Vol. 35 No. 9 情報処理学会論文誌 Sep. 1994

Cベースのオブジェクト指向言語における再コンパイル時間の短縮

安田 和[†] 北山文彦[†] 小野寺民也[†]

現在，C++ をはじめとする静的な型をもつコンパイル方式のオブジェクト指向言語において，クラスの仕様変更にともなって引き起こされる長時間の再コンパイルが，プログラム開発効率上の大きな問題となっている．我々はプログラム開発効率を向上させることを目的に，独自のCベースのオブジェクト指向言語 COB を開発したが，その高いモジュラリティをもつ言語仕様によって再コンパイル時間を短縮できることを示した．また，スマート・コンパイル (smart recompilation) と呼ばれる機構をCベースのオブジェクト指向言語向けのソフトウェア・ツールとして実現し，再コンパイルの量を削減することに成功した．スマート・コンパイルとは，プログラムのヘッダーファイルに変更があった際に，その変更の内容を解析し，真に影響の及ぶファイルのみに対して再コンパイルを行うものである．スマート・コンパイルは当初はこの COB のツールとして実現したが，C++ に対しても，実際に開発のプログラムからデータを収集してその有効性を調べ，COB の場合と比較した．

Reducing Recompilation Time for C-based Object-oriented Languages

Kazu Yasuda,[†] Fumihiko Kitayama[†] and Tamiya Onodera[†]

With the increasingly widespread use of statically typed object-oriented languages such as C++, the large overhead for recompiling modified programs is becoming a serious problem. To overcome this problem, we designed our own C-based object-oriented language, COB, and showed hor recompilation can be reduced by improving the modularity of a language. In addition, to control the recompilation process, we implemented a software tool, which employs a mechanism called *smart recompilation*. This mechanism examines the contents of modified header files, and determines for each implementation that depends on the header files whether recompilation is necessary. The smart recompilation tool was first implemented for COB, but we have shown that it can also reduce the recompilation time if it is used for C++. However, it is more efficient in COB, which has higher modularity.

1. はじめに

近年のオブジェクト指向言語の普及にはめざましいものがあるが，これにはコンパイル方式による実行効率のよいオブジェクト指向言語，中でも C++ の果たした役割が大きい．しかしこれらの言語によって書かれるシステムが大きくなるにしたがい，クラスのインターフェースの変更にともなう再コンパイル時間の長大化が問題となっている．

C++ をはじめ，コンパイル方式によるオブジェクト指向言語の多くでは，あるクラスのオブジェクトを利用するコード（クライアント）のコンパイルの際には，そのクラスのインターフェースの書かれたヘッダーファイルを参照する必要がある．このため，これらの言語で書かれたプログラムは，従来のプログラム言語に比して，ファイル間の依存関係が複雑になりがちであり，これが長時間の再コンパイルに結びついているといわれている．さらに実際には，言語仕様あるいはプログラム構築ツールの不備が，この長時間化に拍車をかけていることが指摘できる．

具体的には，言語によってサポートされるクラスの情報隠蔽機能が不十分であるために，ファイル間の依存関係が大きくなってしまうことや，もっとも広く使われているプログラム構築ツールである make の機能が不十分なために，必要のないファイルまで再コンパイルの対象となってしまうことがあげられる．不必要な再コンパイルを避けるために，人為的にファイルの日付を操作し，“make をだまし” た経験のあるプログ

† 日本アイ・ビー・エム(株)東京基礎研究所
IBM Research, Tokyo Research Laboratory

ラマも少なくないであろう．当然ながら，人手によって“make をだます”ことは，煩わしくかつ誤りを生じやすい．

我々は，こういった問題を解決するため，まず言語の改良の観点からプログラム開発効率をあげることを目標として，独自のCベースのオブジェクト指向言語である COB を開発したが，その言語仕様の設計の際には，再コンパイル時間の短縮，特にクラスのインターフェースに変更があった場合の再コンパイル時間を短縮することが重視された．この COB 言語の特徴について第2章で簡単に説明した後，その実現法，および COB による再コンパイル量の削減効果については第3章で，C++ の場合との差に重点をおいて紹介する．

次に，スマート・コンパイル (smart recompilation) とよばれる，クラスのインターフェースの変更の影響が及ぶ範囲をより細かく求めることによって再コンパイル量を少なくする手法を，オブジェクト指向言語に合うように拡張し，COB 用のソフトウェア・ツールとして実現した．このツールの機構を第4章で説明した後，その効果の評価を第5章で行う．この評価にあたっては，COB および C++ による実際のプログラム開発におけるファイル変更の履歴を収集したデータに基づき，COB についてはツールを適用した結果を，C++ についてはシミュレーションによる評価の結果を使用して比較・検討した．

2. COB

数多くのオブジェクト指向言語の中で，C++ が成功した大きな要因に，実行効率の重視という点があげられる．C++ においては，クラスは型として取り扱われ，そのデータ構造，例えばインスタンスの大きさやインスタンス内でのインスタンス変数へのオフセット値は，プライベートなメンバーに関するものも含めて，コンパイル時点で静的に決まっている．このことが C++ の実行効率のよいコード生成を可能にしている．

しかし一方で，C++ のような静的データ結合を実現するためには，クラスのインターフェースにプライベートなメンバーまで書かねばならない．このことは，ファイル間の依存関係を増大させ，プライベートな部分の変更が大量の再コンパイルを引き起こすことにつながりプログラムの開発効率を低下させてしまう．

