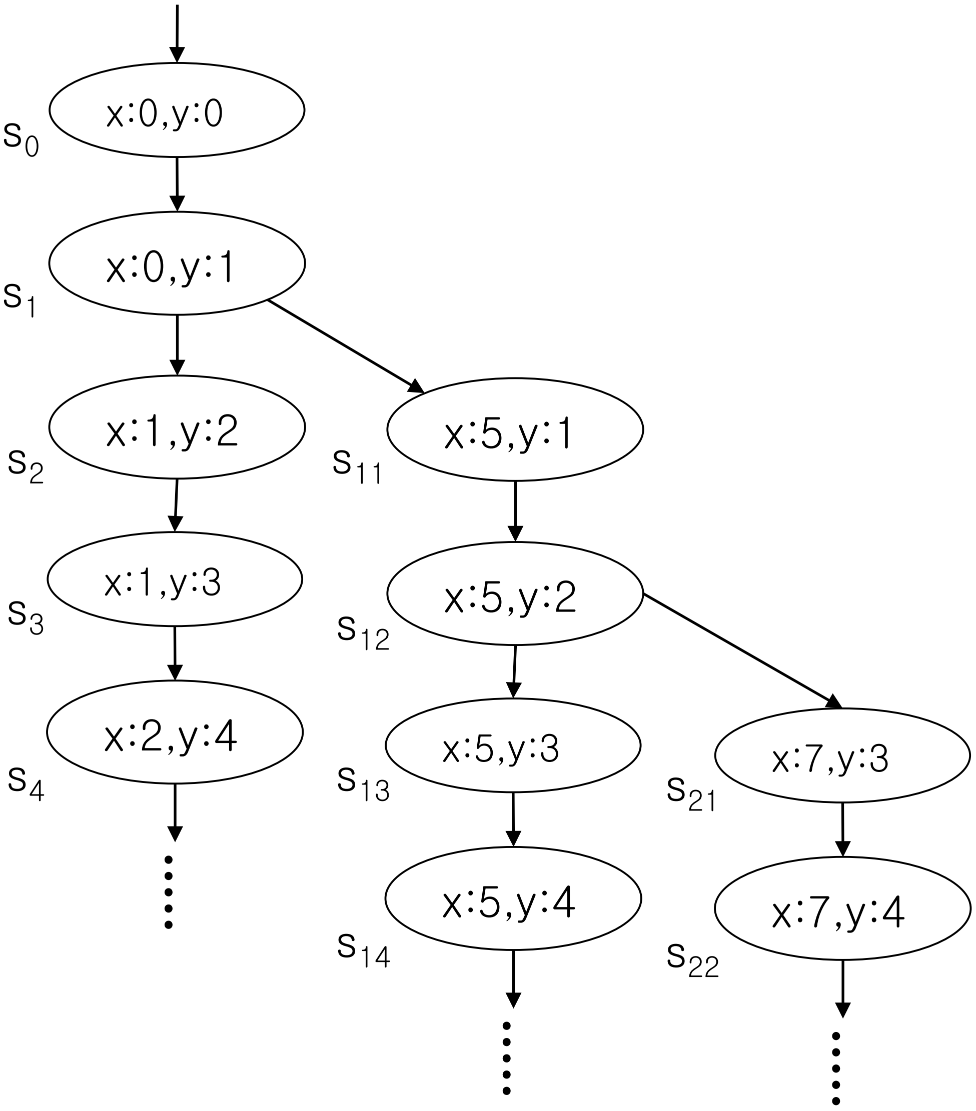
Software Model Checking

Moonzoo Kim

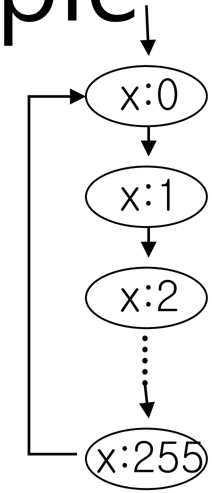
Operational Semantics of Software

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



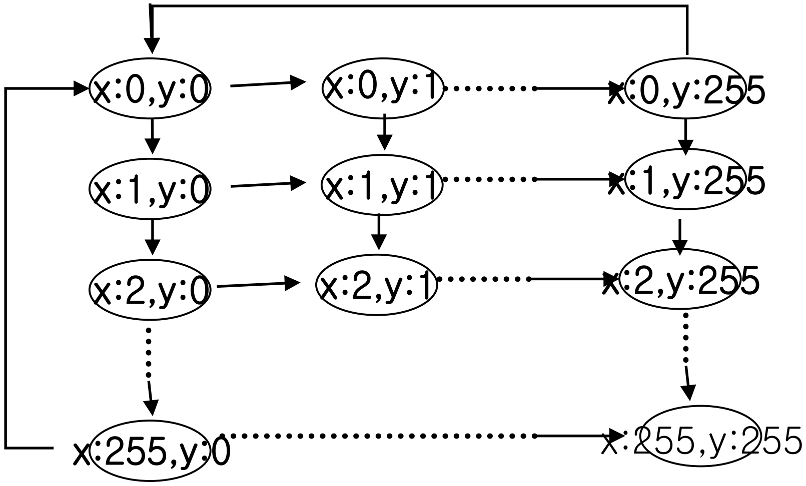
Example

```
active type A() {
byte x;
again:
     x=x+1;;
     goto again;
}
```



```
active type A() {
byte x;
again:
     x=x+1;;
     goto again;
}

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



Note that <u>model checking</u> analyzes ALL possible execution scenarios while <u>testing</u> analyzes SOME execution scenarios

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

SAMSUNG

INRIA

IBM

ESTEREL Technologies

cadence

NEC Empowered by Innovation

Sun microsystems

