Purely top-down software rebuilding

by

Alan Grosskurth

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

©Alan Grosskurth, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software rebuilding is the process of deriving a deployable software system from its primitive source objects. A *build tool* helps maintain consistency between the derived objects and source objects by ensuring that all necessary build steps are re-executed in the correct order after a set of changes is made to the source objects. It is imperative that derived objects accurately represent the source objects from which they were supposedly constructed; otherwise, subsequent testing and quality assurance is invalidated.

This thesis aims to advance the state-of-the-art in tool support for automated software rebuilding. It surveys the body of background work, lays out a set of design considerations for build tools, and examines areas where current tools are limited. It examines the properties of a next-generation tool concept, `redo`, conceived by D. J. Bernstein; `redo` is novel because it employs a purely top-down approach to software rebuilding that promises to be simpler, more flexible, and more reliable than current approaches. The details of a `redo` prototype written by the author of this thesis are explained including the central algorithms and data structures. Lastly, the `redo` prototype is evaluated on some sample software systems with respect to migration effort between build tools as well as size, complexity, and performances aspects of the resulting build systems.

Keywords: software manufacture, system building, software rebuilding, build tool, build automation, redo

Acknowledgements

Thanks to Mike Godfrey for his guidance and advice over these past two years, not just as a supervisor but as a mentor and a friend. Thanks to Steve MacDonald and Andrew Malton for their comments and suggestions, which improved this thesis. Thanks to Ric Holt and everyone else in the Software Architecture Group (SWAG) for shaping my perspective about software through many stimulating discussions. Thanks especially to Davor Svetinovic and Cory Kapser for keeping me going. Thanks to Daniel J. Bernstein for his brilliant ideas, beautiful software, meticulous precision, and inspiring dedication. Thanks to my loving family for supporting me through everything I have done in my life. Thanks to my wonderful Florina for being there when times were toughest, and for helping me through—I couldn't have done it without you.

Alan Grosskurth
Palo Alto
17 January 2007

For all my grandparents, who inspired me to aim high

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Building deliverable software from source code is a crucial but often overlooked phase of the software engineering cycle. While much work has been done to study and improve the design, programming, and testing phases, the build process is rarely examined critically and systematically. This thesis aims to take steps to rectify this situation. It treats building as a serious software engineering activity, and aims to improve engineers' ability to practice this activity by investigating a novel form of tool support for automated software rebuilding.

The build process for obtaining deliverable software from source code must be fast, automatic, reliable, and repeatable. It is, in a sense, "where the rubber meets the road" in software engineering; all programming effort is wasted if you cannot efficiently get deliverable software into the hands of the customer. Not surprisingly, the quality of the build process can often directly influence the success or failure of the project. Errors and oversights in the build process cause disruptions in the flow of code changes from programmers to testers and, later, to customers, resulting in lost revenue and marketshare.

1.1 Motivation

Results of an informal questionnaire by Kumfert and Epperly [85] indicate that the perceived overhead required to maintain the build infrastructure for a software project is approximately 12%, on average. This represents a significant proportion of resources that could be better spent on tangible improvements, such as implementing features and fixing bugs. The results also suggest that many of the developers surveyed are frustrated with current tool support for software building; they believe much of their time spent on this activity is

unproductive or "wasted," and that the build infrastructure may be limiting the progress of the project.

In his classic paper on the difficulties of software engineering, Brooks makes the distinction between *essence* and *accidents*[32]. Essential difficulties relate to an abstract construct of interlocking concepts including data, relationships, algorithms, and modules. These difficulties are inherent in the nature of software and cannot be eliminated through the use of improved technologies or methods. Accidental difficulties, on the other hand, are not inherent. They only exist because we, as software engineers, have not yet developed appropriate means to effectively address and eliminate them.

Presently, it is unclear exactly what proportion of the difficulty of maintaining the build infrastructure for a software project with the technologies and methods available today is essential, and what proportion is accidental. However, the collective experience to date strongly suggests that a significant amount of this difficulty may in fact be accidental. Hence, it may be possible to realize significant gains in productivity, reliability, and simplicity with respect to build infrastructure by pursuing improved technologies and methods that aim to reduce build-related complexity.

1.2 Contributions

This thesis makes three contributions to the area of system building. First, it provides a comprehensive survey of background work in the area of software rebuilding. Second, this thesis presents a detailed list of design issues related to implementing a build tool. The choices made by tool authors with respect to these design issues directly affect the types situations where their tool will be effective. Third, this thesis presents the first publicly available prototype of D. J. Bernstein's `redo` tool concept, a novel approach that uses a simpler, purely top-down model of software rebuilding. The prototype provides a proof-of-concept starting point for further research into patterns and techniques for applying purely top-down build tools to software systems.

1.3 Organization

The organization of this thesis is as follows. Chapter 2 surveys the existing body of research on software rebuilding, including work on software manufacture theory, standalone build tools, integrated build tools, selective recompilation, and higher-level build management. In Chapter 3 a set of design decision

categories for build tools is presented and then discussed in relation to current build tools. Chapter 4 explains the design of D. J. Bernstein's `redo` tool concept; novel features such as dynamic dependency generation, dependencies on nonexistent files, and the purely top-down approach employed by the tool are discussed. Chapter 5 presents the details of my `redo` prototype, including the central build algorithm used, design decisions faced, and opportunities for future work. Chapter 6 describes some results of an evaluation of `redo` on some sample software systems including effort required for migration, build system size and complexity, and speed of different types of rebuilds. Chapter 7 presents conclusions.

Chapter 2

Background

This chapter surveys background work in the area of software rebuilding. It explains the advances made in the past thirty years in the areas of software manufacture theory, standalone build tools, integrated build tools, selective recompilation, and higher-level build management.

2.1 Software manufacture theory

Borison presented the first formal treatment of software rebuilding in 1986. Borison describes a model for *software manufacture*[28] that generalizes and combines ideas from existing technology at the time. The purpose of the model is to help better understand and explore the effect of change on software manufacture, as well as serve as a basis for evaluating current build tools and designing new ones. The first part of the model defines a configuration as a directed acyclic graph of components and manufacturing steps; this provides a basis for identifying and delimiting the scope of changes. The second part uses difference predicates to determine when components are out of date, as well as what steps need to be taken to incorporate a set of changes.

In [86], Lamb examines several problem areas in software manufacture related to the abstraction mechanisms provided by build tools. He aims to design a few simple mechanisms to minimize what the programmer must specify to cause the build tool to construct the correct manufacture graph for the system. First, he proposes that build tools should allow the programmer to modularize, abstract, and reuse multi-step manufacturing idioms in the same fashion that most programming languages allow code to be modularized. Second, he examines how build tools could capture different *kinds* of dependencies. For example, *transitive* or *implied* dependencies are caused by a C source file

including a header file which in turn includes another header file. *Derived* dependencies are caused by a C source file including another module's header file, thus necessitating that the other module's object file be linked in as well to provide the symbols. (These examples are specific to C code, but implied and derived dependencies could be useful in modeling other parts of builds.) In [87], Lamb provides more detail about how different kinds of dependencies could be implemented and used in practice.

In [115], Singleton and Brereton present DERIVE, which employs deductive meta-programming to allow a build to be realized as a pure query evaluation. Abstract interpretation and partial evaluation are used, and minimal recompilation is supported using memoisation. Because tools are treated (or repackaged to behave) as pure functions, it is possible to schedule build tasks with a high degree of concurrency.

In [63], Gunter presents an abstract model of build dependencies based on concurrent computation over a class of Petri nets called p-nets. He develops a general theory of build optimizations and discusses how build abstractions can be created during a build and later used to optimize subsequent builds. The state of a p-net is an assignment of values to variables of the p-net, and build abstractions are additional states representing other assignments of values to variables. Gunter also uses this theory to show how correctness properties can be proved, and how optimizations can be composed and compared.

2.2 Build architecture, reverse engineering, reengineering, and program comprehension

The build process for a software system can serve as a basis for program comprehension, architectural analysis, and reverse engineering efforts. In [77], Holt, Godfrey, and Malton argue that the comprehension process for a large software system should mimic the build process. In [133], Tu and Godfrey consider describe the *build-time* architectural view and explain how it can help bridge the gap between other well-known architectural views. They examine the build architecture of three software systems to show evidence of architectural styles at the build level.

De Jonge applies component-based software engineering (CBSE) principles at the build level in [47], [46], and [48]. He introduces the notion of build-level components, defines rules for developing such components, and presents a semi-automatic process for source-tree decoupling to migrate existing software

to sets of build-level components. In [21], Ammons describes Grexmk, a tool suite for speeding up builds so they can be executed incrementally or in parallel. The suite uses one dynamic analysis to divide the build into mini-builds, and another dynamic analysis to verify that a set of mini-builds executes safely.

2.3 Standalone build tools

This section discusses the design of standalone build tools, which do not integrate with version control systems.

2.3.1 `make` and its derivatives

> Currently, over 30 different flavors of Make have been proposed and used ... Few original propositions have been made. ... There is a clear need to do better than Make; but it is a serious challenge.
>
> —J. Estublier, [53, Section 2]

Written by Stuart Feldman in the mid-1970s and first distributed with Seventh Edition UNIX, `make`[54] was a revolution. `make` gave programmers a well-defined way to capture dependencies in their software, allowing them to automatically rebuild only the necessary components after making a change. `make` operates on a set of source files, transforming and combining them to produce target files. `make` reads a text file called a `Makefile` that describes the target files to be maintained. `make` then queries the environment to determines which files have changed, and then deduces the set of commands necessary to bring the relevant target files up to date.

Consider a program that checks whether or not a mailbox has new mail. Figure 2.1 shows a sample `Makefile` for this program, along with the corresponding dependency graph constructed by `make`. As seen in the figure, the sample `Makefile` has several types of constructs. Line 2 shows a macro definition; the text `-g -Wall` will be substituted for all instances of `$(CFLAGS)` in the `Makefile`. Line 5 shows an explicit rule; it specifies that the program `mailcheck` depends on three object files, `mailcheck.o`, `buffer.o`, and `maildir.o`. Line 6 shows the commands that are issued to rebuild `mailcheck`; the special `Makefile` variables `$@` and `$<` are used to refer to the target and dependency list, respectively. Line 10 shows and implicit rule; this specifies how an object file is rebuilt from the corresponding source file. The "old-fashioned suffix rule" notation is used here, although newer `make`'s such as GNU `make` support more general pattern rules. Lines 14–19 show extra rules that specify dependencies of the object files on the appropriate source and header files.

```
# Macro definition
CFLAGS = -g -Wall
# Explicit rule
mailcheck: mailcheck.o buffer.o maildir.o
        gcc $(CFLAGS) -o $@ $<
# Implicit rule
.c.o:
        gcc $(CFLAGS) -c -o $@ $<
# Possibly auto-generated rules
mailcheck.o: mailcheck.c maildir.h buffer.h
buffer.o: buffer.c buffer.h
maildir.o: maildir.c maildir.h buffer.h
```

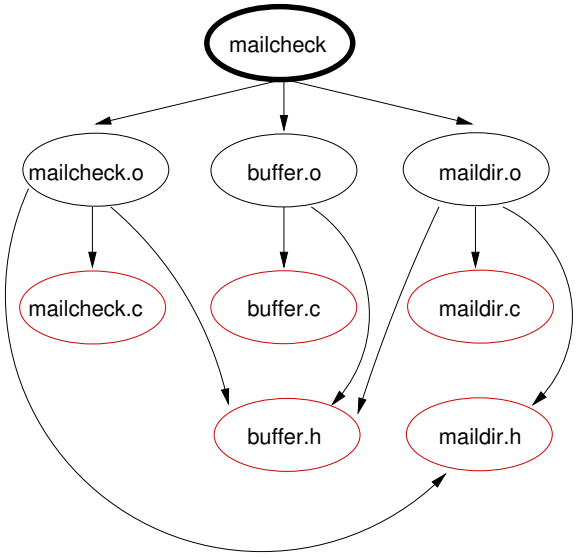

Figure 2.1: A sample `Makefile` and corresponding dependency graph

These rules are often generated prior to running `make` using a program like `makedepend`[137].

In [55], Feldman notes that `make` was not suitable for describing large programs. Regardless, `make` is still used today, over thirty years later, as the de facto build tool for extremely large software systems. Part of `make`'s success owes to the simplicity and generality of the design, while part of it owes to the many enhancements and extensions that have been added over the years; in [56], Feldman summarizes the goals and limitations of the tool as well as changes made during the first decade of use.

In [75], Hirgelt describes an enhanced version of `make` with new conditionals, expressions, loops, includes, and improved error handling. In [51], Erickson and Pellegrin describe `build`, an extension to `make` in which a *viewpath* environment variable can be defined to contain lists source directories to be overlayed on each other; this is useful because it allows developers to share a common source tree and keep copies of only the files they change in a separate tree, thereby saving disk space. This feature is now found in many modern `make` variants such as GNU `make`[122, Section 4.5.1].

In [59], Fowler describes *Fourth Generation* ***make*** (later known as `nmake`), which implements major semantic and syntactic enhancements to the original `make`, including support for source files distributed among many directories, an efficient shell interface that allows concurrent execution of update commands, dynamic generation of header file dependencies, dependencies on conditional compilation symbols, and a new meta-language for constructing default rules. In [60], Fowler explores how several ideas and features implemented in `make` derivatives (such as `nmake`) can be used to improve the loosely coupled, tool-based model of software rebuilding and make this approach favourable to the tightly coupled, seamless model employed by integrated development environments (IDEs).

In [80], Hume describes `mk`, which features improved efficiency, transitive closure on implicit rules, parallel execution of update commands, and regular expression implicit rules. (Most of these features are now also found in GNU `make`.) `mk` differs from `nmake` in that it aims to be a general-purpose tool for maintaining file dependencies, without any built-in knowledge about C programming. `mk` reuses syntax and concepts from original `make` wherever possible, but also generalizes many of `make`'s features. For example, metarules are generalized to full regular expressions, transitive closure is allowed on all rules including metarules, and parallel execution is allowed for any set of update commands. `mk` also eliminates special cases; for example, `mk` variables are simply shell variables and update commands are shell scripts. In [81], Hume and Flandrena explain how `mk` is used to maintain files in the Plan 9 environment.

In [118], Somogyi describes `cake`, a "fifth generation" rewrite of `make` from the ground up. `cake` uses the standard C preprocessor to perform macro substitution and include description file fragments. `cake` also provides transitive implicit rules, and permits the use of variables in implicit rules for increased flexibility. Finally, `cake` adds support for conditional rules and dynamic dependencies, which can be used to `make` build scripts more flexible.

