Vol.2019-SE-202 No.12
2019/7/13

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

石谷 涼[1,a)]　新田 直也[1,b)]

概要：ソフトウェア工学のさまざまな分野において動的解析技術が用いられている．動的解析とは, プログラムの実行時の情報を収集し解析を行うプログラム解析技術で, 解析対象となるプログラムの実行終了後に解析を行うオフライン解析と, 実行の途中でそれまでに収集された情報の解析を行うオンライン解析に分類することができる．本稿では, オンラインおよびオフライン解析ツールの開発支援を目的として本研究室で開発した動的解析プラットフォームの紹介を行う．また, 本プラットフォームを利用して Java プログラムを対象としたオブジェクトフロー解析ツールを開発したので報告する．

Online and Offline Dynamic Analysis Platform and Its Application to Object Flow Analysis

1. はじめに

プログラムの動的解析技術は, 統計的デバッギング [7], [11], [14], コードカバレッジ分析 [2], [3], 動的プログラムスライシング [1], [13] など, ソフトウェア工学におけるさまざまな分野で用いられている．動的解析とは, プログラムの実行時の情報を収集し解析を行うプログラム解析技術である．プログラムの実行中に収集した実行時情報の系列を**実行トレース**という．動的解析は, 対象プログラムの実行終了後に実行トレースを解析する**オフライン解析**と, 対象プログラムの実行の途中でそれまでに収集された実行トレースの解析を行う**オンライン解析** [5] に分類することができる．オフライン解析では, 通常実行トレースの内容はプログラム終了時にファイル (トレースファイル) に記録され, プログラム終了後トレースファイルを参照することによって, いつでも解析を行うことができる．一方オンライン解析では, 対象プログラムの実行を継続した状態で実行トレースの解析が行われ, 解析後も実行トレースの収集が続けられる．

本研究では, オンラインおよびオフライン解析ツールの開発を支援するため, Eclipse 上で動作する動的解析プラットフォーム (以下, 本プラットフォームと呼ぶ) を開発する．具体的には, 以下の 2 つの特徴を持つプラットフォームを開発することによって, 解析ツールの開発の効率化を図る．

- これまで, 特にオンライン解析においてツール毎に個別に実装されてきた実行トレースの収集処理を共通化し, 単一の収集処理で, さまざまな動的解析手法で利用可能な実行トレースの収集ができるようにする．
- 収集された実行トレースの解析を Eclipse 上で行うための共通基盤を提供する．

まず, 実行トレースの収集処理の共通化について考える．基本的に実行トレースに記録すべき内容は, 動的解析の手法によって異なる．例えば統計的デバッギングでは, プログラムを構成する各コンポーネントがその実行の中でそれぞれ何回実行されたかなどの比較的粒度の粗い情報が, 動的プログラムスライシングでは, 実行時にソースコードのどの行がどういう順序で実行され, 各行の実行により変数の値がどう変化したかなどのより詳細な情報が必要となる．実行トレースをあらゆる動的解析の手法で利用できるようにするためには, 対象プログラムの詳細な実行時情報を収集しなければならず, その収集処理が対象プログラムの実行に大きな負荷を与えかねない．そこで本プラットフォームでは, 対象プログラムの実行に多大な負荷を与えることなく, できる限り詳細な情報を収集できるよう Javassist[*1] を

[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/

© 2019 Information Processing Society of Japan

用いて, 収集処理の実装を行う.

次に, オンラインおよびオフライン解析を Eclipse 上で行うための共通基盤の提供について考える. 特にオンライン解析では, 実行トレースの収集処理を継続したまま解析処理を行えるようにする必要があり, そのための実装が複雑になる傾向がある. Eclipse 上でデバッグ実行中の Java プログラムに対するオンライン解析を実現するには, Eclipse を実行している Java 仮想マシン (解析 JVM と呼ぶ) と解析対象プログラムを実行している Java 仮想マシン (対象 JVM と呼ぶ) の間でプロセス間通信を行う必要がある. しかしながらこの通信量が増大すると, 解析処理への負荷が大きくなると予想されるため, 本プラットフォームでは, 対象 JVM 上で収集された実行トレースの情報を解析 JVM 側へは転送せず, 対象 JVM 上で解析処理を行うことによって通信量の削減を図る設計を行う.

本稿では, 上記設計を採用した場合と採用しない (解析処理のすべてを解析 JVM 上で行う) 場合での解析速度の比較を行った. その結果, この設計を採用しない場合と比べて, 採用した方が 1000 倍以上解析処理が高速化されることを確認した. また, 本プラットフォームを利用して, Java プログラムを対象としたオブジェクトフロー解析 [8] のオンライン解析ツールおよびオフライン解析ツールを開発し, オンライン解析版とオフライン解析版で, 解析アルゴリズムの実装を完全に共通化できることを確認した. なお, 本プラットフォームは統合開発環境 Eclipse のプラグインとして実装されており, ソースコードを GitHub 上に LGPL ライセンスにて公開している[*2].

