A UNIX tool for managing reusable software components

John Plaice* William W. Wadge†

Abstract

SLOTH is a simple UNIX-based tool for creating and maintaining large C programs built from reusable modules. SLOTH is a collection of UNIX shell commands for creating and editing modules, and for producing executables by linking compiled code from the modules comprising a program. SLOTH in a sense extends C by providing module facilities similar to those built in to newer languages such as MODULA–II. The SLOTH extensions, however, are at the shell level only. No changes are required in C itself; in fact, the SLOTH commands invoke the standard UNIX C compiler.

A SLOTH module contains a set of C procedures and local declarations, a set of global definitions and variable declarations, an initialization routine and an import list of other modules containing declarations referred to, but not defined by, the module in question. The SLOTH link command has one argument, the name of a root directory. The link command automatically identifies all those modules imported directly or indirectly by the root module, checks that the object code for each module is up-to-date and recompiles where necessary. It then creates a main program in which the various initialization routines are executed in the correct order (most primitive first), compiles it and links it with all the object files and produces an executable program.

Individual modules can be compiled without any knowledge of the application that uses, directly or indirectly, the module in question. Two applications that both use a collection of modules can therefore share the *object* code for these modules.

The use and implementation of SLOTH are presented, as are the experience in using it, programming techniques developed to take full advantage of its facilities, as well as several extensions, either completed or planned.

Keywords

programming environments, software engineering, programming languages,
reusability, version control, modules for C

Introduction: modular programming in C

The C programming language is currently enormously popular, especially for UNIX applications, but it hardly represents the state of the art. It was originally designed in the early seventies and therefore lacks several very important features found in more modern languages, such as ADA or MODULA I-III. For example, C compilers do almost no static type checking; there are no objects and no information-hiding module constructs, nor is there any inheritance.

Researchers have followed two different approaches in the various attempts to remedy C's defects without a complete break with its basic design, with the close connection with UNIX or with the huge

*Address: John Plaice, Dpartement d'informatique, Universit Laval, Ste-Foy, Qubec, Canada G1K 4P7; e-mail: John.Plaice@ift.ulaval.ca .

†Address: William W. Wadge, Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., Canada V8W 3P6; e-mail: wwadge@csr.csc.uvic.ca .

amounts of existing C software. The first is to extend the language itself, by adding new constructs and entities. This internal approach is the one taken in C++ [9, 10], an object-oriented extension of C. The second is to leave the language unchanged but to develop new C programming tools that handle at least some aspects of the problem. One good example of this external approach is LINT, the C verifier utility. LINT checks for many kinds of possible errors and does a more thorough job of type checking than do many C compilers, which do not even count arguments to functions.

There are clearly advantages and disadvantages to both approaches. The internal approach is usually more thorough, more convenient for the programmer and often compiled more efficiently, as no extra passes are needed. On the other hand, extending C is a major task because it involves altering the compiler. To make matters worse, some of these alterations may be machine-dependent. It may be a long time before the extension is accepted, if ever: it has taken several years for the ANSI and POSIX.1 definitions of C to become widespread on UNIX systems. Finally, if two separate problems are each remedied with separate extensions, integrating the two can be costly.

The external approach, by contrast, is usually much simpler. Creating a C programming tool is usually order(s) of magnitude simpler than designing and implementing a language extension. Furthermore, if the tool is written in a standard version of C, it is available immediately. Programmers do not have to switch to a new, possibly unreliable compiler. The main disadvantage is that there are some problems that cannot be completely solved outside the language.

This paper describes an external approach to adding modules to C. The SLOTH system consists of a simple scheme for representing C modules as UNIX directories, together with shell commands for building and modifying modules and for configuring applications. SLOTH modules have procedure definitions, local and exportable declarations, initialization routines and import lists. The SLOTH configuration command takes care of all the routine bookkeeping, such as collecting indirectly needed modules, keeping `.o` files up-to-date (using the standard compiler), generating `#include`'s and ensuring that initializations are performed once each and in an acceptable order. The main weakness is that SLOTH is unable to enforce information hiding between modules.

The module import relation

Modularity is a deceptively simple idea, but one that can be difficult to achieve in practice. Modularity is especially difficult if modules are to be reusable, i.e., if a given module can be included as a component of two or more different applications (programs).

The main complication is that, in general, the modules used to build applications are not independent. Procedures in one module may call external procedures used in other modules, manipulate variables declared in other modules or declare variables to be of types used in other modules; in general, a module may reference entities in other modules. Furthermore, the modules from whom these entities are imported may in turn import entities from other modules, which in turn might need other modules; the dependency relation can be arbitrarily long.

