Java 全知デバッガを用いた複雑な繰り返し実行のデバッグを効率的に支援する繰り返し要約機能

久米 出　新田 直也　中村 匡秀　柴山 悦哉

全知デバッガはプログラムのトレース (実行履歴) を用いてプログラム実行をやり直す事無く過去の実行内容を調査する機能を実現する。しかしながら、繰り返し命令文の実行に伴う複雑な制御やデータの流れを効率的に調査するためには、この機能だけでは不十分である。

本論文では Java プログラムの繰り返し命令文の各回の処理内容を、作業者が指定した式の値や評価の時機によって抽象化する機能である、**繰り返し要約**を提案する。繰り返し要約は我々が過去に開発した、トレース中のバイトコード命令をソースコード内の式や命令に位置付ける技術を用いて実現される。本論文ではプログラミングコンテストの答案として作成された例題プログラムを用いて繰り返し要約の用法と効果を説明する。

Omniscient Debuggers process execution traces in order to implement a feature which enables back-in-time step examination without rerunning programs under debug. This feature is useful but its sole use is not so effective for efficient examination of complex control and data flow by loop execution.

In this paper, we propose *Iteration Summary*, a novel debug feature for Java Programs. This feature abstracts each of iterated processes in a loop execution based on the runtime values of expressions which are specified by developers. The feature is implemented based on our technology that locates executed bytecode instructions at expressions. In order to explain the use and effects of Iteration Summary, we show a case study to apply it to an example Java program which has been developed as an answer to a problem of a programming contest.

1　はじめに

現在広く用いられているデバッガは利用者がブレイクポイントで指定した行でプログラム実行を一時停止し、実行時の状態の調査を可能としている。デバッガは停止時より以前の実行内容は記録していないため、例えばループ命令の以前の繰り返し実行を調査する為にはしばしばプログラムの再実行、及び場合によってはブレイクポイントの条件等の再設定が必要となる。この問題を解決するために、トレース (プログラムの実行履歴) を用いて任意の実行時点の参照や実行時点間の移動を可能とする、全知デバッガ [12] [6] [13] [7] [14] [16] [18] の研究が進められている。

全知デバッガは実行時点の容易な移動を可能としているが、複雑な処理を効率的にデバッグするためにはこれとは異なる種類の支援機能が必要となる。本論文では特に Java プログラムの繰り返し命令文の実行を対象としてこの問題を考える。多数回の繰り返しのどこかで感染 [19] の発生が疑われる場合、何度目に実行された処理を調査すべきかを作業者が効率的に判断出来る事が望ましい。

こうした判断に必要な手掛かりの効率的な取得を支援する事を目的として、我々は**繰り返し要約** *(Iteration Summary)* と呼ばれる新しい機能を提案する。

A Feature to summarize Iterated Steps for Efficiently Debugging Complex Loop Executions using An Omniscient Debugger.

[This is an unrefereed paper. Copyrights belong to the Authors.]

Izuru Kume, 奈良先端科学技術大学院大学情報科学研究科, Graduate School of Information Science, NAIST.

Naoya Nitta, 甲南大学大学院自然科学研究科, Graduate School of Natural Science, Konan University.

Masahide Nakamura, 神戸大学大学院工学研究科, Graduate School of Engineering, Kobe University.

Etsuya Shibayama, 東京大学情報基盤センター, Information Technology Center, The University of Tokyo.

繰り返し要約は繰り返される処理の各回の実行内容を、作業者が指定する形で抽象化する機能である。抽象化された実行は局所変数の値によって状態 (mode) 分割され、それぞれの状態で式 (Expression) がどのような順序で評価されている (或いはされていない) のかを明らかになる。作業者は有用な要約結果を得るためにデバッガに適切な局所変数と式を指定する事が求められる。

全知デバッガの機能として繰り返し要約を実現するためには、任意の実行時点に於ける局所変数の値をトレースから特定出来る必要がある。また任意の式に対して、その評価の時機と結果 (値) もトレースから特定出来る事が求められる。後者の実現にはバイトコード命令とソースコード中の式を自動で対応付ける技術が必要となる。

我々は過去に開発したトレース処理技術 [9] と、バイトコード命令の位置付け技術 [8] [10] を用いてこれらの実装上の問題を解決する。

本節の最後に本論文の残りの部分の構成を述べる。第 2 節で繰り返し命令の実行をデバッグする場合の問題を説明する。第 3 節で解決手法として、調査すべき繰り返し処理の決定を支援する**繰り返し要約**とその用法、及び実装を説明する。第 4 節で繰り返し要約を活用してデバッグを実施した作業事例を紹介する。第 5 節で関連研究を、第 6 節で将来課題をそれぞれ述べ、第 7 節で結論を述べる。

2 問題提起

本論文では、文字列中の各文字や配列・リストの要素、テキスト文書の各行のように、一般に複数の要素の集合に対してその一部或いは全てを逐次的 (並列処理無しで) 参照、中間処理の結果を統合して最終的な処理結果を出力するような Java プログラムをデバッグの対象とする。さらに、こうした処理が繰り返し命令文によって実装されていると想定する。本節では繰り返し命令文によってこうした処理を実装する事例として、我々の過去の論文 [10] でも紹介したソースコード例を図 1 に示す。

