머신러닝을 통한 소프트웨어 테스팅의 탐색전략 자동생성 기술

(Automatically Generating Search Heuristics of Software Testing with Machine Learning)

Sooyoung Cha
Korea University

18 October 2018 @Samsung R&D Campus
I am Sooyoung Cha

- **Research Areas:** software engineering.
  - Data-driven software testing.
  - **Publications:** top-venues in software engineering.
    - ICSE’18, ASE’18, FSE’19

- **Enjoy presenting a talk.**
  - Apr. 2018. Oxford Univ @UK
  - Jun. 2018. ICSE’18 @Sweden
  - Sep. 2018. ASE’18 @France
Software Analysis Lab (SAL)

• Goal: **Automated** programming / testing / debugging

```
Spec          SW (buggy)          Failures          SW (safe)
```

- Programming (manual)
- Testing (manual)
- Debugging (manual)

- **Program Synthesis**
  - **PAT** (IJCAI’18)
  - **FixML** (OOPSLA’18)

- **Program Analysis**
  - **ParaDySE** (ICSE’18)
  - **ConTest** (ASE’18)

- **Program Repair**
  - **MemFix** (FSE’18)

[http://prl.korea.ac.kr](http://prl.korea.ac.kr)
Today’s Talk

• Concolic Testing (Dynamic Symbolic Execution)
  - An effective software testing method.
  - Concrete and Symbolic executions.

• Open-Source Tool:
  - A state-of-the-art technique.
  - Parametric Dynamic Symbolic Execution.
Today’s Talk

• Concolic Testing (Dynamic Symbolic Execution)
  - An effective software testing method.
  - Concrete and Symbolic executions.

• Open-Source Tool: ParaDySE
  - A state-of-the-art technique.
  - Parametric Dynamic Symbolic Execution.
  (https://github.com/kupl/ParaDySE)
IoTcube

- Web service implementation (Users: 10,851)
  - Discovered **security vulnerabilities** in IoT devices.
  - Implemented our technique on **ctest** in IoTcube.

Center for Software Security and Assurance (CSSA)

(https://iotcube.korea.ac.kr)
Concolic Testing
Software Testing

- **Motivation: Unsafe Software**
  - Software failures everywhere

Concolic Testing

• An effective software testing method.
  – Combine concrete and symbolic executions.
  – Enhance random testing.

• Goal
  – Improve code coverage in a limited budget.
  – Find software vulnerability.
  • SAGE: Find 30% of all Windows 7 security bugs.
Limitation of Random Testing

- Probability of the error?
  \((1 \leq x, y \leq 100)\)

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if (z == x) {
        if (x > y + 10) {
            error;
        }
    }
}
```
Limitation of Random Testing

- Probability of the error?
  \( (1 \leq x, y \leq 100) \)
  
  0.4 %

- Random testing requires 250 runs.
- Concolic testing finds it in 3 runs.

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```
Concolic Testing

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```

**Concrete State**
- $x = 22, y = 7$

**Symbolic State**
- $x = \alpha, y = \beta$

**PathCond (PC): true**

**Initial Random Input**
- $x=22, y=7$

1st iteration
Concolic Testing

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```

Concrete State

Symbolic State

1st iteration
Concolic Testing

int twice (int v) {
return 2 × v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
Concolic Testing

\begin{itemize}
\item \textbf{Concrete State:} \texttt{x = 22, y = 7, z = 14}
\item \textbf{Symbolic State:} \text{Solve: } 2\beta = \alpha \\
\text{Solution: } \alpha = 2, \beta = 1
\end{itemize}

\begin{verbatim}
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
\end{verbatim}

1st iteration
Concolic Testing

int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}

Concrete State

x = 2, y = 1

Symbolic State

x = α, y = β

PC: true

2nd iteration
Concolic Testing

```
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```

2nd iteration

Concrete State

Symbolic State

\[ x = 2, y = 1, z = 2 \]

\[ x = \alpha, y = \beta, z = 2\beta \]

PC: true
Concolic Testing

int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}

Concrete State

Symbolic State

\( x = 2, y = 1, z = 2 \)

\( x = \alpha, y = \beta, z = 2\beta \)

PC: \( 2\beta = \alpha \)

2nd iteration
Concolic Testing

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z  = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```

