Vol.2021-SE-209 No.11
2021/11/25

# 関心を反映した実行の抽象化と比較によるデバッグ支援に向けて

久米 出[1,a)]　新田 直也[2,b)]　中村 匡秀[3,c)]　柴山 悦哉[4,d)]

**概要**：プログラム実行が失敗した時に、その実行が成功する場合と比較する事によってデバッグの手掛かりを得られる事が期待される。しかしながら実用規模のプログラムの実行は非常に複雑であるため、従来のデバッガを用いた比較では有用な情報を効率的に取得する事は困難である。
複雑性の問題を解決するために、我々はブレークポイントとステップ実行の繰り返しに代わる、従来とは全く異なる方式のデバッグを可能とする**関心指向デバッギング (Concern-Oriented Debugging** を提案する。関心指向デバッギングでは (1) プログラム実行への作業者の関心を媒介変数とする**変数付き抽象化 (Parametric Abstraction)** の適用と、(2) 抽象化された実行同士の比較を対話的に実施する形でデバッグ作業を遂行する。
本稿では関心指向デバッギングによる複雑性への対処を、我々が現在開発しているデバッガを用いた実例を用いて説明する。さらにこの実例で用いられるデバッガが提供すべき機能とその用法をそれぞれ説明する。デバッガの機能はトレース (プログラムの実行履歴) を解析する事によって実現される。本稿ではこの実現に向けた技術的課題に関して議論する。

**キーワード**：デバッグ，実行比較，関心事，抽象化，動的解析

# Pursuing a Support by Abstraction of program Executions based on Concerns and their Comparison

***Abstract:*** We can expect that comparison of a failed program execution with a successful one gives us useful hint of debugging. However, an execution of a practical program is so complex that we can't effectively find the cause of the failure by widely used debuggers.
In order to tackle this complexity problem, we introduce **Concern-Oriented Debugging**, a novel debugging method which enables a novel debugging style instead of ordinary one based on breakpoints and step executions. In the style of Concern-Oriented Debugging, developers interactively (1) apply abstraction parameterized by developers' concerns to program execution and (2) comparison of abstracted results.
In this paper, we explain the ability of Concern-Oriented Debugging to cope with the complexity problem by a case study in which a developer uses a debugger, which is under development. The features of the debugger and their uses in the case study are also explained. Our debugger uses program execution traces to implement its features. We discuss technical issues about this implementation.

***Keywords:*** Debug, Execution Comparison, Concerns, Concerns, Dynamic Analysis

## 1. はじめに

ソフトウェア開発の現場でプログラム実行を調査する作

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[2] 甲南大学
Konan University
[3] 神戸大学
Kobe University
[4] 東京大学
The University of Tokyo
[a)] kume@is.naist.jp
[b)] n-nitta@konan-u.ac.jp
[c)] masa-n@cs.kobe-u.ac.jp
[d)] etsuya@ecc.u-tokyo.ac.jp

業者の多くは意識するしないにかかわらず実質的に科学的デバッグ [1] を実践している [2] 。この種の調査を実施する作業者はデバッガを用いて、デバッグ対象に対してブレークポイントを設定してプログラムの実行を一時停止し、ステップ実行を繰り返す作業を繰り返す。

こうした従来のブレークポイントとステップ実行によるデバッグではプログラム実行の複雑性のために膨大な労力と時間が消費される [2] 。複雑性は所謂**不実行の誤り (Execution Omission Error)**[3] の解決を一層複雑にする。

本稿では実行時のオブジェクトグラフとループ処理が絡み合う形で複雑な実行が形成され、その中で不実行の誤りが発生する状況を想定し、その不具合箇所の特定を支援する問題とその解決に向けた我々の試みを説明する。対象言語としては Java を想定する。

複雑性の問題を解決するために、我々はブレークポイントとステップ実行とは全く異なる新しいデバッグ方式である**関心指向デバッギング (Concern-Oriented Debugging** を提案する。関心指向デバッギングでは (1) プログラム実行への作業者の関心を媒介変数とする**変数付き抽象化 (Parametric Abstraction)** の適用と、(2) 抽象化された実行同士の比較を対話的に実施する形でデバッグ作業を遂行する。