A small example may help clarify the notion of a module import relation. Suppose that the `calc` module contains a desk calculator program. The calculator manipulates expressions, uses a stack and performs various mathematical operations. Suppose further that these functions are not implemented in `calc`, but that instead `calc` imports procedures, types and constants found in modules `expr`, `stack` and `math`. Using the standard terminology, `calc` *imports* the modules `expr`, `stack` and `math`, i.e., these three constitute the *import list* of the `calc` module. Suppose also that both expressions and stacks are implemented using lists, as provided by a `list` module, and that the `list` module in turn imports a `heap` module. Finally, suppose that the `ed` module contains a separate line editor application, that the editor uses stacks and arrays, that arrays are provided by a module `arr`, which also imports `heap`, and that the editor shares the `stack` module with the desk calculator.

If there are no other imports, Figure 1 describes the import relation between the eight modules. It also illustrates the kind of sharing that is typical, or at least possible, even in small software projects. The calculator and editor directly share the `stack` module but also indirectly share the `list` and `heap` modules. There is no direct reference to the list or heap primitives in the calculator or editor source, but in

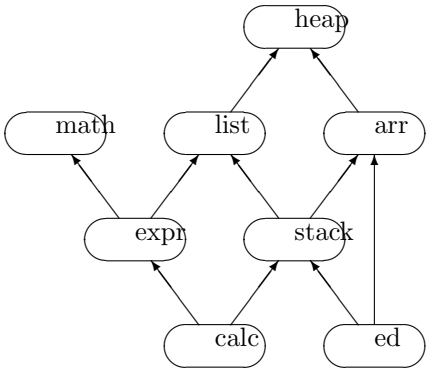

Figure 1: Dependency relation betwen modules

configuring either application these modules must be, at some point, compiled and linked in. Figure 1 also illustrates what might be called internal sharing. The calculator application uses lists, but in two distinct ways: to implement expressions and to implement stacks. The `list` module is shared by two others and all three are components of the same system.

In general, the import relation between modules can be very complex. The only restriction that Sloth imposes is that there be no cycles, i.e., that the module dependency relation forms a directed acyclic graph (dag).

Problems with sharing

The above example also illustrates some of the problems that sharing, or even just importing, causes.

First, when configuring an application to produce an executable, all up-to-date object files for all the modules, directly used or indirectly used, must be linked in. In other words, the *transitive closure* of the import relation must be computed. There do exist simple devices (such as allowing nested `#include`'s) that sometimes compute the transitive closure correctly, but these devices usually fail in the presence of sharing, because they include multiple copies of popular modules. This last problem can be remedied by using `#ifdef`'s and `#ifndef`'s, but only at the cost of complicating the include files.

The C macro facility also complicates the situation. Macro expansion can introduce references to external primitives that are not clear in the source. The result is that even in compiling a module it is safest to include the external declarations of all modules used directly *or* indirectly.

Furthermore, it is usually not enough just to compute the unordered *set* of modules used directly or indirectly. Initialization is a case in point. Modules that encapsulate a hidden data structure—such as a stack—usually need an initialization procedure that must be performed when the application is launched—at *powerup*, so to speak. Care must be taken that these powerup procedures are performed in a consistent order, so that no attempt is made to initialize a particular module without first initializing all the modules it depends on. A module's powerup routine cannot simply initialize the modules it uses, because then some popular modules will be initialized many times.

As before, a list must be compiled, without repetitions, of all the modules imported directly or indirectly by the root module. And for the purposes of initialization, the list must be be in a most-primitive-first order: this is the *uselist* of the module, created by a topological sort on the module import relation.

This calculation is easy enough to perform by hand but is error-prone and becomes tedious when there are more than a dozen or so modules involved. The problem is aggravated by the fact that the module structure will change as the application(s) evolve. For example, if a primitive module is split into two, the uselist of every module that imports it directly or indirectly must be changed.

The traditional C/unix approach

The C language does not have any module construct similar to those provided by MODULA–II and even by several PC PASCAL systems. It does, however, allow programs to be divided into a number of `.c` and `.h` files, and these can be compiled separately.

