An Inconsistency Management System
(M.S. Dissertation)

Harris M. Morgenstern
Columbia University
Department of Computer Science
New York, NY 10027

thesis advisor: Prof. Gail E. Kaiser

March 1987

CUCS-284-87

Abstract

This technical report consists of a M.S. thesis from the CONMAN project. It describes an implementation of 'smarter recompilation' for C. Smarter recompilation is an extension of Tichy's Smart Recompilation algorithm that permits multiple versions of the same symbol within a system provided that the symbol does appear in any link-time interface, that is, the system can be automatically partitioned into a subset that uses the old version and a subset the uses the new version.

CONMAN is a joint project with Siemens Research and Technology Laboratories, Princeton, NJ.

An Inconsistency Management System

Master's Thesis

Harris M. Morgenstern
Columbia University
25 March 1987

Table of Contents

1. Problem and Goal 1
2. Background Work 1
  2.1. Make 1
  2.2. Smart Recompilation 2
  2.3. Inconsistency Management 4
  2.4. Comparison of the Three Algorithms 6
3. Plan 7
4. Description 7
5. Design 9
  5.1. The Compiler 9
  5.2. The Preprocessor 15
  5.3. Cdiff 16
  5.4. Cppcdiff 20
  5.5. Smartmake 21
6. An Example 26
7. Conclusion 29

1. Problem and Goal

A variety of software management systems aid in the development of large scale software systems such as Cedar [08 83] which aids the programmer who works in a distributed environment and Gandalf [09 85] which has such tools as intelligent editors using incremental parsing. None, however, provide a facility for inconsistency management [11 86] among modules. A module is defined as an independent compilation unit which imports and exports resources such as types, procedures, variables and constants. A system is said to be inconsistent if it contains two versions of some logical source component. Such is the case in the C programming language [04 84] [07 78] when two different compilation units include different versions of the same include file. In Ada [01 82], inconsistency occurs when two different versions of an interface to a module (*with* directive) are used in a system. On the outset, inconsistency seems harmful to the extent that it should not be allowed to exist. But considering that for very large systems, correcting the inconsistency may prove very costly, wasting many programming man hours waiting for the system to fully recompile, we realize the need to manage it rather than to avoid it.

2. Background Work

2.1. Make

Make [03 79] was the first system modeling tool. A system model describes how the modules for system should be compiled and linked to produce the final system. The fundamental goal of *Make* is to ensure that all the source units upon which a given module depends on, exist and are up to date. So, if an include file is modified then all the modules which depend on it are recompiled. Dependencies in *Make* may either be linkage dependencies (i.e. object file *foo* depends on *a.o*, *b.o* and *c.o* if *a.o*, *b.o* and *c.o* need to be linked in order to produce *foo*) or compilation dependencies (i.e object file *a.o* depends on *lib1* if *a.c* or some file that *a.c* includes has an include directive that includes *lib1*). *Make* also supports other dependencies relating to the various software tools available (*.o* files depend on *.y* files for YACC). As input, *Make* expects a *Makefile*, which is just a representation of an acyclic dependency graph, showing the dependencies among modules.

Below, is an example of the use of *Make*. If *prog* consists of three object modules linked together, *a.o*, *b.o* and *c.o* where *a.o* and *b.o* depend on *def1* while *c.o* depends on *def2*, then we have the following *Makefile*:

```
prog:   a.o b.o c.o
        cc -o prog a.o b.o c.o
a.o b.o :       def1
```

```
c.o        :        def2
```

The object files are assumed to be dependent on their corresponding source files so entries in the *Makefile* like *a.o: a.c* are unnecessary. If a change occurs to *def1* then *a.o* and *b.o* are recompiled and relinked with *c.o* to produce a new version of *prog*. The dependency of *a.o* and *b.o* on *def1* implies that there is an *#include def1* in each of their corresponding source files.

Make is not capable of handling multiple versions (alternate designs) of modules. This may not be important with small or medium scale software, but with large systems where experimental or exploratory programming is used, the ability to handle multiple versions is crucial.

Make determines whether compilation is necessary by comparing the creation time of an object file with all of its dependent files. If a dependent files was created after the object file was created, then *Make* will remake the object file. Using time stamps as a means of determining whether to recompile is crude, leading to many unnecessary compilations. Consider the case where a comment is added to an include file which is included in many modules.

2.2. Smart Recompilation

The notion of *Smart Recompilation* [12 86] [13 84] was the first attempt to avoid using time stamps as a means of determining whether recompilation of modules is necessary. *Smart Recompilation* is language dependent. The abstract syntax tree or symbol table of the new version of an include file is compared with corresponding abstract syntax tree of the old version to see whether recompilation is warranted.

Tichy [12 86] defines a context as a specification of which external objects a compilation unit may reference (import) and which internal objects it must provide (export). In the C programming language, the include file is an example of a manually prepared context. For the programmer, a context is a convenient means of implanting common declarations into several compilation units without having to retype them into every unit. Library references made during compilation is an example of an automatic context.

A compilation unit depends on a context if it references any declaration defined in that context. The conventional compilation rule used in *Make* recompiles on one of two occurences:

1. The compilation unit changes or

2. A context changes upon which the compilation unit depends

Rule (2) causes unnecessary compilations as illustrated by the following:

defl

```
#define foo 7
#define baz 10
```

prog.c

```
#include defl

main()
{
        char list[baz];   /* foo is not
       ...                   used anywhere */
}
```

prog.c only needs to be recompiled if *baz* were changed. If *foo* were changed, *Make* would recompile *prog.c* while under Tichy's algorithm *prog.c* would be left alone.

Tichy's algorithm effectively minimizes the number of recompilations that result when an include file is modified. His algorithm requires a modified compiler, which not only compiles, but also returns the history attributes for every context used in the compilation. History attributes contain:

1. $DECL_i$: The identifiers declared in M_i;
2. REF_i: The identifiers declared in M_i and transitively referenced in some other context or compilation unit M_j $(1 \le i \le n,\ 0 \le j \le n,\ i \ne j)$

for compilation unit M_0 and contexts $M_1,\ldots,M_n$ which M_0 includes. When a new version of an include file is created the new and old versions are compared. The following tests take place:

1. Analyze M_x^{new} syntactically and semantically. The rules of the programming language apply except that free identifiers are legal. If there are any errors, then the configuration is illegal.
2. Compare the context inclusion directive of M_x^{old} and M_x^{new}. If they are not identical, recompile.
3. Create the following sets:
   - ADD_x: The identifiers declared in M_x^{old}, but not in M_x^{new}.
   - DEL_x: The identifiers declared in M_x^{old}, but not in M_x^{new}.
   - MOD_x: The identifiers declared in both M_x^{old} and M_x^{new} whose declaration differ.
   - $FREE_x$: The identifiers which were not declared, but used in M_x^{new}.
   - $AMREF_x$: The identifiers transitively referenced by declarations in ADD_x and MOD_x
4. If $AMREF_x \cap FREE_x \ne \emptyset$ then an added or modified declaration references a free identifier and recompilation is necessary.
5. If $DEL_x \cap FREE_x \ne \emptyset$ then a deleted identifier is referenced in M_x^{new} and the configuration is illegal.
6. If $MOD_x \cap REF_x \ne \emptyset$ then a local declaration that is referenced elsewhere changed, and recompilation is necessary.

7. If $ADD_x \cap DECL_j \neq \emptyset$ for some j, $0 \leq j \leq n$, $j \neq x$ then M_x^{new} introduced a declaration that conflicts with an external one, and the configuration is illegal.

8. If $DEL_x \cap REF_x \neq \emptyset$ then M_x^{new} is missing a declaration that is referenced externally, and the configuration is illegal.

In the previous example the reference set of *def1* with respect to *prog.c* is *baz*. If *baz* were changed or deleted, compilation would be necessary. This algorithm was successfully implemented for Berkely Pascal.