Concrete State

Symbolic State

2nd iteration

x = 2, y = 1, z=2

x = α, y = β, z = 2*β

PC: (2*β = α) \land (α ≤ β+10)
Concolic Testing

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```

Concrete State

Symbolic State

Solve: $(2*\beta = \alpha) \land (\alpha > \beta+10)$

Solution: $\alpha=22$, $\beta=11$

$x = 2$, $y = 1$, $z=2$

$PC: (2*\beta = \alpha) \land (\alpha \leq \beta+10)$

2nd iteration
Concolic Testing

int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}

Concrete State

Symbolic State

3rd iteration
Concolic Testing

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
```

Concrete State: $x = 22, y = 11, z = 22$

Symbolic State: $x = \alpha, y = \beta, z = 2\beta$

3rd iteration

PC: true
int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y+10 ) {
            error;
        }
    }
}
Concolic Testing

```c
int twice (int v) {
    return 2 * v;
}

void main (int x, int y) {
    z = twice (y);
    if ( z == x ) {
        if ( x > y + 10 ) {
            error;
        }
    }
}
```

Concrete State: $x = 22, y = 11, z = 22$

Symbolic State: $x = \alpha, y = \beta, z = 2\beta$

PC: $(2\beta = \alpha) \land (\alpha > \beta + 10)$

3rd iteration

error-triggering input: $x = 22, y = 11, z = 22$
Challenge

• Path Explosion
  - # of execution paths: \(2\)
  - ex) grep-2.2(3,836) : \(2^{3,836}\) paths (worst case)
  - Exploring all paths is impossible.
Diverse Solutions

• Goal: Mitigate the path-explosion.
Search Heuristic

• Select branches that are likely to maximize code coverage.
• Numerous search heuristics have been proposed.
  – DFS, BFS, Random, Generational, CFDS, CGS, etc
Search Heuristic

• Select branches that **are likely to maximize** code coverage.

• **Numerous search heuristics** have been proposed.
  
  – DFS, BFS, Random, Generational, CFDS, CGS, etc
Search Heuristic

• Select branches that are likely to maximize code coverage.

• Numerous search heuristics have been proposed.
  – DFS, BFS, Random, Generational, CFDS, CGS, etc
Open-Source Tool: ParaDySE
Motivation

- No existing heuristics consistently achieve high coverage.
- Designing new heuristic is highly nontrivial.
  - Search Heuristic → 🧠 (ICSE, FSE, ASE, NDSS, ...)

![Graph showing coverage over iterations for vim-5.7 and expat-2.1.0](Image)
Goal

• **Automatically Generating** Search Heuristics

    - **Parameterized** Search Heuristic.
    - Effective **Parameter Search** Algorithm.

Key ideas 🧠

Input: C programs (e.g., vim, expat)

Output: Search Heuristics.
Effectiveness

- **Considerable increase** in branch coverage.

  - gawk-3.0.3: ./gawk `'\Vo\n^000000000000070000000'` file
  - grep-2.2: ./grep `'\(\)\(\)\+***` file
    - Trigger the error in grep-3.1 (the latest version)
Effectiveness

- **Considerable increase** in branch coverage.

- **Found real-world performance bugs.**
  - `gawk-3.0.3`: `./gawk 'V0\n^000000000000070000000'` file
  - `grep-2.2`: `./grep '\(\)\1\+**'` file
    - Trigger the error in grep-3.1 (the latest version)
In academia

- Make the tool **publicly available.** (Recently)

  ![Image of ParadysE tool]

- **A good opportunity** to use the latest techniques.
I. Parameterized Search Heuristic
Existing Search Heuristics

Software 1

Software 2

CFDS

CGS

DFS
Parameterized Search Heuristic
Parameterized Search Heuristic

Software 1
- CFDS
- CGS
- Param 1: b, b
- Param 2: b, b
- DFS

Software 2
- CFDS
- CGS
- Param 1: b
- Param 2: b
- DFS
Parameterized Search Heuristic

• Search Heuristic$_\theta$: Path $\rightarrow$ Branch

  − Generating a “good search heuristic” $\rightarrow$ Finding a “good parameter $\theta$”

  \[
  \begin{align*}
  \text{score}_\theta(B1) &= 0.1 \\
  \text{score}_\theta(B2) &= 0.7 \\
  \text{score}_\theta(B3) &= -0.5
  \end{align*}
  \]
Parameterized Search Heuristic

- Search Heuristic$\theta$: Path $\rightarrow$ Branch

  - Generating a “good search heuristic” $\rightarrow$ Finding a “good parameter $\theta$”

  $\text{score}_{\theta_2}(B1) = 0.3$
  $\text{score}_{\theta_2}(B2) = -0.2$
  $\text{score}_{\theta_2}(B3) = 0.9$
**Parameterized Search Heuristic**

(1). Represent branches as feature vectors

- A feature: a boolean predicate on branches.
  - ex1) the branch in **main function**?
  - ex2) true branch of a **case statement**?

