

Software Model Checking I

Dynamic v.s. Static Analysis

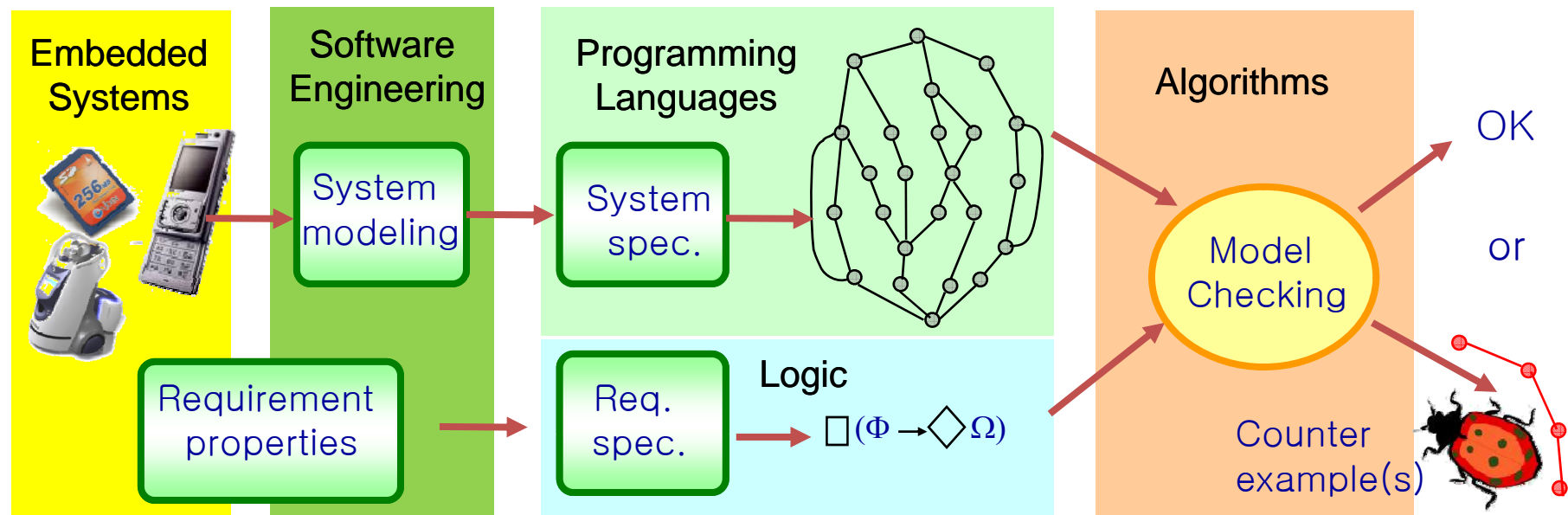
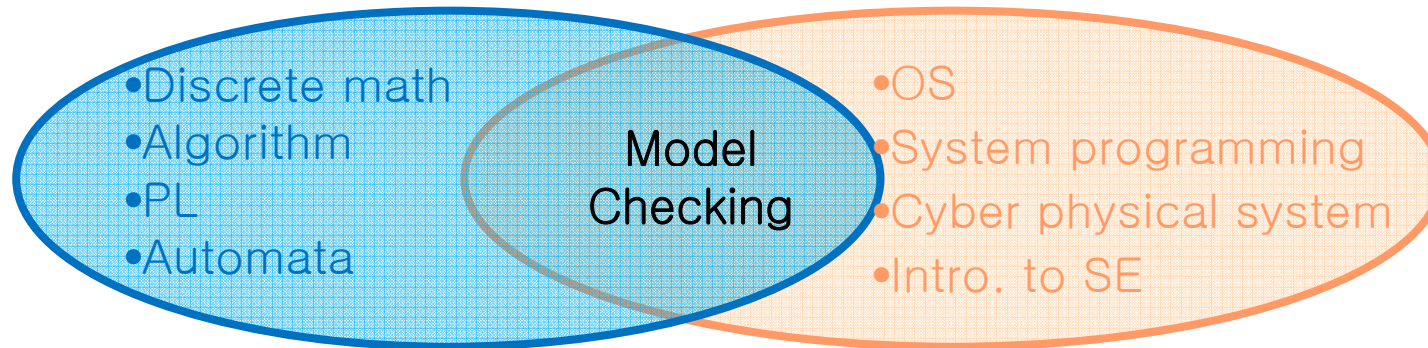
| | Dynamic Analysis (i.e., testing) | Static Analysis (i.e. model checking) |
|------|---|---|
| Pros | <ul style="list-style-type: none">•Real result•No environmental limitation•Binary library is ok | <ul style="list-style-type: none">•Complete analysis result•Fully automatic•Concrete counter example |
| Cons | <ul style="list-style-type: none">•Incomplete analysis result•Test case selection | <ul style="list-style-type: none">•Consumed huge memory space•Takes huge time for verification•False alarms |