我々はこういった点を念頭において，完成したプログラムの実行効率を落とすことなくプログラムの開発効率を向上させることを目的とする独自のCベースのオブジェクト指向言語 COB を開発した[7]．

COB は，C言語の拡張であること，強力な型をもち，クラスが型として扱われることなど，C++ と多くの類似点を持っている．インスタンス変数およびインスタンス関数へのアクセスやインスタンスの生成についても，C++ と類似した方法をとっている．

一方で C++ との相違点として，クラスの情報隠蔽機能を高め，ファイル間の依存関係をできるだけ少なくすることを重視した．具体的には，クラスのインターフェースとインプリメンテーションとの完全な分離という点が挙げられる．クラスのインターフェースにはパブリックな情報だけが書かれ，プライベートに関する情報は一切現れない．一方で，プライベートな情報は，クラスのインプリメンテーションとして扱われ，そのクラスのメンバー関数の実行コードとともにひとまとめにインプリメンテーション・ファイルに書かれる．

図1に COB のプログラムの例を示す．< は継承を表すが，継承に関してもプライベートなものはインプリメンテーションに書かれ，外部からは見えない．common: はそれ以降で宣言されたメンバーが，C++ の用語でいう static なメンバーであることを示す．

この例で，クラスCはパブリック・スーパークラスとしてクラスXをもちプライベート・スーパークラスとしてクラスYをもつ．クラスCのクライアントは，クラスCおよびXのインターフェースに依存しているが，これらのクラスのプライベートな変更からは何ら影響を受けない．したがってクラスYのインターフェースにも依存しないことになる．

その他の言語の特徴として，クラスのインターフェースの自動インクルード機能があげられる．C++ と同様に COB においても，クラスのインターフェースはヘッダーファイルに書かれるのが普通であり，そのクラスのインプリメンテーションやクライアントのコードを処理するときには，そのヘッダーファイルを先に読み込まなければならないが，COB コンパイラは，未知のクラスに出会うと，そのクラスのインターフェースの書かれたヘッダーファイルを自動的に探して読み込むようになっている．つまりクラスのインターフェースに関する限り，#include をプログラム中で指定する必要はない．このことは，不必要なクラス

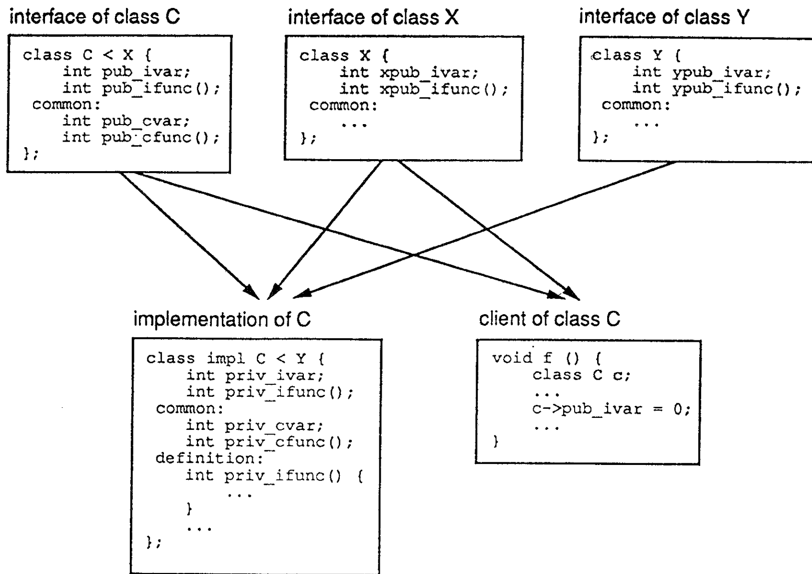


図 1 COB のプログラム例
Fig. 1 Example of COB language.

のインターフェースを #include してしまう心配をなくし，間接的ながらもファイル間の依存関係を減らすことに貢献している．

3. 変数オフセット方式

3.1 COB の処理系

COB は現在 C コンパイラへのトランスレータとして実現されている．COB はクラスのインプリメンテーション・ファイルを単位とする分割コンパイルをサポートしている．

クラスおよびそのインスタンスの実現は標準的な枠組みでなされている．COB のインスタンス関数はすべて C＋＋ でいう virtual 関数であり，実際に呼び出される関数は，各クラスに1個ずつ用意されたメソッド・テーブルを通じて，C＋＋ の virtual 関数の場合と同様，実行時に決定される．メソッド・テーブルは，このクラスのすべてのインスタンス関数へのポインタの配列である．

一方，それぞれのクラスのインスタンスは，自分自身およびそのすべてのスーパークラスで定義されたインスタンス変数を保有する1個のレコードであり，その先頭には当該クラスのメソッド・テーブルを指すポインタが置かれている．例として，図1で定義されたクラスCのインスタンスの構造を**図 2** に示す．

こういった枠組みにおいて，あるクラスを使用する

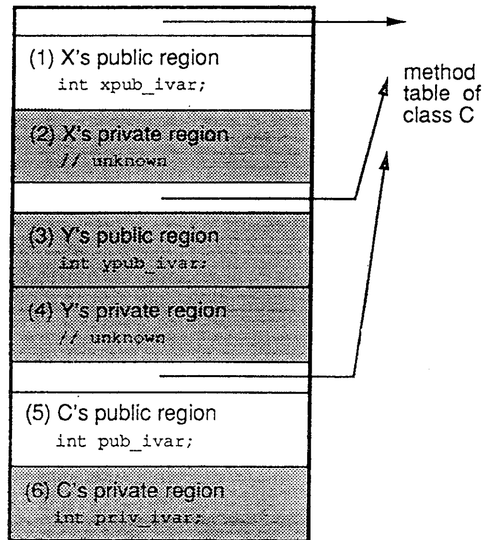


図 2 COB インスタンスの構造例
Fig. 2 Data structure of a COB instance.

