[技術・研究報告] 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

著者	石谷 涼, 新田 直也
雑誌名	甲南大学紀要. 知能情報学編
巻	13
号	2
ページ	163-185
発行年	2021-02-10
URL	http://doi.org/10.14990/00003696

技術・研究報告

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

石谷涼[a], 新田直也[b]

[a] 甲南大学大学院 自然科学研究科 知能情報学専攻
神戸市東灘区岡本８―９―１, 658-8501
[b] 甲南大学 知能情報学部 知能情報学科
神戸市東灘区岡本８―９―１, 658-8501

（受理日 2020年11月16日）

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．本論文では, 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を開発したので報告する．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時情報の収集処理を共通化し, さらに収集された実行時情報を解析するための共通基盤を Eclipse 上で提供することによって開発の効率化を図る．本プラットフォームが提供する収集処理は, 幅広い動的解析手法で利用できるよう詳細な実行時情報を効率良く収集できるという特長を持つ．またオンライン解析の共通基盤としては, 解析アルゴリズムをオンライン化することによって生じるオーバーヘッドをできる限り抑えるよう設計上の工夫を行っている．本論文ではその設計を採用した場合と採用しない場合の解析速度の比較を行い, 設計の妥当性について評価を行ったので報告する．

キーワード: 動的解析, オンライン解析, Eclipse

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 [1]-[3], コードカバレッジ分析 [4], [5], 動的プログラムスライシング [6], [7] 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** [8] 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.

本論文では, 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. 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].

以降の論文の構成は次の通りである. 2 章でプログラム解析についての一般的な解説を行う. 3 章で本プラットフォームの開発目的および要件を説明し, 4 章でそれらを満たすアーキテクチャについて議論する. 5 章で本プラットフォームの利用例について述べ, 6 章で本プラットフォームの実装が 3 章で述べた非機能的要件を満たしているか否かを評価する実験を紹介する. 最後に 7 章で本論文のまとめを述べる.

[1] http://www.javassist.org/
[2] https://github.com/nitta-lab/org.ntlab.traceAnalysisPlatform

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. 静的解析は, 対象となるプログラムの実行環境を構築しなくても解析することができるという特長を持ち, プログラムの特定の実行に依存せずあらゆる可能な実行において成り立つ性質を調べることに適しているが, 実際には起こりえない実行まで想定してしまうという問題がある. 一方動的解析は, 実際に行ったプログラムの実行に関する情報が得られるが, それ以外の実行に関する情報は得られない. また, 実際に動作可能なプログラムがなければ解析することができないという問題もある.

本論文では, 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汎用の動的解析プラットフォームの紹介を行う. 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** [8] 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

前章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: 実行トレースの収集に伴う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-R4 が本プラットフォームの機能的要件に相当し, R5-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. 詳細は次章で述べるが, 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. 次章で, 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詳細に説明する.

4 動的解析プラットフォームのアーキテクチャ

4.1 Javassist の利用

2 章でも述べたように, 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. 前者のツールの例としては, Reticella [9] が挙げられる. ただし, 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の利用には適さない. そこで後者の方法の間で, 収集できる情報の種類および対応可能な動的解析の手法の比較を行う. 後者の方法で使用可能なツールとして, ASM, BCEL, Javassist, AspectJ などが挙げられる. ASM または BCEL を利用すると, バイトコードを直接書き換えることができるため, 任意の詳細度で実行時情報を収集できる反面, 利用にあたってはバイトコードの知識が必要となる. 一方, Javassist および AspectJ はより高い抽象度でバイトコードを書き換えることができるが, 収集できる実行時情報の詳細度は低下する[3]. 収集可能な情報は, 詳細度が高い順に ASM および BCEL, Javassist, AspectJ となる. AspectJ 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イベントに関する情報を取得することができる. Javassist ではこれらに加えて, 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. これらの書き換えツールとそれらによって対応可能な動的解析の手法を表 1 にまとめる.

表 1 中の動的解析手法とその分類には, 文献 [27] に若干の拡張を加えたものを用いている. オンライン解析およびオフライン解析については, 4.4 節で説明する. 表 1 から分かるように, 要件 R2 を満たすことだけを考えれば, ASM もしくは BCEL を用いるのが最も適切である. しかしながら, 要件 R5 および以下に列挙する観点を考慮して, 本プラットフォームでは Javassist を利用することとした.

- ASM もしくは BCEL を利用して詳細な実行時情報を収集できるようにするためには多大な開発工数が必要になると考えられる.
- ASM や BCEL の利用にはバイトコードの知識が必要とされるため, 開発だけではなく保守作業に必要な学習コストも高くなってしまう.
- JVM のバージョンアップへの対応が困難になることが予想される.

[3]ただし, Javassist はバイトコードレベル API を使用するとバイトコードを直接書き換えることができる.

