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`) (1/2)

- `--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 f.0:64,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
- `--show-loops`
  - Show loop ids which are used in `–unwindset`
- `--trace`
  - Generate a counter example
- `--trace-show-code`
  - Show original code in plain trace
- Example:
  - `cbmc --unwindset f.0:64,main.1:64,max_heapify:3  max-heap.c`

CBMC Options (`cbmc --help`) (2/2)

- `--unwinding-assertions`
  - Convert unwinding assumption `__CPROVER_assume(!(i<10))` into `assert(!(i<10))`
- `--dimacs`
  - Show a generated Boolean SAT formula in DIMACS format
- `--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

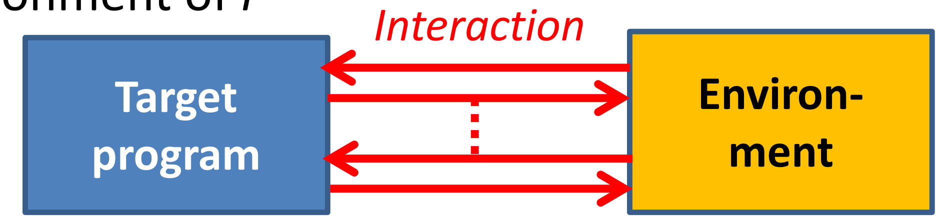
Loop Unwinding Example

```
1 int main() {
2     int sum=0, i=0;
3     for(i=0; i < 3; i ++ ) {
4         sum+=i;
5     } /* assert(! (i<3)); */
6     assert(0); // cbmc --unwind 3 does NOT report the violation
7                // cbmc --unwind 4 does     report the violation
8                // cbmc --unwindset main.0:4 report the violation
9 }
```

```
moonzoo@verifier3:$ cbmc --unwinding-assertions --unwind 3  loop1.c
...
Solving with MiniSAT 2.2.1 with simplifier
72 variables, 11 clauses
...
Runtime decision procedure: 0.000262811s

** Results:
[main.unwind.0] unwinding assertion loop 0: FAILURE

loop1.c function main
[main.assertion.1] line 6 assertion 0: SUCCESS

** 1 of 2 failed (2 iterations)
VERIFICATION FAILED
```

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 <u>non-deterministic</u>
   - 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 assigning a return value of any undefined function (e.g., x= non-det();)
   2. By using uninitialized local variables (e.g., f() { int x; int a[10];...})
   3. By using function parameters (e.g., f(int x) {...})
2. Refine/restrict an environment with __CPROVER_assume(assume)
   - CBMC generates $P \wedge$ assume $\wedge \neg A$

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

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

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

Key Difference between Manual Testing and Model Checking

- Manual testing (unit testing)
  - A user should test one concrete execution scenario by checking a pair of concrete input values and the expected concrete output values
- Model checking (concolic testing)
  - A user should imagine all possible execution scenarios and model a general environment that can enable all possible executions
  - A user should describe general invariants on input values and output values

Ex1. Binary Search

```c
#include <stdio.h>
#define N 3

char  bin_search(char  a[], char  size, char  x) {
  char  low=0;

  char  high=size-1;

  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    char  mid = low + (high - low) / 2;

    if (a[mid] == x) return mid;

    if (a[mid] < x)  low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}


int main() {
  char  a[N],i, key, result;
  for(i=0; i <N; i++) {
    a[i] = non_det();
    __CPROVER_assume(i==0 || a[i-1]<a[i]); //a[] should be sorted
  }

   key = non_det();
   result = bin_search(a,N,key);

   if(result!= -1) {
     assert(a[result] == key);
   } else {
     for (i=0; i < N; i++)
        assert(a[i]!=key);
   }
}
```

Ex2. Circular Queue of Positive Integers

```c
#include<stdio.h>
#define SIZE 12
#define EMPTY 0

// We assume that q[] is
// empty if head==tail
unsigned int q[SIZE],head,tail;

void enqueue(unsigned int x)
{
    q[tail]=x;
    tail=(++tail)%SIZE;
}

unsigned int dequeue() {
    unsigned int ret;
    ret = q[head];
    q[head]= EMPTY;
    head= (++head)%SIZE;
    return ret;}
```

Step 1)

0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	15	6	9	8	4	0
						head=6					tail=11

