COMP 91－22
SS 91－1

ソフトウェア保守におけるファイル間の依存関係管理ツールの試作
A Management Tool for the Dependency between Software Artifacts

島 和之　飯田 元　井上 克郎　鳥居 宏次
Kazuyuki SHIMA, Hajimu IIDA, Katsuro INOUE, Koji TORII
大阪大学 基礎工学部 情報工学科
Department of Information and Computer Sciences
Faculty of Engineering Science
Osaka University

Abstract Managing the artifacts which are produced in software development processes is important. In this paper, we consider the software maintenance process that is performed by several people in a group, and we point out some difficulties in managing the artifacts. To reduce these difficulties, We have developed a tool that supports the management of the dependency between artifacts. This tool displays the dependency and the status of the consistency of artifacts. By using of this tool, users can easily recognize the dependency between files and can know which file should be modified.

1 はじめに

ソフトウェア開発の支援においては,作業者一人一人の作業の支援と同時に,共同開発における作業者間,グループ間の情報交換や資源(例えばファイル)管理の支援を行うことが重要である.

一般に共同開発とは,複数の作業者の協力による開発のことである.ソフトウェアの共同開発において,作業者を数人で構成されたグループに分け,各グループがデバグや機能拡張のような目的別の変更を加えるような場合がある.このとき,各グループの変更すべき部分が完全に分離されているのが理想である.しかし,複数のモジュールから参照される共通のヘッダファイルやサブルーチンのように分離すると,かえって矛盾を招きやすくなるものもある.また,デバグや機能拡張の作業の途中では,ソフトウェアは矛盾を含んでいることが多く,各グループの作業がリアルタイムに全グループに伝わると悪影響となりかねない.

本報告では,ソフトウェアの共同開発の中でも,上記の例のようなデバグ工程と保守工程において,ファイル間の依存関係に基づいた開発支援を行なう方法について述べる.また,この方法に基づいた開発支援に用いるための依存関係管理ツールgradについて述べる.

2 ファイル間の関係

ソフトウェアの開発過程で生成されるファイルの中に,プログラミング言語を用いて記述されたソースファイルと,そのソースファイルをコンパイルすることによって生成されたオブジェクトファイルが含まれている場合を考える.このときソースファイルに何らかの変更を加えた場合,一般にはそれを再コンパイルして新しいオブジェクトファイルを生成する必要がある.このようなソースファイルとオブジェクトファイルとの間には依存関係が存在している.ソースファイルとオブジェクトファイルとの間の依存関係だけではなく,ソースファイル間や,自然言語で記述されたファイル間や,ソースファイルと自然言語で記述されたファイルとの間や,オブジェクトファイルと自然言語で記述されたファイルとの間にも依存関係が存在する.

・ソースファイル間の依存関係

モジュールAとモジュールBが別々のファイルに記述されていて,モジュールAがモジュールBを呼び出して利用している場合があげられる.この場合,モジュールBのパラメータの型が変わるとモジュールAも変更しなければならないので,モジュールAとモジュールBを記述した2つのファイルの間にも依存関係が成立している.

・自然言語で記述されたファイル間の依存関係

要求仕様書と設計仕様書との間の関係があげられる.要求仕様書に機能追加や変更があれば,設計仕様書も機能追加や変更を行なわなければならないことがあるからである.

・ソースファイルと自然言語で記述されたファイルとの間の依存関係

ソースファイルとそのモジュール設計仕様書との間の関係があげられる.

・オブジェクトファイルと自然言語で記述されたファイルとの間の依存関係

モジュールのオブジェクトファイルとそのモジュールについての検査成績書との間の関係があげられる.検査成績書とは,モジュールの設計仕様書に基づいて作成された検査仕様書のテスト項目に従って,モジュールをテストした結果を記録したものである.モジュールのソースファイルが変更されてオブジェクトファイルが変更されると,その検査成績書のテスト結果は信頼性が下がるので,再テストして検査成績書を変更する必要が生じる.

上記の例のように,ファイルbの内容にファイルaの内容の一部または全部が影響を受ける場合,「ファイルaがファイルbに依存している」という.また,このときファイルaの内容がファイルbの内容を正しく反映している場合(この判断は人間がファイルの内容を調べて非形式的に行なわなければならない場合もある),ファイルaに対して変更を加える必要のない状態を,「ファイルaとファイルbとの間の整合性がとれている」という.

