Vol.2020-SE-206 No.12
2020/11/27

デルタ抽出に基づく逆戻りデバッガの開発とその評価

石谷 涼[1,a)]　新田 直也[1,b)]

概要：大規模なソフトウェアの保守・再利用においては, プログラムの読解や調査に膨大な時間が費やされる. プログラムの読解にはデバッガが有効であるが, 実行と逆方向に解析したい場合に何度もデバッグ実行をしなければならず, 作業コストの増大を招いている. そこで, 実行時の内部情報を記録し, その情報を参照しながら実行と逆方向にデバッグすることができる逆戻りデバッガの研究が行われている. しかしながら, 一般に記録される情報が膨大になるため, その中から必要な情報を少ない手間で取り出すにはどうすればよいかは明らかではない. 本稿では, プログラム理解の効率化を目的に, 膨大な情報をデルタ抽出を用いて効率良く探索できる汎用かつ高速な逆戻りデバッガを開発したので紹介する. また, 実際の OSS を対象としてプログラムの読解作業に関する被験者実験を行い, 本デバッガの有効性の評価を行った. その結果, 通常のデバッガと比較して, 理解度を落とすことなく読解に要する時間を大幅に短縮することができたので報告する.

Development and Evaluation of Back-in-Time Debugger Based on Delta Extraction

1. はじめに

保守や再利用作業を適切に行う上で, 対象となるプログラムの理解は必要不可欠である. 一般にプログラムを理解するには, プログラムを実行せずにソースコードを読み進めていく方法と, デバッガを用いてプログラムを実行しながらソースコードを辿っていく方法がある. 本研究では, 後者のデバッガを用いたプログラム理解を対象とする. デバッガを用いてプログラムを読み進めていく場合に, プログラムの実行とは逆向きにソースコードを辿ることが要求される場合がある. しかしながら, 通常のデバッガを用いて実行と逆向きにソースコードを辿るには, ブレークポイントを入れる場所を変えながら, 同じ条件でデバッグ実行を何度も行う必要があり, 利用者の大きな負担となっている.

そこで, プログラム実行時の変数の値やメソッドの呼び出し関係などの内部情報を時系列にファイルに記録し, 実行終了後にそのファイルを参照しながら実行とは逆方向にデバッグすることができる逆戻りデバッガの研究が行われている. しかしながら, 一般に実行時の情報は非常に膨大になるため, 実行時情報を詳細に記録するようにすると,

- 解析対象プログラムの実行における実行時情報の収集処理の負荷が増大し, 実用的な規模のプログラムでの収集が困難になる,
- 同時に逆戻りデバッガが扱う実行時情報も膨大になり, 逆戻りデバッガ自身の処理速度も低下する,

といった問題を生じる. また, 逆戻りデバッガの使用時に, 膨大な実行時情報の中から, 利用者がどのようにして少ない手間で有用な情報を取り出すようにすればよいかも明らかではない.

そこで本研究では, 膨大な情報を効率よく探索できる汎用かつ高速な逆戻りデバッガの開発を目指す. 具体的には, Javassist[*1]を用いて実行時情報を収集し,

- フィールドへの値の代入時点まで遡る,
- コレクションへのオブジェクトの追加時点まで遡る,
- 現在の実行時点を登録し, 任意の実行時点から登録された実行時点に飛ぶ,
- デルタ抽出 [5] を行う,

といった機能を有した逆戻りデバッガを開発する. 本稿では, 開発した逆戻りデバッガを用いて, オープンソースソフトウェア (以下, OSS と略) の実際の機能を理解する被験者

[1] 甲南大学大学院 自然科学研究科 知能情報学専攻
Konan University, Kobe, Hyougo 658–8501, Japan
[a)] m1924001@s.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

[*1] http://www.javassist.org/

実験を行い, 逆戻り機能によって作業がどの程度効率化されるかを計測した. その結果, 対象とした 4 つの機能のうちの 3 つの機能で, 逆戻りデバッグ機能が有意に作業を効率化することを確認した. また, デルタ抽出機能を用いることによって作業がさらに効率化されることを確認するとともに, デルタ抽出機能の利用が理解度の向上に寄与することも確認した.

2. 逆戻りデバッガ

デバッガを用いてプログラムを読解する場合に, プログラムの実行とは逆向きにソースコードを辿ることが要求される場合が少なくない. たとえば文献 [3] では, 不具合が発生した時点での呼び出しスタック中にその原因を特定できる情報が含まれていないケースが, 全体の約 5 割を占めていることが報告されている. そのような場合, デバッガ上で不具合の発生を確認しても, すでにその原因を特定するために必要な情報が失われているため, 不具合の原因特定のために, プログラムの実行とは逆向きにソースコードを追跡していく必要がある. しかしながら通常のデバッガを用いて実行と逆向きにソースコードを辿るには,

- 次にブレークポイントを入れるべき場所を探す必要がある,
- 新しいブレークポイントを入れた後, 再び最初から同じ条件でデバッグ実行を行う必要がある,
- オブジェクトの ID が実行する度に変わるため, デバッグ実行するたびに対応するオブジェクトを特定する必要がある,
- 同一ブレークポイントで停止する回数は 1 回とは限らないため, その中から次に調べるべき停止を探す必要がある,

といった煩雑な作業が要求される.

