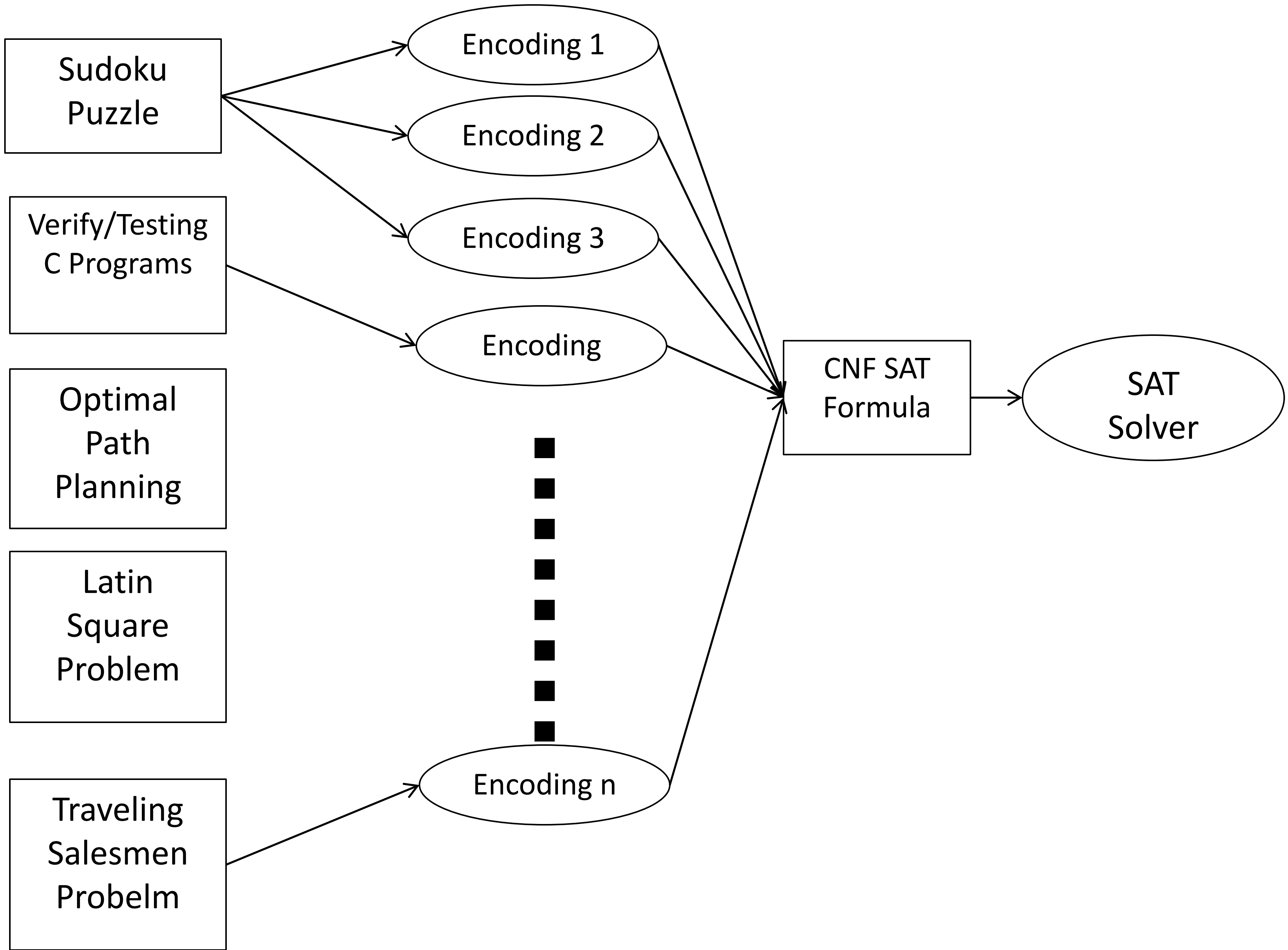
Application of Propositional Logic II - How to Test/Verify my C program?

