Reliable and Productive Programming thru Formal Methods

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

URL: http://www.cs.utah.edu/~ganesh

Software is "eating" the world

- Quote by Marc Andreessen
  - https://www.wsj.com/artts tsicles/SB10001424053111903480904576512250915629460
- Software is everywhere
  - Its integrity is important for everything we do
    - From door locks to automobiles
- Software bugs extract a huge societal toll
  - Bugs lead to exploits
- In many areas, the complexity of programming is huge
  - Systems in this area cannot be designed without deep-testing methods
  - Example:
    None of today's microprocessors could be designed without formal methods

Example: How Intel verifies microprocessors

International Conference on Computer Aided Verification

CAV 2009: Computer Aided Verification pp 414-429 | Cite as

Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation

Authors | Authors and affiliations

Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, Armaghan Naik

Rajnish and Sudhindra did their MS degrees at Utah. Rajnish was my student.

Software correctness and productivity

Research done under “formal methods” aims to achieve much higher standards of correctness and productivity

- Deep-testing
  - Aims to test a piece of SW for all inputs
  - This is achieved by modeling software symbolically
- Rigorous (Formal) specifications
- Anticipated directions
  - Automated software synthesis

“Formal Methods” is a well-recognized term now

contributed articles

DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

S3 is just one of many AWS services that store and process data our customers have entrusted to us. To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks. There are many such algorithms in the literature, but combining them into a cohesive system is a challenge, as the algorithms must usually be modified to interact properly in a real-world system. In addition, we have found it necessary to invent algorithms of our own. We work hard to avoid unnecessary complexity, but the essential complexity of the task remains high.

Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching a service, we need to reach extremely high con-

Amazon's use of formal modeling of their S3 system

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

Why test systems for all inputs?

- We have to recount a story from the mid 1990’s

A little story about Primes

- What is the smallest prime?

A little story about Primes

- What is the smallest prime?
  - 2

    - 1 is not taken to be a Prime for several reasons

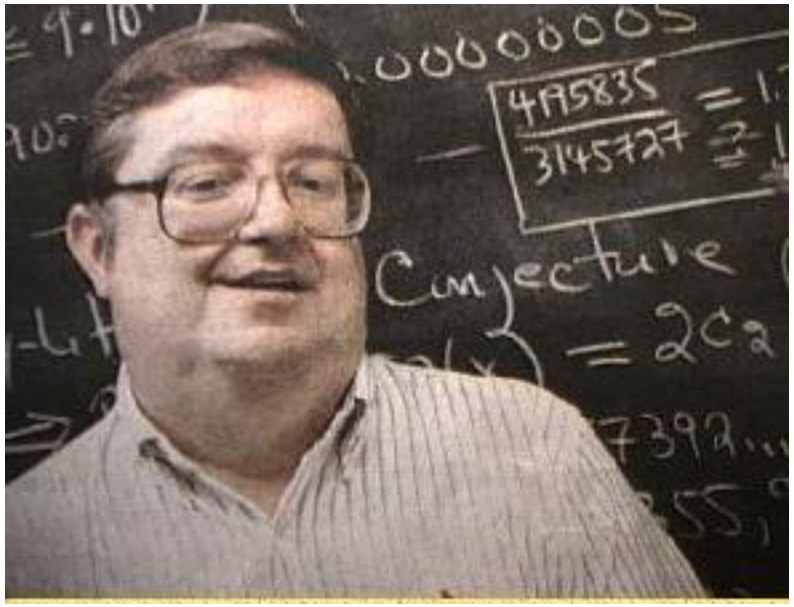
A little story about Primes

- There are twin primes
  - 2,3….5,7……11,13……17,19…..
  - Conjecture : Are there an infinite number of twin primes?
  - Open problem, but mathematicians are closing in
- Another curious fact:
  - The sums of reciprocals of twin primes converges to the so called Brun’s constant

Brun’s constant

- (1/3+1/5) + (1/5+1/7) + (1/11+1/13) + … comes to about
  - 1.90216054….

Enter Prof. Thomas Nicely



Enter Prof. Nicely

- Prof. Nicely was hoping to calculate a better approximation to Brun's constant using the (then) new Pentium processor (in 1994)
- To his utter amazement, he obtained really incorrect answers!
- After checking everything, he realized that Pentium could not divide correctly !!

Prof. Nicely's statements

> The [Pentium] co-processor is designed to give you 19 digits correct.... For it to give you only 10 is just utterly atrocious To get a result that poor from that co-processor is like having the transmission fall out of your Ford.
>
> — Thomas R. Nicely

Prof. Nicely's statements

> Usually mathematicians have to shoot somebody to get this much publicity.
>
> — Thomas R. Nicely

How Formal Methods came into HW Design: Intel FDIV

- A – B * (A/B) = 0 (math)
- A – B * (A/B) = 256 (Pentium)

- FROM: Dr. Thomas R. Nicely Professor of Mathematics Lynchburg College 1501 Lakeside Drive Lynchburg, Virginia
- TO: Whom it may concern RE: Bug in the Pentium FPU DATE: **30 October 1994**

- 4195835.0 - 3145727.0*(4195835.0/3145727.0)

What Intel did

- **Some degree of denial…**
- **Soon they bowed to the reality**
- **A massive recall of 0.5 Billion dollars worth of Pentium chips was issued**

Reason for the bug

- **Intel used aggressive optimizations in their division algorithm**
- **Predict the next quotient digit during division**
- **Used a table-lookup**
- **That table missed a few entries**
- **So the answers wer “mostly right” - and often terribly wrong!**

Reason for the bug

- **All companies designed their division units in “good faith”**

- **But they could not test it for all inputs; why?**

Reason for the bug

- **All companies designed their division units in “good faith”**
- **But they could not test it for all inputs; why?**
  - **Two 32-bit inputs to a divider results in 2^{64} input combinations**

What Intel did

- Some degree of denial…
- **Soon they bowed to the reality**
- **A massive recall of 0.5 Billion dollars worth of Pentium chips was issued**
- **Then they hired the best available Formal Methods talent**
  - **Randal Bryant and Carl Seger were among them**
  - **They built for Intel a method to verify arithmetic circuits for ALL inputs**

How do Formal Methods allow all inputs to be tested?

- **Do not "blindly" test all inputs in terms of bit combinations**
- **Rather, build a circuit model**
- **Feed Boolean variables as inputs!**
  - **Called "Symbolic Execution"**
- **Crank the division algorithm**
  - **This results in Boolean Expressions (instead of specific answers)**
  - **You can check that these expressions are mathematically correct (for all variable values)**

Needed a new data structure for Boolean functions

- Binary Decision Diagrams (BDD)
- A compact way to represent (and manipulate) large Boolean functions

