Vol.2016-SE-192 No.2
Vol.2016-EMB-41 No.2
2016/6/2

ステップ実行時に注目すべき変数を提示するデバッガの開発

冨永 真司[1,a)]　石尾 隆[1,b)]　井上 克郎[1,c)]

概要：デバッグとは，開発者が期待した通りに動作しないプログラムに対してその原因を分析し解消するように修正を行う作業である．本研究では，開発者がデバッガを使用して分析に必要な変数を閲覧する作業に費やす時間と労力を削減するために，次の命令で使用される変数を注目すべき変数として提示する変数ビューを提案する．ここでの次の命令とはステップ実行を 1 回実行したときに実行される命令群を指す．また，使用される変数とは値をメモリから読みだされる変数のことである．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，提案する変数ビューの機能を実現した．試作した変数ビューが変数の並びに与える影響を評価するために，実際のバグ事例を集めたベンチマーク Defects4J に収録された 4 つの事例と，授業で使用されている教材プログラム Lifegame の動作の分析に対して適用し，変数が多数使われるメソッド中であっても，次の命令で使用される少数の変数をビューの先頭に集められることを確認した．

1. はじめに

ソフトウェアの開発およびソフトウェア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．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 [1]．そのため，バグが発見された場合，それを取り除く作業は避けられないことが多い．

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 [2]．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 [3]．

デバッガには様々な形態があるが，Eclipse[4] 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．変数に不正な値や想定外の値が入っているとそれがバグの原因となることが多いので，開発者は，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，変数に期待通りの値が入っているかどうかを確認する．

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 [3]．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．

開発者が閲覧する変数ビューの項目の数は，その命令の地点で参照可能な変数の数と，その中に含まれるオブジェクトや配列の数，データ構造の複雑さによって決まる．プログラムの実行中の情報を使った閲覧のための支援というのは行われていないため，開発者は変数ビューから興味ある変数を自分で探す必要がある．たとえば `array[idx]` という配列の参照をしている命令があったとき，Eclipse JDT は単純に `array` という配列の全要素のリストと変数 `idx` の値を提供し，開発者がその中から手作業で `array[idx]` に対応する値の要素を閲覧する必要がある．また，オブジェクトのフィールドについても，すべてのフィールドが

[1] 大阪大学
Osaka Uniersity, Suita, Osaka 565-0871, Japan
[a)] t-shinji@ist.osaka-u.ac.jp
[b)] ishio@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

アルファベット順で表示され，それらの中から注目する値を開発者が調べる必要がある．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．

本研究では，開発者が閲覧する必要のある変数を探し出す作業に費やす時間と労力を削減するために，次の命令で使用される変数を注目すべき変数として提示する変数ビューを提案する．ここでの次の命令とはステップ実行を 1 回実行したときに実行される命令群を指す．また，使用される変数とは値をメモリから読みだされる変数を指す．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うちの一部であることが多いことから，プログラムの規模が大きく, それに伴い参照可能な変数も多くなったとしても，開発者は変数ビューの上端を確認するだけで，使用される変数の値を迅速に確認することができるようになる．

具体的には，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ことで実現した．この変数ビューは，一時停止したプログラムから，次に実行される命令のリストと変数の値を取得し，デバッガ側で変数の値を使って命令を疑似的に実行することで，デバッグ対象プログラム自体の状態には影響を与えず，参照されるメモリ領域の情報だけを抽出する．解析可能な範囲で実際に使われる変数，オブジェクトおよび配列の一覧を特定し，それらの名前と値をビューの先頭に移動して表示する．

試作した変数ビューが変数の並びに与える影響を評価するために，実際のバグ事例を集めたベンチマーク Defects4J に収録された 4 つの事例と，授業で使用されている教材プログラム Lifegame の動作の分析に対して適用し，次の命令で使用される変数がどの程度変数ビューの先頭に集まり，開発者の閲覧作業を短縮することが期待できるかを定量的に評価する実験を行った．

以降，2 章では本研究の関連研究について述べる．3 章では提案手法について述べる．4 章では提案手法に対する評価実験について述べる．5 章ではまとめについて述べる．

