Relations in Software Manufacture

David Alex Lamb

March, 1991
External Technical Report
ISSN-0836-0227-
90-292a

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

Version 1.4
Document prepared Friday, October 7, 1994

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. One deficiency of *make* and similar systems is that they manage only a single dependency relation. This paper shows how to solve several software manufacture problems by generalizing such tools to support additional relations and relational expressions.

Keywords and phrases: relational algebra, configuration management, software manufacture, programming environments, 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 manufacture is the term Borison introduced to describe the activity of constructing a software system from a collection of source objects [1]. The archetypical software manufacture tool is the UNIX *make* program [3]. Make-like tools work well for systems with simple structure, but for complex ones do not provide appropriate abstraction mechanisms in their input languages [6].

Our goals are to design a few simple mechanisms to minimize what someone (whom we call the *system builder*) must write to specify the graph of relationships among objects in the system (the *manufacture graph*). We propose extending manufacture tools to handle several user-defined relations and relational expressions, and show how this solves several software manufacture problems. We use examples from C and UNIX, but the ideas would be equally useful with other languages and systems. We presume readers are familiar with *make*.

1.1. Context and Background

The kinds of systems this work addresses have several properties. A system consists of several subsystems, many of them reused in several other systems. Subsystems evolve, each controlled by a separate project with its own development schedule. The system builder can choose a version of a subsystem, but cannot otherwise control its contents. Each subsystem consists of *exported* (visible) components plus *hidden* components; components include programming language modules, tools, and data files. 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; tools may generate portions of the system, and there may be long chains of tools producing input for other tools.

1.2. Relational Expressions

For concreteness we suggest a particular way of encoding relations as an incremental change in the current *make* input language; a full redesign of such tools might suggest better alternatives. An *explicit* relation names the particular files involved in the relation. Thus

```
generated_file :(relation_name) file1 file2 ...
```

says that the given generated file (named before the colon) bears the given relation (named in parentheses) to all the other files (after the relation name). One defines an *implicit* relation via a relational expression.

```
name := expression
```

Figure 1 defines the operations permitted in such expressions. If several such definitions occur, the relation is their union; this permits a form of distributed specification of a relation. One specifies when to rebuild a generated file by naming the relation that governs generation.

Operation	Infix	Postfix	Other
( )			parentheses (grouping)
;	composition		
!		inverse	
+	union	transitive closure	
–	set difference		
*	intersection	reflexive transitive closure	

Figure 1: Relational Operations

```
output_file (expression):
    command ....
```

means that, whenever any file related to `output_file` by the given relational expression changes, one executes the given command to rebuild the output file. We permit only one such declaration per output file; in future work we will investigate methods of disambiguating among several of them.

Certain dependencies are common enough to require some form of rule. Thus we allow

```
&.c :(c_derives) &.o
```

to mean that any file with type `.c` has relation `c_derives` to a file of the name name with type `.o`. This is similar to the *metarules* of the *mk* program [5]; however, we allow naming several separate dependencies, while *mk* allows more complex name pattern matching.

1.3. Sets

Any set of files corresponds to an identity relation. The expression `(rel!);rel` defines an identity relation for the range of `rel`; similarly, `rel;(rel!)` corresponds to the domain. Certain useful sets correspond to such domains and ranges; we introduce `dom(relation)` and `rng(relation)` to represent them. The set of all object modules generated by the C compiler is thus `rng(c_derives)`. Composing a relation with a set means restricting the domain or range of the relation; thus `rel;set` is the subset of `rel` with range restricted to `set`. We can say `{file1,file2,...,fileN}` to explicitly construct a set of N named files.

Tools may need to process lists of files obtained from relations; we thus give a way to generate a string from a set or relation. The expression `@[relation]` constructs a string consisting of the names of all the files in the domain and range of the relation, separated by spaces. `@[option relation]` gives a sequence of files in a topological sort of *relation*; if file A is related to file B, then the < (>) option requires that A appear before (after) B in the result string. `@[option relation[file]]` gives a topological sort of the named relation starting with the given file instead of with the natural roots of the relation. `@[relation[file]]` is the set of files related to the given file by the given relation; it is a shorthand for `@[rng({file};relation)]`. As topological sorting is related to transitive closure, `@[option relation+]` gives the same results as `@[option relation]`.