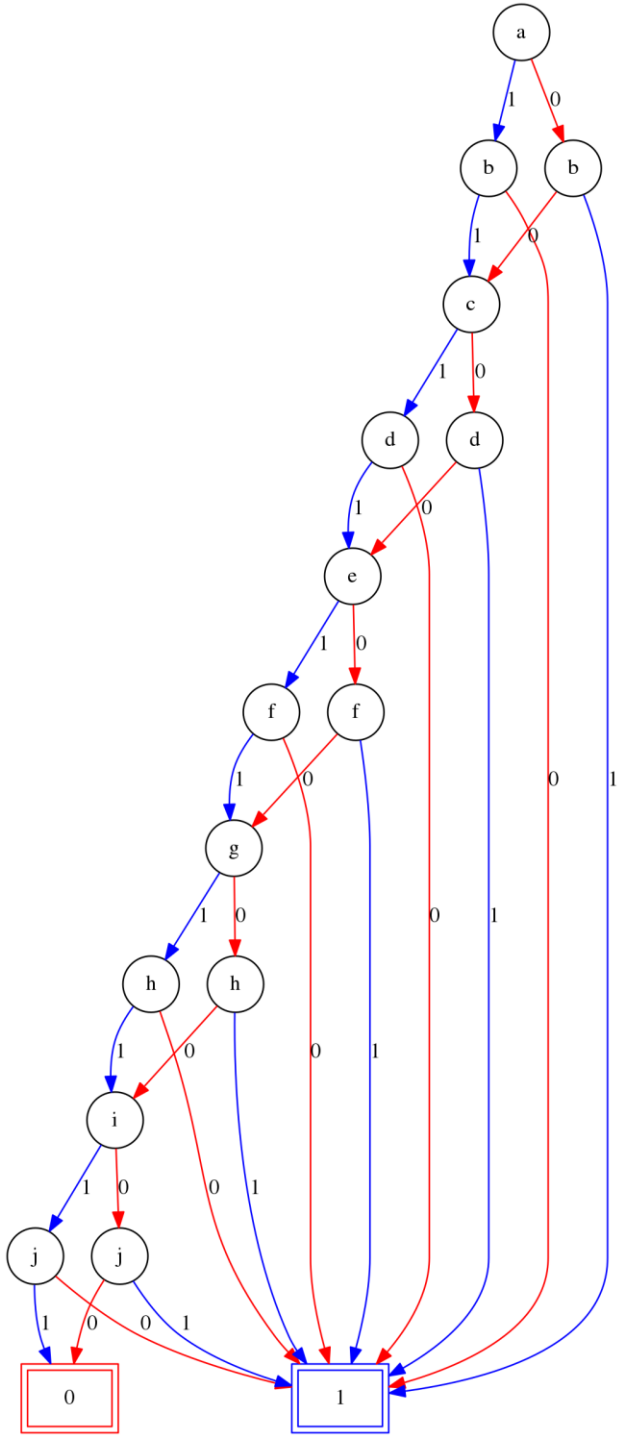
- Observation: one cannot represent a truth-table for a function of 64 inputs
- So we can't even represent a Boolean function of 64 inputs easily (let alone manipulate such functions)

- With BDDs, we can prove equality of Boolean functions!

BDD example

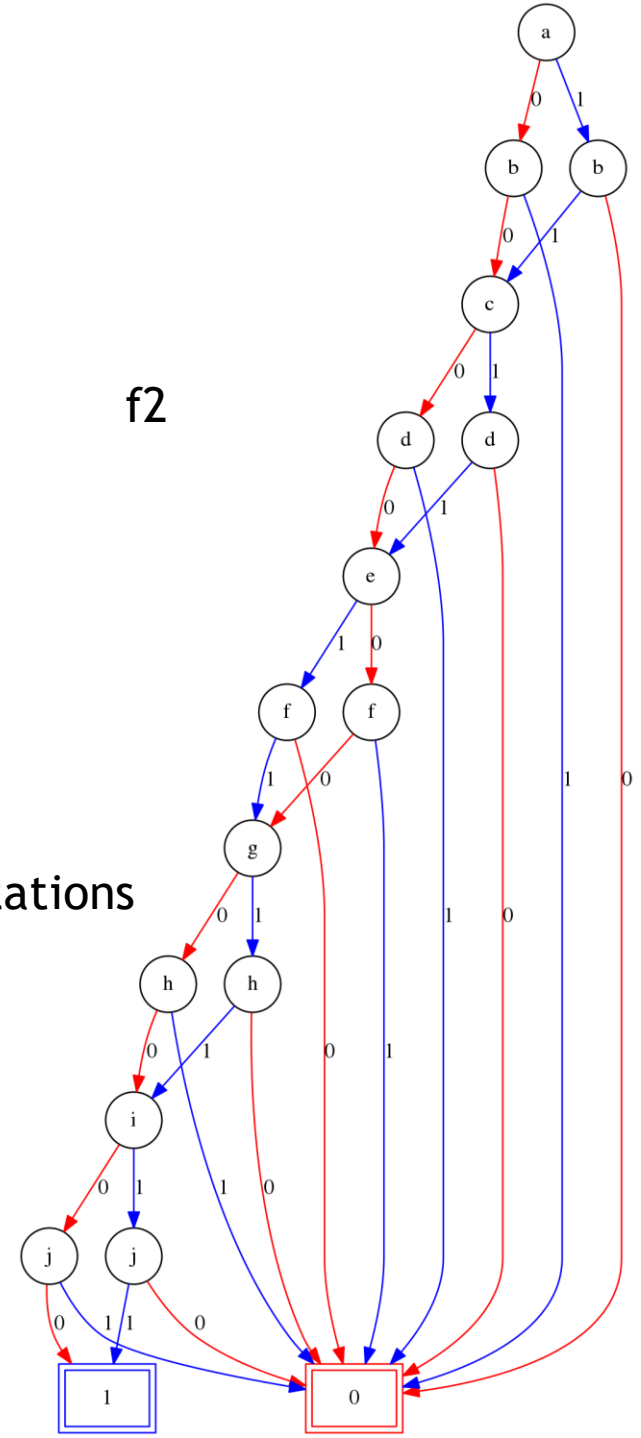
```
Var_Order : a b c d e f g h i j

f1 = (a <=> b) && (c <=> d)  && (e <=> f) && (g <=> h) && (i <=> j)

f2 = (a XOR b) || (c XOR d)  || (e XOR f) || (g XOR h) || (i XOR j)

Main_Exp : f2 <=> !f1
```

f1



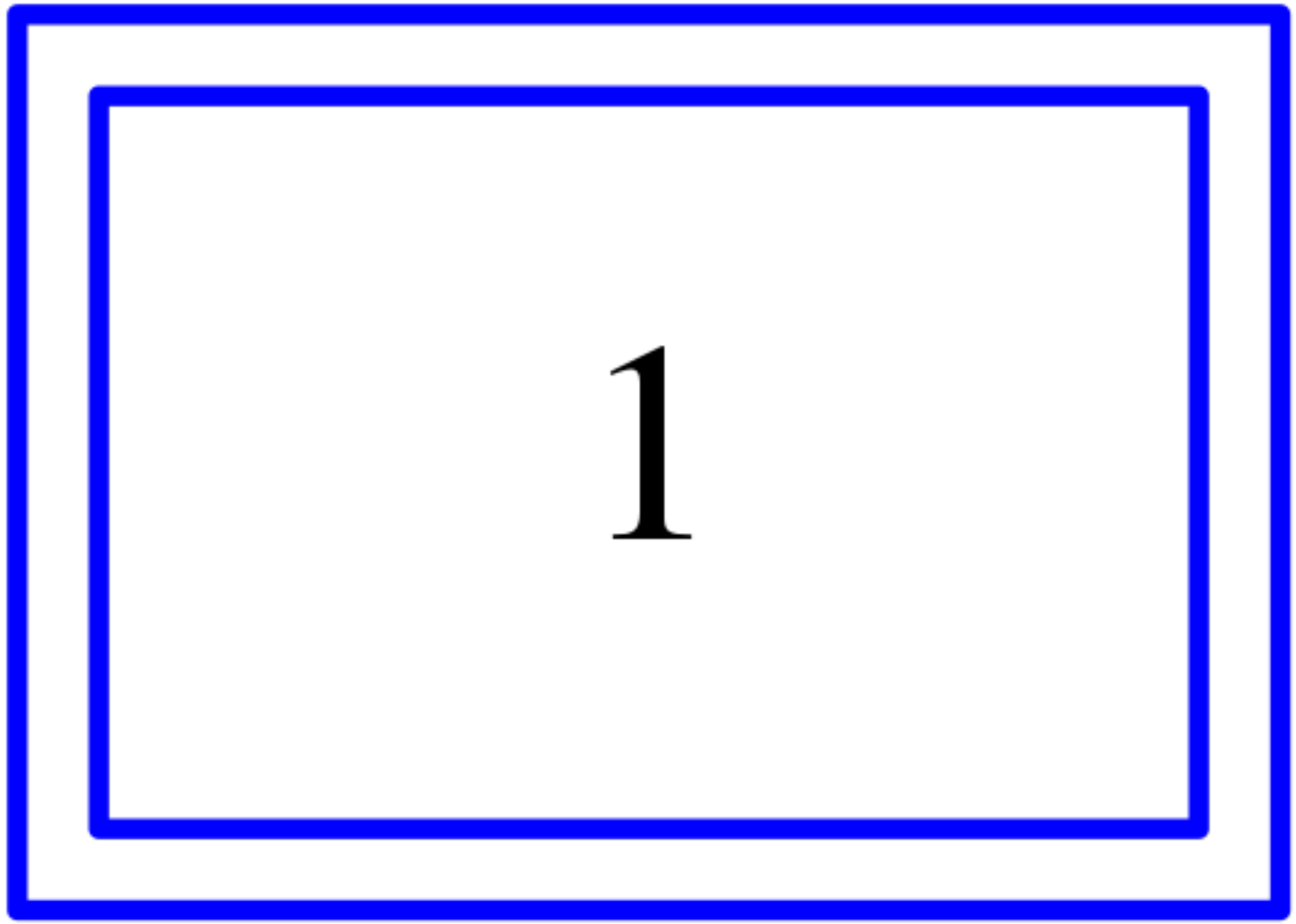
These functions have
Linearly-sized representations

