CMOD : Enforcing Modularity in C Code

Saurabh Srivastava
University of Maryland, College Park
saurabhs@cs.umd.edu

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Abstract

Many modern languages have sophisticated linguistic support for modular programming. In these languages, the different components of a system can be developed independently, and when combined together the resulting program will be type correct. The C programming language does not contain direct support for modules, but over time programmers have developed a discipline of modular programming which treats `.c` *files as modules and* `.h` *header files included by C's preprocessor* `cpp` *as interfaces. However, without proper enforcement, a mistake can easily lead to a confusing error and time wasted tracking it down. This paper presents* CMOD*, a tool that enforces rules to ensure that common idioms for modularity are used correctly, and thus the program is compiled correctly.* CMOD *easily fits within the development process as a front-end for the compiler and linker (e.g.,* `gcc` *and* `ld`*). We have tested* CMOD *on a number of open source software systems and found that each exhibits a small number of violations of our rules, all of which can be easily fixed.*

1 Introduction

Informally, a module is a collection of programming language definitions—for functions, types, variables, etc. A program is a collection of modules which may refer to (and thus depend on) one another; e.g., a function defined in one module may call a function defined in another module. Originally, module systems were necessary to allow a program to be compiled piecemeal, because as libraries grew in size it became too expensive to compile an entire program at once [3]. Modules also have the benefit of making programs easier to compose and understand [17].

The module systems of modern languages, such as ML [4, 15], Haskell [13], Ada [1], and Modula-3 [9], ensure that when separately-compiled files are linked together the resulting program is type correct. This is achieved through the use of *interfaces* (or *signatures*), which describe the externally-visible definitions of a module, and can safely stand in, during compilation, for the modules which they represent. The C programming language has no linguistic support for modular programming. Nonetheless, programmers often informally treat source files as modules, and `.h` header files as interface files, imported into a client source file via the preprocessor directive `#include`.

Unfortunately, the analogy is only skin deep. Defined and undefined symbols in compiled object files are *untyped*, so that the linker resolves the definitions by name only. This means that the programmer may fail, through mistake or carelessness, to use an external symbol in a manner conforming to the type of the symbol in its defining module. As a result, his code compiles and links correctly even if (1) he makes no reference to the "interface," if one exists, that the symbol supplier provides, instead including `externs` within his own code as necessary; or (2) he compiles against an outdated or incorrect version of the interface; or (3) his use of `#define` directives in the code or from the command line cause definitions in included headers to be treated inconsistently. In our experience, linking errors of this sort arise regularly. Though they can often be caught through testing, this seems unsatisfying since they could easily be caught automatically. Just as ANSI C improved on K&R C by mandating that prototypes match their definitions within a compilation unit, it seems only natural to take the next step to ensure this occurs across compilation units as well.

This paper presents a tool called CMOD that aims to enforce the modular convention already used by C programmers. The goal is to ensure that linking will always yield type correct programs, while trying to maintain as much backward compatibility with existing code as possible. We achieve this by enforcing three rules during compilation and linking:

1. Whenever one file links to a symbol defined by another file, both files must include a header that contains the type of that symbol.

2. With the exception of a designated `config.h`, header file inclusion must be order-independent. That is, `#defines` or `#undefs` appearing in one header should not affect the types of declarations appearing in other headers.

3. Header files must not contain (non-inlined or non-`static`) code and data definitions.

CMOD is built as a front-end to the C compiler and linker, so it easily fits into standard build processes. Its implementation largely draws on existing tools so as to easily adapt to other C dialects that make use of the preprocessor, including C++ [24] and Cyclone [10]. We have run CMOD on several open source C programs. In our experience, programs generally adhere to the above rules, and when they do not they can be made to by changing just a few lines of code. The result is more modular and maintainable code.

The remainder of this paper is organized as follows. First, we describe the current practice and problems of modular programming in C (Section 2), and how CMOD solves these problems, describing both its rules (Section 3) and implementation (Section 4). Section 5 presents our experience using CMOD on several open source software packages. We conclude with a discussion of related and future work (Sections 6 and 7).

2 Modular Programming in C

Many programming languages provide support for modular programming using interfaces for defining module boundaries. In particular, if a function in module A depends on the function `foo` provided by module B, to compile A does not require an implementation of B, but merely an interface for B (call it B_I) that mentions the type of `foo`. When using separate compilation, the linker performs *link checking* to ensure that the combined files were compiled against compatible interfaces. For example, when the linker combines modules A and B, it must be sure that B indeed *implements* the interface B_I against which A was compiled, which is to say that the definitions in B have the same (or compatible) types as the declarations appearing in B_I.

Figure 1(a) illustrates how this informal example would be expressed in the module system of the language Objective Caml (OCaml), a dialect of ML [4]. The interface file for B is defined in the file `b.mli`, which includes the type of the function `foo`.[1] Meanwhile the module A, implemented by the file `a.ml`, defines a function `f` that calls B's `foo` function; it includes the signature for B using the syntax `open B`. The implementation `a.ml` can be compiled without knowing how `foo` is implemented; all that is needed is `foo`'s type to generate code for A. Finally, an implementation `b.ml` for B is shown that implements `foo` as the "add one" function. This implementation is acceptable because `foo`'s implementation type in B matches its type in B's interface; if it did not, either the compiler, or in some cases the linker, would reject the program.

