大規模ソフトウェアの概要把握支援のための動的解析結果の静的解析を用いた一元化

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 プログラムに対して，利用している構造体型を特徴付ける手法を提案する．大規模なソフトウェアを理解する初期段階において，構成要素を役割に基づいて分類することは有用である．構造体型に関する利用頻度は，プログラムの実行段階により変動する．この変動を可視化することで，構造体型のプログラム中における役割が推測できる．そこからプログラムの構成要素が分類できる．2 つのソフトウェアによる適用実験から提案手法の検証を行った．

Summarization of Dynamic Analysis Result using Static Analysis for Perspective Software Comprehension

Isao Takeji[1,a)]　Hirotaka Ohkubo[2,b)]　Hideto Kasuya[2,c)]　Shinichiro Yamamoto[2,d)]　Kunihiko Saito[3,e)]

Abstract: This paper characterizes the use of struct types in C programs. It is useful to classify software components by their roles in the initial phase of large-scale software comprehension. Frequency of use of the struct types is not steady but varies according to execution phases. By visualizing these fluctuation, the roles of struct types in a program can be inferred, and the components of the program can be classified. We present two experimental results using open-source software to evaluate our approach.

1. はじめに

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．開発プロセスでは，生産性と信頼性の向上を目的として既存ライブラリを利用したりソフトウェア部品を再利用する．その際に開発者は使用するライブラリや部品を理解し，その挙動が目的に適しているのかを確認する必要がある．保守プロセスでは，保守対象のプログラムを理解することによりどのような機能を持ち，どのような振る舞いをするのかを把握しなければならない．このようにソフトウェア理解はプログラムの品質改善や機能拡張に先立って行われる．そのため，理解の度合いが不十分であると不必要な変更や新たなバグの混入など成果物の品質に悪影響を及ぼす可能性がある．

また一方で，近年ソフトウェアの大規模化・複雑化によりソフトウェア理解はより困難なものになっている．ソースコードやドキュメントが増加しているために現実的な時間とコストでソフトウェア全てを理解できる状況ではない．そのため，開発者は改良・保守に必要な部分を特定するために，大規模で複雑になったソフトウェアの理解に必要な箇所を求めることになる．したがって，ソフトウェア

[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

理解を行う開発者に対して，それを支援するための手掛かりを提供することが必要となる．

本論文の目的は，ソフトウェアの概要把握を行う理解の初期段階にある開発者に対して，その手掛かりとして構造体型を特徴付ける手法を提案することである．本論文の貢献は次の 2 点である．

- 動的解析による複数の実行履歴を統一的に扱うために，静的解析を用いて**時点軸**という概念を提案した．それにより，動的解析の結果を大局的に把握することができる．
- 構造体型を利用頻度に基づいて特徴付けることにより，開発者はソフトウェアの機能を分類し，概要を把握することが可能になる．

概要把握では機能を大まかに分類しソフトウェアの全体像を捉えることを目的とする．そのため，機能と関連が高い構造体型を本手法によって特徴付けることで，概要把握に必要な機能分類の支援になると考えられる．プログラム実行時の構造体型の利用頻度から特徴付けを行う．提案手法では，代表的な手続き型言語である C 言語を対象とする．C 言語への本手法の適用により，その後継であるオブジェクト指向言語へ本手法を拡張できる可能性も考えられる．

2. ソフトウェアの概要把握

2.1 ソフトウェア理解の概要

ソフトウェア理解は概要把握と詳細動作の理解の 2 つの工程によって進められる．

概要把握　モジュールの役割とその入出力の把握，モジュール間の相互関係についての把握

詳細動作の理解　ソースコードの解読，プログラムの実行による挙動の観察

ソフトウェア理解の初期段階にある開発者は，対象ソフトウェアの全体像を捉えるために概要把握をする必要がある．概要把握では，モジュールの振る舞いを把握し大まかな構造を掴む．モジュール内部の理解に先立ちどこを理解すべきかを検討したり，改良や変更を加えるための見当を付けることが目的である．この段階にある開発者は，理解対象のソフトウェアに対する知識が少ない．詳細動作の理解はモジュールの内部について確認をする．ソースコードレベルで具体的な変更を加える箇所を特定することを目的とする．この段階ではソフトウェアの全体構造については把握済みである．

