ソフトウェアの定量的評価によるコーディング手法

平川　太海[†1,a)]　中嶋　卓雄[†1,b)]

概要：
ソフトウェアの品質を向上させるためには品質の定義を明確にし、ソフトウェアの定量的評価が必要となる。ソフトウェアメトリックスや複雑性などが議論されてきたが、必ずしも定性的な手法が確立されているとは言えない。本研究では、コーディング手法の定性化に注目し、広範囲に利用されている GCC(GNU C Compiler) のソースコードを解析する。GCC はマイナーや、メジャーバージョンアップなどが繰り返し行われてきているので、改良手法についても定量化が可能である。解析するシステムを設計、実装し実験した結果、プロの開発者が暗黙のうちに利用しているコーディング手法の一部を抽出することができた。

キーワード：ソフトウェアメトリックス，コーディング手法，ソフトウェア開発

Cording Method based on the Quantitative Evaluation of Software

Abstract: To clarify the definition of quality and quantitative evaluation of software are required to improve and maintain the software quality. Software metrics and complexity has been discussed, however, the qualitative evaluation method has not been established. In this research, we focus on the qualitative approach of coding method, and analyze the source code of GCC(GNU C Complier). GCC has been repeatedly updated with major and miner version. The quantitative evaluation enable to resolve how each version up has improved. As the results of experiments, we extracted the part of coding method developed by the expertise.

Keywords: Software metrics, Coding method, Software development

1. はじめに

近年、デジタルメディアの一般化に伴いソフトウェア開発の需要が高まると予想される。そのため、能力の異なる開発者が設計するソフトウェアの品質を一定以上に保つことは重要である。ソフトウェアの品質を向上させ、維持するためには品質の定義を明確にするとともに、ソフトウェアの定量的評価が必要となる。すでに、ソフトウェアの規模を表す評価指標としては、ステップ数（行数：Line Of Code）ではなく、入力、出力、照会、およびマスタファイルの原データに基づいて計算されるファンクションポイント (FP) が利用されている。しかし、FP 自身は、大きな評価指標であり必ずしも詳細な分析には十分ではない。一方、ソフトウェアのメトリックスおよび複雑性についても検討されてきた。ソフトウェアの構造を詳細に解析することによって、表記上の構造のみならず、意味構造をも対象としてソフトウェアに関する品質が議論されてきた。本研究では、ソフトウェア開発および意味構造をどのように分散させるのかなどのコーディング手法の定性化に注目していることもあり、ソフトウェアメトリックスに注目をして定量的な解析に基づき定性化を試みる。

定量化に関しては具体的なプログラムが必要となる。そのソフトウェアは一般的に利用されているものであり、プロのソフトウェア開発者が開発したものが好ましい。本研究では広範囲に利用されている GCC(GNU C Compiler) のソースコードを解析する。GCC は、オープンソースとして提供されているコンパイラであり、変数、コメント、などの記述量、分岐や繰り返しなどの構造が含まれ、インクルードファイルなどの外部ファイルとの関係性とソースファイルを特徴づけられる可能性がある指標は多岐にわたっている。また、GCC はマイナーバージョンアップ、メジャーバージョンアップなどが繰り返し行われてきている

[†1] 現在，東海大学
Presently with Tokai Uniersity
[a)] 3bijm001@mail.tokai-u.jp
[b)] taku@ktmail.tokai-u.jp

© 2015 Information Processing Society of Japan

ので、個々のバージョンアップにおいて、どのような改良が進んだのかについても定量化が可能である。

本研究では、実際に利用される C 言語用のコード生成機能のみを除外したコンパイラを設計し実装した。実装したシステムを利用すれば、他にも一般的に普及している多くのソフトウェアを解析可能である。ソースコードには、記述した開発者の経験が特徴として表れており、特徴を抽出することが可能であり、具体的なコーディング手法を示す事ができれば、教育機関や企業におけるソフトウェアの開発手法に応用することが可能となる。

2. ソフトウェアの定量化

2.1 ソフトウェアメトリックス

ソフトウェアの定量的な評価は、その製品の評価にもなりコストに反映するだけではなく、メインテナンス的視点からの評価にもつながる。しかし、すべてに渡って有効なメトリックスを見つけるのは困難であり、製品としての評価という視点から、FP 法などが用いられてきた。しかし、FP 法で評価されるのは、ユーザが関心を持つ 5 つの要素である。「入力」、「出力」、「照会」、「論理ファイル」および「インターフェース」の重みを付けた総和でソフトウェアのサイズに関係した指標であり、複雑性などの意味情報を含んだ解析手法ではない。

本研究では、ソフトウェアのサイズだけではなく、コーディング手法の定性化に注目していることもあり、ソフトウェアのメトリックスおよび複雑性について検討する。ソフトウェアのメトリックスについては、Beizer[1] らが分類しており、次に示すように、語句的メトリックス、構造的メトリックス、および複合メトリックスの 3 つに分類される。

語句的メトリックス：プログラムや仕様を記述したテキストの特性に基づくメトリクスであり、テキストを変形せず、意味を解釈せずに計測する。具体的には、プログラムの行数、ステートメント数、オペレータ（変数）の出現頻度、オペレーション（演算子）の出現数、オペーレタの出現総数、オペレーションの出現総数、キーワードの出現総数、トークンの総数などである。

