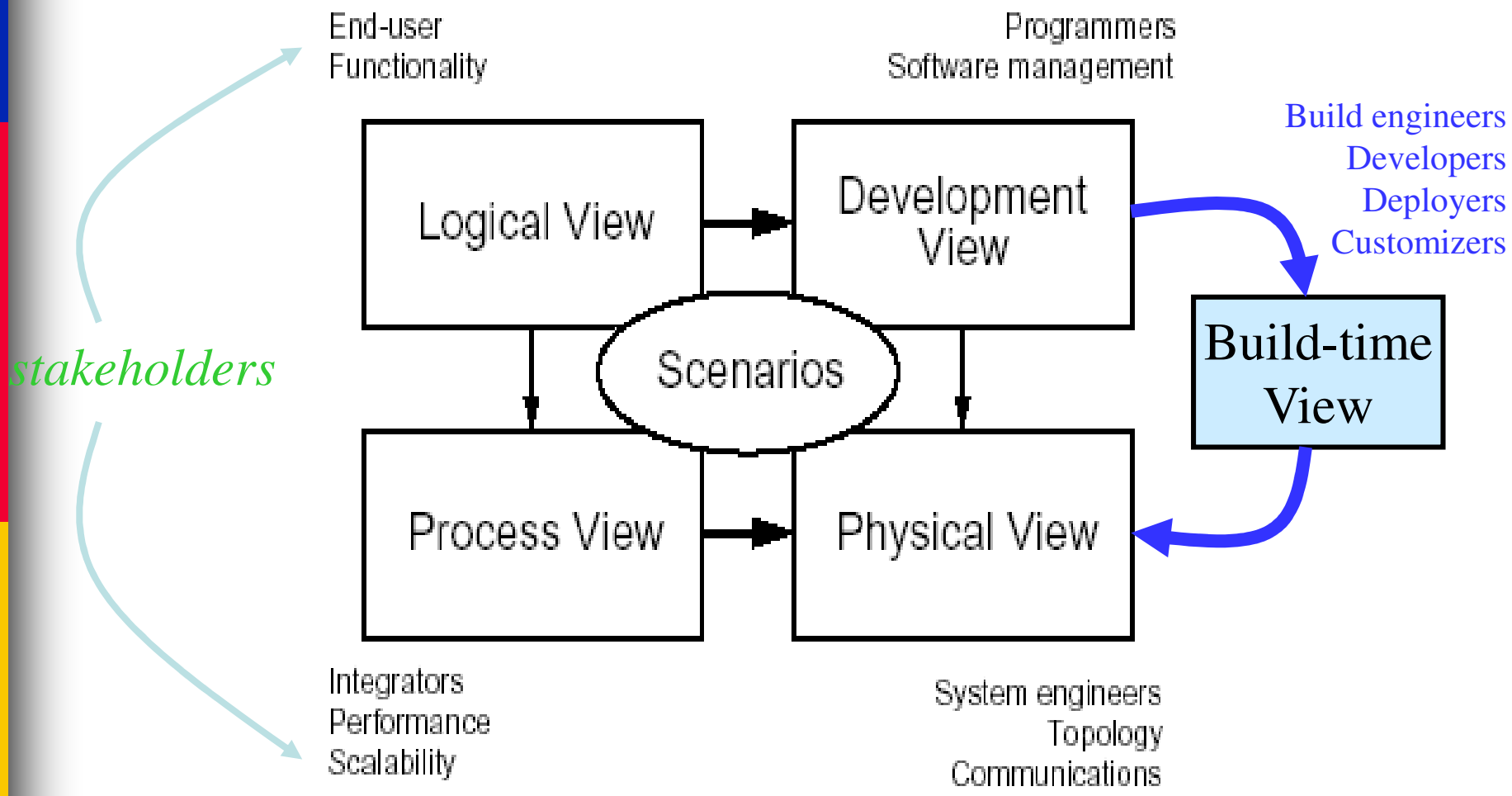
The need for multiple views

- Stakeholders have different experiences of what the system “looks like”
  - One size does not fit all.
  - “Separation of concerns”

- Kruchten’s “4+1” model:
  - Logical, development, process, physical “**+**” scenarios
  - Each view has different elements, different meaning for connectors, *etc.*

    [Hofmeister *et al.* proposed similar taxonomy of four views]

The (4+1)++ model


End-user
Functionality
Programmers
Software management
Build engineers
Developers
Deployers
Customizers
Logical View
Development View
Scenarios
Build-time View
Process View
Physical View
stakeholders
Integrators
Performance
Scalability
System engineers
Topology
Communications

Why the build-time view?

- Many systems do not have very interesting build-time properties …
  - Straightforward, mostly static **`Makefile`**-like approach is good enough.

- … but some systems do!
  - They exhibit interesting *structural* and *behavioural* properties that are apparent only at *system build time*.
  - Want to extract/reconstruct/document interesting build properties to aid program comprehension.

Why the build-time view (BTV)?

- Want to document interesting build processes to aid program comprehension
- Targeted at different stakeholders: anyone affected by the build process
  - System “build engineers”
  - Software developers
  - End-users who need to build or customize the application
- Separation of concerns
  - Configuration/build management
- Of particular interest to open source projects
  - “built-to-be-built” (*./configure*)