2.3. Inconsistency Management

Interphase inconsistency is said to in exist when two or more versions of a logical source component are used to build a derived object. Kaiser and Schwanke [11 86] classify inconsistency into five categories:

1. Nominal: two versions of a source unit are used to build the object code for some executable object.
2. Potential: a nominal inconsistency where the interface specifications of the two versions are different whether compatible or contradictory.
3. Derivation: a nominal inconsistency that results in a derived object different from what would have been obtained had either version been used alone.
4. Actual: a derivation inconsistency that results in an inconsistent interface between two objects.
5. Affective: an actual inconsistency that affects the observable behavior of the derived object.

Only inconsistency (5) is harmful because of the possibility that the bad interface caused by the actual inconsistency will be executed at some point. Nevertheless, it is bad form to allow an actual inconsistency to exist in any form and would prove computationally expensive to decipher an affective inconsistency from an actual inconsistency. Therefore it suffices to just prevent an actual inconsistency.

Tichy's algorithm prevents derivation inconsistencies. Kaiser and Schwanke [10 86] present an extension of Tichy's algorithm which avoids actual inconsistencies, while allowing the user the option of eliminating derivation inconsistencies. Using *Smarter Recompilation*, a programmer can test a new version of an include file on a small set of compilation units, without having to recompile every unit which, if not recompiled, would cause a derivation inconsistency.

When an include file changes, the following steps take place according to the algorithm:

1. Use Tichy's algorithm to generate the set of changed symbol definitions, and to determine candidates for recompilation. Present this list of files to the programmer.
2. Some subset of these files is selected by the programmer. Recompile this subset and generate their new symbol tables.

3. For the remaining files selected by Tichy's algorithm, compare the derived and external symbol tables pairwise with the new symbol tables from the previous step to detect interface errors.
4. If an interface error is detected, recompile the candidate file and add it to the list of new symbol tables that must be considered in the previous step.

Besides using the history attributes of Tichy's algorithm, this algorithm requires a new set of symbols called derived symbols. A derived symbol is one which is exported to the linker (either a function or global variable in C). For every derived symbol, we must record the transitive closure of all the resources that it references. We also need to know which resources are imported from the linker (which resources are external to a module in C). Given two modules *a* and *b*, for every symbol *k*, such that *k* is a derived symbol in *a* and is external to *b*, if one of the types that *k* references is different in *b* than *a*, then there exists an interface error and an actual inconsistency.

An example of the use of this algorithm for C is provided below.

lib1.1
```
typedef int T;
```

lib1.2
```
typedef float T;
```

a.c.1
```
#include "lib1"

f()
{
T foo;
...
}
```

b.c.1
```
#include "lib1"

g()
{
/* doesn't use T */
}
```

In the above configuration, if *lib1.2* were given to Kaiser and Schwanke's algorithm to substitute for *lib1.1*, Tichy's algorithm would return *a.c* as a candidate for recompilation. Assuming we chose to recompile, we would be left with a potential inconsistency between *a.c* and *b.c* because the interface specifications of the two versions of *lib1* differ, and the object code produced would not have changed, had *lib1.2* been used with *b.c* instead of *lib1.1*.

Suppose a new versions of *b.c* is used which actually references type *T* as shown below.

b.c.2
```
#include "lib1"

g(baz)
T baz;
{

...
}
```

Tichy's algorithm, would return both *a.c* and *b.c* as candidates for recompilation when *lib1.2* is

substituted for *lib1.1*. Choosing to recompile one candidate and not the other, a derivation inconsistency would exist because we would have a derived object that is different had either version of *LIB1* been used alone. This form of inconsistency is harmless because no resource referencing type *T* is not passed between *a.c* and *b.c* so an actual inconsistency does not exist.

Given this new version of *a.c*, we note that the interface between the two modules is more complex.

a.c.2

```
#include "lib1"

f()
{
T  foo;
...
g(foo);
}
```

Since *g* is external to *a.c*, *g* is added to the set of external symbols of *a.c*. In *b.c* *g* is a derived symbol and all the typedefs, and structures, unions and enum tags which *g* transitively references must be calculated. In this case the only symbol that *g* references is the typedef *T*. Once again, Tichy's algorithm will return *a.c* and *b.c* as candidates for recompilation. This time, however, when the external symbols of *a.c* are intersected with the derived symbols of *b.c* *g* is produced. For every symbol which *g* transitively references, we must determine whether it belongs to the *mod* set of Tichy's algorithm. In this case *T* is contained in Tichy's *mod* set, so an actual inconsistency exists and the system must be made consistent.

2.4. Comparison of the Three Algorithms

Make does not consider a version of a module as an immutable object; instead it assumes that only one version of the module exists and as changes are made to it the old versions are destroyed. Hence, not even nominal inconsistencies are allowed. *Smart Recompilation*, however, considers each version of a module as immutable and will eliminate all forms of inconsistency except nominal and potential (note that if the inconsistency is nothing more than nominal, then it cannot come about in *Smart Recompilation*, since there is no possibility of recompiling if comments are added or if spacing is changed). The previous example, however, shows that derivation inconsistency is often harmless and need not be corrected during the testing or debugging of a software system. Kaiser and Schwanke's inconsistency management algorithm, using derived symbol tables, is able to determine the exact type of inconsistency and whether recompilation is warranted.

3. Plan

I propose to implement Kaiser and Schwanke's algorithm for the C programming language and show that allowing the programmer the option of testing changes to include files on as small a subset of modules as possible can hasten and even aid in the development and testing of a C software system.

The first stage in my research will be to modify the C compiler to return Tichy's history attributes as well as the derived symbol tables and external symbol tables needed to detect actual inconsistencies. The C preprocessor must also be modified to return Tichy's history attributes, but it need not return derived symbols since macros do not cause interface errors.

The next stage in this system's development will be the creation of a program called *cdiff* (context difference) which takes two C include files and executes tests 1 through 5 of Tichy's algorithm. This can be achieved by eliminating the code generation functions from the C compiler, leaving only the parts which build the symbol table and check for semantic and syntactic errors. The output of *cdiff* will be Tichy's change sets: *add*, *mod* and *del*.

In the final stages of development, I will modify the system modeling tool *Make* to handle multiple versions of both compilation units and include files. The new version of *Make* will execute the last 3 steps of Tichy's algorithm to determine candidates for recompilation. The user will be able to choose any subset of these candidates. The modified *Make* will recompile at least these chosen candidates, recompiling more if interface errors are detected.

4. Description

The Inconsistency Management System (IMS) is a successful implementation of Kaiser and Schwanke's *Smarter Recompilation* algorithm [10 86], providing the programmer with the ability to test changes to include files in a software system written in C [04 84] [07 78], on as small a number of modules as he likes, while preventing interface errors. The IMS tool is targeted for large scale software development and testing where multiple versions of modules and include files are kept.

In the directory that IMS is invoked there must exist a *Makefile* [03 79] in the following strict format:

```
linked object code file: module1.o module2.o ... modulen.o
        smartcc flags module1.o module2.o ... modulen.o

module1.o : includefile1 includefile2 ... includefilen
        smartcc flags module1.c
```

```
module2.o : includefile1 includefile2 ... includefilen
        smartcc flags module2.c
...
```

The linked object code file may be created by linking any number of object code files, while each object code file is dependent upon any number of include files. The source code for the modules is expected to exist in the directory where IMS is executing, although include files may exist in any directory. The specific versions of the source units are not mentioned in the *Makefile*. All the usual macro substitution facilities of *Make* may still be used. When IMS is invoked, the user is asked to enter the names of new versions of modules or include files that he would like to substitute for old versions. When a new version of a module is entered into IMS, the module will become a candidate for recompilation. When a new version of an include file is entered, IMS will decide, using Tichy's algorithm [13 84] [12 86], which modules the changes to the include file affects to the extent that recompilation is warranted, and designate those as candidates for recompilation. The user will later be asked to choose the candidates he wishes to recompile. The system will compile at least the modules that the user chooses; more will be recompiled if an interface error is detected.

