Smart Recompilation

Walter F. Tichy

Mark C. Baker

Report Number:
84-498
Smart Recompilation

Walter F. Tichy
Purdue University
Department of Computer Sciences
West Lafayette, IN 47907

Mark C. Baker
AT&T Technologies, Inc.
2600 Warrenville Road
Lisle, IL 60532

ABSTRACT

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 definitions, 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 actually affect only a small fraction of the overall system.

This paper presents an efficient method for significantly reducing the set of modules that must be recompiled after a change. The method is based on reference sets and the isolation of differences. The cost of determining whether recompilation is necessary is negligible compared to the cost of compilation. The method is easily added to existing compilers, and can be extended to provide guidance to programmers if the change requires software updates.

CSD-TR 498

8 November 1984

1. Introduction

For performing type checking across module boundaries, modern compilers use contexts. 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 data base, and is read in by the compiler during processing of a compilation unit. Examples of manually created contexts are Ada package specifications, Mesa definitions modules, 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 enclosing block generates a context which specifies the items which the inner block may reference. When the inner block is processed, the compiler initializes its symbol table by reading the context it had produced earlier. Ada subunits and Simula classes can be implemented with automatic contexts.

Contexts have three major uses. First, contexts are effective for implanting common declarations into several 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 with an unfamiliar system is more
prone to error than working with one's own system. For this reason, type checking of interfaces is more likely to detect errors than intra-module 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 Reference 4.

With current compiler technology, contexts have a serious disadvantage. 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 developer teams while everything is periodically recompiled from scratch. High compilation costs also tend to convolute system structure, because programmers try 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 was intended to provide. The reason is that the usage relation among contexts and compilation units is on too coarse a level. 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.
2. Overview of Smart Recompilation

Consider a compilation unit and a set of contexts. We say that a compilation unit "depends" on a context if it references any declaration defined in that context. The conventional recompilation rule is as follows.

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.

Rule (2) guarantees that any context modification propagates into the dependent units. It also has the effect of ensuring that the change has not introduced any syntactic or semantic errors. The MAKE program\(^6\) implements these rules. The Ada and Mesa language manuals\(^1,2\) prescribe similar rules, with the additional aspect that each context is a compilation unit in its own right.

Compilations triggered by rule (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 corresponding 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 with respect to a context consists of those items that were defined in the context, but referenced elsewhere (in the compilation unit or in another context). If the intersection of change set and corresponding reference set is empty, then the compilation unit need not be reprocessed.

The example below illustrates. File \texttt{f.ext} is a context containing declarations for a hypothetical traffic light control program. File \texttt{progp} is a compilation unit dependent upon context \texttt{f.ext}. All examples are formulated in Berkeley Pascal, which is equipped for separate compilation. The directive \#\texttt{include} instructs the compiler to read in a context.
Assume the following sequence of events. File `prog.p` is compiled, which produces an object module and a reference set with respect to `f.cxt`. The reference set is `{NumOfStreets, NumOfAvenues, FourWayIntersect, TrafficLights}`. Note that in order to determine this set, one needs to compute 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.

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 set is not empty, and the smart compiler therefore reprocesses `prog.p`.

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 erroneous. Second, a context may refer to
undefined identifiers, but supply no indication where the declaration can be found. For example, it is legal to split \texttt{f.ext} into two files and include them both in \texttt{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 undeclared item cannot be checked by analyzing the context alone. The following section shows how to eliminate these problems.

3. The Smart Recompilation Mechanism

This section states precisely how smart recompilation works. Initially, overloading of identifiers is not permitted. Thus, each occurrence of an identifier has exactly one meaning. The extension for overloading is discussed in Section 5.

3.1. The Multi-Version Model

The following simple model clarifies what changing a context and comparing versions of contexts means. We assume that "changing" a context or a compilation unit does not really change it; instead, a new one is created. Thus, contexts and compilation units are immutable. Furthermore, contexts or compilation units derived from the same initial context or compilation unit via editing are collected into sets called \textit{revision groups}. Members of a revision group are simply called \textit{revisions}. Revisions within a group are distinguished by \textit{revision numbers} unique within the group.

We assume that the program database can store arbitrarily many revisions per group. By saving only the differences, the space requirements for multiple revisions are modest, as has been demonstrated by a number of systems\cite{7,8,9}.

