Co-evolution of source code and build systems

Bram Adams

Supervisor(s): Herman Tromp (UGent), Wolfgang De Meuter (VUB)

Abstract— **Software evolution entails more than just redesigning and reimplementing functionality of, fixing bugs in, or adding new features to source code. Corresponding changes have to be made on the build system too, in order to keep having an executable system at all times. To investigate this phenomenon, we take a look at the Linux kernel and its build system, starting from the earliest versions up until present day. Linux is a perfect case because of its complexity, development model and huge user base steering its future. First, we analyse the kernel's build traces for interesting idioms and patterns, using MAKAO, our re(verse)-engineering framework for build systems. Then, we contrast these findings with the kernel's publicly chronicled development history. Finding a good balance between obtaining a fast, correct build system and migrating in a stepwise fashion turns out to be the general theme throughout the evolution of the Linux build system.**

Keywords—**build system evolution, case study, Linux kernel, MAKAO**

I. Introduction

PROGRAMMING in the large requires more control of how a system gets built than just invoking a compiler: selecting only the minimum files to be rebuilt, determining the build order, choosing the right libraries, etc. In 1977, Feldman introduced a dedicated build tool named "make" [1]. It features explicit declarative specifications of the dependencies between targets (executables, source files, etc.) in textual "makefiles", combined with imperative "recipes" (list of shell commands) for building a target. A time stamp-based updating algorithm considerably improved incremental compilation and enhanced the quality of builds. To form a complete build system, a configuration layer is needed, which resolves platform-specific information like correct include directories and processor-specific compiler flags, but this is outside the scope of this paper.

We claim that the build system co-evolves with the source code. Every time new source code gets added, or existing modules are moved, one is forced to deal with the build system to get a working system again. An agile build system gives developers more freedom to restructure the source code. In this paper, we will examine this peculiar relation by means of the Linux kernel. Considering major kernel releases from the very beginning to the latest ones, we will look for indications of change (section III), study those maintenance actions (section IV) and check our findings with the kernel's chronicled development history. To easily understand an actual build, we developed a framework called MAKAO which will be introduced in the next section.

II. MAKAO

Most of today's build tools are still based on the Directed Acyclic Graph (DAG) model introduced by Feldman [1]. Hence, if one could extract this DAG either from the makefiles or from a trace of an actual build, some graph-based build system re(verse)-engineering tool could be created. Five requirements should certainly be met by such a framework: visualisation of a build in a manageable way, querying support, filtering of too much detail, refactoring from a higher-level than mere makefiles and flexible validation of a build's correctness.

B. Adams is a member of the Ghislain Hoffman Software Engineering Lab (GH-SEL), INTEC, Ghent University, Ghent (Belgium). He is supported by a BOF grant. Website: http://users.ugent.be/~badams/. E-mail: bram.adams@ugent.be.

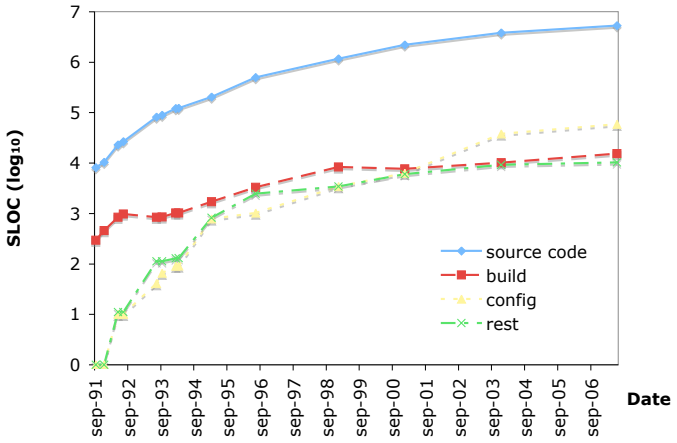

Fig. 1. Linux kernel complexity (SLOC).

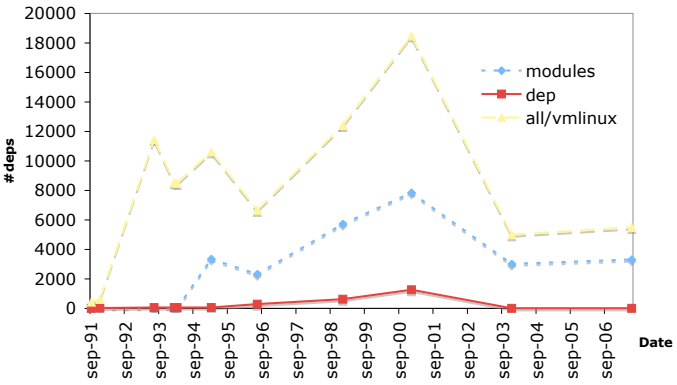

Fig. 2. Linux build explicit dependencies.

We constructed such a system, which takes a DAG obtained from a concrete build as input, but with explicit links to static information like makefile names, line number, etc. We called the resulting framework "MAKAO" [2] (Makefile Architecture Kernel featuring AOP). It is based on GUESS, a graph manipulation tool which offers both interactive and programmatical graph access. Every node and edge are objects (in the object-oriented programming sense) with their own state and behaviour. We exploited this in a number of scripts which customise GUESS for dealing with build DAGs. In the next sections, we will use MAKAO to study the Linux kernel build system.