Motivation for Software Model Checking

- Data flow analysis (DFA): fastest & least precision
 - “May” analysis,
- Abstract interpretation (AI): fast & medium precision
 - Over-approximation & under-approximation
- Model checking (MC): slow & complete
 - Complete value analysis
 - No approximation
- Static analyzer & MC as a C debugger
 - Handling complex C structures such as pointer and array
 - DFA: might-be
 - AI: may-be
 - MC: can-be or should-be

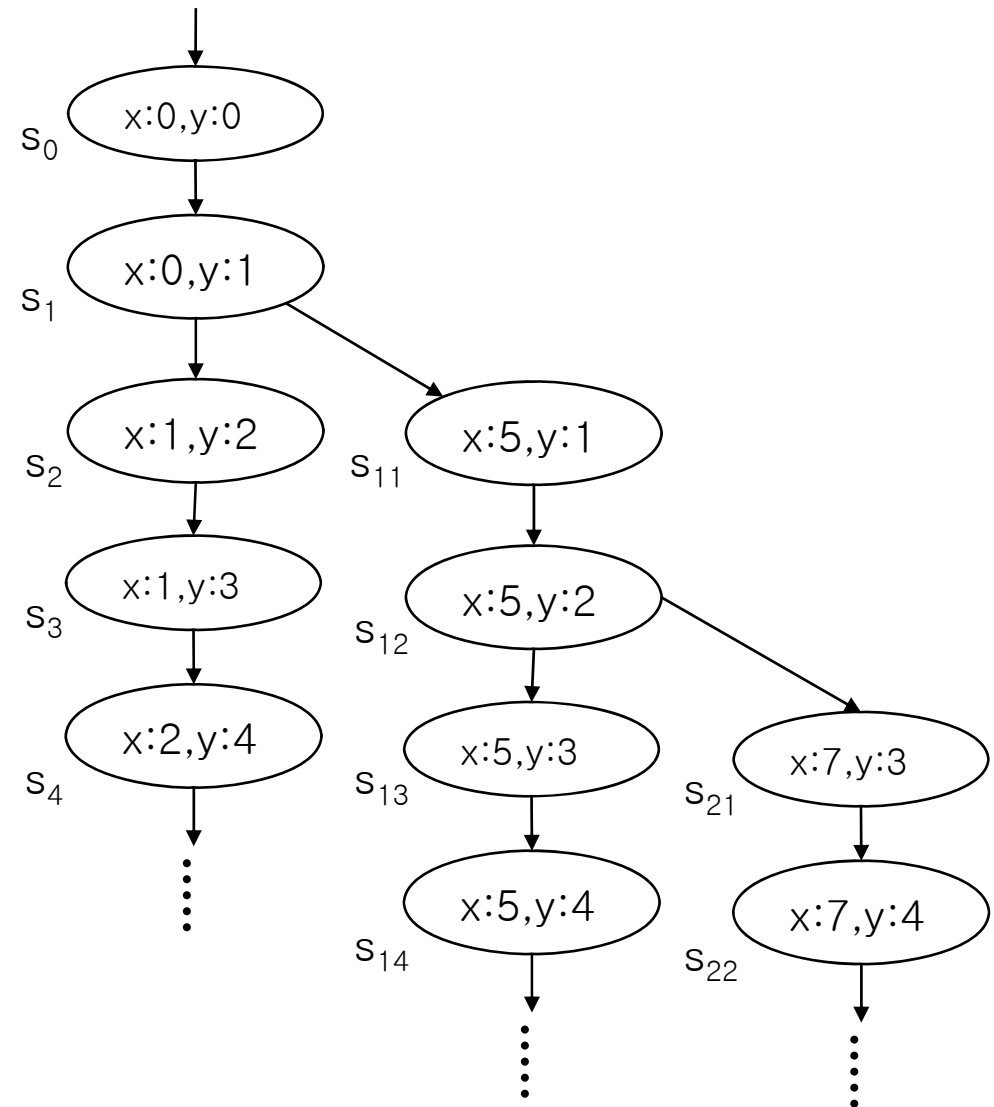
Model Checking Background

- Undergraduate CS classes contributing to this area



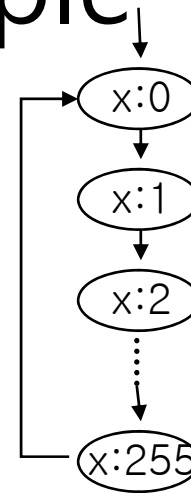
Operational Semantics of Software

- A system execution σ is a sequence of states $s_0 s_1 \dots$
 - A state has an environment $\rho_s: Var \rightarrow Val$
- A system has its semantics as a set of system executions



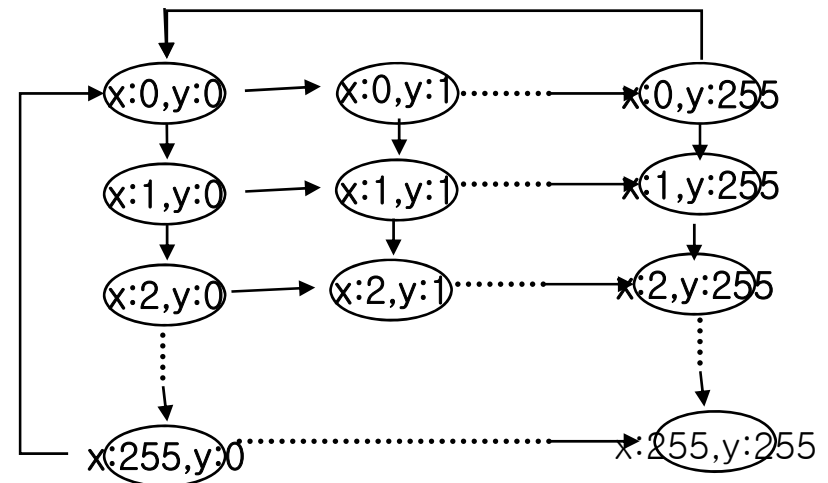
Example

```
active type A() {
  byte x;
again:
  x++;
  goto again;
}
```



```
active type A() {
  byte x;
again:
  x++;
  goto again;
}
```

```
active type B() {
  byte y;
again:
  y++;
  goto again;
}
```



Pros and Cons of Model Checking

- Pros
 - Fully automated and provide complete coverage
 - Concrete counter examples
 - Full control over every detail of system behavior
 - Highly effective for analyzing
 - embedded software
 - multi-threaded systems
- Cons
 - State explosion problem
 - An abstracted model may not fully reflect a real system
 - Needs to use a specialized modeling language
 - Modeling languages are similar to programming languages, but simpler and clearer

