Smart Recompilation

WALTER F. TICHY
Purdue University

With current compiler technology, changing a single line in a large software system may trigger massive recompilations. If the change occurs in a file with shared declarations, all compilation units depending upon that file must be recompiled to assure consistency. However, many of those recompilations may be redundant, because the change may affect only a small fraction of the overall system.

Smart recompilation is a method for reducing the set of modules that must be recompiled after a change. The method determines whether recompilation is necessary by isolating the differences among program modules and analyzing the effect of changes. The method is applicable to languages with and without overloading. A prototype demonstrates that the method is efficient and can be added with modest effort to existing compilers.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces, software libraries*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Intelligent software tools, overloading, separate compilation, type checking

1. INTRODUCTION

Modern compilers use contexts for performing type checking across module boundaries. A context specifies which external items a compilation unit may reference, or which internal items the compilation unit must provide. A context is either prepared manually or automatically, stored in the program library, and read in by the compiler during processing of a compilation unit. Examples of manually created contexts are Ada package specifications [1], Mesa and Modula definition modules [8, 15], and "include-files" in C, Berkeley Pascal, and other languages. Automatically generated contexts are computed by the compiler from an existing program unit. For instance, if a block-structured programming language permits an inner block to be compiled separately, compiling the

An extended abstract of this paper appeared in the *Conference Record of the 12th ACM Symposium on the Principles of Programming Languages*, New Orleans, La., January 1985.

Author's current address: Carnegie Group Inc., 650 Commerce Court, Station Square, Pittsburgh, PA 15219.

enclosing block generates a context that specifies the items which the inner block may reference. When the inner block is processed, the compiler initializes its symbols table by reading the context it had produced earlier. Ada subunits and Simula classes [3] can be implemented with automatic contexts.

Contexts have three major uses. First, contexts are effective for implanting common declarations into multiple compilation units, without having to retype them in every unit. Thus, contexts permit the sharing of a single copy of declarations, with the obvious advantages for updates. Second, contexts assure global type correctness for separate compilation. Using the proper contexts, the compiler can check that each unit uses its imported interfaces properly, and implements its exported interface as expected. The value of checking interfaces can hardly be underestimated, because programmers must routinely deal with the interfaces of complex and unfamiliar subsystems. Interfacing to an unfamiliar system is more prone to error than working within one's own system. For this reason, type checking of interfaces is more likely to detect errors than intramodule type checking.

A third use of contexts applies only to automatically generated contexts. An automatic context transmits symbol table information to nested, separately compiled blocks. It contains declarations of all objects visible to the nested blocks, plus additional code generation attributes. These attributes are typically block levels, offsets, and sizes of data structures. Thus, automatic contexts also perform some of the functions traditionally implemented by linkers. A survey of compilation mechanisms using contexts appears in [11].

With current compiler technology, contexts have a serious drawback. To guarantee consistency, all compilation units using a changed context must be recompiled, no matter how small the change. For instance, changing a comment or adding a new declaration to a pervasive context may cause the unnecessary recompilation of the entire system. Similarly, revising a context item that is used in only a few units triggers the recompilation of all units using that context, rather than the few units using the item.

With modern high-level languages, redundant compilations are a serious obstacle. The processing cost of making a minor change or adding a few declarations to a large system may be so great that it retards the growth and evolution of the system. At the very least, it imposes hours of idle time on development teams while everything is periodically recompiled from scratch [6]. High compilation costs also tend to convolute system structure, because they force programmers to incorporate changes in ways that minimize the number of recompilations, rather than preserve well-structuredness.

In effect, guaranteeing system-wide type consistency with contexts may nullify the time savings that separate compilation is supposed to provide. The reason is that the usage relation among contexts and compilation units is too coarse. By refining this relation, this paper arrives at a simple and effective mechanism that causes recompilation of only those units that are affected by a given context change. The next section presents the basic idea of this mechanism, while Section 3 discusses the mechanism in detail. Section 4 describes a prototype implementation using the Berkeley Pascal compiler, and presents some performance results. An extension for languages that permit overloading of identifiers follows. Additional refinements appear in Section 6.

2. OVERVIEW OF SMART RECOMPILATION

Consider a single compilation unit and a set of contexts. We say that a compilation unit "depends" on a context if it may reference identifiers declared in that context. The following recompilation rule applies.

Conventional Recompilation Rule: A compilation unit must be recompiled whenever

(1) the compilation unit changes, or
(2) a context changes upon which the compilation unit depends.

The purpose of part (1) of the rule is obvious. Part (2) guarantees that any context modifications propagate into the dependent units. It also has the effect of checking for syntactic or semantic errors that the change might have introduced. The MAKE program [5] implements the rule. The Ada [1] and Mesa [8] language manuals prescribe similar rules, with the additional aspect that each context is a compliation unit in its own right.

Compilations triggered by part (2) may be redundant. The smart recompilation mechanism presented here eliminates most redundant recompilations. The basic idea is as follows. If a context is modified, a change analysis of the old and new contexts produces a *change set*, which is intersected with the *reference set* of each dependent compilation unit. The change set consists of those context items that were either added, changed, or deleted. The reference set of a compilation unit records which context identifiers were used outside their declaring contexts. If the intersection of these two sets is empty, then the compilation unit need not be reprocessed.

Figure 1 illustrates. File *f.cxt* is a context containing declarations for a hypothetical traffic light control program. File *prog.p* is a compilation unit dependent upon context *f.cxt*. All examples are formulated in Berkeley Pascal, which is equipped for separate compilation. The directive *#include* instructs the compiler to read in a context.

