Symbolic Methods for Verifying Software

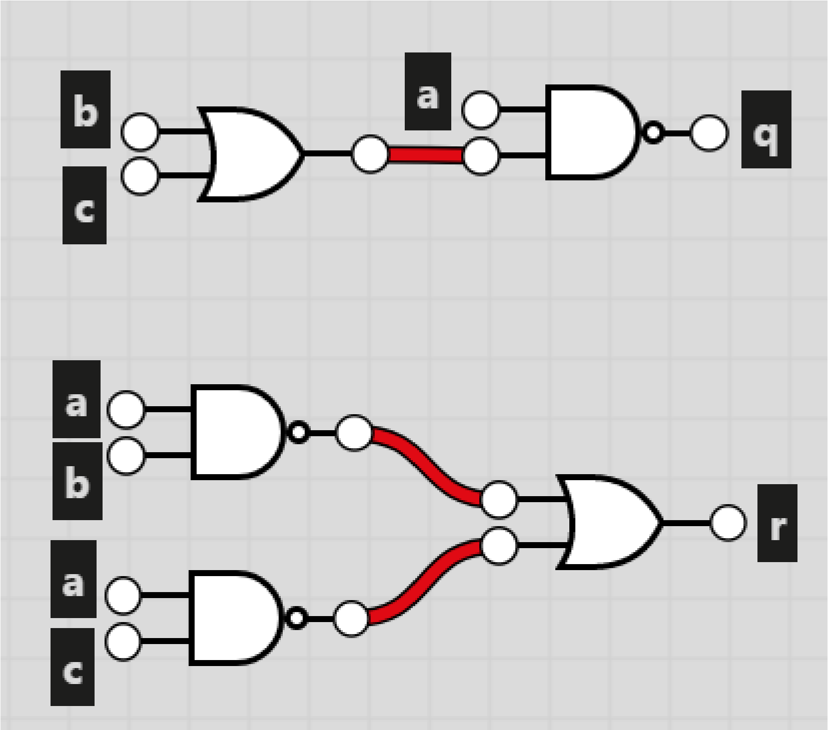
V&T

Roderick Bloem

IAIK – Graz University of Technology
Roderick.Bloem@iaik.tugraz.at

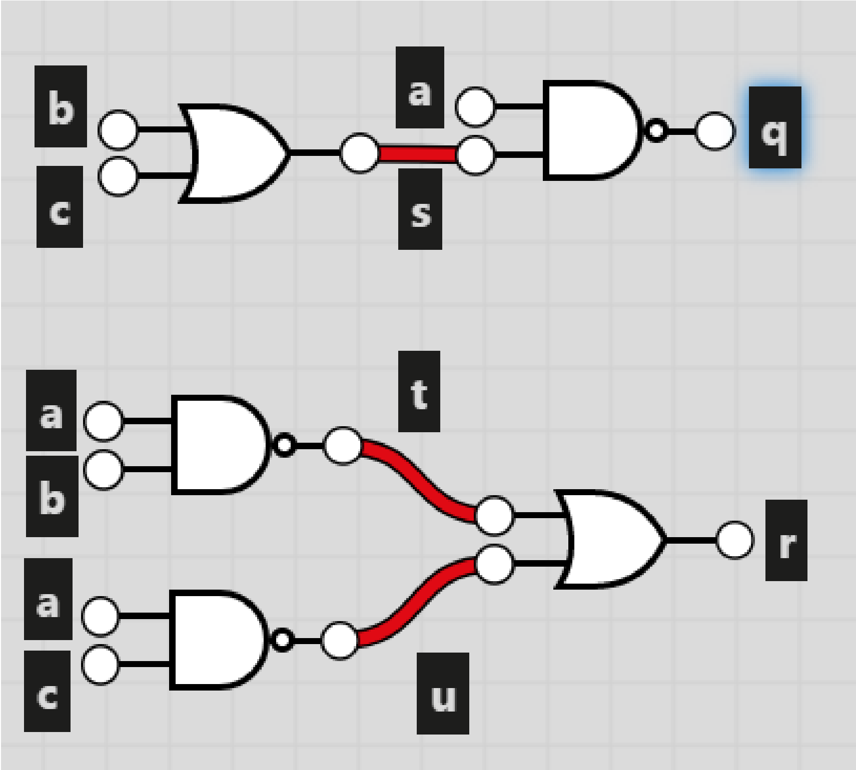
Note to me: see code under Code/Cbmc

Circuit Equivalence



https://logic.ly/demo

Circuit Equivalence



$\Phi = (s \leftrightarrow b \vee c) \wedge (q \leftrightarrow a \wedge s).$

$\Psi = (t \leftrightarrow a \wedge b) \wedge u \leftrightarrow (a \wedge c) \wedge (r \leftrightarrow t \vee u).$

Circuits are different iff following is satisfiable

$$\Phi \wedge \Psi \wedge (q \neq r)$$

Z3

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(declare-const s Bool)
(declare-const t Bool)
(declare-const u Bool)

(assert (= s (or b c)))
(assert (= q (and a s)))

(assert (= t (and a b)))
(assert (= u (and a c)))
(assert (= r (or t u)))

(assert (not (= q r)))

(check-sat)
(get-model)
```

https://rise4fun.com/Z3

Circuit Equivalence

- *Combinational* circuits (no memory elements): Use **Tseitin transformation**
  - Give each wire a name