このプログラムは二つのテキスト文書と、diff 命令によって作成されたそれらの差分テキストを入力とし

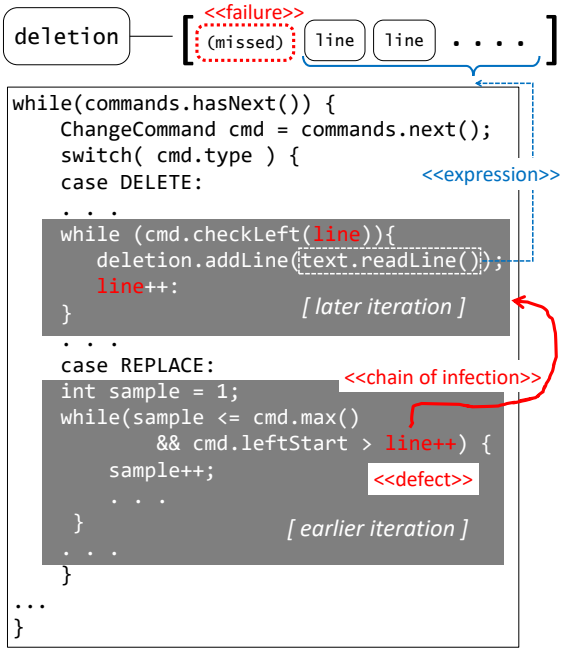

図 1　複雑な繰り返し構造

て、二つのテキスト文書の差分を表現するオブジェクトのリストを出力として生成するものである。コード中の変数 `line` は差分テキストの行番号を指している。この事例ではある回数番目に実行された処理の中で変数 `line` の値が誤って設定され、それが引き金となって感染の連鎖 ([19]) が発生し、最終的には有るべきオブジェクトが生成されないという形で実行が失敗するに至っている。(図 1)

感染の起点 (不具合部分の実行 [19]) は処理が多数回繰り返されるうちの何番目かの実行に潜んでいる。デバッグの最初の作業は、この感染の起点と感染の連鎖の仕組みを特定する**診断** ([19]) である。これによってコード中の不具合箇所が特定出来る。

診断は通常、プログラムの障害が発生した (オブジェクトがリストに含まれていない事が判明した) 時点から過去の実行を遡る形で開始され、実行時点間を主に過去に辿りながら調査が進められる。全知デバッガは過去の実行時点を参照するためにプログラムを再実行する必要が無いため、既存のデバッガに比べて効率的な作業を可能とする。

しかしながらこの事例に示されるような、繰り返し

命令文の実行の本質的な複雑性に対処すると場合には、全知デバッガの実行時点の移動を支援する機能のみでは、不十分である。本論文では全知デバッガでも対処が困難な複雑性が生じる三つの要因を取り上げる。一つ目は処理が繰り返される回数であり、二つ目は各処理そのものの複雑性である。三つめは各処理間の依存関係である。

本事例では diff 命令が作成した差分テキストを一行ずつ読みながら処理が実行される。毎回の繰り返しと読み込まれる行は必ずしも一対一対応していないが、繰り返しの回数は差分テキストの行数に基本的に依存している。同じ種類の誤りを再現する小さな入力事例を生成できない場合には処理が多数回繰り返される実行を調べなければならない。

本事例では繰り返し命令文の毎回の処理に `switch` 命令文と、さらにその `case` 節の中に繰り返し命令文が含まれている。こうした二重、三重の制御構造の入れ子の存在は、作業者がプログラム実行を段階的に追跡する時間と労力を増大させる要因である。

毎回実行される繰り返し処理では複数の変数が互いに依存する形で更新参照されている。例えば変数 `sample` の値は `line` の値に依存する形で計算される。依存関係は個別の処理に留まる場合と比べて、処理の複数の実行にまたがる場合により把握が困難になる。

ソフトウェア開発の実務家の教え [1] に従えば、複雑性への対処の鍵はプログラム実行を追跡する以前の、プログラムの理解と調査範囲の限定に有る。本研究が対象とする繰り返しの命令文実行の複雑性に対しては、各繰り返し処理の実行の内容を少ない労力で事前に把握し、何度目の繰り返しの内容を実行を詳細に追跡 (遡上) すべきか判断する事が効率化に繋がると考えられる。各回の処理の実行の概要を作業者に提示し、作業者の判断を支援する機能が必要とされている。

3 提案手法

我々が提案する**繰り返し要約**は、繰り返し命令文によって実行される毎回の処理それぞれに対して、その実行内容をデバッグ作業者が関心を持つ局所変数や式の値によって抽象化する機能である。この抽象化によってデバッグ作業者はそれぞれの処理の実行中の異常の有無を効率的に調査する事が可能となる。異常有りと判断した回の処理の実行を、その前後の回のそれを合わせる事によって、第 2 節の最後の述べた調査範囲の限定が可能となる。

