

# Feedback-Directed Differential Testing of Interactive Debuggers

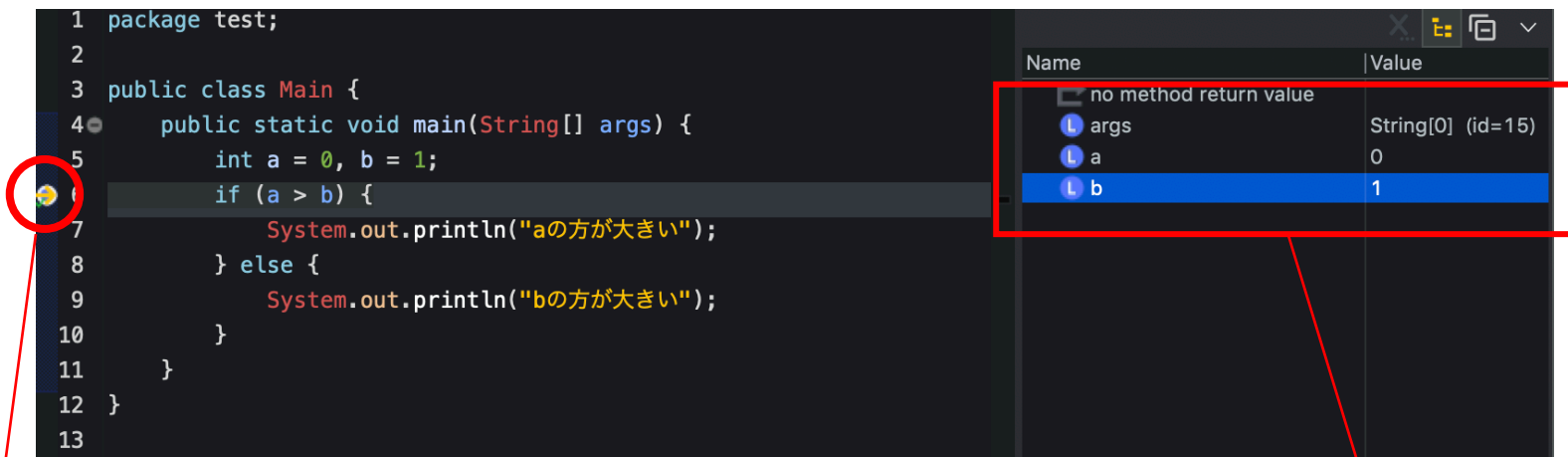
Daniel Lehmann   Michael Pradel

ESEC/FSE 2018 Proceedings of the 2018 26th ACM Joint  
Meeting on European Software Engineering Conference and  
Symposium on the Foundations of Software Engineering  
Pages 610-620

# 対話型デバッガ

- プログラムの欠陥の特定を対話的に行うデバッグツール

eclipseのデバッガを使用している様子



ブレークポイント：プログラムを  
その行で一時停止させる

一時停止した時点の各変数の値を  
見ることができる

# 対話型デバッガの欠陥

- 対話型デバッガに欠陥が含まれている場合、欠陥の特定が不可能になる

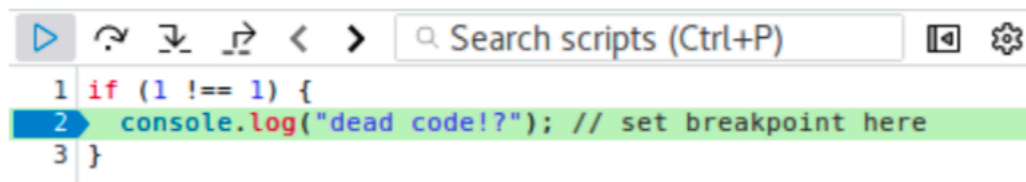


**Figure 1: JavaScript debugger of Firefox 54 incorrectly pauses at a breakpoint in dead code. Bug found with DBDB.**

2行目にブレークポイントを置いたところ1行目の条件式がFalseの場合でもFirefoxのデバッガが2行目のコードを実行してしまった

# 目的・アプローチ

- 目的
  - 対話型デバッガの欠陥を検出する手法の構築
- アプローチ
  - 2つのデバッガに対して同じデバッグ操作を行い、途中で振る舞いに差異が出るかどうかを確認める
  - 振る舞いの差異が実際に欠陥であるかどうかを確認める

# 提案手法の概要

- デバッガを有限状態トランスデューサに変換
- デバッガ間の振る舞いの差異の定義
- デバッガ自動操作アルゴリズムの構築
- アルゴリズムで得られた振る舞いの差異の組み分け

