様　式　Ｃ－１９、Ｆ－１９－１、Ｚ－１９　（共通）

科学研究費助成事業　　研究成果報告書

令和　　元 年　　６ 月 ２５ 日現在

機関番号：１４６０３
研究種目：挑戦的萌芽研究
研究期間：2015～2018
課題番号：１５Ｋ１２００９

研究課題名（和文）ブレイクポイントを用いないデバッグを可能とする捜査地図デバッガの実現

研究課題名（英文）An Omniscient Debugger with novel visualization called "Investigation Map"

研究代表者
久米　出（Kume, Izuru）

奈良先端科学技術大学院大学・先端科学技術研究科・助手

研究者番号：１０３０１２８５

交付決定額（研究期間全体）：（直接経費）　　2,800,000円

研究成果の概要（和文）：ソフトウェア開発ではプログラムの実行を失敗させる記述上の誤り(不具合)を特定・修正する、デバッグと呼ばれる作業が不可欠である。デバッグは通常多大な時間と労力を要するため、効率化のための新しい技術が必要である。
本研究は「全知デバッガ(Omniscient Debugger)」と呼ばれる新しいデバッグ支援ツールを開発し、作業の効率化の効果を評価する事を目的としている。本デバッガはプログラムの実行履歴(トレース)を「操作地図」と呼ばれる新しい方式で可視化する点に大きな特徴を有している。開発に当たって技術的に困難な目標に遭遇するも、新しいソースコード変換手法を開発する事によって問題解決を実現した。

研究成果の学術的意義や社会的意義
本研究ではデバッガの開発と並行して既存のデバッガを用いた作業の分析も実施した。これによって既存のデバッガの問題点を整理し、トレースをデバッグにより効果的に利用する知見を得るに至った。この知見は我々のデバッガの仕様策定に反映されている。本デバッガの仕様には実行時点を命令文や式の単位の細かい粒度で指定する機能が含まれている。これによって、複雑な式の値を容易に取得したり、自身が関心を有する実行時点同士の関連性を可視化によって把握する事が可能となる。この新しい機能によって従来のデバッガより遥かに少ない操作で複雑な制御やデータ構造の調査が可能となり、デバッグ作業の大幅な効率化が実現される。

研究成果の概要（英文）：Debug is a software development task to detect and correct defects in a program, which make its execution fail. Because debugging practical programs is usually time consuming, a new supporting technology is necessary. The goal of our study is to develop and evaluate a new omniscient debugger for Java programs. Our debugger enables developers to efficiently examine program traces by a novel visualization, called "Investigation Map".
We are still on the way to the completion of our development because of an unexpected technical problem to implement our visualization. We found that it is very difficult to locate Java byte code instructions at statements and expressions instead of source code lines. This problem comes from the specification of Java class files. We have resolved this problem by introducing a novel Java source code translation technique.

研究分野：ソフトウェア工学

キーワード：デバッガ　ソフトウェア開発　動的解析　プログラムトレース　プログラム変換　可視化　Java 言語　バイトコード

様　式　Ｃ－１９、Ｆ－１９－１、Ｚ－１９、ＣＫ－１９（共通）

１．研究開始当初の背景

ソフトウェア開発では様々な作業が実施される。近年様々な支援手法・ツールが開発され、多くの作業の効率化が実現されるようになったが、デバッグに関しては依然として効率化の恩恵を受けられない状況が続いている。実用的なプログラムはその構造も処理の過程も極めて複雑であり、その複雑性がデバッグ作業の効率化を阻害する大きな要因となっている。
ソフトウェア開発では通常デバッガと呼ばれる支援ツールが利用される。既存のデバッガはブレイクポイントを指定して実行を停止させ、停止させた実行時点に於けるプログラムの状態の調査を可能とする。こうした従来のデバッガの支援機能は極めて限定されたものであるため、作業者はプログラムの実行過程の点(実行時点)と点をその内面で結びながら状態の異常や不具合箇所を特定しなければならない。
こうした従来のデバッガの制約を乗り越えるべく、プログラムのトレース(実行履歴)を記録し、作業者による参照を可能とする「全知デバッガ(Omniscient Debugger)」と呼ばれる新しい思想のデバッガが注目されてきた。しかしながら現在ではこれらの研究は下火になってしまっている。下火になった主な原因の一つとしてしばしばトレースのデータ量の問題が挙げられる。全知デバッガが注目され始めた時期はハードウェアやソフトウェアの性能があまりにも低かったため、実用的なプログラム実行が膨大な量のトレースを安価な方法で処理する事が困難であった。結果としてデバッグ支援にトレースを利用する事によって作業の効率化に如何程の効果が有るのかが明らかにされていないまま、本分野の研究は決して活発とは言えない状況が続いている。

