OUKA
The University of Osaka
Institutional Knowledge Archive

Title	限られた保存領域を使用する Java プログラムの実行トレース記録手法
Author(s)	Shimari, Kazumasa; Inoue, Katsuro; Ishio, Takashi
Citation	コンピュータソフトウェア. 2019, 36(4), p. 107-113
Version Type	VoR
URL	https://hdl.handle.net/11094/92582
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は日本ソフトウェア科学会に帰属します. 本著作物は著作権者である日本ソフトウェア科学会の許可のもとに掲載するものです. ご利用に当たっては「著作権法」に従うことをお願いいたします.
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

限られた保存領域を使用する Java プログラムの実行トレース記録手法

嶋利 一真　石尾 隆　井上 克郎

本研究では，ソフトウェアの内部状態の分析を行うための，限られた保存領域を効率的に使用する実行トレースの記録手法を提案する．提案手法はプログラムの各命令が使用したデータの最新の系列を，命令ごとに一定個数記録するような領域を個別に作成する．命令ごとの記録個数を調整することで，開発者が保有する保存領域に応じた実行トレースの記録が可能となる．

In this paper, we propose a method to collect an execution trace to analyze the internal states of software with a planned amount of storage. The method uses separated data buffers to record the actual values used by individual instructions. Changing the buffer size of each instruction, the method can record an execution trace using a limited size storage.

1　まえがき

デバッグとは，外部から観察できるソフトウェア障害の症状から，その原因となるソースコード内の欠陥を突き止める活動である [14]．効果的なデバッグの実施において重要となるのが，ソースコード中に書かれた命令の実行順序や変数の値の観測である [11]．特に，障害が発生する場合としない場合のソフトウェアの実行を観測し，実行中の様々な時点で条件分岐や変数の値にどのような違いがあるかを調査することが有効である [2] [5]．

開発者がソフトウェアの実行の様子を観測する手段として，ソフトウェアの処理の進行状況を示すメッセージや重要なデータをプログラムの外部に出力するロギング処理が広く用いられている [6]．しかし，ロギングによって記録されるのは開発時点で選定されたデータのみであり，実際のデバッグにあたって必要な変数の値がログに含まれていない場合も多い [13]．デバッグの効率的な実施を可能とするためには，プログラムの実行を網羅的に観測，記録する手法が必要である．

デバッグに必要となる変数の値を自動的に，かつ網羅的に収集する方法の 1 つとして，プログラムの実行中のメモリ状態を時系列で完全に記録する Omniscient Debugging [7] が提案されている．しかし，このアプローチは Java プログラムを対象にした場合で，実行 1 秒あたり 10 MB 程度の実行トレースデータの記録を必要とする [10]．また，実行トレース全体の大きさは，解析対象となるソフトウェアの規模や実行する機能の種類によって大きく異なる [3]．そのため，開発者が現在直面しているバグに対して手法の適用が可能であるのか判断することが難しい．

本研究は，実行トレース記録に使用される保存領域の大きさを開発者が制御可能とし，かつ，デバッグに必要となる命令の実行順序や変数の値を可能な限り網羅的に収集する方法として，プログラムにおける各命令が使用したデータをそれぞれ最新 k 回だけ保存する実行トレース記録手法を提案する．通常の実行ト

An execution trace recording method using a limited size storage for Java.

Kazumasa Shimari, Katsuro Inoue, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

Takashi Ishio, 奈良先端科学技術大学院大学先端科学技術研究科, Graduate School of Science and Technology, Nara Institute of Science and Technology.

コンピュータソフトウェア, Vol.36, No.4 (2019), pp.107–113.
[研究論文 (レター)] 2018 年 11 月 6 日受付.