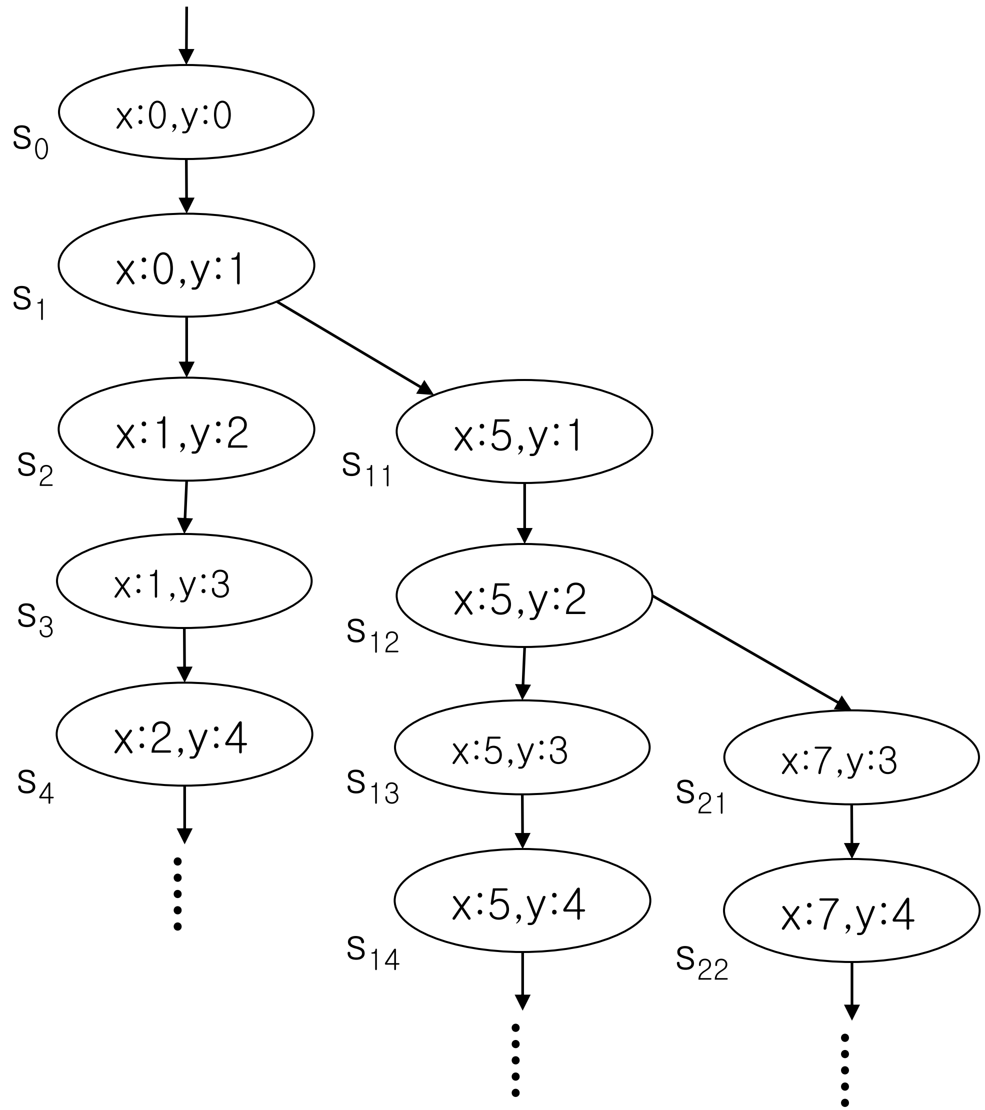
Moonzoo Kim

Solving Various Problems using SAT Solver



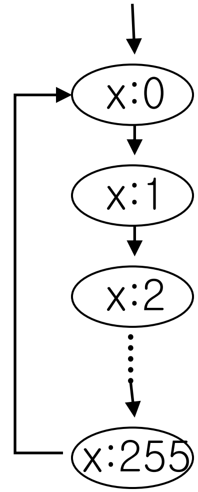
Operational Semantics of Software

- 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



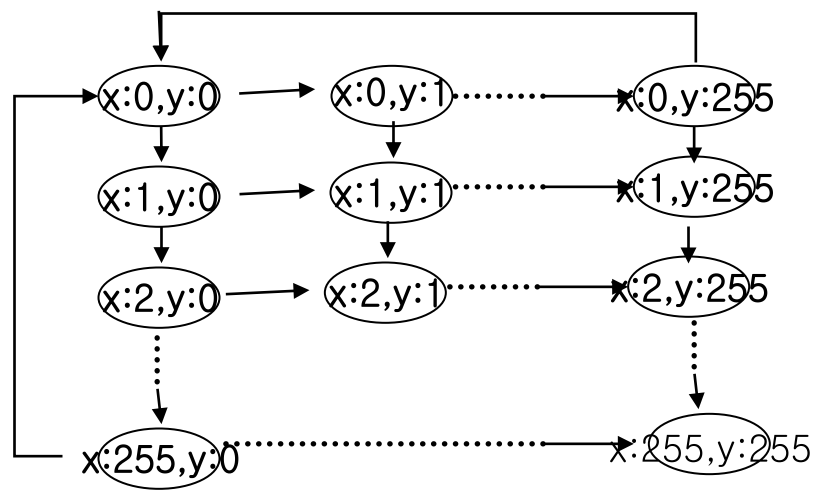
Example of Model Checking

```
thread A() {
unsigned char x;
again:
    x++;
    goto again;
}
```




```
thread A() {
unsigned char x;
again:
    x++;
    goto again;
}

thread B() {
unsigned char 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

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)