２．研究の目的

研究の目的は以下の二つである。

- 全知デバッガの実用性を示す
- 全知デバッガによる支援の効果を明確にすると共にそれを定量的に評価する

全知デバッガを実現する上で最大の障害はトレースのデータ量とされてきた。しかしながら、我々は過去の研究成果より実用的なプログラムであっても生成されるトレースのデータ量は必ずしも膨大なものにならない事を知っていた。さらに障害を再現するテストケース等を適切に導入する事によって障害発生に至る過程のトレースのデータ量を抑える事も可能だと考えている。さらにビッグデータに関する昨今の研究の進展に伴い、かつては現実的な不可能であったトレースの処理を可能にするデータ工学的な技術が確立されている。

より重要な事はデバッグにトレースを利用する利点と効果の明確化にあると我々は考えている。全知デバッガの利点として「過去の実行を遡って追跡出来る」事実がしばしば喧伝されてきた。既存のデバッガでは現在停止させている実行時点より以前の実行を調べるためにはプログラムを再度起動し直さなければならないが、全知デバッガにはそうした操作を必要としない。しかしながら問題はこの操作の軽減がプログラムの構造や実行の複雑性に対して如何程の効力を発揮しているのか、その効果そのものを定量的に評価する枠組みは確立されているとは言い難い。本研究はこうした問題の解決を目的として開始された。

３．研究の方法

研究は以下の二つの作業を可能な限り同時並行的に実施しながら推進した。

- デバッグ作業の分析
- 全知デバッガの開発

開発した全知デバッガを用いた評価実験を通じてトレースをデバッグに利用する効力を評価する予定であったが、技術的な問題の発生によってデバッガの開発を完遂出来なかったため、評価実験を実施出来なかった。

デバッグ作業の分析に当たっては現実のデバッグ作業の記録を取得し、ソフトウェア開発の実務者の知見も踏まえて作業を進めた。分析は作業者の内面の表出や画面の操作に止まらず、実行内容とも関連付ける形で進められた。こうした関連付けは作業者の対象プログラム(及びその実行)の理解の適切性や作業の効率性を定量的に評価するために我々が導入したものである。分析を重ねる事によってその有効性を示すとともに、既存のデバッガの機能の制約の影響を明らかにし、我々のデバッガの実現すべき機能の検討にも分析結果を反映させる事を意図していた。

我々の全知デバッガはトレースを利用して単純に過去の実行を調査するだけでなく、作業者の各作業時点に於ける関心を反映させる形でトレースを抽象的に可視化するユーザインタフェイ

ス有する予定であった。我々はこのユーザインタフェイスを「捜査地図」と呼ぶ。本デバッガの作業者は実行時点をある特定の命令文(statement)や式(expression)、あるいはその部分式の単位で指定出来る。指定した実行時点のうち、自身が関心を有するものをデバッガに指定出来る。捜査地図は、作業者が現在調査している実行時点と先に指定した実行時点の関連を可視化する。これによって例えば「実行時点 A に於いて ++x が実行されたために実行時点 B のメソッド呼び出しが本来の回数より 1 余分に繰り返されている」という関係を図式として提示する。現在のデバッガではこのような関係を特定するまで多大な操作が必要とされる。操作地図を導入する意図はこうした調査のための操作数を大幅に削減する事にある。

４．研究成果

研究成果として以下の三点が挙げられる。

- 全知デバッガの実用性に関する確認
- デバッグ作業者の内面に関する知見の獲得
- 捜査地図の実装に必要な技術上の知見の獲得

[実用性の確認]