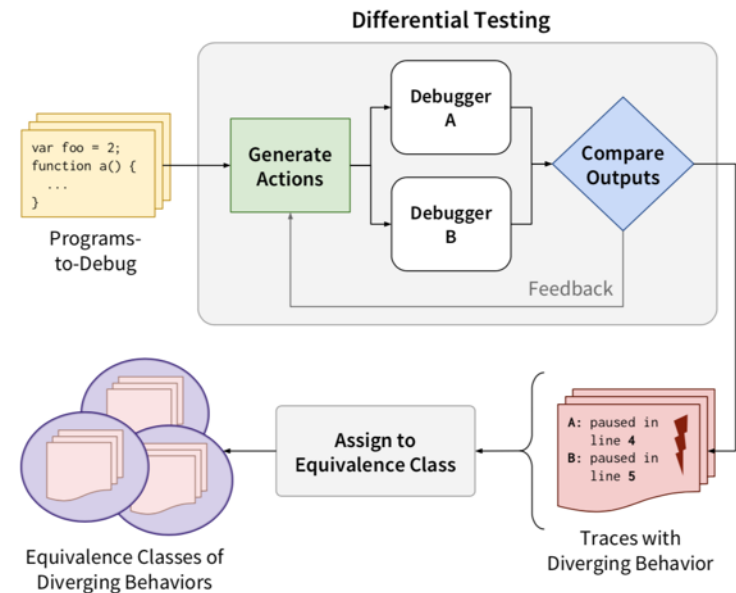
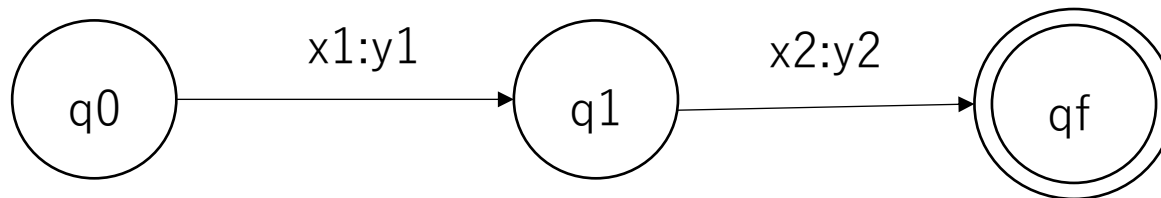


**Figure 2: DBDB overview: differential testing finds diverging behaviors, which are then assigned to equivalence classes.**

# 有限状態トランスデューサ

- 入力記号を与えると遷移だけでなく出力記号を出力する有限状態機械



# デバッガの有限状態トランスデューサ

- 定義： $(Q, q_0, \Sigma, \Gamma, \delta_P)$ 
  - $Q$ ：有限の状態集合
  - $q_0$ ：初期状態
  - $\Sigma$ ：入力シンボル（ユーザのデバッグ操作）
  - $\Gamma$ ：出力シンボル（デバッガの振る舞い）
  - $\delta_P$ ：遷移パターン,  $(q, x, y, r) \in \delta_P$

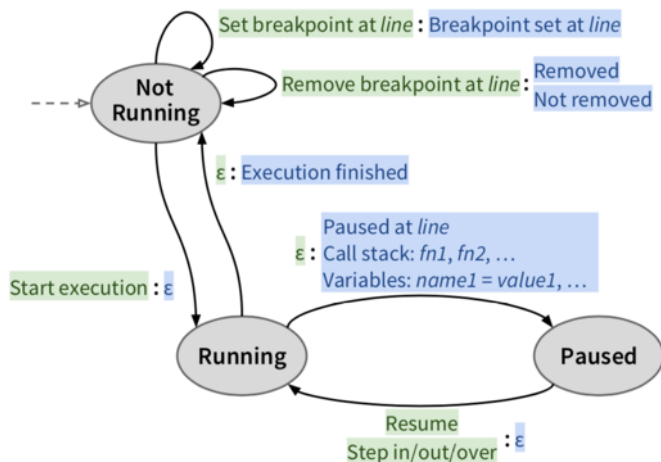


Figure 5: Interactive debuggers as finite state transducers. In transitions, “:” separates inputs (green) from outputs (blue).

