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

久米 出　中村 匡秀　新田 直也　柴山 悦哉

デバッグはプログラム実行の失敗からその原因となった不具合個所を特定、修正する作業である。デバッグ作業者は不具合個所を特定するために様々な実行時点を観察しながら仮説を立て、それらの受け入れや棄却を繰り返す。我々はこうしたデバッグ作業者による思考の適切性を評価する実験手法の開発を目指している。

思考の適切性を評価するためには、実験に参加する作業者がプログラム実行の失敗の過程をどのように理解しているのかを明らかにした上で客観的な方法でその正誤を判定する必要がある。本論文ではこうした正誤判定の支援を目的として、プログラム実行が失敗する過程で発生する「感染の連鎖 (Chain of Infection)」をデータ化する動的解析手法を提案する。

Debug is a task to find the defect that causes a program failure, and to fix it. In order to find the defect a debugging maintainer repeatedly makes, accepts, and rejects hypothesis based on their investigation of execution points. Our research goal is to establish an experimental method to evaluate the relevance of maintainers' thoughts.

In order to achieve our goal, we need to make it clear how maintainers understand a program failure, and to evaluate the correctness of their understanding in an objective way. In this paper, we propose a dynamic analysis to materialize *the chain of infection* that represents a program failure in order to support the correctness evaluation.

1　はじめに

プログラム実行の**失敗** *(failure)* は利用者に観測される実行の誤りであり、プログラムコードの記述中に含まれる誤り、誤った入力、或いは設定の誤りによって引き起こされる。以下、本稿ではプログラムコードの誤りに起因する実行の失敗のみを取り扱う。失敗を引き起こすコードの誤りを**不具合** *(defect)* と呼ぶ。

デバッグとはプログラム実行の失敗を受けてその原因たる不具合を特定し修正するソフトウェア保守作業に他ならない。

デバッグにはいきあたりばったりに進められるものと科学的な態度に基づいて定型的に遂行されるものに二分される。いきあたりばったりでないデバッグでは誤り個所を特定し、それがプログラムの失敗を引き起こす過程を説明する**診断** *(diagnosis)* と呼ばれる仮説を形成する作業が含まれる [25]。本稿では診断を形成しようとしている作業者を**診断者**と呼ぶ。

診断とは現実の実行が本来有るべき実行から逸脱する過程を説明するものである。よって診断の形成は必然的に診断者による本来の実行と逸脱に関する認識に大きく依存する。[25]。診断者は通常デバッガを用いた多数の実行時点の観察を通じてこうした認識を形成、更新する事が求められる [1]。しかしながら実用的なプログラムの実行が失敗する過程は一般に非常に複雑であり、そのため診断者は誤った手掛かり

Failure Materialization for Cognitive Analysis of Debugging Maintainers

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.

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

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

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

に導かれて多大な労力と時間を浪費してしまう事が珍しく無い[5]。

プログラムの実行が失敗する過程を複雑化させる重要な要因として**不実行の誤り** *(execution omission error)*[26] と**偶然による修正** *(Coincidental Correction)*[24] の二つが知られている。両者は元々デバッグの不具合個所を特定する自動化手法の研究過程で特定された概念である。不実行の誤りはある時機に本来であれば実行されるべき命令文が実行されず、結果として実行時の状態に誤りが発生させるものである。偶然による修正は一旦発生した実行時の誤りが何らかの命令文の実行によって偶々正常なものに修正される事象を指す。

我々はこのような複雑さに直面した診断者が、診断の礎となる適切な認識を獲得するに至った (或いは獲得に失敗した) 内面的な過程を明らかにする手法の開発を目指している。一般に作業者の内面的な分析はソフトウェア工学ツールの設計の論拠を与える上で重要な役割を果たす[22][7]。我々はさらにデバッガの評価実験の妥当性を担保する上でもこうした内面分析が重要であると考えている[11]。

認識の過程を明らかにするためには診断者の内面的な認識を表出させると共に、その認識の適切さを客観的なデータに基づいて判定する手段が必要となる。作業者の内面の表出、分析に関してはデバッグを含むソフトウェア保守や、デバッグとも関連が深いプログラム理解の分野で様々な手法が開発されている[5][4][16][21][20][14][8] 。その一方で診断形成過程に於ける診断者の認識に関しては、我々の知る限りその妥当性の定義すら皆無である。