構造的メトリックス：プログラムの構成要素間の構造的関係をもとにしたメトリックスであり、通常、制御フローグラフやデータフローグラフの特性を計測する。この例としては、リンクの数、ノードの数、ネストの深さなどがある。

複合メトリックス：プログラムの語句特性と構造的特性を組み合わせたり、あるいは、語句的特性と構造的特性の両方の機能にもとづいて測定したものである。

ソフトウェア開発における複雑性の解析は従来からソフトウェアのテストに関する研究から検討されてきた [4]。プログラムの静的解析から、プログラムの動作をグラフ化し複雑さを定義し解析することによって、ソフトウェア開発とそのテストとして利用されてきた。テストを目的としたプログラムの静的構造のグラフ化および状態遷移なども使いながら、テストの自動化システムも開発されてきている。また、グローバル化による地理的に離れた文化も異なるチームにおけるプログラム開発の難しさについて研究されてきた [5]。グローバルなチームの分散やチームユーザの動的な変化は協調的なソフトウェア開発の効率性においては負の影響をもたらすことが示されている。ソフトウェア開発が人間が直接的に開発する、記述する製品であるため、このような要素とソフトウェアの品質との関係についても検討する必要がある。また、Sherif[2] らは，ソフトウェアメトリックを決定することがリアルタイム組み込みシステムの開発と維持に有効であることを証明するために、16 種類におよぶメトリックスを定義し、200 以上のモジュールを持つソフトウェアに対して検証を行った。16 のメトリックスを相互関係から関連メトリックスを定義し、関連メトリックスによるソフトウェアの評価を行ってきた。また、Zheng[3] らは COCOMO81 dataset を対象として、製品の複雑度と実際の開発量の関係を複数の関連規則として設定して定義している。結果として、複雑性の増加および開発作業量の負の関係は、特定の条件では合致しないことが証明されている。しかし、この文献では複雑度の定義が詳細化されておらず、複雑さと知識との関係が明確になっていない。

本研究では、語句的メトリックスに主に注目しながら、構造的メトリックスについても解析対象とする。また、メトリックス間の関連性および複合化したメトリックスの定義については、今後の研究テーマとした。

2.2 提案するメトリックス

本研究では、まず研究の第一段階として対象をプリプロセッサに注目し、指標の選別と意味づけを検討する。具体的には、プリプロセッサ命令の頻度、1 ファイルからインクルードされるファイル数、インクルードの入れ子構造の深さなどを計測する。また、プログラム本文では、行数、コメント行数およびステートメント数を計測する。特に、インクルード構造の深さとして、ファイルがどの程度他のファイルから参照され、およびどの程度ファイルを参照するのかに関する参照構造を計測する。

また、複数のバージョンに関する情報を抽出し、バージョンアップにともなう変更手法についても考察する。

3. 評価システムの設計と実装

3.1 C ソースの解析対象

本研究で解析対象としている GCC は 当初 GNU プロジェクトが開発した C のコンパイラであり、ANSI 規格 (ANSI X3.159-1989) 準拠の C 言語コンパイラ処理系として最も普及している。現時点（2015 年 1 月時点）で、最新

© 2015 Information Processing Society of Japan

のバージョンは GCC 5 が 2014 年 11 月に Stage 3 としてスタートしているが、一般的に普及しているシステムで動作する最新のバージョンは GCC 4.9.2(2014 年 10 月) である。以下では、バージョン番号の先頭数字をメジャー番号、それ以下の数値をマイナー番号と呼ぶことにする。1987 年 5 月にバージョン 1.0 が公開されてから頻繁にバージョンアップが繰り替えされ、1992 年 2 月にバージョンが、1.42 から、2.0 に移行し、2001 年 1 月に GCC 3.0 がリリースされ、2005 年 7 月に GCC 4.0.1 がリリースされ、現在のバージョンに至っている。2.X から、3.X に移行する時期に、c のコンパイラのバージョンだけではなく GCC のバージョンとして公開されている。マイナーバージョンについては、バージョンアップの連続性と時系列は同じではなく、マイナーバージョンの一桁目については、枝分かれしながら、バージョンアップが繰り替えさえる場合もある。

3.2 評価システムの概要

本研究で実装するプログラムでは、すべての処理をトークンという単位に変換して構文解析処理を行なっている。また、プリプロセッサ処理は　#error、#warning、#line、#pragma、#include、#define、#ifdef 、#ifndef、#if defined、#if !defined、#if、#elif などの多岐にわたる。プログラム中で処理の対象とするのは、#include、#define 、#ifdef、#ifndef 、#undef とする。#define、#ifdef、#ifndef、#undef などのユーザ定義マクロに関するプロセッサーの処理は、マクロの管理テーブルを実装する。マクロ管理テーブルは、ユーザが定義した定数や、式を登録し、C プログラムの実行部分に含まれることが確認できた時、実際の値に置き換えるために用意するテーブルである。#include は基本的にローカルなものだけに限定し、再帰的な呼び出し、さらにはその禁止処理などを除いた処理を行う。#if, #elif などの式の評価に関した処理は行わない。システムですでに定義されている定義済みマクロは事前に登録が必要となり、さらにはシステムによって異なるため処理しない.。

