Co-evolution of Source Code and the Build System

Supervisors:
Herman Tromp  
GH-SEL, Ghent University

Wolfgang De Meuter  
SOFT, Vrije Universiteit Brussel

Bram Adams  
SAIL, Queen's University

http://sailhome.cs.queensu.ca/~bram/

Monday, September 28, 2009
Source code and build system co-evolve.

We need tools and techniques to understand this co-evolution.
Building a Car
Building a Car: Configuration

1. features

2. tools
Building a Car: Actual Building

1. prescriptions

2. dependencies
Building Software

Windows

A fatal exception OE has occurred at 0020:C9811E36 in UX3 UMM(01) + 00010E36. The current application will be terminated.

- Press any key to terminate the current application.
- Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue.
Configuration Layer

I. features
1. features
2. tools
I. prescriptions
Source Code

Build System
Component Reuse

Mozilla Suite

Source code reuse

Thunderbird
Sunbird
Firefox
SeaMonkey
NVU
Camino
Build System Reuse?

Monday, September 28, 2009
Source code and build system co-evolve.

We need tools and techniques to understand this co-evolution.
1. Research Hypothesis

2. Tool Support to Understand Build Systems

3. Evolution of Linux Kernel Build System

4. Conceptual Reasons of Co-evolution

5. The Pitfalls of PhD Research

6. Conclusion
Understanding the Build System

MAKAO

embedded Gython

[ICSM '07]
Quake 3

- Server
- Game logic
- Client UI
- Client renderer
- Network
- Daemon
- quake3.exe

Monday, September 28, 2009
Quake 3

original game

expansion pack

conditional compilation
(error==0).visible=0

dead code
1. Research Hypothesis

2. Tool Support to Understand Build Systems

3. Evolution of Linux Kernel Build System

4. Conceptual Reasons of Co-evolution

5. The Pitfalls of PhD Research

6. Conclusion
Linux Case Study

1.2.0
2.2.0
3.2

is sufficient to discover and document patterns of coarseness. We consult archived mailing lists, websites, design documents, and software repositories to record restructuring decisions or about hot maintenance topics, puzzling build system constructs and idioms.

We can expose coarse-grained restructurings in the source code and build targets by comparing these findings across all snapshots. To learn about dependencies of an unstable series, we skipped odd releases like 0.4.0 because these are development series that have undergone major reengineering. The directory structure has remained more or less stable, which spans fifteen years of development. We chose a large tool such as GNU Make, but both build system layers have undergone significant reengineering. The directory structure has undergone a major reorganization.

3.8.2
3.8.3
3.8.6

The following two sections will look in depth at the changes introduced by the Linux 2.4.0 build process (with header file targets). Extensive build system documentation is available from the 4.4 series onwards.

Figure 3. Evolution of the average number of source files per configuration option and conditional compilation checks for configuration script documentation can be consulted to clarify any constraints between them as well as the number of places in the source code which depend on the yed selection of a feature.

Figure 4.8: Build phase

•

The following graphs show the evolution of the average number of SLOC, build and configuration scripts per directory. The "recursive make" symptoms mentioned in Section "recursive make" surface in isolation because they need various environment variables to be set by the build system written by Kai Gerschewski. At the eventual "..." build system written by Kai Gerschewski.

Table 1 shows the SLOC of source code, build scripts, and configuration scripts per file type. For each file type, the number of non-comment, non-white space lines ySLOCz and for each file the number of non-comment lines ySLOCz. The build and configuration scripts per directory are shown in isolation because they need various environment variables to be set by the build system written by Kai Gerschewski. At the eventual "..." build system written by Kai Gerschewski.

4. Metrics

4.1. Developer Communication

4.2. Exploring Directory Structure

4.3. Reading Documentation

4.4. Documenting Patterns

5. Developer Communication

5.1. Exploring Directory Structure

5.2. Reading Documentation

5.3. Documenting Patterns

6. Documenting Patterns

As a measure for the size and modularity of the source code and the build system, we first calculate the SLOC of source code, build scripts, and configuration scripts per file type. For each file type, the number of non-comment, non-white space lines ySLOCz and for each file the number of non-comment lines ySLOCz. The build and configuration scripts per directory are shown in isolation because they need various environment variables to be set by the build system written by Kai Gerschewski. At the eventual "..." build system written by Kai Gerschewski.

Table 1 shows the SLOC of source code, build scripts, and configuration scripts per directory. The "recursive make" symptoms mentioned in Section "recursive make" surface in isolation because they need various environment variables to be set by the build system written by Kai Gerschewski. At the eventual "..." build system written by Kai Gerschewski.
Figure 4.2: Evolution of the number of source code files, build and configuration scripts in the Linux kernel build system.

A first thing to note is the sheer order of magnitude exhibited by the build system and Figure 4.2. Our measurements confirm the claim made in [1] about the source code’s superlinear evolution in SLOC and in file count, but suggest similar findings for the build system on a lower scale. The build layer has grown from $\text{SLOC}$ in `build` scripts to $\text{SLOC}$ in `files`. As for the configuration layer and build support, this is even more impressive as it has evolved from nothing into $\text{SLOC}$ in `-files` and $\text{SLOC}$ in `-files` respectively. By way of reference, the source code has exploded from $\text{SLOC}$ in `-files` to $\text{SLOC}$ in `-files`. These figures suggest a very high complexity, not only in the build system and source code themselves, but also on the scale of induced changes. Further, these build scripts and programs to extract symbol tables, install kernel components, etc. As many of them have to be compiled at the beginning of the build, the Linux build system conforms to a simple version of the “Code Robot” architectural style [2]. For this, we excluded the “Documentation” directory introduced in the “9x series, as it merely contains documentation about the kernel and its build system.
Figure 4.5: Evolution of the number of explicit dependencies during the compilation of the Linux kernel.

