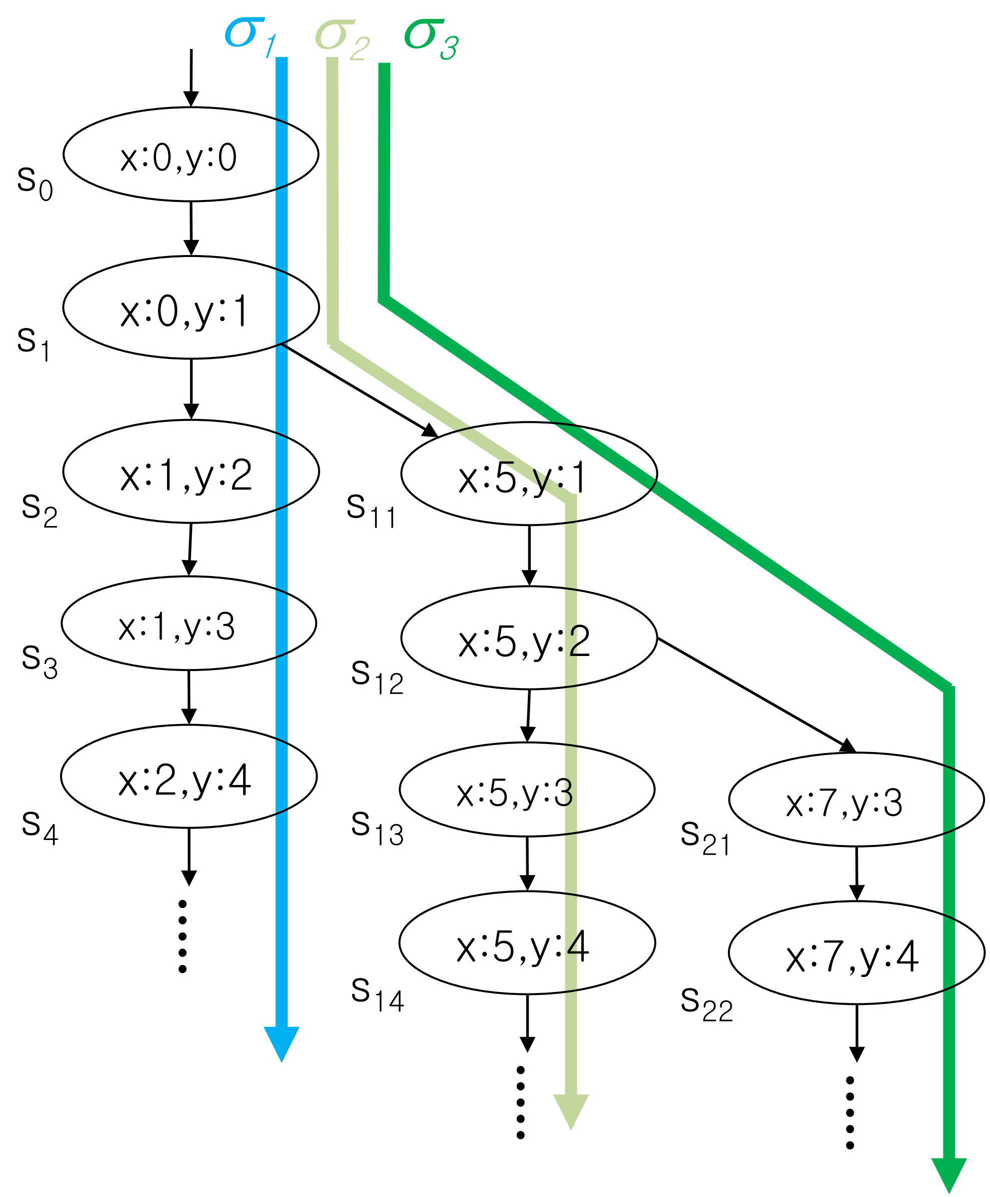
Software Model Checking

Moonzoo Kim

Operational Semantics of Software

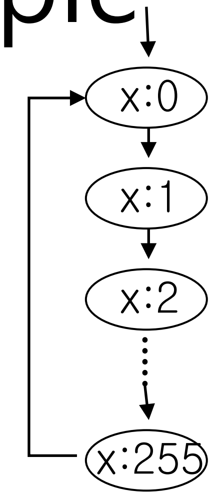
- A system has its semantics as a set of system executions σ's