Companies Working on Model Checking

Microsoft



cadence



Model Checking History

| | | |
|------|---|-------------|
| 1981 | Clarke / Emerson: CTL Model Checking Sifakis / Quielle | 10^5 |
| 1982 | EMC: Explicit Model Checker Clarke, Emerson, Sistla | |
| 1990 | Symbolic Model Checking Burch, Clarke, Dill, McMillan | 10^{100} |
| 1992 | SMV: Symbolic Model Verifier McMillan | |
| 1998 | Bounded Model Checking using SAT Biere, Clarke, Zhu | 10^{1000} |
| 2000 | Counterexample-guided Abstraction Refinement Clarke, Grumberg, Jha, Lu, Veith | |



Example. Sort (1/2)

| | | | |
|---|----|---|-----|
| 9 | 14 | 2 | 255 |
|---|----|---|-----|

- Suppose that we have an array of 4 elements each of which is 1 byte long
 - unsigned char a[4];
- We want to verify sort.c works correctly
 - `main() { sort(); assert(a[0]<= a[1]<= a[2]<=a[3]);}`
- Hash table based **explicit model checker** (ex. Spin) generates at least 2^{32} ($= 4 \times 10^9 = 4\text{G}$) states
 - 4G states x 4 bytes = 16 Gbytes, no way...
- Binary Decision Diagram (BDD) based **symbolic model checker** (ex. NuSMV) takes 200 MB in 400 sec

Example. Sort (2/2)

