Vol.2013-SE-179 No.2
2013/3/11

大規模ソフトウェアの概要把握の支援を目的とした構造体型の特徴付け

竹治 勲[1,a)]　大久保 弘崇[2,b)]　粕谷 英人[2,c)]　山本 晋一郎[2,d)]　齋藤 邦彦[3,e)]

概要：本論文では，C 言語を対象として，プログラム中で利用されている構造体を特徴付けする手法を提案する．特徴付けによる構造体の分類は，大規模なソフトウェア理解の初期段階における全体構造把握の手がかりとなる．特徴付けは，構造体に対して，関数の呼び出し文脈を考慮したデータ流量に基づいて行う．プログラムの実行結果から構造体のアクセス頻度を計測し，その結果から対象プログラム中の構造体を特徴付けする．提案手法を検証するために 2 つのソフトウェアで実験を行った．

1. はじめに

ソフトウェアの開発プロセスでは，生産性と信頼性の向上のためにライブラリを利用したり既存のソフトウェア部品を再利用する．その際に，開発者は利用する部品の挙動を確認し，開発の目的に適しているかを検討する．また，保守プロセスではソフトウェアの品質維持・向上や機能拡張を行う．これらの工程を進めていくためにはソフトウェアのモジュール構成，機能，動作を正確に把握する必要がある．このようにソフトウェアの開発・保守において，ソフトウェア理解は重要である．ソフトウェア理解の不十分さは，プログラムへの不必要な修正やバグ混入など成果物の品質に重大な影響を及ぼす．

しかし，近年のソフトウェアの大規模化・複雑化に伴い，ソースコードやドキュメントが増加しているため，ソフトウェアを全て理解することが困難になってきている．ある目的にしたがい理解しようとしても，どこの箇所から理解を始めればよいか検討することは難しい．そのためソフトウェアを理解しようとする開発者に対して，理解すべき箇所を支援する情報を提供することが要求される．

[1] 愛知県立大学大学院情報科学研究科
Graduate School of Information Science and Technology, Aichi Prefectural University
[2] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
[3] 滋賀大学経済学部
Faculty of Economics, Shiga University
[a)] takeji@yamamoto.ist.aichi-pu.ac.jp
[b)] ohkubo@ist.aichi-pu.ac.jp
[c)] kasuya@ist.aichi-pu.ac.jp
[d)] yamamoto@ist.aichi-pu.ac.jp
[e)] saito@biwako.shiga-u.ac.jp

本論文の目的は，ソフトウェア理解の初期段階にある開発者に対するソフトウェアの概要把握の支援を目的とし，構造体型の特徴付けを行う手法を提案する．構造体型はプログラム中の機能と関連性が高いため，本手法により得られる構造体型の特徴は，大規模なソフトウェアに対してソフトウェアのどこを理解すべきかの指標となる．関数の呼び出し文脈を考慮した，プログラム実行時の構造体型のデータ流量，すなわちアクセス回数に基づいて特徴付けを行う．提案手法は C 言語を対象とする．C 言語は UNIX をはじめとする OS やコンパイラなどの大規模ソフトウェアの実装に用いられており，提案手法の対象として適切と考えられる．

2. ソフトウェア理解の工程

ソフトウェア理解の工程は大きく全体構造の理解と詳細動作の理解の 2 つに分けられる．各工程では以下の作業を行う．

全体構造の理解　各モジュールの役割の把握，各モジュールの入出力の把握，モジュール間の相互関係の把握

詳細動作の理解　ソースコードの解読，プログラムを実行することによる挙動の観察

2.1 全体構造の理解

全体構造を理解する段階では，開発者はモジュールの振る舞いを確認し，ソフトウェアの概要を把握したり，詳細動作を理解する箇所の見当を付ける．この段階では，対象ソフトウェアに対する理解が不十分である開発者が大まかな処理の流れを把握しようとする．

開発者に情報を推薦するという観点での関連研究として

文献 [1] がある．文献 [1] はソフトウェアの大規模化や開発期間の短縮化に伴い，全てのモジュールに力を注ぐことは難しいため，力を注ぐべきモジュールを特定する手法を提案している．ソフトウェアの開発履歴に着目し，メトリクス値の変化をもとに力を注ぐべきモジュールを特定する．メトリクスは CK メトリクスを用い，エントロピーやハミング距離，ユークリッド距離などからメトリクス値の変化を観察し，その変化が不安定なものを力を注ぐべきモジュールとしている．

本手法もソフトウェアの概要把握の見当付けるための情報を提供することが目的であり，問題設定は異なっているが関連している．

2.2 詳細動作の理解

詳細動作の理解では，モジュール内部の内容を確認する．ソフトウェアの改良保守の際に，具体的にどのモジュールに変更を加える必要があるかというソースコードレベルでの変更箇所を特定することが目的である．この段階では，開発者は既にソフトウェアの全体像については把握している．

