実行トレース中の繰り返しの発見による プログラムアニメーションの短縮

樋口 直志* 寺田 実†

2002年1月10日

概 要

大規模ソフトウェアを対象としたプログラムアニメーションにおいては上映の長時間化が問題となる。そこでアニメーションの基となる実行トレースから繰り返し部分を抽出し、アニメーションに反映することによって上映時間の短縮を試みた。繰り返し部分の抽出においてはスライド辞書法を参考にした。その結果プログラムアニメーションの長さを 1/2 ～ 1/5 に短縮することに成功した。

1 背景

ソフトウェアビジュアライゼーションの一種としてプログラムビジュアライゼーション (PV) がある [1] 。これはプログラムのコードやデータ構造を視覚化するものである。

PV の研究は数多く行われているが、視覚化対象となるソフトウェアが十分に小さい場合のみ実用に耐えるというものがほとんどである。もちろん実際に社会で使用されているソフトウェアは大規模なものが多く、既存の PV 技術では対応できない。

そこで我々は大規模ソフトウェアを対象とした PV システムの製作と評価を行った [2] 。このシステムのねらいはプログラムの全体的な構造の概略を提示することであり、ユーザは視覚化結果を利用することでプログラムソースのどの部分を読めばどのような情報が得られるかについての “あたり” を得ることができる。

*Naoshi Higuchi
東京大学大学院修士課程 higuchi@sanpo.t.u-tokyo.ac.jp
†Minoru Terada
東京大学 terada@sanpo.t.u-tokyo.ac.jp

ところで [2] のシステムは出力として静止画形式と動画形式の両方を備えており、動画形式においてはプログラムアニメーション (PA) と呼ばれるソフトウェアのコードが実行されていく様子を時系列を追って提示する手法を採用している。しかしながらこの手法ではプログラムの実行ステップ数に比例して上映時間も長くなる。[2] では空間的な情報省略 (画面に一時に表示する情報を減らす) には注力したが、時間的な情報省略の必要性には着目していなかった。結果として大規模ソフトウェアを対象とした PA ではユーザがそれを見終わることができないという問題が残った。

しかし、PA にはプログラムがいくつのフェーズから構成されているのかがわかるなど静止画には無い利点も多く、あきらめてしまうには惜しい。よって我々は PA の上映時間を短くする研究を行うことにした。

2 本研究の目的

PA の上映時間を短縮することによってユーザがそれを最後まで見ることができるようにするのが本研究の目的である。ただし上映時間の短縮によって、プログラムの実行状況をわかりやすくユーザに提示するという、PA 本来の機能が損なわれてはならないことに注意されたい。むしろ冗長な部分を取り除くことによってユーザの理解度が向上するのが望ましい。

3 本論文の概略

我々のPAシステムの入力データはプログラムの実行トレースである。この実行トレースから冗長な部分を発見し、それを視覚化ツールに反映することにより上映時間を短縮する。冗長部分の発見に関しては汎用のデータ圧縮アルゴリズムを参考にしている。

本論文の以降の構成は次のようになっている。第4章では本研究のベースとなった我々のPAシステムの概略を紹介する。次に第5章でPAの上映時間の短縮の為にシステムに加えた改良について説明する。その後に第6章において改良したシステムの実験結果について述べ、第7章で関連研究との比較を行う。最後の第8章、第9章は結論と今後の課題についてである。

4 ベースとなったシステムの説明

本研究はPAの短縮を目的としている。この章ではPA短縮のための改良をする前のベースとなったシステムについて説明する[1]。

4.1 ベースシステム概要

本研究のベースとなったソフトウェア視覚化システムは、

- 1万行以上のJavaプログラムが対象
- トレーサ・フィルタ群・表示ツール群の三層構造
- フィルタの組合せで大規模プログラムに対応

といった特徴をもっている（図1）。視覚化においてはクラスをノード、クラス間のメソッド呼び出しをアークに対応させたグラフ構造を元にした画像をユーザに提示する(図11)。

以降三層構造のそれぞれの層について詳しく説明する。

4.2 トレーサ

本システムのトレーサはJavaのデバッガ作成用アーキテクチャであるJPDA [3] のうちJVMDI [4]

