# Compositional Dynamic Test Generation

Patrice Godefroid

Microsoft Research

# Motivation

- Problem: automatic code-driven test generation
  - Given a sequential program with a set of input parameters, generate a set of tests that maximizes code coverage
- How? (1) Static test generation ([King76,…])
  - Static analysis to partition the program's input space
  - Ineffective whenever symbolic reasoning is not possible
    - which is frequent in practice…

Example:

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

Can't statically generate values for x and y that satisfy "x==hash(y)" !

# DART = Directed Automated Random Testing

- How? (2) Dynamic test generation
  - Run the program starting with some random inputs, gather symbolic constraints on inputs at conditional statements, use a constraint solver to generate new test inputs
  - Repeat the process until a specific program path or statement is reached (classic dynamic test generation [Korel90])
  - Or repeat the process to attempt to cover ALL feasible program paths (DART = systematic dyn. test gen. [PLDI'05])
    - detect crashes, assert violations, use runtime checkers (Purify,…)

Example:

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

Run 1: pick x and y randomly.
Run 2: keep same value for y but set x to hash(y), known from Run 1.
All program paths are now covered !

Dynamic is more powerful than static

# Compositionality = Key to Scalability

- Problem: executing all feasible paths does not scale !
- Idea: compositional dynamic test generation
  - use summaries of individual functions (arbitrary program blocks) like in interprocedural static analysis
  - If f calls g, test g separately, summarize the results, and use g's summary when testing f
  - A summary $\varphi(g)$ is a disjunction of path constraints expressed in terms of input preconditions and output postconditions:

    $\varphi(g) = \vee\, \varphi(w)$ with $\varphi(w) = pre(w) \wedge post(w)$
    expressed in terms of g's inputs and outputs
  - g's outputs are treated as symbolic inputs to a calling function f

# SMART = Scalable DART

- Unlike interprocedural static analysis:
  - Summaries may include information about concrete values (to allow partial symbolic reasoning)
  - Each summary needs to be grounded in some concrete execution (to guarantee that no false alarm is ever generated): here, "must" summaries, not "may" summaries !
  - Bottom-up strategy for computing summaries is problematic (generates too many spurious summaries and too few relevant summaries – see paper)
  - Top-down strategy to compute summaries on a demand-driven basis from concrete calling contexts: SMART algorithm
  - SMART = Systematic Modular Automated Random Testing
  - Same path coverage as DART but can be exponentially faster!
  - See paper…

# 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;
}
```

Program P={top,is_positive} has 2^N feasible whole-program paths

DART will perform 2^N runs

SMART will perform only 4 runs !

- 2 to compute the summary $\Phi = (x>0 \wedge ret=1) \vee (x=<0 \wedge ret=0)$ for function is_positive()
- 2 to execute both branches of (*), by solving the constraint $[(s[0]>0 \wedge ret_0=1) \vee (s[0]=<0 \wedge ret_0=0)] \wedge [(s[1]>0 \wedge ret_1=1) \vee (s[1]=<0 \wedge ret_1=0)] \wedge \ldots \wedge [(s[N-1]>0 \wedge ret_{N-1}=1) \vee (s[N-1]=<0 \wedge ret_{N-1}=0)] \wedge (ret_0+ret_1+\ldots+ret_{N-1} = 3)$

# Results

- Theorem: SMART provides same path coverage as DART
  - Corollary: same branch coverage, assertion violations,…
- Complexity: if b bounds the number of intraprocedural paths, number of runs by SMART is linear in b (while number of runs by DART can be exponential in b)
  - Similar to interprocedural static analysis, Hierarchical-FSM/Pushdown-system verification…
- Notes: arbitrary program blocks ok, recursion ok, concurrency is orthogonal (but arguably inherently non-compositional in general…)

# Conclusions

- DART is a promising new approach
  - Already detected hard-to-find bugs in several applications…
- Two main limitations: constraint solver + path explosion
- Here, drastic solution to path explosion !
  - compute symbolic test summaries that are grounded in concrete executions ("must") for compositional dynamic test generation
  - completely eliminates path explosion due to interprocedural (interblock) paths, by using formulas with lots of disjunctions
  - those formulas can be solved using existing constraint solvers
- Bottom-line: A SMART search is necessary to make the "DART approach" scalable to large programs !

# Back-up slides

# Example with Bounded Recursion

```
#define N 100
int s[N]; // N inputs
int rec_is_pos(int i) {
  if (i == N) return 0; //(**)
  if (s[i]>0)
     return 1+rec_is_pos(i+1);
  return rec_is_pos(i+1);
}
void top() {
  int cnt;
  cnt=rec_is_pos(0);
  if (cnt == 3) error(); //(*)
  return;
}
```

Program P={top,is_positive} has 2^N feasible whole-program paths (test (**) is input independent!)

DART will perform 2^N runs

SMART will perform only 4 runs !

- 2 to compute the summary $\Phi = (in>0 \wedge ret=1) \vee (in=<0 \wedge ret=0)$ in inner-most call to rec_is_pos()
- 2 to execute both branches of (*), by solving the recursive constraint $[(s[0]>0 \wedge ret_0=1+ret_1) \vee (s[0]=<0 \wedge ret_0=ret_1)]$ $\wedge \ldots \wedge [(s[N-1]>0 \wedge ret_{N-1}=1) \vee (s[N-1]=<0 \wedge ret_{N-1}=0)]$ $\wedge (ret_0 = 3)$

# Note on Unbounded Recursion

- Example: if there is no bound N in the previous example, program P={top,is_positive} has infinitely many feasible whole-program paths
- Thus DART and SMART (as is) do not terminate !
- For finite-state programs with unbounded recursion, use dynamic programming techniques as in interprocedural static analysis and pushdown system verification
  - Example:

```
int foo(int x) {
  if (x>0) return f(x)
  else return f(-x)
}
```

- Otherwise, use techniques for infinite-state program analysis (example: loop/stack invariants for unbounded inputs – see paper)