Assume the following sequence of events. File *prog.p* is compiled, which produces a reference set for *f.cxt* and an object module. The reference set is {*NumOfStreets*, *NumOfAvenues*, *PrimaryIntersect*, *TrafficLights*}. This set is determined by computing the transitive closure of dependencies among declarations. For instance, the variable *Grid* depends directly upon the first three declarations in the set, and indirectly upon the fourth. Note that the enumeration literal *red*, though referenced in *prog.p*, is not included. Enumeration literals are subordinate to the declaration of the enumeration type, and including the identifier of the enumeration type suffices. Whenever anything about an enumeration literal is changed, the entire enumeration type is considered changed, and recompilation will be triggered properly. Record types are treated in a similar way: Reference and change sets summarize access to, and change of, record fields, by listing the corresponding record types.

Now assume that we redefine the type *TrafficLights* (by adding, say, the literal *RedBlink*). When the program is reprocessed, the smart compiler detects that *prog.p* depends on a changed context. It therefore computes the change set by comparing the old and new versions of the context. (The comparison also assures that the new context is syntactically and semantically correct.) The change set contains *TrafficLights*. Its intersection with the previously computed reference

```
File f.cxt:
  const
    NumOfStreets    = 40;
    NumOfAvenues    = 20;
  type
    TrafficLights   = (red, amber, green);
    MorningRush     = 7 .. 9;
    EveningRush     = 16 .. 18;
    PrimaryIntersect = record S, N, E, W: TrafficLights end;
File prog.p:
  #include "f.cxt"
  var
    Grid: array[1 .. NumOfStreets, 1 .. NumOfAvenues] of PrimaryIntersect;
      . . .
  Grid[i, j]. S := red;
```

Fig. 1. Compilation unit *prog.p* with one context.

set is not empty, and the smart compiler therefore reprocesses *prog.p.* No recompilation would occur if instead we had changed the type of *MorningRush*, added a new declaration, inserted a comment, or rearranged the textual layout.

The procedure outlined above still has two flaws. First, it may mask redeclarations and overloading errors. For instance, adding another declaration for *Grid* to the file *f.cxt* will trigger no recompilation and therefore no error message, although the program would obviously be illegal. Second, a context may reference free identifiers, but supply no indication where the corresponding declaration can be found. For example, it is legal to split *f.cxt* into two files and include them both in *prog.p.* In this case, the second context may refer to an item that is not declared in it. The problem with this situation is that the semantic correctness of the reference to the free identifier cannot be checked by analyzing the second context alone. The following section shows how to eliminate these problems.

3. SMART RECOMPILATION WITHOUT OVERLOADING

This section states precisely how smart recompilation works. Overloading of identifiers and qualification of referenes are not permitted, but Section 5 will lift these restrictions. Overloading means that there may be more than one declaration for the same identifier in the same scope. Qualification means that an identifier reference is syntactically associated with the identifier's declaring context. Usually, this association is made by simply concatenating context name and identifier or by special syntax in import-clauses. Qualification prevents conflicts if two or more contexts introduce the same identifier. Simple file inclusion, for example, works without qualification and can lead to errors caused by multiply declared identifiers. We first describe smart recompilation without qualification, and then outline the simplifications possible with qualification.

To avoid unnecessary detail, we assume that separate compilation is only permitted at the global level. Removing this restriction is merely a matter of bookkeeping and involves automatically generated contexts for each separately compiled, inner block as outlined in the Introduction.

3.1 The Multiversion Model

We assume that “changing” a program module (i.e., a context or a compilation unit) does not really change it; instead, a new one is created. Normally, the newly created module starts out as a copy of an existing one and can be modified with an editor until the editing session terminates. At this time, the new module becomes immutable.

We postulate the presence of a version control system that composes configurations. The version controller decides which version of a module to pass to the compiler for processing. For this purpose, the controller keeps track of a special relation among modules, called the *Revision-Of* relation. This relation specifies for each module, from which other module (or modules) it was created by manual editing. This relation is important for implementing version selection rules. For example, a selection rule useful during software development is to always choose the newest revision of a module. More sophisticated rules appear in [7, 12].

Of course, the version control system must be able to store arbitrarily many revisions per module. By saving only the differences between successive revisions, the space requirements are modest, as has been demonstrated by a number of systems [9, 7, 13, 14].

3.2 The Compilation Model

Contexts and compilation units contain a number of declarations. The term declaration stands for any named construct that can be defined in a programming language. Examples are symbolic constants, types, variables, macros, generics, subprograms, subprogram headers, processes, and abstract data types. A declaration introduces an identifier and associates that identifier with a body. The body can be accessed elsewhere (in particular, in the body of another declaration) by using the identifier in some referencing construct (for instance, in an assignment statement).

A declaration may introduce subordinate identifiers. Examples are enumeration literals, record fields, keyword parameters, and operations of abstract data types. The following definition allows us to summarize references to subordinate identifiers as references to the main identifier.

Definition: Declaration A depends on declaration B iff the body of A references the identifier of B or one of B’s subordinate identifiers.

For example, if procedure P assigns to field F of variable V, if V is of type R, and if R is a record type with a field F of type T, then P depends on V, which depends on R, which depends on T. Obviously, the summarization discards some information. Section 6 discusses a more sensitive smart recompilation mechanism that retains this information.

Transitive dependencies of declarations are important for determining the effect of changes. In the above example, if T changes, P must be recompiled. Note that circular dependencies are permitted. Circularities are needed, for instance, for declaring records containing pointers to each other.[1]

[1] Ada and Mesa prohibit circular dependencies to cross context boundaries by imposing a partial ordering on contexts. However, simple text file inclusion can easily lead to intercontext circularities. The mechanism described here treats circular references properly.

An identifier introduced in a context by a declaration is said to be "declared" in that context. An identifier that is referenced in a context, but not declared in it, is said to be "free" in that context. There are two ways of supplying declarations for free identifiers. The first one is if the context itself contains nested inclusion directives for obtaining missing declarations from other contexts. This approach is taken in Ada, Mesa, and Modula. Alternatively, a context may inherit declarations from the compilation unit or other contexts in which it is used. Include-files are typically applied in this fashion. A problem with free identifiers is that they complicate the process of computing differences between contexts and ascertaining the semantic correctness of contexts. Another problem is that it is generally impossible to compute the transitive closure of dependencies by starting a search in the context with the free identifiers. The complete transitive closure is only available when all contexts are combined during compilation. The mechanism introduced here permits nested context inclusion as well as free declarations.

