Pruning the Search Space in Path-based Test Generation

Sébastien Bardin
sebastien.bardin@cea.fr
CEA-LIST, Software Security Labs

(joint work with Philippe Herrmann)
Automatic test data generation from source code (STDG)

The test suite must achieve a **global** structural coverage objective

- all instructions, all branches, etc.

Do not consider the oracle generation issue: assume an external **automatic** oracle

- perfect oracle (back-to-back testing)
- partial oracle (assertions / contracts)
Symbolic Execution (SE) is a very fruitful approach for STDG

- efficiency
- robustness

SE in a nutshell

**Constraint-based reasoning**: translate a part of the program into a logical formula $\varphi$, such that a solution of $\varphi$ is a relevant TD

**Path-based approach**: focus on a single path at once + enumerate (bounded) paths

- simple formulas, only conjunctions (no quantifier / fixpoint)

**Concolic paradigm**: combination of symbolic and dynamic execution

- robustness to “difficult-to-model” programming features
A few prototypes

**PathCrawler (CEA)** 2004
**Dart** (Bell Labs), **Cute** (Uni. of Illinois / Berkeley) 2005
**Exe** (Stanford) 2006
**Jpf** (NASA) 2007
**Osmose** (CEA), **Sage** (Microsoft), **Pex** (Microsoft) 2008
Main Limitations

Two major bottlenecks for Symbolic Execution

1. constraint solving (along a single path)
2. \# paths

Path explosion phenomenon

- nesting loops and conditional instructions
- inlining of function calls

Moreover: SE require a user-defined path-bound $k$

- things get worse if $k$ is over-estimated
- sometimes, very long paths to exhibit specific behaviours

Our goal: lower the path explosion in SE
Motivation

Irrelevant paths

- In practice, SE enumerates all k-paths
- But the true goal is to cover “items” (instr., branches)
- Some paths are very unlikely to improve the current coverage

Idea: detect a priori irrelevant paths to discard them and lower the path explosion

Our results

1. three complementary heuristics to prune likely redundant paths
2. implementation in the Osmose tool and experiments
Outline

- Context
- Symbolic Execution
- Heuristics
- Experiments
- Conclusion
π a finite path of the program $P$

$D$ the input space of $P$

$V \in D$ an input vector

---

**Path predicate**

A path predicate for $\pi$ is a formula $\varphi_\pi$ interpreted on $D$ s.t. if $V \models \varphi_\pi$ then the execution of $P$ on $V$ exercises $\pi$ at runtime.

More formally: let $\pi = t_1 \to t_2 \to \ldots \to t_n$

- the greatest path predicate
  $$\bar{\varphi}_\pi = \text{wpre}(t_1, \text{wpre}(t_2, \ldots \text{wpre}(t_n, \top)))$$

- a path predicate
  $$\varphi_\pi$$

such that $\varphi_\pi \Rightarrow \bar{\varphi}_\pi$

A path predicate is typically computed via strongest postcondition.
Path-based test data generation

1. choose an uncovered \((k\text{-bounded})\) path \(\pi\)
2. compute one of its path predicates \(\varphi_\pi\)
3. solve \(\varphi_\pi\) : solution = TD exercising path \(\pi\)
4. update coverage, if still something to cover then goto 1

Parameter 1 - Logical theory : not relevant here
Parameter 2 - Path enumeration strategy : here, standard DFS
Extension - Concolic execution
Symbolic Execution, Basic Procedure (BP)

Motivation

SE

Heuristics

Experiments

Conclusion

Symbolic Execution, Basic Procedure (BP)

choose path
compute path predicate, solve it, update cover
choose the next path by DFS backtracking, and so on
Symbolic Execution, Basic Procedure (BP)

**Motivation**

**SE**

**Heuristics**

**Experiments**

**Conclusion**

Symbolic Execution, Basic Procedure (BP)

Symbolic Execution, Basic Procedure (BP)

- **choose path**
- compute path predicate, solve it, update cover
- choose the next path by DFS backtracking, and so on
Symbolic Execution, Basic Procedure (BP)

choose path
compute path predicate, solve it, update cover
choose the next path by DFS backtracking, and so on
Symbolic Execution, Basic Procedure (BP)

Motivation
SE
Heuristics
Experiments
Conclusion

Symbolic Execution, Basic Procedure (BP)

choose path
compute path predicate, solve it, update cover
choose the next path by DFS backtracking, and so on
Symbolic Execution, Basic Procedure (BP)

Motivation
SE
Heuristics
Experiments
Conclusion

Symbolic Execution, Basic Procedure (BP)

choose path
compute path predicate, solve it, update cover
choose the next path by DFS backtracking, and so on

READ(X), READ(Y)

X < Y
C := 0
Y >= 20

X >= Y
C := X - Y
C < 20
C >= 20
Heuristic 1: Look-Ahead (LA)

Procedure BP tries to cover a new path at each iteration

BUT this new path does not necessarily cover new items

- the resolution time is wasted
- more useless paths will be explored from this prefix

On the example, full coverage requires at most 3 TD, while there are \( \approx 2^{k+1} \) paths of length \( \leq k \).
Check if uncovered items may be reached from the current instruction. If not, solve the current prefix but do not expand it.

