修士学位論文

題目

Java プログラムのコールスタック状態に着目した実行履歴の対話的な分析ツール

指導教員
井上 克郎 教授

報告者
ZHAO ZEAN

平成 29 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 28 年度 修士学位論文

Java プログラムのコールスタック状態に着目した実行履歴の対話的な分析ツール

ZHAO ZEAN

内容梗概

ソフトウェア保守においては，現時点でのプログラムの構造や動作についての理解が重要である．開発者がプログラムを理解するための手掛かりとして，プログラムを実行してその動作を調べることが有用であることが知られており，プログラムが実行中に行った様々な動作を実行履歴として記録する手法が提案されている．しかし，プログラムは実行中に様々な処理を実行していくため，実行時の状態は膨大な数に及ぶことがあり，また，イベントの系列としての実行履歴にはプログラムの構造情報が欠落してしまうことから，開発者が興味を持つ特定の処理を対話的に閲覧していくことが困難である．本研究では，Java プログラムの実行履歴をコールスタック情報に基づいて木構造で可視化するとともに，コールスタックが同一である実行状況については複数の節点を横断して移動できるような検索機能を提供する可視化ツールを提案する．データベース上に木構造を表現し，対話的な検索を可能とすることで，開発者が実行履歴全体の概要から順に詳細へと進み，必要な実行時情報だけを閲覧することを可能とした．対話的な閲覧が可能であるかどうかを評価するために，複数の大規模な実行履歴に対して，実行履歴中の様々な時点が選択された場合の木構造の検索および可視化に必要な時間を計測する評価実験を行った．

主な用語

実行履歴
コールスタック状態
ソフトウェア可視化

目 次

1　はじめに

ソフトウェアを開発・保守するためには，既存プログラムへの理解が開発者にとって重要である [5]．派生開発における再利用元となるプログラムのように開発者にとって未知のプログラムに限らず，開発者自身が過去に開発したプログラムであってもその構造や振舞いを完全に記憶しておくことは困難であることから，プログラムを動作させ，理解することが重要となる．

プログラムを理解するにあたって重要となるのが，プログラムの実行順序や実行時データの変化を知ることである．しかし，オブジェクト指向プログラムの動作は，動的束縛や例外処理などの実行時に決定される要素が多いことから，理解が困難であるといわれている [12, 2]．そのため，プログラムを実行してその動作を調べることが有効であると考えられている．

プログラムの実行時情報を収集する方法には様々なものがある．たとえばプログラムの状態を知るための古くからある手段の 1 つがデバッガのブレイクポイントである．これは，ソースコード上の文またはバイナリコードのアドレスを指定しておくと，実行が指定された位置に到達した時点で一時停止され，そのときのコンピュータの状態を調べることが可能になるという仕組みである．しかし，データを収集する位置ごとにブレイクポイントをセットする必要があり，実行の再開も手動で指示する必要があることから効率が悪く，また，途中で見落としが生じたと分かっても，過去の状態に戻れないという制限もある．

このような弱点を補うために，プログラムの実行を完全に記録し，実行状況を再現することで分析を行う Omniscient Debugging [6] が提案されている．デバッガと同様のインタフェースを用いて，あらかじめ記録された実行履歴をあたかも実行しているかのように操作することが可能であり，実行を進めるだけでなく，見落とした場合の後戻りも行うことができる．このアプローチを用いると，すべての実行時情報にアクセスすることが可能であるが，画面に表示される情報は常にある特定の時点でのプログラムの状態であるため，プログラム実行の全体像がつかみにくい．特に近年のコンピュータは演算が非常に高速であり，短時間の実行であってもプログラム内部での状態遷移は膨大であり，単純にプログラムの実行を進める，巻き戻すという操作だけでは，開発者にとって興味のある機能にたどり着くことが難しい．

本研究では，既存手法の問題点を解決するため，コールスタック状態に着目した実行履歴の対話的な分析ツールを提案する．このツールは，実行履歴全体に対し，メソッド呼び出しの階層構造を表現したコールツリーを構築し，対話的に探索させることで実行過程を直観的に理解させることを図る．また，同一のコールスタック階層を持つ複数回のメソッドの呼び出しを検索可能とすることで，似たような状況での実行の対照比較を可能とし，理解を深

めさせることを図る．複数回の実行の対照という着眼点は Matsumura ら [8] と同様であるが，コールスタックによる実行状況の考慮という観点を新たに導入した閲覧を可能とした．Java プログラムの実行履歴は，通常の実行であればメソッド呼び出しを要素とする木構造に変換できるが，実際には実行時例外の発生による処理の打ち切りなど，必要な情報が欠落する場合があり，本研究はそのような場合にも木構造の復元を自動的に行うような手法を構築した．

提案ツールは，前処理プログラムと GUI ビューアの 2 つによって構成されており，開発者が分析したいプログラムの実行履歴を記録したのち，前処理プログラムが実行履歴のデータベース構築を行い，GUI ビューアがそれを対話的に検索，閲覧する機能を提供する．

最悪ケースの実行時間を測るため DaCapo ベンチマーク [1] を対象プログラムとして実験した結果，GUI の反応時間上限を 1 分にすれば，対応できるログファイルのイベント数上限は約 163 万である．

以降，2 章では関連研究について，3〜5 章ではツールの詳細について，6 章では実験結果について述べる．

2　関連研究

プログラムの実行ログを記録し分析するための技術として，Omniscient Debugging [6] が提案されている．この手法は，デバッガと同様のインタフェースを用いて，プログラムの実行を時系列に沿って閲覧することができるが，利用者がどの時点の実行を閲覧しているのかという情報は提供していない．櫻井らの Traceglasses [13] は，実行履歴のコールスタックの深さの変化を時系列でプロットすることで，閲覧している実行時点が実行履歴のいつに相当するのかを可視化している．また，特定の変数に対する操作を検索可能とすることで，実行履歴の調査を可能としている．しかし，同一のソースコードの複数回の実行などを対比して調べるといった機能は提供していない．

Repeatedly-Executed-Method Viewer [8] は，実行ログファイルに対し同じメソッドの複数回実行を抽出して表示し比較させる手法である．この方法で実行の詳細が見られ，実行の対照ができる．しかし，プログラムの実行中に呼び出されたあらゆる状況が並べて表示されるため，動作の違いが使い方によるものか単に入力によるものかに関係なく同一視され，使い方を限定した実行だけ調べたいときは使いにくい．本研究では，コールスタック状態を区別することで，この問題を解消する．

