プログラムの動的解析効率化のための参照頻度を考慮したグラフ属性分割格納法

楠　和馬† 久米　出†† 波多野賢治†††

† 同志社大学大学院文化情報学研究科　〒 610–0394 京都府京田辺市多々羅都谷 1–3
†† 奈良先端科学技術大学院大学情報科学研究科　〒 630–0192 奈良県生駒市高山町 8916–5
††† 同志社大学文化情報学部　〒 610–0394 京都府京田辺市多々羅都谷 1–3
E-mail: †kusu@ilab.doshisha.ac.jp, ††kume@is.naist.jp, †††khatano@mail.doshisha.ac.jp

あらまし　プログラム実行時の依存関係を情報として含むトレースはデバッグ支援に多大な効力を発揮することが知られている．しかし，トレースは命令や値を節点，依存関係を関係辺とする複雑なグラフ構造であり，各グラフ要素には属性が付随しているためデータ量が容易に大きくなる．デバッグ時に作業者が随時解析処理を呼び出し，迅速なフィードバックを得られるように，トレースの解析効率を向上させることが現在我々に課せられている問題であり，この課題を解決することが本研究の目的である．本研究では動的解析中に計算機のメモリに読み込まれる属性のデータ量の削減を実現するグラフデータベースへのグラフの格納方法および処理方法を提案する．また，その提案手法の効果を最も高められる提案手法の適用基準を考案する．さらに，本研究で提案した手法が効率的な動的解析の実現に貢献しているか評価するために，解析範囲がトレース全体に及ぶ動的解析を実行して実験を行う．効率性に関しては解析に要したメモリ消費量及び解析時間によって判断する．

キーワード　グラフデータベース, Back-In-Time デバッガ, 動的解析, グラフ走査, Java

1. はじめに

デバッグではプログラムの実行時の状態と依存関係の調査が不可欠とされてきた [15,16]．現在普及しているデバッガはブレークポイントによって指定された箇所でプログラムを停止させ，その時点での状態の調査を可能とする．しかし，ブレークポイント以前の実行内容は参照できないため，依存関係を逆に辿り不正な状態の原因を特定する作業を効率的に実施できない [11]．

10 年前からこうした既存のデバッガの問題を解決するために，依存関係を情報として含むトレースを利用した逆回しデバッガ（Back-In-Time デバッガ）と呼ばれる新しい方式のデバッガが開発されてきた [3,6,9]．依存関係を含むトレースを利用することで，これらのデバッガは変数の値を代入した命令の特定や [6]，命令が実行された（あるいはされなかった）理由の調査 [3]，また，既に呼出し完了したメソッドの実行内容の調査 [9] のように，局所的な視点での依存関係の解析を実現する．

一般に依存関係を含むトレースの処理に関しては動的解析の解析範囲への対応の問題が指摘されている [11]．近年のハードウェアおよびソフトウェア環境の急速な進歩がこの問題の解決を容易にしつつある．実際，我々の先行研究 [5] では不具合を含む実用的な Java のフレームワークアプリケーションに対してその感染を示唆する兆候を特定する動的解析を実現している．

このように我々の先行研究 [5] はトレースに含まれている依存関係に関する動的解析の規模の問題解決に明るい見通しを与えるものであるが，一方でその実行効率に関しては大きな課題を残している．実行効率の低下の主な原因はトレースのモデルに豊富なデータが含まれているところに存在する．我々のトレースはバグの原因となるような兆候の特定以外の側面で解析する際の要求を満たすために，Wang 等の研究 [14] のようにトレースのデータ量の抑制を目指す代わりにデータの豊富さを追求している．

既存のデバッガ [3,6,9] は特定の命令に関する局所的な動的解析を実行しているが，我々の先行研究 [5] ではプログラム実行時の全ての状態変更命令 (注1) を解析の対象としている．そのため，規模の対応は可能であっても実行効率に関しては良い結果が得られていない．この問題を解決するためには，動的解析の基本的な処理である参照関係や依存関係のような依存関係を辿るパフォーマンスを向上させる必要がある．

そこで本研究では，我々の提案する動的解析のような解析範囲がトレース全体におよぶ動的解析をより実用的に実行するために，動的解析処理の効率を考慮した動的解析環境を構築する．また，構築した動的解析環境において解析処理の効率性を損なう要因を明確化し，その要因を解消するためのグラフの管理方法および処理方法を提案する．さらに，本研究で提案した手法を適用後の解析の処理パフォーマンスに関する評価を行う．

2. 関連研究

現在広く使われているデバッガはプログラムコード中のブレークポイントで指定された箇所でプログラム実行を停止させ，作業者が停止時の状態を調査するための機能を提供している．この時点で既に呼出しが完了したメソッドの実行内容はデ

(注1)：インスタンス変数，クラス変数，配列への代入

バッガに記録されていない．プログラムの不具合や感染 (注2) はしばしば既に呼出しが完了したメソッド内に発見される [9]．既存のデバッガでこうしたメソッド呼出しを調査するためにはブレークポイントの設定と実行のやり直しが必要とされ，これがデバッグ作業の効率性を阻害する大きな要因となっている [11]．

プログラムの実行履歴を利用することによってこうした既存のデバッガの限界を克服しようとする研究が最近の 10 年間で進められている．これらの研究の発端となった全知デバッガ [6] はある実行時点の変数の値に対してそれを代入した命令文を特定する機能を実装している．また，Ko らによる Whyline [3] はある命令文が実行された，あるいはされなかった過程を対話的に再現することを可能としている．

Lienhard らによる Dynamic Object Flow 解析 [7] はオブジェクトの視点から依存関係を解析，可視化する機能を実現している．Object Flow 解析はオブジェクトに対する参照に焦点を置いておりメソッドの依存関係解析 [8] であると同時にデバッガの利用を念頭に置いたオブジェクトの流れを表現している．

