Design of a Modified Concolic Testing Algorithm with Smaller Constraints

Yavuz Koroglu       Alper Sen

Department of Computer Engineering
Bogazici University, Turkey
yavuz.koroglu@boun.edu.tr
depend.cmpe.boun.edu.tr

International workshop on Constraints in Software Testing, Verification and Analysis CSTVA@ISSTA 2016
Definition

If a testing technique uses a constraint solver to generate test cases, it is called **Constraint-Based Testing (CBT)**.
The term coined in 1991 by Offut and DeMillo.

**Symbolic Execution** (dates back to 1975),
- Considered *impractical*, lack of powerful constraint solvers.

**Revival** in the last two decades,
- Availability of powerful constraint solvers (Yices, Z3 etc.),
- **Concolic testing** is proposed.

**Constraint solving bottleneck**, 
- **Scalability** issues.
- Constraint solving optimizations (Concolic Unit Testing Engine (CUTE) offers **three** optimizations).
  - Did not completely solve the issue.
Our Motivation

What did we aim?

Design a modification on the current constraint solving methodology which

- Decreases the burden on the constraint solver,
- Still gets the same coverage as the previous CBT approaches and
- Allows new heuristics and optimizations to be implemented.
Our Motivation

What did we see?

CBT approaches make **few large queries** to the constraint solver.

- Instead, make **thousands of small queries**.
- In model checking domain, IC3 uses this strategy.  
  (SAT-Based Model Checking Without Unrolling, Aaron R. Bradley, VMCAI2011)
- Can we better utilize constraint solvers in CBT?
Concolic Testing

- Also called **Dynamic Symbolic Execution (DSE)**.
- Combines *concrete* and *symbolic* execution.
- The idea dates back to **2005** (CUTE and DART).
- We implement our approach on top of Concolic Testing.
Example: Greatest Common Divisor (GCD)

```c
int gcd(int a, int b) {
    if (a <= 0) {     // L0
        return ERROR;   // L1
    }
    if (b <= 0) {     // L2
        return ERROR;   // L3
    }
    while (a != b) {  // L4
        if (a > b) {    // L5
            a = a - b;   // L6
        } else {        // L7
            b = b - a;
        }
    }
    return a;        // L8
}
```
1) Generate random inputs: let $a = 4$, $b = 0$. 

\[
\begin{align*}
L_0 & \quad a \leq 0 \\
L_1 & \quad a \leq 0 \quad \text{ERROR} \\
L_2 & \quad a > 0 \\
L_3 & \quad b \leq 0 \quad \text{ERROR} \\
L_4 & \quad a = b \\
L_5 & \quad a \leq b \\
L_6 & \quad a > b \\
L_7 & \quad b \leftarrow b - a \\
L_8 & \quad a = b
\end{align*}
\]
2) gcd(4,0) traverses the following execution path: $L_0 \rightarrow L_2 \rightarrow L_3$. 

![Graph diagram of gcd(4,0) execution path]
Test GCD using Concolic Testing

3) Gather $\pi_0 = (a > 0) \land (b \leq 0)$ during execution.
4) $\pi_0$ is a **full path constraint**.
5) Full path constraint is the conjunction of all path conditions on an execution path.
6) Generate $\phi_1 = (a > 0) \land (b > 0)$.
7) Let $CS(\phi_1)$ be $a = 4$ and $b = 6$. 
Test GCD using Concolic Testing

8) gcd(4,6) traverses
$L_0 \rightarrow L_2 \rightarrow L_4 \rightarrow L_5 \rightarrow L_7 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_8$.

\[ a \leftarrow a - b \]
\[ b \leftarrow b - a \]
\[ a \leq 0 \]
\[ b \leq 0 \]
\[ a = b \]
\[ a > 0 \]
\[ b > 0 \]
\[ a \neq b \]
\[ a \leq b \]
\[ a > b \]
9) Gather $\pi_1 = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] = b - a)$. 

**Diagram:**

- $L_0$: $a \leq 0$
- $L_1$: $b \leq 0$
- $L_2$: $a > 0$
- $L_3$: $b > 0$
- $L_4$: $a = b$
- $L_5$: $a \leq b$
- $L_6$: $a > b$
- $L_7$: $b \leftarrow b - a$
- $L_8$: $a \leftarrow a - b$
10) Solved only a small constraint ($\phi_1$) to get an input which satisfies a large constraint ($\pi_1$).
11) After a few iterations constraints get very large.
12) Generate $\phi_2 = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$. 

```
L0
L2
L4
L6
L8

a ← a − b
b ← b − a
```

```
L1
L3
L5
L7

ERROR
ERROR
```
13) Let $\text{CS}(\phi_2) = a = 5, b = 6$. 

