修士学位論文

題目

プログラムの一時停止時に将来の実行情報を提供するデバッガ

指導教員
井上 克郎 教授

報告者
冨永 真司

平成 30 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 29 年度 修士学位論文

プログラムの一時停止時に将来の実行情報を提供するデバッガ

冨永 真司

内容梗概

デバッグとは，プログラムに対してテスト入力を与えたときに期待した通りに動作しない問題，すなわちバグの原因を特定し，その原因を解消するように対象プログラムの修正を行う作業である．デバッガは，デバッグ作業の中でも，バグの原因の分析や位置の特定の際に使用される支援ツールであり，開発者が指定したブレークポイントでプログラムの実行を一時停止したとき，プログラムの現在の状態，すなわちプログラムの中に出現する変数名とその値を変数ビューという形式で開発者に提示する．デバッガを用いると，プログラムの任意の時点の状態を調査することが可能となるが，一方で，一時停止した時点からプログラムの実行がどのように進行するかは，開発者が変数の値を手掛かりにしながら目視でプログラムを読解しなくてはならない．多くのデバッガはステップ実行という機能も備えているが，プログラムの実行を進めると後戻りが困難であることから，現在の状態をもとに将来の実行情報を推測する作業が必要となる．

本研究では，開発者がプログラムの実行経路や，そこで参照されるデータの値，実行時エラーの発生の有無を調査する作業の労力を軽減するために，デバッガで得られたプログラムの実行状態をもとに将来の実行経路を先読みし，それ以降の命令で使用される変数の値やメソッドの返り値，実行時エラーの発生可能性を提示するデバッガを提案する．具体的な方法として，デバッガが変数ビューに表示するためにデバッグ対象プログラムから取得する変数情報を用いて，デバッグ対象プログラムの命令を実行した場合に発生するメモリアクセスを計算し，対象プログラムには影響を与えずに，プログラムの実行経路を計算する．Eclipse JDT で利用できるデバッガの変数ビューの機能を拡張して，計算された将来の実行経路で参照される変数および実行時エラーの発生可能性を提示する機能を実現した．

主な用語

デバッガ
動的解析
Java バイトコード解析

目 次

1 前書き **3**

2 背景 **5**

2.1 デバッグ作業 5

2.2 実行トレースによるデバッグ支援 5

2.3 実行トレースの分析 6

3 提案手法 **7**

3.1 プログラムの一時停止場所に対応するクラスファイルの特定 9

3.2 クラスファイルのバイトコード命令の読み取り 9

3.3 バイトコード命令から使用される変数やメソッドの特定 10

3.4 未知の値を含む実行の先読み 15

3.5 実行時エラーの発生の先読み 16

3.6 提案手法の制限 16

4 ツール実装 **18**

4.1 実装方法 18

4.2 実行例 18

4.3 有効性の考察 21

5 まとめ **22**

謝辞 **23**

参考文献 **24**

1　前書き

ソフトウェアの開発および保守における開発者の作業の1つにデバッグがある．デバッグとは，プログラムに対してテスト入力を与えたときに期待した通りに動作しない問題，すなわちバグの原因を特定し，その原因を解消するように対象プログラムの修正を行う作業である．この作業では，障害につながる命令の実行順序や変数の値を観測、調査することが重要である [13].

プログラムのバグによって発生する障害による損失は一般に大きく，また，その修正に必要なコストも大きい [3]．理想的には，情報システムが運用される期間に発生するあらゆる状況を事前にテストし、すべての欠陥を事前に取り除くことが望まれるが，現実には難しい。たとえば2007年10月に首都圏で発生した自動改札機の動作障害 [5] は，1年間のテストと半年間の運用を経た後に初めて発生したものであり，開発者は短期間でのデバッグおよび修正作業を行っている．

デバッグ作業は，バグによって引き起こされる問題を再現するようなテスト入力の特定，それを手掛かりとしたバグの原因の分析，問題のある命令の位置の特定，バグの修正，そしてテストによる修正の確認という手順からなる [15].

デバッガはこれらの作業の中でも，バグの原因の分析や位置の特定の際に使用される支援ツールである．計算機が実行する本来の命令は機械語であるが，デバッガは開発者が記述したプログラミング言語の表現を使って実行中のプログラムの情報を調査することを可能とする [11]．開発者はデバッグ対象プログラムを理解するために，ソースコードを読むだけでなく，デバッグを用いてプログラムの実行時の状態を調査する [14].

デバッガには様々な形態があるが，Eclipse[9] などの統合開発環境として広く利用されているデバッガはブレークポイントと呼ばれる機能を提供している．これらのデバッガは，開発者が指定したブレークポイントでプログラムの実行を一時停止し，実行中のプログラムのある時点でのメモリの状態を確認することを可能とする．変数の値が不正な値や想定外の値であるとそれがバグの原因となることが多いため，デバッガに対してステップ実行を指示することによってプログラムの実行を1行ずつ進めながら，変数の値やメソッドの返り値が期待通りのものであるかどうか確認する．

デバッガにおいて変数の値を確認する機能は変数ビューと呼ばれている [11]．変数ビューは，プログラムの中に出現する変数名とその値を開発者に提示する．Java の開発環境である Eclipse JDT のデバッグ環境に実装されている変数ビューの場合は，現在実行中のメソッドで参照可能なローカル変数の名前と値の組が表示され，また，オブジェクトに関してはその構成要素であるフィールドの一覧が，配列に関してはその要素の値がリストとして表示される．

デバッガを用いると，プログラムの任意の時点の状態を調査することが可能となるが，一方で，一時停止した時点からプログラムの実行がどのように進行するかは，開発者が変数の値を手掛かりにしながら目視でプログラムを読解しなくてはならない．多くのデバッガはステップ実行という機能も備えているが，プログラムの実行を進めると後戻りが困難であることから，現在の状態をもとに将来の実行情報を推測する作業が必要となる．プログラムに書かれた式の形からその値を可視化するインスペクタというツールも提供されているが，開発者が内容を閲覧したい式を個別に指定する必要があるほか，対象プログラムの機能を呼び出すことで実現されているため，対象プログラムの状態自体に予期せぬ変化を与えてしまう可能性がある．そのため，プログラムの実行に影響を与えずに変数の値の変化などを確認したい場合は，開発者は各メソッドで実行される命令を把握し，自分で計算する必要がある．