詳細動作の理解支援を行う手法として Feature Location がある．Feature Location は機能と実装を対応付ける，機能レベルでの理解支援を行う手法である．ソフトウェアの全体構造を理解している開発者がプログラムの改良やデバッグを行う際に用いることを想定している．大きく分けて静的手法 [2]，動的手法 [3]，開発履歴を用いた手法 [4] が存在する．

本手法では，構造体のアクセス回数を求めるために様々なコマンドライン実行のテストケースを用いる．本手法と同様にテストケースを用いる動的手法として Dynamic Feature Traces: Finding Features in Unfamiliar Code [3] がある．文献 [3] は TDD (Test Driven Development) で開発されたシステムを対象に，そこで用いられるテストケースを「機能を表現するテストケース」と「機能を表現しないテストケース」に分類する．各テストケースを実行した結果から機能とそれを実装しているコード断片を対応付ける．

3. 構造体とその特徴

3.1 構造体の役割

構造体は C 言語で用いられるデータ型の 1 つであり，複数の型をひとまとめにして扱う機能である．ソフトウェア中の様々なデータ構造を実現するのに用いられる．本論文で述べるデータ構造とは，配列，リスト，キュー，スタックなどの一般的なデータ構造ではなく，特定の問題を解決するために実現されるデータ構造を指す．このようなデータ構造を実現するために構造体は重要な役割を果たしている (**表 1**).

ソフトウェアが扱うデータは，そのソフトウェアが扱う問題領域と対象に変更がない限り変化しない．それに対して，ソフトウェアの振る舞いを実現する方法は保守改良によって変化する．例として，圧縮・展開ツールを考える．圧縮時に入力対象であるデータやプログラム中で保持される符号化データは，ツールが扱う問題領域が圧縮・展開でその対象が任意の入力ファイルである事実が変わらない限り同じである．しかし，圧縮という振る舞いを実現するための方法は圧縮アルゴリズムが変化するとそれに伴い変化する．そのため基本的にソフトウェアの保守改良による大幅な変更の影響を受けにくいデータ構造は，ソフトウェア理解支援に有用な情報である．したがって，提案手法はデータ構造の実現に用いられる構造体を対象とする．

表 1 構造体の利用例

ソフトウェア	構造体により実現されるデータ構造
圧縮・解凍ソフト	ハフマン木
コンパイラ	構文木，記号表
OS	プロセス制御ブロック，ページテーブル

3.2 構造体の分類方法

分類方法には以下のような方法が考えられる．

- ソースコードやドキュメントの調査による分類
- プログラム実行時の挙動観察による分類

ソースコードやドキュメントの調査により分類する方法は，プログラム中の構造体が他のプログラムやライブラリで用いられるような汎用性のある構造体であるか，もしくは，特定の箇所やアルゴリズムでのみ利用されるような専門的な構造体であるかを分類することができる．この手法は，構造体を細かい粒度で分類する場合に有効であると考えられる．

プログラム実行時の挙動観察を行うことで分類する手法は，実行時の構造体の利用状況から分類を行う．特定の入力を与えた場合の構造体の利用状況の変化やプログラムの実行開始から終了までの構造体の利用状況を観察するなどの方法が考えられる．また，利用状況を測る尺度としては以下のようなものが考えられる．

- 変数や型のアクセス回数
- インスタンスの生成数
- インスタンスの生存時間

この方法は構造体を粗い粒度で分類する場合に有効な手段であると考えられる．

3.3 概要把握のための構造体型の特徴付け

プログラムの実行フェーズは大きく「入力–処理–出力」の 3 段階に分割することができる．ソフトウェアの概要把握の際にこれらのフェーズごとに理解を進める方法が考えられるが，そのためには各フェーズの境界を特定しなければならない．しかし，この境界を特定するにはソフトウェ

アの設計書やソースコードを解読する必要がある．

そこで本論文では境界の特定のために構造体型のアクセス回数に着目し，関数の呼び出し文脈に基づいて構造体型のアクセス回数を計測する．関数の呼び出し文脈は，ある関数からの呼び出しがどの経路からの関数呼び出しであるかの把握を可能にする．実行フェーズの境界を求めるために各関数ごとに構造体型のアクセス回数を計測し，その挙動から構造体型を「入力–処理–出力」の各フェーズに特徴付ける．

4. 呼び出し文脈を考慮したアクセス回数の計測

4.1 概要

本手法は次の手順で構造体の特徴付けを行うためのデータを取得する．

(1) 関数呼び出し文脈に基づいたアクセス回数の計測
(2) 計測結果の重ね合わせ

プログラムの「入力–処理–出力」の各フェーズに対応して構造体型のアクセス回数を計測できるように，main 関数から始まる関数の呼び出し文脈に基づいて計測を行う．また，ソフトウェアの概要把握を行うという目的から様々なコマンドラインオプションでの実行を計測し，その結果を重ね合わせる．