NASA Jet Propulsion Laboratory
California Institute of Technology

The MathWorks
Accelerating the pace of engineering and science

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)

- Suppose that we have an array of 5 elements each of which is 1 byte long
  - unsigned char a[5];

9	14	2	200	64

- We wants to verify sort.c works correctly
  - main() { sort(); assert(a[0]<= a[1]<= a[2]<=a[3]<=a[4]);}
- Hash table based explicit model checker (ex. Spin) generates at least 2^{40} (= 10^{12} = 1 Tera) states
    - 1 Tera states x 1 byte = 1 Tera byte memory required, no way…
- Binary Decision Diagram (BDD) based symbolic model checker (ex. NuSMV) takes 100 MB in 100 sec on Intel Xeon 5160 3Ghz machine

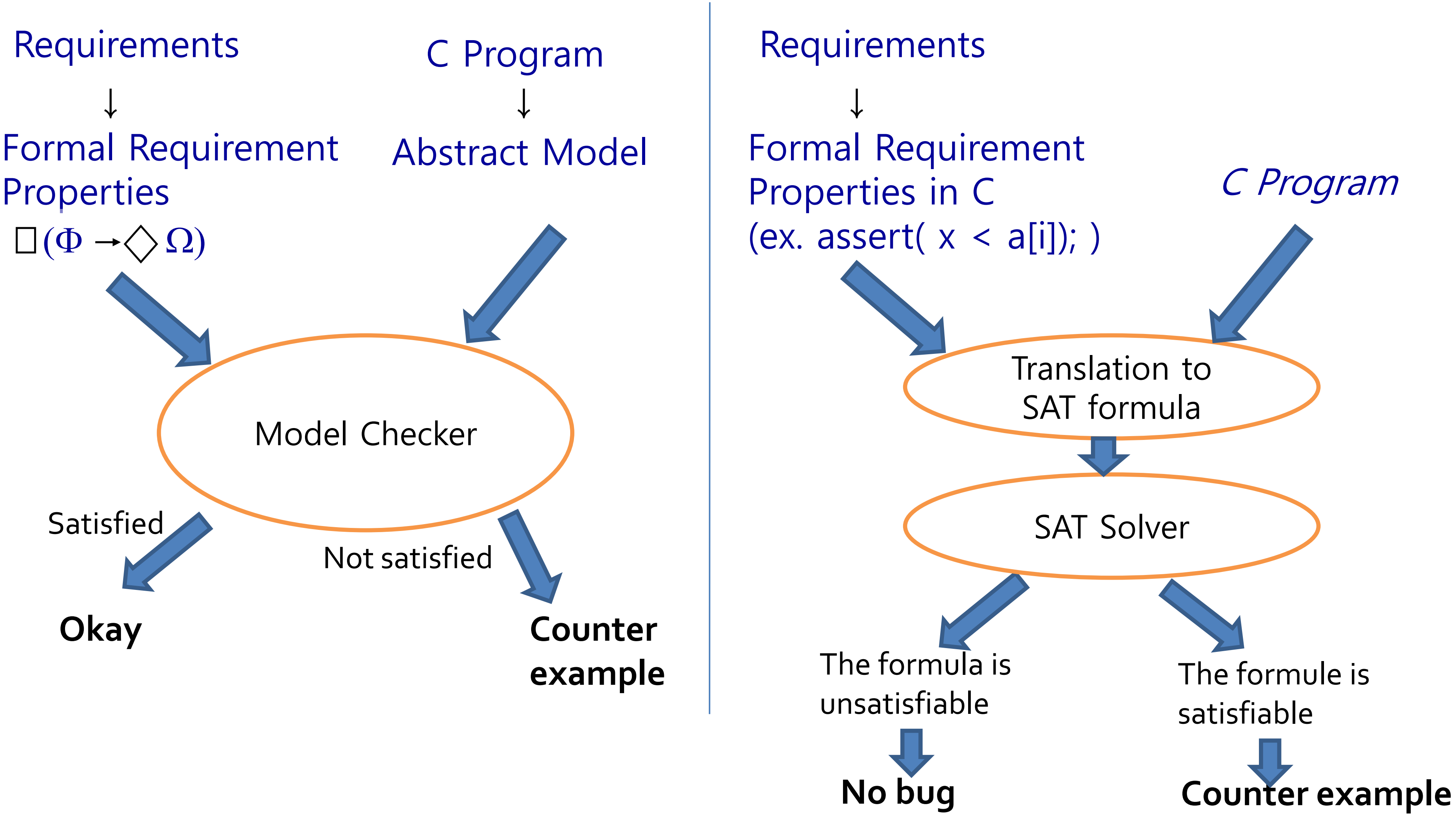
Example. Sort (2/2)