When the system is initially compiled, IMS will create a subdirectory in the *Makefile's* directory, called *makedb* (standing for *Make data base*) which stores sets of files which contain information necessary to perform Kaiser and Schwanke's algorithm and a *.versions* file corresponding to every compiled module. Each *.versions* file contains a list of the versions of each logical source unit used in the creation of the object code. Every time a module is recompiled using this tool, the module's corresponding *.versions* file is updated. If a module's *.versions* file is missing (which is the initial state), then the module will be recompiled and a new *.versions* file is created which, by default, will list version 1 as the current version to be used for every include file that the module is dependent on. If a module's corresponding object file is missing, the module will automatically be recompiled using, by default, the versions listed in the current *.versions* file.

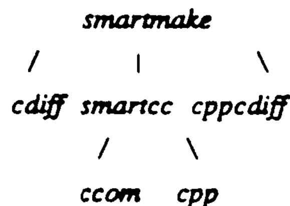
IMS also provides the user with the ability to eliminate inconsistency. If the user does not specify any new versions of source units to exchange with old versions, he will be asked to enter the names of versions of source files to be substituted across the system. So, any inconsistency with respect to the include file the user specifies will be eliminated.

5. Design

The system is composed of six programs: *smartcc, ccom, cpp, cdiff, cppcdiff* and *smartmake*. *smartcc* is a modified version of the C compiler which not only compiles, but also returns Tichy's history attributes (*ref* and *decl*) and Kaiser and Schwanke's derived symbols. *smartcc* serves as a front end of the compiler which calls the modified C preprocessor, *cpp*, the modified compiler, *ccom*, and the loader. The only changes to *smartcc* were to call the modified versions of *cpp* and *ccom* instead on the ones in */lib* on UNIX [06 84].

cdiff compares two preprocessed include files, returning the change sets (*del, mod* and *add*). *cdiff* also executes tests 4 and 5 of Tichy's algorithm, returning an error code of 1 if either is failed or if there were any syntactic or semantic errors in the new version of the include file. The include files are preprocessed by *cppcdiff* which takes as input the name of a module and the name of an include file, returning as its standard output, the preprocessed include file. Macros and their text as well as a list of include directives within the include file are also returned for later comparison.

smartmake is a modified version of the software configuration tool *Make* which handles multiple versions of source files. *smartmake* calls all the aforementioned programs, determining the change sets for macros, performing tests 6, 7 and 8 of Tichy's algorithm and checking for interface errors. The calling relationship between all of the programs is shown below.



5.1. The Compiler

The modified Portable C Compiler (PCC) [05 78] [02 86] assumes the existence of a subdirectory called *makedb* within which it will store files containing the history attributes needed for Tichy's algorithm and the external and derived symbols needed to discover interface errors. When compilation unit *foo.c* is compiled successfully, four files are created: *foo.c.sym*, *foo.c.dersym*, *foo.c.refsym* and *foo.c.macsym*. *foo.c.sym* contains *foo.c's* history attributes for all symbols other than macros. *.sym* files have the following format:

```
name1    file declared in 0/1 t/o
name2    file declared in 0/1 t/o
...
```

A 1 appears in the third column if the corresponding symbol is transitively referenced in a logical source unit other than the one in which it is declared. The fourth column, *t/o*, exists because structure, union and enum tags may have the same names as other symbols such as typedefs and variables, so an extra field (*t* for tag, *o* for other) is used to distinguish the two types.

foo.c.macsym contains Tichy's history attributes for macros. The format is the same as that for the *.sym* files except that the *t/o* field is absent.

foo.c.dersym contains Kaiser and Schwanke's derived symbols along with all of the typedefs and structure, union and enum tags each derived symbol transitively references. *.dersym* files have the following format:

```
derived symbol1  t/o_name1 t/o_name2 ...
derived symbol2  t/o_name1 t/o_name2 ...
...
```

Symbols to the right of the derived symbol are those that the derived symbol transitively references. Once again the *t/o* field is needed.

foo.c.refsym contains a listing of the symbols which are external to the object code of *foo.o* and are imported from the linker. This includes all functions which were not defined in the module. No information about symbols that they transitively reference is kept.

Although compiled as one program, the Portable C Compiler is actually a two pass compiler where the first pass translates C into intermediate code for the second pass to translate into assembly code. The first pass does lexical analysis, parsing, symbol table maintenance, construction of trees for expressions and declarations, initialization and some machine dependent activities. The second pass and the machine dependent parts of the first pass will not be discussed further, since all the modifications made to the compiler were to the portable parts.

Twenty two source files make up PCC. Modifications occured to just four of them: *scan.c*, *cgram.y*, *pftn.c* and *pass1.h*. *scan.c* does lexical analysis, initialization and control for the first pass. The lexical analysis routine, *yylex*, is repeatedly called by the parser and returns a token number representing the type of token found such as names, constants, operators and keywords. *yylex* keeps track of the current file name and line number, resetting this information as a result of preprocessor control lines, using the function *lxtitle*. *yylex* calls the routine *lxcom* to skip comments, although this routine is really obsolete because the C preprocessor eliminates all comments. *yylex* also deals with various constant forms (i.e. octal, hex, etc.) as well as character strings. The routine *mainp1* controls first pass of the compilation process and initializes the symbol table and the *dimtab* array which is used for storing the sizes of data objects, the dimensions of arrays and for locating members of structures, unions and enums.

cgram.y contains the grammar for the C compiler, while the terminal definitions are defined in *pcclocal.h*. When processed with the parser generator YACC [05 75], an LALR parser for C is produced. Some syntax tree construction routines are also included in this module.

pftn.c contains all the symbol table maintenance routines, most notably, the routines *defid*, *lookup* and *dclstruct*. *defid* takes as parameters a syntax tree node, constructed when parsing a declaration, and a class (i.e. *extern*, *static* etc.) and enters the declaration into the symbol table after checking whether the symbol was already declared (which may signify a semantic error). *lookup* takes a symbol name and an integer constant, indicating whether the symbol is a structure, union or emum tag, a member of a structure or union, or neither and returns a symbol table index to the entry. *dclstruct* is called by the parser just after the full definition of a structure, union or enum is seen. *dclstruct* takes a stack offset, containing various information about the structure such as the symbol table indices of the members and the tag, and stores the information in the *dimtab* array. *dclstruct* also computes and checks the sizes of structures and union members for legality.

pass1.h is an include file which contains declarations of various macros and external variables, but most notably the declaration of the symbol table structure, *symtab*. The actual symbol table, *stab* (declared in *xdefs.c*) is an array of *SYMTSZ* elements of structure type *symtab*. Three symbol tables are actually stored in *stab* because structure, union and enum tags, and structure and union members may have the same names as other symbols; thus *lookup* in *pftn.c* must know ahead of time the class of the symbol to look for. Since the size of the symbol table is known at compile time, it is possible to use such structures as bit vectors to represent sets of elements such as Tichy's change sets. Bit vectors are used throughout the design, both in the compiler and *cdiff*. The unmodified symbol table is shown below:

```
struct  symtab {
#ifndef FLEXNAMES
        char    sname[NCHNAM];
#else
        char    *sname;
#endif
        struct  symtab *snext;  /* link to other symbols in the
                                same scope */
        TWORD   stype;          /* type word */
        char    sclass;         /* storage class */
        char    slevel;         /* scope level */
        char    sflags;         /* flags, see below */
        int     offset;         /* offset or value */
        short   dimoff;         /* offset into the dimension
                                table */
        short   sizoff;         /* offset into the size table */
        int     suse;           /* line number of last use of
                                the variable */

};
```

stype indicates the type of the symbol (i.e. *int*, *char* etc.) while some bits may be additionally set to indicate that the symbol is a function, pointer or array. *class* indicates the class of the symbol (i.e. *extern*, *static* etc.). Symbols of global scope have 0 in their *slevel* fields, 1 for parameters and 2 or greater for each level of nesting inside of functions. *dimoff* provides an index to the *dimtab* array to find the array dimensions of a symbol. *sizoff* provides an index to the *dimtab* array to locate structure, union and enum members in *stab* and also to determine the size of the identifier. *offset* gives the offset of structure or enum members relative to the beginning of the structure or enum.

