逆回しデバッグ支援の個別化に向けて

久米 出　中村 匡秀　波多野 賢治　柴山 悦哉

現在広く使われているブレークポイント主体のデバッガに代わって、実行履歴 (トレース) を利用した「逆回しデバッガ (Back-in-Time Debugger)」の研究が進められている。逆回しデバッガはデバッグ作業をブレークポイントの制約から解放する反面、膨大な履歴情報から有用な情報を効率的に獲得する手法が必要となる。本研究は個々の作業者に合わせた情報獲得支援の実現を最終的な目標としている。

デバッグに於ける診断では作業者が対象プログラムの理解に基づいて感染の連鎖を特定する。我々はこれまでに感染やプログラム理解の手掛かりを示唆する挙動をトレースから抽出する手法を研究してきた。我々は現在作業の履歴から作業者の知識や理解を推定し、それに合わせて抽出内容を可視化する逆回しデバッガを開発中である。本論文ではデバッガの概要とその有効性を評価する実験計画に関して説明する。

Back-in-Time debuggers which use execution traces are thought to replace breakpoint based debuggers, which are commonly used now. Users of back-in-time debuggers must retrieve useful information effectively in a large size of execution trace, although they are free from the restrictions imposed by breakpoint. The goal of our research is to establish a way to provide a personalized support of information retrieval for each of developers.

For diagnosis in debugging, developers find a chain of infection based on their understanding of the system under debug. We have developed a trace analysis method to extract behaviors that suggest infection or useful hints for program comprehension. We are developing a back-in-time debugger that visualizes the results of our dynamic analysis according to developers' knowledge and understanding which are obtained from their operations history. In this paper, we explain the overview of our debugger and our plan of an experiment to evaluate the usefulness of our debugger.

1 はじめに

プログラム内の**不具合** *(defect, bug)* を含む箇所が実行されると不正な状態、所謂**感染** *(infection)* が発生する。感染が新たな感染を生じさせる**感染の連鎖** *(chain of infection)* が発生し、最終的には**障害** *(failure)* に至る。デバッグはプログラムの実行時に発生した障害からプログラムコード中の不具合を含む箇所を特定・修正する作業である [†1]。本論文ではデバッグ対象を Java プログラムに限定する。

実用的なプログラムの不具合箇所と障害は直結せず感染 (実行時の不正な状態) の連鎖によって結ばれることが珍しくない。このような不具合の解決には、感染の連鎖から障害の発生に至る過程を説明する検証可能な仮説である**診断**が重要な役割を果たす [14]。

診断を下すためには感染が発生している実行時

Toward Personalized Support for Back-in-Time Debugging.

Concurrent Operations on Splay Trees.

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, Nara Institute of Science and Technology.

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

Kenji Hatano, 同志社大学文化情報学部文化情報学科, Faculty of Culture and Information Science, Department of Culture and Information Science, Doshisha University.

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

†1 本論文中で用いられる不具合、障害、感染、感染の連鎖という用語は文献 [14] の巻末の定義に従っている。

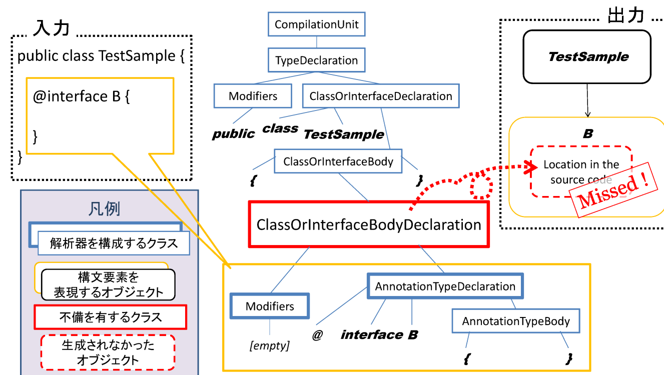

図 1　構文解析器の不具合

点を特定し、それらが依存関係によって連鎖形成している事を確認する必要がある。ブレークポイントを用いる従来のデバッガは依存関係の解析に大きな制約を有している。この制約を解消する有力な手段としてプログラムのトレースを利用した**全知デバッガ** *(Omniscient Debugger)*、或いは**逆回しデバッガ** *(Back-in-Time Debugger)* と呼ばれる †2 新しい原理のデバッガが研究されている [10] [11] [3]