f2



The tautology f1 ⇔ !f2 is established through “symbolic reasoning”

The BDD for f1 ⇔ !f2 is below (demo from http://formal.cs.utah.edu:8080/pbl/BDD.php)



Now, “Formal” has rescued HW debugging

- **Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation,** Roope Kaivola and team **(CAV 2009)**
  - **Essentially, they let-go simulation / testing teams**
  - **Coverage through “formal” was measurably superior**
- 2013 *Microsoft* Research Verified Software Milestone *Award*
- This is why microprocessors, with all their complexity, largely work

Power of SAT-Solving is Behind Modern FM

- **SAT = Boolean Satisfiability**
- **Supposed to be NP-complete**
- **Yet runs surprisingly fast**
- **Power of SAT in your browser (demo)**
  - https://msoos.github.io/cryptominisat_web/

Power of SMT-solving Powers Modern FM

- SMT = Satisfiability Modulo Theories
  - Like “SAT” except for a mix of theories
    - Int, List, String, Array, Functions, …
- Uses SAT underneath
  - Use abstraction
  - Convert to SAT
  - If no success, detail abstraction
  - Repeat

Illustration of SMT : ”Send More Money”

- **Find assignments to s,e,n,d,m,o,r,y such that this addition holds (trivial solutions not accepted)**

```
    s e n d +
    m o r e
----------------
  m o n e y
```

Try using Z3Py (demo)

Installing and running Z3

- http://www.cs.toronto.edu/~victorn/tutorials/z3_SAT_2019/index.html
- https://ericpony.github.io/z3py-tutorial/guide-examples.htm

- Also see "SAT/SMT by Example" by Dmitry Yurichev: https://sat-smt.codes/SAT_SMT_by_example.pdf

Run the provided sudoku.py (next few slides) under python after doing
pip install z3solver

Sudoku encoding from https://ericpony.github.io/z3py-tutorial/guide-examples.htm

```
from z3 import *

# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]

# each cell contains a value in {1, ..., 9}
cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
             for i in range(9) for j in range(9) ]

# each row contains a digit at most once
rows_c   = [ Distinct(X[i]) for i in range(9) ]

# each column contains a digit at most once
cols_c   = [ Distinct([ X[i][j] for i in range(9) ])
             for j in range(9) ]
```

X is a list of lists containing free variables x_i_j of type Int where i and j are between 1 and 9. Notice that X itself has rows that are indexed 0 thru 8. The lists within X are also indexed 0 thru 8.

The cells_c constraint conjoins all the x_i_j variables to be in the indicated ranges of values (I.e. between 1 and 9)

The row_c constraint asserts that all the list of x_i_j variables are distinct per row of X

The cols_c constraint asserts that all the list of x_i_j variables are distinct per column of X where the columns of X are formed by slicing the nine X vectors along their similar indices

Sudoku encoding from https://ericpony.github.io/z3py-tutorial/guide-examples.htm

```
# each 3x3 square contains a digit at most once
sq_c     = [ Distinct([ X[3*i0 + i][3*j0 + j]
                        for i in range(3) for j in range(3) ])
             for i0 in range(3) for j0 in range(3)

sudoku_c = cells_c + rows_c + cols_c + sq_c
# sudoku instance, we use '0' for empty cells
instance = ((0,0,0,0,9,4,0,3,0),
            (0,0,0,5,1,0,0,0,7),
            (0,8,9,0,0,0,0,4,0),
            (0,0,0,0,0,0,2,0,8),
            (0,6,0,2,0,1,0,5,0),
            (1,0,2,0,0,0,0,0,0),
            (0,7,0,0,0,0,5,2,0),
            (9,0,0,0,6,5,0,0,0),
            (0,4,0,9,7,0,0,0,0))
```

The sq_c constraint further enforces the embedded "squares" within X are also comprised of distinct variables

Sudoku constraints are a collection of the aforesaid constraints

We now make-up a tuple of 9-tuples

This template will allow us to introduce a suitable set of constraints as in the next slide

Sudoku encoding from https://ericpony.github.io/z3py-tutorial/guide-examples.htm

```
instance_c = [ If(instance[i][j] == 0,
                  True,
                  X[i][j] == instance[i][j])
               for i in range(9) for j in range(9) ]
```

This now allows us to pick-up the non-zero items of the tuple of 9-tuples and force those values. For the "0" positions, we set the constraint as True. Those can be assigned by the solver!

```
s = Solver()
s.add(sudoku_c + instance_c)
if s.check() == sat:
    m = s.model()
    r = [ [ m.evaluate(X[i][j]) for j in range(9) ]
          for i in range(9) ]
    print_matrix(r)
else:
    print("failed to solve")
```

We now make a solver instance and add the constraints to it. If the solver's constraints are SAT, we ask for a model – and a unique model is obtained thanks to the "sudoku constraints"

While Sudoku is NP-complete for n^2 x n^2 grids comprised of n x n blocks, these versions solve pretty fast!

Power of SMT-solving Powers Modern FM

- **SMT solvers power many FM engines in**
  - **Compiler analysis (safe compilation)**
  - **Security**
  - **Memory access overlap detection**
  - **Synthesizing solutions (constraint solving)**
  - **Spread-sheets (Flash-Fill feature)**

Illustration of the power of SMT

- The KLEE Symbolic + Concrete ("Concolic") testing tool

Demo-1 : a hard-to-test program under conventional testing and then KLEE

```c
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
  if (x < RAND_MAX/2) {
    if (x > RAND_MAX/2 - 3) {
         assert(0);
      }
  }
  else
    return(1);
}

int main() {
  int a, i;

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
  rare(a);
  #endif


  #ifdef PRINTON
  #ifndef KLEEON
  printf("randmax = %d\n", RAND_MAX);
  for (i=0; i < 10000000; i++) {
  a = rand(); //% 100000000;
  rare(a);
  }
  #endif
  #endif
}
```

```
klee@f88364af576a:~/klee_src/examples/hardToHit$ gcc -DINITON -DPRINTON -I ../../include hardToHit
klee@f88364af576a:~/klee_src/examples/hardToHit$ ./a.out
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ !!;!!
./a.out;./a.out
randmax = 2147483647
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ !!;!!
./a.out;./a.out;./a.out;./a.out
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ !!;!!;!!;!!
./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a
out;./a.out;./a.out;./a.out
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "BOSS, CAN I SHIP THE CODE NOW ?!?!"
```

This assertion
Is hard to hit!