The first stage in the construction of IMS was to modify PCC to return Tichy's history attributes. For a given module, Tichy's history attributes describe what symbols were declared, what contexts they were declared in, and whether they were referenced in another context. To aid in gathering this information, two fields were added to the symbol table structure: *char *decl* and *int ref*. *decl* stores the name of the file in which the symbol was declared and ref is set to 1 if the symbol was referenced in another context.

The first problem that had to be tackled was how to determine where each symbol was declared. The text of all the include files is substituted for their corresponding *#include* directive while the name of the included file, left as a result of preprocessing, is written to the preprocessor's standard output directly above the text of the included file. The function *lxtitle* in *scan.c* is called to set the character pointer *ftitle* to point to the current file name during lexical analysis. *ftitle* is used by the compiler to provide the location of errors to the user. The contents of *ftitle* are copied to the *decl* field of the symbol table when a symbol is declared in the function *defid*.

Determining whether symbols are referenced in a context other than the one in which they are declared is a more difficult process. For example, consider the following configuration:

defl

```
struct foo {
        int a;
        int b;
};
```

prog.c

```
include "defl"

main()
{
        struct foo {
            char a;
        };

        ...
}
```

Structure *foo* is declared in include file *defl* and redeclared in function *main*, a perfectly legal program since the second declaration occurs at a different scope level than the first. If the lexical scanner were to blindly include *foo* in Tichy's *ref* set as soon as it is seen in another context, then

foo in *defl* would be considered as *ref* although it certainly is not referenced. The solution to this problem is to actually see how a symbol, declared in another context, is used before marking it as *ref*. This strategy involves modifications to the parser *cgram.y* as well as to the lexical scanner *yylex*. Semantic actions were added to productions in the YACC grammar which used structure, union and enum tags to declare identifiers. For example, in the production *enum_dcl --> ENUM NAME*, the *decl* field of the symbol table entry whose index is returned by *NAME* ($2 in YACC) is compared with *ftitle*; if the two strings differ, the *ref* field of the symbol table entry is set to 1. Similar actions were added for structure and union tags in the production *struct_dcl --> STRUCT NAME*. References to all variables are handled in the production *term --> NAME*, by using the symbol table index returned from *NAME* and comparing the symbol table's *decl* field with *ftitle*. Functions are handled similarly in the production *function_idn --> NAME LP*.

The lexical scanning routine, *yylex*, determines whether a symbol is a typedef or some other identifier, using the function *lookup* in *pftn.c*; therefore, determining whether typedefs are referenced in another context occurs in the lexical analyzer. PCC does not allow typedefs to be redefined within other scopes, allowing *yylex* to blindly mark a typedef's *ref* field as soon as one appears in a context, other than its declaration context. The example below is semantically incorrect, according to PCC.

defl

```
        typedef foo int;
```

prog.c

```
        #include "defl"

        main()
        {
                typedef foo int;

                ...
        }
```

Once compilation has completed, the function *test_sym*, found in *scan.c* is called. *test_sym* handles the generation of Tichy's history attributes for all symbols other than macro's, and the generation of derived and external symbols (to be discussed later) needed for interface error detection. *test_sym* calls the functions *transitive_closure* for all symbols whose *ref* field is set in their symbol table, setting the *ref* field of all the symbols that the given referenced symbol transitively depends on. For structure and union tags, as well as identifiers which are structures or unions, recursive calls are made on all of their members. As mentioned earlier, the *sizoff* field in the symbol table, provides access to a structure, union or enum's tag entry in the symbol table. If *n* is stored in a structure/union/enum's *sizoff* field, then *dimtab[n + 3]* is the symbol table entry for its corresponding tag or -1 if the symbol has no tag associated with it. Note that for a structure/union/enum tag, *dimtab[n + 3]* is the symbol table index for itself.

Locating the members of a structure or union is more difficult than described in [05 78]. According to [05 78], *dimtab[n]*, *dimtab[n + 1]* and *dimtab[n + 2]* store size and alignment information of the structure, *dimtab[n + 3]* stores the symbol table index of the tag, or -1 if none exists and *dimtab[n + 4]*, *dimtab[n + 5]* ...*-1* store the symbol table indices of the structure, where *n* is stored in the *sizoff* field in the structure's symbol table entry. However, consider the following declaration where one structure is declared within another.

```
struct s1 {
        int e;
        struct s2 {
                int a;
                int b;
        } f;
        int g;
};
```

If the *sizoff* entry in the symbol table for structure tag *s1* or some identifier which references *s1* is *n*, then *dimtab[n + 3]* stores the symbol table index for *s1*, but *dimtab[n + 7]* stores the symbol table index for *s2* followed by the symbol table indices of *a* and *b*, -1 and the symbol table indices of *e f* and *g*. Instead of using the *sizoff* field to reference structure members, another field, called *mem_ref* has been added to the symbol table structure for this purpose. The function *dclstruct* in module *pftn.c* has been modified to set the global variable *mem_index* to the *dimtab* index where the symbol table index of the first member of some structure or union is found. When a symbol is defined in the function *defid*, its *mem_ref* symbol table field is set to *mem_index*. Because structure, union and enum tags are defined before their members are seen, their *mem_ref* fields are set in the function *dclstruct*.

Typedefs must also be handled in an unusual way. Given the declaration: *typedef type1 int* then the two declarations of *foo*: *int foo* and *type1 foo* are structurally equivalent. The symbol table provides no indication that the first declaration of *foo* is any different than the second. In order to determine whether a given symbol references a typedef, the field *typedef_ref* has been added to the symbol table structure. Whenever a typedef is scanned by the lexical scanner, *yylex*, the global variable *typedef_store* is set to the symbol table index of the scanned typedef. The function *defid* was modified to set the *typedef_ref* field of a symbol's symbol table entry to the current value of *typedef_store*. *typedef_store* is initialized to -1 in all productions of *data_def* for the declaration of global data and in all productions of *declaration* for the declaration of function parameters.

After calculating the transitive closure of all symbols marked *ref*, *test_sym* determines the sets of derived and external symbols. Symbols that have no class specification, *int foo* for example, are considered to be both derived and external since the linker will only allocate storage once if another declaration of *int foo* if found in another module. Symbols that have the *extern*

specification and functions which have no bodies (just declarations) are written into the external symbols file. The field *ext* has been added to the symbol table to distinguish between variables which are explicitly defined to be *extern*, where no storage is allocated, and those variables with no class specifications (the *class* field of the symbol table stores *EXTERN* for both types). The function *defid* assigns the *class* field of the symbol being defined to *curclass*, a global variable which stores the last class seen. If *curclass* equals *EXTERN* then the *ext* field of the symbol being defined is set to 1 and the symbol is external; if *curclass* equals 0 then the *ext* field is not set and the symbol is both external and derived. Functions with bodies are derived only; functions without bodies are external only.

test_sym finally calls the function *param_ref* on all the derived symbols which are not functions. *param_ref* takes a pointer to the symbol table for a derived symbol and determines which symbols (structure, union and enum tags and typedefs) it transitively depends on, marking the entries of bit vector *paramref_list* corresponding to the indices of symbols in *stab* that the derived symbol depends on. *param_ref* recursively calls itself on the members of structures and unions.

Since there is no way of associating a function's parameters with the function once compilation has completed, it is necessary to analyze the symbols a function depends on, just after the function header has been parsed. The function *dclargs* in *pftn.c* declares parameters whose symbol table indices were pushed onto *paramstk*. To calculate all the symbols that the last function references, *dclargs* calls *param_ref* with every parameter as well with the function itself. The derived symbol and the symbols it depends on are written to the file *.dersym* which will later be changed to *module-name.dersym* if there were no compilation errors (i.e. if a new object file is produced); otherwise it will be deleted.

5.2. The Preprocessor

Before C code is actually compiled, it first undergoes a preprocessing step which expands macros and executes conditional compilation directives. The C Preprocessor (CPP) has a macro symbol table that is an array of structures, much like the symbol table for the C compiler. The fields *char *decl* and *int ref* were added to the symbol table structure to indicate where a given macro was declared and whether it was referenced in another context. The function *lookup* in *cpp.c* is used to check the existence of some macro name in the symbol table and also to enter macro names. *lookup* is passed two parameters: *char *namep* and *int enterf*. If *enterf* equals 1 *lookup* will enter *namep* into the symbol table; otherwise *lookup* will just return a pointer to the symbol's symbol table structure. *lookup* has been modified so that when *enterf* equals 1 the name of the current context, found in the array *fnames* indexed by *ifno*, is copied to the *decl* field of the

corresponding symbol table entry. When *enterf* equals 0 the name of the current context is compared with the name found in the symbol table's *decl* field; if the names are different then the symbol's *ref* field in the symbol table is set to 1.