Several `make` variations allow parts of the build to be parallelized. Version 8 `make` requires the user to explicitly indicate which commands can execute in

parallel. Concurrent Make[42] (also known as `cmake`) is written in Concurrent C and based on Version 8 UNIX `make`; it executes commands in parallel by default and explicitly distributes commands among multiple processors. `cmake` offers special rules to force update commands to run locally or to suppress parallelism for certain targets. `pmake`[24], implemented as a module for UNIX SVR2 `make`, also executes commands in parallel by default but delegates to the underlying operating system to make efficient use of multiple processors. As mentioned before, `nmake` and `mk` both provide support for parallelization. *Parmake*[106] provides concurrent execution of operations that have no mutual dependencies; a set of heuristics is used to balance the local load while another program called *dp* schedules distant processes based on machine-load statistics. DYNIX `make` (also known as `dmake`), provides a mechanism to specify parallelism explicitly by placing an ampersand next to the colon in a rule. PGMAKE[92] extends GNU `make` using clustering software that makes heterogeneous networks of parallel and serial computers appear as one computational resource. (When compiling software, the different machines must be able to generate code for the target architecture by cross-compiling, if necessary.)

Optimistic make[34][35][105] monitors the file system for out-of-date targets, and executes the commands necessary to bring targets up to date before the user types `make`. In [78], Holyer describes an "automatic" `make` facility called `bmake`, which is implemented using command tracing. Rather than maintaining an actual `Makefile`, the user only needs to build the target files in some way, either by hand or using shell scripts; command transactions are monitored by a user shell called `brush`.

GNU `make`[122][98] is likely the most popular `make` derivative in use today; it incorporates many features from other `make`s such as conditionals, functions, `include` statements, and parallel execution. The GNU `make` Standard Library (GMSL) is a collection of functions implemented using native GNU Make functionality that provide list and string manipulation, integer arithmetic, associative arrays, stacks, and debugging facilities. Makepp[76] is a drop-in replacement for GNU `make` written in Perl; it also contains some extensions. Variations of BSD `make` are also in wide use to install and maintain software on FreeBSD, NetBSD, and OpenBSD. Object Make[119][120] is a modern variation of `make` that aims to allow build scripts to be separated into reusable components.

Table 2.1 summarizes the various `make` derivatives that have been created over the years. Note that there also exist Microsoft's `nmake` and Kitware's `cmake`[97] that share their names with older, unrelated variations. The large number of `make` implementations makes it difficult to write portable `makefiles`; some of the differences between them are outlined in [103].

Year	Name	Alternate name	Backwards compat.?
1979	`make`	"original" or "classic" `make`	—
1984	`build`	—	yes
1985	`nmake`	Fourth Generation `make`	no
1986	`cmake`	Concurrent Make	yes
1987	`mk`	—	no
1987	`cake`	Fifth Generation `make`	no
1987	*Parmake/dp*	N/A	yes
1987	`dmake`	DYNIX `make`	no
1987	omake	Opus `make`	N/A
1988	`pmake`	N/A	yes
1988	MMK	MadGoat `make`	unknown
1988	`gmake`	GNU `make`	yes
1989	—	"optimistic" `make`	N/A
1995	—	Object Make	yes
2000	`bmake/brush`	"automatic" `make`	N/A
2003	`makepp`	—	yes

Table 2.1: Various `make` derivatives

Numerous tools have also been developed to work in cooperation with `make`, tailoring or generating `Makefiles` based on other files or the results of platform probes and dependency calculations. Mm4[94] uses the M4 macro processor to customize `Makefiles`. Tilbrook's `pmak`[130] serves as a front-end to `make` that aims to abstract the underlying platform differences and aid with the construction of large-scale software. QEF[129] represents a later, more-refined tool by the same author. The GNU autotools (`autoconf`, `automake`) are currently used in a wide variety of free and open source software projects; `imake`[50] was used for many years as the basis for the X11 build system.

Some work has been done to examine problems with the fundamental `make` model and the way it is used; in particular, out-of-dateness checks based on timestamps have been shown to be unreliable in practice [100]. Many newer build tools calculate cryptographic hashes of file content for use as the basis of out-of-dateness checks. The popular idiom of "recursive make," where `make` is reinvoked by itself, has been shown to be error-prone and inefficient[99].

2.3.2 `make` alternatives Jam[141] is a free, fast, customizable build tool written by Christopher Seiwald. Jam has its own description file language, which includes control-flow statements, variables, rules, and actions. The Jam language separates operating system-independent descriptions from operat-

ing system-dependent descriptions. Jam has considerable built-in knowledge about how to build C and C++ programs; indeed, this is the type of project where it is commonly used. Several variants of Jam have developed including FT-Jam[135], Boost.Jam[19], and `KJam`[18].

Odin[39][40][41] is a replacement for `make` that focuses on extensibility and efficiency. A central part of Odin is the *derivation graph*, which describes relationships between tools in the environment. Another interesting feature of Odin is support for *compound derived objects*, which consist of sets of derived objects; this allows all build steps to be abstracted as producing a single output.

2.3.3 XML-based build tools Apache Ant[45] is an open source Java-based build tool designed for cross-platform use. Rather than using the shell to issue update commands, Ant is extended using Java classes. Instead of a custom build description file syntax, Ant uses an XML-based language. Instead of variables, Ant uses *properties*, which are immutable once initially set. Ant includes many built-in tasks to support the typical structure of Java projects. For example, the Java compilation task understands the directory structures involved with Java packages, and is able to compile all Java source files in a directory at once. An add-on package provides extra tasks for C++ support, including automatic dependency checking. It is also possible to write Ant build scripts in Java-based scripting languages, such as Groovy[123].

NAnt[113] is a variation of Ant written for Microsoft's .NET platform. It is designed to support C# projects, but also has built-in support for C++. In practice, NAnt uses a slightly different syntax for build descriptions that Ant. MSBuild is Microsoft's own .NET-based Ant variation. Geant, part of Gobo Eiffel[27], is an Eiffel-based Ant variant.

2.3.4 Scripting language-based build tools In the past decade, the popularity of scripting languages has given rise to a number of build tools that operate on build descriptions expressed as code written in a single scripting language. This eliminates the complexity caused by interactions between a build tool language and a build command language. For example, GNU `make`'s `Makefile` syntax provides variables, conditionals, various functions for transforming text. However, the actual commands are written in shell script, which provides its own syntax for the same features. Scripting language build tools often incorporate many improvements found in various `make` derivatives, such as the use of cryptographic hashes rather than timestamps to determine if files are out of date.

Cons[114], which uses Perl, is one of the earliest scripting language-based build tools. Cons uses cryptographic hashes to determine out of dateness, and focuses on full and accurate dependency analysis. It has built in rules for C++. SCons[83] originally began as a port of Cons to Python, but is now its own full-fledged build tool. SCons is designed in an object-oriented style; `Builder` objects do the actual construction, `Emitter` objects manipulate lists of files, and `Scanner` objects parse source files and extract implicit dependencies. This object-oriented approach makes SCons different from other build tools. SCons has built-in knowledge about several kinds of projects, including C++. A-A-P[101] is another build tool is also based on Python. tmk[108] and Bras[82] use TCL, Rake[138] and Rant[89] use Ruby.

2.3.5 Compiler caches A compiler cache is a tool that acts as a wrapper for a compiler. Each time a file is compiled, the wrapper stores the result in a cache. Upon subsequent recompiles, the wrapper can fetch the result from the cache rather than invoking the compiler again. This can speed up build times because it is much more expensive to invoke the compiler than simply fetching the resulting object file from the cache.

The System Modeller for the Cedar programming environment is an early example of a compiler cache; it used a "48-bit version stamp which is constructed by hashing the name of the source object, the compiler version and switches, and the version stamps of any interfaces which are parameters of the compilation."[88]. More recent examples of compiler caches include *compilercache*[125] and *ccache*[132].

Compiler caches are often able to store many different compilation results for a particular input file. This can help speed up builds if a change is reverted, or if multiple different variants of a software system are frequently built, such as versions with and without debugging symbols. On its own, a compiler cache can be used to implement single-phase system building. Build commands can be stored in a script, which is run in its entirety each time a rebuild is requested. A compiler cache can also be used in conjunction with a generic build tool such as `make`. In this case, the compiler cache acts as a sort of safety net for `make`'s dependency checking. That is, the compiler cache will save time if `make` requests a file that has not actually changed to be recompiled.

In [84], Koehler and Horspool explain how a caching compiler can be used to maintain a database of partially processed header files to eliminate redundant work of reprocessing the same headers over and over. This is a different sort of compiler cache than those mentioned above.

2.4 Integrated build tools

An integrated development environment (IDE) is a program that assists programmers in writing software. An IDE often includes its own build tool, along with other components such as a code editor, compiler or interpreter, and debugger. The tightly-coupled nature of these components allows them to share information. A build tool that is part of an IDE often has access to the code's abstract syntax tree, which may be extracted by the code editor in real-time.

DSEE[90] was an early commercial product that integrated building with versioning; it was based on system models, which describe the version-independent structure of a system, and *configuration threads*, which describe the version requirements. `shape`[95] was an extension of `make` that provided full access to the version control system and support for *configuration rules*, which control the selection process for component versions during rebuilding. `shape` is built on top of an *attributed filesystem*, that provides a more generalized scheme for document identification than does a regular filesystem.

Vesta[65][67][69][71] is an advanced software configuration management system designed to support two important tasks: versioning and building. System models are specified in a functional system description language (SDL) and, when executed, the results of function calls are cached and later used to transparently speed up rebuilds[72]. In order to improve repeatability, builds are done on a separate isolated build server.

2.5 Selective recompilation

Selective recompilation mechanisms reduce unnecessary or redundant compilations by considering the semantic effect of changes to source files. Tichy's *smart recompilation* [126] is one of the first examples; Schwanke and Kaiser's *smartest recompilation* [110] allows certain harmless inconsistencies to remain in the build. The idea has also been applied to Java[49][23]. There have been many other variations on the idea[112][38].

Selective recompilation techniques must balance the time taken to determine which recompilations are unnecessary with the time taken to simply perform those recompilations; if the selection algorithm is too complex it may actually take more time that simply recompiling the files in question right away[29][20]. Yu, Dayani-Fard, and Mylopoulos investigated using a graph algorithm and programming tool to remove false dependencies that result in lengthy preprocessing time[143].

2.6 Higher-level build management

Higher-level build management overlaps with the area of automated release management. It addresses problems such as scheduling builds among a pool of build servers, notifying the appropriate people when builds fail, running test suites after builds finish, and copying and archiving deliverables to file stores. Examples of higher level build tools include Apache's Maven and IBM Rational's BuildForge. The term *continuous integration* is often used to describe techniques that ensure higher-level build management runs around the clock, giving developers constant feedback about the status of the project.

2.7 Summary

This chapter has surveyed the background work related to software rebuilding. I have explained the advances made in software manufacture theory, standalone build tools, integrated build tools, selective recompilation, and higher-level build management over the past thirty years.

Chapter 3

Build tool design

This chapter discusses the topic of build tool design. In order to evaluate and compare build tools, it is important to be aware of the various dimensions along which build tools can differ. This chapter lays out a set of design considerations for build tools developed by the author of this thesis. It also discusses how these issues relate to current build tools.

3.1 Design considerations

This section presents set of design considerations that can help in analyzing different build tools. Each point represents a design decision made by tool authors, and the choices made directly affect how the build tool will be used as well as which types of projects it will be suited to.

3.1.1 Operation What does the build tool do when it is invoked? How and when does the build tool parse the description files? What are the performance characteristics of the build tool? How long does the build tool take before building a single target? How long does the build tool take to determine nothing needs to be built (known as a *null build*)?

Most build tools look for a build description file with a well-known name in the current directory. This description file may include or call description files in other directories, which are processed as well. In order to effectively organize and maintain a build system for a project, it is essential that it be possible to break down the build description into separate files or modules.

3.1.2 Organization Where are the description files located with respect to the source files? Where are the target files stored?

The build tool may leave this decision up to the programmer to specify. It may be desireable to store some build files in the same location as the source files they operate on, while other build files may be kept together as a build module. As a system grows, it is increasingly important for the build tool to allow build description files to be distributed throughout the source tree, since the system may be composed of individual, reusable subsystems.

3.1.3 Syntax What syntax is used for the build description files? Are there multiple different syntaxes used or a single unified syntax? Is dependency information specified using the same syntax as command sequences?

Traditionally, build tools such as `make` have used their own custom syntax to specify description files. However, at some point, the build tool needs to be able to execute commands to actually generate target files. Therefore, there must be some interface to the underlying operating system. `make` uses the shell to execute commands, while Ant allows commands to be written in Java. Newer scripting language-based build tools allow build description files to be specified entirely in a single chosen scripting language.

3.1.4 Interaction with the environment Can information be passed to the build tool from the environment? Does the build tool attempt to prevent information from the environment from accidentally influencing the build?

Many aspects of a user's environment can influence build results. For example, the value of the `$PATH` environment variable will influence which particular versions of tools will be used, provided they are not specified with absolute paths. `make` allows all environment variables to be directly visible inside the build description files, while SCons requires a variable to be explicitly declared before it can be visible.

3.1.5 Integration with versioning Is the build tool aware of multiple versions of the same source file?

Some build tools operate with no knowledge of multiple versions of source files; they are only aware of the current versions. This type of approach seems to be tailored towards monolithic projects with one large source repository. However, when software is split into multiple separate repositories, each evolving separately, it may be helpful for the build tool to cooperate with the versioning tool to use knowledge about what the different variants of the components are and what a valid configuration consists of.

3.1.6 Caching Does the build tool keep track of multiple versions of target files?

Target files are not traditionally kept under version control. However, if a build tool stores previous targets in a cache, it may be able to reduce build times if old versions of the targets are ever needed again. For example, if a change is rolled back and the software is rebuilt, the previously cached versions of targets can be used to speed up the build process.

3.1.7 Variants Does the build tool consider that the user may want to build multiple versions of target files side-by-side using different configuration parameters (source files)?

It is common in many large organizations to require several different types of builds at once, such as one with debugging information enabled and one with more aggressive code optimizations enabled. While it is possible to get two different types of builds by building the software twice separately, it may be desireable to build many variants at once.

3.1.8 Built-in knowledge Does the build tool know how to build certain types of target files? How is this knowledge represented? How is it extended by the user?

Most build tools have at least a small amount of innate knowledge about how to rebuild certain types of derived objects. For example, `make` has a set of default built-in implicit rules that dictate how to rebuild object files from C source files, and how to generate C parser source code from Yacc grammars. Some build tools, such as SCons, provide even more sophisticated built-in rules that take into account information about the current build platform when deciding how to invoke necessary tools.

On one hand, built-in knowledge may be attractive because it allows the build tool to be used out-of-the-box without extensive effort. On the other hand, built-in knowledge is often useless for projects of any significant size, since these projects invariably have slightly different assumptions and requirements. In these cases, it may be better for the tool to let the project specify all of the necessary knowledge and build description information explicitly.

3.1.9 Grouping How can multiple different targets be grouped together and built the same way?

Most projects build many different target files that are all of the same type. For example, a C program will typically be composed of several different object files, which are built from corresponding source files. It is not realistic to expect

the programmer to specify the build description for each of these individually; the build tool must offer a way to group these files together and allow them to share common build description information. Many tools, such as `make`, allow this to be done with suffix matching. For example, the programmer can specify how a file with a `.o` extension is rebuilt from the corresponding file with a `.c` extension. Although this is usually sufficient for most projects, it may be desireable to use other matching mechanisms, such as the arbitrary pattern matching facility provided by Fowler's `nmake` tool.