全知デバッガや Whyline による支援はある特定の命令文に対してその関連する依存関係を辿る機能によって実装される．こうした支援を必要とする作業者の関心は特定の命令に限定されており，作業者の関心の範囲が反映される形で制御やデータに関する局所的な依存関係が解析される．一方で制御やデータに関してトレース全体を解析する動的解析は我々の過去の研究 [5] 以外のものは我々の知る限りでは存在せず，メソッド呼出しか Object Flow 解析のようなオブジェクトの参照程度しか扱われていない．

局所的な解析に基づく支援は感染が疑われる変数値のようにデバッグの問題解決に直接寄与する状態を発見した場合には有効である．しかし，現実のデバッグ作業の作業者はこうした情報を発見するためにプログラムの実行過程全体を対象に状態を把握し実行の挙動を理解することが求められる [1]．我々の過去の研究 [5] は解析範囲がトレース全体におよぶ制御と値の依存性の解析によってこの種の要求を満たすことが目的である．

文献 [5] では動的解析の一つである Outdated-State 解析手法を提案している．この動的解析手法は同じオブジェクトの異なる二つ以上の状態に影響を受け実行された命令を検出する．このような実行過程のパタンはオブジェクトのコレクションの状態を参照して繰返し制御を行う時の事例がある．この実行過程のパタンは直接不具合の要因となることや潜在的な不具合の原因となるため，このパタンは検出されれば修正するべき実行パタンの一つとなる．従来の動的解析を実行する環境で上記のようなトレースの全体を解析対象とする動的解析を行う場合，トレースの節点をメモリ上に読み込んだ上で依存性解析を行うため，トレースのデータ量が大きくなると実行ができない問題があった．また，依存性解析を行うと多数の節点同士の比較や，トレースに記録された依存関係をもとに一つないしは多数の節点の導出が効率的に実行できない問題もある．これら問題を解決するためには，トレースのデータ構造を把握した上でより効率的な解析を支援できる動的解析環境を構築する必要がある．

3. 動的解析環境の構築

本研究ではデバッグにはさまざまな種類の動的解析を実行することが必要であると想定しており，実際に過去の研究でも複数の解析手法を開発している [5]．

さまざまな動的解析手法の適用を可能にするために我々のトレースにはあらゆる動的解析にも対応することができるデータモデルが採用されている．その代償として実用的なプログラムのトレースは膨大かつ複雑になり易く，それが解析の効率の大きな妨げとなり易い．したがって，効率的な動的解析の実現にはトレースに対する動的解析を効率的に実行する動的解析環境が要求される．

図 1 に Java プログラムの実行から動的解析までを実施する動的解析環境の概要を示す．図中のトレース生成部は Java Bytecode instrumentation 技法を利用することによってトレースを生成する．トレース処理部は生成されたトレースをグラフデータベース（GDB）に格納し，関係性を用いた解析の効率的な実装を支援する．

3.1 動的解析環境に必要な機能

先行研究 [5] では，図 2 のような動的解析環境を構築している．図 2 の動的解析環境ではデバッグ対象のプログラムを実行し，そのプログラムの実行過程を記録したトレースを生成する．ここで記録されているトレースに含まれている命令や値，依存関係は POJO (Plain Old Java Object)(注3) のオブジェクトとして生成しメモリ上に格納する．また，動的解析中において，メモリに格納されたトレースを参照し，命令や値，依存関係を調査することにより動的解析を実行している．2. で述べたように，トレースの局所的な範囲を解析する手法とトレース全体を解析範囲とする手法が提案されており，当然ながら後者の手法は解析にメモリを多く消費する．したがって，既存の動的解析環境においてトレースのデータ量が大きくなるほど解析に利用できるメモリ領域が不足することになり，膨大なトレースに対して解析範囲がトレース全体に及ぶ動的解析手法は実行することができない．

以上より動的解析環境に要求されることとしては，あるデータから特定の関係性を辿り別のデータを取得する処理（グラフ走査）を高速に行える必要が第一にある．また，トレースのデータ量の大小にかかわらず，解析範囲がトレース全体に及ぶ動的解析手法を実行可能にするためには，トレースの管理や解析処理で消費するメモリを削減する必要がある．さらに，本研究で取り扱う動的解析手法では必要ないが，異なるトレース同士を比較することでプログラムの挙動を解析する手法も存在するため，それら手法を将来的に適用することを想定する必要がある．