レースには，重要な機能を持たないユーティリティ関数の実行の繰り返しなどが含まれる[3]．本研究ではそのような繰り返し実行される命令について記録するデータ量に上限を設けることで，限られた保存領域でプログラム全体の変数情報を記録する．具体的には，観測対象の命令ごとに大きさ k のバッファを個別に準備し，実行回数が k より少ない命令に対する観測値はすべて保存し，それを超えて実行された命令に関しては最新 k 回分の観測値のみを保持する．観測範囲となるプログラムを構成する命令の数を実行前に数え上げておくことで，開発者が保有する保存領域に応じて k の値を設定し，実行トレースを記録することが可能となる．

2 関連研究

2.1 実行トレースの削減手法

プログラムの実行トレースのデータ量がソースコード等と比べて大きくなる要因の 1 つは，プログラムが繰り返し同じ命令を実行することにある．実行トレースを削減するために，繰り返しの実行が互いに類似している，あるいは重要度が低いと仮定した，データ圧縮やサンプリングの適用が提案されている．

Wang ら[12] は，プログラムがアクセスしたメモリアドレスの系列からなる実行トレースを，データ圧縮によって効果的に保存する方式を提案している．プログラムの多くは繰り返し同じ命令を実行するが，それらの命令の多くが配列等の連続的なデータに対する同一の操作になることから，差分エンコーディングが高い圧縮率を達成できることを示した．しかし，この手法で記録可能なのはデータ依存関係と呼ばれるデータの流れのみであり，本研究のような変数の具体的な値の記録にはそのまま適用できない．また，圧縮後のデータの大きさがソフトウェアに依存するという課題も残されている．

Cornelissen ら[3] は，メソッド呼び出しの系列からなる実行トレースに対して，実際に使用できるデータ量の目標値が設定された場合，重要な機能を持たないユーティリティ関数の呼び出しの繰り返しを除去することが，単純なサンプリングよりも情報量を保つ傾向にあることを報告した．一方で，この研究は完全な実行トレースを保存した後，他の解析を行う前にフィルタリングを行うことを前提とした分析を行っていた．本研究では，繰り返しのデータの除去を，プログラムの実行中に行うことを提案している．

Hirzel ら[4] は，実行トレースの効果的なサンプリング方法として，データの観測を行う実行区間，行わない実行区間を切り替える Bursty Tracing 手法を提案した．この方式は，観測頻度を均等に低くする通常のサンプリングよりも，プログラムの実行経路について多くの情報を取得することが可能である．しかし，サンプリングで得られるデータはプロファイリング対象に大きく依存しており，デバッグに必要なデータを収集できない可能性もあるほか，収集するデータ量の上限をプログラムの実行前に設定することはできないという制約もある．

2.2 低オーバーヘッドの動的解析

実行トレースの収集には一定のコストがかかるため，特定の解析目的に対して，実行時間あるいは解析に伴うデータ量を最小限にするような解析技術が提案されている．たとえば Liu ら[8] は，バッファオーバーフローやメモリリークなどの欠陥検出を目的として，通常は軽量なメモリ監視を実行し，疑わしい挙動に対して詳細情報の収集を行う解析手法を提案している．Zhang ら[15] は動的にデータ依存関係解析を実行するにあたって，プログラムの実行トレースをそのまま保存するのではなく，実行中にプログラムスライスに変換しメモリに保持することで，記録量を大幅に削減できることを示している．これらの手法は，それぞれ，実行に伴うオーバーヘッドを削減するが，本研究が対象としているような命令の実行順序や具体的な変数の値の取得という目的とは異なる．

3 提案手法

本研究は，Java バイトコードの命令ごとに最新 k 回の実行に関する具体的なデータ値を記録する手法を提案する．最新の実行を残すことで，プログラムのクラッシュなどが生じた場合には，異常な動作の値が記録として残る可能性が高い．一方で，プログラムの実行開始時などに一度だけ実行されるような処理も

