Smarter Recompilation

ROBERT W. SCHWANKE
Siemens Research and Technology Laboratories
and
GAIL E. KAISER
Columbia University

Tichy's Smart Recompilation method can be made smarter by permitting benign type inconsistencies between separately compiled modules. This enhanced method helps the programmer to make far-reaching changes in small, manageable steps.

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 Works and Phrases: Intelligent software tools, overloading, separate compilation, type checking

1. INTRODUCTION

Tichy has described a "smart recompilation" algorithm that minimizes the number of recompilation steps necessary to restore consistency after a change to a source file [3, 5]. His algorithm recompiles a module only if its own implementation has changed or if it references a symbol defined elsewhere whose definition has changed. Tichy's implementation is very efficient, producing a net savings in processing time if at least one unnecessary recompilation is avoided.

However, we believe that Tichy's definition of consistency is too strict. There are many real-life situations in which a programmer would be willing to sacrifice compilation consistency to reduce turn-around time, if he could do so without introducing interface errors. We therefore propose a more relaxed definition of consistency, which reduces the need for recompilation even further. Our

Authors' addresses: R. W. Schwanke, Siemens Research and Technology Laboratories, 105 College Road East, Princeton, NJ 08540; G. E. Kaiser, Department of Computer Science, Columbia University, New York, NY 10027.

```
MessageQueue.spec:                           MessageQueue.body:
  type Message = string                        procedure Send is ...
  procedure Send (M:Message)                   procedure Empty is ...
  procedure Empty returns boolean

Sender.body:                                 Filer.body
  with MessageQueue                            with MessageQueue
  procedure Submit is                          var messages: array of Message
    var M: Message                             procedure Status is
    call Send(M)                                 if Empty then ...
```

Fig. 1. Part of a message system.

algorithm based on this definition can be implemented by extending Tichy's strategies, leading to comparable time savings. Perhaps more importantly, our relaxed definition helps a programmer to introduce a far-reaching change in manageable steps, thus improving productivity.

2. CONSISTENCY

Tichy's method defines consistency in system-wide, source code terms. The current source code modules provide exactly one definition for each source code symbol. Every compiled module must be consistent with those definitions. This implies that if a single, widely-used type definition is changed, then every module that uses the type must be recompiled.

We define consistency less strongly, by noticing that widely used symbols are often used independently in different parts of a system. Consider a generic hash table package written in Ada®. It might be used in one part of a compiler to store identifiers and in another part to detect common subexpressions. Conceptually, we partition the system according to these independent uses, and require consistency only within a partition.

More precisely, two separately compiled modules have a *link-time interface* consisting of the identifiers defined in one object module and referenced in the other. Each module contains object code that defines or uses these identifiers. That object code is derived from a set of source code symbols. The two modules are consistent with each other if they agree upon the definitions of these symbols. Consider the three modules given in Figure 1: *MessageQueue*, *Sender*, and *Filer*. *MessageQueue* defines a type *Message* and a procedure *Send*(*M*: *Message*), which is called from the *Sender* module. The *Filer* module contains an array of *Messages*, and also calls the *Empty* procedure of the *MessageQueue* module. The link-time interface between *MessageQueue* and *Sender* contains the identifier *Send*, which is defined by the source code symbols *Send* and *Message*. The link-time interface between *Filer* and *MessageQueue* contains the identifier *Empty*, defined by the corresponding source code symbol.

A system is *pairwise consistent* if every pair of modules has been compiled using equivalent definitions of the source code symbols that affect the link-time interface between them. If a system is pairwise consistent, then every procedure

® Ada is a registered trademark of the United States Government (Ada Joint Program Office) call and every variable reference between separately compiled modules is type-safe. This degree of consistency is sufficient to permit most kinds of debugging; a simple exception is when two modules communicate through an ASCII stream file interface.

3. A SMARTER RECOMPILATION ALGORITHM

Tichy's algorithm uses conventional symbol table information collected during compilation to determine whether symbols defined in one file are referenced in another file. This information is saved between compilations. When a modified file is recompiled, his algorithm notes for each symbol definition whether it has been added, modified, or deleted. Other files are recompiled only if they depend on the added, modified, or deleted symbols.

Our extension to Tichy's algorithm requires only pairwise consistency. We assume that a programmer selects a set of files that must be recompiled, based on his testing strategy or other factors. We call this the "test set." Our algorithm compiles these files, then invokes Tichy's algorithm to identify other modules that are candidates for recompilation. For each candidate, it analyzes the link-time interface between the candidate and the test set. If that interface is affected by any symbol that has changed, the candidate is recompiled and added to the test set. When no more candidates need to be recompiled, the new object files from the test set can be safely linked with the old object files from the other modules to form a functional system.

3.1 Link-Time Interface Specifications

An *interface error* exists when two distinct versions of a symbol affect the link-time interface between two object code files. The interface between two object files involves the procedures defined by one object and called by the other, the variables declared by one object and used by the other, the types of the arguments and results of these procedures and the types of these variables. Types are often composed of other types, so all types whose definitions contribute to the interface must be considered.

To detect the pairwise inconsistencies that imply interface errors, we extend the symbol tables required by Tichy's algorithm. We add a *derived symbol* record for each symbol definition or symbol reference that must be resolved by the linker. A derived symbol *definition* represents a procedure or a variable declared inside the file and accessible from other, separately compiled files. A derived symbol *reference* represents a procedure or a variable that the file needs to access; this symbol must therefore be provided by some other object file.