[1]本章の内容は我々の先行研究 [2] と重なる部分が多い。

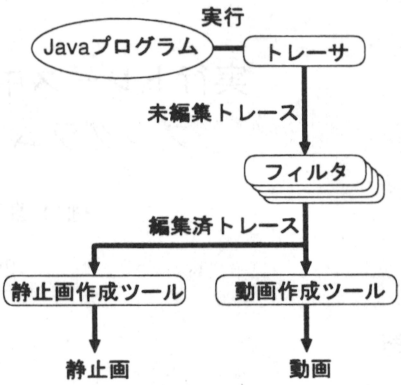

図 1: ベースシステムの概要

というインタフェースを使用して作成されている。このインタフェースに準拠してトレーサはC言語で記述され、シェアードライブラリとしてコンパイルされている。そしていくつかの引数を付けてJava実行系を起動することにより、実行系にトレーサがダイナミックリンクされて動作するようになる。

トレーサはメソッドの呼び出し・復帰、例外の生成・補足の各時点において、

Action CALL / RETURN / THROW / CATCH

FC コールスタック中のフレーム数

Thread ID カレントスレッドのID

Method Modifier static / virtual / native

Receiver ID レシーバオブジェクトのID

Receiver Class レシーバオブジェクトのクラス

Method Class メソッドの定義されているクラス

Method name メソッドの名前

Method Signature メソッドの引数・返り値の型

についての情報を出力する。

トレーサは簡単なフィルタ機能を有しており、**Receiver Class** を基準にして出力を抑制することができる。現状ではクラス名が `java/`、`javax/`、`sun/` 等ではじまっている場合（Javaの標準クラスライブ

ラリ中のクラス）には出力を抑制している。ここで、**Method Class** ではなく**Receiver Class** を基準にしていることに注意されたい。これによってユーザが標準ライブラリ中のクラスを継承して作成したクラスに関する情報は、抑制されることなく出力される。

4.3 フィルタ群

フィルタ群では、

- 例外情報の補完
- 不要な情報の切り捨て

という 2 種類の作業を行う。

例外情報の補完とは、大域脱出で通過した途中のメソッドをリストアップする作業である。例えば:

表 1: 例外情報の補完

Action	FC	Method
THROW	5	thrower()
CATCH	2	catcher()
	⇓	
THROW	5	thrower()
THROUGH	4	through1()
THROUGH	3	through2()
CATCH	2	catcher()

フィルタ内部でコールフレームの Push / Pop をシミュレートすることにより、このように大域脱出の情報を補完することができる[2]。

フィルタ群の機能のうち、不要な情報の切捨ては大規模プログラムの視覚化において重要な役割を果たす。何をもって不要な情報とするかについては様々な基準が考えられるが、ここでは様々な Java アプリケーションに対応できる package を基準とした切り捨てを挙げておく。

これは着目する package を引数にしてフィルタを起動するとレシーバオブジェクトのクラスが所属するパッケージの比較を行い、非着目パッケージ内でのメソッドの呼び出しについての情報を切り捨てるというものである (図 2)。

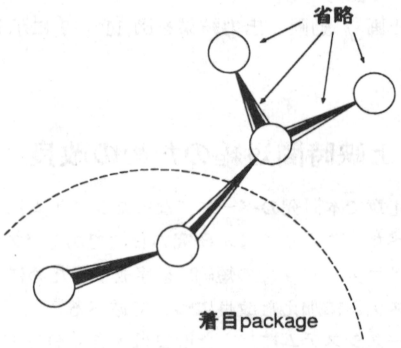

図 2: package を基準にした切捨て

なお我々の経験上、初見のソフトウェアを視覚化する際のデフォルトの操作として、main メソッドを持つクラスが所属する package を引数にしてこのフィルタをかけるというのが有効である。

4.4 表示ツール群

表示ツール群はトレースから静止画を生成するツールと動画を生成するツールからなる。

両ツールに共通する手法は以下のとおりである。

まず実行トレースを元にクラスをノード、相異なるクラス間のメソッド呼び出しをアークとする有向グラフを生成する。

次にノードにクーロン電荷、アークにバネ係数を定義することにより有向グラフを物理モデル（バネモデル）に変換する。そして最後に緩和計算を行うことによってノードの配置を決定する。

両ツールの差異は有向グラフの生成と緩和計算、そして結果出力のタイミングの違いである。

静止画作成ツールでは最初にトレースを全て走査することにより、トレース中のクラス全てをノードとして持った有向グラフを作成してしまう。その後に緩和計算を行い、結果を静止画として出力する。