Demo-1 : a hard-to-test program under conventional testing and then KLEE

```c
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
  if (x < RAND_MAX/2) {
    if (x > RAND_MAX/2 - 3) {
          assert(0);
      }
  }
  else
    return(1);
}

int main() {
  int a, i;

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
  rare(a);
  #endif


  #ifdef PRINTON
  #ifndef KLEEON
  printf("randmax = %d\n", RAND_MAX);
  for (i=0; i < 10000000; i++) {
  a = rand(); //% 100000000;
  rare(a);
  }
  #endif
  #endif
}
```

```
klee@f88364af576a:~/klee_src/examples/hardToHit$ gcc -DINITON -DPRINTON -I ../../include hardToHit
klee@f88364af576a:~/klee_src/examples/hardToHit$ ./a.out
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ !!;!!
./a.out;./a.out
randmax = 2147483647
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ !!;!!
./a.out;./a.out;./a.out;./a.out
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ !!;!!;!!;!!
./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./a.out;./
out;./a.out;./a.out;./a.out
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
randmax = 2147483647
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "BOSS, CAN I SHIP THE CODE NOW ?!?!"
```

We run 10M iterations many times without any luck!

Would you put your child / father / sister … in an airplane that contains this code, hoping the assert is never hit?

If not, what would you, as a responsible engineer, do today for **short but important** pieces of codes??

Enter

KLEE !!

```c
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
  if (x < RAND_MAX/2) {
    if (x > RAND_MAX/2 - 3) {
        assert(0);
      }
  }
  else
    return(1);
}

int main() {
  int a, i;

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
  rare(a);
  #endif


  #ifdef PRINTON
  #ifndef KLEEON
  printf("randmax = %d\n", RAND_MAX);
  for (i=0; i < 10000000; i++) {
  a = rand(); //% 100000000;
  rare(a);
  }
  #endif
  #endif
}
```

```
klee@f88364af576a:~/klee_src/examples/hardToHit$ clang -DKLEEON -I ../../include -emit-llvm -g -c -O0 -Xclang -disable-O0-optnone hardToHit.c
hardToHit.c:18:1: warning: control may reach end of non-void function [-Wreturn-type]
}
^
1 warning generated.
klee@f88364af576a:~/klee_src/examples/hardToHit$ klee hardToHit.bc
KLEE: output directory is "/home/klee/klee_src/examples/hardToHit/klee-out-3"
KLEE: Using STP solver backend
KLEE: ERROR: hardToHit.c:13: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 27
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "see we already hit the assert in concolic"
see we already hit the assert in concolic
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "now to see the tests synthesized"
now to see the tests synthesized
klee@f88364af576a:~/klee_src/examples/hardToHit$ ls -ld klee-last
lrwxrwxrwx 1 klee klee 49 Feb 10 18:34 klee-last -> /home/klee/klee_src/examples/hardToHit/klee-out-3
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "this directory has the tests"
this directory has the tests
klee@f88364af576a:~/klee_src/examples/hardToHit$ ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['hardToHit.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
klee@f88364af576a:~/klee_src/examples/hardToHit$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['hardToHit.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\xfd\xff\xff?'
object 0: hex : 0xfdffff3f
object 0: int : 1073741821
object 0: uint: 1073741821
object 0: text: ...?
klee@f88364af576a:~/klee_src/examples/hardToHit$ ls klee-last
assembly.ll  messages.txt  run.stats             test000001.kquery  test000002.ktest  warnings.txt
info         run.istats    test000001.assert.err  test000001.ktest   test000003.ktest
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "ah ha , test 000001 is the one that hit assert"
ah ha , test 000001 is the one that hit assert
```

Compiling the same C program using the KLEE Symbolic Executor causes the tool to synthesize test-inputs automatically to cover all the branches. This Generates a bunch of test cases. The test-tool also records which test hit the error!

Enter

KLEE !!

```c
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
  if (x < RAND_MAX/2) {
    if (x > RAND_MAX/2 - 3) {
        assert(0);
      }
  }
  else
    return(1);
}

int main() {
  int a, i;

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
  rare(a);
  #endif


  #ifdef PRINTON
  #ifndef KLEEON
  printf("randmax = %d\n", RAND_MAX);
  for (i=0; i < 10000000; i++) {
  a = rand(); //% 100000000;
  rare(a);
  }
  #endif
  #endif
}
```

Magnified view of the same test, below.
Test000001 hit the error !

```
klee@f88364af576a:~/klee_src/examples/hardToHit$ gcc -DKLEEON -DPRINTON -I ../../include -L /home/klee/klee_build/lib hardToHit.c -lkleeRuntest
klee@f88364af576a:~/klee_src/examples/hardToHit$ KTEST_FILE=klee-last/test000002.ktest ./a.out
klee@f88364af576a:~/klee_src/examples/hardToHit$ KTEST_FILE=klee-last/test000001.ktest ./a.out
a.out: hardToHit.c:13: rare: Assertion `0' failed.
Aborted
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "BOOM - we hit the assert "
BOOM - we hit the assert
klee@f88364af576a:~/klee_src/examples/hardToHit$
```

Demo-2
Binary
Search

Looks
OK?

Ship it?!

```c
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
  printf("searching for %d in:\n[", target);
    for (int i=0; i < size-1; i++) {
      printf("%d, ", arr[i]);
    }
    printf("%d]\n", arr[size-1]);
}

int binary_search(int arr[], int size, int target) {
  #ifdef PRINTON
    print_data(arr, size, target);
  #endif
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            low = mid + 1;
        }
        if (arr[mid] > target) {
            high = mid - 1;
        }
    }
    return -1;
}
```

```c
int main() {

  int a[4]; // was a[10]

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
  klee_assume(a[0] <= a[1]);
  klee_assume(a[1] <= a[2]);
  klee_assume(a[2] <= a[3]);

  klee_make_symbolic(&x, sizeof(x), "x");
  #endif

  #ifdef INITON
  a[0]=rand()%30;
  a[1]=a[0]+rand()%30;
  a[2]=a[1]+rand()%30;
  a[3]=a[2]+rand()%30;
  #endif

  int result = binary_search(a, 4, x); // was 10

  #ifdef PRINTON
  printf("result = %d\n", result);
  #endif
  // check correctness

  if (result != -1) {
    assert(a[result] == x);
  } else {
    // if result == -1, then we didn't find it.
rray
    for (int i = 0; i < 3; i++) { // was 10

      assert(a[i] != x);
    }
  }
  return 1;
}
```

Demo-2
Binary
Search

Looks
OK?

```c
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
  printf("searching for %d in:\n[", target);
    for (int i=0; i < size-1; i++) {
      printf("%d, ", arr[i]);
    }
    printf("%d]\n", arr[size-1]);
}

int binary_search(int arr[], int size, int target) {
  #ifdef PRINTON
    print_data(arr, size, target);
  #endif
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            low = mid + 1;
        }
        if (arr[mid] > target) {
            high = mid - 1;
        }
    }
    return -1;
}
```

```c
int main() {

  int a[4]; // was a[10]

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
```

```
klee@f88364af576a:~/klee_src/examples/binsrch$ gcc -DINITON -DPRINTON -I ../../include binsrch.c
klee@f88364af576a:~/klee_src/examples/binsrch$ ./a.out
searching for -336381299 in:
[13, 29, 56, 81]
result = -1
klee@f88364af576a:~/klee_src/examples/binsrch$ clang -DKLEEON -I ../../include -emit-llvm -g -c -O0 -Xclang -disable-O0-optnone binsrch.c
klee@f88364af576a:~/klee_src/examples/binsrch$ klee binsrch.bc
KLEE: output directory is "/home/klee/klee_src/examples/binsrch/klee-out-18"
KLEE: Using STP solver backend
KLEE: WARNING: undefined reference to function: printf

KLEE: done: total instructions = 638
KLEE: done: completed paths = 9
KLEE: done: generated tests = 9
```

Demo-2 Binary Search

Looks OK?

