Abstraction Problems in Software Manufacture

David Alex Lamb

February, 1989
External Technical Report
ISSN-0836-0227-
1989-243

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Version 1.1
Document prepared Thursday, November 20, 1997

Abstract

Software manufacture is the process of building or re-building a large software system from myriad components. The UNIX *make* program is the most well-known software manufacture tool. *Make* and similar systems have several deficiencies when used for complex multi-language, multi-subsystem programs; this work addresses those problems related to abstraction mechanisms. Particular problem areas discussed include specifying indirect and derived dependencies, manufacture subprocesses, and handling systems composed of independently-developed subsystems.

Keywords and phrases: configuration management, software manufacture, abstraction, *make* program

Computing Reviews categories: D.2.6 (Software Engineering: Programming Environments). D.2.9 (Software Engineering: Management - Software Configuration Management).

General Terms: Design

Table of Contents

List of Figures

1. Introduction

Software configuration management consists of several related sub-disciplines. *Software manufacture* is the term Borison introduced to describe the activity of constructing a software system from a collection of source objects [Bori86]. The archetypical software manufacture tool is the UNIX *make* program [Feld77]. Although *make* and similar tools work well for systems with simple structure, they have some difficulties in manufacturing larger, more complex systems. This paper discusses some of the areas of difficulty, and sketches some approaches to solutions. The difficulties we will explore are each a form of failure to provide appropriate abstraction mechanisms in the input language to the manufacture tool.

The kinds of systems this work addresses have several properties.

- A system is composed of several subsystems, each of which may have its own substructure. Each subsystem might be reusable, and thus might be part of several distinct systems.
- Subsystems evolve; each subsystem might be under the control of a separate project, and might have its own independent development schedule. The manager of a project using subsystems can choose the version of a subsystem to use, but beyond that cannot control the contents of a subsystem.
- Each subsystem consists of *exported* (visible) components, plus *hidden* components. Components might be programming language modules, or tools, or data files. Other subsystems might know the visible components, but should be able to ignore the hidden components. However, system-wide processing tools, such as the linker, need to know all the components, or perhaps all components with certain properties, of all subsystems.
- Source components are not all written in the same language. This property comes not so much from mixing ordinary programming languages as from having tools that generate portions of the system. The input notation for each tool is a different language. Furthermore, there may be long chains of tools producing output that becomes the input to yet other tools.

In this work, I assume the basic software manufacture tool follows Borison's model. The key aspects of this model are as follows: The tool works from a manufacture graph, which is a directed acyclic graph (DAG). Such graphs have two kinds of nodes: *processing* nodes, where work takes place, and *object* nodes, which are the inputs to and outputs from processing steps. An important difference between such a graph and the dependency graphs of programs like *make* is that a manufacture graph must represent *all* dependencies, including not only the input files for a tool, but the tool itself and any "hidden" files it references.

To understand the last point, consider the example of Figure 1, which shows a *make* input file for a small system consisting of a main program and two submodules.[1] Each line that starts in the left margin says that the target file, whose name precedes the colon (:), depends on all those files whose names follow the colon; if any of them change, *make* must use the immediately following command to rebuild the target. The dependency of prog on main.o arises because main.o is a relocatable file we pass to the linker to make up the executable program. The dependency of main.o on main.c arises because main.c is the source file from which we get the relocatable file main.o; the dependencies on mod1.h and mod2.h arise because main.c incorporates these two files by file inclusion. However, there are several more hidden dependencies.

- mod3.o also depends on the command line arguments "${CCFLAG} -DDEBUG"; if we change these options, we must recompile mod3.o, but *make* does not detect this. The fourth generation *make* program [Fowl85] lets a file depend on the contents of macros; by putting each distinct set of command line options in a macro we can record these dependencies, but this is clumsy for a system designer to do.
- The linking step implicitly depends on the C library, which on the particular version of UNIX we run at Queen's happens to be the file /lib/libc.a. This may not seem important in a system where such files rarely change; however, in some environments (such as those using a compiler and library that themselves are under development) such hidden files can change rapidly. Furthermore, if we had this information we could use it to find all the files we would need to

```
CCFLAG=-c -g
prog: main.o mod1.o mod2.o mod3.o
    ld -o prog main.o mod1.o mod2.o mod3.o
main.o: main.c mod1.h mod2.h
    cc ${CCFLAG} main.c
mod1.o: mod1.c mod1.h mod3.h
    cc ${CCFLAG} mod1.c
mod2.o: mod2.c mod2.h mod3.h
    cc ${CCFLAG} -DLISTOPTION=short mod1.c
mod3.o: mod3.c mod3.h
    cc ${CCFLAG} -DDEBUG mod1.c
```

Figure 1: Simple *make* File

[1] This example ignores *make*'s mechanisms for specifying *default rules*, which could shorten the example somewhat, but do not solve the problems we address here.

archive to tape to be able to faithfully rebuild an old version of the system.