本稿ではこの種の複雑性の下で発生した不実行の誤りを、これら二つの操作を組み合わせる事によって特定出来る事を上述の事例を用いて説明する。さらにこれらの操作をデバッガの機能の実装に関して、我々が先に研究したトレース処理 [4], [5] とソースコード変換の手法 [6], [7] との関わりを説明する。

本稿の以降の部分は以下の内容から構成されている。第 2 節で例題として用いるグラフ処理プログラムを紹介し、その複雑性に起因するデバッグの問題を説明する。第 3 節では複雑性の問題に対処可能な新しいデバッグ方式を提供する関心指向デバッギングの概要を述べる。

## 2. 問題提起

### 2.1 例題プログラム

本論文では一貫してある実用規模のグラフ処理プログラムをデバッグの例題として採用する。処理されるグラフは各辺に向きが付けられており、循環構造は存在しないものとする。このプログラムは各節点で枝分かれした分岐がその先のどの節点で合流するのか、あるいはしないのかを判定する事を目的としている。

グラフの節点と辺はそれぞれがクラスのインスタンスとして実装されており、辺とその始点と終点は相互に参照し合っている。図 1 に実行時のグラフの構成が示されている。角丸が節点と、楕円が辺を表している。図を見易くするために、ここでは節点から辺への参照は省略されている。

この図中の節点 f の分岐はその先にある節点 Join(f) で合流する。一方節点 s は節点 Join(s) で合流する。辺が一本しか伸びていない (分岐していない) 節点はその辺の終点で合流する。

この例題プログラムはグラフの始点から終点を、各辺を一度だけ辿りながらグラフ上の節点を訪問するようにアルゴリズムが設計されている。各節点は訪問時にこれまでの経路とその処理に関する情報が渡される。この情報を用いてその節点で合流した分岐の有無が判定される。全ての節点の訪問を終えた時点で合流判定されなかった分岐は合流点が存在しないと判定される。

図 1 の右下に示されている繰り返し命令は、上述した節点の訪問と経路情報の処理を実行するよう設計されている。コード中の変数 `p` は新たに訪問された節点が代入され、メソッド `start()` の呼出しにより経路情報の処理が開始される。この節点から別の節点が訪問される際にはそれらがメソッド `start()` の返り値として返され、処理待ちのスタック `processes` に積まれる。まだその先の節点を訪問する時機でない時には `null` 値が返される。

### 2.2 複雑性に起因するデバッグの問題

この例題プログラムを実行すると、合流する箇所が正しく特定される節点とそうでない節点が存在する。図 1 の例だと節点 Join(s) に対しては正しく判定されるが、節点 Join(f) に対しては合流点の特定に失敗してしまう。本節では通常のデバッガを用いてこの失敗の原因を特定する状況を想定する。

まず、本事例の実行時の複雑性に関して説明しよう。本例題プログラムは第 2.1 節の最終節で説明したようにグラフ中の節点を参照している。オブジェクトがここで参照される順番は、はグラフの構造と、繰り返される処理の結果の積み重ねに依存する。

第 2.1 節で述べた、節点に渡される経路情報は、それ自体があるクラスのインスタンスとして表現されている。これらのオブジェクトが自身の状態として保有する経路の内容も、節点の参照順序と同様にグラフの構造と、繰り返された処理結果の積み重ねによって決定される。

上述したグラフ構造と繰り返し処理の組み合わせは明らかに実行時の複雑性を形成するものである。(図 1) この複雑性はて従来のデバッグ方式の大きな障害となる、所謂 “長い実行 (long execution)“[2] と、その長い実行での途上で辺を辿った軌跡の履歴という形で顕在化する。

作業者はデバッグの手掛かりを得るために、正しく処理されたオブジェクトとそうでないものを特定し、それぞれの処理を比較試みるかもしれない。こうしたオブジェクトを特定しようとする作業は上述の複雑性のために困難が予想される。多大な労力と時間を掛けて特定したとしても、その作業で得られた情報は “一時的なもの (ephemeral)”[8]