The C programming language has no formal, linguistic module system as in OCaml, but rather an informal one that is used similarly. In particular, programmers typically treat individual `.c` files as modules, and `.h` "header files" as interfaces. An example of this is shown in Figure 1(b). Here, the implementation of module A is in the file `a.c`, and the interface of module B is stored in the file `b.h`. This interface is included via the `cpp` ("C pre-processor") directive `#include "b.h"` at the top of its implementation `b.c`. When compiling `a.c`, `cpp` inlines the contents of `b.h` and thus the compiler "sees" the prototype for `foo` so that it can compile `f` successfully. In C programming practice, there are many variations on this basic theme.

2.1 Problems with Current Practice

Because modular programming in C is a convention not enforced by the compiler or linker, program errors can arise from a number of sources:

No connection between a module and its interface The compiler and linker do not know that the "interface" of `b.c` is the header `b.h`. This could lead to two problems. First, the types of a module's definitions may not be compatible with the types appearing in its interface. In our example, the type of `foo`'s implementation in `b.c` could be incompatible with `b.h`'s declaration of `foo`'s type. To illustrate how, say that in `b.c` we decide to change `foo` to be

```
int foo(int *px) { return *px+1; }
```

but we forget to change the interface `b.h`. This would mean that `a.c` would be compiled to think `foo` had the old type, and when it was linked against `b.c` the call to `foo` would be type-incorrect, since it would be passing an integer when a pointer to an integer is expected.

Second, a client of some module may ignore the module's interface by not `#include`ing it, and instead write declarations for what it thinks are the types of that module's members. For example, we could change `a.c` to remove the `#include "b.h"` and replace it with a prototype of `foo`, as follows:

[1]The notation `int -> int` means that `foo` takes an integer argument and returns an integer result.

b.mli:
```
val foo : int -> int
```

b.ml:
```
let foo (x:int) = x+1
```

a.ml:
```
open B
let f (y:int) =
  1+foo(y)
```

(a) Objective Caml Modules

b.h:
```
int foo(int);
```

b.c:
```
int foo (int x) {
  return x+1;
}
```

a.c:
```
#include "b.h"
int f(int y) {
  return 1+foo(y);
}
```

(b) C "Modules"

Figure 1: Simple module example in Objective Caml and C.

```
int foo(int);
int f(int y) { return 1+foo(y); }
```

Unfortunately, if the maintainer of B changes `foo` as described above and modifies the interface file `b.h` accordingly, we run into the same error because `a.c` does not refer to the (now updated) header file.

No link checking The C linker resolves symbols between modules only by their names, and does not consider their types. This means that a file compiled against one version of an interface could be incorrectly linked against another incompatible version of the implementation. Suppose in our example that the user changes `b.c` and `b.h` as described above and recompiles them, and then immediately relinks `b.o` with the *old* `a.o`; i.e., without recompiling `a.c` using the new interface `b.h`. The code in `a.o` was compiled against the old interface, and therefore will call `foo` at the incorrect type. If `a.c` had been recompiled, the compiler would observe the mismatch and issue a warning. We would like the linker to signal that such recompilation is necessary.

Bad preprocessor interpretation of header contents Of course, `#include` is just one of many `cpp` directives that appear in `.c` and `.h` files. Programmers often define `cpp` macros and use conditional operations like `#ifdef` and `#ifndef` to support platform independence or different implementation strategies. If not used consistently, this can lead to problems.

For example, suppose the programmer rewrote our example to support both implementations of `foo` that we have considered, say to allow backward compatibility. This is shown in Figure 2. For this to work, clients like `a.c` must be compiled with the same `#define` macro directives with which `b.c` was compiled. That is, if we compiled both `a.c` from Figure 1(b) and `b.c` from Figure 2 with `-DOLD`, then the `#ifdef OLD` operation will be true in both cases, and we will end up with the same program as in Figure 1(b). Likewise, if we compile both without `-DOLD`, then we will get a type error, which is a good thing, because the type of `foo` in the new `b.h` will mismatch with the caller in `a.c`, and thus be prevented from linking incorrectly with `b.c`. The problem arises when we mistakenly compile `a.c` with `-DOLD` but *do not* compile `b.c` that way. Once again we will end up with inconsistent assumptions.

b.h:
```
#ifdef OLD
 int foo(int);
#else
 int foo(int *);
#endif
```

b.c:
```
#ifdef OLD
 int foo (int x) {
   return x+1;
 }
#else
 int foo (int *x) {
   return *x+1;
 }
#endif
```

Figure 2: Conditionally-compilable `b.c` and `b.h`.

It is useful to think of a `.h` file like the one shown in Figure 2 not so much as an interface *file*, but as an interface *program*, which takes as arguments an environment of macro definitions and returns as results an interface file and a modified list of macros. The fact that a header can define and undefine macros can actually make the contents of a header file depend upon the *order* in which it was `#include`d: one header may `#define` a macro that affects the interpretation of a later header, whereas if the order of the two were reversed the headers would be interpreted differently. While this can be useful in a controlled way (e.g., a `config.h` file that defines platform-specific information that always appears first in the list of included headers), general order dependence can make local reasoning about interfaces and the modules they represent difficult, and can make software configuration a real nightmare.