2. プログラム解析

ソフトウェア開発のさまざまな工程において, プログラムを読解したり調査する作業が発生する. 特に大規模なソフトウェアの開発では, それらの作業に膨大な時間が費やされることが少なくない. プログラム解析はそのような作業を支援するための技術であり, 大きく静的解析と動的解析に分類することができる. **静的解析**はプログラムを実行することなく基本的にソースコードなどの成果物から得られる情報のみに基づいて解析を行うプログラム解析技術で, **動的解析**はプログラムを実際に実行し実行時の情報 (コンポーネントの実行順序や変数の値など) を収集して解析を行うプログラム解析技術である.

静的解析と動的解析のそれぞれに長所・短所が存在するが, 本研究では動的解析に着目し, 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汎用の動的解析プラットフォームを開発する. 動的解析において, プログラムの実行中に収集した実行時情報の系列を**実行トレース**という. 動的解析を行う方法として, 対象プログラムの実行終了後に実行トレースを解析する**オフライン解析**と, 対象プログラムの実行の途中でそれまでに収集された実行トレースの解析を行う**オンライン解析** [5] がある. オフライン解析では, 通常実行トレースの内容はプログラム終了時に**トレースファイル**に記録され, プログラム終了後トレースファイルを参照することによって, いつでも解析を行うことができる. 一方オンライン解析では, 対象プログラムの実行を継続した状態で実行トレースの内容が解析され, 解析後も実行トレースの収集が続けられる. また, 解析結果をそれ以降のプログラムの実行に反映させることができるという特長を持つ.

動的解析には, 目的に応じてさまざまな手法が存在し, また手法によって用いられる実行時情報も異なる. 例えば, コードカバレッジはテストの網羅性を測るひとつの尺度であるが, その計算には各命令文や基本ブロック, メソッドなどが実際に実行されたか否かについての情報が用いられる. また, 統計的デバッギングは不具合の要因となっているコンポーネントを推測する技術であるが, 失敗したテストケースと成功したテストケースのそれぞれにおいて, プログラムを構成する各コンポーネントが何回ずつ実行されたかについての情報が用いられる. 動的プログラムスライシングは, ある時点の変数の値に影響を与えたソースコード中の行をすべて抜き出す技術であるが, その計算にはソースコードのどの行がどういう順序で実行され, 各行の実行により変数の値がどう変化したかなどのより詳細な情報が必要となる.

3. 動的解析プラットフォームの開発目的および要件

前節で紹介したように動的解析にはさまざまな手法が存在しているが, それらの手法のツール開発で利用することができる共通のプラットフォームの提供を目的として, 汎用な動的解析プラットフォーム (以下, 本プラットフォームと略) を開発する. 具体的には, これまで, 特にオンライン解析においてツール毎に個別に実装されてきた実行トレースの収集処理を共通化し, さらに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ことによってツール開発を支援することを目指す. 本プラットフォームの共通基盤はオンライン解析とオフライン解析の両方に対応し, 統合開発環境 Eclipse に組み込んで利用できるようにする. 一般に動的解析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. 本プラットフォームの要件を以下にまとめる.

R1: オンライン解析とオフライン解析の両方に対応する.
R2: さまざまな動的解析の手法で利用できるよう実行トレースの収集処理を共通化する.
R3: プラットフォーム自身は個々の動的解析に依存しない

[*2] https://github.com/nitta-lab/org.ntlab.traceAnalysisPlatform

© 2019 Information Processing Society of Japan

よう汎用性を高く保つ.

R4: Eclipse に組み込んで利用できるようにし, 動作中の Java プログラムだけでなく, Eclipse のデバッガ上で一時停止しているプログラムに対してもオンライン解析ができるようにする.

R5: 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.

R6: 同一の解析アルゴリズムをオフライン解析とオンライン解析で実装したときに, オフライン解析に対するオンライン解析での処理速度の低下をできる限り抑える.

R7: 実行トレースへのアクセスコードを, オンライン解析であるかオフライン解析であるかに依存することなく記述することができる.

要件 R1, R2, R3, R4 が本プラットフォームの機能的要件に相当し, R5, R6, R7 が非機能的要件に相当する. 要件 R4 の詳細については, 4.2 節で述べる. Eclipse への組み込みは, 特にデバッガと組み合わせたオンライン解析において有効であると考えている. 例えば, ブレークポイントで停止した状態から, 過去に遡って実行を追跡できるようなデバッガへの応用などを想定している.