- All the C compilations depend on the C compiler. However, the compiler itself runs several other programs, such as the C preprocessor. Furthermore, what C compiler we get depends on the value of the search path variable (PATH) in the shell; in an environment with several possible versions of a tool, such considerations become important.

Even if we could in principle record all these dependencies, forcing a system designer to specify the full manufacture graph in such detail would cause enormous headaches. Adding independent subsystems causes even more problems, since now portions of the manufacture graph for a system would be under the control of the developers of different subsystems.

The goals of this work are to design a few simple mechanisms to minimize what a system builder must specify to cause the manufacture tool to construct the correct manufacture graph for the system. Such mechanisms must make it easy to change the graph in response to changes in the relationships among the components of the system, and must track changes to the subsystem structure, not just the contents of files.

2. Problem Areas

This section considers several specific problem areas with current software manufacture input specification languages:

- Reusable abstractions.
- Implied dependencies.
- Derived dependencies
- System-wide processing steps.

2.1. Manufacture Abstractions

Anyone who does much complex software manufacture eventually develops a collection of idioms she uses over again. Such idioms are patterns of multi-step manufacture subprocesses; they have several characteristics:

- They are reusable in the same system, or different systems.
- They produce several objects.
- The builder might want to selectively reconstruct individual objects, rather than all the objects in the subprocess.
- There should be a way to hide some of the intermediate objects in the subprocess; their existence should not really be visible outside the subprocess.

make has no abstraction mechanism that can express such idioms; the only way to capture them is to use tools such as the *sed* stream editor to generate portions of a *make* input file.

For example, Figure 2 shows the process I used to produce individual chapters of a Software Engineering textbook [Lamb88]. The & symbol stands for the name of a chapter. The & object is the main exported result; it involves first making the printable form of the chapter (&.hp), then postprocessing some auxiliary outputs to update global book-wide data files. The &.chk object involves building three different objects (&.spl, &.styl, &.dic), which contain reports from simple machine proofreading programs; you might

---

```
&: &.hp
	sed -f /staff/dalamb/book/bin.sed &.tm >&.tm1
	cp book.xaux book.oxaux
	dsplit -key LB &.xxaux -omit \( TC FC IX LX SX HX \) -key SO &.xso &.tm1
	part -part book.so &.xso > &.so1
	mv &.so1 book.so
	part -part book.xaux &.xxaux > &.xaux1
	mv &.xaux1 book.xaux
	- diff book.xaux book.oxaux >book.xdif
&.chk: &.use &.spl
&.use: &.dic &.styl
&.dic: &.de ${DICTIONFILE}
	diction ${DICTIONFLAGS} &.de >&.dic
&.spl: &.de1 ${DICTIONARY}
	spell ${SPELLFLAGS} &.de1 >&.spl
&.styl: &.de
	style ${STYLEFLAGS} &.de >&.styl
&.de0: &.n
	SOelim -noso &.n | sed -e "s/@.*@/EQUATION/g" >&.de0
&.de: &.de0 ${REM}
	cat ${REM} &.de0 | nroff -Tlpr - | sed -e "/^.TG/d" >&.de
&.de1: &.de0 ${SPELL} ${REM}
	cat ${SPELL} &.de0 | nroff -Tlpr - | sed -f ${BKDIR}/remspell.sed >&.de1
&.pre: &.n ${SOFILES}
	SOelim ${SOFLAG} -part book.so &.n | sed -e "//d" | \
		docpre -phase ${CITEPART} ${XREFFL} | number > &.pre
&.hp: &.pre ${MACS}
	- cp &.tm &.tmold
	eqn &.pre | troff -me -Ttrlj 2> &.tm | ditplus -dtrlj > &.hp
```

Figure 2: Book Chapter Abstraction

---

want to select each of them independently. Some objects serve as intermediate processing points; for example, &.de is the result of filtering out text formatter markup that would confuse the proofreading programs; it is input to two of the proofreaders.

We can look to standard ideas from programming language design to solve these problems. Given any language semantic unit X, such as a manufacture step, we can introduce the idea of an X-valued abstraction [Tenn81]. Thus the input language for a manufacture tool should allow parameterized abstractions. The parameters would be objects (such as files or strings). Thus, for example, I might invoke a Chapter abstraction as:

```
intro: chapter(intro.n)
```

An abstraction would act somewhat like a generic module in a language like Ada. After you invoke such an abstraction, you could reference individual exported components (such as intro.use, or intro.hp). You could independently build any exported component. The manufacture tool might support a library mechanisms for such abstractions, so you could reuse them.