```
#include <stdio.h>
#define N 20
int main(){//Selection sort that selects the smallest # first
     unsigned int data[N], i, j, tmp;
     /* Assign random values to the array*/
     for (i=0; i<N; i++){
          data[i] = nondet_int();
     }
    /* It misses the last element, i.e., data[N-1]*/
     for (i=0; i<N-1; i++)
          for (j=i+1; j<N-1; j++)
               if (data[i] > data[j]){
                    tmp = data[i];
                    data[i] = data[j];
                    data[j] = tmp;
               }
 /* Check the array is sorted */
     for (i=0; i<N-1; i++){
          assert(data[i] <= data[i+1]);
     }
}
```

- SAT-based Bounded Model Checker
  - Total 161,311 CNF clause with 41,646 boolean propositional variables
  - Theoretically, $2^{41,646}$ choices should be evaluated!!!

N	Exec time (CBMC 4.6 i5 3.4Ghz)	Mem	# of var	# of clause
20	2 sec	25M	41,646	161,311
30	41 sec	167M	92,961	363,586
40	156 sec	400M	165,826	648,811
50	430 sec	686M	261,141	1,018,486
100	14 hours	5.9 GB	1,060,216	4,108,876
1000	33 hours	OOM (>64GB)	?	?

Overview of SAT-based Bounded Model Checking



Requirements

↓

Formal Requirement Properties

$\Box(\Phi \rightarrow \Diamond \Omega)$

C Program

↓

Abstract Model

Model Checker

Satisfied

Okay

Not satisfied

Counter example

Requirements

↓

Formal Requirement Properties in C

(ex. assert(x < a[i]);)

C Program

Translation to SAT formula

SAT Solver

The formula is unsatisfiable

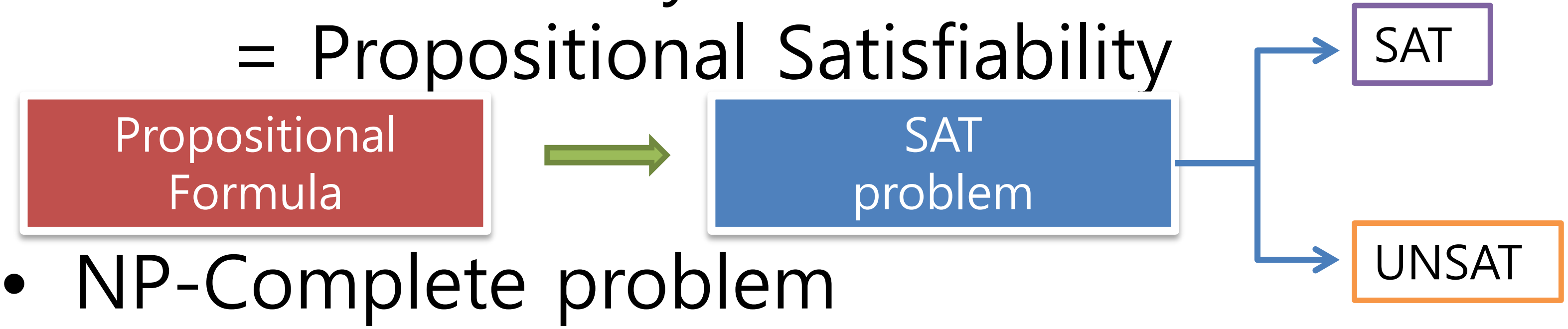
No bug

The formule is satisfiable

Counter example

SAT Basics (1/3)

- SAT = Satisfiability
  = Propositional Satisfiability



Propositional Formula

SAT problem

SAT

UNSAT

- 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=true satisfies clause 1
  - x_1=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/
solution

o	x_1	x_2	x_3	x_4	*f*
o_1	T	T	T	T	T
o_2	T	T	T	F	T
o_3	T	T	F	T	T
o_4	T	T	F	F	T
o_5	T	F	T	T	T
o_6	T	F	T	F	F
o_7	T	F	F	T	T
o_8	T	F	F	F	F
o_9	F	T	T	T	T
o_{10}	F	T	T	F	T
o_{11}	F	T	F	T	F
o_{12}	F	T	F	F	F
o_{13}	F	F	T	T	T
o_{14}	F	F	T	F	T
o_{15}	F	F	F	T	T
o_{16}	F	F	F	F	T

Model Checking as a SAT problem (1/6)

- Control-flow simplification
  - All side effect are removed
    - `i++ => i=i+1`;
  - Control flow is made explicit
    - `continue, break => goto`
  - Loop simplification
    - `for(;;), do {…} while() => while()`

Model Checking as a SAT problem (2/6)

- Unwinding Loop

Original code