  - Use standard formula for each gate
  - conjoin formulas
- Note: linear construction
- More complicated for *sequential* circuits (with memory)
  - model checking using a SAT solver, interpolation

This is a Bad Idea

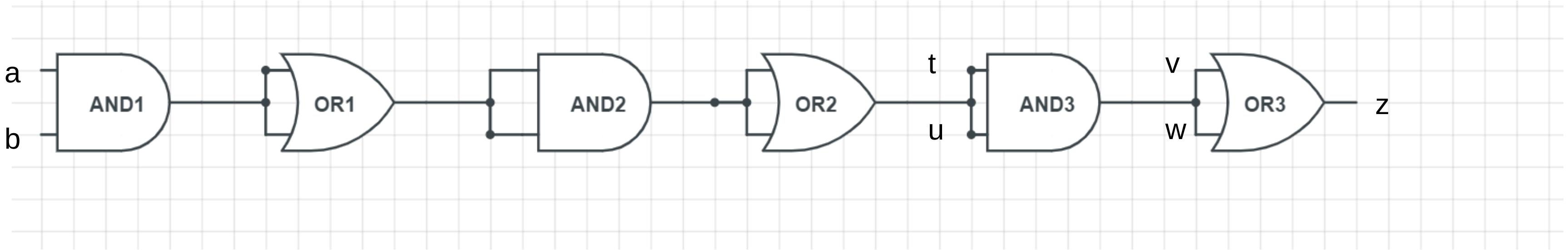
Exponential blowup

$z = x \vee y$ [substitute x and y]

$z = (v \wedge w) \vee (v \wedge w)$ [substitute v, w]

$z = \big((t \vee u) \wedge (t \vee u)\big) \vee ((t \vee u) \wedge (t \vee u))$ [now t,u]

etc etc



Verification Condition

Given a Program P, a **verification condition** is a formula ϕ such that

(ϕ is satisfiable) implies (P is buggy).

For the circuit example, the verification condition is $\Phi \wedge \Psi \wedge q \neq r$

Note: we have extra variables besides a, b, q, u, but these are not a problem

From Circuits to Software

Find out if the assertion can be violated

```
Boolean a, b;
if(a){
  if(!b)
    assert(false);
}
```

How do I get here?

ϕ?

Assertion reached iff ϕ satisfiable.

From Circuits to Software

Find out if the assertion can be violated