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 wants 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} (= 4×10^9 = 4G) 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)

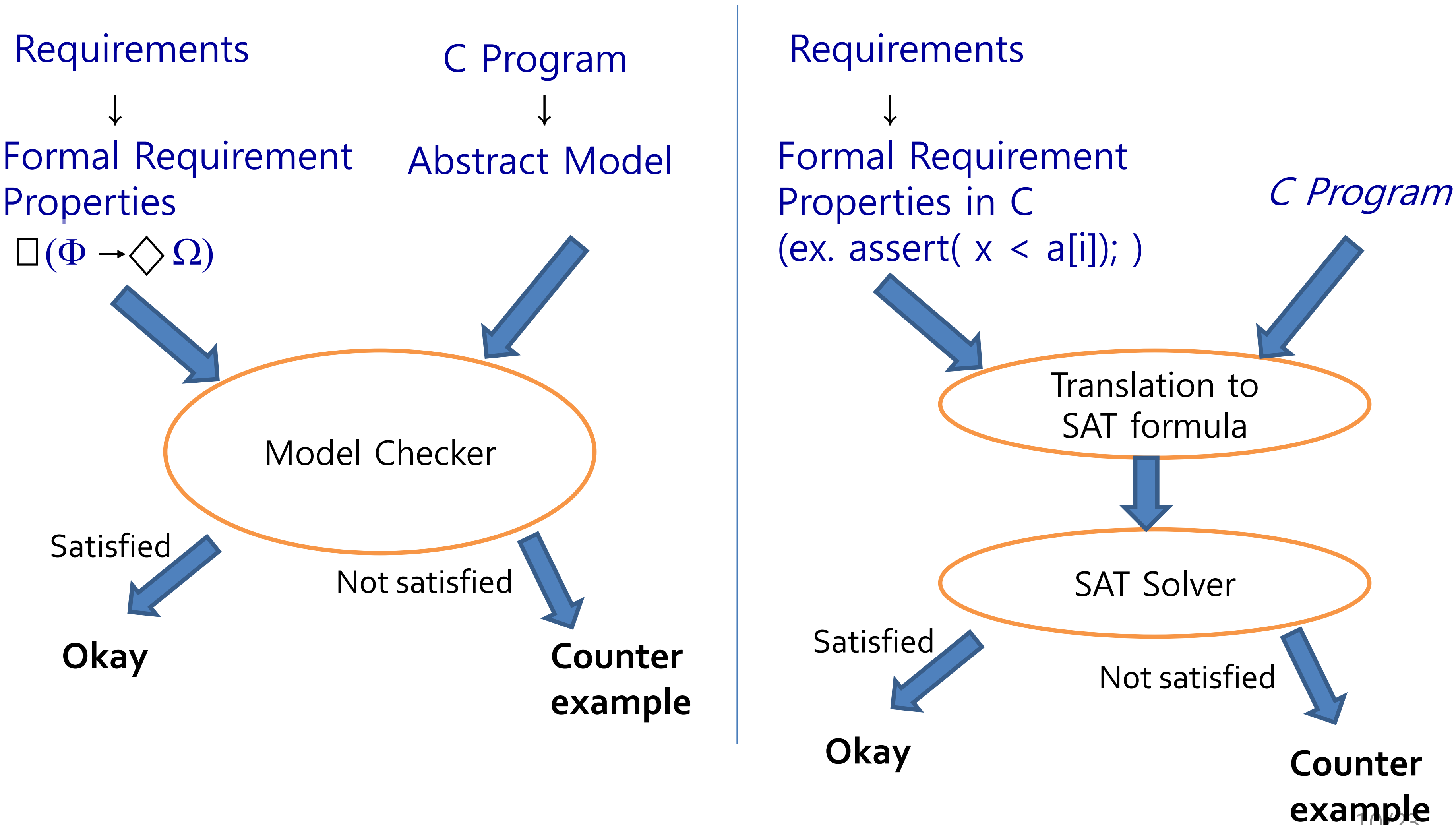
```
#include <stdio.h>
#define N 5
int main(){
    int data[N], i, j, tmp;
    /* Assign random values to the array*/
    for (i=0; i<N; i++){
        data[i] = nondet_int();
    }
    /* It misses the last element, i.e., data[N-1]*/
    for (i=0; i<N-1; i++)
        for (j=i+1; j<N-1; j++)
            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 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