2. 関連研究

本研究は，開発者が注目するであろう変数をウィンドウ上で優先的に表示する手法であるが，変数の値の閲覧に関する研究としては，変数の値を収集し可視化する方法が主な研究となっている．Alsallakh ら [5] は，Tracepoint と呼ばれる，実行時の変数の値を観測するためのブレークポイントの一種を提案した．プログラムの実行が Tracepoint に到達した時点での変数の値を記録していき，時間の経過によって変数の値が変化することを可視化する手法である．Omniscient Debugging [6] は，プログラムの実行をすべて記録しておき，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とするデバッグ手法である．Whyline[7] は，Omniscient Debugging と同様に実行履歴をすべて保存しておき，その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．質問内容は実行履歴の中での制御フローやデータフローに対するクエリであり，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，といった質問を選択すると，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．この方式は，プログラムの振る舞いに仮説を立て，それをツール等により確認するという体系的なデバッグ作業 [2] の一種であると考えられる．

データの操作に注目してプログラムの実行を制御するデバッガについても研究が行われている．Object-Centric Debugging[8] 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ことを可能としている．Relative Debugging[9] は，2 つの類似プログラムに同じ入力を与えて実行を比較し，異なる動作が観測された時点でプログラムの実行を停止することでバグの症状の発生場所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．

実行中に観測された変数の値を使った異常データの発生の検出もまた，デバッグ支援技術の 1 つである．たとえば Daikon[10] はプログラムの実行を分析し，その値域から不変条件を推定するツールである．あらかじめ試験データから不変条件を推定しておくことで，これまでの入力になかった異常なデータの発生を検出することが可能である．

3. 提案手法

本研究では，デバッガにより一時停止したプログラムにステップ実行を指示したとき，次の一時停止までに使用される変数を変数ビューの先頭にまとめる機能を搭載したデバッガを提案する．変数ビューが行う処理は，デバッグ対象プログラムのある行 l でプログラムの実行を停止したとき，その位置でプログラムが参照可能なメモリ領域の中から表示すべき領域を選定し，開発者から閲覧できるメモリ領域のリスト $V(l)$ を計算する処理と考えられる．

Java プログラムがあるメソッド m の中のある行 l で停止したとき，あるメソッドが定義されたクラスを c とすると，以下のメモリ領域が参照可能である．

- m がインスタンスメソッドである場合，メソッドを実行するオブジェクトへの参照 `this`
- m の引数

- 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

Java では定数は `final` 修飾子をつけた変数で表現されるため，定数も参照可能な変数に含まれる．また，上記の参照可能なメモリ領域 v を用いて，以下に列挙する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

オブジェクト参照及び配列参照は，他のオブジェクトのフィールドや配列の要素になりうるため，参照可能な変数や配列の要素は参照可能な変数を基点とした階層構造として表現できる．

本研究で試作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 で停止したとき，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．`static` フィールドや `final` を付与したフィールド変数は，参照可能で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ではプログラムがメソッド m のある行 l で停止したとき，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．

- l と同一行にある命令を実行したときに（デバッガにおける Step Over 操作を一度だけ適用したときに）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` を付与されたフィールド変数は除く．

上記の項目は従来の変数ビューにおいてツリー構造の中にのみあったが，本研究で提案する変数ビューではツリー構造内の変数を残したまま個別にツリー構造外に追加する．例えば，`obj.var` というフィールド変数や `num[2]` 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しか存在しなかったが，本研究で提案する変数ビューでは，`obj.var` や `num[2]` という独立した項目をツリー外に追加する．

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変数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ときに値を参照される変数や配列の要素：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を実装している．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とき，その命令行 l に記述された命令を知るために，まず停止位置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だけからではクラスを判別することができない．そのため，本研究では，Java Debugger 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) 現在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) `Class` オブジェクトが持つ `ProtectionDomain` オブジェクトを取得し，そのクラスの読み出し元を表現する `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，クラスを読み出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保持するツリー構造である．
(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．たとえば，“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の使用される変数の特定

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 $p_i(1 \leq i \leq n)$ から，別の行番号の命令に到達するまで命令を順に実行し，参照される変数のリストを作成する．ただし，1 行で完結するような for 文等によって実行がループする場合は，そのループを一周だけ実行するものとする．