本研究では，開発者がプログラムの実行経路や，そこで参照されるデータの値，実行時エラーの発生の有無を調査する作業の労力を軽減するために，デバッガで得られたプログラムの実行状態をもとに将来の実行経路を先読みし，それ以降の命令で使用される変数の値やメソッドの返り値，実行時エラーの発生可能性を提示するデバッガを提案する．対象プログラムの実行に影響を与えないようにするために，デバッガが変数ビューに表示するためにデバッグ対象プログラムから取得する変数情報を用いて，デバッグ対象プログラムの命令を実行した場合に発生するメモリアクセスをデバッガ側で計算する．

具体的な実現方法としては，Eclipse JDT で利用できるデバッガの変数ビューの機能を拡張する形とした．各ステップで実際に使用される変数はビューに並んだ変数の一部であることが多いことから，ブレークポイントやステップ実行によってプログラムの実行が一時停止したとき，それからの実行で使用される変数の値や実行時エラーの発生可能性を，変数ビューに追加する形で可視化する．

以降，2 章では本研究の背景について述べる．3 章では提案手法について述べる．4 章ではまとめについて述べる．

2　背景

デバッグ作業に関しての現状と基本的なデバッグ作業に対する支援，プログラムの実行を記録・再生する実行トレースを用いたデバッグ支援，および実行トレースの分析手法について述べる．

2.1　デバッグ作業

デバッグは重要な作業であるが同時に難しい作業とも言われている [15]．Perscheid[8] らは 4 つのソフトウェア開発会社とオンラインでデバッグ作業に関する聞き取り調査を行った．その結果，多くの開発者は printf デバッグや assert 文，ステップ実行といった機能を頻繁に利用する一方で，そのほかの高度なデバッガの機能については知らないか，知っていても利用していないことが判明した．

Whyline[6] は，実行履歴に対して質問を重ねていくことで問題箇所を絞り込むデバッガである．デバッグ作業は，プログラムの振る舞いに疑問を持ち，それをツール等により確認するという特徴があり，それをうまく活用した手法であると言える．質問内容としては，なぜ対象地点に到達したのか，なぜある変数にはその値が入っているのか，といった質問を行うことができ，その質問内容に合致するプログラム実行上の時点へ移動することができる．

Object-Centric Debugging[10] は，オブジェクトに着目したデバッグ手法である．通常のデバッガでは，ソースコード上にブレークポイントを設置し，プログラムの実行がその地点に到達した際に，プログラムが停止する．一方，Object-Centric Debugging では，オブジェクトに対しブレークポイントを設定し，指定したオブジェクトに対して変更された際にプログラムが停止する．

ブレークポイントによるデバッグはプログラムが一時停止する一方，printf デバッグではあらかじめ必要な値を想定してプログラムを改変しておく必要がある．嶋利ら [16] はソフトウェアを書き換えず停止もさせないモニタリング手法により、printf デバッグの柔軟な実現を可能にした．

2.2　実行トレースによるデバッグ支援

Omniscient Debugging[7] はプログラムの実行を全て記録し，プログラムの状態を再現することで，任意の地点のプログラムの状態を閲覧可能なデバッガである．Omniscient Debugging では一度に閲覧可能なのはある 1 つの時点での実行時情報のみであるため，プログラム中の同じ命令を複数回実行した場合にはそれぞれを調査する必要がある．

この問題に対処するために，松村ら [18] は Java メソッドの複数回の実行経路を可視化するツール REMViewer を提案した. バイトコード実行を記録しておくことで、メソッドを指

定すると, そのメソッドの実行すべてを再生し，実行経路によって分類したものを提示する．これにより複数回の実行を分類し整理して可視化することができる．一方で，これらの手法は実行トレースを保存しておく必要があり，REMViewer では数秒の実行で 1GB 近い履歴が生成されるなどのコストがかかる．

実行トレースの取得方法を工夫することで，コストを削減する試みが行われている．Bond ら [2] は Null Pointer Exception の発生原因を特定することに特化した手法を考案した．すべての実行トレースを取得せず null が使用されたときに限ってマーカーを与え，そのマーカーを使用することにより軽量な実行トレースの追跡を実現している．

2.3 実行トレースの分析

Relative Debugging[1] は,2 つのプログラムに同じ入力を与えて実行を比べることで欠陥を特定する技術である．正しく動作していた旧バージョンのプログラムに改変を加えたところ，新バージョンでは正しく動作しなくなった場合などに，旧バージョンと新バージョンの実行を比べることで，効率的なデバッグが可能となる．

松村ら [17] および松田 [19] は，2 つのプログラムの実行を比較し差分を自動的に抽出する手法を提案した．松村らの手法では，グラフの比較により差分のある命令およびその差分が影響を及ぼす先の命令を検出することができる．松田の手法ではグラフをテキストで表現したものに対し diff を用いることで，検出対象が差分のある依存関係に絞られる代わりに巨大なグラフの比較を可能にした．

Daikon[4] はプログラムの実行を分析し，実行における不変条件を検出するツールである．

3　提案手法

本研究では，ブレークポイントによりプログラムが停止している状態で，プログラムの将来の実行をプログラム自身の実行とは独立に計算し，使用される変数や例外発生の可能性を変数ビューに提示する機能を搭載したデバッガを提案する．このデバッガは，デバッグ対象のプログラムがある行 l でプログラムの実行を停止した時，その位置から参照可能なメモリ領域のリスト $M(l)$ を取得し，それに基づいて将来の実行で参照されるメモリ領域のリスト $V(l)$ を計算する処理だと考えられる．

本研究では，Java プログラムを対象とした実装を行った．提案手法の考え方そのものは特定の言語に依存するものではないが，実装は Java バイトコードという解析が容易なバイナリ形式に依存したものである．

Java プログラムがあるメソッド m の中のある行 l で停止した時，m があるクラス c で定義されているとすると，以下の変数が参照可能である．

- m の中で宣言されたローカル変数
- c で宣言されたフィールド
- c の親クラスのフィールドのうち，可視性の制約を満たすもの
- 可視性の制約を満たす `static` フィールド

また，上記の参照可能な変数を v とすると，以下のメモリ領域も参照可能である．

- v を通してアクセス可能なフィールド
- v を通してアクセス可能な配列の個別の要素

オブジェクトや配列や配列の要素は，他のオブジェクトのフィールドや配列の要素となりうる．そのため，参照可能な変数や配列の要素は，参照可能なメモリ領域を基点とした階層構造として表現できる．

本研究で実装するデバッガの基礎である Eclipse JDT の従来の変数ビューでは，プログラムがあるメソッド m のある行 l で停止した時，以下の項目を表示する．