本稿では診断者の認識の妥当性評価の実現に向けて、プログラムが失敗する過程で本来あるべき実行からの逸脱の発生を具体的なデータとして表現する**失敗の実体化** *(Failure Materialization)* と呼ばれる手法を紹介する。失敗の実体化は診断実験に利用する事を想定している。**感染の連鎖**[25] と呼ばれる考え方に基づいて本来有るべき実行からの逸脱の定義を与える。また不具合を有するプログラムの実行履歴を修正版のそれと比較する事によって本来あるべき実行からの逸脱の発生とそこから派生する効果を具体的な形で明らかにする。こうした逸脱の表現は診断者の認識の妥当性を判定する際の客観的な根拠を与えるものになると我々は期待している。

本稿の以降では以下のように構成される。まず、本研究の背景であるプログラム実行や診断に関する基本的な用語を第 2 節で定義する。次に研究の動機を第 3 節で、失敗の実体化について第 4 節で説明し、その課題に関して第 5 節で議論する。第 6 節で関連研究を、結論を第 7 節で述べる。

2 プログラム実行の失敗その診断

本論文ではプログラム実行の失敗過程を説明するために Zeller [25] による用語を導入する。Zeller [25] に依ればプログラム実行が失敗する過程はコードの誤りである**不具合** *(defect)* の実行によって開始される。不具合の実行によって実行時の状態に誤りが発生する。これを**感染** *(infection)* と呼ぶ。感染は新たな感染を引き起こし、最終的にはプログラム実行の失敗を示す**障害**に至る。

不具合はプログラムの記述の誤りであり、一方感染も障害もプログラム実行の誤りである。障害はプログラムの利用者に実行の失敗を示す何らかの出力を伴うが、感染はプログラム内部の状態によって表現される。感染は作業者がデバッガを用いて該当する実行時点を観察し、それが本来あるべき状態から逸脱している事を**適切に認識**してその存在が初めて特定される。

診断 *(diagnosis)* はプログラムの不具合を特定し、不具合の実行から障害の発生に至る過程の説明を与える仮説である。本論文では診断の形成を試みる作業者を**診断者** *(diagnoser)* と呼ぶ。デバッグの実践者は診断者がプログラムを十分理解した上で、デバッガを用いて感染の存在を特定し、一定の戦略に基づいて感染の連鎖を遡る事を推奨している[1]。よって感染の存在を特定する能力、ひいては有るべき実行からの逸脱の認識の妥当性こそが診断形成の成否や効率性を大きく左右する重要な要因であると考えられる。

3 問題提起

第 2 節で述べたように、診断形成の効率的な遂行は、有るべき実行からの逸脱に関する診断者の認識の

妥当性に大きく左右される。診断者の認識の妥当性は、観察された各実行時点が実際に逸脱しているか否かに基づいて評価されるべきである。よって認識の妥当性評価を実現する上で、指定された実行時点に対する逸脱の有無を判定する仕組みが不可欠である。

こうした仕組みを実現する上で、実用的なプログラムの実行が失敗する過程の複雑さに対する対処が必要となる。実行の失敗を表現する感染の連鎖の詳細は一般に想像される以上に複雑である。

偶然による修正 *(Coincidental Correction)* [24] は感染の連鎖を消滅させる。診断者が感染を特定出来なかった実行時点の直前まで感染が連鎖していたかもしれない。本来あるべき実行を理解していない診断者が**不実行の誤り** *(execution omission error)* [26] を発見する事は極めて困難である事は容易に想像出来る。本節では例題プログラム (プログラムリスト 1) を用いて不実行の誤りと偶然による修正による複雑性の導入を説明する。

```
1
2   public class SomeClass {
3
4     private int _negative;
5
6     public int doIt() {
7
8       int ret = callMe(3);
9       int ret2 = callMe(0);
10      int ret3 = callMe(-11);
11      return (ret+ret2+ret3)*this.
            _negative;
12    }
13
14    public int callMe(int x) {
15
16      if(x > 0) { //Must be (x < 0).
17
18        this._negative+=x;
19      }
20      return x;
21    }
22  }
```

Example Code 1　誤てるプログラム例

この例題プログラムではクラス `SomeClass` が宣言されている。このクラスはメソッド `doIt` と `callMe` を実装している。メソッド `doIt` が呼び出されると、メソッド `callMe` が、引き数 `3`、`0`、`-11` を渡されて三回呼び出される。

メソッド `callMe` は不具合を有している。コード中の 16 行目の条件分岐の条件式は、本来 `x < 0` であるべきところが `x > 0` となっている。メソッド `doIt` からの三度の呼び出しの結果は全てこの誤った条件式に依存しているが、その結果に関する詳細はそれぞれ異なっている。

引き数として `3` を与えた 8 行目の呼び出しでは、本来更新されるはずでなかったインスタンス変数 `_negative` の値が更新されている。この誤った値の変更によって感染が発生している。この感染の発生の直接的な原因である