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，デバッグ対象の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としておき，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，変数の値を書き換える場合，このキャッシュに格納されたコピーの値のみを書き換え，プログラムの実際の変数の値は変更しない．

具体的な計算手順としては，表示する変数列 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 命令ぶん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，それぞれの分岐先を探索するために C を複製し，各分岐先の命令実行をそれぞれ行って 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 を参照する命令があれば，その変数を V に登録するが，既に登録されている場合は重複して登録はしない．また，無限ループの評価に陥ることを防止するため，プログラムカウンタ pc として使用した値は記録しておき，同じ命令の実行は高々 1 回しか行わないものとした．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変数のリスト V を得る．

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．

バイトコード命令のうち，IINC_INSN，VAR_INSN，

表 1 ASM におけるバイトコード命令の分類
Table 1 Categories of bytecode instructions in ASM

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	フィールド操作
INSN	命令処理
LOOKUPSWITCH_INSN	条件分岐
TABLESWITCH_INSN	条件分岐
MULTIANEWARRAY_INSN	配列操作
LDC_INSN	定数のロード
INT_INSN	int 型変数の操作
TYPE_INSN	型キャスト
JUMP_INSN	条件分岐，ジャンプ命令
IINC_INSN	変数のインクリメント
VAR_INSN	ローカル変数の読み書き
FRAME，LABEL，LINE	何もしない

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

FIELD_INSN，INSN の場合が変数の読み書きを行う．IINC_INSN，VAR_INSN の場合はローカル変数の読み出しが，FIELD_INSN の場合はフィールド変数の読み出しが，INSN の場合は配列の要素の読み出しが行われる可能性がある．これらの処理については，$C(v)$ および V の更新を行う．

メソッド呼び出しについては，実行することによって何らかの副作用が生じる可能性があるため，本研究ではメソッド呼び出しの実行そのものは行わないものとした．メソッドを実行しないとメソッドの返り値も計算できないため，便宜上その値を表現する数値 `unknown` を導入し，`unknown` を含む演算等の結果をすべて `unknown` とすることで，値の判明する範囲でのみ命令の実行を分析する．たとえば，`number` という配列があり，`number[1]` の値が読みだされるとすると，V には `numer` という配列変数の参照と `numer[1]` という配列の要素の参照が追加されるが，値として 1 を返すようなメソッド `one()` を用いて `number[one()]` といった形で表記されている場合は変数 `number` を使用することは検出できるが，`number[1]` という項目は追加されない．変数の値が確定する範囲，たとえば `i=0` の場合に `number[i+1]` という命令が実行される場合は，変数 `number`，`i` の参照に続けて `number[1]` が V に追加される．

3.4 実装上の制限

本研究におけるバイトコードの解析は，以下の制限を持っている．

- `for` 文の初期化子と条件節などで 1 行のソースコードが複数のバイトコード命令のブロックに分割して表現されている場合は，それぞれのブロックを個別に解析し，それらの命令をすべて列挙し，使用される可能性のある変数として列挙する．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クスを持っているが，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．
- `null` が代入されたオブジェクトの参照や `a[-1]` のような存在しない配列要素への参照など，不正なメモリ参照が行われる場合は，単に変数ビューには反映しないだけとし，これからエラーが起きる可能性の報告は行わない．
- 条件分岐は，分岐先が確定する場合もすべての分岐先を探索する．既に変数の値が確定していれば論理演算の結果も特定できるため，条件分岐先を可能な限り確定するような分析も可能である．
- 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出される保証ができないためである．

図 1　フィボナッチ数列を計算するサンプルプログラム
Fig. 1　An example program computing Fibonacci sequence

```
 1:  package test;
 2:  public class Main {
 3:    public static void main(String[] args) {
 4:
 5:      int[] fibo = new int[11];
 6:      fibo[1] = 1;
 7:      for (int i=2; i<=10; ++i) {
 8:        fibo[i] = fibo[i-1] + fibo[i-2];
 9:        System.out.println(fibo[i]);
10:      }
11:    }
12:  }
```

