PySE: Automatic Worst-Case Test Generation by Reinforcement Learning

Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi
Electrical and Computer Engineering
Purdue University
{kooj, cgusthin, milind, sbagchi}@purdue.edu
Stress testing

• Stress testing
  o Testing the software beyond its normal operational capacity, and investigates the behavior of a program when subjected to heavy loads.

• The goal of such tests
  o To identify performance bottlenecks
  o To identify algorithmic complexity attacks
  o To identify scale-dependent bugs

• The key challenge
  o How to find the input that can lead to the worst-case complexity.
Symbolic execution

- Runs a program using symbolic variables as inputs, instead of concrete values.
- Can explore all the possible execution paths, including the ones of worst-case complexity.
- On each path that is executed, symbolic execution collects a set of symbolic conditions, called a path condition.
- Then, it invokes a constraint solver, such as OpenSMT [7] or Z3 that generates concrete test input values.
- Path explosion: the number of paths to search increase exponentially with the size of the input.
WISE-like algorithms

- WISE[1] and SPF-WCA[2]
  - Learn a branching policy that results in a path of the worst-case complexity for small input sizes by using exhaustive search, and
  - Then apply the learned branching policy to perform a guided search for a large input size.

The worst-case branching policy

Insertion sort: always True

Limitations of WISE-like algorithms

• Assumes a continuous program behavior across scales
  o Some conditional blocks are activated only when the input size is larger than a certain threshold.

• Irregular branching policy

Dijkstra implemented with min-priority queue: no simple way to describe the worst-case branching policy
Limitations of WISE-like algorithms

- Assumes a continuous program behavior across scales
  - Some conditional blocks are activated only when the input size is larger than a certain threshold.

- Irregular branching policy

Can we avoid/minimize these issues of white-box based approaches for large-scale test generation?
PySE Solution approach

• PySE: learns the worst-case branching policy using Q-learning, a model-free reinforcement learning.
  o Uses symbolic execution to collect behavioral information of a given branching policy
  o Updates the policy based on Q-learning.

Observe the behavior of a branching policy, i.e., the length of a path

Symbolic execution → Policy update

Give a new branching policy
The main objective of PySE

- To find out a branching policy $\pi(s_t)$ for a given state $s_t$ at the $t$-th branch condition that it encounters while a program is being symbolically executed.
  - The branching policy $\pi(s_t)$ determines a branching decision $a_t = \pi(s_t) \in \{True, False\}$, which we also call action.
  - The state $s_t$ mainly consists of the current branch condition, previous $L$ branch conditions, and actions taken there.
    - $L$: the history length
  - The branching policy $\pi(s_t)$ continues evolving in such a way that the length of an execution path increases.
Workflow of PySE

• Step 1: (SYMBOLIC EXECUTION)
  o Execute a program by the branching policy \( \pi(s_t) \).
  o Collect resulting behavioral information such as which branch points the program visits, actions taken at each branch, and feasibilities of the actions.

• Step 2: (POLICY UPDATE)
  o Update the branching policy \( \pi(s_t) \) in a way that an undesirable action that caused a program to terminate quickly can be avoided in the future.
    ✓ Q-learning
Branching policy $\pi(s_t)$

• Design the branching policy $\pi(s_t)$ as:

$$\pi(s_t) = \arg \max_{a_t} Q(s_t, a_t)$$

• $Q(s_t, a_t)$ is made from an artificial neural network (ANN), whose inputs are $s_t$ and its output layer produces two values, $Q(s_t, True)$ and $Q(s_t, False)$.
  o $\pi(s_t)=True$ if $Q(s_t, True) \geq Q(s_t, False)$.
  o $\pi(s_t)=False$ if $Q(s_t, True) < Q(s_t, False)$. 
State representation

- \( s_t = (s_{t0}, s_{t1}, \ldots, s_{tL}) \)
  - \( s_{tl} \): an integer vector encoding the \((t - l)\)-th branch condition and the action taken there.

- Encoding of a state when \( L = 2 \).
  - F2: unique identifier for each branch point (e.g. line number)
  - F3: action taken at the branch point (1 = TRUE, 0 = FALSE)
How to update the branching policy (1/3)

• Symbolic execution takes action $a_t$ at a given state $s_t$ and observes its consequence.
  o Whether the execution path is still feasible.
  o Feasibility can be checked by using a constraint solver like Z3.