引き数 `0` を与えた 9 行目の呼び出しでは偶然による修正が働いている。この呼び出しではこのインスタンス変数の値は更新されない。 正しいコードでもインスタンス変数の値は更新されないのでこれは本来あるべき実行である。このようにコードの不具合部分が実行された時の状態 (この場合は `callMe` の引数 `x` の値) によっては不具合箇所が実行されても本来の実行結果と同じ結果が得られる事がある。

引数として`-11` を与えた 10 行目の呼び出しでは不実行の誤りが発生している。この呼び出しでは本来変更されるべきインスタンス変数の値が変更されていない。これは命令文の実行が誤って省かれた事例である。このメソッド終了時にインスタンス変数 `_negative` は本来そうあるべき値が代入されていない形で感染が発生している。

感染の発生は必ずしもその直後に感染の連鎖を生じさせるとは限らない。上記の 8 行目と 10 行目で呼び出された `callMe` の実行によって感染が発生しているが、呼び出しの返り値は不具合部分の影響を受けないために本来有るべき値である。そのため `callMe` の実行を追跡せずに呼び出された側の実行のみを追う限り本来あるべき実行とその内容は全く変わらない。これはこれらのメソッドが呼び出されてその返り値が変数に代入される時点では感染が発生していない事を意味する。

一方、メソッド `doIt` の実行がさらに進み、11 行目で値が計算されて返り値として呼び出し側に渡さ

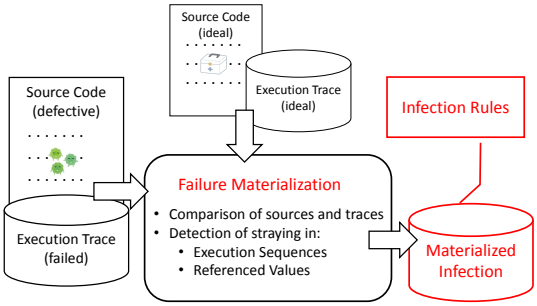

図 1　失敗の実体化：概要

れている。この時インスタンス変数 `_negative` の誤った値が利用されている。結果として不具合による感染は二度の `callMe` 呼び出しで実行された条件分岐命令と、値を返す命令の実行によって断続的に連鎖する事になる。

有るべき実行からの逸脱、すなわち感染に関する診断者の認識は、現実の感染の連鎖と照らし合わせる事によってその妥当性を評価出来る。しかしながら上の例が示すように現実の感染の連鎖はあまりにも複雑であるために、デバッガを用いてステップ実行しながら感染の連鎖を確認するような素朴な手法は非現実的である。妥当性評価の円滑な遂行を支援するための手法が必要とされている。

4　感染の実体化

4.1　手法の概要

現在我々は我々は診断者による感染の認識の正しさの評価する実験手法を開発中である。実験手法には診断者による有るべき実行からの逸脱に関して、その認識の妥当性を評価する方法が含まれる。認識の妥当性を評価するためには実験を通じて診断者による認識の表出を促し、表出された認識に対して客観的なデータに基づいた評価の手続を適用する必要がある。以降では診断を形成するプログラムは Java 言語によって記述されているものと想定する。

第 3 節で述べたようにプログラム実行中に於ける感染の連鎖は非常に複雑であり、実行時点に関する感染の有無を手作業で判断する事は現実的では無い。この問題を解決するために我々は**失敗の実体化** *(Failure Materialization)* と呼ばれる動的解析手法を導入する。この動的解析手法では我々が先に開発したトレース生成ツール [9] を利用している。

図 1 に失敗の実体化の概要を示す。失敗の実体化は誤てるプログラムの実行に対して、そこに含まれる各実行時点で感染が発生しているか否かを特定する事を目的としている。実行全体に渡る特定によって感染の連鎖が形成される筈である。我々はこうして形成された感染の連鎖の健全性を確認するために**感染規則** *(Infection Rule)* を定める。

失敗の実体化では診断形成の実験に用いるための不具合を有するプログラムと、その不具合を修正した版が共に準備されている事を想定する。不具合を有する版を**誤てるプログラム** *(defective program)* と呼ぶ。またそのソースコードとその実行をそれぞれ**誤てるソースコード** *(defective source code)*、**誤てる実行** *(defective execution)* と呼ぶ。また不具合を修正したプログラム、そのソースコード、及び実行をそれぞれ**有るべきプログラム** *(ideal program)*、**有るべきソースコード** *(ideal source code)*、**有るべき実行** *(ideal execution)* と呼ぶ。誤てる実行と有るべき実行には同じ入力が与えられているものとする。