類似した実行履歴をまとめる手法として，繰り返し同一コードが実行された場合のイベント系列を検出する手法が谷口ら [14] によって提案されている．ループの検出は連続的に実行されたイベントだけが対象に限られており，本研究では，コールスタック情報を用いることで異なる時刻での類似イベント列の発生に対応する．

プログラムの実行を特定のメソッドごとに分けるという考え方は Lienhard ら [7] も用いている．この手法では，あるメソッドの実行開始から終了までに生じたイベントを，プログラムの状態変化として要約する．プログラムの実行の概要を理解するための手法であり，本研究のように実行履歴中のイベント列の詳細を閲覧するという観点とは異なる．

メソッド呼び出し関係に基づく実行履歴の可視化手法には，Dynamic Object Process Graph [10] がある．それは，1 つのオブジェクトに対しメソッドの呼び出し系列を抽出して実行経路を可視化する手法である．この方法でメソッドの呼び出し関係が可視化されるが，データの変化などの実行時情報が省略された．また，動作が異なる同じメソッドの複数回実行が同じノードに吸収され，その間の差が比較不能となる．

3　ツールの概要

実行履歴ビューアは，対象プログラムのある実行の履歴に対し，その実行中に発生したイベントをコールスタック状態に基づきツリー構造に表示する．また，あるイベントのノードに対し，それと同じコールスタック状態の呼び出し群を検索して，その結果を部分木の集まりの形で表示できる．さらに，その検索結果のノードに対し，元の木での対応したノードをハイライトすることができる．

上記の機能を実現するためには，実行ログが必要で，かつコールツリーに関する情報が必要である．また，ログ内の履歴データを高速にアクセスする必要である．さらに対話的操作のため，GUI が必要である．

ログファイルの入手には selogger [4] を利用することにした．ただし selogger の出力はイベントの発生した系列であって木構造を提供するわけではないため，その点はツールで計算する必要がある．データの高速アクセスのためデータベースライブラリを利用することにした．GUI 表示のため Win32API を利用することにした．

何 GB のログファイルになると，コールツリー計算および実行履歴をデータベース化する時間が長くなることが予想され，データベース化と GUI 表示は別々のプログラムで行うことにした．よって本研究のツールは，selogger の出力を入力としてデータベース化する前処理プログラムと，そのデータベースに基づいた GUI ビューアプログラムで構成された．

本章の残りは各モジュールの外部仕様について述べる．

3.1　selogger について

selogger [4] は Java プログラムの実行履歴を取るためのツールである．分析したい Java プログラムのクラスファイルに対して selogger 内の Weaver と呼ばれるツールを実行し，ログファイル出力用のコードを埋め込むバイナリ変換を行う．そのバイナリ変換後のクラスファイルを実行することで，実行ログがファイルとして出力される．

本研究では，出力をデータベース用に加工しやすいように改造を加えたバージョンを使用した．プログラムを実行すると，下記のファイルが出力される．

- classes.txt：csv 形式の配列，クラスファイルの情報を持つ．
- methods.txt：csv 形式の配列，メソッドの情報を持つ．
- Location ＃＃＃＃＃.txt：csv 形式の配列，イベントがあり得る詳細情報を持つ．“＃＃＃＃＃” は通し番号である．
- LOG $ ObjectTypes01.txt：csv 形式の配列，オブジェクトの情報を持つ．

- LOG $ Types.txt：csv 形式の配列，型情報を持つ．
- LOG $ String ＃＃＃＃＃.txt：csv 形式の配列，文字列の情報を持つ．
- t-＃-＃＃＃＃＃.slg : ObjectOutputStream でのバイナリデータ，イベント情報を持つ．“＃＃＃＃＃” は通し番号である．

3.2 前処理について

前処理は，selogger が出力されたログファイルをデータベース化する過程である．前処理の意味は以下の通りである．

- ログファイル自体が配列の形であり，かつ何ギガバイトのサイズとなる可能性があるため，データベース化することよりメモリ使用の軽減，アクセス高速化およびツール開発の簡易化を図る．
- コールスタック状態やメソッドの開始終了位置などの情報はログファイルの含んでいないため，計算して保存しておく必要がある．

前処理はビューアと別のプログラムで処理を行う．入力は selogger が出力された各ログファイルで，出力は「log.db」というデータベースファイルである．

データベースファイルのスキーマを以下に示す．追加情報と右側に注釈のついた項目についての詳細は後述する．

```
classes(    //classes.txt
    ClassID INTEGER PRIMARY KEY,
    Container TEXT,
    Filename TEXT,
    ClassName TEXT,
    LogLevel TEXT,
    MD5 TEXT
)
methods(    //methods.txt
    ClassID INTEGER,
    MethodID INTEGER PRIMARY KEY,
    ClassName TEXT,
    MethodName TEXT,
    MethodDesc TEXT,
    Access INTEGER,
    SourceFileName TEXT
)
Location(    //Location#####.txt
```

```
    DataID INTEGER PRIMARY KEY,
    ClassID INTEGER,
    MethodID INTEGER,
    Line INTEGER,
    InstructionIndex INTEGER,
    DataName TEXT,
    DataType TEXT,
    Label TEXT
)
ObjectTypes01(    //LOG$ObjectTypes01.txt
    ObjectID INTEGER PRIMARY KEY,
    ObjectHash INTEGER,
    TypeID INTEGER
)
Types(    //LOG$Types.txt
    TypeID INTEGER PRIMARY KEY,
    TypeName TEXT,
    ClassLocation TEXT,
    ParentTypeID INTEGER,
    ComponentType INTEGER
)
String(    //LOG$String#####.txt
    StrID INTEGER PRIMARY KEY,
    StrLen INTEGER,
    StrData TEXT
)
threads(    //追加情報
    ThreadNum INTEGER PRIMARY KEY,
    ThreadID INTEGER
)
t#(    //t-#-#####.slg, スレッドごとに表を作る
    EventNumber INTEGER PRIMARY KEY,    //追加情報
    CallStackID INTEGER,    //追加情報
    EventType INTEGER,    //追加情報
    MTime INTEGER,    //追加情報
    InsDir INTEGER,    //追加情報
    IOFlag INTEGER,    //追加情報
    DataID INTEGER,
    Value INTEGER,
    Timestamp INTEGER
)
CallStack(    //追加情報
    CallStackID INTEGER PRIMARY KEY,
```

```
    TopMethodID INTEGER,
    Count INTEGER,
    CallStackData TEXT
)
MethodIO#(    //追加情報，スレッドごとに表を作る
    EventNumber INTEGER PRIMARY KEY,
    EventNumberEnd INTEGER,
    CallStackID INTEGER,
    MethodID INTEGER
)
```