```
Boolean a, b;
if(a){
  if(!b)
    assert(false);
}
```

How do I get here?

$$\phi = a \land \neg b$$

Assertion reached iff ϕ satisfiable.

Satisfying assignment = input to reach assertion

Adding Assignments

```
Boolean a, b;
if(a){
  a = (a&&b);
  if(!a)
    assert(false);
}
```

```
Boolean a0, b0, a1;
if(a0){
  a1 = (a0&&b0);
  if(!a1)
    assert(false);
}
```

Single Static Assignment (SSA)

Let $\phi =$

Assertion reached iff ϕ satisfiable

Adding Assignments

```
Boolean a, b;
if(a){
  a = (a&&b);
  if(!a)
    assert(false);
}
```

```
Boolean a0, b0, a1;
if(a0){
  a1 = (a0&&b0);
  if(!a1)
    assert(false);
}
```

Single Static Assignment (SSA)

Let $\phi = \mathrm{a0} \land (a1 \leftrightarrow a0 \land b0) \land \neg a1$.
Assertion reached iff ϕ satisfiable

Adding Arithmetic

```
int a, b, c;
if(a != 0){
  c = (a + b);
  if(c > 0)
    assert(false);
}
```

Let's pretend ints have four bits

a != 0

Adding Arithmetic

```
int a, b, c;
if(a != 0){
  c = (a + b);
  if(c > 0)
    assert(false);
}
```

Let‘s pretend ints have four bits

c > 0

Adding Arithmetic

```
int a, b, c;
if(a != 0){
  c = (a + b);
  if(c > 0)
    assert(false);
}
```

Let‘s pretend ints have four bits

c = a + b

Adding Arithmetic

```
int a, b, c;
if(a != 0){
  c = (a + b);
  if(c > 0)
    assert(false);
}
```

Let's pretends `int`s are 4 bits: a_3, a_2, a_1, a_0

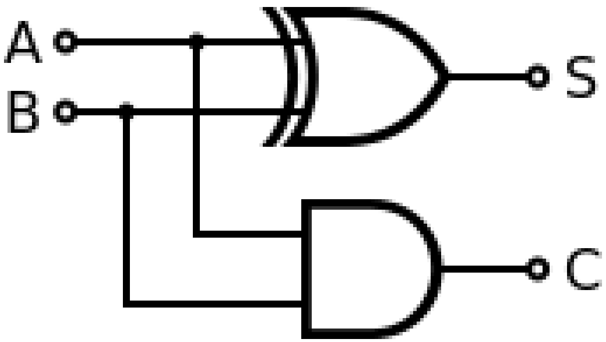
`(a != 0)` becomes $a_0 \vee a_1 \vee a_2 \vee a_3$

`(c>0)` becomes $\neg c_3 \wedge (c_2 \vee c_1 \vee c_0)$

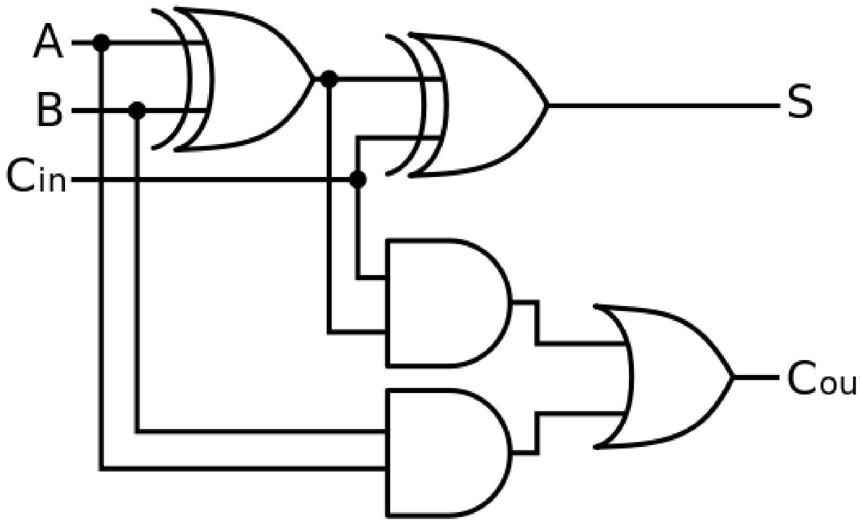
What about addition?

One-Bit Adder

Half Adder

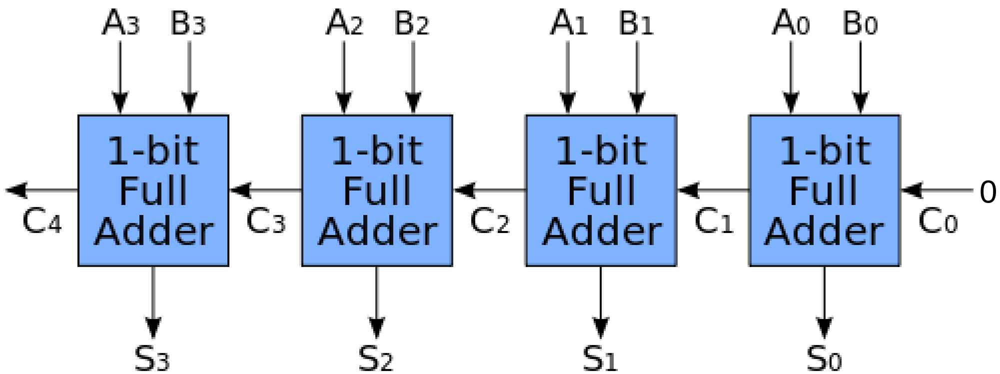


Full Adder



Halfadder: Wikipedia, public domain; full adder: **Colin M.L. Burnett**, GDFL

4-bit Adder



Write formula $\phi(a_3, a_2, a_1, a_0, b_3, b_2, b_1, b_0, s_3, s_2, s_1, s_0)$ such that $\phi(a, b, s)$ is true iff $s = a + b$.

Note: there are extra variable in ψ that don't bother us (why not?)

Software

