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

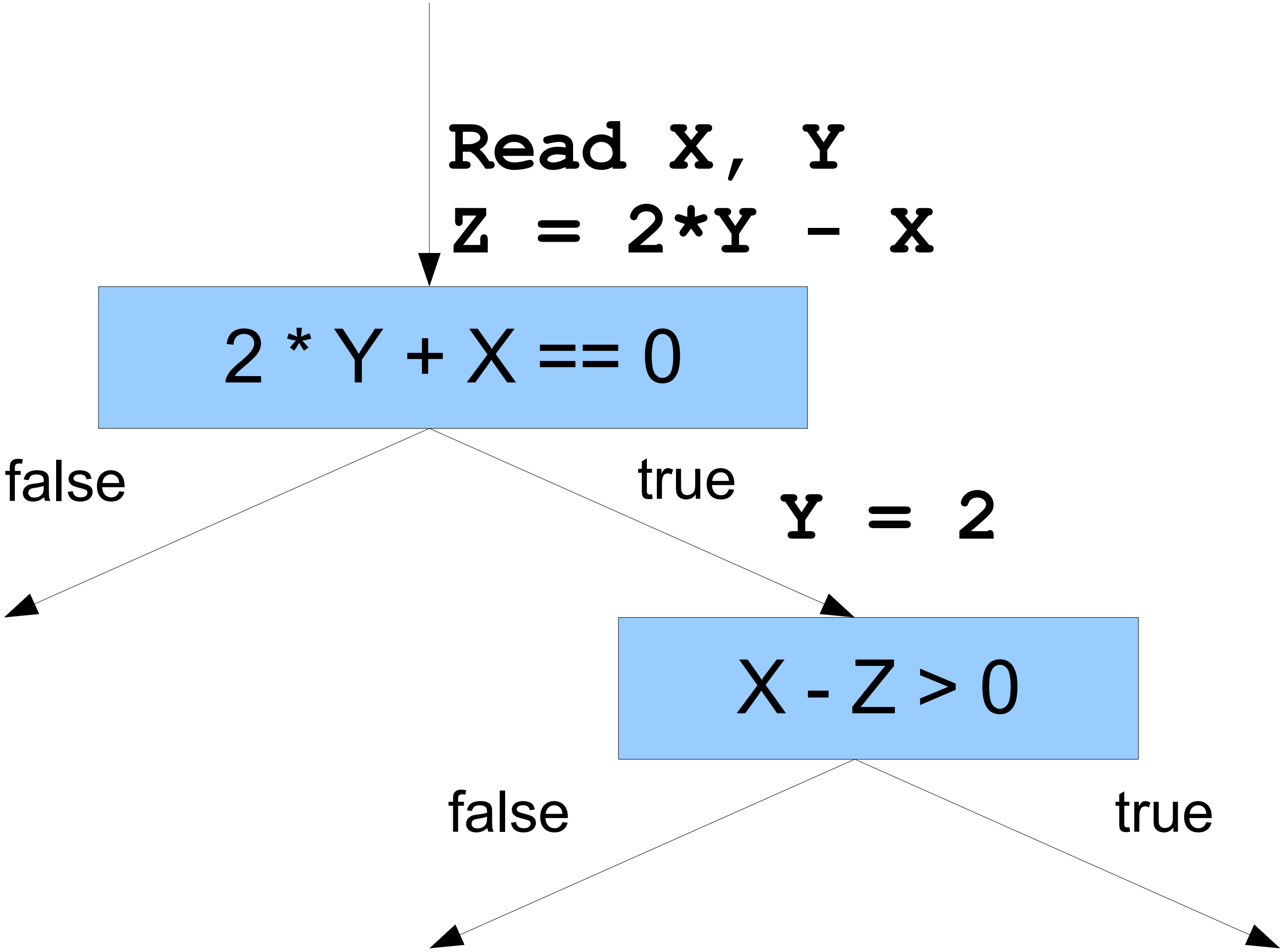
- Automate unit testing
- Automate test generation itself
  - Generate test inputs that examine desired features
    - search for bugs
    - avoid code fragility
    - Integrate into nightly builds
- Automatically detect failures
  - or extensible to allow failure detection

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 to derive a representative
  - Better path coverage increases chances of discovery

Familiar Example



```
Read X, Y
Z = 2*Y - X
```

2 * Y + X == 0

false

true

```
Y = 2
```

X - Z > 0

false

true

Approach A: Random Testing

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

Consider:

```
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

```
test_me(int x){
   if( (x%10)*4!=17 ){
      ERROR;
   } else {
      ERROR;
   }
}
```