そこで, プログラム実行時の変数の値やメソッドの呼び出し関係などの内部情報 (**実行トレース**) を収集し, トレースファイルとして時系列に記録して, 実行終了後にトレースファイルから読み込んだ情報に基づいて, 実行とは逆向きにデバッグすることを可能にする**逆戻りデバッガ**が研究されている. オムニサイエントデバッガ (ODB)[2] は, 実行時のすべての変数への代入を記録し, 実行の任意の時点まで遡ることが可能な逆戻りデバッガであるが, トレースファイルのサイズが膨大になるため, 実用的な規模のプログラムに対して適用が困難であるという問題がある. トレース指向デバッガ (TOD)[6] は, トレース全体を複数の部分に分割し分散データベースに保管することによって, 逆戻りデバッギングの実行効率の向上を図っている.

しかしながら, 対象プログラムの実行時の情報をすべて残すこれらのアプローチは, トレース情報の収集処理が対象プログラムの実行に非常に大きな負荷をかけてしまうという問題がある. また, 記録したトレース情報が膨大になることから, 逆戻りデバッギング自体の処理効率も実用レベルに到達させるのは難しい. これに対してオブジェクトフロー [4] は, オブジェクトがどこから来たのかを逆向きに追跡することに特化することによって, トレースファイルに記録すべき情報の大幅な削減に成功している.

逆戻りデバッギングのもう一つの問題は, 有用な情報を探し出す上での利用者の操作上の負担である. 単純に通常のデバッガと逆方向の操作を用意するだけでは, 膨大なトレース情報の中から有用な情報を探し出すには不十分である. 逆戻りデバッギングにおいて, 如何にしてより少ない手間で有用な情報を取り出せばよいかは今のところ明らかではない. オブジェクトフローは有用な情報を探索する上での一つの有効な手段であるといえる.

3. 逆戻りデバッガの開発目的および要件

前節で紹介したように, 実行時の情報を収集し, 実行とは逆向きにデバッグすることができる逆戻りデバッガの研究が行われている. 本研究では, プログラム理解の効率化を目的に, 膨大な情報を少ない手間で探索できる汎用かつ高速な逆戻りデバッガ (以下, 本デバッガと略) を開発する. 具体的には, 読み込んだトレースファイルに記録された情報を基にして, プログラムの実行とは逆向きにデバッグ実行ができるほか, デルタ抽出をはじめとするいくつかの探索機能を利用して必要な情報を素早く探し出し, プログラムの読解作業にかかる時間を大幅に短縮できるようにすることを目指す. 本デバッガは統合開発環境 Eclipse に組み込んで利用できるようにする. 一般に逆戻りデバッガなどの動的解析 [7] では膨大な量の実行トレースを収集し解析する必要がある. そのため, 収集および解析時のパフォーマンスには細心の注意を払う必要がある. 本デバッガの要件を以下にまとめる.

R1: 実行時の情報を可能な限り細かいステップで収集する.
R2: 実行トレースを収集することによる解析対象プログラムの実行速度の低下をできる限り抑える.
R3: 逆戻りデバッガ上での高速な解析を実現する.
R4: 膨大な実行時の情報を少ない手間で探索できるようにする.
R5: Eclipse に組み込んで利用できるようにする.
R6: 通常のデバッガのユーザインタフェースにできる限り近づける.

要件 R1, R4, R5 が本デバッガの機能的要件に相当し, R2, R3, R6 が非機能的要件に相当する. 要件 R5 の詳細については 5.1 節で述べる.

これらの要件の間にはいくつかのトレードオフ関係が存在している. 例えば, 実行時の情報を可能な限り細かいステップで収集する (要件 R1) ことによって解析対象プログラムの実行速度が大幅に低下する恐れがある (要件 R2). また, 要件 R1 と逆戻りデバッガ上での解析処理の効率性 (要

件 R3) にもトレードオフ関係が存在している．したがって, 上記の要件を同時に満たすことは容易ではない．そこで, 本デバッガの開発に際しては, 本研究室で開発された**動的解析プラットフォーム** [8] を利用する．動的解析プラットフォームは, さまざまな動的解析手法のツール開発で利用できるようなプラットフォームであり, 上記の要件 R1, R2, R3 を同時に満たせるように工夫して設計されている．詳細については 4 節で述べるが, 動的解析プラットフォームは, 解析対象プログラムに対して実行トレースの収集処理を提供するほか, 収集された実行トレースを Eclipse 上で解析するにあたっての共通基盤としての役割を果たす．

4. 動的解析プラットフォームについて

前節でも述べたように, 本デバッガの開発にあたっては, 本研究室で開発した動的解析プラットフォームを利用する．動的解析プラットフォームは, さまざまな動的解析手法のツール開発で利用することができるプラットフォームであり, 具体的には, さまざまな動的解析で利用できる実行トレースを収集するための処理と, 収集された実行トレースを解析するための共通基盤を提供する．動的解析プラットフォーム自体は Eclipse のプラグインとして開発されており, 各動的解析ツールはその拡張プラグインとして開発することができる．