The following restriction guarantees that extra declarations can be added to contexts and that unused declaration can be removed or changed, without affecting already generated code. This restriction is important because it allows contexts to be changed without necessarily forcing recompilations.

Restriction 1. Code generated for referencing some declaration D declared in a context may only be derived from D itself and from declarations that D depends on transitively.

This restriction precludes optimization techniques that add dependencies which are not apparent from explicit references in the source program. Interprocedural optimization usually adds a great number of such dependencies. As an alternative for Restriction 1, one could record all the additional dependencies introduced and add them to the reference sets. This technique is not difficult to implement, but since it is strongly language- and compiler-dependent, we chose the more general approach reflected by Restriction 1. Furthermore, note that the restriction only applies to declarations in contexts. Hidden dependencies among declarations appearing in compilation units are allowed. For a compilation unit (not a context), any change requires reprocessing, which reanalyzes embedded, hidden dependencies. Thus Restriction 1 is not as severe as it appears at first. (See also the discussion of attribute dependencies in Section 6.)

A subtle example where the restriction applies is the following. The address of a global variable introduced by a context should not be assigned by the compiler, since that address may depend on the size and number of preceding, unrelated variables. The address must either be determined by a later phase (e.g., the loader), or must be made an explicit part of the declaration. For global variables, the former approach is the one taken by most language systems; automatically generated contexts employ the latter technique. As an example for the latter, suppose a subprogram P at block level $n (n \geq 2)$ is compiled separately. When compiling the block enclosing P, a context is generated that contains all declarations visible to P, with associated code generation attributes. In particular, each visible variable V at block level $b (1 \leq b < n)$ has the address (b, d) associated with it, where d is the offset of V in its activation record. If the block containing V is changed, recompilation of that block produces a new context with a possibly altered address for V. The determination of whether P needs to be recompiled must take into account the entire declaration of V, including the old and new addresses.

Restriction 2. Within a context, directives for context inclusion must appear before declarations.

This restriction is needed for two reasons: First, it makes it easier to determine whether context inclusion, and therefore system structure, have changed. Second, it makes it feasible to deal with inherited declarations and their effects on semantic correctness. Without the restriction, the directives could be intermixed with declarations, making it difficult to determine whether a context remains semantically correct when the directives are moved around, deleted, or added.

Note that Restriction 2 is easily satisfied, because a context with an inclusion directive in the middle can always be split. Moreover, the restriction applies only to contexts, and not to compilation units. The restriction is already enforced by some programming languages, for example Ada, Mesa, and Modula.

3.3 Problem Statement

Given a compilation unit M_0 and contexts $M_1, \ldots, M_n$, assume that the configuration $C = \{M_0, \ldots, M_n\}$ is legal and was compiled successfully. The compilation resulted in a translation with an associated history attribute containing the following sets:

$DECL_i$:	The identifiers declared in $M_i (0 \le i \le n)$;
REF_i:	The identifiers declared in M_i and transitively referenced in some other context or compilation unit $M_j (1 \le i \le n, 0 \le j \le n, i \ne j)$. ($REF_0$ is not needed.)

Given a new context $\bar{M}_x (1 \le x \le n)$, inspect only M_x, $\bar{M}_x$ and the sets $DECL_i$ and REF_i to determine the following:

(a) Is the configuration $\bar{C} = \{M_0, \ldots, \bar{M}_x, \ldots, M_n\}$ legal, that is, syntactically and semantically correct?

(b) Are the translations generated by the compiler for C and $\bar{C}$ functionally equivalent?

3.4 Solution

The following decision procedure answers the two questions above by performing a change analysis.

Change Analysis 1 (*No Overloading, No Qualification*)

Test 1.	Analyze $\bar{M}_x$ syntactically, as specified by the programming language manual. If there are any errors, then $\bar{C}$ is illegal.
Test 2.	Compare the context inclusion directives (if any) in M_x and $\bar{M}_x$. If they are not identical, recompile.

Test 3. Analyze $\bar{M}_x$ semantically. The rules specified by the programming language manual apply, except that occurrences of free identifiers are legal. If there are any other errors then $\bar{C}$ is illegal.
Compare M_x and $\bar{M}_x$ and determine the following sets.

ADD_x:	The identifiers declared in $\bar{M}_x$ but not in M_x.
DEL_x:	The identifiers declared in M_x but not in $\bar{M}_x$.
MOD_x:	The identifiers declared in both M_x and $\bar{M}_x$ whose declarations differ.
$FREE_x$:	The identifiers free in $\bar{M}_x$.
$AMREF_x$:	The identifiers transitively referenced by declarations in ADD_x and MOD_x.

Test 4. If $AMREF_x \cap FREE_x \neq \varnothing$, then an added or modified declaration references a free identifier and recompilation is necessary.

Test 5. If $DEL_x \cap FREE_x \neq \varnothing$, then a deleted identifier is referenced in $\bar{M}_x$, and $\bar{C}$ is illegal.

Test 6. If $MOD_x \cap REF_x \neq \varnothing$, then a local declaration changed that is referenced elsewhere, and recompilation is necessary.

Test 7. If $ADD_x \cap DECL_j \neq \varnothing$ for some j, $0 \leq j \leq n$, $j \neq x$, the $\bar{M}_x$ introduced a declaration that conflicts with an external one, and $\bar{C}$ is illegal.

Test 8. If $DEL_x \cap REF_x \neq \varnothing$, the $\bar{M}_x$ is missing a declaration that is referenced externally, and $\bar{C}$ is illegal.