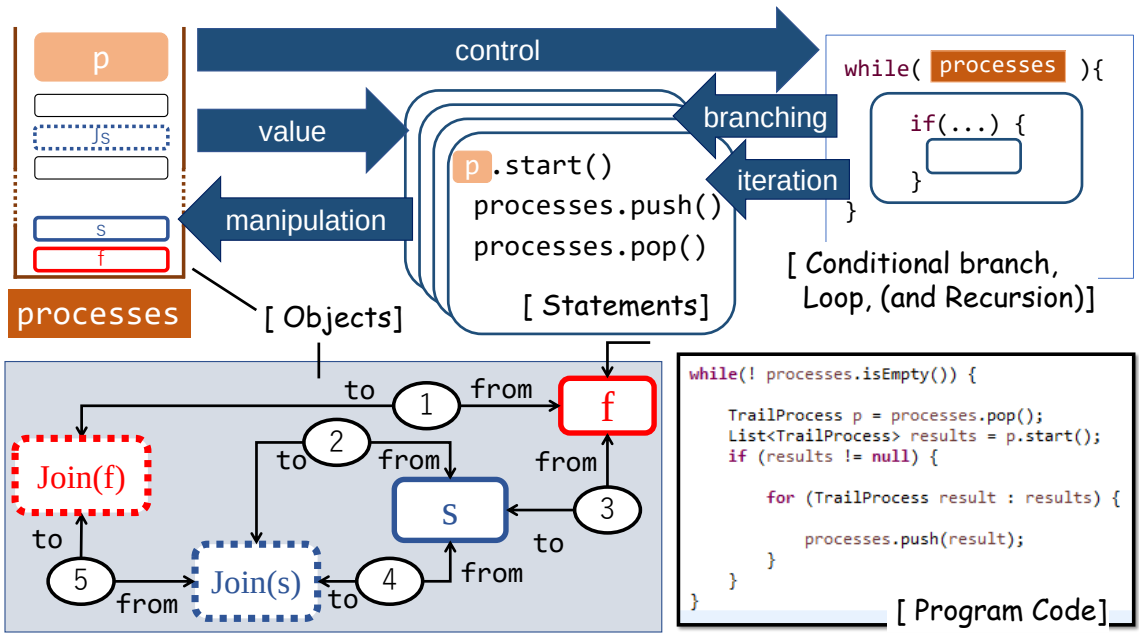


図 1　ループとオブジェクトグラフの相互作用による複雑性の形成

であるため、実行を遡って調査しようとした時点で消滅してしまう。

実行の流れの追跡でメソッドの呼び出しに遭遇すると、しばしばその返り値から実行時の異常の兆候の有無を確認しようとする。メソッド `start()` 内部では合流判定の成功時に分岐元と合流先の対が作成登録されるが、合流判定の結果は返り値には反映されていない。よって呼出しの効果を調べるためにはメソッド内部の処理の追跡が必要である。

メソッド内部の処理では経路情報が参照される。本実行例では `null` 値のような、明確に不正な値は出現しない。よって経路情報の正誤を判定するためには、過去に辺を辿った履歴との整合性確認しなければ分からない。上述した複雑性の存在のため、通常のデバッガではこの履歴を効率的に取得する事は困難である。

## 3. 提案手法

**関心指向デバッギング (Concern-Oriented Debugging)** はデバッグ対象プロセス (debuggee) をトレースとして表現し、そこから作業者が必要とする情報を効率的に取得するための二種類の操作を提供するものである。一つ目は作業者が指定した関心事を反映する形でデバッグ対象プロセス全体或いはその一部を抽象化するもので**変数付き抽象化 (Parametric Abstraction)** と呼ばれている。もう一つは抽象化された結果同士を比較するもので、**双模倣性追跡 (Bisimulation Tracking)** と我々は呼んでいる。

変数付き抽象化を適用するためには抽象化される範囲を入力として指定すると共に、媒介変数値として、抽象化の対象となる実行の側面を指定する。このような側面はオブジェクトやデータの流れ、或いは条件分岐等の制御の流れとして表現する事が可能である。