実行時情報の収集は, 解析対象プログラムのバイトコードに対して実行時情報を収集するコードを埋め込むことで行う (以下, この操作を**インストゥルメンテーション**と呼ぶ)．動的解析プラットフォームでは, 要件 R1 を満たすよう, インストゥルメンテーションに Javassist を用いている．Javassist を用いると, メソッド実行の開始および終了, コンストラクタ実行の開始および終了, フィールドの更新や参照などの基本的なイベントに関する情報に加えて, 配列の生成, 配列要素の更新および参照, 基本ブロック間の制御の移動などのイベントに関する情報も取得することができる．インストゥルメンテーションが行われたプログラムを起動すると, その中に埋め込まれた収集コードが実行時イベントの発生に応じて実行される．これによって, 実行時情報が実行トレースとして収集され, 収集された実行トレースは JSON 形式に変換されてトレースファイルに記録される．ただし, 要件 R2 を満たすよう, 解析対象プログラムの実行中はトレース情報をメモリ中に蓄積したまま, 解析対象プログラムの実行終了時にその情報をまとめてトレースファイルに書き出して記録するようになっている．収集された実行トレースの解析時には, トレースファイルに記録されたトレース情報がメモリ中に復元され, 動的解析プラットフォームはそのトレース情報にアクセスするためのインタフェースを提供する．各動的解析ツールはそのインタフェースを用いて解析ライブラリからアクセスを行い, トレース情報を解析に利用することができる．

5. 逆戻りデバッガプラグインの開発

5.1 Eclipse との統合

統合開発環境 Eclipse 上で動作するように, 本デバッガを Eclipse のプラグインとして開発する．以下では, このプラグインを逆戻りデバッガプラグインと呼ぶ．Eclipse プラグインは, Eclipse の本体を拡張して固有の機能を組み込むことができる Java プログラムで, Eclipse 本体自体も多数のプラグインによって構成されている．本節では, 要件 R5 を以下のような形で実現することを考える (画面構成は図 1 を参照)．

(1) Eclipse のワークスペース上にある任意の Java プロジェクトに対してインストゥルメンテーションを行い, トレース情報を収集することができる (動的解析プラットフォームが提供)．
(2) 本デバッガ上でトレースファイルを読み込み, トレース情報をメモリ中に復元することができる．
(3) 解析対象プログラムに配置された Eclipse 標準のブレークポイントを取り込み, 読み込んだトレース情報上に設定することができる．
(4) 本デバッガによるデバッグ中に, 必要に応じて解析対象プログラムのソースファイルを Java エディタ上で自動的に開くことができる．
(5) 本デバッガによる解析の結果を, Eclipse の画面上に表示することができる．

5.2 逆戻りデバッガの機能と特徴

本デバッガでは, 基本的なデバッグ操作として, 通常のデバッガにおけるステップ実行 (ステップオーバー, ステップイン, ステップリターン) や再開機能のほか, それらを逆方向に行うステップバック実行 (ステップバックオーバー, ステップバックイン, ステップバックリターン) や逆方向への再開機能も利用できる．ステップバックオーバーでは, 実行時点をインストゥルメンテーションを行った行単位で遡ることができる．ステップバックインでは, 実行時点を実行文単位で遡り, 呼び出し先のメソッドがあった場合は, その呼び出し先メソッドに実行の時系列上で後ろから入ることができる．ステップバックリターンでは, 現在一時停止しているメソッドが呼び出される直前まで遡ることができる．逆方向への再開機能では, 有効なブレークポイントが入っている行のうち, 現在の実行時点よりも過去の実行時点で最初に遭遇する行にまで実行が遡り, そこで実行を再び一時停止させることができる．

3 節でも述べた通り, 一般に収集された情報は膨大になるため, その中から必要な情報を少ない手間で探し出すには, 上述の基本的なデバッグ操作だけでは不十分である．そこで, 要件 R4 を以下の機能の実装によって実現することを考える．