プログラムの実行履歴は，プログラムの実行が完了すると追記が起こることはないため，使用されたスレッドの個数に対応する数だけ表を作成し，表の大きさを小さくしている．

各表が持つ追加情報とは，selogger が提供しないデータである．それぞれ以下の内容を保持する．

- threads はスレッド番号情報を持つ表である．特別に，ThreadNum=-1 の項目を持ち，その ThreadID はスレッドの総数を表す
  - ThreadNum はスレッド番号に振られた連続の ID である
  - ThreadID はログ上のスレッド番号である
- t ＃は各スレッドのイベント列の表である．
  - EventNumber はイベントに振られた連続の ID である
  - CallStackID はイベント発生時のコールスタック状態の ID である
  - EventType はオブジェクト生成またはオブジェクトに対するメソッド呼び出しのフラグである
  - MTime はここまで発生したメソッド開始およびメソッド終了のイベント数である
  - InsDir は GUI 側で木を構築際に直前のノードに対する置き方のフラグである
  - IOFlag はメソッド開始またはメソッド終了のフラグである
- CallStack はコールスタック情報を持つ表である
  - CallStackID はコールスタック状態に振られた連続の ID である
  - TopMethodID はコールスタックの一番上のメソッドの ID である
  - Count はメソッド呼び出しよりこのコールスタック状態になった回数である
  - CallStackData はメソッド ID で表したコールスタックの内容である

- MethodIO ＃は各スレッドのメソッド開始と終了情報を持つ表である
  - EventNumber はメソッド先頭のイベント ID である
  - EventNumberEnd はメソッド末尾のイベント ID である
  - CallStackID はメソッド内でのコールスタック ID である
  - MethodID はこのメソッドの ID である

上記にデータベース化することより，GUI ビューアのプログラムの機能は SQL 文で処理できるようになる．

3.3 GUI ビューアについて

GUI ビューアは，データベース化された実行ログを木構造に表示する GUI プログラムである．入力ファイルは前処理プログラムの出力である log.db で，出力は GUI 画面である．

```
1
2  public class Main {
3      static int x,y;
4      private static int fibo(int x){
5          if (x<3) return 1;
6          return fibo(x-2)+fibo(x-1);
7      }
8
9      public static void main(String args[]){
10         x=7;
11         y=fibo(x);
12         System.out.println(y);
13     }
14 }
```

図 1: サンプルプログラム

図 1 のフィボナッチ数列を再帰で計算するプログラムに対し，図 2 の画面を出す．

図 2 のように，ウインドウを 2 分割して，左は各スレッドおよび検索結果の木を表し，右は木から選んだノードの詳細内容を表す．また，右下は計測した反応時間を出力する．

操作方法は，タブでスレッド番号を指定し，木を展開することコールスタックを対話的に指定する．その間ノードを選択することより詳細内容を右のウインドウで確認する．

あるノードと同じコールスタック状態の呼び出し群を検索する場合，そのノードにダブルクリックして,「search result」タブに移動する．図 3 に示したように，検索結果を部分木の集まりで表す．ただし，検索はすべてのスレッドに対して行う．

検索結果から元の位置を調べる場合，検索結果画面内のノードにダブルクリックすると，そのノードが属するスレッドのタブはハイライトされ，その中の木は対応したノードが出るまで展開されてハイライトされる．

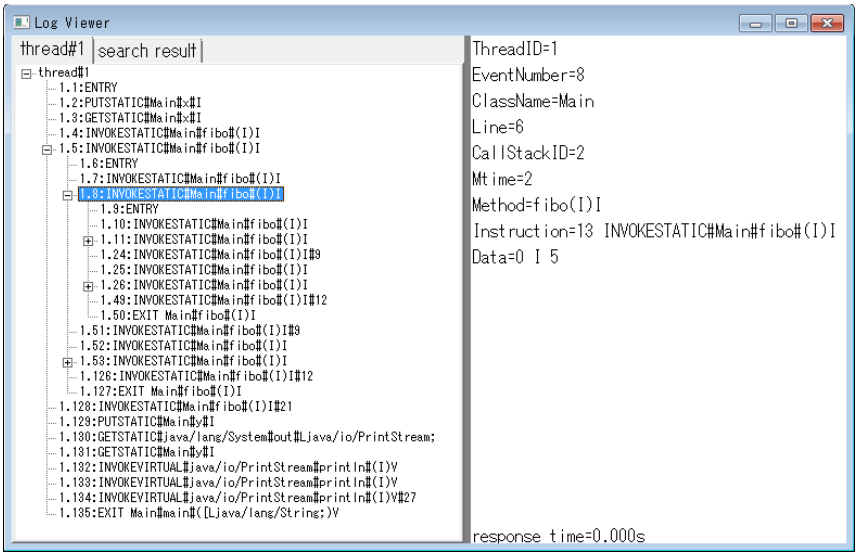

図 2: メイン GUI 画面

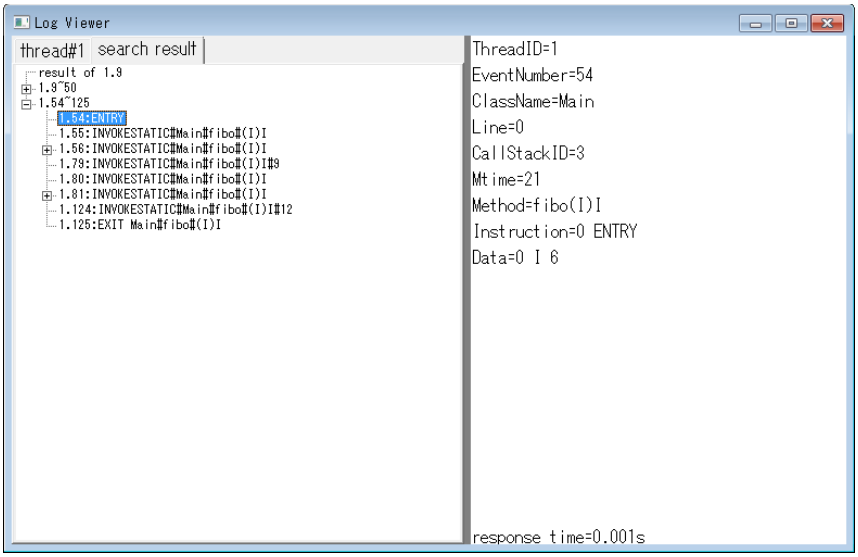

図 3: 検索結果画面

4　ツールのアルゴリズム

本章にツールを設計する際重要な内部論理について述べる．

4.1　コールスタック状態の自動計算およびコールツリーの構築

selogger の生データではコールスタック状態を持たない，ただイベントごとにメソッドの開始や終了であるかどうかが判定できる．そこでイベントを読み込む際にメソッドの開始終了およびコールスタックの状態を記憶してデータベースに記録すればコールスタック状態が復元できる．