• Depending on the feasibility, the consequence of the action $a_t$ at the state $s_t$ is scored by a reward $r_t$:
  o $r_t = 1$ if feasible, and $r_t = P$ if not feasible.
  o $P = -20$ so that the infeasible decision is more distinguishable from the feasible one.
How to update the branching policy (2/3)

- We want \( \pi(s_t) \) to converge to the optimal branching policy \( \pi^*(s_t) \) that maximizes the expected sum of future rewards, \( E\left(\sum_{k=t}^{T} r_k \mid s_t\right) \).
  - \( T \) denotes the last branch condition before a program terminates normally or falls in an infeasible path condition.
  - Thus, equivalently, it maximizes the length of a feasible execution path.

- Define the optimal action-value function \( Q^*(s_t, a_t) \) as the maximum expected sum of future rewards, after taking action \( a_t \) at a state \( s_t \):
  \[
  Q^*(s_t, a_t) = \max_{\pi} E\left(\sum_{k=t}^{T} r_k \mid s_t\right)
  \]

- \( Q^*(s_t, a_t) \) can be re-written recursively as:
  \[
  Q^*(s_t, a_t) = E(r_t + \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}))
  \]
How to update the branching policy (3/3)

• We try to learn $Q^*(s_t, a_t)$ by a sample mean $Q(s_t, a_t)$:
  $$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}))$$

  - $\alpha$ is called a learning rate.
  - By the law of large numbers, $Q(s_t, a_t)$ can converge to $Q^*(s_t, a_t)$ after iterations for a sufficiently small value of $\alpha$.
  - Such an update for learning $Q(s_t, a_t)$ without knowing the underlying probability distribution model is referred to as Q-learning in the reinforcement learning literature.
Q-network architecture

• In practice, updating $Q(s_t, a_t)$ separately for each $(s_t, a_t)$ is unattainable.
  
  o This is because the state is a multi-dimensional integer vector and thus the number of possible states can be too large.

• Thus, a function approximator is commonly used to estimate the function $Q(s_t, a_t)$ with the limited number of observations for state-action pairs.

• PySE also represents $Q(s_t, a_t)$ by using an ANN-based function approximator, which we refer to as a Q-network.
Algorithm of PySE

Algorithm 1 Basic mode of PySE

1: procedure SYMBOLIC EXECUTION
2: for $t$ from 1 to $T$ do
3:   Choose a number $u$ randomly over $[0, 1]$.
4:   if $u < \epsilon$ then
5:     Choose $a_t$ randomly. ▷ $\epsilon$-greedy.
6:   else
7:     $a_t = \pi(s_t)$.
8:   Execute $a_t$, and observe $r_t$ and $s_{t+1}$.
9:   if the experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is new then
10:      Add $e_t$ in $E$.
11: Delete old experiences in $E$ to keep $|E| \leq N_e$.
12: procedure POLICY UPDATE
13: Sort experiences in $E$ in a random order.
14: for $i$ from 1 to $|E|$ do
15:    Read the $i$-th experience from $E$.
16: Update weights.

• Exploration of a new path by $\epsilon$-greedy strategy.
  ○ With $\epsilon$ probability, take random action instead of $\pi(s_t)$.

• Symbolic Execution step collects what is called the experience.
  ○ $e_t = (s_t, a_t, r_t, s_{t+1})$

• Policy Update step uses these experiences to update the Q-network.
Unique Path Finder (UPF)

• UPF attempts to help us gather at least one new experience in each symbolic execution step.

• Virtual execution:
  o Defined as a sequence of state transitions using $\pi(s_t)$ with an $\epsilon$-greedy strategy over an observed computation tree, which means a computation tree built up by all of observed experiences.
  o Namely, the virtual execution is not an execution of a real program, but a simulation of state transitions among states that have been already observed.
  o Such a simulation takes negligible time to run.

Unique Path Finder that discovers a prefix (P1) of a brand-new execution path by virtual execution, which is a run over a computation tree built by observed experiences. Symbolic execution that follows is guided by the prefix P1 and finds out the remaining (P2) of the new execution path.
Experiments

• Class 1 programs:
  o The worst-case branch behavior is continuous and follows a simple pattern like “always True” or “always False”
  o These are the programs where WISE is effective, and SPF-WCA works exactly the same as WISE.