破棄されないため，プログラム実行時の環境設定などの確認も可能である．

3.1 実行トレースのモデル

本研究で取り扱う実行時情報は，Omniscient Debugging の実装の 1 つである松村らの手法 [9] に基づくものである．この手法は，Java プログラムの任意の時点での挙動を確認するために，メソッドの境界を超えて受け渡されたすべてのデータの値を Java バイトコードの命令単位で記録する．具体的には，メソッド呼び出し命令の実行開始および完了，メソッド実行の開始および完了におけるすべての実引数・仮引数・戻り値・例外の値，フィールドおよび配列の読み書き命令の実行における対象オブジェクト，添え字，読み書きされた値を記録する．ここで，オブジェクトについては，参照を区別するための ID 値を保存するものとする．また，引数や戻り値のないメソッドの呼び出しについては，呼び出しの開始や終了の発生を記録するために，それぞれ 1 個のデータを記録する．これらの値には，たとえばフィールドに書き込んだ値とそこから読み出した値の組のように，冗長な情報も含まれるが，複数のスレッドによる同時操作の発生可能性や，観測範囲外となるネイティブコードやライブラリからのアクセスの可能性も考慮して，それぞれ個別に記録を行う．ローカル変数の値は，メソッドの外部から受け取ったデータから計算によって再現可能であるため，引数等を除くローカル変数の操作は記録しない．

本研究では，このような実行トレースを観測値の系列として表現する．観測値はそれぞれ，観測地点の ID p，命令を実行したスレッドの ID t，観測された値 v の 3 つ組 $\langle p, t, v\rangle$ として考える．観測地点は，データ授受を引き起こした Java バイトコードの命令とデータの種別を識別する．1 つの命令でも，たとえば引数と戻り値という異なる値が観測されるものであれば，それぞれ異なる観測地点とみなす．

3.2 実行トレースの記録方法

提案手法は，観測地点 p が同一であるような観測値は最新の k 個だけを残すように実行トレースを記録するというものである．プログラムの実行中に，観測地点 p ごとに長さ k のバッファを用意して，観測値と，実行全体での観測順序の情報を，メモリ内に保持する．プログラムの終了時に，Java 仮想マシンのシャットダウンフック機構を利用して，蓄積したデータをまとめて保存する．なお，従来研究，たとえば松村ら [9] の実行トレースは，すべての観測値を時系列で単純に保存することに相当する．

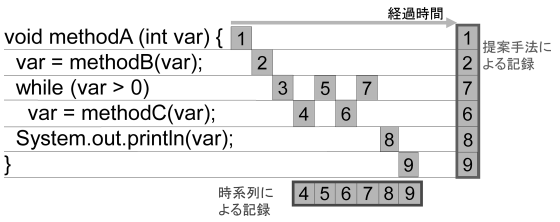


図 1 提案手法による記録イメージ

提案手法の特徴を図 1 の例を用いて説明する．プログラムの各行の右側にある 1 から 9 までの数字が，ある 1 回の実行における命令の実行順序であり，実行トレースとして保存する観測値を表現しているとする．実行トレースの保存領域が限られている場合，一般的なサーバ等での時系列に基づくロギングであれば，最新の観測値から順に保存領域の容量分だけ記録するため，保存容量の大きさが 6 であれば，図 1 の下側にあるように記録が行われる．それに対して，提案手法は同じ容量を，図 1 の右側にあるように，6 個の観測地点それぞれに対し，1 つの観測値を保存するために使用する．これにより，ループで繰り返し実行された命令の観測値は最新の値を除いて破棄することになるが，プログラムの先頭部分などの実行回数が少ない観測地点における情報を完全に保持する．

プログラムが持つ観測地点の数 m はプログラムの Java バイトコードから静的に数え上げができるため，実行トレースの保存に必要な保存領域の大きさ $N = k \times m$ を事前に知ることができる．Java では動的にバイトコードを生成する技術も利用されているが，通常の利用であればプログラム中の大きな割合を占めることはなく，m に余裕を持たせた見積もり値の設定が可能であると考えている．