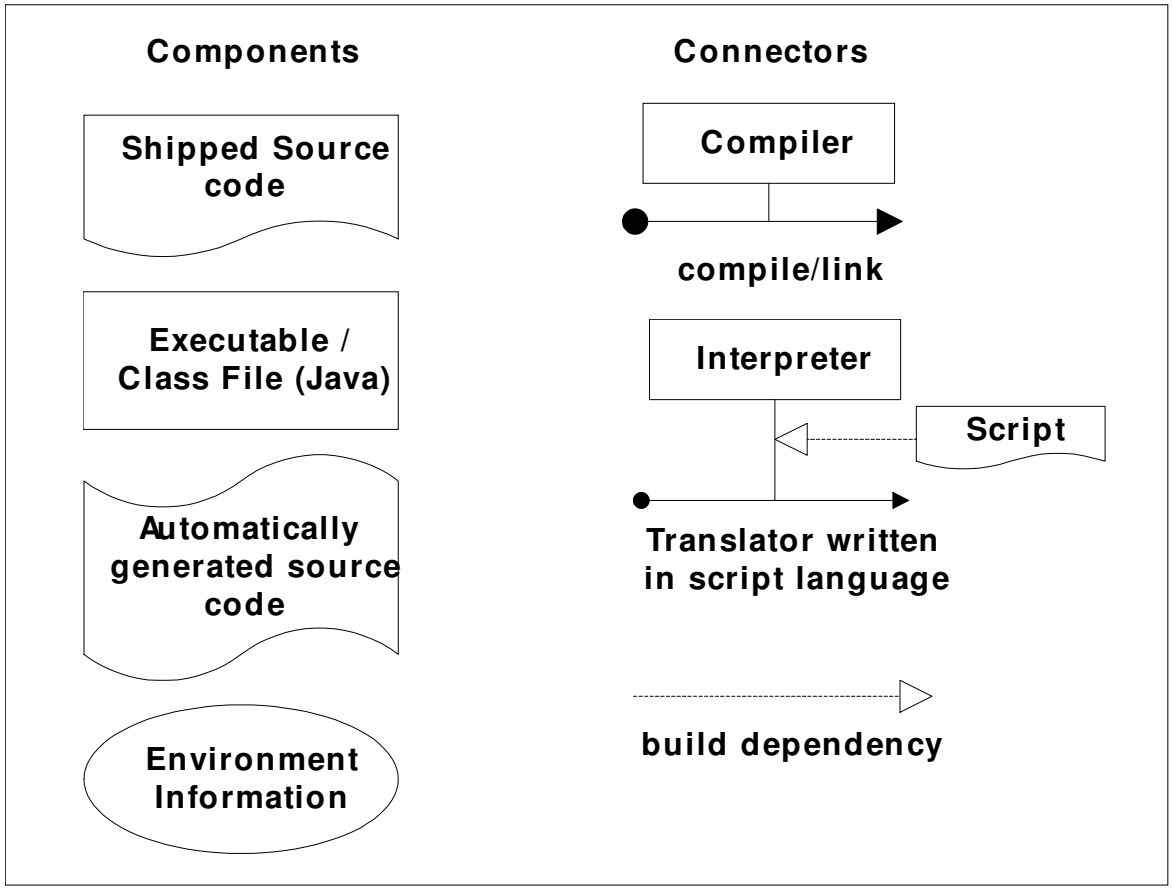
Some interesting build-time activities

- Automatic “source” code generation
  - Build-time vs. development-time (*e.g.,* GCC *vs.* JDK/JNI)
  - Targeted at a large range of CPU/OS platforms
    - Implementation (algorithms) are highly platform dependent.
    - Conditional compilation is not viable.
- Bootstrapping
  - Cross-platform compilation
  - Generation of VMs/interpreters for “special languages”
- Build-time component installation
- Runtime library optimization
  - *e.g.,* VIM text editor
- **...**

Common reasons for interesting build-time activities

- System building is simply a complex process
  - A “software system” is more than a set of source code files
- Software aging
  - Older systems gather cruft which is most easily dealt with by build-time hacks
  - Native source language no longer widely supported
  - Ports to new environments dealt with at build-time
- Complex environmental dependencies which must be resolved by querying the target platform
  - Especially true for open source software *(“built-to-be-built”)*
  - Common for compiler-like applications

