Bounded Model Checking Of C Programs: CBMC Tool Overview

Prateek Saxena

CBMC – Developed and Maintained by Dr Daniel Kröning
www.cprover.org/cbmc/

Model Checking

Given a model M *of a system and a property* P*, check* :

- if $M \models P$ (M models P), *P holds in M*

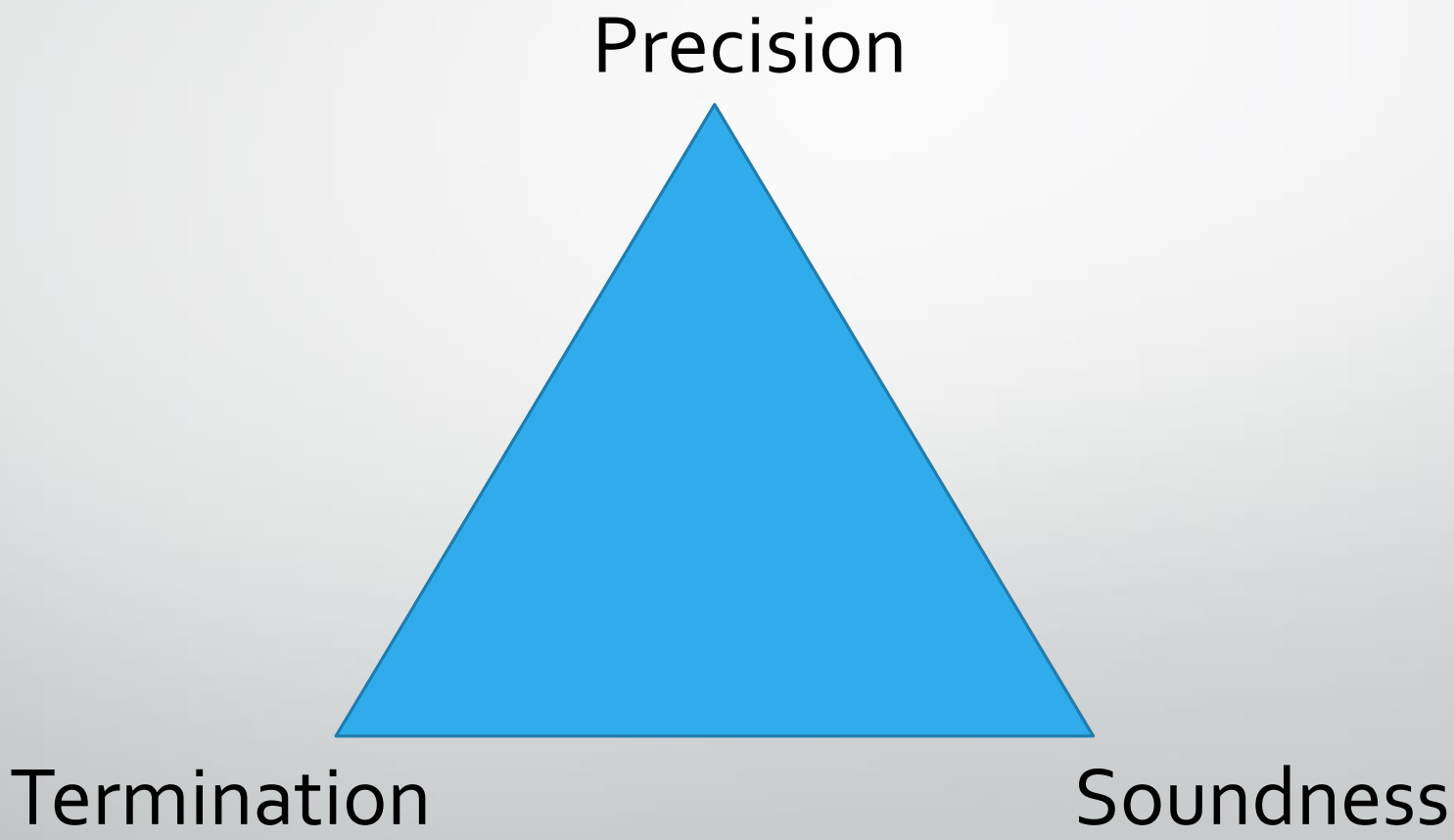
i.e. the system functions according to P.