逆回しデバッガによって従来のデバッガによる制約は解消されたが、これが直ちに診断作業の効率化を実現するわけではない。感染が疑われる箇所の特定と感染の有無の判定は個別の作業者のプログラム理解に依存する、属人性の強い作業である。またオブジェクト指向プログラム理解には**設計の非局在性問題** [9] と呼ばれる古くて新しい問題の解決が必要となる。

我々は過去にソフトウェアシステムのアーキテクチャを利用して感染を示唆する挙動をトレースから抽出する動的解析手法の研究を進めてきた [7] [8]。属人性を有さないトレース解析結果を個々の作業者に理解し易い形で表現する**動的解析の個別化**によって診断作業に於ける非局在化の問題解決を図る。我々は現在、動的解析の個別化機能を有する逆回しデバッガを開発中である。本デバッガは作業者が指定したクラスと解析のパターンに基づいて呼び出し構造を簡素化し、かつソースコードの表示に動的解析の結果を反映させる。

本論文の残りは以下のように構成される。第 2 節で研究の動機を具体的な事例を用いて説明し、第 3 節で動的解析の個別化の概要を説明する。第 4 節で実験の計画を、第 5 節で関連研究を、そして第 6 節でまとめを述べる。

2　研究の動機

2.1　不具合事例

一般に実用的なオブジェクト指向プログラムの機能は複雑な階層構造を形成するクラス群が実装する、粒度が細かい多数のメソッド同士の呼び出しによって実装される。こうしたオブジェクト指向プログラムに固有な設計の性質を**設計の非局在性** [9] と呼ぶ。細分化されたメソッド相互の関連はしばしば文書化されない。文書化されていないメソッ相互の関連をボトムアップに復元する作業は一般に多くの人手を時間を必要とする。

設計の非局在化が診断の障害となったプログラムの不具合例 †3 を紹介する。問題となったプログラムは Java のソースコードを入力とする構文解析器であり、クラスやインタフェイス †4 宣言を表現するオブジェクトを出力する。宣言同士の包含関係はオブジェクト間の参照関係として表現され、宣言がソースコー

†2　本論文ではこれらのデバッガの総称を逆回しデバッガとして統一する。

†3　この構文解析器は本論文の著者の一人が開発したものであるが、問題が発覚したのは開発の数ヶ月後である事から設計の非局在化の問題が避けられなかった。

†4　Java の注釈 (annotation) はインタフェイスの一種である。

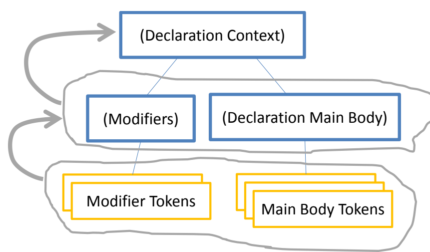

図 2　正例を通じた理解

ド上に占める範囲情報はオブジェクトのフィールド値として参照される。

図 1 に問題となった入力部分とその処理を行った構文解析器のメソッド呼び出し、そして誤った出力を示す。入力には宣言 `TestSample` とそれに包含される宣言 `B` が含まれている。図 1 にはメソッド名の代わりにメソッドを実装するクラスが記載されている。これらのクラスは Java の文法が定める構文要素に一対一に対応しており、その数は三桁にのぼる。

図 1 の実行では構文解析が出力 (生成) した宣言 `B` を表現するオブジェクト (以降では単に「オブジェクト `B` と呼ぶ) の宣言範囲が不正な `null` 値となっていた。本来この値を設定すべきクラス (図 1 で赤い太枠で表現されている) の実装から設定処理を呼び出すコードが抜け落ちていたのである。

我々は初めにオブジェクト `B` の生成から宣言範囲の値の不正が明らかになるまでの過程を調査した。ここで範囲設定処理が抜け落ちている可能性に思い当たり、正しく範囲設定が成されている実行例を調査した。問題となった実行を正しい実行例と比較する事によって処理が抜けていたクラスの特定に成功した。

正しい実行例の理解の過程を図 2 に示す。この調査では構文解析の過程で生成される、字句を表現する `Token` オブジェクトを利用した。それぞれの字句が構文解析器のどのクラスインスタンスに処理されるのかを調査した。その結果、図 2 に示す三つのクラスインスタンス集団同士の関係が明らかになった。図 2 の矢印はそれぞれのインスタンスの状態や挙動から別のインスタンスの正しい状態や挙動が決定された、正誤判定に関する依存関係を示している。