2. Problems and Solutions

This section uses the facilities of Section 1.2 to solve several software manufacturing problems.

2.1. Transitive Dependencies

Current manufacture tools create some difficulty in describing transitive dependencies. For example, consider the C source file `x.c`, which incorporates several `.h` files by file inclusion (#include directive):

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

Suppose we are trying to simulate abstract data types in C. Module `lib1` might define a type with some fields intended to be visible to its clients, and others intended to be hidden. Suppose we add a “hidden” field 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. With the relational extension, we can define an `includes` relation to capture one-level file inclusion, and

```
lib.h :(includes) adt1.h adt2.h
x.c :(includes) x.h lib1.h lib2.h lib3.h

&.c :(c_derives) &.o
o_from_c := c_derives!
c_depends := o_from_c;includes*
&.o (c_depends):
    cc ${CCFLAGS) &.c
```

Figure 2: Transitive Includes in C

use transitive closure to discover the full set of dependencies, as Figure 2 shows.

2.2. Derived Dependencies

Suppose source file `mod.c` depends directly on `lib1.h`. Often, `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`; we say that `mod.h implies_link mod.o`. The simplicity of Ada-specific or Modula-specific manufacture tools comes from their deducing these kinds of dependencies automatically.

Current systems manage this problem in one of two ways, both of which have some difficulties. The system builder might explicitly list all relocatable object files for the linker. This is tedious and error-prone if done manually. Alternatively, the system builder might place all such relocatable object files in linker libraries (one per subsystem), and rely on the library external symbol resolution rules to link in all appropriate components. This works well if there is an appropriate order in which to pass these libraries to the linker. However, many linkers cannot handle backward references from one library to another (although they may handle such references within a library); this may force a project to make its own super-library from all the individual libraries, which may cost too much space for the copies.

A system builder can solve this problem by using the *includes* and *c_derives* relations of the previous section, and by defining new relations (see Figure 3). `linked_from` means that `main.o` contains the main program (entry point) of the `exec` program. Consider Figure 4 (which omits how to build `x.o` and `z.o`, for brevity). To build `exec`, the linker must put together `main.o`, `x.o`, `y.o`, and `z.o`; this is exactly the set of files related to `exec` by the relation `object_files`.

2.3. Multiple Inclusion

Suppose `m.c` includes `y.h` and `z.h`, and `n.c` includes `z.h`. Now suppose a maintainer changes `z.h` to define field of some type declared in `y.h`, but some clients of `z.h` will

```
&.h :(implies_link) &.o
exec :(linked_from) main.o
object_files := linked_from;(o_from_c;includes+;implies_link)*

exec (object_files):
    ld -o exec @[object_files[exec]]
```