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. 詳細は次節で述べるが, さまざまな動的解析の手法で利用できるよう実行トレースの収集処理を共通化 (要件 R2) するためには, 詳細な実行時情報が収集できるような収集処理を用意する必要がある. しかしながら, そのような収集処理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 R5). また, 収集処理の効率性 (要件 R5) と, オンライン解析処理の効率性 (要件 R6) およびプラットフォームの汎用性 (要件 R3) の間にもある種の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. 次節で, これらの要件を同時に満たすために行ったアーキテクチャ設計上の選択について説明する.

4. 動的解析プラットフォームのアーキテクチャ

4.1 実行トレース収集処理の共通化

2 節でも述べたように, 実行トレースに記録すべき内容は動的解析の手法によって異なる. さまざまな動的解析の手法で利用できるよう実行トレースの収集処理を共通化 (要件 R2) するためには, できる限り詳細な実行時情報を収集するよう収集処理を設計する必要がある. Java 仮想マシン (以下, **JVM** と呼ぶ) 上では, クラスのロードとアンロード, スレッドの開始と終了, メソッド実行の開始と終了, 基本ブロック間の遷移, 命令の実行などさまざまなタイミングでイベントが発生するが, より詳細な実行トレースを収集するためには, より頻繁に実行時情報を収集する必要がある. しかしながら, 実行時情報の収集は解析対象プログラムの実行に少なからぬ負荷を与えるため, 一般に要件 R2 と R5 はトレードオフの関係になる.

実行時情報の収集は, 解析対象プログラムをデバッグ実行している JVM との間の通信を介して行う方法と, 解析対象プログラムのソースコードもしくはバイトコードに実行時情報を収集するコードを埋め込んで行う方法があるが, 前者の方法は解析対象プログラムの実行速度を大幅に低下させるため, 本プラットフォームでは後者の方法として, Javassist を用いる. 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ため, 広い範囲の動的解析手法に応用できることが期待できる.

4.2 Eclipse との統合

統合開発環境 Eclipse 上で動作するように, 本プラットフォームを 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**動的解析プラットフォームプラグイン** (以下, プラットフォームプラグインと略) 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 R4 を以下のような形で実現することを考える.

(1) Eclipse のワークスペース上にある任意の Java プロジェクト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ができる. (以下, この操作を**インストゥルメンテーション**と呼ぶ.)

(2) インストゥルメンテーションされた Java プログラムを起動し, 実行時情報を収集することができる.

(3) インストゥルメンテーションされた Java プログラムをデバッグ実行し, 実行中の任意の時点 (デバッガ上で一時停止している状態も含む) において, それまでに収集された実行時情報に対するオンライン解析を行うことができる. 解析後も, 実行および実行時情報の収集は継続される.

(4) オンライン解析にあたって, 現在デバッガ上で停止している場所 (スレッド, メソッド, 行番号など) の情報を利用することができる.

(5) オンラインおよびオフライン解析の結果を Eclipse の画面上に表示することができる.

以上の仕様を実現する上での前提知識として, Eclipse における Java プログラムの実行およびデバッグの仕組みについて解説する. Eclipse は起動後 JVM インスタンス上で実行されるが, Eclipse から Java プログラムを起動すると, そのプログラムは新しく起動した別の JVM インスタンス上で実行される. Java プログラムをデバッグモードで起動し

© 2019 Information Processing Society of Japan

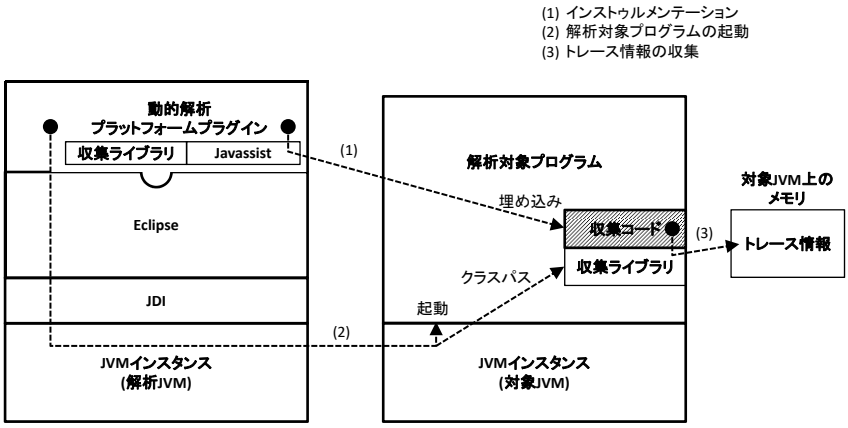

図 1　動的解析プラットフォームの基本アーキテクチャ

た場合は, JDI(Java Debug Interface) を通じて, Eclipse を実行している JVM インスタンスとデバッグ中のプログラムを実行している JVM インスタンスの間で通信が行われ, Eclipse 側からの制御が行われる (図 1 参照). ここで, JDI を介した通信は両方の JVM インスタンスの実行に負荷を与えるので注意が必要である. 以下では, Eclipse を実行している JVM インスタンスを**解析 JVM**, 解析対象プログラムを実行している JVM インスタンスを**対象 JVM** と呼ぶ.