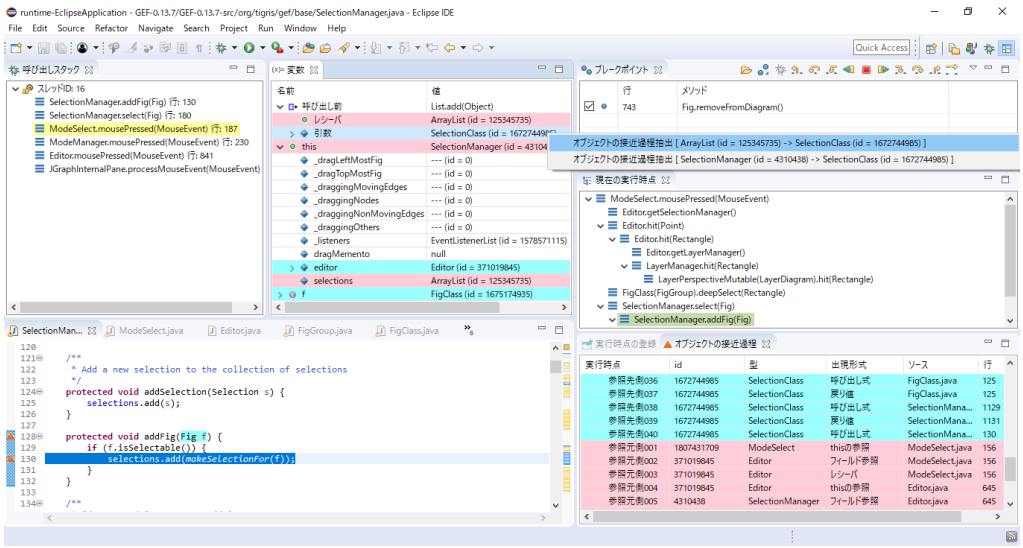

図 1　本デバッガの画面構成

(1)「値の代入時点に飛ぶ」
(2)「オブジェクトの追加時点に飛ぶ」
(3)「実行時点の登録」および「登録した実行時点に飛ぶ」
(4)「オブジェクトの接近過程抽出」(デルタ抽出)

(1) の機能は, 選択したフィールドが保持している値に対して, その値が実際に代入された時点まで実行を遡ることができる機能である. (2) の機能は, リストなどのコレクションから取得してきたオブジェクトに対して, そのオブジェクトがコレクションに対して実際に追加された時点まで実行を遡ることができる機能である. (3) の機能は, 逆戻りデバッグ中に, 現在一時停止している実行時点を登録する機能であり, 登録した実行時点には後からいつでも移ることができる. (4) の機能は, デルタ抽出という動的解析手法を用いて, 2 つのオブジェクトが関連付けられた経緯を表すコードを抜き出す機能であり, 抜き出したコードに対応する実行時点へはいつでも移ることができる. デルタ抽出の詳細については 6 節で述べる.

本デバッガによるデバッグ実行は, 実際に解析対象プログラムを実行するのではなく, プログラム実行時の情報が記録されたトレースファイルを読み込み, その情報を基に実行を再現するものである. その性質上, 本デバッガは, 通常のデバッガと比較すると以下のような違いがある.

(1) 実行トレースを記録したトレースファイルを読み込むため, 何度でも同じ実行を再現することができる.
(2) オブジェクト ID がトレースに記録されているため, 何回実行しても同一オブジェクトの ID は変わらない.
(3) ローカル変数の値はトレースに記録されていないため表示されない.
(4) 全ての行をトレースに記録しているわけではないので, ステップ実行の際に途中の行を飛ばすことがある.
(5) 部分的には一行よりも細かい単位で記録しているので, 一行よりも細かい単位でのステップ実行も可能である.
(6) 操作や画面の情報が残っていないので, その実行でユーザがアプリケーションをどのように操作し, アプリケーションでどんな画面が表示されたかはわからない.

何度でも同じ実行を再現できることは, 解析対象プログラムを理解する上では有効であると考えられる. これは, 通常のデバッガではユーザが毎回同じ操作を行う必要があり, その上完全に同じ実行を再現できるとも限らないのに対し, 本デバッガでは完全に同じ実行を再現することができる上, ユーザによる操作も必要なくなるためである. また, 何回実行しても同一オブジェクトの ID が変わらないことも同様に有効であると考えられる. これは, 通常のデバッガで複数回デバッグ実行をした場合, デバッグ実行の度にオブジェクトの ID が変わるため, 同一オブジェクトを探し出す作業が必要となるためである.

5.3　逆戻りデバッガの画面構成

本研究で開発した逆戻りデバッガプラグインの画面構成を図 1 に示す. 逆戻りデバッガプラグインの画面は, 7 つのビュー (Java エディタ, ブレークポイントビュー, 呼び出しスタックビュー, 変数ビュー, 実行時点の登録ビュー, オブジェクトの接近過程抽出ビュー, 現在の実行時点ビュー) で構成される. 各ビューは, 要件 R6 を満たすよう, 通常のデバッガのユーザインタフェースに似せて画面設計を行った. Java エディタは, Eclipse 標準のものをそのまま利用する. ブレークポイントビューでは, 逆戻りデバッグを実行するにあたって必要なブレークポイントの設定が行えるほか, 実際に逆戻りデバッグの実行を制御する操作をユーザに提供する. 呼び出しスタックビューでは, 逆戻りデバッグの実行によって現在一時停止しているメソッド実行をトッ

プとする呼び出しスタックを表示する．変数ビューでは，現在のスタックフレームから見える仮引数や this の値が表示され，その値がオブジェクトへの参照の場合は，そのクラス名と ID を確認することができる．なお，ローカル変数の値は本デバッガでは表示されない．そのかわりに，本デバッガではフィールドへの代入や参照の際には参照元側オブジェクトと参照先側オブジェクトの値が，メソッド呼び出しの際にはレシーバや引数や戻り値といったオブジェクトの値が表示されるようになっている．また，変数ビューに表示されている値を右クリックすることで，5.2 節で述べた探索機能のうち，(1), (2), (4) を必要に応じて利用することができる．実行時点の登録ビューでは，探索機能 (3) を利用することができる．オブジェクトの接近過程抽出ビューおよび現在の実行時点ビューは，探索機能 (4) のデルタ抽出の結果を表示するためのものである．オブジェクトの接近過程抽出ビューでは，デルタ抽出によって抜き出されたコードに関連する実行時点の一覧が表示される．必要に応じてこの一覧をクリックすることで，これらの実行時点にいつでも移ることができる．現在の実行時点ビューでは，抜き出されたコードに関連するメソッド呼び出しの階層構造が表示されるほか，表示されているメソッド呼び出しの中に現在一時停止しているメソッド呼び出しが存在する場合は，そのメソッドがハイライト表示される．これにより，現在一時停止している実行時点が，デルタ抽出で抜き出された範囲内のどこにあるのかを確認しながら探索することができる．