Software Model Checking as a SAT problem (1/4)

- 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()`

Software Model Checking as a SAT problem (2/4)

- Unwinding Loop

Original code

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

Unwinding the loop 1 times

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

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 (3/4)

- 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)

$x_1=x_0+y_0;$
$\text{if } (x_1 != 1)$
$\quad x_2=2;$
else
$\quad x_3=x_1+1;$
$x_4=(x_1!=1)?x_2:x_3;$
$\text{assert}(x_4<=3);$

Generate constraints

$C \equiv x_1=x_0+y_0 \wedge x_2=2 \wedge x_3=x_1+1 \wedge (x_1!=1 \wedge x_4=x_2 \vee x_1=1 \wedge x_4=x_3)$
$P \equiv x_4 <= 3$

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

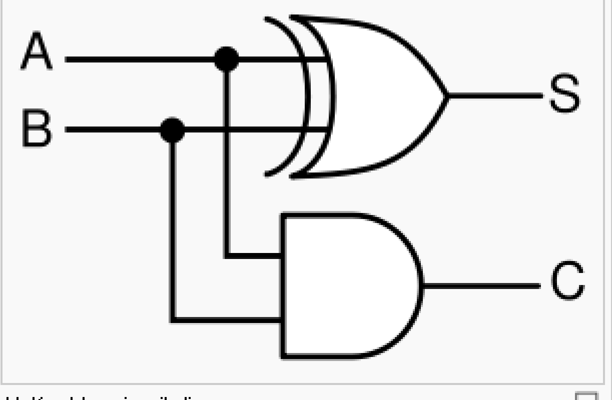
$C \wedge \neg P$ is converted to Boolean logic using a bit vector representation for the integer variables $y_0, x_0, x_1, x_2, x_3, x_4$

Model Checking as a SAT problem (4/4)