3 Rules for Enforcing Module Boundaries

Given these problems, one might decide that the practice of module writing in C is broken and we should design a new linguistic module system that avoids `cpp` altogether, e.g., by defining means to support components [16, 21].

prototypes.h:
```
#ifndef _PROTOTYPES_H
#define _PROTOTYPES_H

void iterate();
int quit();
void init_counter(int);

#endif // _PROTOTYPES_H
```

globals.h:
```
#ifndef _GLOBALS_H
#define _GLOBALS_H

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
extern int counter;

#endif // _GLOBALS_H
```

main.c:
```
#include "prototypes.h"
#include "globals.h"

void iterate() {
  while (!quit()) {
    printf("%d\n", counter--);
  }
}

int main(int argc, char **argv) {
  assert(argc==2);
  init_counter(atoi(argv[1]));
  iterate();
  return 0;
}
```

helpers.c:
```
#include "prototypes.h"
#include "globals.h"

int counter;

void init_counter(int init) {
  counter = init;
}

int quit() {
  return (counter!=0);
}
```

Figure 3: Example illustrating generalized interfaces supported by Rule 1.

While this makes sense in some contexts, it seems unnecessary in general: when certain reasonable conventions are properly followed, `.h` files *do* make reasonable interface files—what is lacking is a way to *enforce* those reasonable conventions. We have developed CMOD to fill this gap. Our approach has been to identify three rules that if enforced will ensure that a program linked together from separately-compiled modules will be type-correct. These rules are consistent with those of more formal module systems, while at the same time adhere to practical usage scenarios for C programs.

Rule 1: Whenever one file links to a symbol defined by another file, both files must include a header that contains the type of that symbol. We want to prevent the situation in which the type of a definition in B's implementation `b.c` does not match its corresponding type in `b.h`, and we want to ensure that a client `a.c` of B always includes B's interface `b.h`, rather than guessing the types of B's members. A simple way of achieving both goals together is to require that both A and B include a header file that defines members in B that are linked to by A, i.e., `b.h`. By doing so, we ensure that members in `b.c` have types that are compatible with the advertised interface `b.h`, since if `b.h` mentioned a definition at the wrong type, the C compiler would complain. Likewise, we ensure that `a.c` has the correct view of B's definitions, since it always includes B's interface file.

This rule also permits a more relaxed view of interface files. In particular, it is common practice in C programming to define a single `.h` to include declarations for many `.c` files, or even multiple `.h` files to define the interface for more than one `.c` file. For example, Figure 3 defines two `.c` files and two `.h` files in which the combination of both `.h` files serves as the interface for both `.c` files. This satisfies our rule because both files include both headers, and thus when each links a symbol defined by the other file, they are sure to be including a common header file, and thus are treating symbols at the correct types.

Rule 2: With the exception of a designated `config.h`, header file inclusion must be order-independent. As mentioned earlier, the fact that `.h` files may contain preprocessor directives like `#if` and `#ifdef` means that a `.h` file is really a *function* from which `cpp` computes an interface file. Treating interface files as functions can be quite useful, and is possible in other languages such as in Haskell [12] and ML [19, 20]. However, the tricky part here is that an included header file may also include

precision.h:
```
#if defined(__linux__) || \
    defined(__FreeBSD__)
  #define COMPRESSED
#endif
```

protos.h:
```
#ifdef COMPRESSED
  int *bitmap;
  void set_bit(int *, int);
  void initialize_bitmap(int *);
#else
  int **bitmap;
  void set_bit(int **, int);
  void initialize_bitmap(int **);
#endif
```

order-indep.c:
```
#include <stdlib.h>
#include "protos.h"     // <--
#include "precision.h"  // <--

int main(int argc, char **argv) {
  initialize_bitmap(bitmap);
  set_bit(bitmap, atoi(argv[1]));
  ...
  return 0;
}
```

order-impl.c:
```
#include "precision.h"
#include "protos.h"

#ifdef COMPRESSED
  void set_bit(int *bitmap, int bit) { ... }
  void initialize_bitmap(int *bitmap) { ... }
#else
  void set_bit(int **bitmap, int bit) { ... }
  void initialize_bitmap(int **bitmap) { ... }
#endif
```

Figure 4: Example illustrating violation of order-independent header inclusion.

`#defines` which affect the compilation of subsequent headers.

Consider the project shown in Figure 4. The code aims to implement a library for bitmap manipulation. In an attempt to write space-optimized code for Linux systems, the designer has chosen to do individual bit operations on those systems, as determined by the `COMPRESSED` flag, while otherwise he uses an array of `ints`. Such an implementation difference often occurs in the implementation of file descriptor sets.

It is easy to see that there is a bug in the code introduced because of the interdependencies in the header files. In particular, the headers `precision.h` and `protos.h` are included in a different order in `order-indep.c` than in `order-impl.c`. As a result, within `order-indep.c` on a Linux system the header `protos.h` is compiled with the `COMPRESSED` flag disabled, while for `order-impl.c` it is compiled with the flag enabled. Despite the resulting type mismatch, the objects are linked (by name) without complaint, so that compilation proceeds smoothly but introduces runtime bugs. Moreover, this program does not violate our first rule: the common symbols `set_bit` and `initialize_bitmap` appear in `protos.h`, which is shared between the two `.c` files.