(注2)：不具合はプログラムコードの誤りを，感染は不具合箇所の実行に起因する実行時の誤りを意味する [16]．

(注3)：フレームワークのような規約に縛られないように設計した Java オブジェクトのこと．

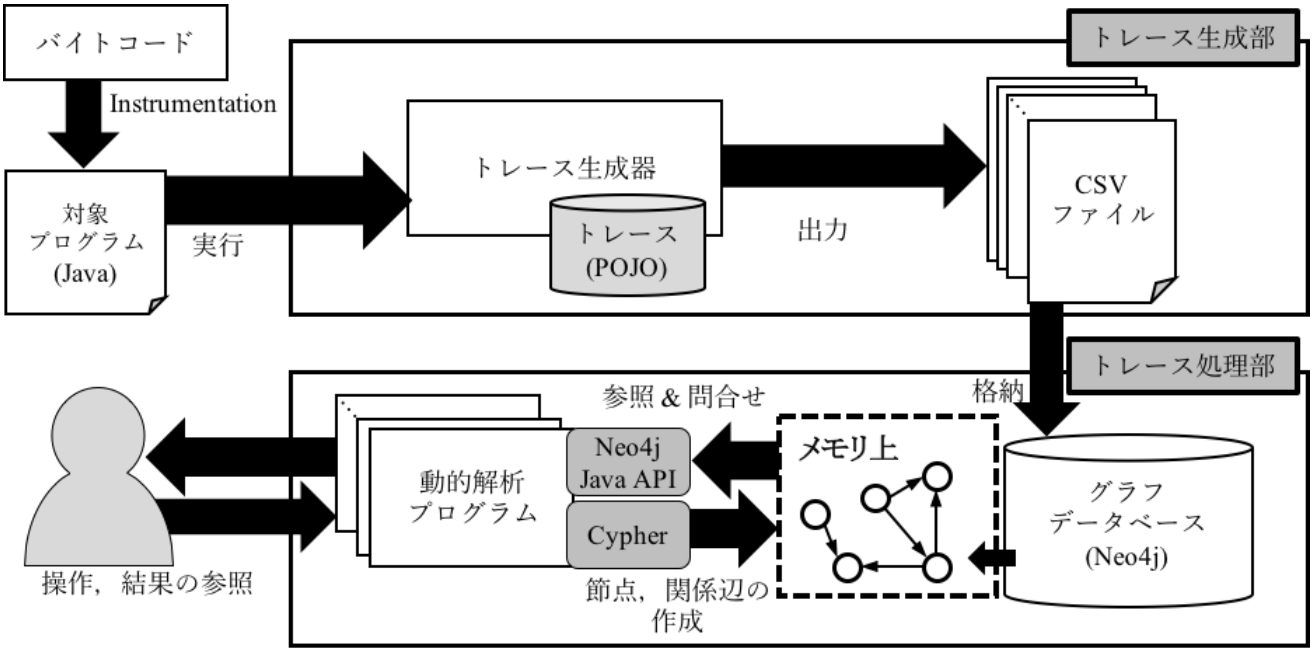


図 1: 本研究で構築する動的解析環境

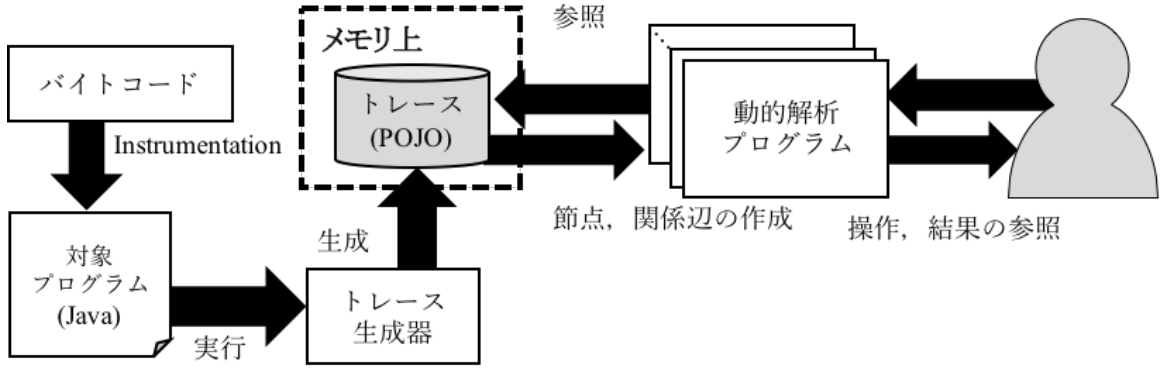


図 2: 先行研究の動的解析環境

3.2 トレースのデータモデル

我々が開発した動的解析 [5] は個別のメソッド内部およびメソッド間に跨る依存関係を解析の対象とする．この時，値を生成および参照する命令や，これらの命令が実行されるメソッドに関する情報も解析に利用される．我々は更にある特定の条件を満たすデータに依存する動的解析手法も現在開発中である．

こうしたさまざまな解析の要求に答えるために我々のトレースには以下の概念を表現する要素が含まれている．

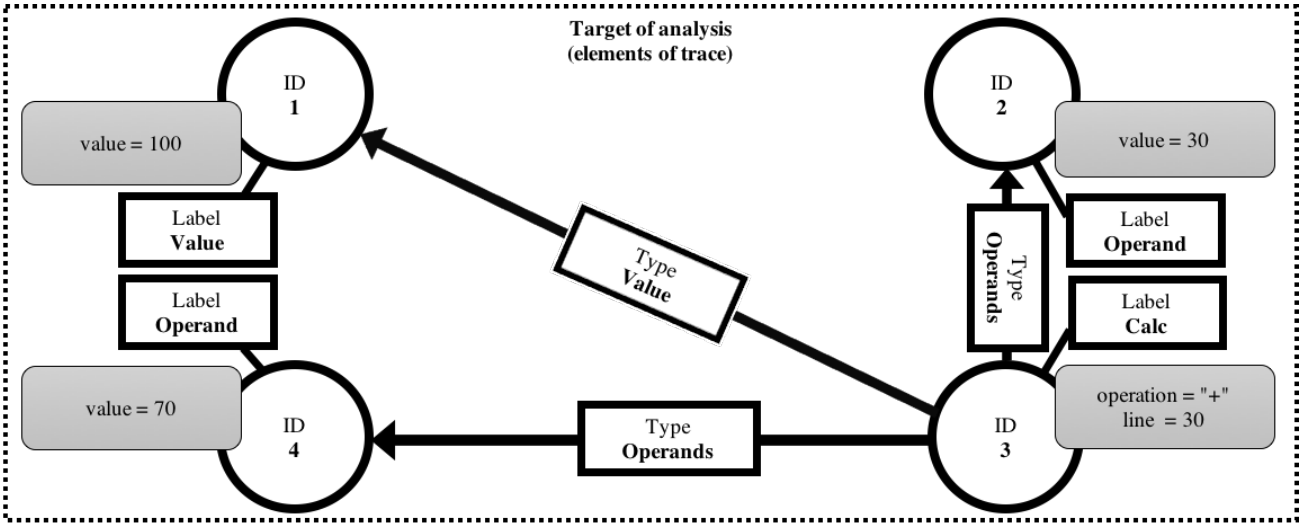
- メソッド呼出し構造
- メソッドで実行されたバイトコード命令
- バイトコード命令による値の生成参照
- 参照される値