2.2 概要把握とモジュールの分類

理解するための手掛かりをほとんど持っていない状況では，モジュールを大まかに分類することで概要把握せざるを得ない．概要把握できるようなモジュールの分類方法として考えられるのは，STS 分割に代表されるように**図 1** のような形で各モジュールを実行フェーズに分類することである．STS 分割ではプログラムを入力，変換，出力という 3 つの役割に分割する．モジュールを実行フェーズに分類することができれば，モジュールの役割や相互関係の把握やその後の詳細動作の理解に有用である．

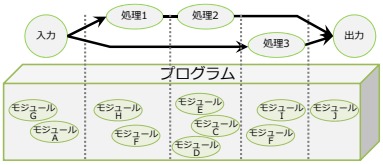

図 1　モジュールの分類
Fig. 1　Classification of modules

3. 構造体型について

3.1 構造体の役割

構造体型は C 言語で用いられるデータ型の一つである．複数のデータ型をひとまとめにして扱うことができ，プログラム中の様々なデータ構造の実現に用いられる．データ構造は大きく分けて，配列，リスト，キュー，スタックなどの一般的なものと**表 1** に示すような特定の問題を解決するために実現されるものに分けることができる．本論文では特定の問題を解決するためのデータ構造を対象とする．

3.2 構造体型と機能の関係について

プログラムはデータ構造とそれに対する手続きの繰り返しである．その関係を表すと**図 2** のように考えることができる．図中で，円のノードは手続きである．データを生成する手続きを生産 (P: Producer)，データを消費する手続きを消費 (C: Consumer)，データを変更する手続きを変更 (M: Modifier) と呼ぶ．四角のノードはデータ構造である．本論文は C 言語を対象としており，データ構造の実装には構造体を用いるため四角のノードは構造体としている．

例として圧縮の機能を表す青色で囲まれた箇所に着目する．ここではハフマン木という構造体に対して生成，更新，消費を行う．これらの手続きは圧縮の一部であるが，それぞれの手続きは独立しているため手続き間の関連は低い．一方，ハフマン木の構造体は生成，更新，消費のすべての手続きと関連があることが分かる．個々の手続きは独立していても，構造体を通じてまとまった 1 つの機能として捉えることができる．したがって構造体は手続きよりも機能の推測がしやすいと考えられる．そのため提案手法は理解支援に構造体型を用いる．

表 1　構造体の利用例
Table 1　Example of structures

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

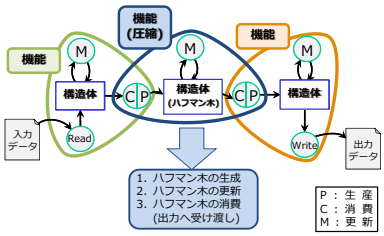

図 2　手続きと構造体の関係
Fig. 2　Relation of procedure and struct

4. 特徴付けのための構造体型の利用頻度の計測

機能が構造体型と対応すると考えると，実行する機能により構造体型ごとの利用頻度が異なるという仮説を立てることができる．複数の機能を持つプログラムに対し，その中のある機能を実行すれば，結果としてその機能に関連する構造体型が他の構造体型に比べて多く利用され，それ以外の構造体型は利用頻度が低いか全く利用されないことが予想できる．したがって，各機能には構造体型の利用頻度にそれぞれ特徴があると考えられる．また，概要把握のためにモジュールを各実行フェーズへ分類するには，機能ごとに構造体型の利用頻度が異なるという仮説から，実行フェーズごとに各機能の利用頻度の推移に特徴が現れると考えることができる．これらの理由から本手法では構造体型の利用頻度に着目して，実行フェーズへ構造体型を特徴付ける．

4.1 本手法のアプローチ

本論文では概要把握のために構造体型の利用頻度を用いて構造体型を実行フェーズに特徴付ける．利用頻度をここでは構造体型のアクセス回数とする．

アクセス回数を計測するために動的解析によりプログラムの実行履歴を取得する必要がある．プログラムは様々な構成要素からなり，ある 1 回の実行ではその一部しか利用されない．すなわち，1 つの実行履歴から得られる構造体型のアクセス回数は，プログラムのある側面での機能という断片的な情報しか表さない．プログラムの全ての構成要素を調査するためには，網羅的に条件を変えて実行履歴を取得する必要がある．実行条件を変えるには入力データとコマンドラインオプションの 2 つを操作する．入力は種類が無限に存在すると考えられるので，本手法ではコマンドラインオプションに限定して実行履歴を取得することで網羅性を確保する．これは入力ファイルのサイズやファイル形式はその種類に際限がないが，あるプログラムが持つコマンドラインオプションは有限であるため，実行履歴の取得にそれほどコストを掛けることなく網羅性を確保できるからである．