One sign that an idea is a good one is for it to solve elegantly problems other than the ones you planned for. The serendipity with abstractions is that the same mechanism works for subsystem descriptions; a subsystem is nothing more than an abstraction that you happen to invoke only once per system.

2.2. Implied dependencies

make has difficulty handling transitive dependencies. For example, consider the C source file x.c, which incorporates several .h files by file inclusion:

```
x.o: x.c x.h lib1.h lib2.h lib3.h
     cc $(CCFLAG) x.c
```

Now suppose we are trying to simulate abstract data types in C. Module lib1 might define some type with some fields intended to be visible to its clients, and others intended to be hidden. Suppose we add some ‘‘hidden’’ fields, and so modify lib1.h to include adt1.h. Unfortunately, we cannot really hide such fields in C, even though we know we will not reference them in clients of lib1; we must recompile x.c whenever adt1.h changes, since the sizes of some records defined in lib1.h might change. Existing approaches have problems.

- We could make x.o depend on adt1.h. However, we must do this for all the clients of lib1; this is a repetitious, error-prone editing job.
- We could add the step:

  ```
  lib1.h: adt1.h
       touch lib1.h
  ```

  to the manufacture graph. When adt1.h changes, this forces lib1.h to change, too. However, this fails if lib1.h is part of a reusable subsystem, since clients of the subsystem would typically not have permission to modify the date-of-last-change of such files. Furthermore, because of cross-subsystem dependencies, adt1.h and lib1.h might be in separate subsystems; two different clients of the same version of lib1.h might sensibly select different versions of adt1.h. Finally, even without these problems, your environment may have other uses for accurate modification dates that this mechanism would defeat.

- We could cause our manufacture tool to scan source files automatically for inclusion directives, and deduce dependencies from them. In some environments this is a good solution. However, this requires a (potentially) language-specific scanning tool for each language in our toolset. Furthermore, in some commercial software environments, this approach places too much control of the system structure in the wrong place; managers may want a ‘‘freezable’’ description of the system interdependencies, rather than distributing this information throughout the source code. It would be better to find an approach that would allow us to specify such dependencies by hand when we had to, but allow tools to help when we have time to build them.

I believe the best approach is to separate out different kinds of dependencies. Thus x.o has a clear-cut direct dependency on x.c; however, depending on x.c implies a dependency on lib1.h and so on; depending on lib1.h implies a dependency on adt1.h. We can construct a database of such *implied dependencies*. If x depends directly on y, and y implies a dependency on z, then changing z means rebuilding x, without rebuilding y. We can build such a database by hand if we have no appropriate tools. When we can afford to schedule the effort to build one, a database editor can invoke language-specific scanners to find or check such dependencies when asked to do so.

2.3. Derived dependencies

Suppose source file mod.c depends directly on lib1.h. Normally, lib1.h is the specification part of a module whose implementation is in lib1.o. This means that mod.o implies a dependency on lib1.o; that is, any system that includes mod.o must also include lib1.o.

Current systems manage this problem in one of two ways, both of which have some difficulties.

1. The system builder might explicitly list all relocatable object files for the linker. This is a tedious and error-prone process if done manually.
2. The system builder might instruct the manufacture tool to place all such relocatable object files in a linker library, and rely on the library external symbol resolution rules to link in all appropriate components.

The second solution can work if each separate project using several subsystems can afford the space for a library file large enough to hold copies of all the object files of all the subsystems, and if the linker can correctly resolve backward references in such a library. However, many projects cannot afford the space for so many copies; instead, they expect to have separate libraries for each subsystem, shared among all the users of a subsystem. Furthermore, many linkers are not clever enough for such a search strategy; they may expect new object modules to refer only to previously-linked symbols, or to symbols that will appear in later object modules. Even if they do allow arbitrary references within a library file, they may not allow it across a list of library files; thus, projects that cannot afford so many copies of object modules are out of luck if there are any loops in the relationships among subsystems.

A related problem for systems that must explicitly list what object modules to link is subset management. Suppose a subsystem exports k modules, each of which depends on different subsets of its hidden modules. Different clients might import different subsets of the exported modules. The problem is that there are potentially $2^k - 1$ distinct collections to manage.