3 ファイル管理の問題

ソフトウェア開発では,開発過程で作られる様々なファイルを効果的に管理していくことが重要である.ファイル管理の機能はどのような開発支援システムにも必要とされる基本的な機能であると考えられる.

ファイル管理の問題には,次のようなものがある.

(1) 複数の人間が同一のファイルに変更を加えるために生じる問題

例えば,二人の作業者が同一のファイルをエディタに読み込んでから,それぞれの変更を加えて書き込みを行ったとき,先に書き込んだ作業者の変更が無駄になるといった問題である.

(2) ファイル間の依存関係を把握できなくなる問題

複数のファイルの内容の間に矛盾がないかどうかを調べるとき,通常,関連のある2つのファイルの内容を見比べて調べるが,ファイルの数が多くなって,ファイル間の依存関係を把握できなくなると,チェック漏れが起きたり,関連のない2つのファイルの内容を調べるという無駄が生じる.

問題(1)を解決するファイル管理の方法を以下の2種類に大別することができる.

(a) 複数の作業者が共有の作業領域で作業する方法

各作業者の作業割当を行い,共有のファイルを操作する時はロックをかけるようにする方法.開発の設計・製造工程において有効である.

また,ファイルにロックをかけてから起動するエディタさえ用意すれば,簡単に実現できる.

(b) 作業者が固有の作業領域で作業する方法

各作業者が共有の領域から各自の作業領域にファイルを取り出してから,作業を行い,ファイルを共有の領域に書き戻す方法.開発のテスト・デバグ・保守工程において有効である.

この方法によるファイル管理ツールが既にいくつか開発されている[4].

本報告では,デバグ・保守工程に着目しているので,ファイル管理の方法としては(b)の方法を用いることにする.

4 既存のファイル管理ツール

複数の人間が共同でプログラミングするとき,互いの変更が矛盾しないように管理をするのは適当なツールの手助けなしにはきわめて困難である.また,過去に作成したソフトウェアにバグが発見された時,関連するソースコードを正しく再構成する場合も,適切な管理機構なしには多大な労力を必要とする.

このような問題を解決するために,さまざまなファイル管理ツールが提案されてきた.ここでは代表的なツールであるRCS/SCCS,CVS,makeについて簡単に述べる.

4.1 RCS/SCCSについて

RCSとSCCSはUNIX上での代表的なバージョン管理ツールである[4].これらのツールはファイルの変更履歴を保存することにより,過去の任意のバージョンをとり出すことを可能とする.また,変更中のファイルにはロックをかけることで,複数の開発者による変更の競合を防止できる.

RCSとSCCSの問題点として,以下のものをあげることができる.

・個々のファイル単位でしか管理できず,グループ化/階層化されたファイル群に対するサポートがない

ファイルの集合に対しては,別に自分でシェルスクリプト等を書いて対処しなければならない.1つ1つのファイルについては変更の理由を記述しておくことができるが,グループ全体でのバージョンにコメントを残すことはできない.普通は,そのために特別なファイルをつくって対処する.

・ある作業者がファイルを変更している間,別の作業者はそのファイルを変更できない

ファイルを変更するときにはそのファイルにロックをかけて取り出さなければならないが,この方法では同じファイルの異なる部分に,それぞれ同時に変更を加えることができず,一人の作業が完了するまでもう一人の作業が中断されてしまうといったように,プロジェクトの生産性を下げることになる.Makefileや共有変数を定義したヘッダファイルなどはその一例である.

・関連するほかのファイルの変更までは監視してくれない

ユーザがAというファイルを取り出して編集して,そのファイルを共通ディレクトリに書き戻そうとするときに,ほかのファイルとの整合性がとれているかどうかはチェックされない.例えば,AがBというファイルに依存していて,Aを編集している間にBが変わっていたとしても,ユーザが調べない限り矛盾があることには気づかない.

4.2 CVS

CVS(Concurrent Versions System)は,前節で述べた問題点を改善するものとして考えられたシステムである.

CVSの特徴としては以下のものがある.