Optimistic check based on the CFG abstraction of the program.

The Look-Ahead heuristic enjoys nice properties:

- **Soundness**: discard only redundant paths.
- **Relative completeness**: BP+LA achieves always the same coverage than BP.
- **Path reduction**: BP+LA explores always less path than BP.

Difficulty: efficient computation of the (CFG) reachability set.
Procedure ReachSet : node $\rightarrow$ Set(node)

Standard worklist algorithm has the following problems in our context:

- all reachability sets are computed at the same time, even if BP will not use all of them
- not designed for interprocedural or context-sensitive analysis
Reachability Set Computation (2)

Efficient interprocedural analysis

- Lazy computation
- Computation cache

Interprocedural analysis

- Compact representation of sets of nodes: manipulate CFG nodes and Call Graph (CG) nodes
- Function summaries: propagate reachable CG nodes (from CG)
- Lazy computation and computation cache extend to CG
Reachability Set Computation (3)

Context-sensitive analysis

the current stack is passed as an argument, if the current node can reach a ret instruction, then the procedure is recursively launched on the top of the stack (return site)

\[
\text{ReachSet-context}(\text{node}, \text{stack}, \text{rset}) : \\
\begin{align*}
\blacktriangledown & \quad c := \text{ReachSet}(\text{node}); r := c \cup \text{rset} \\
\blacktriangledown & \quad \text{if (stack.empty or ret} \notin c) \text{ then return } r; \\
\blacktriangledown & \quad \text{else return ReachSet-context(stack.top, stack.tail, r)}
\end{align*}
\]

Remark : the computation cache is still a map from \textit{node} to \textit{set}, rather than a map from \textit{(node, stack)} to \textit{set}
Heuristic 2: Max-CallDepth (MCD)

Nested function calls are often **the** major source of path explosion.

BP explores all the paths in callees.

But in unit testing, need to cover only paths of the top-level function.

Example: only two TD to cover the main function, but $\approx 2^{k+1}$ paths
Idea

Motivation
SE
Heuristics
Experiments
Conclusion

(claim) top-level paths rarely depend only on specific behaviours in deep function calls

MCD heuristic : prevent backtracking in deep nested function calls

Implementation : a user-defined mcd parameter, a counter depth updated by call and ret, performs branching only if depth ≤ mcd

Theoretically : take care, the MCD heuristic is not sound
Empirically : experimental results show a very large pruning and no loss in coverage (see after)
Heuristic 3: Solve-First (SF)

DFS has two main drawbacks in our context:

- If # TD is limited, DFS focuses only on a deep narrow portion of the program (slow coverage speed).
- Longer (and more complex?) prefixes are solved first.

Example: assume #node = 2n+1, all paths are feasible, goal = instruction coverage

- Only two TD are necessary.
- BP+LA: n+1 TD.
Idea

Slight modification of the concolic DFS procedure

- on a choice point, choose which branch B1 will be followed (symbolically) first
- immediately solve the other branch B2 (TD2), execute TD2 and update coverage info, store TD2
- execute symbolically the procedure through branch B1 (as usual)
- when backtracking through B2, TD2 can be retrieved if needed

Mostly the DFS symbolic execution, except than along a given prefix, every alternative branch has been concretely expanded once

- minimal overhead
- along a path, shorter prefixes are solved first
- some distant portion of the program (in a DFS ordering) are exercised very early
Idea (2)
Idea (2)

Motivation
SE
Heuristics
Experiments
Conclusion
Idea (2)
Idea (2)
<table>
<thead>
<tr>
<th>Heuristics</th>
<th>Relative completeness</th>
<th># path reduction</th>
<th>Implementation in BP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look-Ahead</td>
<td>yes</td>
<td>always</td>
<td>efficient reach. test</td>
</tr>
<tr>
<td>Max-CallDepth</td>
<td>no</td>
<td>not sure</td>
<td>easy</td>
</tr>
<tr>
<td>Solve-First</td>
<td>yes</td>
<td>not sure</td>
<td>easy (concolic setting)</td>
</tr>
</tbody>
</table>
Outline