有るべきプログラムは不具合を有しない有るべきソースコードとその有るべき実行結果 (トレース) を提供するものである。失敗の実体化の基本的な着想は、誤てるプログラムのソースコードと実行結果を有るべきプログラムのソースコードと実行結果を比較する事によって不具合と、実行の逸脱を特定しようというものである [10]。特定された不具合コードの実行と感染の連鎖は誤てる実行を表現するトレース内の属性として実体化される。トレース上に表現されたこの本来あるべき実行からの逸脱を我々は**実体化された感染** *(Materialized Infection)* と呼ぶ。

4.2　感染規則

我々は誤てるプログラムの失敗した過程に含まれる全ての命令文の実行を対象として以下の分類を与える:

Defective:　この分類は不具合コードに含まれる命令文の実行を意味する。

Ideal: この分類は有るべき実行と同じ時機に同じ命令文が実行されている事を意味する。プログラムを命令文の制御の流れ図 ([23]) として表現した時に有るべきプログラムと同じ経路で命令文が実行されている事を示す。ここに分類される命令文は不具合個所に含まれない。

Straying: 本分類は不具合個所に含まれない命令が感染によって本来とは異なる時機に実行されている事を意味する。

誤てるプログラムで実行される命令文は不具合コードが実行されるまで全て ideal に分類される。ideal 属する命令文に関して関して以下の規則を満たす事が求められる：

Infection Rule 1 *(Ideal Statement Execution) ideal* に分類された命令文の実行は有るべき実行に含まれる命令文の実行と一対一に対応付けられなければならない。命令の実行 I に対応する本来のプログラムの命令実行を $id(I)$ とすると、I と $id(I)$ はそれぞれの構文木上の一致部分の同じ節点に位置し、構文要素として同じ内容でなければならない。誤てるプログラムと有るべきプログラムをそれぞれ制御の流れ図として表現した時に、I を含む実行経路と $id(I)$ を含む実行経路は一致していなければならない。

誤れるソースコード内の不具合個所は有るべきソースコードと字面上の比較によって特定出来る。よって命令文の実行が Defective か否かはソースコードの字面の解析によって決定できる。これを以下の規則として記述する:

Infection Rule 2 *(Defective Statement Execution)* 不具合部分に含まれる、或いは不具合部分の一部を含む命令の実行は *Defective* に分類される。

straying に分類された命令文の実行は本来有るべき実行からの逸脱、即ち感染を表現している。感染したメソッド呼び出しや条件分岐によって実行される命令文も感染を表現する。即ち straying な命令文に関しては以下に述べる推移律が成立する。

Infection Rule 3 *(Straying Statement Execution)* ある命令文の実行 *(*R とする*)* がメソッド呼び出し或いは条件命令 *(C)* の実行に依存すると仮定する。C が *Defective* 或いは *Straying* に分類される時、R は *Straying* に分類されなければならない。

感染が連鎖するのは感染した呼び出しや条件分岐によって実行される命令文のみである。呼び出しからの復帰や分岐が終了した後に実行される命令文にはこれらの呼び出しや分岐に関する推移律は適用されない。

Java プログラムでは expression の評価や expression statement の実行によって所謂 *Def-Use* [23]、さらにオブジェクト指向言語に固有な aliasing [15] によってデータの参照に関する関係が形成される。clean な命令文に含まれる expression や expression statement に関して感染に関する以下の分類を導入する:

Clean: expression の評価や expression statement が表現する値が感染していない事を示す。

Latent: expression の評価や expression statement が表現する値が感染しているが、たまたま有るべき実行で得られるものと同じである事を示す。

Infectious: expression の評価や expression statement の値が感染しており、有るべき実行とは異なっている事を示す。

有るべき実行と誤れる実行で生成参照される値が「同じか否か」を考える場合、基本値に関しては単純に比較すれば良い。しかしながら両者はそれぞれ別の実行であるため、オブジェクト同士の関係に関しては必ずしも自明ではない。両者の対応関係を明確にすべく、誤てる実行で生成参照されるオブジェクトに関して以下の定義を導入する:

Definition 1 *(*対応オブジェクト*)* プログラム実行で参照されるオブジェクト *obj* に対して、*obj* を生成或いは初めて参照する命令文をその導入と呼ぶ。

obj を誤てるプログラム実行で参照されるオブジェクト、*I* を *obj* の導入、*I* で *obj* を表現する *expression* を *e* とする。