- 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，配列の個別の要素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．
- メソッド呼び出しは，ただ不定値を返す扱いとしている．メソッド呼び出しによってフィールドの値などが更新される可能性は考慮していない．
- 変数の参照順序をバイトコード上での命令の出現順に依存して決定しているため，たとえば `x ? y : z` などの条件分岐を伴う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，コンパイラに依存している可能性がある．

3.5 実行例

実行例として，フィボナッチ数列を計算するプログラムを**図 1** に示す．7 行目の `for` 文で繰り返し処理を実行しており，8 行目で配列に順番に値を代入している．このサンプルプログラムを実行し，プログラムの 8 行目で実行を一時停止した状況のスクリーンショットを**図 2** に示す．変数 `i` の値は現在 6 となっており，8 行目の計算で使用される配列 `fibo`，変数 `i`，配列の要素 `fibo[i-1]` に対応する `fibo[5]`，`fibo[i-2]` に対応する `fibo[4]` の値が変数ビューの上から順番に表示されている．そして，この行では使用されない変数 `args` が一番下に表示されている．変数 `fibo[5]` のように使用される要素を直接変数ビューに表示することで，開発者が `fibo` の配列参照からツリーを展開して `i` の値に対応する位置を探すという作業を不要としている．

4. 評価実験

本研究で試作したデバッガにより，命令ごとに使用される変数がどの程度リストの先頭に固まるかを定量的に評価する為の実験を行った．評価指標として各変数の「変数ビュー内の位置」の数値を使用して，Eclipse 4.5.1 標準での変数 v の表示位置と，提案する変数ビューでの変数 v の表示位置を比較する．この値が小さいほど，変数 v が変数