```
B1 = ⟨1, 0, 1, 1, 0⟩
B2 = ⟨0, 1, 1, 1, 0⟩
B3 = ⟨1, 0, 0, 0, 1⟩
```
Parameterized Search Heuristic

- Design 40 features.
  - 12 static features
  - extracted without execution.
    (e.g., true branch of a loop)
  - 28 dynamic features
    - Extracted at runtime.
      (e.g., branch newly covered in the previous execution)

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>branch in the main function</td>
</tr>
<tr>
<td>2</td>
<td>true branch of a loop</td>
</tr>
<tr>
<td>3</td>
<td>false branch of a loop</td>
</tr>
<tr>
<td>4</td>
<td>nested branch</td>
</tr>
<tr>
<td>5</td>
<td>branch containing external function calls</td>
</tr>
<tr>
<td>6</td>
<td>branch containing integer expressions</td>
</tr>
<tr>
<td>7</td>
<td>branch containing constant strings</td>
</tr>
<tr>
<td>8</td>
<td>branch containing pointer expressions</td>
</tr>
<tr>
<td>9</td>
<td>branch containing local variables</td>
</tr>
<tr>
<td>10</td>
<td>branch inside a loop body</td>
</tr>
<tr>
<td>11</td>
<td>true branch of a case statement</td>
</tr>
<tr>
<td>12</td>
<td>false branch of a case statement</td>
</tr>
<tr>
<td>13</td>
<td>first 10% branches of a path</td>
</tr>
<tr>
<td>14</td>
<td>last 10% branches of a path</td>
</tr>
<tr>
<td>15</td>
<td>branch appearing most frequently in a path</td>
</tr>
<tr>
<td>16</td>
<td>branch appearing least frequently in a path</td>
</tr>
<tr>
<td>17</td>
<td>branch newly covered in the previous execution</td>
</tr>
<tr>
<td>18</td>
<td>branch located right after the just-negated branch</td>
</tr>
<tr>
<td>19</td>
<td>branch whose context (k = 1) is already visited</td>
</tr>
<tr>
<td>20</td>
<td>branch whose context (k = 2) is already visited</td>
</tr>
<tr>
<td>21</td>
<td>branch whose context (k = 3) is already visited</td>
</tr>
<tr>
<td>22</td>
<td>branch whose context (k = 4) is already visited</td>
</tr>
<tr>
<td>23</td>
<td>branch whose context (k = 5) is already visited</td>
</tr>
<tr>
<td>24</td>
<td>branch negated more than 10 times</td>
</tr>
<tr>
<td>25</td>
<td>branch negated more than 20 times</td>
</tr>
<tr>
<td>26</td>
<td>branch negated more than 30 times</td>
</tr>
<tr>
<td>27</td>
<td>branch near the just-negated branch</td>
</tr>
<tr>
<td>28</td>
<td>branch failed to be negated more than 10 times</td>
</tr>
<tr>
<td>29</td>
<td>the opposite branch failed to be negated more than 10 times</td>
</tr>
<tr>
<td>30</td>
<td>the opposite branch is uncovered (depth 0)</td>
</tr>
<tr>
<td>31</td>
<td>the opposite branch is uncovered (depth 1)</td>
</tr>
<tr>
<td>32</td>
<td>branch negated in the last 10 executions</td>
</tr>
<tr>
<td>33</td>
<td>branch negated in the last 20 executions</td>
</tr>
<tr>
<td>34</td>
<td>branch negated in the last 30 executions</td>
</tr>
<tr>
<td>35</td>
<td>branch in the function that has the largest number of uncovered branches</td>
</tr>
<tr>
<td>36</td>
<td>the opposite branch belongs to unreach functions (top 10% of the largest func.)</td>
</tr>
<tr>
<td>37</td>
<td>the opposite branch belongs to unreach functions (top 20% of the largest func.)</td>
</tr>
<tr>
<td>38</td>
