Modelling and Extracting the Build-Time Architectural View

University of Waterloo

Michael W. Godfrey

Software Architecture Group (SWAG)

migod@uwaterloo.ca

Joint work with …

- Grad students
  - Qiang Tu
  - Xinyi Dong
- Faculty colleagues
  - Ric Holt
  - Andrew Malton
- Also see:
  - "The Build-Time Software Architecture View",
    - *Proc. of ICSM 2001*
  - The BTV Toolkit
    - **http://www.swag.uwaterloo.ca/~xdong/btv/**

Overview

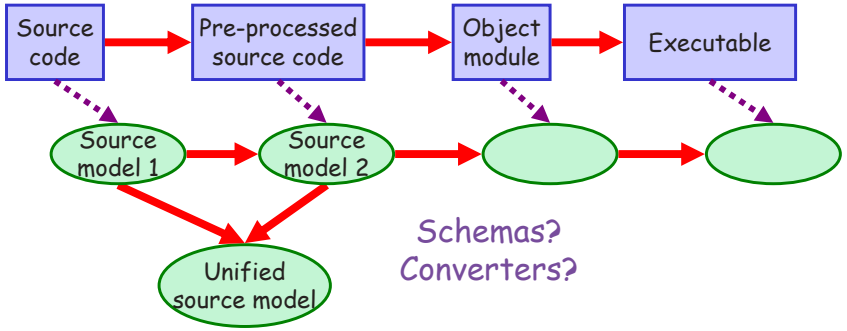
- The build / comprehend pipelines
  - Software architecture views
- The build-time software architecture view
  - What and why
  - Examples: **`GCC, Perl, JNI`**
  - The "code robot" architectural style
  - Representing build-time views in UML
- Demo of the *BTV toolkit*

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 by most fact extractors

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

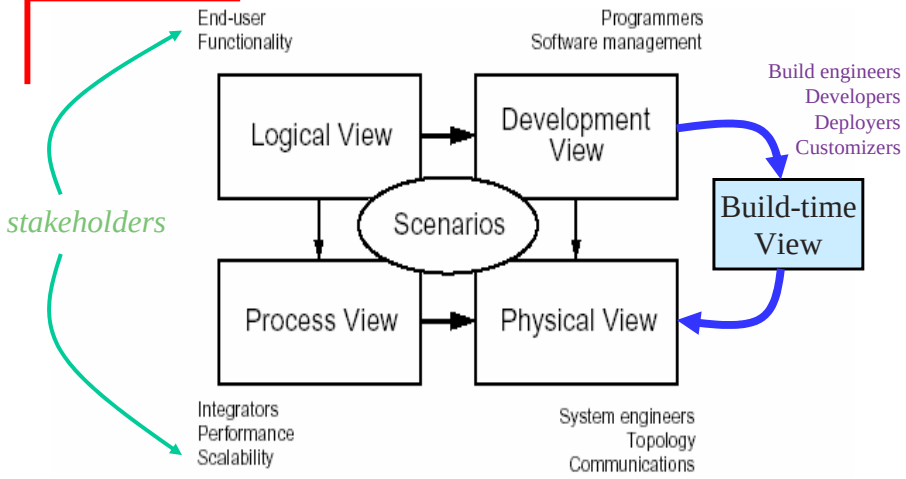
- **What:**
  - Consists of descriptions of:
    - components, connectors, rationale/constraints, …
  - Shows *high-level structure*
    - Composition and decomposition, horizontal layers and vertical slices
  - Reflects *major design decisions*
    - Rationale for why one approach taken, what impact 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



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"