Near the end of preprocessing, in the function *refill*, the symbol table is scanned and written to the file *.macsym*. If no compilation errors are detected when *ccom* is run, this file will later be renamed to *module-name.macsym*, otherwise it will be deleted.

Note that no transitive closure routine is needed for macros because macros are not expanded within one another. Consider the following example:

def1

```
#define mac1 33 * 33
#define mac2 mac1 + 44
```

def2

```
#include "def1"

#define mac3 mac2 * 2
```

prog.c

```
#include "def2"

int foo[mac3];
```

mac1 is not expanded within *mac2* and *mac2* is not expanded within *mac3* until *mac3* is actually used in *prog.c*. *lookup* is called three times with *mac3*, *mac2* and *mac1* and each symbol's *ref* field in their symbol table is marked.

5.3. Cdiff

cdiff is a program whose purpose is to take two preprocessed versions of an include file and determine which symbols were deleted, modified or added going from one version to the next. Tests 4 ($AMREF \cap FREE \neq \emptyset$) and 5 ($FREE \cap DEL \neq \emptyset$) of Tichy's algorithm must also be computed here. *cdiff* was built out of the declaration parser of PCC. The symbol tables of both version of the include file are constructed and then compared.

The first step in the construction of *cdiff* was the removal of all functions from the compiler which did not pertain to the parsing of C declarations, the checking of the semantic correctness of the constructs and the maintenance of the symbol table. All of the functions from *scan.c* and some of the initialization functions in *common.c* were included in the file *cdiffscan.c*. The YACC grammar for declarations, all the functions in *cgram.y*, the symbol table maintenance functions in *pftn.c* and the tree construction functions which dealt with declarations were stored in *cdiffcgram.y*. Below is a list of the most significant semantic functions included in *cdiff*, all of which reside in *cdiffcgram.y*.

- *bstruct (idn, q) int idn; NODE *q;*: This is called when a structure is being defined. *idn* stores the symbol table index of the tag or -1 if there is no tag (i.e. *struct tag {...}* vs. *struct {...}*). *bstruct* calls *defid* to define the tag if one exists. *bstruct* pushs some semantic information onto *paramstk* to be accessed when *dclstruct* is called.

- *dclstruct (oparam) int oparam;*: Stores the structure, union or enum member symbol table indices in the *dimtab* array. *mem_index* is set to the *dimtab* index where the members can be found. If a tag exists, its *free* field is set to 0. If a tag's *free* field was 2, then an error is flagged -- an undeclared tag was used to declared some non-pointer.

- *defid (q, class) NODE *q; int class;*: declares a symbol represented by tree node pointer *q* in the symbol table after performing some extensive checking if the symbol was previously declared. Code handling automatic and register variables was eliminated. Modified to handle free structure, union and enum tags.

- *falloc (p, w, new, pty) struct symtab *p; NODE *pty;*: needed to perform semantic tests on bit fields of structures (such as checking for negative or 0 bit fields) and to compute bit offsets for structures.

- *fixclass (class, type) int class; TWORD type;*: Checks for illegal storage classes. *fixclass* has been modified to allow SNULL storage classes for symbols declared without a storage class (i.e. to distinguish between *int foo* and *extern int foo*).

- *fixtype (p, class) NODE *p; int class;*: Checks a declaration node for various semantic errors like function declarations within structures.

- *mknonuniq (idindex) int *idindex;*: Locates a symbol table entry for a nonunique structure member. Called by *defid*.

- *nidcl (p) NODE *p;*: Calls *defid* to define an unininitialized declaration. Some code generation lines have been eliminated.

- *noinit()*: Used to return EXTERN as the default class for global identifiers declared without a class specification. Now returns SNULL to distinguish between *extern int foo* and *int foo*.

- *oalloc (p, poff) struct symtab *p; int *poff;*: Updates the offset of a member of a structure whose symbol table is pointed to by *p*.

- *NODE *rstruct (idn, soru) int idn, soru;*: Calls *defid* to define some structure, union or enum tag with no definition (*struct tagname foo*, for example). *defid* checks whether the symbol was previously declared and, if it wasn't, *defid* enters the symbol in the symbol table. Allowing tags to be used before their declaration enables programmers to define structures and unions with cycles (structures that have pointers to each other). *rstruct* was modified to set the *free* field of the symbol table of some tag to 1, if the tag has not been previously defined.

- *talign(ty, s) unsigned ty; int s;*: Computes the alignment of an object of type *ty* and *sizoff* index *s*. This is needed for semantic tests.

- *tsize(ty, d, s) TWORD ty; int d, s;*: Computes the size of a type *ty* with dimension offset *d* and sizoff offset *s*. Needed for semantic tests. Modified so that it does not flag undeclared structure, union or enum tags.

- *NODE *tymerge (typ, idp) NODE *typ, *idp;*: Merges two tree nodes, one some identifier to be declared, and the other a type, into one.

- *TWORD types (t1, t2, t3) TWORD t1, t2, t3;*: Returns a basic type from three or less types.

- *tyreduce(p) NODE *p;*: Stores away dimensions of identifiers and checks for semantic errors like 0 dimensioned arrays.

In the YACC grammar for the C compiler, all of the productions which dealt with C statements and most of the productions for expressions were removed. The four basic numeric operations (+, -, *, /) as well as *a:b?c*, (if *a* then *b* else *c*) were allowed for integers in order to compute array indices and bit fields.

The most difficult part in the construction of *cdiff* was allowing free (undeclared) identifiers (typedefs and structure, union and enum tags) to be used without issuing error messages. When a free identifier is seen, *cdiff* declares it in the symbol table, setting the *free* field in the symbol table to 1. Recall that the lexical scanner, *yylex*, distinguishes between typedefs and other symbols, using *lookup*. If the identifier is indeed a typedef, *TYPE* is returned; otherwise *NAME* is returned. To allow for free typedefs, the production *type --> NAME* has been added to the grammar. When this production is used in parsing, the free identifier is declared as *typedef int*.

Handling free structure, union and enum tags is more complicated because tags may be used before being declared in the declaration of pointers to structures. When a structure tag without a declaration is used as in *struct s1 *foo* the parser calls the function *rstruct* which calls *defid* to declare the tag. If the tag was not previously declared, the *free* field in the tag's symbol table is set to 1 in *rstruct*. In the following declaration *struct s1 foo*, assume that *s1* has not previously been declared. At the conclusion of *nidcl*, the function *commdcl* calls *tsize* which flags *s1* as being an undeclared (zero sized) structure. *commdcl* has been eliminated in *cdiff* so this call is never made. However, it was necessary to eliminate the error messages in *tsize* because *tsize* is called in *dclstruct* on structure and union members. If a member of a structure or union references a free structure, union or enum tag then *tsize* would flag the error had this message not been eliminated. *defid* has been modified to set the *free* field of the symbol table of a *free* tag to 2 (from 1) to indicate that the free tag was used in the declaration of some non-pointer. It is a semantic error for a tag to be used to declare a non-pointer before its definition appears. *dclstruct* is called after the definition of a structure, union or enum has been parsed. If the *free* field in a tag's symbol table was previously set to 2, an error message is issued, indicating that before the tag was declared, it was used to declare some non-pointer. The free field in the tag's symbol table structure is set to 0 to reflect the fact that the tag is no longer free (if it was).

Like the modified C compiler, *cdiff* uses the global variable *typedef_store* to store the symbol table index of the last typedef seen. *typedef_store* will be stored away in the identifier's *typedef_ref* field in the symbol table, once the identifier has been defined. This is important because the comparisons between the old version and new version of the include file check for

name equivalence and not structural equivalence. So, the two declarations: *type1 a* and *int a* are considered different even if *type1* is a typedef of *int*.