End Change Analysis 1

The purpose of Test 1 is obvious. Test 2 is rather conservative, in that it triggers recompilation whenever inclusion directives change. This test could be refined to allow addition of inclusion directives if no multiple declarations arise. Tests 1 and 3 combined determine whether the context is legal internally, as far as that is possible. Although free identifiers are permitted, the rule "declaration before reference" applies, if the programming language specifies it.

Free identifiers in $\bar{M}_x$ present a problem since their use cannot be checked locally for legality. However, M_x is legal by assumption. Thus, if a free identifier is referenced in exactly the same way in both M_x and $\bar{M}_x$, then the use of that free identifier is correct in both. Otherwise, only a recompilation can check legality. Test 4 implements this analysis.

Deletions can result in free identifiers that are not necessarily included in $AMREF_x$; Test 5 checks for those. Test 6 causes a recompilation if a referenced declaration was modified. Test 7 assures that any new declaration does not interfere with existing ones. This check must be relaxed if overloading is permitted. Test 8 prints an error message if a deleted declaration is still referenced externally.[2]

THEOREM. *If Change Analysis* 1 *detects no errors* (*Tests* 1, 3, 5, 7, *and* 8) *and triggers no recompilation* (*Tests* 2, 4, *and* 6), *then*

(*i*) $\bar{C}$ *is a legal configuration and*
(*ii*) *the translations of* C *and* $\bar{C}$ *are functionally equivalent.*

[2] The tests do not have to be executed in the order implied by their numbers. For example, Test 1 (syntax analysis) and Test 3 (semantic analysis) may be performed in an overlapped fashion as is usual in modern compilers. Tests 4 through 8 are independent and may be performed in any order.

Table I. Classes of Changes and Possible Errors Introduced

	Difference	Test
A.	Textual layout (spacing, comments)	No effect
B.	Syntax errors	Test 1
C.	Inclusion directives	Test 2
D.	Order of declarations	
	Declaration before reference (if applicable)	Tests 3, 4
E.	Modified declaration	
E.1.	Modified declaration references local declarations	Test 3
E.2.	Modified declaration references external declarations	Test 4
E.3.	Modified declaration introduces undeclared identifiers	Test 4
E.3.	Modified declaration is referenced locally	Test 3
E.4.	Modified declaration is referenced externally	Test 6
F.	Additional declaration	
F.1.	Additional declaration references local declarations	Test 3
F.2.	Additional declaration references external declarations	Test 4
F.3.	Additional declaration introduces undeclared identifiers	Test 4
F.3.	Redeclaration of existing local declaration	Test 3
F.4.	Redeclaration of existing external declaration	Test 7
G.	Deleted declaration	
G.1.	Deleted declaration is referenced locally	Test 5
G.2.	Deleted declaration is referenced externally	Test 8

PROOF. (i) Consider how M_x and $\bar{M}_x$ may differ. The simplest differences are (A) different textual layout and (B) syntactic errors in $\bar{M}_x$. Clearly, change of textual layout like spacing and commenting has no effect in free-form languages, as long as the order of the individual tokens is the same. Syntactic errors are detected by parsing.

In the absence of syntactic errors, the only other differences are (C) different inclusion directives, (D) different order of declarations, (E) declarations that have the same identifier but different bodies, (F) additional declarations in $\bar{M}_x$, and (G) declarations missing from $\bar{M}_x$. Table I classifies the possible differences, and shows which test in Change Analysis 1 determines whether a difference constitutes an error.

(ii) Consider configurations C and $\bar{C}$. By (i), both are legal. $\bar{C}$ differs from C in that it may have deleted, added, or modified declarations. By Tests 5 and 8, deleted declarations are not referenced anywhere in $\bar{C}$. By Restriction 1, code generated for accessing other declarations is not affected. Consequently, $\bar{C}$'s translation differs from C's translation only in that some "deadwood" was eliminated.

Added declarations are not referenced externally because of Test 7 and because C is legal. Modified declarations are not referenced externally because of Test 6. Added and modified declarations may be referenced locally, though. (For example, there may be a declaration in $\bar{M}_x$ that references a modified declaration in the same context.) However, since REF_x reflects the transitive closure, the locally referencing declarations cannot be referenced externally. Thus they are all "deadwood," and can be left uncompiled. Recompilation will be triggered if a reference to one of them is inserted later. □

```
File f1.cxt:
  const
    NumOfStreets  = 40;
    NumOfAvenues = 20;
File f2.cxt:
  #include "f1.cxt"
  type
    TrafficLights  = (red, amber, green);
    StopLights     = (redblink, yellowblink);
    MorningRush = 7 .. 9;
    EveningRush  = 16 .. 18;
File f3. cxt:
  type
    PrimaryIntersect    = record S, N, E, W: TrafficLights end;
    SecondaryIntersect = record S, N, E, W: StopLights end;
File prog.p:
  #include "f2.cxt"
  #include "f3.cxt"
  var
    Grid: array[1 .. NumOfStreets, 1 .. NumOfAvenues] of PrimaryIntersect;
      . . .
  Grid[i, j].S := red;
```

Fig. 2. Compilation unit *prog.p* and three contexts.

```
⟨f1.cxt⟩  NumOfStreets, NumOfAvenues |
⟨f2.cxt⟩  TrafficLights, StopLights | MorningRush, EveningRush
⟨f3.cxt⟩  PrimaryIntersect | SecondaryIntersect
⟨prog.p⟩  Grid |
```

Fig. 3. History attribute generated for *prog.p*.

3.5 Example

Consider Figure 2, a contrived example using three contexts. For simplicity, we assume that contexts can only be requested at the outermost block level. It does not matter whether a context is included by the compilation unit or another context; the inclusion directives can always be rearranged such that they appear in the compilation unit only. In fact, automatically "flattening" the file inclusion in this manner is the best way to handle inherited free identifiers.

