Automated Testing: CUTE & SMART

Nick Sumner
with material from:
Patrice Godefroid
Koushik Sen
Recall from class

• Automated tests that are “played on demand”
  – Avoiding interaction
    • introduce fewer errors
    • cheaper

• Difficulties
  – Fragility
    • Interface evolution
    • Code evolution
  – Deciding correctness
  – Developing test suite
Focus

• Automate unit testing

• Automate test generation itself
  – Generate test inputs that examine desired features
    • search for bugs
    • avoid code fragility
    • Integrate into nightly builds

• Automatically detect failures
  – or extensible to allow failure detection
Execution Model

• Execution is viewed as *Computation Tree*
  – Nodes are predicates
  – Edges are straight line code
  – HALT or ABORT lie in the leaves

• Each path from root is an equivalence class
  – Goal in error finding to derive a representative
  – Better path coverage increases chances of discovery
Familiar Example

Read $X, Y$

$Z = 2Y - X$

$2 \cdot Y + X = 0$

false $\rightarrow$

true $\rightarrow$

$Y = 2$

$X - Z > 0$

false $\rightarrow$

true $\rightarrow$
Approach A: Random Testing

- Though successful, cannot reach tightly constrained code
  - Random distribution cannot hit discrete points with even $2^{32}$ options

Consider:

```c
if ( x == 42 )
    abort();
```
Approach B: Symbolic Execution

- Symbolic execution executes programs abstractly
  - Program is function on abstract input variables
  - Collect general constraints along execution paths
  - Attempt to solve in terms of input variables

- Scales poorly

- Limited by conservative static analysis

```c
int obscure(int x, int y) {
    if ( x==hash(y) )
        error();
    return 0;
}
```

```c
int obscure(int x, int y) {
    if ( x==hash(y) )
        error();
    return 0;
}
```

Falls to over or under approximation
CUTE and SMART

- CUTE and SMART: 2 tools using similar approaches
  - Both developed from DART

- Force exploration of all possible execution paths on valid input
  - potentially complete feasible path coverage
  - Combine static and dynamic analysis
    - Randomized testing *supporting* symbolic execution
General Approach

• Perform DFS for errors on computation tree
  – every path from root to (leaf | infinity)

• Solve constraints over tree iteratively to drive execution along all possible paths
  – Upon reaching difficult constraints, use the concrete values from execution to enable pushing past them

```c
int obscure(int x, int y) {
    if ( x==hash(y) )
        abort();
    return 0;
}
```
More Refined

1) Generate randomized inputs for entry method, stored in *input map*

2) Collect symbolic constraints or a best guess along execution path while executing

3) Negate last constraint and solve to generate new path, marking when done.

4) Return to step 2

Original DART had verification flavor

- Never stopped testing until all paths executed (infinite?)
- Ran forever if theory violation led to unexpected paths
CUTE

- **CUTE: A Concolic Unit Testing Environment for C**
- Concolic: Concrete and Symbolic
- More pragmatic
  - Bounded DFS - full *completeness* not realistic
  - Analyzes pointer graphs and constraints
  - Includes efficiency heuristics
  - Theory prediction violation only restarts analysis

[Example: Sen FSE'05 Slides 9-35]
CUTE

• Reconsider yet again:

```c
int obscure(int x, int y) {
    if ( x==hash(y) )
        abort();
    return 0;
}
```

• Expression of $\text{hash}(y)$ is irrelevant.
  - Could be a library or instrumented function; it doesn't matter
CUTE

- Tool available by request, also for Java
- Implementation:
  - Translate into simplified representation (CIL)
  - Instrument source
    - Maintain symbolic memory map over function calls and operations
  - Compile
  - Run cute, which executes instrumented program
Constraint Optimizations

Even using optimized linear solvers is costly

• **Fast unsatisfiability** (60-95% fewer checks)
  - Negation of previous path constraint $\rightarrow$ unreachable

• **Common subconstraints** (64-90% fewer constr.)