表 1: 書き換えツールと対応可能な動的解析手法

カテゴリ	動的解析手法の名称	書き換えツール			オンライン解析 /オフライン解析
		ASM および BCEL	Javassist†	AspectJ	
実行経路の分析	コードカバレッジの計算 [4], [5]	○	△‡	△§	オフライン (オンラインも可)
	振舞いの可視化 [10], [11], [12], [13]	○	○	○	オフライン (オンラインも可)
	フェイズ検出 [14], [15]	○	○	○	オンライン
欠陥の分析	統計的デバッギング [1], [2], [3]	○	○	○	オフライン
	動的スライシング [6], [7]	○	×	×	オフライン (オンラインも可)
	Capture & Replay[16]	○	○	○	オフライン
オブジェクトの追跡	Object Flow[17]	○	△¶	×	オンライン
	デルタ抽出 [18]	○	△‖	×	オフライン (オンラインも可)
振舞い仕様のマイニング	プロトコルマイニング [19], [20]	○	○	○	オンライン
	動的不変条件の検出 [21]	○	×	×	オフライン (オンラインも可)
	Live Sequence Chart の抽出 [22]	○	○	○	オフライン (オンラインも可)
機能捜索	ソフトウェア偵察 [23], [24]	○	○	△**	オフライン
	フレームワーク利用例抽出 [25], [26]	○	△††	△‡‡	オフライン (オンラインも可)

†: ただし, バイトコードレベル API を使用しないものとする.
‡: 行単位のカバレッジを計算できない.
§: 行単位および基本ブロック単位のカバレッジを計算できない.
¶: メソッド内のフローを追跡できない.
‖: メソッド内のフローを追跡できない.
**: 基本ブロック単位のソフトウェア偵察ができない.
††: メソッド内のフローを追跡できない.
‡‡: メソッド内のフローを追跡できない.

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**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うえ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).

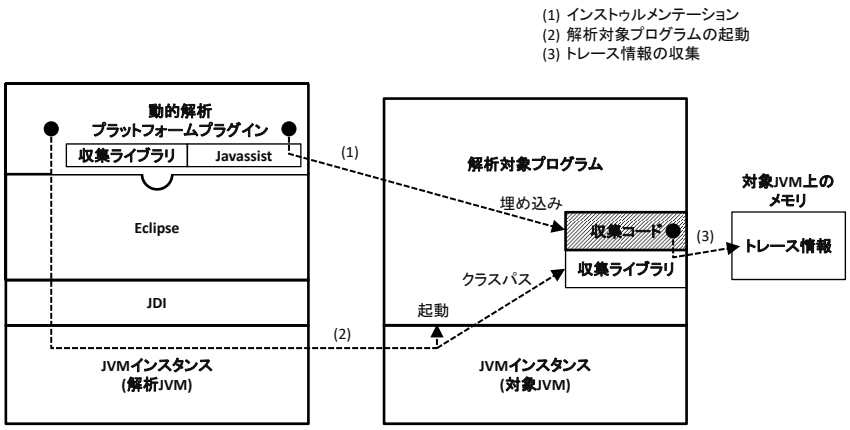

図 1: 動的解析プラットフォームの基本アーキテクチャ

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**, 解析対象の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. しかしながら後者の方法は, 対象プログラムを起動する際に, バイトコードの書き換え処理を行う専用のクラスローダを指定する必要があり, 起動時に独自のクラスローダを指定するようなプログラム (例えば jEdit) に対して適用することができない. そこで, 本プラットフォームでは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). 収集ライブラリは動的解析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. そのため, 解析対象プログラムの実行は, 動的解析プラットフォームプラグインが提供する独自の実行コマンドによって行う (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. 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, 動的解析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**と呼ぶ (図 2 参照).