The turbulent course of Figure 4.5 culminates in a huge growth up to September (kernel 2.6), followed by a serious dip. We also notice that eventually the number of dependencies rises again, albeit at a slower pace. Typically, the number of dependencies grows when more build targets appear in a build, redundant checks are made, artificial dependencies have been added, etc. However, there are also periods when the number of dependencies decreases. If we combine this with the observation that the number of targets does only grow partly because the compiled configuration is extended, this means that the build system complexity fluctuates a lot. Because every new target adds at least one extra edge to the build dependency graph, the reduction in complexity can only be the product of human intervention, i.e., maintenance operations in the build system. Deeper investigation is needed to verify the true nature of these changes. This is done in the next section.

Figure 4.6 shows that there is also a steady growth in the number of implicit dependencies. This means that the number of relationships the build system knows nothing about is on the rise. This is not only problematic when trying to understand the build system, it also constitutes a potential source of build errors, and at best may lead to suboptimal builds. Luckily, most of the implicit dependencies originate from temporary files created during the dep phase, i.e., files which contain the dependency makefile snippets extracted from the source code’s #include relations. Extraction of dependencies only seems to do a depth-first iteration of dependencies.

As seen in (0.2.4.0-6 implicit dependencies are relationships which are not explicitly declared as makefile rule dependencies, but rather are buried inside a rule’s command list.
Linux 2.6.16.18
could interpret a phony target as having a time stamp far ahead in the future. If it is used as a rule dependee, the rule's target always has to be remade as it is older than any phony target. Hence, phony targets can force execution of a rule's command list. Newer implementations like GNU Make have explicit support for phony targets by means of the declarative "4PHONY", whereas the traditional "FORCE" idiom lists the phony target as the target of a rule without any dependencies or command list.

Even when building incrementally, the command list of a build rule with a phony dependee will be executed, hence there is no visual difference with a full build graph.

At first sight, this seems counter-intuitive. One of "make"'s strengths is to only rebuild what is required, i.e., only execute the command list if needed. Here, the kernel build developers explicitly bypass this by using phony targets as dependees.

To understand what is happening, the relevant build logic is shown on Figure 4.14. Lines 99–9. represent the build rule for compiling .c files into object files. The "FORCE" target is indeed a prerequisite of each object file. The "FORCE" target is declared as phony in two ways here xlines 9" and 9-y for backward compatibility. Explicit usage of the "4PHONY" declarative is said to be more efficient than the emulation [98-]. The heart of the Linux kernel's FORCE idiom is the call to the GNU Make function if_changed_rule on line 90. The definition of this function xlines 8–.y contains a complicated if-test with the condition spread over lines 8–0 and the conditional action on line .4 In the case of

If that rule has a command list, this will be executed anywhere the phony target is listed as a dependency. This is the equivalent of a function call during dependency processing [8-]. Except for the addition of header file dependencies starting from the second xreybuild4.

Build Idioms
1. Research Hypothesis

2. Tool Support to Understand Build Systems

3. Evolution of Linux Kernel Build System

4. Conceptual Reasons of Co-evolution

5. The Pitfalls of PhD Research

6. Conclusion
1. **Modular** source code needs a **modular** build system

- Pure recursive make
- List-style recursive make (8 years)
- Non-recursive make (2 years)
- Recursive make with external build directory (3 years)
2. The Build System is an Executable Specification of the Architecture
3. **Correctness Trumps Efficiency**

Speculatively removing source code dependencies to **speed up** the build

| inconsistent build products |  
|-----------------------------|---|
|                            |   |
4. Configuration Layer Controls the Static Variability of Source Code
1. Research Hypothesis

2. Tool Support to Understand Build Systems

3. Evolution of Linux Kernel Build System

4. Conceptual Reasons of Co-evolution

5. The Pitfalls of PhD Research

6. Conclusion
I'm Grateful my Supervisors ...

• gave me the freedom to develop my "hobby project" into a PhD dissertation
• stimulated me to attend conferences and workshops
• taught me to learn from rejected papers
I Should Have Known that ...

- a concise dissertation is more impressive than a wordy one ;-)
- even vegetarians like salami slicing
- statistics is your friend
1. Research Hypothesis
2. Tool Support to Understand Build Systems
3. Evolution of Linux Kernel Build System
4. Conceptual Reasons of Co-evolution
5. The Pitfalls of PhD Research
6. Conclusion
Questions?

Linux Case Study

- Source Code
- Build System

Conceptual Reasons of Co-evolution

1. **Modular** source code needs a **modular** build system
2. The Build System is an **Executable** Specification of the **Architecture**
3. **Correctness** Trumps **Efficiency**
4. Configuration Layer **Controls** the Static **Variability** of Source Code