Programmers often use this facility to provide at least an informal module structure. Each of these modules consists of a `.c` file of procedures and variable declarations and a `.h` file of declarations—including procedure headers, type definitions and `#define`'s—for use in other files. The `.c` file roughly corresponds to the implementation part of a module, and the `.h` file roughly corresponds to the interface. Makefiles are used to ensure that compiled code is up-to-date.

The traditional approach is adequate for single projects (no sharing) that involve a small number of modules (one or two dozen). It becomes impractical when there are very many modules, shared among more than one application. The problem is that programmers must do almost everything by hand. They must make sure that all the required `.h` files are included and that externally defined procedures and variables are so declared. The programmers must write initialization code for each application; this code must call all of the individual powerup procedures in a consistent order. Finally, they must write a makefile and ensure that a `.o` file is defined in terms of all the `.h` files that the corresponding `.c` file needs.

New problems are added when new modules are introduced and/or the import relations change. The situation is complicated by the fact that a `.c` file may—because of macro calls—need to include `.h` files for a module even when there are no explicit references to names declared in the corresponding `.h` file. As a result, changes made even in a module not directly used will still require editing the `.c` file and updating the makefile entry.

Suppose, for example, that a module `A` now depends on a new module `B`. In general, `#include`'s to `A.c` must be added to bring in `B`'s declarations. But that is not enough: the same must be done for *all* of the modules that need `A`'s declarations; as was just explained, this could be all modules that use `A`, directly or indirectly. The makefile entries for each of these modules must be updated, and the initialization sequence for any application involving `A` must also be recomputed. In other words, it might be necessary to change almost all of the `.c` files in an application, and almost all the lines in its makefile.

This process can clearly be very time consuming. It is also very error-prone. The same information (essentially, the uselists) is encoded in at least three separate places: in the `#include` lists, in the makefile entries and in the main initialization routine. The danger is that these three forms will become inconsistent, so that, for example, some powerup routine is never called.

The traditional approach therefore creates a powerful disincentive to modularization. As the number of modules increase, the overhead becomes unmanageable, since the size of the makefiles and `#include` lists grows more or less quadratically as a function of the number of modules. As a result, programmers tend to use large modules that, because of their size, are unlikely to be reusable without alteration.

The Sloth approach

SLOTH originated as a fairly simple attempt to generalize the traditional C/UNIX approach described above and to automate the bookkeeping involved. It allows the programmer to use UNIX's file and directory facilities to impose a modular structure on a C application. SLOTH takes over the job of computing and maintaining uselists, generating `#include`'s and external declarations, and producing initialization code. It also takes over from MAKE the task of ensuring that `.o` files are up-to-date. In so doing it relieves the programmer of the burden of maintaining makefiles. SLOTH even generates the commands to compile and link an application.

A SLOTH module consists of a number of components, some, like the import list, provided by the programmer; others, like the object files, automatically produced by SLOTH. Each component is represented as a separate file. The files are gathered into a single directory that represents the module as a whole.

The SLOTH user does not normally look inside these module-directories, or directly manipulate the files therein, not even those edited by the user. Instead, the programmer uses the `vm` command to view components and the `mm` command to modify components. There is also an `mkm` command for creating new

blank modules. These three commands together shield the user from the details of SLOTH's representation of modules as directories.

Executables are produced using SLOTH's `lkm` command. One simply invokes the `lkm` command with the name, minus extension, of the main module of an aplication as argument. Assuming no errors are detected, the command eventually terminates and leaves, in the same directory, an executable file with the module's name.

The `lkm` command automatically tracks down all those modules required directly or indirectly by the main program. It checks that each `.o` file is up-to-date, recompiling if necessary. Next it creates a small C program whose main procedure simply calls the initialization routines of all the modules used directly or indirectly. These routines are called in a most-primitive-first order, so that the initialization routine for a given module is called only after calling the initialization routines of those modules used by the given module. The powerup routine of the root module is the last to be called, and therefore constitutes the program proper. Finally, `lkm` compiles the main program just described and links it with all the other `.o` files, yielding the executable version.

Notice that the powerup code of the main module is the body of the application configured using `lkm`. SLOTH does not need to distinguish between modules that can only be parts of programs and modules that are themselves programs.

The user-supplied components of a Sloth modules

A SLOTH module consists of a number of different components. While some of these components are user-created and maintained, others are created and maintained automatically. SLOTH users do not normally have to concern themselves with the latter.