- A system execution σ is a sequence of states $s_0 s_1 \ldots$
  - A state has an environment $\rho_s: Var \text{-> } Val$

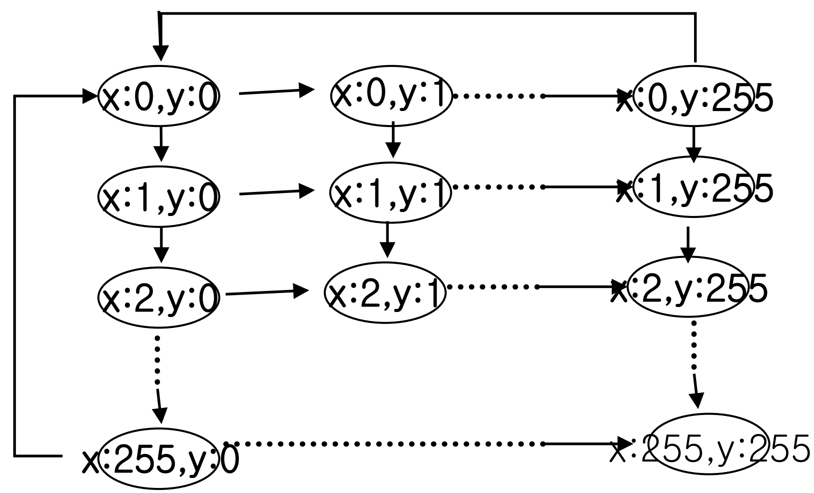


Example

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



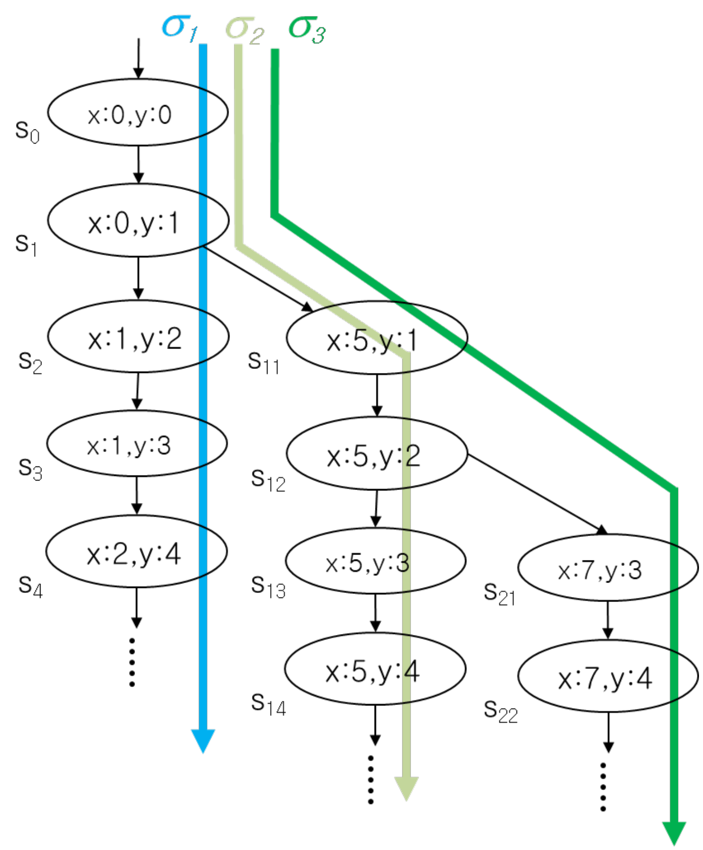

```
active type A() {
byte x;
again:
    x=x+1;;
    goto again;
}

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

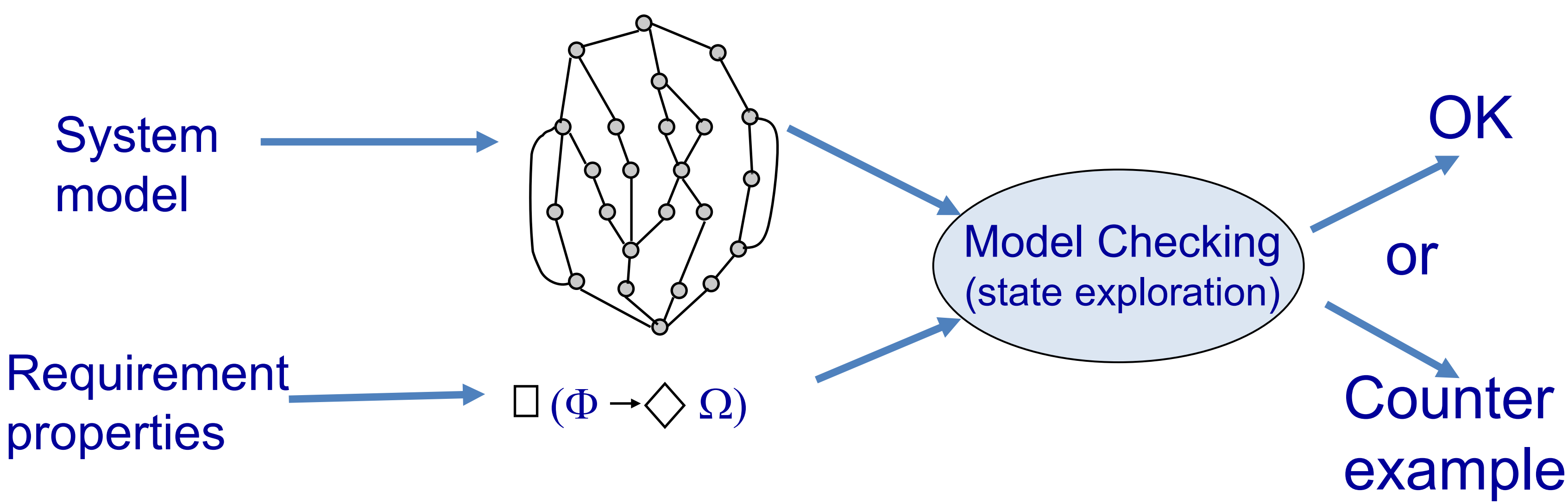
Bug Detection vs. Verification

- Bug detection (testing):
  - a given assert statement (at a given code location) is violated
    - proof: **for a some execution** like σ_1, a given assert is violated
      - ex. σ_1 violates the assert(2x != y) at s_2 and s_4
- Verification (model checking):
  - a given assert statement will be never violated (i.e., **always** satisfied)
    - proof: **for every possible execution** σ_1, σ_2, σ_3, and so on, a given assert is satisfied
      - ex. there is no execution σ such that assert(x >= 0) is violated.



Verification: State Exploration Method

- Model checking
  - Generate possible states from the model/program and then check whether given requirement properties are satisfied within the state space
    - On-the-fly v.s. generates all
    - Symbolic states v.s. explicit state
    - Model based v.s. code based



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 10^5
Sifakis / Quielle
1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking 10^{100}
Burch, Clarke, Dill, McMillan
1992 SMV: Symbolic Model Verifier
McMillan

1998 Bounded Model Checking using SAT 10^{1000}
Biere, Clarke, Zhu
2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith

Example. Sort (1/2)

- Suppose that we want to verify `sort(unsigned char* a)` on an array of 5 elements each of which is 1 byte long
  - `unsigned char a[5];` // 40 bits

9	14	2	200	64

- We wants to **verify** if `sort()` works correctly on every unsigned char array `a[5]`