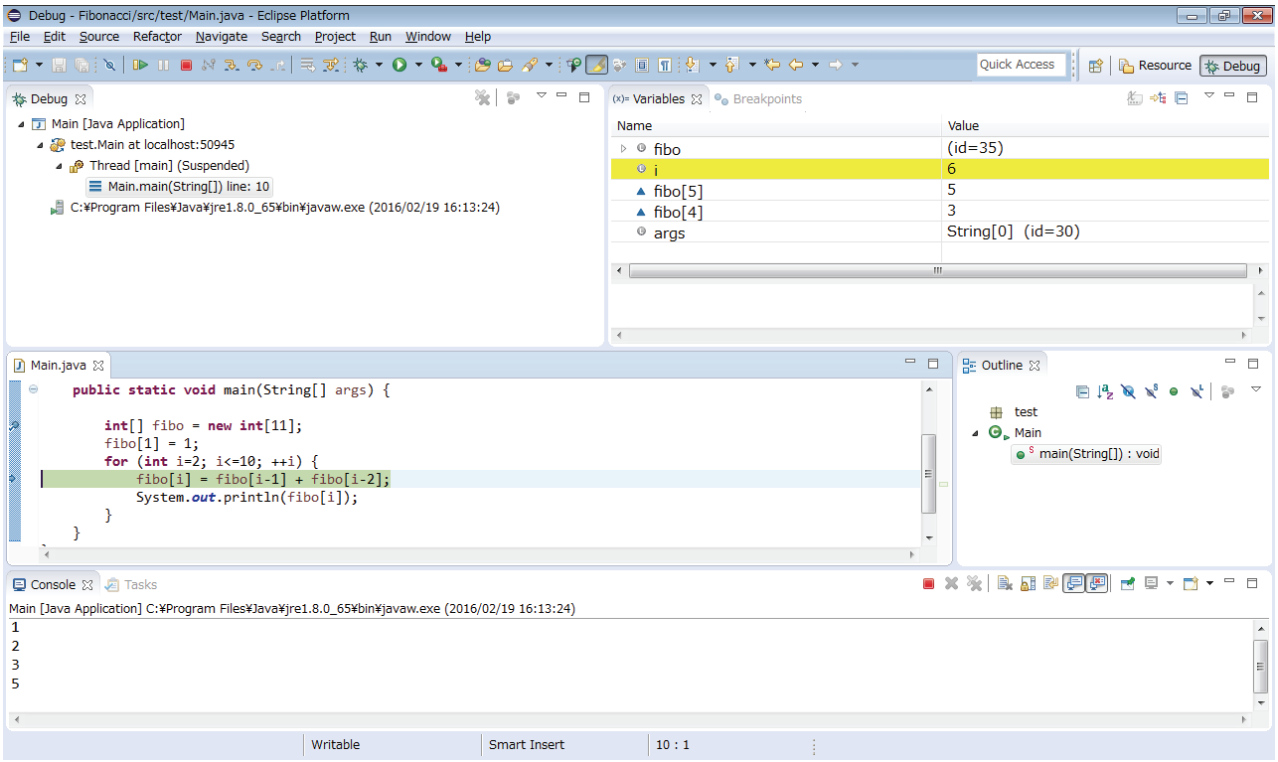

図 2 サンプルプログラムに対する実行例のスクリーンショット．一時停止している行の `fibo[i-1]` という式で使用される変数 `fibo`，`i` が変数ビューの上端に表示されている．

Fig. 2 A screenshot of an execution of the example program. The variables view shows variables `fibo` and `i` on the top because they are used by the expression `fibo[i-1]` at the line where an execution is suspended.

ビューの上側に表示され，開発者が変数ビューから目的の変数を探し出す労力が小さいと考える．

評価指標である変数リスト V 中の変数 v の位置を表す数値を $P(v,V)$ とする．$P(v,V)$ の値は変数ビュー内の先頭から数えて何番目であるかで表す．つまり，先頭は 1 である．たとえば，変数 a が変数リスト $V(l)$ において先頭から 3 番目にあるとき，$P(a,V(l))=3$ である．オブジェクトのフィールドや配列の個別の要素はツリーを展開した中にあるので，$P(v,V)$ は次のように定義する．

(1) フィールド：そのフィールドを保持するオブジェクト o の位置 $P(o,V)$ に，フィールド内での変数の位置を加算したもの．
(2) 配列の要素：その要素を格納する配列 a の位置位置 $P(a,V)$ に，配列内部での順序（添え字+1 の値）を加算したもの．

これらの値は，該当するフィールドあるいは配列の要素を探して変数ビューの必要な要素のみツリーを展開したときに得られる要素の位置である．たとえば，オブジェクト `A` が変数リスト $V(l)$ の先頭から 3 番目にあり，そのフィールド `var` が `A` を展開したツリーの先頭から 5 番目にある場合，$P(\texttt{A.var},V(l))=3+5=8$ である．3 個の要素を持つ配列 a が変数リスト V_l の 4 番目にある場合，$P(\texttt{a[2]},V_l)=4+3=7$ である．オブジェクトや配列が入れ子になっている場合も同様にして求める．

評価指標の計測対象は，プログラム中である注目するメソッドを 1 つ選択し，その 1 回の実行を Step Into 機能で開始から終了まで計測するものとした．Step Into 機能は，メソッド呼び出しがある場合，その呼び出し先まで追跡するステップ実行である．ステップ実行が停止した各行 l において，Eclipse JDT が持つ従来の変数ビューで得られる変数リスト $V_E(l)$ と本研究が提案する変数ビューで得られる変数リスト $V(l)$ を取得し，l の実行中に参照される各変数 v について $P(v,V_E(l))$,$P(v,V(l))$ を求めた．Step Into 機能はライブラリのメソッド等もステップ実行してしまうため，Eclipse 標準の Step Filtering 機能を利用して”`java.*`”などの標準パッケージ，テスト実行用の JUnit パッケージは観測対象から除外した．また，ステップ実行を行う作業については，`DebugSetEventListener` インターフェースを用いて自動的にステップ実行を行う処理を記述した．

実験対象とするプログラムは Defects4J から抜粋したバグ事例および Lifegame のグライダーパターンとする．Defects4J はバグが混入したコードのベンチマークである．バグが発見された 4 つのケース，1b，2b，3b，4b を Defects4J から抜粋した．これらは Apache Commons Lang プロジェクトのバグが未修整の状態のプログラムであり，

表 2　実験対象メソッド
Table 2　Subject methods of the experiment

プログラム ID	クラスおよびメソッド
Lang 1b	org.apache.commons.lang3.math.NumberUtilsTest.testLang747
Lang 2b	org.apache.commons.lang3.LocaleUtilsTest.testParseAllLocales
Lang 3b	org.apache.commons.lang3.math.NumberUtilsTest.testStringCreateNumberEnsureNoPrecisionLoss
Lang 4b	org.apache.commons.lang3.text.translate.LookupTranslatorTest.testLang882
Lifegame	lifegame.BoardModel.next