Javassist を用いたインストゥルメンテーションには, クラスファイル中のバイトコードを直接書き換える方法と, クラスロード時にバイトコードを書き換えながらメモリ中に読み込む方法があるが, 本プラットフォームでは汎用性を考慮して, クラスファイルを直接書き換えるインストゥルメンテーション法を採用する (図 1 の (1) 参照). インストゥルメンテーションが行われたプログラムは対象 JVM 上で起動し, 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対象 JVM 上で収集される. このとき, 実行時イベント間で共通する処理を収集ライブラリとしてまとめておく (図 1 の (3) 参照). 収集ライブラリはプラットフォームプラグイン内に含まれているが, 対象 JVM 上で呼び出すことができるように, 解析対象プログラムの起動時にクラスパスとして指定する (図 1 の (2) 参照).

4.3　オンラインおよびオフライン解析処理

オフライン解析の実現は比較的容易である. 対象 JVM 上で収集した情報をトレースファイルに書き出すようにすれば良い. ただし本プラットフォームでは, 要件 R5 を満たすよう, 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, 対象プログラムの実行終了時にその情報をまとめてファイルに書き出すように工夫している. 一方, R5 を満たすようオンライン解析を実現するには, “対象 JVM 上で収集されたトレース情報を, どのようにして解析 JVM 側に伝えるか?” について考慮する必要がある. なぜなら, 最終的にオンライン解析の結果は Eclipse 上, すなわち解析 JVM 側で表示されるためである. 2 つの JVM 間でトレース情報を転送する方法として, 以下の 2 通りが考えられる.

(1) 対象 JVM 上で実行時情報が収集される度に, JDI を通じて対象 JVM 側から解析 JVM 側に収集された情報を転送する.
(2) 対象 JVM 上でトレース情報を蓄積し, オンライン解析を行う時点で, 解析に必要なトレース情報もしくは解析結果のみを JDI 経由で解析 JVM 側に転送する.

上記 (1) の方法は, データの転送頻度が高くなり転送されるデータ量も膨大になることから, 対象 JVM の実行に大きな負荷を与えるため, R5 に適合しない. 一方, (2) の方法を用いると, (1) の方法と比較して明らかにデータの転送頻度が低くなり, 転送量の削減も図ることができる. そこで本プラットフォームでは, (2) の方法を採用する. (2) の方法の採用にあたって, 対象 JVM 側にどのような情報をいつまで保持するか, 解析 JVM 側にどのような情報を転送するかについては以下で議論する.

まず, 解析 JVM 側に転送する情報について考える. 要件 R6 を考えると, トレース情報の解析処理をすべて対象 JVM 上で行い, 解析結果のみを解析 JVM 側に転送する方が転送量が少なく明らかに効率的である. しかしながらそうした場合, 対象 JVM 上で実行される解析処理の内容を動的解析の手法に応じて変える必要があり, そのことが要件 R3 の成立に与える影響について考えなければならない. まず要件 R3 を満たすため, 本プラットフォームでは, プラットフォームプラグインに対して, 動的解析の手法に応じた固有のプラグインを追加するものとする. 以下では, このプラグインを**動的解析拡張プラグイン** (以下, 拡張プラグインと略) と呼ぶ (図 2 参照). 次に, 解析処理を対象 JVM 上で実行させる方法について, 以下の 2 つを考える.