- if $M \not\models P$ (M doesn't model P), *P doesn't hold in M,*

and a counterexample is produced, i.e. an execution of the system that does not satisfy P

Why Bounded Model Checking

In general, there is a compromise to be made between the precision of the analysis and its decidability



Bounded Model Checking

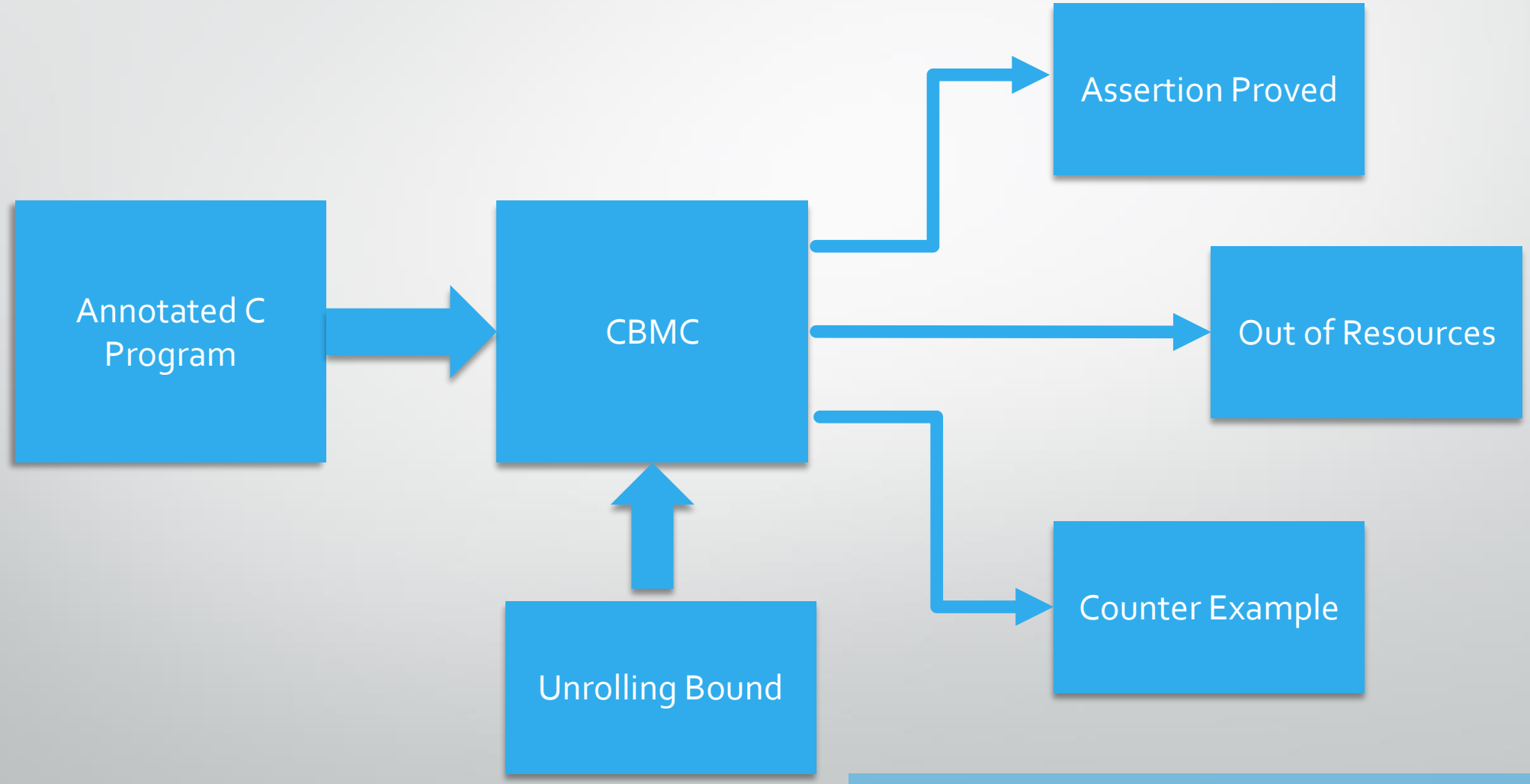
Given a model M of a system, a property P and a bound k (>0)