```
int a, b, c;
if(a != 0){
   c = (a + b);
   if(c > 0)
      assert(false);
}
```

Let's pretends `int`s are 4 bits: a_3, a_2, a_1, a_0

$\psi(a,b,c) =$
$(a_0 \vee a_1 \vee a_2 \vee a_3) \wedge$ `a != 0`
$\phi(a,b,c) \wedge$ `c = a +b`
$\neg c_3 \wedge (c_2 \vee c_1 \vee c_0)$ `c > 0`

ψ satisfiable iff assertion reachable.

Summarizing

We know how to represent a single path in a formula

From now on, I will use arithmetic in my functions

How do we deal with multiple paths and conditions? Two options:

1. Bounded Model Checking
2. Concolic Testing

Bounded Model Checking

- Create a formula that says a bug exist, give to SMT solver.
- Formula: *Is there a path of length* $\leq k$ *to a bug?*

Tool: CBMC

From Path to Program: BMC

Program

```
int a, b, c;
if(c > 0){
  assert(c < a);
else
  assert (c > a);
```

Formula

Path condition

$$\phi = (c > 0) \land \neg(c < a) \lor \neg(c > 0) \land \neg(c > a)$$

ϕ is true iff the program contains a bug.

idea: represent all paths in a formula

From Path to Program: BMC

Program

```
int a, b, c;
if(c > 0){
  assert(c < a);
else
  assert (c > a);
```

Formula

$$\phi =$$

ϕ is true iff the program contains a bug.

idea: represent all paths in a formula

Loop unrolling assuming that b>0

Program

```
int a, b, as, bs;
as = a;
bs = b;
while(b>0){
   a = a + 1;
   b = b - 1;
}
assert(a == as + bs);
```

Formula

BMC: Loop Unrolling

Program