導入 *I* が *ideal* であり、*id*(*I*) 内の *expression* *e* が表現するオブジェクトを *obj*′ とする。*id*(*I*) が *obj*′ の導入である時、*obj*′ は *obj* の対応オブジェクト候補 (a candidate of corresponding object) であると言う。

obj が唯一の対応オブジェクト候補 *obj*′ を有する時、*obj*′ を *obj* の対応オブジェクトと呼び、*corr(obj)* と表記する。

対応オブジェクトを用いる事によって、誤てる実行と有るべき実行の間で以下のように値の対応関係を定義する:

Definition 2 *(*値の対応*)* 誤てる実行中に参照される値を *v*、本来の実行で参照される値を *v*′ と表現する。以下のどちらかの条件を満たす時に *v* と *v*′ は同値 (equivalent) であると定義し、*v* ≡ *v*′ と表記する。

- *v* と *v*′ は基本値であり、*v* = *v*′ が成立する。
- *v* と *v*′ は同じ型のオブジェクトであり、*v*′ = *corr*(*v*) が成立する。

clean な値はそのデータを運んだデータの流れが感染の連鎖に依存せずに形成されている事を示す。この性質は以下の規則によって保証される:

Infection Rule 4 *(*一般的な *clean* 値*)* 命令文中に含まれる *expression (*ただし局所変数、クラス変数、インスタンス変数、配列要素の参照を除く*)* が表現する値は以下に述べる条件が全て満たされる場合にその場合に限り *clean* である。

- その命令文が *ideal* であり、
- *expression* 中に含まれる全ての *expression* が表現する値が *clean* であり
- 本来の値と同値である

Infection Rule 5 *(clean* な副作用*)*

クラス変数、インスタンス変数、配列要素の参照を表現する *expression* は上記の制約を満たし、かつその変数や配列要素への代入命令に関して以下の制約を満たす場合にその場合に限って *clean* となる。

- 代入命令内に含まれる全ての *expression* が表現する値が *clean* である
- 代入命令が *ideal* である。

At last, we introduce a rule to specify latent and infectious values: 最後に latent と infectious に関する制約を述べる:

Infection Rule 6 *(dirty* な値*)* *I* を誤てるプログラムにおける *ideal* な命令文の実行、*v* を *I* に含まれる *expression (e)* が表現する値とする。*id*(*I*) が *e* として値 *v*′ を参照するとする。*v* の *e* での参照が *latent* である場合、*v* ≠ *v*′ でなければならない。*v* の *e* での参照が *infection* である場合、*v* ≡ *v*′ でなければならない。

続けて第 3 節で紹介したプログラムコード (プログラムリスト 1) に関して上記の規則を適用する。以降ではメソッド `doIt` 呼び出しは常に ideal であると想定する。

`if` 文の条件式が誤っているので規則 2 よりこの条件文の実行は全て defective である。これと規則 3 インスタンス変数に対する代入文は全て straying に分類される。一方、`doIt` 内での `callMe` の呼び出しは、`doIt` が ideal である限り規則 1 によって全て ideal となる。

`callMe` が呼び出した `doIt` の返り値は上述の条件分岐に依存していない。よって `doIt` の呼び出しが ideal である限り規則 4 によってこの返り値は clean となる。一方で `callMe` によって代入される `_negative` の値は規則 3、規則 4、規則 6 によって infectious となる。`doIt` の返り値も同様に infectious となる。

プログラム例 2 にクラス `SomeClass` のもう一つのメソッド `doItAlt` が示されている。このメソッド内

での呼び出し `callMe(-1)` は上述の不具合のため、本来であれば変更される筈の `_negative` の値が変更されない。よってこのメソッド呼び出しに続く復帰文 (return statement) で参照されている `_negative` の値は規則 4 と規則 6 によって infectious となる。この事例では省略による副作用による感染が特定されている事に注意して欲しい。

```
1
2   public class SomeClass {
3
4     ...
5
6     public int doItAlt() {
7
8       callMe(-1);
9       return this._negative;
10    }
11  }
```

Example Code 2　不実行の誤り例

4.3　感染モデル

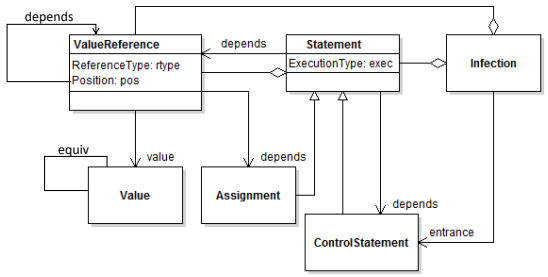

図 2　感染モデル

我々は実体化された感染をクラス図としてモデル化している (図 2)。このモデルを**感染モデル**と呼んでいる。感染モデルは我々が先行研究 [9] で開発したトレースのモデルを拡張する形で定義されている。モデル中のクラス `Statement` は命令文の実行を表現している。分岐命令 (`if` 文と `switch` 文) 及びメソッド呼び出しはこのクラスの下位クラス `ControlStatement` として抽象化される。さらに Java 仮想機械内部での例外処理の割り当てもこの仮想的な命令文の実行として `ControlStatement` によって表現される。`Statement` の属性 `exec` は命令文の感染に関する分類を表現する。