グラフデータベースシステム(neo4j)を導入する事により、大容量のトレースの処理が可能である事を確認した。市販の PC 上でデータベースシステムを対象としたベンチマーク用のプログラムを実行し、2 ギガ程度のトレースの生成と処理が可能である事を確認した。現在の実装ではトレース生成時にデータベースを利用していないため、生成出来るトレースのデータ量には限界が有る。生成部分の実装にデータベースを組込む事によってより大量のトレースの生成が可能となる見込である。

[内面に関する知見]

実用的なプログラムのデバッグの進め方として、プログラム実行が失敗した段階でその失敗を再現するテストケースを作成し、その実行を調べる事によって不具合を特定する事が望ましいとされている。我々は現実のソフトウェア開発の過程で実際にこうした作業が実施された事例を記録し、作業時の発話や捜査を作業後の作業者の記憶と照らし合わせる事によってその内面を分析した。デバッグの対象は全て Java 言語で記述されたプログラムであり、作業者は統合開発環境 Eclipse の標準のデバッガを使用している。現実のソフトウェア開発時に発生した障害を対象としているため、不具合箇所の所在は作業が完了して初めて明らかにされる。

上記の形で作業記録を収集したため作業に要する時間は事例毎にそれぞれ異なっているが、凡そ 30 分から 1 時間の間に収まるものを選択して分析した。分析作業では、まず作業者の発話と操作動画からこれらの意図と目的をそれぞれ分類した。そしてその分類に基づいて一連の操作によって遂行される作業(タスク)の特定を試みた。特定が可能であった作業に関してその適切性を評価した。また各作業に対して作業者が追跡していたプログラム実行の内容に関しても特定を試みた。

上記の分析作業は一つの事例毎に多大な労力と時間を要するため、実際に分析を実施した作業記録は数件に過ぎない。しかしながらその分析結果は既存の内面分析の研究で得られた知見との間の整合性が見出されているため、これが我々の分析の妥当性を示していると考えられる。

我々が最も関心を有していたのは作業者が追跡したプログラム実行の内容と、その追跡作業の妥当性であった。追跡された時の内面の忠実な再現は実現出来なかったが、プログラムコードの特定箇所で特定のデータ構造(List 型データの内容)を何度も確認する等、作業者が関心を有する実行時点を特定する事には成功した。作業の妥当性に関して、作業者は(結果として)必ずしも必要ではないデータ参照を何度も繰り返すという傾向を確認出来た。デバッグ時の内面分析に関する既存の研究によって、デバッグ作業者の短期記憶として保有出来る情報量は極めて限定される事が知られている。上記の必ずしも適切でないデータの参照はこうした短期記憶の制約が原因であると推測出来る。

我々の分析では作業者は少なからぬ数の操作によって完遂される同じような作業を何度も繰り返していた。本分析で用いられた Java プログラムは我々が分析のために準備したものではなく、現実のソフトウェア開発で作成されたものである。よってプログラム実行時の制御やデータは複雑な構造を有している。こうした複雑な構造が操作数を増大させていると考えられる。こうした操作数の増大はデバッグに要する労力と時間に直接関係する要因である。

作業の途中でしばしば作業者は操作を行わずに内面でプログラムの実行を再現している。この

時作業者は変数値だけでなく数値演算や変数のインクリメント前後の値等、より複雑な式の値も内面で計算している事が判明した。変数値と異なりデバッガを用いて一般の式の値を直接確認するためには、条件付ブレイクポイント等操作の手間を要する方法に依るしかない。そのため作業者はしばしばこうした内面の作業を遂行する事になる。先に述べた短期記憶の制約によりこうした内面の作業はデバッグ作業者の負担を増大させ、不必要なデータ参照の回数を増大させると考えられる。

分析によって得られた知見:
実用的なプログラムの複雑性とデバッグ作業者の短期記憶の制約により必ずしも適切ではないデータの参照のために同じような操作が何度も繰り返され、それによってデバッグに要する労力と時間が増大していると考えられる。短期記憶の制約を解消し、内面に於ける実行の再現を不要とし、複雑性に起因する操作回数の増大を抑制するという点で、我々が開発するデバッガの仕様が合目的である事を確認出来た。

[捜査地図の実装に関する知見]