動画作成ツールでは 1) トレースを 1 ステップ読む、2) 必要があれば有向グラフにノードを追加する、3) 読み込んだ 1 ステップのトレースに関連するノー

[2]簡単の為にトレースはかなり省略してある

ドとアークを光らせる[3]、4) 緩和計算を 1 ステップ行う、という一連の動作を繰り返し、その様子を動画として出力する。

静止画 / 動画の出力結果を図 11、12 に示しておく。

5 上映時間短縮のための改良

第 4 章で本研究のベースになったシステムについて述べた。この章では本研究の目的である “プログラムアニメーションの短縮” を達成するためにベースシステムに加えた改良について述べる。

ベースシステムに加えた改良は大きく分けて二ヶ所である。一つはフィルタ群に “繰り返しを発見して印をつけるフィルタ” を新たに追加したことである。

もう一つは繰り返し情報を反映できるように既存の動画生成ツールに手を加えたことである。

なおベースシステムのうち、トレーサについてはなんの改良も加えていない。また、静止画作成ツールについては本研究の対象外なのでこれも改良を加えてはいない。

5.1 繰り返しの定義

本研究では、トレース中の部分系列 A と部分系列 B について A が実行順序的に先行しており、かつ B が A と 同様である と認められる場合に「B は A の繰り返しである」と定義する。

“同様” が何を意味するかは、PA において対象プログラムをどのようなモデルで扱うかに依存する。本システムでの “同様” とは、トレースの部分系列が各ステップにおいて

- Action が等しい
- Thread ID が等しい
- Receiver Class が等しい
- Method Name が等しい

という条件を満たすことである。ここで Receiver ID を考慮していないのは、本システムでは同じクラスに属するオブジェクト群は一つのノードで代表してモデル化されるからである。もちろん異なるモデルを採用する PA においては “同様” の定義も変わりうる。

[3] 例えばトレースがメソッド呼び出しに関するものであれば、呼び出し元、呼び出し先、その間のアークを光らせる。

5.2 繰り返しの種類

この章では繰り返しを大きく二種類に分けて考える視点を導入する。1 つは繰り返しの単位となる系列（単位系列）が連続して現れるもので、これを本研究では Repeat と呼ぶ。もう一つは単位系列が時間的に離れて現れるもので、これは Re-run と呼ぶことにする。

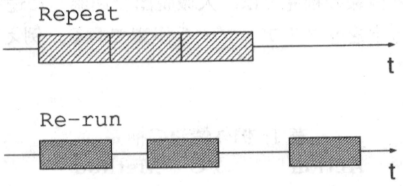

図 3: Repeat と Re-run

なお単位系列は “単位” とはいえ 1 ステップだけではなく 2 ステップ以上の実行トレースを含みうることに注意されたい。

さて Repeat と Re-run 両者を分けて考えるのは Repeat の方が Re-run に比べてアニメーションとしてのユーザへの提示が容易なためである。すぐ後で詳しく説明するが、我々の PA システムでは Repeat・Re-run ともに単位系列を 1 フレーム（PA の 1 コマ）でユーザに提示する。Repeat の場合はユーザに単位系列を表現したフレームが連続して提示されることになるので、ユーザは特に注意しなくても個々のフレームが同様の単位系列に対応していることを把握することができると予想される。問題は Re-run の場合であり、単位系列を表現するフレームが時間的に離れて現れた場合にユーザが個々のフレームを結びつけて把握できるためには、“見せ方” において何らかの工夫が必要であると考えられる。

5.3 繰り返しの発見手法

現在のところ繰り返しを発見するために “スライド辞書法” を参考にした手法をトレースに適用している。これは最近のトレースの履歴を固定長バッファに保存しておき、そこから最長一致で繰り返し系列を検索するものである。

ただし単純にスライド辞書法を用いるだけでは 1)Re-run において先行して現れた系列と後に現れる系列の間が離れすぎると繰り返しが発見できない、2) 繰り返しに ID をつけるなどして対応関係を保存することができない、といった問題がある。そこで我々は既に発見した繰り返し系列を Trie 構造に保存し、a) 繰り返し発見において固定長バッファだけを探索するのではなく Trie 中も探索する、b)Trie に保存された系列に ID として通し番号をつける、という改良を加えるによって上記の問題点を解決した。

なおスライド辞書法を用いて繰り返しを発見するということは実行トレースの構造について、1) 木構造などを仮定しないコード列、2) 同様な部分系列が何度も現れる、という最低限の仮定しか置いていないことになる。本システムに関して言えばメソッド単位の実行トレースは木構造をとるという条件を用いていないことになる。この実行トレースに最低限の仮定しか置かないことについては第 6.2 章で再度触れる。