それぞれの要素が表現している概念から要素間にさまざまな関係が導かれる．メソッド呼出しに関しては呼出し側と被呼出し側の関係が導かれる．条件分岐命令やメソッド呼出しのように「制御する」命令とそれらによって実行される命令の間には制御の依存関係（制御依存関係）が形成される．また値の生成と参照の関係を通じて命令同士にデータの依存関係（データ依存関係）が形成される．値とその生成は一対一関係を形成し，値とその参照の間には一対多関係が生成される．トレースはこれらの要素を節点，要素間に導入された関係を辺とする有向グラフとして表現される．

まず，我々のトレース生成手法により生成されるトレースは各構成要素に行番号やスレッド番号のような属性を持つため，図 3 のようなプロパティグラフモデルで表現できる [13]．プロパティグラフモデルは Apache の TinkerPop プロジェクトで定義されているデータモデルである (注4)．また，これら属性は動的解析の際に参照されるデータや，動的解析の結果として理解されやすいように表示の際に参照されるデータが含まれている．

動的解析を行う際には，オブジェクトの状態の変遷やプログラムのエラーを追跡など，命令や値の依存関係を辿ることになる．したがって，動的解析では制御依存関係やデータ依存関係などの依存関係を辿る（グラフ走査）処理が頻繁に行われることが想定され，グラフ走査処理の効率化が要求される．

(注4)：Apache TinkerPop: https://tinkerpop.apache.org/（閲覧日：2017-01-16）



Nodes

Node ID	Attribute Key	Attribute Value
1	value	100
2	value	30
3	operation	”+”
	line	30
4	value	70

Relationships

Start Node ID	End Node ID	Attribute Key	Attribute Value
3	1	-	-
3	2	-	-
3	4	-	-

図 3: プロパティグラフモデル

3.3 動的解析の処理方法

動的解析は解析範囲の広狭に関わらず解析の起点となる命令や値からデータ依存関係や制御依存関係などの依存関係を辿ることによって行うことができる．したがって，グラフ走査の起点となる節点（走査開始節点）を取得後は依存関係を辿りながらトレースを調査していくため，グラフ走査処理の効率化を図ることが動的解析の効率化につながる．また従来の場合，解析中は全てのトレースの要素がメモリ上に読み込まれ，その上，トレースの要素の依存関係を解析するため組合せ爆発のような問題が発生し，トレースの全体を解析対象とする動的解析に対応できなかった．そのため，膨大なトレースに対する解析を可能にするために，メモリ上に読み込まれるのは解析に必要なデータのみにする必要がある．

そこで，本研究ではグラフ走査処理のパフォーマンス向上やディスク上での管理といった二点の必要性を考慮して，トレース処理部の基盤にはオンディスクデータベースである GDB を用いる．GDB にはさまざまなソフトウェアが存在するが，本研究では Neo4j(注5) を採用することにした．これは，Neo4j がトレースを格納するのに適したプロパティグラフモデルを採用しており，同時に GDB の中でグラフ走査処理の性能が高いことが示されていたからである [2,4]．また，GDB の中でも関係性を利用したグラフ解析に最適化している GDB はグラフ走査の対象となっている節点に直接関係がないデータを読み込まずに解析を行うことができるため，データ量が膨大なトレースに対しても全体的に解析範囲がおよぶ動的解析手法の適用を可能にする．

Neo4j のデータベースに格納されたトレースに対する動的解析の実装は Java のグラフ走査処理を実装するライブラリである Neo4j Traversal API および，グラフ問合せ用のクエリ言語 Cypher を用いることにより行う．Cypher により解析の起点となるデータを問合せを行い，Neo4j Traversal API を用いることにより命令や値の種類に合わせて条件付けしたグラフ走査を実現することができる．

4. 属性参照の頻度を考慮した節点属性の分割

3. では，GDB の Neo4j を用いたトレースの格納方法，処理方法について説明した．データをディスク上で管理することにより，動的解析に必要ないデータはメモリ上に存在しないため，メモリ消費量の削減につながっている．しかし，我々の動的解析環境ではグラフ走査の際に動的解析に不要な命令や値の属性を読み込んでいることにより，無駄なディスクアクセスが増えていることがボトルネックになっていると考えられる．そこで，本節ではこのボトルネックを改善するグラフの変換方法について提案する．

4.1 オンディスクデータベースの挙動

Neo4j のデータベースに格納されているデータに対して問合せを行う際には Cypher による問合せ，もしくは Neo4j Traversal API によるグラフ走査によらず，まず，走査開始節点の問合せから開始される．次に，走査開始節点から関係性を辿ることで次の節点を取得することを繰り返すことでメモリ上にグラフ

(注5)：Neo4j HP. http://neo4j.com/（閲覧日: 2017-01-16）

データを展開している．そのため，グラフ走査がデータベースに格納されている全データに対して行われる際には，ほとんど全ての節点や関係辺がメモリ上に読み込まれることになる．