4. 最新バージョンの定量化とコーディング手法

GCC 4.8.0 に基づいてソースファイルを分析した結果、以下のようなソフトウェアのコーディング手法が発見できた。以下で記述する「行数」とは、ファイル中に使われている改行コードの数をカウントした指標であり、「ステートメント数」とは、セミコロンで区切られる命令コードをカウントした指標である。また、「コメント行数」とは、ファイル中のコメントの存在する行の改行コードをカウントした指標である。したがって、「行数」の中には「コメント行数」が含まれているが「ステートメント数」には「コメント行数」は含まれない。

4.1 サイズに関する特徴

ソフトウェアプログラムに中のファイル数を計測した場合、.c ファイルが 76%を占め、.h ファイルが 24%を占めていた。行数毎のファイルサイズ分布を計算した結果を図 1 に示す。横軸がファイルの行数、縦軸がファイルの頻度である。偏りがあまりに大きいため、図の下方にその頻度の値を記述した。

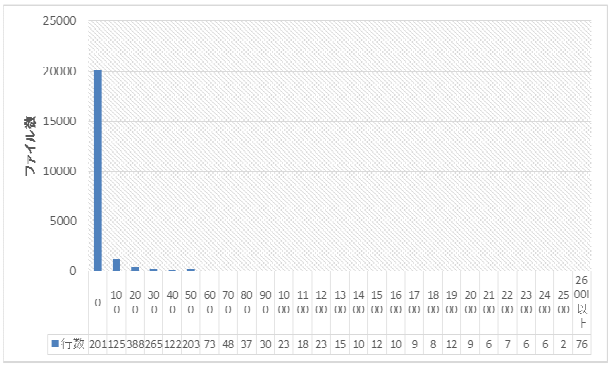


図 1　行数によって分類したファイル分布

ファイルの約 65%は 200 行以下の小さなファイルから構成されており、さらに約 8 割は 1000 行以下の行数で記述されている。8000 行を超える . c ファイルのほとんどに複数の関数が定義されている。分布としては Longtail な分布となっており、UNIX OS などのファイルサイズ分布と同様の特徴を持っている。図 2 にコメント行数によるファイルの分布を示す。

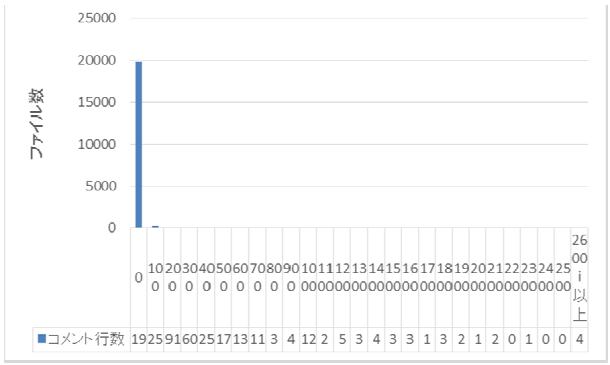


図 2　コメント行数によって分類したファイル分布

この図 2 からも明らかなように、ほとんどのファイルは 100 行のコメントである。行数による分布と比較してコメント行数による分布の方が、偏りが大きくなっており、結果として、コメント行数は一定量で十分であることがわかった。

トップファイル以外は、ほぼ 200 行以下の小さな構造に分割されているのが分かった。行数による分布と比較して、ファイル数の頻度は少ない。その結果から、 1 つの行に複数のステートメントが記述されていることを示してい

© 2015 Information Processing Society of Japan

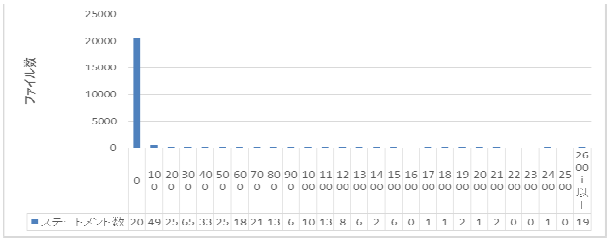


図 3 ステートメント数によって分類したファイル分布

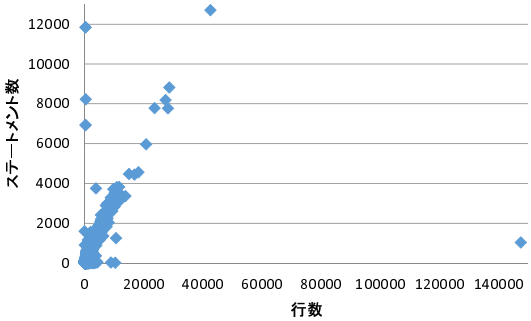


図 4 行数とステートメント数の関係

る。図 4 に、行数とステートメント数の関係を示す。

結果から、１行は２ステートメントから構成されているのがわかる。以降の議論では、行数、コメント行数およびステートメント数について、各指標の量は一定の範囲に集中しているので、最もファイル数の多い指標の範囲を中心に詳細化する。

図 5 は行数 2000 行以下のファイルの行数とコメント行数を.c および.h ファイルを分離してプロットしたものである。

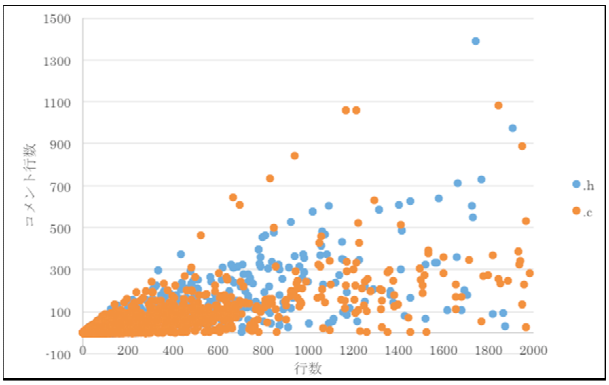


図 5 行数とコメント行数の関係

図に基づきデータを詳細に検討した結果、コメントの内容は次のように分類できる。（１）ファイル全体の説明（２）インクルードされるファイルの説明　（３）列挙体構造体クラスなど、ユーザ定義型についての説明（４）関数の仕様の説明