(1) インストゥルメンテーション時に, 収集処理に加えて解析処理を行うコードも対象プログラム中に埋め込む

© 2019 Information Processing Society of Japan

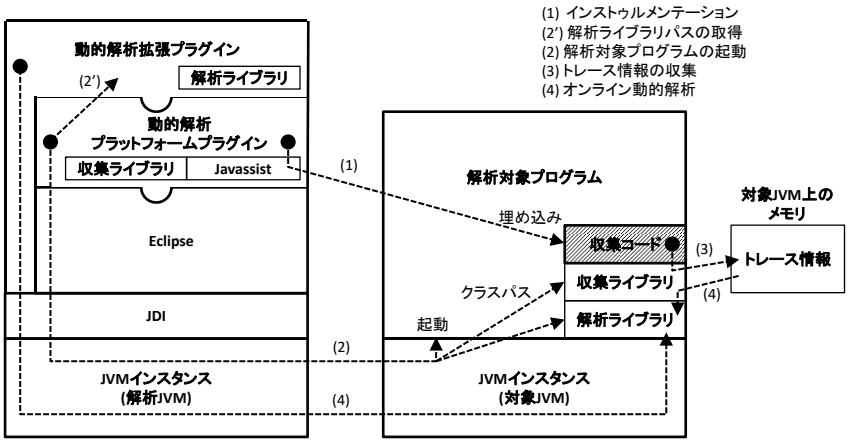

図 2　動的解析システム全体のアーキテクチャ

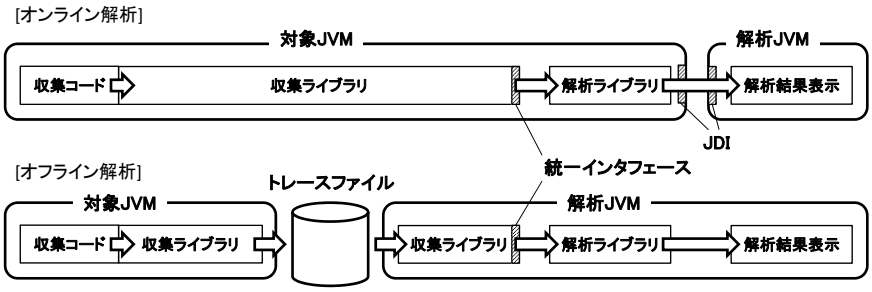

図 3　オンライン解析およびオフライン解析の処理とデータの流れ

方法.

(2) 対象 JVM の起動時に, クラスパスとして解析処理のライブラリを指定する方法.

1 つめの方法を採用した場合, 要件 R3 を満たすことは容易ではない. なぜなら, 動的解析の手法に依存した解析コードを, プラットフォームプラグイン内部のインストゥルメンテーション処理 (図 2 の (1) 参照) の中で埋め込まなければならないためである. そこで, 本プラットフォームでは, 上記 (2) の方法を採用する.

次に, 対象 JVM 側にどのような情報をいつまで保持するかについて考える. 上で述べたようにオンライン解析において, 対象 JVM 上に読み込まれた解析ライブラリは, メモリ中に蓄積されたトレース情報を元に解析処理を行った後, 解析結果のみを解析 JVM 側に返す. このとき, 解析処理で用いたトレース情報が以降の解析において必要とされない場合は, 解析処理終了後直ちにそれらの情報を破棄する方が, メモリの使用量を抑制するうえでは望ましい. しかしながら, 後の解析処理のために解析後のトレース情報を保持しておく必要があるか否かは, オンライン解析のアルゴリズムによって異なる. そのため, 本プラットフォーム単体ではトレース情報の破棄を行わず, 対象 JVM 上に読み込まれた解析ライブラリの内部で, 必要に応じて解析処理終了後にトレース情報の破棄を行う.

4.4　実行トレースへのアクセス

オンライン解析では対象 JVM 上でトレース情報の収集および解析が行われる. このとき図 2 で示したように, トレース情報の収集には収集ライブラリが, 解析には解析ライブラリが利用される. 収集ライブラリによってメモリ中に蓄積されたトレース情報への解析ライブラリからのアクセスは, 収集ライブラリを経由して行われる (図 2 の (4) および図 3 参照). 一方オフライン解析では, 収集ライブラリによって蓄積されたトレース情報が JSON 形式に変換されてトレースファイルに出力され, 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, 解析ライブラリからのアクセスが行われる (図 3 参照).

ここで要件 R7 を考えると, 解析ライブラリはオンライン解析とオフライン解析の間で共通化できることが望ましく, さらに解析ライブラリからは, トレース情報がメモリ中に蓄積されたのかトレースファイルから読み込まれたのかによらず, 同一のインタフェースでトレース情報にアクセスできることが望ましい. そこで収集ライブラリが, トレース情報にアクセスするための統一インタフェースを提供する. オンライン解析およびオフライン解析の処理およびデータの流れを図 3 にまとめる.

なお, トレース情報にアクセスするインタフェースを統一することによって, オフライン解析ツールの解析アルゴリズムを, その実装をほとんど変えることなく, オンライン

© 2019 Information Processing Society of Japan

解析ツールの解析アルゴリズムとして再利用できる可能性がある. ただし, 複数のトレースファイルを利用するようなオフライン解析や, トレース情報を後の実行で使用することを前提としたオフライン解析などは, 本質的にオンライン解析化することができない.

5. オブジェクトフロー解析への応用

本プラットフォームの有効性を確認するために, プラットフォームプラグインを拡張してオブジェクトフロー解析 [8] を行う拡張プラグインの開発を行った. 具体的には, オンライン解析版のオブジェクトフロー解析プラグインと, オフライン解析版のオブジェクトフロー解析プラグインの開発を行い, 実装をどの程度共通化できるかを調査した.