$\Sigma ::= BpAction \mid ExecAction \mid \varepsilon$  (Actions)  
 $BpAction ::= Set\ breakpoint\ at\ line \mid Remove\ breakpoint\ at\ line$   
 $ExecAction ::= Start\ execution \mid Resume \mid Step\ in \mid Step\ out \mid Step\ over$   
 $\Gamma ::= BpOutput \mid ProgramState \mid Execution\ finished \mid \varepsilon$  (Outputs)  
 $BpOutput ::= Breakpoint\ set\ at\ line \mid Removed \mid Not\ removed$   
 $ProgramState ::= \langle line, callStack, vars \rangle$   
 $callStack ::= name^*$   
 $vars ::= (name : type = value?)^*$   
 $line \in \mathbb{N}$ , line numbers (Meta Variables)  
 $name \in identifiers$   
 $type \in types$   
 $value \in primitive\ values$

Figure 4: Grammars of debugging actions and outputs.

状態  $q$  から操作  $x$  によって状態  $r$  に遷移し、  
得られた振る舞いは  $y$

$$\begin{matrix} x:y \\ q \rightarrow r \end{matrix}$$

# 差異の定義

- 2つのデバッガ  $(Q, q_0, \Sigma, \Gamma, \delta_P)$ ,  $(\tilde{Q}, \tilde{q}_0, \tilde{\Sigma}, \tilde{\Gamma}, \tilde{\delta}_P)$  が次の状態遷移を示すとき,

$$q_0 \xrightarrow{x:y} \dots \xrightarrow{x':y'} r \quad \text{and} \quad \tilde{q}_0 \xrightarrow{\tilde{x}:\tilde{y}} \dots \xrightarrow{\tilde{x}':\tilde{y}'} \tilde{r}$$

$q_0 = \tilde{q}_0$ ,  $x = \tilde{x}$ ,  $y = \tilde{y}$  を満たすが,  $y' \neq \tilde{y}'$  または  $r \neq \tilde{r}$  を満たす場合, 振る舞いの差異が発生したとする

# 自動操作アルゴリズム

入力：2つのデバッガと対象プログラム  
出力：2つのデバッガのトレース

トレースの定義：デバッガへの操作と振る舞いをまとめたもの

プログラムの再実行を繰り返す  
振る舞いの差異：

- デバッガの状態が異なる
- プログラムの状態が異なる

## Algorithm 1 Interactive differential analysis

**Input:** Debugger  $D_A, D_B$  and program  $P$

**Output:** Traces  $t_A, t_B$  of actions and outputs

```
1: Assume:  $D_A.state = D_B.state = not\ running$   
2:  $BPs \leftarrow \emptyset$  ▷ Manipulate breakpoints:  
3: while  $|BPs| < \text{max number of breakpoints}$  do  
4:    $l \leftarrow \text{randLine}(P)$   
5:   if  $l \in BPs$  then  
6:     continue  
7:    $l_A \leftarrow D_A.setBp(l); l_B \leftarrow D_B.setBp(l)$   
8:   if  $l_A \neq l_B$  then  
9:     return "Diverging behavior: Bp location"  
9:   if  $\text{randProb}() < \text{prob. of removing breakpoint}$  then  
10:     $ok_A \leftarrow D_A.rmBp(l_1); ok_B \leftarrow D_B.rmBp(l_2)$   
11:    if  $ok_A \neq ok_B$  then  
12:      return "Diverging behavior: Bp removal"  
12:  else  
13:     $BPs \leftarrow BPs \cup \{l_A, l_B\}$   
14:  $D_A.startExec(); D_B.startExec()$  ▷ Start program execution:  
15: repeat ▷ Step and resume:  
16:   Wait until  $D_A.state \neq running$  and  $D_B.state \neq running$   
17:   if  $D_A.state = D_B.state = not\ running$  then  
18:     return "Program finished"  
18:   if  $D_A.state \neq D_B.state$  then  
19:     return "Diverging behavior: Termination"  
19:   if  $D_A.programState \neq D_B.programState$  then  
20:     return "Diverging behavior: Program state"  
20:    $action \leftarrow \text{randPick}(\{resume, stepIn, stepOut, stepOver\})$   
21:    $D_A.action(); D_B.action()$   
22: until max number of execution actions
```