コードを生成するためには，そのクラスのインスタンスおよびメソッド・テーブルのサイズやその構造（各メンバーのオフセットなど）について十分な情報が必要である．しかしながら，COB においては，クラスのプライベートな構成要素（メンバー，スーパークラ

ス）がインターフェースから隠されているため，これらはいずれもクライアントのコンパイル時には決定できない．

例えば，図2において網掛けになっている部分((2)(3)(4)(6))のサイズがクライアントからは見えないため，クラスCのパブリックなメンバー変数(図の(5)の部分）のインスタンス内でのオフセットも，クライアントのコンパイル時にはわからない．例えば，図1の関数f()のコンパイル時には，そこでアクセスしている変数 pub_ivar のオフセットは決定できないわけである．ちなみに C++ ではこれらの値はすべてコンパイル時に定数となっている．

この問題を解決するために，COB コンパイラは，これらメソッド・テーブルやインスタンスのサイズ，メンバーのオフセットを参照するのに，一群の変数（アクセス変数）を用いてコードを生成するようにした．これらの変数は，実行時にその値が必要となる前に，各クラスのインプリメンテーションごとに生成された初期化ルーチンを呼び出すことによって，値を決定される．

例えば，**図3**のように，クラスCのインスタンスのサイズ _size_C_instance は，その初期化ルーチン init_C() の中で，そのスーパークラスXおよびYの初期化ルーチンを呼び出した後に決められ，new_C() におけるクラスCのインスタンス生成時にはこの変数が使われる．

C++ のようにコンパイル時など実行時に全データ構造が定数によって決められる方式（定数オフセット方式，static binding）に対して，このようにデータ構造を変数で表し，その値を実行時に決定する方式を本論文では変数オフセット方式（late binding）と呼ぶ．

3.2 性能評価

COB では変数オフセット方式を採用したことによって，再コンパイル時間の短縮が期待される一方，アクセス変数を使用することによって実行時の性能が低下することが予想される．そこで，COB によるコンパイル効率および生成コードの実行効率を測定した実験データを紹介する．比較のため，C++ のようにプライベートな情報をクラス・インターフェースに書いた場合のコンパイル効率，および定数によってコードを出力した場合の実行効率も，COB コンパイラでシミュレーションを行うことによって測定した．サンプルとして**表1**に示す2種類のプログラムを使用した．

コンパイル効率 **表2**は，それぞれの方式によって実行可能コードを生成するのに必要なコンパイル所要時間を示すものである．*all* とある行はすべてのファイルをコンパイルした場合であり，その他の行は，特定のクラスのプライベート・メンバーが修正された場合の再コンパイル時間である．

これらのコンパイル時間の比をとってみると，全コンパイルの場合には，変数オフセット方式の方が1割から3割ほど遅くなっているが，再コンパイルの場合は2分の1から 11 分の1の時間しかかからず，変数オフセット方式を採用することによって再コンパイル

```
// access variables for class C
int _size_C_instance = 0;
int _size_C_public = sizeof(int);
int _size_C_private = sizeof(int);
...

// initialization of class C
void init_C(void) {
    if (_size_X_instance == 0) init_X();
    if (_size_Y_instance == 0) init_Y();
    _size_C_instance = sizeof(void*)
                     + _size_X_instance
                     + _size_Y_instance
                     + _size_C_public
                     + _size_C_private;
    ....
}

// creation of C's instance
void* new_C() {
    void* ptr;
    if (_size_C_instance == 0) init_C();
    ptr = malloc(_size_C_instance);
    init_C_instance(ptr);
    return ptr;
}
```

図 3 COB における変数オフセット方式
Fig. 3 Late binding in COB.

表 1 サンプル・プログラム（1）
Table 1 Sample programs.

program	行 数	クラス数	説 明
maze	843	7	create/solve a maze
lisp	1128	45	a lisp interpreter

表 2 再コンパイル時間 (PS/2 AIX)
Table 2 Compilation time (under PS/2 AIX).

program	modified class	再コンパイル時間 (sec)	
		static binding (simulation)	late binding (COB)
maze	*all*	98.3	108.7
	class A	36.7	8.1
	class B	41.8	10.6
lisp	*all*	147.9	198.4
	class C	33.8	13.0
	class D	147.9	13.4

時間を劇的に短縮できることがわかる．この効果はプログラムが大きくなるほど顕著となり，コード・サイズが約2万行のプログラムで測定したところでは，最大 1：50 という結果が出た．

実行効率　表1のプログラムについて実行時間を測定した結果，変数オフセット方式による実行時間の増加は定数オフセット方式の時と比べ，lisp の場合で10%，maze の場合で3% 程度であった．

我々は同じサンプルを C++ でもインプリメントした．COB には，実行時の型チェック，配列の境界チェックなど C++ にはない機能があるため，この実行効率の結果を C++ の場合と直ちに比較することはできない．ただし COB には，完成したプログラムの変数オフセットを定数に置き換えることによって最適化を行う機能が付け加えられており，それによって生成されたコードからは C++ に匹敵する効率が得られた[10]．このことから，プログラム開発時には変数オフセット方式を用いたコード生成を行うことによって開発効率を高め，実行効率を重視する場合には，プログラムの開発が終了した段階で，定数オフセット方式による最適化を行う手法が，有効であると言えるだろう．