正確なアクセス回数の計測を行うためには実行トレースを詳細に解析する必要があるが，本論文では解析の効率と正確さのトレードオフから Sapid [5] による静的解析とラインカバレッジによる動的解析を利用して計測を行う．

4.2 計測方法

図 1 に示すプログラムを例に計測方法を説明する．ソースコードの左側にある数字は実行回数を表す．

4.2.1 関数の再帰呼び出しと繰り返し構文による関数呼び出し

図 1 のサンプルプログラムの実行トレースは**図 2** のようになる．

本手法は計測の後に実行パスの異なる実行結果を重ねる．そのために，関数呼び出しの順序を揃える必要がある．そこで，実際の計測において以下の制御から呼び出される関数はまとめてアクセス回数をカウントする．

- 関数の再帰呼び出し (自己再帰と相互再帰)
- 繰り返しの制御構造 (for, while, do-while)

図 2 の場合，for 文による func0 の呼び出し (2 行目と 3 行目) を 1 つにまとめる．また，func1 の自己再帰の部分 (4〜6 行目) を 1 つにまとめる．相互再帰は呼び出し先が戻って再帰になる部分でまとめる．7〜10 行目は，func2 と func3 の呼び出しの後 func2 に呼び出しが戻るため，func2 と func3 の呼び出しを一組にまとめる．したがって，これらを考慮した関数の呼び出しの順序は**図 3** のようになる．

```
1       1:  int main(void) {
2       -:    int i;
3       -:
4       3:    for (i = 0; i < 2; i++) {
5       2:      func0(i);
6       -:    }
7       1:    func1(2);
8       1:    func2(3);
9       -:
10      1:    return 0;
11      -:  }
12      -:
13      2:  void func0(int i) {
14      2:    printf("func0\n");
15      2:  }
16      -:
17      3:  void func1(int j) {
18      3:    if (j == 0) {
19      1:      printf("func1\n");
20      -:    } else {
21      2:      printf("func1\n");
22      2:      func1(j - 1);
23      -:    }
24      3:  }
25      -:
26      2:  void func2(int k) {
27      2:    if (k == 0) {
28  #####:      printf("func2\n");
29      -:    } else {
30      2:      printf("func2\n");
31      2:      func3(k - 1);
32      -:    }
33      2:  }
34      -:
35      2:  void func3(int m) {
36      2:    if (m == 0) {
37      1:      printf("func3\n");
38      -:    } else {
39      1:      printf("func3\n");
40      1:      func2(m - 1);
41      -:    }
42      2:  }
```

図 1 サンプルプログラム

```
1  => main()
2      => func0(0)
3      => func0(1)
4      => func1(2)
5          => func1(1)
6              => func1(0)
7      => func2(3)
8          => func3(2)
9              => func2(1)
10                 => func3(0)
```

図 2 サンプルプログラムの実行トレース

4.2.2 変数のアクセス回数の計測

本手法では，計測の方法としてラインカバレッジを用いる．ラインカバレッジは，プログラム実行時にソースコード中の各行が何回実行されたかを測る．ラインカバレッジから得られる情報を次に示す．

- コードのカバレッジ
- 実行された行の回数

ラインカバレッジでは，ソースコード中の各文がどの関数からの文脈で実行されたかを知ることはできない．正確に

```
1  => main()
2      => func0()
3
4      => func1()
5
6
7      => func2()
8          => func3()
9
10
```

図 3 再帰呼び出しと繰り返し構文を畳み込んだトラバース

```
=> main()
    => func0()
<= main()
    => func1()
<= main()
    => func2()
        => func3()
    <= func2()
<= main()
```

図 4 関数の呼び出し前後を区別したトラバース

アクセス回数をカウントする方法としてはデバッガやアスペクト指向を利用した方法が考えられる．デバッガの場合は，1 回のプログラムの実行で監視できる変数のサイズがアーキテクチャに依存するため，すべての構造体型変数に対してアクセス回数を計測するには膨大な時間がかかる．また，アスペクト指向を用いる場合は，ログ取得をするためのアスペクトをソースコードに埋め込む必要があり，この作業に対するコストが大きい．

したがって，本手法ではラインカバレッジによる行の実行回数と関数の呼び出し回数から各文の実行回数を計算する方法を選択した．実際の計測方法については 5.2 節で述べる．

4.2.3 関数の呼び出し前後を考慮した計測

プログラムの実行フェーズである「入力–処理–出力」を考慮して関数ごとに構造体のアクセス回数を計測するために，関数の呼び出しの前後を区別することで図 3 をよりも詳細に計測する．図 3 の main 関数から func0 の呼び出しについて考える．この場合，main の呼び出し時に main に含まれるすべての文が実行されるわけではない．実際のプログラムの実行の流れでは func0 の呼び出し直前までが実行される．したがって計測を行う．その他の関数も呼び出し前後を区別すると**図 4** のようになる．

