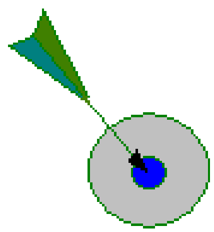


DART:
Directed Automated Random Testing

Patrice Godefroid — Bell Labs

Nils Klarlund — Bell Labs

Koushik Sen — UIUC

Motivation

- Software testing: "usually accounts for 50% of software development cost"
  - "Software failures cost $60 billion annually in the US alone"

  [Source: "The economic impacts of inadequate infrastructure for software testing", NIST, May 2002]

- Unit testing: applies to individual software components
  - Goal: "white-box" testing for corner cases, 100% code coverage
  - Unit testing is usually done by developers (not testers)

- Problem: in practice, unit testing is rarely done properly
  - Testing in isolation with manually-written test harness/driver code is too expensive, testing infrastructure for system testing is inadequate
  - Developers are busy, ("black-box") testing will be done later by testers…
  - Bottom-line: many bugs that should have been caught during unit testing remain undetected until field deployment (corner cases where severe reliability bugs hide)

- Idea: help automate unit testing by eliminating/reducing the need for writing manually test driver and harness code → DART

DART: Directed Automated Random Testing

1. Automated extraction of program interface from source code
2. Generation of test driver for random testing through the interface
3. Dynamic test generation to direct executions along alternative program paths

- Together: (1)+(2)+(3) = DART
- DART can detect program crashes and assertion violations.
- Any program that compiles can be run and tested this way:

  No need to write any test driver or harness code!

- (Pre- and post-conditions can be added to generated test-driver)

Example (C code)

```
int double(int x) {
    return 2 * x;
}

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

(1) Interface extraction:

- parameters of toplevel function
- external variables
- return values of external functions

(2) Generation of test driver for random testing:

```
main(){
    int tmp1 = randomInt();
    int tmp2 = randomInt();
    test_me(tmp1,tmp2);
}
```

Closed (self-executable) program that can be run

Problem: probability of reaching abort() is extremely low!

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
        abort(); /* error */
  }
}
```

Concrete Execution

x = 36, y = 99

Symbolic Execution

create symbolic variables x, y

Path Constraint

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete Execution

x = 36, y = 99,
z = 72

Symbolic Execution

create symbolic variables x, y

z = 2 * x

Path Constraint

DART Step (3): Directed Search

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
      if (y == x+10)
          abort(); /* error */
  }
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 36, y = 99, z = 72	create symbolic variables x, y z = 2 * x	2 * x != y

Solve: 2 * x == y

Solution: x = 1, y = 2

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 1, y = 2	create symbolic variables x, y	

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 1, y = 2, z = 2	create symbolic variables x, y z = 2 * x	