Figure 3: Using the implies_link Relation

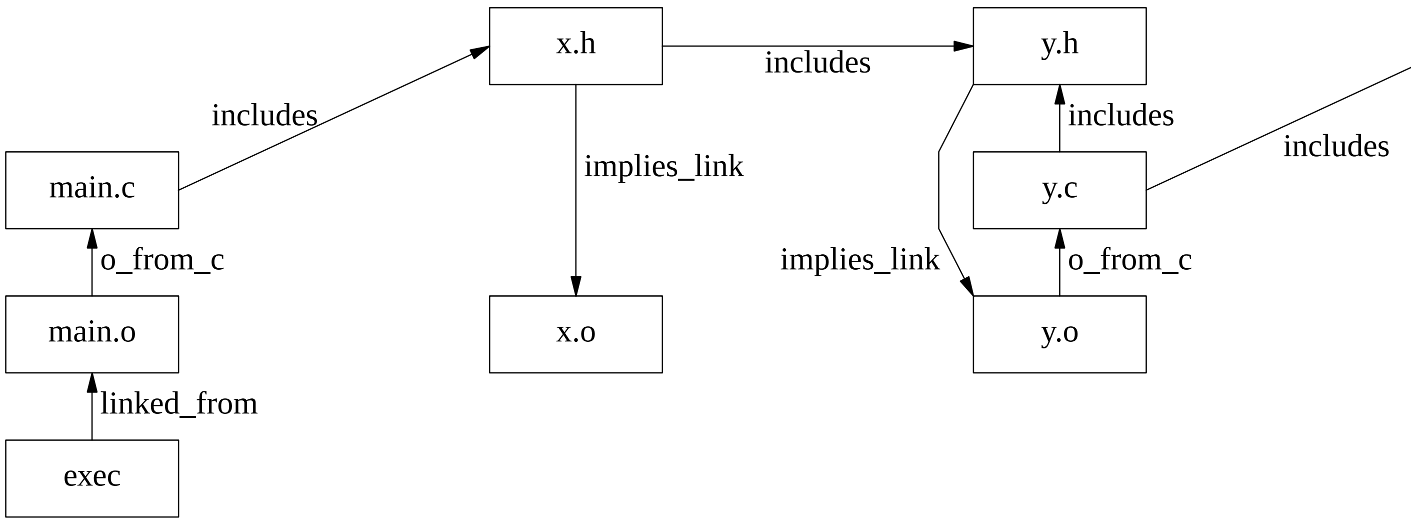

Figure 4: Implies_Link Relationships

never manipulate such fields. How should the system builder record this new dependency? If `z.h` includes `y.h`, there is a double inclusion of `y.h` in compilations of `m.c`; this is inefficient, and may lead to spurious errors about redefinitions. Conditional inclusion is possible with the C preprocessor, but lacks elegance.

What is really happening here is that the preprocessor's include directive has at least two distinct uses: "compose a source file by combining several partial files," and "ensure that you've read the following definitions before you compile this program." We can solve the multiple inclusion problem by separating these two uses. Whenever header file `f1.h` defines fields of types defined in `f2.h`, we declare

```
f1.h :(requires) f2.h
```

We *do not* include `f2.h` within `f1.h`. If `x.c` directly uses definitions from `f1.h`, we declare

```
x.c :(requires) f1.h
```

We similarly avoid including `f1.h` within `x.c`. If for some reason we are composing `x.c` from code fragments in several files, we declare

```
x.c :(includes) fragment1 fragment2 ...
```

and this time we *do* include `fragment1` at the appropriate place in `x.c`. We then topologically sort `requires`, create a list of files to which `x.c` is related by the result, and concatenate these files (in reverse topological order) together with `x.c` as input to the C compiler (see Figure 5; we assume a C compiler that acts as a filter). This approach ensures that the C compiler sees each header file only once, and in a correct order. To allow included files to have their own ‘‘requires’’ relations, we can say

```
c_depends := o_from_c;(includes*;requires)*
```

2.4. Testing

Suppose we have relations `has_stub`, relating each C source file to a stub module that has the same interface but trivial implementation, and `has_driver`, relating each C file to a corresponding test driver. Figure 6 shows how we can rebuild an executable test program when the C module, its driver, or any stubs it uses change. `ident` is a (predefined) identity relation that relates each file to itself. Given a relation between a C source file and corresponding test cases (inputs to the driver program), we could also rerun all the test cases whenever any of them change. Running just the changed test cases would require a mechanism to select the set of changed files.

```
  m.c :(requires) y.h z.h
  n.c :(requires) z.h
  z.h :(requires) y.h

  c_depends := o_from_c;(requires*)
  &.o (c_depends):
      cat @[>c_depends[&.o]] | cc >&.o
```

Figure 5: Use of Requires Relation

```
uses := requires;implies_link;o_from_c
&.c :(executable_test) &.test
test_link := executable_test!;(ident+(uses;has_stub)+has_driver);c_derive

&.test (test_link):
    ld -o &.test @[test_link[&.test]]
```

Figure 6: Controlling Testing

3. Discussion

We have shown four examples of using relational expressions to solve software manufacture problems; we have found others, and undoubtedly system builders can think of their own. Existing tools manage one relation and its transitive closure; extending them to handle several relations should be reasonably easy.