4. スマート・コンパイル

プログラム開発においては，プログラムの構成要素間の依存関係を調べ，ある構成要素に変更が与えられたときに，その影響の及ぶ範囲を特定することによって無駄な再コンパイルを抑えるツール（コンパイル機構）が不可欠である．

現在もっとも普通に使われているコンパイル機構（UNIX の make など）では，依存関係をファイル間の関係に限って調べ，ファイルの日付のみにもとづいてその変更の有無を判定するものである．そのため，変更の内容がコメントの書き換えなどで何らコードには影響がない場合ですら，無駄な再コンパイルが行われてしまうことがある．

また，こういったコンパイル機構では，あるクラスのインターフェースの書かれたヘッダーファイルに修正があった場合には，変更の内容にかかわりなく，そのクラスのクライアントすべてについて再コンパイルが行われる．しかし実際には，クラスのインターフェースの一部に変更があった場合でも，そのクラスのすべてのクライアントが，真に再コンパイルを必要としているとは限らない．例えば，特定のメンバーに修正があったからとしても，それがすべてのクライアントで使われているとは限らない．

スマート・コンパイルは，プログラム中の依存関係および変更の有無の判定を，ファイルだけではなく，そこに書かれたプログラムの内容に踏み込んで行い，ある変更の影響の及ぶ範囲を計算によって求めることによって，再コンパイルされるファイルをさらに少なくするための機構である．

我々は Tichy[9] の手法を参考にして，これをオブジェクト指向言語に向くように改良した．つまり，文献 9) が Pascal のような従来の手続き型言語を対象とし，依存関係やプログラム変更の解析を，各宣言文，例えばひとつのレコード型の全体を単位として行っていたのに対し，我々はこれを，クラスの内容まで調べ，クラスのメンバー単位での解析を行えるようにした．

一般に，コンパイル機構においては，解析の単位の粒度を小さくすることによって，ひとつの変更の影響の及ぶ範囲をより小さく限定できることになり，コンパイル量削減の効果が高まる．スマート・コンパイルはこの原理を利用したものであり，われわれの改良はその効果をさらに高めている．

図4はそれを例示するもので，この図中のプログラムにおいて，クラスBのメンバー関数 g() の型に変更があった場合，make であれば(A)(B)(C)の3つ

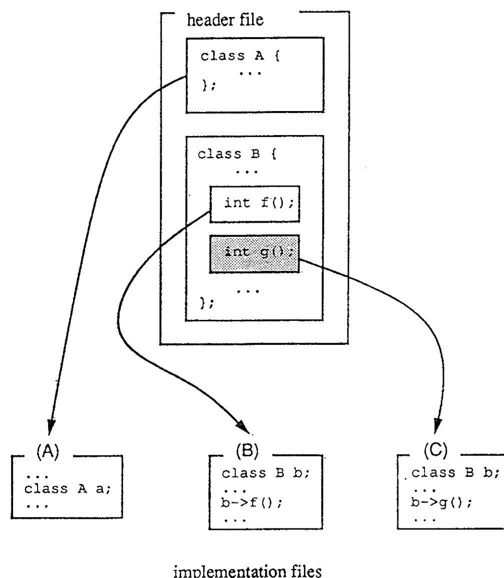


図 4　プログラム中の依存関係
Fig. 4　Dependency in program.

のインプリメンテーション・ファイルが再コンパイルの対象となるが，Tichy のようなクラス単位の解析ならば(B)(C)の2つのファイル，われわれの採用したメンバー単位の解析を行うと(C)のみが再コンパイルされることになる．

われわれの実現したスマート・コンパイルのアルゴリズムは，3つのステップからなりたっている．すなわち，あるヘッダーファイルに変更があった場合，

1. 変更されたヘッダーファイルの新旧の内容を比較し，その中で宣言されているシンボル（クラスおよびそのメンバー）について差分（diff）をとる．
2. 変更されたヘッダーファイルに依存するインプリメンテーション・ファイルの各々について，それが diff に現れるシンボルを使用しているかどうかを調べる．
3. 使用していた場合，そのインプリメンテーション・ファイルの再コンパイルを行う．

ヘッダーファイルに変更がない場合の再コンパイルは，make と同様に，変更のあったインプリメンテーション・ファイルについてのみ行われる．

スマート・コンパイルは，まず COB 用のソフトウェア・ツールとして実現された．その後，C++ に対するスマート・コンパイルの有用性を調べるために，C++ 上でのシミュレーションを行った．

4.1 COB のスマート・コンパイル

COB のスマート・コンパイルは大きく分けて，COB コンパイラ組み込みのルーチンおよびスマート・コンパイル・ドライバから成り立っている．

COB コンパイラ COB コンパイラは，スマート・コンパイルを行うために必要なプログラムに関する情報を，コンパイル時にファイルに書き込む．これらのプログラム情報は，ヘッダーファイルごとに出力される decl 情報と，インプリメンテーション・ファイルごとの use 情報との2種類に分かれる．decl 情報には，そのヘッダーファイルで宣言された各クラスのスーパークラスやメンバー，およびそれらのオフセットや型など，シンボルに関する属性情報が含まれており，use 情報にはそのインプリメンテーション・ファイルで実際に使っているシンボル名が含まれている．