6. デルタ抽出

本研究では，膨大な実行時情報を少ない手間で探索するための手法として，デルタ抽出を利用する．デルタ抽出とは，2 つのオブジェクトの参照が，プログラムの実行中にどのように渡されてくることによって関連付けられたのかを解析する手法である．デルタ抽出を行うことで，2 つのオブジェクトが関連付けられた経緯を表すソースコードを抜き出すことができる．デルタ抽出は，オブジェクトフローよりも広範囲のソースコードを抽出することができるが，動的スライス [1] よりは抽出範囲が狭い．

デルタ抽出の概要について，図 2 に示したプログラムを例に説明する．例えば，図 2 の 25 行目では代入文によってクラス E のインスタンスとクラス C のインスタンスが関連付けられている．ここでは，クラス C のインスタンスがクラス E のインスタンスに渡された経緯，すなわちクラス E のインスタンスがクラス C のインスタンスをフィールド c によって参照できるようになるまでの経緯について考える．ここで，E のインスタンスを参照元，C のインスタンスを参照先オブジェクトと呼ぶ．クラス C のインスタンスは過去に 15 行目で生成されていたものが，17 行目で戻り値として呼び出し元に返され，それが 11 行目で実引数としてクラス E のインスタンスに渡されることで，最終的に 25 行目に到達する．ここで，17, 11, 24, 25 行目に出現するオブジェクトの参照は，全て同一のクラス C のインスタンスへの参照である．このとき，クラス C のインスタンスがクラス E のインスタンスに受け渡されるまでの経緯には，クラス C のインスタンスだけでなく，それを参照しているクラス B のインスタンスも間接的に関与する．なぜならば，このクラス B のインスタンスが，もしクラス B の別のインスタンスであった場合，11 行目の `getC()` メソッドの呼び出しによって取得されるクラス C のインスタンスも別のものに変わり得るからである．ここでは，クラス C のインスタンスおよびクラス B のインスタンスを参照先側に関連するオブジェクトとする．同様に，クラス E のインスタンスもまたクラス D のインスタンスによって参照されており，これらは参照元側に関連するオブジェクトとする．このとき，参照先オブジェクトが参照元オブジェクトに渡されるためには，参照元オブジェクトと参照先オブジェクトの両方を知っているオブジェクトが少なくとも一つ必要となる．図 2 のプログラムにおいては，クラス A のインスタンスがそれに相当する．したがって，クラス E のインスタンスからクラス C のインスタンスへのオブジェクト間参照は，クラス A のインスタンスが `m()` メソッドによって参照先オブジェクトを参照元オブジェクトに紹介することによって生成されたとみなすことができる．この例においては，25 行目で生成されたクラス E のインスタンスからクラス C のインスタンスへのフィールド c によるオブジェクト間参照に対してデルタ抽出を行うことで，クラス A のインスタンスの `m()` メソッドを実行の時系列上での開始時点，参照が生成された時点を実行の時系列上での終了時点とするデルタを抽出することができ，抽出したデルタに関連するコードとして，参照先側オブジェクト (B, C) が参照されている 5, 10, 11, 17, 24, 25 行目と，参照元側オブジェクト (D, E) が参照されている 5, 11, 25 行目のコードを抜き出すことができる．

```
 1: class A {                    14: class B {
 2:     B b = new B();           15:     C c = new C();
 3:     D d = new D();           16:     C getC() {
 4:     void m() {               17:         return c;
 5:         d.passB(b);          18:     }
 6:     }                        19: }
 7: }                            20: class C {
 8: class D {                    21: }
 9:     E e = new E();           22: class E {
10:     void passB(B b) {        23:     C c = null;
11:         e.setC(b.getC());    24:     void setC(C c)
12:     }                        25:         this.c = c;
13: }                            26:     }
                                 27: }
```

図 2 サンプルプログラム

7. 評価実験

7.1 実験の概要とリサーチクエスチョン

本研究で開発した逆戻りデバッガが実際のプログラムの機能理解において有効であるかどうかを検証するため, 実際の OSS を対象としたプログラムの読解作業について, 本デバッガを用いた場合と, そうでない場合を比較する評価実験を行う. 比較にあたり, 本デバッガの機能を一部制限したもの (以下, 順方向デバッガと略) を用意した. 順方向デバッガは, 本デバッガから逆戻りデバッガ特有の逆戻り操作や探索機能を取り除き, 通常のデバッガに存在する機能のみを利用できるようにしたものである. ただし, 順方向デバッガについても, トレースファイルを読み込んでデバッグ実行を再現するという点で, 通常のデバッガとは異なる特徴を持つことに注意する必要がある.