- Encode all executions of M of length k into a formula M_k
- Encode all executions of M of length k that violate P into $\neg P_k$
- If $(M_k \wedge \neg P_k)$ is un-satisfiable then P holds in M of length k
- if $(M_k \wedge \neg P_k)$ is satisfiable then P doesn't hold in M of length k, and a ***counterexample is produced***

Simplified Safety Properties

- Array bounds (Buffer Overflows)
- Division by zero
- Pointer checks (i.e., NULL pointer dereference)
- Arithmetic overflow
- Custom assertions (i.e., assert (i > j))

C Bounded Model Checker



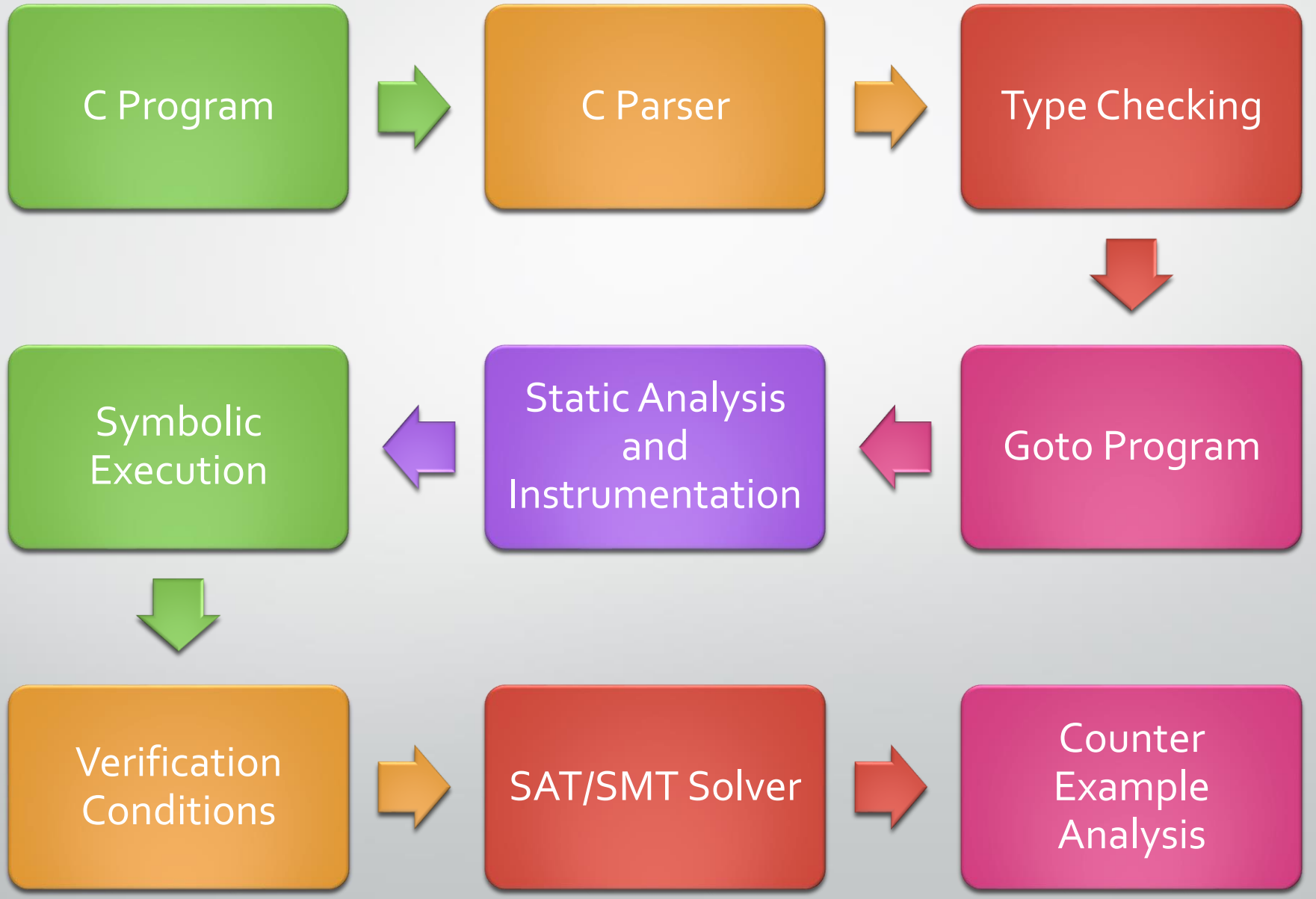
CBMC – Developed and Maintained by Dr Daniel Kröning
www.cprover.org/cbmc/