Figure 3 shows the history attribute generated by compiling *prog.p*. The sets *DECL* and *REF* of a context can be represented overlapped, because the former is a superset of the latter. The vertical bar (|) separates the reference set from the rest of the declarations.

The reader is encouraged to check what happens if declarations in the contexts are added, deleted, or modified. For instance, removing or changing the constant *EveningRush* has no effect, whereas removing the constant *NumOfAvenues* causes an error message, and changing it triggers a recompilation. Similarly, changing *StopLights* causes a recompilation. Note that the only reference to *StopLights* is in the unreferenced declaration *SecondaryIntersect*. However, since *SecondaryIntersect* appears in a different context, recompilation is necessary to check the legality of the change.

3.6 Putting It All Together

When comparing two contexts during change analysis, the textual layout of declarations and other syntactic variations should have no effect. Comparing abstract syntax trees filters out these differences. Two declarations are identical if they have the same identifier and their abstract syntax trees are identical. This test can be carried out by a simple, recursive program.

The version control system maintains a pool of object modules which were compiled previously. Whenever the object module of a compilation unit is requested, the version controller checks whether one already exists. If so, saving the recompilation may be possible. First the version controller inspects the history attribute of the candidate object module to determine which contexts were used to generate it. If an exact match with the desired configuration is found, the object module can be used as is. Otherwise change analysis of the old and the new configurations is necessary. If this analysis finds no errors and triggers no recompilation, the existing object module can be reused.

When an object module is reused, a new history attribute is added to it. The new history attribute is the same as the old one, except that the set $DECL_x$ is updated to reflect the replaced context. The reference sets remain unchanged. This approach guarantees that future change analyses operate reliably.

When more than one context is replaced, the contexts are handled one after the other, producing a new history attribute after every step. If declarations are moved from one context to another, this technique may generate spurious error messages. For example, Tests 5 or 8 would find an error in the context from which the declartion was removed, whereas a fresh recompilation would not report that error. Thus, during multicontext changes, errors detected by Tests 5, 7, and 8 should actually trigger a recompilation. Saving the potentially redundant compilation is possible with a more detailed analysis that checks whether errors reported by these tests cancel. This analysis needs data structures that record precisely where each identifier is referenced. The details are left to the reader.

A similar inefficiency results from the fact that reference sets do not contract after suppressed compilations. For example, suppose declaration A in context M is not referenced, and is the only declaration that references B in context N. When A is deleted, no recompilation is necessary, but the reference set of N still lists B as in use. When B is deleted at a later time, an unnecessary recompilation (or error message) results. This slight inefficiency could be corrected by associating a reference count with each element in the reference sets, or by using the method mentioned in the previous paragraph. Note, however, that the mechanism is safe in that it will never omit a necessary recompilation.

4. PROTOTYPE

We implemented a prototype by modifying the Berkeley Pascal Compiler, *pc*, running on the UNIX® operating system. Version control was provided by MAKE[5] and RCS[14]. RCS, short for Revision Control System, collects revisions of modules into revision groups. It conserves space by storing only deltas.

® UNIX is a trademark of AT&T.

4.1 Implementation

Adding the generation of the reference sets to the Pascal compiler was straightforward. Each symbol table entry was expanded with a reference bit and a pointer to the file name in which the declaration appeared. The lexical analyzer of the compiler turns on the reference bit whenever it encounters an identifier that had been declared in a different context. Computing the transitive closure of dependencies is simply a matter of following all links emanating from a symbol table entry, and setting the reference bit in the reached declarations. For example, if an array variable is used, the declaration of that variable, the type of the array elements, and the index type(s) are marked. If compilation succeeds, a simple program scans the symbol table and writes the sets *DECL* and *REF* of each context into a file.

The change analysis consists of two separate phases. The program *cdiff*(short for context *diff*erence) implements the first phase. It takes a pair of context revisions and performs Tests 1, 2, 3, 4, and 5. It also assures that the inclusion directives are at the beginning of the context. Furthermore, *cdiff* produces the sets *ADD*, *DEL*, and *MOD* as output. The collection of these sets is called the *change set*. The change set is needed for Tests 6, 7, and 8 later. Essentially, these tests compare the change set with the history attribute of an object module. This arrangement has the advantage that the change set is computed once and can then be matched against the history attributes of several object modules. This division saves time, because a context change normally affects several compilation units which must all be brought up to date. If *cdiff* detects any errors, it produces no change set and triggers a recompilation if the user wants to have more detailed error messages.

Cdiff was easy to build. It is essentially the declaration parser of the Pascal compiler. It reads in two contexts, say M and $\bar{M}$, builds up a symbol table for each, and then compares individual entries. Each entry in the symbol tables is the abstract syntax tree of a declaration. To produce the change set, the symbol table for context M is traversed. For each identifier declared in M, the corresponding identifier is looked up in the symbol table for context $\bar{M}$. If the two declarations are not identical, then the identifier is included in the set *MOD*; if the identifier is not declared in $\bar{M}$, it is added to the set *DEL*. In order to detect declarations that have been added, identifiers are marked as they are looked up in the symbol table for $\bar{M}$. Any entries still unmarked in $\bar{M}$ after scanning M's table are part of the set *ADD*. Hashing looks up entries quickly.

A controlling program, called *spc* (for Smart Pascal Compiler), provides the mechanism for comparing the change set with the history attribute. The job of determining which contexts and compilation units must be analyzed is left to MAKE. If anything appears out of date, MAKE invokes *spc*, which retrieves old contexts from RCS, computes the change sets, intersects them with the reference sets, and, if necessary, starts the modified Pascal compiler.