The *import* component was presented above; it is a list of all the modules declaring entities referred to directly in code in the given module. It is entirely the user's responsibility to ensure that the import list is complete; SLOTH does not check, nor can it, since it does not deal with the semantics of C. The largest component of a module is usually the *procedure* component, which consists of C procedure definitions together with other declarations, e.g., of variables, intended to be local to the module. The module initialization is provided by the *body* component, which consists of C code to be executed on powerup; this code may call procedures defined in the procedure component and use other entities declared there. The body can also have references to external entities defined in modules on the import list. The *variable* component contains declarations of variables that the programmer wants to be visible to modules using the module in question. Finally, the *definition* component contains other declarations that the programmer wants to be visible to other modules using the module in question. This component would typically contain `#define`'s, `typedef`'s, `struct` declarations and declarations for procedures in the procedure component that are meant to be available to modules using the module in question.

Each of these components is represented by a single file inside the directory that represents the module as a whole. The import component is in file `import`, the procedure component is in file `proc.i`, the body component in file `body.i`, the variable component in file `var.i` and the definition component in file `define.i`. The SLOTH user does not normally need to use these names because the `vm` and `mm` commands support one letter abbreviations. For example, the command `mm pbi list` will invoke the editor to modify, in turn, the procedure, body and import components of the `list` module.

For pragmatic reasons there are some simple syntactic constraints on the format of the data in these files. For example, the import list is represented one to a line, no leading spaces, with the module indicated as a relative path name without the `.m` extension. For example, if the import file contains the following three lines

```
../list
stack
sys/heap
```

it means that the module in question imports the `list` module in the parent directory, the `stack` module in the current directory and the `heap` module in the `sys` subdirectory. The order in which imported modules

appear is irrelevant.

The system-maintained components

The SLOTH user normally is not even aware of the module components created and maintained by SLOTH. However, to explain how SLOTH works and how the `lkm` command creates executables, the purpose of these components and the relationships that SLOTH maintains are presented.

The *uselist* component, already referred to, is a topologically sorted list of all modules used directly or indirectly by the module in question. The fact that it is topologically sorted means just this: if module `M` appears in the list and `M` uses module `N` indirectly, so that `N` is also in the list, then `N`, being more primitive, appears *before* `M` in the list. The uselist component is stored in a file of the same name, one per line, as *absolute* pathnames, without the `.m` extension.

The *external* component consists of all the external declarations required by modules using the one in question. When the external component is up-to-date it is the result of merging the definition component and the result of adding the keyword `external` before all the declarations in the variable component.

The `prog.c` file is a C source file that can be separately compiled. It consists of `#include`'s for all external declarations of all files on the uselist, an `#include` of the procedure component and an initialization procedure whose body is a `#include` of the body component of the module; the initialization procedure's name is `InitM`, where `M` is the name of the module. The `prog.c` file is up-to-date if it reflects the current or up-to-date state of all the files contributing to it. The `prog.o` file is the result of compiling the `prog.c` file. It is up-to-date if it is the result of compiling an up-to-date `prog.c` file.

Finally, application (root) modules also have a `main.c` file. This file consists of `#include`'s for all the external declarations plus a procedure `main()` whose body is a series of calls to the initialization routines of the procedures on the uselist, terminated by a call to the module's own initialization routine.

Notice that each of the system components is constructed from a number of other user or system components in the same module or in other modules on the uselist. A system component is called correct or up-to-date if it is the result of performing this construction on correct or up-to-date—as appropriate—components. For example, the uselist component is correct or up-to-date if it correctly reflects the *current* contents of the various import files involved.

Inside some simple modules

Two examples are given to clarify the presentations. The first example consists of all of the user-supplied components of a simple-minded `stack` (of integers) module implemented with an array and a pointer. The second example illustrates the text files automatically generated by SLOTH during the configuration of the `calc` module described in Figure 1.

The largest user-supplied file in a module is typically the `proc.i` file:

```
int starr[STSIZE];

void stpop()
/* pop the stack */
{ if (stptr==0)
     err("stack underflow");
  stptr--;
}

int sttop()
/* the top of the stack */
{ if (stptr==0)
     err("top of empty stack");
  return starr[stptr];
}
```

```
void stpush(i)
int i;
/* push i onto the stack */
{ if (stptr+1==STSIZE)
     err("stack overflow");
arr[++stptr] = i;
}
```

The variable `starr` is declared in the `proc.i` file, so it is local to the module and cannot be used outside of it. On the other hand, the `stptr` variable must be made be available to modules using this one (for reasons explained later), so it is declared in the `var.i` file:

```
int stptr;
```