```
main() {
  assign all possible values to a;
  sort(a);
  assert(a[0]<=a[1]<=a[2]<=a[3]<=a[4]);}
```

a) 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…

b) Binary Decision Diagram (BDD) based symbolic model checker (ex. NuSMV) takes 100 MB in 100 sec on Intel Xeon 5160 3Ghz machine

c) **Bounded model checker (i.e., CBMC)** takes less than 100 MB in 1 sec

Bounded Model Checking

```c
grepbuf(beg, lim)
      char *beg;
      char *lim;
{
  int nlines, n;
  register char *p, *b;
  char *endp;

  nlines = 0;
  p = beg;
  /* just for Understand analysis by YHK */
  EGexecute(p, lim-p, &endp);
  Fexecute(p, lim-p, &endp);
  while ((b = (*execute)(p, lim - p, &endp)) != 0)
    {
      /* Avoid matching the empty line at the end of the buffer. */
      if (b == lim && ((b > beg && b[-1] == '\n') || b == beg))
  break;
      if (!out_invert)
  {
    prtext(b, endp, (int *) 0);
    nlines += 1;
  }
      else if (p < b)
  {
    prtext(p, b, &n);
    nlines += n;
  }
      p = endp;
    }
  if (out_invert && p < lim)
    {
      prtext(p, lim, &n);
      nlines += n;
    }
  return nlines;
}
```

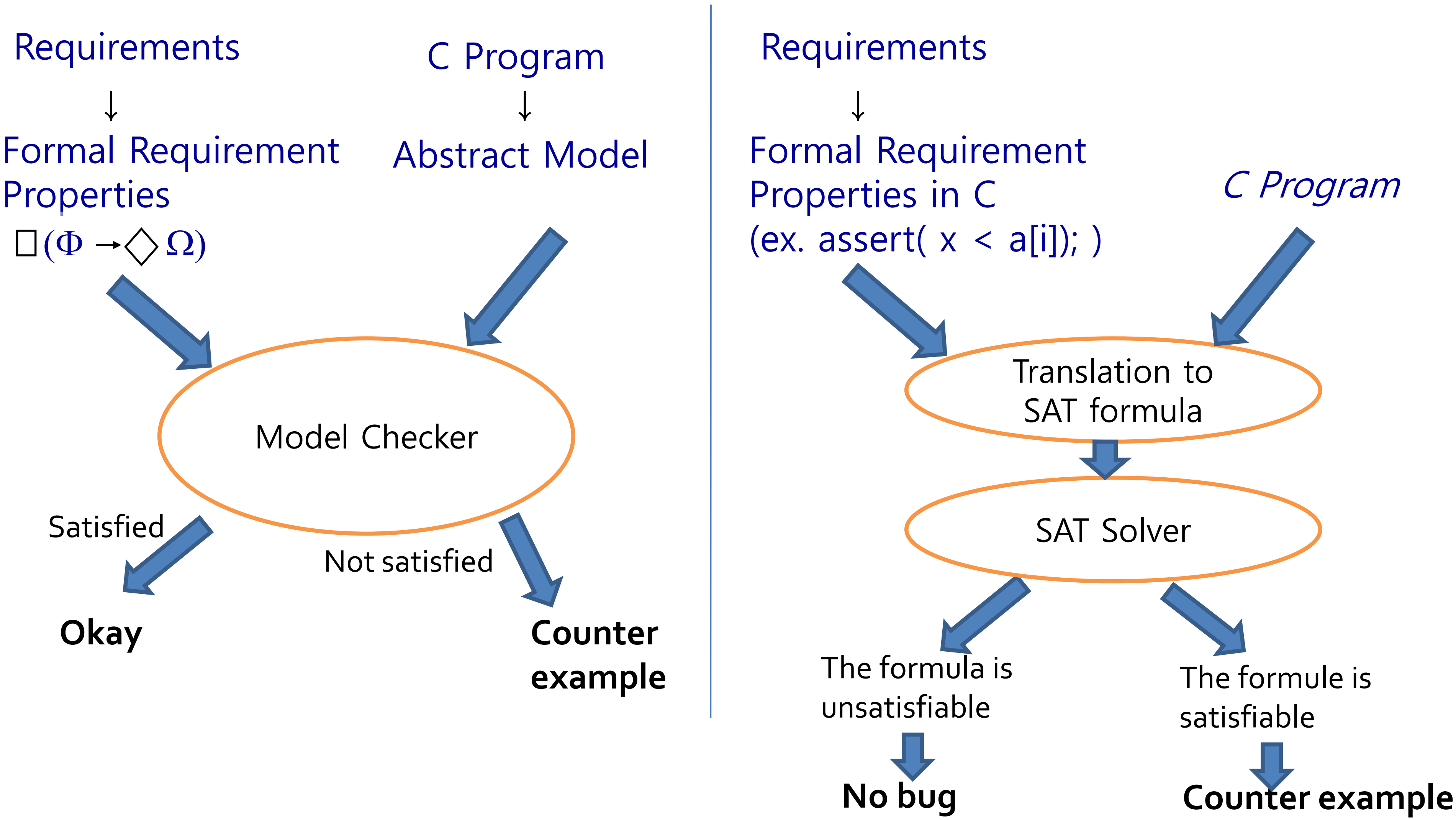
C program source code

$$
\begin{aligned}
&(\bar{a} \vee m \vee u) \wedge (a \vee n \vee u) \wedge (\bar{a} \vee r \vee x) \wedge (\bar{c} \vee \bar{e} \vee s) \\
&\wedge (c \vee \overline{m} \vee \overline{w}) \wedge (\bar{c} \vee p \vee x) \wedge (c \vee q \vee s) \wedge (e \vee p \vee s) \\