表 1 ベンチマークプログラムおよび実行トレースの規模

	クラス	メソッド	観測地点数	実行された観測地点数	観測値の総数	データ量
avrora	527	3,456	87,736	38,772	7,880,412,751	117.4 GB
batik	1,122	9,163	218,775	72,630	601,350,099	9.0 GB
fop	1,198	10,401	318,795	127,226	325,806,544	4.9 GB
jython	2,244	23,342	670,054	214,300	5,461,124,807	81.4 GB
luindex	231	2,467	73,094	25,838	1,542,871,620	23.0 GB
lusearch	199	2,140	58,127	16,256	5,503,253,333	82.0 GB

提案手法の実装は GitHub に公開されている †1．この実装では，Java プログラムがクラスをロードする際にバイトコード変換を実行し，対象プログラムに対して観測地点 ID を割り当てながら，観測のための命令を埋め込む．観測地点 ID およびスレッド ID の保存にそれぞれ 4 バイト，値の保存に 8 バイトを使用するため，1 つの観測値あたり 16 バイトの保存領域を必要とする．実際にはプログラムの命令情報など，付加的な情報も出力するが，それらの量はプログラムとしてロードされたクラスの数，バイトコード命令の量にのみ依存して決まる．実行時には順序の保存のために 8 バイトのインデクスを同時に保存するため，記録する観測値 1 つあたり 24 バイトのメモリと，それらを管理する配列等を使用する．

文字列データに関しては，観測値として記録されたオブジェクトについて，一定の長さまでの固定長の記録と内容のハッシュ値の記録によって，一定の範囲で有用な情報を保存することが考えられる．ただし，プログラム中で使用される文字列の種類は多岐にわたるため，文字列の適切な保存方法は今後の課題とし，現在の実装では，文字列の内容に関しては可変長データを用いて完全に保存するようになっている．

4 評価

提案手法によって記録される実行トレースが，限られた保存領域でどれだけの情報を保存可能か，以下の 2 つの観点から評価する．

1. 観測値が完全に保存される観測地点の割合
2. データ依存関係が保存される割合

前者は開発者がランダムに観測地点を選んだ場合にその地点に関するデータがすべて揃っている確率であり，事前にどのデータを必要とするか分からない状況での有用性を評価する．後者は，実行トレースを系列としてみたときに，データの流れに関する順序関係が正しく保存されているかどうかを評価する．

実行トレースの計測には，DaCapo Benchmarks [1] のバージョン 9.12-bach に収録された 14 個のベンチマークのうち，動作が確認できた 6 個を使用した．実行トレースは，ベンチマーク内部での標準ライブラリへの呼び出し等を含んでいるが，Java 標準ライブラリ内部の動作は記録していない．表 1 に，各ベンチマークのプログラム規模と実行トレースの大きさを示す．クラス数は，ベンチマークの実行のためにロードされたクラスから，Java 標準ライブラリ (`java`, `javax`, `sun` などのパッケージ名で識別されるもの) を除いた数である．メソッド数，観測地点数は，それらのクラスに対して数え上げた値であり，実行されなかったメソッド，命令に対応するものも含んでいる．観測値の総数は，完全な実行トレースを記録した場合の値であり，データ量は 1 観測値 16 バイトとして完全な実行トレースを保存した場合の値である．

提案手法のパラメータである k の値として $16, 32, 64, 128, 256$ を使用する．それぞれの k の値に対しての提案手法の記録データ量を図 2 に示す．図中の ALL は，完全な実行トレースを保存した場合のデータ量である．提案手法は観測地点 1 つあたり 16 バイトの観測値を k 個保持することから，観測地点数が最大の jython を $k = 256$ で実行した場合で，観測地点数 $670,054 \times 256 \times 16 = 2.7 \times 10^9$ となり，最大で約 2.6GB の実行トレースを収集することになるが，すべての観測地点が実行されるわけではないために，実際の値はそれよりも小さくなっている．提案手法で $k = 256$ とした際の記録データ量は，完全な実行トレースを記録した場合と比較して平均で 1.0 %以下

†1 https://github.com/takashi-ishio/selogger, develop branch, commit 2519626.