We postulate the presence of a configuration management program that composes configurations. The configuration manager decides which revision of a group to pass to the compiler for processing. When a compiler encounters a directive to read in a context, the directive will usually not specify the revision number of the context. In that case, the configuration manager decides which revision from the given group to select. A selection rule useful during software development is to always choose the newest revision. Additional rules are discussed in Reference 10.
3.2. Dependencies among Declarations

We assume that each declaration introduces an identifier naming the declaration. A reference to a declaration is simply the occurrence of its identifier in another declaration. We define a dependency relation among declarations as follows.

Definition:

Declaration $A$ depends on declaration $B$ iff $A$ references $B$.

For example, if procedure $P$ assigns to variable $V$, $V$ is of type $S$, and $S$ is a record type with one field of type $T$, then $P$ depends on $V$, which depends on $S$, which depends on $T$. Transitive dependencies of declarations are important for determining the effect of changes.

A declaration that is referenced, but not defined in a context is called "free" in that context. There are two ways of supplying definitions for free declarations. The first one is if the context specifies directly which other contexts it needs. This approach is taken in Ada. Alternatively, a context may inherit extra declarations from the compilation unit or other contexts in which it is used. Include-files typically allow both alternatives. The problem with inherited declarations is that it is impossible to compute the transitive closure of dependencies by starting a search in the context with the free declarations. The complete transitive closure is only available when all contexts are pulled together during compilation. The mechanism discussed here permits inherited declarations.

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

Assumption:

Code generation attributes for some declaration $D$ appearing in a context may only be derived from $D$ itself and from code generation attributes of those declarations that $D$ depends on transitively.

As an example, the address of a variable introduced by a context cannot be assigned by the compiler, since that address may depend on the size and number of potentially unrelated variables. The address must either be determined by the loader, or must be made explicit. The latter choice is used in automatically
generated contexts. Note that the assumption applies only to declarations appearing in contexts. Hidden dependencies among declarations appearing in compilation units are allowed. Since any change of a compilation unit requires reprocessing, hidden dependencies are automatically re-analyzed. Alternatively, hidden dependencies could be made explicit.

3.3. Problem Statement

Given a compilation unit \( M_0 \) and contexts \( M_1, \ldots, M_n \), where each \( M_i \) is a revision with number \( r_i \) in revision group \( M_i (0 \leq i \leq n) \). Assume that the configuration consisting of \( M_0, \ldots, M_n \) was compiled successfully. The compilation resulted in an object module with an associated history attribute containing the following sets.

\( \text{DECL}_i \):

The identifiers of declarations defined in \( M_i \) (0\( \leq i \leq n \));

\( \text{REF}_i \):

The identifiers of declarations defined in context \( M_i \), and referenced transitively in some other context or compilation unit \( M_j \) (\( i \neq j, 1 \leq i \leq n, 0 \leq j \leq n \)).

Given a new context revision, \( M_\ast \), (1\( \leq x \leq n \)), inspect only \( M_x, M_\ast \) and the sets \( \text{DECL}_x, \text{REF}_x \) to determine the following:

1) Is the new configuration \( M_0, \ldots, M_x, \ldots, M_n \) syntactically and semantically correct?

2) Are the object modules generated for the two configurations identical?

3.4. Solution (no overloading)

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

Change Analysis:

Step 1:

Analyze \( M_x \) syntactically and semantically. Treat all occurrences of free declarations as legal. If there are any other errors, then the new configuration
is erroneous.

Step 2:

Compare $M^*_x$ and $M^*_{x'}$ and determine the following sets.

$FREE^*_{x'}$:

The identifiers of free declarations in $M^*_x$;

$FREEREF^*_{x'}$:

The identifiers of declarations that reference elements of $FREE^*_{x'}$;

$ADD^*_{x'}-\hat{r}_x$:

The identifiers of declarations defined in $M^*_x$ but not in $M^*_{x'}$; this set includes declarations that are free in $M^*_x$ but defined in $M^*_{x'}$;

$DEL^*_{x'}-\hat{r}_x$:

The identifiers of declarations defined in $M^*_x$ but not in $M^*_{x}$; this set includes declarations that are defined in $M^*_x$ but free in $M^*_{x}$;

$MOD^*_{x'}-\hat{r}_x$:

The identifiers of declarations that are defined in both $M^*_x$ and $M^*_{x'}$, but differ.