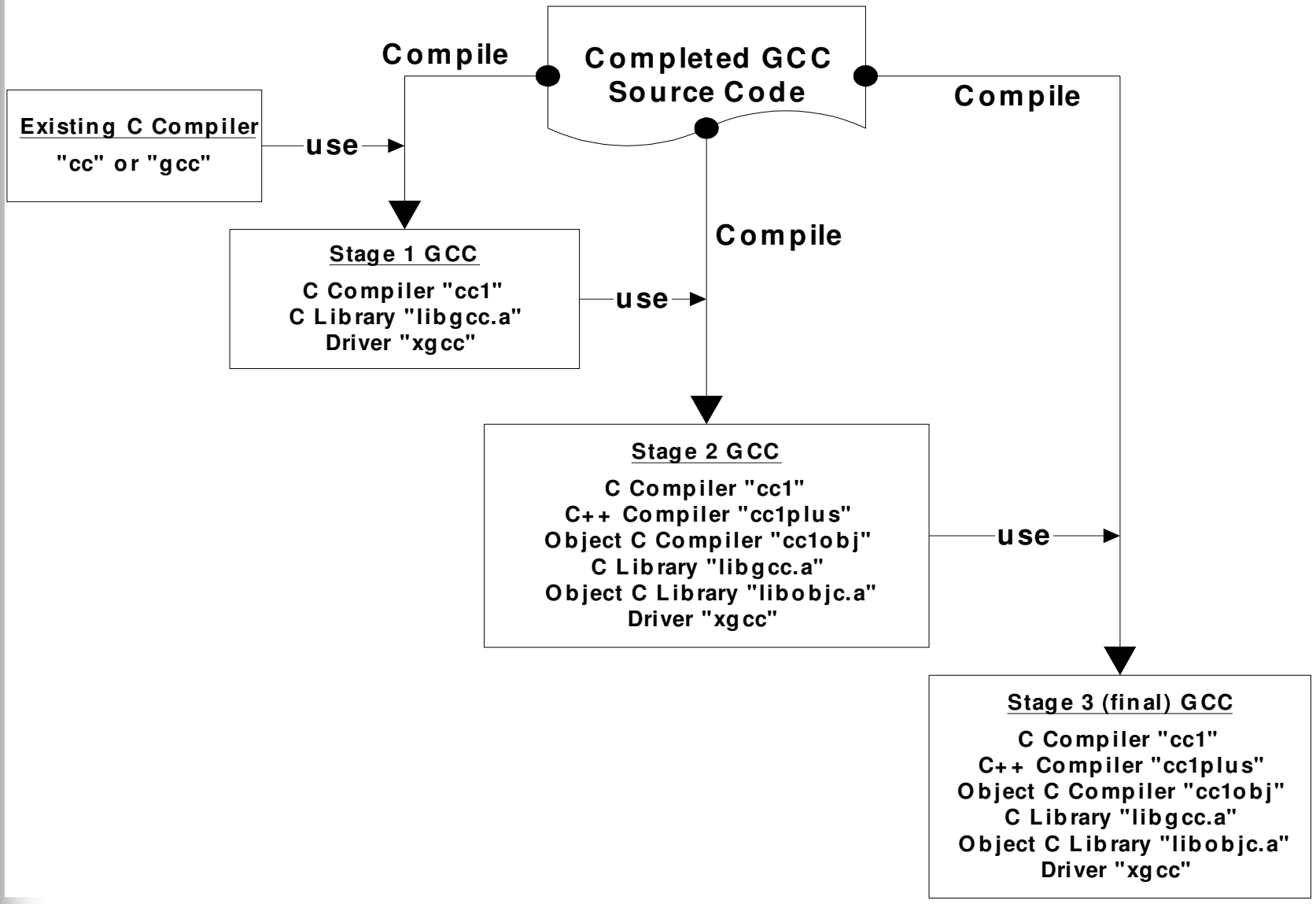
Build-time view schema



Example 1: GCC bootstrapping

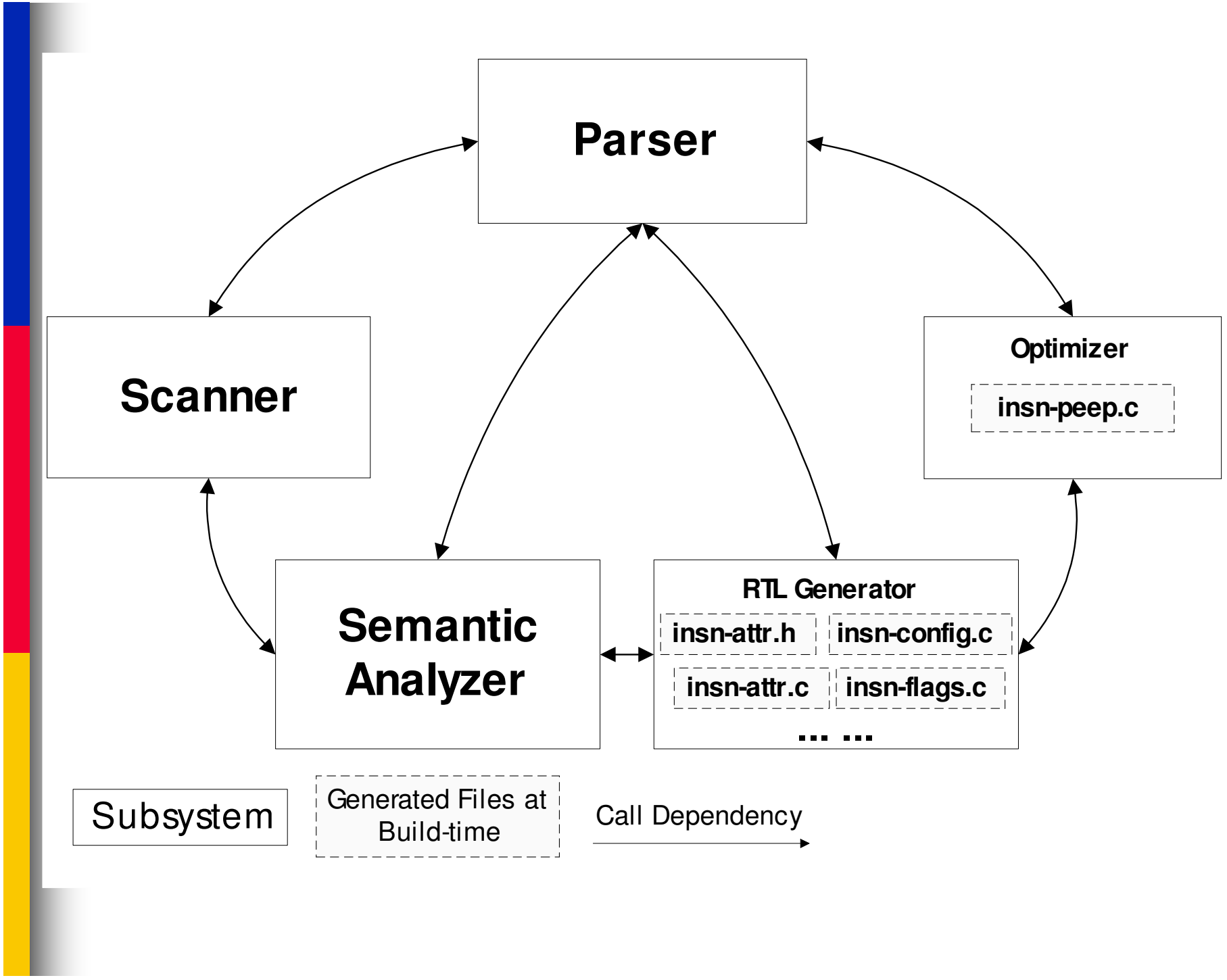
- Same source code is compiled multiple times
  - Each time by a different compiler!
    - Usually, the one built during the previous iteration.
  - Different source modules are **`include`**d and configured differently for some iterations

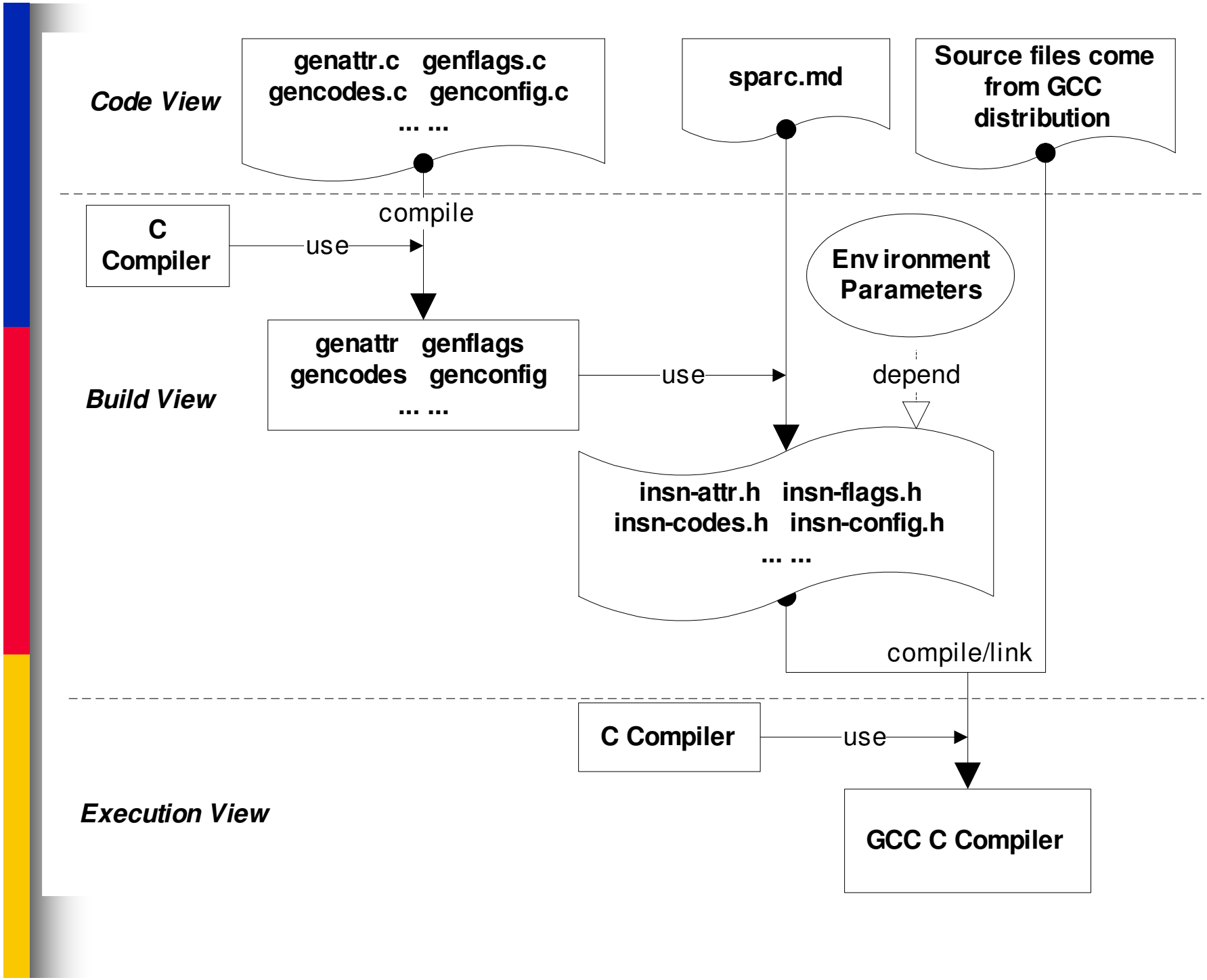
- Static analysis (reading) of the **`Makefile`**s doesn't help much in understanding what's going on.
  - **`Makefile`**s are templated, control flow depends on complex interactions with environment.
  - Need to instrument and trace executions of build process, build visual models for comprehension