&\wedge (e \vee q \vee \bar{y}) \wedge (e \vee r \vee y) \wedge (\bar{e} \vee r \vee z) \wedge (\bar{g} \vee r \vee x) \\
&\wedge (g \vee v \vee \bar{y}) \wedge (m \vee \bar{n} \vee u) \wedge (m \vee \bar{o} \vee \bar{u}) \wedge (m \vee o \vee v) \\
&\wedge (\overline{m} \vee \bar{q} \vee s) \wedge (\overline{m} \vee \bar{r} \vee \bar{s}) \wedge (m \vee \bar{u} \vee \bar{v}) \wedge (\overline{m} \vee x \vee \bar{z}) \\
&\wedge (\bar{n} \vee r \vee \bar{y}) \wedge (o \vee r \vee \overline{w}) \wedge (\bar{p} \vee q \vee s) \wedge (r \vee \overline{w} \vee \bar{x}) \\
&\wedge (r \vee w \vee \bar{y}) \wedge (r \vee w \vee \bar{z})
\end{aligned}
$$

Boolean logic formula
(propositional logic)

A key idea: representing every
(bounded) execution path
as a pure Boolean logic formula

Overview of SAT-based Bounded Model Checking



Example. Sort (2/2)

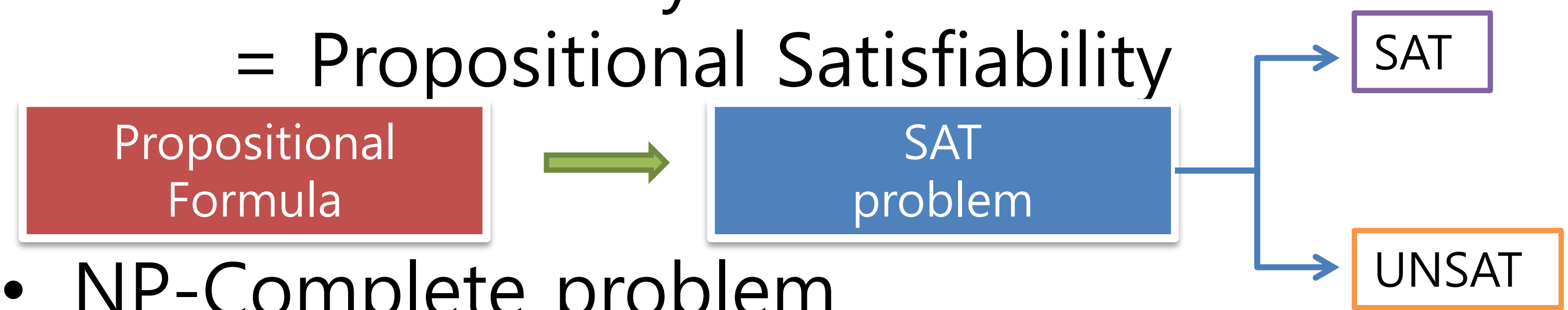
```
#include <stdio.h>
#define N 5
int main {
    unsigned char data[N], i, j, tmp;
    /* Assign random values to the array*/
    for (i=0; i<N; i++){
        data[i] = nondet_int();
    }
   /* It misses the last element, i.e., data[N-1]*/
    for (i=0; i<N-1; i++)
        for (j=i+1; j<N-1; j++)
            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 (CBMC v 5.11) converts the (fixed version) of the left code to a Boolean formula
    - Total 2277 CNF clause with 905 boolean propositional variables
    - Theoretically, 2^{905} choices should be evaluated!!!