4.3 時間軸の重ね合わせ

関数の呼び出し文脈に基づいて構造体型のアクセス回数を計測した後に，その結果を重ね合わせる際，時間軸について考慮しなければならない．例えば，if 文により実行の制御が分岐する場合，true ブロックと false ブロックが同じ実行パスで実行されることはないため，単純に実行パスを重ね合わせることはできない．したがって，本論文では全体構造を把握するという観点から，UML のシーケンス図の複合フラグメント (**図 5**) を参考にする．

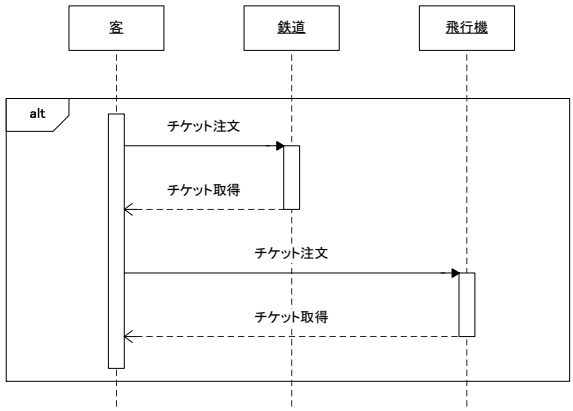

図 5 複合フラグメントを用いたシーケンス図

シーケンス図は，オブジェクト指向開発において，オブジェクト間のメッセージの関係を時系列で図示したものである．複合フラグメントは，シーケンス図内で制御構造を表すときに用いられる表記法である．制御の分岐や繰り返しを時間軸の中に組み込むことで，オブジェクト間の相互関係の概要を把握することが可能となる．本論文もこれを基に制御構造を含めて，プログラムの実行結果を重ね合わせることにする．

5. 適用実験

5.1 実験方法

gzip-1.2.4 [6] と tar-1.26 [7] を対象に 4 節で述べた手法を用いて，構造体型の特徴付けが可能かを観察するために実験を行った．プログラム中の各構造体型が各関数ごとに何回アクセスされたかを計測する．プログラムの実行結果を重ね合わせるときに基本的に次のような場合を計測対象とした．

- 単独で使用できるオプションは，単独で使用する
- 他と組み合わせる必要があるオプションの場合は，できるだけ少ないオプションで済むように選択する
- 計測の対象とする構造体型変数
  - グローバル変数
  - ローカル変数
  - 関数の引数

5.1.1 コマンドラインスイッチの指定について

あるプログラムにおけるオプションは，複数個組み合わせて使用できる場合が多い．しかし，そのすべてを計測の対象とするとオプションの組み合わせの数が爆発的に増加する，少なくとも各オプションを 1 度は指定してプログラムを実行するように上記の条件を設定した．

5.1.1.1 gzip のオプション

gzip-1.2.4 は 24 個のオプションを持つ．そのうち主要な 13 個のオプションを実行の対象とした．そして，対象のオプションから上記の条件に従い 17 通りを実行した．具体的には，`-r` オプション (指定先ディレクトリを再帰的に探索し，すべてのファイルを圧縮，展開) と `-S` *suf* オプション (*suf* を拡張子として認識する) では圧縮と展開を実行し，`-[1..9]` オプション (圧縮レベルの指定) は `-1` と `-9` を実行した．また，オプション指定なしでの実行を含め，18 通りをアクセス回数の計測対象とした．

5.1.1.2 tar のオプション

tar-1.26 は 135 個のオプションを持つ．tar の場合は各オプションを 1 度だけ実行するようなケースを作成すると何度も同じオプションが実行される．tar の一般的な使用によって実行されるコマンドは以下の 5 つである．

- アーカイブの生成 (`-c`)
- アーカイブからのファイル削除 (`--delete`)
- アーカイブへのファイル追加 (`-r`)
- アーカイブ内の一覧表示 (`-t`)
- アーカイブからのファイル抽出 (`-x`)

この 5 つのコマンドを基に一般的に利用されるオプションである gzip や bzip2 を指定してアーカイブの生成，抽出を行うオプションの組み合わせを選択し 11 通りをアクセス回数の計測対象とした．

5.2 アクセス回数の計測

4.2.2 節で述べたように本手法ではラインカバレッジにより変数のアクセス回数を計算する．そのため，本実験では文脈の違いによる実行の誤差について実行回数の平均値を取ることで近似する．例として，サンプルプログラム (図 1) の func1 (8 行目) の呼び出しにおける変数 j の読み取り回数の求め方を**図 6** に示す．