Step 2) After adding 17, 3, 5 to the queue

3	5	0	0	0	0	15	6	9	8	4	17
		tail=2				head=6					

Step 3) After poping up 15 from the queue

3	5	0	0	0	0	0	6	9	8	4	17
		tail=2					head=7				

```c
#include<stdio.h>
#define SIZE 12
#define EMPTY 0

unsigned int q[SIZE],head,tail;

void enqueue(unsigned int x)
{
    q[tail]=x;
    tail=(++tail)%SIZE;
}

unsigned int dequeue() {
    unsigned int ret;
    ret = q[head];
    q[head]=0;
    head= (++head)%SIZE;
    return ret;
}
```

```c
// Initial random queue setting following the script
void environment_setup() {
    int i;
    for(i=0;i<SIZE;i++) { q[i]=EMPTY;}

    head=non_det();
    __CPROVER_assume(0<= head && head < SIZE);

    tail=non_det();
    __CPROVER_assume(0<= tail && tail < SIZE);

    if( head < tail)
        for(i=head; i < tail; i++) {
            q[i]=non_det();
            __CPROVER_assume(0< q[i]);
        }
    else if(head > tail) {
        for(i=0; i < tail; i++) {
            q[i]=non_det();
            __CPROVER_assume(0< q[i]);
        }
        for(i=head; i < SIZE; i++) {
            q[i]=non_det();
            __CPROVER_assume(0< q[i]);
        }
    } // We assume that q[] is empty if head==tail
}
```

```c
void enqueue_verify() {
    unsigned int x, old_head, old_tail;
    unsigned int old_q[SIZE], i;
    __CPROVER_assume(x>0);

    for(i=0; i < SIZE; i++) old_q[i]=q[i];
    old_head=head;
    old_tail=tail;

    enqueue(x);

    assert(q[old_tail]==x);
    assert(tail== ((old_tail +1) % SIZE));
    assert(head==old_head);
    for(i=0; i < old_tail; i++)
            assert(old_q[i]==q[i]);
    for(i=old_tail+1; i < SIZE; i++)
            assert(old_q[i]==q[i]);
}
```

```c
void dequeue_verify() {
    unsigned int ret, old_head, old_tail;
    unsigned int old_q[SIZE], i;

    for(i=0; i < SIZE; i++) old_q[i]=q[i];
    old_head=head;
    old_tail=tail;
    __CPROVER_assume(head!=tail);

    ret=dequeue();

    assert(ret==old_q[old_head]);
    assert(q[old_head]== EMPTY);
    assert(head==(old_head+1)%SIZE);
    assert(tail==old_tail);
    for(i=0; i < old_head; i++)
            assert(old_q[i]==q[i]);
    for(i=old_head+1; i < SIZE; i++)
            assert(old_q[i]==q[i]);}
```

```c
int main() {// cbmc q.c –unwind
SIZE+2
    environment_setup();
    enqueue_verify();}
```

```c
int main() {// cbmc q.c –unwind
SIZE+2
    environment_setup();
    dequeue_verify();}
```

Model checking v.s. random sequence of method calls

- You may try to test the circular queue code by calling enqueue and dequeuer randomly
  - Ex.

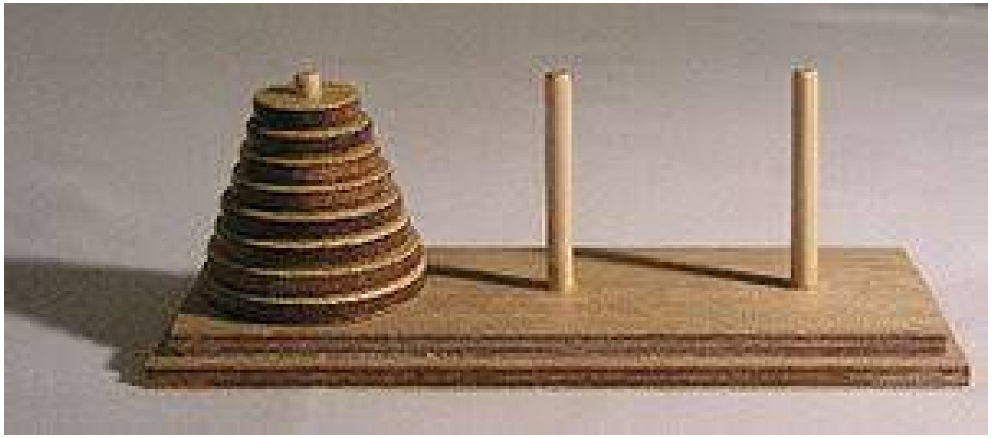
```
void test1(){e(10);r=d();assert(r==10);)
void test2(){e(10);e(20);d();e(30);r=d();
   assert(r==20);)
void test3(){...} …
```

- Note that model checking covers all test scenarios of the above random method sequence calls
  - Note that a random sequence of method calls just provide ONE input instance/state to the circular queue
  - MC provides ALL input instances/states through environment/input space modeling

Ex3. Tower of Hanio

Write down a C program to solve the Tower of Hanoi game (3 poles and 3 disks) by using CBMC

– Hint: you may non-deterministically select the disk to move
– Find the shortest solution by analyzing counter examples. Also explain why your solution is the shortest one.
    • Use non-determinism and `__CPROVER_assume()` properly for the moving choice
    • Use `assert` statement to detect when all the disks are moved to the destination



```
// cbmc hanoi3.c –unwind 7
// Increase n from 1 in –unwind [n] to find the shortest solution
signed char disk[3][3] = {{3,2,1},{0,0,0},{0,0,0}};

// The position where the top disk is located at.
// If the pole has no disk, top is -1
char top[3]={2,-1,-1};

int main() {
    unsigned char dest, src;

    while(1) {
    src = non_det();
    __CPROVER_assume(src==0 || src==1 || src==2);
    __CPROVER_assume(top[src] != -1);

    dest= non_det();
    __CPROVER_assume((dest==0||dest==1||dest==2) && (dest!= src));
    __CPROVER_assume(top[dest]==-1 ||
                     (disk[src][top[src]] < disk[dest][top[dest]]));

    top[dest]++;
    disk[dest][top[dest]]=disk[src][top[src]];

    disk[src][top[src]]=0;
    top[src]--;

    // Check if the final state (i.e., all disks are moved to the
    // pole 2) is reached or not
    assert(!(disk[2][0]==3 && disk[2][1]==2 && disk[2][2]==1));}
}
```

top[3]

2	-1	-1

disk[3][3]

	0	1	2
2	1	0	0
1	2	0	0
0	3	0	0

	0	1	2
2	0	0	1
1	0	0	2
0	0	0	3