コマンドラインオプションは有限個であるが，網羅的な実行の数はその組み合わせにより指数的に増大する．それら全ての実行履歴を個々に調査するのは難しい．そこで本手法は，すべての実行履歴をひとまとめにする．一つ一つの実行履歴は断片的な情報であるが，網羅的な実行により得られたものである．それらをまとめることで対象ソフトウェアの全ての実行履歴の情報を俯瞰的に持つと考えられる．そのために，ステップ数が異なる実行履歴をどのようにまとめるかを考えなければならない．すべての実行履歴を統一した基準で扱うために**時点軸**という概念を提案する．時点軸は静的解析によって導出する．

異なる実行ステップを統一して扱うために，時点軸には以下に示す性質が要求される．

- 関数呼び出しの深さと回数の統一
- 分岐制御の区別
- 関数の呼び出し文脈を考慮

実行が異なれば，関数の呼び出しの深さや回数は異なるのでどのような実行でも同じように扱うために何らかの形で統一する必要がある．if 文や switch 文などの分岐制御は実行毎に選択される処理が異なるため，これを区別する必要がある．また，関数はどのような文脈を辿って呼び出されたのかという情報も各実行によって変わるためこれも考慮する必要がある．したがって，時点軸とは再帰による関数呼び出しと繰り返しを考慮しない静的な文脈である．

4.1.1 時点軸の導出

時点軸は次の方法によって導出する．

(1) プログラムのすべての関数呼び出しをインライン化
(2) `main` 関数の 1 行目からのすべての行を列挙

はじめに，プログラム中の関数呼び出しを全てインライン化する．**図 3** をインライン化すると**図 4** のようになる．これにより，複数個所から呼び出されている関数定義は複製される．図 3 で関数 `g1` は関数 `f1` と `f2` から呼ばれているため，2 か所インライン化される．また，関数呼び出しの深さと回数を統一するために，再帰による関数呼び出しのようなコールグラフが循環する箇所については展開を行わない．図 3 では関数 `g2` が再帰呼び出ししており，その部分についてはインライン化しない．

次に，インライン化後の `main` 関数の 1 行目から順に行を列挙したものが時点軸になる．分岐制御の区別には，各行を順に列挙するため自然に区別することが可能である．また，全ての関数呼び出しをインライン化しているため，関数の呼び出し文脈を考慮している．これにより生成される時点軸は**表 2** になる．表 2 はインライン化した後のソースに対して，インライン化前の行番号が対応している．

4.1.2 実行結果と時点軸の重ね合わせ

4.1.1 節で導出した時点軸に対して，対象プログラムの実行履歴を重ねる．時点軸はプログラムに対して静的に一意に定まるソースの行を列挙したものであるので，各実行履歴の実行ステップを統一して扱うことができる．例として 3 つの実行履歴を時点軸の上に重ね合わせたものを**図 5** に

```
1  main() {
2    ...
3    f1();
4    ...
5    f2();
6    ...
7  }
8
9  f1() {
10   ...
11   g1();
12   ...
13 }
14
15 f2() {
16   ...
17   if () {
18     g1();
19     ...
20   } else {
21     g2();
22   ...
23   }
24   ...
25 }
26
27 g1() {
28   ...
29 }
30
31 g2() {
32   ...
33   g2();
34   ...
35 }
```

図 3　サンプルソース

Fig. 3　Sample source

```
main() {
  ...
  {
    ...
    {
      ...
    }
    ...
  }
  ...
  {
    ...
    if () {
      {
        ...
      }
      ...
    } else {
      {
        ...
        g2();
        ...
      }
      ...
    }
    ...
  }
  ...
}
```