- m がインスタンスメソッドである場合，m を実行しているオブジェクト参照 `this`
- m の中で宣言され，l よりも前に値を代入されたローカル変数

`this` を通して参照可能なフィールドは，`this` の子要素として `this` のツリー構造内に表示される．なお，`static` や `final` を付与したフィールド変数は参照可能であっても変数ビューには表示されない．

本研究で提案する変数ビューは，プログラムがメソッド m のある行 l で停止した時，従来の変数ビューで表示されるものに加え，さらに以下の項目を表示する．

- 行 l と同一行にある命令を実行した時に値を参照されるオブジェクトのフィールドや配列の要素．ただし `static` や `final` を付与されているものを除く．
- 行 l と同一行にある命令を実行した時に呼び出されるメソッド

ここでの命令の実行とはデバッガにおける Step Over 操作を一度だけ適用することを指す．オブジェクトのフィールドや配列の要素は，従来の変数ビューではツリー構造の中にのみ存在したが，本研究で提案する変数ビューでは，ツリー構造の中の項目は残したまま個別にツリー構造の外に追加する．例えば，obj.var というフィールド変数や num[1] という配列の要素は従来の変数ビューでは obj や num の項目を展開したツリーの中にのみ存在したが，本研究で提案する変数ビューでは，obj や num と同じ階層に obj.var や num[1] という項目を追加する．

また，メソッドに関してはメソッド名とその返り値の型を表示する．

従来の変数ビューの変数の表示順は以下の通りである．

1. ローカル変数：変数の宣言順
2. フィールド変数：アルファベット順

一方，本研究で提案する変数ビューの項目の表示順は以下の通りである．

1. 行 l と同一行にある命令を実行した時に値を参照される変数や配列の要素および呼び出されるメソッド．これらは命令の実行順序に従い，参照順に表示する．
2. 上記以外の変数．これらは従来の変数ビューと同じ順番で表示する．

メソッド m と命令行 l から変数ビューに表示する変数のリスト $V(l)$ を求める操作は，Eclipse4.5.1 において `JavaStackFrameContentProvider` クラスの `getAllChildren` メソッドに実装されている．このメソッドには，デバッグ対象プログラムにアクセスして，m や l の情報を持つ `JDIStackFrame` オブジェクトから，$V(l)$ に対応するオブジェクトの配列を計算する処理が実装されている．本研究ではこのメソッドを拡張して，変数を並べ替え，さらにメソッドを追加した $V(l)$ を計算する処理を実現した．

$V(l)$ の具体的な計算手順は以下の通りである．

1. プログラムの一時停止場所に対応するクラスファイルを特定する
2. 特定したクラスファイルからバイトコード命令を読み取る

3. 読み取ったバイトコード命令を解析し，使用される変数およびメソッドを特定する

以降，各手順について順番に説明し，最後に実装上の制限について述べる．

3.1 プログラムの一時停止場所に対応するクラスファイルの特定

プログラムの実行が一時停止した時，その命令行 l に記述された命令を知るために，まず停止場所のクラスがどのクラスファイルであるかを特定する．

Java は実行時にファイル名等から任意のクラスをロードする仕組みを持つため，単純に実行時に与えられたオプションや対象プログラムの記述からだけではクラスを判別することが出来ない．そのため，本研究では，Java Debug Interface(JDI) と呼ばれるデバッグ対象プログラムの仮想マシンにアクセスするためのインターフェースを使用して，デバッグ対象プログラムを実行している Java 仮想マシンで以下の処理を実行させる．

1. メソッドを実行中のクラスに対応する `java.lang.Class` オブジェクトを取得する．
2. そのオブジェクトに対して，`getProtectionDomain().getCodeSource()` メソッドを呼び出し，クラスの呼び出し元を表現する `CodeSource` オブジェクトを取得する．
3. 得られた `CodeSource` オブジェクトの `getLocation()` メソッドを使用して，クラスを読みだしてきたファイルの `URL` を取得する．
4. `URL` に対して `toExternalForm()` メソッドを呼び出して文字列表現に変換し，JDI を通じてデバッガ側がその文字列表現を得る．

この方式で得られた URL は計算機上の任意のディレクトリあるいは JAR ファイルの中から読みだされたクラスファイルを一意に特定しており，Java 標準ライブラリの `URLClassLoader` クラスを使うことでそのファイルの内容をデバッガ側に読み込む．

3.2 クラスファイルのバイトコード命令の読み取り

バイトコード命令の読み取りは，ASM[1] という Java バイトコードの操作や分析を行うためのライブラリを用いて実装した．手順は以下の通りである．

1. 解析対象のクラスファイルを入力として，ASM の `ClassNode` オブジェクトを取得する．このオブジェクトは，クラス内に宣言されたすべてのメソッド，メソッドごとの命令をツリー構造で保持する．

[1] `http://asm.ow2.org/`

2. 実行が一時停止したメソッドの名前と引数の型情報を `JDIStackFrame` オブジェクトから取得し，`ClassNode` が保持するメソッドの中から対応するメソッドを見つける．

3. 見つけたメソッドの命令リストを調べ，プログラムの停止箇所に行番号が一致する命令の開始位置をすべて列挙する．

以上の手順で，行番号の一致点を命令の実行開始点とする．行 l に対して，複数のバイトコード命令群が同一行に対応する場合は，それぞれを命令の実行開始点とする．例えば，"`for (int i = 0; i < 10; i++) {`"という行があった場合，コンパイルされたバイトコードは `i = 0` の初期化処理と，`for` 文の本体の実行終了後の `i++`の実行の 2 つの命令群に分割される．本研究では，このようなバイトコード命令については，`i = 0` と `i++`に対応する 2 つの命令群の両方をバイトコードでの出現順にそれぞれ解析し，登場した変数のリストを連結して最終的な $V(l)$ を作成する．

3.3 バイトコード命令から使用される変数やメソッドの特定

前節で特定した命令の実行開始点集合 $P = \{p_1, p_2, ..., p_n\}$ の各点 $p_i(i \leq i \leq n)$ から，順に命令を解釈し，そのメソッドの `return` 文に到達するか，あるいは `for` 文等によるループを 1 周する（既に訪問済みの命令に到達する）まで仮想的な実行を行う．