宣言を構成する字句列はそれぞれ修飾子列と宣言本体に分けられて二つのクラスインスタンス ([Modifiers] と [Declaration Main Body]) によって処理される。これらの処理結果は宣言が置かれた場所を表現するクラスインスタンス ([Declaration Context]) によって統合される。宣言の範囲はこのインスタンスが設定する。不具合事例でこの三つのインスタンスに対応するインスタンス (図 1 の太枠で表示されたもの) を特定し、そのコードを調査する事によって診断が完了した。

2.2　逆回しデバッガでも解決が困難な問題

上記の作業ではブレークポイントを用いて標準的な機能を有するデバッガを利用した。しばしばブレークポイントで実行が停止される以前の内容の調査が必要となった。そのためブレークポイントの設定とプログラムの再実行が何度も繰り返された。こうした問題はプログラムの実行過程をトレースに記録する逆回しデバッガ (Back-In-Time) デバッガ [10] [11] [3] の実用化と普及によって解決される事が期待出来る。

上記作業では結果として必要でなかった実行やコードの調査も避けられなかった。字句オブジェクトと構文解析器のクラスインスタンスを関連付ける着想を得るまで少なからぬ作業時間が消費された。またこの着想を得た後も問題解決に直接寄与しない実行過程の調査が避けられなかった。

図 1 、 2 には示されていないのだが、構文解析器のクラスインスタンス同士は構文規則に従って処理の流れを制御する、フレームワーク的な役割を果たすオブジェクトによって介在されている。これらのコードは本構文解析器の開発で利用したコンパイラコンパイラによって自動生成されているため、解読が極めて困難である。診断作業の効率化のためには字句オブジェクトのようなプログラム理解の手掛かりの取得と、問題解決に必要な箇所の特定を支援する仕組みが不可欠である。

3　動的解析の個別化

以下の議論で我々は診断作業者が逆回しデバッガを用いて診断作業を実施するものと仮定する。作業者は障害が発生した過程を記録した実行トレースと実行

の正例を含むトレースを作業に利用出来るものと仮定する。

3.1　診断作業モデル

診断作業では作業者による実行時点の選択とその状態の正不正の判定が不可欠である。状態はクラス変数やそのインスタンス変数の値、呼び出されたメソッド内部の挙動とその効果から構成される。本論文ではこのような正不正判定の対象となる状態を**判定対象**と呼ぶ。状態を構成するクラス変数やインスタンス変数を宣言する、或いはメソッドの実装を与えるクラスを**判定対象クラス**と呼ぶ事にする。

一般に作業者は判定対象単独で、或いは他の実行時の状態を根拠として判定を下す。こうした根拠は一般に複数のクラスやそれらのインスタンスの状態、或いは複数のメソッド呼び出しから構成される。作業者が判定を下す際にその根拠としたこれら実行時の事象を**判定の根拠**と呼ぶ。判定の根拠を構成するクラス変数やインスタンス変数、メソッドに対してそれを宣言する、或いはその実装を与えるクラスを**根拠クラス**と呼ぶ事にする。

第 2.1 節で紹介した正しい実行例の調査では、修飾子列と定義の本体を表現するクラスが判定対象クラスであり、字句を表現するクラスがそれらの根拠クラスとして選択された。一方で修飾子列と定義の本体を表現するこれらのクラスは、宣言が置かれた場所を表現するクラスの根拠クラスとしての役割も果たしている。これらのクラス間の関係は図 2 に示された正誤判定に関する依存関係に他ならない。

図 2 ではある判定対象に対してその根拠を直接的に求めるのではなく、中間的な判定対象を設定してその判定の根拠を求める作業を繰り返して最終的な問題解決に至っている事が示されている。診断作業ではこうした段階的な理解がしばしば必要と我々は考えている。デバッグ作業に於ける問題解決がしばしば入れ子構造を形成している事実[2] は我々の主張を支える傍証である。

判定対象からその根拠を求めるためには多数のソースコードファイルの表示を切り替えながらプログラムの実行を追う作業がしばしば必要となる。図 1 に示

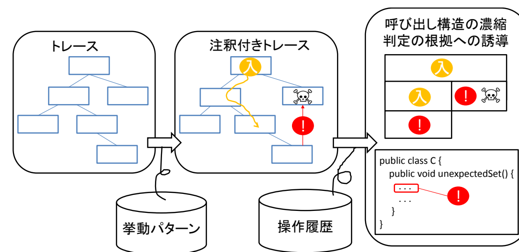

図 3　動的解析の個別化処理の概要

される構文解析器内部の関係の特定に於いてもコンパイラコンパイラによって自動生成された難解なコードも交えた手間暇を費す調査が時には必要であった。こうした判定対象クラスから根拠クラスを特定する作業を支援する事によって、診断作業全体の大幅な効率化が期待出来る。