本実験では，ラインカバレッジの計測に，C 言語のコードカバレッジツールである gcov [8] と gcovr [9] を使用した．

5.3 gzip-1.2.4 に対する適用結果

gzip-1.2.4 に対する適用結果を**図 7** と**図 8** に示す．「入力–処理–出力」というプログラムの実行フェーズに対して，構造体型の特徴付けが可能かを考察するために，予めグラフの背景に処理に該当する範囲に対して色を付けている．水色になっている部分は圧縮に関係する関数で，橙色になっている部分は圧縮に関係する関数である．縦軸は各構造体型の各関数の中での利用比率である．横軸は 4 節で述べた関数呼び出し文脈を考慮した関数呼び出しトラバースの順に関数が列挙されている．

$$
\begin{aligned}
&\text{変数 j の読み取り回数} \\
&= \text{func1 の呼び出し回数} \times \frac{\text{変数 j の読み取り実行回数}}{\text{func1 の被呼び出し回数}} \\
&= \text{8 行目} \times \frac{\text{19 行目} + \text{23 行目}}{\text{18 行目}} \\
&= 1 \times \frac{3+2}{3} \\
&= 1.67 \text{ 回}
\end{aligned}
$$

図 6 変数 j の読み取り回数の計算例

表 2 gzip の構造体型

構造体型名	実行フェーズ	機能
`stat`	入力，出力	ファイル処理
`option`	入力	オプション処理
`config`	処理	圧縮
`tree_desc`	処理	圧縮
`ct_data`	処理	圧縮
`huft`	処理	展開

gzip には 6 種類の構造体が定義されており，「入力–処理–出力」で分類すると**表 2** になる．

5.3.1 考察

図 7 に関して，ファイル処理に関する構造体である `stat` はグラフの序盤と終盤にのみ出現しているため，プログラムの実行フェーズの開始と終了部分でのみ利用されていることが分かる．

オプション処理に関する構造体である `option` はグラフの序盤で出現しているため，プログラム実行フェーズの序盤であると判断できる．

圧縮処理に関する構造体である `tree_desc` と `ct_data` はグラフの中盤で何回か利用されているため，プログラムの実行フェーズの中盤で利用されていることが確認できる．また，`config` は圧縮処理の序盤で利用されている．この構造体は圧縮時のパラメータを保持する構造体である．プログラム中で一度参照された後は利用されず，グラフ中でも 1 つの関数でのみ利用されており，利用率も 1.0 となっている．

展開に関する構造体である `huft` はグラフ上では終盤に位置している．これは 4 節で述べた関数呼び出しのトラバースで計測していることと繰り返し構文をまとめていることが原因である．gzip の場合 1 回分の実行で圧縮と展開のどちらか一方が実行される．しかし，計測方法は関数呼び出しのトラバースを用いており，グラフ上の関数の並び順は圧縮，展開となる．また，展開処理では主に 1 つの関数 `inflate_codes` を繰り返しによって何度も実行しており，畳み込みによって 1 つにまとめているため圧縮に比べて極端にグラフの横軸が狭くなっている．

図 8 は図 7 と同じ情報をアクセス回数の変化をより明確にするために累積グラフで表したものである．ファイル処理の構造体 `stat` は，序盤と終盤でアクセス回数が急激に増加している．オプション処理 `option` と圧縮パラメータ