```
int a, b, as, bs;
as = a;
bs = b;
while(b>0){
   a = a + 1;
   b = b - 1;
}
assert(a == as + bs);
```

Program'(unroll 0)

```
int a, b, as, bs;
as = a;
bs = b;
if(b>0){
   stop;
}
assert(a == as + bs);
```

print a warning unrolling not long enough!

BMC: Loop Unrolling

Program

```
int a, b, as, bs;
as = a;
bs = b;
while(b>0){
   a = a + 1;
   b = b - 1;
}
assert(a == as + bs);
```

Program'(unroll 1)

```
int a, b, as, bs;
as = a;
bs = b;
if(b>0){
   a = a + 1;
   b = b - 1;
   if(b>0) stop;
}
assert(a == as + bs);
```

BMC: Loop Unrolling

Program

```
int a,b,as,bs;
as = a;
bs = b;
while(b>0){
  a = a + 1;
  b = b - 1;
}
assert(a==as+bs);
```

Program'(1)

```
int a,b,as,bs;
as = a;
bs = b;
if(b>0){
  a = a + 1;
  b = b - 1;
  if(b>0) stop;
}
assert(a==as+bs);
```

Program''(2)

```
int a,b,as,bs;
as = a;
bs = b;
if(b>0){
  a = a + 1;
  b = b - 1;
  if(b>0){
    a = a + 1;
    b = b - 1
    if(b>0) stop;
  }
}
assert(a==as+bs);
```

BMC: Loop Unrolling

Program'

```
int a, b, as, bs;
as = a;
bs = b;
if(b>0){
   a = a + 1;
   b = b - 1;
   if(b>0) stop;
}
assert(a == as + bs);
```

Program' (SSA)

```
int a, b, as, bs;
as = a;
bs = b;
if(b>0){
   a1 = a + 1;
   b1 = b - 1;
   if(b1>0) stop;
} else {
   a1 = a; b1 = b;
}
assert(a1 == as + bs);
```

Verification Condition

```
int a, b, as, bs;
as = a;
bs = b;
if(b>0){
  a1 = a + 1;
  b1 = b - 1;
  if(b1>0) stop;
} else {
  a1 = a; b1 = b;
}
assert(a1 == as + bs);
```

Finding assertion violation

$\phi_1 = (as = a \land bs = b \land b \leq 0 \land a_1 = a \land b_1 = b \land a_1 \neq as + bs)$

$\phi_2 = (as = a \land bs = b \land b > 0 \land a_1 = a + 1 \land b_1 = b + 1 \land a \neq as + bs)$

Verification condition: $\phi = \phi_1 \lor \phi_2$

Have we unrolled enough?

Let

$\psi = (as = a \land bs = b \land b > 0 \land a_1 = a + 1 \land b_1 = b - 1 \land b1 > 0)$

If ψ is satisfiable, verification is not complete: unroll the loop further!

Loop unrolling

Check for bugs that occur when the loops are unrolled *k* times, for some *k.*

Good:

- Find **all** bugs for any input up to that depth

Bad:

- Expressions quickly become complicated; you will not go *deep* into a program

What if we want to test deeply?

Concolic Testing

- Idea: combine random testing with symbolic execution. Then, systematically look for inputs that take a different path.
- Formula: *Can this path lead to a bug?*

Tools: DART, CUTE
(see also KLEE for symbolic execution)

Concolic Testing Example

- Start with random input
- Execute program with concrete inputs and symbolically at the same time. Concrete values determine path
- Negate part of condition to obtain different path
- Give to solver to obtain new values
- Repeat

Path Condition

Path condition: formula that states how to get to a given location.

printf reached with path condition ?

