繰り返し構造に潜む不具合を効率的に特定するためのソースコード変換手法

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

概要：
標準的な Java デバッガを用いてプログラム実行のログを取得するためにはソースコードや条件付きブレイクポイントのコード編集が必要である。本論文ではコード編集の代わりにソースコード上の簡単な操作によって必要最小限の情報を含むログを生成する全知デバッガの機能と、その実装に用いたソースコード変換手法を紹介する。本ログ生成機能は繰り返し構造の中に潜む不具合を効率的に特定する作業の支援を目的としている。

Source Code Analysis for Efficient Diagnosis of Chain of Infection in Loop Statement

1. はじめに

現代に至るまでデバッグ支援のための様々な手法が研究される一方で、現実のソフトウェア開発ではコーディングによって実行時の特定の式の値をログ出力する古典的な printf デバッグや条件付きブレイクポイントが採用されている [3], [6], [8] 。

こうした手法は例えばプログラムの繰り返し構造の各回の処理の内容を、式の評価値によって把握する場合に有効である。しかしながら現在広く用いられているデバッグ環境下ではこうした作業には作業者によるコーディングが必要であり、これがログ生成の効率化の妨げとなっている。

本論文ではコーディングを経ずに必要な式の値をログ出力する全知デバッガの機能と、それを実現するソースコード変換手法 [5] を説明する。本ログ生成機能は繰り返し構造を有する Java プログラムのデバッグ支援を目的としている。本変換手法によってバイトコード命令の実行はソースコード中の式に自動的に対応付けられる。これによって作業者が指定した式がその都度評価される値を特定する事が可能になる。

[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] 東京大学
The University of Tokyo
[4] 神戸大学
Kobe University
[a)] kume@is.naist.jp
[b)] n-nitta@konan-u.ac.jp
[c)] etsuya@ecc.u-tokyo.ac.jp
[d)] masa-n@cs.kobe-u.ac.jp

2. 背景

デバッグの各朱の作業には、既存のデバッガによる支援が困難なものが含まれている事が明らかにされている [2]。実際 printf デバッグのような旧来の手法が現在でもなお利用されている事から [6] 、既存のデバッガでは必ずしも効率的に遂行出来ない問題が存在する事が分かる。

古典的な printf デバッグではソースコードに出力のための命令文を追加して実行する事によって特定の式のログを取得する。既存のデバッガを用いる場合でもブレイクポイントの停止条件ノコーディングガ必要となる [3]。こうしたコーディングが必要である事から効率的なログ取得は一般に困難である。

ソースコード中の式や命令文に対してそれらを評価するバイトコード命令を応付けられれば、それらをマウス操作等で指定するだけでログの取得が可能となり、コーディングは不要となる。しかしながら標準的な Java コンパイラはバイトコード命令のソースコード中の位置に関してその行番号以上の情報を与えない。行番号のみをから任意の式

にバイトコードの実行を対応付けるのは一般に困難である。

3. 繰り返し構造を対象としたログ生成機能

繰り返し構造を有するコードのどこかに不具合が潜んでいる場合、繰り返される処理の何度目かの実行で感染 (状態の誤り) が発生し、以降の繰り返しの過程で連鎖して障害 (外部から観測可能な実行の誤り) の発生に至る。このような過程で発生する感染は繰り返しの過程で何度も評価される式の値のログを作成し、異常な値の発生を調べる事によって感染の発生源 (不具合箇所の実行時点) の特定に繋がる手掛かりの取得を期待出来る。

必要な情報を過不足無く取得するためには対象となる式の適切な選択が必要となる。しかしながら選定された式が適切か否かはログの内容を調べてみなければ分からない。よって適切な選択に至るまでには試行錯誤がしばしば不可避であると考えるべきである。ログ生成の試行錯誤の過程で何度もコーディングを繰り返さなくても済む事が望ましい。

我々は現在、ソースコード上のマウス操作等によって局所変数宣言と式を指定し、繰り返しの各回毎に (1) 局所変数値が変更されない範囲、及び (2) 式の評価値 (もし評価されている場合) をログ出力する全知デバッガの機能を開発している。本機能は我々が過去に開発したトレース生成・解析フレームワーク [4] 上に実装されている。

本機能の核心部分はトレース上に記録されたバイトコード命令の実行列から指定された式の評価を行っているものを特定する**式計測器** *(Expression Meter)* である。式計測器は我々が開発した**超改行法** *(Extreme Line Feeding)* と呼ばれるソースコード変換技法 [5] を用いて行番号のみでバイトコード命令を適切な命令文や式と対応付けられるようにする。超改行法はソースコードを字句毎に改行し、かつ構文上の特定のパターンに関して例外的な処理を施す事によって適切な対応付けを実現する。

4. 評価

我々は式計測器による対応付けを試験し、その挙動の正しさを検証した。試験のために様々な文法構造を含む計 56 個、3771 行の例題プログラムを作成し、それらの実行トレースを取得した。

トレース中のバイトコード命令に対して式計測器を適用し、バイトコード命令列の実行に対応する式と命令文を実行順に並べたものを印字した。印字された内容が Java の言語仕様に定める評価・実行の順序を満たす事を確認する事によって式計測器の処理結果の適切性を検証した。

式計測器は式は配列の初期化や、拡張 For 文の集合/配列式や局所変数への代入に関してもバイトコード命令を適切に対応付けている。また and/or 条件式に関してもそれらのどの部分式がどこまで実際に評価されたのかを正しく再現している。

5. 関連研究

近年、デバッグに於ける様々な作業やその遂行手順 (scenario) を類型化し、特定の類型に特化を対象とした機能の開発や拡張に関する研究が進められている [1], [2], [7] 。本研究は繰り返し構造のデバッグを対象とした支援手法として位置付けられる。

6. おわりに

本論文では繰り返し構造のデバッグを対象とした、コーディングを必要としないログ生成機能と、その実現に必要なソースコード変換手法を紹介した。

参考文献

[1] Chiş, A., Denker, M., Gîrba, T. and Nierstrasz, O.: Practical domain-specific debuggers using the Moldable Debugger framework, *Computer Languages, Systems & Structures*, Vol. 44, pp. 89 – 113 (online), DOI: https://doi.org/10.1016/j.cl.2015.08.005 (2015).

[2] Dupriez, T., Polito, G., Costiou, S., Aranega, V. and Ducasse, S.: Sindarin: A Versatile Scripting API for the Pharo Debugger, *15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2019)*, ACM (2019).

[3] 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 (2015).

[4] 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, pp. 26–40 (online), available from ⟨http://dx.doi.org/10.4018/IJSI.2015070103⟩ (2015).

[5] 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).

[6] 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).

[7] 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).

[8] Spinellis, D.: Modern Debugging: The Art of Finding a Needle in a Haystack, *Commun. ACM*, Vol. 61, No. 11, pp. 124–134 (online), DOI: 10.1145/3186278 (2018).

本研究は科研費による助成を受けている。(15K12009, 17H00731,16H02908)