図 4　インライン化したサンプルソース

Fig. 4　In-lined sample source

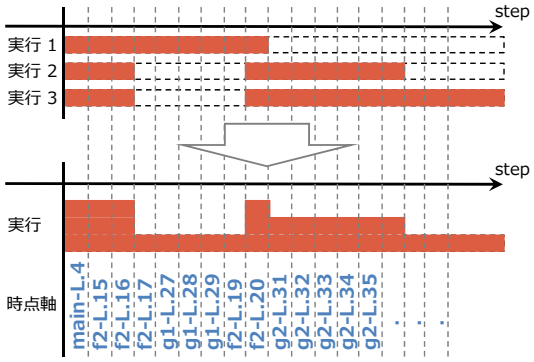

図 5　実行と時点軸の重ね合わせイメージ

Fig. 5　Picture of aggregating runs by the sequentialized time axis

示す．各実行履歴はステップ数は異なるが，ステップごとに時点軸に対応付けることができる．各実行履歴における各構造体型のアクセス回数をそれぞれ時点ごとに重ね合わせて集計することで，網羅的な実行における構造体型の重ね合わせたアクセス回数を計測する．計測の対象とするのは構造体の生成や参照など，構造体型変数へのアクセス全てを対象とする．

5. 適用実験

5.1 実験方法

gzip-1.2.4[4] と tar-1.26[7] を対象に 4 節で提案した手法を用いて，構造体型の特徴付けの実験を行った (**表 3**)．本

表 2　時点軸

Table 2　Sequentialized time axis

時点軸		インライン化後ソース	呼び出し文脈
関数名	行番号		
main	1:	main() {	
	2:	...	
f1	9:	{	main
	10:	...	
g1	27:	{	main → f1
	28:	...	
	29:	}	
f1	12:	...	main
	13:	}	
main	4:	...	
f2	15:	{	main
	16:	...	
	17:	if () {	
g1	27:	{	main → f2
	28:	...	
	29:	}	
f2	19:	...	main
	20:	} else {	
g2	31:	{	main → f2
	32:	...	
	33:	g2();	
	34:	...	
	35:	}	
f2	22:	...	main
	23:	}	
	24:	...	
	25:	}	
main	6:	...	
	7:	}	

実験では提案手法による分類と人手による分類により構造体型を実行フェーズに分類する．そして，分類結果を比較することで提案手法による特徴付けの有効性を観察する．

提案手法による分類

(1) 構造体型のアクセス回数を計測し，時点軸の上に集約しグラフを作成

(2) 作成したグラフの形から構造体型を各実行フェーズに分類

人手による分類

(1) ソースを読み，各構造体型を実行フェーズに分類
今回の実験では，著者が分類を行った

5.1.1 アクセス回数計測の条件

プログラムに対する実行を網羅的に行うために様々なコマンドラインオプションを選択してプログラムを実行する．一般的に，あるプログラムに対するコマンドラインオプションは，複数個を組み合わせて指定できる場合が多い．しかし，全ての組み合わせを網羅するようにオプションを選択すると組み合わせの数が爆発する．そのため，各オプ

表 3　実験対象のプログラム

Table 3　Experimental program

プログラム	LOC	構造体型の数
gzip-1.2.4	約 6,000	6
tar-1.26	約 17,000	39

ションが少なくとも 1 度は指定して実行されるように選択する．したがって，次の方針に該当するオプションをアクセス回数の計測対象とする．

- 単独で使用可能なオプションはそのまま使用する
- 単独で使用できない場合は，最小限の組み合わせになるようにオプションを選択する

また，計測対象とする構造体型変数はグローバル変数，ローカル変数，関数の引数とし，メンバ変数が構造体型であるものについては計測の対象外とした．

5.1.2 gzip-1.2.4 のオプション指定

gzip-1.2.4 は全部で 24 個のオプションが存在する．その中で主要な 13 個のオプションを実行の対象とした．そして，方針に従い 17 通りのオプションとオプションを指定せずに実行する場合を含む計 18 通りをアクセス回数の計測対象とした．次のオプションについては圧縮と展開をそれぞれ実行した．

- 圧縮・展開時に同名のファイルが存在する場合に強制的に上書きする (`-f`)
- 指定したディレクトリ内のファイルを再帰的に圧縮・展開する (`-r`)
- *suf* を拡張子として認識する (`-S` *suf*)