```
x=0;
while(x < 2){
  y=y+x;
  x=x+1;;
}
```

Unwinding the loop 1 times

```
x=0;
if (x < 2) {
  y=y+x;
  x=x+1;;
}
/* Unwinding assertion */
assert(!(x < 2))
```

Unwinding the loop 2 times

```
x=0;
if (x < 2) {
  y=y+x;
  x=x+1;;
  if (x < 2) {
    y=y+x;
    x=x+1;;
}}
/* Unwinding assertion */
assert(!(x < 2))
```

Unwinding the loop 3 times

```
x=0;
if (x < 2) {
  y=y+x;
  x=x+1;;
  if (x < 2) {
    y=y+x;
    x=x+1;;
    if (x < 2) {
      y=y+x;
      x=x+1;;
}}}
/*Unwinding assertion*/
assert (! (x < 2))
```

Ex. Constant # of Loop Iterations

```
/*# of loop iter. is constant*/
for(i=0,j=0; i < 5; i++) {
    j=j+i;
}
```

```
/*# of loop iter. is constant*/
for(i=0,j=0; j < 10; i++) {
    j=j+i;
}
```

```
/* Complex but still constant
# of loop iterations */
for(i=0; i < 5; i++) {
    for(j=i; j < 5;j++) {
        for(k= i+j; k < 5; k++) {
            m += i+j+k;
        }
    }
}
```

```
/* # of loop iter. Is unknown */
for(i=0,j=0; i^6−4*i^5 −17*i^4 != 9604 ; i++) {
    j=j+i;
}
```

Ex. Variable # of Loop Iterations Depending on Input

```
/* x: unsigned integer input
   It iterates 0 to 2^32−1 times*/
for(i=0,j=0; i < x; i++) {
    j=j+i;
}
```