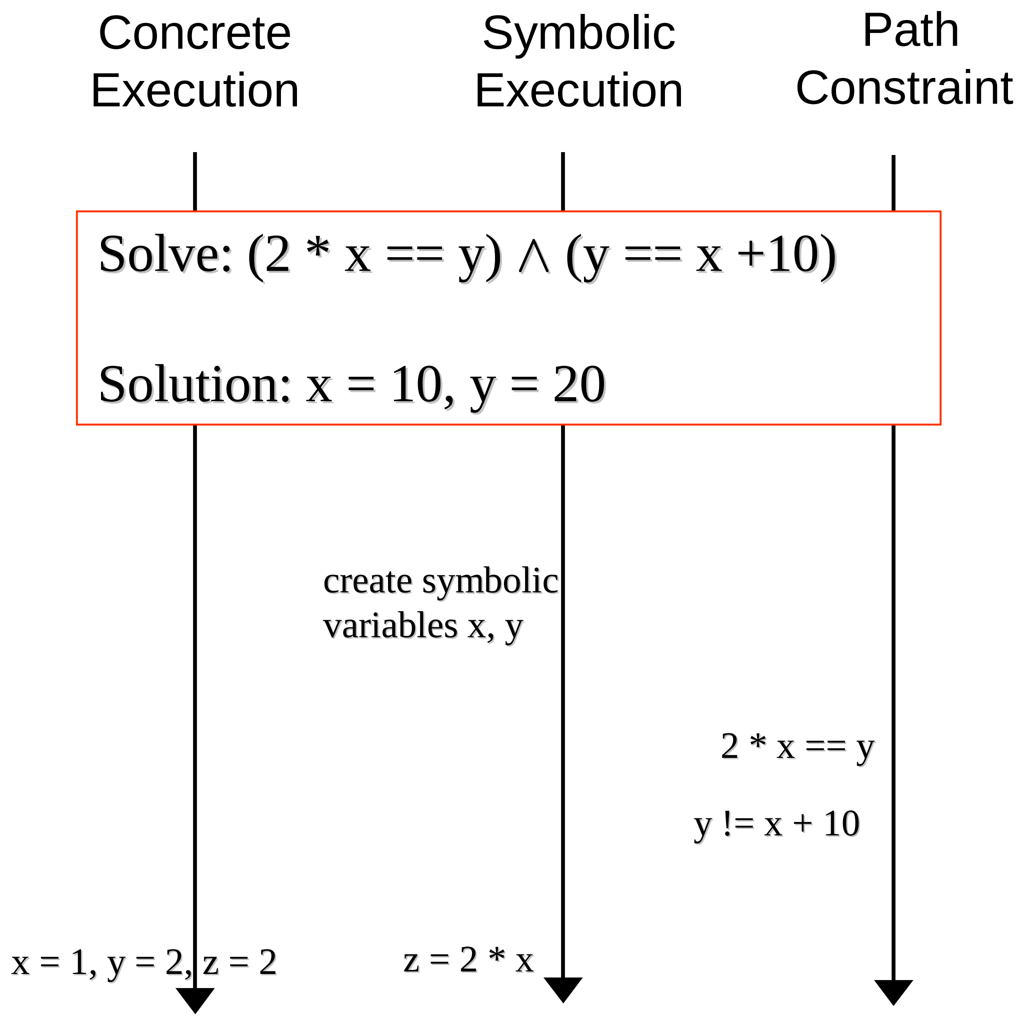
DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables x, y	
x = 1, y = 2, z = 2	z = 2 * x	2 * x == y