The pointer is initialized in the `body.i` file:

```
stptr = 0;
```

The `define.i` file contains global declarations for the stack access procedures and defines a macro for fast stack popping:

```
void stpop(), stpush();
int sttop();
#define SIZE 200
#define fastpop() stptr++
```

It is because the `fastpop` macro produces a reference to the variable `stptr` that this variable must be declared globally in the `var.i` file. Finally, since this simple-minded implementation does not use lists or a separate array module, these do not need to be included in the import list. But the implementation does use an error procedure, assumed to be defined in module `error`. Therefore, the `import` file lists only this module:

```
error
```

The `calc` module is used to present SLOTH-created components. (To produce these examples a network of stub modules, with the given import relation, was created; the command `lkm calc` was then executed.)

The `uselist` brings in all the modules used directly or indirectly by `calc`:

```
/ursamajor/a/wwadge/projects/heap
/ursamajor/a/wwadge/projects/math
/ursamajor/a/wwadge/projects/list
/ursamajor/a/wwadge/projects/arr
/ursamajor/a/wwadge/projects/stack
/ursamajor/a/wwadge/projects/expr
/ursamajor/a/wwadge/projects/ed
```

Notice the order of the `uselist`: for example, the `list` module is used by the `stack` and `expr` modules, and appears before them in the list. Next, the `prog.c` file allows the recompilation of the module's procedures and body in the presence of all of the required external declarations:

```
#include "/ursamajor/a/wwadge/projects/heap.m/extern.i"
#include "/ursamajor/a/wwadge/projects/math.m/extern.i"
#include "/ursamajor/a/wwadge/projects/list.m/extern.i"
#include "/ursamajor/a/wwadge/projects/arr.m/extern.i"
#include "/ursamajor/a/wwadge/projects/stack.m/extern.i"
```

```
#include "/ursamajor/a/wwadge/projects/expr.m/extern.i"
#include "/ursamajor/a/wwadge/projects/ed.m/extern.i"

#include "/ursamajor/a/wwadge/projects/calc.m/var.i"
#include "/ursamajor/a/wwadge/projects/calc.m/proc.i"

Initcalc(argc,argv)
int argc;
char **argv;
{
#include "/ursamajor/a/wwadge/projects/calc.m/body.i"
}
```

Finally, the `main.c` file calls the powerup routines, including its own, in the correct order:

```
main(argc,argv)
int argc;
char **argv;
{
  Initheap(argc,argv);
  Initmath(argc,argv);
  Initlist(argc,argv);
  Initarr(argc,argv);
  Initstack(argc,argv);
  Initexpr(argc,argv);
  Inited(argc,argv);
  Initcalc(argc,argv);
}
```

Creating an application

The whole point of SLOTH, of course, is to produce executable files that correspond to the current contents of the application source code, as distributed throughout the relevant modules. These executables are created with the `lkm` command, which in the simplest case has exactly one argument, namely the name of the application's root module. For example, to create the calculator program described above, issue the command `lkm calc`. Similarly, to create the editor, issue the command `lkm ed`.

The executable file produced by `lkm` has the same name as the root module. It is produced by compiling the root module's `main.c` file and then linking it with the `.o` files from all the modules on the uselist. The `lkm` command ensures that it is up-to-date, i.e., the result of compiling an up-to-date `main.c` and linking it with up-to-date `.o` files from all the modules on an up-to-date `uselist`.

The simplest way to do this is to recreate from scratch all the system files in the root module and in all the modules imported directly or indirectly. Recomputing from scratch is, of course, very time-inefficient and it is particularly important to avoid unnecessary C compilations. Therefore, SLOTH follows the strategy of MAKE and determines, by examining UNIX timestamps, which files are (possibly) obsolete, and computes only those that are obsolete (or missing). In general, a file is obsolete if, according to the timestamps, it is older than any of the files used to create it. Thus, for example, an `extern.i` file is obsolete if it is older than either the corresponding `var.i` file or the corresponding `define.i` file.

The update process begins with the uselists of the root module. Changes to import lists are relatively rare, so SLOTH first checks that all the relevant uselists are up-to-date. To do this, it examines the timestamps on the root's `import` file and on the `import` files of all the modules on the root's current uselist. Then it does the same for the corresponding uselists. SLOTH is looking for an `import` file that is younger than at least one `uselist` file. If no such files are found, the uselists are all up-to-date; otherwise, some may need recomputing.