CMOD prevents such an ordering dependence between headers to make local reasoning more transparent, with one exception. It is common for systems to define a single `config.h` that contains a list of `#defines` used to configure the system, e.g., to enable or disable certain features during compilation. CMOD supports such a `config.h` provided it is included first by every linked source file, ensuring that its effects are uniform. We similarly require that each linked file is compiled with the same command-line `-D` options and search path (`-I` flags). In the example in Figure 4, we could specify to CMOD that `precision.h` serves as the configuration file; CMOD would emit an error indicating that this file should appear first in all sources files that include it, fixing the error.

Finally we note that this rule naturally supports the common `#ifndef` pattern used to prevent recursive inclusion of a header file. This pattern is shown for `prototypes.h` in Figure 3. The `#define _PROTOTYPES_H` ensures that if `prototypes.h` ends up getting included multiple times while compiling a given source file, the second time this happens the `#ifndef _PROTOTYPES_H` directive will no longer be true. While in this pattern a `#define` is affecting how a header file is processed, it is a temporal *self*-dependence rather than a dependence between headers, and so it does not violate rule 2.

Rule 3: Header files must not contain (non-inlined or non-`static`) code and data definitions. The C compiler places no restrictions on the text in included files. It is therefore entirely possible to place actual definitions in header files, rather than just their types. However, this practice is undesirable for two reasons. First, if we are treating a `.h` file as an interface, which is meant to

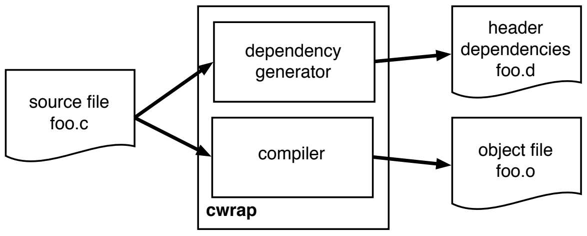

Figure 5: Operation of CMOD compiler wrapper *cwrap*.

describe the type of code, it would not be expected by the programmer that the file contain code itself. Second, if code is included it could be unsafe if other files link against the included definitions, since they may do so at incorrect types. For example, consider the following idiom, in which there is a header `globals.h`:

```
#ifdef MAIN
  int g;
  int* px;
#else
  extern int g;
  extern int *px;
#endif
```

By design, `globals.h` is to be included by those files in the system that must access global variables. One file, prior to including this file, would `#define MAIN`, thereby including the definitions, rather than the declarations of the variables. While this idiom is convenient, the definitions in the first branch are never checked against the prototypes appearing in the second branch, which violates our first rule, and in a more complicated setting with many global variables could lead to mistakes. We believe that moving the first part of this file into a `.c` file directly and removing the `#ifdef` is of little burden, is safer, and makes the resulting code easier to read.

We allow `static` definitions in header files; though perhaps a questionable practice, these cannot result in a link error. We also allow `inline` functions, since once again each copy of the definition will be used consistently in the including file (as long as the macro environment is consistent in each case, which we enforce).

4 Implementation

CMOD is implemented as two programs, *cwrap* and *lwrap*, which are wrappers for the C compiler and linker, respectively. To use CMOD simply requires modifying the `Makefile` or environment variables to use these wrappers. We also have a variant of *lwrap* that wraps `ar` for checking library creation.

Figure 5 illustrates the usage of *cwrap*. It takes as input the compiler source invocation—the source file

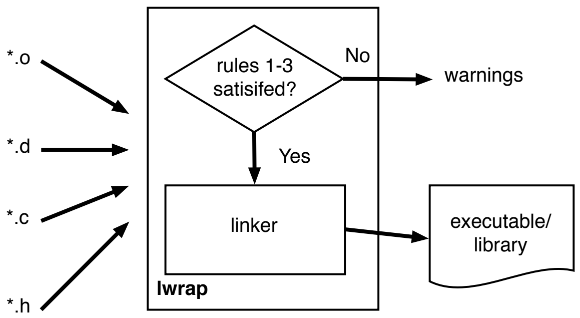

Figure 6: Operation of CMOD linker wrapper *lwrap*.

`foo.c` and any compile flags. From this information it generates a *dependency file* `foo.d` that lists all of the non-system header files (recursively) included by `foo.c`. Preprocessor definitions and search path information (i.e., `-D` and `-I` flags) is added to the end of the file for later checking. After generating this file the compiler is invoked to generate the object file as usual.

Figure 6 illustrates the usage of *lwrap*. It takes as input all of the `.o` files it would normally link, as well as the corresponding source and dependency files; the relevant `.h` files are gathered from the contents of the provided `.d` files. *Lwrap* then examines these files to check whether our three rules are satisfied, as we explain in detail below. The linker is invoked as usual to generate the library or executable; any violations of the rules are emitted as warnings to be examined by the user.

Our goal has been to develop these wrappers by using existing tools as much as possible. For example, we generate `.d` files by invoking `gcc -c -MMD`. We check for rule violations by using a combination of the system preprocessor (e.g., `gcc -E`), `ctags` for extracting information about definitions and declarations, the *objutil* utilities (like `nm`) for examining object file symbols, and `grep` for finding order dependency information, along with roughly 2K lines of glue code. This keeps our implementation simple, and makes it easy to incorporate improvements and bug fixes in the supporting tool chain. Since none of these tools is source language-specific (i.e., our rules ensure type compatibility is checked by the underlying compiler), we believe our tool could be used for other C-like languages that use preprocessor-based modularity conventions, such as C++ or Cyclone.