また，メモリ上へ節点や関係辺を読み込む際にはそれらデータを取得するためにディスクアクセスが発生していると考えられる．したがって，トレースに対して全体的に動的解析を実行する場合，ほとんどのトレースのデータをディスク上から取得しようとし，グラフ走査が非効率になる．他にも以上の問題はメモリへ読込むデータ量が多くなることから，動的解析に要するメモリ領域を確保するためにも無駄なトレースの要素の読込みは削減できるべきである．

4.2 メモリ消費量削減のための節点属性の分割

本研究で構築する動的解析環境は GDB を利用することにより，動的解析の基本的な処理である依存性の解析を効率化するだけでなく，解析に必要な節点のみをメモリ上に読み込むため膨大なトレースに対しても解析範囲が全体的な動的解析に対応することも可能になる．節点の読込みと一緒にその節点の属性もメモリに読み込まれるといった，本研究で構築する動的解析環境のボトルネックは，不要な属性を異なる節点に格納しておくことにより削減できるものと考えられる．

そこで，解析の対象となる節点（解析対象節点）とその節点の属性が格納されている節点（属性用節点）の二つに分けて管理することで上記のボトルネックを除去する方法を提案する．また，解析対象節点に対応する属性用節点を取得できるようにするために，図 4 のようにそれらの間には関係辺（属性辺）を作成する．そのため，トレースの全要素に適用すると次のようなグラフ構造の変更点が発生する．

（1） GDB に格納されている節点数は元の倍になる．

（2） 各節点に接続辺が一つ増える．

しかし，Neo4j のようなネイティブグラフデータベースではグラフの走査はある節点に接続している節点のみを対象とするため，変更点 (1) のような節点の増加の影響はほとんど受けない [10, 12]．また，変更点 (2) により，解析に不要な節点の属性の読込みを削減できる代わりに，属性辺が一つ読み込まれるようになる．さらに，グラフ走査の際に辿る必要のある辺を確認する対象が全節点で一つ増加するが，ほとんどのグラフ走査は関係性を指定するため，関係辺の候補は実質増えずグラフ走査の効率への影響はほとんど無い．ただ，動的解析において必ず属性の参照が必要な節点ラベルが存在するため，全てに対して属性の分割を適用することはグラフ走査の処理を増やすことになり非効率化につながる．したがって，各節点ラベルに対して節点属性の分割を適用するか適用しないかを判断する基準を設ける必要がある．

4.3 節点属性の分割が必要な節点の導出方法

本研究の提案手法の目的は節点に付随する属性のメモリへの読込みを削減することで，グラフ走査の効率性を向上させることである．一方，節点の分割を施すことにより懸念される解析の非効率化の要因としては，解析時に属性が必要な節点に対しても節点属性の分割を行うことで属性辺を辿る処理が発生することである．属性辺のグラフ走査は元々のトレースの構造に対する解析時には発生しない処理であるため，属性辺を辿る処理が発生するほど解析の非効率化につながる．これらのことから，動的解析中に発生する節点属性の読込み数および属性辺のグラフ走査回数を最小にできるようにグラフ変換する節点を決定する必要がある．

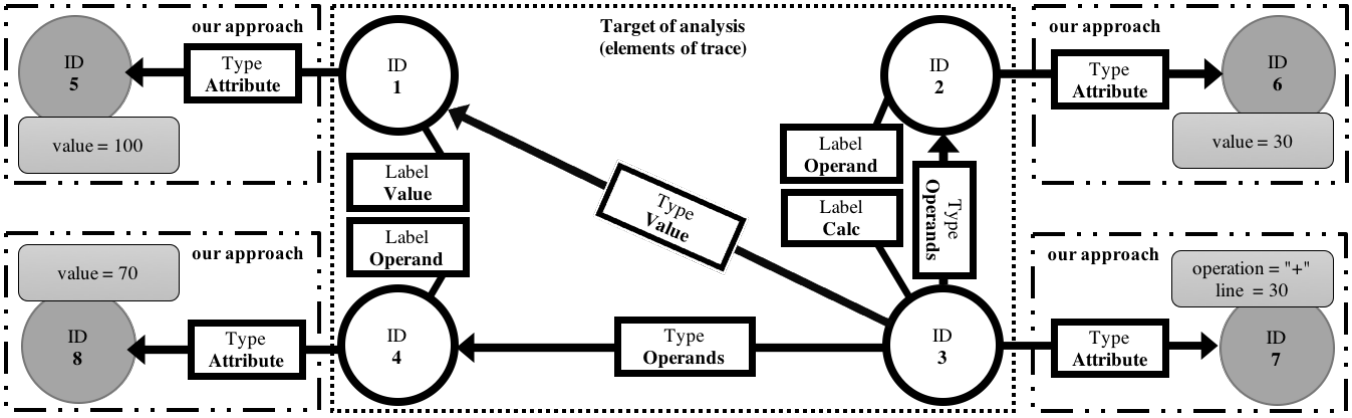
Algorithm 1 属性を分割する節点の自動決定アルゴリズム

Require: $N_{node}, N_{load}, N_{trav}$
  for each $l \in L$ **do**
    $\boldsymbol{f}[l] \leftarrow$ **false**
  end for
  for each $l \in L$ **do**
    $before \leftarrow S_{load}(\boldsymbol{f}, N_{load}, N_{node})$
    $\boldsymbol{f}[l] \leftarrow$ **true**
    $after \leftarrow S_{load}(\boldsymbol{f}, N_{load}, N_{node})$
    $traversal \leftarrow S_{trav}(\boldsymbol{f}, N_{trav}, N_{node})$
    if $before > after$ **and** $traversal = 0$ **then**
      continue
    else
      $\boldsymbol{f}[l] \leftarrow$ **false**
    end if
  end for
  return $\boldsymbol{f}$