最後に注意してほしいのは、繰り返し発見フィルタの処理が 1pass であるために繰り返しの第一回目には ID がつかない。より正確に言うと、第一回目の繰り返しは繰り返しとして認識されず、第二回目からが繰り返しとして認識されるということである。またある部分系列が 3 回現れるとして 1 回目には ID がつかず、2 回目には ID がつく。そして 3 回目には 2 回目と同じ ID がつく。2 回目と 3 回目で異なる ID がつくわけではないことにも注意してほしい。

5.4 アニメーションにおける繰り返しの表現手法

この章では Repeat および Re-run に関する情報を PA に反映する手法について述べる。また個々の手法の効果を示す量として (短縮率) = (工夫した場合の上映時間)/(何も工夫しない場合の上映時間) を導入する。短縮率の値が小さいほどその手法の効果が大きいことを示している。

Repeat の表現手法 Repeat に関する情報を PA の短縮に結びつけるために用いられる手法は二つの工夫からなっている。一つの工夫は Repeat の単位系列は PA 上では 1 フレームでユーザに提示されるというものである。すなわち単位系列中の処理に関連するノードとアーク全てが同時に光ることになる (図 4)。この工夫による短縮率は 1/(単位系列の長さ) であり、単位系列が長いほど効果が大きくなる。

もう一つの工夫は単位系列に対応するフレームを連続してユーザに提示する際に提示回数 (Repeat 回数) を省略するというものである。Repeat 回数省略の式は $n = 9.9\log(0.1N + 0.8) + 2.1$ (省略前 N、省略後 n) となっており、基本的には N の対数をとるのであるが $N = 1, 2, 3$ については $n = 1, 2, 3$ となるように調整することにより PA のユーザを混乱させないようにしてある。代表的な N に対する n の値は次の表のとおりである。

表 2: Repeat 回数の省略

N	1	2	3	10	100	1000
n	1	2	3	7	25	47

この表からわかるように、二つ目の工夫の効果は Repeat 回数によっている。Repeat 回数が 1、2、3 の時の短縮率は 1(全く短縮されない) であるが、回数が 100 ならば 1/4 、1000 ならば約 1/20 に達する。

以上において二つの工夫について説明してきたわけだが、注意してほしいのは全実行トレース中の 50% が Repeat 系列であったとして、そのことだけでは PA 全体に与える短縮効果が測れない点である。全ての Repeat が単位系列の長さ 100、Repeat 回数 1000 の系列からなっていたとすると、Repeat 部分全体の短縮率は $1/(100 \times 20)$ である。しかし全て長さが 2 で Repeat 回数 3 の系列からなっていたとすると、短縮率は $1/(2 \times 1)$ にしかならない。すなわち Repeat が PA 短縮に与える効果を見積もるには、実際の実行トレース中での Repeat の分布 (単位系列の長さ・Repeat 回数ごとの出現頻度) を調べる必要がある。

Re-run の表現手法　現在のところ、Re-run 系列に関しては単位系列に関連するノード・アーク群を全て光らせることにより PA の短縮を目指している (図 4)。よって短縮率は Re-run の単位系列の長さだけで決まり、1/(Re-run の単位系列の長さ) となる。

このことから短縮率の観点からすると Repeat に比べて Re-run は不利であると見ることもできるが、Repeat と同様に全 PA を通しての Re-run の効果は実際の実行トレース中での分布にも依存するので, 一概には言えない。単位系列の長い Re-run が高い頻度で実行トレース中に出現することが望ましい。

また第 5.2 章でユーザが Re-run の単位系列間の対応関係を把握しやすいように見せ方を工夫する必要があると述べたが、現在のところ何の対策もとっていない。

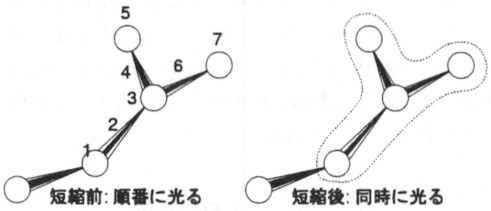

図 4: 繰り返しの単位系列の表現手法

6　実験結果

この章では本システムの評価を行うための実験結果を示す。本来ならば本システムの評価を行うために必要な実験として少なくとも下記の三種類が挙げられる。

実験 1 実際のトレース中での繰り返しの現れ方とその結果としての PA の短縮率を調べる

実験 2 本システムが発見する繰り返しはプログラム中の意味要素に対応しているか調べる

実験 3 本システムがユーザのプログラム理解を助けることができるか試験する

現在のところ実験 1 および 2 を開始したという状態であり、実験 3 はまだ行われていない。

なお実験においては実用アプリケーションソフトウェアである、[5] [6] [7] [8] [9] のトレースを取得および分析した。そのうちの [6] [7] [8] について表 3 に概略を示す。