クラス `Assignment` は局所変数、クラス変数、インスタンス変数、そして配列要素の代入を抽象化するものである。クラス `ValueReference` は expression 或いは expression statement として表現される値の生成や参照を表現する。`ValueReference` の属性 `rtype` と `pos` は感染に関する値の分類と、自身を含む命令文に置ける構文上の位置を表す。`ValueReference` から参照されるクラス `Value` は参照される値そのものを表現している。第 4.2 節で定義された値の同値性が関連 equiv として表現されている。

上記のクラス間を結ぶ関連 `depends` は制御の依存関係やデータの依存関係を表現している。これらの依存関係は第 4.2 節で導入した感染規則をトレースの要素に適用する際に参照すべき要素の特定に利用される。クラス `Infection` は defective 或いは straying に分類される `Statement` と、latent 或いは infectious に分類される `ValueReference` より構成される。

4.4　アルゴリズム

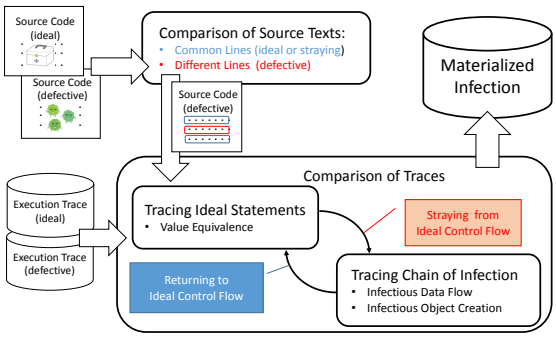

図 3　実行の比較

現在我々は誤てるプログラム実行から図 2 によって定義されるデータを生成するアルゴリズムを現在実装中である。我々は過去に開発した独自のトレース生成フレームツール [9] を用いてプログラムのトレースを生成する。本トレースは Java のバイトコード命

令 (byte code instruction)[†1] の実行を記録したものである。図 2 の `Statement` は複数のバイトコード命令によって実装されている。トレースにはバイトコード命令による値の生成や参照関係が記録されており、これらが `ValueReference` を実装している。

トレース中のバイトコード命令や値の生成参照関係には特定の解析目的に用いるためのデータを追加するための拡張点 (extension point) が予め設定されている。命令文の実行と値の参照の種類を表現するデータはこの各頂点を用いて実装する。

アルゴリズムの概要を図 3 に示す [†2]。本アルゴリズムは二つのソースコードの字面の比較とトレースの比較の二つの処理過程から構成される。ソースコードを比較する目的は誤てるプログラムのソースコードの不具合部分の特定である。次のトレースの比較過程では、この不具合部分の情報と有るべきトレースを利用して誤てる命令実行と値を分類する。

誤てるソースコードの不具合部分の特定処理では、それぞれのソースコードの文字列を比較する。この処理に従って行単位で不具合個所が特定される。Java の byte code には属性値として行番号が付されており、これによって実行されるバイトコード命令と命令文を行単位で対応付けられる。

トレースの比較過程では二つのトレースのバイトコード命令を一つ一つ順に辿る。この処理過程はさらに二つの処理過程に分割され、双方を行き来ながら解析が進められる。分割された一つ目の処理過程では ideal な命令文を実装するバイトコード命令を追跡しながら命令文の実行を再現する。もう一つ過程では defective 或いは straying な命令文に対応するバイトコード命令の実行による感染の連鎖を追跡する。

先に述べたようにプログラムの実行は ideal な命令の実行によって開始される。defective なバイトコード命令が実行されるか、或いは二つのトレースの間で異なる分岐が実行されるか、異なるメソッドが選択される時に感染の追跡に移る。分岐を抜けるかメソッドの呼び出しが完了 (或いは例外処理で終了) した時点で有るべき制御の流れに復帰したか否かを確認する。復帰している場合は ideal な命令文の実行の再現に移る。トレースの比較はどちらか片方のバイトコード命令が最後まで辿られた時点で終了する。

ideal な命令文の追跡では、誤てるトレースと有るべきトレースの間の命令文の対応を確認する。両者がオブジェクトの生成命令である場合、生成される二つのオブジェクトは第 4.2 節で定義した同値関係にある。この同値関係を用いて以降の値の参照を分類する。