また，`-[1..9]` オプション (圧縮レベルの指定) は`-1` と`-9` を実行した．

5.1.3 tar-1.26 のオプション指定

tar-1.26 は全部で 135 個のオプションが存在する．tar では大半のオプションが別のオプションと組み合わせて使用される．そのため，各オプションを 1 度だけ実行するような選択をすると何度も選択されるオプションがいくつか存在する．また，特殊な場面で使用されるオプションも数多く存在する．したがって，tar を使用する際に一般的に選択されるアーカイブを操作するオプションを基にする．

- 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を抽出 (`-x`)

この 5 つのオプションと頻繁に組み合わせて使用される次のオプション

- アーカイブの生成・抽出を行う際に gzip を使用 (`-g`)
- アーカイブの生成・抽出を行う際に bzip2 を使用 (`-j`)
- アーカイブ対象のファイルを選択 (`-f`)
- 処理中のファイルの詳細を表示 (`-v`)

を組み合わせて 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ツールについて

本実験を行うために実装したツールについて述べる．本ツールは静的解析により時点軸を導出し，動的解析によりコードカバレッジを取得することで，各時点の構造体型のアクセス回数を計測する．静的解析には細粒度ソフトウェアリポジトリに基づいた CASE ツール・プラットフォームである Sapid[9] を用いた．また，動的解析には C 言語のコードカバレッジツールである gcov[2] と gcovr[3] を使用した．gcov はコンパイル時にプロファイルに必要なコードを埋め込み，プログラム実行時にソースの各行が何回実行されたかを計測することができる．また，gcovr は gcov で得られるカバレッジの情報を XML 形式に変換する Python スクリプトである．これらの解析結果から時点軸に沿って構造体型のアクセス回数を計測するツールを実装した．

```
1    3:   void f() {
2    3:     g();
3    3:     h();
4    3:   }
5    -:
6    6:   void g() {
7    6:     printf("Call g\n");
8    6:   }
9    -:
10   3:   void h() {
11   3:     g();
12   3:     printf("Call h\n");
13   3:   }
```

図 6 ラインカバレッジの例
Fig. 6 Example of line coverages

5.2.1 文脈の違いによる行の実行回数

本手法ではラインカバレッジを用いて構造体型のアクセス回数を計測する．ラインカバレッジはプログラムの実行結果を行ごとに集約するため，文脈の違いを考慮しない．そのため，本実験では文脈の違いによって生じる実行の誤差は実行回数の平均を取ることで近似する．文脈を考慮した各行の実行回数は次のように求める．

$$\text{関数 f の } l \text{ 行目の実行回数} = \frac{\text{文脈を考慮した関数 f の呼び出し回数} \times l \text{ 行目の実行回数}}{\text{関数 f の被呼び出し回数}}$$

例として，**図 6** のラインカバレッジを挙げる．ソースの左側の数字は，プログラム実行時に各行が何回実行されたかを表す．ここで 7 行目の実行回数を文脈を考慮して計算することを考える．関数 g は，f からの呼び出しと f → h という文脈からの呼び出しの 2 通りある．f → h から呼び出された場合の関数 g の 7 行目は以下のように計算する．

$$\begin{aligned} &\text{f} \to \text{h という呼び出し文脈における関数 g の 7 行目の実行回数} \\ &= \frac{\text{文脈 f} \to \text{h からの g の呼び出し回数} \times 7 \text{ 行目の実行回数}}{\text{関数 g の被呼び出し回数 (6 行目)}} \\ &= \frac{3 \times 6}{6} \\ &= 3 \text{ 回} \end{aligned}$$

5.2.2 関数ポインタによる関数呼び出しの扱い

関数ポインタによる関数の呼び出しでは，プログラム実

```
1  main() {
2    ...
3    switch (opt) {
4      case 'f':
5        func = f;
6        break;
7      case 'g':
8        func = g;
9        break;
10   }
11
12   (*func)();
13
14
15
16
17
18   return 0;
19 }
```

図 7 関数ポインタによる関数呼び出し

Fig. 7 Function call using function pointer

```
1  main() {
2    ...
3    switch (opt) {
4      case 'f':
5        func = f;
6        break;
7      case 'g':
8        func = g;
9        break;
10   }
11
12   if (func == f) {
13     f();
14   } else if (func == g) {
15     g();
16   }
17
18   return 0;
19 }
```