• **Incremental solving** (1/8 constr. set)
  - Only constraints related to last on path need be used in calculation of new input map.
    • Constraint set to solve reduced considerably
Constraint Solving

- The set of constraints to solve is *either*
  - Numerical
    - Solved by lp_solve
  - Pointer graph
    - Use equivalence graph from disequalities
    - Ensure no edge when equality added
    - Ensure unequivalence when disequality added

- Locality of reference in computation tree ensures minimal modification to pointer graph between rounds
Data Structure Testing

- Often, programs require valid pointer graphs to function properly
  - e.g. doubly linked lists
- Can provide API to enable proper construction of data structures.
- Can utilize data structure invariant checker when producing structures
  - constraints from invariant checker adjunct to path constraints in input derivation.
Limitations

- Obviously faces effects of path explosion
- Approximation in pointer theory requires direct predicates to push through constraints

```c
a[i] = 0;
a[j] = 1;
if (a[i] == 1)
    abort();
```

- Bounded DFS clearly lacks completeness over looping. Loop intensive programs become intractable.
- Library functions with side effects are clearly analyzable, but relatively glossed over.
Evaluations

• Used in combination with Valgrind to analyze itself.
  – memory leaks discovered in its own source code
  – exhibits orthogonality as a driver to other analyses

• Analysis of SGLIB - Open Source Data Structure
  – Use of structure invariant checker
  – 2000 lines of C
  – Discovered 2 bugs
Evaluations

<table>
<thead>
<tr>
<th>Name</th>
<th>Run time in seconds</th>
<th># of Iterations</th>
<th># of Branches Explored</th>
<th>% Branch Coverage</th>
<th># of Functions Tested</th>
<th>OPT 1 in %</th>
<th>OPT 2 &amp; 3 in %</th>
<th># of Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Quick Sort</td>
<td>2</td>
<td>732</td>
<td>43</td>
<td>97.73</td>
<td>2</td>
<td>67.80</td>
<td>49.13</td>
<td>0</td>
</tr>
<tr>
<td>Array Heap Sort</td>
<td>4</td>
<td>1764</td>
<td>36</td>
<td>100.00</td>
<td>2</td>
<td>71.10</td>
<td>46.38</td>
<td>0</td>
</tr>
<tr>
<td>Linked List</td>
<td>2</td>
<td>570</td>
<td>100</td>
<td>96.15</td>
<td>12</td>
<td>86.93</td>
<td>88.09</td>
<td>0</td>
</tr>
<tr>
<td>Sorted List</td>
<td>2</td>
<td>1020</td>
<td>110</td>
<td>96.49</td>
<td>11</td>
<td>88.86</td>
<td>80.85</td>
<td>0</td>
</tr>
<tr>
<td>Doubly Linked List</td>
<td>3</td>
<td>1317</td>
<td>224</td>
<td>99.12</td>
<td>17</td>
<td>86.95</td>
<td>79.38</td>
<td>1</td>
</tr>
<tr>
<td>Hash Table</td>
<td>1</td>
<td>193</td>
<td>46</td>
<td>85.19</td>
<td>8</td>
<td>97.01</td>
<td>52.94</td>
<td>1</td>
</tr>
<tr>
<td>Red Black Tree</td>
<td>2629</td>
<td>1,000,000</td>
<td>242</td>
<td>71.18</td>
<td>17</td>
<td>89.65</td>
<td>64.93</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 11: Results for testing SGLIB 1.0.1 with bounded depth-first strategy with depth 50

- Branch coverage and run time on live code act as metrics, as is common.
- Examples from live code provide validity
- An interesting metric used by CUTE is the # of iterations of the framework.
SMART

Compositional Dynamic Test Generation

• Again, the verification flavor of DART is present
• While dynamic testing is powerful, it faces tractability setbacks for large scale programs.
• Repeated analysis of code within the computation tree is unnecessary in a specific theory.

• Analyze each function or module separately and reuse the analysis as possible.
  – Systematic Modular Automated Random Testing
SMART Summaries

- A summary is a disjunction of logical constraints in a particular constraint theory.
- Individual terms are conjunctions of
  1. Preconditions on function inputs for the term's summary to apply
  2. Postconditions of effect constraints on the output of a function under the preconditions.
- Only preconditions expressible within the predetermined theory $T$ are admitted
SMART

• Just as with the computation tree, equivalence classes have been defined
  - Classes over call flow graph based on preconditions
  - Equivalence is sound only if all constraints along path are within theory T.