Compile
Completed GCC
Source Code
Compile
Existing C Compiler
"cc" or "gcc"
use
Stage 1 GCC
C Compiler "cc1"
C Library "libgcc.a"
Driver "xgcc"
use
Compile
Stage 2 GCC
C Compiler "cc1"
C++ Compiler "cc1plus"
Object C Compiler "cc1obj"
C Library "libgcc.a"
Object C Library "libobjc.a"
Driver "xgcc"
use
Stage 3 (final) GCC
C Compiler "cc1"
C++ Compiler "cc1plus"
Object C Compiler "cc1obj"
C Library "libgcc.a"
Object C Library "libobjc.a"
Driver "xgcc"

Example 2: GCC build-time code generation

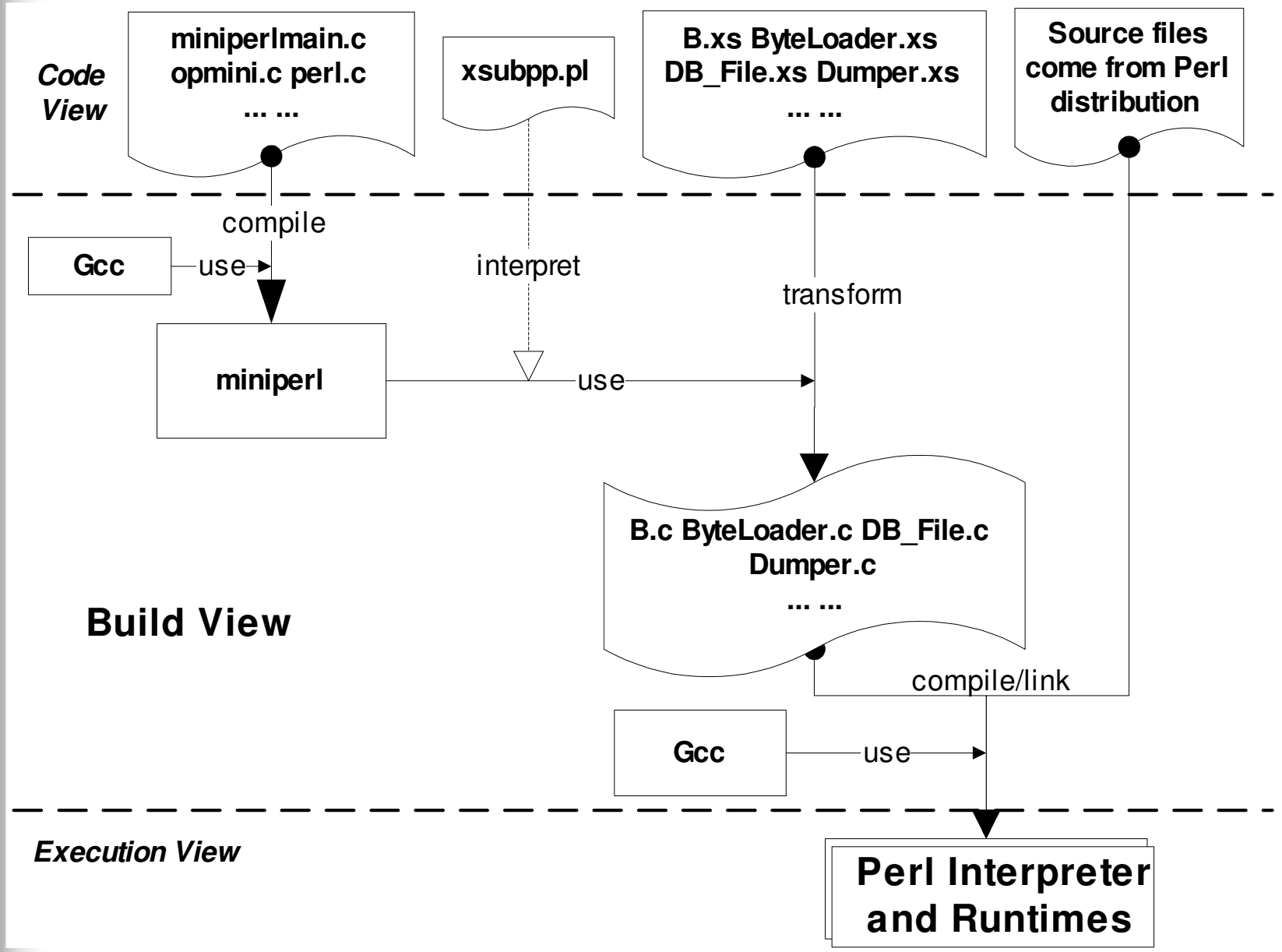
- In GCC, the common intermediate representation language (*i.e.,* post-parsing) is called the Register Transfer Language (RTL)
  - The RTL is hardware dependent!
  - Therefore, the code that generates and transforms RTL is also hardware dependent.
- RTL related code is generated at build-time
  - Information about the target environment is input as build parameters.


Parser
Scanner
Optimizer
insn-peep.c
Semantic Analyzer
RTL Generator
insn-attr.h
insn-config.c
insn-attr.c
insn-flags.c
... ...
Subsystem
Generated Files at Build-time
Call Dependency


Code View
genattr.c genflags.c
gencodes.c genconfig.c
... ...
sparc.md
Source files come from GCC distribution
compile
C Compiler
use
Environment Parameters
genattr genflags
gencodes genconfig
... ...
Build View
use
depend
insn-attr.h insn-flags.h
insn-codes.h insn-config.h
... ...
compile/link
C Compiler
use
Execution View
GCC C Compiler