3.1.10 Batching Does the build tool allow multiple targets to be created from a single command sequence? Or does each command sequence build exactly one target?

While compilers and other tools typically translate one or some input files into a single output file, some newer compilers perform better when invoked in "batch" mode. For example, the Visual C++ 2005 compiler will compile several source files faster if invoked once in batch mode, as opposed to many times in single-output mode. This may create a problem for the build tool if it is designed around the model that each build step produces exactly one output. One way for such build tools to cope with batching is to output a single archive file, such as a JAR, that contains all outputs. Then, subsequent build steps can retrieve individual files from the archive, keeping the model consistent. Odin's concept of *composite* targets works in a similar way.

3.1.11 Granularity On what types of artifacts does the build tool operate? Does it treat source files as opaque or transparent?

Most build tools, such as `make`, operate strictly on the file level. They generate target files from source files, and they treat files as opaque objects. Interestingly, `make` has another level of granularity in description files: variables; however, `make` does not allow dependencies on variables. That is, if the compilation flags are stored in a `Makefile` in a variable called `CFLAGS`, `make` will not notice if this variable is changed between builds and rebuild the corresponding object files, unless each object file explicitly depends on the `Makefile`. This is discussed in more detail in Section 4.1.3. By contrast, SCons allows dependencies on variables and directories, as well as files.

Techniques such as smart recompilation treat source files as transparent; they determine what steps are necessary to rebuild the system by parsing the source files and using language-specific knowledge to determine the impact of changes.

3.1.12 Composability How are build systems for disjoint subsystems or subcomponents composed into a single build system? How much effort is required to incorporate an existing standalone component with its own build system into a larger codebase?

As a system grows larger, it will invariably be composed of distinct, reusable subsystems. While the code of these subsystems is designed to be reusable separately, the build description files are often not. In order to integrate a new subsystem into a system, the build files for the overall system must be able to utilize the subsystem's build files at the appropriate point, passing in any necessary information.

Currently, many open source software systems use GNU `autoconf` to create a configuration script that is run before the system is built, filling in values in a set of `Makefile` templates. Unfortunately, when projects using `autoconf` are composed, the same tests are often repeatedly run for each subsystem.

3.1.13 Concurrency What parts of the build process run in parallel? Does the programmer need to explicitly indicate the build tool?

Modern software systems are built on machines with multiple processors and multiple processing cores. In order to take full advantage of the hardware, build tools should support running different parts of the build process concurrently. Unfortunately, it is not always trivial to determine what parts can be run concurrently, and incorrectly parallelizing build steps that should be executed serially can jeopardize the correctness of the build. It is even possible that the build will succeed, but the final system will not be correct, resulting in bugs that are difficult to track down.

It is beneficial for a build tool to allow the programmer to indicate which steps are safely parallelizable, either implicitly or explicitly. The build tool can then exploit this information to reduce build times by performing different parts of the build concurrently. The amount of parallelization possible may be directly influenced by the organization of the build description files. For example, a large description file for a project must be read serially. However, if it is broken up into fragments, organized in a tree-like formation, different branches of the tree could be processed in parallel.

3.2 Discussion with respect to current build tools

Most current build tools do not make an effort to integrate with versioning tools. One reason could be related to the large amount of open source software in use today. These projects are often distributed as a source code archive over the web, to be downloaded and built by the user prior to installation. By not integrating with versioning tools, it makes this installation process simpler for the user.

Composability is an area where most current tools have difficultly. It would be desireable to be able to drop in a new subcomponent into an existing project and have the project's build system interface properly with the subcomponent's build system. In theory, this is not difficult if all the files are located in the project root's directory, and hence share the same namespace. However, in reality, the issue becomes complicated by the fact that most projects isolate subcomponents into separate subdirectories. Source and target files are specified relative to the project root directory, and because the subcomponent is no longer in the top-level project directory the paths in its build scripts must be modified to include an extra directory component. One workaround for this is for the build tool to change to the subcomponent's directory before building it, as is done with recursive `make`. This effectively changes the root of the namespace to what the subcomponent expects. However, it also throws away all the benefit of being able to analyze dependencies in a whole-project manner.

Although most current build tools have at least some facility for building targets in parallel, the way the tools are used often prevents parallelization from happening in practice. Consider the GNU `autoconf` program, which is used as a de facto standard in the open source community in conjunction with GNU `make`. `autoconf` creates a configuration script that the user needs to run before running `make` in order to execute platform probes to customize the `Makefile`s. While these probes could executed in parallel, the scripts generated by `autoconf` always execute in serial. The result is a long period with no parallelization at the start of the build while this script is run. For large projects, this period may not be significant with respect to the total build time; however, for small projects, running the configuration script may take more time than actually executing `make`.

3.3 Summary

In this chapter, I have described several important issues relevant to build tool design. How these issues are handled by the build tool directly affects how the build tool is used, and which types of projects it is appropriate for. I have discussed how these design issues are handled by current tools current tools and the consequences of these design decisions.

Chapter 4

Purely top-down software rebuilding with `redo`

4.1 Specification

In 2003, D. J. Bernstein published four web pages describing a new build tool concept called `redo` in [26]. At present, this is the only information available `redo`. While Bernstein has not published a version of the software, he has made available several `redo`-style build scripts in the `bib` directory of his FTP server. These build scripts are presumably used along with an unpublished `redo` implementation to translate bibliographic information stored as simple TEXinto HTML and BibTeX, and serve as an example of how `redo` might be used in practice. The following is a high-level specification of `redo`'s features as inferred from Bernstein's web pages and bibliography build scripts.

4.1.1 Semantics `redo` operates in purely top-down manner. Given a target to build, the corresponding build script for that target is read and executed. For example, if asked to rebuilt `hello.o`, `redo` will use the build script `hello.o.do`. If that build script does not exist, it will fall back to using `default.o.do`.

Dependency information is embedded in the build script commands, causing dependencies to be recursively rebuilt as a side-effect of building the target itself. This behaviour is similar to the way functional languages use lazy evaluation to delay a computation until the result is known to be needed. Dependency information can be expressed using one of two programs:

- "`redo-ifchange hello.o`" means "Remember that the current target depends on `hello.o`; and if `hello.o` is not currently up to date, rebuild

it."

- "`redo-ifcreate hello.o`" means "Remember that the current target depends on `hello.o` not existing."

4.1.2 Atomic target rebuilding Invoking programs in the simplest way from a typical `Makefile` results in each target file being rebuilt one disk block at a time. This can confuse a program that uses a target file because the file is truncated while it is being built. It can also confuse `make` because, if the build process is suddenly halted, the truncated file remains and is mistaken on subsequent runs as being correct and up to date. `redo` creates the new target under an alternate filename and, once complete, atomically replaces the old target with the new one.

For example, the build script `a.do` for a target `a`

```
redo-ifchange c
sed 's/World/Waterloo/' < c
```

is comparable to the `Makefile`

```
a: c
	sed 's/World/Waterloo/' < c > a
```

but is actually as safe as the `Makefile`

```
a: c
	sed 's/World/Waterloo/' < c > a---redoing
	mv a---redoing a
	fsync
```

Note that the target filename is implicit in the build script name.

4.1.3 Isolated build descriptions A typical `Makefile` may contain multiple explicit rules that describe how particular targets are rebuilt, as well as multiple implicit rules that describe how particular classes of targets are rebuilt. `redo` requires the rule for each target or class or targets to be isolated in its own file. For example, the rule that describes how to rebuild a target named `buffer.o` is stored in a file called `buffer.o.do`. The rule that describes how to rebuild a `.o` file from a `.c` file is stored in a file called `default.o.do` (however, this will only be used if a target-specific file does not exist).

Isolated build descriptions allow `redo` to avoid rebuilding targets that are not affected when part of the system's build description changes. For example,

suppose the file `c` contains one line with the string "Hello, world". Figure 4.1 shows a `Makefile` that can be used to build two targets, `a` and `b`, each performing a text substitution on `c`. The bold circle represents a top-level `Makefile` target that depends on both `a` and `b`. A change to either the commands or the dependency lists will not trigger a rebuild, since `make` only checks that the dependency list is up to date (the targets do not depend on the Makefile). This is a problem because it forces the programmer to manually determine which targets are out of date and rebuild just those. Or, to be safe, the programmer will simply rebuild everything.

```
# Makefile:
it: a b

a: c
        sed 's/World/Waterloo/' < c > a

b: c
        sed 's/Hello/Goodbye/' < c > b
```

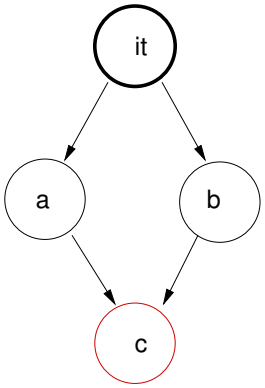

Figure 4.1: No dependencies on build script with `make`

This problem can be mitigated somewhat by making all targets depend on the Makefile and encapsulating each of the command sequences in a separate script. The resulting `Makefile` is shown in Figure 4.2. Now the correct target will be rebuilt if its command sequence changes, and everything will still end up getting rebuilt if the Makefile changes, although this may be unnecessary.

Figure 4.3 shows how `redo` handles this situation. Because each build script is isolated in a separate file and each target implicitly depends on its build script (indicated by the dependencies in grey circles) a change to a build script is guaranteed to only trigger a rebuild of the corresponding target.

4.1.4 Homogeneous build descriptions A typical `Makefile` rule has two separate components. The first component is list of targets along with a list of associated prerequisites. The second component is a sequence of commands that, when executed, brings the targets up to date. A `redo` rule is homogeneous: there is only a sequence of commands that, when executed, brings the target up to date. Prerequisite information is embedded in the

```
# Makefile:
it: a b Makefile

a: c build a Makefile
        sh build-a > a

b: c build-b Makefile
        sh build-b > b

# build-a:
sed 's/World/Waterloo/' < c

# build-b:
sed 's/World/Waterloo/' < c
```

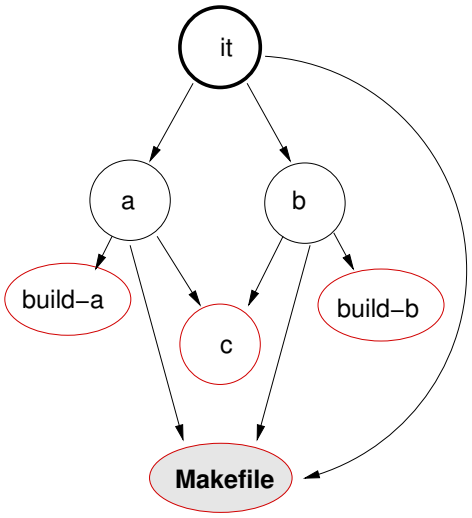

Figure 4.2: Partial dependencies on build script with `make`

commands by invoking the `redo-ifchange` and `redo-ifcreate` programs with appropriate arguments.

Homogeneous build descriptions can be thought of as the defining characteristic that gives `redo` its power. Because the dependencies for a target are not determined until the target starts building, the target's build script can use up to date information from other parts of the build to decide what the dependencies should be. This technique is discussed further in Section 4.1.6.

4.1.5 Up to dateness not determined by timestamp `make` considers a target file up to date if its timestamp is newer than the timestamp of all of its dependencies. `redo` does not use the newness of a target file as a reason to consider the target file up to date; `redo` considers the target file up to date only after it finishes and records what happened in `.redo`, a file in the top-level build directory containing build metadata.

Several build tools are able to determine whether a file is up to date using the file contents, rather than the timestamp. The key idea about `redo` is that it records what happens at each build step, not just isolated pieces of information about each file. This is discussed more in Section 5.4.1.

4.1.6 Dynamic prerequisite generation `make` does not allow to allow the content of a prerequisite to itself be a list of prerequisites. Since all prerequisite information is processed only once at the beginning of a build when the `Makefile` is read, `make` will not reexamine such a list if it changes after the

```
# it.do:
redo-ifchange a b

# a.do:
redo-ifchange c
sed 's/World/Waterloo/' < c

# b.do:
redo-ifchange c
sed 's/Hello/Goodbye/' < c
```

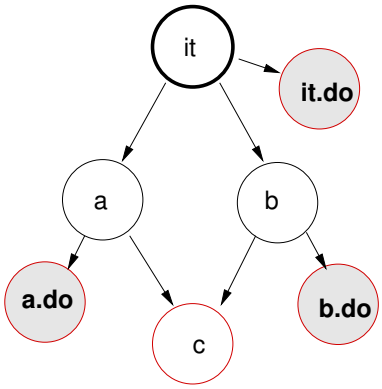

Figure 4.3: Full dependencies on build script with `redo`

prerequisite is rebuilt. GNU `make` allows the content of a prerequisite to itself be a `Makefile` fragment which can be `included` by the primary `Makefile`. Upon reading a `Makefile`, GNU `make` will immediately notice this scenario; if the fragment is out of date, GNU `make` will rebuild it and then start again from the beginning. `redo` is designed to allow the content of one prerequisite to itself be a list of prerequisites. Because targets are rebuilt in a strictly top-down manner, the build scripts can be written to ensure the prerequisite will always properly rebuilt before its content is extracted and used in further stages of the build.

`make` traverses nodes in a post-order fashion, rebuilding the current node after rebuilding each of its children. `redo`, on the other hand, rebuilds child nodes in the middle of rebuilding the current node (by invoking `redo-ifchange` from within the current node's build script). After a particular `redo-ifchange` command returns, indicating the requested child nodes have been built, `redo` then continues building the current node.

Consider the example of compiling a C source file into an object file. The source file may include header files, on which the object file should also depend. These dependencies are often called *implicit dependencies*. The common solution to this problem when using `make` is to add a separate pre-build stage that runs a program such as `makedepend` to generate a `Makefile` fragment expressing these dependencies on the header files. `makedepend` cannot be called from within the `Makefile` since the new fragment won't be re-evaluated. (The exception is GNU `make`, which is able to re-evaluate the new fragment by going back and starting at the beginning.) In order to be safe, this pre-build stage should be run before each build. However, this is obviously a waste of time if `make` is, for example, being invoked to build documentation targets rather

than program targets.