節点属性を分割する節点の自動決定には動的解析環境に実装されている動的解析手法の解析アルゴリズムを事前に分析しておく必要がある．すなわち動的解析手法ごとに，節点ラベル別に依存関係を辿る処理や節点属性の取得する処理などが定義されており，それらを事前に集計しておく必要がある．本節で導入する節点属性を分割する節点の自動決定では，ある節点ラベルの依存性を解析する際に発生する属性読込みの情報を用意しておく必要がある．ただ，依存性解析は基本的に節点属性の値によって場合分けされる可能性があり正確に測ることができない．そのため，解析されるかどうかわからない対象も依存性解析がなされると仮定することで最悪の場合を想定する．事前に用意した節点ラベル別に解析時に発生しうる属性読込みの情報をもとに，属性読込みが発生する可能性がない場合はその節点の属性を分割するが，属性読込みが発生する可能性がある場合は節点の属性を分割しないようにする．

以上の節点属性の分割を行う節点の自動決定方法を実現したアルゴリズムを Algorithm 1 に示す．Algorithm 1 では，節点ラベルの集合を L としている．このとき，$N_{node}(l)$ は GDB に格納されている節点ラベル $l \in L$ ごとの個数であり，$N_{load}(l)$ は節点ラベル $l \in L$ の節点が共通して持ちうる属性の個数である．また N_{trav} は，節点ラベル $l \in L$ に到達した際に発生する節点ラベル $m \in L$ に対する属性の読込み回数である．ただし，節点ラベル l，m は同一な節点ラベルの場合（$l = m$）も考慮する．

次に，節点の属性を分割する基準となる $S_{load}(L)$，$S_{trav}(L)$ はそれぞれ解析を実行後の属性の累計読込個数，属性辺のグラフ走査累計回数である．これらは，N_{node}，N_{load}，N_{trav} により推定することが可能である．$S_{load}(L)$，$S_{trav}(L)$ の算出方法は式 (1)，(2) に示す．また，式中の $\boldsymbol{f}$ はラベルごとに節点属性を分割す



Nodes

Node ID	Attribute Key	Attribute Value
1	-	-
2	-	-
3	-	-
4	-	-
5	value	100
6	value	30
7	operation	”+”
	line	30
8	value	70

Edges

Start Node ID	End Node ID	Attribute Key	Attribute Value
1	5	-	-
2	6	-	-
3	1	-	-
3	2	-	-
3	4	-	-
3	7	-	-
4	8	-	-

図 4: 節点属性の分割後のトレース

るかしないかを区別する辞書型データであり，節点ラベル名ごとに属性を分割するのであれば *true*，属性を分割しない場合は *false* としている．

$$S_{load}(L) = \sum_{l \in L} s_{load}(l, \boldsymbol{f}[l]) \tag{1}$$

ただし

$$s_{load}(l, \boldsymbol{f}[l]) = \begin{cases} N_{load}(l) \cdot N_{node}(l) & if\ \boldsymbol{f}[l] = false \\ 0 & otherwise \end{cases}$$

$$S_{trav}(L) = \sum_{l \in L} s_{trav}(l, \boldsymbol{f}) \tag{2}$$

ただし

$$s_{trav}(l, \boldsymbol{f}) = \begin{cases} \sum_{m \in L} N_{trav}(l, m) \cdot N_{node}(m) & if\ \boldsymbol{f}[m] = true \\ 0 & otherwise \end{cases}$$

式 (1) 中の $s_{load}(l, \boldsymbol{f}[l])$ は節点ラベル l の節点を読み込んだ際に同時に読み込まれる節点属性の個数を， GDB に格納されている節点ラベル l である節点の個数で掛けた値である．式 (2) 中の $s_{trav}(l, \boldsymbol{f})$ は節点ラベル l，m において，節点ラベル l の依存性を解析する際に節点ラベル m の節点属性を読み込む回数を，GDB に格納されている節点ラベル m である節点の個数と掛けることを全ての組合せで行い，それらを総和した値である．ただし，$s_{load}(l, \boldsymbol{f}[l])$ は節点属性の分割が適用されている節点に対しては節点属性の読込みが発生しないため 0 となり，$s_{trav}(l, \boldsymbol{f})$ は節点属性の分割が適用されていない節点に対しては属性辺のグラフ走査が発生しないため 0 となることに注意する．

属性辺のグラフ走査回数 $S_{trav}(L)$ を 0 回に維持しながら，属性の読込み回数 $S_{load}(L)$ を最小にする節点属性の分割を適用する節点ラベルの組合せ $\boldsymbol{f}$ を導出し，トレースのグラフ構造の変換時に利用する．

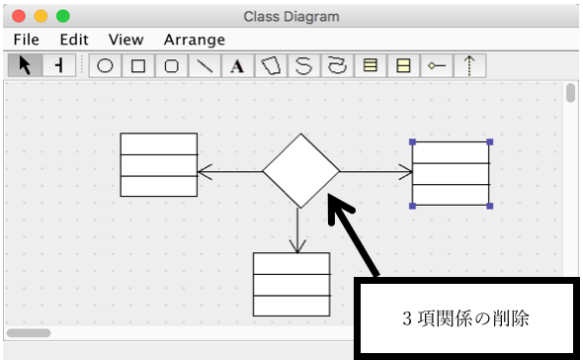
5. 評価実験

4.1 で示したグラフ走査の際に不要なデータでさえもメモリに読み込まれるといったボトルネックを解消する方法を 4.2 で提案した．本節では本提案手法がグラフ走査の処理パフォーマンスにどのように効果をもたらすのか明らかにするため実験を行う．トレースの全域を対象とする動的解析は解析時間だけでなく，メモリを効率的に利用できているか評価する必要がある．本研究の提案手法を適用することで解析範囲がトレース全体に及ぶ動的解析の処理パフォーマンスが改善されているかどうかについて評価を行う．この実験ではトレースの構造ごとに処理パフォーマンスの比較を行うために，トレース全体を解析対象とする動的解析の解析時間，動的解析実行中のメモリ消費量を計測する．本実験は OS: Linux(注6)，CPU: 2.26 GHz 4 core(注7)，RAM: 64GB の Kernel-based Virtual Machine の環境で行う．