4.1.1　原理

t-＃-＃＃＃＃＃.slg の DataID に対応した Location ＃＃＃＃＃.txt の Label の先頭が「EXIT」または「EXCEPTIONAL EXIT」（以下は「EXIT」と統合する）である場合，そのイベントはメソッド終了イベントに一対一対応である，一方メソッド開始イベントは必ず Label の先頭が「ENTRY」である（一対一対応でないのは，同じメソッドの「ENTRY」が連続の場合，それぞれのイベントは引数 1 つ 1 つ指定するイベントで，真のメソッド開始イベントはその「ENTRY」列の最初のイベントである）．従って，前処理プログラムは slg ファイルからイベントデータを読み込む度にその DataID に対応した Location ＃＃＃＃＃.txt の Label の先頭が「ENTRY」か「EXIT」かを判定し，オートマトンの状態を遷移させ，そのオートマトンの状態に従ってコールツリーを構築することができる．また，そのコールツリーを元に再構築関連の情報を残せば，GUI ビューアで木が再現できる．

4.1.2　データ構造

コールツリーを表現するために，以下のデータ構造を用いる．

- calltree 構造体 (CT)．この構造体はコールツリー上表示できるノードの役割を果たし，コールスタック関連の情報の他，親ノードおよび子ノードへのポインタを持つ．
- ctnext 構造体（CTN）．calltree の子ノードの数が不定であるため，それをこの構造体の双方向リストで補う．ctnext 構造体は付属する calltree 構造体へのポインタを持ち，リスト前後へのポインタを持つ．また，calltree の子へのポインタを ctnext 双方向リストの先頭および末尾に指すポインタで表現する．すると，calltree 構造体と ctnext 構造体でコールツリーが表現できる．

- methodio 構造体 (MIO)．メソッドの開始終了のイベント番号を記録する．コールツリーと別の層で，MIO で双方向リスト形のスタックを構築する．同一の MIO が複数の CT 構造体に共有されるため，メモリ上は有向グラフの形として見える．

各構造体の実際の定義は以下の通りである．

```
struct calltree{
    int id;    //CallStackID
    int top;    //MethodID
    int count;
    struct calltree *up;
    struct ctnext *first;
    struct ctnext *last;
};

struct ctnext{
    struct calltree *ct;
    struct ctnext *prev;
    struct ctnext *next;
};

struct methodio{
    int start;    //EventNumber
    struct calltree *ct;
    struct methodio *prev;
    struct methodio *next;
};

struct location{
    int mid;
    int iof;
    int type;
};
```

準備動作として，Location＃＃＃＃＃.txt を読み込む際に，location の構造体配列に Label の先頭の情報を残す．具体的に,「ENTRY」の場合 1 を,「EXIT」の場合-1 を，その他の場合 0 を iof に書き込む．

slg ファイルを読み込む際，DataID を添え字として location の構造体配列から iof を取得し，その値でオートマトンを駆動する．その際にコールツリーの構築およびメソッド開始終了の記録を行う．

構築を繰り返し，図 4 のような構造となる．

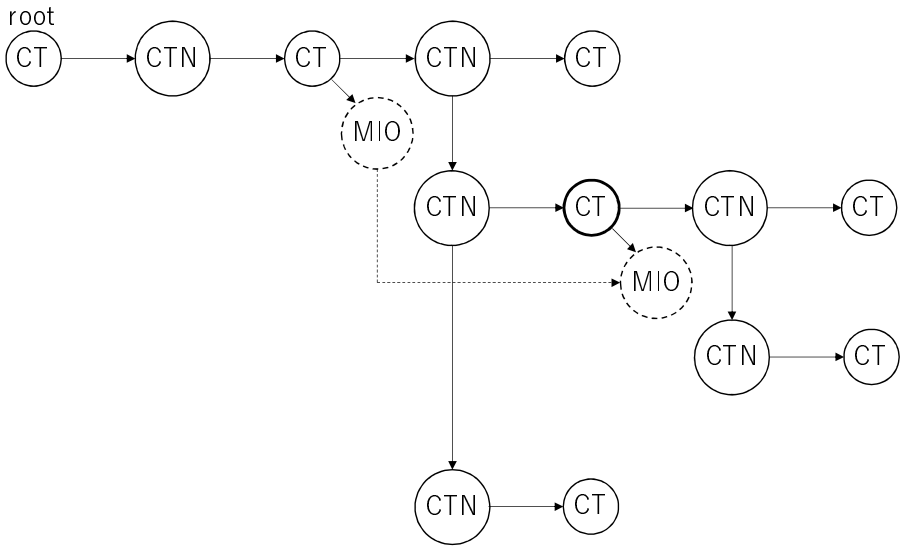

図 4: 木構造となる構造体

実行時の操作は下記で示す．コールツリーを扱うため現在位置のポインタが必要であり，図 4 の例では，太線の calltree 構造体が現在位置に指定されているものとする．

- メソッド開始

  まず現在位置のすべての子に一致したメソッドがあるかどうかを検索し，もし一致があれば現在位置をその場所に移す．もし一致がなければ，現在位置の ctnext のリストの末尾に ctnext 構造体を追加し，その ctnext 構造体に calltree 構造体を追加して，現在位置を新しい calltree 構造体に移す．いずれにせよ，最後に methodio のスタックに新しい methodio 構造体をプッシュする．