4.2 Performance

Performance of the implementation was surprisingly good. Measurements on about 20 compilation units (of up to 700 lines, with an average of 200 lines) and 3 contexts (of up to 190 lines, with an average of 140 lines) were performed on a

VAX/780 running the Berkeley UNIX system 4.2. Results consistently indicated that saving a single compilation more than amortizes the cost of the extra analysis; any additional recompilation that is suppressed constitutes a net saving.

For calculating the potential savings achievable with *spc*, consider the following costs: (a) generating the history attribute, (b) generating the change set, and (c) comparing change set and history attributes. The time for generating the history attribute was not measurable with the limited accuracy of the UNIX clock. Given a clock accuracy of 0.1 seconds and an average compilation time of about 20 seconds per module, it follows that writing the history attribute takes less than 1 percent of the total compilation time. This is not surprising, since compilers are basically I/O bound, and the history attribute represents a tiny fraction of total I/O. The additional file space needed is also quite small, and could be nearly eliminated by redesigning the object module format. Much of the information contained in the history attribute is already buried in the object module, where it is needed for the debugger.

The cost of producing the change set is, on the average, less than a third of the cost of a compilation. The reason is that the amount of input is small, and output is even less. Furthermore, producing the change set is a one-time cost: It is computed once, but will be intersected with the history attributes of many object modules.

Finally, computing the intersections is fast. For nonempty change sets, the average time was about 0.2 seconds (less if the change set was empty). This is a mere 1 percent of compilation cost. Stated another way, determining whether recompilation is necessary is two orders of magnitude faster than compiling.

In summary, the only nonnegligible cost is in computing the change set. However, this cost is already more than amortized by suppressing a single compilation. Thus, substantial savings can be obtained even in systems of moderate size. Note that the contexts used for measurement were fairly large; in languages like Ada or Modula, contexts are probably smaller, reducing the cost of computing the change set. Furthermore, the compilation units were small; with larger units, the advantage of saving compilations becomes more pronounced. One should also consider that the Berkeley Pascal compiler is already reasonably fast. Highly optimizing compilers or compilers for complicated languages are slower, and even greater savings are possible.

5. OVERLOADING

A number of programming languages allow overloading of identifiers (i.e., several declarations for the same identifier in the same scope). This section describes how to extend smart recompilation for overloading.

Languages that permit overloading also specify rules for overload resolution (i.e., for resolving the ambiguities introduced by overloading). For each identifier reference, overload resolution inspects all alternative declarations of that identifier and succeeds if it can unambiguously select a single declaration from this set. Overload resolution may fail in two ways: A reference of an identifier may be *unresolvable*, that is, there may not exist a declaration that overload resolution can select (there simply may be no declaration of that identifier, or there may be several, none of which fit the constraints of the reference) or a reference may be *ambiguous*, that is, there may exist two or more candidate declarations that could be selected equally well. An efficient algorithm for overload resolution in Ada appears in [2].

A complicating factor is that in most languages, overloading is restricted to certain classes of identifiers. In Ada, for example, only subprogram identifiers, operators, enumeration literals, and entry identifiers may be overloaded. For clarity, we first consider unrestricted overloading (i.e., languages in which any identifier may be overloaded) and then outline the changes for restricted overloading.

5.1 Unrestricted Overloading

Assume that configuration C has been compiled successfully, and that the sets $DECL_i$ and REF_i have been computed as before. Because of overloading, the sets $DECL_i$ may have pairwise nonempty intersections. Although overload resolution inspects all declarations with the same identifier, a set REF_j records an identifier only if a declaration in M_j with that identifier was actually selected.

A case analysis similar to the proof in Section 3 yields the decision procedure for smart recompilation. An important consideration is that the identifier of any added, modified, or deleted declaration, as well as any identifier referenced by added or modified declarations, may be overloaded. Furthermore, overloading may be caused by a context other than the one being analyzed.

Change Analysis 2 (*Unrestricted Overloading, No Qualification*)

Test 1′. Syntax analysis of $\bar{M}_x$ (same as in Change Analysis 1).

Test 2′. Comparison of context inclusion directives in M_x and $\bar{M}_x$ (same as in Change Analysis 1).

Test 3′. Semantic analysis of $\bar{M}_x$. The rules specified by the programming language manual apply, except that occurrences of unresolvable identifiers are legal. If there are any other errors, then $\bar{C}$ is illegal.
Compare M_x and $\bar{M}_x$ and determine the following sets.

ADD_x:	The identifiers declared in $\bar{M}_x$ but not in M_x.
DEL_x:	The identifiers declared in M_x but not in $\bar{M}_x$.
MOD_x:	Consider the set of identifiers declared in both M_x and $\bar{M}_x$. To obtain MOD_x, delete from this set every identifier ID for which the following holds: For every declaration of ID in M_x there is an identical declaration of ID in $\bar{M}_x$, and vice versa.
$UNRES_x$:	The identifiers unesolvable in $\bar{M}_x$;
$AMREF_x$:	The identifiers transitively referenced by declarations in ADD_x and MOD_x.
$CREF_x$:	The indices of those contexts other than $\bar{M}_x$ whose declarations can be referenced in $\bar{M}_x$. (This set is determined by the context inclusion directives.)

Test 4′. If $(AMREF_x \cap UNRES_x \neq \emptyset) \vee (AMREF_x \cap DECL_j \neq \emptyset)$ for some $j \in CREF_x$, then recompile.

Test 5′. If $DEL_x \cap UNRES_x \neq \emptyset$, then recompile.

Test 6′. If $(MOD_x \cap REF_x \neq \emptyset) \vee (MOD_x \cap DECL_j \neq \emptyset)$ for some j, $0 \leq j \leq n, j \neq x$, then recompile.

Test 7′. If $(ADD_x \cap REF_x \neq \emptyset) \vee (ADD_x \cap DECL_j \neq \emptyset)$ for some j, $0 \leq j \leq n, j \neq x$, then recompile.

Test 8′. If $DEL_x \cap REF_x \neq \emptyset$, then $\bar{C}$ is illegal.