Figure 4.4 shows how `redo` solves the problem of implicit dependencies. The object file can depend on not only a file listing the relevant header files, but also each of the listed headers; The UNIX `cat` command can be used to extract the list of headers from the dependency file.

```
# hello.o.do:
redo-ifchange compile hello.c hello.o.deps
redo-ifchange `cat hello.o.deps`
./compile hello.c

# hello.o.deps.do
redo-ifchange findincludes hello.c
./findincludes hello.c
```

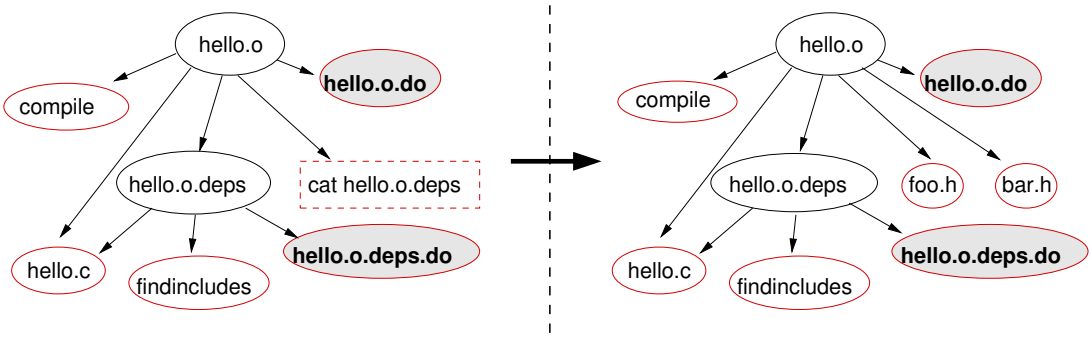

Figure 4.4: Dynamic dependencies with `redo`

4.1.7 Dependencies on nonexistent files `make` will only notice one type of change made by the user: the modification or removal of an existing file. `redo` can notice this type of change, too. `redo` can also notice when a particular file that did not exist before is created and then incorporate this into the build process. The way this is accomplished is through the use of a command called `redo-ifcreate`. While the line

```
redo-ifchange buffer.h
```

indicates that the current target should be rebuilt if the existing file `buffer.h` is modified or removed, the line

```
redo-ifcreate buffer.h
```

indicates that the current target should be rebuilt if the nonexistent file `buffer.h` is created.

For example, consider a software package containing several small C programs. These programs are all compiled in roughly the same way, using a set of standard flags. However, it may be desirable to compile some programs with extra flags. With `redo`, each program can have a dependency on a special file, which may or may not exist containing extra flags for the build of that program. If the file is not found by the build script, it simply uses the default flags. If the file is created at some point, `redo` will notice and rebuild the program, even if nothing else has changed. This is illustrated in Figure 4.5.

```
# it.do:
redo-ifchange hello.o

# hello.o.do:
redo-ifchange cflags
cflags=‘head -1 cflags‘
if [ -e hello.cflags ]; then
  redo-ifchange hello.cflags
  cflags="$cflags ‘head -1 hello.cflags‘"
else
  redo-ifcreate hello.cflags
fi
redo-ifchange hello.c
gcc -o /dev/stdout $cflags hello.c

# hello.cflags:
-O3
```

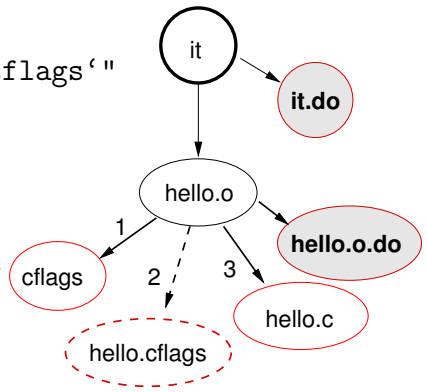

Figure 4.5: Dependency on a nonexistent file with `redo`

4.2 Discussion

The features of `redo` work together in a consistent way. By making build scripts ordinary shell scripts, flexibility is gained in where dependencies can be declared. For example, the dependency declarations can be intermixed with commands that output parts of the target file; this naturally allows for dynamic prerequisite generation because information from earlier dependencies in the script can be used to calculate later dependencies.

In most situations where dependencies on nonexistent files are desireable, it is also desireable to be able to specify what happens if the nonexistent file actually exists. In this case, a normal conditional shell construct can be used to check if the file exists or not, and then trigger the appropriate invocation of either `redo-ifchange` or `redo-ifcreate`. Also, problems could be encountered if it were possible for a target to add dependencies for a target other than itself. `redo` eliminates this possibility by only allowing dependencies for a particular target to be declared inside the corresponding target's build script.

`redo`'s purely top-down style of rebuilding allows for an enormous amount of parallelization compared to existing build tools. Because a `redo-ifchange` command builds each of the requested dependencies in parallel, entire subtrees of the dependency graph can be built in parallel. This is different than existing build tools, such as `make` and SCons, which only build different nodes of the dependency graph in parallel. One consideration to keep in mind might be that `redo` needs a way to limit parallelization. Some commands, such as linking a set of object files and libraries together into an executable, can require a considerable portion of the system's memory. If multiple link commands happen to be executed in parallel, the system may run out of memory.

4.3 Summary

This chapter has described in detail the behaviour of D. J. Bernstein's `redo` tool concept, elaborating on the description provided in his web pages. It has outlined the major features and explained the difference between the approach used by `redo` and the approaches used by other tools, such as `make`. It has discussed how these features fit together and combine to make purely top-down software rebuilding possible.

Chapter 5

Implementation of a `redo` prototype

This chapter discusses the details of a prototype implementation, written by the author of this thesis, of D. J. Bernstein's `redo` tool concept. It explains the types of data stored between rebuilds, design decisions faced, and limitations of the prototype. It also presents some directions for future work related to `redo` implementations.

5.1 Overview

The overriding design goal of this prototype is to produce correct build results while being as simple as possible. The prototype consists of approximately 250 lines of Bourne shell code (available as Appendix A).

5.2 Metadata stored by `redo`

`redo`, unlike `make`, records explicit state between builds in a file (or perhaps a directory) called `.redo`. Bernstein does provide any details about what pieces of information are stored or the format of this file. In the prototype described here, this database is modeled as an associative array, with keys that map filenames to a data value or list of data values. For both source and target files,

- `type` is either "s" (for source) or "t" (for target);
- `md5` is a string containing the MD5 cryptographic hash of the file, such as "feb254e8f28fdcf679414457248e4fa1."

For targets only,

- `prereqs` is a list of prerequisites; the target file should be rebuilt if any of these prerequisites are modified or removed;
- `prereqsnonexist` is another list of prerequisites; the target file should be rebuilt if any of these prerequisites are created;
- `result` is an integer between 0 and 254 representing the exit code of the target file's build script.

For files which are neither sources nor targets, but have been mentioned in a call to `redo-ifcreate`, an empty `nonexist` key is created for that file. This allows subsequent runs to properly detect dependencies on nonexistent files and rebuilt targets appropriately. `redo` could also possibly store the output or error messages from building a particular target in `.redo`; this is discussed in more detail below in the subsection titled "Error handling."

It is nontrivial to implement associative arrays efficiently in a persistent, disk-based format. One option is to use a database package like Berkeley DB or UNIX's DBM. However, for simplicity, I implemented the database as a collection of text files, where the filename is the key and the contents of the file are the value. Lists of values are stored one value per line in the file. For example, the prerequisites for a file called `maildir.o` are stored in `.redo/maildir.o.prereqs`, whose contents might be

```
maildir.h
buffer.h
```

5.3 Algorithm

Part 1 shows the beginning of the algorithm used by `redo`. When the `redo` command is invoked, it attempts to build the top-level target named `it`, by invoking `redo-ifchange`. A limitation of it is that it always rebuilds the top-level `it` target; it cannot rebuild a subset of arbitrary targets. Notice that all the `uptodate` keys are removed prior to starting; this forces `redo` to assume that each file is out of date until it actually examines the file, verifies whether or not it is up to date, and records the decision.

Part 2 shows the beginning of `redo-ifchange`. When the procedure is invoked, it first creates the `.redo` database if it does not already exist. It then loops over all the files passed as arguments, rebuilding each of them. Limitation: My prototype rebuilds these files in series, rather than parallel.

Algorithm 1 `redo`, part 1: REDO procedure

```
1: procedure REDO(args)
2:     delete *.uptodate
3:     redo-ifchange it
4: end procedure
```

The program next determines whether the current file being rebuilt is a source or a target. If there is no record of whether this file is a source or target, we check if the file exists: if it does, we assume it is a source file; if not we assume it is a target. Finally, if there is already an `uptodate` entry for this target in the database, we can assume this target has already been rebuilt and can stop building.

Algorithm 2 `redo`, part 2: REDO-IFCHANGE procedure

```
5: procedure REDO-IFCHANGE(args)
6:     create .redo database if it does not already exist
7:     for each argument i do
8:         if i.type does not exist then
9:             if i exists then
10:                i.type ← source
11:            else
12:                i.type ← target
13:            end if
14:        end if
15:        if i.uptodate exists then
16:            record i as a regular prerequiste for its parent
17:            continue
18:        end if
```

In Part 3, we handle the case where we have been asked to rebuild a source file. In this case, no actual build commands are issued; we only need to determine whether or not the file is up to date. If we have already stored an MD5 hash for this file, we calculate the MD5 hash of the file's current contents and compare the two. If they match, we record that the file is up to date. If no previous MD5 hash exists, we record the file is out of date. We also check if a special key has been created to indicate that one or more targets depend on the nonexistence of this file, and if this key exists we remove it. (This key will be created again if the file is removed.) Finally, we can exit, since there is nothing left to be done to rebuild a source file.

Algorithm 3 `redo`, part 3: continuation of REDO-IFCHANGE procedure

```
19:     if i.type = source then
20:         if i.md5 exists and it matches the current MD5 hash then
21:             i.uptodate ← yes
22:         else
23:             i.uptodate ← no
24:             if i.nonexist exists then
25:                 delete i.nonexist
26:             end if
27:         end if
28:         i.md5 ← current MD5 hash
29:         record i as a regular prerequiste for its parent
30:         continue
31:     end if
```

If we have made it to Part 4, we know the file is a target file. Thus, it can have two types of prerequisites: regular and nonexistent. There is a list of files for each prerequisite type; these lists were created dynamically the last time the target's build script was run. The idea behind this part of the algorithm is to calculate a boolean value, `uptodate`, for this target, which tells us whether or not we can skip running the actual build script this time. We can examine each file in the list of regular prerequisites, `prereqs`, rebuilding it if necessary (shown in Part 6). If all of these files are up to date, then we can consider the current target up to date as well, without building it. This works because the build script itself is listed as a prerequisite; the key to this algorithm is that if the build commands are the same, and the inputs to those commands are the same, then the output must be the same.

We can then examine each of the files in the list of nonexistent prerequisites, `prereqsnonexist`. If all of these files are still missing, then the current target must be up to date. (Note that this list is also one of the prerequisites for the target, so if the list changes then the target will be properly rebuilt.) Finally, if the current target is up to date, we can exit.

If we have made it to Part 5, we know we have a target file that is out of date, and therefore must be rebuilt. This part of the algorithm determines which build script to use. The idea is that we want to use the specialized build script if it exists, but fall back to the generic build script for our target type otherwise. The complication is that we would like to ensure that if we start off using the generic build script, we can automatically switch to the specialized one for if it is created. Finally, if no appropriate build script exists for this

Algorithm 4 `redo`, part 4: continuation of REDO-IFCHANGE procedure

```
32:     if i.prereqs exists then
33:         i.uptodate ← yes
34:         for each file j in i.prereqs do
35:             if j.uptodate does not exist then
36:                 redo-ifchange j
37:                 if j.uptodate = no then
38:                     i.uptodate ← no
39:                 end if
40:             end if
41:         end for
42:     end if
43:     if i.prereqsnonexist exists then
44:         for each file j in i.prereqsnonexist do
45:             if j exists then
46:                 i.uptodate ← no
47:             end if
48:         end for
49:     end if
50:     if i.uptodate = yes then
51:         break
52:     end if
```

target, we have an error.

Part 6 is the last part of the `redo-ifchange` procedure. Its purpose is to run the build script determined in the previous part, and to record the result. If there is no error and the new MD5 hash matches the old MD5 hash, we can mark this target as up to date. (This may prevent targets that depend on it from being rebuilt.) Otherwise, we record the new MD5 hash and the status of the target remains out of date.

Part 7 shows the logic for `redo-ifcreate`, which is very simple compared to `redo-ifchange`. If the current file exists, we have an error because the fact that this procedure was invoked means some other file is depending on the current file's nonexistence. Otherwise, we record it as a prerequisite for the appropriate parent, delete any previous MD5 hash (in case the file existed at one point), and create a `nonexist` key for this file.

Algorithm 5 `redo`, part 5: continuation of REDO-IFCHANGE procedure

```
53:         if build file i.do exists then
54:             redo-ifchange i.do
55:             i.buildfile ← i.do
56:         else
57:             calculate filename for default build script
58:             if if default build script exists then
59:                 redo-ifchange default
60:                 redo-ifcreate i.do
61:                 i.buildfile ← default
62:             else
63:                 error: no build script for i
64:             end if
65:         end if
```

Algorithm 6 `redo`, part 6: continuation of REDO-IFCHANGE procedure

```
66:         execute the build script for i and store the result
67:         if result ≠ 0 then
68:             error: build failed
69:         end if
70:         if i.md5 exists and matches the current MD5 hash then
71:             record i.uptodate = yes
72:         end if
73:         record i.md5 = current MD5 hash
74:     end for
75:     record i as a regular prerequiste for its parent
76: end procedure
```

5.4 Design decisions

I faced several major design decisions when implementing my `redo` prototype. The choices made directly affect the efficiency, reliability, and complexity of the prototype. In this section, I explain the alternatives considered for each of these decisions, and the trade-offs between them.

5.4.1 Determining whether a target is up to date. Bernstein explains in his web pages that `redo` does not use the newness of a file to determine whether or not it is up to date. He goes on to say that a "target file isn't considered up to date until `redo` finishes and atomically records what happened

Algorithm 7 `redo`, part 7: REDO-IFCREATE procedure

```
77: procedure REDO-IFCREATE(args)
78:     for each argument i do
79:         if i exists then
80:             error
81:         end if
82:         record i as a nonexistent prerequiste for its parent
83:         delete i.md5
84:         create i.nonexist
85:     end for
86: end procedure
```

in `.redo`."[26] As described in the previous section, the prototype calculates the MD5 hash of the file and compares it to the previous MD5 hash. If the MD5 hashes match, the file is up to date. The MD5 hash for each file is stored in `.redo`. MD5 is suitable for this purpose because it is fairly fast to calculate and, for all practical purposes, two files will not have the same MD5 hash unless they have the same contents.

There is a problem with this approach, however. Consider the case where only a subset of the targets are built in a particular run. For example, suppose both `A` and `B` depend on `Z`. Suppose that first `A` is built, then `Z` is modified, then `B` is built. Now when `A` is rebuilt, `redo` incorrectly believes `A` is up to date because its only prerequisite, `Z`, has not changed since the previous run. The correct way to handle this case is to not store a single MD5 hash for each file, but to instead store an MD5 hash for each prerequisite. This way, if a particular file is a prerequisite for two targets, its MD5 hash is stored in two different places in `.redo`. If one of the targets is rebuilt, the MD5 hash of the prerequisite is updated to reflect the state of that target.

For simplicity, my prototype only stores a single MD5 hash for each prerequisite and requires that all targets be rebuilt each time `redo` is run.