表 3: 実験に用いたアプリケーションソフトウェア

	説明 クラス数 / 行数
DRS [6]	ソースコード読解支援ソフト 32 / 2912
Jisgaw [7]	HTTP サーバ 928 / 121650
Kawa [8]	Scheme インタプリタ 516 / 72651

6.1　実験結果 1—繰り返しの現れ方

6.1.1　繰り返しが実行トレース中に占める割合

実行トレース中に占める繰り返しの割合が十分に高くなければ、本研究で提案する PA 短縮のための手法は有効とはいえない。そこでいくつかのアプリケーションに対してトレース中に占める繰り返しの割合を実際に調べた。下記は調査結果を表にまとめたものである。ただし (cut) という表記は第 4.3 章で述べたように package を基準にトレースを編集してサイズを減らしたことを示す。

この結果から言えることは、

- 繰り返しが実行トレースに占める割合は、アプリケーションのドメインに依らずに 50%を超えることが期待できる。
- 繰り返しに着目した PA の上映時間の短縮では「桁違い」な短縮は期待できない。
- cut による PA の短縮効果はきわめて大きく、文字通り桁違いな短縮が可能である。

といったところである。

我々のシステムでユーザが PA を見終わることができる上限は、経験上 20000 ステップ (およそ 1 時間に対応) となっている。Jigsaw、Kawa については

表 4: 繰り返しが実行トレース中に占める割合

	step	Repeat + Re-run	Repeat
CSS Validator (cut)	443	4 %	0 %
Jigsaw (cut)	12026	41 %	23 %
CSS Validator	12375	73 %	43 %
Kawa (cut)	14620	79 %	54 %
DRS	73540	72 %	53 %
Kawa	92170	56 %	20 %
Jigsaw	583100	90 %	50 %
Raja	36744497	77 %	1 %

本研究による PA の短縮手法を使うまでもなく、cut によって 20000 ステップ以下に収めることができる。しかし DRS については本研究の手法を用いて初めて 20000 ステップに収めることが可能となる。

なお Raja はコンピュータグラフィックスのためのレイトレーサであり、実行ステップ数が多くなるのは期待通りである。しかし繰り返しの割合、特に Repeat の割合が低くなっているのは予想外であった。これが実験の不手際に依る誤ったデータなのか、本当に繰り返しの割合が低いのかについては現在調査中である。誤ったデータが得られる可能性の一つとして考えられるのは、スライド辞書法における固定長バッファの長さが不足していることであるが、バッファ長を 8 倍にして再調査した際にも全く同じ結果が出たことを付記しておく。

6.1.2 Repeat の分布

PA の工夫のしやすさという観点からいうと、Repeat の方が Re-run より扱いやすいことを第 5.2 章で述べた。また、Repeat の割合が 50%を超える場合もあることを第 6.1.1 章で示した。しかし以上をもって「Repeat の表示を工夫することにより PA を容易に短縮できる」ということはできない。第 5.4 章で述べたように、Repeat が PA の短縮に与える効果を評価するには実際の実行トレース中での Repeat の分布を調べる必要がある。

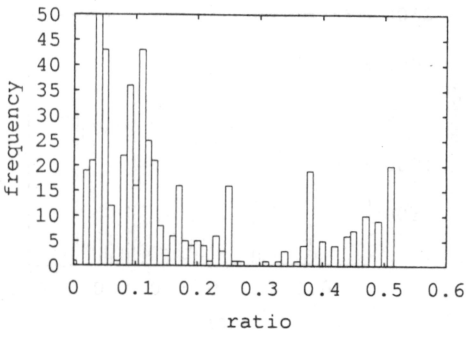

図 5: DRS における短縮率のヒストグラム

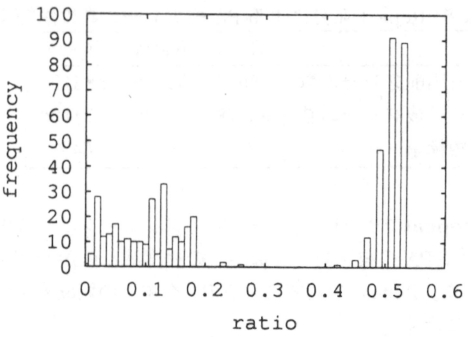

図 6: Kawa における短縮率のヒストグラム

よってこの章ではプログラム実行トレース中から Repeat だけを発見し、Re-run は発見しないようなフィルタを用いて実験を行った。

Repeat の分布 (1) まず実行トレース中の個々の Repeat 系列について短縮率を計算し、次に短縮率について 0.01 刻みの区間に分けて各区間に入る Repeat 系列の出現頻度を調べることによってヒストグラムを作成した。図 5、6、7 がそのヒストグラムである。