図 8 呼ばれる可能性のある関数を列挙した近似プログラム

Fig. 8 Approximated program by enumerating functions potentially invoked

表 4 関数ポインタを含む時点軸

Table 4 Sequentialized time axis with function pointer

関数名	行番号
main	1
main	...
main	11
f	1
f	2
f	...
g	1
g	2
g	...
main	13
main	14
main	15

行時に呼び出される関数が決定される．そのため本ツールでは条件分岐によって呼び出される可能性のある関数を全て列挙することで時点軸を導出する．呼び出される可能性のある関数を平均的に扱い近似することは，網羅的な実行履歴を用いた近似としては十分であると考えられる．

例えば**図 7** のような関数ポインタ func に代入される関数が f と g である場合について考える．この場合は 12 行目で関数ポインタ func による呼び出しがある．時点軸では**図 8** のように，12 行目で関数 f と g が条件分岐によって呼び出されるものとする．その結果**表 4** のようになる．

関数の列挙は，ソースを静的解析し関数ポインタへの代入箇所を調査することで対応する．したがって，関数ポインタが指す関数が静的に決定する場合を対象とする．

5.3 gzip-1.2.4 に対する適用実験

gzip-1.2.4 に対する提案手法の適用結果を**図 9** に示す．今回 gzip に与えた入力ファイルは linux-3.6.3 のソース (linux-3.6.3.tar: 466MB) である．網羅性を確保するためにある程度の規模があるファイルを選択した．

グラフ中で，縦軸は各構造体型の各時点における利用比率である．横軸は時点軸であり，目盛上には各時点の関数名を記載している．全ての時点を列挙すると極端に長くなるため，同じ関数において時点が連続する部分については関数ごとにまとめている．各時点において構造体型のアクセス回数の変化が明確になるように累積グラフとして表している．また，グラフの上部に色が付けられている部分は，18 通りの実行から自動的に時点を分類したグラフである．gzip で行う 18 通りの実行は圧縮か展開が事前に分かっている．そのため，圧縮のみ実行される時点については青色，展開のみ実行される時点ついては赤色，圧縮と展開の両方で実行される時点については灰色で示している．時点軸は関数呼び出しを静的に列挙したものである．gzip では時点軸を列挙したときに圧縮，展開の順に関数呼び出し箇所が出現するため，グラフ上の色分けは圧縮，展開の順となる．

gzip 中で定義される 6 種類の構造体型は**表 5** である．また，各構造体型に対して「入力–処理–出力」という実行フェーズに分類した結果を人手による分類と提案手法による分類を併せて示す．

5.3.1 考察

各構造体型について考察を行う．

stat 構造体型 stat はファイル処理に関する構造体であり，グラフの序盤と終盤でアクセスされることが分かる．また，時点の分類グラフから圧縮と展開の両方で利用されていることから，入力と出力で利用される構造体型であると判断できる．

option 構造体型 option はオプション処理に関する構造体である．グラフの序盤で利用されており，時点の分類グラフからも圧縮と展開の両方で利用されているため，実行フェーズの入力で利用される構造体であると判断できる．

tree_desc と ct_data これらの構造体型は圧縮処理で利用される．グラフ中盤で利用されており，時点の分類グラフの圧縮部分で利用されているため，実行フェーズの処理に該当する構造体であると判断できる．

config 構造体型 config は圧縮の際のパラメータを保持し，圧縮処理のはじめにプログラム中で 1 度だけ参照される．時点の分類グラフから圧縮部分の序盤で利用

表 5 gzip の構造体型

Table 5 Struct types of gzip

構造体型名	機能	実行フェーズの分類	
		人手	提案手法
stat	ファイル処理	入力，出力	入力，出力
option	オプション処理	入力	入力
config	圧縮	処理	処理
tree_desc	圧縮	処理	処理
ct_data	圧縮	処理	処理
huft	展開	処理	処理