```
/* j: unsigned integer input */
for(i=0; j < 10; i++) {
    j=j+i;
}
```

```
/* a: unsigned integer array input */
for(i=0,sum=0; (i<2) || (sum<10) ;i++) {
    sum += a[i];
}
/* Minimum # of iteration? Maximum # of iteration? */
```

Model Checking as a SAT problem (3/6)

- From C Code to SAT Formula

Original code

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

Static single assignment (SSA)

$$\begin{array}{l}
\texttt{x}_1\texttt{==x}_0\texttt{+y}_0\texttt{;}\\
\texttt{if (x}_1\texttt{!=1)}\\
\quad\texttt{x}_2\texttt{==2;}\\
\texttt{else}\\
\quad\texttt{x}_3\texttt{==x}_1\texttt{+1;}
\end{array}$$

Generate SSA constraint of the original code:

$$P \equiv \quad x_1 == x_0 + y_0 \\ \wedge\ x_2 == 2 \\ \wedge\ x_3 == x_1 + 1$$

Every feasible execution scenario of the original code has its corresponding solution of *P* and vice versa.

Note that solutions/models of *P* represent feasible execution scenarios of the original code

Ex1. W/ initial values x=1 and y=0, x becomes 2 at the end.
See that *P* is true w/ the following corresponding solution $(x_0,x_1,x_2,x_3,y_0) = (1,1,2,2,0)$

Ex2. See that P is false w/ $(x_0,x_1,x_2,\mathbf{x_3},y_0) = (1,1,2,\mathbf{3},0)$.
Note that no corresponding execution scenario of the original code

Model Checking as a SAT problem (4/6)

- From C Code to SAT Formula

Original code

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

Convert to static single assignment (SSA)

$x_1 == x_0 + y_0;$

$\texttt{if}\ (x_1 != 1)$

$\quad x_2 == 2;$

$\texttt{else}$

$\quad x_3 == x_1 + 1;$

$x_4 == (x_1 != 1) ? x_2 : x_3;$

$\mathbf{assert}(x_4 <= 3);$

Generate constraints

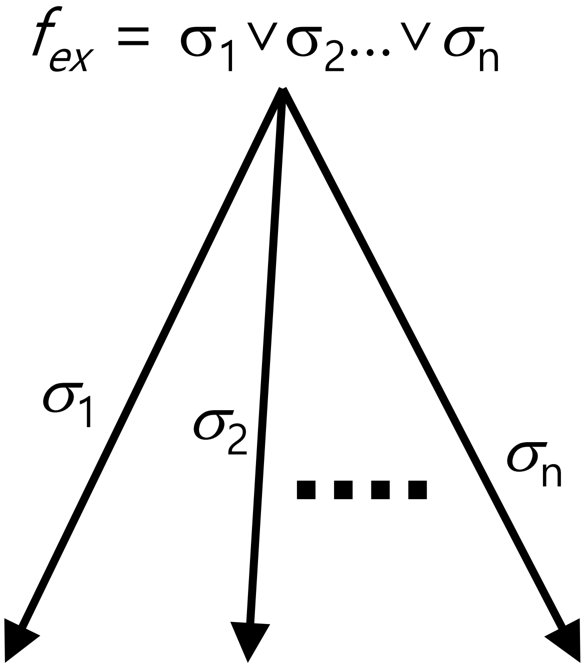
$P \equiv x_1 == x_0 + y_0 \wedge x_2 == 2 \wedge x_3 == x_1 + 1 \wedge ((x_1 != 1 \wedge x_4 == x_2) \vee (x_1 == 1 \wedge x_4 == x_3))$

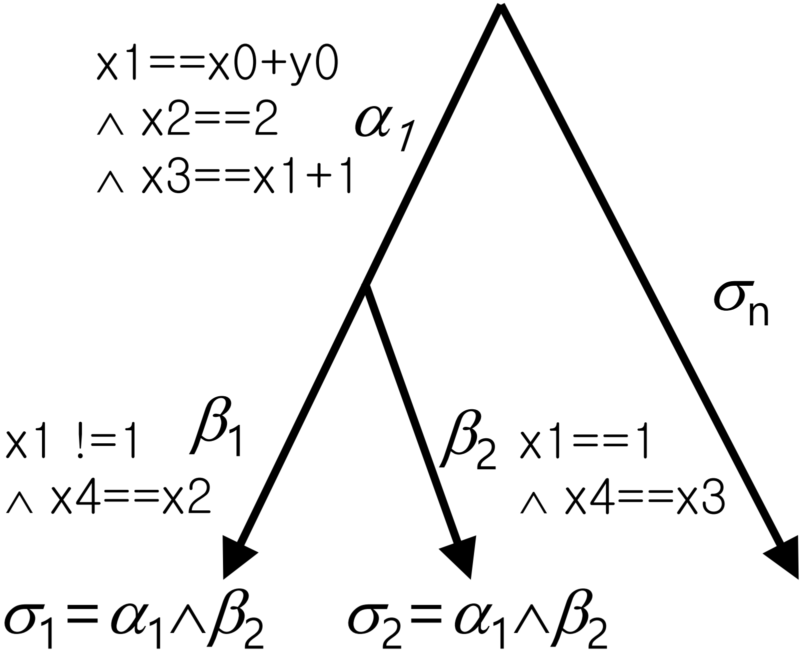
$A \equiv x_4 <= 3$

Check if $P \wedge \neg A$ is satisfiable.

- If it is satisfiable, the assertion is violated (i.e., the program is buggy w.r.t *A*)
- If it is unsatisfiable, the assertion is never violated (i.e., program is correct w.r.t. *A*)

Question: Why not $P \wedge A$ but $P \wedge \neg A$?





Note that a whole execution tree (i.e. all target program executions) can be represented as a single SSA formulae.

- A whole execution tree can be represented as a disjunction of SSA formulas each of which represents an execution (i.e. $f_{ex} = \vee\ \sigma_i$) since $\vee$ represents different worlds/scenarios.
  - Each execution can be represented as a SSA formula (saying σ_i)
  - Each execution can be represented using $\wedge$ and $\vee$ for corresponding execution segments

Model Checking as a SAT problem (5/6)

Original code

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

Convert to static single assignment (SSA)