End Change Analysis 2

As an example, consider modified declaration D in context $\bar{M}_x$. If D is referenced externally (first half of Test 6′), or if the body of D references an unresolvable identifier (first half of Test 4′), then recompilation is necessary as in Change Analysis 1. If D is overloaded with declarations in other contexts (second half of Test 6′), recompilation checks whether D introduced an overloading error (whether actually selected or not). Finally, recompilation is required if D references an identifier for which there are competing declarations in other contexts, even if the reference could have been resolved locally (second half of Test 4′). The recompilation checks whether the competing declarations cause ambiguities in the body of D. Semantic analysis of $\bar{M}_x$ (Test 3′) merely checks the legality of strictly local overloading of D and the identifiers referenced by D.

Added declarations are handled in exactly the same way as modified ones. Deleted declarations do not necessarily introduce errors, because they may be overloaded. Unchanged declarations need not be reanalyzed.

For single context changes, Test 5′ can be accelerated as follows. If the declaration deleted from $\bar{M}_x$ is not overloaded in $\bar{C}$ (i.e., the last declaration of that identifier was removed), then $\bar{C}$ is known to be illegal without a recompilation.

In general, overloading requires more recompilations. However, if added, deleted, and modified declarations are not overloaded and reference no overloaded identifiers, then Change Analysis 2 produces no more compilations than Change Analysis 1. The correctness proof of Change Analysis 2 is left as an exercise.

5.2 Restricted Overloading

Restricted overloading means that only certain identifiers may be overloaded. For example, identifiers of variables are usually not overloadable. Accordingly, the sets $DECL_i$, REF_i, ADD_i, DEL_i, and MOD_i must each be split into two classes: one containing the overloadable identifiers, the other the nonoverloadable identifiers. Smart recompilation executes Change Analysis 2 on the overloadable identifiers, and Change Analysis 1 on the rest.

A further complication arises because the same identifier may not simultaneously be both overloadable and nonoverloadable. For example, the identifier of a variable in one context may not be used as a subprogram identifier in another context. Although compilation of configuration C guarantees that C is free from such overlaps, $\bar{M}_x$ may introduce them into $\bar{C}$. The following simple steps guard against these errors. First, if a declaration is changed so that it becomes overloadable though it was nonoverloadable before (or vice versa), this modification is treated like an addition to the overloadable identifiers and a deletion from the nonoverloadable identifiers (or vice versa). Thus, before the two change analyses are executed, the sets MOD_x and DEL_x must be set up to reflect the migration of identifiers from one class to another. Second, no overlap is permitted between the nonoverloadable and overloadable identifiers. This test, although seemingly expensive, can be executed efficiently by observing that significant changes can only be introduced by ADD_x and MOD_x. The details are again left to the reader.

5.3 Qualification of References

Recall that qualification means that every identifier reference must be syntactically associated with the context declaring the identifier. Qualification simplifies smart recompilation because it eliminates intercontext conflicts. Change Analysis 1 becomes both simpler and faster, because Test 7 can be eliminated. In Change Analysis 2, the second halves of Tests 4′, 6′, and 7′ can be removed, since overload resolution cannot cross context boundaries. A minor complication is that locally declared identifiers normally must be referenced unqualified. A simple solution is to add the qualification implicitly to all local references.

Some programming languages provide facilities for both qualified and unqualified references, for example Modula and Ada. However, Modula's *import*-clause satisfies our criterion for qualification. The clause uniquely specifies from which context an identifier is taken, even if it is not preceded by a context name. Thus the missing qualification can be added implicitly, and the simpler form of Change Analysis 1 applies.

Ada's context inclusion facilities are more complex. They have two parts, a *with*-clause and a *use*-clause. The *with*-clause introduces qualified identifiers from a context. If the *use*-clause is added, qualification may (but need not) be omitted. Unqualified references may introduce intercontext conflicts. For example, adding a declaration to a context causes an error if (1) there is another declaration with the same identifier in a competing context, (2) at least one of the declarations makes the identifier not overloadable, and (3) the identifier is referenced unqualified. This situation is similar to simple file inclusion, except that two conflicting declarations may coexist in separate contexts as long as all references are qualified. The simplest solution is to treat qualified and unqualified references separately. The qualified references are handled with the simplified analyses described above. For the unqualified references, Change Analyses 1 and 2 are executed, but only for those contexts that appear in use-clauses, and restricted to those identifiers that are actually referenced unqualified.

A further complication in Ada comes from the interaction between overloading and qualification. For example, it is legal to have the same identifier simultaneously overloadable and not overloadable, provided the corresponding declarations are in separate contexts. However, in that case, *all* references to that identifier must be qualified. This rule involves the treatment of identifiers that migrate from one class to the other, and the test for the intersection among the two classes. It appears that the interaction between restricted overloading and unqualified references in Ada is unnecessarily complex.

6. REFINEMENTS AND EXTENSIONS

The mechanisms described here do not eliminate all redundant compilations. Several improvements are possible, even beyond the ones already discussed in Sections 4 and 5. First, the handling of free or unresolvable identifiers could be more sophisticated. It is usually possible to derive some information about a free identifier from its use. For instance, if it is obvious from context that a certain declaration is a range type, then using it as a range in a new declaration need not trigger recompilation (Tests 4 and 4′). Note also that the history attribute contains enough information to determine where missing declarations can be found, and an exact legality check could be done, possibly saving redundant recompilations. Further experimentation is necessary to determine whether the gain outweighs the cost.

Another refinement would be to replace declaration dependencies with attribute dependencies, as proposed by Dausmann [4]. Attribute dependencies record which attributes of declarations were actually used during compilation. For instance, assume a compilation unit uses only information about certain fields in a record, but never the record's size attribute. Then recompilation is not needed if a new field is appended to the record type. Similarly, adding a default parameter to a subprogram need not trigger recompilation of all calling units, provided subprograms are implemented such that default parameters are supplied by the callee and not the call site. Clearly, this refinement is strongly language- and compiler-dependent, and the code generator must be designed with smart recompilation in mind.