Use Cases

- Verification of operating systems
- Verification of Linux device drivers
- Equivalence Checking (Code Generators / Translation Validation / FBD -> C)
- Runtime Verification, Reachability Verification
- Model Based testing
- WCET analysis
- ….

www.cprover.org/cbmc/applications

CBMC Architecture



CBMC Simplifications and Transformations

- Removal of side effects
- Transforming all explicit loops into while loops
- All non-linear control flow is replaced by guarded goto.
- Generates one CBMC goto Program per function
- Performs light weight static analysis to resolve function pointers.
- Implicit assertions are introduced.

Goto Program

```
int main() {
  int a,SIZE;
  a =2;
  SIZE = 8;
  int i, sn=0;
  for(i=SIZE; i>=0; i--)
 {
    if (i<6)
    sn = sn + a;
  }
 assert(sn<=SIZE*a);
}
```

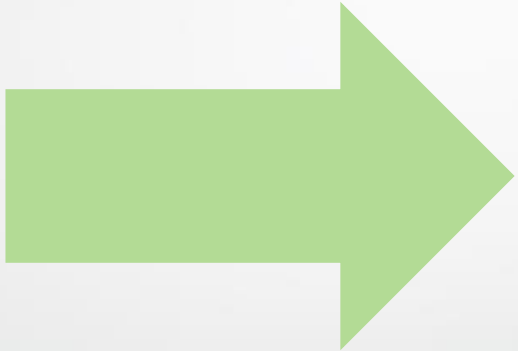
```
main /* main */
      signed int a;
      signed int SIZE;
      a = 2;
      SIZE = 8;
      signed int i;
      signed int sn;
      sn = 0;
      i = SIZE;
   1: IF !(i >= 0) THEN GOTO 3
      IF !(i < 6) THEN GOTO 2
      sn = sn + a;
   2: i = i - 1;
      GOTO 1
   3: ASSERT sn <= SIZE * a
      IF !(sn <= SIZE * a) THEN GOTO 4
   4: dead sn;
      dead i;
      dead SIZE;
      dead a;
      main#return_value = NONDET(signed int);
      END_FUNCTION
```

Generation of Verification Conditions

```
int a;
int b = 10,c = 3;

if( a>3)
        b  = c +3;
else
        c = b -7;

assert(b == 10 || b
==6)
```