配列などの参照を正しく取り扱うには，四則演算等の計算も実行しなければならない．しかし，代入命令などの副作用を直接デバッグ対象のプログラムに実行させてしまうと，デバッグ対象プログラムの状態を破壊してしまう．そこで本研究では，デバッグ対象プログラムのメモリ領域のコピー C を用意する．このメモリのコピーは，実行を開始する時点では空にしておき，メモリ領域 v を参照する場合にそのコピー $C(v)$ が存在するかどうかを確認し，もし存在していなければ，JDI を通じてデバッグ対象プログラムにアクセスし，値のコピーを $C(v)$ とする．変数 v に対して値の書き込みを行う場合は，$C(v)$ の値のみを書き換える．このようにキャッシュに格納された値のみを書き換えることで，プログラムの実際の変数の値は変更しないようにして，デバッグ対象プログラムの状態を変えないようにしている．

具体的な計算手順としては，参照するメモリ領域のリスト V の初期状態を空とし，命令の各実行開始点 $p_i(1 \leq i \leq n)$ について以下の処理を実行する．

1. 変数のコピー C を空で初期化する．

2. 仮想プログラムカウンタ pc を p_i で初期化する．

3. 命令位置 pc にある命令を実行し，V と C の更新を行って pc を 1 命令分進める処理を繰り返す．条件分岐命令により次に実行される命令が複数存在する場合，それぞれの

表 1: ASM におけるバイトコード命令の分類

種類	処理
METHOD_INSN	メソッド呼び出し
INVOKE_DYNAMIC_INSN	メソッド呼び出し
FIELD_INSN	フィールド処理
INSN	命令処理
LOOKUPSWITCH_INSN	条件分岐
TABLESWITCH_INSN	条件分岐
MULTIANEWARRAY_INSN	配列処理
LDC_INSN	定数のロード
INT_INSN	int 型変数の処理
TYPE_INSN	型キャスト
JUMP_INSN	ジャンプ命令処理
IINC_INSN	int 型変数のインクリメント処理
VAR_INSN	ローカル変数の読み書き
FRAME	何もしない

分岐先を探索するために C を複製し，各分岐において命令実行により V を更新する．V そのものは複製せず，各分岐先の命令を深さ優先探索で実行した結果を順番に格納する．

変数リスト V は変数 v やメソッド m を参照する命令があれば，v や m を V に登録する．ただし，変数に関しては既に登録されている場合は重複して登録はしないが，メソッドに関しては既に登録されている場合は (シグネチャが一致していても) 重複して登録する．実行が停止した後，得られた V に，元の変数リストから V の要素を取り除いたリストを連結して，変数ビューに表示するリスト V になる．

各バイトコード命令の実行は，基本的にはバイトコードの仕様通りである．ASM におけるバイトコード命令の分類を表 1 に示す.

提案手法の動作を，具体例を用いて説明する．以下は，コマンドライン引数を入力として受け取り，引数があればその中の最後の要素を出力し，引数がなければメッセージを出力して終了するメソッドである．左に付与したのは実際のソースコードにおける行番号である．

```
5:   public static void main(String[] args) {
6:       int i = args.length;
```

```
 7:        if (i > 0) {
 8:            System.out.println(args[i-1]);
 9:        } else {
10:            System.out.println("No arguments");
11:        }
12:    }
```

たとえば引数として 3 つの文字列 “a”, “b”, “c” を格納した配列が引数として与えられた場合，このメソッドは 8 行目に進み，“c” を出力して実行を終了する．

このメソッドを Eclipse JDT でコンパイルした場合，以下のような Java バイトコードで表現される．このバイトコード表現は SOBA [12] によって出力したものを，ソースコードの行番号に対応するように整形したものである．行番号との対応は，コンパイルされたクラスファイルに格納された行番号表に基づいたもので，デバッガのステップ実行やブレークポイント機能の実現に用いられるものと同様である．また，(L00021) の表記は，実際のバイトコード命令ではなく，ジャンプ命令の飛び先を示すラベルを便宜上挿入したものである．

```
 6: ALOAD 0 (args)
    ARRAYLENGTH
    ISTORE 1 (i)
 7: ILOAD 1 (i)
    IFLE L00021
 8: GETSTATIC java/lang/System#out: java/io/PrintStream
    ALOAD 0 (args)
    ILOAD 1 (i)
    ICONST_1
    ISUB
    AALOAD
    INVOKEVIRTUAL java/io/PrintStream#println(Ljava/lang/String;)V
 9: GOTO L00027
10: (L00021)
    FRAME-OP(1)
    GETSTATIC java/lang/System#out: java/io/PrintStream
    LDC No arguments
    INVOKEVIRTUAL java/io/PrintStream#println(Ljava/lang/String;)V
12: (L00027)
```

```
    FRAME-OP(3)
    RETURN
```

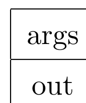
Java 仮想マシンはスタックマシンとして定義されており，たとえば 6 行目に対応する 3 つの命令

```
 6: ALOAD 0 (args)
    ARRAYLENGTH
    ISTORE 1 (i)
```

は，引数として渡された配列への参照をオペランド操作用のスタックにロードし，次にスタックに積まれた配列の参照を配列の長さに変換し，得られた結果をローカル変数 `i` に代入する．7 行目にブレークポイントを設置し，引数として 3 つの文字列 “a”, “b”, “c” を格納した配列を与えてこのメソッドを実行した場合，デバッガは変数 `i` には 3 が代入された状態で，7 行目に属する命令を実行する前に，プログラムの実行を一時停止する．そして，ローカル変数である配列 `args` と整数 `i` の状態をデバッグ対象プログラムの Java 仮想マシンからデバッグ用の API を介して取得し，表示を行う．

本研究で提案するデバッガは，この表示状態からプログラムの将来の実行の先読みを開始し，以下のような手順で計算を行っていく．

1. デバッグ対象のプログラムとは独立した空のオペランドスタックを準備する．
2. 7 行目の `ILOAD 1` を解釈し，オペランドスタックに実際の `i` の値である 3 を読み込む．
3. 続く `IFLE L00021` という命令は，オペランドスタックの一番上の値を取り出し，それが 0 以下であれば指定のラベルに分岐する命令である．ここでは値が 3 なので，条件分岐が発生せず，次の命令に進む．オペランドスタックは空に戻る．
4. 8 行目の最初にある `GETSTATIC` 命令は，Java のクラスフィールドの値をオペランドスタックに読み込む命令である．ここでは `java.lang.System` クラスの `out` フィールドがスタックに積まれることになる．続く `ALOAD 0` 命令は引数 `args` に格納された配列の参照をスタックに積む．この時点でのスタックの状態は以下のようになる．



5. `ILOAD 1` 命令は変数 `i` の値である 3 をスタックに積み，続く `ICONST_1` は定数 1 をスタックに積む．この時点でスタックの状態は以下のようになる．