A smart recompilation mechanism using attribute dependencies and exact legality checks may achieve the theoretical optimum of never causing an unnecessary recompilation. However, only experimentation can answer the question whether the potential savings justify the cost of the additional analysis. Perhaps the simple mechanism described here is already so close to the optimum in the majority of cases that only marginal gains can be accomplished, or perhaps the additional analysis is so expensive in storage and runtime costs as to completely overwhelm the incremental improvement. On the other hand, very large software projects might benefit.

The Desoto project [10] attempted to build a smart recompilation mechanism for Mesa. The basic ideas were similar, but the project failed, partly because of the lack of an adequate version control system. A system for storing multiple revisions of source code reliably and economically is indispensable. Furthermore, each object module must carry a history attribute that unambiguously specifies how the object module was produced. In the prototype discussed here, the decision of when to start *spc* was based on MAKE's timestamp mechanism rather than on inspecting history attributes. Timestamps are an approximation of history attributes, and it is possible to construct situations where MAKE does not start *spc* although it should. Further development in version control systems is needed.

An important extension of smart recompilation helps programmers update modules after changes. Recompilation is often not sufficient to bring a system back up to date; some reprogramming may be necessary. For example, recompilation suffices if a record type is expanded or the fields are reordered. However, if a parameter is added to a subprogram, or the type of a variable changes, then an adaptation of the using compilation units is required. The following, somewhat more sophisticated change analysis can help with the updating.

Suppose the change analysis examines changed declarations in old and new contexts, and determines whether recompilation is sufficient or not. In conjunction with the history attributes, the analysis can then offer two services. First, the programmer revising a context can be informed (or warned) about the impact his changes may have on the rest of the system, in terms of the number of modules to be recompiled and the number of modules to be edited. Second, if the change is to be carried out, change information is passed to an editor, which steps the programmer through the discrepancies, displaying the old and new revisons of the appropriate declarations, and perhaps even proposing corrections. Furthermore, if the programmer accepts a change regarding a particular item, the editor can apply a similar change throughout.

This functionality would form the basis for a *maintainer's assitant,* an intelligent program that helps a maintainer carry out changes. The programmer would do the creative work of initiating modifications, whereas the machine would perform the task of bringing the system into a consistent state. This latter task is tedious and error prone for humans, and therefore a prime candidate for automation.

7. CONCLUSIONS

The smart recompilation mechanism described here eliminates most redundant compilations. It is simple and efficient, and the potential time savings in large systems are significant. The mechanism is based on change analysis, which can be added with modest effort to existing compilers, since almost all of the data structures are already present, and syntactic and semantic analyses can be reused. The mechanism can be extended for languages with overloading and with facilities that help programmers bring a system up to date.

ACKNOWLEDGMENTS

Many thanks go to Mark Baker, who built the prototype and performed the measurements. He cheerfully implemented numerous versions until I finally got the algorithm right. Insightful comments by Ellen Borrison, John Nestor, Erhard Ploedereder, and a reviewer greatly improved the paper.

REFERENCES

1. *Ada Programming Language, Military Standard.* J. D. Ichbiah, United States Department of Defense, 1983.
2. BAKER, T. P. A one-pass algorithm for overload resolution in Ada. *ACM Trans. Program. Lang. Syst. 4,* 4 (Oct. 1982), 601-614.
3. BIRTWISTLE, G., ENDERIN, L., OHLIN, M., AND PALME, J. *DECsystem-10 Simula Language Handbook Part* 1. C8398, Swedish National Defense Research Institute, Mar. 1976.
4. DAUSMANN, M. Reducing recompilation costs for software systems in Ada. In *System Implementation Languages: Experience and Assessment, Proceedings of the IFIP WG2.4 Conference* (Canterbury, UK, 1984) North-Holland, Amsterdam.
5. FELDMAN, S. Make—A program for maintaining computer programs. *Softw. Pract. Exper. 9,* 3 (Mar. 1979), 255-265.
6. LAUER, H. C. AND SATTERTHWAITE, E. H. The impact of Mesa on system design. In *Proceedings of the 4th International Conference on Software Engineering* (Sept. 1979), ACM, IEEE, ERO, GI, 174-182.
7. LEBLANG, D. B., AND CHASE, R. P. Computer-aided software engineering in a distributed workstation environment. *SIGPLAN Not. 19,* 5 (May 1984), 104-112.
8. MITCHELL, J. G., MAYBURY, W., AND SWEET, R. *Mesa Language Manual.* Tech. Rep. Xerox Palo Alto Research Center, Feb. 1978.
9. ROCHKIND, M. J. The source code control system. *IEEE Trans. Softw. Eng. SE-1,* 4 (Dec. 1975), 364-370.
10. SWEET, R. E. The Mesa programming environment. *ACM SIGPLAN Not. 20,* 7 (July 1985), 216-229.

11. TICHY, W. F. Software development control based on module interconnection. In *Software Development Environments*, A. I. Wasserman, Ed., IEEE Computer Society Press, 1981, 272–284. Also published in *Proceedings of the 4th International Conference on Software Engineering.* (Sept. 1979), IEEE, New York.
12. TICHY, W. F. A data model for programming support environments and its application. In *Automated Tools for Information System Design and Development*, H-J. Schneider and A. I. Wasserman, Eds., North-Holland, Amsterdam, 1982.
13. TICHY, W. F. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984), 309–321.
14. TICHY, W. F. RCS—A system for version control. *Softw. Pract. Exper. 15*, 7 (July 1985), 637–654.
15. WIRTH, N. *Programming in Modula-2.* Springer-Verlag, New York, 1985.

Received February 1985; revised November 1985; accepted December 1985