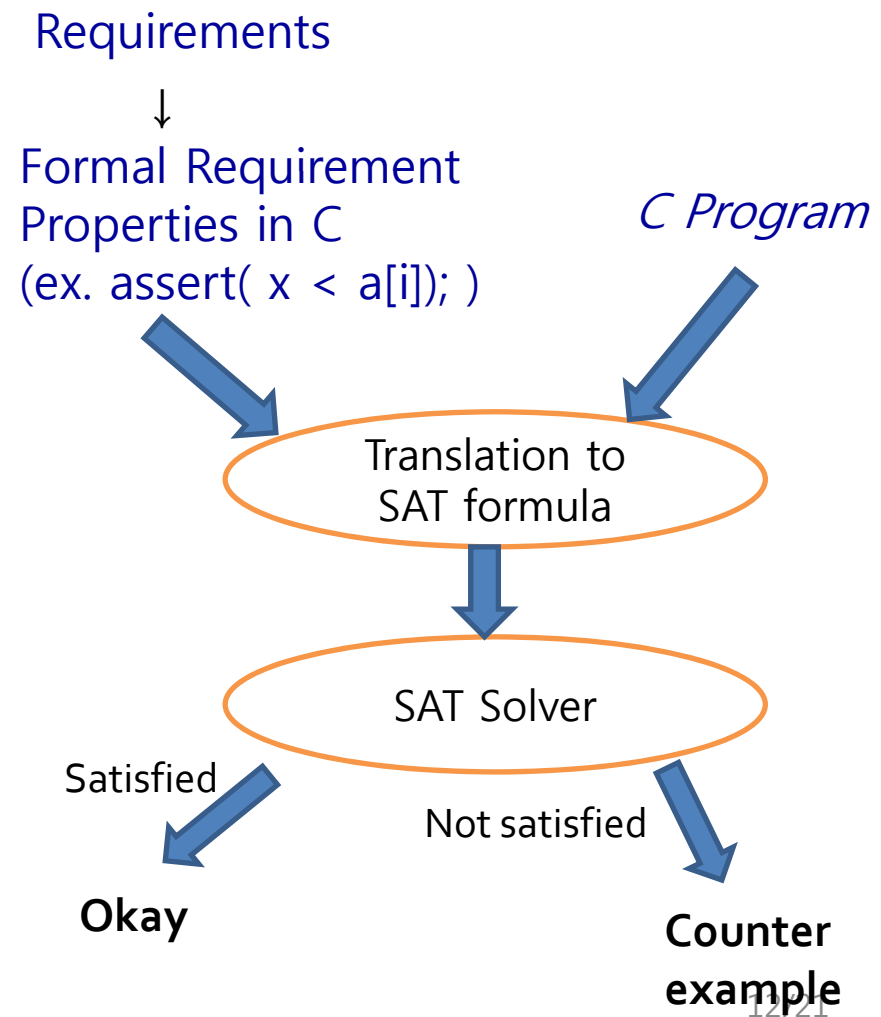
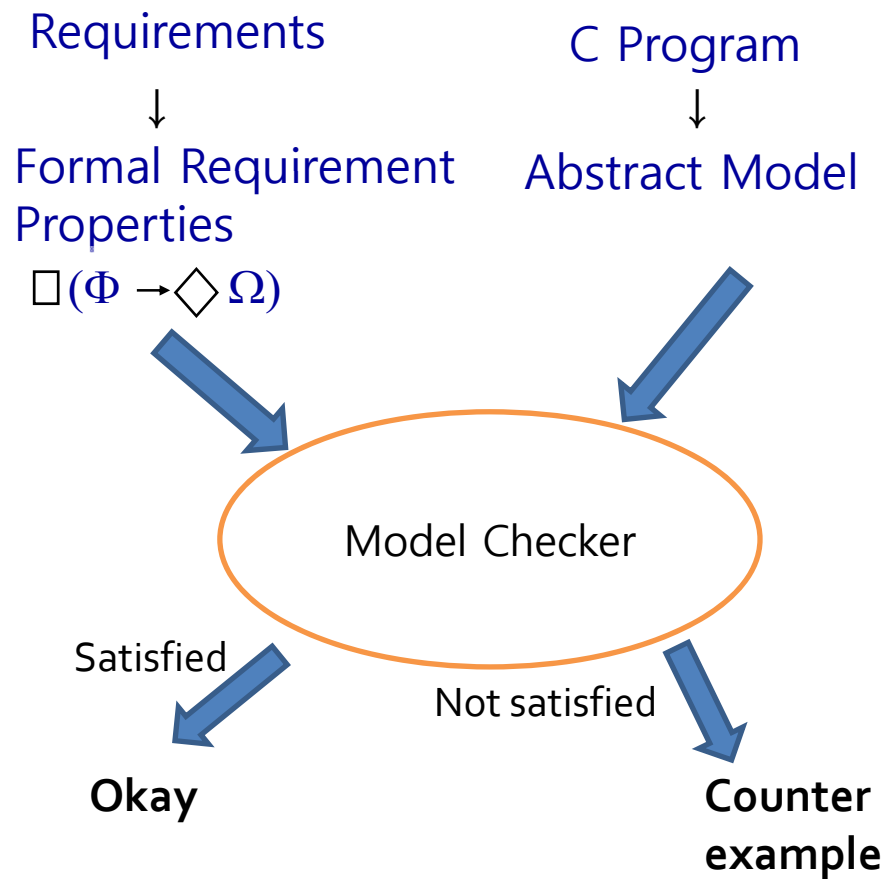
```
1. #include <stdio.h>
2. #define N 5
3. int main(){
4.     int data[N], i, j, tmp;
5.     /* Assign random values to the array*/
6.     for (i=0; i<N; i++){
7.         data[i] = nondet_int();
8.     }
9.     /* It misses the last element, i.e., data[N-1]*/
10.    for (i=0; i<N-1; i++){
11.        for (j=i+1; j<N-1; j++){
12.            if (data[i] > data[j]){
13.                tmp = data[i];
14.                data[i] = data[j];
15.                data[j] = tmp;
16.            }
17.    /* Check the array is sorted */
18.    for (i=0; i<N-1; i++){
19.        assert(data[i] <= data[i+1]);
20.    }
21. }
```

- SAT-based Bounded Model Checker
 - Total 19099 CNF clause with 6224 boolean propositional variables
 - Theoretically, 2^{6224} choices should be evaluated!!!

| SAT | VSIDS |
|-----------|-------|
| Conflicts | 73 |
| Decisions | 2435 |
| Time(sec) | 0.015 |

| UNSAT | VSIDS |
|-----------|--------|
| Conflicts | 35067 |
| Decisions | 161406 |
| Time(sec) | 1.89 |