Once the root uselist is updated, all the relevant modules are visited to ensure that the `extern.i` and `prog.o` files are correct. In each case the `extern.i` file is recreated if it is missing or older than either the `var.i` or `define.i` file. Next, if the `prog.o` file is present its timestamp is compared with the stamps on the module's uselist, on its `proc.i` file, on its `extern.i` file, on its `body.i` file, and on all the `extern.i` files of all the modules on the uselist. If the `prog.o` is absent or if any of these stamps are more recent, the `prog.c` file is(re)created and compiled, yielding an up-to-date `prog.o` file.

Finally, when the root module's own `.o` file is brought up-to-date, SLOTH decides if the executable needs relinking. It does so if there has been any change to the uselist, or if any of the relevant `prog.o` files were found to be missing or obsolete.

Bottom-up computation

The approach described above avoids unnecessary recompilations but can still be inefficient if implemented naïvely. For example, in determining whether or not a module's `prog.o` file is up-to-date, its timestamp must be compared with the most recent timestamp found on any of the `extern.i` files in modules on the uselist. If this requirement is implemented in a simple-minded way, the more primitive modules will have their timestamps read many times in the course of a single `lkm` operation.

Instead, the current implementation uses a number of bottom-up algorithms in which information, such as the youngest timestamp, is accumulated and passed from the leaves up through the import dag to the root. These algorithms save time because results of the calculations on a given module are available when the modules that import it are processed. These bottom-up algorithms are superior to more naïve, quadratic algorithms that visit each module on the root uselist, and that in turn visit each module on its uselist.

These algorithms are illustrated by pseudo-code for a procedure `upp` that updates uselists and a procedure `cmp` that ensures `prog.o` files are current. Assume that before either procedure is called, an internal representation of the part of the import dag accessible from the root module has been constructed.

```
/* the various components of modules are represented as arrays */

import,uselist : array[module] of list[module];
proc,body,var,define : array[module] of textfile;
prog : array[module] of objectfile;

procedure uup(m:module) : stamp;
/*
 * ensures that the uselists of m and all its descendants are
 * up-to-date;  returns the most recent timestamp found on
 * any of these modules' import file.
 */
var m,k : module;
    s : stamp;
    im : list[module];

begin
    im := import[m];
    s := maxlist (uup(k) : k in im);
    s := max(s,stamp(im));
    if s> stamp(uselist[m]) then
       uselist[m] := Tmerge(im,(uselist[k] : k in im));
    return s;
end;
```

```
procedure cmp(m:module) : stamp;
/*
 * ensures that the prog.o file of m and its descendants are
 * all up-to-date, by recompiling if necessary;  returns the
 * most recent timestamp found on any of these modules'
 * extern.i files, which are updated if necessary.
 */
var s,t,r : stamp;
    k : module;
begin
    s := maxlist(cmp(k) : k in import[m]);
    r := max(stamp(var[m]),stamp(define[m]))
    if stamp(extern[m]) > r then
       extern[m] := mkextern(var[m],define[m]);
    t := max(s,r,stamp(body[m]),stamp(proc[m]));
    if t > stamp(prog[m]) then
       compile(m);
    return max(s,stamp(extern[m]));
end;
```

The code here uses a number of other fairly obvious functions whose declarations are given below:

```
function Tmerge(I:list[module],L:list[list[module]]) : list[module];
/*
 * takes the import list and the imported modules' uselists
 * and merges them to produce the uselist.
 */

procedure compile(m:module);
/*
 * creates m's prog.c file and compiles it,
 * yielding m's prog.o file.
 */

function mkextern(v,d:textfile) : textfile;
/* externalizes the declarations in v and appends d */

function stamp(m:file) : timestamp;
/* returns the timestamp on a file */

function max(m1,m2,m3,...:timestamp) : timestamp;
/* returns the most recent of m1, m2, m3, ... */

function maxlist(L:list[timestamp]) : timestamp;
/*
 * returns the most recent component of L
 * (or -infinity if L is empty).
 */
```

Experience with Sloth

The current version has been in use since 1985. It has been heavily used by one of the authors (Wadge) and some of his graduate students to manage projects involving thousands of lines of code and about 200 different modules. SLOTH has also been used in several graduate and undergraduate courses to manage course projects, some of which were built on the software mentioned above and some of which were smaller, independent projects. On the whole, SLOTH has proved very successful. Indeed, many of these projects, such as the implementation of the Intensional Spreadsheet [11], would have been difficult or impossible to do manually, with only the standard UNIX tools.