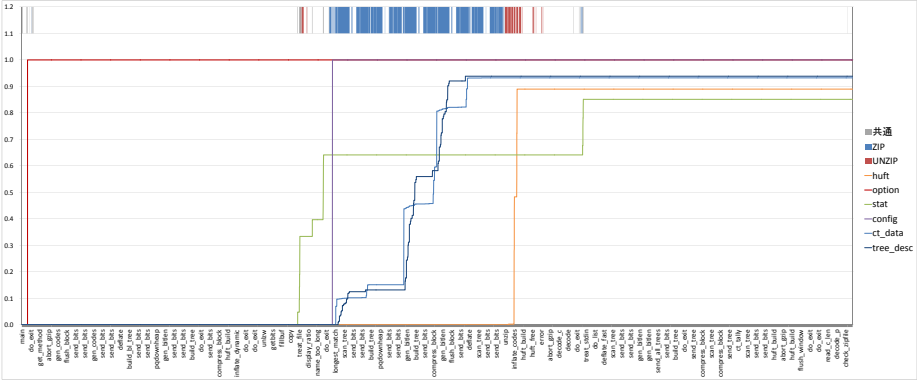

図 9 gzip-1.2.4 のアクセス頻度グラフ
Fig. 9 Access frequency chart of gzip-1.2.4

されていることが分かる．そのため，実行フェーズの処理であると判断できる．

huft 構造体型 huft は展開処理で使用される構造体である．時点の分類グラフ中で中盤の展開処理に該当する部分で利用されているため，実行フェーズの処理に該当すると判断できる．展開処理の部分が極端に短くなっているのは，時点軸を導出する際に繰り返し構文の展開を 0 段で打ち切っているためである．gzip の展開処理は関数 inflate_codes を繰り返しによって何度も呼び出すことで実現している．

5.4 tar-1.26 に対する適用実験

tar-1.26 に対する提案手法の適用については分類の結果のみを**表 6** に示す．tar に与えた入力は gzip と同様に linux-3.6.3 のソースである．

tar で定義されている関数の一部には，ライブラリから呼び出される関数が含まれている．時点軸の導出には，gzip と同様にライブラリ部分は含めずに tar 中で定義されている範囲のソースのみを対象とした．時点軸はすべての関数呼び出し部分をインライン化するが，その呼び出し部分を特定しない場合には展開をしない．インライン化されない関数から呼び出されている関数は時点軸には現れず，関数の呼び出し文脈を考慮することができない．そのため，時点軸に現れない関数からの関数呼び出しについては構造体型のアクセス回数を正確に計測することができない．

tar-1.26 中で定義される構造体型は 39 個である．そのうち，前述した影響を受けない構造体型と tar 中でアーカイブ処理に関係する中心的な役割を果たす tar_stat_info を結果として掲載する．

表 6 にある tar_stat_info 以外の構造体型は，動的解析のために選択したオプションにより，変数の初期化時に利用されているが，特徴付けできていると考えられる．

5.5 適用実験のまとめ

gzip については人手による分類と提案手法による分類が一致したので，「入力–処理–出力」という実行フェーズに各構造体を特徴付けることができたと判断できる．

tar では，tar で定義される関数のみを時点軸の対象としたため十分な結果を得ることができていない．これについてはライブラリから tar 中で定義される関数を呼び出している部分を時点軸に含めることで対応できると考えられる．

本実験ではプログラムを実行するためのオプションを手動で選択した．実行の網羅性を確保するためには，設定可能なオプションを全て選択する必要がある．また，実験に用いたプログラムは「入力–処理–出力」という形式に該当するプログラムである．プログラムの種類にはそれ以外のものいくつか存在すると考えられるため，そのようなプログラムに対して実験を行うことで提案手法がどのような種類のプログラムに適用可能であるかを調査する必要がある．

文脈の違いによる行の実行回数の近似について正確に実行回数を計測した場合と精度を比較する必要がある．しかし，本論文は概要把握の支援という観点から大まかにアクセス回数の変化を捉えたいので，近似で十分であると考えられる．

表 6 tar の構造体型 (一部)
Table 6 Struct types of tar (part)

構造体型名	機能	実行フェーズの分類	
		人手	提案手法
tar_stat_info	アーカイブの生成，追加，削除，抽出，リスト表示	入力，処理，出力	—
checkpoint_action	チェックポイントで実行するアクションを指定	入力，処理	入力
deferred_unlink	削除するファイルのリストを保持	入力，処理	処理
keyword_list	キーワードを保持	入力，処理	処理
string_list	アーカイブ抽出時に利用	処理	処理
posix_header	posix 形式ヘッダの情報を保持	処理	処理