我々は全知デバッガを Web アプリケーションとして開発を進めている。デバッグの対象となる言語は Java である。本デバッガのサーバ側はトレースの処理と捜査地図のモデルに相当するデータ生成を担当し、このデータがブラウザ側に渡されて Web ページとして表示される。サーバは Java 言語で、ブラウザ上の表示は JavaScript によって実装される。

我々のデバッガの利用者はソースコード上の任意の命令文や式の形で実行時点を指定する。Java プログラムの実行トレースはバイトコード命令の実行列として表現される。よって捜査地図の要求仕様にはトレース中のバイトコード命令をソースコード中の命令文/式と対応付ける機能が含まれている。この機能の実装は技術的な困難を伴うものであり、結果として研究を完遂するために研究期間を一年延期せざるを得なかった。延期された期間内で本機能を実装するライブラリの開発を進め、独自の Java ソースコード変換手法を開発して完成にこぎつけるに至った。本報告書を作成している時点でライブラリの最終的なテストを遂行している段階である。

一般にデバッガは実行可能なコード(例: Java のバイトコード)とソースコードの記述を対応付ける事が求められる。対応付けに必要な情報はコンパイラによって与えられるため、デバッガの対応付けは実行可能コードを生成したコンパイラに大きく依存する。標準的な Java コンパイラは対応付けに必要な情報としてソースコードの行番号しか与えないため、より粒度の細かい命令文や式をバイトコード命令に対応付ける事は困難であった。実際、行番号より細かい粒度(命令文や式)の単位でバイトコード命令との対応を実装している実用的な Java デバッガは我々の知る限り存在しない。

我々は型推論を伴わない構文解析とソースコード変換を用いて、元のソースコードとバイトコード命令の対応付けを実現した。変換は基本的には行番号と式を対応付けるために改行を入れるだけであるが、コンパイラによっては我々の予想を超えた不適切な行番号の割り当てを実施する場合がある。本研究ではこうした不適切な行番号の割り当てを「誤誘導(misdirection)」と呼ぶ。我々の対応付けライブラリは Eclipse の Java Compiler に対して誤誘導のパターンを特定し、自動的に修正する機能を実現している。我々が用意したテスト用の例題に対して対応付けを実施している事例を以下に示す。

```
package org.lines;

public class CombinedOperations2 {

  public static void main(String[] args) {

    int x = args.length;
    int y = x;
    x =  (x+2)*(y+x);
  }
}
```

上記の Java プログラムには配列長参照、数値演算、変数代入が含まれている。コンパイラが生成したバイトコードの実行に対して以下に示すように構文要素が対応付けられる。

```
[0]invoke org.lines.CombinedOperations2.main([Ljava/lang/String;)V

    [14]load  --> args
    [10]arraylength  --> args.length
    [18]store  --> x = args.length
    [19]load  --> x
    [20]store  --> y = x
    [21]load  --> x
    [27]constant 2  --> 2
    [29]iadd  --> x+2
    [31]load  --> y
    [33]load  --> x
    [35]iadd  --> y+x
    [37]imul  --> (x+2)*(y+x)
    [39]store  --> x =    (x+2)*(y+x)
    [40]return  --> <No Location>
```

行頭の数値はバイトコード命令の実行毎に割り振られた ID 値である。これらの ID はデバッガの内部で特定の実行時点を参照するのに用いられる。次にバイトコード命令の記号表現(mnemonics)が続く。局所変数の参照や代入を表す正式な記号表現は例えば iload のように先頭に変数のデータ型を表すアルファベットから始まるのであるが、上記の表記ではこうした型の差異を示さない記法が用いられている。矢印 [-->] の後に各バイトコード命令に対応付けられる構文要素が示されている。例えば [14]load に対しては配列を参照する変数 args が対応付けられている。

バイトコード命令とソースコードの構文要素の対応付けの実現に際して、研究当初予想していなかった技術的な問題のために研究期間内で捜査地図の実装を完成させる事は出来なかった。この対応付けは捜査地図の根本である実行時点の可視化の実現には不可欠な要素であり、この問題を避けて通る事は出来なかった。研究期間を一年延長した事によりこの技術的な難題を解決出来た事はソフトウェア工学上の大きな成果であると考えられる。時間の制約によって止むを得ず今回の対応付けは Eclipse のコンパイラのみに限定されている。今後は他の実用的な Java コンパイラに対しても誤誘導の特定と修正を試みる予定である。