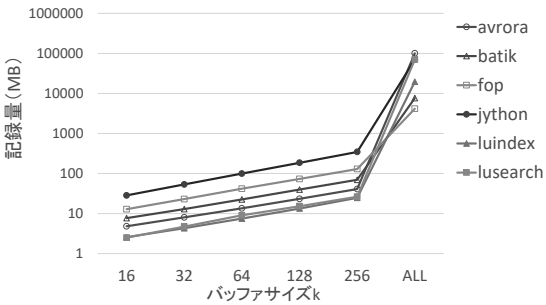


図 2 提案手法の実行トレースのデータ量

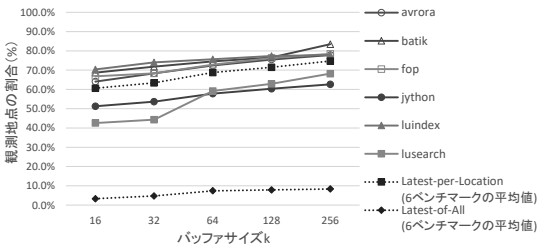


図 3 観測値が完全に保存される観測地点の割合

である．

使用できる保存領域が制限された環境での性能のベースラインとして，時系列に基づく方式，すなわち実行トレース全体における最新 N 個の観測値を保存する方法を用いる．以降，提案手法を Latest-per-Location，ベースラインとなるこの方式を Latest-of-All と記載して区別する．

4.1 観測値が完全に保存される観測地点の割合

観測地点ごとに保存する観測値の個数 k を変化させたとき，それぞれのベンチマークで実行された観測地点のうち，完全なデータを保存できる (すなわち実行回数が k 以下である) 観測地点の割合の変化を図 3 に示す．また，同図は Latest-per-Location，Latest-of-All の 2 手法における 6 つのベンチマークの値の平均も示している．

提案手法である Latest-per-Location は，$k = 16$ のときで 60%，$k = 256$ のときで 74%の命令に関して完全なデータを保存し，残る命令に関しては最新の k 回のデータを保存する．したがって，従来手法では巨大な実行トレースとなるようなプログラムの実行であっても，1%以下の限られたデータで，多くの文について完全な情報を参照することができる．これに対して Latest-of-All は，約 10%の命令に関してのみデータを保存しており，ほとんどの文に関しては，利用者は完全な観測値を参照できない．

4.2 データ依存関係の正確さ

提案手法がデータの流れに関する順序関係を正しく保存しているかどうかを評価するために，提案手法の実行トレースから算出したデータ依存関係を完全な実行トレースから算出したデータ依存関係と比較する．ここでのデータ依存関係は，フィールドあるいは配列 (ヒープ領域) を経由してデータを受け渡す，代入命令と参照命令の組とする．なぜなら，これらは単純なメソッド呼び出しの系列やメソッド内部の静的解析からは得られないためである．フィールドに関しては，代入命令 d がオブジェクト obj のフィールド f に対して値 v を書き込んだ後，その値が上書きされることなく参照命令 u で読みだされたとき，つまり u がオブジェクト obj のフィールド f から値 v を読みだしたときに，d から u へのデータ依存関係が存在すると考える．配列に関しても同様に，読み書きされた添え字をフィールド f のかわりに用いて，個々の要素を区別したデータ依存関係を抽出する．書き込んだ値と読み込んだ値を比較することで，代入命令の実行が記録されなかった場合や，実行の観測範囲外のコードから値の書き換えが行われた場合などに対応する．

完全な実行トレースから求めたデータ依存関係の集合を正解として，Latest-per-Location と Latest-of-All の 2 手法で得られた実行トレースから得られるデータ依存関係の適合率と再現率，F 値を求めた．表 2 に，すべてのベンチマークから得られたデータ依存関係の和集合を用いて算出した結果を示す．

Latest-per-Location は，繰り返し実行されるデータ読み書き命令の実行の一部を記録しないため，それによるデータ依存関係の誤検出や欠落が発生する．特に，オブジェクトの状態遷移を表現するグローバル変数のように，多数の場所で，ある一定の範囲の値の代入，参照が行われるようなフィールドが，多数の誤検出