ドライバ ドライバはコマンドの形で用意され，実際にスマート・コンパイルを使用するには，これに必要なソース・ファイル名を引数として与えて起動する．ドライバの動作は次のとおりである．

1. 各ソース・ファイルの日付を調べ変更の有無を判定する．
2. インプリメンテーション・ファイルに変更があった場合は，それをコンパイルする．
3. ヘッダーファイルに変更があった場合は，
   (a) 変更されたヘッダーファイルを単独でコンパイルして decl 情報を得る．
   変更されたヘッダーファイルから decl 情報を得るためには，一般にはクラスのインターフェースだけを単独でコンパイルできる必要がある．文献 9) においてはこのための専用のパーザーを用意しているが，COB では未知のクラスのインターフェースを自動的に読み込む機能があるため，コンパイラをわずかに変更するだけですんだ．
   (b) このヘッダーファイルにインターフェースが書かれている各クラスと，このヘッダーファイルに依存するインプリメンテーション・ファイルの組み合わせそれぞれに対してチェンジ・アナライザを呼ぶ．
   (c) チェンジ・アナライザが true を返すときは，COB コンパイラを起動してそのインプリメンテーション・ファイルを再コンパイルする．

ここで，チェンジ・アナライザ change_analysis とは，あるクラスXのインターフェースに変更があったときに，そのクライアントであるインプリメンテーション・ファイル（Y.cob とする）を再コンパイルする必要があるかどうか判定するルーチンであり，次の形で呼ばれる．

Bool change_analysis(X.decl, X.ndecl, Y.use)

ここで X.decl は変更前のXの decl 情報，X.ndecl は変更後のXの decl 情報，Y.use は Y.cob の use 情報である．このルーチンは，次のような一連の条件判定を行っている．細部はコンパイラのコード生成方式に依存するものであり，ここでは省略する．

1. X.decl と X.ndecl が同一であれば，false を返す．
2. X.decl と X.ndecl の間でスーパークラスに変更があった場合は，true を返す．
3. Xのメンバーが削除されていて，かつそれが Y.use の中に現れていた場合は，true を返す．

4. Xのメンバー変数の型に変更があって，かつそれがY.useの中に現れていた場合は，trueを返す．
5. Xのメンバー関数のインターフェースに変更があって，かつそれがY.useの中に現れていた場合は，trueを返す．

⋮

4.2 C++ のスマート・コンパイル

C++ でスマート・コンパイルを行う場合も，基本的な原理はCOBの場合と同じであるが，以下の点に留意する必要がある．

まず，C++ では，第3章でも述べたように，インスタンスの構造はコンパイル時点で静的に決定されている．例えばあるクラスのメンバー変数の，インスタンス内でのオフセット値なども定数で表現されている．この値は，COB では変数で表現され実行時に値を決定されるため，このクラスのスーパークラスの構造にある程度の変化があってもコードに影響が及ばないが，C++ においてはスーパークラスのインスタンスの大きさに変化があれば，このオフセット値にも変更が必要となる．つまり，ヘッダーファイルに同様の変更があった場合でも，C++ とCOBとはスマート・コンパイルの効果が異なる場合がある．

また，C++ ではinline関数として，ヘッダーファイルに実行コードを書くことができる．inline関数の目的も実行効率を高めることであるが，inline関数の存在も依存関係の複雑さを増す結果につながっている．

今回，C++ については，ヘッダーファイルのみのコンパイルをサポートするコンパイラが我々の手元になく，ヘッダーファイルの変更の解析に必要な情報をコンパイラから得ることができなかったため，シミュレーションによって，C++ にスマート・コンパイルを適用した場合の有効性を調べることにした．つまり，通常のmakeを使用して再コンパイルを行ったのち，再コンパイルされた各ファイルについて，本当に再コンパイルが必要であったかどうかの判定を行う方法をとった．この判定にあたっては，我々が開発したC++ 用のプログラム・ブラウザー[5]に組み込まれているプログラム・データベース[6]のスキーマを変更して使用し，このプログラム・データベースの問合せ言語として採用されているPrologでチェンジ・アナライザを記述した．

また前述のように，ヘッダーファイルに同様の変更があった場合の，C++ とCOBとでのスマート・コンパイルの効果の差を調べるため，それぞれの言語仕様に基づき異なった判定基準をもつチェンジ・アナライザを2通り用意し，計算することにした．

inline関数については，変更の内容の解析が困難であることから，inline関数の定義を含むファイルが変更されたときには，その中のinline関数はすべて変更されたものと見なすこととした．

5. スマート・コンパイルの効果

スマート・コンパイルを使うことによって，コンパイル量を減らすことができる場合があるのは明らかだが，スマート・コンパイルの真の効果は，これを適応した場合のプログラム開発期間中における総コンパイル時間の変化を見なければならない．

スマート・コンパイルを使用した場合の総コンパイル時間の変化を評価するためには，次のような点を考慮する必要がある．

スマート・コンパイルのオーバーヘッド　これには，
- コンパイラがスマート・コンパイル用のプログラム情報を出力することに起因するオーバーヘッドと，
- その情報をもとに再コンパイルの必要性を判定するための解析のオーバーヘッド

とが含まれる．

スマート・コンパイルの有効量　プログラムの開発の際には，ファイルの変更と再コンパイルのサイクルが繰り返し行われる．経験上，そのうちのほとんどは，ヘッダーファイルには変更のない，インプリメンテーション（ほとんどの場合そのメンバー関数）内部の変更であることが多い．全体のコンパイルの回数のうち，スマート・コンパイルが有効となるものがどの程度を占めるのか，またそれによってどれだけコンパイルすべきコードの量が減るのかは，スマート・コンパイルのトータルな有効性を左右する．