一般に, プログラム理解の効率を評価するにあたっては, 対象プログラムの読解作業に費やされる時間を計測すればよい. しかしながら, たとえ読解作業を大幅に短縮できたとしても, 対象プログラムに対する理解度そのものが低下してしまうのであれば, プログラム理解を効率化できたとは言えない. したがって, 本実験では, 本デバッガによって, プログラムの機能理解に必要十分な情報だけを効率よく探索し, 理解度を落とすことなく読解作業を削減できるか否かを調べる必要がある. また, ユーザインタフェースの使い易さも評価すべき課題である. なぜなら, ユーザインタフェースが使い難いと, それだけ本デバッガの学習コストや作業コストが高くなるためである. 実験を行うにあたってのリサーチクエスチョンを以下にまとめる.

RQ1: 機能理解において, 本デバッガからデルタ抽出機能を除いたものを用いた場合, 順方向デバッガを用いた場合よりも速く探索できるか.

RQ2: 機能理解において, 本デバッガからデルタ抽出機能を除いたものを用いた場合の理解度と, 順方向デバッガを用いた場合の理解度に差があるか.

RQ3: 機能理解において, 本デバッガにおけるデルタ抽出は探索を効率化するか.

RQ4: 機能理解において, 本デバッガにおけるデルタ抽出は理解度に影響を与えるか.

RQ5: 本デバッガのユーザインタフェースは使い易いか.

RQ1 および RQ2 が逆戻りデバッグ機能自体での, RQ3 および RQ4 が逆戻りデバッグ機能とデルタ抽出の組み合わせでのプログラム理解の効率化に関する問いであり, RQ5 がユーザインタフェースの使い易さについての問いである.

本実験に用いる対象プログラムは ArgoUML ver. 0.34[*2] と JHotDraw ver. 7.6[*3] の 2 つである. その中で, ArgoUML からは図形削除機能と図形選択機能の 2 つを, JHotDraw からは図形移動機能と図形選択機能の 2 つを選び, それらの機能に関するソースコードを読解する課題を用意する. また, 被験者として, Java の実務経験が 3 年以上のエンジニアをクラウドソーシングサービスで募集する. 本実験の全体的な流れは次の通りである. まずは, 簡単な例題プログラムを題材に, 順方向デバッガと本デバッガについて, 各機能を実際に利用してもらいながら一通りの操作の練習を行う. その後, 上述の各課題について, 順方向デバッガと本デバッガのいずれかを用いながら操作時間の計測と理解度のアンケートを行う. デバッガの評価実験時にデルタ抽出の評価実験もそれぞれ行い, 各実験の終了時にアンケートを実施する. 4 つの課題全ての評価実験の完了後, 最後に最終アンケートを実施する. 本実験では, RQ1 および RQ2 を逆戻りデバッガの評価実験で, RQ3 および RQ4 をデルタ抽出の評価実験で検証し, それぞれが成り立っているか否かを明らかにする. また, 最終アンケートの設問の一つとして, 本デバッガのユーザインタフェースが適切であったか否かを問い, RQ5 が成り立っているか否かを明らかにする. なお, 逆戻りデバッガの評価実験の詳細については 7.2 節で, デルタ抽出の評価実験の詳細については 7.3 節でそれぞれ述べる.

[*2] https://github.com/argouml-tigris-org/argouml
[*3] https://sourceforge.net/projects/jhotdraw/

7.2 逆戻りデバッガの評価実験

本節では, リサーチクエスチョン RQ1, RQ2 が成り立っているか否かを検証する実験について説明する. 具体的には, 前節で述べた順方向デバッガを用いる場合と, 本デバッガを用いる場合とで, 対象プログラムの読解作業に要した時間を分単位でそれぞれ計測し, 所要時間の比較を行う. ただし, ここでの読解作業では, 本デバッガの機能のうち, デルタ抽出機能は用いないものとする. ここで, 実験を実施するにあたっては, 順序効果に注意する必要がある. まず, 本実験において順方向デバッガと本デバッガを用いる順番であるが, 順方向デバッガは通常のデバッガと同様の機能を持つため先に用いて 2 つの課題に取り組み, その後に本デバッガを用いて残りの課題に取り組む. 次に, 読解作業を行う対象プログラムの順番であるが, 本実験では, 先に順方向デバッガによる ArgoUML の課題に取り組んでから本デバッガによる JHotDraw の課題に取り組むグループと, 先に順方向デバッガによる JHotDraw の課題に取り組んでから本デバッガによる ArgoUML の課題に取り組むグループの 2 つのグループに被験者を分け, 順序効果をなくすようにする. 各グループには, Java の実務経験年数が偏らないよう, 被験者を割り当てる. 被験者はグループに割り当てられた順番にしたがって各プログラムの読解作業を進めていき, それぞれの読解作業が完了するごとに, 読解作業に要した時間と, その機能の理解度を問うアンケートに回答してもらう. アンケートの結果から, RQ1 および RQ2 が成り立っているか否かについてそれぞれ評価を行う.