![Diagram showing the GCD algorithm with states and transitions labeled with conditions and actions.]

- $L_0$: $a \leq 0$
- $L_1$: $L_0 \rightarrow L_1$
- $L_2$: $a > 0$
- $L_3$: $L_0 \rightarrow L_3$
- $L_4$: $L_2 \rightarrow L_4$
- $L_5$: $L_4 \rightarrow L_5$
- $L_6$: $L_5 \rightarrow L_6$
- $L_7$: $L_4 \rightarrow L_7$
- $L_8$: $L_4 \rightarrow L_8$

Actions:
- $a \leftarrow a - b$
- $b \leftarrow b - a$

Conditions and Transitions:
- $a \leq 0$
- $b \leq 0$
- $a = b$
- $a > b$
- $a \neq b$
- $a > 0$
- $b > 0$
- $a \leq b$
14) gcd(5,6) traverses $L_0 \rightarrow L_2 \rightarrow L_4 \rightarrow L_5 \rightarrow L_7 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow \ldots$
One of the first concolic testers, CUTE, proposes three optimizations for constraint solving:
1) Fast Unsatisfiability Check.
2) Common Sub-Constraints Elimination.
3) Incremental Solving.
Main Idea

- Check if a path condition is **syntactically the negation** of any preceding ones in the full path constraint. 
  e.g. \( \pi = \ldots \land (a = b) \land \ldots \land (a \neq b) \land \ldots \)

- If it is, the full path constraint is decided to be **infeasible** without solving.

**OPT1,**

- Reduces the number of constraint solver queries by 60-95% in general.
- Reduction in the GCD example: 0%.
(OPT2) Common Sub-Constraints Elimination

Main Idea

- Identify and eliminate common sub-constraints.

OPT2,

- Reduces common sub-constraints by 64-90% in general.
- Reduction in the GCD example: 0%.
Main Idea

- Remember that $\pi_0 = (a > 0) \land (b \leq 0)$ and $\phi_1 = (a > 0) \land (b > 0)$ from the GCD example.
- $\pi_0$ and $\phi_1$ only differ by one condition.
- Let the conjunction of all conditions on $\phi_1$ that depend on $(b > 0)$ be $\phi_1' = (b > 0)$.
- Let the solver fix $a$ to its previous value and find a solution for $\phi_1'$ instead of $\phi_1$.

OPT3,

- On average, $|\phi'| \approx |\phi|/8$ in general.
- On the GCD example: No significant improvement.
Partial Path Constraints ($\phi$)

**Definition**

Any **overapproximation** of the Full Path Constraint $\pi$ is called a Partial Path Constraint ($\phi$).

**Example**

- Let $\pi = (a > 0) \land (b \leq 0)$.
- Then, the possible partial path constraints are
  - $\phi_0 = T$,
  - $\phi_1 = (a > 0)$,
  - $\phi_2 = (b \leq 0)$ and
  - $\phi_3 = (a > 0) \land (b \leq 0)$. 
There are subsumed path conditions.

1. In the GCD example, $\phi_2$ contains both $p = (a \neq b - a)$ and $q = (a > b - a)$.
2. Trivially, $q \rightarrow p$.
3. So, $p$ is redundant.
4. We should eliminate redundant path conditions.
Consider $\pi = (a > 0) \land (b > 0) \land (a = b)$.

Let $\phi = (a = b)$.

Probability of $\text{CS}(\phi)$ also satisfies $\pi$ is 0.25.

For $\phi' = (b > 0) \land (a = b)$, probability becomes 0.50.

Danger!

Usage of partial path constraints may cause path divergence.

Therefore, some feasible execution paths may not get executed (incompleteness).
Incremental Partial Path Constraints (IPPC)

Main Idea

- **Same** as concolic testing.
- We **replace** the constraint solver call with IPPC.
- IPPC tries a **small** partial path constraint.
- Learns **larger** $\phi$ and tries again until the answer is found.
Incremental Partial Path Constraints (IPPC)

Algorithm

1. Start from a partial path constraint $\phi$ where $\pi \rightarrow \phi$.
2. Generate test input $i$ that satisfy $\phi$.
3. If $\phi$ is infeasible, then $\pi$ must be infeasible.
4. Else if $i$ satisfies $\pi$, return $i$.
5. Find out the first path condition $c_d$ which $i$ does not satisfy.
6. Let $\phi \leftarrow \phi \land c_d$.