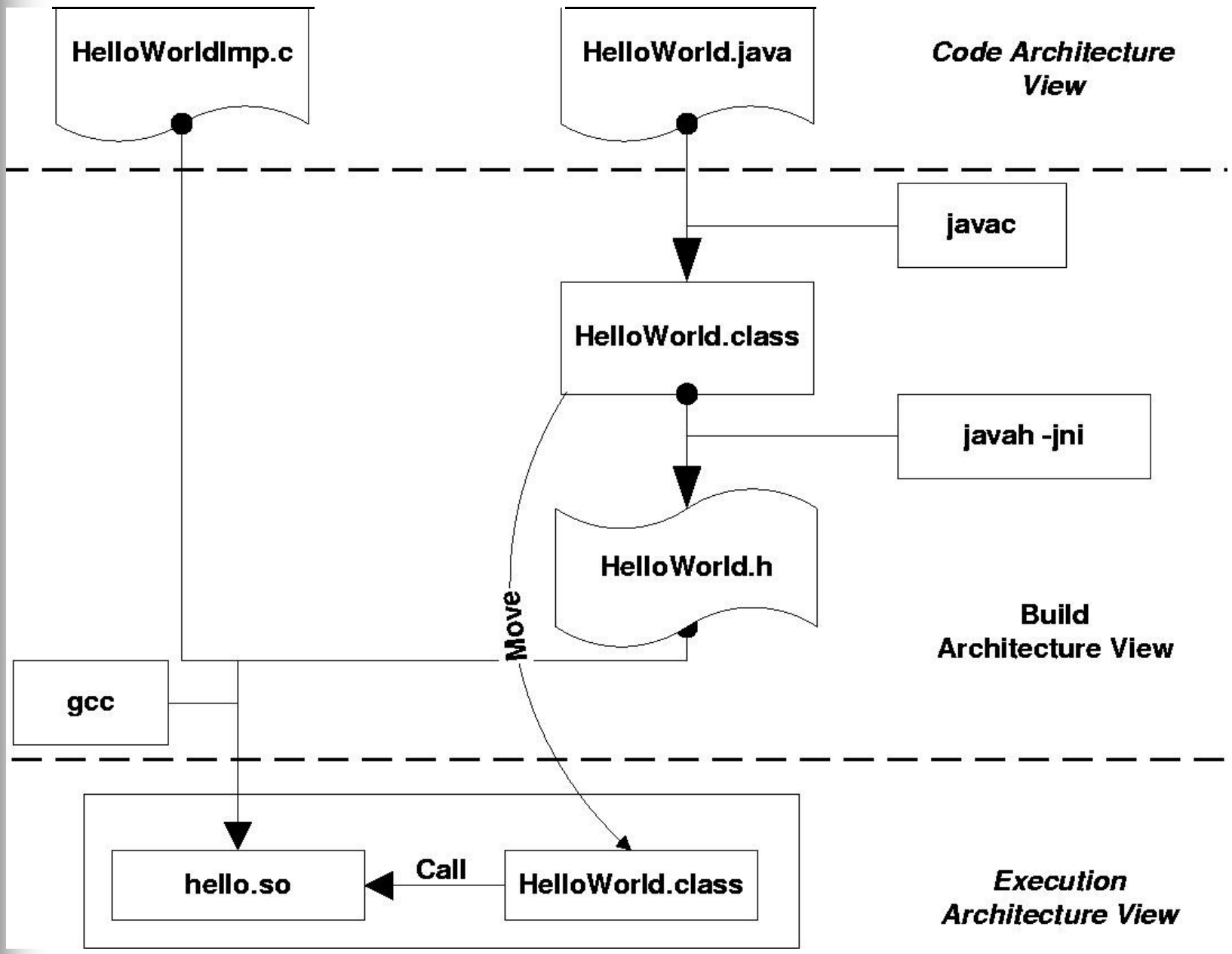
Example 3: PERL building procedures

- PERL build process exhibits both *bootstrapping* and *build-time code generation*.
  - The PERL build process is so complex that is an open source project in its own right!
- Templates written in XS language are transformed at build-time to generate C files that bridge PERL runtime with Unix runtime libraries.
  - These C files are OS dependent.


Code View
miniperlmain.c
opmini.c perl.c
... ...
xsubpp.pl
B.xs ByteLoader.xs
DB_File.xs Dumper.xs
... ...
Source files
come from Perl
distribution
compile
Gcc
use
interpret
transform
miniperl
use
B.c ByteLoader.c DB_File.c
Dumper.c
... ...
Build View
compile/link
Gcc
use
Execution View
Perl Interpreter
and Runtimes

Example 4: Use of Java Native Interface (JNI)

- May want your Java program to make use of an existing C/C++ program for performance or other reasons.
- Need to go through several steps to customize the interaction between the two systems.
  - Similar to Perl XS mechanism, but done for each Java application that requires access to “native” code


HelloWorldImp.c
HelloWorld.java
Code Architecture View
javac
HelloWorld.class
javah -jni
HelloWorld.h
Move
Build Architecture View
gcc
hello.so
Call
HelloWorld.class
Execution Architecture View

“Code Robot” architecture style

- An *architectural style* is a recurring abstract pattern of high-level software system structure [Shaw/Garlan]

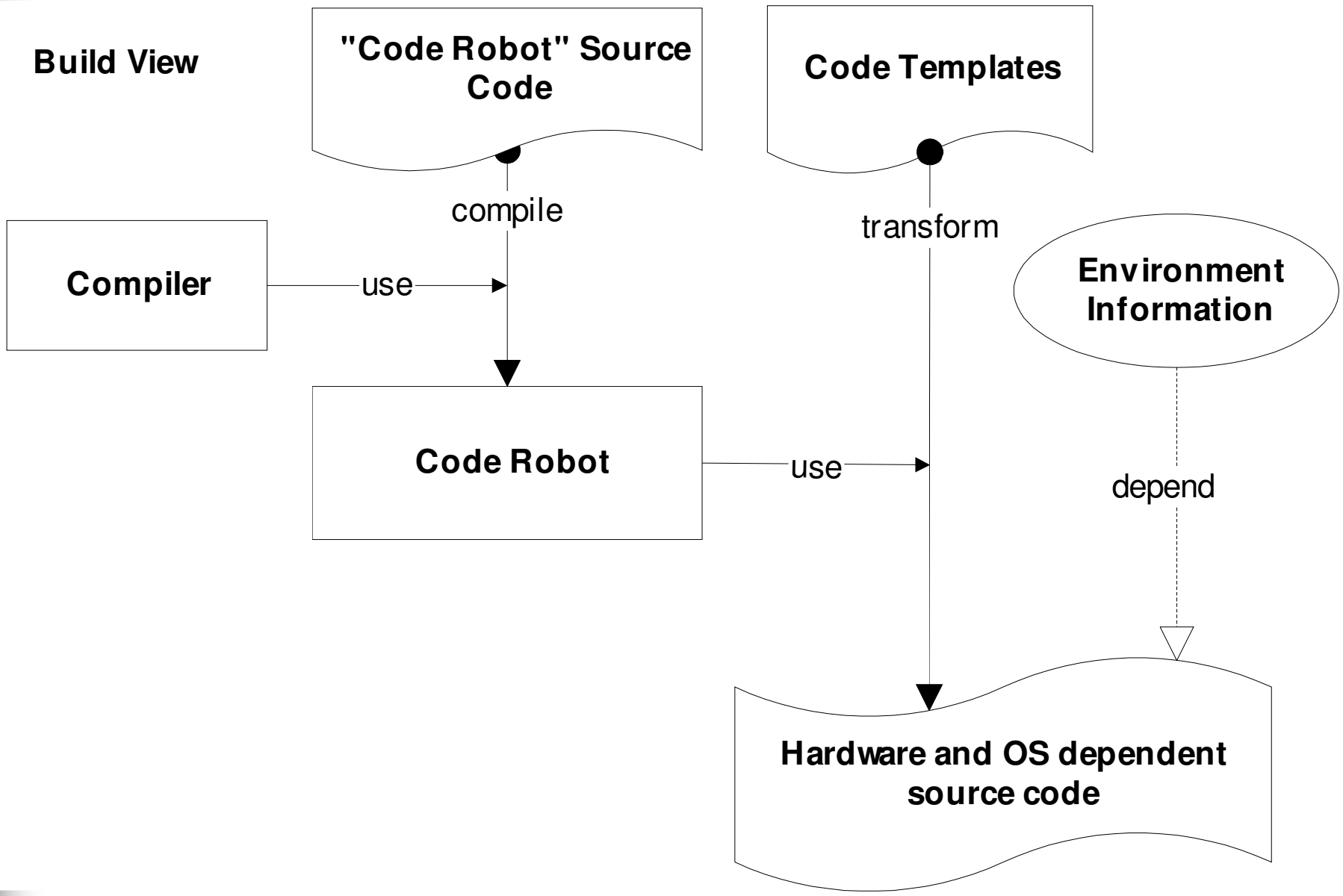
“Code Robot”

Problem: – desired behavior of software depends heavily on hardware platform or operating systems.

Solution: – create customized “source” code at build-time using auto code generator, code templates, other environment-specific customizations.