5.4.2 Communication between parent and child nodes. One tricky part of implementing `redo` is managing communication between parent target files and their children. Recall that a dependency on a child is specified implicitly by invoking a regular program, `redo-ifchange`, from within the parent's build script. Because `redo-ifchange` does not take any other parameters than the files to rebuild, there is no obvious way for `redo-ifchange` to record the dependency information in `.redo`.

For example, consider the case where a target called `A` depends on two

children, `Y` and `Z`. Target `B` also depends on `Z`. The build script for `A` contains a call

```
redo-ifchange Y Z
```

while the build script for `Z` contains a call

```
redo-ifchange Z
```

Now, rebuilding `A` will cause `Y` to be rebuilt. After `Y` is rebuilt, we would like to record that `Y` is a prerequisite of `A`. Furthermore, rebuilding `A` will also cause `Z` to be rebuilt. But `Z` may already be up to date if `B` has already been rebuilt, since `B` also depends on `Z`. Either way, we would like the end result to be that `Z` is recorded as a prerequisite for both `A` and `B`.

A naiive solution might be create a convention that the last parameter to `redo-ifchange` is not the name of a prerequisite, but the name of the current target. While this would work, it is redundant. Even worse it changes the specification of `redo-ifchange` and adds unnecessary complexity.

A second solution is to use an environment variable to store the current target being built; say, `$REDOPARENT`. Each time the build script for a target is executed by `redo-ifchange`, `$REDOPARENT` is set appropriately. One drawback with this approach is that it clutters the global environment namespace. Another is that it is also potentially unreliable—programs or build scripts could clear or change this variable.

A third solution to this problem is to use an associative array in `.redo` to map the UNIX process ID of each running build script to the name of the target it is building. When the child process `redo-ifchange` is executed, it can use the `getppid()` system call to find out the UNIX process ID of its parent. The child process can then determine the name of the parent using the associative array in `.redo`, and properly record itself as a prerequisite. Although this solution is likely the best, communication between parent and child was implemented using an environment variable, since this is simpler to implement.

5.5 Future work

There are several areas where future work could be focused on my `redo` prototype.

5.5.1 Concurrency At present, my prototype does not rebuild targets in parallel as described by Bernstein. However, parallel rebuilding would likely not be overly difficult to implement. Since different targets are built by separate UNIX processes, these processes can naturally run in parallel with one another. However, care must be taken to ensure that the processes do not interfere with one another. Prior to starting rebuilding of a target, the process must obtain an exclusive lock for that target. Upon completion of the build, the process releases the lock file. If the process was not able to obtain the lock, it knows that another process is currently rebuilding the target.

5.5.2 Error handling Error handling is an area of my prototype that could be improved in the future. At present, `redo` will exit when it encounters an error while executing a build script. However, if my prototype were to execute multiple build scripts in parallel, it would be desirable to continue as far as possible in each branch of the build.

Another issue with error handling, and build output in general, is the user interface. With multiple build scripts executing at once, output may appear interleaved in the terminal. It would be better if `redo` saved a copy of the build output for each target in a separate file, making it easy for the user to examine what happened at a later time.

5.5.3 Target cache As discussed in Chapter 2, target caches currently exist in for certain types of target files, such as those produced by C compilers (*e.g.* `ccache`). More general-purpose target caches also exist in integrated build tools such as `Vesta`. It may be useful to add a similar facility to `redo`. In large systems, changes are occasionally made and later reverted; a cache would save time in this situation because it would enable `redo` to "remember" versions of targets beyond the last issued build. Furthermore, if it was possible to share such a cache among developers, then this would reduce the time necessary to build the system from scratch for the first time.

5.5.4 Different scripting languages Some users may wish to write build scripts in a language other than Bourne shell. One reason for this might be that many software systems are developed on both UNIX and Windows simultaneously, and the Windows platform does not typically contain a port of the Bourne shell. A scripting language such as Python, Perl, or Ruby could be used; however, it is likely that build speed will be reduced because these interpreters take longer to start up than Bourne shell, and `redo` is designed run a separate interpreter process for each target.

In my prototype, while the default interpreter is currently the Bourne shell, other interpreters could be used by invoking them on the command line. For example, a simple Python script can be passed to the Python interpreter with the `-c` option:

```
python -c 'print "hello, world"'
```

A more complex script could be stored in a separate file and used via

```
redo-ifchange complex.py
python complex.py
```

5.5.5 Batch tools Some compilers operate more efficiently by processing several sources at once to produce several targets. For example, the Visual C++ 8 compiler performs considerably better if run in batch mode. However, these types of tools pose a problem for `redo` because each build scripts must produce exactly one target by writing the data to standard output.

One solution to this that would work with the current semantics of `redo` would be to write a build script that produces a single composite target, such as a JAR or ZIP file, that actually contains several targets concatenated. (The Odin build tool also uses composite targets[41].) The individual targets may then be extracted in subsequent build steps. Unfortunately, this approach would require twice the storage space. However, a key benefit of `redo`—atomic creation of targets—is maintained.

5.5.6 Dependency on environment At present, `redo` does not rebuild targets when variables in the user's environment change. Although changes to these environment variables can result in a different set of derived objects, this is often not the case. An example of a variables whose values can directly affect build results are the various search paths; for example, `$PATH`. However, other variables, like `$OLDPWD` should almost certainly not influence the build result.

5.5.7 Speed The goal of my `redo` prototype is to serve as a proof-of-concept implementation of `redo`. As a result, it was written to be as simple as possible, without seriously considering the speed of the program. It would be beneficial in the future write a proper implementation of `redo` to run at high-speed. This would allow more comprehensive tests to be done to verify the suitability of `redo` as a build tool for software systems with millions of lines of code.

5.6 Summary

In this chapter, I have discussed the details of a prototype I have written of D. J. Bernstein's `redo` build tool. I have explained the central build algorithm and how control flows through the program. I have discussed the design decisions made with respect to metadata storage, communication, and determining whether a file is out of date. Lastly, I have discussed directions for future work on the prototype.

Chapter 6

Results and evaluation

This chapter describes some results of an evaluation of `redo` on some sample software systems. While a large-scale, in-depth case study is beyond the scope of this thesis, the evaluation in this chapter aims to provide preliminary feedback about

- the amount of effort required to migrate between `redo` and an existing build tool;
- the size and complexity of a build system that uses `redo` compared to the size and complexity of a functionally equivalent build systems that uses an existing build tool; and
- the speed of full, incremental, and null rebuilds with `redo` compared to the speed of these types of builds with an existing tool.

Note that the third point is an assessment of the speed of the particular `redo` prototype described in the thesis, which is not designed to run at high-speed. A high-speed implementation of `redo` would run much faster. The reason for assessing the speed of the prototype is simply to determine if it is tolerable for use in its current form.

All speed tests were done on a system running Ubuntu 6.10 with a Core 2 Duo 2.13GHz processor, 2GB RAM, and a 7200RPM disk. A full rebuild is a rebuild from scratch with no existing targets. A null rebuild is a rebuild when nothing has changed, and hence no targets are rebuilt. A minor rebuild is a build when only a few files have changed; in this case I changed 2 source files and this resulted in 11 target files getting rebuilt.

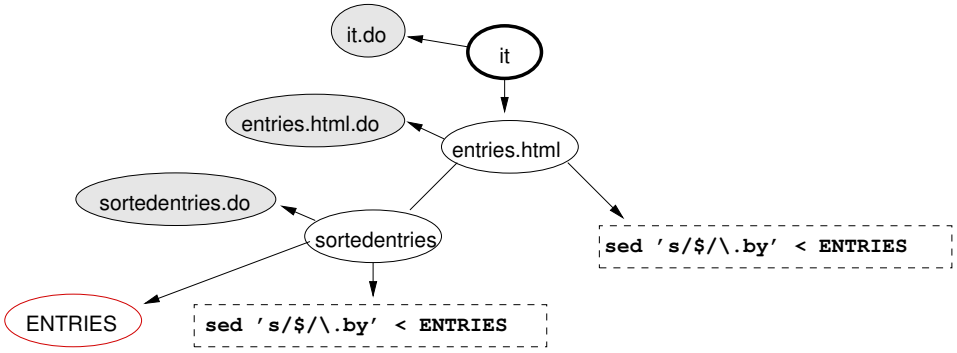

Figure 6.1: Dependencies in an online bibliography system, part 1

6.1 Online bibliography system

This section focuses on an online bibliography system: a set of scripts written by D. J. Bernstein to maintain a list of bibliographic references for the web in a variety of forms. This system is unique because it is designed to build with `redo`, rather than another build tool, and hence it takes advantage of `redo`-specific features such as dynamic dependencies and dependencies on nonexistent files.

The interface to the system is relatively simple. The user maintains a directory of files, where each file is a raw bibliography entry. The syntax of the raw bibliography entries is straight TEX using a set of macros. These raw entries are then processed to produce properly cross-referenced TEX entries, as well as BibTeX entries and HTML snippets. The HTML snippets are then assembled together to form the final web page, which links to the generated TEX and BibTeX entries.

Figure 6.1 shows an overview of the top-level dependencies in the system. Note that certain child nodes in the dependency graph are determined dynamically during the build from the output of UNIX commands that take as input the values of sibling nodes. Figure 6.2 shows the complete dependency graph produced after `redo` works its way down the tree. Note that certain source nodes may or may not exist. If they do not exist, `redo-ifcreate` will remember so if they are created at a later time by the user, they will be incorporated into the next build.

6.1.1 Migration effort The author of this thesis migrated the build scripts for the online bibliography maintenance system from `redo` to `make`. There were several phases involved:

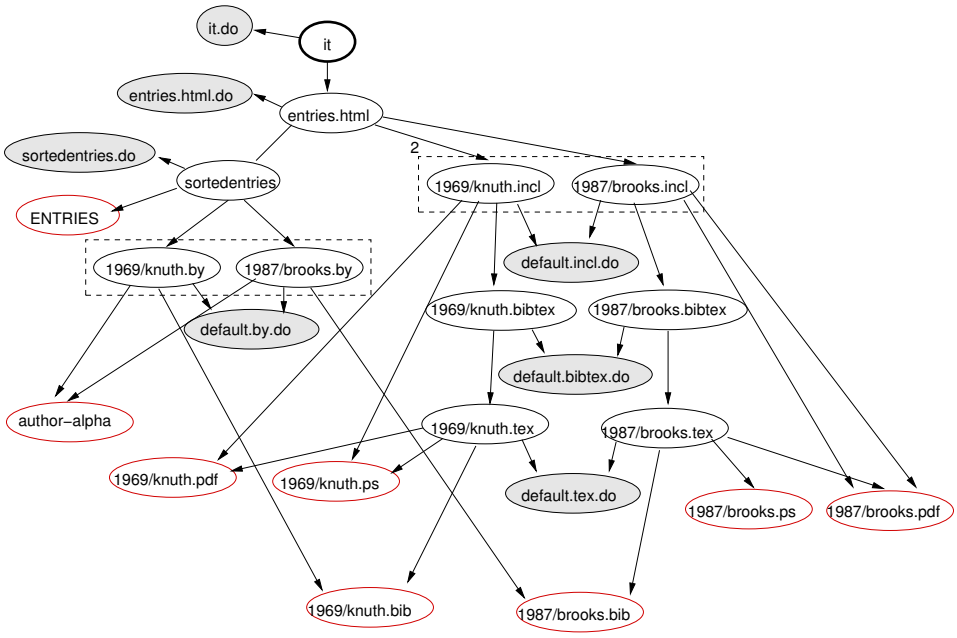

Figure 6.2: Dependencies in an online bibliography system, part 2

1. A `Makefile` was created to encode the dependencies expressed by calls `redo-ifchange` in the `.do` scripts.

2. Each of the `.do` scripts was converted to a `make` script by removing all calls to `redo-ifchange` and `redo-ifcreate`.

3. Any dynamic dependency information was moved to the start of the `Makefile` where it is computed at the start of the build and stored in variables.

The following `Makefile` was obtained:

In total, approximately two hours were spent doing the migration. However, it should be noted that the `make`-based build system has flaws that the `redo`-based build system does not have. The most significant flaw is that not all of the dynamic dependencies were able to be captured at this time, such as the possible dependencies of processed bibliography entries on files that map journal names to ISSNs. In order to capture this, more time would need to be spent to figure out how to pre-compute this information for storing in a `Makefile` variable. Another flaw is that pre-computing values to avoid dynamic dependencies results in extra work being done for each build. In order to avoid this, the GNU `make`-specific technique described in [122, Section 4.14]

```
SHELL = /bin/sh

ENTRIES = $(shell cat ENTRIES)
ALLBY = $(patsubst %,%.by,$(ENTRIES))
SORTEDENTRIES = $(ENTRIES)
ALLINCL = $(patsubst %,%.incl,$(ENTRIES))
ALLBIB = $(patsubst %,%.bib,$(ENTRIES))

all: entries.html

entries.html: $(ALLINCL)
./make-entries-html $(SORTEDENTRIES) > $@; chmod 644 $@

%.incl: %.tex %.bibtex
./make-incl x $* > $@; chmod 644 $@

%.bibtex: %.tex
./make-bibtex x $* > $@; chmod 644 $@

%.tex: %.bib
./make-tex x $* > $@; chmod 644 $@

clean:
-rm -f entries.html */*.incl */*.bibtex */*.tex

.PRECIOUS: %.incl %.bibtex %.tex
```

Figure 6.3: Migrated `Makefile` for an online bibliography system

Build type	`redo` prototype	`make`
Full	1m28s	0m15s
Null	0m26s	0m01s
Minor	0m28s	0m01s

Table 6.1: Speed measurements comparing the `redo` prototype with `make` for an online bibliography system

could be used. However, this technique is not as straightforward and would certainly increase the effort required for migration.

6.1.2 Size and complexity The original `redo`-based build scripts were 389 lines of code in total. The migrated `make`-based build scripts were 425 lines of code in total. Most of the `.do` scripts needed to be modified only slightly in order to be driven by a `Makefile` rather than `redo`. Hence, the difference between the two build systems was the `Makefile`, which was the most complex part of the new `make`-based build system.

6.1.3 Speed tests Table 6.1 shows some speed measurements from using the `redo` prototype to rebuild an online bibliography with 283 entries; this translates to 500 source files and 1135 target files.

The table shows that it takes around one and a half minutes to rebuild the bibliography from scratch with `redo`. While this is not fast, it is likely acceptable for most users for general use. Null rebuilds and minor rebuilds take around half a minute; again this is usable. Although `make` builds much faster, reliability is sacrificed since `make` will not notice all types of changes (for example, adding a PDF mirror of an entry). This could result in the user frequently having to run `make clean` to be ensure correct results.

6.2 Links web browser

This section focuses on the Links web browser, which consists of 27,385 physical lines of code. Links is a lightweight text-only web browser that supports colour terminals, bookmarks, customizable keys, multiple languages, and multiple character sets. Links is written entirely in C, although it uses four custom scripts to generate header files that allow the use off different character sets and languages.