- Context
- Symbolic Execution
- Heuristics
- Experiments
- Conclusion
### About experiments

Heuristics implemented in the **Osmose** tool (SE for executable files)
Small C programs cross-compiled to C509 and PPC architectures
Configuration: Intel Pentium M 2Ghz, RAM 1.2 GBytes, Linux

<table>
<thead>
<tr>
<th>program</th>
<th>#I</th>
<th>#Br</th>
<th>#F</th>
<th>CD</th>
<th># T</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-pressure</td>
<td>59</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>square 3x3</td>
<td>272</td>
<td>46</td>
<td>1</td>
<td>0</td>
<td>43</td>
</tr>
<tr>
<td>square 4x4</td>
<td>274</td>
<td>46</td>
<td>1</td>
<td>0</td>
<td>123</td>
</tr>
<tr>
<td>hysteresis</td>
<td>91</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>merge</td>
<td>56</td>
<td>24</td>
<td>3</td>
<td>1</td>
<td>70</td>
</tr>
<tr>
<td>triangle</td>
<td>102</td>
<td>38</td>
<td>5</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>ppc-square 4x4</td>
<td>226</td>
<td>30</td>
<td>1</td>
<td>0</td>
<td>125</td>
</tr>
<tr>
<td>ppc-hysteresis</td>
<td>76</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>251</td>
</tr>
<tr>
<td>ppc-merge</td>
<td>188</td>
<td>16</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ppc-triangle</td>
<td>40</td>
<td>18</td>
<td>3</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>

#I : n. of instructions  
#F : n. of functions  
#T : n. of tests (full Br cover)  

#Br : n. of branches  
CD : maximal call depth
Motivation  
SE  
Heuristics  
Experiments  
Conclusion

Results

Notations: BP (Basic Procedure), UT (Unit Testing)

Comparisons

- BP+LA vs BP
- BP+UT+MCD vs BP+UT
- BP+SF vs BP

<table>
<thead>
<tr>
<th></th>
<th>average benefit</th>
<th>win-loss</th>
<th>max benefit</th>
<th>max loss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(time</td>
<td>#path)</td>
<td>W/D/L</td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>-57%</td>
<td>-57%</td>
<td>7/2/1</td>
<td>8/2/0</td>
</tr>
<tr>
<td>MCD</td>
<td>-85%</td>
<td>-72%</td>
<td>5/1/0</td>
<td>5/1/0</td>
</tr>
<tr>
<td>SF+LA</td>
<td>-61%</td>
<td>-80%</td>
<td>4/0/5</td>
<td>5/0/4</td>
</tr>
</tbody>
</table>

Sébastien Bardin, Philippe Herrmann
### Summary (2)

<table>
<thead>
<tr>
<th></th>
<th>theoretical</th>
<th>empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>relative</td>
<td># path reduction</td>
</tr>
<tr>
<td>LA</td>
<td>yes</td>
<td>always</td>
</tr>
<tr>
<td>MCD</td>
<td>no</td>
<td>not sure</td>
</tr>
<tr>
<td>SF+LA</td>
<td>yes</td>
<td>not sure</td>
</tr>
</tbody>
</table>
Other experiments

LA overhead: reachability set is computed, but test inclusion always answers yes

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Mean</th>
<th>Variability</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS computed on backtrack only</td>
<td>+0%</td>
<td>+0% - +1%</td>
</tr>
<tr>
<td>RS computed at each branch</td>
<td>+2.4%</td>
<td>+0% - +7%</td>
</tr>
</tbody>
</table>
Path enumeration strategy for better coverage speed

- best-first search (EXE, SAGE, PEX): active prefixes are ranked, and the best one is expanded
- hybrid search (CUTE): DFS + random

Redundant paths

- discard a path prefix if similar to an already expanded path prefix (EXE), state caching / state abstraction (JPF)
- discard a path prefix when it cannot reach an interesting state (YOGI) and the Synergy approach

Concurrent systems and interleaving

- dynamic partial orders (CUTE)
Related work (2)

Function calls

- when the maximal depth is reached, a call returns $\top$ (JPF)
- function concretisation (CUTE) can also be used for path pruning

Other techniques

- lazy handling of function calls via uninterpreted symbols (SAGE)
- incremental construction of a summary function (DART)
- user-defined function specification (PathCrawler)
We propose three heuristics to perform path pruning in Symbolic Execution:

- easy to implement, whatever the path enumeration strategy is
- all the three techniques are complementary

Very encouraging results for Look-Ahead and Max-CallDepth on limited benchmarks.

Solve-First shows a positive global gain, but much more variability.

Future work:

- experiments on larger programs and with other path search methods
- application to search-based testing?