Ship it?!

```c
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
  printf("searching for %d in:\n[", target);
    for (int i=0; i < size-1; i++) {
      printf("%d, ", arr[i]);
    }
    printf("%d]\n", arr[size-1]);
}

int binary_search(int arr[], int size, int target) {
  #ifdef PRINTON
    print_data(arr, size, target);
  #endif
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            low = mid + 1;
        }
        if (arr[mid] > target) {
            high = mid - 1;
        }
    }
    return -1;
}
```

```c
int main() {

  int a[4]; // was a[10]

  #ifdef KLEEON
  klee_make_symbolic(&a, sizeof(a), "a");
  klee_assume(a[0] <= a[1]);
  klee_assume(a[1] <= a[2]);
  klee_assume(a[2] <= a[3]);

  klee_make_symbolic(&x, sizeof(x), "x");
  #endif

  #ifdef INITON
  a[0]=rand()%30;
  a[1]=a[0]+rand()%30;
  a[2]=a[1]+rand()%30;
  a[3]=a[2]+rand()%30;
  #endif

  int result = binary_search(a, 4, x); // was 10

  #ifdef PRINTON
  printf("result = %d\n", result);
  #endif
  // check correctness

  if (result != -1) {
    assert(a[result] == x);
  } else {
    // if result == -1, then we didn't find it.
rray
    for (int i = 0; i < 3; i++) { // was 10

      assert(a[i] != x);
    }
  }
  return 1;
}
```

```
KLEE: done: generated tests = 9
klee@f88364af576a:~/klee_src/examples/binsrch$ ls klee-last
assembly.ll  messages.txt  run.stats         test000002.ktest  test000004.ktest  test000006.ktest  test000008.ktest  warnings.txt
info         run.istats    test000001.ktest  test000003.ktest  test000005.ktest  test000007.ktest  test000009.ktest
klee@f88364af576a:~/klee_src/examples/binsrch$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['binsrch.bc']
num objects: 2
object 0: name: 'a'
object 0: size: 16
object 0: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
object 0: hex : 0x00000000000000000000000000000000
object 0: text: ................
object 1: name: 'x'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x00'
object 1: hex : 0x00000000
object 1: int : 0
object 1: uint: 0
object 1: text: ....
klee@f88364af576a:~/klee_src/examples/binsrch$ ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['binsrch.bc']
num objects: 2
object 0: name: 'a'
object 0: size: 16
object 0: data: b'\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x01\x00\x00\x00\x01'
object 0: hex : 0x00000000000000010000000100000001
object 0: text: ................
object 1: name: 'x'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x00'
object 1: hex : 0x00000000
object 1: int : 0
object 1: uint: 0
object 1: text: ....
klee@f88364af576a:~/klee_src/examples/binsrch$ ktest-tool klee-last/test000009.ktest
ktest file : 'klee-last/test000009.ktest'
args       : ['binsrch.bc']
num objects: 2
object 0: name: 'a'
object 0: size: 16
object 0: data: b'\x00\x00\x00\x006\x00\x00\x00\xfe\x00\x00\x00\x00\x00\x00\x03'
object 0: hex : 0x0000000036000000fe00000000000003
object 0: text: ....6...........
object 1: name: 'x'
object 1: size: 4
object 1: data: b'\x00\x01\x00\x00'
object 1: hex : 0x00010000
object 1: int : 256
object 1: uint: 256
object 1: text: ....
klee@f88364af576a:~/klee_src/examples/binsrch$
```

Demo-2 Binary Search Looks OK. Closer to shipping.

What was achieved?
* tests that cover every path

* default C-level checks such as array out of bounds, null de reference

i.e. FEWER BUGS

```
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
  printf("searching for %d in:\n[", target);
```

```
klee@f88364af576a:~/klee_src/examples/binsrch$ ktest-tool klee-last/test000009.ktest
ktest file : 'klee-last/test000009.ktest'
args       : ['binsrch.bc']
num objects: 2
object 0: name: 'a'
object 0: size: 16
object 0: data: b'\x00\x00\x00\x006\x00\x00\x00\xfe\x00\x00\x00\x00\x00\x00\x03'
object 0: hex : 0x0000000036000000fe00000000000003
object 0: text: ....6...........
object 1: name: 'x'
object 1: size: 4
object 1: data: b'\x00\x01\x00\x00'
object 1: hex : 0x00010000
object 1: int : 256
object 1: uint: 256
object 1: text: ....
klee@f88364af576a:~/klee_src/examples/binsrch$ gcc -DKLEEON -DPRINTON -I ../../include -L /home/klee/klee_build/lib binsrch.c -lkleeRuntest
klee@f88364af576a:~/klee_src/examples/binsrch$ KTEST_FILE=klee-last/test000001.ktest ./a.out
searching for 0 in:
[0, 0, 0, 0]
result = 1
klee@f88364af576a:~/klee_src/examples/binsrch$ KTEST_FILE=klee-last/test000002.ktest ./a.out
searching for 0 in:
[0, 16777216, 16777216, 16777216]
result = 0
klee@f88364af576a:~/klee_src/examples/binsrch$ KTEST_FILE=klee-last/test000009.ktest ./a.out
searching for 256 in:
[0, 54, 254, 50331648]
result = -1
klee@f88364af576a:~/klee_src/examples/binsrch$
```

Take-away: The symbolic tester automatically produces inputs (initial array values and the item to be searched by the Binary Search program) so that all branches are covered.

What does KLEE do? (Cadar's slides)

KLEE

[OSDI 2008, Best Paper Award]

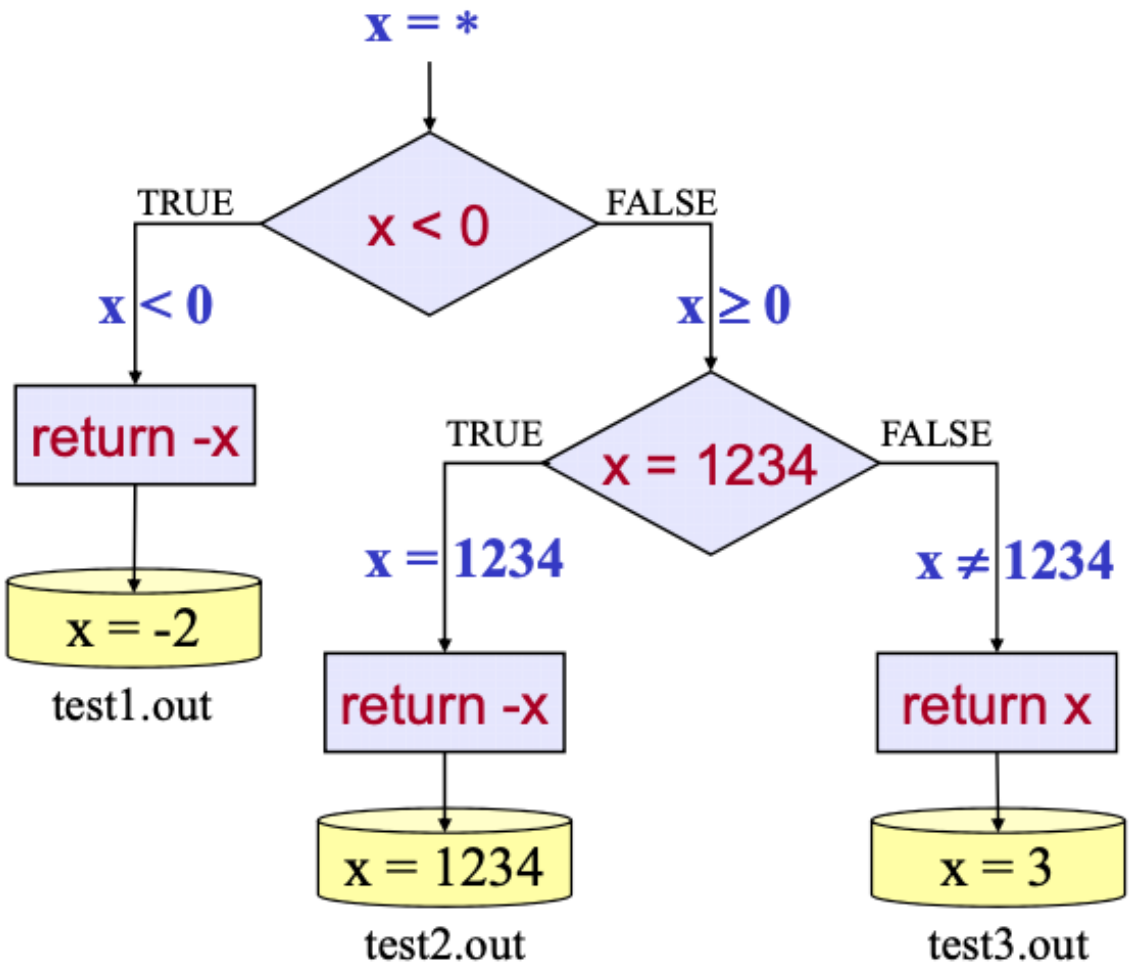
- Based on symbolic execution and constraint solving techniques