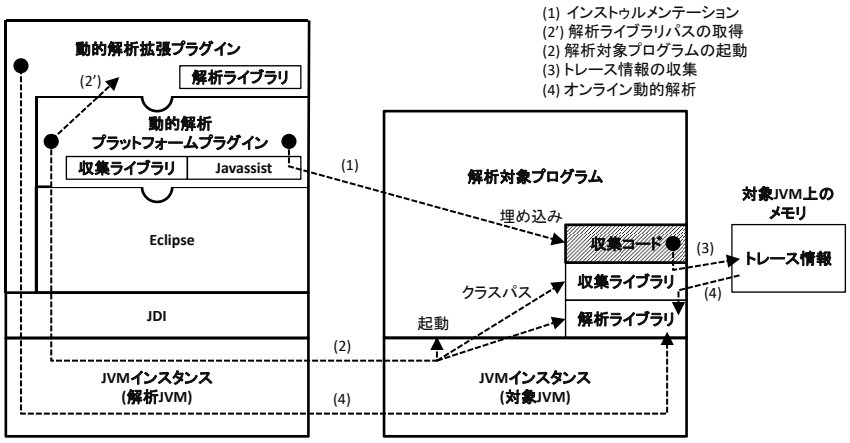

図 2: 動的解析システム全体のアーキテクチャ

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 つ目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, 動的解析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. 一方 2 つ目の方法を採用した場合でも, 要件 R3 を満たすために考慮しなければならないことがある. 動的解析プラットフォームプラグイン内部で行う対象 JVM の起動処理 (図 2 の (2) 参照) において, 動的解析の手法によって異なる解析ライブラリのパスを, 起動時のクラスパスとして如何に指定するかである. しかしながら, この問題は Eclipse プラグインの**拡張ポイント**を用いて比較的容易に解決できる. 拡張ポイントとは, Eclipse のプラグイン A にプラグイン B を追加する際に, A 側のメソッドから B 側のメソッドを呼び出せるようにする仕組みで, 具体的には A 側に拡張ポイントを定義しておき, その拡張ポイントを B 側のメソッドで拡張することによって実現される. 動的解析プラットフォームプラグインに, 解析ライブラリのパスを取得するための拡張ポイントを定義しておき, その拡張ポイントを動的解析拡張プラグインのメソッドが拡張して内部の解析ライブラリのパスを返すようにすればよい (図 2 の (2') 参照). そうすることによって, 動的解析の手法に応じた解析ライブラリのパスを対象 JVM の起動時に指定できるようになり, R3 も満たされる.

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 (フォーマットの概要は付録参照) に出力され, 解析時にはトレースファイルからメモリ中に復元されたトレース情報に対して解析が行われる (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. オフライン解析は対象プログラムの実行終了後に実行されるため, 実行にあたっては動的解析拡張プラグインが内包している解析ライブラリが呼び出される. このとき, トレースファイルは動的解析プラットフォームプラグインが内包している収集ライブラリの中で読み込まれ, 統一インタフェースを経由して解析ライブラリに復元されたトレース情報が提供される. 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.

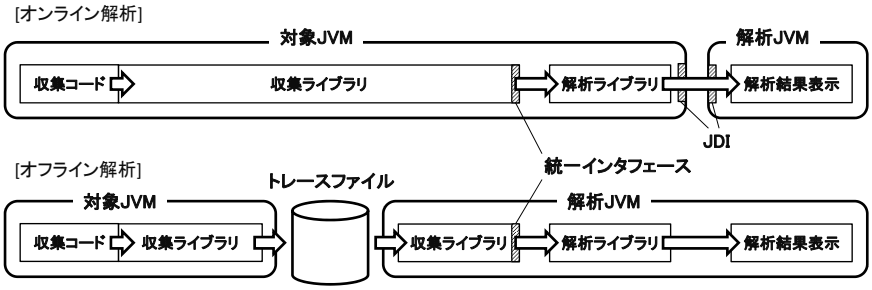

図 3: オンライン解析およびオフライン解析の処理とデータの流れ

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. そこで, それぞれの動的解析手法が元々オンライン解析であるのかオフライン解析であるのか, オフライン解析であるならオンライン解析化することができるのかについて, 表 1 に記載する.

5 利用例

5.1 SampleAnalyzer

動的解析拡張プラグインの一例として SampleAnalyzer[4]を紹介する. SampleAnalyzer は, デバッガによって現在停止している場所のメソッドがプログラムの起動時から現在までに何回呼び出されたかをそれまでに蓄積された実行トレースを元にオンラインで計算する Eclipse プラグインで, 動的解析プラットフォームプラグイン `org.ntlab.traceAnalysisPlatform` に追加するプラグインとして開発を行った.

SampleAnalyzer では, 次のように解析が実行される.

1. デバッグ実行しているインストゥルメンテーション済みの対象プログラムがブレークポイントで停止しているときに, Eclipse のツールバー上に配置されたボタンを押下すると, 解析が始まる.
2. 当該メソッドの呼び出し回数が, 実行トレースに記録されている範囲内での最初のメソッド実行を起点としたメソッド呼び出し木の深さ優先探索によって求められる.
3. 解析結果が Eclipse のダイアログ上に表示される.

SampleAnalyzer には, 対象 JVM 上で実行される通常の解析ライブラリと, 実行速度の比較用に作成した解析 JVM 上で実行される解析ライブラリが含まれている. 解析 JVM 上で実行される解析では, トレース情報が対象 JVM 上に蓄積されているため, 呼び出し木中の新しいノードを訪問する度に, JDI を経由したプロセス間通信でノードに関する情報が取得される.