５．主な発表論文等

〔雑誌論文〕（計 2 件）

① Kazuma Kusu, Izuru Kume, Kenji Hatano, A Graph Partitioning Approach for Efficient Dependency Analysis using a Graph Database System, 査読有り, International Journal on Advances in Networks and Services 10(3&4) 82-91 Dec 2017

② Izuru Kume, Masahide Nakamura, Naoya Nitta, Etsuya Shibayama, A Case Study of Dynamic Analysis to Locate Unexpected Side Effects Inside of Frameworks, 査読有り, International Journal of Software Innovation (IJSI) 3(3) 26-40 Sep 2015, https://doi.org/10.4018/IJSI.2015070103

〔学会発表〕（計 11 件）

① Izuru Kume, Etsuya Shibayama, Masahide Nakamura, Naoya Nitta, Cutting Java Expressions into Lines for Detecting their Evaluation at Runtime, 査読有り, 2nd International Conference on Software and Services Engineering(ICSSE 2019) Mar 2019, https://doi.org/10.1145/3318236.3318259

② 久米 出、新田直也、柴山 悦哉、中村 匡秀, Java デバッガによる式の監視機能の必要性と実現に関して, 査読無し, ウィンターワークショップ 2019・イン・福島飯坂 2019 年 1 月

③ Izuru Kume, Masahide Nakamura, Naoya Nitta, Revealing Implicit Correspondence between Bytecode Instructions and Expressions Determined by Java Compilers, 査読有り, 25th

Australasian Software Engineering Conference (ASWEC) and Australasian Software Week (ASW) Nov 2018, https://doi.org/10.1109/ASWEC.2018.00025

④ 久米 出、中村 匡秀、新田 直也、柴山 悦哉，デバッグ作業者の内面分析支援を目的とした障害発生過程の実体化手法，査読無し，日本ソフトウェア科学会大会 第 34 回大会 予稿集 2017 年 9 月

⑤ Izuru Kume, Masahide Nakamura, Naoya Nitta, Analyzing execution traces of failed programs for materializing chain of infection, 査読有り,2nd International Conference on Big Data, Cloud Computing, and Data Science Engineering (BCD 2017) Jul 2017

⑥ Kazuma Kusu, Izuru Kume, Kenji Hatano, A Node Access Frequency based Graph Partitioning Technique for Efficient Dynamic Dependency Analysis, 査読有り, The Ninth International Conferences on Advances in Multimedia (MMEDIA 2017) 73-78 Apr 2017

⑦ 楠 和馬、久米 出、波多野 賢治，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，査読無し，第 9 回データ工学と情報マネジメントに関するフォーラム （第 15 回日本データベース学会年次大会） 2017 年 3 月

⑧ Kazuma Kusu, Izuru Kume, Kenji Hatano, A Trace Partitioning Approach for Memory Efficiency on a Trace Analysis Environment, 査読有り, 4th International Conference on Applied Computing & Information Technology (ACIT 2016) Dec 2016, https://doi.org/10.1109/ACIT-CSII-BCD.2016.036

⑨ 楠 和馬、久米 出、波多野 賢治，デバッグ支援を目的とした大域的動的依存性解析の効率化，査読無し，FIT2016 情報科学技術フォーラム 講演論文集第 1 分冊，B-017 2016 年 9 月

⑩ Izuru Kume, Masahide Nakamura, Yasuyuki Tanaka, Etsuya Shibayama, Evaluation of diagnosis support methods in program debugging by trace analysis: An exploratory study, 査読有り, 15th IEEE/ACIS International Conference on Computer and Information Science 1-6 Jun 2016, https://doi.org/10.1109/ICIS.2016.7550875

⑪ 久米 出、中村 匡秀、田中 康之、新田 直也、柴山 悦哉，デバッグ時の診断作業の質的な分析に向けた事例研究，査読無し，日本ソフトウェア科学会第 33 回大会, FOSE3-2 2016 年 9 月