1
3
args
out

6. ISUB 命令は，オペランドスタックの上 2 つの要素を取り出し，減算を実行する．ここでは値が 3, 1 と分かっているので，実際の計算を行って 2 をスタックに積みなおす．この時点で，スタックには out, args, 2 の 3 つの要素が積まれた状態となる．

2
args
out

7. 次に続く AALOAD 命令は，オペランドスタックから配列の参照と添え字を取り出し，配列要素の読み出しを行う．ここで args[2] の内容が参照されることが分かるので，引数として渡された “c” に対応する文字列オブジェクトをデバッガから取得し，スタックに積む．スタックの状態は以下のようになる．

“c”
out

8. 8 行目最後の INVOKEVIRTUAL 命令は，println メソッドの呼び出しである．メソッド呼び出し先で何が起きるかは計算の先読みでは関知せず，この呼び出しが引数としてスタックに積まれた out, “c” を使用することだけを検知する．

9. メソッド呼び出しが正常に終了することを仮定して，実行の先読みを継続する．9 行目の GOTO L00027 命令で，12 行目に移動する．

10. FRAME-OP はバイトコード検証用の情報なので無視して，RETURN 命令でメソッドの終端まで到達したことを検知し，実行の予測を終了する．

この一連の流れで，7 行目のブレークポイント以降，実行を継続した場合は条件分岐で 8 行目に進み，args[2] を参照して println メソッドを呼び出すという将来の実行経路を取得することができる．今回は自明な例であるが，条件分岐の論理演算が複雑な場合や，配列の要素参照が複雑な場合でも，実行経路や実際に使われる値を抽出することが可能である．

この先読み処理は，デバッグ対象プロセスから必要なメモリの内容をコピーすることになるが，実装の基盤とした Eclipse JDT の場合，スコープ内で有効なほぼすべての変数と，それらがオブジェクトや配列である場合はそれぞれフィールドや配列の要素を画面に表示するために取得しており，先読みにあたってメモリコピーのオーバーヘッドはほとんど生じない．

3.4 未知の値を含む実行の先読み

システムコールのように，実行しなければ値が分からないメソッドに到達した場合は，その結果を計算することが不可能である．しかし，たとえば複数の変数の値を計算する処理がメソッド中に書かれていた場合，一部に未知の値があっても，他の計算については実行の先読みが可能な場合がある．本研究では，このような場合に対応するために，計算不可能な値を表現する仮想的な値 u (Unknown) を導入する．

バイトコードの先読みにおいて，未知の値 u は，以下のように振る舞う．

- u を少なくとも 1 つのオペランドとして含むような四則演算，ビット演算，論理演算の結果は u である．
- オブジェクト参照が u であるようなフィールド参照や配列参照の結果得られる値は u である．
- 条件分岐に用いる値が u となった場合は，そこで命令の先読みを中止するのではなく，両方の可能性があるとして，2 つの経路の探索を行う．

未知の値 u は，記号実行に用いられるシンボル値に類似しているが，計算の過程を保存しない軽量な表現である．

未知の値を用いた計算の例を示す．以下のコード片は，既に代入された変数 x, y, z, a の値と 2 行目で外部から読み込む論理値を使って，変数 sum と prod を計算している．

```
1:   int sum = x + y + z;
2:   if (stream.readBoolean()) {
3:     sum += a;
4:   }
5:   int prod = x * y * z;
6:   System.out.println(sum);
7:   System.out.println(prod);
```

1 行目から実行予測を開始した場合，2 行目のメソッド呼び出し `readBoolean` の結果を取得することができない．そこで，メソッドの戻り値として仮の値 u を使用し，条件分岐がどちらにも進みうると仮定する．提案手法では，実行予測は，3 行目を実行した場合の `sum` の値と，3 行目を実行しなかった場合の `sum` の値の 2 通りを内部的に計算することになる．また，`prod` に関しても同様に，2 通りの経路での値を計算し，結果的に同一の値を取るという情報を得ることになる．

3.5 実行時エラーの発生の先読み

実行時エラーとして，ヌル参照を使用したメモリアクセス（NullPointerException），配列の範囲外アクセス (ArrayIndexOutOfBoundsException)，ゼロ除算（ArithmeticException）を取り扱う。これらは未知の値 u が登場しない計算において，オブジェクト参照が必ず null であることが確定する場合や，配列の大きさとインデクスが既知の場合など，確実にエラーが起きる場合に報告する．

たとえば以下のようなコード片が与えられたとき，もし変数 z, k の値が等しければ，実行経路予測の計算中に `x / 0` の計算が要求され，例外 ArithmeticException が発生する．

```
y = z - k;
System.out.println(x / y);
```

このような場合、そのまま実行経路予測を進められる分だけ進め，その結果を変数ビューに反映する．

3.6 提案手法の制限

本研究におけるバイトコード解析は，以下の制限を持っている．

- バイトコード命令リストの走査において，指定行の命令をすべて列挙している．デバッガ自体は次に実行されるバイトコード命令のインデックスを持っているが，バイトコード解析の基盤に用いたライブラリ ASM が命令ごとのバイトコードのインデックスを特定する機能を提供していなかったためである．
- メソッド呼び出しは例外を発生させないと仮定して，結果を未知の値 u と仮定し，その次の命令の実行を先読みする．メソッド呼び出し先がライブラリ等であり，間接的に呼び出し側のオブジェクトの状態を改変するような実行がなされると，実行経路の予測に誤りが生じる可能性はある．
- 変数の値が実行途中に変更されて使用される場合でも，変数ビューに表示される値は現在の状態に限られてしまう．これはデバッガが表示する変数の値をデバッグ対象のプログラムと結び付けており，たとえば「7 行目における変数 i の値」「8 行目における変数 i の値」というように，同一の変数の異なる時点を区分して変数ビューに表示することが困難であったためである．
- フィールドの値の上書きには対応していない．フィールドの値については異なるスレッドによる上書きの可能性もあり，書き込まれた値がそのまま次に読みだされる保証ができないためである．

- 演算は計算結果のみを保存しており，計算手順は示さない．したがって，配列の個別の読み出しにおいて，たとえば a[i+3] は a[4] というように，添え字は最終的な値のみを出す．
- 命令の実行順はバイトコード依存であるため，x ? y : z などの条件演算での変数の参照順序がソースコード上の変数の状態と一致するかどうかは，コンパイラ依存の可能性がある．

4　ツール実装

4.1　実装方法

提案手法を Eclipse JDT の拡張という形式で実現した．具体的には Eclipse 4.5.1 (Mars 1) のリリース版をベースに，`org.eclipse.jdt.debug.ui` プロジェクトにコードを追加している．このプロジェクトは Eclipse が持つデバッガのユーザインタフェースを実現しているものである．

提案手法の実現には, 内部 API に当たる `org.eclipse.jdt.internal.debug.ui.variables` パッケージに所属する `JavaSTackFrameContentProvider` クラスに対する拡張が必要であった．このクラスは変数ビューに表示する内容を決定する処理を担当しており，`getAllChildren` メソッド中にて実行される

```
Object[] children = super.getAllChildren(parent, context, monitor);
Object[] filtered = JavaContentProviderFilter.filterVariables(
                        children, context);
```

という処理の結果得られる変数ビューの項目一覧に，実行予測の結果得られた変数情報を追記する形で実現した.