- Automatically generates high coverage test suites
  - Over 90% on average on ~160 user-level apps
- Finds deep bugs in complex systems programs
  - Including higher-level correctness ones

How does KLEE work? (Cadar's slides)

Toy Example

```
int bad_abs(int x)
{
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
```

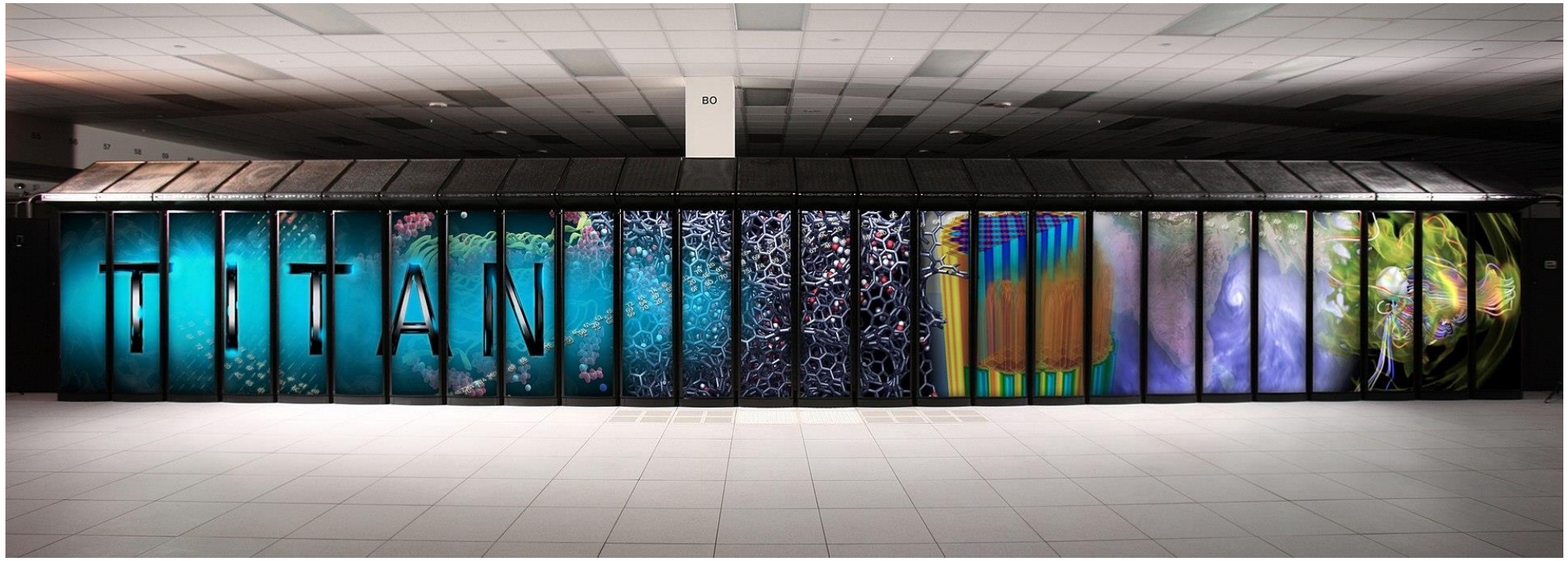


Formal Synthesis: Another fruit of FV research

- At the forefront of automating programming
- Correct-by-construction programs from higher level specifications
- Often programs are generated by generalizing the provided examples
  - E.g. the FlashFill feature of Excel works this way!

Overview of High Performance Computing

HPC powers scientific discoveries through **computational methods.** ***Our research helps make HPC software less buggy***



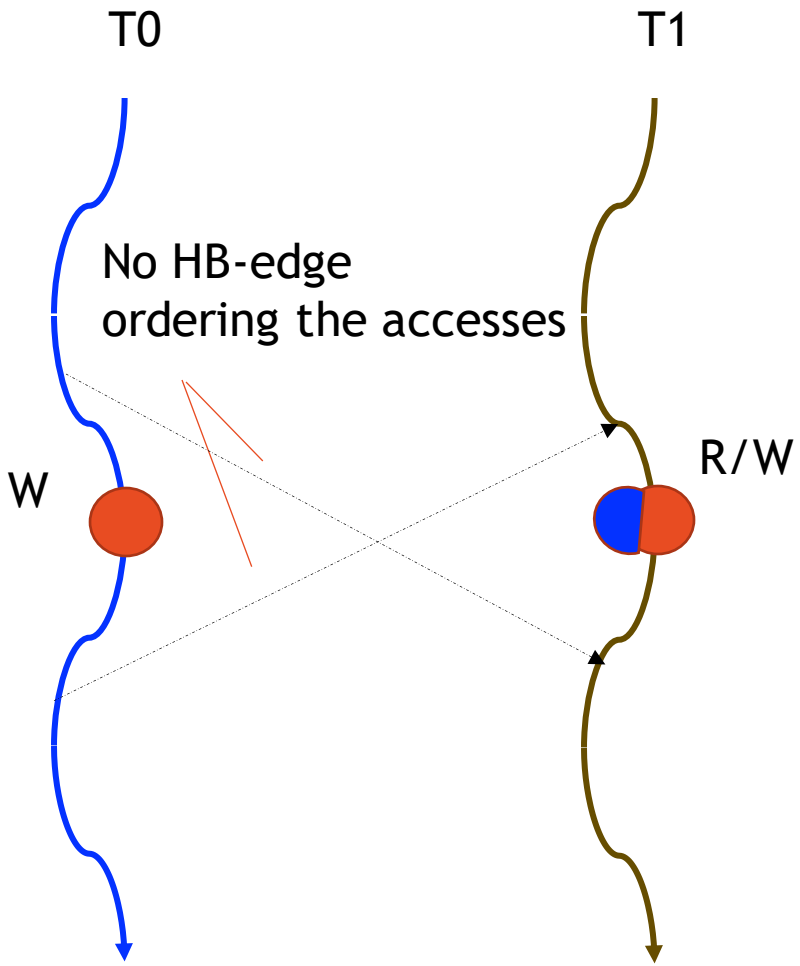
Our Correctness Projects in HPC

- Dynamic analysis tools for PThreads/MPI programs
- Symbolic verification methods for GPU programs
  - Adaptation of KLEE for GPUs
- Floating-point precision tuning tools
- Data Race Checking of OpenMP Programs
  - Archer tool released and in use within LLNL
- Reproducibility (trusting compilation) of compilations
  - FLiT tool released and within use within LLNL
- System Resilience: analysis and protection

Focused study: Data Race Checking

- Definition of data races
- Consequences of races : often overlooked
  - A real-world software failure due to an innocuous-looking race
- **Our contributions :**
  - **Two tools:**
    - **Archer -- in Production use at LLNL (IPDPS'16)**
    - **Sword -- getting to the state of Archer in about a year (IPDPS'18)**
- **Use of Formal Methods:**
  - Clear definition of data races
  - Design of the race detection algorithm that provides some guarantees