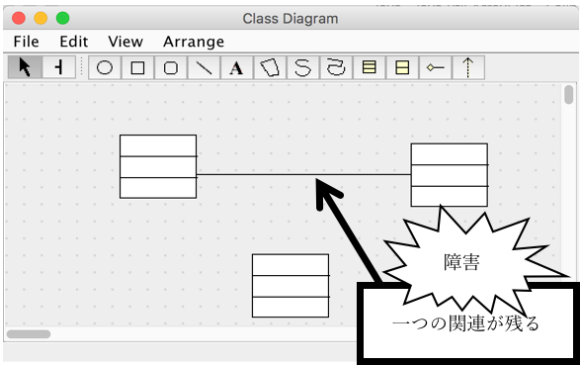
5.1 UML エディタ “GEFDemo”

5.2 の解析に利用するトレースには Graph Editing Framework のデモプログラム (GEFDemo) (注8) の実行に関するトレースを用

(注6)：CentOS 7.2-1511
(注7)：QEMU Virtual CPU ver. 0.9.1
(注8)：GEFDemoUML-0.10.5beta-src.zip を実行．`http://gefdemo.stage.tigris.`

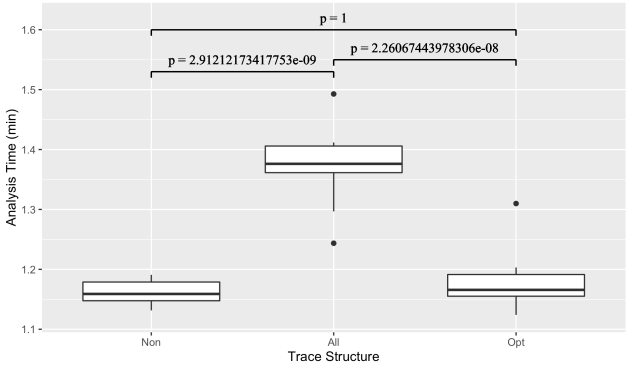


(a) クラスと 3 項関係の作成

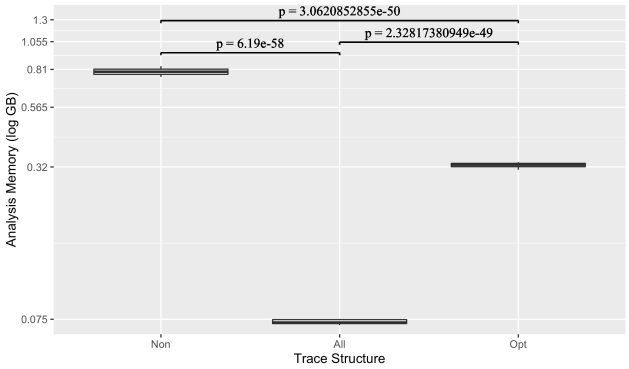


(b) 3 項関係の削除と例外の発生

図 5: GEFDemo の操作



(a) 動的解析の平均解析時間



(b) 動的解析の平均メモリ消費量

図 6: 動的解析パフォーマンスの評価

いる．GEFDemo は図 5 に示すようなアプリケーションフレームワークを用いた簡易な UML エディタである．GEFDemo にはプログラムの実装に欠陥があり，3 項関係を削除する操作を行うと図 5(b) のような障害が発生することが知られている．

また，図 5(b) に示すような障害が発生する原因は手作業により確認されているため [5]，動的解析の実装が正確かどうかについて検証が可能である．本実験において利用するトレースは図 5 のように，以下の手順で故意に例外を発生させたプログラムの実行過程を記録している．

（1） エディタ上にクラスを三つ作成する．

（2） 一つのクラスから他のクラスに対して関連を作成する．

（3） 関連を作成していないクラスから関連の線に対して関連を作成する．

（4） 3 項関係を削除する．

5.2 Outdated-State 解析

5.1 で述べた GEFDemo の障害の原因は Java プログラムのコレクションの状態を参照した繰返し命令の途中でコレクションの状態が変更されたため発生している．本研究では GEFDemo の障害の原因を検出するため，命令で同じオブジェクトの異なる状態を利用している命令を検出することができる動的解析の一つである Outdated-State 解析を用いる．

動的解析環境において，以下の手順で Outdated-State 解析を実行する．

（1） メソッド呼出しを実行順に一つずつ調査する．

（2） 一つのメソッド呼出しに対して，そのメソッドで実行されている複数の命令の種類ごとにオブジェクトの状態との依存関係を調査する．

（3） 状態変更命令を解析した際に，その値の変更が何回目か記録した節点を GDB に作成する．

（4） 手順 (3) で作成した節点から同じオブジェクトの新しい状態と古い状態の組合せと依存関係を持つ命令が存在するか調査する．

Outdated-State 解析は手順 (1) で示すように，プログラムの実行全体を解析する必要があるため，通常の実装では動的解析にメモリ空間が多く必要となる．

5.3 動的解析の処理パフォーマンスの評価と考察

動的解析の処理パフォーマンスを評価するために，評価項目として動的解析時間およびメモリ消費量の計測を行う．動的解析時間は解析時間の開始時と終了時の時間を記録して差分を算出する．一方，動的解析中の最大メモリ消費量は UNIX コマンドの vmstat で実行中の毎秒のメモリ消費量を計測することで，その最大値と動的解析実行前のメモリ消費量の差分を算出する．