(１) は、プログラムの目的、機能、全体のプログラム中に位置付け、ファイル権利の所在なども記載される。このタイプのコメントはファイルに一度だけあり、ファイルサイズにほとんど左右されない。一定のサイズ以上、具体的には２００行を超えるプログラムに記載されていた。(２) は、インクルードされるファイルの概要や、インクルードファイルの中で定義される構造の説明である。コメントのサイズは１つのファイルに付き３から４行であり、インライン関数など機能を持ったものが多い。(３) は、構造体や列挙体などのユーザ定義型の用途やメンバー変数が記載される。変数やメンバーに１対１に記載されることが多い。(４) は、関数の機能を記載であり、機能の複雑さによって大きくに左右され、関数の機能が限定的であれば、関数のサイズが大きくなってもコメント量は増えることはない。しかし、オプションによって関数の挙動が変わるような場合、それぞれのコメントの割合が増える。

コメント行数と行数の関係について相関係数 0.75 の相関が確認できた。行数 1000 行以下のファイル中 93.4%でコメントは行数中 50%以下である。

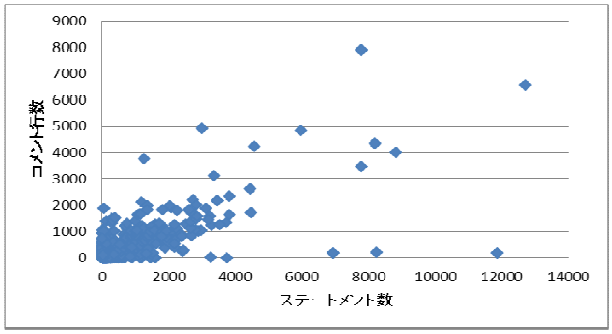


図 6 ステートメント数とコメント数

図 6 はすべてのファイルについてコメント行数とステートメント数の二つの指標の関係を示している。行数とコメント行数では、相関係数は 0.58 とほとんど相関はないが、ステートメント数とコメント行数の間では、0.73 の相関がみられた。この違いが生まれる理由として考えられるのは、空白の改行である。プログラムでは、可読性を高めるためであり、関数部に多く使われている。

4.1.1 プリプロセッサ命令に関係した特徴

まず、プリプロセッサ命令の頻度を計測した。図 7 は、ソースプログラム中の１つのファイルにおけるプリプロセッサ命令の平均的な出現頻度を.c および.h に分けて示したものである。.c ファイルが主に include 文、undef 文、ifdef 文を利用しており、.h ファイルは define 文を利用している。define 文などの定義部は.h ファイルに記載する傾向がある。

次に、プリプロセッサ命令の include 文に注目し、その参照関係を計測した。ここで、ファイルがインクルードされた回数を「in 回数」、インクルードしたファイル数を「out 回数」と呼ぶ。また、.h ファイルに関しては GCC が

© 2015 Information Processing Society of Japan

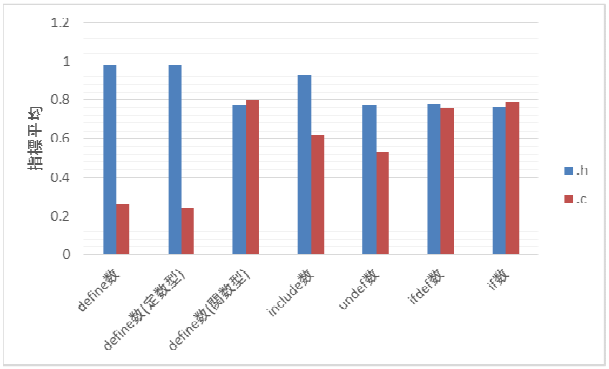


図 7　ファイル中のプリプロセッサ命令の平均出現頻度

提供しているファイルに記載している include 文についてはチェックしているが、それから呼ばれるシステムの、.h ファイルの中の構造については解析していない。

図 8、9 はインクルードによる in 回数と out 回数の関係を.h .c のそれぞれをファイルごとにプロットした図である。In 回数の平均は.h が 11.5、.c が 6.4 回となり、. h が多く参照されている。一方 out 回数が一回以上のファイルは.h では、4062 個.c では、7497 であり、平均では. h が 1.6 回、. c が 2.1 回となっている。また、.h のファイルで out 回数が多いファイルは、複数のファイルをインクルードする際に、 1 つのファイルイをンクルードするだけで済むように複数の include 命令から構成されるファイル、または、システムが提供するインクルードファイルであった。また、out 回数が 1 以上のファイルについて、システムおよびユーザーファイルを比較すると、平均でシステムインクルードでは 1.4 回インクルードされているのに対し、ユーザインクルードは、1 回となりシステムファイルのほうが多くインクルードされている。