4.3 節で述べたとおり, SampleAnalyzer は動的解析プラットフォームプラグインが提供する拡張ポイントを拡張して実装している. 図 4 に SampleAnalyzer のプラグイン定義ファイルを示す.

```
:
<plugin>
   <extension
          point="org.ntlab.traceAnalysisPlatform.additionalClasspaths">
       <classpath
             class="org.ntlab.sampleanalyzer.analyzerProvider.SampleAnalyzerLaunchConfiguration">
      </classpath>
   </extension>
    :
</plugin>
```

図 4: SampleAnalyzer のプラグイン定義ファイル (plugin.xml)

動的解析プラットフォームプラグインが提供している拡張ポイントは `org.ntlab.traceAnalysisPlatform.additionalClasspaths` である. 図 4 はそれを拡張したクラスが `SampleAnalyzerLaunchConfiguration` クラスであることを示している. 図 5 は `SampleAnalyzerLaunchConfiguration` クラスのソースコードの一部である.

[4]https://github.com/nitta-lab/org.ntlab.sampleAnalyzer

```
public class SampleAnalyzerLaunchConfiguration implements IAdditionalLaunchConfiguration {
   public static final String ANALYZER_PATH
                              = "org/ntlab/sampleanalyzer/analyzerProvider/SampleAnalyzer.class";
      :
   public String[] getAdditionalClasspaths() {
      :
      List<String> classPathList = new ArrayList<>();
      String analyzerClassPath = FileLocator.resolve(this.getClass().getClassLoader()
                                                     .getResource(ANALYZER_PATH)).getPath();
      :
      return classPathList.toArray(new String[classPathList.size()]);
      :
   }
}
```

図 5: SampleAnalyzerLaunchConfiguration クラス

このクラスは動的解析プラットフォームプラグイン内に含まれる `IAdditionalLaunchConfiguration` インタフェースを実装しており, `getAdditionalClasspaths()` メソッドによって SampleAnalyzer 内部の解析ライブラリのパスを動的解析プラットフォームプラグインに返す.

5.2 研究における今後の応用

本プラットフォームは Javassist を用いて実行トレースを収集しているため, 表 1 に示した Javassist が対応していない動的スライシングや動的不変条件の検出などの手法には適用することができない. 例えば動的スライシングに適用するためには, 局所変数の更新や参照, 算術演算や論理演算などのイベントを検出する必要があるが, 本プラットフォームはそれらのイベントには対応していない.

Object Flow [17] は, オブジェクトの参照がどのように渡されてきたかを実行と逆向きに追跡する技術であり, Smalltalk プログラムを対象とした実装が存在している. Java プログラムを対象とした Object Flow の実装は現時点では確認できないが, これを厳密に実装するためには, Java プログラムにおける局所変数の更新や参照などのイベントを検出する必要がある. しかしながら, Object Flow によって追跡されるオブジェクトの参照は, 算術演算や論理演算などによって値 (参照先) が変更されないため, 実用上は, メソッドやコンストラクタの呼び出しと復帰, フィールドの更新と参照, 配列の生成および配列要素の更新と参照などのイベントによるオブジェクト参照の受け渡しを追跡できれば十分であり, Javassist で対応可能である. 同様のことは, 2 つのオブジェクトがどのようにして関連付けられたかを追跡する技術であるデルタ抽出 [18] にもいえる. [18] では AspectJ を用いてデルタ抽出ツールを実装していたが, Javassist に基づいた本プラットフォームを利用することによって, 配列に格納されたオブジェクト参照も追跡できるようになり, 解析精度の向上が期待できる.

また, 本プラットフォームは Eclipse に組み込まれているため, Eclipse のデバッガと組み合わせることによって, Object Flow に基づく逆戻りデバッガを自然な形で実現することができると期待される. さらに, Eclipse が提供している抽象構文木解析などの静的解析手法と組み合わせることによって, 静的解析と動的解析のハイブリッド解析手法を容易に実装できるようになると期待される. また, Eclipse の内部には SWT や JFace といった高機能な GUI ライブラリが用意されており, 他にも GEF

(Graphical Editing Framework) のような Eclipse 上でグラフィック表示を行うためのプラグインも利用することができる. それらを用いて, 収集した実行トレースを容易に可視化することができる. 現在, トレースファイルを読み込んで, 実行トレース中の呼び出し木を折り畳み・展開表示できる拡張プラグインを開発中であり, 今後オンライン解析にも対応させていくことを予定している.

6 アーキテクチャの評価

6.1 Javassist 利用の評価

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トレースの収集コードの埋め込みに Javassist を利用した. 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動的解析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 [17] を参考にテスト用に作成した小規模プログラム Sample, JHotDraw ver. 7.6[5], ArgoUML ver. 0.34[6], jEdit ver. 4.3[7]の 4 つである. 表 2 および図 6 にそれぞれの計測結果を示す.