4.1 Rule checking with *lwrap*

Cwrap gathers relevant information at compile time while *lwrap* checks for rule violations at link time. We consider each rule in turn.

Rule 1: Whenever one file links to a symbol defined by another file, both files must include a header that contains the type of that symbol. To enforce this rule, for each symbol S *lwrap* must find which `.o` file provides S, and which `.o` files refer to S. Call the former `defS.o` and the latter `useS.o`. Having done this, *lwrap* must find a header included by both `defS.c` and `useS.c` that defines the type of S. This is done by finding the headers in `defS.d` and `useS.d`, and then seeing if the type of S appears in one of these. To determine this, for each common header we run `ctags` over the preprocessed file (using the flags stored in the dependency file), looking to see whether at prototype for S appears. If there is no such prototype for all headers, then this results in a rule violation.

In our implementation, the information used to make these checks is loaded into caches in memory, to improve performance. For example, we generate a hash table that maps symbol names to records, where the first field contains the name of the defining file (e.g., `defS.o`), and the second field is a list of the referring files (e.g., `useS.o`). We similarly use an in-memory representation of the `.d` file, and a hashtable that maps variable names to the `.h` files that include them.

Rule 2: With the exception of a designated `config.h`, header file inclusion must be order-independent. We check order-independence in a piecewise manner, considering all pairs of `.h` and `.c` files to look for dependencies. In particular, for a pair of headers I_a, I_b, we have $\mathit{Indep}(I_a, I_b)$ if and only if every preprocessor conditional branch in I_a references macro names which are not (un)defined (by `#define` or `#undef`) in I_b and vice versa. We similarly define $\mathit{Indep}(I_a, S_b)$ where S_b is some source `.c` file if and only if every preprocessor conditional branch in I_a references macro names which are not (un)defined in S_b.

Let S_k be the kth source file and $\mathcal{I}_{S_k}$ be the set of headers included by S_k, as determined by its dependency file (S_k`.d`). Then we can define $\mathbf{clean}(S_k)$ as follows:

$$\begin{array}{lll}\mathbf{clean}(S_k) & \mathit{iff} & \forall I_a \in \mathcal{I}_{S_k}.\mathit{Indep}(I_a, S_k) \text{ and} \\ & & \forall I_b \in \mathcal{I}_{S_k}.\mathit{Indep}(I_a, I_b) \\ & & \quad (\text{when } I_a \neq I_b)\end{array}$$

From this, we can say that the program's interfaces are order-independent as long as (1) $\forall S_k, \mathbf{clean}(S_k)$ and (2) each S_k was compiled using the same preprocessor directives and search path (`-D` and `-I` flags), as determined by its `.d` file.

We implement the $\mathit{Indep}(I_a, I_b)$ as a script using `grep`. In particular, for each file I_a and I_b we extract out two sets: D_I contains those macro names that are either defined or undefined by I, and C_I contains those macro names mentioned in conditional directives within I, like `#ifdef` and `#ifndef`. If either $C_{I_a} \cap D_{I_b}$ or $C_{I_b} \cap D_{I_a}$ is non-empty, then we have found a potential dependency.

The use of a `config.h` file as is typical in open source software could violate this check, since it may define macro names that are designed to effect the compilation of other headers. To support this idiom's safe usage, we do two things. First, we do not consider `config.h` as part of the independence check. Second, we ensure that `config.h` is included as the first header in every source file S_k. This way, we are sure that its definitions are used uniformly throughout the linked code and its headers. Note that we must also use the definitions present in `config.h` when preprocessing files in CMOD, e.g., prior to using `ctags`.

Rule 3: Header files must not contain (non-inlined or `static`) code and data definitions. By now we are aware of all header files in use by the system, so we simply preprocess these files, and use `ctags` to extract their definitions. We flag an error if any of these are definitions, rather than declarations, unless they are `static` or specified to be in-lined.

5 Experimental Results

To see how effective CMOD could be in practice, we incorporated it into the build process of a variety of open-source programs. This section reports on our experience and measures the CMOD's performance.

5.1 Benchmark Programs

Our benchmark programs fall into a range of application classes: network servers and daemons (`vsftpd`, `xinetd`, and `zebra`), a remote login utility (`openssh`), and various utility programs (`bc`, `gzip`, `flex`, and `rcs`). In some of these applications, security is an important consideration, so we would expect careful coding practices. In total, these applications comprise more than 225K lines of code. They are summarized in Table 1. In general, incorporating CMOD into the build process required only trivial changes to the environment, with no changes required to the code itself.

These programs were structured using similar modular patterns. Preprocessor directives to varying degrees, with `bc` and `gzip` being the most heavy [7]. Nearly all programs used the `config.h` idiom, where this file was most often generated by `autoconf`. Most programs followed the practice of defining a separate header for each source file. However, `rcs` was unusual in that for 20 source files there were only 4 header files containing relevant shared definitions. Nonetheless, as we will see, this practice largely conformed to rule 1, keeping CMOD happy.

Program	Lines of Code	Description
gzip-1.2.4	8k	compression utility
bc-1.06	11k	text-based calculator
vsftpd-2.0.3	14.5k	“Very Secure” FTP daemon
flex-2.5.4	15.5k	Code generation tool
rcs-5.7	18.5k	Version control management system
xinetd-2.3.14	24.4k	Generic network services daemon
zebra-0.94	57.4k	Back end for various routing daemons
openssh-4.2p1	76k	Secure Shell remote login

Table 1: Benchmark Program Characteristics.