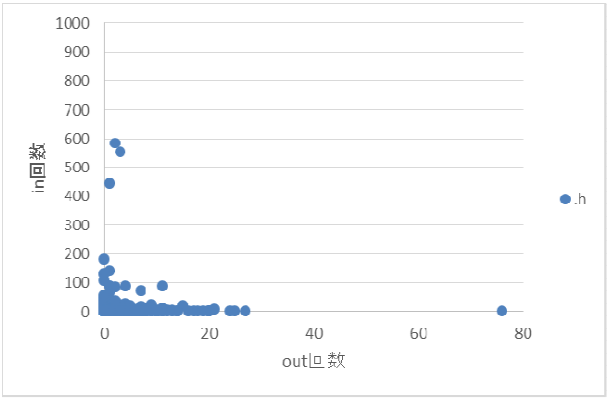


図 8　.h ファイル中での in 回数および out 回数の関係

次に、c ファイルの中で in 回数が多い場合、定数の定義よりもプロトタイプ宣言やインライン関数の定義などが多く、それらで定義されているインライン関数は、複数の変数が使われており、ループ構造をもったより関数に近いものであった。また.c ファイルのインクルードは.h とは逆にユーザインクールドが平均 1.3 回であり、システムインクルードが 0.8 回であった。

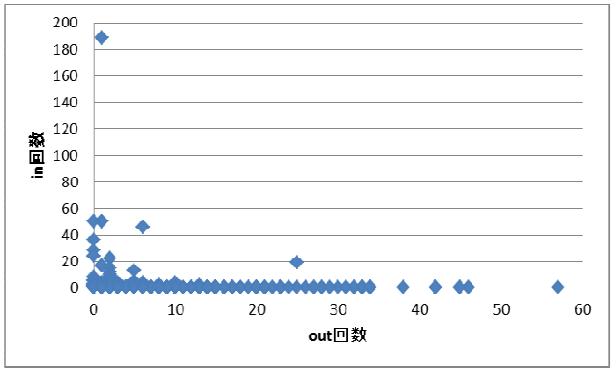


図 9　.c ファイル中の in 回数および out 回数の関係

c ファイルで in 数が多い場合は.h と同様に定義や小規模な関数が記述されている。.c の場合.h に比べ関数の定義に重点を置いたものが多く、処理のサイズでは、. h で in が多い、. h で out が多い、さらに、. c で in が多く、最後に. c で out が多くなる。逆に定義部分が減る傾向にある。

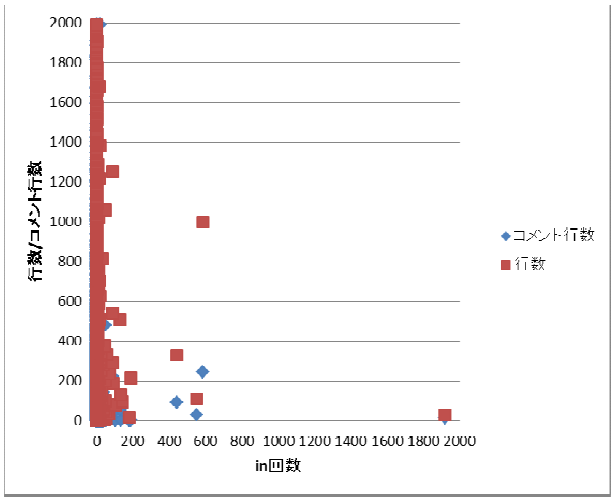


図 10　参照される in 回数のサイズからみた特徴

図 10 から、参照回数の多いファイルは行数、コメント行数ともに減少する傾向がある。参照回数の多いファイルの中には、このパッケージのコピーライトや補足などをコメントの形で記載しているファイルが存在する。

次に、include 文のネスト階層とファイルの特徴を計測した。.c ファイルから最初に参照される.h ファイルを 0 階層のファイルとし、その 0 階層から参照されるファイルを 1 階層とする。このようなネスト階層のレベルを階層の数値として表記する。

図 11 は、インクルードのネスト構造の中で何階層でインクルードされたのか階層ごとのファイル数を表したものである。

ネスト階層とファイル数の関係は、指数オーダーで減少

© 2015 Information Processing Society of Japan

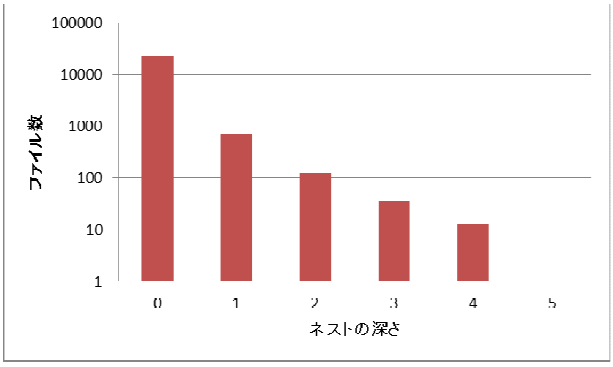


図 11 ネスト階層毎のファイル数

し、ネストの深さ 0 のファイルは 96% を占めており、3 階層まで含めると 99%以上になる。深い階層になるにつれて、内部変数の定義や数値のなどの基本的な定義データから成るフィアルが多くなっていることが明らかになった。