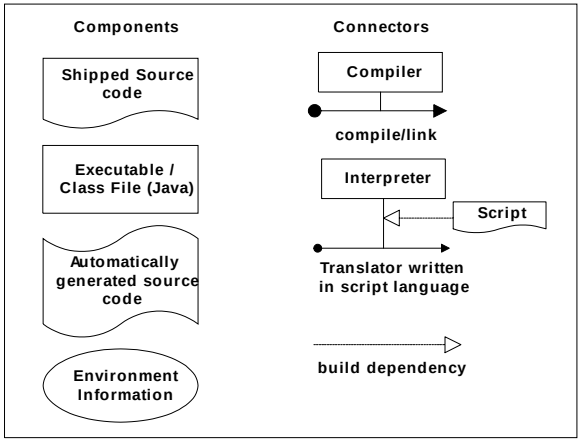
Some interesting build-time activities

- 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

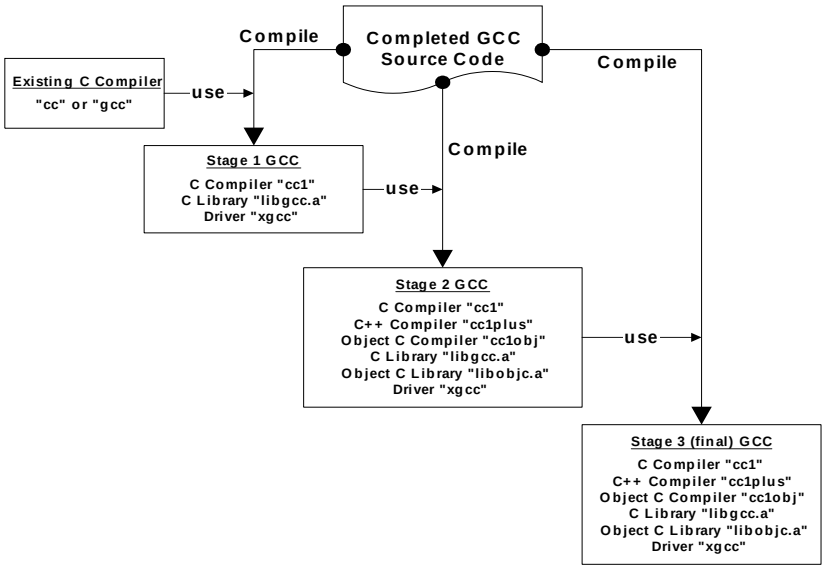
- System building is simply a complex process
  - A "software system" is more than a set of source code files
- Software aging
  - Older systems gather cruft which is most easily dealt with by build-time hacks
  - Native source language no longer widely supported
  - Ports to new environments dealt with 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