プロセズ全体のように抽象化の対象範囲が広い場合、関心指向デバッギングはある特定のオブジェクトに関連した指定を推奨する。このように指定されるものには Object Flow[9] や、オブジェクトに対する操作の履歴 [10], [11] が含まれるがそれらにのみに限定されない。

オブジェクトやデータの流れ以外にも、あるメソッド呼び出しの内側、あるいは個々のメソッドの枠を超えた制御の流れを指定する事も可能である。関心指向デバッギングではある特定のイベントに対して、それの制御上の依存関係を特定のメソッド呼出しの内部限定、或いはメソッド呼出しの階層も遡った分析結果を**条件付き制御の連鎖**と呼び、抽象化の媒介変数値として指定する事が出来る。

媒介変数のもう一つの値として、指定された範囲、側面の実行から作業者に提示する情報の種類を指定する。媒介変数の値として指定される側面はそれぞれ自身の実行を表現する複数の属性を定めている。媒介変数の二つ目の値として作業者に提示すべき属性を指定する。

双模倣性追跡はコード中の同じ箇所から開始される二つの実行過程を比較し、その相違点を抽出する操作である。この操作を適用するためには、比較される二つの実行過程の抽象化の実行の側面同士が比較可能でなければならない。また比較の結果もこれらの側面によって決定される。

比較されるもの同士の側面によっては本質的な差異は同一視するように、比較の厳密性を調整する事が可能である。こうした調整は双模倣性追跡の省略可能な (optional) な引数として指定出来る。

変数付き抽象化と双模倣性追跡に与えられる値やその結果には、実行を構成する様々な種類の要素が含まれている。

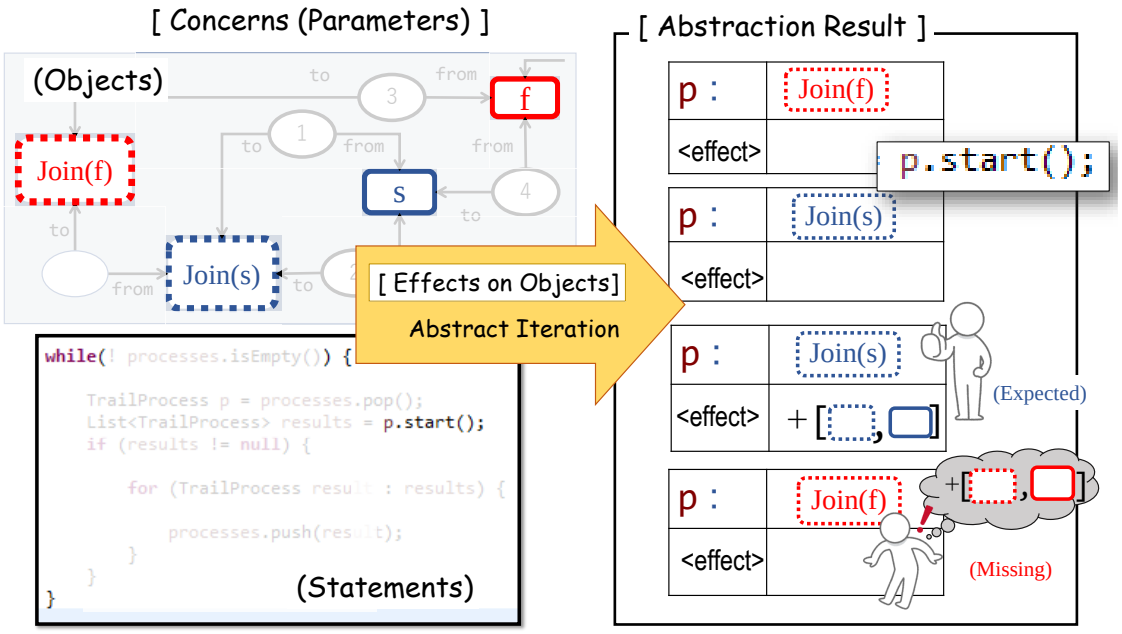


図 2　特定の節点オブジェクトの流通に関する側面で繰り返し命令内のメソッド呼出しを抽象化し、登録による参照構造の導入の有無を提示させた結果

操作対象となる範囲はその開始と終了に当たる命令文の実行や式の評価によって指定される。特定のオブジェクトを用いてその Object Flow を表現する。条件付き制御の連鎖は互いに制御依存関係にある条件分岐命令文の実行が含まれる。