7.3 デルタ抽出の評価実験

5.2 節でも述べたように, 膨大な実行時情報を少ない手間で探索できるよう, 本デバッガでは実行時情報の探索機能の一つとしてデルタ抽出を利用できるようにした. 本節では, デルタ抽出の有効性に関わるリサーチクエスチョン RQ3, RQ4 を調べる実験について説明する. 具体的には, 各課題について, 前節の実験によるプログラム読解作業の終了後, デルタ抽出を利用してソースコードを抜き出す作業を行い, 抜き出す作業および抜き出されたコードが下記の項目を満たしているか否かについて, それぞれアンケートを実施して検証する.

(1) プログラムの読解作業のどのようなときにデルタ抽出を利用すればよいかがわかるか. (利用目的)
(2) デルタ抽出はプログラムの読解作業を効率化するか. (効率化)
(3) デルタ抽出によって抜き出されたソースコードは過不足ないものであるか. (抽出範囲)
(4) デルタ抽出によって抜き出されたコードは対象プログラムの機能理解を容易にするか. (容易化)

アンケート項目 (1) および (2) の結果から RQ3 が成り立っているか否かについて, (3) および (4) の結果から RQ4 が成り立っているか否かについて評価を行う.

8. 考察

8.1 逆戻りデバッガの考察

7.2 節で述べた実験での, プログラムの読解作業に要した時間の比較結果を図 3 に示す. なお, 図 3 では, 本デバッガからデルタ抽出機能を除いたものを, “逆方向” と表記している. 実験の結果から, 本デバッガからデルタ抽出機能を除いたものは, 順方向デバッガと比較してプログラムの読解に要する時間が, 4 つの課題の平均で 1.52 倍速くなることを確認できた. また, 3 つの課題において, 順方向デバッガを用いた場合の所要時間と, 本デバッガからデルタ抽出機能を除いたものを用いた場合の所要時間に有意差があることも確認することができた. 有意差が確認できなかった JHotDraw の移動機能に関する課題については, 他の課題と比較して機能理解にあたってのプログラム読解のための作業量が少なく, 順方向デバッガを用いた場合でも比較的短時間で読解作業を終えることができたからだと考えられる. これらの結果から, リサーチクエスチョン RQ1 については正しいといえる. また, 各課題についての理解度の比較結果を図 4 に示す. ここでは, 理解度について 0〜10 の 11 段階で評価を行っており, 10 に近いほど理解度が高いことを示している. 実験の結果から, いずれの課題においても順方向デバッガを用いた場合の理解度と, 本デバッガからデルタ抽出機能を除いたものを用いた場合の理解度に有意差を確認することはできなかった. したがって, リサーチクエスチョン RQ2 については, これら 2 つを用いた場合の理解度に差はないといえ, RQ1 と RQ2 の結果を合わせると, 逆戻りデバッグ機能は, 課題の理解度を落とすことなく読解作業の効率化に成功しているといえる.

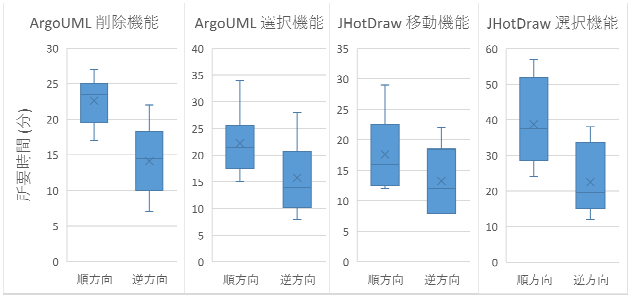

図 3 順方向デバッガを用いた場合と本デバッガからデルタ抽出機能を除いたものを用いた場合とでの各課題の所要時間比較

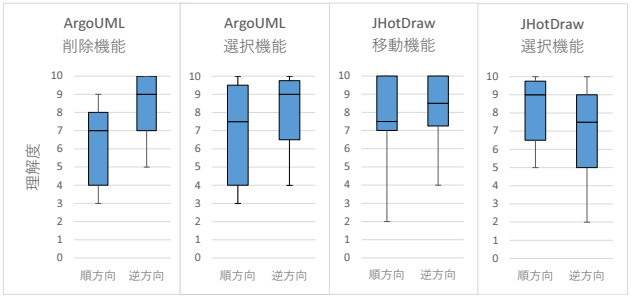

図 4 順方向デバッガを用いた場合と本デバッガからデルタ抽出機能を除いたものを用いた場合とでの各課題の理解度比較

8.2 デルタ抽出の考察

7.3 節で述べた実験での, デルタ抽出の有効性についての実験結果を図 5 に示す. ここで, アンケート項目 (1) および (3) は 0〜10 の 11 段階, (2) および (4) は 0〜5 の 6 段階で評価を行っている. (1), (2), (4) は値が高くなるほど良い結果であることを表している. (3) はデルタ抽出によって抜き出されたソースコードの範囲について, 0 に近いほど不足していることを, 10 に近いほど過剰であることを表しており, 5 に近いほど良い結果となる. アンケート項目 (1) および (2) の結果から, いずれの課題においても, 被験者は機能理解を行うにあたってデルタ抽出を適切な場面で利用することができ, それによって実際に作業が効率化されると考えていることが確認できる. 本実験での各課題においては, 機能理解に必要な範囲を 1〜3 個のデルタでカバーすることができており, 本デバッガにおける作業ステップのかなりの割合をデルタ抽出によって置き換えることができる. 各デルタは高々 2.5 秒で抽出することができるため, 読解作業に要する時間のかなりの部分を削減できると考えられる. 以上のことから, リサーチクエスチョン RQ3 については正しいといえる. また, アンケート項目 (3) および (4) の結果から, デルタ抽出は機能理解に過不足ない範囲の