3.1 手法適用の前提

必要な情報を抽象化によって取得するためには、作業自身が適切な変数や式 (Expression) を全知デバッガに指定する必要がある。指定を行うためには、指定対象となる変数や式が繰り返し命令文の実行に果たす役割に関して作業者が仮説を有している事が求められる。適切な指定によって有用な情報が得るためには作業者が正しい仮説を有している (つまり作業者が役割を正しく理解している) 事が望ましい。

まず大前提として作業者は診断の対象となる繰り返し命令文が、第 2 節の冒頭で述べたような、入力された集合内の個別の要素の参照、中間的な処理とその統合が実施しており、そういう性質の実行が実施されている事を作業者が理解している必要が有る。言い換えれば診断対象が本研究が対象とするような性質を有するものであり、その事を作業者自身が理解している事が必要である。

我々が対象とする繰り返し命令文は要素の参照と共に処理が進む、言い換えれば要素の参照によって駆動されるような性質を有している。よって参照される要素の単位でコードの実行の状態 (モード) を導入出来ると考えられる。そうしたコードでは参照される個別の要素そのものを参照する変数や、或いは要素を指す索引を参照する変数が存在していると考えられる [†1]。第 2 節の事例では変数 `line` がこの役割を果たす。こうした性質の変数を本論文では**原始状態変数** *(primary mode variables)* と呼ぶ。

繰り返し命令文の中では原始状態変数が表現する状態よりも抽象度が高い状態が定義されているかもしれない。例えば第 2 節の事例では、一般に複数の行が同じ差分オブジェクトに属しているため、その差分オブジェクトに属する行に対応する `line` のそれぞれ

†1 そういう変数が存在しないようなコードは提案手法の適用範囲外となる。

の値は同一の状態を表していると考えられる。このような粒度の細かい状態を統合した状態が存在し、その状態を表現する変数が存在する場合、そのような変数を**発展状態変数** *(advanced mode variables)* と呼ぶ。

作業者はコード中に含まれる局所変数の中から原始状態変数を仮説として特定していなければならない。また原始状態変数に依存する発展状態変数に関しても仮説として特定出来る事が望ましい。診断作業の進展と共にそれまで気が付かなかった発展状態変数の存在を認識するかもしれない。以降では原始状態変数と発展状態変数を合わせて**状態変数**と呼ぶ事にする。

ソースコード中に記述される式 (Expression) の中には、状態変数の値と関連付けてその評価や実行の正誤を判定出来るものが有る。例えば図 1 の上部で読み込まれている文字列の値 (`text.readLine()`) の値は状態変数が示す状態と整合性が無いかもしれない。或いは現在の状態が続いている間、`text.readLine()` によって読み込まれるべき文字列が読み込まれていないかもしれない。

このような不整合の発生は作業者が詳細な実行の追跡をどこから開始すべきかを決定する上で重要な手掛かりとなる。よって作業者は可能な限り多くの式について、状態変数と関連付け出来る事が望ましい。このような作業者によって状態変数と関連付けられた式を**整合性検証式** *(consistency checking expression)* と呼ぶ事にする。

状態変数の値と整合性検証式の評価に関する正誤は、双方の組合せに関して診断者の仮説上不整合が生じている事以上の事は示してない事に注意すべきである。誤っているのは状態変数の値かもしれないし、整合性検証式の方かもしれない。双方が誤っているかもしれない。もちろん、作業者の仮説が誤っているのかもしれない。その場合は仮説の形成から遣り直す必要が有るが、こうした手戻りを必要としない、プログラムを正しく理解している作業者は “正しい” 不整合を特定して作業の効率化を実現出来る機会をより多く得られるものと期待出来る。

3.2 繰り返し要約 (Iteration Summary)

本論文が提案する**繰り返し要約** *(Iteration Summary)* は指定された状態変数と式 (必ずしも状態検証式とは限らない) に対して、繰り返し命令文によって実行される各回の繰り返し処理の内容を以下の様式で抽象化するものである:

- 処理全体を同じ状態が続く区間毎に分割
- 各区間内に於ける式の評価の有無とその値 (もし値を持てば) の提示

こうした抽象化の目的は情報を取捨選択する事によって「どの状態の時にどの式がどの値に評価されたか (或いはされなかったのか」を明確化する事にある。指定された式が状態検証式である場合には抽象化によって不整合の有無が明らかになる。状態変数に関する処理が正しい事が確定している場合、ある式が状態検証式であるかどうか判定するために抽象化を試すために利用する事も出来る。

繰り返し要約を利用する作業者は全知デバッガに入力として以下の三つを与える:

- 抽象化の対象となる処理を実行する繰り返し命令文
- ソースコード中の状態変数の宣言
- ソースコード中の任意の式

これらを入力された繰り返し要約は以下の事例が示す様式の出力を生成する:

```
-----------------------------------

(4 回目)

str1.charAt(i)  -->  'd'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  'd'
(Line:112, Column:52 -- Column:65)

i++  -->  3
(Line:137, Column:49 -- Column:51)

[ i : 4, j : 3]

j++  --> 3
(Line:137, Column:53 -- Column:55)

[ i : 4, j : 4]

-----------------------------------
```

```
(5 回目)

str1.charAt(i)  -->  '\"'
(Line:111, Column:37 -- Column:50)

num = 1  -->  1
(Line:114, Column:49 -- Column:55)

i++  -->  4
(Line:121, Column:49 -- Column:51)

[ i : 5, j : 4]

j++  --> 4
(Line:121, Column:54 -- Column:56)

[ i : 5, j : 5]
```

事例 1　繰り返し処理の抽象化

事例 1 ではあるコードの繰り返し命令による四回目と五回目の処理の実行が抽象化された結果が印字されている。この例では状態変数として i と j が指定されている。これらの変数の値の更新によって以下に示す順に状態が変化している。

- [i : 4, j : 3]
- [i : 4, j : 4]
- [i : 5, j : 4]
- [i : 5, j : 5]

以下の式は状態判定式として指定されたものである:

- str1.charAt(i)
- str2.charAt(j)
- num = 1

最初の式が四回目の繰り返しで評価された時の値は 'd' であり、五回目の値がダブルクォート文字である事が示されている。また式と値の対応の下の括弧内の内容は、この式のソースコード中の位置を表している。二つ目の三つめの式も同様に評価値とソースコード中の位置が示されている。二つ目の式は四回目は評価されているが、五回目は評価されていない。三つめの式は逆に四回目は評価されていないが、五回目で評価されている。

以下に示す変数 i と j の増加演算子は作業者によって指定されたものではなく、状態が変更する契機となった式の評価結果を示すために示されたものである:

- i++
- j++

既存の Java デバッガの条件付きブレイクポイントを利用してプログラムの実行時の式の値の列を標準出力やファイルに出力する技法 [4] が知られている。こうした既存のデバッグ手法による出力と繰り返し要約によって生成された出力は似て非なるものである事に注意すべきである。

既存手法が示すのはブレイクポイントで指定された実行時点で式を評価した結果である、この評価は既存の Java デバッガが実行時に挿入するものである。評価される式がプログラムのソースコード中に含まれていたとしても、一般にその同じ式がこの時機に (則ちデバッガがこの評価自体を挿入した時機に) 実行されるわけではない。

一方で、我々の繰り返し要約が示すのは、プログラムのトレースの中から、ソースコード中の指定された位置の式の評価を抜き出したものに他成らない。繰り返される処理の中に繰り返し命令が含まれていない場合、処理の各実行毎に式の値は高々一回だけ評価される。こうした評価を行う実行時点を実行の順序に従って並べたものである。状態変数の値と合わせて提示しているものである。

3.3　実装

一般に Java プログラムはコンパイラによってソースコードから生成されたバイトコード列を Java 仮想機械が処理する形で実行される。多くの場合 Java プログラムのトレースは BCEL [3] 等のバイトコード変換技術を利用して、バイトコードを変換してその実行時にトレースを生成するような拡張を加える事によって生成される。こうしたバイトコードの変換は一般に可測化 (instrumentation) と呼ばれる。我々が過去に開発したトレース生成技術 [9] も BCEL を用いた可測化によって実現されている。

第 3.2 節で示したように、繰り返し要約に対する入力としてソースコード中の構文要素 (繰り返し命令、局所変数宣言と式) が与えられる。繰り返し要約は出力すべき情報をトレースから取得するために、与えられた構文要素に対応する実行時点を選択、処理出

来なければならない。特に与えられた式に対してそれを評価するために生成されたバイトコード列を特定する必要が有る。

構文要素を選択するという事は、則ちトレースに記録されているバイトコード命令の実行をソースコード中の構文要素と対応付ける事に他ならない。バイトコードには構文要素と対応する手掛かりとして、局所変数表 (LovcalVariableTable) と、バイトコード命令とソースコードの行番号を対応付ける行番号表 (LineNumberTable) が標準的な Java コンパイラによって与えられる。特に何か特別なコンパイル技術を用いるのでなければこの二種類の情報のみを用いて対応付けを実現しなければならない。

Java 言語の構文規則では同じ行に複数の式が含まれる事が許されており、バイトコード命令の行番号から対応する式を特定する事は容易では無い。我々はコンパイラが本来対応付けられるべき式を指す適切な行番号をバイトコードに与えるように誘導するソースコード変換手法を開発してきた [8] [10]。この変換手法を応用する事によって Eclipse 上の Java6 環境でバイトコード命令に対する適切な式の対応付けを実現する事に成功した。

繰り返し要約の入力はソースコード中の構文要素である。円滑な入力のためにはソースコードを表示しながら入力する構文要素を選択出来るユーザインタフェイスを開発すべきである。また抽象化もテキスト形式でなく、図式や図形を用いて可視化される事が望ましい。対応付けの核心部分は実装されているが、ユーザインタフェイスは可視化の処理は本原稿を投稿した時点では未だ完了していない。