N	Exec time (CBMC 5.11 on i5-9600K)	Mem	# of var	# of clause
10	50 sec	43 M	2,895	9,382
20	1317 sec	150 M	10,175	37,842
40	2 hours	900 M	37,935	151,762
100	40 hours	18 GB	226,815	949,522

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=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)

$x_1 == x_0 + y_0;$
$\text{if } (x_1 != 1)$
$\quad x_2 == 2;$
else
$\quad x_3 == x_1 + 1;$

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 <u>feasible</u> 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)

```
x1==x0+y0;
if (x1!=1)
   x2==2;
else
   x3==x1+1;
x4==(x1!=1)?x2:x3;
assert(x4<=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 $P \wedge \neg A$?

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)

```
x1==x0+y0;
if (x1!=1)
   x2==2;
else
   x3==x1+1;
x4==(x1!=1)?x2:x3;
assert(x4<=3);
```

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

$A \equiv x_4 <= 3$

<u>Observations on the code</u>

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

<u>Observations on the *P*</u>

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

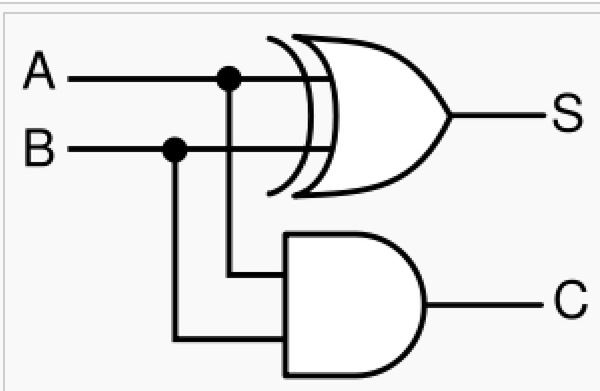
Model Checking as a SAT problem (6/6)

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

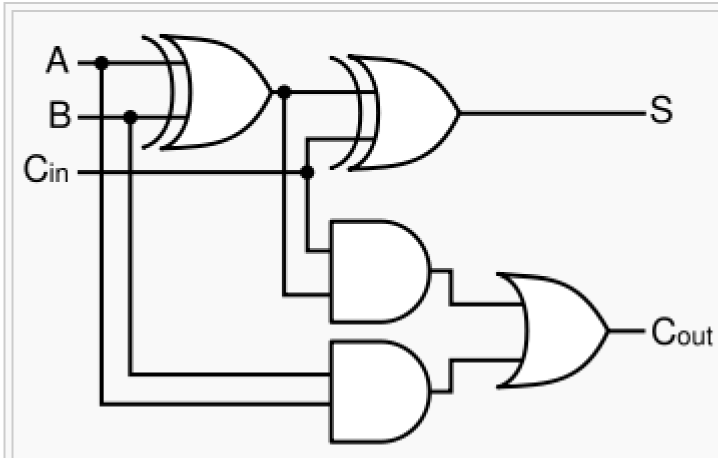
- Example of arithmetic encoding into pure propositional formula

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 $\equiv$ ($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



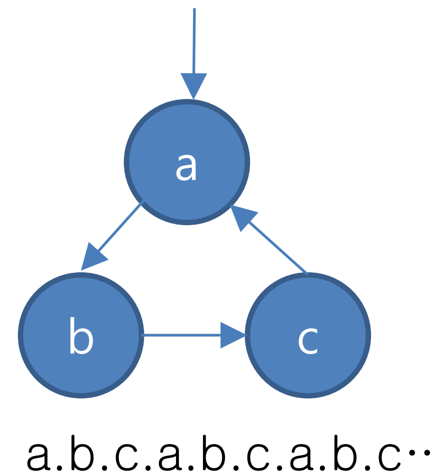
Full adder circuit diagram
Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}

Example

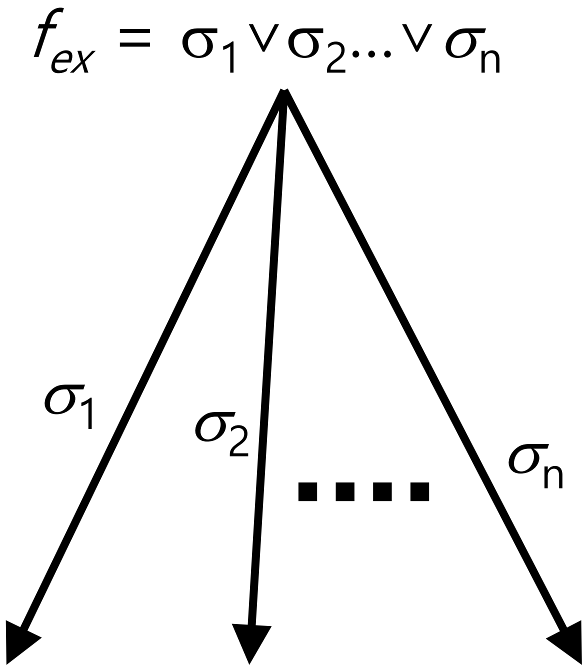
```
/* Assume that x and y are 2 bit
unsigned integers */
/* Also assume that x+y <= 3 */
void f(unsigned int y) {
    unsigned int x=1;
    x=x+y;
    if (x==2)
        x+=1;
    else
        x=2;