- メソッド終了

  現在位置を親に移し，methodio のスタックの最新情報をポップしてデータベースに出力する．

また，最後にコールツリーを壊す際，再帰呼び出しをなくすため，以下の手順で行う．

1. 現在位置を根ノードにする
2. 現在位置に子を持たず，かつ親を持たなければ，終了
3. 現在位置に子を持たなければ，6 へ進む
4. 現在位置を最初の子に移し，現在位置の情報をデータベースへ出力する

5. 3 に戻る

6. 現在位置を親に移し，現在位置の最初の子を削除する

7. 2 に戻る

4.1.3 オートマトンの状態遷移

```
in tree      out state
 0 ***        0   2
 1     ENTRY  1   4
 1     ENTRY  0   0
 0     ***    0   2
-1     EXIT   0   1
 0 ***       -1  -2
 1     ENTRY  1   4
 0     ***    0   2
-1     EXIT   0   1
 1     ENTRY  0   0
-1     EXIT   0   1
 0 ***       -1  -2
```

まず入出力について説明する．入力は location の構造体配列の iof で，出力は InsDir である．InsDir は iof と同様に 1，-1 と 0 で表すが，意味について，1 の場合はノードが直前ノードの子であり，-1 の場合は直前ノードの親の兄弟であり，0 の場合は直前ノードの兄弟である．

注意すべきなのは，EXIT の直後に ENTRY が来た場合である．その場合 ENTRY とその直前のノードが同じ階層関係となる必要があるため，その ENTRY の親ノードが存在しないこととなる．同じ階層であることを再現するため，ENTRY ノードの出力を 0 にする必要がある．

そこで，図 5 と表 1 で示した状態遷移を持つムーア型ステートマシンモデルを構築した．

表 1 は図 5 の実線の遷移を抜粋したものである．状態番号の振り方に工夫して，右に 2bit シフトした値は出力と一致し，二進数で 1 の桁は 0 入力および-1 入力の遷移先の分類に使い，二進数で 10 の桁は 1 入力の遷移先の分類に使うように設計した．

図 5 の破線の遷移は連続 ENTRY の際にメソッド ID が途中で変わった場合の例外用の遷移である．メソッド ID が異なる場合，その ENTRY の意味がパラメータ指定でなく新しいメソッドの先頭となるため，例外の遷移が必要となる．実装する際に，例外があり得る状態ではまず例外を検証してから，実線の遷移または破線の遷移に従って動作する．

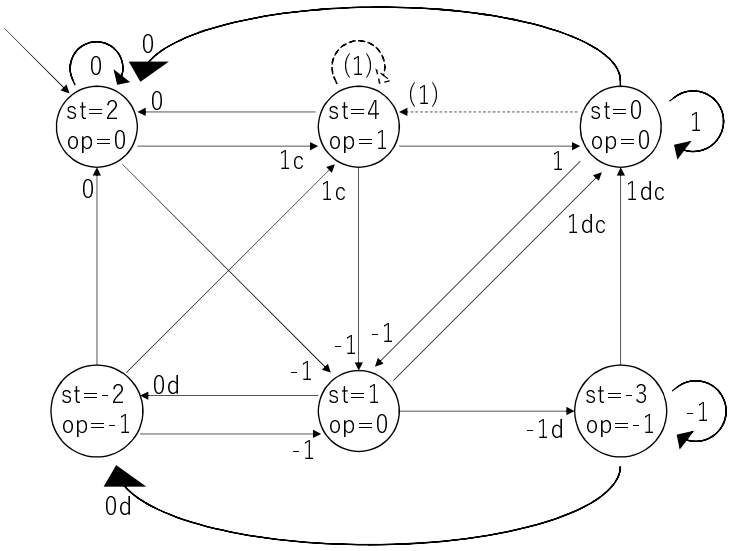

図 5: 状態遷移図

表 1: メイン状態遷移表

状態	状態（二進数）	出力	0	1	-1
→ 2	0010	0	2	4c	1
-2	1110	-1	2	4c	1
0	0000	0	2	0	1
4	0100	1	2	0	1
1	0001	0	-2d	0dc	-3d
-3	1101	-1	-2d	0dc	-3d

また，状態の後ろに「d」や「c」が付属したのは状態遷移の際にデータ構造への操作命令である.「d」は destruct の d で，メソッド終了の場合の methodio スタックのポップ動作を行う.「c」は construct の c で，メソッド開始の場合のコールツリーにノード追加および methodio スタックのプッシュ動作を行う.「dc」は「d」の動作を行って「c」の動作を行う.

4.2 メソッド区間のレコードおよびツリーの展開

InsDir を元に，ツリービューでノードを追加すれば木構造が再現できる．ただし，最初から最後まで全部読み込むのは長時間かかり，かつ大量のメモリを占めるため，必要でない情報を読み込まないようにする必要がある.

つまり，ノードを展開する際，その展開で表示する部分だけ読み込んで表示すればいい．そのため，メソッド開始番号から終了番号が検索できる MethodIO ＃の内容が不可欠である．展開の際に展開可能ノードを付加する場合，その中身を読み込まず，中にダミーノードを子として付加し，終了番号まで飛ばす.

また，展開する際の動作は下記で示す.

1. 展開するノードの最初の子が内容のあるノードであれば，終了
2. 展開するノードの最初の子が特別なダミーである場合，8 へ進む
3. 展開するノードの最初の子を削除し，展開するノードの次の位置を開始番号とし，それに対し MethodIO ＃から終了番号を取得する
4. 終了番号の直後の番号に対し MethodIO ＃から捜し，ヒットしたかどうかをフラグに残す
5. i を開始番号として，i が終了番号以下である間以下のことを行う
   (a) i 番の内容を読み込んでノートを追加する
   (b) もし iof が 0 か-1 の場合 i を 1 だけ増やしてループを再開する
   (c) ダミーを子ノードとして付加する
   (d) i 番に対し MethodIO ＃から終了番号を取得して，成功した場合 i を終了番号+1 に更新してループする
6. フラグ内容がヒットした場合，続き展開専用ノードおよびその子とした特別なダミーを付加する
7. 終了

8. 展開するノードの位置を開始番号とし，それに対し MethodIO ＃から終了番号を取得する
9. 展開するノードからの部分木を削除する
10. 4 に戻る

4.3 検索および検索結果からの逆検索

検索はすべての MethodIO ＃から現在ノードのコールスタック ID でメソッド開始および終了番号を検索して，その結果を展開可能ノードとして結果画面に表示する．

検索結果からの逆検索は，そのノードのスレッド ID およびイベント ID を取得して，そのスレッドの木からそのイベント ID のノードが出るまで展開してハイライトする．