3.2　支援の概要

作業者が自身が判定対象とその根拠クラスの候補を指定し、我々が開発するデバッガは判定対象からその判定の根拠を求める作業の支援を実現する。本デバッガはプログラムのアーキテクチャ情報を利用した動的解析機能を有している。作業者はこのデバッガに対して判定対象とその判定の根拠の候補を指定し、動的解析の結果を利用して両者の関係を表示させる事が出来る。作業者が指定する候補はその理解に依存しているため、このような形で**動的解析の個別化**が実現される。

3.2.1　パターンを利用した動的解析

本デバッガが扱うトレースにはプログラムの実行から障害の発生に至るまでのメソッドの呼び出し構造と、各メソッド呼び出しで実行された命令列が記録されている。本トレースは文献[13] に述べられた動的スライスと同程度の依存関係を含む細粒度なデータ構造を有している。本トレースの命令には変数代入、条件分岐のようなソースコード上で表現される命令に加えて、フィールド値の参照、メソッド呼び出し、オブジェクトの生成、値の演算も含まれる。

フレームワークとそのアプリケーション固有部分の区別のようなアーキテクチャに関する情報をトレースの属性として付与出来る。この属性を利用して不正な

状態を示唆する挙動をトレースから抽出する動的解析手法を我々は開発して来た[6][8]。これと並行して現在我々は、字句クラスのようなプログラムの入出力と直接的な関わりを有するクラスを特定し、そのインスタンスやメソッド呼び出しが実行過程に与えた影響を抽出する解析手法を開発中である。本論文ではこうした解析される挙動のパターンがデバッガに事前に登録され、解析の意味するところが利用者に周知されている状況を想定している。

動的解析の個別化を実現する処理の概要を図 3 に示す。まず属性が付与されたトレースに対して予め登録された挙動のパターンを用いた解析が行われる。各パターンに対してパターンが表現する挙動を構成する命令集合が複数抽出される。各命令集合には命令同士の依存関係が含まれている。これら抽出された集合の要素に対して、パターンを特定する注釈情報が付与された結果**注釈付きトレース**が生成される。

3.2.2 動的解析の個別化

トレースに付与された注釈を現在観察している判定対象と関連付ける事によって、例えばある分岐の選択がフレーンムワークの誤用に依存している事に早期に気が付いたり、或いは構文要素と対応するソースコード中の字句の特定が容易になる事が期待出来る。こうした形でその正不正の判定や判定対象の特定を効率化出来ると我々は期待している。

この関連付けを可視化するために我々は二種類の表示機能を実装中である。一つ目はメソッドの呼び出し構造を濃縮した表示である。この表示の目的は判定対象とその判定の根拠の関係を構造が簡素化されたメソッド呼び出しの視点から関連付ける事にある。このメソッドの呼び出し構造の中で実行された命令はそれらの効果によって抽象化される。抽象化された効果は濃縮によって隠された詳細を辿る足掛かりとしての役割も果たす。

二つ目の表示機能は表示されたソースコードに動的解析の結果を付与するものである。ソースコードに記述された命令に対して上記の効果が対応付けられる。副作用のようにその効果が複数のメソッドに跨がる場合には、その範囲が上記の濃縮された呼び出し構造状表示される。

作業者はデバッガに対して判定対象を指定し、デバッガが提示するクラスの中から自らの主観に基づいて判定の根拠クラスを選択する。この時デバッガは作業者の操作履歴に基づいて提示されるクラスを決定する。作業者はさらにデバッガに観察したい挙動パターンを指定する。

デバッガはこれらの指定に応じて判定対象が実行されるメソッドの呼び出し構造を濃縮する。濃縮は反対対象が直接実行されているメソッド、挙動パターンが実装するメソッド、そしてパターンが個別の表示を要求するメソッド以外の全てのメソッド呼び出しをその呼び出し元に再帰的に吸収させる。判定の根拠クラスのメソッドでも同じ受け手に対して呼び出されているメソッドの連鎖も呼び出し元に吸収させる。