4 デバッグ事例

本節では二つの文字列を比較する Java プログラムの不具合箇所を実際に特定した事例を用いて繰り返し要約の効果を示す。このプログラムは 2017 年度に実施されたプログラミングコンテストで出題された問題 [2] の答案として作成されたものである。答案そのものは C 言語で実装されているが、Java 言語で書き直したものを作業で利用した。

このプログラムは入力された二つの文字列が、(1) 一致しているか、(2)“殆んど一致” しているか、(3) 一致しないか、を判定する事を目的としている。文字列に零個以上偶数個のダブルクォートが含まれている。ダブルクォートが含まれている場合、奇数番目と偶数番目でくくられている部分が一箇所だけ異なっている場合のみ “殆んど同じ” と判定される。それ以外は (一致していなければ一致していないものと判定される。

例えば以下の二つの文字列はダブルクォートで括られた箇所が二つ含まれており、どちらも異なっているので「一致しない」と判定されるべきである。しかしながら我々の例題プログラムは「“殆んど一致”」していると判断してしまう。

```
read\"C1\";solve;output\"C1ans\";
read\"C2\";solve;output\"C2ans\";
```

事例 2　一致していない入力対

例題のソースコードの中から実際に比較を行っているメソッドのコードのみを抜き出した物を以下に示す。左端に元のソースコードの行番号が与えられている。原稿の紙面に合わせるため、各行の字下げは本来よりも狭くなっている。

```
public static void compare(String str1
    , String str2) {


97:  if (str1!=null && str2 !=null &&
    str1.equals(str2)) {
98:
99:      System.out.println("IDENTICAL
    ");
100:  } else {
101:
102:    int i = 0;
103:    int j = 0;
104:    int num = 0;
105:    int close = 0;
106:
107:    while (true) {
109:      if (i == str1.length() || j
    == str2.length())
109:          break;
110:
111:      if (str1.charAt(i) == '"'
112:              || str2.charAt(j) ==
     '"') {
113:      if (num == 0) {
114:              num = 1;
```

```
115:            }
116:            else {
117:                num = 0;
118:            }
119:            if (str1.charAt(i) == '"
    ,
120:                    && str2.charAt(j
    ) == '"') {
121:                i++; j++;
122:            } else if (str1.charAt(i
    ) == '"') {
123:                while (true) {
124:                    j++;
125:                    if (str2.charAt(
    j) == '"') {
126:                        break;
127:                    }
128:                }
129:            } else if (str2.charAt(j
    ) == '"') {
130:                while (true) {
131:                    i++;
132:                    if (str1.charAt(
    i) == '"') break;
133:                }
134:            }
135:        } else {
136:            if (str1.charAt(i) ==
    str2.charAt(j)) {
137:                i++;j++;
138:            } else {
139:                 if (num == 0) {
140:                     close = 0;
    break;
141:                 } else {
142:                     close = 1; i
    ++; j++;
143:                 }
144:            }
145:        }
146:    }
147:    if (close == 1) {
148:        System.out.println("CLOSE"
    );
149:    }
150:    else {
151:        System.out.println("
    DIFFERENT");
        }
      }
    }
```

事例 3　例題のソースコード

まず、原始状態変数を特定する。2つの入力文字列 str1 と str2 に含まれる文字を指している変数 i と j を原始状態変数と判断する。このメソッドの最後、147 行目から 151 目の条件分岐命令文で‘「‘殆んど一致”(CLOSE)」か「一致しない (DIFFERENT)」が出力されているが、これらの出力は変数の close の値で決定されている。よってこの変数の値が実質的な出力と考えられる。

変数 close は処理の冒頭 (105 行目) で初期化され、139 行目から 143 行目の条件分岐命令文で値が代入されている。この時代入される値は変数 num の値によって決定されている。変数 num の役割は現時点では不明であるが、この変数が関係する処理がいずれかの繰り返しの過程で感染のきっかけとなっている可能性が有る。よって変数 num の役割を特定する目的も兼ねて繰り返し要約を利用する事を決断した。

繰り返し要約に与える状態変数として i と j を指定した。与える式には 109 行目の str1.charAt(i) と str2.charAt(j) を加える事にした。これは i と j が参照する文字が何であるのか表示する事によって抽象化の内容を分かり易くするためである。

さらに 114 行目、117 行目、140 行目、142 行目の num と close への代入式を繰り返し要約の入力として追加した。これらの代入が実施されている時機と、その時比較されている文字の対の関係を明らかにしようという意図に基づいている。