表 2: 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	速度低下 (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

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, 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.

[5]https://sourceforge.net/projects/jhotdraw/
[6]https://github.com/argouml-tigris-org/argouml
[7]http://www.jedit.org/

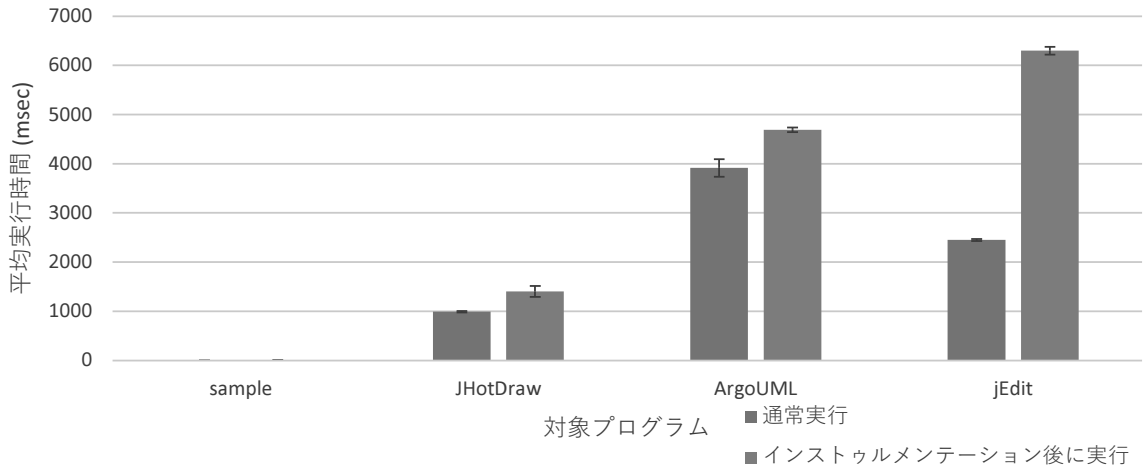

図 6: インストゥルメンテーションの有無による速度低下

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. 以下では, 他のいくつかの動的解析ツールによる速度低下の実験結果と比較することによって本プラットフォームによる速度低下の評価を行う. 比較にあたって, まず, 文献 [16] のように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を限定する手法を, 比較の公平さのため, 比較対象から除外する. そのうえで, 1) Java を対象とし, 2) ツール使用による速度低下について記載があり, かつ 3) 使用したインストゥルメンテーションツールについて記載のある, 文献 [10], [14], [25], [26], [28] のツールを比較対象とした.

まず, 文献 [10], [14], [25], のツールでは, 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に収集コードが実行されていると考えられ, 実行時間は文献 [10] で 1.27~2.39 倍, 文献 [14] で 2 倍, 文献 [25] で 2~3 倍になったことが報告されている. 次に, 文献 [26] のツールでは, AspectJ を用いることで, これらに加えてフィールドの更新や参照時にも収集コードが実行され, 実行時間が 2.8 倍になったことが報告されている. 続いて, 本プラットフォームでは, Javassist を用いることで, さらに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も収集コードが実行され, 実行時間は表 2 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. 最後に, 最も収集コードの実行頻度が高くなる文献 [28] のツールでは, BCEL を用いることで, 対象プログラムのすべての命令の実行時に収集コードが実行され, 実行時間は 4~14 倍, 最大で 93 倍になったことが報告されている.

これらの比較から, 調査したツールの範囲内では, 速度低下の割合は概ね収集コードの実行頻度の差によって決まっていることがわかる. なお, インストゥルメンテーションに用いたツールによる差を調べるため, AspectJ を用いたツールのなかで, 本プラットフォームに最も収集コードの実行頻度が近い文献 [26] のツールと, 本プラットフォームの比較を行ってみる. そうすると, 文献 [26] のツールの方が, 想定される収集コードの実行頻度が低いにもかかわらず, 速度低下の差はほぼ見られない

ことがわかる. これが, 実際の実行頻度にあまり差がなかったことを意味するのか, AspectJ と比較して Javassist の方が実行効率が良いことを意味するのかは, この比較のみでは分からない. 要件 R5 に関しては, より収集コードの実行頻度が低いと考えられる文献 [10], [14], [25], [26] の手法と比較しても, 本プラットフォームによる速度低下は大きく異ならないため, 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, 5.1 節で紹介した SampleAnalyzer を動的解析拡張プラグインとして利用して, 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 3 にそれぞれの計測結果を示す.

表 3: 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

表 3 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. 図 7 に解析処理を対象 JVM 上で実行することによる速度向上 (倍率) を示す.