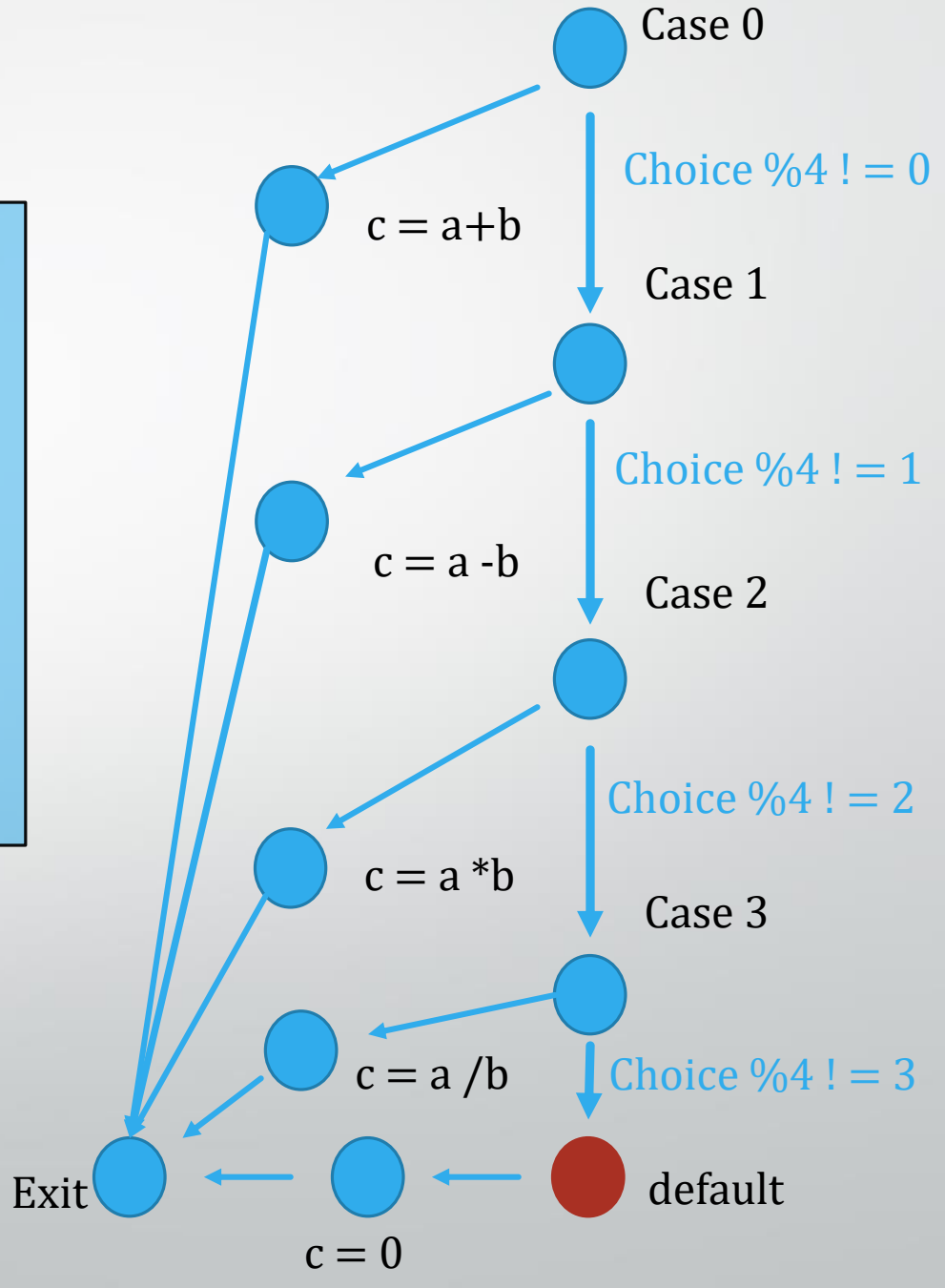
```
b1 = 10
c1 = 3
\guard1  = a1 >=4
b2= c1 + 3
c2  = c1
c3  = b1 – 7
b3  = b1
b4 = (\guard1? b2: b3)
c4= (\guard1? c3:c2)

==============
b5 ==10 || b5 == 6
```

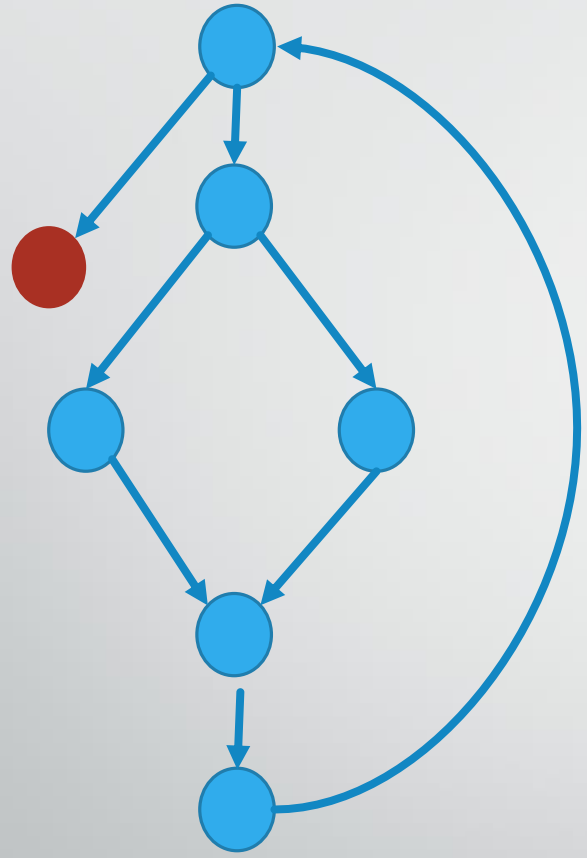
```
int choice;
if(choice > 0)
switch (choice%4)
{
case 0:
        c = a+b;
        break;
case 1:
        c= a-b;
        break;
case 2:
        c = a*b;
        break;
case 3:
        c = a/b;
        break;
default:
        assert(0);
        c = 0;
        break;   }
```