表 2　提案手法によって得られるデータ依存関係の正確さ (全ベンチマークの合計)

k	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
16	52,061	42,603	0.914	0.748	0.823	3,010	1.000	0.058	0.109
32	52,061	45,115	0.903	0.782	0.838	3,144	1.000	0.060	0.114
64	52,061	47,472	0.892	0.813	0.851	3,938	1.000	0.076	0.141
128	52,061	49,826	0.881	0.843	0.861	4,043	1.000	0.078	0.144
256	52,061	51,941	0.872	0.870	0.871	4,054	1.000	0.078	0.144

表 3　提案手法によって得られるデータ依存関係の正確さ (ベンチマーク別, $k = 16$)

ベンチマーク	正解の依存関係	Latest-per-Location				Latest-of-All			
		依存関係	適合率	再現率	F 値	依存関係	適合率	再現率	F 値
avrora	4,325	3,269	0.904	0.683	0.778	289	1.000	0.067	0.125
batik	7,054	6,366	0.951	0.858	0.902	50	1.000	0.007	0.014
fop	13,906	12,143	0.968	0.845	0.902	2,149	1.000	0.155	0.268
jython	20,460	15,342	0.849	0.637	0.728	137	1.000	0.007	0.013
luindex	4,098	3,699	0.952	0.859	0.903	385	1.000	0.094	0.172
lusearch	2,218	1,784	0.915	0.736	0.816	0	N/A	0.000	N/A

と欠落の原因となる．たとえば，ベンチマーク avrora に含まれる avrora.arch.legacy.LegacyInterpreter クラスは，与えられたプログラムを実行するインタプリタの実装であり，100 以上のメソッドがプログラムカウンタに関するフィールド (pc, $nextPC$) を繰り返し操作していた．それぞれの代入，参照命令の実行回数が多いことから，Latest-per-Location では命令の実行順序が十分に保存されず，多数のデータ依存関係が欠落するとともに，同一の値を使用した異なる命令の実行を誤ってデータ依存関係として検出することにつながった．ただし，このような誤検出の要因は少数のフィールドに限られており，全体での適合率は 0.9 以上，再現率は 0.8 前後と，保存するデータを 1%に抑えた状態でも高い数値を保つことが確認できた．一方で Latest-of-All は最新の実行トレースのみを保持するため，誤ったデータ依存関係が認識されることはないが，プログラム全体でみるとほとんどのデータ依存関係を破棄してしまうため，プログラム全体の調査には不向きである．

表 3 は，$k = 16$ におけるベンチマークごとの詳細な結果である．k の値が変わってもベンチマークごとの傾向は変わらなかったため，他の k の値に関する結果は省略する．この結果から，どのベンチマークに関しても提案した実行トレースの削減手法が高い数値を示しており，Latest-per-Location が実行トレースの削減手法として有望であることが分かる．なお，lusearch の適合率・F 値が N/A となったのは，Latest-of-All の手法ではフィールド・配列に関するデータ依存関係が得られなかったためである．

5　まとめと今後の課題

本研究では，ソフトウェアの詳細な内部状態の観測を行う手段として，使用する保存領域を制御可能な実行トレース記録手法を提案した．観測地点ごとに最新 k 個の観測値を保持することで，完全な実行トレースの 1%程度のデータ記録量でも，60%から 74%の観測地点に対して完全な観測値を保持し，また，フィールドや配列に関するデータ依存関係の多くを追跡可能である．

本研究の結果をもとに，開発者が多くの環境で常に実行トレースを収集し続けられるようなデバッグ環境を構築することが今後の課題である．また，デバッグに適した適切な文字列データの記録方法の実現や，データ依存関係などをより高い精度で保存できるように静的解析を活用することも今後の課題として挙げられる．

謝辞　本研究は科研費 JP18H04094，JP18H03221，JP15H02683 の助成を得た．

参考文献

[1] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, in *Proc. of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 169–190.