During this time a great deal was (re)discovered about how to create and manage reusable C software; inadequacies in SLOTH and corresponding possibilities for improvement were revealed. These experiences are discussed below, as are the improvements suggested by them. Some of these improvements have already been implemented, while others are planned.

The scope for modularity

At first SLOTH was used simply as a substitute for the MAKE/`#include` approach, for single applications divided into a dozen or so modules with very little sharing. SLOTH proved to be a perfectly adequate substitute and somewhat easier to use than the traditional approach. However once SLOTH was available to take care of the bookkeeping, it became clear that even a relatively small project could be easily (and profitably) divided up into literally dozens of logically simple, reusable units.

For one thing, large numbers of test modules were created, sometimes nearly one for every useful module. The files in the test modules are mostly empty, since the test modules import the module to be tested, and have short bodies containing mainly calls to the procedures to be tested. Test modules allow the new software to be exercised without altering it, or even recompiling it. These are not really user applications, but they are complete programs in the sense that `lkm` is used to create an executable. And they are maintained, because of the need to retest modules that have been modified.

The granularity of the useful modules has also been steadily reduced. For example, originally there was one module that implemented, in C, a collection of data types and operations similar to those of LISP; the I/O code was then put into its own module; the implementation, using cells and pointers, was then separated from the others; finally, the garbage collection code was extracted. These modifications hardly constitute earth-shaking advances in modular programming; but they would have been quite impractical with the traditional approach, because of the overhead involved in writing and rewriting ever larger header and makefiles.

Originally, a number of application programs were seen as monolithic pieces of software, e.g., a complete language interpreter, a complete desk calculator or a full-featured spreadsheet. As the project developed, however, the formats of the applications became the subjects for experiments. For example, it often proved helpful to break a single application into several separate executable programs linked through UNIX files or pipes. The software became easier to test, allowing for reusability at the UNIX command level. For example, the input to the calculator might be piped through a parser and then through a compiler into a low-level stack-based interpreter.

On the other hand, there is still a place for the self-contained application whose use requires minimal knowledge of UNIX. Fortunately, with SLOTH, one does not have to choose between the two approaches. Most of the software remains in multipurpose SLOTH modules. The different applications just mentioned took the form of relatively small SLOTH driver modules that simply imported those modules that perform the actual compilation or interpretation. The small driver modules are like test modules; most of their code is in the body, which consists largely of input/output and calls to imported procedures that do the real work.

In this way SLOTH makes it practical to present the same functionality in many different forms, without modifying the software that implements the functionality. In other words, the form of the software can be separated from its content. The many different application programs correspond to different forms; the modules they import hold the content.

Variations in form accounted for some but not all of the many applications developed. Some of these applications corresponded to different functionalities, and implemented different versions of the language. For example, a desk calculator that supports complex arithmetic, and not just real arithmetic, might be needed. Sometimes these variations were easy to achieve using SLOTH, simply by enlarging or reducing the root module's import list. Other variations were not so easily handled, forcing a study of the notion of versions, discussed below. At any rate, it became clear that even a simple project is capable of generating large numbers of distinct but useful application programs that share large amounts of code.

Extensible procedures

SLOTH is basically external to C and therefore imposes no real restrictions on the nature of the C code being maintained—not even information hiding. There is, however, one apparently nonlinguistic constraint that, in the authors' experience, had a major influence on the structure of the projects.

The constraint is that there be no circular dependencies among modules. In other words, if module `A` uses module `B`, then module `B` cannot use module `A`. The reason for this constraint is obvious: if `A` and `B` use each other, then any application that uses one must include the other. The two modules are inseparable and, as far as reusability is concerned, they constitute a single component.

The problem arises with certain kinds of software components that (apparently) cannot be completed without knowing how they are to be used. A typical example is the module that manages a list heap and is called to perform a standard mark-and-sweep algorithm. What exactly should it mark?

This module only knows about the basic structures common to all applications, such as the free list. It cannot know about the data structures internal to modules that use it. These structures, however, may be implemented with lists and must be marked before a collection. The circularity seems unavoidable: the application module must call the heap module to get storage (and initiate a collect), but the garbage module must call the application module routines that mark its own storage.

The problem is not simply an inadequacy of SLOTH: how does one write a reusable, completely generic mark-and-sweep procedure? The solution is to write the code that is, in a sense, dynamically extensible. For example, a garbage collector was implemented to use a *jobjar*, which is an enlargeable set of marking procedures, implemented as an array of function pointers. When a collect is triggered, the generic garbage procedure first marks all the structures it knows about; it then it calls in turn all the procedures in the jobjar and only then begins the sweep.