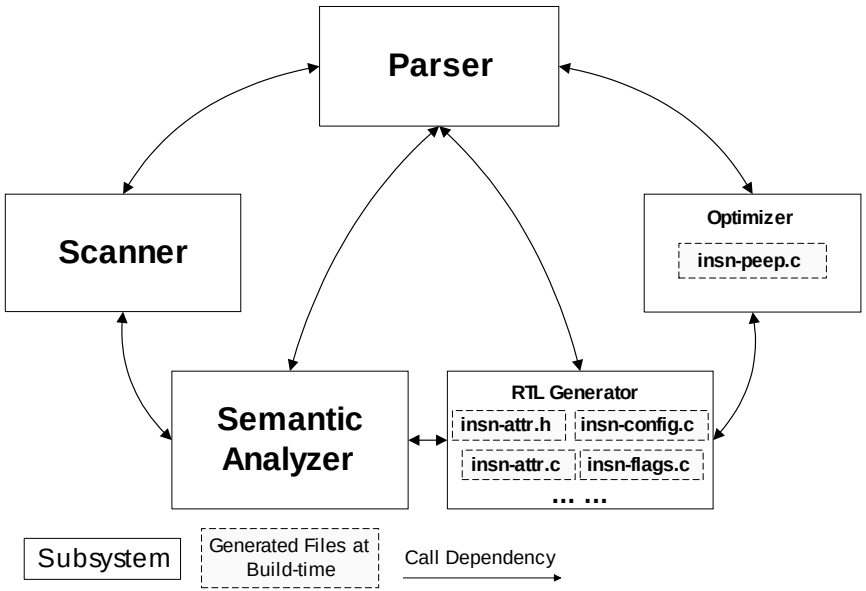
Example 1: GCC bootstrapping

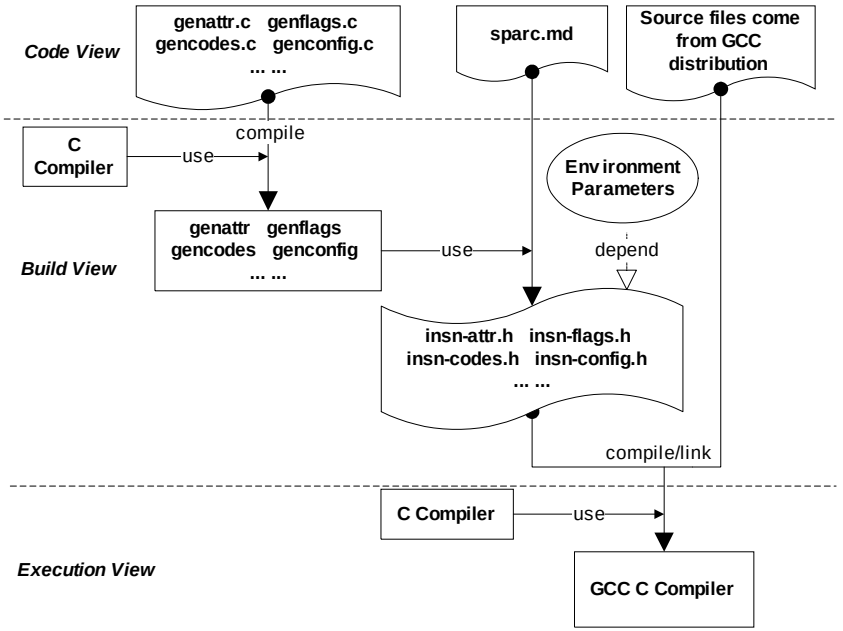
- Same source code is compiled multiple times
  - Each time by a different compiler!
    - Usually, the one built during the previous iteration.
  - Different source modules are **`include`**d and configured differently for some iterations
- Static analysis (reading) of the **`Makefile`**s doesn't help 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



Example 2: GCC build-time code generation

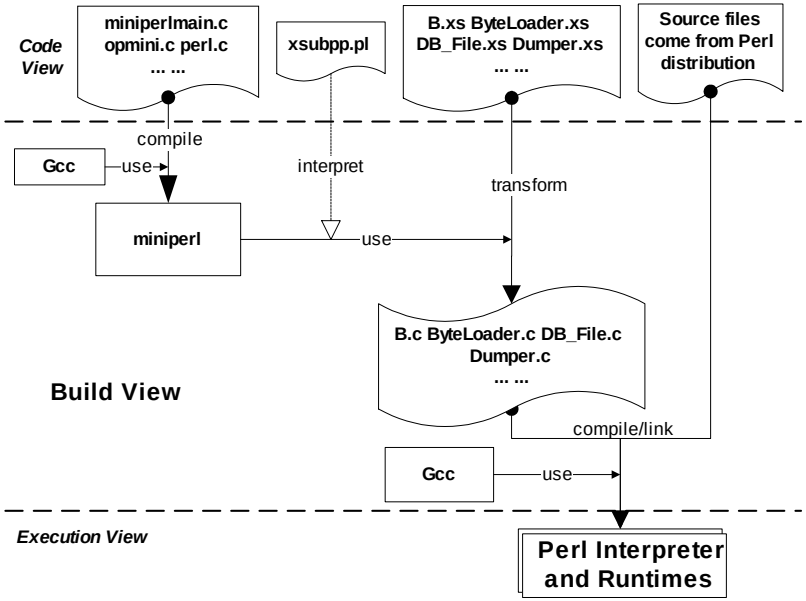
- In GCC, the common intermediate representation language (*i.e.*, post-parsing) is called the Register Transfer Language (RTL)
  - The RTL is hardware dependent!
  - Therefore, the code that generates and transforms RTL is also hardware dependent.
- RTL related code is generated at build-time
  - Information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