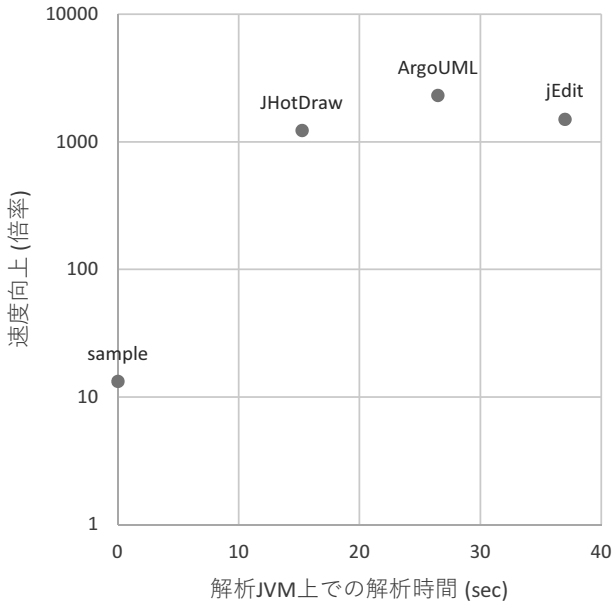

図 7: 解析処理を対象 JVM 上で実行することによる速度向上

図 7 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.

6.3 オンライン解析処理のメモリ使用量の評価

実用的なプログラムを対象に解析を行う場合, メモリ使用量も問題の 1 つとなる. 4.3 節で述べたように, 本プラットフォーム単体では, 収集されたトレース情報を破棄することはない. しかしながら, メモリの制約が厳しい場合や, 長時間に渡って継続的に解析を行う必要がある場合は, 動的解析拡張プラグイン側で解析を行う度に, 解析に用いたトレース情報を破棄することも可能である. 今回, 実験を行うにあたり, 実際に文献 [14] のようにリアルタイムでオンライン解析を行う動的解析拡張プラグインを実装した. この動的解析拡張プラグインでは, 100 [msec] 毎に, 指定したスレッド上での, 指定したメソッドの総呼び出し回数をカウントし, 結果を Eclipse のダイアログ上に表示する. 本節では, このプラグインを用いて, 解析を行う度にトレース情報を破棄した場合と, 破棄しなかった場合とで, メモリ使用状況の変化について比較を行う. 一般に, インストゥルメンテーションを行ったプログラムを実行すると, 実行開始から時間が経過する毎にメモリ使用量は増加していく. メモリ使用量が増加し続けると, やがて必要なメモリを確保することができなくなり, 解析不能状態に至る. そのため,

本実験では, 対象プログラムの実行にあたって使用する最大ヒープサイズを指定したうえで, 対象プログラムの起動直後から, 解析不能になるまでに要した時間を, 実行可能時間として計測した. なお, 実行可能時間の計測にあたっては, 順序効果をなくすため, それぞれの計測を独立して行った. 計測に用いた対象プログラムは, 本実験を行うにあたり作成した, 無限ループによって実行トレースを生成し続ける簡単なプログラムである. 表 4 にそれぞれの計測結果を示す.

表 4: 最大ヒープサイズ指定による対象プログラムの実行可能時間

	平均実行可能時間 (sec) †	
最大ヒープサイズ (MB)	トレース情報破棄なし	トレース情報破棄あり
128	8.74	57.15
256	11.83	83.14
512	19.32	172.72
1024	26.15	361.03

†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

表 4 の値は, 初回の実行を除いて 5 回実行した結果の平均値である. いずれの場合でも, 解析に用いたトレース情報を破棄する方が, トレース情報を破棄しない場合と比較して, 少ないメモリ使用量でより長時間の実行が可能になることが確認できる. また, メモリの使用状況の変化がわかるよう, 実行開始後, 表 4 における平均実行可能時間が経過したときに最大ヒープサイズのメモリを消費しているものと仮定して, メモリ使用量の時間変化をグラフ化したものを図 8 に示す.