At the end of parsing each include file, we are left with two symbol tables, *stab* and *oldstab*, and two dimension tables, *dimtab* and *olddimtab*, both corresponding to the new and old versions of the include files, respectively. If any syntactic or semantic errors were found in parsing the new version of the include file, no further processing is done and *cdiff* returns an error code of 1. Otherwise, the function *difference* which computes Tichy's change sets and the set *amref*, is called, followed by a call to the function *cdiff* which executes test 4 and 5 of Tichy's algorithm. If one of these tests fails, *cdiff* returns an error code of 1, along with the files which contain the change sets.

The three change sets, *del*, *mod* and *add*, and *amref* are represented in bit vectors *delv*, *modv*, *addv* and *amrefv* respectively. For every symbol in *oldstab*, *difference* calls the routine *newlookup* which returns the symbol table index of the corresponding symbol in *stab* or -1 if no symbol is found. *newlookup* uses hashing to look up members especially quickly. If -1 is returned by *newlookup*, then *delv* is marked, indicating that the symbol was deleted. If a number between 0 and *SYMTSZ* is returned (the symbol table index), then a corresponding symbol has been found in the new version of the include file, and the new symbol's *mark* field which was added to the symbol table structure is set to 1. The two symbol table indices are passed to the function *diff* to determine whether the two identifiers are the same or different. *diff* compares the type, class and array dimensions of the two versions of the symbol. If the symbol is a structure or a union (tag or otherwise), then *diff* calls *struct_diff* to check whether the two structures are the same. *struct_diff* compares the bit offsets of each member of the old and new version of the structure and calls *diff* on all the members of the structure or union (*struct_diff* and *diff* are mutually recursive). The global variable, *testoffset* is used to keep a running count of the size of bit offsets and their types (additional bits are set to record the type i.e. *int :3 vs. char :3*). The field *testoffset* was added to the symbol table structure to make note of these gaps between structure members.

If the symbol is an enum, *diff* calls *enum_diff* to compare the members; the *offset* field in the symbol table which stores the value of each enum member is compared.

If two versions of the same symbol are different, the function *mod* is called, marking the *modv* vector for corresponding symbol table index. *mod* calls *amref*, marking the *amrefv* bit vector for all symbols which the changed symbol transitively references. *amref* recursively calls itself on the members of structures and unions. At the end of *difference*, *stab* is scanned for symbols whose *mark* field was not set; these symbols have been added to the include file. The function *add* is called on each of them, marking the *addv* bit vector and calling *amref*.

When *difference* finishes, *cdiff* is called to perform tests 4 and 5 of Tichy's algorithm. This involves doing a scan of the entire new symbol table for symbols which are free. When a free symbol's *amrefv* entry is 1, test 4 has failed; when a free symbol's *delv* entry is 1, test 5 has failed.

outfiles, the last function called, creates the files, *del*, *mod* and *add* containing lists of symbols which were deleted, added and modified respectively, going from the old to the new version of the include file. All change files have the following format:

```
name1 t/o
name2 t/o
...
```

5.4. Cppcdiff

cppcdiff is a modified version of the C preprocessor whose purpose is to preprocess include files to be used with *cdiff*. *cppcdiff* takes as arguments, the names of a module and an include file which is included in the module, returning as its standard output the preprocessed include file. The include file must be preprocessed in the context of a module that includes it, because macros which the include file uses may be previously declared in the module or in some other include file, included in the module. *cppcdiff* also creates two files: *includes* which lists the names of the include files included within the specified include file in the order they were included, and *macsym* which lists the macro names, their parameters and text in the following format:

```
name
text
n param1 param2 ... paramn
...
```

The function *doincl* has been modified to print out the names of include files to the file *includes* when the preprocessor is in the midst of processing the specified include file (the name of which is pointed to by the global character pointer *context*). The current file being preprocessed, stored in *fnames[ifno]*, is compared with *context* to determine whether the names should be printed out.

The function *dodef* which handles the preprocessing of macros, was modified to print out the macro names with their associated parameters and text to the file *macsym*. At the end of the *for* loop which reads in the macro body, the character pointer *psave* is left pointing to the macro text. Unfortunately, within the *for* loop, there exists code, which in dealing with macro parameters, leaves control characters in their place in the text pointed to by *psave*. Therefore before the parameter handling code is executed, the macro text body is copied to the storage area

pointed to by the character pointer *save*. The formal parameters for macros are stored in the array of character pointers, *formal*, while the count of parameters is kept in the variable *param*.

The function *dump* which writes the preprocessed text to the standard output, was modified to just write out the text found in the include file specified as an argument to *cppcdiff*. Before writing out text to the standard output, *dump* checks whether *context* (the specified include file) is the same as *fnames[ifno]* (the current include file); if they are the same, then the character is written to the standard output; otherwise the character pointer for the text is just incremented without writing anything.

5.5. Smartmake

The *Make* software configuration tool is composed of eight source files, including *doname.c* which contains the basic algorithm in the function *doname*. *doname* does a depth first search on the acyclic dependency graph. A given vertex within the dependency graph will be remade if the creation times of at least one of his sons is later than his own. Vertices in the dependency graph are represented by structures called *nameblocks* which store such information as the name of the file and the file's creation time. Each *nameblock* has a pointer to a structure called a *lineblock* and a pointer to a list of *nameblocks*. Each *lineblock* has a pointer to other *lineblocks*, a pointer to a structure called a *shblock* and a pointer to a structure called a *depblock*. *shblocks* store the shell command lines which are executed to remake a file. Each *depblock* has a pointer to a list of *nameblocks* which the *nameblock* one level higher depends on and a pointer to a list of *depblocks* pointing to other *nameblock* lists which the *nameblock* one level up also depends on. These structures are shown below in their unmodified form.

```
struct nameblock
        {
        struct nameblock *nxtnameblock;
        char *namep;
        char *alias;
        struct lineblock *linep;
        int done:3;
        int septype:3;
        TIMETYPE modtime;
        };

struct lineblock
        {
        struct lineblock *nxtlineblock;
        struct depblock *depp;
        struct shblock *shp;
        };

struct depblock
        {
        struct depblock *nxtdepblock;
```

```
        struct nameblock *depname;
        };

struct shblock
        {
        struct shblock *nxtshblock;
        char *shbp;
        };
```

The structure of the dependency graph was simplified to a three level tree to facilitate using multiple versions of source files. The top level file contains executable object code; the second level contains object files which are linked to produce the object code on the top level; the third level contains those source files which are needed to produce each object file on the second level. This structure simplifies the locating of source files which an object file depends on for the creation of *.versions* files. Recall that a *.versions* file for an object file contains a list of the versions of source units used in the creation of the current object file. *.versions* files are stored in the *makedb* subdirectory which is created (if it doesn't exist already) upon entry to *doname*.

Besides modifications to *doname*, three additional modules were added: *diff.c*, *dersym.c* and *unify.c*. *diff.c* contains functions dedicated to executing Tichy's algorithm and handling multiple versions of source units. *dersym.c* contains functions dedicated to the checking of interface errors between a pair of modules. *unify.c* contains functions dedicated to the elimination of inconsistency for a give include file.

doname requests that the user enter the full names (*name.number*) of new versions of source units that are to be substituted for old ones. The function *read_changes* in *diff.c* reads in a list of names of source files that the user specified into a linked list of structures of *info* type, pointed to by *changes*. The *info* structure stores the full name of the file (i.e. *foo.c.2*), the name of the source unit (i.e. *foo.c*) and the version number (2). For every module whose object code and *.versions* file exists, the function *diff* is called. *diff* does initialization on the various symbol tables used and calls *read_versions* to read in the *.versions* file containing the versions of source units used to create the current object file. *version_list* and *info* structures are used to store version information. Each *version_list* stores the name of a module, a pointer to a list of *version_list* structures and a pointer to a list of *info* structures. The structures are shown below:

```
/* store the name and version number of a source unit */
struct info{
  int number;
  char *name;
  char *fullname;
  struct info *next;
};

/* stores the names of the versions of files needed to create a
   compilation unit */
```

```
struct version_list {
  struct info *versions;
  char *compilation_unit;
  struct version_list *next;
};
```

The *versions* pointer is set to the version list corresponding to the current module. *diff* finally calls *intersect* which reads in Tichy's history attributes for both macros and other symbols into their corresponding symbol tables. *intersect* compares the *info* list pointed to by *versions* and the list pointed to by *changes*. If a different version of the module is used, *intersect* returns 1, signifying that the module should be considered a candidate for recompilation. If a different version of an include file is used within the module, then the function *compare* determines whether recompilation is warranted. *compare* first checks whether the new version of the include file previously failed one of the first five of Tichy's tests; the names of include files which failed one of the first five of Tichy's tests are stored on a linked list of structure *compile_store* to avoid repeated testing.