• When constraints lie outside of T, summaries are inaccurate/incomplete, leading to incomplete analysis

```plaintext
1 int g(int x) {
2   int y;
3   if (x < 0)
4     return 0;
5   y = hash(x);
6   if (y == 100)
7     return 10;
8   if (x > 10)
9     return 1;
10  return 2;
11 }

(x >= 0 ^ x <= 10 ^ ret = 2)
```
Computing Summaries

- Upon function f's termination, preconditions are easily observed as the path constraints within f.
- Postconditions are the constraints of any externalized value (or false for termination)
- Every time f is analyzed (on new preconditions) a term is added to its summary

- Top down or bottom up attack:
  - Bottom up may not generate needed and may generate unneeded terms
  - Top down best. Memoize symbolic procedures.
Correctness

• SMART is, within a quantified theory $T$, equivalent to DART.
  – Terminates on known full coverage
  – Terminates on sound bug
  – Nonterminating otherwise

• This is explicitly within $T$, which Godefroid says is seldom consistent.
  – Exchange does have benefits...
Complexity

• Suppose $\exists$ a bound $b$ on path branches within any given function.
  - No function is analyzed more than $b$ times. If there are $N$ functions, SMART search is $O(N)$.

• DART search, as mentioned before, has path explosion
  - Potential complexity is actually $O(2^N)$.

• SMART overhead from summary propositions?
  - Not time intense, as precondition matching can be fast
Example

- $2^N$ program paths
- SMART does 4 runs
  - 2 for summary:
    \[ \Phi = (x>0 \land \text{ret}=1) \lor (x=<0 \land \text{ret}=0) \]
- 2 to execute both branches of (*), by solving the constraint
  \[ [(s[0]>0 \land \text{ret}_0=1) \lor (s[0]=<0 \land \text{ret}_0=0)] \]
  \[ \land [(s[1]>0 \land \text{ret}_1=1) \lor (s[1]=<0 \land \text{ret}_1=0)] \]
  \[ \land \ldots \land [(s[N-1]>0 \land \text{ret}_{N-1}=1) \lor (s[N-1]=<0 \land \text{ret}_{N-1}=0)] \land (\text{ret}_0+\text{ret}_1+\ldots+\text{ret}_{N-1} = 3) \]

```c
int is_positive(int x) {
    if (x>0)
        return 1;
    return 0;
}
#define N 100
void top(int s[N]) { //N inputs
    int i, cnt=0;
    for (i=0; i<N; i++)
        cnt = cnt + is_positive(s[i]);
    if (cnt == 3) error(); //(*)
    return;
}
```
Clarification

- Memoized constraints from the summaries are adjunct to the path constraints
  - Similar to the data structure invariant constraints
  - Cannot be negated by the path forcing process.
  - Different explicit path traversal than DART / CUTE
Case Study

- Implementation of SMART created, though not obviously available.
- Comparison made between DART and SMART on limited subset of oSIP code.

![Graph showing comparison between DART and SMART](image)

**Figure 4.** Experimental comparison between DART and SMART
Metrics

- Only real metric present is comparison to DART in Number of Runs vs. Input Size
  - Memo storage could devour significant enough space to slow the process considerably.
  - While the asymptotic behavior in terms of test runs is, as expected, significantly different, the unlisted factors are interesting enough to not ignore.
  - Still, no real trials of noteworthy size have been performed with Concolic test units.
Interesting Results

• Interleaving opposing methodologies can yield more benefit than either alone.
  - Degree of integration seems to increase over time.

• Never forget:
  - reduction of subproblems to equivalence classes
  - cache or choose single representative
Trade Offs

- Consider the message summary approach used in SMART.
  - Is the message summary efficiency gain worth being restricted to a feasible theory in analysis?
    - no more piggybacking of Valgrind, etc.
- Is the limitation of automated testing only to unit tests reasonable for the coverage provided by CUTE?
Possibilities

• What are the possible advantages or disadvantages of loop invariant analysis within CUTE?

• A lattice on pre and post conditions in SMART is given. What sorts of heuristics would be beneficial to the goal of increasing the precision, and therefore completeness?
  – e.g. Checking that postconditions validate

Godefroid. “Compositional Dynamic Test Generation,” POPL'2007 Talks