以上の実験を 10 回試行し，動的解析時間の算出結果を図 6(a)，最大メモリ使用量の算出結果を図 6(b) に示す．図中の p 値は対応のある t 検定により導出した値であり，有意水準 5 % としたときにこの水準を下回らなければ帰無仮説 (H_0)「2 群間の平均値に差はない」を採択し，下回る場合は帰無仮説を棄却し対立仮説 (H_1)「2 群間の平均値に差がある」を採択する．

---

org/.（閲覧日: 2017-01-16）

図 6(a) より，トレースの全ての節点に対して属性分割を適用した場合 (All)，グラフの構造を変換しなかった場合 (Non) と比べて統計的に差が見られ All は非効率化につながっている．属性分割を適用する節点ラベルを 4.3 で導入した Algorithm 1 により自動決定した場合 (Opt)，Non と同等の解析時間で解析を実行することができていることが分かる (注9)．All では，属性辺を辿るグラフ走査が多数発生することにより解析時間を悪化させていたが，Opt では Algorithm 1 の効果により属性辺のグラフ走査が発生しない Non と同様な解析時間で実行することができていることが確認できた．次に図 6(b) ではトレース構造ごとに最大メモリ使用量に大きく差が出たため，便宜上 y 軸は対数軸にしている．図 6(b) より All はどのグラフ構造よりもメモリ消費量を大きく削減できている．これは，本研究で提案した手法を全ての節点に対して適用したことにより，節点属性の読込みが全く発生しなくなったことによる結果である．一方，Opt ではメモリ消費量の削減に関して All よりも劣るものの Non の平均メモリ消費量 15.77 GB から 43.1 % 削減することができ，動的解析を効率的に実行することが実現できた．以上をまとめると本研究で導入した Algorithm 1 の効果により，属性辺のグラフ走査を発生させずに節点属性の読込みを削減でき，元々のトレース構造に対する動的解析の速度を維持しつつ，大幅にメモリ消費量を削減すること可能にした．

本研究における提案により，グラフ走査により到達した節点に付随する属性の読み込み時に発生すると想定していた，ディスクアクセスおよびメモリへの属性の読込みのうち後者は改善することができていることが実験で確認できた．前者のディスクアクセスの削減により処理時間を短くできなかったのは Neo4j の実装において節点の属性をディスクから取得する処理が最適化されていることにより，処理の遅延に大きく影響していなかったと判断できる．

6. おわりに

本研究では膨大で複雑なトレースの処理の効率化のために，依存関係を辿ることにより解析する動的解析の特徴に着目し，グラフ走査の処理が最適化された GDB を用いて動的解析環境を構築した．また，動的解析において解析に不要なデータも一緒にメモリに読み込まれるといった非効率的な処理を抑制するために，節点の属性の読込みを削減する格納方法を提案した．さらに，本手法は全ての節点に対して適用すると動的解析の非効率化につながるため，節点に対して提案手法を適用するか適用しないか自動決定する方法を導入した．動的解析時間およびメモリ消費量の観点から動的解析パフォーマンスの評価実験を行った結果，本研究で提案した非効率化を抑制するように工夫したトレースの節点属性を分割する手法は元々のトレース構造の場合よりも動的解析時間を維持しつつ，メモリ消費量はトレースの構造を変化しない場合よりも 43.1 % 削減することを可能にした．

(注9)：それぞれの平均解析時間は，Non が 1.15 分，All が 1.37 分，Opt が 1.17 分である．

今後の課題としては，今回は Outdated-State 解析のみ適用した場合を考えているため，他の動的解析手法を適用した際の効果検証も行う必要がある．また，本研究で提案した手法が動的解析の高速化には寄与できなかったため，動的解析中に多数発生している命令や値の比較回数を少なくする必要がある．

謝　辞

本研究は同志社大学ハリス理化学研究所研究助成事業および栢森情報科学振興財団研究研究助成事業，人工知能研究振興財団研究助成事業，日本学術振興会科学研究費助成事業 25240014 および 26280115，15K12009 の助成を受けて遂行された．

文　献

[1] David J. Agans. *Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. Amacom, 2002.

[2] Salim Jouili and Valentin Vansteenberghe. An empirical comparison of graph databases. In *Proceedings of the 2013 International Conference on Social Computing*, SOCIALCOM '13, pages 708–715. IEEE Computer Society, 2013.

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

[4] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlýnková. Experimental comparison of graph databases. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, 2013.

[5] Izuru Kume, Masahide Nakamura, Naoya Nitta, and Etsuya Shibayama. A case study of dynamic analysis to locate unexpected side effects inside of frameworks. *International Journal of Software Innovation*, 3(3), 2015.

[6] Bil Lewis. Debugging Backwards in Time. In *Proceedings of the Fifth International Workshop on Automated Debugging*, 2003.

[7] Adrian Lienhard. *Dynamic Object Flow Analysis*. Lulu.com, 2009.

[8] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Exposing side effects in execution traces. In *International Workshop on Program Comprehension through Dynamic Analysis*, pages 11–17, 2007.

[9] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. *Practical Object-Oriented Back-in-Time Debugging*. 2008.

[10] Sonal Raj. *Neo4J High Performance*. Packt Publishing, 2015.

[11] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *International Conference on Software Engineering*, pages 485–495, 2012.

[12] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. 2015.

[13] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *Computing Research Repository*, abs/1006.2361, 2010.

[14] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *International Conference on Software Engineering*, pages 512–521, 2004.

[15] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449, 1981.

[16] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.