Program	Distribution of Warnings			Changes required			Build time	
	rule 1	rule 2 (false alarms)	rule 3	rule 1	rule 2	rule 3	stock	CMOD (% overhead)
gzip	2	0	0	*impos*	0 / 0	0 / 0	2.9s	6.0s (107%)
bc	0	2	0	0 / 0	-2 / +2	0 / 0	12.4s	18.3s (48%)
vsftpd	1	6	0	0 / +1	-10 / +4	0 / 0	13.7s	22s (61%)
openssh	12	28	0	-7 / +8	-5 / +5	0 / 0	2m 22s	5m 6s (114%)
flex	0	4	0	0 / 0	-4 / +4	0 / 0	1.73s	2.6s (52%)
rcs	0	2	0	0 / 0	-2 / +2	0 / 0	3.1s	8.2s (164%)
xinetd	2	0 (10)	0	0 / +2	0 / 0	0 / 0	6.2s	11.5s (85%)
zebra	46	10	0	0 / +18	-10 / +10	0 / 0	31.1s	50.6s (63%)

Table 2: Output and performance of CMOD on benchmark programs.

5.2 Warnings emitted by CMOD

Table 2 summarizes the results of using CMOD when building these applications. Each package is listed in the first column, and the next three columns tabulate the warnings issued by CMOD, separated by which of the three rules defined in Section 3 were violated. For rule 2 we also list false alarms in which the tool conservatively reports a potential order dependence, but in face no such dependence exists. The next three columns indicate the number of lines of code that needed to be changed to eliminate the respective warnings. The last two columns report performance measurements which we discuss in the next subsection.

There are three immediately noteworthy aspects of the table. First, the number of total warnings is fairly small. The largest category is rule 1 for zebra. Second, the warnings can be fixed with small changes to the code base. Finally, there were no violations of rule 3. Note that in no case did we discover any true linking errors in which one file treated a symbol as having a type different than the type of the symbol's actual definition.

In the remainder of this section we elaborate on the kinds of coding practices that led to the reported rule violations.

Practice 1: Prototypes in C code rather than header files. According to rule 1, all prototypes for linkable functions or other definitions appearing in a module should appear in a header file. This file should be included by users of that module. However, in the case of `zebra` we found this rule was frequently violated by placing prototypes for external functions directly in C code instead of including the appropriate header. Here is an example in which the programmer has chosen to include the prototypes for `rib_weed_tables` and `zebra_vty_init` as local declarations within `main`:

```
int  main (int argc, char **argv) {
  char *p;
  ...
  void rib_weed_tables ();
  void zebra_vty_init ();
  ...
}
```

25 of the 46 rule 1 violations of `zebra` were of this flavor, and most rule 1 violations in other programs were as well.

Practice 2: A file does not include its own header. To satisfy rule 1, all `.c` files that refer to a certain symbol—whether defining it or referring to it—must include a header file that includes the symbol's type. In many cases we found that the file defining a symbol *did* have a corresponding header file, but did not include that file itself. For example, in openssh there exists a file sshlogin.c whose interface is sshlogin.h, but the file does not include this interface itself. The rule 1 violation for vsftpd was of the same category, and the remaining 21 of the 46 rule 1 violations in zebra were of this flavor. Interestingly, whether or not a file included its own header was not consistent: in many cases, files in these packages *do* include their own header file, but sometimes they do not. We feel this illuminates the ad hoc nature by which programmers use headers as interfaces owing to the fact that the development tool set provides no enforcement.

Practice 3: Order dependence between headers and source files: One way in which rule 2 could be violated is by having an order dependence between a source file and a header file. In particular, the source file might perform a #define prior to including a header file, and this changes the way the header's contents are interpreted. Consider the following set of code fragments taken from vsftpd:

str.h:

```
#ifdef VSFTP_STRING_HELPER
#define str_alloc_memchunk \
        private_str_alloc_memchunk
...
```

filestr.c:

```
#define VSFTP_STRING_HELPER
#include "str.h"
...
```

secutil.c:

```
#include "secutil.h"
#include "str.h"
...
```

The definition of str_alloc_memchunk in str.h depends on whether VSFTP_STRING_HELPER is defined. Thus the programmer has essentially parameterized the header file (str.h) on the macro. However, notice how filestr.c includes str.h after defining VSFTP_STRING_HELPER while secutil.c includes str.h without doing so. This is dangerous because in these cases the inclusion of the header leads to non-uniform declarations.

On the other hand, CMOD does allow idioms like that used in str.h, but requires that #defines like VSFTP_STRING_HELPER occur in a single config.h that is included first in every source file. This convention ensures that all headers are treated uniformly.

Practice 4: Order dependencies between headers (other than config.h) that contain global #defines: In some cases we found that programmers distribute compilation-affecting #defines over several header files, rather than accumulating them all in a single config.h, and subsets of these headers were included by different source files, potentially leading to subtle inconsistencies. Most of the 28 order-dependency violations in openssh exhibited this practice. These were localized among a few files were easily resolved by moving a few #defines into config.h. The order dependences in zebra, bc, and flex were also predominately of this flavor.