感染の追跡では defective 或いは straying な命令文を実装するバイトコード命令によるデータフローの形成を追跡し、オブジェクトの生成を記録する。再現された ideal の命令文によるこれらの値の参照は latent 或いは infectious に分類される。さらに有るべきトレースのバイトコード命令も辿りながら、ideal な命令文の追跡に移るまでに実行された配列やインスタンス/クラス変数への代入を調査する。これらの代入は誤てる実行で実行されなかった「失われた」副作用を実装している。ideal な命令文の再現で、対応する有るべき実行の命令文がこの「失われた」代入値を参照する時、ideal な命令文での代入値の参照は latent 或いは infectious に分類される。

5 議論

第 4.4 節で説明したように、誤てるプログラムの不具合部分はソースコード同士の単純な文字列の比較によって実装される。よって不具合部分の特定処理は文字列の比較アルゴリズムに実質的に依存している。また第 3 節で紹介した例のように不具合の修正が単純である場合には良いが、再構築 (refactoring) [3] が必要な場合にはこの方法が上手く行かない事は容易に予想される。

我々のトレースに含まれている情報は命令文では無く命令文を実現するバイトコード命令の実行である。よってバイトコード命令とソースコード中の命令文を対応付けが必要となる。デバッガの実装でも本質的に同じ問題が発生する [19]。バイトコード命令は行番号に基づいて命令文の対応付けられる。同じ行に同じ種

†1 ただしスタックを操作するものは除外されており、局所変数の読み込みはデータ依存関係に統合されている。

†2 詳細に関しては [10] を参照。

類の命令文が並んでいない場合には我々の現在の実装は問題無いかもしれないが、そうでない場合には問題解決が困難になる事が予想される。現実的ではないが、例えばソースコード中の構文要素を全て一行に並べたプログラムに対して不具合部分の特定は極めて困難である。

実体化された感染はトレースを用いて実装される。我々のトレースは依存関係に関する情報を含んでいるため実用的なプログラムの実行トレースは容易にビッグデータ化してしまう。そのため所謂**逆回しデバッギング** *(Back-In-Time Debugging)* [18] の実現に関するものと本質的に同じ問題を抱えている。我々は現在この問題に関してグラフデータベースを用いた解決を模索中である [12] [13]。

6 関連研究

デバッガを含むソフトウェア工学ツールの有効性を検証する上で**制御実験** *(Controlled Experiment)* は重要な役割を果たす。しかしながら制御実験の結果は、評価されるツールの優劣よりも作業を実際に行う個々の被検者の技量に大きく左右される可能性を排除出来ない。結果として期待された結果が得られない、或いは実験の正当性が担保されない危険性が制御実験による評価を困難なものとしている [6] 。

我々は先行研究 [11] で診断形成時に対話的にツールを利用する診断者の内面を分析した。その分析結果から被検者の認識の妥当性を評価する事によって実験結果に及ぼす被検者固有の要因を洗い出し、制御実験の失敗の危険性を減少させる事が可能であるとの展望を持つに至った。本稿で述べた失敗の実体化は妥当性評価を効率的に遂行する仕組みを実装するためには不可欠な要素の一つである。

妥当性評価の対象である診断者の認識を定義に関連して、我々は (1) 診断者による手掛り (clues) の選択 [5] と (2) 関連性の理解 [2] の二点に着目している。手掛かりには診断者デバッガを用いて観察した実行時点が含まれる。手掛かりの選択によって診断の成否や作業の効率が大きく左右される [5]。また関連性の理解は被検者の熟練度と強い相関関係を有している事が実証されている [2] 。

我々は診断者が自身が注目する関連性に基づいてこれらを互いに位置付け、或いは新たな手掛かりを取得していると予想している。ツールの利用によって感染の連鎖とより関連が深い手掛かりが選択され、それらが感染の連鎖の追跡に適した形で位置付けられるようになっているか否かを示す事によって、個々の被検者のバイアスを排しツールの効果が実現出来ると考えている。実体化された感染を用いる事によって、こうした手掛かりや関連性の評価が可能になると考えられる。