オブジェクトフロー解析とは, 指定した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きたかを, 実行とは逆向きに追跡する解析手法である. 文献 [8] では, オブジェクトフロー解析で必要となる実行時情報を残すため, VM に対してオブジェクトフロー解析固有の改変を行っているが, VM に同様の改変を行うことは個々の動的解析に依存しないという本プラットフォームの目的には一致しない. そのため, 本プラグインでは本プラットフォームで収集された一般的な実行トレースを元にオブジェクトフローを追跡することを考える. ただし, 本プラットフォームでは変数への代入をイベントとして検出できないため, オブジェクトフローを確実に追跡することはできない. そこで本プラグインでは, オブジェクト ID が演算によって変更されないことを利用して, 以下のようにオブジェクトフローの推定を行う. まず, あるメソッド実行 m の中でオブジェクト o への参照が出現しているとする. このとき, o への参照は m から呼び出されたメソッド m' から戻り値として返されたか, 別のオブジェクト o' のフィールドから取得されたか, ある配列の要素から取得されたか, m の引数として渡されたか, m の中で o の生成と同時に作られたかのいずれかによって m にもたらされたはずである. 本プラットフォームでは, これらすべてのイベントの情報を収集できるため, それらの情報に含まれるすべてのオブジェクト ID を調べることで, o への参照の由来を見つけることができる. 特定のオブジェクトへの参照の由来を再帰的に追跡することによってオブジェクトフローを推定することができる. このとき, 由来の候補が複数存在した場合に, 複数のオブジェクトフローが推定され得るが, 実際のオブジェクトフローは必ずその中に含まれていることに注意されたい. なお, 文献 [8] のオブジェクトフロー解析のもう 1 つの機能であるオブジェクト状態の更新履歴の追跡は, 実装自体は可能だが今回は対象外とした.

本研究で開発したオブジェクトフロー解析プラグインの画面は, オンライン解析版とオフライン解析版のいずれの場合も, 4 つのビュー (Java エディタ, コールスタックビュー, エイリアスビュー, オブジェクトフロービュー) によって構成される. Java エディタは, Eclipse 標準のものをそのまま利用する. エイリアスビューでは, ユーザによって実行トレース中から選択されたメソッド実行の中で出現したすべてのオブジェクト参照を表示する. ユーザはビューに表示されたそれらのオブジェクト参照を右クリックすることで, そのオブジェクト参照の出現を起点としたオブジェクトフロー解析を実行することができる. オブジェクトフロービューでは, 解析の結果として, オブジェクトフローを構成するすべてのオブジェクト参照の出現をプログラム実行の時系列の逆順に表示する. コールスタックビューでは, エイリアスビューで選択されているメソッド実行をトップとするコールスタックを表示する. これらのビューは, オンライン解析版とオフライン解析版の両方で基本的には共通の役割を果たすが, 実装を完全に共通化することはできない. その理由の一つは, オンライン解析とオフライン解析とでは解析のための入力が異なり, ユーザインタフェースを変える必要があるためである. また, オンライン解析版では, 解析結果を解析 JVM 側に転送するために JDI によるプロセス間通信が必要になる.

以上のことを踏まえたうえで, それぞれの拡張プラグインの開発を行った. その結果, オブジェクトフロー解析のアルゴリズムそのものを実装するクラスを, オンライン解析版とオフライン解析版とで完全に共通化することができた. また, 拡張プラグイン全体で見ると, ユーザインタフェースとプロセス間通信に関わる部分以外の実装をすべて共通化することができた.

6. アーキテクチャの評価

6.1 実行トレース収集処理の共通化の評価

4.1 節で述べたように, 要件 R2 と R5 を同時に満たすよう, 本プラットフォームでは実行時情報の収集に, Javassist を用いて対象プログラムに収集コードを埋め込む方法を採用した. 本節では, この選択により R5 が満たされているか否かを検証するため, 対象プログラムをそのままデバッグ実行した場合と, インストゥルメンテーションを行った後にプラットフォームプラグインが提供するコマンドでデバッグ実行した場合とで, それぞれの実行時間の比較を行う. 計測に用いた対象プログラムは, 文献 [8] の例題を参考に作成した小規模プログラム Sample, JHotDraw Ver. 7.6[*3], ArgoUML Ver. 0.34[*4], jEdit Ver. 4.3[*5]の 4 つである.

表 1 にそれぞれの計測結果を示す. これらの値は, 初回の実行を除いて 10 回実行した結果の平均値である. JHotDraw, ArgoUML, jEdit については起動が完了するまでの時間を計測している. いずれの対象プログラムにおいても,

[*3] https://www.jhotdraw.org/
[*4] http://argouml.tigris.org/
[*5] http://www.jedit.org/

© 2019 Information Processing Society of Japan

表 1　インストゥルメンテーションの有無による実行時間の比較

		平均実行時間 (msec)†		
プログラム	総メソッド数	通常実行	インストゥルメンテーション後に実行	実行時間 (倍率)
Sample	26	0.604	1.929	3.191
JHotDraw	5885	993.302	1403.862	1.413
ArgoUML	10201	3915.642	4692.090	1.198
jEdit	5506	2453.021	6302.199	2.569