展開の具体の手順は下記で示す．

1. 根ノードを展開して，現在位置を 1 番のノードとする
2. 現在位置の ID が入力より小さい間，現在位置を次の兄弟へ移すループする
3. 現在位置の ID が入力と一致すれば，ハイライトして終了
4. 現在位置の前のノードがなければ，8 へ進む
5. 現在位置の前のノードを記録し，現在位置を展開して，現在位置を記録したノードの直後に移す
6. 現在位置に子があれば，現在位置を最初の子に移す
7. 2 に戻る
8. 現在位置を展開して，現在位置を最初の子に移す
9. 2 に戻る

5　ツールの実装

ツールは C 言語で開発して，データベースは sqlite3.h ライブラリを利用し，GUI 関連は windows.h ライブラリを利用する．

Win32API の制限より，カウンタが int 型で，各スレッドのイベント数が $2^{31}-1$ 以下（slg ファイルの最大番号が 214 以下）という制限がある．

本章の残りは実装する際に遭った問題に対する解決を述べる．

5.1　sqlite3 関連の高速化

データベース作成時大量のデータ挿入操作が必要である．もしそのまま SQL 文に対し sqlite3_exec で実行するのは遅い．そこで sqlite3 高速化関連のブログ [3] を参考に，ローレベル API を利用して，同じ表への連続挿入動作の高速化ができた．

しかし，ブログの内容のままで実行するとまだ遅い．時間計測で見た現象は CPU 時間の消費より，実時間の消費が何割りないしは何倍長い．なぜなら，slg ファイルを処理する際にデータベース内の表は書き込み専用だけでなく，読み込み専用のと読み書きの表も混在しており，かつ書き込み動作は 2 つの表へ交互に挿入することになるため，同じファイルに対する同期および読み書き状態の切り替えのオーバーヘッドに生じたからである．

そこで考えた解決策は，データベース内容の分割である．最初 slg ファイル以外の部分を読み込むとき，slg ファイルを処理する際に触れない表だけ log.db に直接書き込む．読み書きおよび読み込み専用の表は全部メモリ上で論理的に操作して，処理が全部終わって log.db に書き込む．書き込み専用の表のうち，小さい方は一時ファイル tmp2.db に書き込み，処理が全部終わってマージする．

この方法で，データベースの読み込みをなくし，書き込みを専一にし，高速化の前提に満たすようにした．

5.2　slg ファイルの読み込み

slg ファイルは Java の ObjectOutputStream で出力されたバイナリデータである．Oracle 社のドキュメント [9] を読みながら十六進数エディタで開いて調べると，内部の構造は以下のようになる．

- 固定の 4 バイト：AC ED 00 05
- 0 個以上の 7A，4 バイト 0x100〜0x400 のビッグエンディアン整数，プラスその値バイト分のデータ片

- 最後に 1 個または 0 個 77，1 バイトの符号なし整数，プラスその値バイト分のデータ片

また，同じスレッドからの全ファイル中のデータ片全体の集まりは 20 バイト分（int，long，long 型のビッグエンディアンバージョン）の構造体からなる配列である．ファイル分割は 10000000 レコードごとに行われる．

その仕様より，slg ファイルの内容を抽出するため，余分の分割情報の除去，断片間の連結およびエンディアン変換機能が必要である．

エンディアン変換はバッファのバイト単位逆順書き込みと強制ポインタ型変換で実装した．

分割情報の除去および断片間の連結は，オートマトン式で行っている．注目すべきなのは 1024 と 20 の最大公約数が 4 であることで，断片連結は 4 バイト単位で処理するのが十分，1 バイト単位での処理よりアクセス回数が軽減できる．

スレッドごと最初の slg ファイルに空のバッファを用意し，カウンタを 0 で初期化して，slg ファイルごとに以下の動作を行う．

1. 余分の 4 バイト（AC ED 00 05）を読み飛ばす
2. カウンタが負であれば異常終了する
3. カウンタが 0 でなければ 8 へ進む
4. 1 バイトを読み，読み込み失敗なら正常終了，7A なら 5 へ，77 なら 7 へ進み，その他は異常終了する
5. 4 バイトを読み，エンディアン変換してカウンタに足す
6. 8 へ進む
7. 1 バイトを読み，符号なし整数としてカウンタに足す
8. もしバッファが空でないまたはカウンタが 20 より小さい場合，11 へ進む
9. バッファに 20 バイト分読み込む
10. 13 へ進む
11. バッファに 4 バイト分追加読み込み，カウンタを 4 だけ減らす
12. もしバッファが 20 バイト未満なら 2 に戻る
13. バッファの内容を処理して，空にする
14. 2 に戻る

なお，異常終了はファイルのフォーマットが想定外である場合のみ発生する．

6　評価

ツールの実用性を評価するため，実行時間計測を行う．ツールは前処理部分（コンソールプログラム）とビューア（GUI プログラム）の 2 つに分かれたため，時間計測も両方に対して別々に行う．

6.1　時間の計測

実行時間はログの大きさおよびコールツリーの構造に強く依存する．長い実行履歴を含むログファイルを生成し最悪ケースの実行時間を調べるため，実験でログを取る対象を DaCapo ベンチマーク [1]DaCapo-9.12-bach.jar にした．

前処理プログラムに対しては，処理中標準エラー出力の情報の最後に出力された実行時間を収集する．また，参考のため入出力データのサイズも記録する．

GUI ビューアの内容表示および展開の時間計測シナリオは，スレッド 1 に対し，ハイライトしたノードと同じレベルの展開可能ノードのうち，展開区間の最長ノードを展開して展開時間を計測して，追加された内容の先頭ノードをクリックして内容表示の時間を計測する．このことをできないまで（葉に到達した），または 20 回になるまで繰り返しデータを収集する．

また，GUI ビューアの逆探索時間計測は人間の操作ではやりづらく，時間計測専用バージョンを作って，時間計測を行う．動作は，起動してすぐに ThreadID と EventNumber で指定されたノードを展開する関数を呼び，プログラムが安定する次第起動からの時間を出力して終了する．ノードの位置より展開と検索の時間が大幅に変動するため，実験は全ノードを線状に並べて区間を 100 等分し，その各区間から 1 個ノードをランダムに指定し計測対象ノードとする．