以下に繰り返し要約による出力結果を示す。これは第 3.3 節で説明した機能の核心部分を利用して自動生成されたテキストを見易さのために手動で整形したものである。繰り返し処理の実行を分割し、繰り返しの回数を記載し、かつ原稿の紙面に収めるための改行が施されている。

```
-----------------------------------

(4 回目)

str1.charAt(i)  -->  'd'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  'd'
(Line:112, Column:52 -- Column:65)

i++  -->  3
(Line:137, Column:49 -- Column:51)

[ i : 4, j : 3]
```

```
j++  --> 3
(Line:137, Column:53 -- Column:55)

[ i : 4, j : 4]

-----------------------------------

(5 回目)

str1.charAt(i)  -->  '"'
(Line:111, Column:37 -- Column:50)

num = 1  -->  1
(Line:114, Column:49 -- Column:55)

i++  -->  4
(Line:121, Column:49 -- Column:51)

[ i : 5, j : 4]

j++  --> 4
(Line:121, Column:54 -- Column:56)

[ i : 5, j : 5]


-----------------------------------

(6 回目)

str1.charAt(i)  -->  'C'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  'C'
(Line:112, Column:52 -- Column:65)

i++  -->  5
(Line:137, Column:49 -- Column:51)

[ i : 6, j : 5]

j++  --> 5
(Line:137, Column:53 -- Column:55)

[ i : 6, j : 6]

-----------------------------------

(7 回目)


str1.charAt(i)  -->  '1'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  '2'
(Line:112, Column:52 -- Column:65)

close = 1  -->  1
(Line:142, Column:57 -- Column:65)

i++  -->  6
(Line:142, Column:68 -- Column:70)

[ i : 7, j : 6]

j++  --> 6
(Line:142, Column:73 -- Column:75)

[ i : 7, j : 7]

-----------------------------------

(8 回目)

str1.charAt(i)  -->  '"'
(Line:111, Column:37 -- Column:50)

num = 0   -->  0
(Line:117, Column:49 -- Column:55)

i++  -->  7
(Line:121, Column:49 -- Column:51)

[ i : 8, j : 7]

j++  --> 7
(Line:121, Column:54 -- Column:56)

[ i : 8, j : 8]

-----------------------------------

(9 回目)

str1.charAt(i)  -->  ';'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  ';'
(Line:112, Column:52 -- Column:65)

i++  -->  8
(Line:137, Column:49 -- Column:51)

[ i : 9, j : 8]

j++  --> 8
(Line:137, Column:53 -- Column:55)

[ i : 9, j : 9]

-----------------------------------
```

```
/* 回目～回目の実行部分を削除1020 */

-----------------------------------

(21 回目)

str1.charAt(i)  -->  't'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  't'
(Line:112, Column:52 -- Column:65)

i++  -->  20
(Line:137, Column:49 -- Column:51)

[ i : 21, j : 20]

j++  -->  20
(Line:137, Column:53 -- Column:55)

[ i : 21, j : 21]

-----------------------------------

(22 回目)

str1.charAt(i)  -->  '"'
(Line:111, Column:37 -- Column:50)

num = 1  -->  1
(Line:114, Column:49 -- Column:55)

i++  -->  21
(Line:121, Column:49 -- Column:51)

[ i : 22, j : 21]

j++  -->  21
(Line:121, Column:54 -- Column:56)

[ i : 22, j : 22]

-----------------------------------

(23 回目)

str1.charAt(i)  -->  'C'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  'C'
(Line:112, Column:52 -- Column:65)

i++  -->  22
(Line:137, Column:49 -- Column:51)

[ i : 23, j : 22]

j++  -->  22
(Line:137, Column:53 -- Column:55)

[ i : 23, j : 23]

-----------------------------------

(24 回目)

str1.charAt(i)  -->  '1'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  '2'
(Line:112, Column:52 -- Column:65)

close = 1  -->  1
(Line:142, Column:57 -- Column:65)

i++  -->  23
(Line:142, Column:68 -- Column:70)

[ i : 24, j : 23]

j++  -->  23
(Line:142, Column:73 -- Column:75)

[ i : 24, j : 24]

-----------------------------------

(25 回目)

str1.charAt(i)  -->  'a'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  'a'
(Line:112, Column:52 -- Column:65)

i++  -->  24
(Line:137, Column:49 -- Column:51)

[ i : 25, j : 24]

j++  -->  24
(Line:137, Column:53 -- Column:55)

[ i : 25, j : 25]

-----------------------------------

(26 回目)

str1.charAt(i)  -->  'n'
```

```
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  'n'
(Line:112, Column:52 -- Column:65)

i++  -->  25
(Line:137, Column:49 -- Column:51)

[ i : 26, j : 25]

j++  -->  25
(Line:137, Column:53 -- Column:55)

[ i : 26, j : 26]

-----------------------------------

(27 回目)

str1.charAt(i)  -->  's'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  's'
(Line:112, Column:52 -- Column:65)

i++  -->  26
(Line:137, Column:49 -- Column:51)

[ i : 27, j : 26]

j++  -->  26
(Line:137, Column:53 -- Column:55)

[ i : 27, j : 27]

-----------------------------------

(28 回目)

str1.charAt(i)  -->  '"'
(Line:111, Column:37 -- Column:50)

num = 0  -->  0
(Line:117, Column:49 -- Column:55)

i++  -->  27
(Line:121, Column:49 -- Column:51)

[ i : 28, j : 27]

j++  -->  27
(Line:121, Column:54 -- Column:56)

[ i : 28, j : 28]

-----------------------------------

(29 回目)

str1.charAt(i)  -->  ';'
(Line:111, Column:37 -- Column:50)

str2.charAt(j)  -->  ';'
(Line:112, Column:52 -- Column:65)

i++  -->  28
(Line:137, Column:49 -- Column:51)

[ i : 29, j : 28]

j++  -->  28
(Line:137, Column:53 -- Column:55)

[ i : 29, j : 29]

-----------------------------------

(30 回目)

<<exit>>
```