提案手法を構成するコンポーネントは以下の通りである．

- Java バイトコードを解釈実行する実行予測エンジン．
- デバッグ対象プログラムからバイトコード情報を取得したり，JDT の変数ビューに必要な項目を追加するなど，Eclipse JDT デバッガとのインタフェース．

いずれも Eclipse 同様に Java を用いて実装した．これらのコンポーネントのソースコードは合計で 3200 行程度である．

4.2　実行例

図 1 は，フィボナッチ数列の各項を計算しながら標準出力するプログラムを実行したときのスクリーンショットである．配列 fibo を用意し，計算したフィボナッチ数列の各項を fibo に格納している．現在，41 行目の"fibo[1] = 1"を実行する直前でプログラムが停止している．

図 1 で示している状態からプログラムの実行を再開した場合，`for` 文のループが一回終わるまでプログラムを実行すると，fibo[1] への数値の代入，fibo[i] の計算，fibo[i] の出力がそれぞれ一回ずつ実行される．これらの命令を実行するにあたり，fibo[i] の計算において fibo[i-1] と fibo[i-2]，fibo[i] の出力において fibo[i] が参照される．また，fibo の各要素にアク

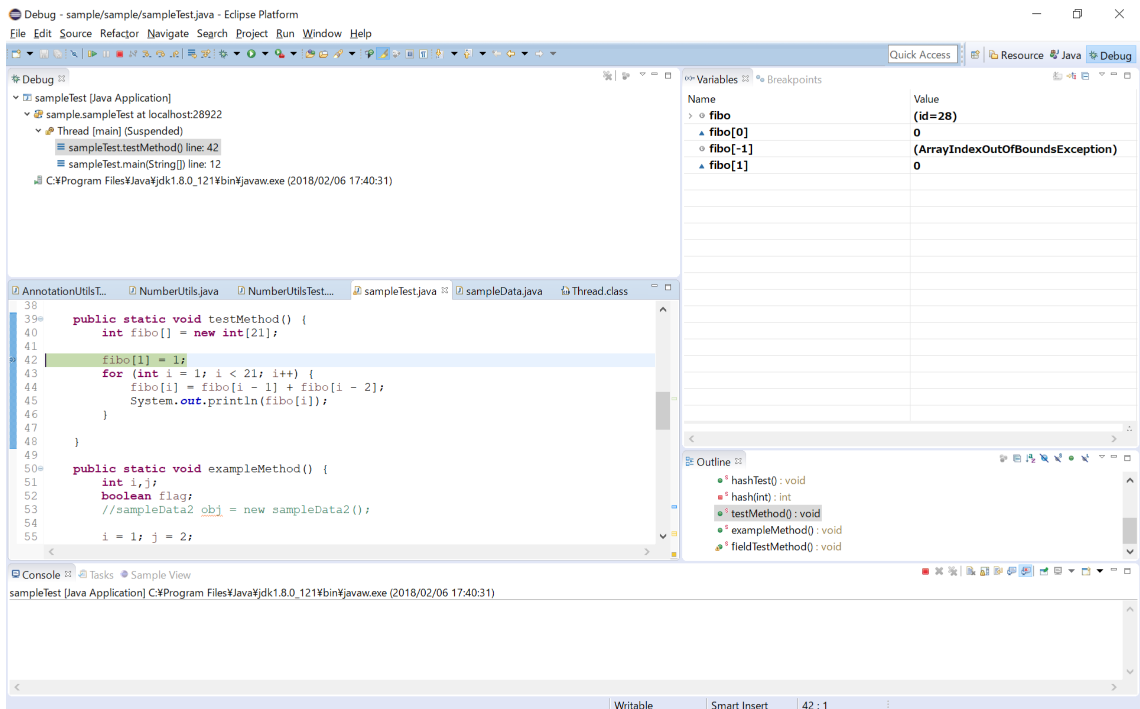