6.2　結果

ツールを cygwin64 の gcc（x86_64-pc-cygwin-gcc.exe，5.4.0 バージョン）でコンパイルして，表 2 のコンピュータ環境で,「small」をパラメータとして各ベンチマークを実行し，実行が成功したかつ制限を満たすログにツールを適用したところ，表 3 と表 4 の結果を得た．

生データサイズについては，0.515GB の静的部分（プログラムだけに依存し，実行に依存しない部分）を含む．

「計測不能」ということは，途中で反応時間が 10 分間以上かかるケースに遭って，実験を中断せざるを得なかったことより，最大値が入手できないことである．その場合の平均値は計測不能のケース抜きで計算する．ちなみに，計測不能のケース抜きの最大値は 1270ms である．

表 2: コンピュータ環境

項目	タイプ	スペック
CPU	Intel Xeon E5-1620	3.6GHz
メモリ	DDR3 1600	32GB
SSD	SanDisk SDSSDHII	894GB
OS	Windows 7 Professional SP1	64bit

表 3: 計測結果（前処理）

名前	生データ サイズ (GB)	前処理時間	DB サイズ (GB)
tradesoap	0.546	1’12”96	0.547
tradebeans	0.546	1’13”13	0.547
pmd	0.717	1’49”36	0.877
fop	1.39	4’24”46	2.06
luindex	1.49	4’25”15	2.14
batik	2.79	8’39”27	4.39
xalan	7.59	25’36”76	12.6
jython	8.86	34’15”62	16.6
lusearch	13.1	44’51”88	22.1
avrora	71.1	263’42”49	128

表 4: 計測結果（GUI ビューア）

名前	起動時間 (ms)	内容表示時間 最大 (ms)	展開時間 最大 (ms)	展開時間 平均 (ms)	展開時間 中央値 (ms)
tradesoap	39	5	44099	4773	20
tradebeans	44	5	44143	4758	20
pmd	37	5	1065	68	9
fop	34	5	934	64	14
luindex	33	5	980	63	8
batik	37	6	計測不能	106	28
xalan	48	5	67	17	8
jython	36	5	400	37	9
lusearch	53	5	578	63	9
avrora	44	5	77	20	13

GUI ビューアの逆探索の代わりの実験については，サイズの一番小さい tradesoap については最大 55096ms，平均 18567ms，中央値 13580ms の結果が得られた．最大時間が約 1 分になったため，より大きいサイズの実行履歴に対し実行時間が 1 分以上かかると予測し，実験を中止した．

展開不能ノードの影響を調べるため，batik について同じ実験を行うと，1 分間以内に成功した実行は 21 個，10 分間以内に成功した実行は 22 個しかない．

6.3 議論

図 6 は生データサイズとデータベースファイルサイズの関係図である．データベースファイルのサイズは生データのサイズに線形関係が見られ，かつほぼ比例する．つまり前処理に追加した情報のサイズは生データサイズにほぼ比例すると意味する．

図 7 はデータベースファイルサイズと前処理時間の関係図である．前処理時間はデータベースファイルのサイズに線形関係が見られ，かつほぼ比例する．これは実行時間のボトルネックはハードディスクの書き込み時間であると考えられる．実際に，もし SSD を機械ハードディスクに変更した場合，時間が何倍もかかるようになる．

GUI の起動時間はデータベースサイズに関係なくほぼ一定の時間がかかる．また，内容表示時間もデータベースサイズに関係なくほぼ一定の時間がかかる．これはデータベースを利用することで高速にアクセスできることを証明した．

展開時間について，結果とサイズの関連の規則性がない．なぜなら，展開は必要の部分だ

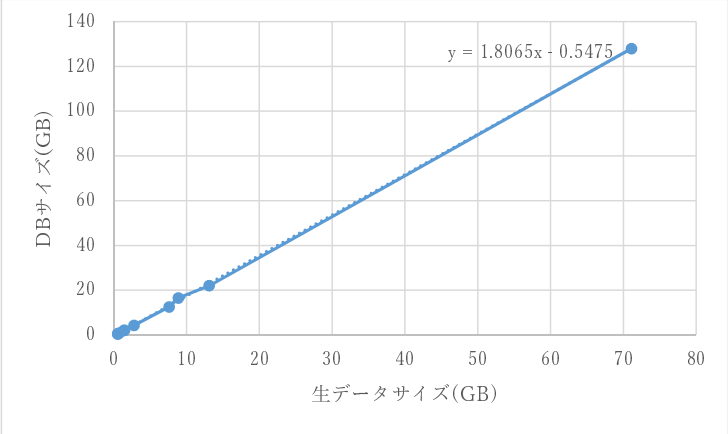

図 6: 生データサイズと DB サイズの関係図

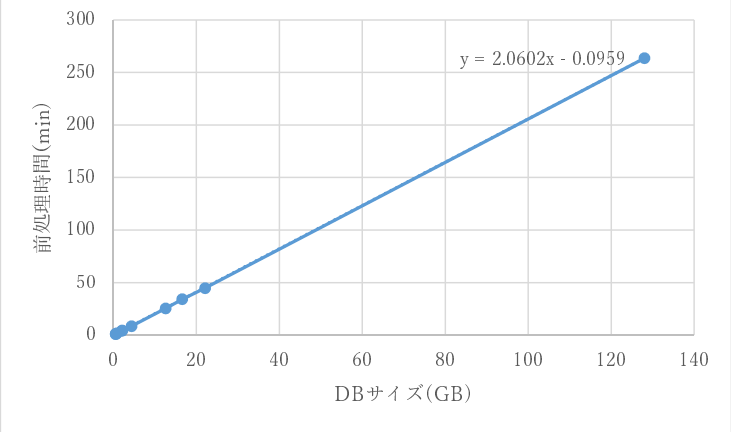

図 7: DB サイズと前処理時間の関係図

け読み込んで表示するため，データベースサイズでなく，実際の実行上同じメソッド内のイベント数に依存する．シナリオどおり実験した結果だけに対しまとめると，対話的に範囲の長い部分を捜そうとしても，1 秒以上かかるケースが少なく，そのうち 1 分間以内に反応できるケースが多い，ただし小さい確率で 10 分以上かかるケースに遭う．計測不能のケースは，展開すると一度に表示すべきノード数が多すぎることである．今回の実験の対象はベンチマーク，つまり時間を計測するためループを意図的に長くしたもので，この状態に陥るのは仕方がない．もしプログラム動作を理解しようとしたら，ループ回数の少ない入力で実行過程を見ることが予想され，その場合計測不能のケースが現れないと予想できる．