Overview of SAT-based Bounded Model Checking



SAT Basics (1/3)

- SAT = Satisfiability
= Propositional Satisfiability



- NP-Complete problem
 - We can use SAT solver for many NP-complete problems
 - Hamiltonian path
 - 3 coloring problem
 - Traveling sales man's problem
- Recent interest as a verification engine

SAT Basics (2/3)

- A set of propositional variables and Conjunctive Normal Form (CNF) clauses involving variables
 - $(x_1 \vee x_2' \vee x_3) \wedge (x_2 \vee x_1' \vee x_4)$
 - x_1, x_2, x_3 and x_4 are variables (true or false)
- Literals: Variable and its negation
 - x_1 and x_1'
- A clause is satisfied if one of the literals is true
 - $x_1 = \text{true}$ satisfies clause 1
 - $x_1 = \text{false}$ satisfies clause 2
- Solution: An assignment that satisfies all clauses

SAT Basics (3/3)

- DIMACS SAT Format

– Ex. $(x_1 \vee x_2' \vee x_3)$
 $\wedge (x_2 \vee x_1' \vee x_4)$

```
p cnf 4 2
1 -2 3 0
2 -1 4 0
```

Model

| x_1 | x_2 | x_3 | x_4 | Formula |
|-------|-------|-------|-------|---------|
| T | T | T | T | T |
| T | T | T | F | T |
| T | T | F | T | T |
| T | T | F | F | T |
| T | F | T | T | T |
| T | F | T | F | F |
| T | F | F | T | T |
| T | F | F | F | F |
| F | T | T | T | T |
| F | T | T | F | T |
| F | T | F | T | F |
| F | T | F | F | F |
| F | F | T | T | T |
| F | F | T | F | T |
| F | F | F | T | T |
| F | F | F | F | T |

Software Model Checking as a SAT problem (1/3)

- Unwinding Loop

Original code

```
x=0 ;  
while (x < 2) {  
    y=y+x ;  
    x++ ;  
}
```

Unwinding the loop 3 times

```
x=0 ;  
if (x < 2) {  
    y=y+x ;  
    x++ ;  
}  
if (x < 2) {  
    y=y+x ;  
    x++ ;  
}  
if (x < 2) {  
    y=y+x ;  
    x++ ;  
}
```

Unwinding assertion: —————→

```
assert (! (x < 2))
```


Examples

```
/* Straight-forward  
   constant upperbound */  
for(i=0,j=0; i < 5; i++) {  
    j=j+i;  
}
```

```
/*Constant upperbound*/  
for(i=0,j=0; j < 10; i++) {  
    j=j+i;  
}
```

```
/* Complex upperbound */  
for(i=0; i < 5; i++) {  
    for(j=i; j < 5;j++) {  
        for(k= i+j; k < 5; k++) {  
            m += i+j+k;  
        }  
    }  
}
```

```
/* Upperbound unknown */  
for(i=0,j=0; i^6-4*i^5 -17*i^4 != 9604 ; i++) {  
    j=j+i;  
}
```

Model Checking as a SAT problem (2/3)

- From C Code to SAT Formula

Original code

```
x=x+y;  
if (x!=1)  
    x=2;  
else  
    x++;  
assert(x<=3);
```

Convert to static single assignment (SSA)

```
x1=x0+Y0;  
if (x1!=1)  
    x2=2;  
else  
    x3=x1+1;  
x4=(x1!=1)?x2:x3;  
assert(x4<=3);
```

Generate constraints

$$C \equiv x_1 = x_0 + Y_0 \wedge x_2 = 2 \wedge x_3 = x_1 + 1 \wedge (x_1 \neq 1 \wedge x_4 = x_2 \vee x_1 = 1 \wedge x_4 = x_3)$$
$$P \equiv x_4 \leq 3$$

Check if $C \wedge \neg P$ is satisfiable, if it is then the assertion is violated