ブレークポイントの配置と削除の操作を行う  
振る舞いの差異：

- ブレークポイントの位置が異なる
- ブレークポイントの削除ができない

プログラムの実行

# 差異の組み分け

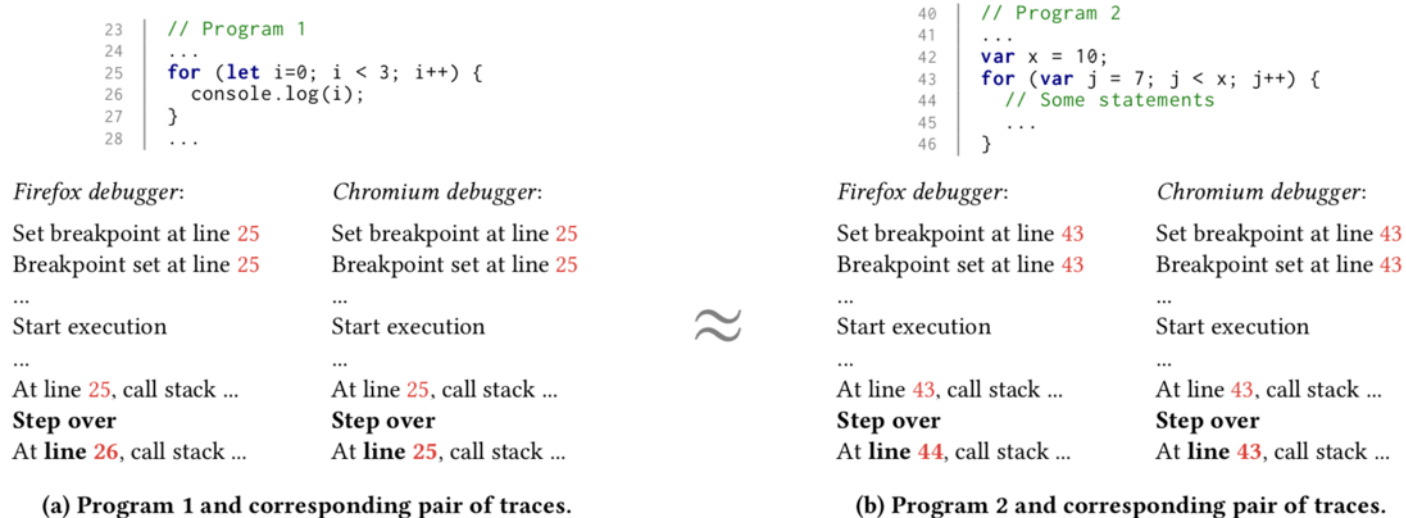


Figure 6: Example of equivalent diverging behaviors exposed by two different programs.

- 膨大な件数のトレース結果から欠陥らしいものを見つけやすくするために同じ流れのトレースの組み分けをする

# 評価実験

- データセット
  - デバッガ
    - Firefox 54.0とChromium 59.0.3071.109
  - ソースコード
    - JavaScriptで記述されたソースコード41
- 実験項目
  - 振る舞いの差異の検出の有効性
  - 組み分けの性能
  - パフォーマンス

# 振る舞いの差異の検出の有効性

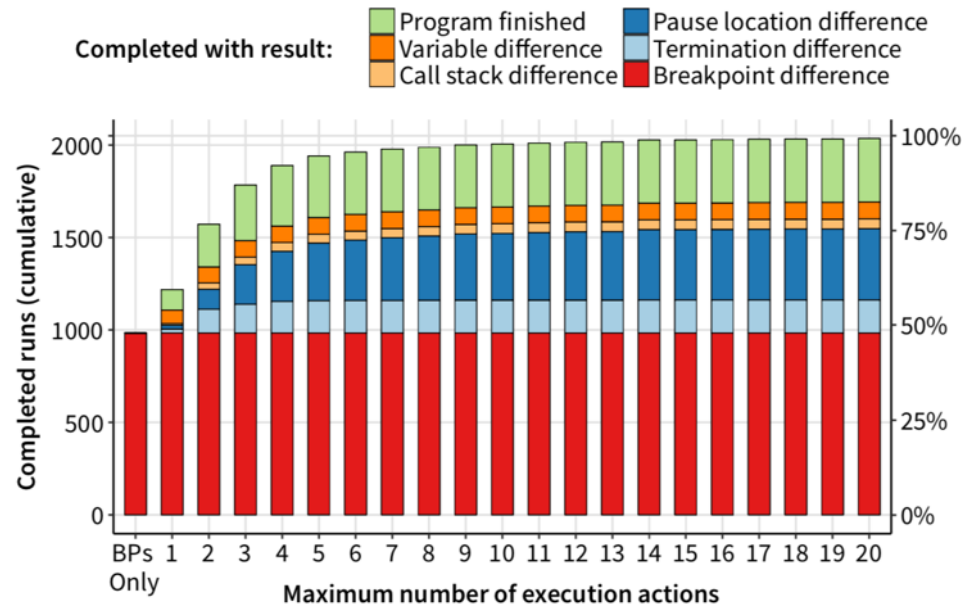


Figure 7: Completed test runs (and their results) per maximum number of generated execution actions.