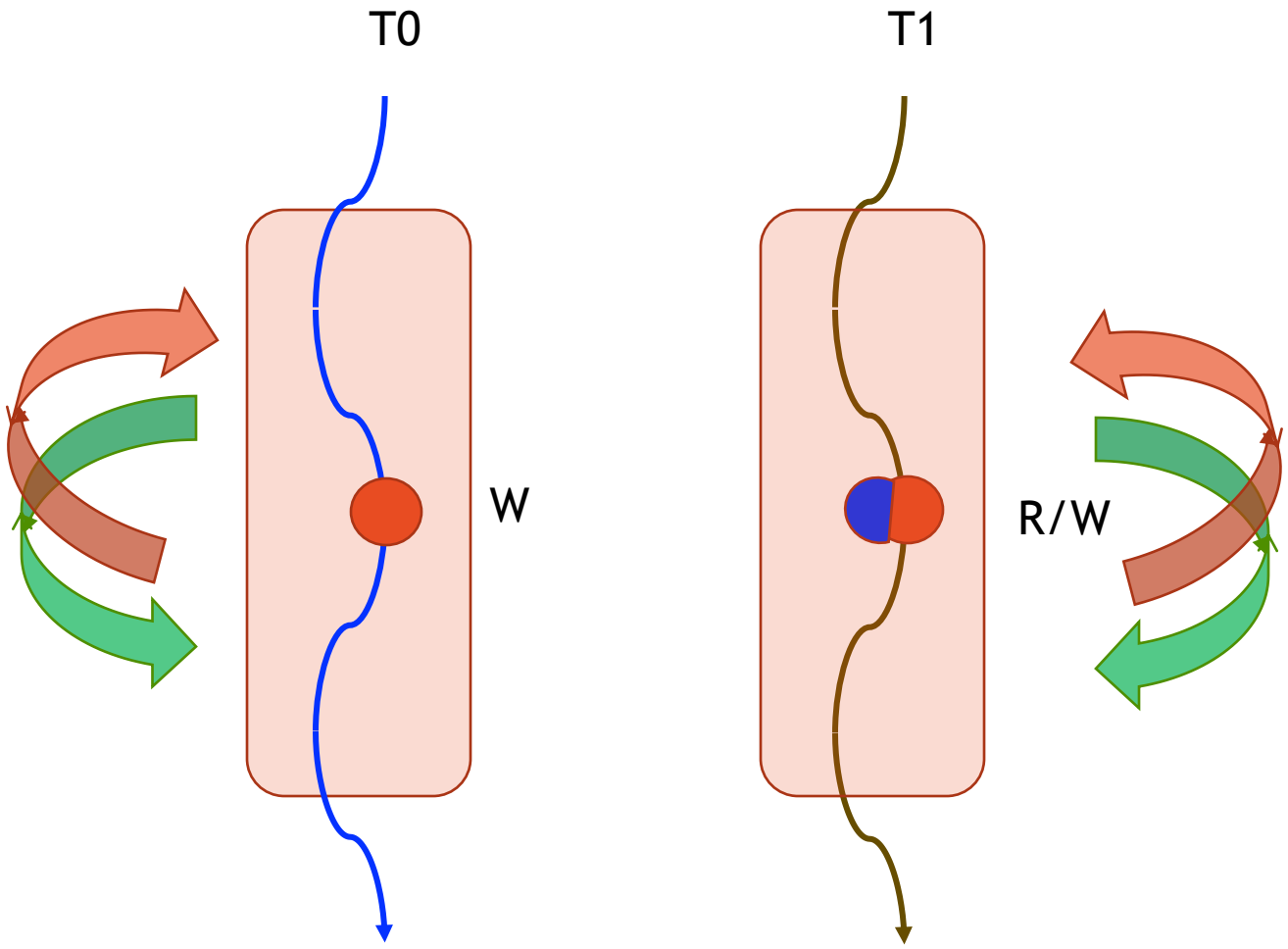
Definition of a data race



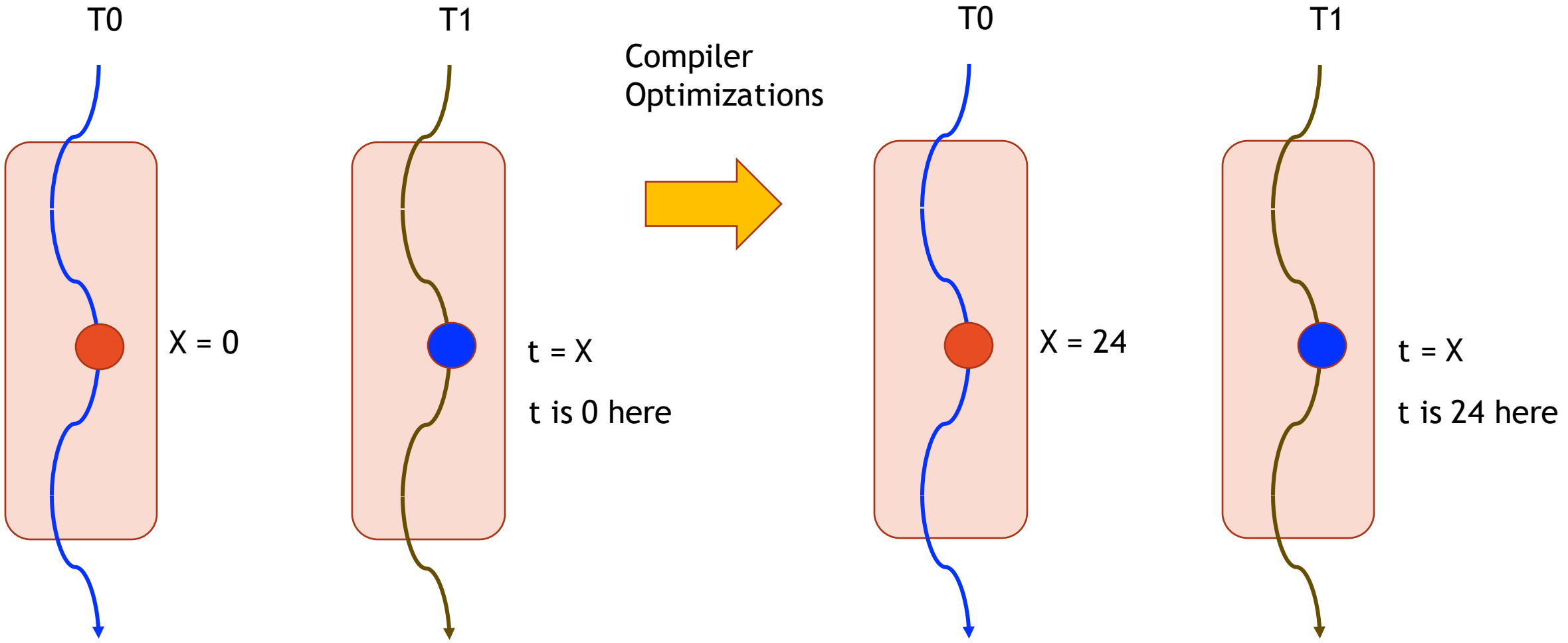
Two concurrent accesses,
one of which is a write
without any ”HB edge”

HB means ***happens-before*** ordering [Lamport]