<td>the opposite branch belongs to unreach functions (top 30% of the largest func.)</td>
</tr>
<tr>
<td>39</td>
<td>the opposite branch belongs to unreach functions (# of branches &gt; 10)</td>
</tr>
<tr>
<td>40</td>
<td>branch inside the most recently reached function</td>
</tr>
</tbody>
</table>
Parameterized Search Heuristic

(2). Scoring

- The parameter: a k-dimension vector.

\[ \theta = \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle \]

- Linear combination of feature vector and parameter

  - \( \text{Score}_\theta(B1) = \langle 1, 0, 1, 1, 0 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = 0.1 \)
  - \( \text{Score}_\theta(B2) = \langle 0, 1, 1, 1, 0 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = 0.7 \)
  - \( \text{Score}_\theta(B3) = \langle 1, 0, 0, 0, 1 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = -0.5 \)

(3). Choosing the branch with the highest score

- B2
2. Parameter Search Algorithm
Parameter Search Algorithm

• Finding good parameters is crucial.

• Naive algorithm based on random sampling.

\[ \theta_1 = \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle \rightarrow \text{Coverage}(519) \]

\[ \theta_2 = \langle -0.9, 0.5, 0.9, -0.2, 1.0 \rangle \rightarrow \text{Coverage}(423) \]

\[ \ldots \]

\[ \theta_n = \langle 0.7, -0.2, -0.9, -0.9, 0.3 \rangle \rightarrow \text{Coverage}(782) \]

- Failed to find good parameters.
  • Search space is intractably large.
  • Performance variation in concolic testing.
Parameter Search Algorithm

- **Iteratively refine** the sample space based on the feedback from previous runs of concolic testing
Experiments
Experiments

- Implemented in CREST
- Compared with five existing heuristics
  - CGS, CFDS, Random, Generational, DFS
- Used 10 open-source C programs

<table>
<thead>
<tr>
<th>Program</th>
<th># Total branches</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>vim-5.7</td>
<td>35,464</td>
<td>165K</td>
</tr>
<tr>
<td>gawk-3.0.3</td>
<td>8,038</td>
<td>30K</td>
</tr>
<tr>
<td>expat-2.1.0</td>
<td>8,500</td>
<td>49K</td>
</tr>
<tr>
<td>grep-2.2</td>
<td>3,836</td>
<td>15K</td>
</tr>
<tr>
<td>sed-1.17</td>
<td>2,656</td>
<td>9K</td>
</tr>
<tr>
<td>tree-1.6.0</td>
<td>1,438</td>
<td>4K</td>
</tr>
<tr>
<td>cdaudio</td>
<td>358</td>
<td>3K</td>
</tr>
<tr>
<td>floppy</td>
<td>268</td>
<td>2K</td>
</tr>
<tr>
<td>kbfiltr</td>
<td>204</td>
<td>1K</td>
</tr>
<tr>
<td>replace</td>
<td>196</td>
<td>0.5K</td>
</tr>
</tbody>
</table>
Evaluation Setting

- The same initial inputs
- The same testing budget (4,000 executions)
- Average branch coverage for 100 trials (50 for vim)
  - 1 trial = 4,000 executions
Effectiveness

- Average branch coverage (6 Smiles 😊)
Effectiveness

- **Reusable over multiple subsequent programs**

  ![Graphs showing average coverage over versions](image)

  - (4 Years)
  - (A Year)

- **Time for obtaining the heuristics (with 20 cores)**
  - vim-5.7(24 h), expat-2.1.0(10h), grep-2.2(5h), tree-1.6.0(3h)
## Important Features

- No winning feature always belongs to top 10 features.
- Depending the program, the role of feature changes. (feature 10)