それぞれ 1 つだけテスト実行が失敗するメソッドが用意されており，それらのメソッドの開始から終了までを観測対象とした．Lifegame は生命の誕生，淘汰などのプロセスを簡易的モデルで表したシミュレーションゲームであり，大阪大学基礎工学部情報科学科 2 年生の Java プログラミング演習教材として教員が作成した実装において，盤面の状態をゲームのルールに従って更新する処理を実装しているメソッドの開始から終了までを観測対象とした．これらプログラムのクラス名，メソッド名を**表 2** に示す．

Defects4J を対象に選んだ理由は実際にバグのある状況で本研究で提案する変数ビューがどれほどの効果があるかを確かめるためである．また，Lifegame を対象に選んだ理由は，セルの表現に二次元配列を用いるため，参照可能な値が多いと考えられるためである．本研究で提案する変数ビューは多数の変数の中から一部の変数を選び出すため，参照可能な変数が多いほど，$P(v,V)$ の削減が期待できる．よって Lifegame においても，削減効果が期待できる．

Defects4J の各プログラムの実行結果から観測された $P(v,V_E(l))$ の分布を pre，$P(v,V(l))$ の分布を post として表記した箱ひげ図を**図 3** に示す．例えば，1b-pre は 1b に手法を適用していない場合の変数の表示位置の分布，3b-post は 3b に手法を適用した場合の変数の表示位置の分布である．同様に，Lifegame のプログラムの実行結果から観測された値の分布を**図 4** に示す．これらの図から，1b，2b の場合を除いて提案手法を適用した方が中央値が小さくなっている．また，いずれの場合も提案手法を適用した方が第三四分点や最大値が小さくなり，全体的に分布が 1 に近づいている．

1b，2b は各行で参照可能な変数が少なく，そのため全体的に $P(v,V)$ の削減量が小さく，中央値が変わらなかったと考えられる．一方，3b においては第三四分点や最大値が大きく減少しており，Lifegame においては外れ値の個数が大きく減少している．3b や Lifegame は各行で参照可能な変数が多いパターンで，全体的に $P(v,V)$ の削減量が大きかったと考えられる．この図から，提案手法を適用することで，参照可能な変数が多いメソッドでは変数ビューの変数の位置を表す数値 $P(v,V)$ が大きく削減される一方，参照可能な変数が少ないメソッドではあまり $P(v,V)$ は削減されないことがわかる．

以上の結果は，使用している変数が多いメソッドでのデバッグにおいて，提案手法による変数の並べ替えが効果を発揮することを示している．使用される変数の数はメソッドごとにも大きく異なるため，デバッグにおける有効性の評価は，今後，さらに適用バグ事例数を拡大しての調査が必要である．また，対象とするメソッドの開始から終了までの全命令をステップ実行したが，実際のプログラム開発ではバグの疑いがある一部分だけをステップ実行することも多く，今回の実験結果が実際のプログラム開発における効果を正確に反映しているとは限らない．この点は，今後の被験者実験などを通じて，実際のデバッグ作業への影響を確認する必要がある．

5. まとめ

本研究では，開発者が閲覧する必要のある変数を探し出す作業に費やす時間と労力を削減するために，次の命令で使用される変数を注目すべき変数として提示する変数ビューを試作した．実際のバグ事例を集めたベンチマーク Defects4J に収録された 4 つの事例と，授業で使用されている教材プログラム Lifegame の動作の分析に対して適用した結果，変数が多数使われるメソッド中であっても，次の命令で使用される少数の変数をビューの先頭に集められることを確認した．変数の位置の平均値が，Defects4J では 2.607 から 1.258 に削減され，Lifegame では 3.915 から 2.718 に削減された．通常，変数ビューの表示面積は一定であると考えられるため，参照可能な変数が多いメソッドで変数の表示範囲が小さく抑えられる効果は大きいと考えられる．