Paths

$$
\begin{aligned}
&choice > 0 \\
&\quad \wedge \\
&choice \,\%\, 4 \neq 0 \\
&\quad \wedge \\
&choice \,\%\, 4 \neq 1 \\
&\quad \wedge \\
&choice \,\%\, 4 \neq 2 \\
&\quad \wedge \\
&choice \,\%\, 4 \neq 3
\end{aligned}
$$



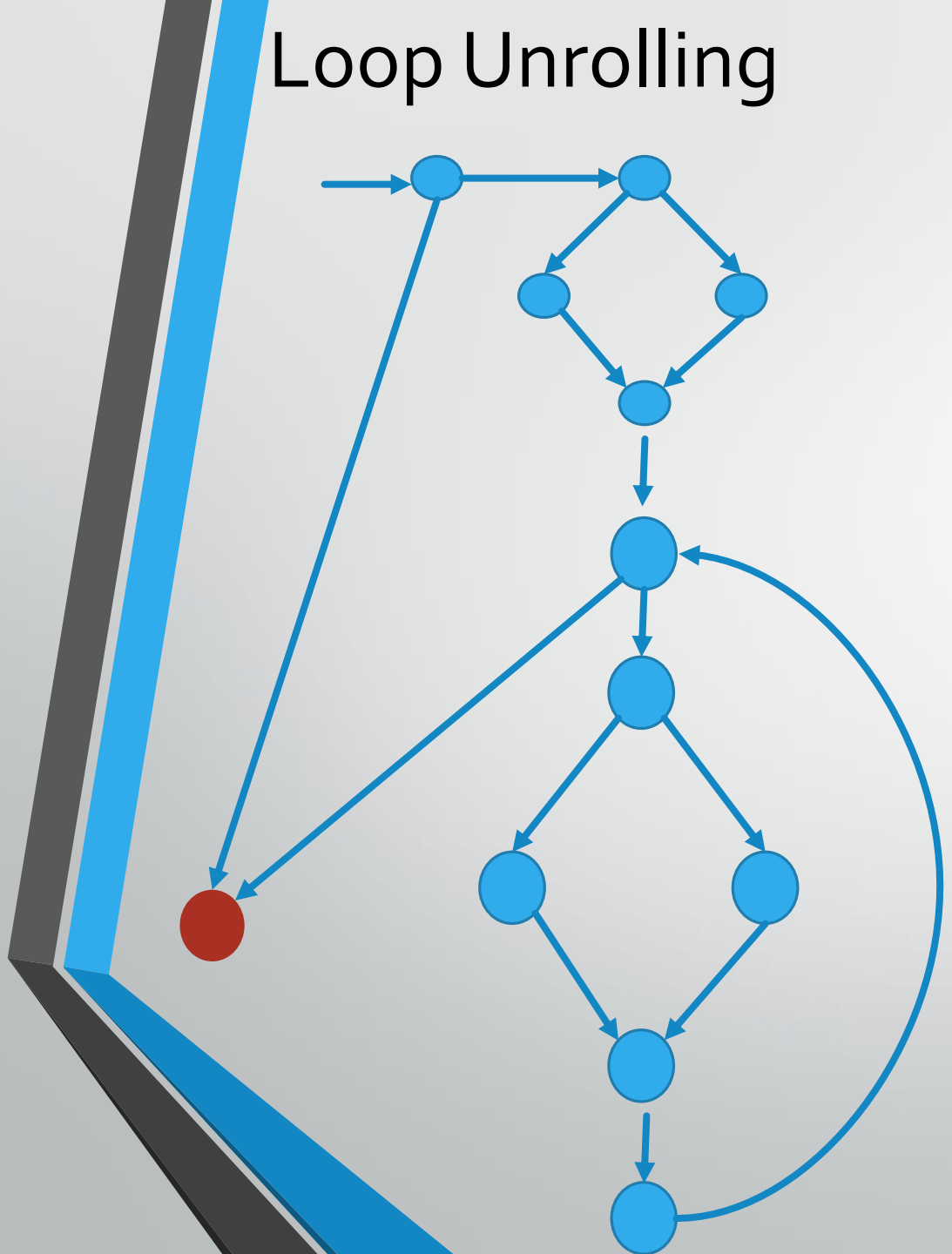
Loops



```
while( some condition holds)
{
  do something here
}
```

```
int a, b;
while(a < b)
{
  a = a+1;
}
```