我々はまず一定の状況を設定してオーバーヘッドの測定を行った．その後，COBおよびC++ による実際のプログラム開発において，各再コンパイル時のファイルの履歴を収集し，スマート・コンパイルの有効量を実測することによって，スマート・コンパイルの効果を推測する手法をとった．その際，単純な仮定として，コンパイルするコード量とコンパイル時間，および情報出力・解析のオーバーヘッドが比例するものとした．

5.1 オーバーヘッドの測定

オーバーヘッドの測定は，スマート・コンパイルのドライバ自身のコードを使用し，RS/6000 530 (AIX 3.1.0) 上で測定した.

まずコンパイラがスマート・コンパイル用のプログラム情報 (decl 情報および use 情報) を出力することによるオーバーヘッドを**表 3** に示す. normal と smart の欄は，それぞれ通常のコンパイル時間と，情報を出力した場合のコンパイル時間であり，情報を出力することによるオーバーヘッドは5% 程度であった.

あるヘッダーファイルが変更された場合，スマート・コンパイルはそのファイルに依存するすべてのファイルに対して，再コンパイルの必要性を判定する. このとき解析を行う対象のコード量は，通常の make を使った場合に再コンパイルされるコード量に等しい. したがって判定のための解析のオーバーヘッドは，通常の make を使った場合のコンパイル時間に比例すると考えてよい. この比率は**表 4** に示すとおり，10% 程度であった.

5.2 プログラム変更の履歴

現実にプログラムの開発あるいは保守に当たる際，ヘッダー・ファイルとインプリメンテーションのファイルは，それぞれどの程度変更され，スマート・コンパイルによって減らすことのできるコンパイル量がどの程度になるかを，3種類の COB のプログラムと1種類の C++ のプログラムを用いて実測した. ただし，C++ におけるスマート・コンパイルの効果は前節で述べたように，測定データを元にシミュレーションによって求めた値であり，同様の変更を COB および C++ で行った場合の両方の数値を掲げた. 実験に用いたプログラムのおよその行数と測定期間を**表 5** に，スマート・コンパイルの効果の結果を**表 6** に示す.

表 6 中のコンパイル回数欄は，それぞれ

表 3 情報出力のオーバーヘッド
Table 3 Overhead of information generation.

ファイル数	行 数	コンパイル時間 (sec)		overhead (%)
		normal	smart	
1	310	14.8	15.6	5.4
3	519	40.1	42.2	5.2
6	718	55.5	59.0	6.3
9	1067	74.9	78.4	4.7

表 4 解析のオーバーヘッド
Table 4 Overhead of change analysis.

ファイル数	time (sec)		overhead (%)
	analysis	compilation	
3	4.6	26.8	17.2
6	6.3	55.5	11.4
9	7.4	74.9	9.9
11	7.5	75.8	9.9

表 5 サンプル・プログラム (2)
Table 5 Sample programs (2).

program	行 数	データ測定期間(日数)
A (COB)	280	2
B (COB)	6400	5
C (COB)	3332	5
D (C++)	7000	15

表 6 スマート・コンパイルの効果
Table 6 Effect of smart recompilation.

program		コンパイル回数	行数 smart	行数 make	行数比(%) (smart/make)
A (COB)	(1)	31	5262	5371	98.0
	(2)	26	3922	3922	100.0
	(3)	5	1340	1449	92.5
	(4)	4	1247	1356	92.0
B (COB)	(1)	9	8024	8851	90.7
	(2)	7	7225	7225	100.0
	(3)	2	799	1626	49.1
	(4)	1	416	1243	33.5
C (COB)	(1)	10	11496	14512	79.2
	(2)	9	9506	9506	100.0
	(3)	1	1990	5006	39.8
	(4)	1	1990	5006	39.8
D(C++*)	(1)	61	627623	807642	77.8
	(2)	48	199230	199230	100.0
	(3)	13	428393	608412	70.4
	(4)	8	297029	477048	62.3
D(COB*)	(1)	61	560836	807642	69.4
	(2)	48	199230	199230	100.0
	(3)	13	361606	608412	59.4
	(4)	8	230242	477048	48.3

コンパイル回数・行数（ヘッダーファイル展開後）の数値はいずれも
(1) すべての場合の合計 ((2)+(3) と一致),
(2) ヘッダーファイルに全く変更がなかった場合,
(3) ヘッダーファイルに変更があった場合,
(4) (3)のうち，スマート・コンパイルが有効であった場合.

また，D(C++*) および D(COB*) は，C++ プログラムのデータにもとづくシミュレーションによる結果.