```
int h(int x, int y) {
  if (x == y)
    if (2*x == x + 10)
      abort(); /*error*/
      else
       printf("hello");
  return 0;
}
```

Path Condition

Path condition: formula that states how to get to a given location.

printf reached with path condition $x = y \land 2x \neq x + 10$

```
int h(int x, int y) {
  if (x == y)
    if (2*x == x + 10)
      abort(); /*error*/
    else
      printf("hello");
  return 0;
}
```

Concolic Testing Example

1. Start with random input
2. Execute program with concrete inputs and symbolically at the same time. Concrete values determine path
3. Negate part of condition to obtain different path
4. Give to solver to obtain new values
5. Repeat

```
int h(int x, int y) {
  if (x == y)
    if (2*x == x + 10)
      abort(); /*error*/
  return 0;
}
```

1. Call `h(12,88)`

2. h takes else branch h. Path condition: $\phi_1 = (x \neq y)$
3. $\neg\phi_1 = (x = y)$
4. Obtain an assignment for $\neg\phi$. Example: x = 42, y= 42

2. new call: h(42,42). Program takes then branch and else branch. Path condition: $\phi_2 = (x \neq y) \wedge (2x \neq x + 10)$.
3. Negate $\phi = \phi_1 \vee \phi_2 = (x \neq y) \vee ((x = y) \wedge (2x \neq x + 10)$
4. Obtain an assignment for $\neg\phi$: x=10, y =10.

2. New call: h(10,10). Finds bug.

Concolic Testing Example

1. Start with random input
2. Execute program with concrete inputs and symbolically at the same time. Concrete values determine path
3. Negate part of condition to obtain different path
4. Give to solver to obtain new values
5. Repeat

```
int h(int x, int y) {
  if (x == y)
    if (2*x == x + 10)
      abort(); /*error*/
  return 0;
}
```

Concolic Testing

In which order do we change conditions?

- Any search order we want.
- Example: always negate last part of condition $\rightarrow$ DFS

1. Start with random input
2. Execute program with concrete inputs and symbolically at the same time. Concrete values determine path
3. Negate part of condition to obtain different path
4. Give to solver to obtain new values
5. Repeat

Dealing with Memory

Random pointers make little sense – prefer NULL pointers, or allocated structs.

Dealing with Memory

```
typedef struct cell{
  int v;
  struct cell *next;
} cell;

int f(int v){ return 2*v+1;
   }

int testme(cell *p, int x){
  if(x > 0)
    if(p != NULL)
      if(f(x) == p->v)
        if(p->next == p)
          ERROR;
  return 0;
}
```

Dealing with Memory

```
typedef struct cell{
  int v;
  struct cell *next;
} cell;

int f(int v){ return 2*v+1;
   }

int testme(cell *p, int x){
  if(x > 0)
    if(p != NULL)
      if(f(x) == p->v)
        if(p->next == p)
          ERROR;
  return 0;
}
```

Start: x=236; p = NULL
path: x>0; p==NULL.
Solve x>0 && p!=NULL

x=236, p->[634,NULL]
path: x>0; p!=NULL; 2x+1 != p->v
solve x>0 && p != NULL && 2x+1==p->v

x=1; p->[3,NULL]
path: x>0; p!=NULL; 2x+1 == p->v; p->next!=p
solve x>0 && p != NULL && 2x+1==p->v && p->next==p

x=1; p->[3,p]
ERROR reached

Conclusions

- Symbolic representation of programs
- Systematic search for all bad behavior

BMC tries all paths simultaneously.

- Query: there is a path of length k to a bug
- Like breadth-first search: wide and shallow

Concolic tries one path at a time

- Query: this path (of length k) leads to a bug
- Like depth-first search: deep and narrow

Literature

- P. Godefroid, N. Klarlund, and K. Sen, DART: Directed Automated Random Testing, *Proc. Programming Language Design and Implementation,* 2005
- K. Sen, D. Marinov, and G. Agha, CUTE: A Concolic Unite Testing Engine for C, Proc. *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2005*