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```



DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y != x+10)
      abort(); /* error */
  }
}
```

Concrete Execution

Symbolic Execution

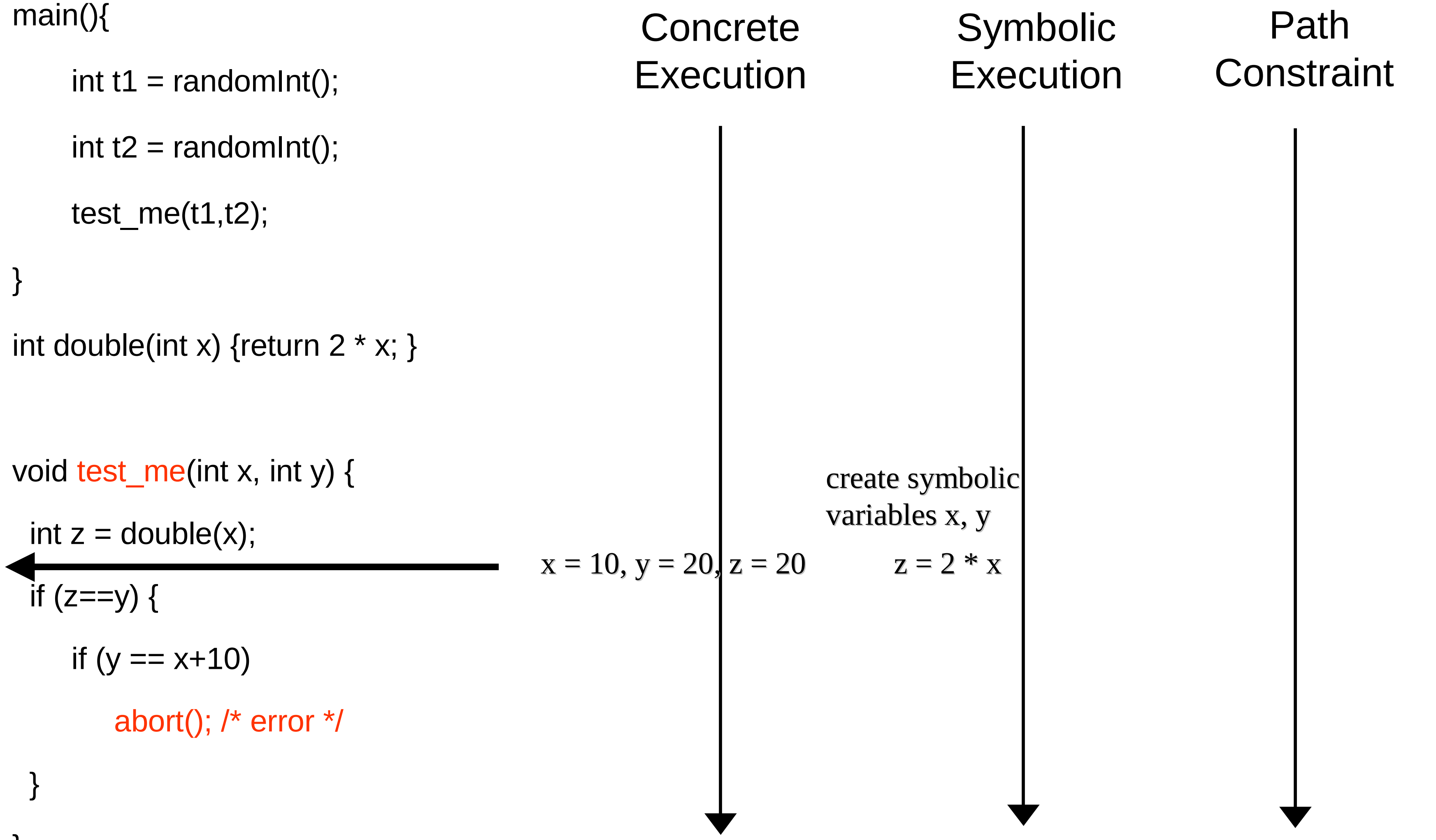
Path Constraint

x = 10, y = 20

create symbolic variables x, y

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```



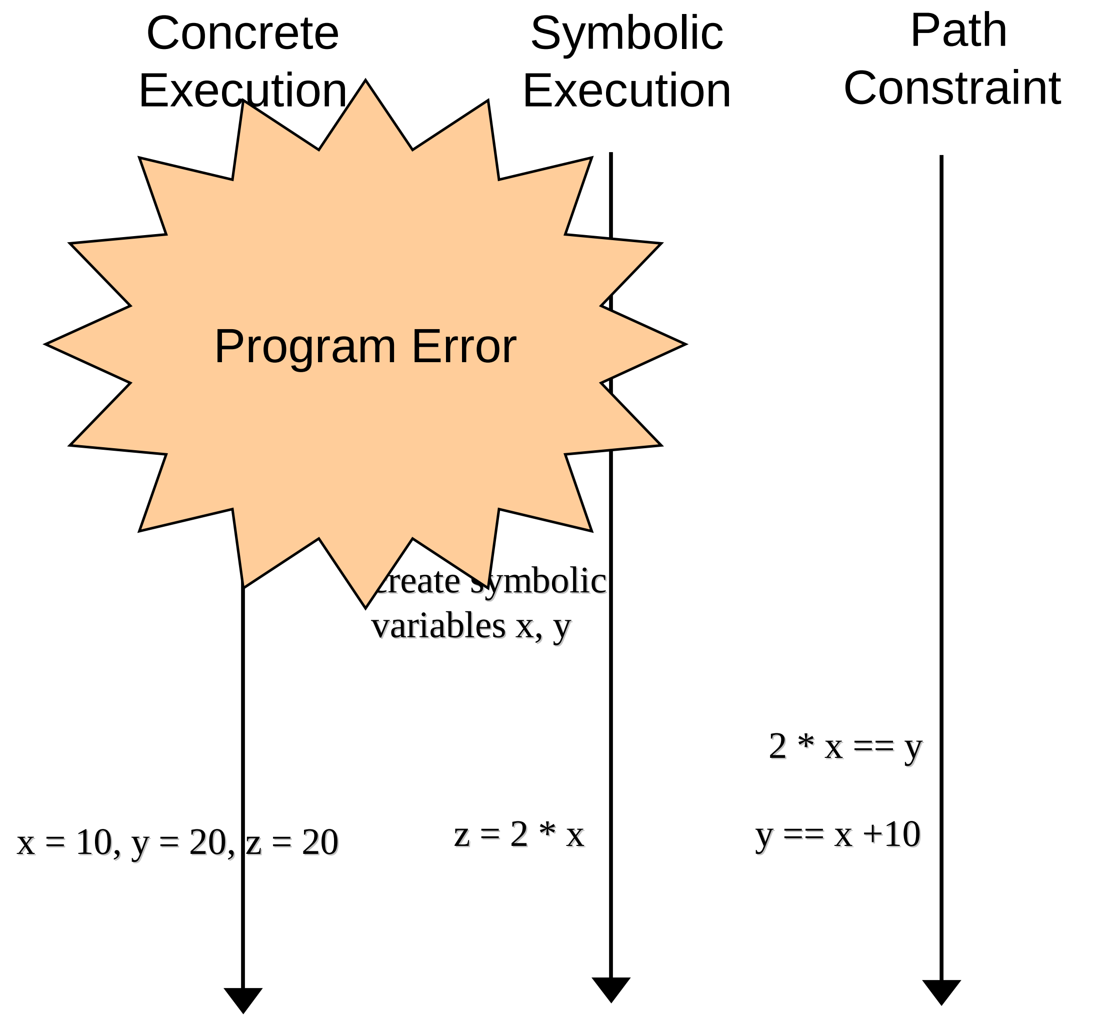
DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
       abort(); /* error */
  }
}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables x, y	
x = 10, y = 20, z = 20	z = 2 * x	2 * x == y

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```



Directed Search: Summary

- Dynamic test generation to direct executions along alternative program paths
  - collect symbolic constraints at branch points (whenever possible)
  - negate one constraint at a branch point to take other branch (say b)
  - call constraint solver with new path constraint to generate new test inputs
  - next execution driven by these new test inputs to take alternative branch b
  - check with dynamic instrumentation that branch b is indeed taken
- Repeat this process until all execution paths are covered
  - May never terminate!
- Significantly improves code coverage vs. pure random testing

Novelty: Simultaneous Concrete & Symbolic Executions