今後の課題としては，まず，副作用を持たないメソッド呼び出し先の実行内容の反映などのバイトコード解析機能の拡大が挙げられる．また，本研究では利用者の負荷軽減を変数の値の表示位置によって評価しているが，変数ビューの実際の利用に与える影響を被験者実験によって評価することも今後の課題である．

謝辞　本研究は JSPS 科研費 26280021 の助成を受けたものです．

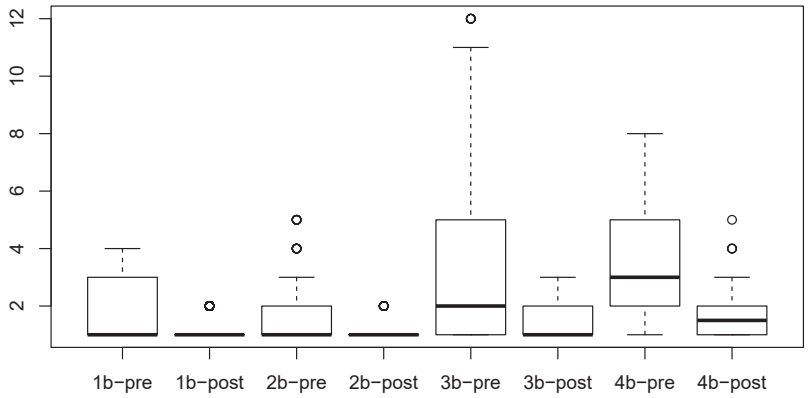

図 3 Defects4J の実行で観測された変数の表示位置の分布．“Pre” および “Post” はそれぞれ提案手法適用前，後の値を意味する．

Fig. 3 Distributions of variable positions observed in executions of Defects4J. “Pre” and “post” indicate the results obtained without/with our method.

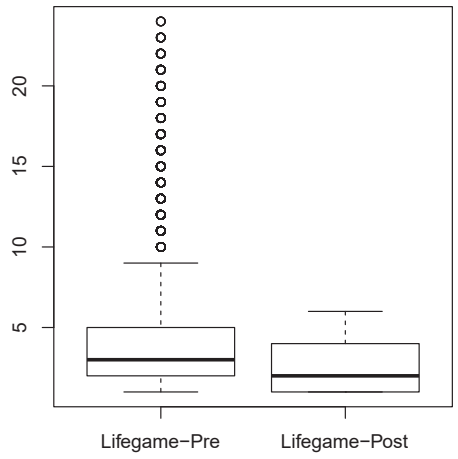

図 4 Lifegame の実行で観測された変数の表示位置の分布．“pre” および “post” はそれぞれ提案手法適用前，後の値を意味する．

Fig. 4 Distributions of variable positions observed in executions of Lifegame. “Pre” and “post” indicate the results obtained without/with our method.

参考文献

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

[2] Zeller, A.: 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, 2nd edition (2012).

[3] Rosenberg, J. B.: デバッガの理論と実装，アスキー出版局 (1998).

[4] Project, E.: The Eclipse Foundation open source community website., https://eclipse.org/.

[5] Alsallakh, B., Bodesinsky, P., Gruber, A. and Miksch, S.: Visual Tracing for the Eclipse Java Debugger, *CSMR 2012*, pp. 545–548 (2012).

[6] Lewis, B.: Debugging Backwards in Time, *Proceedigns of International Workshop on Automated Debugging*, (online), available from ⟨http://arxiv.org/html/cs/0309027v4⟩ (2003).

[7] Ko, A. J. and Myers, B. A.: Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 151–158 (online), DOI: 10.1145/985692.985712 (2004).

[8] Ressia, J., Bergel, A. and Nierstrasz, O.: Object-centric Debugging, *Proceedings of the 34th International Conference on Software Engineering*, pp. 485–495 (2012).

[9] Abramson, D., Chu, C., Kurniawan, D. and Searle, A.: Relative debugging in an integrated development environment, *Software Practice and Experience*, Vol. 39, No. 14, pp. 1157–1183 (2009).

[10] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S. and Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants, *Sci. Comput. Program.*, Vol. 69, No. 1-3, pp. 35–45 (online), DOI: 10.1016/j.scico.2007.01.015 (2007).