Step 3:

If $FREEREF^*_{x'} \cap (MOD^*_{x'}-\hat{r}_x \cup ADD^*_{x'}-\hat{r}_x) \neq \emptyset$, then a free declaration is referenced in an added or modified declaration, and compilation is necessary to check legality.

Step 4:

If $\bigcup_{l=0}^{\infty} DECL^*_l \cap ADD^*_{x'}-\hat{r}_x \neq \emptyset$, then the change introduced a declaration that conflicts with an existing one and the new configuration is erroneous.

Step 5:

If $DEL^*_{x'}-\hat{r}_x \cap REF^*_{x'} \neq \emptyset$, then the change removed a declaration which was referenced externally, and the new configuration is erroneous.

Step 6:

If $DEL^*_{x'}-\hat{r}_x \cap FREE^*_{x} \neq \emptyset$, then the change removed a declaration which was referenced internally, and the new configuration is erroneous.
Step 7:

If \( \text{MOD}_k \cap \text{REF}_k \neq \emptyset \), then a referenced declaration changed and recompilation is necessary to check legality, and, if legal, to produce new object code.

Step 8:

Otherwise, the new configuration is syntactically and semantically correct, and the object code and history attribute that the compiler would generate would be identical to the one for the old configuration. Thus, no recompilation is necessary.

End Change Analysis

Step 1 checks that the new context is legal. Free declarations present a problem, since their use cannot be checked locally for legality. However, an earlier version of the context was compiled successfully. Thus, if a free declaration is used in exactly the same way in both the old and new contexts, then the use of that free declaration is correct in BOTH revisions. Otherwise, only a recompilation can check legality. Step 3 implements this analysis.

Step 4 assures that any new declarations do not interfere with existing ones. This check must be relaxed if overloading is permitted. Step 5 prints an error message if a deleted declaration is still in use in another context or compilation unit. Step 6 performs the same check for internal references. Since step 1 treats all free declarations as legal, step 6 is required. Step 7 causes a recompilation if a referenced declaration was modified. If none of these steps detects an error or triggers recompilation, compiling the new configuration produces an object module identical to the old one. The reason is that all added, deleted, or modified declarations are not referenced, and unreferenced declarations do not affect code generation. Thus, no compilation is necessary; only the history attribute should be extended to reflect the fact that the same code module can be generated from an additional configuration.

3.5. Examples

Checking legality of a new configuration is subtle. For instance, assume context revision \( M^{1d}_1 \) defines a type \( t_1 \), and context revision \( M^{2d}_2 \) declares a type \( t_2 \), where \( t_2 \) depends on \( t_1 \). Assume furthermore that neither \( t_1 \) nor \( t_2 \) are referenced anywhere else. Now suppose a new revision, context \( M^{1w}_1 \), is substituted into the configuration. If the new context changes \( t_1 \), the declaration of \( t_2 \) must be checked for legality, because the change of \( t_1 \) is unpredictable. Recall that \( t_2 \) is referenced nowhere.
Fortunately, during the compilation of the old configuration, \( t1 \) was placed into the reference set \( \text{REF}^{\text{old}}_1 \). Thus, step 7 of the decision procedure above will find that \( \text{MOD}^{\text{old}}_1 \cap \text{REF}^{\text{old}}_1 \) is nonempty, and therefore trigger a recompilation. If \( t1 \) and \( t2 \) had both been in the same context, say \( M^{\text{old}}_1 \), \( \text{REF}^{\text{old}}_1 \) would not contain \( t1 \) and no recompilation would be triggered, because local analysis of \( M^{\text{new}}_1 \) (step 1) is enough to check legality.

If, on the other hand, \( t1 \) and \( t2 \) are in separate contexts and a new revision of \( M^{\text{old}}_2 \) is created, a careful analysis of \( t2 \) is necessary. If \( t2 \) is unchanged, then step 3 will not trigger a recompilation, because a compilation of \( t2 \) was legal in the old configuration. This step assumes that \( t1 \) is unchanged. If, however, \( t1 \) changed also, its context will trigger a recompilation as described above, independent of changes to \( t2 \).