図 1: スクリーンショット．41 行目まで実行した状態で止まっている．

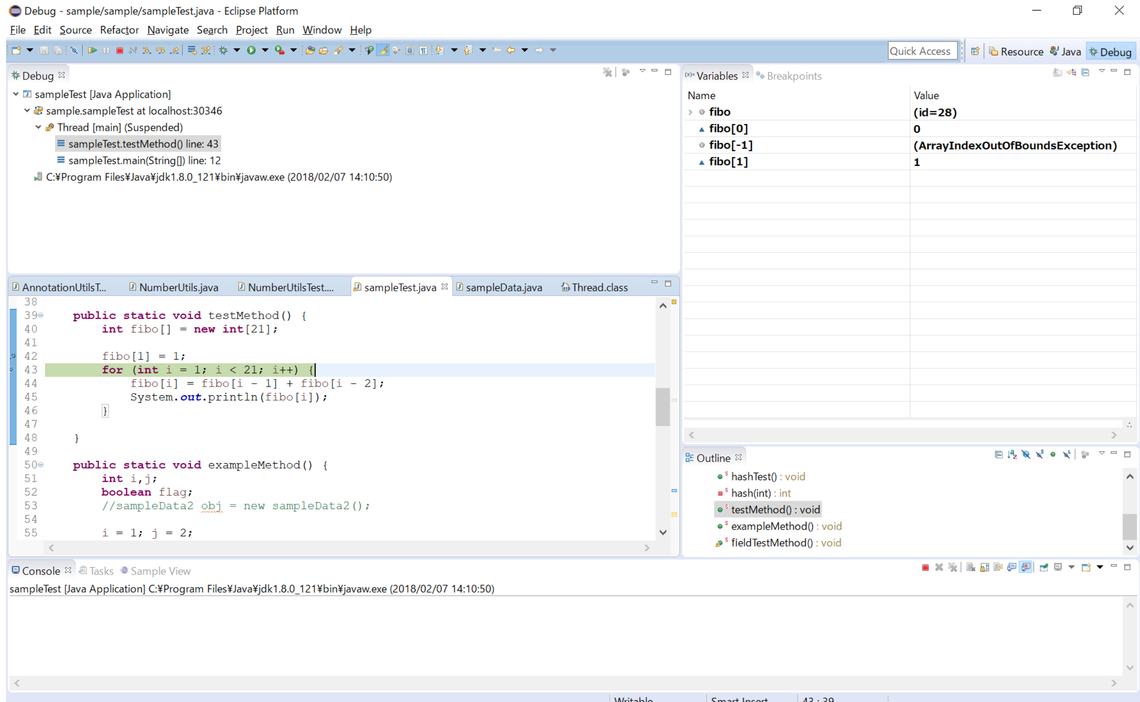


図 2: 42 行目まで実行を進めた状態

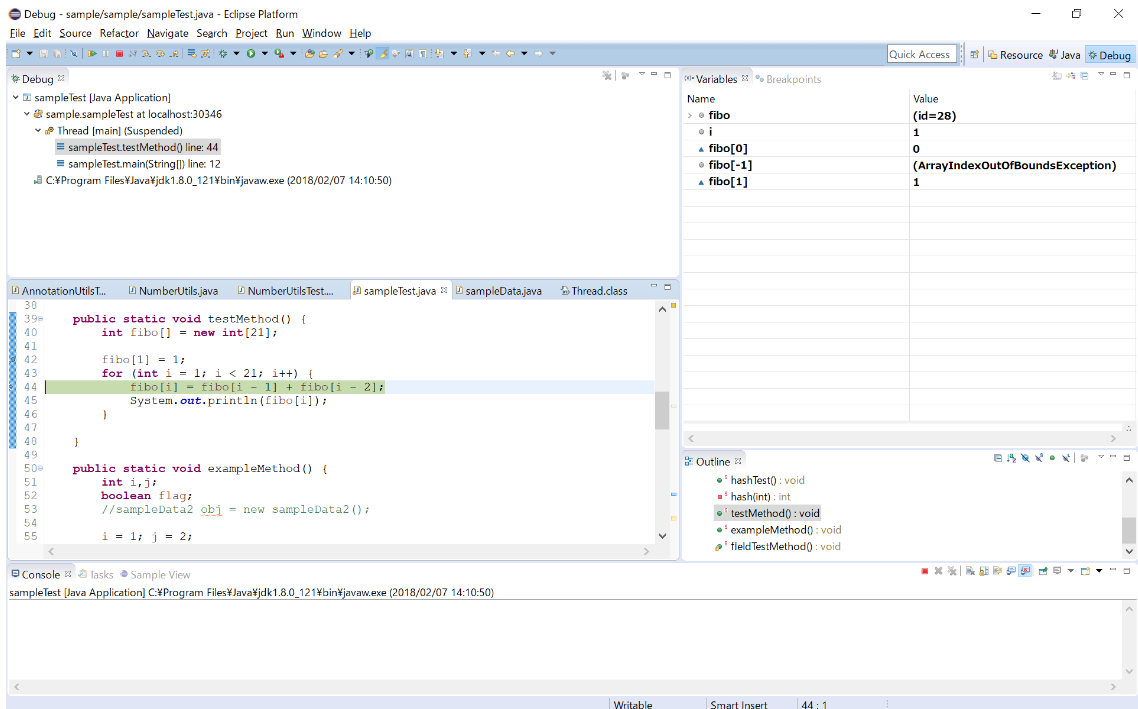


図 3: 43 行目まで実行を進めた状態

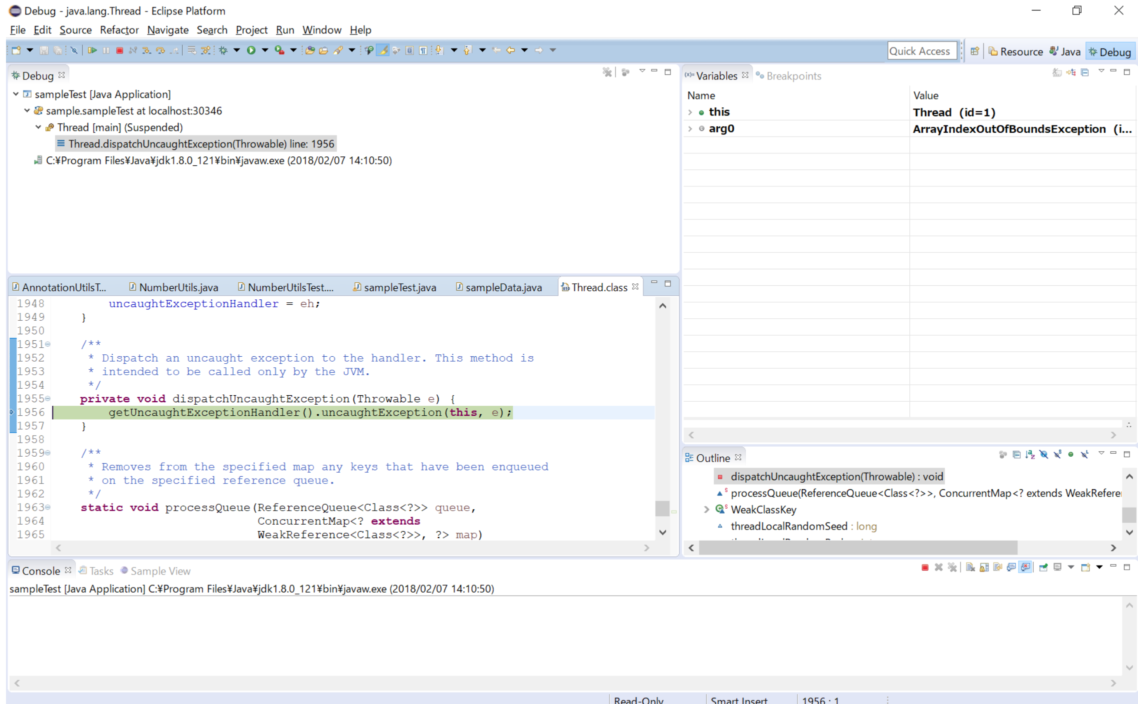


図 4: 例外を検知して例外処理のメソッドに移動した状態

セスする際に fibo にもアクセスするため，fibo および fibo の要素のうち参照されるものが変数ビューに表示される．ただし，fibo[i] の計算において，一回目の `for` 文のループであることから変数 i の値は 1 であるため，fibo[i] の計算式は”fibo[1] = fibo[0] + fibo[-1]” となるが，fibo[-1] は存在しない配列要素であるため不正なメモリ参照であり，Java プログラムでは ArrayIndexOutOfBoundsException という例外が発生する．本研究のツールでは，こうした例外に対しては変数の値やメソッドの返り値の代わりに例外名を示すため，fibo[-1] に対しては ArrayIndexOuOfBoundsException と表示されている．