事例 4　出力結果

事例 4 から判明した重要な事項を以下に述べる。

まず、この実行では原始状態変数 i と j の値が共に毎回 1 づつ単調に増加されている事が分かる。つまりどちらの入力文字列 (str1 と str2) も毎回一文字づつ参照されている。これより繰り返し処理の各実行を、参照される文字対によって特徴付けられる事が明らかとなった。

5 回目、8 回目、22 回目、28 回目の実行内容から変数 num を理解するための手掛かりを得る事が出来る。これらの実行では"'が処理されており、num に対して 0 と 1 が交互に代入されている。これより、num は現在参照されている文字が"'の対の内側に位置するのか外側にあるのかを記録しているのではないかと推定出来る。

変数 close に対しては 7 回目と 24 回目で共に値 1 が代入されている事が分かる。どちらの回も i と j が異なる文字を指している。この前後の num の値と合わせて考えると、これらの異なる文字対は、どちらも"'で括られた内側に位置している事が分かる。つまり、これら二回の実行では異なる文字対を処理しているが、どぎらも"'で括られているために close に

1 が代入されているのではないかと推察出来る。

この繰り返し命令の実行の実質的な処理結果は 24 回目の実行で代入された 1 に他ならないが、これは誤った値である。24 回目の実行では”’ で括られているとはいえ、二つ目の異なる文字対が発見されているわけなので、ここでは 0 が代入されるべきである。そういうわけで、”’ の出現の前後の実行を追う事により、上記の仮説の正しさを確認すると共に、て 139 行目から 143 行目にまたがる条件分岐命令を不具合箇所として特定する事に成功した。

本作業事例でデバッグしたプログラムは行数という面ではそれほど規模が大きくはない。しかしながら繰り返しによって生じる実行過程は手作業で逐一実行を追跡する上では相当な複雑性を有している。作業者が状態変数と整合性検証式を適切に選択する事によって、複雑な処理の詳細に踏み入る事無く感染の疑いが強い箇所を効率的に特定する事が可能である事が示された。結果として本手法の有効性が示されたと我々は考えている。

5 関連研究

全知デバッガ [12] [6] [13] [7] [14] [16] [18] はプログラムのトレース (実行履歴) を記録する事により、利用者が現在参照している実行時点よりも以前の状態を遡って調査する、所謂逆回しデバッギン グ (Back-In-Time Debugging) を可能とする。全知デバッガはしばしば依存関係を利用して、作業者が実行時点間を効率的に移動する機能を実装している。そもそも初期の全知デバッガは「現在参照している変数の値を代入した命令を特定出来る」事を利点として主張していたのである [12]。Treffer 等は slicing 技法を利用してより広範な依存関係を調査する機能を実装している [18] [17] 。

実行時点の移動以外に全知デバッガが備えるべき様々な機能が提案されている。Lienhard 等は Object-Flow と呼ばれる動的解析手法を利用する事によって、オブジェクト同士の参照に由来する依存関係の可視化を実現している [13]。その他の可視化手法として、Andrew Ko 等による Whyline が顕著な成功例として知られてる [7]。Whiline はプログラムの実行、特に条件分岐に関するものを可視化する事によって、「なぜこのコードが実行された (或いはされなかったのか)」とデバッグの初心者でも理解出来るようなユーザインタフェイスを提供した。

デバッグに於いて特定の命令文が実行されなかったり或いは特定のメソッドが呼び出されない、所謂 Execution Omission Errors [20] と呼ばれる不具合の問題が知られている。こうした問題に対処する手法として櫻井等による Omission Finder [16] が知られている。

全知デバッガの問題として扱われるトレースの容量の大きさは二つの観点から問題となる。トレースの容量の問題としてしばしば主張されるのが、デバッガの処理能力を超える可能性である。こうした疑問に答えるべく様々な効率化が研究されている (例えば [15] [14])。また我々自身もグラフデータベースを利用する事によって、ギガバイト単位の容量を大きさを有するトレース処理の高速化が可能である事を確認している [11]。

もう一つの観点はデバッガによる処理が可能であっても、作業者による手作業の限界を超える場合である。我々はこちらの問題の方がより本質的であると考える。我々の提案手法もこちらの問題に焦点を当てている。本提案手法では繰り返し命令文の実行に対象を限定し、抽象化によって複雑性の対処を試みている。より広い範囲の実行内容に対処するためには Goldsmith 等が提案するようなトレースに対する検索手法 [5] も検討すべきと考えている。

6 今後の課題

第 4 節で紹介した事例では処理の繰り返しが 30 回と比較的少数であったが、これが 1000 回規模であるような場合には各回の実行を概観第 5 節の最後で述べたように手作業で扱えるトレースの大きさが問題となる状況下では本論文の提案手法を他の手法と組み合わせるような工夫が必要となる。