Examples – some open source systems (*e.g.,* GCC, PERL)

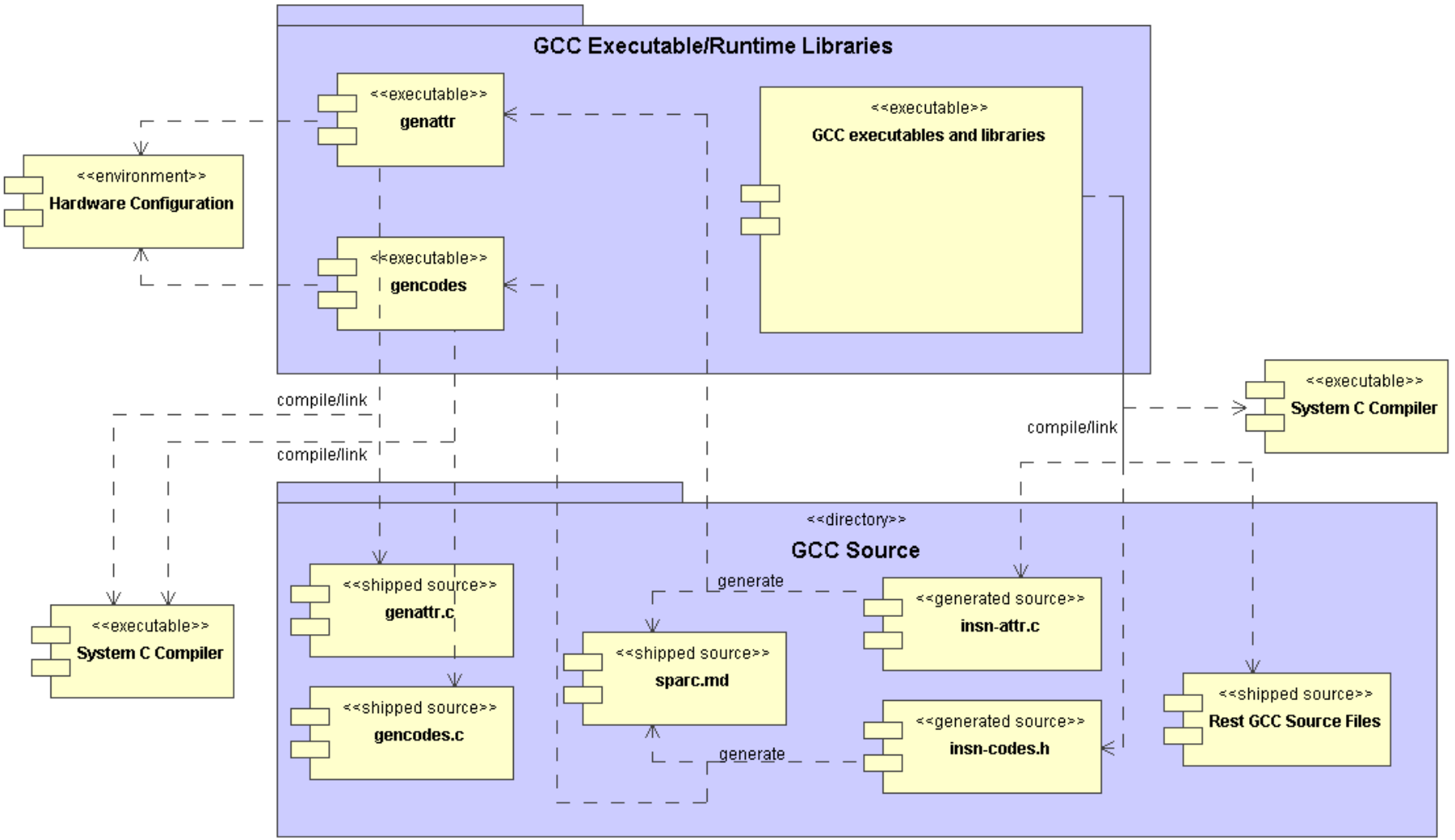
Build View


"Code Robot" Source Code
Code Templates
compile
transform
Compiler
use
Environment Information
Code Robot
use
depend
Hardware and OS dependent source code

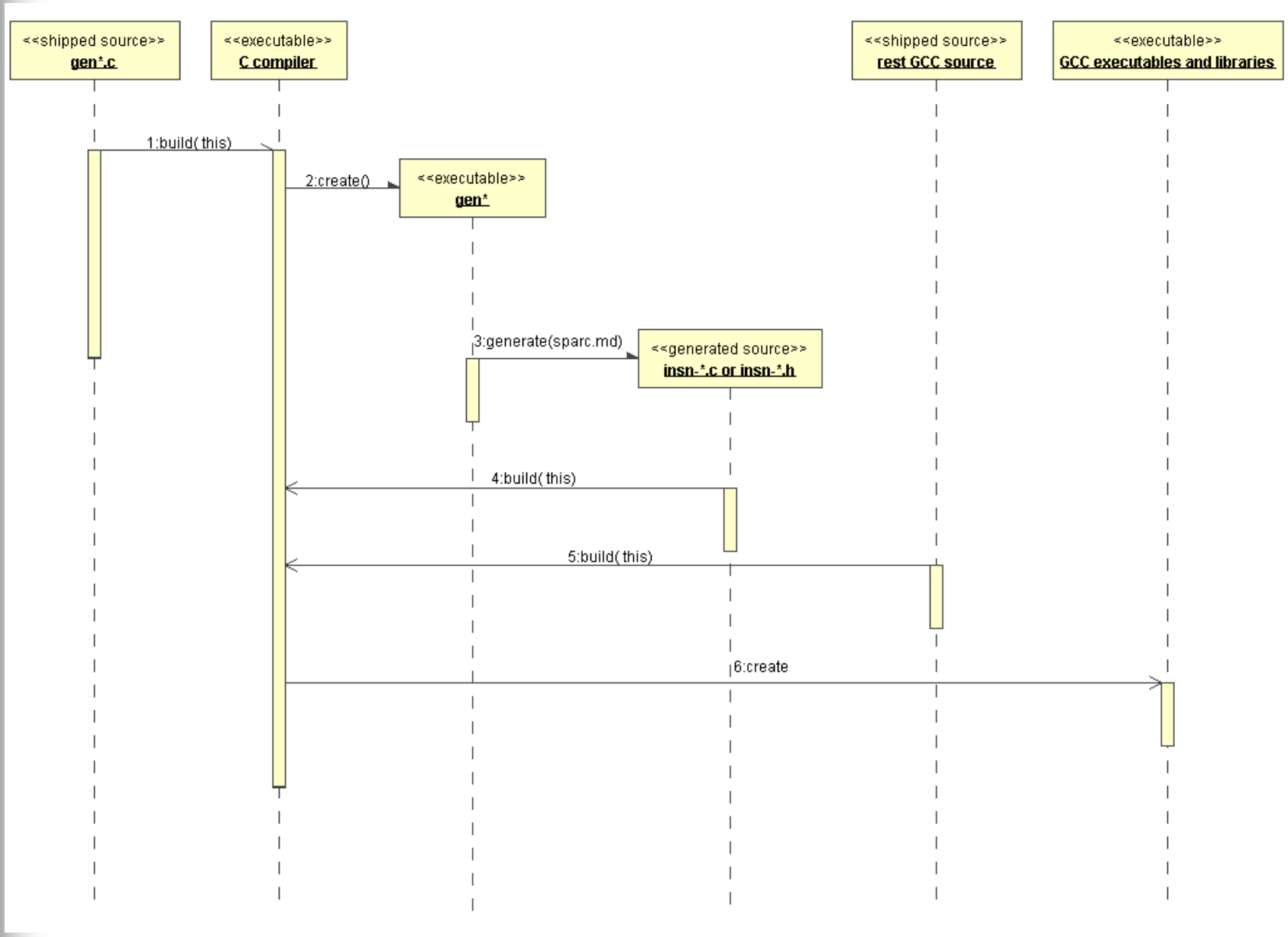
UML Representation

- Static View (UML Component Diagram)
  - Components:
    - Code written at development phase
    - Code generated at build time
    - Library and executables
    - Environment information
  - Relations:
    - Compile/Link
    - Generate
- Dynamic View (UML Sequence Diagram)
  - Model dynamic build procedures