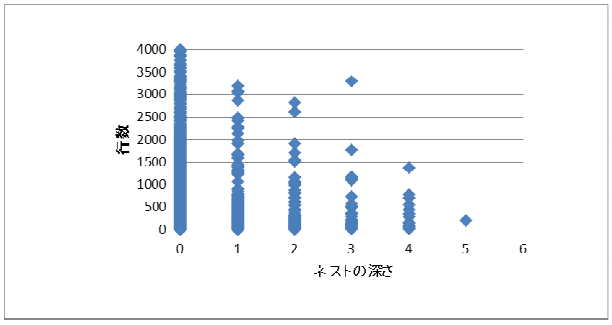


図 12 ネスト階層毎のファイルの行数との関係

図 12 から、ネスト階層が深い。すなわち深い位置で呼ばれる.h プログラムについては、そのサイズが徐々に小さなもとなり、コンパクトな構造となっていることがわかる。

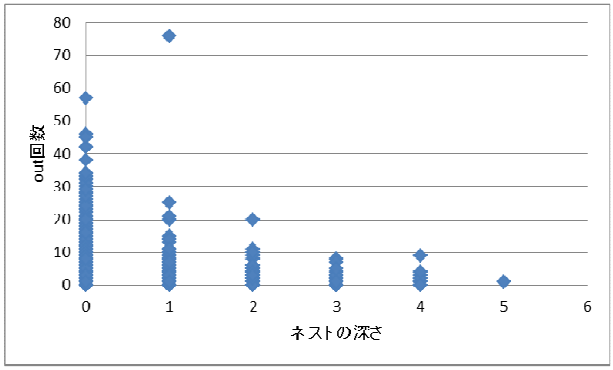


図 13 ネスト階層毎のファイルの数

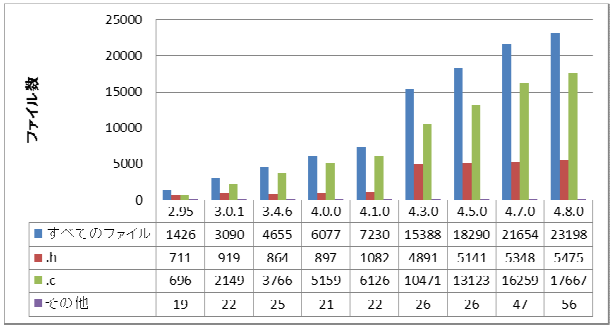
結果から、ファイル数も階層が深くなれば減少している。#include 文は再帰的に呼び出されることもあり、ネストを深くすることが可能である。#include 文については、ネスト階層はそれほど深くなく、2 階層程度の参照が大部分でありそれより深い階層は変化の少ないファイルだと思われる。ライブラリーを除いて、人間が把握できるネスト階層は、それほど大きくないことが明確になった。

5. バージョンの変化による特徴の変化

この章では、バージョンアップによるコーディング量およびソフトウェアの品質の変化について考察する。具体的には、GCC が配布している異なるメジャー番号、およびマイナー番号を持つパッケージ内のファイルについてバージョンの特徴を測定する。

5.1 サイズに関する特徴

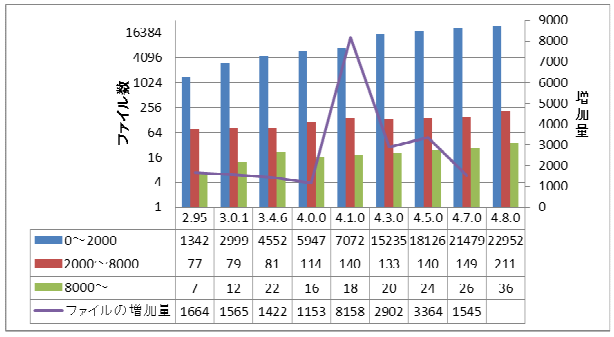
まず、図 14 に示すように、5 つの異なるバージョンについて、ファイル総数、.c および.h ファイル数について計測した。



	2.95	3.0.1	3.4.6	4.0.0	4.1.0	4.3.0	4.5.0	4.7.0	4.8.0
すべてのファイル	1426	3090	4655	6077	7230	15388	18290	21654	23198
.h	711	919	864	897	1082	4891	5141	5348	5475
.c	696	2149	3766	5159	6126	10471	13123	16259	17667
その他	19	22	25	21	22	26	26	47	56

図 14 異なるバージョン毎のファイル数

図から、バージョンアップを重ねるごとにファイル総数、.h ファイル、.c ファイル数は増加している。バージョン 4 の初期段階までは、線形で増加しているが、メジャー番号の 4.3 のから急激にサイズが増加している。線形で増加している部分では、.h ファイル数については大きく増加することなく、.c ファイルが増加しており、機能の追加によってソースコードが増加する結果になったと考えられる。急激に増加している理由としては、機能の拡張ファイルの細分化、バージョン間の互換性の維持のために残されたファイルの蓄積などが考えられる。



	2.95	3.0.1	3.4.6	4.0.0	4.1.0	4.3.0	4.5.0	4.7.0	4.8.0
0~2000	1342	2999	4552	5947	7072	15235	18126	21479	22952
2000~8000	77	79	81	114	140	133	140	149	211
8000~	7	12	22	16	18	20	24	26	36
ファイルの増加量	1664	1565	1422	1153	8158	2902	3364	1545	

図 15 ファイル数のサイズの分布どバージョンアップによる増加量

図 15 はバージョンアップによる行数の分布、ファイルの増加量を示したものである。いずれのバージョンでも、最も増加しているファイルが行数 0 から 2000 のファイル