3.1. Prior Work

There are many make-like programs; most manage a single dependency relation. In contrast, Schwanke et al. [8] describe the BiiN™ Software Management System (SMS), which provides four built-in relations, and allows developers to specify ‘‘dependency rules’’ to deduce the built-in relations from user-defined relations. It then uses the built-in relations to deduce the order in which to invoke processing steps, and what inputs to give to the tools invoked during processing steps. Their `needs` relation is the one that governs when to invoke a processing step: if A needs B, then one rebuilds A when B changes; we would view their relation as the union of several separately-specified relations. Their `derives` relation is simply `(needs*)!` in our notation. Their other two primitives are ternary relations between pairs of objects (files) and tools; our paper does not directly address modeling of tools and automatically constructing inputs to tools, although we expect to address these issues in future work.

The main interesting difference is in how one derives new relations from old. SMS uses a deductive mechanism that repeatedly applies rules until they are translated to primitive ones; we use direct specification of rules with transitive closure. Our approach is unimplemented and theirs is not widely available, so we cannot objectively compare the two; we believe ours would be more efficient, and that system builders would prefer direct specification (where they can picture what is going on more easily).

Other *make* variants have recognized the need for better abstraction facilities. *imake* (available with the X11 distribution) and *cake*[9] both use the C preprocessor, which provides file inclusion, conditional text inclusion, and simple macros.

3.2. Future Work

We have argued that system builders would find our relational mechanisms better than alternatives; to test these claims, we plan to build a prototype tool. Before doing so, however, we expect to continue working on other design issues, including:

- Subsystem interconnection and namespace management issues.
- Managing caches of generated files, perhaps introducing one cache per subsystem. The Odin system exemplifies many of the issues to consider, but appears to maintain a single cache per system [2].
- Inclusion of version management facilities. The *shape* system exemplifies many of the issues [7].
- Efficiency considerations, such as treating the dependency graph as a cache, and reducing the number of *fork* system calls [4], and executing system-building commands in parallel [5].

- Partial system building. When porting a system that uses many UNIX tools to a non-UNIX system, one might run some construction steps on a UNIX host, then transport generated files to the target system.

3.3. Acknowledgements

Discussions with Ellen Borison, Margaret Lamb, Grant Guenther, and Eric Unteregelsbacher helped clarify my ideas. The members of IFIP Working Group 2.4 gave helpful feedback on the talks on which I based this paper; Reidar Conradi observed that particular relational expressions seem to characterize particular implementation languages, and others suggested using different relations to govern different ways of constructing the same output. Margaret Lamb corrected several small deficiencies in an earlier draft; John Gannon, Jim Purtilo, Trevor Thompson, Robert Schwanke, Pablo Straub, David Wonnacott, and Marvin Zelkowitz gave helpful criticisms.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant OPG0000908, the Information Technology Research Centre (ITRC), and the Institute for Advanced Computer Studies at the University of Maryland (UMIACS).

References

1. 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.
2. Geoffrey M. Clemm, *The Odin System: An Object Manager for Extensible Software Environments*, Ph.D. dissertation, CU-CS-314-86, Computer Science Department, University of Colorado, Boulder (February 1986).
3. S. I. Feldman, ‘‘Make - A Program for Maintaining Computer Programs,’’ Technical Report 57, Bell Laboratories (April 1977).
4. Glenn S. Fowler, ‘‘The Fourth Generation Make,’’ in *Proceedings of the 1985 Summer Conference*, USENIX Association (11-14 June 1985). Held in Portland, OR.
5. Andrew Hume, ‘‘*Mk*: a successor to *make*,’’ in *Proceedings Summer 1987 USENIX Conference*, pages 445-457, USENIX Association (1987).
6. David Alex Lamb, ‘‘Abstraction Problems in Software Manufacture,’’ Technical Report ISSN-0836-0227-89-243, Queen’s University Department of Computing and Information Science (February 1989).
7. 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.

8. R.W. Schwanke, E.S. Cohen, R. Gluecker, W.M. Hasling, D.A. Soni, and M.E. Wagner, ‘‘Configuration Management in BiiN SMS,’’ in *11th International Conference on Software Engineering*, pages 383-393, Pittsburgh PA (May 1989).

9. Z. Somogyi, ‘‘Cake: a fifth generation version of make,’’ *Australian UNIX system User Group Newsletter* **7**(6):22-31 (April 1987).