```
x_1==x_0+y_0;
if (x_1!=1)
  x_2==2;
else
  x_3==x_1+1;
x_4==(x_1!=1)?x_2:x_3;
assert(x_4<=3);
```

$$P \equiv x_1 == x_0 + y_0 \wedge x_2 == 2 \wedge x_3 == x_1 + 1 \wedge ((x_1 != 1 \wedge x_4 == x_2) \vee (x_1 == 1 \wedge x_4 == x_3))$$

$$A \equiv x_4 <= 3$$

Observations on the code

1. An execution scenario starting with x==1 and y==0 satisfies the assert
2. The code is correct (i.e., no bug w.r.t. *A*)
   - -case 1: x==1 at line 2=> x==2 at line 6
   - -case 2: x!=1 at line 2 => x==2 at line 6

Observations on the *P*

1. A solution of *P* which assigns every free variable with a value and makes *P* true satisfies *A*
   - ex. (x_0:1, x_1:1, x_2:2, x_3:2, x_4:2, y_0:0)
2. Every solution of *P* represents a feasible execution scenario
3. $P \wedge \neg A$ is unsatisfiable because every solution has x_4 as 2

Model Checking as a SAT problem (6/6)

Finally, $P \wedge \neg A$ is converted to Boolean logic using a bit vector representation for the integer variables $y_0, x_0, x_1, x_2, x_3, x_4$

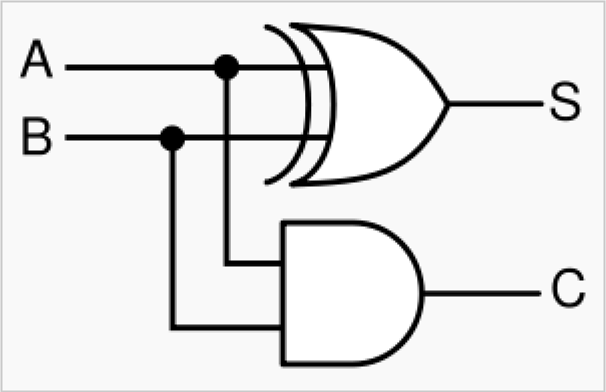
- Example of arithmetic encoding into pure propositional formula

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

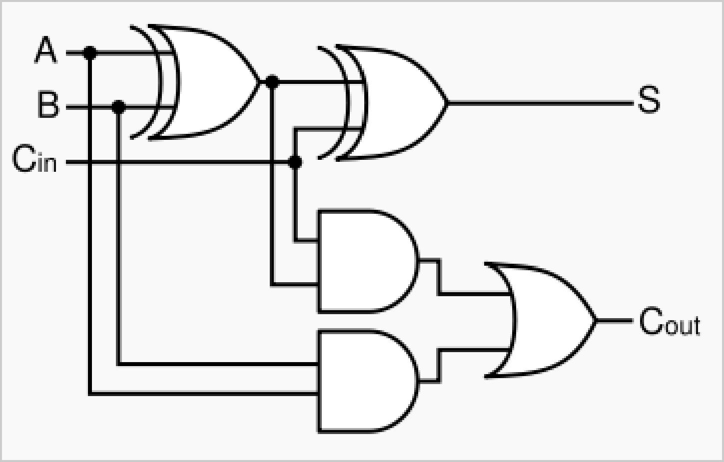
$P \equiv z=x+y \equiv \quad (z_0\$\ (x_0©y_0)©(\ (x_1Æy_1)\ Ç\ (((x_1©y_1)Æ(x_2Æy_2)))$

$Æ(z_1\$\ (x_1©y_1)©(x_2Æy_2))$

$Æ(z_2\$\ (x_2©y_2))$



Half adder circuit diagram



Full adder circuit diagram

Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}

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);
}
```