Static UML View

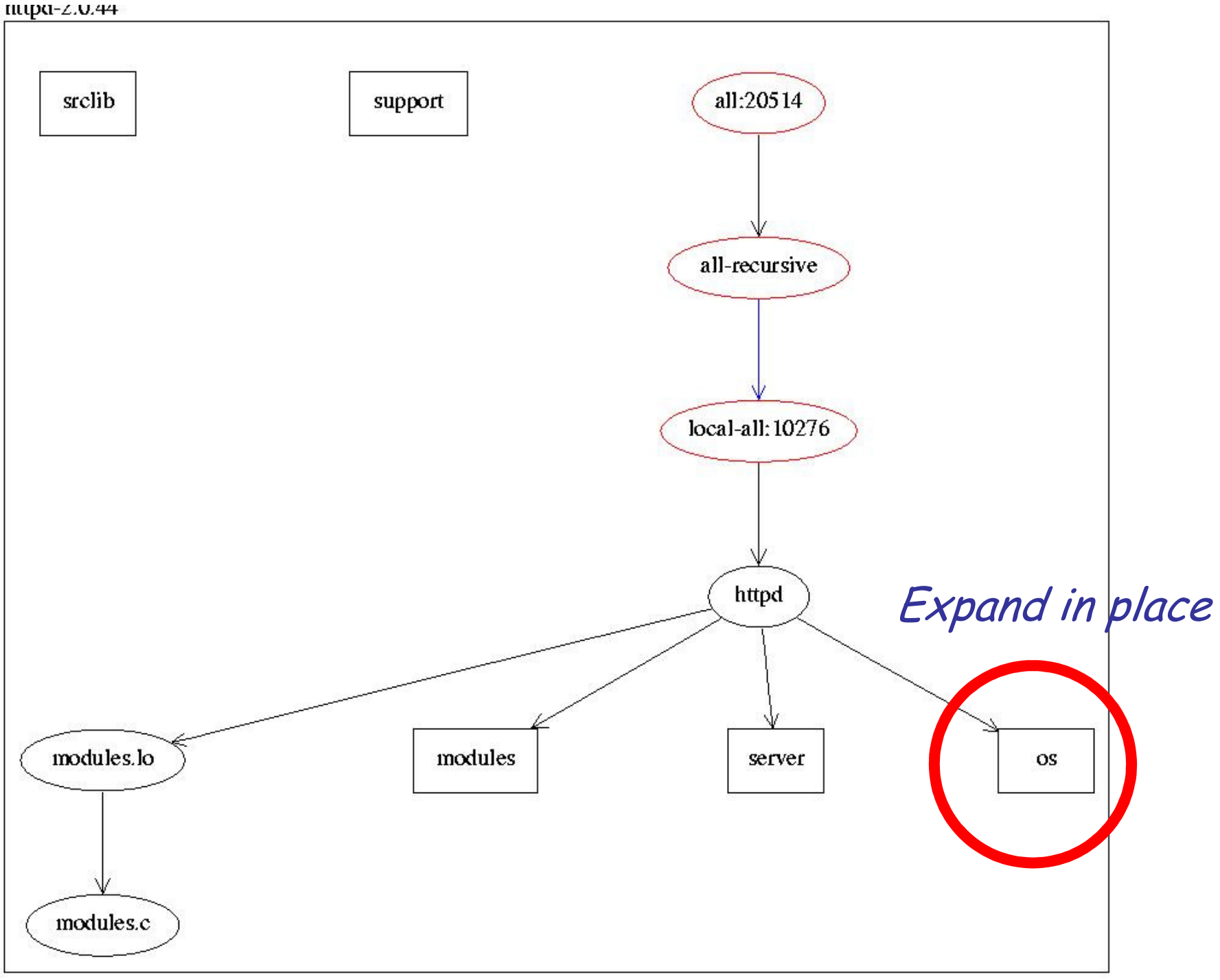


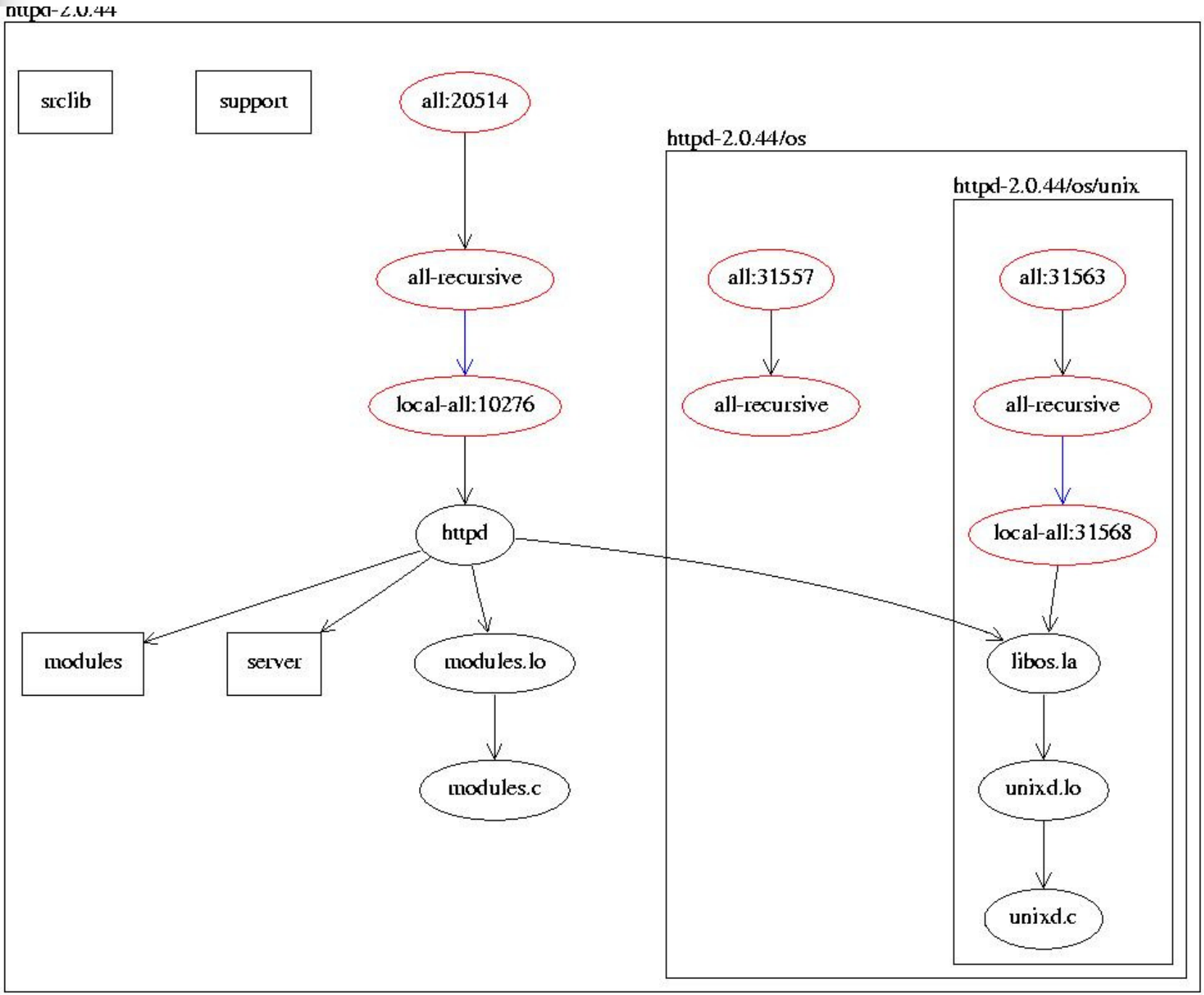
Dynamic UML View

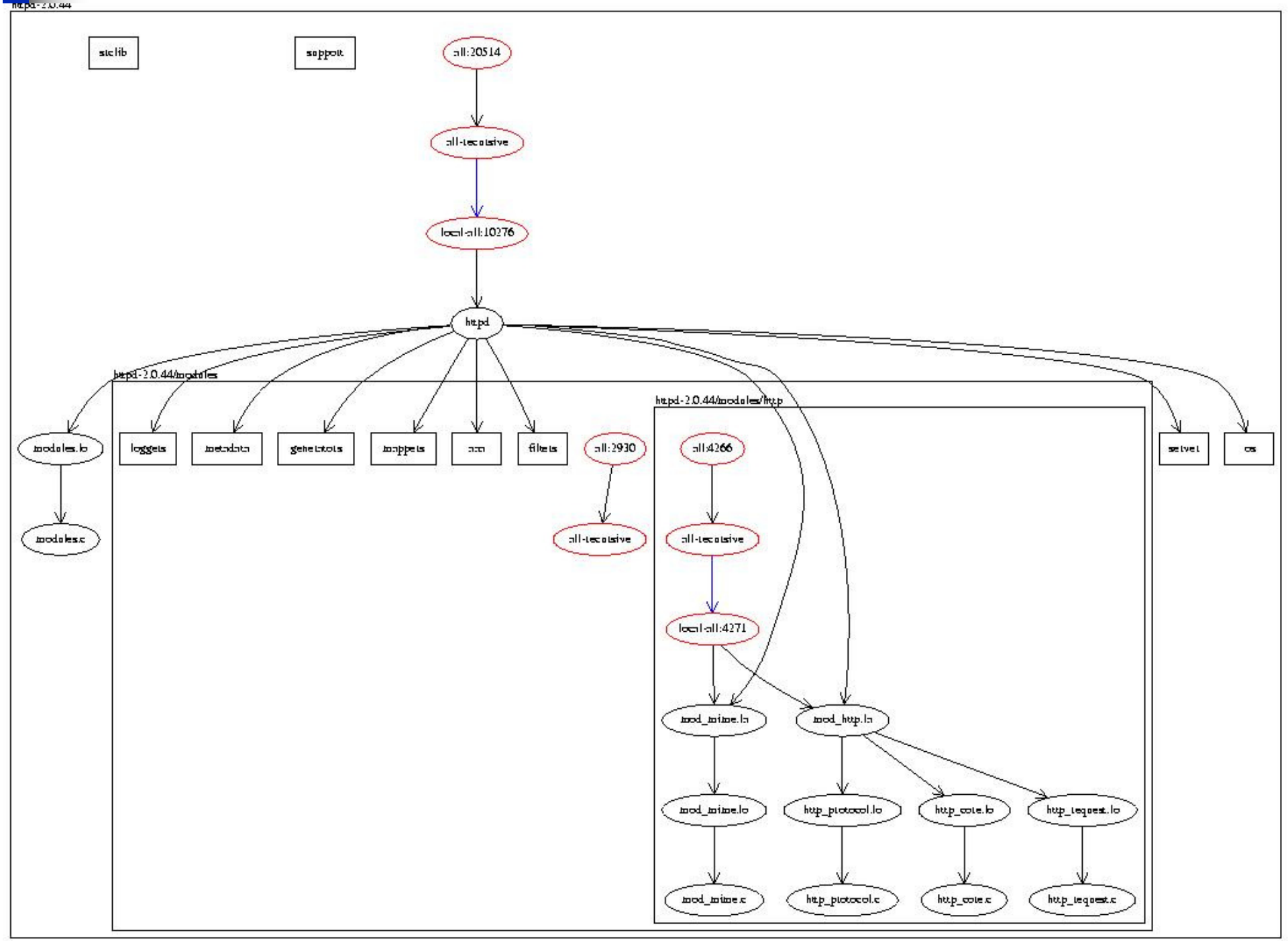


BTV toolkit

- Idea:
  - Record all: **[gmake]**
    - **make** target/subtarget dependencies
      - *shows make deps, not compilation deps*
    - directory locations of targets/files
    - build command actions
  - Resolve common targets to one node **[grok]**
  - Visualization / navigation **[graphviz]**


srclib
support
all:20514
all-recursive
local-all:10276
httpd
modules.lo
modules
server
os
modules.c
Expand in place


httpd-2.0.44
srclib
support
all:20514
all-recursive
local-all:10276
httpd
modules
server
modules.lo
modules.c
httpd-2.0.44/os
all:31557
all-recursive
httpd-2.0.44/os/unix
all:31563
all-recursive
local-all:31568
libos.la
unixd.lo
unixd.c


httpd-2.0.44
srclib
support
all:20514
all-recursive
local-all:10276
httpd
httpd-2.0.44/modules
httpd-2.0.44/modules/http
modules.lo
loggers
metadata
generators
mappers
aaa
filters
all:2930
servlet
os
modules.c
all-recursive
all:4266
all-recursive
local-all:4271
mod_mime.la
mod_http.la
mod_mime.lo
http_protocol.lo
http_core.lo
http_request.lo
mod_mime.c
http_protocol.c
http_core.c
http_request.c

BTV toolkit

- Future work:
  - Timeline info (sequence charts?)
  - Querying
  - Improved navigation
  - Model files that aren't explicit targets [hard]
  - Model effects of actions / scripts [hard]

Summary

- Build-time view captures interesting structural and behavioral properties of some classes of software.
  - Modelling BTV is essential to understanding a system's design
- "Code robot" architectural style
  - Common in systems with interesting BTVs
- *BTV toolkit* can help to explore systems that use **`make`**
- Future work:
  - More case studies and exploration of problem space
    - Discover recurring patterns of build-time activities
  - (More) tools to extract and navigate build-time views