For each derived symbol definition, our extended symbol table records all the source code symbols that contribute, transitively, to its interface specification. This information is produced by the compiler. For example, Figure 2 indicates that the definition of derived symbol *Send* comes from source code symbols *Send* and *Message*. For derived symbol references, such as to *Send* in *Sender* and to *Empty* in *Filer*, only the symbol name is stored; an improvement on our algorithm stores additional information (see Section 3.4).

MessageQueue.body:		
definition.	Send	depends on: procedure Send, type Message
	Empty	depends on: procedure Empty
Sender.body:		
reference.	Send	
Filer.body:		
reference.	Empty	

Fig. 2. Derived symbols.

3.2 Deciding What to Recompile

When source files change, our algorithm follows these steps.

(1) Recompile every changed file and produce its new symbol table, including derived symbols.
(2) Use Tichy's algorithm to generate the set of changed symbol definitions and to determine candidates for recompilation. Present this list of files to the programmer.
(3) The programmer selects the test set. Recompile this set and generate their new symbol tables.
(4) For each member of the test set, compare its symbol table pairwise with related files that are not yet in the test set, to detect pairwise inconsistencies. (Explained further below.)
(5) If an interface error is detected, recompile the related file and add it to the test set, to be checked under Step 4 above.

The only difficult part is the pairwise comparison. Here our algorithm takes advantage of both our derived symbols and the changed set produced by Tichy's algorithm. It detects cases where the same symbol appears as a derived symbol definition in one table and as a derived symbol reference in the other. In each such case it intersects the set of source symbols on which the derived symbol definition depends with the set of changed source symbol definitions. If the intersection is nonempty, then there is an interface error and recompilation is necessary. (If the recompilation detects errors, it may also be necessary to further modify the source code to remove the inconsistency.)

3.3 Example

Consider again the four files in Figure 1. Say the programmer modifies MessageQueue.spec to change type *Message* from "string" to "array of integer." Tichy's algorithm would recompile all three body files, because each uses type *Message*. In contrast, our algorithm tells the programmer that all three bodies reference type *Message* and asks which of them it should recompile. Say the programmer requests only MessageQueue.body. Our algorithm considers whether or not it is necessary to also recompile Sender.body and Filer.body. It recompiles Sender.body because the derived symbol *Send*, referenced in Sender.body, is defined in part by the changed type *Message*. Our algorithm does not recompile Filer.body because the derived symbol *Empty*, referenced in Filer.body, does not depend on the changed type. The old object module for Filer.body can be safely linked with the new object modules for Sender and MessageQueue, because the old and new modules never exchange values of type *Message*.

3.4 Improvements

Our algorithm assumes a baseline that is globally consistent, so interface errors can be introduced only through changed symbol definitions. This assumption is not always correct; an improvement to our algorithm solves this problem. We give each distinct version of a source file a unique version identifier. In the symbol table we mark each source symbol with the version identifier of the source file from which it came. For each derived symbol reference, we record the source symbols (with their version identifiers) used to validate the reference. Then, during pairwise comparison, we ignore the changed sets and instead compare the actual source symbol versions affecting the definition and reference. This extends nicely to handle overloaded symbols, where the version identifier for each overloaded symbol encodes both the version of the source file and the specific overloaded symbol definition within the file.

Our improved algorithm also applies to nonsequential versions of files, such as maintained by RCS [4]. Consider the case where certain source or object files in a configuration are replaced by other versions [1]. If derived symbol tables are available, our algorithm can detect whether the replacements introduce interface errors.

4. MANAGING FAR-REACHING CHANGES

When a programmer must modify a symbol definition that is used in many places in a system, he may be reluctant to edit all of the places where the symbol is used until he is sure that the new definition works for at least a few test cases. Using our algorithm, the programmer could start with only the defining module in his test set, and the algorithm would determine the minimum set of dependent modules that must be edited to test the change. Assuming that the change passes the test, the programmer could then select additional dependent modules to change and apply our algorithm again to discover other modules that must be changed at the same time. In this fashion our algorithm helps a programmer develop and install a widespread change in manageable steps.

5. IMPLEMENTATION

Our algorithm has been implemented as part of Harris Morgenstern's M.S. thesis project—Inconsistency Management System (IMS) [2]—at Columbia University. IMS is targeted for C, and supports smarter recompilation for almost all of C, including macros, unions, anonymous structure fields, and other difficult aspects of the language. Conditional compilation is not supported becuase it is fundamentally inconsistent with smarter recompilation (and smart recompilation as well). The primary limitation of IMS is that makefiles must follow a strict format, although the usual macro substitution facilities may be used. The system consists of six programs, smartee, ccom, cpp, cdiff, cppcdiff, and smartmake, which were written by modifying the corresponding Berkeley UNIX 4.2 utilities.

REFERENCES

1. LEBLANG, D. B., AND MCLEAN, G. D., JR. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large* (June 1985), GTE Laboratories, Inc., Waltham, Mass., 1985, pp. 122–127.
2. MORGENSTERN, H. M. An inconsistency management system. Master's thesis, Tech. Rep. CUCS-284-87, Dept. of Computer Science, Columbia Univ., March 1987.
3. TICHY, W. F., AND BAKER, M. C. Smart recompilation. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (POPL), (Jan. 1985), ACM, New York, 1985, pp. 236–244.
4. TICHY, W. F. RCS—A system for version control. *Softw. Pract. Exper. 15*, 7 (July 1985), 637–654.
5. TICHY, W. F. Smart recompilation. *ACM Trans. Program. Lang. Syst. 8*, 3 (July 1986), 273–291.

Received October 1986; accepted May 1987