・1つのマスターファイルで履歴を集中管理している

マスターファイルには,CVSの管理機構により過去の履歴がすべて保持されている.これによって,ユーザは過去のファイルをユーザ自身で名付けた名前(タグ)か特定の日付で取り出すことができる.マスターファイルが管理されている場所を,"リポジトリ"と呼ぶ.

CVSは階層的なディレクトリ構造のファイル群をまとめて管理することができる.

・複数のユーザが同時に一つのファイルを変更できる

各ユーザは,他のメンバーの変更作業に関係なく必要な部分のファイルをリポジトリから取り出して自由に編集することができる.ユーザがファイルを取り出して作業する場所をワークスペースと呼ぶ.ワークスペースのファイルは,編集が終了した任意の段階でリポジトリに書き戻すことが可能である.

CVSは,同じファイルに対して複数の作業者が同時に更新しようとする場合に矛盾が生じないためのサポートをしている.ただし,矛盾を自動的に解決するわけではなく,問題の箇所を指摘して,マスターファイルのなかに矛盾が持ち込まれるのを防ぐだけである.

4.3 makeについて

makeは定義ファイル(makefile)中にファイル間の依存関係と被依存ファイルの作成の規則を記述す

ることによって,ファイルの構成を管理し,自動的な生成を効率良く行なう.また,makeから上記のバージョン管理システムを利用することによって,バージョンを考慮した構成管理を行うようにもできる.

しかし,makeには次のような問題点があり,本報告で目的としているファイル管理を行なうには十分とはいえない.

・ソースファイル間の依存関係を管理できない

makeでは,ファイルを機械的な作業によって自動的に生成できるとき,元になるファイルをrequisiteファイル,生成されるファイルをtargetファイルと呼び,それらのファイルの変更時間を調べる.そしてrequisiteファイルがtargetファイルよりも新しいときに,targetファイルを再生成することによって,requisiteファイルとtargetファイルとの間の整合性をとっている.しかし,人間の編集作業によって生成されるソースファイルの間では,たとえ2つのファイル間に依存関係があっても,一方を変更するたびに他方を変更しなければならないとは限らず,また,逆に,変更をしたからといって,ファイル間の整合性がとれているとは限らない.従って,ファイルの変更時間の比較だけでは,ファイルを編集すべきかどうかを判断することはできない.

・ファイル間の依存関係を把握しにくい

ファイルの数が多くなると,makefileに並べられたファイル名の羅列を見ただけでは,ファイルの構成を把握することが困難になってくる.

5 依存関係管理ツールgradについて

5.1 gradの特徴

このような問題に対処するために,我々は依存関係管理ツールgradを作成した.

gradの主な機能は,

・ファイル間の依存関係グラフの表示

ファイルaがファイルbに依存していることを,ファイルbからファイルaに向かう矢印で示したグラフを表示する.

・ファイル間の依存関係の整合性の表示

ファイルaとファイルbとの間の整合性がとれているかどうかを,ファイルaとファイルbとの間の依存関係を示す矢印の色で表示する.

・ファイル生成プロセスの起動

指定されたファイルを生成するために予め定義されたプロセスを起動する.

また,主な特徴としては,

・ファイル間の依存関係を視覚的に捉えることができる

・変更すべきファイルが一目で分かる

があげられる.

5.2 gradの仕様

5.2.1 ファイルの種類

ファイルには,ソースコード,ロードモジュール,ドキュメント,オブジェクトコード,テストデータなどがある.

これらのファイルは大きく二種類に分けることができる.一つは作業者がエディタなどを用いて直接作成するものであり,各種ドキュメント,図式,ソースコードなどがこれに含まれる.もう一つは,ツールなどを使って他のファイルから機械的な手続きによって生成できるものであり,オブジェクトコードやロードモジュールなどがこれに含まれる.

本報告では,前者をsourceファイル,後者をtargetファイルと呼ぶ.また,機械的な手続きによってtargetファイルを生成するために必要なファイルをそのtargetファイルに対するrequisiteファイルと呼ぶ.