6. 関連研究

機能レベルでの理解支援として Feature Location がある．Feature Location は機能と実装を対応付ける手法である．プログラムの改良やデバッグの際に用いられ，静的手法 [5]，動的手法 [1] などが存在する．Andrew らは文献 [1] で Dynamic Feature Traces を提案している．TDD (Test Driven Developement) で開発されたシステムを対象に，そのシステムで用いられるテストケースを「機能を表現しているテストケース」と「機能を表現していないテストケース」に分類し，各テストケースの実行結果から機能とソース上の実装箇所の対応付けを行う．文献 [1] は大規模なテストケースが用意されているシステムを対象とすることで機能を実装している箇所を特定している．本手法は入力データの選別にはそれほど注力せず，コマンドラインオプションを全て実行することで網羅性を確保する．入力データの自動生成を用いて本手法を適用することで，構造体型の分類にさらに有用な情報を提供できる可能性があると考えられる．

開発者に開発支援となる情報提供をする研究として村尾らのメトリクスの変化を用いた研究 [8] がある．村尾らはソフトウェアの開発履歴から得られるメトリクスを計測し，その変化を観察することによりどのモジュールに注力すべきかを特定する手法を提案している．メトリクス値からエントロピーや距離を計算し，メトリクスがどの程度安定しているかを表す指標をメトリクス値の恒常性としている．大規模ソフトウェアの支援をするための情報推薦という点で本研究と関連している．

静的解析と動的解析を組み合わせた研究として，Patel らはソフトウェアのコンポーネントを分類する手法を提案している [6]．実行トレースを用いてコンポーネントをクラスタリングし，そこで分類されなかったコンポーネントについては静的解析を用いて既存のグループに追加していく．文献 [6] の手法は，実行トレースの類似性から機能を分類しているのに対して，本手法は実行履歴を集約し，その変化を観察することで機能を分類する点が異なる．

7. おわりに

本論文では，C 言語を対象としてソフトウェアの概要把握を支援するために構造体型を特徴付ける手法を提案した．複数の実行履歴を 1 つにまとめるための方法として，時点軸の導出を行った．関数の呼び出し文脈に基づき構造体型のアクセス回数を計測し，構造体型を実行フェーズに特徴付けた．2 つのソフトウェアを対象に適用し，gzip-1.2.4 について大まかな特徴付けを行うことができた．tar-1.26 はソースの解析範囲を広げることで特徴付けを行うことができると考えられる．

特徴付けの結果は，ある構造体型を利用している関数群を機能としたときにその機能が実行フェーズのどの段階であるかを判断するための指標になる．大規模ソフトウェアは複数の機能から構成されており，正確に機能を分類するにはコストが掛かる．提案手法により特徴付けられた構造体型は，複数の網羅的な実行から特徴付けを行うため，実行フェーズという大まかなグループに分類することができる．したがって，概要把握を行う開発者に対して有用であると考えられる．

今後の課題として 4 つ挙げる．1 つ目に，今回は提案手法を 2 つのソフトウェアに対して適用したが，その他のソフトウェアにも適用することで本手法の有効性について検証する必要がある．特に，「入力–処理–出力」という実行フェーズ以外の形式を持つようなプログラムに対して適用し，本手法の有効な範囲を確認する必要がある．2 つ目に本手法はアクセス回数ではなくアクセスの比率によって実験の考察を行った．比率に正規化するとアクセス回数が極端な場合でもその差が現れないため，回数により比較する必要がある．3 つ目に時点軸で対応できない部分を考える必要がある．時点軸は静的解析によって導出するため，関数ポインタのように呼び出される関数が動的に決定する場合は対応できない．その部分について考慮する必要がある．最後に，本手法の他言語への応用である．本手法は手続き型である C 言語を対象に構造体型の特徴付けを行ったが，オブジェクト指向のクラスのような近い概念を持つ言語に対して本手法が拡張可能であるかを検討する．

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

参考文献

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

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

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

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

[5] 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).

[6] Patel, C., Hamou-Lhadj, A. and Rilling, J.: Software Clustering Using Dynamic Analysis and Static Dependencies, *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, Washington, DC, USA, IEEE Computer Society, pp. 27–36 (2009).

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

[8] 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，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).

[9] 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，情報処理学会論文誌，Vol. 39, No. 6, pp. 1990–1998 (1998).