図 5 によれば DRS では短縮率が 0.1 を下回るような Repeat 系列が多数存在し、効率よく PA を短縮できることが期待される。しかし図 6 および図 7 によれば Kawa や Jigsaw では短縮率が 0.5 付近の Repeat 系列が多く、各 Repeat 系列についてアニメーションのフレーム数を 50%程度にしか減らせないことがわかる。

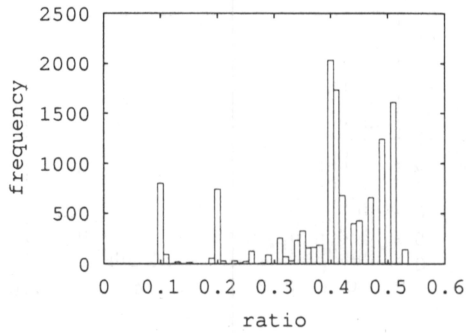

図 7: Jigsaw における短縮率のヒストグラム

表 5: Repeat 発見による PA 総フレーム数の短縮率

	DRS	Kawa	Jigsaw
短縮前フレーム数	73540	92170	583100
短縮後フレーム数	36478	75125	374198
短縮率	49%	81%	64%

Repeat の分布 (2) 各々のアプリケーションにおいて PA の総フレーム数がどの程度減るのかを調査した。Repeat の短縮による PA 全体での短縮率を表 5 に示す。

DRS が最も良い短縮率を示しており、Repeat のみの発見によって 50%まで PA を短縮できる場合があることがわかる。

ここで前出の表 4 は実行トレースの持つ PA 短縮の可能性を示すもので、表 5 は実際に PA を短縮した結果であることに注意されたい。

6.1.3 Repeat と Re-run を併用した場合の短縮率

Repeat と Re-run を併用した場合の PA 全体での短縮率を表 6 に示す。

Repeat の発見を利用する場合と同程度か、より良い短縮率を示している。このことは PA の短縮の為に Repeat だけを利用するのではなく、PA 上での表現の難しさを差し引いても Re-run の発見を積極的に利用すべきであることを意味している。

ここでも表 4 は PA 短縮の可能性を示し、表 6 は実際に短縮した結果を示していることに注意されたい。

表 6: Repeat と Re-run を併用した場合の短縮率

	DRS	Kawa	Jigsaw
短縮前フレーム数	73540	92170	583100
短縮後フレーム数	18214	57476	86035
短縮率	24%	62%	14%

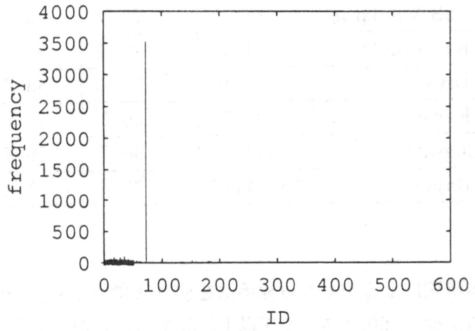

図 8: DRS における特定の繰り返しの出現頻度

6.1.4 特定の Repeat、Re-run の登場頻度

第 5.3 章において繰り返し系列に ID をつけることが可能であることを述べた。ここでは ID 毎に出現頻度を調べた結果について述べる。調査結果は図 8、9、10 に示してある。

この結果からすぐに言えるのは、出現頻度の高い繰り返しはごく限られているということである。いずれのアプリケーションプログラムにおいても 1 ないしは 2 種類の繰り返しだけが極めて高い頻度で出現し、その他の繰り返しの出現頻度に大きく差をつけている。ただし、この結果をどのように PA の短縮もしくは PA の表現力向上に利用できるかは今のところ不明である。

6.2 実験結果 2—繰り返しとルーチンの対応関係

本研究の目的は PA の短縮であるが、これは単に時間的に短縮できれば良いというものではない。PA 本来の目的はソフトウェアを理解することであり、PA の短縮も当然この枠内でなされなければならない。ここでは実行トレース中から自動的に発見され