以上に説明したように作業者は自身の理解に基づいた指定を行い、デバッガはその指定に基づいて表示の内容を決定する。このような形で動的解析の個別化が実現される。

4 評価実験計画

デバッガの開発と同時に、我々はその評価実験の設計も進めている。本デバッガは判定対象判定に対して判定の根拠を特定する作業を効率化するための機能が実装される。被験者を用いた診断作業を実施することによってその機能の有効性を検証する。検証の方式としては被験者を二つの集団に分けて一方では本デバッガを、もう一方では動的解析の個別化機能を削除した版用いて双方の成績を比較する形が考えられる。

それ以外に本デバッガを用いた被験者に対してインタビューを実施してその有効性を検証する実験方式が考えられる。この方式の実験では第 3.2.2 節で説明したユーザインタフェイスに対する作業者の操作履歴を記録する。同時に作業者の内面の思考内容を発話させる所謂 “think aloud” と呼ばれる方式のデータ採取が必要となる。

被験者によるデバッグ作業が完了した直後に操作履歴と発話を再生、操作と内面の思考を関連付けるためのインタビューを実施する。これらのデータから動的解析に個別化が問題解決にどのように貢献したのか、或いはしなかったのかが評価される。

5 関連研究

逆回しデバッガ [3] [10] [11] の歴史は浅く、実用的なプログラムをデバッグする際の実行効率的に関してその実用性を疑問視する意見 [12] も散見される。我々の過去の研究 [6] で得た知見に依れば、通常の PC 上で実用的なプログラムを実行し、文献 [13] で扱われているものとと同等な、極めて詳細な内容を有する実行履歴の処理には数分を要する。我々の現在の実装は実行効率に関して何の工夫もなされておらず、今後の改善も期待出来る。コードの書き換えと実行を頻繁に繰り返さないのであれば実行効率に関しては大きな問題とならないと考えている。

近年では洗練されたユーザインタフェイスを備えた逆回しデバッガが実用化されている [5] [4] [1] 。Whyline [5] [4] は指定した命令文が「何故実行されたのか (或いはされなかったのか)」を作業者が問い合わせるための洗練されたユーザインタフェイスを実装している。プログラムの実行時の依存関係を辿る際にこの機能は大いに効力を発揮している。JIVE はプログラム実行から UML の順序図を作成する、逆工程 (reverse engineering) 機能を有しており、Java のプラグインとして実用化されている。

6 結語

本論文ではデバッグに於ける診断に焦点を当て、その作業の効率化を妨げる要因を考察した。また作業の効率化を目的として現在我々が開発を進めている逆回しデバッガの機能と、その効果を検証するための実証実験の計画を説明した。

謝辞

この研究は栢森情報科学振興財団、人工知能研究振興財団、同志社大学ハリス理化学研究所研究助成金、科研費基盤 (A)24500352、基盤 (B)23300009 及び 26280115、基盤 (C)24500079、挑戦的萌芽 25540150 及び 15K12009 の助成を受けて遂行された。

参 考 文 献

[1] Czyz, J. K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse, *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, New York, NY, USA, ACM, 2007, pp. 31–35.

[2] Halle, J. E., Sharpe, S., and Hale, D. P.: An Evaluation of the Cognitive Processes of Programmers Engaged in Software Debugging, *Journal of Software Maintenance*, Vol. 11, No. 2(1999), pp. 73–91.

[3] 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.

[4] Ko, A. and Myer, B.: Finding Causes of Program Output with the Java Whyline, *SIGCHI: Human Factors in Computing Systems*, ACM, 2009, pp. 1569–1578.

[5] 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.

[6] Kume, I., Nakamura, M., Nitta, N., and Shibayama, E.: Toward a dynamic analysis technique to locate framework misuses that cause unexpected side effects, *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on*, June 2014, pp. 1–6.

[7] 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.

[8] Kume, I., Nakamura, M., and Shibayama, E.: Toward Comprehension of Side Effects in Framework Applications as Feature Interactions, *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, 2012.

[9] Lekovsky, S. and Soloway, E.: Delocalized Plans and Program Comprehension, *IEEE Software*, Vol. 3, No. 3(1986), pp. 44–49.

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

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

[12] Ressia, J., Bergel, A., and Nierstrasz, O.: Object-Centric Debugging, *International Conference on Software Engineering*, IEEE, 2012, pp. 485–495.

[13] Wang, T. and Roychoudhury, A.: Using Compressed Bytecode Traces for Slicing Java Programs, *International Conference on Software Engineering*, IEEE, 2004, pp. 512–521.

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