6.2.1 Migration effort The author of this thesis migrated the build scripts for Links from `make` to `redo`. There were several phases involved:

1. A top-level `it.do` build script was created that triggers the `links` executable to be rebuilt.

2. A `links.do` build script was created that triggers each of the 35 object files to be rebuilt.

3. A `default.o.do` build script was created that triggers a corresponding file containing dependency information to be rebuilt. Each of the dependencies listed within is then triggered to rebuild.

4. A `default.deps.do` build script was created that scans the corresponding C source file for header dependencies.

5. Four C fragments are generated from character set and language tables to be included in `character.c` and `language.c`, respectively. These are triggered to rebuild when `character.o` and `language.o` are rebuilt, since they are listed as dependencies in `character.deps` and `language.deps`.

Overall, it did not require a great deal of effort to migrate to `redo` once the existing build process was understood.

6.2.2 Size and complexity The `make`-based build system relies on `automake` to first generate a 492-line `Makefile.in` template from a 27-line `Makefile.am` skeleton. `autoconf` then generates a 4531-line `configure` script from a 248-line `configure.ac`. There are also 110 lines of scripts for generating Unicode and internationalization information.

By comparison, all the `.do` scripts are 28 lines total, and no large artifacts are generated. This suggests a `redo`-based build system may be easier to debug. Granted, the `redo`-based build system does not provide all the custom configuration tests, although it is unclear exactly how useful these tests are on for most machines.

6.2.3 Speed tests Table 6.2 shows some speed measurements from using the `redo` prototype to rebuild the Links web browser; this compiles 35 object files and links them together in one executable file.

This table shows that the `redo`-based build system is almost as fast as the `make`-based build system. However, it is difficult to get an accurate comparison of the two build systems because the `make`-based build system runs

Build type	`redo` prototype	`make`
Full	0m33s	0m28s
Null	0m11s	0m04s
Minor	0m05s	0m00s

Table 6.2: Speed measurements comparing the `redo` prototype with `make` for the Links web browser

a long configuration script at the beginning of the build to query the host machine, and these tests were not migrated over to the `redo`-based build system. However, this configuration script is generated by separate tools prior to distribution along with `Makefile` containing pre-generated dependencies for the object files, and this part of the `make`-based build system was not counted while object file dependency generation was counted for `redo`. Overall, the key observation is that the `redo` prototype is within the same order of magnitude as `make`.

6.3 Discussion

The author's experience seems to show that a quick migration from one build tool to another retains the idioms of the old tool, which may not be as easily expressed in the new tool. This results in increasing the total size of the new build system. Spending more time on the migration to map the idioms of the old tool properly onto the new one could produce a simpler, more compact build system, but at the cost of increased effort. For large systems, it is likely that it would be too labour intensive to manually migrate from one build tool to another; tools to support automated or semi-automated migration would be necessary.

The slow speed of the `redo` prototype when compared with other build tools is not surprising. The prototype was not designed to run at high-speed; rather it is a proof-of-concept for `redo` and the purely top-down approach to rebuilding. That said, the speed of the prototype appears adequate for use with smaller systems, which indeed the author of this thesis has found to be the case while maintaining an online bibliography on a daily basis.

For future work, it would be interesting to compare `redo`-based build systems with corresponding SCons-based and Ant-based build systems, since they are less similar to `redo` than `redo` is to `make`.

6.4 Summary

This chapter has described some results of an evaluation of `redo` on some sample software systems. These results provide feedback about the following properties of `redo` compared to existing tools: effort required for migration, build system size and complexity, and speed of different types of rebuilds.

Chapter 7

Conclusion

This thesis has explained the concept of purely top-down software rebuilding, wherein build scripts are executed and a dependency graph is built on-the-fly. It has explained how D. J. Bernstein's `redo` tool concept makes use of this approach, along with several other novel features such as dependencies on nonexistent files, to provide a simpler and more consistent model of software rebuilding. It has presented a `redo` prototype written by the author of this thesis, and discussed the algorithms and data structures used by it, as well as design decisions made and directions for future work on the prototype. It has also evaluated the prototype on some sample systems and investigated how much effort is required to migrate between `redo` and other build tools, as well as the size, complexity, and performances aspects of the resulting build systems.

A purely top-down approach to software rebuilding represents a significant departure from the approaches used by conventional build tools over the past thirty years, and offers promising benefits in terms of reducing accidental complexity of build infrastructure for large-scale projects. The goal of this thesis was to provide a starting point for researchers and practitioners interested purely top-down rebuilding; such persons are encouraged to take further steps in developing purely top-down build tools, applying them to large-scale software projects, and evaluating them against competing approaches.

Appendix A

Source code

Listing A.1: redo

```
#!/bin/sh
#
# Alan Grosskurth
# http://grosskurth.ca/xredo/20070117/redo
# Public domain
#

[ -d .redo ] || mkdir .redo
find .redo -name '*.uptodate' -exec rm -f '{}' \;
find .redo -name '*.prereqs.build' -exec rm -f '{}' \;
find .redo -name '*.prereqsne.build' -exec rm -f '{}' \;
redo-ifchange it
```

Listing A.2: redo-ifcreate

```
#!/bin/sh
#
# Alan Grosskurth
# http://grosskurth.ca/xredo/20070117/redo-ifcreate
# Public domain
#

msg_() {
  level_="$1: "
  shift
  case "$level_" in
    info*) level_=
  esac
  echo "redo-ifcreate: ${level_}$@" 1>&2
  case "$level_" in
```

```
16      error*) exit 111 ;;
17    esac
18  }
19
20  for i in "$@"; do
21    [ -d .redo/"`dirname $i`" ] || mkdir .redo/"`dirname $i`"
22    if [ -e "$i" ]; then
23      msg_ error "$i exists"
24    fi
25
26    rm -f .redo/"$i.md5"
27    touch .redo/"$i.nonexist"
28
29    case "$REDOPARENT" in
30      '') error_ "$i: no parent" ;;
31      *)
32        ( if [ ! -e .redo/"$REDOPARENT.prereqsne.build" ]; then
33            echo "$1"
34          elif ! grep "$1" .redo/"$REDOPARENT.prereqsne.build" \
35          >/dev/null 2>/dev/null; then
36            cat .redo/"$REDOPARENT.prereqsne.build"
37            echo "$1"
38          fi
39        ) > .redo/"$REDOPARENT.prereqsne.build"'{new}'
40        mv .redo/"$REDOPARENT.prereqsne.build"'{new}' \
41        .redo/"$REDOPARENT.prereqsne.build"
42        ;;
43    esac
44  done
```

Listing A.3: redo-ifchange

```
1   #!/bin/sh
2   #
3   # Alan Grosskurth
4   # http://grosskurth.ca/xredo/20070117/redo-ifchange
5   # Public domain
6   #
7
8   msg_() {
9     level_="$1: "
10    shift
11    case "$level_" in
12      info*) level_=
13    esac
14    echo "redo-ifchange: ${level_}$@" 1>&2
15    case "$level_" in
16      error*) exit 111 ;;
```

```
17    esac
18  }
19
20  record_() {
21    echo "$1" > .redo/"$2"'{new}'
22    mv .redo/"$2"'{new}' .redo/"$2"
23  }
24
25  record_prereq_() {
26    case "$REDOPARENT" in
27      '') : ;;
28      *)
29        ( if [ ! -e .redo/"$REDOPARENT.prereqs.build" ]; then
30            echo "$1"
31          elif ! grep "$1" .redo/"$REDOPARENT.prereqs.build" \
32          >/dev/null 2>/dev/null; then
33            cat .redo/"$REDOPARENT.prereqs.build"
34            echo "$1"
35          fi
36        ) > .redo/"$REDOPARENT.prereqs.build"'{new}'
37        mv .redo/"$REDOPARENT.prereqs.build"'{new}' \
38        .redo/"$REDOPARENT.prereqs.build"
39        ;;
40    esac
41  }
42
43  for i in "$@"; do
44    [ -d .redo/"`dirname $i`" ] || mkdir .redo/"`dirname $i`"
45
46    if [ -e .redo/"$i.uptodate" ]; then
47      record_prereq_ "$i"
48      continue
49    fi
50
51    # Determine file type
52    if [ -e .redo/"$i.type" ]; then
53      type=`head -1 .redo/"$i.type"`
54    else
55      if [ -e "$i" ]; then
56        type=s
57      else
58        type=t
59      fi
60      record_ "$type" "$i.type"
61    fi
62
63    uptodate=n
64    case "$type" in
```

```
65      s)
66        # Check MD5 checksum
67        if [ -e "$i" ]; then
68          newmd5=`md5sum "$i" | awk '{print $1}'`
69          if [ -e .redo/"$i.md5" ]; then
70            oldmd5=`head -1 .redo/"$i.md5"`
71            if [ "$newmd5" = "$oldmd5" ]; then
72              uptodate=y
73            fi
74          elif [ -e .redo/"$i.nonexist" ]; then
75            rm -f .redo/"$i.nonexist"
76          fi
77          record_ "$newmd5" "$i.md5"
78        fi
79        #case "$uptodate" in
80        #  n) msg_ info "out-of-date source $i" ;;
81        #  *) msg_ info "up-to-date source $i" ;;
82        #esac
83        record_ "$uptodate" "$i.uptodate"
84        record_prereq_ "$i"
85        continue
86        ;;
87    esac
88
89    ### Must be a target file
90
91    # Check regular prerequisites
92    if [ -e .redo/"$i.prereqs" ]; then
93      uptodate=y
94      while read -r line; do
95        if [ ! -e .redo/"$line.uptodate" ]; then
96          env REDOPARENT="$i" redo-ifchange "$line"
97        fi
98        # Check for build errors
99        if [ -e .redo/"$line.result" ]; then
100         case `head -1 .redo/"$line.result"` in
101           0) : ;;
102           *) msg_ error "$i: failed to rebuild prerequisite $line" ;;
103         esac
104       fi
105
106       if [ -e .redo/"$line.uptodate" ]; then
107         case `head -1 .redo/"$line.uptodate"` in
108           n)
109             uptodate=n
110             break
111             ;;
112         esac
```

```
      else
        msg_ ASSERT: "$i: no uptodate file for $line" 1>&2
        exit 111
      fi
    done < .redo/"$i.prereqs"
  fi

  # Check nonexistent prerequisites
  if [ -e .redo/"$i.prereqsne" ]; then
    while read -r line; do
      if [ -e "$line" ]; then
        uptodate=n
      fi
    done < .redo/"$i.prereqsne"
  fi

  case "$uptodate" in
    n)
      # Remove old prerequisites
      rm -f .redo/"$i.prereqs"
      rm -f .redo/"$i.prereqsnonexist"
      # Determine which build script to execute (rebuild it if necessary)
      if [ -e "$i.do" ]; then
        env REDOPARENT="$i" redo-ifchange "$i.do"
        buildfile="$i.do"
      else
        default=`echo "$i" | sed 's/.*\(\.[^\.]*\)$/default\1/'`
        if [ -e "$default.do" ]; then
          env REDOPARENT="$i" redo-ifchange "$default.do"
          env REDOPARENT="$i" redo-ifcreate "$i.do"
          buildfile="$default.do"
        else
          msg_ error "$i: no build script found"
        fi
      fi
      # Execute the build script
      basefile=`echo "$i" | sed 's/\..*$//'`
      env REDOPARENT="$i" \
      sh -e "$buildfile" 0 "$basefile" .redo/"$i---redoing" \
      > .redo/"$i---redoing"
      result="$?"
      record_ "$result" "$i.result"
      case "$result" in
        0)
          newmd5=`md5sum .redo/"$i---redoing" | awk '{print $1}'`
          if [ -e .redo/"$i.md5" ]; then
            oldmd5=`head -1 .redo/"$i.md5"`
            case "$newmd5" in
```

```
161               "$oldmd5")
162                 rm -f .redo/"$i---redoing"
163                 uptodate=y  # No change
164                 ;;
165               *)
166                 mv .redo/"$i---redoing" "$i"
167                 record_ "$newmd5" "$i.md5"
168                 ;;
169             esac
170           else
171             mv .redo/"$i---redoing" "$i"
172             record_ "$newmd5" "$i.md5"
173           fi
174           msg_ info "rebuilt $i" 1>&2
175           ;;
176         *)
177           rm -f .redo/"$i---redoing"
178           msg_ error "failed to rebuild $i"
179           ;;
180       esac
181       ;;
182   esac
183
184   if [ -e .redo/"$i.prereqs.build" ]; then
185     mv .redo/"$i.prereqs.build" .redo/"$i.prereqs"
186   fi
187   if [ -e .redo/"$i.prereqsne.build" ]; then
188     mv .redo/"$i.prereqsne.build" .redo/"$i.prereqsne"
189   fi
190   record_prereq_ "$i"
191   record_ "$uptodate" "$i.uptodate"
192 done
```