（1）再コンパイルが行われた回数，
（2）（1）のうち，ヘッダーファイルに変更がなかった回数，
（3）（1）のうち，ヘッダーファイルに変更があった回数，
（4）（3）のうち，スマート・コンパイルが有効であった回数，

を表している．ヘッダーファイルが変更される回数は，再コンパイルの回数全体に対して多くないこと，また，ヘッダーファイルに変更があるときの大半の場合に，スマート・コンパイルによって，コンパイル量を減らすことができることがわかる．

その次の行数欄は，スマート・コンパイルを使ったときにコンパイルされるファイルの総行数 (smart) と，通常の make を使った場合にコンパイルされるファイルの総行数 (make) で，上の(1)から(4)までそれぞれの場合について示すものである．このときの総行数の比も同時に掲げた．ここで，行数はプリプロセッサ処理の後の数値である．

5.3 総コンパイル時間の評価

まず，ヘッダーファイルに変更があるときのコンパイル（表6の(3)）を考える．このとき変更されたヘッダー・ファイルに依存するコード量，すなわち make による再コンパイル量を a とし，スマート・コンパイルを使用した場合の再コンパイル量を b とすると，make による再コンパイル時間が Th_{make} が $a*C$ (C は定数) であれば，スマート・コンパイルによる再コンパイル時間 Th_{smart} は，解析のオーバーヘッド (10%) と情報出力のオーバーヘッド (5%) を考慮すると，約 $(0.1a+1.05*b)C$ となる．

一方，ヘッダーファイルには変更がない場合，つまりインプリメンテーションのみに変更がある場合（表6の(2)）は，make でもスマート・コンパイルでも再コンパイル量は同じである．したがって，このときのスマート・コンパイルによる再コンパイル時間は，常に make による再コンパイル時間の5% 増しとなる．

結局，ヘッダーファイルに変更がないときの再コンパイル量を c として，make による再コンパイル時間の合計 Tt_{make} を $(a+c)C$ と置くと，スマート・コンパイルによる再コンパイル時間の合計 Tt_{smart} は $(0.1a+1.05(b+c))C$ となる．

ここで，上記の Th と Tt それぞれについて，スマート・コンパイルによる再コンパイル時間の合計と make による再コンパイル時間の合計の比の計算値 (T_{smart}/T_{make}) を，**表7**に掲げる．この表から，プログラムが極端に小さな場合（A）を除いては，スマート・コンパイルによって，コンパイル時間を減らせることがわかる．つまりスマート・コンパイルによる再コンパイル量 b は，十分 $T_{make} > T_{smart}$ となるくらい小さい．C++ については，COB の場合より効果が小さいことがわかる．

5.4 考　　察

以上の結果から，スマート・コンパイルによって，コンパイル時間を減らせることが示されたが，C++ よりも COB におけるほうが効果が大きかったことの主原因は，C++ では定数オフセット方式が使われていることにある．つまり，クラスのプライベートなメンバー変数だけに変更があった場合でも，パブリックなメンバー変数のオフセットの定数値が変わってしまうことがあるなど，定数オフセット方式では，変数オフセット方式の場合に比べ，クラス間の依存関係だけでなくメンバー間でも依存関係も多くなってしまうため，スマート・コンパイルの利点が十分にいかされなくなるのである．

今回の評価では，スマート・コンパイルに必要な各種の解析やコンパイルの時間がプログラムの行数に比例すると仮定し，また解析のオーバーヘッドの比率を比較的少ないデータを基準に決めたが，今後さらにデータを収集することによってより正確な比較を行えるようにしたい．

最後に言及しておきたいことのひとつに，スマート・コンパイルの1回あたりのコンパイルに与える効果がある．つまり，ヘッダーファイルに変更がなく，

表7 スマート・コンパイルの効果（時間比）
Table 7 Effect of smart recompilation (time ratio).

program	T_{smart}/T_{make}	
	h	t
A (COB)	1.07	1.07
B (COB)	0.62	0.97
C (COB)	0.52	0.87
D (C++*)	0.84	0.89
D (COB*)	0.72	0.80

スマート・コンパイルによる再コンパイル時間と通常の make による再コンパイル時間の合計の比．ただし，
h：ヘッダーファイルに変更があった場合の合計
t：すべての再コンパイルにおける合計
また，D(COB*) および D(C++*) は，C++ プログラムのデータにもとづくシミュレーションによる結果．

インプリメンテーションのみが再コンパイルされる場合は，非常に回数が多い割に，1回あたりの再コンパイル時間は短い．一方，ヘッダーファイルの変更は，例えばサンプルDの場合61回中13回と，全コンパイル回数に対し2割程度の頻度でしか起こらないが，他の多くのファイルの再コンパイルを誘発するため，1回あたりの再コンパイルが非常に長くなりがちである．この場合のスマート・コンパイルの効果は非常に大きく，例えばサンプルDの場合，もっとも効果的な場合には，make では 75048 行の再コンパイルが必要な場合に，スマート・コンパイルでは 11327 行で済むケースがあった．一般に，再コンパイル時間が意識して待つほどのものでない場合には，プログラムの開発効率にはさほど影響を与えないのが，ある長さを超えると，コンパイルが終了するまで完全にプログラムの開発は滞ってしまうことはよく経験するところである．この点を考慮すると，特に長時間のコンパイルになるほど効果をより発揮するスマート・コンパイルは，平均の数字に現れている以上に有効であると言ってもよいであろう．

6. 関連研究

Tichy[9] によるスマート・コンパイルは，前述のようにオブジェクト指向言語を対象としてはいなかったが，最近になってC++ において再コンパイル時間を短縮するためにさまざまな試みが現れている．Lucid[3] は，エディタ・コンパイラ・デバッガなどのツールが，決められたプロトコルによって緊密に連係しあった統合環境を提供している．Lucid の環境は，エディタからの情報によって変更された関数だけを再コンパイルするインクリメンタル・コンパイルや，依存関係の解析によって再コンパイル時間を短縮するための機構を備えている．また Palay[8] では，C++ においても COB のように変数を使ってメンバーのオフセットなどを表現することにより，クラスの変更によるクライアントの再コンパイルを削減する試みを行っている．言語の方面では，C++ に対し小規模の言語拡張によって，インターフェースとインプリメンテーションを分離する試みなども行われている[4]．

