プログラム実行時に発生したエラーの原因分析を支援する手法の提案

2020SE055 佐野杏朱

指導教員：名倉正剛

1 はじめに

プログラムの実行時にエラーが発生した際に，Java や Python など，実行時にコードを解釈しながら実行する実行環境を備えた言語では，発生したエラーを例外としてユーザに通知し，実行を停止する．そして通知の際に例外の種類とスタックトレースの情報をエラーメッセージとして表示する．そして開発者は，スタックトレースの情報が示すソースコードの場所を確認することにより，障害原因を確認できる．

2 エラーメッセージを基に原因分析を実施する際の課題

スタックトレースの情報が示す場所を確認しても，エラー原因を解消できない場合がある．例えば，ソースコード 1 に示す Python プログラムでは，関数 `total` は引数 `n` に整数を受け取ることを前提に定義されている．しかしこの関数への 4 行目の呼び出しで引数に指定される変数 `x` は，3 行目で文字列として定義されているため，実行時に変数 `n` が文字列型として扱われ，2 行目の `n + 1` の実行に失敗する（画面 1）．この画面では `Traceback` で始まる部分に `line 2` の場所で実行が中断したことを示しているが，3 行目の変数 `x` の定義に問題があり，エラーメッセージの示すソースコードの場所を確認しても原因を解消できない．変数の定義と利用される箇所の関係や関数の呼び出し関係を追跡しなければ不具合に対応できず，エラーメッセージで示される情報だけを基にエラー原因を分析することは難しい．

3 提案手法

3.1 概要

本研究では，実行時エラーに基づく例外情報を用いてソースコードを解析し，エラーの原因となる可能性のある式を特定する手法を提案する．具体的には，例外に示される文中の変数に関連する式を特定し，例外の種類ごとに特定方法をルール化して開発者の原因分析を支援する．また，ルールを定義する例外については，全てを網羅することはその種類の多さから困難であるため，構文エラーを除く実行時エラーを主として送出頻度の高い例外の調査を行い上位 5 つの例外について実装を行った．

3.2 提案手法の手順

提案手法では, 図 1 に示す次の手順によってエラーの原因の可能性がある式の候補を特定する．

前処理. 入力として受け取った Python ファイルの実行結果を取得

ソースコード 1: ソースコードの例

```
1 def total(n):
2     return sum([i for i in range(1, n + 1)])
3 x = "9"
4 print(total(x))
```

```
$ python3 total.py
Traceback (most recent call last):
  File "/path/to/total.py", line 4, in <module>
    print(total(x))
          ^^^^^^^^
  File "/path/to/total.py", line 2, in total
    return sum([i for i in range(1, n + 1)])
                                    ~~^~~
TypeError: can only concatenate str (not "int") to str
```

画面 1: 実行時に発生するエラーメッセージ[*1]

手順 1. メッセージ内の着目すべき変数の特定
エラーメッセージを解析し，例外種別とプログラム実行の中断箇所を取得する．それらとソースコードによって，エラー発生に関連する変数を特定する．

手順 2. 関連する箇所の特定
ソースコードに対してプログラムスライシングを実施することで，手順 1 で特定した変数に関連する箇所を特定する．

出力. 変数に関連する箇所を提示
手順 1，手順 2 で特定した変数の関連箇所を要修正箇所の候補としてハイライトして表示.

3.2.1 手順 1：メッセージ内の着目すべき変数の特定

実行時に発生するエラーが例外に示される文中の式で利用されている変数に起因する場合，前述のようにその変数の定義や利用する部分の記述に原因があるため，まず最初のステップとしてエラーメッセージの情報からその変数の特定を行う．最初に，エラーメッセージから解析を行ううえで必要となる各情報を抽出する．その後エラーメッセージから抽出した情報を利用して，ルールに則ってソースコードから例外発生に影響している変数を特定する．特定した変数のリストと定数のリストを生成し，変数のリストを手順 2 へ受け渡す.

3.2.2 手順 2：関連する箇所の特定

実際にエラーの原因となっている箇所（要修正箇所）は手順 1 で特定した変数の箇所に限らず，その変数の呼び出し元，定義式などが原因である可能性も高い．したがって，手順 2 では手順 1 で特定した変数に関連する箇所を特定する．手順 1 で変数群を特定した時，それらに対し変数

[*1] PEP 657 （https://peps.python.org/pep-0657/）準拠の Python3.11 以降での表記である.