A possible direction for a solution is to develop a mechanism for the process of constructing mod.o from mod.c to generate an implied dependency of mod.o on lib1.o; we call this a *derived dependency*. The manufacture step for constructing the executable program would specify only a direct dependency on main.o, the object file for the main program. The manufacture tool could deduce all the rest of the object files from implied and derived dependencies. Indeed, the simplicity of Ada-specific or Modula-specific manufacture tools is that they can deduce these two kinds of dependencies automatically.

2.4. System-wide processing

The need to find all the appropriate object files to link is a special case of the general problem of processing all files of certain characteristics from all subsystems as a group. Other examples from the Queen's Program Component Generator (PCG) project [Lamb89] include:

- Pass all run-time symbol tables for IDL-based data structures to a tool that gives each a unique structure number.
- Combine error descriptions from all subsystems to the *ersort* tool to form a single database of all error messages.

A solution to this problem seems to require two mechanisms:

- Abstractions need the ability to add dependencies to a parameter or global object. For example, the IDL translator abstraction could add each structure symbol table to an ‘‘all IDL symbol tables’’ object.
- There should be some means for turning a set of dependencies into input for a tool.

3. Other Considerations

There are several other problem areas for a software manufacture specification language that I have not addressed in the previous section, and whose solutions require further design.

Subsystems can depend on each other in complex ways that seem inherently to require some interaction with the version selection process. For example, for a graduate course I provided an input subsystem to teach the students how to use a parser generator. The generated parser called an error subsystem, which expected both to print errors on the terminal and to incorporate them into a listing. Thus, to learn about the parser the students had to learn about several other subsystems. Ideally, there should have been a way for the manufacture tool to notice the subsystems the main system was using, and to select different versions of the error reporting subsystem depending appropriately. If the students showed they knew of the listing subsystem by calling it from their main programs, the manufacture tool should select the version that reports errors in the listing; otherwise, it should select the version that only prints on the terminal.

There needs to be some mechanism to handle *component migration*, where a module or group of modules split off from their former subsystem as the set of tools evolve. To use a component, a second component should not need to mention the subsystem of the first component; there should be a separate mechanism to associate such use names with appropriate subsystems. Perhaps the *provides/requires* facility of typical module interconnection languages is an appropriate mechanism.

There needs to be a way to integrate version selection with manufacture; the *shape* system at the Technical University of Berlin does so for *make*, and its mechanisms may be appropriate for other manufacture tools [Mahl88]. Selecting versions of subsystems may be more complex, since different versions of a subsystem may have different manufacture sub-graphs.

4. Conclusion

We have shown several problem areas where current *make*-like tools do not meet the needs of complex systems, and have sketched possible solutions. During 1989 we plan to flesh out the notational sketches of this paper, and design a prototype translator and tool. We expect that the PCG tools environment will be a reasonable one in which to test the ideas of the prototype; the tools are small and well-defined, but have interesting relationships.

I would like to thank Ellen Borison for several enlightening discussions of software manufacture, and for the students in Queen's course CISC 835 (The Tools Approach) in Winter 1988 for pointing out deficiencies in my earlier approaches to these problems. The members of IFIP Working Group 2.4 gave helpful feedback on the talk on which I based this paper. Margaret Lamb corrected several small deficiencies in an earlier draft. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant OPG0000908, and in part by the Information Technology Research Centre (ITRC), which is part of the Ontario Centres of Excellence Program.

References

Bori86. Ellen Borison, "A Model of Software Manufacture," in Reidar Conradi, editor, *Proceedings of the International Workshop on Advanced Programming Environments*, pages 197-200, Springer-Verlag, Berlin (June 1986). Lecture Notes in Computer Science 244.

Feld77. S. I. Feldman, "Make - A Program for Maintaining Computer Programs," Technical Report 57, Bell Laboratories (April 1977).

Fowl85. Glenn S. Fowler, "The Fourth Generation Make," in *Proceedings of the 1985 Summer Conference*, USENIX Association (11-14 June 1985). Held in Portland, OR.

Lamb88. David Alex Lamb, *Software Engineering: Planning for Change*. Prentice-Hall, Englewood Cliffs, NJ (1988).

Lamb89. David Alex Lamb, "Program Component Generator Project Implementor's Guide - Version 2.3," Internal Report ISSN-0836-0235-89-IR-02, Queen's University Department of Computing and Information Science (October 1989).

Mahl88. Axel Mahler and Andreas Lampen, "*shape* — A Software Configuration Management Tool," in *Proceedings of the International Workshop on Software Version and Configuration Control*, German Chapter of the ACM (January 1988). Held in Grassau, West Germany.

Tenn81. R. D. Tennent, *Principles of Programming Languages*. Prentice-Hall International, London (1981).