図 2，図 3，図 4 は図 1 の状態からステップ実行を行い，プログラムの実行を進めた場合のスクリーンショットである．図 1 の状態からステップ実行により 42 行目まで実行を進めると，図 2 の状態となる．42 行目が実行されたことにより fibo[1] の値が更新され，同時にビューに示されている fibo[1] の値も更新されている．さらに 43 行目まで実行を進めると図 3 の状態になる．さらにステップ実行を進めると図 4 の状態になる．これは命令を実行した結果，例外が発生したことを表している．具体的には，先述の通り，fibo[-1] という存在しない配列にアクセスしたことにより ArrayIndexOutOfBoundsException という例外が発生している．

なお，本研究のツールの実行時間は，例えば上記の図 1 の状態では 1ms 以下であった．

4.3 有効性の考察

提案手法は，ブレークポイントで停止した際に，それ以降のプログラムの実行情報を付加するという形で効果を発揮する．そのため，たとえば「`if` 文で特定の経路に進む場合」にその `if` 文の直前の振舞いをステップ実行で分析したい場合など，特定の実行経路を用いることが有効に働くと考えられる．また， `switch` 文における `break` の欠落のような予期せぬ実行経路の発生が，本来使用しないはずの変数が変数ビューに表示されるという形で可視化されるといった効果もある．

本研究の実装では，無限ループへの突入を回避するため，実行経路が同一命令に複数回到達したときには自動停止するものとした．しかし，`for` 文における条件節やインクリメントの設定ミスなどのバグは，たとえば 2 回程度ループを巡回した結果を示すことで，症状を可視化することも可能であると考えられる．

なお，提案手法は，システムの現在のメモリ状態から実行を予測するため，外部システムの状態に起因する問題のデバッグには有効に働かない．このようなバグに対しては，実行トレースを記録し，メモリ状態の履歴を分析する Omniscient Debugging のような手法が適している．

5　まとめ

本研究では，開発者が閲覧したい変数の値やメソッドの返り値などのデータを把握するのに費やす時間と労力を削減するために，プログラムの実行を先読みし，まだ実行されていない命令で使用される変数の値や実行時エラーの発生可能性を提示するデバッガを提案した．実行の先読みはインタプリタ実行の一種であるが，計算不可能な値を仮想的な未知の値として表現し，一部計算結果が未知の状態でも実行を可能な限り継続する形式で実現した．今後の課題としては，バイトコード解析機能の拡大，評価実験による性能や有用性の評価などが挙げられる．

謝辞

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 教授には研究について様々なご助言を賜りました．深く感謝申し上げます．

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 准教授には研究について様々なご意見を賜りました．深く感謝申し上げます．

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 神田哲也 特任助教には研究について様々なご意見を賜りました．また，論文執筆においてもご援助とご指摘を賜りました．深く感謝申し上げます．

奈良先端科学技術大学院大学情報科学研究科ソフトウェア工学研究室 石尾隆 准教授には研究について様々なご指導を賜り，多大にご支援いただきました．また，論文執筆においてもご援助とご指摘を賜りました．深く感謝申し上げます．

最後に，井上研究室の皆さまには様々な場面でご指導，ご意見をいただき，研究生活を支援していただきました．深く感謝申し上げます．

参考文献

[1] David Abramson, Clement Chu, Donny Kurniawan, and Aaron Searle. Relative debugging in an integrated development environment. *Software Practice and Experience*, Vol. 39, No. 14, pp. 1157–1183, July 2009.

[2] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pp. 405–422, New York, NY, USA, 2007. ACM.

[3] Cambridge News. Experts battle £192bn loss to computer bugs. `http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html`.

[4] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, Vol. 69, No. 1-3, pp. 35–45, December 2007.

[5] ITMedia ニュース．260 万人の朝の足を直撃　プログラムに潜んだ“ 魔物 ”．`http://www.itmedia.co.jp/news/articles/0710/12/news117.html`, 2007.

[6] Andrew J. Ko and Brad A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 151–158, 2004.

[7] Bil Lewis. Debugging backwards in time. In *Proceedings of International Workshop on Automated Debugging*, 2003.

[8] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, Vol. 25, No. 1, pp. 83–110, Mar 2017.

[9] Eclipse Project. The eclipse foundation open source community website. `https://eclipse.org/`.

[10] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 485–495, 2012.

[11] Jonathan B. Rosenberg. デバッガの理論と実装. アスキー出版局, 1998.

[12] Soba. `https://kir.ics.es.osaka-u.ac.jp/pub/kir/proj/SOBA`.

[13] D. Spinellis. 66 specific ways to debug software and systems. Addison-Wesley Professional(2016).

[14] Rebecca Tiarks Tobias Roehm. How do professional developers comprehend software? In *International Conference on Software Engineering*, pp. 255–265, 2012.

[15] Andreas Zeller. デバッグの理論と実践–なぜプログラムはうまく動かないのか. O'Reilly Japan, 第 2nd 版, 2012.

[16] 嶋利一真, 石尾隆, 井上克郎. ソフトウェアの実行を分析するための低侵襲なモニタリングツールの試作. ソフトウェアエンジニアリングシンポジウム 2017 論文集, pp. 224–227, aug 2017.

[17] 松村俊徳, 石尾隆, 井上克郎. 動的スライスを用いたバグ修正前後の実行系列の差分検出手法の提案. 情報処理学会研究報告, No. 8, pp. 1–8, mar 2016.

[18] 松村俊徳, 石尾隆, 鹿島悠, 井上克郎. REMViewer: 複数回実行された Java メソッドの実行経路可視化ツール. コンピュータ ソフトウェア, Vol. 32, No. 3, pp. 137–148, 2015.

[19] 松田直人. 回帰テストにおける実行系列の差分の効率的な検出手法. 大阪大学基礎工学部情報科学科特別研究報告, 2017.