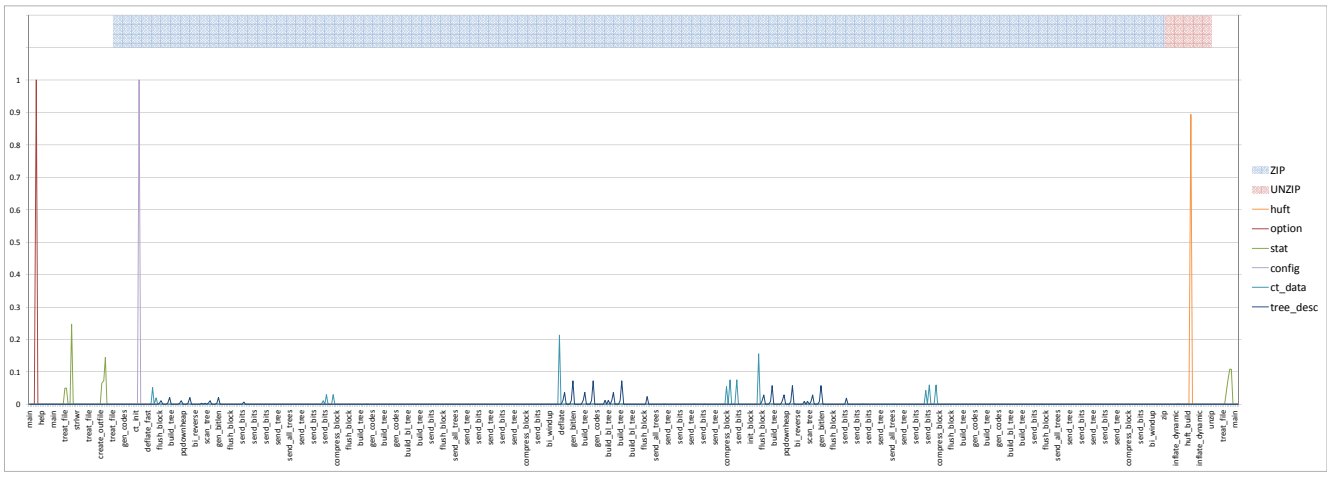

図 7　gzip-1.2.4 に対する実験結果

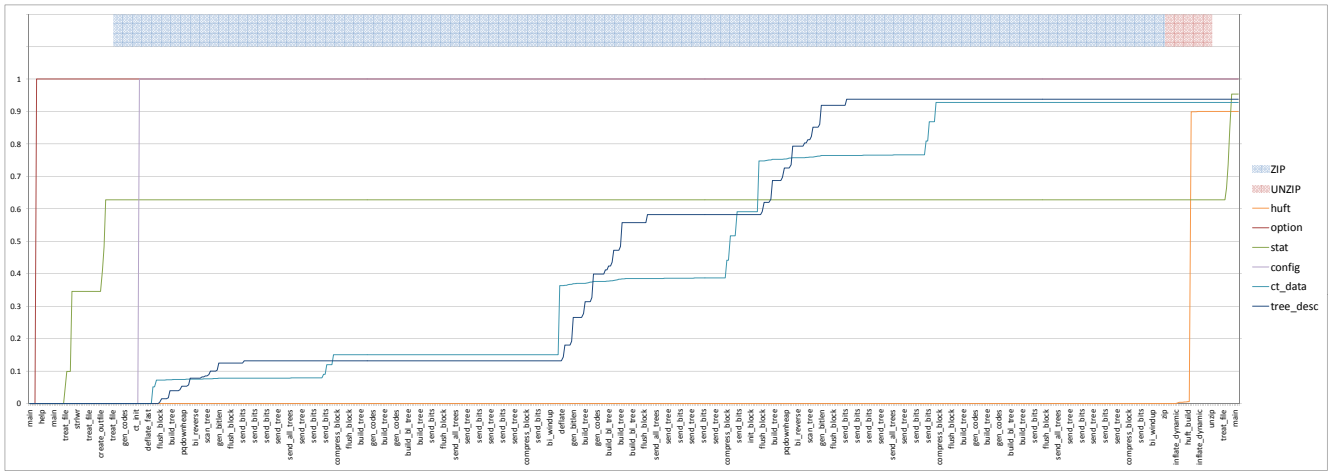

図 8　gzip-1.2.4 に対する実験結果 (累積グラフ)

用の構造体 `config` も序盤で急激に増加している．また，圧縮に関する構造体である `tree_desc` と `ct_data` はグラフの中盤で徐々に増加することが分かる．

展開に関する構造体である `huft` は図 7 と同様に関数呼び出しトラバースと繰り返し構文の畳み込みによって，終盤に実行されているように見える．

累積グラフは全体的な処理の流れがどの時点で切り替わるのかが把握しやすい．

5.4　tar-1.26 に対する適用結果

tar-1.26 に対する適用結果を**図 10** と**図 11** に示す．gzip-1.2.4 と同様な理由からグラフの背景に色を付けている．赤色はアーカイブへの追加，緑色はアーカイブからのファイル削除，紫色はアーカイブの生成，水色はアーカイブからのファイル抽出，橙色はアーカイブの一覧表示である．グラフの縦軸と横軸については gzip の適用実験結果のグラフと同じである．

tar のソースコード中で定義された関数 (`src` ディレクトリ) が tar 中で定義された関数以外から呼ばれることがある．make 時に gnu ディレクトリ内に gnu のライブラリを生成する．プログラム実行時は，そのライブラリから tar 定義の関数を呼ぶ場合がある．今回，Sapid で解析しているのは `src` ディレクトリあり，tar 中で定義されたソースコード部分であるため，ライブラリ側から `src` ディレクトリの関数を呼んでいる場合に関しては正確に構造体のアクセス回数を計測することができない．したがって，その部分に影響を受けていない構造体ついてのみ考察を行う．

tar では 39 種類の構造体が定義されている．tar の中心的な処理であるアーカイブの生成，追加などの機能で利用される構造体は `tar_stat_info` である．`tar_stat_info` は *.tar* 形式のファイルの情報を保持している．また，解析を行った範囲が不十分であることに影響を受けていない構造体も合わせて**表 3** に示す．

5.4.1　考察

図 10 について，`link` は実行フェーズの中盤であるアーカイブの追加と生成の箇所で利用されている．

`bufmap` はプログラム実行の中盤の中盤であるアーカイブの追加，削除，生成，抽出，リスト表示の箇所で使用さ

表 3 tar の構造体型 (一部)