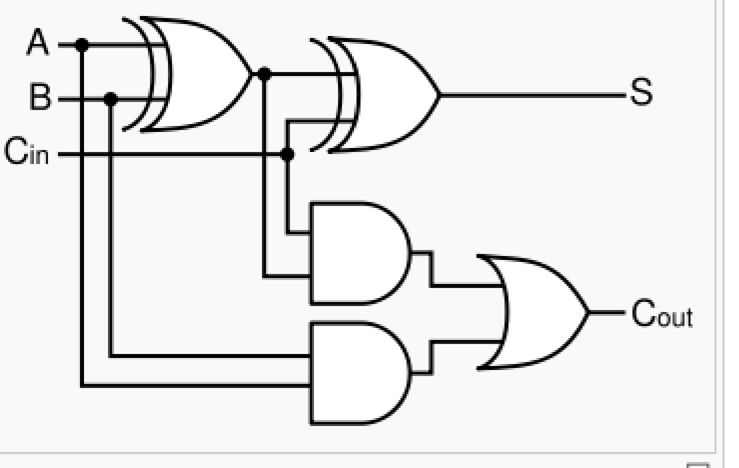
- Example of arithmetic encoding into pure propositional formula

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

  $$C \equiv z=x+y \equiv \quad (z_0 \leftrightarrow (x_0 \oplus y_0) \oplus ((x_1 \wedge y_1) \vee (((x_1 \oplus y_1) \wedge (x_2 \wedge y_2))))$$
  $$\wedge\ (z_1 \leftrightarrow (x_1 \oplus y_1) \oplus (x_2 \wedge y_2))$$
  $$\wedge\ (z_2 \leftrightarrow (x_2 \oplus y_2))$$