```
# Top 10 positive features

<table>
<thead>
<tr>
<th>Rank</th>
<th>vim</th>
<th>gawk</th>
<th>expat</th>
<th>grep</th>
<th>sed</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#15</td>
<td>#10</td>
<td>#27</td>
<td>#14</td>
<td>#13</td>
<td>#36</td>
</tr>
<tr>
<td>2</td>
<td>#18</td>
<td>#13</td>
<td>#30</td>
<td>#40</td>
<td>#2</td>
<td>#15</td>
</tr>
<tr>
<td>3</td>
<td>#35</td>
<td>#12</td>
<td>#23</td>
<td>#24</td>
<td>#29</td>
<td>#5</td>
</tr>
<tr>
<td>4</td>
<td>#40</td>
<td>#38</td>
<td>#31</td>
<td>#1</td>
<td>#3</td>
<td>#25</td>
</tr>
<tr>
<td>5</td>
<td>#31</td>
<td>#14</td>
<td>#4</td>
<td>#30</td>
<td>#8</td>
<td>#40</td>
</tr>
<tr>
<td>6</td>
<td>#7</td>
<td>#9</td>
<td>#9</td>
<td>#38</td>
<td>#30</td>
<td>#9</td>
</tr>
<tr>
<td>7</td>
<td>#13</td>
<td>#35</td>
<td>#8</td>
<td>#32</td>
<td>#35</td>
<td>#13</td>
</tr>
<tr>
<td>8</td>
<td>#3</td>
<td>#31</td>
<td>#15</td>
<td>#17</td>
<td>#6</td>
<td>#39</td>
</tr>
<tr>
<td>9</td>
<td>#12</td>
<td>#4</td>
<td>#25</td>
<td>#31</td>
<td>#21</td>
<td>#30</td>
</tr>
<tr>
<td>10</td>
<td>#10</td>
<td>#33</td>
<td>#7</td>
<td>#29</td>
<td>#16</td>
<td>#22</td>
</tr>
</tbody>
</table>

# Top 10 negative features

<table>
<thead>
<tr>
<th>Rank</th>
<th>vim</th>
<th>gawk</th>
<th>expat</th>
<th>grep</th>
<th>sed</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#17</td>
<td>#26</td>
<td>#39</td>
<td>#20</td>
<td>#11</td>
<td>#10</td>
</tr>
<tr>
<td>2</td>
<td>#11</td>
<td>#8</td>
<td>#35</td>
<td>#39</td>
<td>#32</td>
<td>#35</td>
</tr>
<tr>
<td>3</td>
<td>#34</td>
<td>#16</td>
<td>#33</td>
<td>#22</td>
<td>#19</td>
<td>#6</td>
</tr>
<tr>
<td>4</td>
<td>#33</td>
<td>#29</td>
<td>#37</td>
<td>#25</td>
<td>#40</td>
<td>#24</td>
</tr>
<tr>
<td>5</td>
<td>#22</td>
<td>#3</td>
<td>#38</td>
<td>#26</td>
<td>#38</td>
<td>#7</td>
</tr>
<tr>
<td>6</td>
<td>#21</td>
<td>#6</td>
<td>#2</td>
<td>#19</td>
<td>#18</td>
<td>#12</td>
</tr>
<tr>
<td>7</td>
<td>#26</td>
<td>#22</td>
<td>#24</td>
<td>#27</td>
<td>#5</td>
<td>#23</td>
</tr>
<tr>
<td>8</td>
<td>#25</td>
<td>#11</td>
<td>#22</td>
<td>#21</td>
<td>#20</td>
<td>#2</td>
</tr>
<tr>
<td>9</td>
<td>#37</td>
<td>#19</td>
<td>#10</td>
<td>#33</td>
<td>#34</td>
<td>#27</td>
</tr>
<tr>
<td>10</td>
<td>#20</td>
<td>#28</td>
<td>#32</td>
<td>#37</td>
<td>#26</td>
<td>#11</td>
</tr>
</tbody>
</table>
```

Search Heuristic should be adaptively tuned for each program.
Take away

- **Q1. 'Concolic Testing'** is what kind of technology?
  - SW error detection is a testing technique.

- **Q2. 'ParaDySE'** is what kind of tool?
  - It is an open-source tool that improves SW testing performance through machine learning.

- **Q3. What do you want to say in the end?**
  - Robot > Person