† Intel(R) Xeon(R) CPU E5-1603 v4 @ 2.80GHz, メモリ 32.0GB,
Java(TM) SE Runtime Environment (build 1.8.0_181-b13) で計測

インストゥルメンテーションを行った場合に実行速度が低下することが確認できる. 4 つの対象プログラムの計測結果を比較すると, Sample がインストゥルメンテーションを行った後に最も大きな割合で実行速度が低下していることがわかる. 他の対象プログラムの実行速度があまり低下しなかった理由は, これらのプログラムにおいて, インストゥルメンテーション時に収集コードを埋め込むことができない Java の標準クラスや外部ライブラリが多く利用されており, 収集コードを埋め込めない部分の割合が高くなったことによるものと考えられる. 一般に実行トレースの収集を行う場合, 収集コードの実行やメモリの消費などにより解析対象プログラムの実行速度は著しく低下する. 具体的には, 収集コードの実行頻度が高いほど速度低下は大きくなり, 解析対象プログラムの性質や, インストゥルメンテーションに用いたツールの種類, ファイル出力を伴うか否か等の収集コードの処理内容, インストゥルメンテーションの対象となるコンポーネントの範囲の大小なども, 速度低下に影響を与える要因となる. 本プラットフォームでは, メソッドおよびコンストラクタ実行の開始および終了時, フィールドの更新や参照時, 基本ブロック間の制御の移動, 配列の生成, 配列要素の更新および参照時に収集コードが実行され, 実行時間は表 1 からもわかる通り, 実用規模のプログラムに対しては 1.2〜2.6 倍, 最大で 3.2 倍となっている. これらの速度低下は, 本手法より実行時情報の収集頻度が低い動的解析手法 [4], [6], [9], [12] の結果 (1.27〜3 倍) と比較しても大きく異ならないため, 要件 R5 に関しては満たされているといえる.

6.2　オンライン解析処理の設計の評価

4.3 節で述べたように, 要件 R5 と R6 を満たすため本プラットフォームでは, 解析処理をすべて対象 JVM 上で行うよう設計した. 本節では, この選択により R6 が満たされているか否かを検証するため, 4.3 節の設計を採用した場合と採用しなかった場合とで, オンライン解析の実行時間がどの程度変わるかを調べる. 具体的には, 対象プログラムに対する解析処理を対象 JVM 上で行う場合のオンライン解析と, 解析 JVM 上で行う場合のオンライン解析の実行時間の比較を行う. 実行時間の計測にあたっては, 指定したメソッドが現在までに何回実行されたかを解析する簡単な拡張プラグインを利用し, 対象 JVM 上で実行される解析ライブラリと解析 JVM 上で実行される解析ライブラリを用いて, それぞれについて解析を開始する直前から終了までを計測した. 計測に用いた対象プログラムは, 前節と同様 Sample, JHotDraw, ArgoUML, jEdit の 4 つである.

表 2 にそれぞれの計測結果を示す. 表 2 の値は, 初回の実行を除いて 10 回実行した結果の平均値である. いずれの対象プログラムにおいても, 解析処理を対象 JVM 上で行う場合の方が解析 JVM 上で行う場合よりも顕著に高速であった. これは, 解析 JVM と対象 JVM との間での JDI を利用した通信の回数が大きく異なるためであると考えられる. 解析処理を対象 JVM 上で行う場合, 解析 JVM は対象 JVM 上にある解析処理を JDI 経由で 1 回呼び出すが, その後は対象 JVM 上でメモリ中に蓄積された実行トレースの解析処理がすべて行われ, 解析 JVM はその結果を受け取るだけでよい. そのため, 解析 JVM と対象 JVM との間の通信は最初の 1 回だけとなる. 一方で, すべての解析処理を解析 JVM 上で行う場合, 対象 JVM 上の実行トレースを取得しようとする度に JDI を利用した通信が発生することになる. 一般に JVM 内での処理と比較して JVM 間の通信には遥かに多くの計算コストが費やされるため, 解析処理を対象 JVM 上で行い通信回数を削減する方が, より短い実行時間で解析処理を完了することができると考えられる.

また, 図 4 からわかるように, Sample のみ速度向上が約 10 倍程にとどまっているが, これは Sample が他の 3 つと比較して非常に小規模であるために, JDI を利用した通信を 1 回以上行う際に発生するオーバーヘッドの影響が相対的に大きくなったためと考えられる. 実用的な規模のプログラムを対象とする場合, 解析 JVM と対象 JVM との間での通信量が飛躍的に増大し, オーバーヘッドの影響が相対的に無視できるようになるため, 実際には他の 3 つのプログラムと同様 1000 倍以上の高速化が見込まれると考えられる. これらのことから, 解析処理を対象 JVM 上で行う本プラットフォームの設計によって速度低下を十分に抑えられているといえ, 要件 R6 は満たされているといえる.