© 2015 Information Processing Society of Japan

である。また、バージョン 4.1 から 4.3 では急激に増加しており、4.2 から 4.3 においてマルチコア CPU 対応したことによるものであると考えられる。また、4.3 から 4 .5 における増加は c の機能拡張と思われる。

図 16 に示すようにバージョンが進むにしたがって、行数もコメント数も減少傾向にある。これはサイズの小さいファイルが増えて平均を押し下げていることとファイルサイズが削減されたことの 2 つが考えられる。

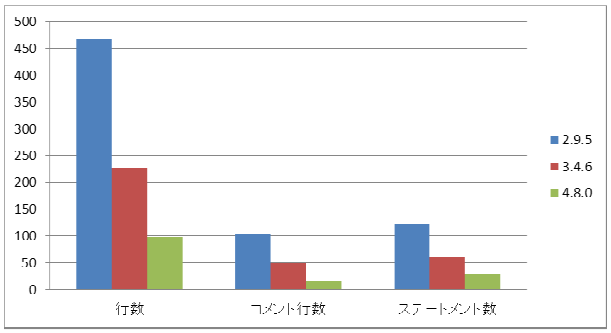


図 16 バージョン毎のプログラム構成要素の平均個数

5.2 プリプロセッサ命令に関係した特徴

バージョンが新しくなるにつれて 1 ファイルあたりのサイズがどのように変化しているのか検証した。図 17 はバージョン内の各プリプロセッサ文の平均を示したものである。

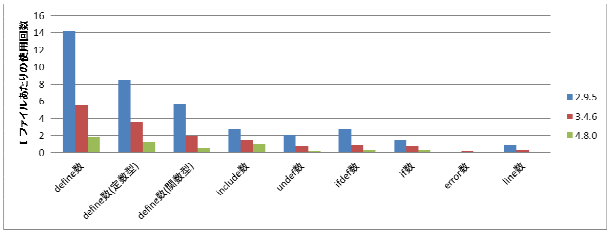


図 17 バージョン毎のプリプロセッサ文の平均個数

1 ファイルあたりの各指標の平均値はバージョンが進むにあたって減少傾向にあり、機能の分散化が行われている。また、プリプロセッサ命令が記載されないファイルの増加も平均値を減少させている原因だと思われる。difine 命令の場合、命令が一切記載されないファイルが、バージョン 2.9.5 で 454 個だったものが、3.4.6 では 3139 個、4.8.0 では 14923 まで増加した。次に、1 回以上 define 命令を実行しているファイルにつて、define 命令の平均数は、それぞれ 20.6、

17.1、5.2 となり、減少傾向となった。Include ついて平均したところ、いずれも 3. 4 程度であったため 1 ファイルあたりのインクルードは 3 回程度が適度な回数だと思われる。

バージョンアップによるファイルの変化について、2 つのバージョン、2.9.5 および 3.4.6 を比較することでファイルの内容の変化を調査した。Include 数、すなわち out 回数はユーザとシステムインクルードファイルの合計である。

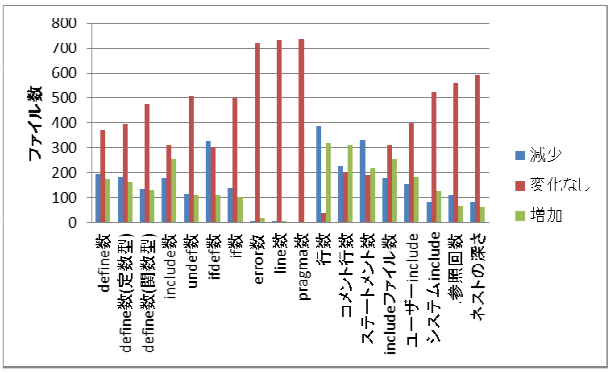


図 18 バージョンアップにともなう構文要素の変化

図 18 は、2 つのバージョンの同名ファイルの各構文要素について、減少、増加および変化のないものについてファイル数を示したものである。各構文要素に関して増加したファイル数と減少したファイル数がほぼ同じ場合が多い。一定の構文要素について追加されたファイルと同等に減少される場合が多い。しかし、ifdef 文については減少しているファイル数が多くなっている。理由としては、ファイルの分割・分離が進み、ファイル中で ifdef 文によって分離する必要がなくなったことなどが考えられる。

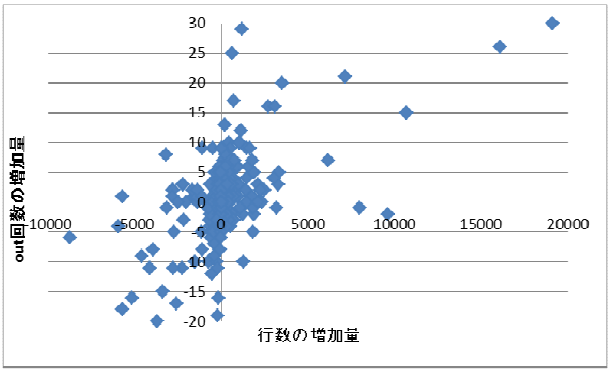


図 19 行数の増加量と inluce 文の増加量の関係

結果として、include 文が増加しているファイルに関しては行数も増加している場合が多い。

バージョンアップによるステートメント数の増加量とコメント行数の増加量の間には 0.76 の相関があるが，相関がないところについては，コメント行数が一定の量で抑えられている。

include 文のネスト階層が 2 レベルであっても、多く参照回数が変化している。これはシステム全体の構造の変化に伴って参照される回数が大きく変化していることによる。Include 文の参照回数については、バージョンアップにより大きく変化する場合もあることが明らかになった。