At this point, the routine *search_list* is called to determine whether the change sets (both macro and non-macro) have already been computed and stored away, so that *compare* can skip preprocessing the two include files and running *cdiff* to doing tests 6 ($MOD \cap REF \neq \varnothing$), 7 ($ADD \cap DECL \neq \varnothing$), and 8 ($DEL \cap REF \neq \varnothing$) of Tichy's algorithm. If the change sets were not previously computed, *compare* uses the routine *setup* to copy the current versions of source units used to create the object code to their unnumbered counterparts (i.e. *lib.1* --> *lib*). *compare* then runs *cppcdiff* on the module and include file. Recall that *cppcdiff* not only provides a preprocessed version of the include file, but also provides a listing of its macros and the include directives found within the include file. The include file names are copied to a linked list; the macros are copied to a macro symbol table called *mactable1*. The new version of the include file is copied to its unnumbered counterpart and *cppcdiff* is run a second time. The names of the include files, included in the include file, are copied to another linked list and the macros are read into *mactable2*. If the include sequences of the two versions of the include file differ (test 2 of Tichy's algorithm) then the new include file is added to *compile_list* and the module becomes a candidate for recompilation.

compare calls *cdiff* on the two preprocessed include files. If the return status is nonzero, then the new include file failed one of tests 1, 3, 4, or 5 of Tichy's algorithm and is placed on *compile_list*. However, instead of immediately returning, *compare* reads in the change sets if they exist (if tests 4 or 5 were failed, the change sets were produced). The change sets are recorded because they may later be needed to test for interface errors. Note that it was not deemed important enough to run *cdiff* on include files whose include directives change, just to get the change sets for interface error detection; the cost of running *cdiff* is too high to warrant its use

in this case. *mac_compare* performs tests 6, 7 and 8 of Tichy's algorithm for macros and *compare2* performs the same tests on all other symbols. Before *mac_compare* performs these tests, it first calls *mac_compare2* to calculate the change sets. *mac_compare2* checks main memory to see whether the change sets have already been computed, calling *macro_search_list* which returns a structure of type *macro_context_changes* containing pointers to the three change lists. If the change sets are not found, *mac_compare2* compares the macro symbol tables *mactable1* and *mactable2* to produce the change sets. The change sets are stored in main memory using the two structures shown below:

```
/* stores macro change sets */
struct macro_change_list {
  char *name;
  struct macro_change_list *next;
};

/* stores macro change sets for the whole context */
struct macro_context_changes {
  char *context;
  struct macro_change_list *del_list;
  struct macro_change_list *mod_list;
  struct macro_change_list *add_list;
  struct macro_context_changes *next;
};
```

Each *macro_context_changes* structure contains pointers to three linked lists of *macro_change_list* structures, corresponding to Tichy's *del*, *mod* and *add* sets. *mac_compare2* returns pointers to lists of the three change sets through its parameters. *mac_compare* uses the three lists and the *macsym* symbol table containing Tichy's history attributes for macros to perform tests 6, 7 and 8 of Tichy's algorithm.

compare2 is the analogue of *mac_compare* for non-macros. *compare2* calls *search_list* to search the *context_changes* (analogous to *macro_context_changes*) for the change sets. If the change sets are not found in main memory, the change sets are read in from the *add*, *mod* and *del* files and stored away. *compare2* uses Tichy's history attributes, stored in *sym* as well as the change sets to perform tests 6, 7 and 8 of Tichy's algorithm for non-macros.

For every module determined to need recompilation by *diff*, their *recompile* field (added to the *nameblock* structure) in their corresponding *nameblock* structure is set to 1, indicating that the module is a candidate for recompilation. Modules whose object file is missing or whose *.versions* file is missing, have their *recompile* field set to 2 and 3 respectively, indicating that the module will be recompiled with all the changes (the user has no option with these). Those modules that were recompiled have their *nameblocks* placed on the *recompiled* linked list; those modules that were candidates for recompilation, but the user refused to recompile are place on the *refused* list. For every pair of modules *a.c* and *b.c* where *a.c* is on the *recompiled* list and *b.c* is on the *refused*

list, the function *compare_dersym* is called to determine whether *b.c* must be recompiled due to an interface error with *a.c*. Two loops achieve this analysis, with the recompiled list being scanned on the outer loop and the refuse list being scanned on the inner loop. Whenever a *refused* module is determined to require recompilation due to an interface error, the module is recompiled with all the changes and the corresponding *nameblock* structure is removed from the *refused* list and added to the end of the *recompiled* list. Note that when a module is recompiled, the list of versions used to create the module's corresponding object code is updated both in main memory and in the *.versions* files only if the the compilation was successful.

Much effort goes into reading in the derived and external symbols from files. Time is saved by storing away the derived and external symbols of both the refused and recompiled modules, so they do not have to be read in from files each time they are compared with some other module's derived and external symbols. Derived symbols are stored using the three structures shown below:

```
struct dersym_node {
  char *name;
  char tag;
  struct dersym_node *next;
};

/* list of reference symbols for a given derived symbol */
struct dersym_list {
  struct dersym_node *sym_node;
  char *name;
  struct dersym_list  *next;
};

/* source file and their derived symbol lists */
struct dersym_source {
  char *source;
  struct dersym_list *list;
  struct dersym_source *next;
 };
```

Each module has a list of derived symbols and each derived symbol has a list of symbols which it transitively references. Storage structures for external symbols are simpler because the symbols which each external symbol transitively references need not be stored.

```
/* stores external symbol */
struct refsym {
  char *name;
  struct refsym *next;
};

/* lists of pointers to refsym lists */
struct refsym_source {
  char *source;
  struct refsym *list;
  struct refsym_source *next;
```

```
};
```

Before performing Kaiser and Schwanke's pairwise comparison, the common include files included in the two modules being compared is determined by the function *intersect*. This information is used when searching the change sets for some symbol, so that only the change sets corresponding to different versions of a common include files are searched. If a derived symbol in one module is an external symbol in the other module being compared, then the routine *changed* is called. *changed* searches the *mod* sets of common include files for symbols (enums, structures or union tags or typedefs) which the derived symbol transitively depends on. If the new version of one of these referenced symbols was modified, then *changed* returns 1 and the refused module is recompiled.

6. An Example

This scenario consists of two modules: *a.c*, *b.c* and one include file *lib1*. The *Makefile* and the initial versions of each file is shown below.

Makefile

```
test: a.o b.o
        smartcc -o test a.o b.o
a.o: lib1
        smartcc -c a.c
b.o: lib1
        smartcc -c b.c
```

lib1.1

```
        typedef int T;
```

a.c.1

```
        #include "lib1"

        f()
        {
        T foo;
        }

        main()
        {}
```

b.c.1

```
        #include "lib1"

        g()
        {
        }
```

smartmake **initializes the system as** follows:

```
Smartmake: Enter the full file names of the files you wish to
exchange.  End with quit.

enter: quit
Object file a.o missing, must remake.
Versions file for a.o missing, must remake.
Object file b.o missing, must remake.
```

```
Versions file for b.o missing, must remake.
smartcc -c a.c
smartcc -c b.c
smartcc -o test a.o b.o
```

The object file and the *.versions* file of each module is missing, so the system is automatically recompiled.