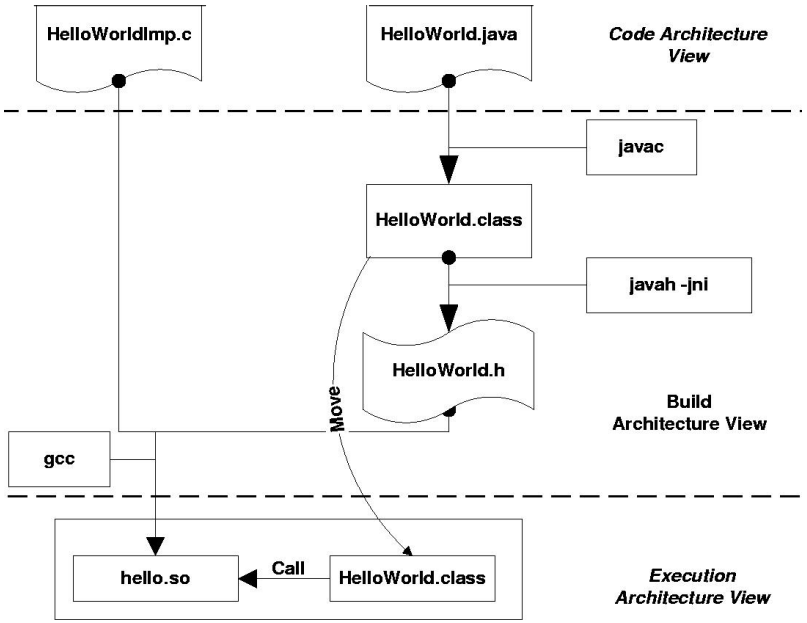
Example 3: PERL building procedures

- 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.



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



"Code Robot" architecture style

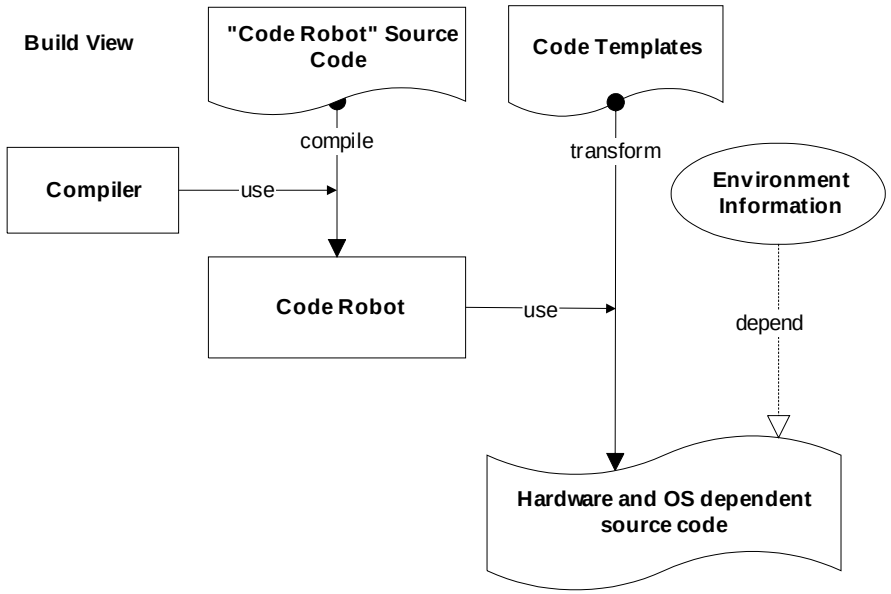
- An *architectural style* is a recurring abstract pattern of high-level software system structure [Shaw/Garlan]

"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.

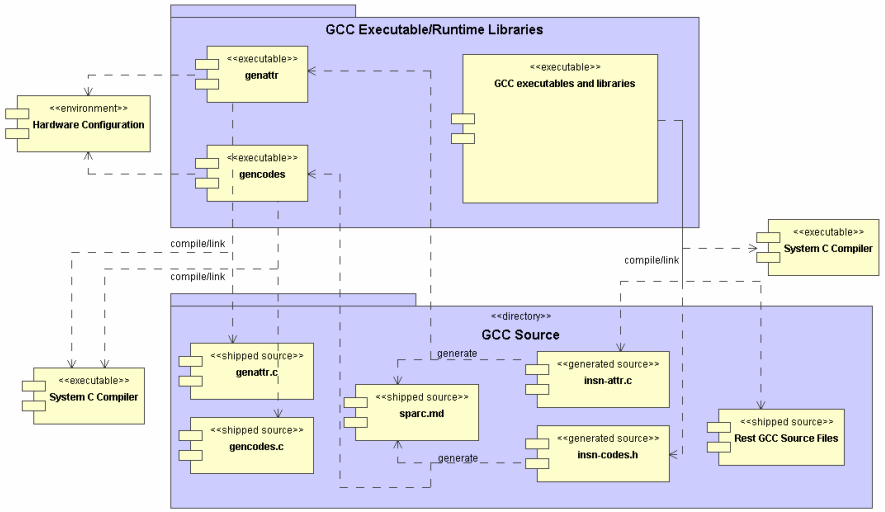
Examples – some open source systems (*e.g.*, GCC, PERL)