© 2015 Information Processing Society of Japan

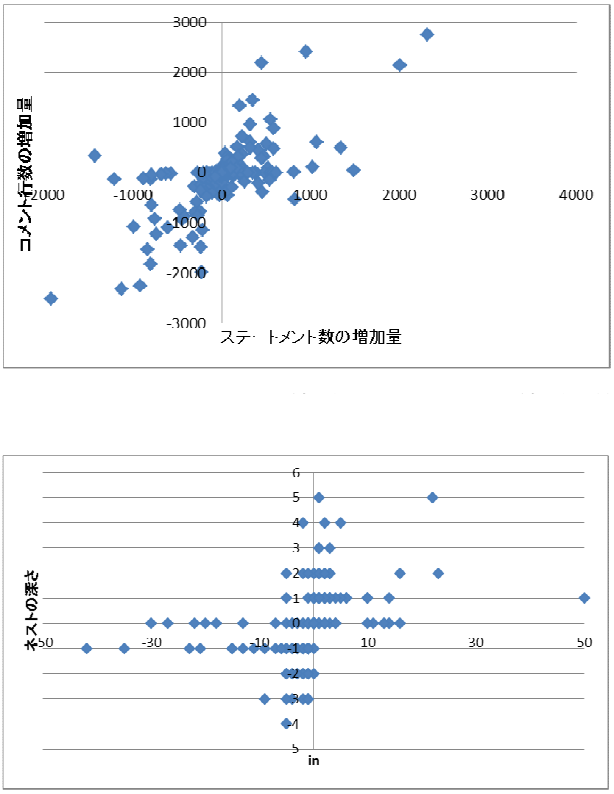


図 21　参照回数の増加量とネスト階層の増加量の関係

6. おわりに

一般的に普及しているシステムで動作する最新のバージョンは GCC 4.9.2(2014 年 10 月) であるが、本研究では汎用的に利用されている GCC 4.8.0 を最新バージョンとして、そのプロダクトに基づいてソースファイルを分析した。結果以下のようなソフトウェアのコーディング手法が発見できた。

まず、最新バージョン 4.8.0 においては次のような特徴を抽出することができた。（１）ファイル分布としては、Longtail な分布となっており UNIX OS などのファイルサイズ分布と同様の特徴を持っている。（２）コメント行数は一定量で十分である。（３）行数とステートメント数は比例関係にあり、ステートメント数の５倍の行数から構成されており、見やすさが大きく行数に影響している。（４）c ファイルの中で in 回数が多いものの特徴として、定数の定義よりもプロトタイプ宣言やインライン関数の定義などが多い。（５）include 文のネスト階層の調査から、ネスト階層とファイル数の関係は、指数オーダーで減少する。深い階層になるにつれて、内部変数の定義や数値のなどの基本的な定義データから成るファイルが多くなっている。（６）ネスト階層が深い.h プログラムについては、そのサイズが徐々に小さなもとなり、コンパクトな構造となっている。

また、異なるバージョンのソースファイルを比較すると次のような特徴を抽出することができた。（１）バージョンアップを重ねるごとにファイル総数、.h ファイル数、.c ファイル数は増加している。バージョン４の初期段階までは、線形で増加しているが、メジャー番号の４の途中から m 急激にサイズが増加しており、機能の拡張ファイルの細分化、バージョン間の互換性の維持のために残されたファイルの蓄積などによる影響と思われる。（２）１ファイルあたりの各指標の平均値はバージョンが進むにあたって減少傾向にあり、機能の分散化が行われている。（３）バージョンアップによるステートメント数の増加量とコメント行数の増加量の間には 0.76 の相関があるが、相関がないところについては、コメント行数が一定の量で抑えられていると思われる。

今回は、１種類の GCC というソースプログラムのバージョンアップにより、ソースコードにどのように構造的が変化しプログラムの質が改善されたのかについて検証したが、今後は複数のプログラムについて検証していきたい。

参考文献

[1] Boris Beizer: *Software Testing Techniques (Second Edition)*, ソフトゥエアテスト技法, Van Nostrand Reinhold (1990), 日経 BP.

[2] J.S. Sherif, J.M. Hops: *Development and Application of Composite Complexity Models and a Relative Complexity Metric in a Software Maintenance Environment*, WESCON96, pp.514–526 (1996).

[3] Zheng Li, Liam O'Brient, and Ye Yang: *Impact of Product Complexity on Actual Effort in Software Developments: An Empirical Investigation*, 23rd Australian Software Engineering Conference(ASWEC),pp.170–179 (2014).

[4] Norman F. Schneidenwind: *Application of Program Graphs and Complexity Analysis to Software Development and Testing*, IEEE Trans. on Reliability, Vol.R-28, No. 3, pp.192–198 (1979).

[5] Gnaw Lee, J. Alberto Espinosa, and William H. DeLone: *Task Environment Complexity, Global Team Dispersion, Process Capabilities, and Coordination in Software Development*, IEEE Trans. on Software engineering, Vol. 39, No.12, pp.1753–1771 (2013).

© 2015 Information Processing Society of Japan