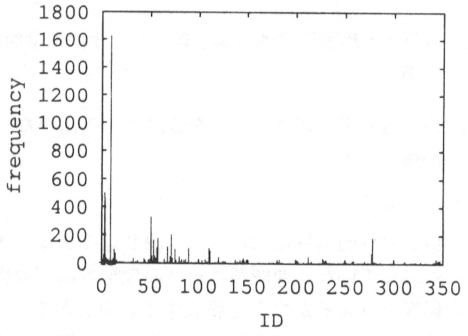

図 9: Kawa における特定の繰り返しの出現頻度

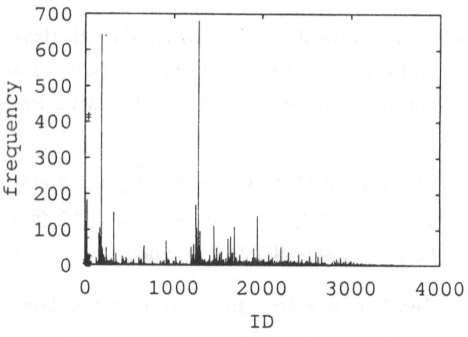

図 10: Jigsaw における特定の繰り返しの出現頻度

た繰り返しについて、それらが何らかのルーチン（意味的に一まとまりの処理の系列）と対応しているかを調べた。以下で二種類のアプリケーションについての調査結果を述べる。

まず Jigsaw(cut) では Repeat と Re-run ともにルーチンとの対応がとりやすかった。例えば MimeParser から read したものを HttpBuffer に書き込むルーチンに対応した Repeat や、ClientSocket を kill するルーチンに対応した Re-run などが存在していた。

次に Kawa (cut) においては Hash を引くことに関する繰り返しが多く見られた。また Repeat とルーチンの対応は簡単につくが、Re-run とルーチンの対応がとりにくかった。この原因として考えられるのは、Repeat では基本系列の最初と最後のネストレベルが等しくなる場合がほとんどなのに対して、Re-run では基本系列の最初と最後でネストレベルが異なる場合が多かったことである。これは動的コールツリーにおいて Repeat の基本系列はいずれかのサブツリーに対応するが、Re-run の基本系列には対応するサブツリーが無いことを示している。

我々は、Re-run の基本系列がサブツリーに対応していないことが Re-run の意味を取りにくくしている一因ではないかと考えている。またサブツリーに対応しないのは、第 5.3 章で述べたように実行トレースが木構造になるという情報を繰り返し発見時に利用していないことが原因であると考えている。

しかしサブツリーに対応しないがプログラムを理解する上で有用な繰り返しもありうる。例えば `f()` と `g()` は必ずこの順序で連続して呼び出されるとする。ここで `f()` および `g()` を呼び出す親メソッドはいつも同じとは限らない場合には、この `f(); g();` はサブツリーに対応しない。

以上をまとめると次の三点である。1) 本システムが実行トレース中から発見する繰り返しが常に何らかのルーチンに対応しているとはいえない。 2) このことは PA を利用してソフトウェア理解を図る際に障害となる可能性がある。 3) しかしながら我々はサブツリーに限定した繰り返しの発見によって問題を解決できるとは考えていない。

7 関連研究

[10] ではソフトウェアの動的分析のための三層フレームワークが提唱されており、これは我々のシステムとよく合致する。論文内ではトレースの圧縮についても述べられているが、何か特定の応用を意図したものではない。またトレースの分析手法として、run-length encoding、 context-free grammer encoding および building of finite state automata が挙げられている。特に run-length 以外の二手法はトレースからプログラムの構造を推測し、ソフトウェア理解のためのモデルを構築しようという意欲的なものとなっており、我々のシステムへの応用も考えられる。

[6] では PA を行うメインウィンドウとは別に動的コールツリーを表示したウィンドウが用意されている。このウィンドウは Windows のエクスプローラの左ペインと似たウィジェットを用いており、コールツリーのうちアニメーションを省略したいサブツリーを閉じることによって PA の時間短縮を図ることができる。これは我々の研究の目的とするところとよく似ているが、省略したい部分の発見が自動ではない点で異なる。

8 結論

本研究では PA の短縮の為に実行トレース中から繰り返し部分を発見し、それを PA に反映することについて述べた。

まず実行トレース中の繰り返しを大きく Repeat と Re-run の二種類に分ける視点を導入した。次にトレース中から Repeat および Re-run を発見するツールを実装し、実用にたるアプリケーションプログラムから取得したトレースに適用した。