- $c_d$ is called the **Cause of Divergence**.
- Steps 5-6-7 occurs only if generated $i$ causes a **path divergence**.
Determining Initial $\phi$: Most Basic Strategy

<table>
<thead>
<tr>
<th>Motivation</th>
<th>We negate <strong>only one condition</strong> on the previously satisfied full path constraint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>Take the <strong>negated condition</strong> as the initial $\phi$.</td>
</tr>
<tr>
<td>Advantage</td>
<td>Incremental Solving optimization (OPT3) has more chance to satisfy $\pi$ by fixing some of the inputs.</td>
</tr>
</tbody>
</table>
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).
Example Returned: GCD

1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

2) Let us solve this constraint using IPPC instead of a CS call.
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

3) $\phi^1 = (a - [b - a] \neq b - a)$. 

\[
\begin{align*}
    &\text{ERROR} &\text{ERROR} &\quad a \\
    &L_1 &L_3 &L_8 \\
    &a \leq 0 &b \leq 0 &a = b \\
    &L_0 &L_2 &L_4 \\
    &a > 0 &b > 0 &b \leftarrow b - a \\
    &L_6 &L_5 &L_7 \\
    &a \leftarrow a - b &a \neq b &a \leq b
\end{align*}
\]
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

4) Yices in incremental mode generates $a = 2, b = 3$ for $\text{CS}(\phi^1)$. 

\[ a \leftarrow a - b \]
\[ b \leftarrow b - a \]
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

5) $(a, b) = (2, 3)$ does NOT satisfy $\pi$ due to $c_d^1 = (a \neq b - a)$.
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

6) $\phi^2 = \phi^1 \land c^1_d = (a - [b - a] \neq b - a) \land (a \neq b - a)$.
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

7) Yices generates $a = 4$, $b = 6$ for $\text{CS}(\phi^2)$ which satisfies the $\pi$. 
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

8) Standard concolic tester solves 1 path constraint of size 7.
1) Consider $\pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a)$ ($\phi_2$ of the previous example).

9) IPPC solves 2 path constraints of sizes 1 and 2.
1) Consider \( \pi = (a > 0) \land (b > 0) \land (a \neq b) \land (a \leq b) \land (a \neq b - a) \land (a > b - a) \land (a - [b - a] \neq b - a) \) (\( \phi_2 \)) of the previous example.

10) More smaller queries vs. Few larger queries
Experimental Environment

The Environment

- Virtual Linux guest with 1024MB memory and one CPU,
- MacBook Pro host with an Intel Core i7 2.9 GHz GPU and 8GB Memory.

The Framework

CREST, is a known concolic testing framework developed by J. Burnim.

- **Source code** available.
- It uses **Yices**.
- It implements **different** concolic testing **strategies**.
List of benchmarks used in the experiments are as follows:

<table>
<thead>
<tr>
<th>UUT</th>
<th>KLOC</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>bsort</td>
<td>0.05</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>prime</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>factor</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>replace</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>ptokens</td>
<td>0.6</td>
<td>40</td>
</tr>
<tr>
<td>grep</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>
All conditions are guaranteed to be correctly solvable by Yices.

<table>
<thead>
<tr>
<th>UUT</th>
<th>KLOC</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>bsort</td>
<td>0.05</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>prime</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>factor</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>replace</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>ptokens</td>
<td>0.6</td>
<td>40</td>
</tr>
<tr>
<td>grep</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>
Benchmarks

## Benchmarks

Benchmarks are in different sizes.

<table>
<thead>
<tr>
<th>UUT</th>
<th>KLOC</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>bsort</td>
<td>0.05</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>prime</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>factor</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>replace</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>ptokens</td>
<td>0.6</td>
<td>40</td>
</tr>
<tr>
<td>grep</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>
We made 10 executions for each configuration.

<table>
<thead>
<tr>
<th>UUT</th>
<th>KLOC</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>bsort</td>
<td>0.05</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>prime</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>factor</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>replace</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>ptokens</td>
<td>0.6</td>
<td>40</td>
</tr>
<tr>
<td>grep</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>
We measured branch coverage via a script which uses gcov.