The final two cases apply when a declaration changes from free to defined and vice-versa. Assume \( t1 \) is free in \( M^{\text{old}}_2 \), but is defined in \( M^{\text{new}}_2 \) (and, as before, in \( M^{\text{old}}_1 \)). In this situation, \( \text{ADD}^{\text{new}}_2 \) contains \( t1 \) and step 4 triggers an error message because of multiple declaration of \( t1 \). For the opposite case, assume the declaration \( t1 \) occurs in \( M^{\text{old}}_1 \), but not in \( M^{\text{new}}_1 \). In that case, step 5 results in an error message, because a used declaration was removed. An additional error message may be caused by internal use. Assume \( t1 \) is also used in \( M^{\text{new}}_1 \), in the same context from which \( t1 \) was deleted. Step 6 causes recompilation in this case.

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 include 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 declarations. In Figure 2, revision numbers, shown as superscripts in the preceding discussion, are attached to the file name, with a colon as separator.
File f1.cxt:1:
  const
  NumOfStreets = 40;
  NumOfAvenues = 20;

File f2.cxt:1:
  #include "f1.cxt"
  type
  TrafficLights = ( red, amber, green );
  MorningRush = 7 .. 9;
  EveningRush = 16 .. 18;

File f3.cxt:1:
  type
  FourWayIntersect = record
    S, N, E, W : TrafficLights
  end;

File prog.p:1:
  #include "f2.cxt"
  #include "f3.cxt"
  var
  Grid : array[1..NumOfStreets, 1..NumOfAvenues] of FourWayIntersect;

  ... 
  Grid[i,j].S := red;

Figure 2: Compilation unit prog.p, revision 1, and three contexts.

The history attribute generated by compiling prog.p follows. The reference set and the declarations set of a context can be represented overlapped, because the former is a subset of the latter. We use the vertical bar (|) to separate the reference set from the rest of the declarations.

< f1.cxt:1> NumOfStreets, NumOfAvenues |
< f2.cxt:1> TrafficLights | MorningRush, EveningRush
< f3.cxt:1> FourWayIntersect |
< prog.p:1> Grid |

Figure 3: History attribute generated for prog.p:1
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.

3.6. Practical Considerations for Change Analysis

When two revisions of a context are compared during change analysis, comments, the textual layout of declarations, and other syntactic variations should have no effect. Comparing the 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.

So far, we have omitted treatment of the directives for context inclusion. The problem is that these directives are embedded in the program text, and can be changed freely from one revision to the next. Any such change can totally alter the composition of a configuration, and make the change analysis discussed above worthless. To detect this situation, we need to add the following rules.

(a) In a context, directives for context inclusion must appear before any declarations.

(b) If two revisions of a context are compared during change analysis and their context inclusion directives are not identical, then recompilation is necessary.

Rule (a) makes it much easier to deal with inherited declarations and their effects on semantic correctness of the context. The rule does not represent a serious restriction, since a context that has an inclusion directive in the middle can always be split into two contexts. Rule (2) is rather conservative. It could be refined to allow addition of include directives if no multiple declarations arise. However, change of inclusion directives in contexts is relatively rare, and the potential saving in compilation time provided by additional analysis is marginal.

Note that the rules above do not apply to the compilation unit itself. When the compilation unit changes, it must be recompiled anyhow, at which time legality of the entire configuration will be checked.
3.7. Putting it all together

The configuration management program maintains a pool of object modules which were compiled previously. Whenever the object module of a compilation unit is requested, the configuration manager checks whether suitable ones already exist. If none exist that match the compilation unit in both name and revision number, recompilation is necessary. Otherwise, saving the recompilation may be possible. First, the configuration manager inspects the history attribute of the candidate object modules to determine which revisions of which contexts were used to generate them. If an exact match with the desired configuration is found, the object module can be reused as is. Otherwise, change analysis of the old and the new configurations is necessary. For each pair of corresponding context revisions that differ, change analysis is performed. If this analysis finds that none of the context changes affect code generation, the existing object module can be reused. Its history attribute should be extended to reflect the compatibility with the new set of contexts.

4. Prototype

We implemented a prototype by modifying the Berkeley Pascal Compiler, pc, running on the UNIX* operating system. Configuration management was provided by MAKE and RCS. RCS, short for Revision Control System, manages revision groups. It conserves space by storing only deltas. 9

4.1. Implementation

Adding the generation of the reference sets to the Pascal compiler was straightforward. We expanded each item in the symbol table 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, the history attribute is generated by scanning the symbol table for each context, and writing the sets DECL and REF of each context into a file.