Bibliography

1. — (no editor), *UNIX time-sharing system: UNIX programmers manual, seventh edition, volumes 1, 2A, 2B*, Bell Telephone Laboratories, Murray Hill, New Jersey, 1979; reprinted as [2] and [3]. URL: `http://cm.bell-labs.com/7thEdMan/`.
2. — (no editor), *UNIX time-sharing system: UNIX programmers manual, seventh edition, volume 1*, Holt, Rinehart, and Winston, New York, 1983; previously printed as Volume 1 of [1]. ISBN 0–03–061742–1.
3. — (no editor), *UNIX time-sharing system: UNIX programmers manual, seventh edition, volume 2*, Holt, Rinehart, and Winston, New York, 1983; previously printed as Volumes 2A and 2B of [1]. ISBN 0–03–061743–X.
4. — (no editor), *Proceedings of the USENIX summer 1983 technical conference held in Toronto, July 13–15*, USENIX Association, Berkeley, 1983. See [75], [94].
5. — (no editor), *Proceedings of the 1st international conference on computer workstations held in San Jose, California, November 11–14, 1985*, IEEE Computer Society, Los Alamitos, 1985. ISBN 0–8186–0649–5. See [90].
6. — (no editor), *Proceedings of the USENIX summer 1985 technical conference held in Portland, Oregon, June 11–14*, USENIX Association, Berkeley, 1985. See [59].
7. — (no editor), *Proceedings of the USENIX summer 1986 technical conference held in Atlanta, Georgia, June 11–13*, USENIX Association, Berkeley, 1986. See [130].
8. — (no editor), *Proceedings of the USENIX summer 1987 technical conference held in Phoenix, Arizona, June 8–12*, USENIX Association, Berkeley, 1987. See [80].
9. — (no editor), *Proceedings of the USENIX summer 1992 conference held in San Antonio, Texas, June 8–12*, USENIX Association, Berkeley, 1992. See [128].
10. — (no editor), *Plan 9 programmer's manual: the manual, the documents*, 2nd edition, Bell Laboratories, Murray Hill, New Jersey, 1995; reprinted as [11].
11. — (no editor), *Plan 9 programmer's manual: the manuals, the documents*, 2nd edition, Harcourt, Orlando, Florida, 1995; see also newer version [12];

previously printed as [10]. ISBN 0–03–01742–3.
12. — (no editor), *Plan 9 programmer's manual: the manuals, the documents*, 3rd edition, Harcourt, Orlando, Florida, 2000; see also older version [11].
13. — (no editor), *Proceedings of the sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000) held in Tokyo, Japan, September 11–15, 2000*, IEEE Computer Society, Los Alamitos, 2000. ISBN 0–7695–0583–X. See [120].
14. — (no editor), *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM'01) held in Florence, Italy, November 07–09, 2001*, IEEE Computer Society, Washington, 2001. ISBN 0–7695–1189–9.
15. — (no editor), *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Association for Computing Machinery, New York, 2002. ISBN 1–58113–471–1. See [49].
16. — (no editor), *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on collaborative research*, International Business Machines, Toronto, 2003. See [143].
17. — (no editor), *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03) held in Amsterdam, The Netherlands, September 22–26, 2003*, IEEE Computer Society, Washington, 2003. ISBN 0–7695–1905–9. See [133].
18. — (no editor), *KJam Preview (0.35)*, Oroboro Interactive, Arlington, Massachusettes, 2006. URL: `http://www.oroboro.com/kjam/`. Citations in this document: §2.3.2.
19. David Abrahams, *Boost.Jam 3.1.13*, 2006. URL: `http://www.boost.org/tools/build/jam_src/`. Citations in this document: §2.3.2.
20. Rolf Adams, Walter Tichy, Annette Weinert, *The cost of selective recompilation and environment processing*, ACM Transactions on Software Engineering and Methodology **3**.1 (1994), 3–28. ISSN 1049-331X. Citations in this document: §2.5.
21. Glenn Ammons, *Grexmk: speeding up scripted builds*, in [64] (2006), 81–86. Citations in this document: §2.1.
dell'Informazione
22. Davide Ancona, Giovanni Lagorio, *Stronger typings for separate compilation of Java-like languages*, Technical report, Dipartimento di Informatica e Scienze, 2003; see also newer version [23]. URL: `http://www.macs.hw.ac.uk/DART/reports/D5.1/AL03a.pdf`.
23. Davide Ancona, Giovanni Lagorio, *Stronger typings for separate compilation of Java-like languages*, Journal of Object Technology **3**.6 (2004), 5–25; see also older version [22]. ISSN 1660–1769. URL: `http://www.jot.fm/jot/issues/issue_2004_06/article1/article1.pdf`. Citations in this document: §2.5.
24. Erik. H. Baalbergen, *Design and implementation of parallel make*, Computing

Systems **1**.2 (1988), 135–158. ISSN 0895–6340. Citations in this document: §2.3.1.

25. Luciano Baresi, Reiko Heckel (editors), *Fundamental approaches to software engineering, 9th international conference, FASE 2006, held as part of the joint European conferences on theory and practice of software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, proceedings*, Lecture Notes in Computer Science, 3922, Springer-Verlag, Berlin, 2006. ISBN 3–540–33093–3. See [73].
26. Daniel J. Bernstein, *Rebuilding target files when source files have changed*, Last modified: 2003.07.12; accessed: 2006.07.07 (2003). URL: `http://cr.yp.to/redo.html`. Citations in this document: §4.1, §5.4.1.
27. Eric Bezault (editor), *Gobo Eiffel 3.4: free and portable Eiffel tools and libraries*, 2005. URL: `http://gobo-eiffel.sourceforge.net/`. Citations in this document: §2.3.3.
28. Ellen Borison, *A model of software manufacture*, in [44] (1986), 197–220. URL: `http://grosskurth.ca/bib/entries.html#1986/borison`. Citations in this document: §2.1.
29. Ellen Borison, *Program changes and the cost of selective recompilation*, Ph.D. thesis, Technical Report CMU–CS–89–205, Computer Science Department, Carnegie Mellon University, 1989. Citations in this document: §2.5.
30. Jan Bosch, Charles Krueger (editors), *Software reuse: methods, techniques and tools: 8th international conference, ICSR 2004, Madrid, Spain, July 5–9, 2004, proceedings*, Lecture Notes in Computer Science, 3107, Springer-Verlang, Berlin, 2004. ISBN 3–540–22335–5. See [46].
31. Pearl Brereton, Paul Singleton, *Deductive software building*, in [52] (1995), 81–87. URL: `http://grosskurth.ca/bib/entries.html#1995/brereton`.
32. Frederick P. Brooks, Jr., *No silver bullet: essence and accidents of software engineering*, Computer **20**.4 (1987), 10–19. Citations in this document: §1.1.
33. Mark R. Brown, John R. Ellis, *Bridges: tools to extend the vesta configuration management system*, Research Report 108, Digital Systems Research Center, 1993. URL: `ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-108.pdf`.
34. Rick Bubenik, Willy Zwaenepoel, *Performance of optimistic make*, in [57] (1989), 39–48. Citations in this document: §2.3.1.
35. Rick Bubenik, Willy Zwaenepoel, *Optimistic make*, IEEE Transactions on Computers **41**.2 (1992), 207–217. ISSN 0018–9340. Citations in this document: §2.3.1.
36. Jim Buffenbarger, *A software tool for maintaining file and macro build dependencies*, Journal of Software Maintenance: Research and Practice **8**.6 (1996), 421–431. ISSN 1040–550X. URL: `http://grosskurth.ca/bib/entries.html#1996/buffenbarger`.
37. Craig Chambers, Jeffrey Dean, David Grove, *A framework for selective recompilation in the presence of complex intermodule dependencies*, Techni-

cal Report 94–09–07, Department of Computer Science and Engineering, University of Washington, 1994; see also newer version [38]. URL: `ftp://ftp.cs.washington.edu/tr/1994/09/UW-CSE-94-09-07.PS.Z`.

38. Craig Chambers, Jeffrey Dean, David Grove, *A framework for selective recompilation in the presence of complex intermodule dependencies*, in [104] (1995), 221–230; see also older version [37]. Citations in this document: §2.5.
39. Geoffrey M. Clemm, *The Odin system: an object manager for extensible software environments*, Ph.D. thesis, Technical Report CU–CS–314–86, Department of Computer Science, University of Colorado, 1986. Citations in this document: §2.3.2.
40. Geoffrey M. Clemm, *The Odin specification language*, in [142] (1988), 144–158. URL: `http://grosskurth.ca/bib/entries.html#1988/clemm`. Citations in this document: §2.3.2.
41. Geoffrey M. Clemm, *The Odin system*, in [52] (1995), 241–262. URL: `http://grosskurth.ca/bib/entries.html#1995/clemm`. Citations in this document: §2.3.2, §5.5.5.
42. Robert F. Cmelik, *Concurrent Make: a distributed program in Concurrent C*, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986. Citations in this document: §2.3.1.
43. Reidar Conradi (editor), *Software configuration management: proceedings of the ICSE '97 SCM-7 workshop, Boston, MA, USA, May 18–19, 1997*, Lecture Notes in Computer Science, 1235, Springer-Verlag, Berlin, 1997. ISBN 3–540–63014–7. See [141].
44. Reidar Conradi, Tor M. Didriksen, Dag H. Wanvik (editors), *Advanced programming environments: proceedings of an international workshop held in Trondheim, Norway, June 16–18, 1986*, Lecture Notes in Computer Science, 244, Springer-Verlag, Berlin, 1986. ISBN 0–387–17189–4. See [28].
45. James Duncan Davidson (editor) (editor), *Apache Ant 1.7.0*, 2006. URL: `http://ant.apache.org`. Citations in this document: §2.3.3.
46. Merijn de Jonge, *Decoupling source trees into build-level components*, in [30], 215–231; available as Technical Report UU–CS–2004–024, Institute of Information and Computing Sciences, Utrecht University. URL: `http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-2004/2004-024.pdf`. Citations in this document: §2.1.
47. Merijn de Jonge, *Build-level component-based software engineering*, Technical Report UU–CS–2004–046 (2004). URL: `http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-2004/2004-046.pdf`. Citations in this document: §2.1.
48. Merijn de Jonge, *Build-level components*, IEEE Transactions on Software Engineering **31**.7 (2005), 588–600. ISSN 0098–5589. Citations in this document: §2.1.
49. Mikhail Dmitriev, *Language-specific make technology for the Java programming language*, in [15] (2002), 373–385. URL: `http://www.dcs.gla.ac.uk/~misha/`

papers/p022-dmitriev.pdf. Citations in this document: §2.5.
50. Paul DuBois, *Software portability with imake*, 2nd edition, O'Reilly, Sebastopol, California, 1996. ISBN 1–56592–226–3. Citations in this document: §2.3.1.
51. Verlyn B. Erickson, John F. Pellegrin, *Build—a software construction tool*, AT&T Bell Laboratories Technical Journal **63**.6 (1984), 1049–1059. ISSN 0005–8580. URL: http://grosskurth.ca/bib/entries.html#1984/erickson. Citations in this document: §2.3.1.
52. Jacky Estublier (editor), *Selected papers from the ICSE SCM-4 and SCM-5 workshops on software configuration management*, Lecture Notes in Computer Science, 1005, Springer-Verlag, London, 1995. ISBN 3–540–60578–9. See [31], [41].
53. Jacky Estublier, *Software configuration management: a roadmap*, in [58] (2000), 279–289. URL: http://www.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finalestublier.pdf. Citations in this document: §2.3.1.
54. Stuart I. Feldman, *Make—a program for maintaining computer programs*, Software—Practice & Experience **9**.4 (1979), 255–265; alternate version, [55]. ISSN 0038–0644. URL: http://grosskurth.ca/bib/entries.html#1979/feldman-jour. Citations in this document: §2.3.1.
55. Stuart I. Feldman, *Make—a program for maintaining computer programs*, in [1] (1979), ; alternate version, [54]. URL: http://wolfram.schneider.org/bsd/7thEdManVol2/make/make.pdf. Citations in this document: §2.3.1.
56. Stuart I. Feldman, *Evolution of MAKE*, in [142] (1988), 413–416. URL: http://grosskurth.ca/bib/entries.html#1988/feldman. Citations in this document: §2.3.1.
57. Domenico Ferrari (chairman), *SIGMETRICS '89: Proceedings of the 1989 ACM SIGMETRICS international conference on measurement and modeling of computer systems held Oakland, California, May 23–26*, Association for Computing Machinery, New York, 1989. ISBN 0–89791–315–9. See [34].
58. Anthony Finkelstein (chairman), *Proceedings of the conference on the future of software engineering held in Limerick, Ireland, June 04–11, 2000*, Association for Computing Machinery, New York, 2000. ISBN 1–58113–253–0. See [53].
59. Glenn Fowler, *The fourth generation make*, in [6] (1985), 159–174. URL: http://www.research.att.com/~gsf/publications/make-1985.pdf. Citations in this document: §2.3.1.
60. Glenn Fowler, *A case for make*, Software—Practice & Experience **20**.S1 (1990), S1/35–S1/46. ISSN 0038–0644. URL: http://www.research.att.com/~gsf/publications/make-1990.pdf. Citations in this document: §2.3.1.
61. David Garlan (editor), *Proceedings of the 4th ACM SIGSOFT symposium on foundations of software engineering held in San Francisco, California, October 16–18, 1996*, Association for Computing Machinery, New York, 1996. ISBN 0–89791–797–9. See [62].

62. Carl A. Gunter, *Abstracting dependencies between software configuration items*, ACM SIGSOFT Software Engineering Notes **21**.6 (1996), 167–178; see also newer version [63]; also printed in [61]. ISSN 0163–5948.
63. Carl A. Gunter, *Abstracting dependencies between software configuration items*, ACM Transactions on Software Engineering and Methodology **9**.1 (2000), 94–131; see also older version [62]. ISSN 1049-331X. Citations in this document: §2.1.
64. Neelam Gupta (conference chair), Andy Podgurski (conference chair), *Proceedings of the 2006 international workshop on dynamic systems analysis, Shanghai, China, May 23–23*, Association for Computing Machinery, New York, 2006. ISBN 1–59593–400–6. See [21].
65. Chris B. Hanna, Roy Levin, *The Vesta language for configuration management*, Technical Report 107, DEC Systems Research Center, Palo Alto, California, 1993. URL: `ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-107.pdf`. Citations in this document: §2.4.
66. Peter Henderson (editor), *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments held in Boston, Massachusetts, November 28–30, 1988*, Association for Computing Machinery, New York, 1989. ISBN 0–89791–290–X. See [96].
67. Allan Heydon, Jim Horning, Roy Levin, Timothy Mann, Yuan Yu, *The Vesta-2 software description language*, Technical Note 1997–005c, Compaq Systems Research Center, 1997. URL: `http://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-1997-005c.html`. Citations in this document: §2.4.
68. Allan Heydon, Roy Levin, Timothy Mann, Yuan Yu, *The Vesta approach to software configuration management*, Research Report 1999–001, Compaq Systems Research Center, 1999; see also newer version [69]. URL: `ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-1999-001.pdf`.
69. Allan Heydon, Roy Levin, Timothy Mann, Yuan Yu, *The Vesta approach to software configuration management*, Research Report 168 (2001); see also older version [68]. URL: `ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-168.pdf`. Citations in this document: §2.4.
70. Allan Heydon, Roy Levin, Timothy Mann, Yuan Yu, *The Vesta software configuration management system*, Research Report 177, Compaq Systems Research Center, 2002; see also newer version [71]. URL: `ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-177.pdf`.
71. Allan Heydon, Roy Levin, Timothy Mann, Yuan Yu, *Software configuration management using Vesta*, Springer-Verlag, New York, 2006; see also older version [70]. ISBN 0–387–00229–4. Citations in this document: §2.4.
72. Allan Heydon, Roy Levin, Yuan Yu, *Caching function calls using precise dependencies*, ACM SIGPLAN Notices **35**.5 (2000), 311–320. ISSN 0362–1340. URL:

http://vestasys.org/doc/pubs/pldi-00-04-20.pdf. Citations in this document: §2.4.
73. Jason Hickey, Aleksey Nogin, *OMake: designing a scalable build process*, in [25] (2006), 63–78; see extended version [74].
74. Jason Hickey, Aleksey Nogin, *OMake: designing a scalable build process* (2006); see shorter version [73]. URL: http://caltechcstr.library.caltech.edu/551/01/omake-fase06-tr.pdf.
75. Edward S. Hirgelt, *Enhancing MAKE or re-inventing a rounder wheel*, in [4] (1983), 46–58. URL: http://grosskurth.ca/bib/entries.html#1983/hirgelt. Citations in this document: §2.3.1.
76. Gary Holt (editor), *Makepp 1.40.1a: Compatible but improved replacement for make*, 2004. URL: http://makepp.sourceforge.net. Citations in this document: §2.3.1.
77. Richard C. Holt, Michael W. Godfrey, Andrew J. Malton, *The build/comprehend pipelines*, Presented at the second ASERC workshop on software architecture held in Banff, Alberta, Canada, February 18–19, 2003 (2003). URL: http://plg.uwaterloo.ca/~migod/papers/aserc03.pdf. Citations in this document: §2.1.
78. Ian Holyer, Huseyin Pehlivan, *An automatic make facility*, Technical Report CSTR–00–001, Department of Computer Science, University of Bristol, 2000. URL: http://www.cs.bris.ac.uk/Publications/Papers/1000438.pdf. Citations in this document: §2.3.1.
79. Steve Holzner, *Ant: the definitive guide*, 2nd edition, O'Reilly, Sebastopol, California, 2005; see also older version [131]. ISBN 0–596–00609–8.
80. Andrew Hume, *Mk: a successor to make*, in [8] (1987), 445–457. URL: http://grosskurth.ca/bib/entries.html#1987/hume. Citations in this document: §2.3.1.
81. Andrew G. Hume, Bob Flandrena, *Maintaining files on Plan 9 with mk*, in [11] (1995). URL: http://cm.bell-labs.com/sys/doc/mk.pdf. Citations in this document: §2.3.1.
82. Harald Kirsch, *bras: another kind of 'make': user manual & reference, March 4, 2002*, 2002. URL: http://bras.berlios.de/bras.pdf. Citations in this document: §2.3.4.
83. Steven Knight, *Building software with SCons*, Computing in Science & Engineering **7**.1 (2005), 79–88. ISSN 1521–9615. Citations in this document: §2.3.4.
84. Brian Koehler, R. Nigel Horspool, *CCC: A caching compiler for C*, Software—Practice & Experience **27**.2 (1997), 155–165. ISSN 0038–0644. URL: http://www.csr.uvic.ca/~nigelh/Publications/ccc.pdf. Citations in this document: §2.3.5.
85. Gary K. Kumfert, Tom G. W. Epperly, *Software in the DOE: the hidden overhead of "the build"*, Release Number UCRL–ID–147343, U. S. Department of Energy, Lawrence Livermore National Laboratory, 2002. URL: http://www.

llnl.gov/tid/lof/documents/pdf/244668.pdf. Citations in this document: §1.1.