7. まとめと今後の課題

我々は本研究において，言語仕様の改良，具体的にはクラスのインターフェースおよびインプリメンテーションを分離し，クラス間の依存関係を減らすことによって，オブジェクト指向言語の再コンパイル量を減らせることを独自のCベースのオブジェクト指向言語 COB の開発によって示した．

同時に，適当なスマート・コンパイル機構によって再コンパイル量を短縮できることを，理論上の可能性としてだけでなく，実際のソフトウェア開発におけるプログラム変更の履歴を集め，その内容を解析することによって実証した．

スマート・コンパイルの機構は，当初 COB のソフトウェア・ツールとして実現したが，これが C++ でも有効であることも示された．しかしたがら，その効果は COB の場合に比べればまだ小さく，クラスの情報隠蔽機能を高めたことの効果がここでも現れている．つまり再コンパイルを短縮するには，言語処理系などのプログラミング環境だけでなく，COB のように言語仕様の改良を含めて考えることが必要であるといえる．

我々のスマート・コンパイルにはまだ改良の余地がある．特にチェンジ・アナライザは「疑わしきはコンパイル」という原則に基づくどちらかといえば大雑把なものである．これを精密なものにすることによって，さらにコンパイル量を減らすことができるであろう．しかし，精密な分析をすることは同時にスマート・コンパイルのオーバーヘッドを増加させる．したがって，プログラムの開発過程でどのような変更がソース・コードに加えられることが多いかといった点に関する実地のデータを，今後さらに収集することにより，効果的な手法を検討していきたい．特に，C++ は COB にくらべて言語仕様が複雑であり，inline 関数の存在などによって，ファイルの依存関係が大きくなっている分，独自の工夫によってツールをより有効なものにすることができると思われる．

謝辞 COB 言語の設計・開発の際には日本アイ・ビー・エム(株)東京基礎研究所の上村務氏・横内寛文氏の協力があった．また，本研究をまとめる際には筆者の同僚たちの助言に恵まれたことをここに記し，感謝をささげたい．

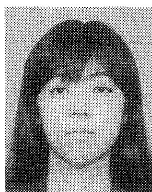
参考文献

1) Dausmann, M.: Reducing Recompilation Costs for Software Systems in Ada, *System Implementation Languages: Experience and Assessment, Proceedings of the IFIP WG 2.4 Conference* (1984).

2) Ellis, M. A. and Stroustrup, B.: *The*

Annotated C++ Reference Manual, Addison-Wesley (1990).
3) Gabriel, R. P. et al.: Foundation for a C++ Programming Environment, *Proceedings of C++ at Work-90* (1990).
4) Martin, B.: The Separation of Interface and Implementation in C++, *USENIX C++ Conference Proceedings*, pp. 51-63 (1991).
5) 三ツ井欽一, Javey, S., 久世和資：拡張可能なC++ ソースコード・ブラウザー基本設計, 第45回情報処理学会全国大会論文集（5）, pp. 133-134 (1992).
6) 中村宏明, 安田　和, 三ツ井欽一, Javey, S.：拡張可能な C++ ソースコード・ブラウザープログラム・データベース, 第 45 回情報処理学会全国大会論文集（5）, pp. 135-136 (1992).
7) Onodera, T. and Kamimura, T.: *COB Language Manual*, IBM Research, Tokyo Research Laboratory (1990).
8) Palay, A.: C++ in a Changing Environment, *USENIX C++ Conference Proceedings* (1992).
9) Tichy, W. F.: Smart Recompilation, *ACM TOPLAS*, Vol. 8, No. 3, pp. 273-291 (1986).
10) 安田　和, 小野寺民也, 北山文彦, 久世和資, 上村　務：Cベースのオブジェクト指向言語における再コンパイルの短縮, 情報処理学会プログラミング―言語・基礎・実践―研究会報告 1-5, pp. 39-48 (1991).

（平成 5 年 3 月 11 日受付）
（平成 6 年 5 月 12 日採録）



安田　和（正会員）

1964年生. 1987年東京大学理学部情報科学科卒業. 1989年同大学院理学系研究科情報科学専攻修士課程修了. 同年日本アイ・ビー・エム(株)に入社. 東京基礎研究所に勤務. プログラム言語, ソフトウェア開発環境に関する研究に従事. 現在日本アイー・ビー・エム・システムズ・エンジニアリング(株)に出向中. ソフトウェア科学会会員.

北山　文彦（正会員）

1964 年生. 1988 年東京大学理学部情報科学科卒業. 1995年同大学院理学系研究科修士課程修了. 同年より日本アイ・ビー・エム(株)東京基礎研究所に勤務. オブジェクト指向言語, オブジェクト指向分析・設計ツール, ユーザー・インターフェース作成法の研究に従事.

小野寺民也（正会員）

1959 年生. 1983 年東京大学理学部情報科学科卒業. 1988年同大学院理学系研究科情報科学専門課程修了. 理学博士. 同年 4 月日本アイ・ビー・エム(株)入社, 現在東京基礎研究所勤務. コンピュータ・グラフィクスの形式化, プログラム言語処理系, オブジェクト指向プログラミングなどの研究に従事. 著書に「A Formal Model of Visualization in Computer Graphics Systems」(共著, Springer-Verlag) がある. 日本ソフトウェア科学会, ACM, IEEE, USENIX 各会員.