ブレイクポイントとステップ実行によるデバッグでは、こうした実行時の要素を直接指定する手段が提供されていない。命令文や式の実行に関しては、それらが記述されているソースコード行にブレークポイントを設定する事によってその実行の時機を拾う事が可能であった。

関心指向デバッギングで利用されるトレースでは、こうしたオブジェクト、命令文の実行、式の評価全てが作業者が参照出来る第一級データ (first class data) として表現されている。繰り返し命令文の実行時の個別の繰り返しも同様である。

## 4. 適用事例

変数付き抽象化と双模倣性追跡の使用例を示すために、第 2 節で紹介した例題に対してこれらを適用する作業シナリオを紹介する。このシナリオではこれらの適用によって、感染の連鎖 [1] の途上で発生した不実行の誤りの特定に成功する。

まず、図 1 で示される繰り返し命令文のある実行によって繰り返されているエソッド呼出し全てを対象として変数付き抽象化を適用する。作業者は合流点として特定された節点オブジェクトと、判定されるべきにもかかわらず判定されなかったオブジェクトの二つに関心を有している。以下の手順でこれらから抽象化に変数値として渡される実行の側面を生成する。

第 2.2 で紹介した処理が成功する節点オブジェクトと、失敗するものとして、図 1 に示される Join(s) と Join(f) を選択する。前者は正しく節点 s の合流点として登録されているが、後者は登録されていない。選択された二つのオブジェクトそれぞれに対して、Object Flow を作成する。

上述の Object Flow からメソッド呼出し `p.start()` の呼出しの受け手として参照が開始されている断片を抽出する。抽出された Object Flow の各断片に対して以下のように提示内容を指定する。

まず、起点となる呼出しの受け手と、合流点として受け手に向けられた新たな参照構造の二つを作業者に対して提示情報として指定する。これによって各メソッド呼出しの中から上記の正例負例を体現している二つのオブジェクトに対してそれらが処理されている `p.start()` の特定と、合流点としての登録の有無を調べた結果が図 2 に示されている。

特定されたメソッド呼出しは実行された順番に従って並べられている。これらを調べた結果、正しく処理された節点 Join(s) は最後に呼出された `p.start()` の内部で合流点として参照される事が分かった。この実行と Join(f) の最後の呼び出しに対して、双模倣性追跡を適用し、実行内容が分かれた箇所を特定する事によって、後者の実行では前者のような参照構造が形成されなかった、則ち不実行の誤りが発生した過程を明らかにする。

まず、参照構造を正しく形成したイベントに対して、そ

我々の実装ではこれらは Java のインタフェイスのインスタンスであり、インタフェイスの API を通じてこれらの取得や操作が可能である。

のイベントから条件付き制御の連鎖を形成する。これと不実行の誤りが発生した実行そのものを比較する。比較の起点はそれぞれの `p.start()` 呼出しである。二つの呼出しで同じコードが実行される事に留意して欲しい。

合流点判定に失敗した実行は条件付制御の連鎖のどこかで実行が逸れる筈である。なぜなら最後まで逸れなければ、Join(f) を合流点として登録したイベントが Join(f) に対しても同様に発生する筈だからである。双模倣性追跡によって、条件分岐に失敗した箇所の効率的な特定に成功した。(図 3) 以降では

## 5. 関連研究

プログラム実行の比較をデバッグに利用するという着想は新しいものではなく、様々な手法が提案されてきた。その多くは不具合箇所の特定を指向するものであるが、現状ではソフトウェア開発の現場に於ける標準的な支援手法として用いられるには至っていない [2], [12]。

デバッグ支援のためのトレースの主な利用法の一つとして**逆回しデバッギング (Back-In-Time Debugging)** の実現が挙げられる [13], [14], [15], [16], [17], [18], [19], [20] 。標準的なデバッガは現在作業者が調査している実行時点より前に実行された内容を追跡する機能を有していない。そのため、ある変数の値がどの時機に代入されたのか、直前に呼出しが完了したメソッドの内部でどのような処理が実施さらたのかを、再実行せずに調査する事は不可能である。