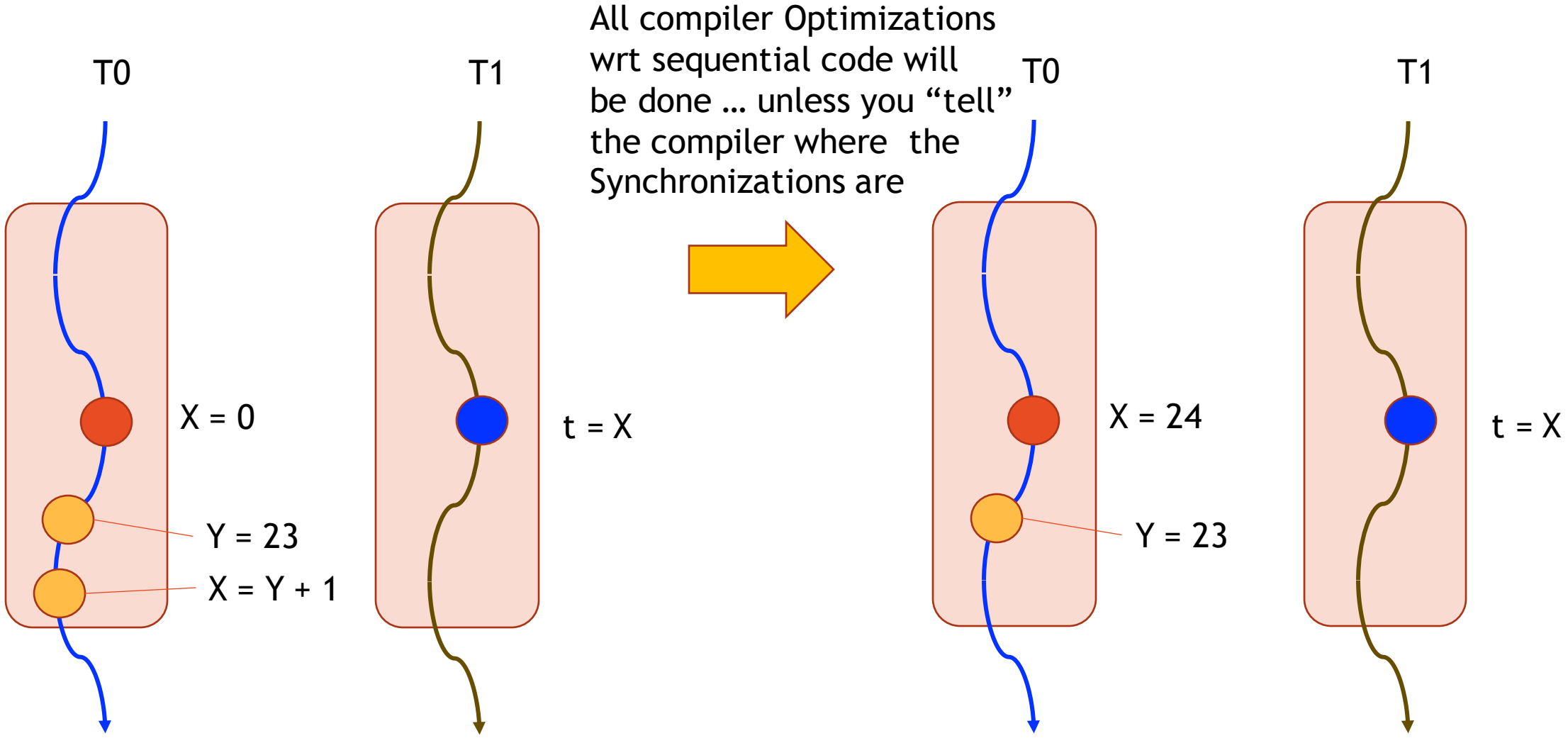
Why problematic: Compiler optimizations!



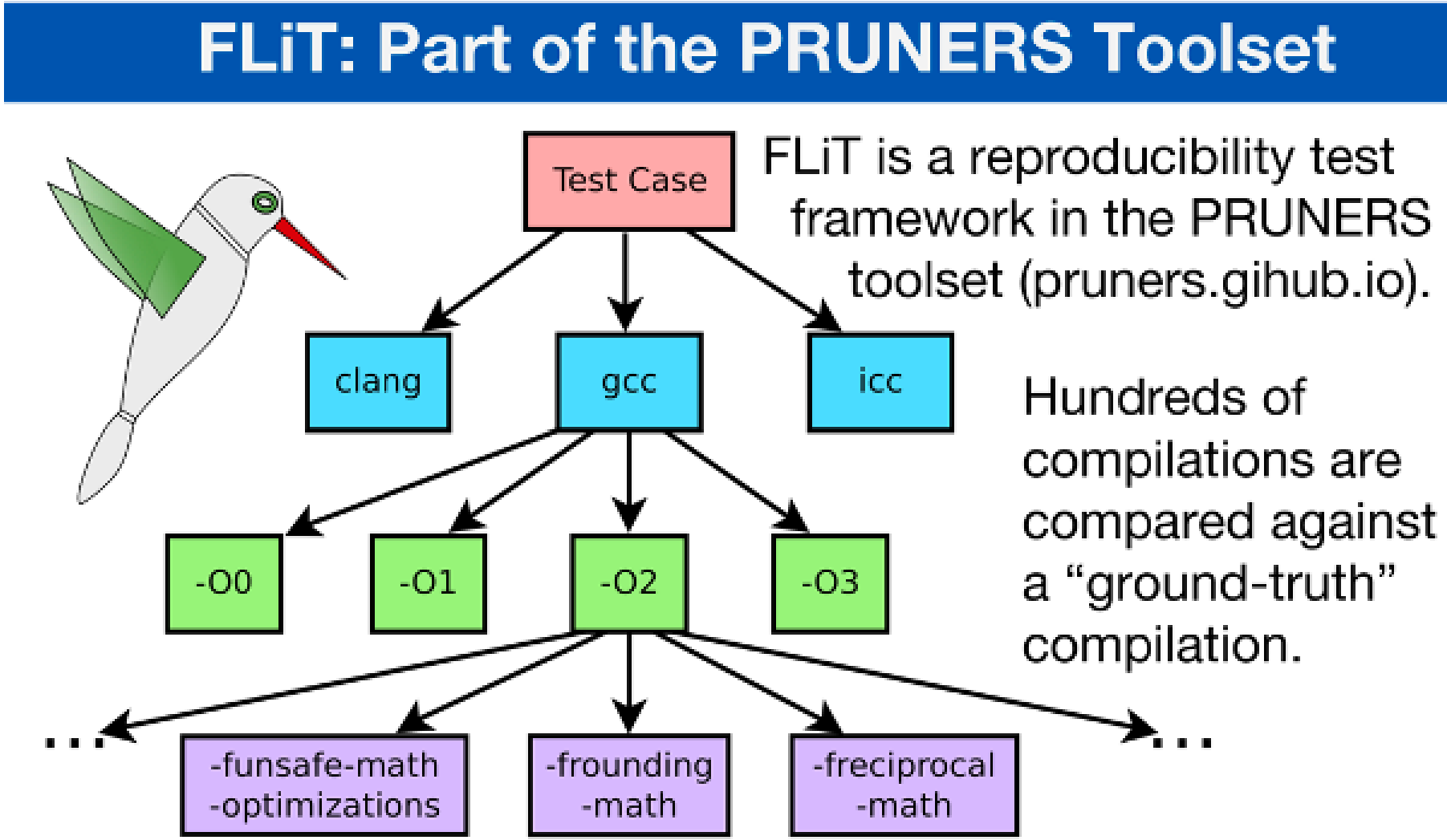
How can this happen after optimization?



Why problematic after optimization



Glimpse of FLiT tool for compiler testing



Concluding Remarks

- Manual testing of code is inadequate
- Formal testing finds deep-seated bugs
- Formal verification tools can prove (in some cases) the absence of bugs
  - Even otherwise, the exercise of formalizing systems allows one to produce unambiguous specifications
- Formal Synthesis tools have started democratizing programming
- Formal methods talent is in very high demand
- If you take CS 5110 in Spring 2022, you can experience some of these state-of-the-art tools, and also do an exciting project
- Case in point: a project in Spring 2021 was conducted by an engineer who took back the technology to his workplace
- You too can get good at formal methods!

```
dog, cat, mouse = Ints('dog cat mouse')
solve(dog >= 1,   # at least one dog
      cat >= 1,   # at least one cat
      mouse >= 1, # at least one mouse
      # we want to buy 100 animals
      dog + cat + mouse == 100,
      # We have 100 dollars (10000 cents):
      #   dogs cost 15 dollars (1500 cents),
      #   cats cost 1 dollar (100 cents), and
      #   mice cost 25 cents
      1500 * dog + 100 * cat + 25 * mouse == 10000)
```

```
# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]
```

```
# each cell contains a value in {1, ..., 9}
cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
             for i in range(9) for j in range(9) ]
```

```
# each row contains a digit at most once
rows_c  = [ Distinct(X[i]) for i in range(9) ]
```