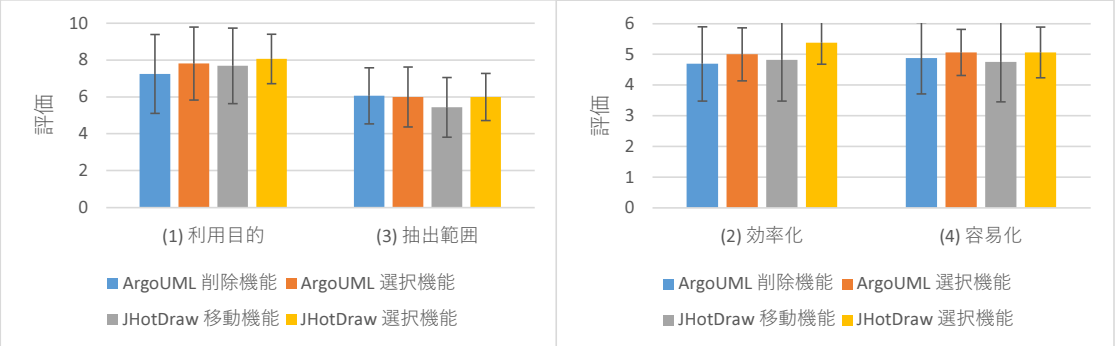

図 5 デルタ抽出の有効性の評価についてのアンケート結果

ソースコードを抜き出すことができており, それによって機能理解がより容易になったことが確認できる. したがって, リサーチクエスチョン RQ4 については, 機能理解において, デルタ抽出を用いることでその理解度を向上させることができるといえる.

8.3 逆戻りデバッガの使い易さの考察

本実験の最終アンケートにおいて, 本デバッガのユーザインタフェースが適切であったか否かについて, 0〜10 の 11 段階で回答してもらい評価を行った. このアンケートでは, 0 に近いほど適切でないことを, 10 に近いほど適切であることを表している. 実験の結果, 平均 8.56 の評価を得ることができた. 一方で, メニューの項目名や逆戻りデバッガ特有の操作のわかりにくさや, デバッグ操作のためのボタンの配置についての指摘も一部見られた. しかしながら, 最も低く評価した被験者についても, 11 段階中で 6 の評価を得ることができ, 本デバッガのユーザインタフェースが適切でないと評価した被験者はいなかった. これらの結果から, リサーチクエスチョン RQ5 については成り立っているといえる.

9. おわりに

プログラム理解の効率化を目的に, 膨大な情報を少ない手間で探索できる汎用かつ高速な逆戻りデバッガを開発した. 本デバッガでは, 通常のデバッガにおける基本的なデバッグ操作や, それを逆方向に行うことができる機能のほか, 探索機能の一つとしてデルタ抽出を取り入れ, 収集された膨大な実行時情報から必要な情報を少ない手間で探し出せるようにすることを目指している. 実際の OSS を対象としてプログラムの読解作業に関する被験者実験を実施し, 逆向きデバッグ機能およびデルタ抽出機能によるプログラム読解作業の効率化および理解度への影響の評価を行った. その結果, 本デバッガは通常のデバッガと比較すると, 解析対象プログラムの理解度を落とすことなく読解に要する時間を大幅に削減することができ, デルタ抽出もまた理解度を落とすことなく作業を効率化できることを確認することができた.

謝辞 この研究の一部は, 私立大学等経常費補助金 特別補助「大学間連携等による共同研究」による.

参考文献

[1] Agrawal, H. and Horgan, J. R.: Dynamic program slicing, *Proc. the Conference on Programming Language Design and Implementation*, pp. 246–256 (1990).
[2] Lewis, B.: Debugging Backwards in Time, *Proc. Fifth International Workshop on Automated Debugging (AADEBUG'03)*, (2003).
[3] Liblit B., Naik M., Zheng A., Aiken A. and Jordan M.: Scalable Statistical Bug Isolation, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 15–26 (2005).
[4] Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugging, *Proc. 22nd European Conference on Object-Oriented Programming*, pp. 592–615 (2008).
[5] Nitta, N. and Matsuoka, T.: Delta extraction: An abstraction technique to comprehend why two objects could be related, *Proc. 31st International Conference on Software Maintenance and Evolution*, pp. 61–70 (2015).
[6] Pothier, G., Tanter, E. and Piquer, J.: Scalable Omniscient Debugging, *Proc. 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, pp. 525–552 (2007).
[7] 石尾 隆: プログラムの動的解析, コンピュータソフトウェア, Vol. 29, No. 1, pp. 47–60 (2012).
[8] 石谷 涼, 新田 直也: オンラインおよびオフライン動的解析プラットフォームの開発とそのオブジェクトフロー解析への応用, 情報処理学会研究報告, 2019-SE-202(12) (2019).