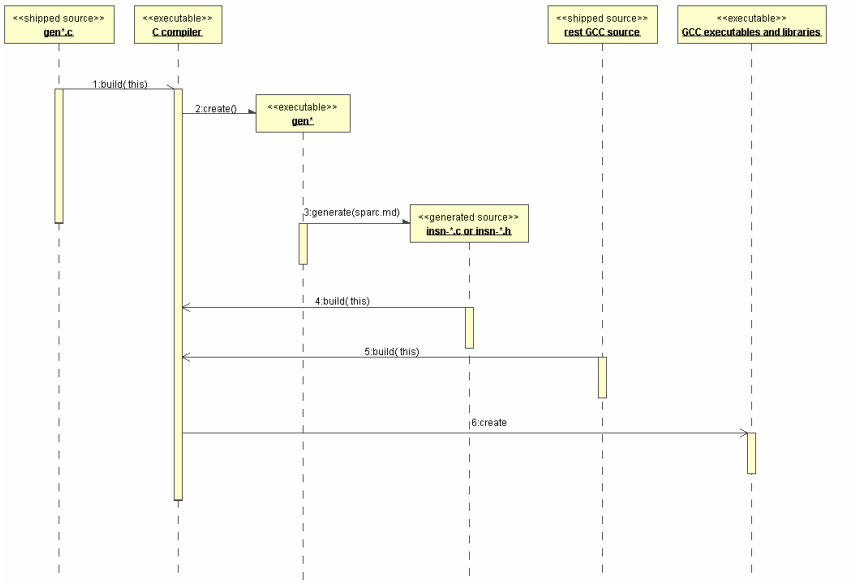
UML Representation

- 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

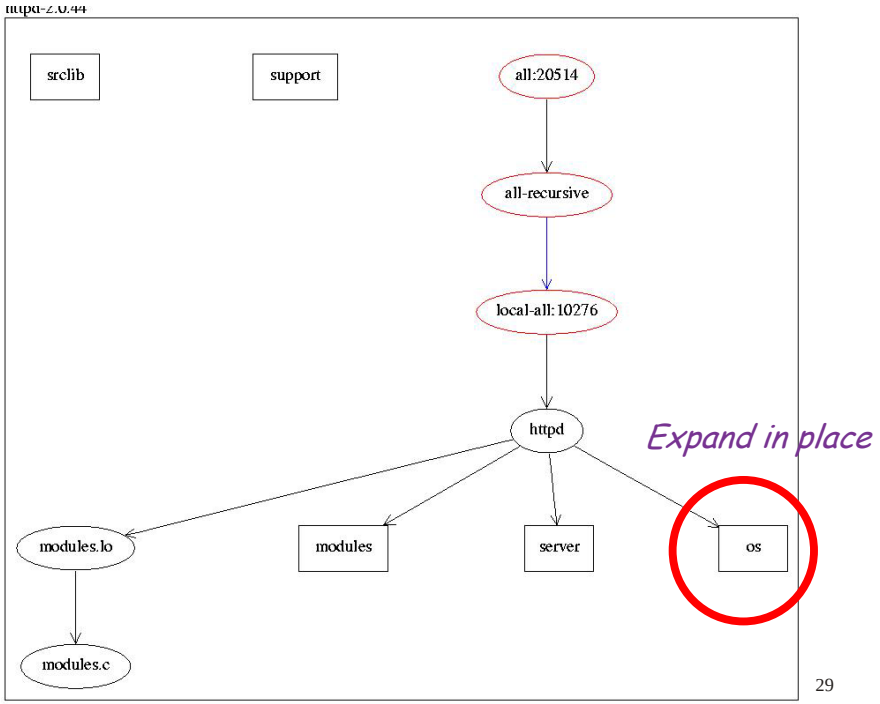


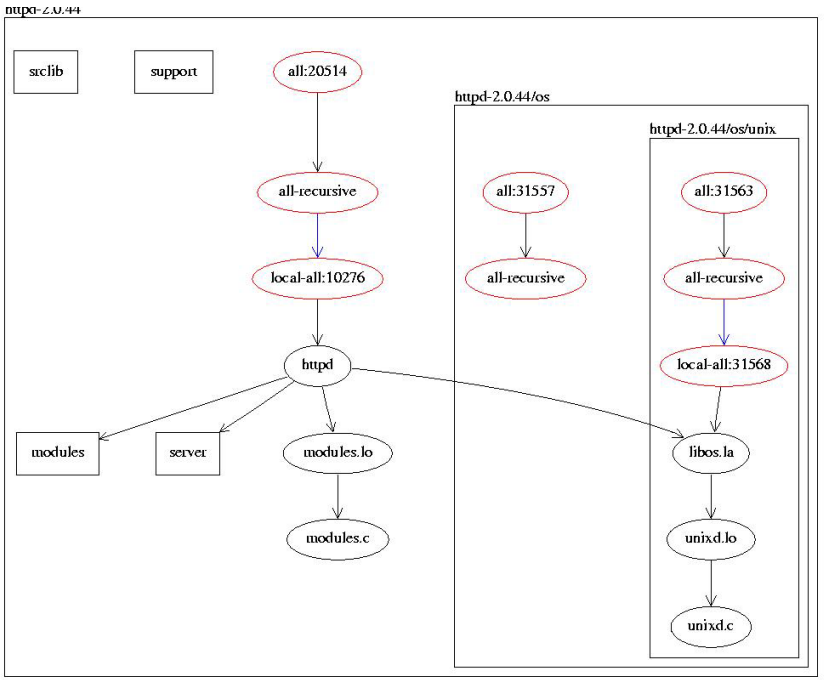
Dynamic UML View

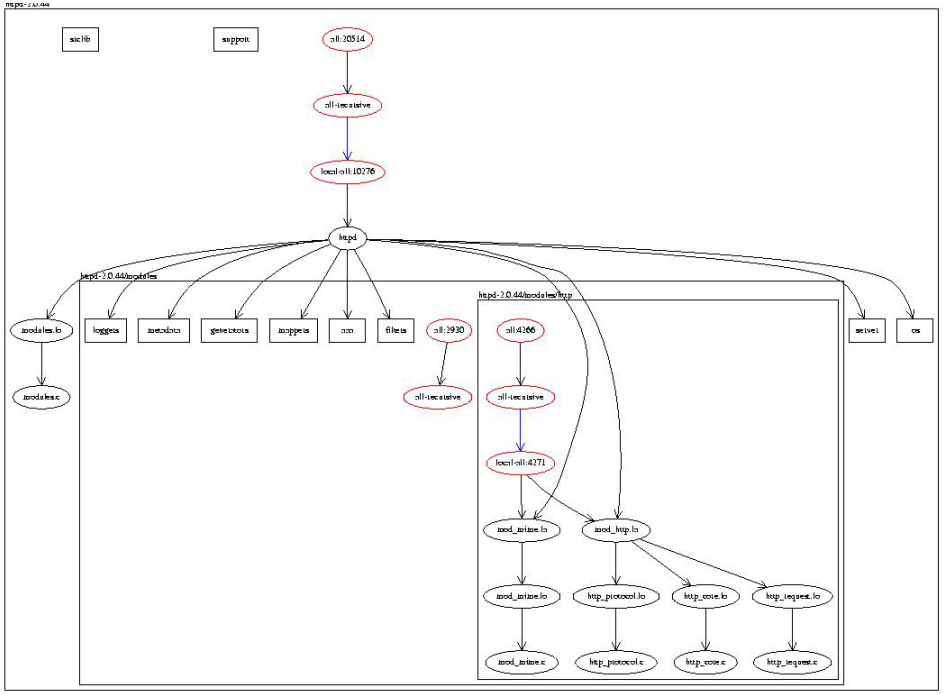


BTV toolkit

- Work of Xinyi Dong; early prototype available from
  http://www.swag.uwaterloo.ca/~xdong/btv/
- 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]`







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 such 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