86. David Alex Lamb, *Abstraction problems in software manufacture*, Technical Report 1989–243, School of Computing, Queen's University, 1989. URL: http://www.cs.queensu.ca/TechReports/Reports/1989-243.pdf. Citations in this document: §2.1.
87. David Alex Lamb, *Relations in software manufacture*, Technical Report 1990–292, School of Computing, Queen's University, 1990. URL: http://www.cs.queensu.ca/TechReports/Reports/1990-292.pdf. Citations in this document: §2.1.
88. Butler W. Lampson, Eric E. Schmidt, *Organizing software in a distributed environment*, in [139] (1983), 1–13. Citations in this document: §2.3.5.
89. Stefan Lang (editor), *Rant 0.5.8: flexible, Ruby-based* `make`, 2006. URL: http://rant.rubyforge.org. Citations in this document: §2.3.4.
90. David B. Leblang, Robert P. Chase, Jr., Gordon D. McLean, Jr., *The DOMAIN software engineering environment for large-scale software development efforts*, in [5] (1985), 266–280. Citations in this document: §2.4.
91. Roy Levin, Paul R. McJones, *The Vesta approach to precise configuration of large software systems*, Technical Report 105, DEC Systems Research Center, Palo Alto, California, 1993. URL: ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-105.pdf.
92. Andrew Lih, Erez Zadok, *PGMAKE: A portable distributed make system*, Techical Report CUCS–035–94, Department of Computer Science, Columbia University, 1994. URL: http://www.am-utils.org/docs/pgmake/pgmake.pdf. Citations in this document: §2.3.1.
93. Andy Litman, *An implementation of precompiled headers*, Software—Practice & Experience **23**.5 (1993), 341–350. ISSN 0038–0644. URL: http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol23/issue3/spe817.pdf.
94. Martin J. McGowan, William L. Anderson, Allen H. Brumm, *Mm4—make with M4 for maintaining makefiles (talk summary)*, in [4] (1983), 59. URL: http://grosskurth.ca/bib/entries.html#1983/mcgowan. Citations in this document: §2.3.1.
95. Axel Mahler, Andreas Lampen, *shape—a software configuration management tool*, in [142] (1988), 228–243. URL: http://grosskurth.ca/bib/entries.html#1988/mahler. Citations in this document: §2.4.
96. Axel Mahler, Andreas Lampen, *An integrated toolset for engineering software configurations*, in [66] (1989), 191–200.
97. Ken Martin, Bill Hoffman, *Mastering CMake 2.2*, Kitware, Clifton Park, New York, 2006. ISBN 1–930934–16–5. Citations in this document: §2.3.1.
98. Robert Mecklenburg, *Managing projects with GNU make*, 3rd edition, O'Reilly, Sebastopol, California, 2004. ISBN 0–596–00610–1. Citations in this document:

§2.3.1.

99. Peter Miller, *Recursive make considered harmful*, Australian UNIX and Open Systems User Group Newsletter **19**.1 (1997), 14–25. ISSN 1035–7521. URL: `http://aegis.sourceforge.net/auug97.pdf`. Citations in this document: §2.3.1.
100. Webb Miller, Eugene W. Myers, *Side-effects in automatic file updating*, Software—Practice & Experience **16**.9 (1986), 809–820. ISSN 0038–0644. URL: `http://grosskurth.ca/bib/entries.html#1986/miller`. Citations in this document: §2.3.1.
101. Bram Moolenaar (editor), *A-A-P 1.089: install and develop software*, 2007. URL: `http://www.a-a-p.org`. Citations in this document: §2.3.4.
102. Tamiya Onodera, *Reducing compilation time by a compilation server*, Software—Practice & Experience **23**.3 (1993), 477–485. ISSN 0038–0644.
103. Andrew Oram, Steve Talbott, *Managing projects with make*, 2nd edition, O'Reilly, Sebastopol, California, 1991; see also older version [124]. ISBN 0–937175–90–0. Citations in this document: §2.3.1.
104. Dewayne Perry (chairman), *Proceedings of the 17th International Conference on Software Engineering*, Association for Computing Machinery, New York, 1995. ISBN 0–89791–708–1. See [38].
105. V. Ramji, Timothy. A. Gonsalves, *Distributed and optimistic make: implementation and performance*, In 11th annual Phoenix conference on computers and communication, April, 1992 (1992), 531–538. Citations in this document: §2.3.1.
106. E. S. Roberts, J. R. Ellis, *Parmake and Dp: experience with a distributed, parallel implementation of make*, in [140] (1987). Citations in this document: §2.3.1.
107. Mansur Samadzadeh, Mansour Zand (editors), *Proceedings of the 1995 symposium on software reusability held in Seattle, Washington, April 29–30*, Association for Computing Machinery, New York, 1995. ISBN 0–89791–739–1. See [119].
108. Hartmut Schirmacher, Stefan Brebec, *tmk—a multi-site, multi-platform system for software development*, Presented at the First European Tcl/Tk User Meeting, June 15–16, 2000, Hamburg (2000). URL: `http://www.tmk-site.org/doc/pub/tmk-devel.pdf`. Citations in this document: §2.3.4.
109. Robert W. Schwanke, Gail E. Kaiser, *Living with inconsistency in large systems*, in [142] (1988), 98–118. URL: `http://grosskurth.ca/bib/entries.html#1988/schwanke-inconsistency`.
110. Robert W. Schwanke, Gail E. Kaiser, *Smarter recompilation*, ACM Transactions on Programming Languages and Systems **10**.4 (1988), 627–632. Citations in this document: §2.5.
111. Zhong Shao, Andrew W. Appel, *Smartest recompilation*, Technical Report CS–TR–363–92, Department of Computer Science, Princeton University, 1992;

see shorter version [112]. URL: ftp://ftp.cs.princeton.edu/techreports/1992/395.ps.gz.

112. Zhong Shao, Andrew W. Appel, *Smartest recompilation*, in [136] (1993), 439–450; see extended version [111]. Citations in this document: §2.5.
113. Gerry Shaw (editor), *NAnt 0.85: a .NET build tool*, 2006. URL: http://nant.sourceforge.net. Citations in this document: §2.3.3.
114. Bob Sidebotham, *Software construction with Cons*, The Perl Journal **3**.1 (1998). ISSN 1087–903X. URL: http://www.foo.be/docs/tpj/issues/vol3_1/tpj0301-0012.html. Citations in this document: §2.3.4.
115. Paul Singleton, Pearl Brereton, *Building software by deduction: why and how*, Technical Report TR92–17, Deptartment of Computer Science, Keele University, 1992. URL: http://grosskurth.ca/bib/entries.html#1992/singleton. Citations in this document: §2.1.
116. Dag I. K. Sjøberg, Ray Welland, Malcolm P. Atkinson, Paul Philbrow, Cathy Waite, *Exploiting persistence in build management*, Software—Practice & Experience **27**.4 (1997), 447–480. ISSN 0038–0644. URL: http://dx.doi.org/10.1002/(SICI)1097-024X(199704)27:4<447::AID-SPE93>3.3.CO;2-J.
117. Ian Sommerville (editor), *Software configuration management: selected papers from the ICSE '96 SCM-6 workshop held in Berlin, Germany, March 25–26, 1996*, Lecture Notes in Computer Science, 1167, Springer-Verlag, Berlin, 1996. ISBN 3–540–61964–X. See [129].
118. Zoltan Somogyi, *Cake: a fifth generation version of make*, Australian UNIX and Open Systems User Group Newsletter **7**.6 (1987), 22–31. ISSN 1035–7521. URL: http://www.cs.mu.oz.au/~zs/papers/cake.ps.gz. Citations in this document: §2.3.1.
119. Yasuhiro Sugiyama, *Object Make: a tool for constructing software systems from existing software components*, in [107] (1995), 128–136; also printed in ACM SIGSOFT Software Engineering Notes **20**.SI. Citations in this document: §2.3.1.
120. Yasuhiro Sugiyama, *Distributed development of complex software systems with Object Make*, in [13] (2000), 82–93. URL: http://ssl.ce.nihon-u.ac.jp/papers/ICECCS2000.pdf. Citations in this document: §2.3.1.
121. Richard Stallman, Roland McGrath, *GNU Make: a program for directing compilation*, Edition 0.28 beta, last updated 23 September 1991 for make version 3.61 beta, Free Software Foundation, Boston, 1991; see also newer version [122].
122. Richard Stallman, Roland McGrath, *The GNU Make manual*, Edition 0.70, last updated 1 April 2006 for GNU make version 3.81, Free Software Foundation, Boston, 2006; see also older version [121]. URL: http://www.gnu.org/software/make/manual/make.pdf. Citations in this document: §2.3.1, §2.3.1, §6.1.1.
123. James Strachan (editor), *Groovy 1.0: an agile dynamic language for the Java platform*, 2006. URL: http://groovy.codehaus.org. Citations in this docu-

ment: §2.3.3.
124. Steve Talbott, *Managing projects with make*, 1st edition, O'Reilly, Sebastopol, California, 1985; see also newer version [103]. ISBN 0–937175–04–8.
125. Erik Thiel, *Compilercache 1.0.10*, 2003. URL: `http://www.erikyyy.de/compilercache`. Citations in this document: §2.3.5.
126. Walter F. Tichy, *Smart recompilation*, ACM Transactions on Programming Languages and Systems **8**.3 (1986), 273–291. Citations in this document: §2.5.
127. Walter F. Tichy, *Tools for software configuration management*, in [142] (1988), 1–20. URL: `http://www.ida.liu.se/~petfr/princprog/cm.pdf`.
128. David Tilbrook, Russell Crook, *Large scale porting through parameterization*, in [9] (1992), 209–216. URL: `http://www.qef.com/html/docs/strfix.pdf`.
129. David M. Tilbrook, *An architecture for a construction system*, in [117] (1996), 76–87. URL: `http://www.qef.com/html/qefwhite.html`. Citations in this document: §2.3.1.
130. David M. Tilbrook, P. R. H. Place, *Tools for the maintenance and installation of a large software distribution*, in [7] (1986), 223–237. URL: `http://grosskurth.ca/bib/entries.html#1986/tilbrook`. Citations in this document: §2.3.1.
131. Jesse Tilly, Eric M. Burke, *Ant: the definitive guide*, 1st edition, O'Reilly, Sebastopol, California, 2002; see also newer version [79]. ISBN 0–596–00184–3.
132. Andrew Tridgell, *ccache 2.4*, 2004. URL: `http://ccache.samba.org`. Citations in this document: §2.3.5.
133. Qiang Tu, Michael W. Godfrey, *The build-time software architecture view*, in [17] (2001); see also newer version [134]. URL: `http://plg.uwaterloo.ca/~migod/papers/icsm01.pdf`. Citations in this document: §2.1.
134. Qiang Tu, Michael W. Godfrey, Xinyi Dong, *Modelling and extracting the build-time architectural view* (2003); see also older version [133]. URL: `http://plg.uwaterloo.ca/~migod/papers/btv-ase03.pdf`.
135. David Turner (editor), *FT-Jam 2.5.2*, 2006. URL: `http://www.freetype.org/jam/`. Citations in this document: §2.3.2.
136. Mary Van Deusen (chairman), Bernard Lang (chairman), *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, Association for Comuting Machinery, New York, 1993. ISBN 0–89791–560–7. See [112].
137. Kim Walden, *Automatic generation of make dependencies*, Software—Practice & Experience **14**.6 (1984), 575–585. ISSN 0038–0644. URL: `http://grosskurth.ca/bib/entries.html#1984/walden`. Citations in this document: §2.3.1.
138. Jim Weirich (editor), *Rake 0.7.1: Ruby make*, 2006. URL: `http://rake.rubyforge.org`. Citations in this document: §2.3.4.
139. John R. White (chairman), Lawrence A. Rowe (chairman), *Proceedings of the 1983 ACM SIGPLAN symposium on programming language issues in software*

systems, San Francisco, California, June 27–29, 1983, Association for Computing Machinery, New York, 1983. ISBN 0–89791–108–3. See [88].

140. Jeanette Wing, Maurice Herlihy, Mario R. Barbacci, *Proceedings from the workshop on large-grained parallelism (2nd) held in Hidden Valley, Pennsylvania on October 11–14, 1987*, Technical Report CMU/SEI–87–SR–5, Software Engineering Institute, Carnegie Mellon University, 1987; accession number: ADA191094. See [106].
141. Laura Wingerd, Christipher Seiwald, *Constructing a large product with Jam*, in [43] (1997), 36–48. URL: `http://www.perforce.com/jam/doc/scm7.html`. Citations in this document: §2.3.2.
142. Jürgen F. H. Winkler (editor), *Proceedings of the international workshop on software version and configuration control, January 27–29, 1988, Grassau*, Teubner, Stuttgart, 1988. ISBN 3–519–02671–6. See [40], [56], [95], [109], [127].
143. Yijun Yu, Homayoun Dayani-Fard, John Mylopoulos, *Removing false code dependencies to speedup software build processes*, in [16] (2003), 343–352. URL: `http://www.cs.toronto.edu/~yijun/literature/paper/yu03cascon.pdf`. Citations in this document: §2.5.