逆回しデバッギングは作業者が調査している実行時点に対して、それから遡った実行に戻って追跡する作業を可能とするものである。しかしながら、単純に過去に戻る機能を提供するだけでは、ステップ実行を用いた従来の調査方式の延長上の手法と位置付けられ、プログラム実行の複雑性の問題への対処という面で課題が残っていると考えられる。

トレースを利用して不実行の誤りの特定を支援する手法も研究されている [3], [15] 。この支援はメソッド役割や呼出し方に関する知識を作業者が有している場合に効力を発揮するものと期待出来る。作業者がこうした知識を有していない場合には関心指向デバッギングを適用した調査が有効であると考えられる。

その他の例として Treffer 等による研究 [20] では動的スライスを対話的に取得する手法が提案されている。第 3 節で我々が説明した、変数付き抽象化の媒介変数の値として動的スライスの利用を我々は検討しており、こうした視点からこの研究に注目している。

## 6. 議論

我々は過去の研究 [4], [5] で、Java プログラムの実行を「ありのまま」トレースとして保存するツールを開発済みであり、これによって第 3 節で説明したトレースの諸要素の第一級データとしての実装を実現している。これによって変数付き抽象化と双模倣性追跡の多様かつ柔軟な適用が可能となっている。

作業者はプログラム実行をソースコードと関連付けて理解している。よって上記の「ありのまま」トレース化された諸要素は、個別の式や命令文の粒度で関連付けられる事が求められる。「比較でどの程度の違いを無視」する際に個別の式の単位で指定する事が可能だからだ。この問題に関しては、既に我々の過去の研究 [6], [7] によってこうした細かい粒度での対応付けが実現済みである。

抽象化の変数値となる実行の側面は様々な種類のものが取り扱える事が望ましいが、一方でそれは取り扱いの効率性とは相反する要求かもしれない。相反する要求の釣り合いを保つような形式化は今後の課題である。

## 7. おわりに

本稿ではプログラム実行の複雑性が効率的なデバッグの遂行を妨げる大きな要因である事を実用的なプログラム例題を用いて説明した。従来のブレイクポイントとステップ実行を用いたデバッグ方式は複雑性の影響を強く受けてしまうため、これらを用いない新しい方式のデバッグを可能とする**関心指向デバッギング (COncern-Oriented Debugging)** を提案した。

関心指向デバッギングは作業者の関心を反映した実行の抽象化と、それらの比較を通じて複雑性に対処しつつプログラム実行を調査する事を可能とする。本稿ではオブジェクトの参照構造と繰り返しの制御構造が相互に作用する形で発生する複雑性に対して、関心指向デバッギングを適用する作業シナリオを通じてその有効性を示した。

### 参考文献

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

[2] Perscheid, M., Siegmund, B., Taeumel, M. and Hirschfeld, R.: Studying the advancement in debugging practice of professional software developers, *Software Quality Journal*, Vol. 25, No. 1, pp. 83–110 (online), DOI: 10.1007/s11219-015-9294-2 (2017).

[3] 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, pp. 415–424 (online), DOI: 10.1145/1250734.1250782 (2007).

[4] 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, pp. 82–91 (2017).