III. Evolution of kernel build system

We looked at most of the pre-1.0 releases of Linux, as well as the major stable post-1.0 releases (up to the 2.6 series). This spans some 15 years of real-world development time. To each of these versions we applied David A. Wheeler's SLOCCount tool[1] in order to calculate the physical, uncommented SLOC (Source Lines Of Code) of source code (.c, .cpp, etc.), build scripts (Makefile, Kbuild, etc.), configuration data (config.in, Kconfig, etc.) and support build files (.sh, .awk, etc.). Figure 1 shows the results. It clearly gives a strong indication of co-evolution between source code and build system, as both more or less follow the same growth pattern. Over the course of fifteen years the SLOC of the makefiles has grown by a factor of about 52. What is also striking from this figure is the amount of work which has been put into the configuration and support system.

Next, we compiled each of the kernels using an initial configuration we enhanced with new kernel features as needed throughout the measurements. The build traces were fed into MAKAO [2] (see section II). Initially, we tried to detect some indicators for the build system complexity by measuring a.o. the number of build dependencies. This is shown in Fig. 2 for the three major build stages. The kernel image is either built by a phase named `all` or `vmlinux` (starting from version 2.6.0 in 2003), while modules are built within the `modules` stage. Extraction of source code dependencies happens during the `dep`-phase. We see that there is a huge growth up to September 2000 (version 2.4 of the kernel), followed by a serious dip. Eventually the number of dependencies rises again, albeit at a slower pace.

Dependencies capture the relationships between targets. As their number grows, so does the complexity of understanding them, which in their turn relates back to physical components of the software system and their interconnections. By "physical components", we mean the decomposition of the source code in such a way that the build system is able to create a working application from it. Hence, growth of the number of dependencies shows that understanding the build system becomes harder.

To investigate this, each DAG was loaded into MAKAO for closer inspection. The next section discusses our findings.

IV. Build maintenance

Having observed that the build system evolves and that its complexity increases throughout, we will now examine efforts of the Linux build developers to reduce this growing complexity. From Figure 1, we can deduce that in the pre-1.0 era (1992-1993) and between 2.2.0 and 2.4.0 (1999-2000) effort has been spent to mitigate build complexity, as both periods show a decrease in SLOC. In the first case, a new build scheme had been introduced, while in the second case the majority of the makefiles was rewritten in a more concise "list-style" manner.

Figure 2 also points at some reduction attempts like e.g. between 1.2.0 and 2.0.0 (1995-1996) when common build logic was extracted into a shared build script. However, the biggest maintenance step occurred between 2.4.0 and 2.6.0 (2001-2003), as the dependency extraction step has been subsumed in the other phases, and the number of dependencies has dropped drastically to one third of the 2.4.0 level, while the number of targets in the `vmlinux` build more than doubled.

[1] `http://www.dwheeler.com/sloccount/`

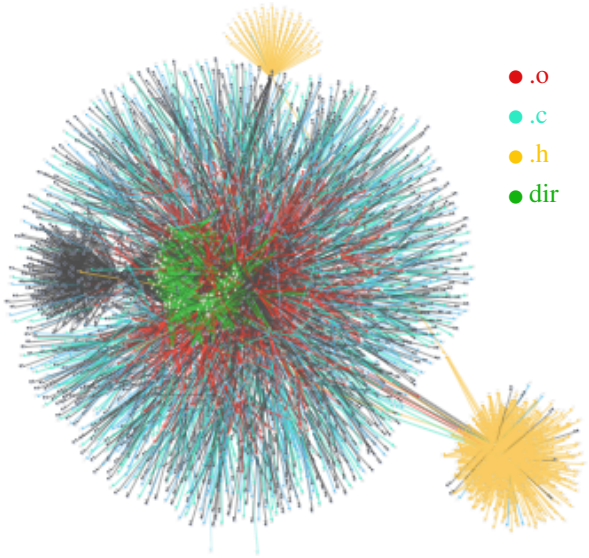

Fig. 3. Linux 2.6.0 build process (phase `vmlinux`).

We compared the two builds with MAKAO [2], and the 2.6.0 kernel (Fig. 3) turned out to be very dense compared to the straightforward 2.4.0 build. There are various reasons for this. First, the `dep` phase now occurs as part of the build itself. Second, a lot of dependencies between object files and build-time files were added. Third, a couple of build idioms are in use, which tie together a lot of build targets. The most obvious one is the "FORCE"-idiom. Target `FORCE` is a so-called "phony" target, i.e. it should always be rebuilt. Many build rules depend on `FORCE` to make sure that their recipes are always executed, as they call a custom make function to decide whether the build target is up-to-date. "Make" is solely used to express build dependencies and not to determine when rebuilding is necessary.

There are still some other peculiarities like "component objects" and the "circular dependency chain" [2]. Using MAKAO's filtering capabilities, we have been able to filter out these idioms from the 2.6.0 kernel build. The resulting DAG resembles the 2.4.0 structure, which is plausible, as the basic directory structure has remained stable between 2.4.0 and 2.6.0.

V. Conclusion

We analysed the growth of the Linux build system using indicators like the number of source lines of code (SLOC) and the number of build dependencies. We then investigated the build changes with MAKAO, our build re(verse)-engineering framework. Hence, (1) the build system evolves, (2) increasing complexity, (3) which is controlled by maintenance. This strengthens our conviction that in order to re-engineer the source code one also has to re-engineer the build system.

References

[1] Stuart I. Feldman, "Make - a program for maintaining computer programs," *Software - Practice and Experience*, 1979.

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter, "Design recovery and maintenance of build systems," in *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, Paris, France, October 2007.