Now a different version of *lib1* is used, with a different version of *T*.

lib1.2

```
        typedef float T;
```

Below is the response of IMS when *lib1.2* is exchanged with *lib1.1*.

```
% smartmake
Smartmake: Enter the full file names of the files you wish to
exchange.  End with quit.
enter: lib1.2
enter: quit
doing change analysis on lib1.1 and lib1.2
symbol T was modified and was referenced in another context
Change analysis has determined that a.o should  be remade.
doing change analysis on lib1.1 and lib1.2
Change analysis has determined that b.o should not be remade.
Do you wish to remake a.o??? y or n: y
smartcc -c a.c
smartcc -o test a.o b.o
```

Since *T* is not used in *b.c*, *b.c* is not a candidate for recompilation.

Suppose a new version of *b.c* , one that references *T*, is now used.

b.c.2

```
        #include "lib1"

        g(baz)
        T baz;

        {}
```

The result of entering *lib1.2* into the system with new version of *b.c* are shown below.

```
% smartmake
Smartmake: Enter the full file names of the files you wish to
exchange.  End with quit.

enter: lib1.2
enter: quit
doing change analysis on lib1.1 and lib1.2
symbol T was modified and was referenced in another context
Change analysis has determined that a.o should  be remade.
doing change analysis on lib1.1 and lib1.2
symbol T was modified and was referenced in another context
```

```
Change analysis has determined that b.o should  be remade.
Do you wish to remake a.o??? y or n: y
smartcc -c a.c
Do you wish to remake b.o??? y or n: n
smartcc -o test a.o b.o
```

Although both *a.c* and *b.c* depend on *T* causing two derivation inconsistencies [11 86], *T* is in not passed between them, so recompiling one and not the other is allowed.

Now a new version of *a.c* is entered into the system, one which shares a common interface with *b.c*.

a.c.2

```
#include "libl"

f()
{
T foo;
g(foo);
}

main()
{}
```

With *a.c.2*, an actual inconsistency exists if *a.c* uses one version of *T* and *b.c* uses the other. *smartmake* does not allow the actual inconsistency to occur.

```
% smartmake
Smartmake: Enter the full file names of the files you wish to
exchange.  End with quit.

enter: libl.2
enter: quit
doing change analysis on libl.1 and libl.2
symbol T was modified and was referenced in another context
Change analysis has determined that a.o should  be remade.
doing change analysis on libl.1 and libl.2
symbol T was modified and was referenced in another context
Change analysis has determined that b.o should  be remade.
Do you wish to remake a.o??? y or n: y
smartcc -c a.c
Do you wish to remake b.o??? y or n: n
Must remake b.o to prevent interface error.
smartcc -c b.c
smartcc -o test a.o b.o
```

7. Conclusion

Although IMS successfully implements Kaiser and Schwanke's algorithm, the system has limitations. One such limitation is that IMS may behave erroneously when conditional compilation constructs are used because the symbols defined may have different definitions, depending upon some previous macro definition. Consider the configuration below:

def1.1

```
#ifdef mac1
    typedef foo int;
#else
    typedef foo char;
#endif
```

def1.2

```
#ifdef mac1
    typedef foo int;
#else
    typedef foo float;
#endif
```

def2.1

```
#define mac1 1
```

module1.c.1

```
#include "def2"
#include "def1"

/* use foo somewhere */
```

module2.c.1

```
#include "def1"

/* use foo somewhere */
```

Now if the change sets for *def1* are computed by first running *cppcdiff* on *module1.c*, then all the change sets will be empty when *cdiff* is run. However, when it's time to decide whether *module2.c* should be recompiled, the same change sets which were computed by first preprocessing *module1.c* are not the same as the ones generated had *module2.c* been used. IMS will fail to recompile *module2.c*, thinking that *foo* did not change. The only solution to this problem is to recompute the change sets for a given include file using conditional compilation, for every module that includes it. This solution, however, severely limits the effectiveness of IMS, since running *cdiff* on every module that includes some include file is very costly.

All the modules must be in the same directory for IMS to be used. This is inconvenient since when large scale software is being developed, the modules are usually distributed across several directories. The problem occurs because *doname* gets the module name from its corresponding object file, which resides in the directory with the *Makefile* and not in the module's directory. The solution to this problem is to force the user to eliminate implied dependencies that occur within *Make* – make the user specify the name of the module which its corresponding object file depends on.

To achieve multiple versions of source units, the numbered version is copied to the unnumbered source file, taking up much file space. A better solution would be to change the

name of the source units, using the *mv* command, when processing is done and later change the names back to the numbered version. This should be the least significant of problems, because in a large scale programming environment, there is usually more than enough file space.

The aforementioned problems notwithstanding, IMS is an efficient means by which a programmer can test changes to include files without having to waste time watching his system recompile. I expect that IMS will be used as a prototype for other implementations of Kaiser and Schwanke's algorithm to be used in future programming environments.

References

[01 82] *Reference Manual for the Ada Programming Language*
1982.

[02 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, 1986.

[03 79] Stuart I. Feldman.
Make -- A Program for Maintaining Computer Programs.
Software -- Practice & Experience, 9(4):255-265, April, 1979.

[04 84] Samuel P. Harbison and Guy L. Steele Jr.
A C Reference Manual
1984.

[05 75] Stephen C. Johnson.
Yacc: Yet Another Compiler-Compiler.
Computer Science Technical Report, (32)July, 1975.

[05 78] Stephen C. Johnson.
A Tour Through the Portable C Compiler.
Bell Technical Journal, 1978.

[06 84] Brian W. Kernighan and Robert Pike.
The UNIX Programming Environment.
Prentice-Hall, 1984.

[07 78] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[08 83] Butler W. Lampson and Eric E. Schmidt.
Organizing Software in A Distributed Environment.
Sigplan annual conference, 1983.

[09 85] David Notkin.
The GANDALF Project.
The Journal of Systems and Software, 5(2)1985.

[10 86] Robert W. Schwanke and Gail E. Kaiser.
Smarter Recompilation.
submitted for publication, 1986.

[11 86] Robert W. Schwanke and Gail E. Kaiser.
Version Inconsistency in Large Systems.
Siemens Research and Technology Laboratories, RTL-86-TR-072, 1986.

[12 86] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems, 8(3):273-291, July, 1986.

[13 84] Walter F. Tichy and Mark C. Baker.
Smart Recompilation.
Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL), :236-244, July, 1984.

How to Use Smartmake

In order to run smartmake from some directory, the following executable object files must be made available: *smartmake*, *cppcdiff*, *cdiff*, *smartcc*, *ccom* and *cpp*. Two lines of *smartcc.c* must be modified to give the locations of ccom and cpp; the other programs that smartcc calls (like the loader) must be available in /lib in UNIX. Smartmake expects a makefile in the following stict format in the directory where it is to be executed:

```
linked object code file: module1.o module2.o ... modulen.o
        smartcc flags module1.o module2.o ... modulen.o

module1.o : includefile1 includefile2 ... includefilen
        smartcc flags module1.c

module2.o : includefile1 includefile2 ... includefilen
        smartcc flags module2.c
...
```

Include files may exist in other directories, however, the modules must be in the directory you plan to run smartmake in. To run smartmake just enter "smartmake <cr>." Smartmake will respond as follows:

```
Smartmake: Enter the full file names of the files you wish to
exchange.  End with quit.

enter:
```

When the system is first being compiled and linked, you won't have any alternate versions of modules of include files to enter, so just enter "quit." Smartmake will use the first version (*file.1*) of version of every source file mentioned in the include file.

When you want to substitute a new version of either a module or an include file, enter the full version name (file.#) after the "enter:" prompt. Enter all of the versions of include files and modules that you wish to substitute. If any module is recompiled, then all of the changes that you specified will be used in the recompilation.

Smartmake also allows you to get rid of inconsistency. If you input "quit" without entering the name of any include files or modules to substitute an nothing gets recompiled, the smartmake requests the following:

```
Smartmake: Do you wish to make a substitution across ever unit???
Enter the names of files to be used.  End with quit.

enter:
```

Here you would enter include files wish you wish to substitue across every module.