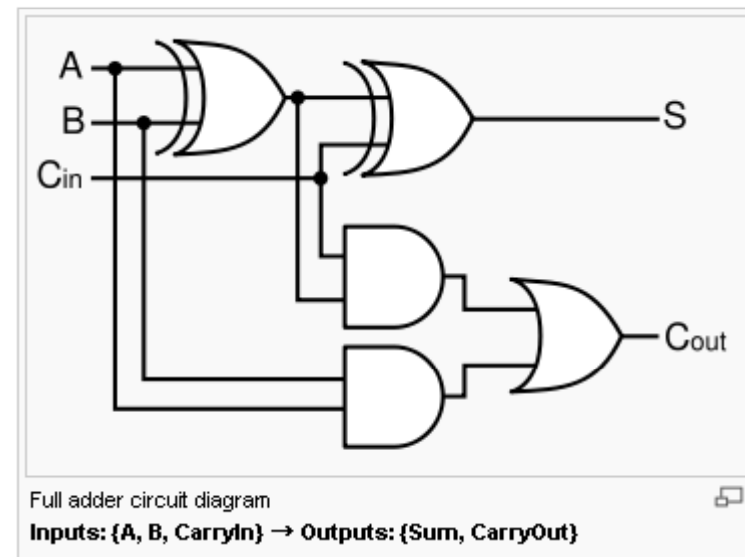
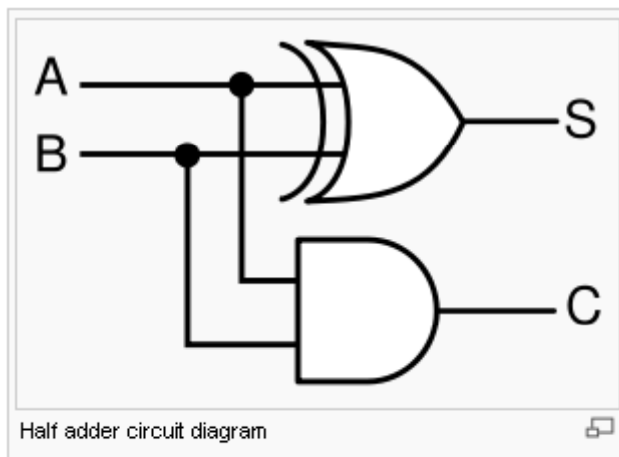
$C \wedge \neg P$ is converted to Boolean logic using a bit vector representation for the integer variables $Y_0, x_0, x_1, x_2, x_3, x_4$

Model Checking as a SAT problem (3/3)

- Example of arithmetic encoding into pure propositional formula

Assume that x, y, z are three bits positive integers represented by propositions $x_0x_1x_2, y_0y_1y_2, z_0z_1z_2$

$$\begin{aligned} C \equiv z=x+y \equiv & (z_0 \leftrightarrow (x_0 \oplus y_0) \oplus ((x_1 \wedge y_1) \vee (((x_1 \oplus y_1) \wedge (x_2 \wedge y_2)))) \\ & \wedge (z_1 \leftrightarrow (x_1 \oplus y_1) \oplus (x_2 \wedge y_2)) \\ & \wedge (z_2 \leftrightarrow (x_2 \oplus y_2)) \end{aligned}$$



Example

```
/* Assume that x and y are 2 bit
unsigned integers */
/* Also assume that x+y <= 3 */
void f(unsigned int y) {
    unsigned int x=1;
    x=x+y;
    if (x==2)
        x+=1;
    else
        x=2;
    assert(x ==2);
}
```

Model Checking as a SAT problem (4/4)

- CBMC (C Bounded Model Checker)
 - Handles function calls using inlining
 - A return value of a undefined function is considered non-deterministic
 - Unwinds the loops a fixed number of times
 - Models various scenarios using non-determinism
 - So that a program can be checked for a set of inputs rather than a single input
 - `__CPROVER_assume (x > 0) ;`
 - Allows specification of assertions which are checked using the bounded model checking