* UNIX is a trademark of AT&T.
The change analysis is performed in two separate phases. The program cdiff (short for Context DIFFerence) implements the first phase. It takes a pair of context revisions, and first tests whether the inclusion directives are at the beginning of the context, and identical (rules (a) and (b) of the previous Section). Cd1iff then performs steps 1, 2, 3, and 6, of the change analysis, and produces the sets ADD, DEL, and MOD as output. The union of these sets is called the change set. The change set is needed for steps 4, 5, and 7 later. Essentially, these steps 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 rules (a) or (b), or steps 1, 3, or 6 detect errors, no change set is produced, forcing a recompilation if the user wants to have more detailed error messages.

Cd1iff was easy to build. It is essentially the declaration parser of the Pascal compiler. It reads in two contexts, say $C^{ol^d}$ and $C^{new}$, builds up a symbol table for both, and then compares individual entries. Each entry in the symbol tables is essentially the abstract syntax tree of a declaration. To produce the change set, the symbol table for context $C^{old}$ is traversed. For each identifier declared in $C^{old}$, the corresponding identifier is located in the symbol table for context $C^{new}$. If the two declarations are not identical, then the identifier is included in the set $MOD_{old-new}$; if the identifier is not declared in $C^{new}$, it is added to the set $DEL_{old-new}$. In order to detect declarations that have been added, identifiers are marked as they are looked up in the symbol table for $C^{new}$. Any entries still unmarked in $C^{new}$ after the comparison is completed are part of the set $ADD_{old-new}$. Hashing is used to look 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. We performed measurements on about 20 files and 3 contexts of various sizes on a VAX/780 running the Berkeley UNIX system 4.2. We found that saving a single compilation more than amortizes the cost of the extra analysis; any additionally recompilation that is suppressed constitutes a net saving.

For evaluating the potential savings achievable with spec, 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 .1 seconds and an average compilation time of about 20 seconds per module, it follows that writing the history attribute takes less than 1% 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 probably be reduced even more 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, to be intersected with the history attributes of many object modules.

Finally, computing the intersections is fast. For non-empty change sets, an average time was about 2s (less if the change set was empty). This is a mere 1% of compilation cost. Stated another way, determining whether recompilation is necessary is two orders of magnitude faster than compiling.

In summary, the only non-negligible 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. One should also consider that the Berkeley Pascal compiler is already reasonably fast. Highly optimizing compilers or compilers for complicated languages are much slower, and even greater savings are possible.
5. Extensions

The smart recompilation mechanism can be extended for languages permitting overloading. With overloading, certain redeclarations are allowed, provided the disambiguation rules can distinguish them. Suppose a context change overloads an identifier. During change analysis, step 4 detects the redeclaration. At this point, the configuration is not necessarily erroneous. Instead of printing an error message, recompilation is started. Its purpose is to check the legality of the overloading, and whether all references to the overloaded identifier can be properly disambiguated. Similarly, if an overloading declaration is deleted, step 5 does not print an error message. Instead, step 5 triggers a recompilation to determine whether all uses of the overloaded identifier refer to the remaining declaration(s).

The mechanism described here does not eliminate all redundant compilations. Several improvements are possible. For example, the handling of free declarations could be more sophisticated. It is usually possible to derive some information about a free declaration 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 (step 3). 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. However, it is unclear whether this approach will be less expensive than a new compilation.

Dausmann goes a step further and proposes to use attribute dependencies, rather than declaration dependencies. Attribute dependencies record which attributes of declarations were used during code generation. For instance, assume a compilation unit uses only field offsets of a record type, but never the size attribute. Then no recompilation is necessary if a new field is added to the record type. However, we estimate that the additional gain in speed with this method is small. The cost of updating a data base of attribute dependencies may well overwhelm that gain. Dausmann's proposal also seems to ignore that suppressed recompilations may mask errors.

Another extension helps programmers update modules after changes. If a context 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 a parameter type changes, then an adaptation of the using compilation units is required. A 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 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. Second, if the change is to be carried out, information about the change can be passed to an editor, which steps the programmer through the discrepancies, displaying the old and new revisions 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 has only been feasible with language oriented editors. This paper demonstrates that change analysis can provide many of the same support functions in traditional, compiler-based environments.

6. Conclusions

The 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 analysis can be reused. The mechanism can be extended with facilities that help programmers in bringing a system up to date.

Acknowledgements: We are grateful to Peter Feiler, Tim Korb, and Tom Murtagh, whose comments greatly improved this paper.

References