[2] Cleve, H. and Zeller, A.: Locating Causes of Program Failures, in *Proc. of the 27th International Conference on Software Engineering*, ACM Press, 2005, pp. 342–351.

[3] Cornelissen, B., Moonen, L., and Zaidman, A.: An assessment methodology for trace reduction techniques, in *Proc. of the 24th IEEE International Conference on Software Maintenance*, 2008, pp. 107–116.

[4] Hirzel, M. and Chilimbi, T.: Bursty Tracing: A Framework for Low-Overhead Temporal Profiling, in *Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001, pp. 117–126.

[5] Johnson, N. M., Caballero, J., Chen, K. Z., McCamant, S., Poosankam, P., Reynaud, D., and Song, D.: Differential Slicing: Identifying Causal Execution Differences for Security Applications, in *Proc. of the 32nd IEEE Symposium on Security and Privacy*, IEEE, 2011, pp. 347–362.

[6] Kabinna, S., Bezemer, C.-P., Shang, W., and Hassan, A. E.: Logging Library Migrations: A Case Study for the Apache Software Foundation Projects, in *Proc. of the 13th International Conference on Mining Software Repositories*, 2016, pp. 154–164.

[7] Lewis, B.: Debugging Backwards in Time, CoRR, cs.SE/0310016, 2003.

[8] Liu, T., Curtsinger, C., and Berger, E. D.: DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis, in *Proc. of the 38th International Conference on Software Engineering*, 2016, pp. 911–922.

[9] 松村俊徳, 石尾隆, 鹿島悠, 井上克郎: REMViewer: 複数回実行された Java メソッドの実行経路可視化ツール, コンピュータ ソフトウェア, Vol. 32, No. 3 (2015), pp. 137–148.

[10] Pothier, G., Tanter, E., and Piquer, J.: Scalable Omniscient Debugging, in *Proc. of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, 2007, pp. 535–552.

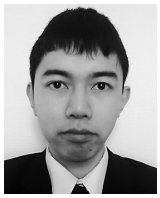
[11] Spinellis, D.: *Effective Debugging: 66 Specific Ways to Debug Software and Systems*, Addison-Wesley Professional, 2016.

[12] Wang, T. and Roychoudhury, A.: Using Compressed Bytecode Traces for Slicing Java Programs, in *Proc. of the 26th International Conference on Software Engineering*, 2004, pp. 512–521.

[13] Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S.: Improving Software Diagnosability via Log Enhancement, *ACM SIGARCH Computer Architecture News*, Vol. 39, No. 1 (2011), pp. 3–14.

[14] Zeller, A.: *Why Programs Fail*, Morgan Kaufmann, second edition, 2009.

[15] Zhang, X., Gupta, R., and Zhang, Y.: Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams, in *Proc. of the 26th International Conference on Software Engineering*, 2004, pp. 502–511.



嶋 利一真

2017 年大阪大学基礎工学部情報科学科卒業．現在，大阪大学大学院情報科学研究科博士前期課程 2 年．プログラム解析に関する研究に従事．



石尾　隆

2003 年大阪大学大学院基礎工学研究科博士前期課程修了．2006 年同大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員 (PD)．2007 年大阪大学大学院情報科学研究科助教．2017 年奈良先端科学技術大学院大学情報科学研究科准教授．2018 年奈良先端科学技術大学院大学先端科学技術研究科准教授．博士 (情報科学)．プログラム解析，プログラム理解に関する研究に従事．



井上克郎

1979 年大阪大学基礎工学部情報工学科卒業．1984 年同大学大学院博士課程修了．同年同大学基礎工学部助手．1984〜1986 年ハワイ大学マノア校情報工学科助教授．1989 年大阪大学基礎工学部講師．1991 年同助教授．1995 年同教授．2002 年大阪大学大学院情報科学研究科教授．工学博士．ソフトウェア工学，特に，ソフトウェア開発手法，プログラム解析，再利用技術の研究に従事．