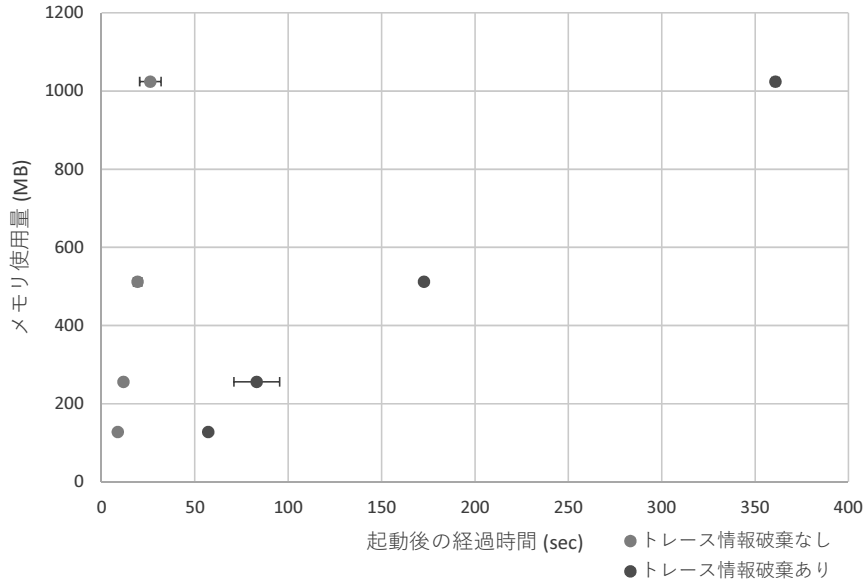

図 8: トレース情報の破棄の有無によるメモリ使用状況の変化

図 8 からわかるように, トレース情報を破棄しない場合と比較して, 経過時間に対するメモリ使用

量の増加率が大きく緩和されていることがわかる. これらのことから, 動的解析拡張プラグイン側でトレース情報を破棄することによって, オンライン解析でのメモリ使用量を大幅に削減できているといえる.

なお, トレース情報を破棄しない場合でも, 実用的なプログラムに対してオンライン解析がどの程度利用できるかを検証するため, 前節でも用いた JHotDraw, ArgoUML, jEdit の 3 つのプログラムに対して, 起動してから何も操作しないまま放置した状態でのメモリ使用状況を調べる実験を, 上記の動的解析拡張プラグインを用いて行った. その結果, JHotDraw では起動完了時に 0.3 GB 前後, jEdit では起動完了時に 1.0 GB 前後のメモリを使用し, その後それ以上のメモリの使用はほぼ見られなかった. 一方で, ArgoUML では起動完了時に 0.5 GB 前後のメモリを使用した後, 時間の経過にしたがって少しずつメモリ使用量が増加していき, 起動から約 8 分が経過した時点で 3.0 GB 前後のメモリを使用することを確認した. このことから, トレース情報を破棄しなくても, 実用的なプログラムに対して一定時間オンライン解析を実行可能であることがわかる.

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, 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の Java を対象としたさまざまな動的解析および静的解析と組み合わせたハイブリッド解析手法への応用が期待される.

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] J. A. Jones, M. J. Harrold and J. Stasko, “Visualization of test information to assist fault localization,” in *Proc. 24th ACM/IEEE International Conference on Software Engineering*, pp. 467–477, 2002.

[2] S. Park, R. W. Vuduc and M. J. Harrold, “Falcon: Fault localization in concurrent programs,” in *Proc. 32nd ACM/IEEE International Conference on Software Engineering*, pp. 245–254, 2010.

[3] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang and X. Wang, “Capturing propagation of infected program states,” in *Proc. 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.

[4] 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.

[5] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional, 1999.

[6] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *Proc. Conference on Programming Language Design and Implementation*, pp. 246–256, 1990.

[7] S. Tallam, C. Tian and R. Gupta, “Dynamic slicing of multithreaded programs for race detection,” in *Proc. 24th IEEE International Conference on Software Maintenance*, pp. 97–106, 2008.

[8] M. B. Dwyer, A. Kinneer and S. Elbaum, “Adaptive online program analysis,” in *Proc. 29th ACM/IEEE International Conference on Software Engineering*, pp. 220–229, 2007.

[9] K. Noda, T. Kobayashi, S. Yamamoto, M. Saeki and K. Agusa, “Reticella: An execution trace slicing and visualization tool based on a behavior model,” *IEICE Transactions on Information and Systems*, vol. E95–D, no. 4, pp. 959–969, 2012.

[10] 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.

[11] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides and J. Yang, “Visualizing the execution of Java programs,” *Revised Lectures on Software Visualization, International Seminar*, pp. 151–162, 2001.

[12] J. Quante and R. Koschke, “Dynamic object process graphs,” *Journal of Systems and Software*, vol. 81, pp. 481–501, 2008.

[13] T. Systa, “Understanding the behavior of Java programs,” in *Proc. 7th Working Conference on Reverse Engineering*, pp. 214–223, 2000.

[14] S. P. Reiss, “Dynamic detection and visualization of software phases,” in *Proc. International Workshop on Dynamic Analysis*, pp. 1–6, 2005.

[15] Y. Watanabe, T. Ishio and K. Inoue, “Feature-level phase detection for execution trace using object cache,” in *Proc. International Workshop on Dynamic Analysis*, pp. 8–14, 2008.

[16] S. Joshi and A. Orso, “SCARPE: A technique and tool for selective capture and replay of program executions,” in *Proc. 23rd IEEE International Conference on Software Maintenance*, pp. 234–243, 2007.