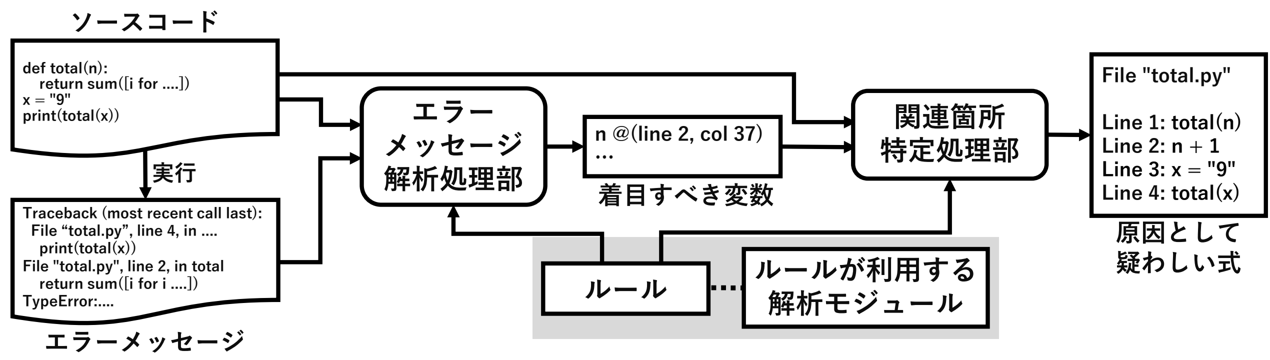

図 1: 提案手法の流れ

に関連する箇所をプログラムスライシングを用いて特定する．このプログラムスライシングの方法も例外の種類により異なるので，あらかじめ例外の種類ごとに用意したルールにしたがって実施する．

3.3 ルール

提案手法を実装するにあたって GitHub 上で Python のハッシュタグが付いているプロジェクトをスター順にソートし，上位 1000 個のプロジェクトについて例外処理を行っている箇所を例外の種類ごとに集計し，登場回数が多い例外をルールの実装対象とした．エラーメッセージ解析処理部は，手順 1 においてエラーメッセージから詳細の情報を抽出した後，マッチしたパターン（`if` 部）に対応する方法（`then` 部）でエラーの原因に関連する変数・定数を特定し，`output` 部で出力する．そして `params` 部で受け渡された特定した変数に対して，関連箇所特定処理部では `method` 部に記述する方法で変数に関連する箇所すなわちソースコード中の要修正箇所の特定を行う．

ソースコード 2: ルールの例

```
1  "type3":{
2    "variable":{
3      "if":"TypeError: can only concatenate \\w+
       \\(not \"\\w+\"\\) to \\w",
4      "then":"extract_left_token",
5      "output":[
6        "type3_var_list",
7        "type3_const_list"
8      ]
9    },
10   "related":{
11     "method":"extract_related_locations",
12     "params":[
13       "type3_var_list"
14     ]
15   }
16 }
```

3.4 実施例

ソースコード 1 で示すプログラムを実行した際に出力される画面 1 のエラーメッセージを利用して，提案手法によって原因箇所を特定すると画面 1 のような実行結果を得る．手順 1 で特定したソースコード上のエラー発生箇所（`n + 1`）と，それに関連する箇所（関数定義 `total(n)`，関数呼び出し `total(x)` および，引数に指定した変数 `x` への代入している箇所）をハイライトして表示している．

```
$ python3 ./errlocator path/to/total.py
Traceback (most recent call last):
  File "/path/to/total.py", line 4, in <module>
    print(total(x))
          ~~~~~~~~
  File "/path/to/total.py", line 2, in total
    return sum([i for i in range(1, n + 1)])
                                    ~~~~~
TypeError: can only concatenate str (not "int") to str
-------------------------------
Cause of error:
/path/to/total.py
1: def total(n):
2:     return sum([i for i in range(1, n + 1)])
3: x = "9"
4: print(total(x))
```

画面 1: 提案手法の実行結果

4 関連研究

実行時エラーの原因分析を行う先行研究としては，スタックトレースから疑わしい関数をランキングする手法 [1] や，スタックトレースの解析結果と Spectrum-Based Fault Localization の結果を組み合わせ，修正すべき箇所のランキングを生成する手法 [2] が挙げられる．後者は本研究での提案手法と同様に，誤りを含む可能性のあるステートメントを特定する手法だが，提案手法では事前に想定していない例外へも対応できるようにするため，その特定方法をルールとして外部定義できるようにしている．

5 おわりに

本研究では，実行時エラーの発生時に開発者に向けて原因分析の支援をすることを目的に，実行時に発生した例外に対して，エラーメッセージが示す情報に基づいてソースコードを解析し，プログラム上で原因となる可能性のある箇所を特定する手法を提案した．

参考文献

[1] R. Wu, H. Zhang, S. Cheung, and S. Kim. Crashlocator: Locating crashing faults based on crash stacks. *2014 International Symposium on Software Testing and Analysis*, pp. 204–214, 2014.

[2] D. Ginelli, O. Riganelli, D. Micucci, and L. Mariani. Exception-driven fault localization for automated program repair. *2021 IEEE 21st International Conference on Software Quality, Reliability and Security*, pp. 598–607, 2021.