5.2.2 依存関係の管理

ファイルシステムの提供するディレクトリ構造などを用いて,ファイルの構成を表現することは,一般に困難であるといわれている[1].

gradでは,構成情報をrequisiteファイルとtargetファイルとの依存関係を用いて管理する.ここで,gradが管理する依存関係を二種類に分ける.一つは,requisiteファイルとtargetファイルとの間の依存関係であり,もう一つは,sourceファイル間の依存関係である.

requisiteファイルとtargetファイルとの間の依存関係では,targetファイルはrequisiteファイルから機械的な手続きによって生成できるので,requisiteファイルが変更されたらtaregetファイルを再生成することによって,依存関係を正しく保つことができる.よって,requisiteファイルとtargetファイルの変更時刻によって,targetファイルを再生成する必要があるかどうかを判断できる.すなわち,requisiteファイルがtarget

ファイルよりも新しければ,targetファイルを再生成する必要が有り,そうでなければ,再生成する必要はない.しかし,sourceファイル間の依存関係では,ファイルの変更日時ではsourceファイルを変更する必要があるかどうか判断できない.よって,sourceファイル間の依存関係が正しいかどうかを記録する仕組みが必要となる.

5.2.3 Folderについて

gradが表示するファイル間の依存関係グラフ上で,ファイルは矩形に囲まれたファイル名のアイコンによって示されている.gradでは,このアイコンをFolderと呼ぶ.全てのFolderは名前を持ち,親を持たない唯一のFolder(Base Folder)以外の全てのFolderは唯一つの親を持つ.すなわち,全てのFolderはBase Folderを根とする木構造を形成する.

Folderのファイル名は,親Folderのファイル名にFolder名をつなげた文字列になる.ただし,Base Folderのファイル名は空文字列である.例えば,Base Folderの子Folderの名前が'source/',その子Folderの名前が'main',その子Folderの名前が'.c'であるとき,ファイル名は'source/main.c'となる(figure 1).

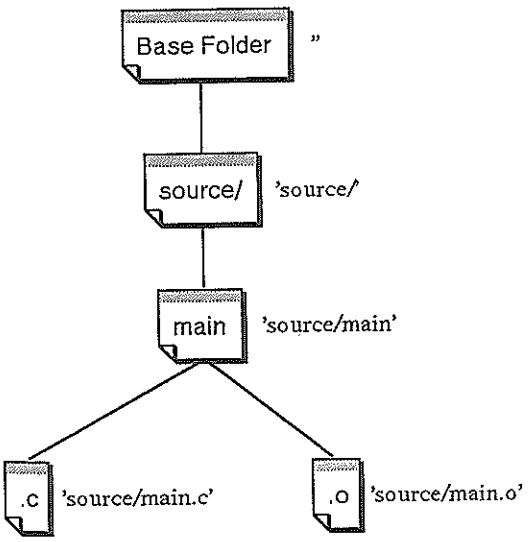

Fig 1. Folderとファイルの対応

5.2.4 ファイル間の整合の保証

例えば,ある大域変数について,その変数定義をしているファイルと外部変数宣言をしているファイルとの間で変数の型が異なっていれば,この2つのファイルは整合性がとれていない.

このような意味的な誤りを自動的に検出することは,一般には不可能である.しかし,作業者がチェックして,2つのファイル間の整合性がとれていた場合,その事を記録しておけば同じチェックを何度も行うという無駄は避けられる.

このように,作業者が2つのファイルをチェックして整合性がとれていたという事を記録した状態を2つのファイル間の整合が保証されている状態という.2つのファイル間の整合を保証した後,2つのファイルの一方でも変更された場合は,そのチェックの信頼性が下がるので,gradは整合の保証を取り消す.このため,作業者は再チェックの必要に気づくことができる.

5.2.5 表示内容

gradは起動すると1つのウィンドウを開き,Folder及びFolder間を結ぶ矢印を表示する.矢印はファイル間の依存関係を示し,矢印の種類はファイル間の整合性を示す.矢印はrequisite Folderからtarget Folderに向かっており,target Folderがrequisite Folderに依存していることを意味している.

ファイルBがファイルAに依存している時,ファイルAからファイルBに向かう矢印の種類は以下の状態を示している(figure 2).