Other Warnings In gzip, CMOD reported a violation of rule 1 in which no common header was found for a symbol common to match.o and deflate.o. Unfortunately, match.o is generated from compiling the assembler file match.S and so this error cannot be fixed.

As mentioned in Section 3, rule 2 supports the standard pattern used to avoid recursive inclusion of header files. However, vsftpd sometimes checked the definedness of a macro name not in the file itself, but in a file that included it, creating a spurious order dependence. For example:

filesize.h:

```
#ifndef VSF_FILESIZE_H
#define VSF_FILESIZE_H
...
#endif /* VSF_FILESIZE_H */
```

session.h:

```
...
#ifndef VSF_FILESIZE_H
#include "filesize.h"
#endif
...
```

Here, as usual, filesize.h uses the macro name VSF_FILESIZE_H to prevent multiple inclusion. However, rather than checking this macro's definedness only within filesize.h itself, the developers also check for it within in other files, here shown in session.h. This unnecessary—the surrounding #ifndef can be safely removed, eliminating the order dependence between the two headers.

The rule 2 violations for xinetd are false alarms, in that the tool is overly conservative in its definition of dependence.

5.3 Performance Overhead

CMOD runs as a wrapper around the system linkers and compilers and hence is bound to incur some overhead.

We measured the time it took to build each software package with and without CMOD; the elapsed time for each case is reported in the last two columns of Table 2; the overhead of CMOD relative to the stock build time is shown in parentheses. These measurements are the average of three runs taken on a dual 2.8 GHz Xeon with 2 GB of DRAM running Fedora Core 3, kernel version 2.4.21.

Overhead ranges from a 48% to a 164%. As might be expected the overhead due to *cwrap* is negligible while the overhead due to *lwrap* is significant as it must do a lot of additional checks before proceeding to link the objects. Thus the build proceeds in stages with large pauses at the intermediate linking operations.

If CMOD is to be used regularly during development, the overhead due to *lwrap* must be reduced. Our current prototype implementation does not take advantage of some obvious optimizations that can be utilized for reducing the complexity of the algorithm. In its current form the complexity is polynomial in the number of includes in each file. But the powers of the polynomial are prohibitively high. We have preliminary ideas on how to reduce this complexity to at most quadratic. We should also be able to make the algorithm incremental so that each time the linker is run, it need only reconsider modules that have changed since the last link. In the meantime, it is easy enough to do a strawman version of this: simply avoid running *lwrap* at every compile, but rather do so only at various milestones or when trying to rule out sources of bugs.

6 Related Work

Modular programming constructs are supported by a variety of modern programming languages, such as ML [4, 15], Haskell [13], Ada [1], and Modula-3 [9]. A nice overview of module programming mechanisms appears in Pierce [19]. Most of these languages have fairly sophisticated module systems in which interface files can be computed, e.g., by providing type parameters [12, 13], or overriding abstract type declarations [15, 4]. Ramsey et al [20] propose to extend ML with support for constructing interfaces from smaller pieces, to better support software evolution. In C programming practice, `cpp` directives are the de-facto module programming language. Its computations can be sophisticated but dangerous; CMOD's rules attempt to find the sweet spot in this space.

There are a number of existing proposals for C or C++ module systems, most of which focus on organizing code into *components*. Examples include Knit [21], Koala [25], and Click [16]. In each setting, components are specified as having explicit *exports* and *imports*, and the linking process is not implicit and by name, but may be directed by the programmer. For example, one can define a Koala component by referring to two existing components, and then using the `connect` primitive to link together the exports of one to the imports of the other. This is similar to *functor application* in ML-like languages [19, 15, 4]. Explicit linking allows a component to be included potentially many times within a single program. Microsoft's Component Object Technologies (COM) model [5] is another component-based system that can be used for organizing C modules within dynamically linked libraries (DLLs). While useful, component-oriented programming is the exception rather than the norm, and these systems do nothing the enforce the programming practice of treating files as modules and header files as interfaces.

Cyclone [10], a type-safe dialect of C, originally contained a module system following earlier work for Modular Typed Assembly Language (MTAL) [8]. Like C code, Cyclone files may `#include` header files, but these files were not treated as interfaces by the language. Instead, a separate interface file also had to be present for each object file, and was based on *post-processed* Cyclone code. Interface files were checked for consistency during compilation and linking. However, the approach was ultimately abandoned because of the disconnect between the C-style convention of using `.h` files as interface files and Cyclone's native interface files—programmers felt it was unnatural to use both. Because CMOD is source-language independent, and uses `cpp` as a subroutine, it could easily be applied to Cyclone programs to support modular programming.

Ernst et al [7] study preprocessor usage in a variety of programs, categorizing and explaining various idioms of use. They observe that C preprocessor use, particularly the use of macro expansions, can be quite complicated. For this reason, Ernst and others suggest that preprocessor use be greatly curtailed (others suggest eliminating `cpp` altogether [14]). Some suggestions concern the use of `cpp` for modular programming. For example, an adaptation of the Indian Hill C Style and Coding Standards [2] suggests that `#ifdef` should only appear in header files, and that header files should not be nested. This is more restrictive, but similar in spirit, to our requirement that headers should be order-independent. Stroustrup [24] suggests `#include` should be used for importing declarations, but would prefer to move `#include` into the proper language, as was done with the `#import` directive for Objective-C [6]. In a sense, CMOD enables this idea by enforcing a particular safe usage of `#include`. Spencer and Collyer [22] suggest that uses of `#ifdef` be eliminated altogether in favor of search path adjustments and scripts for finding different files. CMOD relies on the preprocessor only to include imported definitions so they can be checked for consistency by the C compiler; reduction of other cpp features would simplify and speed up its operation.