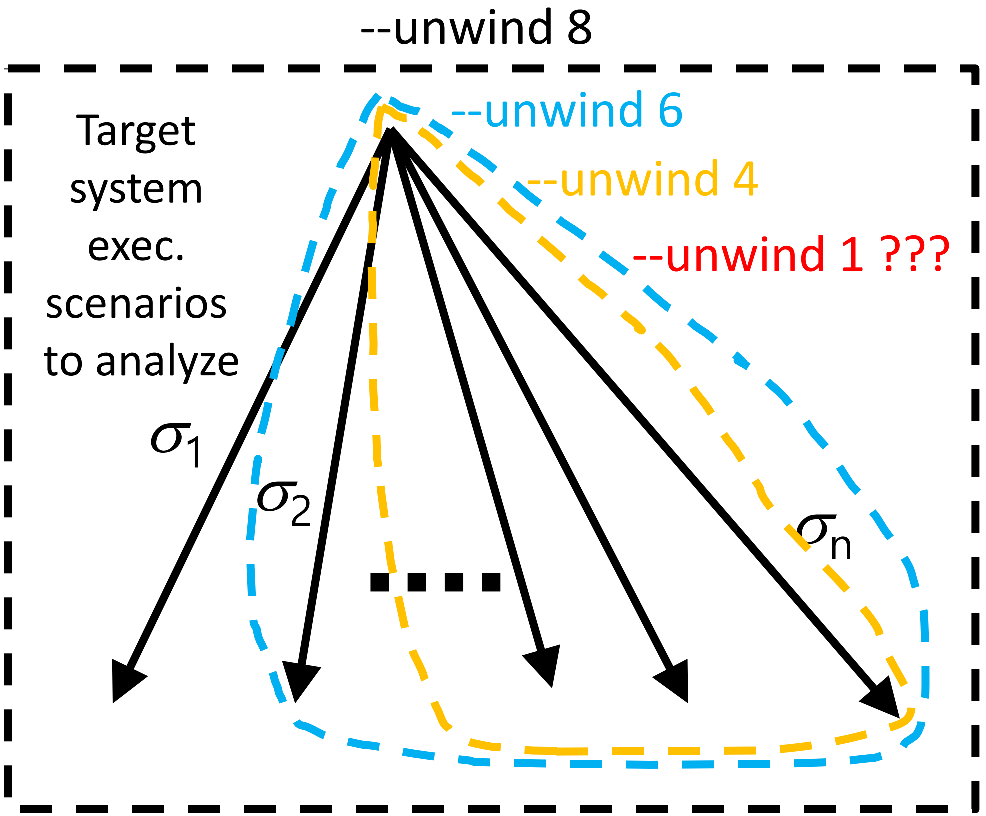
Warning: # of Unwinding Loop (1/2)

```
void f(unsigned int n) { // n can be any number
  int i,x;
  for(i=0; i < 2+ n%7; i++) {
    x = x/ (i-5);// div-by-0 bug
  }//assert(!(i<2+n%7)) or __CPROVER_assume(!(i<2+n%7))
}
```

- Q: What is the maximum # of iteration?
  - A: n_{max}=8
- What will happen if you unwind the loop more than n_{max} times?
  - What will happen if you unwind the loop less than n_{max} times?
    - What if w/ unwinding assertion assert(!(i <2+n%7)) (default behavior of CBMC)?
    - What if w/o unwinding assertion?
    - What if w/ `__cprover_assume(`(!(i <2+n%7))`)`, which is the case w/ `–no-unwinding-assertions`?
- What is the minimum # of iterations?
  - A: n_{min} =2
  - What will happen if you unwind the loop less than n_{min} times w/ `–no-unwinding-assertions`?

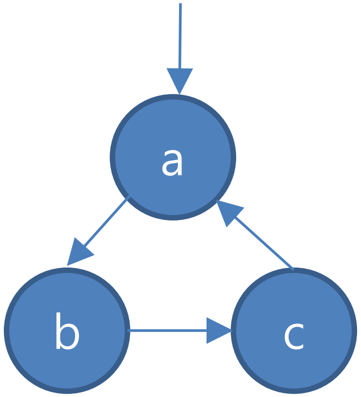
Warning: # of Unwinding Loop (2/2)

```
void f(unsigned int n) {
  int i,x;
  for(i=0; i < 2+ n%7; i++) {
    x = x/ (i-5);// div-by-0 bug
  }//assert(!(i<2+n%7)) or __CPROVER_assume(!(i<2+n%7))
}
```



Note that a bug usually causes a failure even in a small # of loop iteration because a static fault often affects all dynamic execution scenarios (a.k.a., small world hypothesis in model checking)