```
void foo(int x,int y){
    int z = x*x*x; /* could be z = h(x) */
    if (z == y) {
        abort(); /* error */
    }
}
```

- Assume we can reason about linear constraints only
- Initially x = 3 and y = 7 (randomly generated)
- Concrete z = 27, but symbolic z = x*x*x
  - Cannot handle symbolic value of z!
  - Stuck?

Novelty: Simultaneous Concrete & Symbolic Executions

```
void foo(int x,int y){
    int z = x*x*x; /* could be z = h(x) */
    if (z == y) {
        abort(); /* error */
    }
}
```

Replace symbolic expression by concrete value when symbolic expression becomes unmanageable (e.g. non-linear)

NOTE: whenever symbolic execution is stuck, static analysis becomes imprecise!

- Assume we can reason about linear constraints only
- Initially x = 3 and y = 7 (randomly generated)
- Concrete z = 27, but symbolic z = x*x*x
  - Cannot handle symbolic value of z!
  - Stuck?
  - NO! Use concrete value z = 27 and proceed…
- Take else branch with constraint 27 != y
- Solve 27 = y to take then branch
- Execute next run with x = 3 and y = 27
- DART finds the error!

Comparison with Static Analysis

```
1  foobar(int x, int y){
2    if (x*x*x > 0){
3      if (x>0 && y==10){
4          abort(); /* error */
5      }
6    } else {
7      if (x>0 && y==20){
8          abort(); /* error */
9      }
10   }
11 }
```

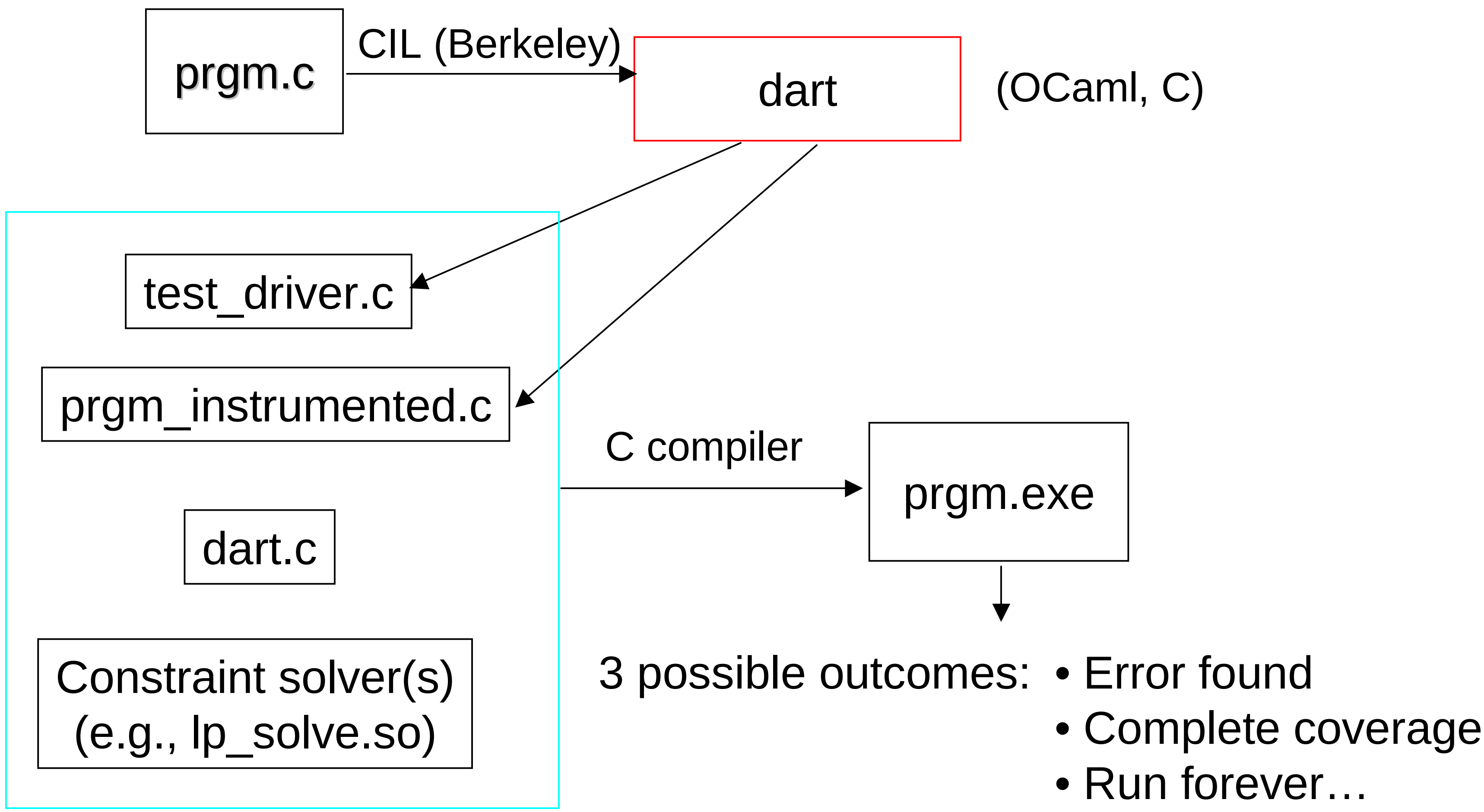
- Symbolic execution is stuck at line 2…
- Static analysis tools will conclude that both aborts may be reachable
  - "Sound" tools will report both, and thus one false alarm
  - "Unsound" tools will report "no bug found", and miss a bug
- Static-analysis-based test generation techniques are also helpless here…
- In contrast, DART finds the only error (line 4) with high probability
- Unlike static analysis, all bugs reported by DART are guaranteed to be sound

Other Advantages of Dynamic Analysis