(1) ファイルAとファイルBとの間の整合を保証された状態…緑(太線)

(2) ファイルAとファイルBとの間の整合を保証されていない状態

(2.1) ファイルAがファイルBよりも新しい状態……赤(点線)

(2.2) ファイルAがファイルBよりも新しくない状態……黒(細線)

5.2.6 ファイルの生成

gradは,ファイルの更新にともなって依存関係の図を変更しなければならないが,UNIXシステムではファイルの更新を検知するには,その都度ファイルの修正時刻を調べる必要がある.しかし,ファイルの更新の後,gradの表示が変更するまでのタイ

ムラグは,せいぜい数秒以内が望ましいのに対して,実際に作業中ファイルが更新される頻度は,数十秒から数十分に1度と非常に少ない.つまり,1回の更新を検知するために10~600回のチェックが必要となる.しかも,これはファイルが1つのときであって,ファイルがn個あるときは,1回の更新につき60n~600n回のチェックを行なうことになり,非効率的である.

この問題を回避するために,gradからファイル生成のプロセスを起動することにした.ファイルを更新するためには,ファイル出力を行なう何らかのプロセスが必要なはずである.よって,それらのプロセスが終了したとき,ファイルの更新をチェックすると効率が良い.UNIXシステムでは,子プロセスの終了時に親プロセスで行なう処理を設定できるので,親プロセスが子プロセスの終了を監視する必要もない.

また,gradからファイルの生成プロセスを起動する場合,生成するファイルに対応するFolderをダブルクリックするだけでよく,コマンド名やファイル名を入力する手間が省けるというメリットもある.

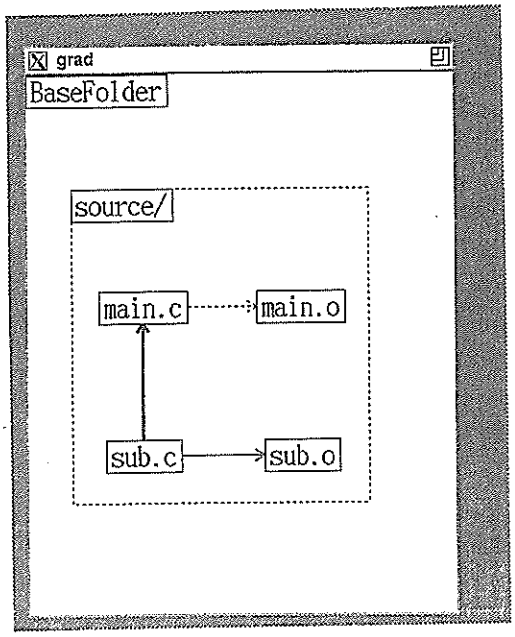

Fig 2. gradの表示例

5.3 ファイル.status

ファイル.statusは,gradが管理するFolderと矢印を保存している.ファイル.statusはエディタで編集可能なテキストファイルなので,ファイル名を一斉に変更するなどの操作をemacsやsedなどで行なうこともできる.

5.4 ファイル.makerc

.makercはgradから起動されてファイル生成を行なう実行形式ファイルである.その第1引数にはtargetファイル名,第2引数にはrequisiteファイル名のリスト,第3引数にはtargetファイルよりも新しいrequisiteファイルの名前のリストが与えられる.

ファイル.makercのインプリメントとしては,cshのシェルスクリプトを採用した.理由は,ほとんどのUNIX上でcshを実行できること,UNIXを利用する多くのユーザーがcshのシェルスクリプトの文法を知っていて,自分用に.makercをカスタマイズできるというメリットがあること,targetファイル名とrequisiteファイル名のリストを元に,targetファイルを生成するために必要な手順を十分に記述できること,の3点である.

6 あとがき

本報告では,ファイル間の依存関係を管理するためのツールgradの試作について述べた.gradを利用することにより,作業者はファイル間の依存関係を視覚的に把握することができ,効率良くファイルの依存関係の整合性を確認することができる.

今後は,実際のソフトウェア開発にgradを用いて,gradの有効性を試したい.

参考文献

[1]坪谷英昭,岸知二,入交晃一,鈴木美智子,猪狩錦光:"構成・版管理ライブラリLifeLineを用いたCプログラム開発環境",情処研報Vol.90,No.62.

[2]細谷倏一,伊東洋一:"大規模プログラミングのための環境",情報処理Vol.30,No.4.

[3]斉藤信男:"米国におけるプログラミング環境の開発",情報処理Vol.30,No.4.

[4]P.D.Stallmen:"Free Software CVS(Concurrent Versions System)",UNIX