構造体型名	実行フェーズ	機能
tar_stat_info	入力，処理，出力	アーカイブの生成，追加，削除，抽出，リスト表示
bufmap	処理	マルチボリュームの情報を保持
deferred_unlink	処理	削除するファイルリストを保持
keyword_list	入力	キーワードを保持
link	処理	ファイルのリンク数を保持

れているため，実行フェーズの中盤であると判断できる．

keyword_list はプログラム実行の中盤であるアーカイブの生成，削除，抽出，リスト表示の箇所で使用されているため，実行フェーズの中盤であると判断できる．

deffered_unlink はプログラム実行の中盤であるアーカイブの追加と生成の箇所で使用されているため，実行フェーズの中盤であると判断できる．

図 11 の累積グラフでは，tar_stat_info についてはアクセス回数を累積した結果が 1.0 にならない．これは先に述べた tar のソフトウェアの構成と本手法の解析範囲が原因である．

bufmap, defered_unlink, keyword_list, link は全体的に中盤から階段状にアクセス回数の利用比率が増加している．tar_stat_info については，不十分な解析ではあるが，アーカイブの抽出とリスト表示の部分で徐々に増加していることが確認できる．tar_stat_info は tar の中心的な処理のすべてで利用されるため，十分な解析を行った上で実験を行えば，アーカイブの追加，削除，生成，抽出，リスト表示の部分でそれぞれ徐々に増加する箇所が現れると考えられる．

5.5 適用実験のまとめ

gzip については，ファイル処理を扱う stat と圧縮処理を扱う tree_desc と ct_data の部分で「入力–処理–出力」という実行フェーズを大まかに分割できる箇所が見られた．しかし，一部の構造体型については実行フェーズの分割箇所を特定するには不十分な結果となるものも存在した．展開処理を扱う huft は，処理部分に該当する構造体であるが，グラフの終盤で利用されている．

tar については，今回示した解析結果に依存しない構造体については大まかに分割できる箇所が見られた．すべての関数呼び出しについて十分に解析ができていないため再度プログラムの解析をする必要がある．解析が不十分である原因は，tar のソフトウェア構成にある．今回は tar が定義しているソースコードを対象に解析を行った．しかし，実際にはライブラリ関数から tar 定義関数を呼んでいる箇所が存在するため呼び出し文脈をすべて考慮できていない．したがって，tar 全体として解析を行うことで，すべての文脈を計測可能にする必要がある．

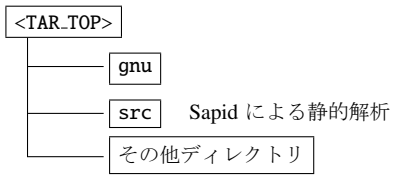

図 9 tar-1.26 のディレクトリ構造 (一部)

6. おわりに

本論文では，ソフトウェアの概要把握を支援するために，C 言語を対象として構造体型を特徴付ける手法を提案した．関数の呼び出し文脈を考慮し，関数の呼び出し前後で各関数ごとにデータ流量から構造体型のアクセス回数を計測することで，構造体型を特徴付けるための実験を行った．2 つのソフトウェアに本手法を適用し，gzip-1.2.4 については「入力–処理–出力」というプログラム実行フェーズに対して大まかに分割できる可能性を結果として見ることができた．tar については，手法の実装が不充分であったため，呼び出し文脈の一部について計測が行えなかった．この点に関してツールを改良してより正確な計測を実現することを今後の課題とする．

6.1 今後の課題

今後の課題として 4 つ挙げる．1 つ目に，今回は 2 つのソフトウェアに対して提案手法を適用したが，他のソフトウェアに対しても適用し本手法の有効性を検証する必要がある．2 つ目に計測方法の妥当性の確認である．関数の呼び出し文脈を考慮した関数呼び出しトラバースにより構造体型のアクセス回数の計測を行ったが，異なる時間軸を合わせて計測を行っている．複数の計測結果を合わせる前の結果と比較し，誤差があるのかを確認する必要がある．3 つ目にアクセス回数以外のパラメータを用いて構造体型のデータ流量を観察することである．パラメータの候補はいくつか考えられるため，それらを用いて計測することでソフトウェア理解に有用であるかを検討する．最後に，他言語への応用である．今回は構造体型の特徴付けを行ったが，概念が近いオブジェクト指向のクラスに関して本手法を拡張できるかを検討する．

謝辞 本研究は科研費 22300011, 24300006 の助成を受けたものである．

参考文献

[1] 村尾憲治，肥後芳樹，井上克郎：ソフトウェアメトリックス値の変遷に基づいた注力すべきモジュールを特定する手法の提案 (ソフトウェア工学)，電子情報通信学会論文誌. D, 情報・システム， Vol. 91, No. 12, pp. 2915–2925 (2008).