[17] A. Lienhard, T. Gîrba, and O. Nierstrasz, “Practical object-oriented back-in-time debugging,” in *Proc. 22nd European Conference on Object-Oriented Programming*, pp. 592–615, 2008.

[18] N. Nitta and T. Matsuoka, “Delta extraction: An abstraction technique to comprehend why two objects could be related,” in *Proc. 31st International Conference on Software Maintenance and Evolution*, pp. 61–70, 2015.

[19] M. Gabel and Z. Su, “Online inference and enforcement of temporal properties,” in *Proc. 32nd ACM/IEEE International Conference on Software Engineering*, pp. 15–24, 2010.

[20] D. Lo, G. Ramalingam, V. P. Ranganath and K. Vaswani, “Mining quantified temporal rules: Formalism, algorithms, and evaluation,” in *Proc. 16th Working Conference on Reverse Engineering*, pp. 62–71, 2009.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[22] D. Lo and S. Maoz, “Mining scenario-based triggers and effects,” in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 109–118, 2008.

[23] N. Wilde and M. C. Scully, “Software reconnaissance: Mapping program features to code,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[24] W. E. Wong, S. S. Gokhale, J. R. Horgan and K. S. Trivedi, “Locating program features using execution slices,” in *Proc. IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, pp. 194–203, 1999.

[25] A. Heydarnoori, K. Czarnecki and T. Bartolomei, “Supporting framework use via automatically extracted concept-implementation templates,” in *Proc. 23nd European Conference on Object-Oriented Programming*, pp. 344–368, 2009.

[26] N. Nitta, I. Kume and Y. Takemura, “Identifying mandatory code for framework use via a single application trace,” in *Proc. 28th European Conference on Object-Oriented Programming*, pp. 593–617, 2014.

[27] 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.

[28] 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, vol. 3, no. 3, pp. 1–17, 2011.

A 付録: 動的解析プラットフォームが使用するトレースファイルの JSON フォーマット

本プラットフォームが使用するトレースファイルの JSON フォーマットの概要を以下に説明する. 本プラットフォームでは, Java プログラム実行時に発生する以下のイベントに対して情報を収集する.

- クラスのロード (`"classDef"`).
- メソッド実行の開始 (`"methodEntry"`).
- メソッド実行の終了 (`"methodExit"`).
- コンストラクタ実行の開始 (`"constructorEntry"`).
- コンストラクタ実行の終了 (`"constructorExit"`).
- フィールド更新 (`"fieldSet"`).
- フィールド参照 (`"fieldGet"`).
- 配列生成 (`"arrayCreate"`).
- 配列要素更新 (`"arraySet"`).
- 配列要素参照 (`"arrayGet"`).
- メソッド呼び出し (`"methodCall"`).
- 基本ブロック間遷移 (`"blockEntry"`).

括弧内の文字列は, トレースファイル内で使用する各イベントのイベントタイプを示している. すべてのイベントに共通する JSON の構造は以下の通りである. ただし, `"lineNum"`と`"time"`は使用しないイベントもある. (フォーマットを読み易いように改行し, インデントを掛けている. 以下同様.)

```
{
   "type":イベントタイプ,
      :
   "threadId":実行されたスレッドの ID,
   "lineNum":行番号,
   "time":実行された時刻
},
```

また, フォーマットに出現する各オブジェクトは以下のような JSON で表す.

```
{
   "class":クラス名,
   "id":オブジェクト ID
}
```

メソッド実行の開始および終了イベントを表す JSON を以下に示す.

```
{
   "type":"methodEntry",
   "signature":メソッドのシグニチャ,
   "receiver":レシーバオブジェクトを表す JSON,
   "args":[引数オブジェクトを表す JSON,···],
   "threadId":実行されたスレッドの ID,
   "time":実行された時刻
}
```

```
{
   "type":"methodExit",
   "shortSignature":メソッドの短縮版のシグニチャ,
   "receiver":レシーバオブジェクトを表す JSON,
   "returnValue":戻り値オブジェクトを表す JSON,
   "threadId":実行されたスレッドの ID,
   "time":実行された時刻
}
```

フィールドの更新および参照イベントを表す JSON を以下に示す.

```
{
   "type":"fieldSet",
   "fieldName":フィールド名,
   "container":コンテナオブジェクトを表す JSON,
   "value":代入された値オブジェクトを表す JSON,
   "threadId":実行されたスレッドの ID,
   "lineNum":行番号,
   "time":実行された時刻
}
```

```
{
   "type":"fieldGet",
   "fieldName":フィールド名,
   "this":実行されたオブジェクトを表す JSON,
   "container":コンテナオブジェクトを表す JSON,
   "value":取得した値オブジェクトを表す JSON,
   "threadId":実行されたスレッドの ID,
   "lineNum":行番号,
   "time":実行された時刻
}
```