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

- Development releases (1.1, 1.3, 2.1, 2.3)
- Stable releases (1.0, 1.2, 2.0, 2.2)
The (4+1)++ model

- Build-time View
- Stakeholders
- Logical View
- Development View
- Process View
- Physical View
- Scenarios

End-user Functionality
Programmers Software management
Integrators Performance Scalability
System engineers Topology Communications

Build engineers Developers Deployers Customizers
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
- Integrators
  - Performance
  - Scalability
- Programmers
  - Software management
- System engineers
  - Topology
  - Communications

Stakeholders:
- Developers
- Deployers
- Customizers
- Build engineers

Views:
- Logical View
- Development View
- Process View
- Physical View
- Build-time View

Scenarios
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

Components

- Shipped Source code
- Executable / Class File (Java)
- Automatically generated source code
- Environment Information

Connectors

- Compiler
  - compile/link
- Interpreter
  - Translator written in script language
  - build dependency
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 included and configured differently for some iterations

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

Stage 1 GCC
C Compiler "cc1"
C Library "libgcc.a"
Driver "xgcc"

Stage 2 GCC
C Compiler "cc1"
C++ Compiler "cc1plus"
Object C Compiler "cc1obj"
C Library "libgcc.a"
Object C Library "libobjc.a"
Driver "xgcc"

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.
Optimizer
Parser
Scanner

Semantic Analyzer

RTL Generator
insn-attr.h  insn-config.c
insn-attr.c  insn-flags.c
...
...

Subsystem

Generated Files at Build-time

Call Dependency

insn-peep.c
Source files come from GCC distribution

Code View

C Compiler

genattr.c genflags.c
gencodes.c genconfig.c
...

Build View

C Compiler

use

compile

Environment Parameters

depend

Environment Parameters

use

insn-attr.h insn-flags.h
insn-codes.h insn-config.h
...

compile/link

Execution View

C Compiler

use

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 it 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.
Source files come from Perl distribution.
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
Dynamic UML View
**BTV toolkit**

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

[gmake]
[graphviz]
**BTV toolkit**

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