<table>
<thead>
<tr>
<th>UUT</th>
<th>KLOC</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>bsort</td>
<td>0.05</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>prime</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>factor</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>replace</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>ptokens</td>
<td>0.6</td>
<td>40</td>
</tr>
<tr>
<td>grep</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>
Standard Concolic Testing and IPPC achieves the same coverage in $N$ iterations.

<table>
<thead>
<tr>
<th>UUT</th>
<th>KLOC</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>bsort</td>
<td>0.05</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>prime</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>factor</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>replace</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>ptokens</td>
<td>0.6</td>
<td>40</td>
</tr>
<tr>
<td>grep</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>
### IPPC Speedup over Standard Concolic Testing (DFS)

IPPC has smaller constraints by a factor of 60.

<table>
<thead>
<tr>
<th>UUT</th>
<th>Avg Const. Size Ratio (DFS / IPPC)</th>
<th>Speedup ($t_{DFS}/t_{IPPC}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>4.4</td>
<td>0.6x</td>
</tr>
<tr>
<td>bsort</td>
<td>20.8</td>
<td>0.79x</td>
</tr>
<tr>
<td>sqrt</td>
<td>21.3</td>
<td>1.25x</td>
</tr>
<tr>
<td>grep</td>
<td>31.8</td>
<td>0.83x</td>
</tr>
<tr>
<td>ptokens</td>
<td>48.4</td>
<td>1.7x</td>
</tr>
<tr>
<td>gcd</td>
<td>97.5</td>
<td>2.77x</td>
</tr>
<tr>
<td>prime</td>
<td>115.6</td>
<td>9.1x</td>
</tr>
<tr>
<td>factor</td>
<td>137.3</td>
<td>9.8x</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td><strong>59.6</strong></td>
<td><strong>3.35x</strong></td>
</tr>
</tbody>
</table>
IPPC has a speedup of 3.35 on average.

<table>
<thead>
<tr>
<th>UUT</th>
<th>Avg Const. Size Ratio (DFS / IPPC)</th>
<th>Speedup ($t_{DFS}/t_{IPPC}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>4.4</td>
<td>0.6x</td>
</tr>
<tr>
<td>bsort</td>
<td>20.8</td>
<td>0.79x</td>
</tr>
<tr>
<td>sqrt</td>
<td>21.3</td>
<td>1.25x</td>
</tr>
<tr>
<td>grep</td>
<td>31.8</td>
<td>0.83x</td>
</tr>
<tr>
<td>ptokens</td>
<td>48.4</td>
<td>1.7x</td>
</tr>
<tr>
<td>gcd</td>
<td>97.5</td>
<td>2.77x</td>
</tr>
<tr>
<td>prime</td>
<td>115.6</td>
<td>9.1x</td>
</tr>
<tr>
<td>factor</td>
<td>137.3</td>
<td>9.8x</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td><strong>59.6</strong></td>
<td><strong>3.35x</strong></td>
</tr>
</tbody>
</table>
IPPC has better speedup when the UUT has more infeasibilities.
In this work, we designed a modification which,

- Eliminates the need for solving large constraints,
  - Largest path constraint sizes found during the experiments:
    1. IPPC: 157
    2. DFS: 2922

- Works better if the UUT has many infeasible paths and
- Is flexible.

We also,

- Gave motivational examples and background for our work.
- Strongly suggested a relationship between speedup and infeasibility.
- Did experiments on the benchmarks.
Caching

- KLEE utilizes caching as a performance improving optimization.
- We use partial path constraints,
  - Therefore we can have **both-way** caching:
  - Inputs have a corresponding full path constraint (input $\rightarrow$ path constraint, reduces UUT execution)
  - Full path constraints are mapped to inputs. (path constraint $\rightarrow$ input, reduces CS execution)
Future Work

Independent Path Conditions as the Initial $\phi$

- Using a greedy algorithm, find a set of independent path conditions.
- Conjunct all the independent conditions to get the Initial $\phi$.
- Maybe we can decrease the total CS calls if we use this initial $\phi$. 
Future Work

Implementation on Different Frameworks and More Benchmarks

- We should find more benchmarks (currently there are 8 benchmarks),
- We should implement IPPC on top of different CBT approaches,
Thank You. Any Questions?