- 20回の操作で1692件の差異を検出

# 組み分けの有効性

- 1162件の差異を112種類のクラスに組み分けた
- 右の上の図はサイズが大きいクラスの上位6位を示している
- 右の下図は欠陥の特定がクラス分けするかしないかでどちらが多いかを示している

Table 2: The top 6 equivalence classes of diverging behavior.

| Diff. Type     | Last Action    | AST Node   | Size | in %  |
|----------------|----------------|------------|------|-------|
| Set breakpoint | Set breakpoint | Program    | 538  | 31.8% |
| Set breakpoint | Set breakpoint | ArrayExpr  | 72   | 4.3%  |
| Set breakpoint | Set breakpoint | BinaryExpr | 64   | 3.8%  |
| Paused line    | Step in        | ForStmt    | 61   | 3.6%  |
| Set breakpoint | Set breakpoint | Property   | 55   | 3.3%  |
| Paused line    | Step over      | ForStmt    | 51   | 3.0%  |

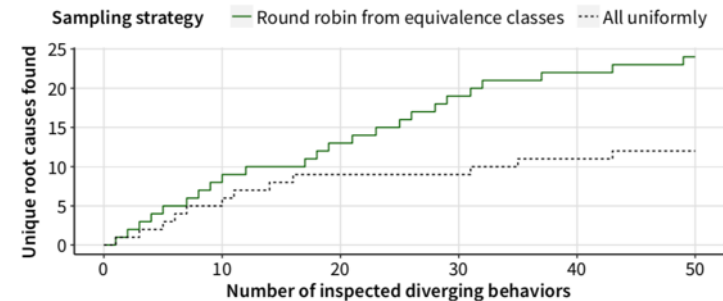


Figure 9: Found unique root causes when sampling difference equivalence classes compared to raw differences.

# パフォーマンス

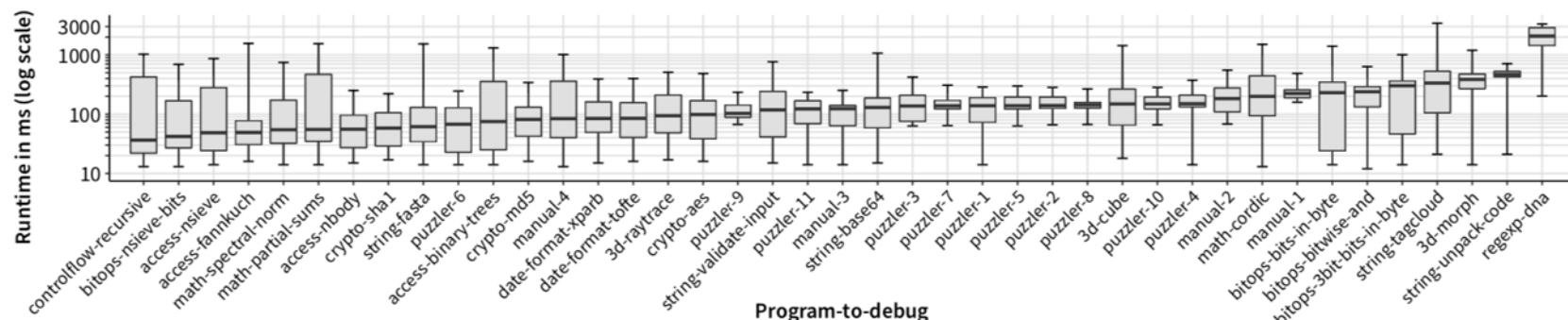


Figure 8: Runtime of the differential testing per program-to-debug. Since actions are randomly generated, medians (middle band) and first and third quartile (bottom and top of box) are shown after 50 runs. Whiskers indicate minimum and maximum.

- 41種のソースコードに対して50回提案手法を実行し，実行時間を計測して箱ひげ図として示した
- 平均時間は448ms

# まとめ

- 対話型デバッガの欠陥を検出する手法を以下のアプローチにより構築
  - デバッガを有限状態トランスデューサに変換
  - 振る舞いの差異の定義
  - 自動操作アルゴリズムの構築
  - 振る舞いの差異の組み分け
- 評価実験により以下の点で有効であることが明らかになった
  - 20回の操作で多くの差異を検出
  - 差異の組み分けにより欠陥の特定が容易になった
  - 実行速度は平均で448ms

# 所感

- デバッガのユーザが欠陥を特定できない事態を防ぐことができるという点は大きな貢献である
- アルゴリズムの実行時間がかなり速いため、実用性も大きいと思った
- 他のデバッガを使う場合に手間がかかりそう