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);
}
```

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

Modeling with CBMC

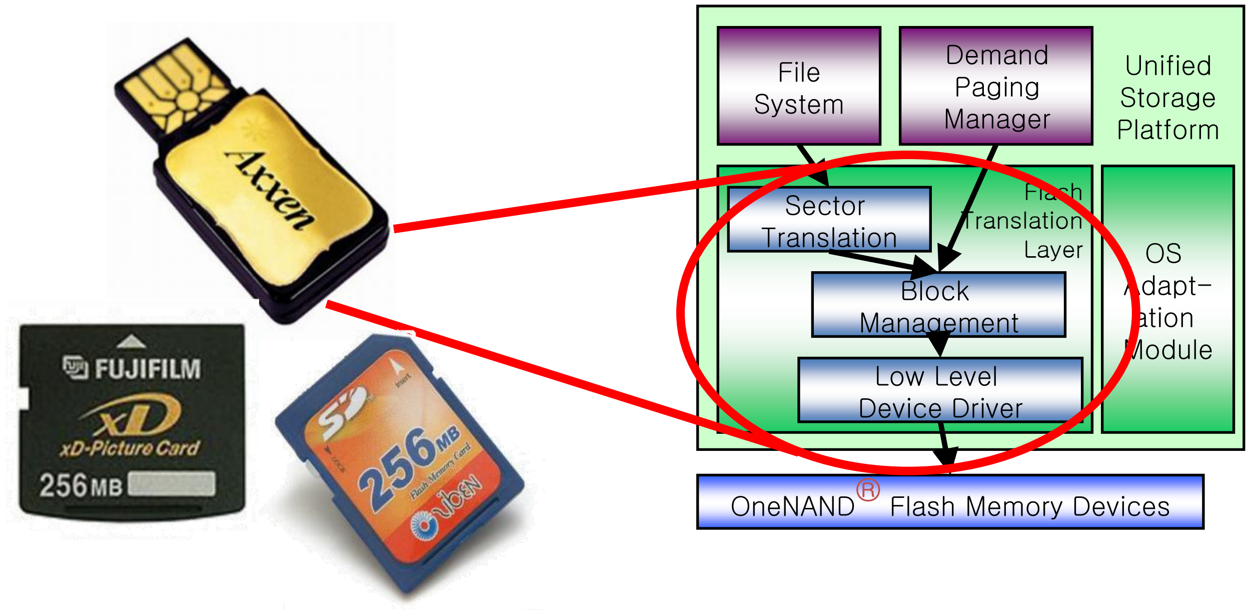
- Models an environment (i.e., various scenarios) using non-determinism
  1. By using undefined functions
  2. By using uninitialized local variables
  3. By using function parameters
  4. By explicitly using `__CPROVER_assume()`

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

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

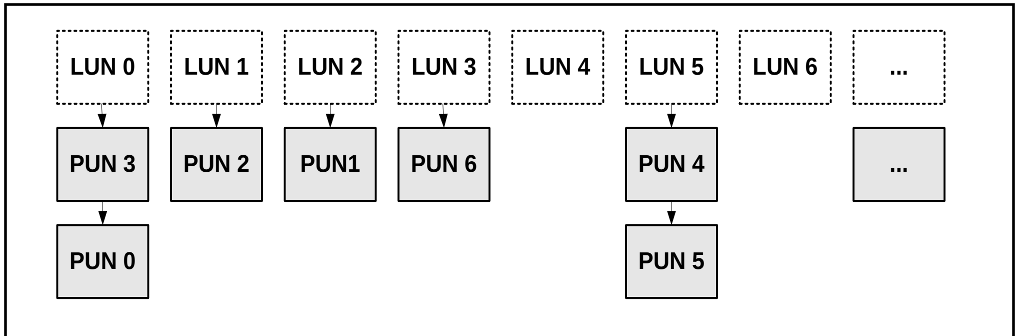
```
int x = nondet();
bar() {
    int y;
    __CPROVER_assume
   (0<x && 0<y);
    if(x < 0 && y < 0)
        assert(0);
}
```

Industrial Application of CBMC [ASE'08, TSE'11]

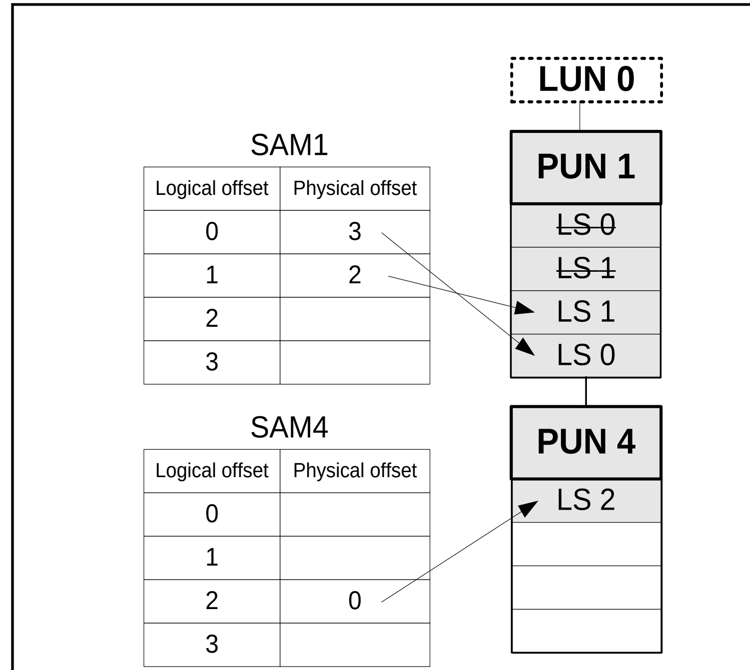


- In 2007, Samsung requested to debug the device driver for the OneNAND™ flash memory
- We reviewed the requirement specifications, the design documents, and C code to identify code-level properties to check.
- Then, we applied CBMC (C Bounded Model Checker) to check the properties