```
struct foo { int i; char c; }

bar (struct foo *a) {
    if (a->c == 0) {
        *((char *)a + sizeof(int)) = 1;
        if (a->c != 0) {
            abort();
        }
    }
}
```

- Dealing with dynamic data is easier with concrete executions
- Due to limitations of alias analysis, static analysis tools cannot determine whether "a->c" has been rewritten
  - "the abort may be reachable"
- In contrast, DART finds the error easily (by solving the linear constraint a->c == 0)
- In summary, all bugs reported by DART are guaranteed to be sound!
- But DART may not terminate…

DART for C: Implementation Details



Experiments: NS Authentication Protocol

- Tested a C implementation of a security protocol (Needham-Schroeder) with a known attack
  - About 400 lines of C code; experiments on a Linux 800Mz P-III machine
  - DART takes less than 2 seconds (664 runs) to discover a (partial) attack, with an unconstrained (possibilistic) intruder model
  - DART takes 18 minutes (328,459 runs) to discover a (full) attack, with a realistic (Dolev-Yao) intruder model
  - DART found a new bug in this C implementation of Lowe's fix to the NS protocol (after 22 minutes of search; bug confirmed by the code's author)
- In contrast, a systematic state-space search of this program composed with a concurrent nondeterministic intruder model using VeriSoft (a sw model checker) does not find the attack

A Larger Application: oSIP

- Open Source SIP library (Session Initiation Protocol)
  - 30,000 lines of C code (version 2.0.9), 600 externally visible functions
- Results:
  - DART crashed 65% of the externally visible functions within 1000 runs
  - Most of these due to missing(?) NULL-checks for pointers…
  - Analysis of results for oSIP parser revealed a simple attack to crash it!

Attack: send a packet of size 2.5 MB (cygwin) with no 0 or "|" character

oSIP version 2.0.9 (August 2004)

```
Int osip_message_parse (osip_message_t * sip,
                        const char *buf)

{ [ … ]
  char *tmp;
  tmp = alloca (strlen (buf) + 2);
  osip_strncpy (tmp, buf, strlen (buf));
  osip_util_replace_all_lws (tmp);
[ etc. ]
```

alloca fails and returns NULL

crash!

oSIP version 2.2.0 (December 2004)

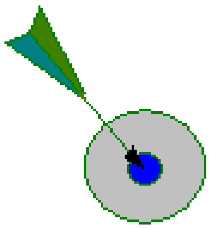
```
Int osip_message_parse (osip_message_t * sip,
                        const char *buf, size_t length)

{ [ … ]
  char *tmp;
  tmp = osip_malloc (length + 2);
  if (tmp==NULL) { [… print error msg and return –1; ] }
  osip_strncpy (tmp, buf, length);
  osip_util_replace_all_lws (tmp);
 [ etc. ]
```

Related Work

- Static analysis and automatic test generation based on static analysis: limited by symbolic execution technology (see above)
- Random testing (fuzz tools, etc.): poor coverage
- Dynamic test generation (Korel, Gupta-Mathur-Soffa, etc.)
  - Attempt to exercise a specific program
  - DART attempts to cover <u>all</u> executable program paths instead (like MC)
  - Also, DART handles function calls, unknown functions, exploits simultaneous concrete and symbolic executions, is sometimes complete (verification) and has run-time checks to detect incompleteness
  - DART is implemented for C and has been applied to large examples
- New: extension to deal with symbolic pointers [Sen et al., to appear in FSE'05]
- New: independent closely related work [Cadar-Engler, to appear in SPIN'05]

Conclusion

- DART = Directed Automated Random Testing



- Key strength/originality:
  - No manually-generated test driver required (fully automated)
    - As automated as static analysis but with higher precision
    - Starting point for testing process
  - No false alarms but may not terminate
  - Smarter than pure random testing (with directed search)
  - Can work around limitations of symbolic execution technology
    - Symbolic execution is an adjunct to concrete execution
    - Randomization helps where automated reasoning is difficult
  - Overall, complementary to static analysis…