• Class 2 programs:
  o Some or all of branch points have a **irregular branch behavior** in the worst case.
  o The worst-case-leading decision at a branch point can change depending on the scale \( (N) \), or the time \( (t) \) that the branch point is visited.
  o WISE cannot handle Class 2 programs efficiently.
  o SPF-WCA can be effective for some of them, *i.e.*, when the pattern can be expressed in terms of the history-length
Class 1 example

<table>
<thead>
<tr>
<th>Benchmark 1: Biopython pairwise2: Smith-Waterman [39]</th>
<th>(N, longest path length)</th>
<th>(3,9)</th>
<th>(4,12)</th>
<th>(5,15)</th>
<th>(10,30)</th>
<th>(20,60)</th>
<th>(30,90)</th>
<th>(100,300)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive search</td>
<td>Paths</td>
<td>127</td>
<td>511</td>
<td>2047</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0:04</td>
<td>0:18</td>
<td>1:14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WISE</td>
<td>Paths</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:01</td>
</tr>
<tr>
<td>PySE</td>
<td>Paths</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0:02</td>
<td>0:02</td>
<td>0:02</td>
<td>0:02</td>
<td>0:02</td>
<td>0:02</td>
<td>0:13</td>
</tr>
</tbody>
</table>

- Exhaustive search: search time exponentially grows
- WISE: small-scale tests predict the worst-case at a larger scale.
- PySE: finds the worst-case within a few trials.
Class 2 example (1/2)

<table>
<thead>
<tr>
<th>GNU grep : Boyer-Moore</th>
<th>(N, longest path length)</th>
<th>(3,3)</th>
<th>(4,3)</th>
<th>(5,3)</th>
<th>(10,9)</th>
<th>(20,18)</th>
<th>(30,30)</th>
<th>(100,99)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exhaustive search</strong></td>
<td>Paths</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>40</td>
<td>1093</td>
<td>88573</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:01</td>
<td>0:31</td>
<td>43:39</td>
<td>-</td>
</tr>
<tr>
<td><strong>WISE</strong></td>
<td>Paths</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>40</td>
<td>1093</td>
<td>88573</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:01</td>
<td>0:32</td>
<td>44:24</td>
<td>-</td>
</tr>
</tbody>
</table>

- WISE cannot handle: GNU grep's worst-case branching behavior shows an irregular pattern
<table>
<thead>
<tr>
<th></th>
<th>(N, longest path length)</th>
<th>(3,3)</th>
<th>(4,3)</th>
<th>(5,3)</th>
<th>(10,9)</th>
<th>(20,18)</th>
<th>(30,30)</th>
<th>(100,99)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPF-WCA trained</td>
<td>Paths</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>243</td>
<td>19683</td>
<td>-</td>
</tr>
<tr>
<td>at N=3,4</td>
<td>Time</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>00:07</td>
<td>10:20</td>
<td>-</td>
</tr>
<tr>
<td>SPF-WCA trained</td>
<td>Paths</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>at N=6,7</td>
<td>Time</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
</tr>
<tr>
<td>PySE pre-trained</td>
<td>Paths</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>276</td>
</tr>
<tr>
<td>at N = 5</td>
<td>Time</td>
<td>0:11</td>
<td>0:11</td>
<td>0:11</td>
<td>0:11</td>
<td>0:12</td>
<td>0:20</td>
<td>48:21</td>
</tr>
<tr>
<td>PySE pre-trained</td>
<td>Paths</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>82</td>
</tr>
<tr>
<td>at N = 10</td>
<td>Time</td>
<td>0:11</td>
<td>0:11</td>
<td>0:11</td>
<td>0:02</td>
<td>0:12</td>
<td>0:20</td>
<td>13:03</td>
</tr>
</tbody>
</table>

- SPF-WCA may handle, but its performance is sensitive to the length of history.
- PySE can handle it and the length of history is not critical.
Concluding remarks

• PySE uses symbolic execution to run a program and collects behavioral information.

• PySE then updates a branching policy using the collected behaviors based on a reinforcement learning framework.

• By iterating the symbolic execution and policy update, PySE gradually increases the length of an execution path towards a path of the worst-case complexity.

• In various Python programs and scales, PySE can effectively find a path of worst-case complexity and has benefits against exhaustive search and WISE-like algorithms.
Thank you!