The extra marking procedures are provided by the application modules and are placed in the jobjar during powerup. For example, if module `T` has a tree, implemented using lists, that will need marking, the programmer writes a procedure—say `treemark`—that marks the tree when called. The programmer then places a call `EnterMarkingProc(treemark)` in the body of the application module; when the call is executed, `treemark` is added to the garbage collector's jobjar.

This situation occurred so many times that a general-purpose extensible function module was written. This technique is hardly new—a very similar approach is used in the implementation of object-oriented languages. However, in practice circular dependencies are completely avoided.

There may, of course, be other forms of circularity that require more sophisticated approaches. It should be noted, however, that the simple technique just described allows *data* circularities at *run time*. For example, if module `A` uses module `B`, then `A`'s data structures can obviously contain pointers into `B`'s data structures. On the other hand, code in `A` can also call `B`-supplied procedures, such as `EnterMarkingProc`, that enter pointers into data structures, such as tables, maintained by `B`.

Discussion and future work

Most work attempting to make the C programming language more modular works at the language level. Modular C [1, 2] introduces a number of C macros to give the impression of import lists and export lists, among other things. Reference [4] does the same thing to create public and private objects. Reference [6] shows how abstract data types can be programmed in C. And, of course, C++ [9, 10] adds classes to C,

thereby allowing for inheritance and improved data encapsulation. Reference [5] shows how C++ can be used to create a Smalltalk-like hierarchy of classes.

However, there seems to be little work on how to arrange modules in such a way that includes are not a headache. Reference [3] addresses the issue, but its solution requires a global include file that knows about all of the modules that can be used. In SLOTH, everything is built automatically through the automatic creation of a uselist.

The SLOTH environment has been useful to the authors and their students. In fact, any tightly-knit group of programmers would find it useful. After all, if one is programming in C, one is forced to address the issues of include files, module initialization and automatic recompilation of required modules. Nevertheless, a large team of programmers, working in several languages, maintaining a number of valid versions, requires much more. Current work deals with expanding the capacities of SLOTH, while keeping the original vision.

Minor extensions allow for the use of compiler options and the specification of libraries other than the standard C library. More elaborate extensions include the addition of versions and allowing a multilingual programming environment. A new language for versions was defined in reference [7]. The set of possible versions of a component or of a complete system forms a lattice. When the link command is asked for a particular version, the most relevant components are selected. For example, when the `French+graphics` version is requested, components only available in one of `French` or `graphics` will be accepted. A new environment, called LEMUR, adds versions to SLOTH.

A generalization of LEMUR, called MARMOSET [8], allows the user to specify the configuration required for a particular programming language. A user can write a meta-makefile to state what must be done for a module in a given language. A module can have several different import files, so users can build systems using several different languages. Actual research deals with questions raised by private workspaces, consistency checks on configured systems and software databases.

References

[1] Stowe Boyd. Modular C. *ACM SIGPLAN Notices*, 18(4):45–54, April 1983.

[2] Stowe Boyd. Free and bound generics: two techniques for abstract data types in Modular C. *ACM SIGPLAN Notices*, 19(3):12–20, March 1984.

[3] J. M. Coggins and G. Bollella. Managing C++ libraries. *ACM SIGPLAN Notices*, 24(6):37–48, June 1989.

[4] Kalyan Dutta. Modular programming in C: an approach and an example. *ACM SIGPLAN Notices*, 20(3):9–15, March 1985.

[5] Keith E. Gorlen. An object-oriented class library for C++ programs. *Software — Practice and Experience*, 17(12):899–922, Dec. 1987.

[6] Giulio Iannello. Programming abstract data types, iterators and generic modules in C. *Software — Practice and Experience*, 20(3):243–260, March 1990.

[7] J. A. Plaice and W. W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 1992. In press.

[8] J. A. Plaice and W. W. Wadge. Reducing the complexity of software configuration. In *1st African Conference on Research in Computer Science*, Yaound, Cameroon, Oct. 1992.

[9] Bjarne Stroustrup. Classes: An abstract data type facility for the C language. *ACM SIGPLAN Notices*, 17(1):42–52, Jan. 1982.

[10] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[11] Du Weichang and W. W. Wadge. An 3-D spreadsheet based on intensional logic. *IEEE Software*, 7(5):78–89, 1990.