    assert(x ==2);
}
```

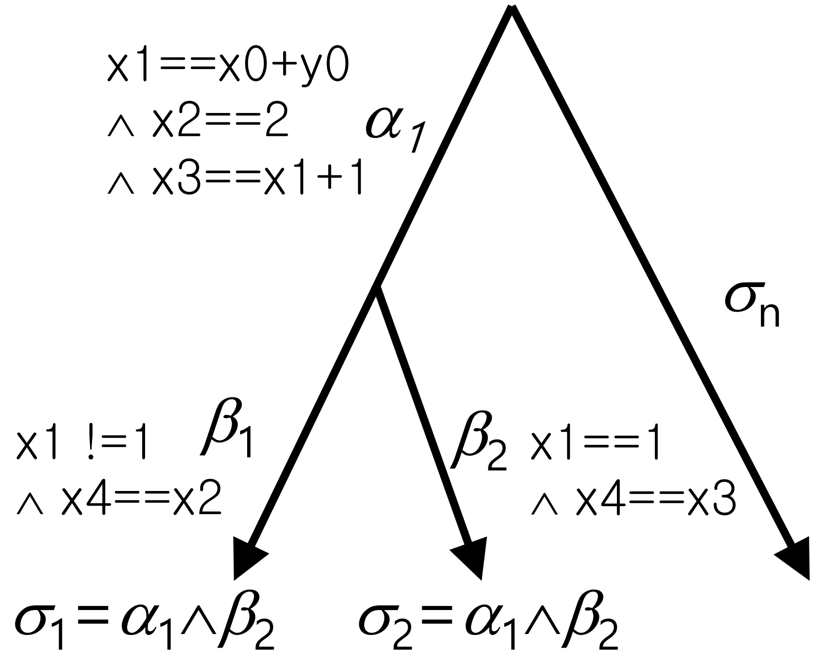
Advanced Issues on Bounded Model Checking

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



- 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





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

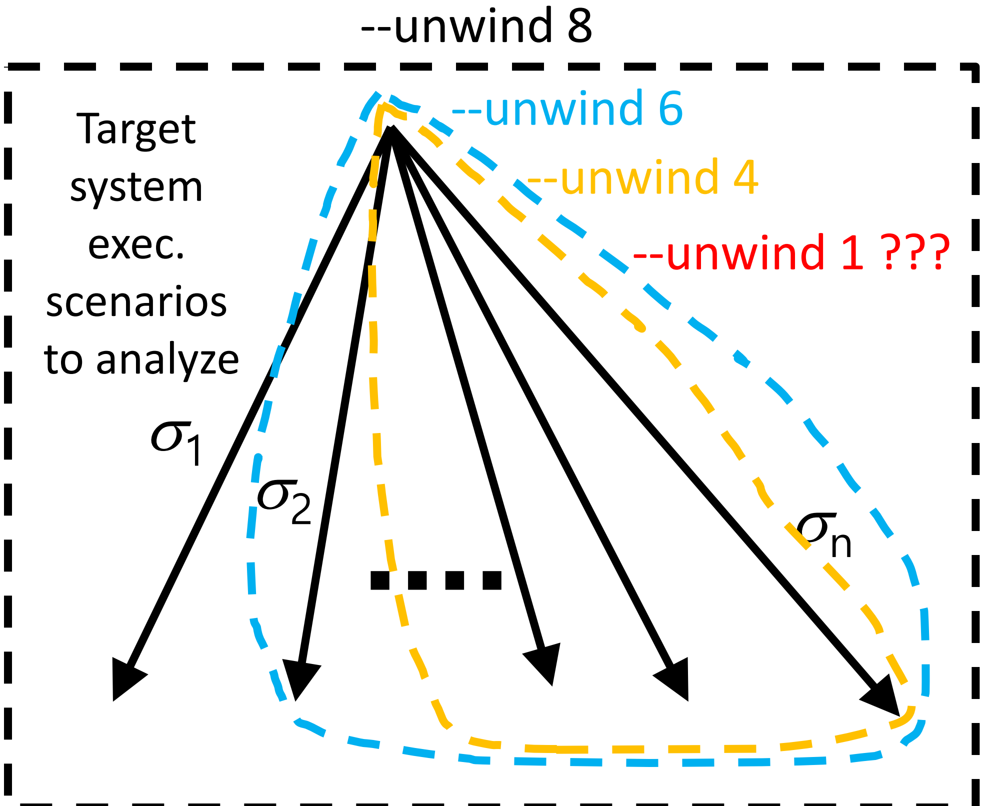
Warning: # of Unwinding Loop (1/2)

```
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))?
    - What if w/o unwinding assertion?
    - What if w/ `__cprover_assume`((!(i <2+n%7))) ?
- What is the minimum # of iterations?
  - A: n_{min} =2
  - What will happen if you unwind the loop less than n_{min} times?

Warning: # of Unwinding Loop (2/2)

```
1:void f(unsigned int n) {
2:   int i,x;
3:   for(i=0; i < 2+ n%7; i++) {
4:     x = x/ (i-5);// div-by-0 bug
5:   }//assert(!(i<2+n%7)) or __CPROVER_assume(!(i<2+n%7))
6:}
```



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)