洗練された可視化手法はこの問題を解決する有力な候補として検討されるべきである。一つ一つの繰り返し処理の内容の類似性に着目して各回の処理を自動的に色分けする事も可能かもしれない。実際、事例 4 の処理結果を眺めてみると、二つの文字列の一

致部分を処理している回では基本的に同じコードが実行されている。その他の解決手法として特定の値やオブジェクトを利用したフィルタリング手法やトレースに処理結果に対する検索手法を検討すべきである。

7 おわりに

本論文では全知デバッガを用いて Java プログラムの繰り返し命令の複雑な実行をデバッグする際の問題を議論し、問題解決の手法として毎回繰り返される処理を抽象化する**繰り返し要約**と呼ばれる新しい機能を提案した。実際に Java プログラムの不具合を特定する事例を通じて提案手法の有効性を示す事に成功した。

参 考 文 献

[1] Agans, D. J.: *Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems*, AMACOM, 2002.

[2] : ICPC Domestic 2017: Almost Identical Programs, http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1617.

[3] : BCEL Project Page, http://jakarta.apache.org/bcel/.

[4] Fester, A.: Dynamic logging in Eclipse during a debug session, https://www.software-architect.net/blog/article/date/2015/10/02/dynamic-logging-in-eclipse-during-a-debug-session.html, Oct 2015.

[5] Goldsmith, S., O'Callahan, R., and Aiken, A.: Relational Queries over Program Traces, *OOPSLA*, ACM, 2005, pp. 385–402.

[6] Hofer, C., Denker, M., and Stéphane Ducasse: Design and Implementation of a Backward-In-Time Debugger, *Proceeding of NODe 2006*, Lecture Notes in Informatics, Vol. P-88, 2006, pp. 17–32.

[7] Ko, A. J. and Myers, B. A.: Extracting and Answering Why and Why Not Questions About Java Program Output, *ACM Trans. Softw. Eng. Methodol.*, Vol. 20, No. 2(2010), pp. 4:1–4:36.

[8] Kume, I., Nakamura, M., and Nitta, N.: Revealing Implicit Correspondence between Bytecode Instructions and Expressions Determined by Java Compilers, *25th Australasian Software Engineering Conference (ASWEC) and Australasian Software Week (ASW)*, IEEE, 2018.

[9] Kume, I., Nakamura, M., Nitta, N., and Shibayama, E.: A Case Study of Dynamic Analysis to Locate Unexpected Side Effects Inside of Frameworks, *International Journal of Software Innovation (IJSI)*, Vol. 3, No. 3(2015), pp. 26–40.

[10] Kume, I., Shibayama, E., Nakamura, M., and Nitta, N.: Cutting Java Expressions into Lines for Detecting Their Evaluation at Runtime, *Proceedings of the 2019 2Nd International Conference on Geoinformatics and Data Analysis*, ICGDA 2019, New York, NY, USA, ACM, 2019, pp. 37–46.

[11] Kusu, K., Kume, I., and Hatano, K.: A Graph Partitioning Approach for Efficient Dependency Analysis using a Graph Database System, *International Journal on Advances in Networks and Services*, Vol. 10, No. 3&4(2017), pp. 82–91.

[12] Lewis, B.: Debugging Backwards in Time, *International Workshop on Automated Debugging (AADEBUG)*, 2003.

[13] Lienhard, A., Fierz, J., and Nierstrasz, O.: Flow-Centric, Back-in-Time Debugging, *TOOLS EUROPE 2009: Objects, Components, Models and Patterns*, Oriol, M. and Meyer, B.(eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2009, pp. 272–288.

[14] Pothier, G. and Tanter, Éric.: Summarized Trace Indexing and Querying for Scalable Back-in-time Debugging, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*, Berlin, Heidelberg, Springer-Verlag, 2011, pp. 558–582.

[15] Pothier, G., Tanter, Éric., and Piquer, J.: Scalable Omniscient Debugging, *OOPSLA*, ACM, 2007, pp. 535–552.

[16] Sakurai, K. and Masuhara, H.: The Omission Finder for Debugging What-should-have-happened Bugs in Object-oriented Programs, *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, New York, NY, USA, ACM, 2015, pp. 1962–1969.

[17] Treffer, A., Perscheid, M., and Uflacker, M.: Bringing back-in-time debugging down to the database, *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Vol. 00(2017), pp. 521–525.

[18] Treffer, A. and Uflacker, M.: The Slice Navigator: Focused Debugging with Interactive Dynamic Slicing, *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Vol. 00(2016), pp. 175–180.

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

[20] Zhang, X., Tallam, S., Gupta, N., and Gupta, R.: Towards Locating Execution Omission Errors, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, ACM, 2007, pp. 415–424.

謝辞 繰り返し要約の着想を得るための議論に参加して下さった株式会社京都ソフトウェアリサーチの田中康之氏に感謝します。本研究のために例題を提供して下さった甲南大学知能情報学部知能情報学科の本郷亜季氏に感謝します。本研究の一部は JSPS 科研費

JP19H01138, JP17H00731 の助成を受けています。