逆探索の代わりの実験について，tradesoap の実行ログに対しては，逆探索の反応時間の上限が約 55 秒であることが判明した．逆探索は展開の繰り返しで，展開の反応時間の上限も約 55 秒である．tradesoap の実行ログには約 163 万イベントの履歴があり，実際プログラム動作を理解するための実行で履歴のイベント数を約 163 万に抑えれば，反応が 1 分間以内で抑えられ，対話的に閲覧可能となる．

計測不能な領域を持つ batik に対して逆探索の代わりの実験を行って，展開時間の長すぎるノードは多数のノードを隠していることが判明した．また，逆探索不能ノードのうち，展開不能なノードに遭ったケースだけでなく，長い連続領域に遭ったケースもある．連続領域は，EXIT イベントの直後に ENTRY イベントが来るという区間の連鎖で，連鎖先頭の部分を表示するのは少ない展開回数で済むが，後ろになればなるほど展開回数が増え，全体的な展開時間が長くなる．これも長いループより生じたもので，動作を理解するためのループ回数の少ない実行では回避できる．

以上の結果より，ツールを実際に応用するとき，ループ回数を少なくする入力を指定することを推奨する．このことよりノード数が減り，ツールの実行がスムーズにでき，かつ注目ポイントが捜しやすくなる．目安として，約 163 万イベントの履歴がほぼ対話的実行の上限となる．その場合，ログの動的部分が少なく，前処理も速くできる．

7 妥当性への脅威

7.1 外部妥当性への脅威

- データベースの設計は selogger が持つオブジェクト，メソッドなどの Java 固有の概念に依存しており，Java 以外の言語への対応が難しい．
- 本研究では，GUI ビューアの実装に Win32API を用いた．この API が持つ内部的な制限により，実行ビューアは単一スレッドに非常に長いログを持つ場合は対応できない．
- Java プログラムが出力した実行履歴データの読み取りに Java の API を使っていないため，もし Java の出力プロトコルが変更された場合プログラムを書き直す必要がある．

7.2 構造妥当性への脅威

- 本研究で報告している性能は，すべてベンチマークを対象とした実験結果である．これらは様々なプログラムの実行例を収録してはいるが，実際のプログラムを対象とした適用の状況とは異なる可能性がある．
- GUI ビューアに対する時間計測は人為的に作られたシナリオに基づいたものである．実際のユーザの利用の方法とは異なる可能性がある．
- 逆検索機能において，時間計測の対象となるノードは区間ごとにランダムに選んだ．これらは代表性を持たない可能性がある．

7.3 内部妥当性への脅威

- 本研究では Java プログラムの実行履歴を分析するツールを構築したが，木構造を損なうような実行履歴の出現パターンは発生した具体的な問題例からの推定であり，今の設計を覆す実行例が存在するかもしれない．

8　結論

本研究では，実行ログを用いて実行履歴を対話的に分析するツールを提案した．このツールは実行履歴をコールツリーで表現し，プログラムの実行過程を便利に調べることができるようにした．さらに同じコールスタック状態のメソッド実行のサブツリーの抽出や，その抽出されたサブツリーから元の位置が逆探索でき，実行過程に対する更なる理解が見込める．

実験から，非常に長いループを含まない実行ログに対し，ツールが対話的に使えることを示した．スムーズに閲覧できる範囲としては，実行履歴のイベント数が約 163 万に抑えることを推奨する．この数字は，プログラミング演習等における学習用プログラムなどのイベントとしては十分である一方でサーバなどに常駐して動作を続けるようなプログラムの分析への適用は難しいと考えられる．

9　今後の課題

ツールはプログラムの理解へ支援するために内部の呼び出し関係や具体的なイベントの系列は可視化するが，各メソッドなどで発生しうるイベントの内容などについては，ソースコードの内容に対する事前知識がないと理解しにくいという弱点がある．この点は，REMViewer [8] など既存の可視化ツールとの連携が重要であると考えられる．

また，本研究ではコールスタックを検索の中心としたが，Object-Centric Debugging [11] で提案されているオブジェクトやデータ内容を中心に検索する機能の拡張も考えられる．

謝辞

本研究において, 常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします.

本研究において, 適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします.

本研究において, 懇切丁寧な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く感謝いたします.

本研究において，客観的な御助言を頂きました元大阪大学大学院情報科学研究科コンピュータサイエンス専攻松村俊徳様に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of OOPSLA*, pp. 169–190, 2006.

[2] L. C. Briand, Y Labiche, and J. Leduc. Towards the reverse engineering of UML sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, Vol. 32, No. 9, pp. 642–663, 2006.

[3] chenguanzhou123. 提升 sqlite 数据插入效率低、速度慢的方法. `http://blog.csdn.net/chenguanzhou123/article/details/9376537`, 2013.

[4] Takashi Ishio. Selogger project. `https://github.com/takashi-ishio/selogger`, 2015.

[5] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 185–194, May 2010.

[6] Bil Lewis. Debugging backwards in time. In *Proceedigns of the International Workshop on Automated Debugging*, 2003.

[7] A. Lienhard, T. Girba, O. Greevy, and O. Nierstrasz. Test blueprints – exposing side effects in execution traces to support writing unit tests. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pp. 83–92, April 2008.

[8] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-executed-method viewer for efficientvisualization of execution paths and states in java. In *Proceedings of the 22nd IEEE International Conference on Program Comprehension*, pp. 253–257, 2014.

[9] Oracle. Object serialization stream protocol. `https://docs.oracle.com/javase/7/docs/platform/serialization/spec/protocol.html`, 2016.

[10] Jochen Quante and Rainer Koschke. Dynamic object process graphs. *Journal of Systems and Software*, Vol. 81, No. 4, pp. 481–501, 2008.

[11] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proc. of ICSE*, pp. 485–495, 2012.

[12] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1038–1044, 1992.

[13] 櫻井孝平, 増原英彦, 古宮誠一. Traceglasses：欠陥の効率良い発見手法を実現するトレースに基づくデバッガ. 情報処理学会論文誌 プログラミング（PRO）, Vol. 3, No. 3, pp. 1–17, 2010.

[14] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. プログラム実行履歴からの簡潔なシーケンス図の生成手法. コンピュータソフトウェア, Vol. 24, No. 3, pp. 153–169, 2007.