7 終わりに

本論文ではデバッグに於ける診断形成に焦点を当て、診断者の思考の適切性を評価するための要素技術として、我々が現在開発している**失敗の実体化** *(Failure Materialization)* と呼ばれる動的解析手法を提案した。失敗の実体化はプログラム実行の失敗に於ける感染の連鎖の発生をデータ化する動的解析手法である。データ化された感染の連鎖は思考の適切性を客観的に評価するための根拠として利用される。本論文では失敗の実体化を構成する三つの要素として、感染を規定する規則、感染の連鎖を表現するデータのメタモデル、動的解析のアルゴリズムを説明すると共にその限界と将来課題を議論した。

Acknowledgment

本研究は以下の助成金を用いて遂行された: 文部科学省科研費挑戦的萌芽 (No. 15K12009)、基盤 (B)(No. 16H02908, 15H02701)、基盤 (A)(17H00731)

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] Fix, V., Wiedenbeck, S., and Scholtz, J.: Mental Representations of Programs by Novices and Experts, *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, New York, NY, USA, ACM, 1993, pp. 74–79.

[3] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] Karahasanović, A., Levine, A. K., and Thomas, R.: Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study, *Journal of Systems and Software*, Vol. 80, No. 9(2007), pp. 1541–1559.

[5] Ko, A., Myers, B., Coblenz, M., and Aung, H.: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks, *Software Engineering, IEEE Transactions on*, Vol. 32, No. 12(2006), pp. 971–987.

[6] Ko, A. J., LaToza, T. D., and Burnett, M. M.: A practical guide to controlled experiments of software engineering tools with human participants, *Empirical Software Engineering*, Vol. 20, No. 1(2015), pp. 110–141.

[7] Ko, A. J. and Myers, B. A.: Designing the whyline: a debugging interface for asking questions about program behavior, *SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2004, pp. 151–158.

[8] Ko, A. J. and Myers, B. A.: A framework and methodology for studying the causes of software errors in programming systems, *Journal of Visual Languages & Computing*, Vol. 16, No. 1–2(2005), pp. 41–84.

[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., Nakamura, M., Nitta, N., and Shibayama, E.: Analyzing Execution Traces of Failed Programs for Materializing Chain of Infection, *International Conference on Big Data, Cloud Computing, and Data Science Engineering (BCD 2017)*, ACIS, 2017.

[11] Kume, I., Nakamura, M., Tanaka, Y., and Shibayama, E.: Evaluation of Diagnosis Support Methods in Program Debugging by Trace Analysis: An Exploratory Study, *15th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2016)*, 2016.

[12] Kusu, K., Kume, I., and Hatano, K.: A Trace Partitioning Approach for Efficient Trace Analysis, *4th International Conference on Applied Computing & Information Technology (ACIT 2016)*, ACIS, 2016.

[13] Kusu, K., Kume, I., and Hatano, K.: A Node Access Frequency based Graph Partitioning Technique for Efficient Dynamic Dependency Analysis, *The Ninth International Conferences on Advances in Multimedia (MMEDIA 2017)*, IARIA, 2017.

[14] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., and Fleming, S.: How Programmers Debug, Revisited: An Information Foraging Theory Perspective, *Software Engineering, IEEE Transactions on*, Vol. 39, No. 2(2013), pp. 197–215.

[15] Lienhard, A.: *Dynamic Object Flow Analysis*, Lulu.com, 2008.

[16] Murphy, G. C., Kersten, M., Robillard, M. P., and Čubranić, D.: The Emergent Structure of Development Tasks, *ECOOP 2005 - Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2005, pp. 33–48.

[17] Murphy, G. C., Kersten, M., Robillard, M. P., and Čubranić, Davor: The Emergent Structure of Development Tasks, *"ECOOP"*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2005, pp. 33–48.

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

[19] Rosenberg, J. B.: *How Debuggers Work: Algorithms, Data Structures, and Architecture*, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[20] Sillito, J., Murphy, G. C., and De Volder, K.: Questions Programmers Ask During Software Evolution Tasks, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, New York, NY, USA, ACM, 2006, pp. 23–34.

[21] Sillito, J., Murphy, G. C., and Volder, K. D.: Asking and Answering Questions during a Programming Change Task, *IEEE Transactions on Software Engineering*, Vol. 34, No. 4(2008), pp. 434–451.

[22] Storey, M.-A. D., Fracchia, F. D., and Müller, H. A.: Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration, *J. Syst. Softw.*, Vol. 44, No. 3(1999), pp. 171–185.

[23] Tip, F.: A survey of program slicing techniques, *Journal of Programming Languages*, Vol. 3(1995), pp. 121–189.

[24] Wang, X., Cheung, S. C., Chan, W. K., and Zhang, Z.: Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization, *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Washington, DC, USA, IEEE Computer Society, 2009, pp. 45–55.

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

[26] 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.