その結果としてプログラム中の繰り返しの割合は 50%を超えることが期待できることがわかった。また繰り返しの現れ方を調べることにより、Repeat と Re-run の発見を併用する必要があり、併用した場合には PA を 1/2 ～ 1/5 に短縮することが可能であった。

現状では発見された繰り返しが意味的に一まとまりの処理系列に必ずしも対応していないこともわかった。これはプログラム理解という PA 本来の目的の障害となる可能性がある。

9 今後の課題

今後やらなくてはならないのは、

1. PA 短縮がプログラム理解に与える影響を実験により評価する
2. スライド辞書法以外の繰り返し発見手法を検討・評価する
3. 繰り返し発見以外に PA を改良する手法がないか検討する

などである。

一番目の評価実験においては、繰り返しがルーチンに対応していない問題がユーザの理解度にどの程度の影響を与えるかなどを調査する必要がある。

二番目のスライド辞書法以外の繰り返し発見手法の検討・評価では、例えば実行トレースが木構造をとることを利用できるような手法を用いた場合に、現行の手法と比較してどのようなメリット/デメリットが得られるのかを調査する必要がある。

三番目の繰り返し発見以外の手法の検討では、関連研究で挙げた context-free grammer encoding 等の、プログラムの構造そのものを推測するような手法が PA に与える影響について調査する必要がある。

参考文献

[1] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization*, chapter 1, pp. 3–27. The MIT Press, 1996.

[2] 樋口直志, 寺田実. 大規模プログラムの実行状況の視覚化. 情報処理学会 第 61 回全国大会（平成 12 年後期）講演論文集, 2000.

[3] Sun Microsystems, Inc. Java Platform Debugger Architecture. http://java.sun.com/.

[4] Sun Microsystems, Inc. Java Virtual Machine Debug Interface (JVMDI). http://www.sun.com/.

[5] W3C . CSS Validator. http://jigsaw.w3.org/css-validator/.

[6] 首藤達生. プログラム実行系列に沿ったソースコード読解の支援. Master's thesis, 東京大学大学院工学系研究科機械情報工学専攻, 2001.

[7] W3C. Jigsaw. http://www.w3.org/Jigsaw/.

[8] Per Bothner . Kawa. http://www.gnu.org/software/kawa/.

[9] Gregoire Sutre and Emmanuel Fleury. Raja. http://raja.sourceforge.net/.

[10] Steven P. Reiss and Manos Reneris. Encoding program executions. In *Proceedings of the 23rd international conference on Software engineering 2001*, pp. 221–230. ISBN: 0-7695-1050-7.

[11] Steven P. Reiss and Manos Renieris. Generating java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*, pp. 71–77, 2000. ISBN: 1-58113-288-3.

[12] Manos Renieris and Steven P. Reiss. Almost: Exploring program traces. In *Proceedings of the workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM international conference on Information and knowledge management 2000*, pp. 70–77, 2000. ISBN: 1-58113-254-9.

A 視覚化の例

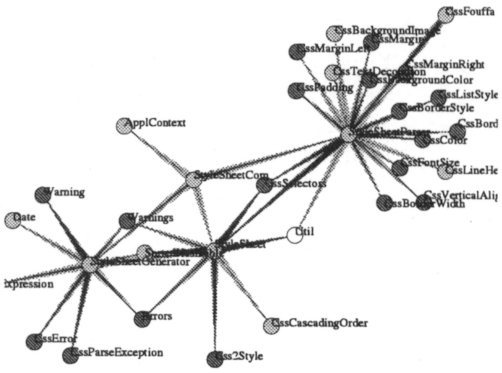

図 11: 静止画の例

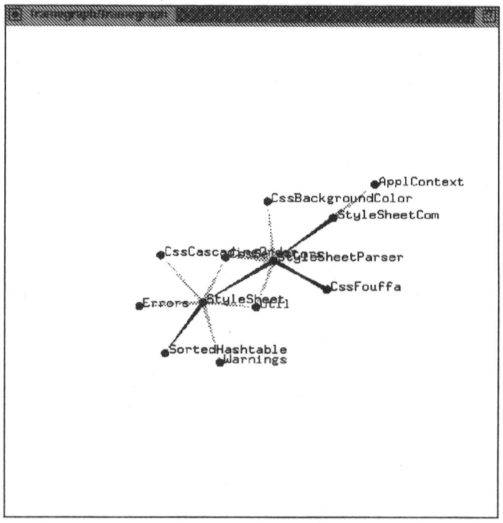

図 12: 動画の例