```
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

```
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:

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

- Expression of `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 → 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

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

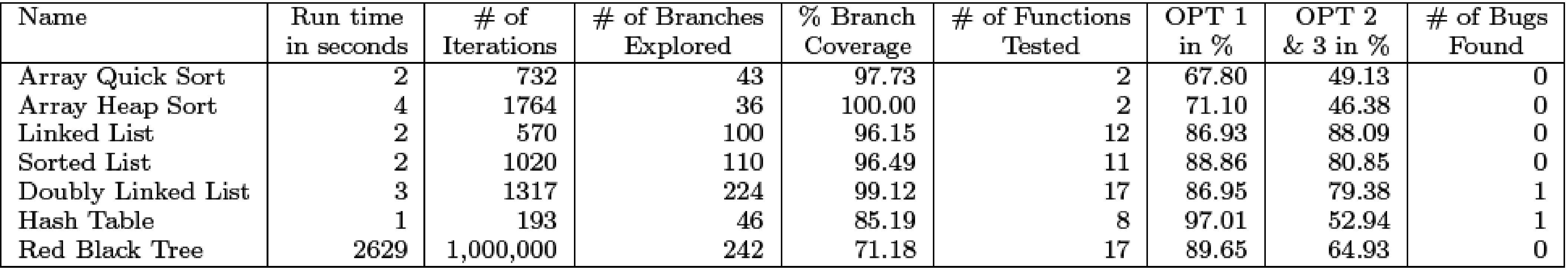
- Bounded DFS clearly lacks completeness over looping. Loop intensive programs become intractable.
- Library functions with side effects are clearly analyzable, but relatively glossed over.

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



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

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

```
int g(int x) {
 int y;
 if (x < 0)
     return 0;
 y = hash(x);
 if (y == 100)
     return 10;
 if (x > 10)
     return 1;
 return 2;
}
```

(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 ∃ 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

```
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;
}
```

- 2^N program paths
- SMART does 4 runs
  - 2 for summary:

$$\mathbf{\Phi = (x>0 \wedge ret=1) \vee (x=<0 \wedge ret=0)}$$

- 2 to execute both branches of (*),by solving the constraint

$$[(s[0]>0 \wedge ret_0=1) \vee (s[0]=<0 \wedge ret_0=0)] \wedge [(s[1]>0 \wedge ret_1=1) \vee (s[1]=<0 \wedge ret_1=0)] \wedge \ldots \wedge [(s[N-1]>0 \wedge ret_{N-1}=1) \vee (s[N-1]=<0 \wedge ret_{N-1}=0)] \wedge (ret_0+ret_1+\ldots+ret_{N-1} = 3)$$

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.

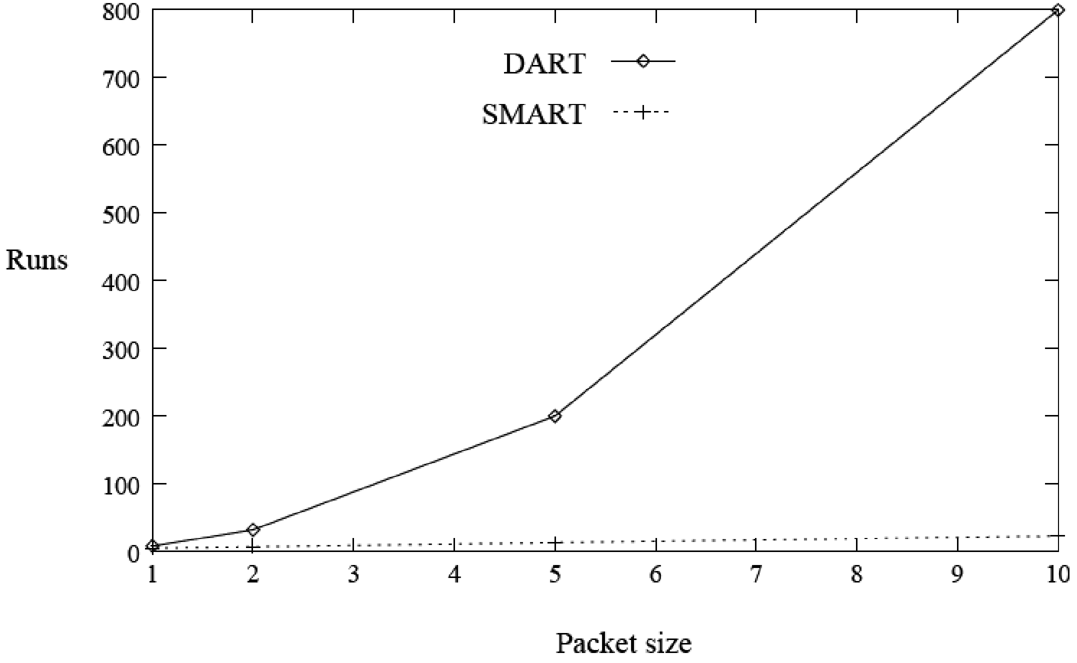


Figure 4. Experimental comparison between DART and SMART

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 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

Thank You

Godefroid. “Compositional Dynamic Test Generation” Proceedings of POPL'2007 (34th Annual ACM Symposium on Principles of Programming Languages), pages 47-54, Nice, January 2007.

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

Patrice Godefroid, Nils Klarlund, Koushik Sen: DART: directed automated random testing. PLDI 2005: 213-223

Sen. "CUTE: A Concolic Unit Testing Engine for C", ESEC/FSE 2005, Lisbon, Portugal, September 8.

KOUSHIK SEN, DARKO MARINOV, and GUL AGHA, "CUTE: A Concolic Unit Testing Engine for C." in 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05), pp. 263-272, Lisbon, Portugal, September 2005.