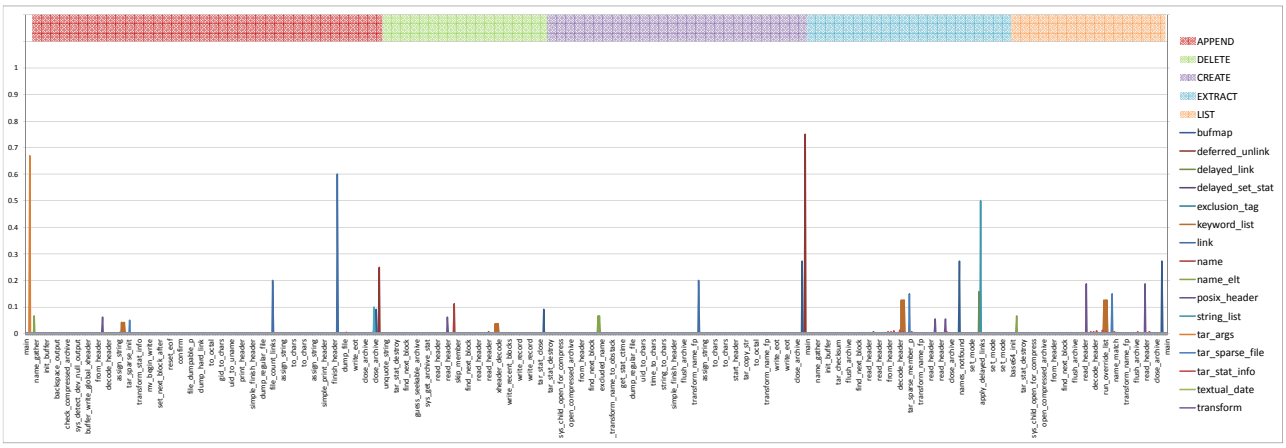

図 10 tar-1.26 に対する実験結果

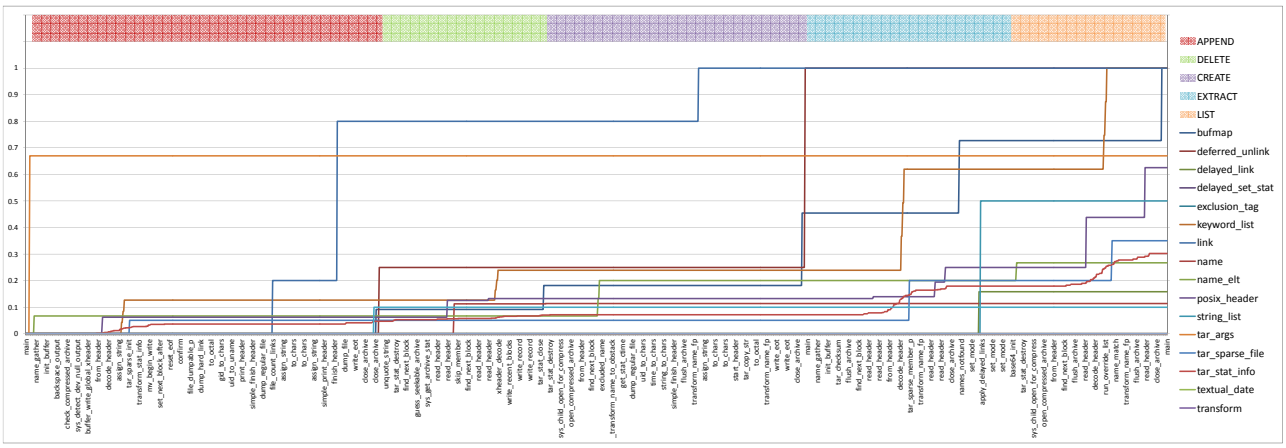

図 11 tar-1.26 に対する実験結果 (累積グラフ)

[2] Kunrong Chen and Va'clav Rajlich: Case Study of Feature Location Using Dependence Graph, *In Proceedings of the 8th International Workshop on Program Comprehension*, IEEE Computer Society, pp. 241–249 (2000).

[3] Eisenberg, A. D. and Volder, K. D.: Dynamic Feature Traces: Finding Features in Unfamiliar Code, *International Conference on Software Maintenance*, pp. 337–346 (2005).

[4] 楠田泰三：メソッドの同時更新履歴を用いたクラスの機能別分類法，修士論文，大阪大学大学院情報科学研究科 (2006).

[5] 福安直樹, 山本晋一郎, 阿草清滋：細粒度リポジトリに基づいた CASE ツール・プラットフォーム Sapid (〈特集〉並列処理)，情報処理学会論文誌， Vol. 39, No. 6, pp. 1990–1998 (1998).

[6] gzip – GNU Gzip: "`http://www.gnu.org/software/gzip/`".

[7] Tar: "`http://www.gnu.org/software/tar/`".

[8] gcov – a Test Coverage Program: "`http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`".

[9] gcovr – simplified gcov reporting: "`https://software.sandia.gov/trac/fast/wiki/gcovr`".