A number of tools automatically check for erroneous or questionable uses of cpp directives, including `lint` [11], PC-lint [18], and Check [23]. However, the detected bug patterns are fairly localized and generally concern problematic macro expansions; none of these tools enforces a CMOD-style discipline of modular programming.

7 Conclusions and Future Work

Though the C programming language does not provide linguistic support for modular programming, most programmers adhere to a discipline of modular programming that treats `.c` files as modules and `.h` files as interfaces. We have developed CMOD, a front-end for the standard C compiler and linker, to enforce correct use of this discipline. Using CMOD allows programmers to more easily detect mistakes that could lead to subtle errors, and ensures their code is more maintainable. Experiments on a variety of open source programs show that by and large programmers adhere to the modular discipline enforced by CMOD, but nonetheless violate it in various ways, all of which are easily fixed. Our results suggest to us that CMOD can be a useful tool in the hands of the disciplined developer.

While we believe that CMOD's rules are correct, in the sense that their use implies the linked program will be type-correct, we have no proof of this. We plan to develop a small formalism that captures the essence of the preprocessor to model its interaction with header and implementation files. The goal will be to define a notion of type-correct linking (e.g., based on the approach of MTAL [8]), and then prove that a program whose modules adhere to our rules will always be type correct when linked. We plan to continue to experiment with CMOD on open source programs, to identify any other rules or idioms that are worth enforcing. We also intend to improve the performance of our prototype implementation by using more clever algorithms and data structures.

Availability

CMOD is free software released under the GNU Public License. The source and documentation for it can be obtained from our web site at

`http://www.cs.umd.edu/~saurabhs/CMod/`

Acknowledgments

This work was funded in part by National Science Foundation grant CCF-0346989. The authors thank Iulian Neamtiu and Nikhil Swamy for helpful comments on drafts of this paper.

References

[1] BARNES, J. Ada 95 rationale: The language, the standard libraries. *Lecture Notes in Computer Science 1247* (1997).

[2] CANNON, L., ELLIOTT, R., KIRCHOFF, L., MILLER, J., MITZE, R., SCHAN, E., WHITTINGTON, N., SPENCER, H., KEPPEL, D., AND BRADER, M. *Recommended C Style and Coding Standards*, sixth ed. 1990.

[3] CARDELLI, L. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)* (New York, NY, USA, 1997), ACM Press, pp. 266–277.

[4] CHAILLOUX, E., MANOURY, P., AND PAGANO, B. *Développement d'applications avec Objective Caml*. O'Reilly, France, 2000. An English translation is currently freely available at `http://caml.inria.fr/oreilly-book/`.

[5] COM: Component object model technologies. `http://www.microsoft.com/com/default.mspx`.

[6] COX, B., AND NOVOBILSKI, A. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[7] ERNST, M. D., BADROS, G. J., AND NOTKIN, D. An empirical analysis of C preprocessor use. *IEEE Transactiosn on Software Engineering 28*, 12 (2002), 1146–1170.

[8] GLEW, N., AND MORRISETT, G. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)* (1999), ACM Press, pp. 250–261.

[9] HARBISON, S. *Modula-3*. Prentice-Hall, 1992.

[10] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), USENIX Association, pp. 275–288.

[11] JOHNSON, S. Lint, a C program checker. Tech. Rep. 65, Bell Labs, Murray Hill, N.J., Sept. 1977.

[12] JONES, M. P. Using parameterized signatures to express modular structure. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL)* (1996), ACM Press, pp. 68–78.

[13] JONES, S. P., AND HUGHES, J., Eds. *Haskell 98: A Non-strict, Purely Functional Language*. `http://www.haskell.org/onlinereport/`, 1999.

[14] MCCLOSKEY, B., AND BREWER, E. ASTEC: a new approach to refactoring C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), ACM Press, pp. 21–30.

[15] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[16] MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)* (1999), ACM Press, pp. 217–231.

[17] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (1972), 1053–1058.

[18] PC-lint/FlexeLint. `http://www.gimpel.com/lintinfo.htm`, 1999. Product of Gimpel Software.

[19] PIERCE, B. C., Ed. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[20] RAMSEY, N., FISHER, K., AND GOVEREAU, P. An expressive language of signatures. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2005), ACM Press, pp. 27–40.

[21] REID, A., FLATT, M., STOLLER, L., LEPREAU, J., AND EIDE, E. Knit: Component composition for systems software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2000), pp. 347–360.

[22] SPENCER, H., AND COLLYER, G. #ifdef considered harmful, or portability experience with C News. In *Proceedings of Usenix Summer Technical Conference* (June 1992), pp. 185–197.

[23] SPULER, D., AND SAJEEV, A. Static detection of preprocessor macro errors in C. Tech. Rep. 92/7, James Cook University, Townsville, Australia, 1992.

[24] STROUSTRUP, B. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[25] VAN OMMERING, R., VAN DER LINDEN, F., KRAMER, J., AND MAGEE, J. The Koala component model for consumer electronics software. *IEEE Software* (2000).