  - Found several bugs
  - Provided high confidence in multi-sector read operation through exhaustive exploration

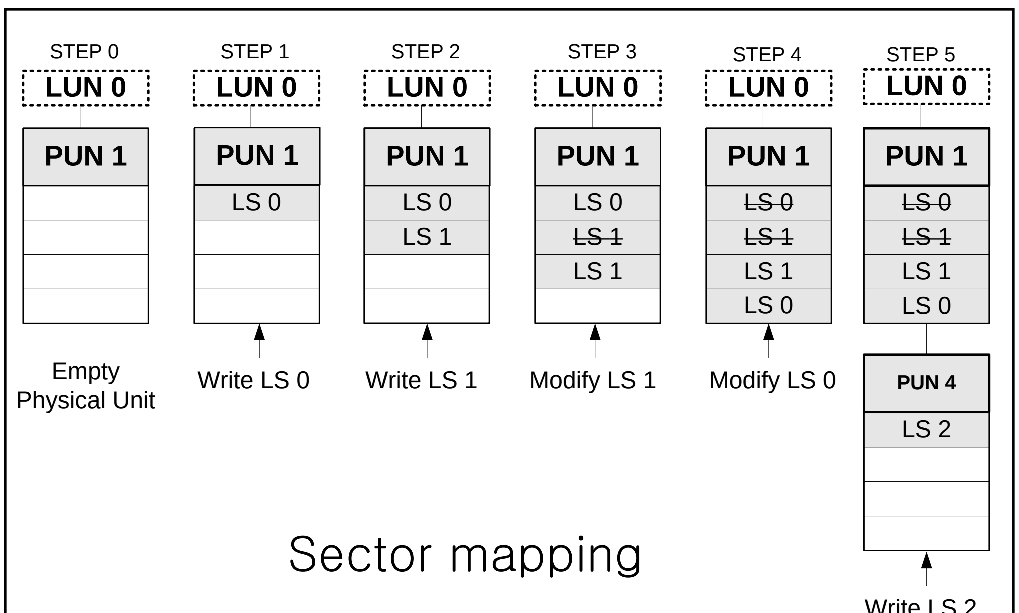
Logical to Physical Sector Mapping



1:N mapping from a LUN to PUNs



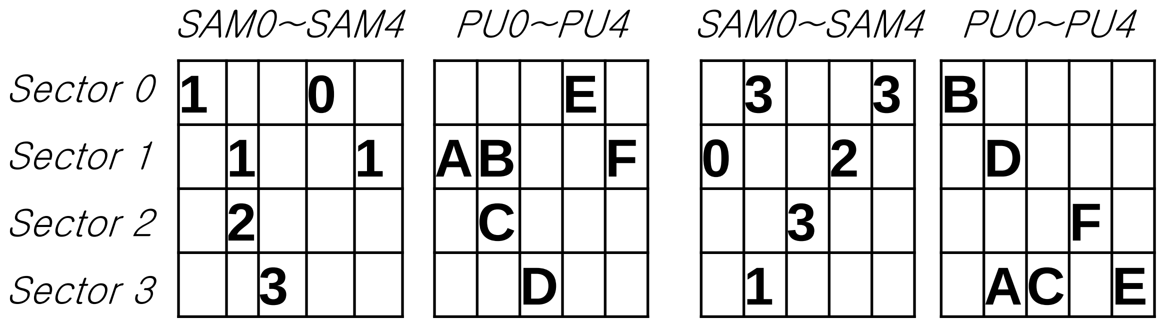
Sector Allocation Map (SAM)



Sector mapping

- In flash memory, logical data are distributed over physical sectors.

Multi-sector Read Operation (MSR)



a) A distribution of "ABCDEF"

b) Another distribution of "ABCDEF"

- MSR reads adjacent multiple physical sectors once in order to improve read speed
  - MSR is 157 lines long, but highly complex due to its 4 level loops
- We built a small test environment for MSR
  - The test environment contains only upto 10 physical units
  - The test environment should follow constraints, which are described by _CPROVER_assume(Boolean exp) statement
    - SAM tables and PUs should correspond each other
    - For each logical sector, at least one physical sector that has the same value exists

Performance Comparison for Verifying Multisector Read