[5] 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)*,

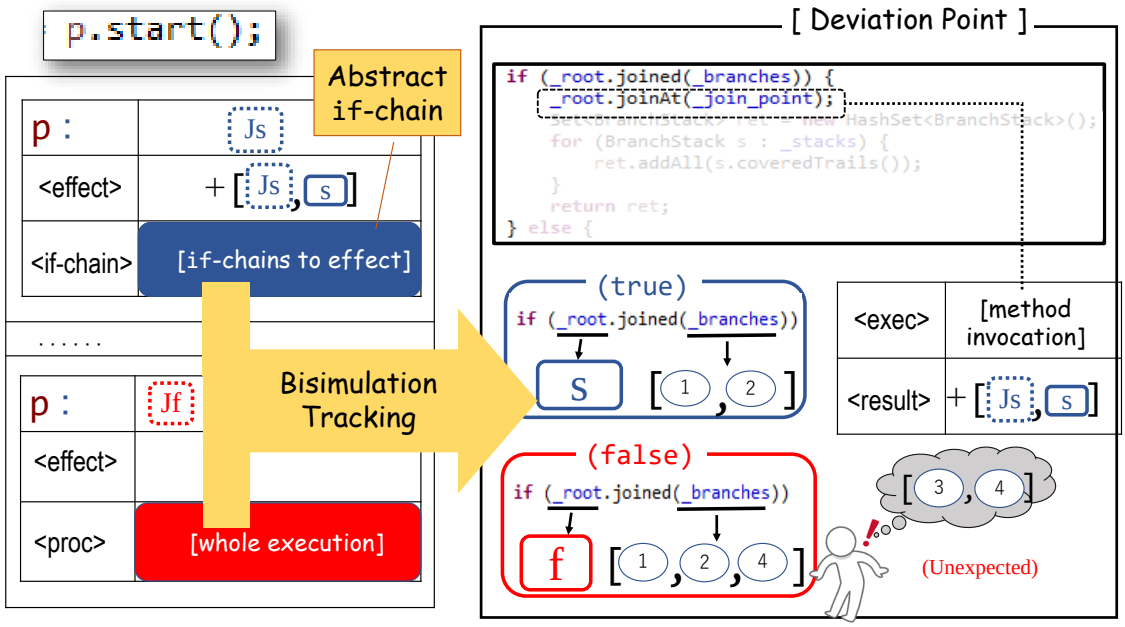


図 3　正しく実行された条件付き制御の連鎖と、実行が失敗したメソッド呼出しそのものとの比較

Vol. 3, No. 3, pp. 26–40 (online), available from ⟨http://dx.doi.org/10.4018/IJSI.2015070103⟩ (2015).

[6] 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, pp. 37–46 (online), DOI: 10.1145/3318236.3318259 (2019).

[7] 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).

[8] Spinellis, D.: Debuggers and Logging Frameworks, *IEEE Software*, Vol. 23, No. 8, pp. 98–99 (online), DOI: https://doi.org/10.1109/MS.2006.70 (2006).

[9] Lienhard, A.: *Dynamic Object Flow Analysis*, Lulu.com (2008).

[10] Quante, J.: Do Dynamic Object Process Graphs Support Program Understanding? – A Controlled Experiment, *International Conference on Program Comprehension*, IEEE, pp. 73–82 (2008).

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

[12] Monperrus, M.: A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair, *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, ACM, pp. 234–242 (online), DOI: 10.1145/2568225.2568324 (2014).

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

[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, pp. 558–582 (online), DOI: 10.1007/978-3-642-22655-7_26 (2011).

[15] 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, pp. 1962–1969 (online), DOI: 10.1145/2695664.2695735 (2015).

[16] Khoo, Y. P., Foster, J. S. and Hicks, M.: Expositor: Scriptable Time-Travel Debugging with First-Class Traces, *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press, pp. 352–361 (online), DOI: 10.5555/2486788.2486835 (2013).

[17] Barr, E. T., Marron, M., Maurer, E., Moseley, D. and Seth, G.: Time-Travel Debugging for JavaScript/Node.Js, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, New York, NY, USA, Association for Computing Machinery, pp. 1003–1007 (online), DOI: 10.1145/2950290.2983933 (2016).

[18] Honarmand, N. and Torrellas, J.: Replay Debugging: Leveraging Record and Replay for Program Debugging, *SIGARCH Comput. Archit. News*, Vol. 42, No. 3, pp. 445–456 (online), DOI: 10.1145/2678373.2665737 (2014).

[19] Treffer, A. and Uflacker, M.: Back-in-Time Debugging in Heterogeneous Software Stacks, *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Toulouse, France, October 23-26, 2017*, pp. 183–190 (online), DOI: 10.1109/ISSREW.2017.62 (2017).

[20] 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, pp. 175–180 (online), DOI: doi.ieeecomputersociety.org/10.1109/ISSREW.2016.17 (2016).