Model checking (MC) v.s. Bounded model checking (BMC)



a.b.c.a.b.c.a.b.c…

- Target program is finite.
- But its execution is infinite
- MC targets to verify infinite execution
  - Fixed point computation
  - Liveness property check : <> f
    - Eventually, some good thing happens
    - Starvation freedom, fairness, etc
- BMC targets to verify finite execution only
  - No loop anymore in the target program
  - Subset of the safety property (practically useful properties can still be checked)
    - assert() statement

C Bounded Model Checker

- Targeting arbitrary ANSI-C programs
  - Bit vector operators (>>, <<, |, &)
  - Array
  - Pointer arithmetic
  - Dynamic memory allocation
  - Floating #
- Can check
  - Array bound checks (i.e., buffer overflow)
  - Division by 0
  - Pointer checks (i.e., NULL pointer dereference)
  - Arithmetic overflow/underflow
  - User defined assert(cond)
- Handles function calls using inlining
- Unwinds the loops a fixed number of times
- By default, CBMC 5.8 (and later) inserts loop unwinding **assumption** to avoid unsound analysis results

CBMC Options (`cbmc --help`)

- `--function <f>`
  - Set a target function to model check (default: main)
- `--unwind n`
  - Unwinding all loops n-1 times and recursive functions n times
- `--unwindset c::f.0:64,c::main.1:64,max_heapify:3`
  - Unwinding the first loop in `f` 63 times, the second loop in `main` 63 times, and `max_heapify` (a recursive function) 3 times
- `--unwinding-assertions`
  - Convert unwinding assumption `__CPROVER_assume(!(i<10))` into `assert(!(i<10))`
- `--show-loops`
  - Show loop ids which are used in –`unwindset`
- `--bounds-check, --div-by-zero-check, --pointer-check`
  - Check corresponding crash bugs
- `--memory-leak-check, --signed-overflow-check, --unsigned-overflow-check`
  - Check corresponding abnormal behaviors

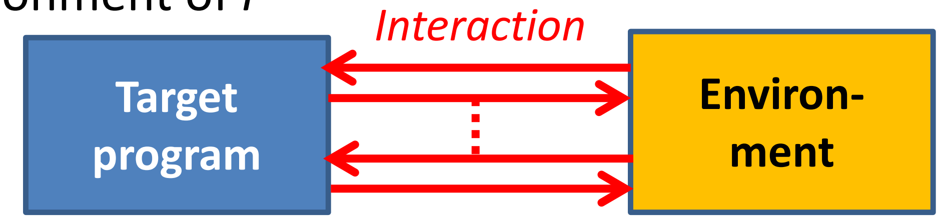
CBMC Options (`cbmc --help`)

- `--cover-assertions`
  - Checks if a user given assertion is reachable. Useful to check if you use __CPROVER_assume() incorrectly or unwind a loop less than minimum number of loop iteration
- `--dimacs`
    - Show a generated Boolean SAT formula in DIMACS format
- `--trace (for cbmc 5.x)`
  - To generate a counter example
- `--unwinding-assertions (for cbmc 5.x)`
  - To enable unwinding assertion

- Example:
  -
```
cbmc --bounds-check –-unwindset c::f.0:64,c::main.1:64,max_heapify:3 --
-–no-unwinding-assertions max-heap.c
```

Procedure of Software Model Checking in Practice

0. With a given C program (e.g.,`int bin-search(int a[],int size_a, int key)`)
1. Define a requirement (i.e., `assert(i>=0 -> a[i]== key)` where `i` is a return value of `bin-search()`)
2. Model an **environment/input space** of the target program, which is non-deterministic
   - Ex1. pre-condition of `bin-search()` such as input constraints
   - Ex2. For a target client program *P*, a server program should be modeled as an environment of *P*



A program execution can be viewed as a sequence of interaction between the target program and its environment

3. Tuning model checking parameters (i.e. loop bounds, etc.)

Modeling an Non-deterministic Environment with CBMC

1. Models an environment/input space using **non-deterministic values**
   1. By using undefined functions (e.g., x= non-det();)
   2. By using uninitialized local variables (e.g., f() { int x; …})
   3. By using function parameters (e.g., f(int x) {…})
2. Refine/restrict an environment with __CPROVER_assume(assume)
   - CBMC generates $P \land$ assume $\land \neg A$

```
void foo(int x) {
   __CPROVER_assume
  (0<x && x<10);
   x=x+1;;
   assert (x*x <= 100);
}
```

```
void bar() {
   int y=0;
   __CPROVER_assume
   ( y > 10);
   assert(0);
}
```

```
int x = nondet();
void bar() {
   int y;
   __CPROVER_assume
  (0<x && 0<y);
   if(x < 0 && y < 0)
      assert(0);
}
```