7.　おわりに

さまざまな動的解析ツールの開発の支援を目的として,

© 2019 Information Processing Society of Japan

表 2　解析処理を実行する JVM による実行時間の比較

		平均実行時間 (sec)†		
プログラム	総メソッド数	対象 JVM 上での解析	解析 JVM 上での解析	速度向上 (倍率)
Sample	26	0.002	0.021	13.258
JHotDraw	5885	0.012	15.250	1223.490
ArgoUML	10201	0.012	26.493	2302.545
jEdit	5506	0.025	37.010	1500.354

† Intel(R) Xeon(R) CPU E5-1603 v4 @ 2.80GHz, メモリ 32.0GB, Java(TM) SE Runtime Environment (build 1.8.0_181-b13) で計測

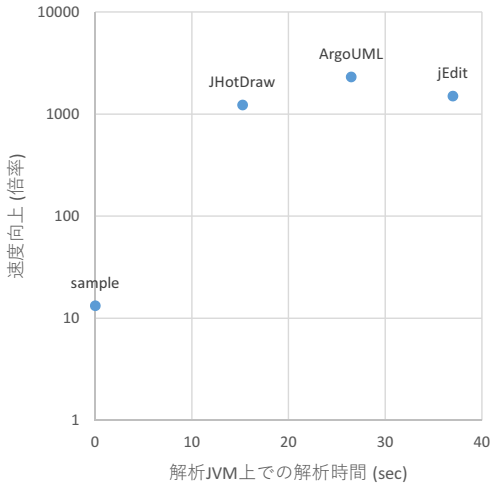

図 4　解析処理を対象 JVM 上で実行することによる速度向上

Eclipse 上で動作する効率の良い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を行った．本プラットフォームは実行トレースの収集処理を共通化し, Eclipse 上でオンラインおよびオフライン解析を行うための共通基盤を提供することによってツール開発の効率化を目指している．本プラットフォームのトレース収集時の実行速度およびトレース解析時の実行速度の評価を行った．その結果, トレース収集時は通常の実行時に対して最大で 3 分の 1 程度しか速度が低下せず, またオンライントレース解析時は本プラットフォームで採用した設計にしたがうことによって 1000 倍以上の高速化ができることを確認することができた．今後本プラットフォームを, 非対称スライス [9] のオンライン化やデルタ抽出 [10] に基づく逆戻りデバッガなど, Java を対象としたさまざまな動的解析ツールの実装に適用していくことを考えている．

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. of the Conference on Programming Language Design and Implementation*, pp. 246–256 (1990).
[2] ボーリス・バイザー: ソフトウェアテスト技法, 日経 BP 出版センター (1994).
[3] Binder, R. V.: *Testing object-oriented systems: Models, patterns, and tools*, Addison-Wesley Professional (1999).
[4] Briand, L. C., Labiche, Y. and Leduc, J.: 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).
[5] Dwyer, M. B., Kinneer, A. and Elbaum, S.: Adaptive online program analysis, *Proc. of the 29th ACM/IEEE International Conference on Software Engineering*, pp. 220–229 (2007).
[6] Heydarnoori, A., Czarnecki, K. and Bartolomei, T.: Supporting framework use via automatically extracted concept-implementation templates, *Proc. of the 23nd European Conf. on Object-Oriented Programming*, pp. 344–368 (2009).
[7] Jones, J. A., Harrold, M. J. and Stasko, J.: Visualization of test information to assist fault localization, *Proc. of the 24th ACM/IEEE International Conference on Software Engineering*, pp. 467–477 (2002).
[8] Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical object-oriented back-in-time debugging, *Proc. of the 22nd European Conf. on Object-Oriented Programming*, pp. 592–615 (2008).
[9] Nitta, N., Kume, I. and Takemura, Y.,: Identifying mandatory code for framework use via a single application trace, *Proc. of the 28th European Conf. on Object-Oriented Programming*, pp. 593–617 (2014).
[10] Nitta, N. and Matsuoka, T.: Delta extraction: An abstraction technique to comprehend why two objects could be related, *Proc. of the 31st International Conference on Software Maintenance and Evolution*, pp. 61–70 (2015).
[11] Park, S., Vuduc, R. W. and Harrold, M. J.: Falcon: Fault localization in concurrent programs, *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 245–254 (2010).
[12] Reiss, S. P.: Dynamic detection and visualization of software phases, *Proc. of the International Workshop on Dynamic Analysis*, pp. 1–6 (2005).
[13] Tallam, S., Tian, C. and Gupta, R.: Dynamic slicing of multithreaded programs for race detection *Proc. of the 24th IEEE International Conference on Software Maintenance*, pp. 97–106 (2008).
[14] Zhang, Z., Chan, W. K., Tse, T. H., Jiang, B. and Wang, X.: Capturing propagation of infected program states, *Proc. of the 7th Joint Meeting of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 43–52 (2009).

© 2019 Information Processing Society of Japan