Loop Unrolling



```
...
if( some condition holds)
{
   do something here
   while( same condition holds)
                do same thing here
}
...
```

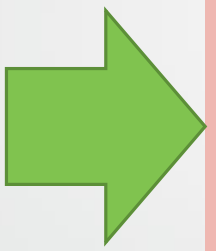
Unwinding Assertions

```
if( some condition holds)
{
  do something here
   if( same condition holds)
  {
    do same thing here
    while( same condition holds)
              do same thing here
}
```

```
if( some condition holds)
{
  do something here
   if( same condition holds)
  {
    do same thing here
    while( same condition holds)
              do same thing here
              assume(!cond)
}
```

```
if( some condition holds)
{
  do something here
   if( same condition holds)
  {
    do same thing here
    while( same condition holds)
              do same thing here
              assert (!cond)
}
```

```
int a, b;
if(a < b)
{
 a = a+1;
 while(a < b)
            a = a+1;
}
```



```
int a, b;
if(a < b)
{
 a = a+1;
 if(a < b)
  a = a+1;
  while(a< b)
   a = a+1;
}
```

```
int a, b;
if(a < b)
{
 a = a+1;
 if(a < b)
  a = a+1;
    if(a<b)
     a = a+1;
 __assume(a >= b)

}
```

Assumptions and Assertions

- The __CPROVER_assume statement restricts the program traces that are considered and allows assume-guarantee reasoning.
- The __CPROVER_assert statement aborts the program successfully if the condition evaluates to false.

```
int onetoten ()
{
int value=nondetint();
__CPROVER_assume ( value>=1 &&
value <=10);
return value ;
}
```

```
value!0@1#2 == nondet_symbol(symex::nondet0)
value!0@1#2 >= 1 && !(value!0@1#2 >= 11)
onetoten#return_value!0#1 == value!0@1#2
a!0@1#2 == onetoten#return_value!0#1
|--------------------------
{1} a!0@1#2 >= 1 && !(a!0@1#2 >= 11)
```

CBMC Checks

- Bounds check
- Div by Zero checks
- Pointer Checks
- Memory Leak checks
- Unsigned Overflows
- Float overflows
- Nans
- Signed Overflow Checks

Supported Language Features: Arithmetic Operators

- Supports all ANSI-C Boolean operators on scalar variables
- Support all integer and floating arithmetic operators on scalar variables
- Full support for arithmetic type casts
- Different rounding modes are currently not supported
- CBMC allows to model user-input by means of non-deterministic choice functions.

Supported Features: Arrays

- Arrays are encoded with the WITH and [] operators.
- Can be mapped to *store* and *select* Operators in theory of arrays

$$
\begin{aligned}
&i = 0; \\
&x[i] = j; \\
&p = x[i] + k; \\
&q = j+k;
\end{aligned}
$$

$$
\begin{aligned}
&i2 = 0 \\
&x2 = x1 \text{ WITH } [0_l := j1] \\
&p2 = x2[i1_l] + k1 \\
&q2 = j1 + k1
\end{aligned}
$$

Supported Features: Pointers

- Exact points-to analysis is performed.

```
if(d ==0)
    x = &a;
else
    x= &b;
*x = c;
```

```
\guard1 = (d1 == 0)
x2 = &a1
x3 = x1
x4 = &b1
x5 = \guard1? &a1 : &b1
b2 = (x5 == &b1)? c1: b1
a2 = (x5 == &b1)? a1: c1
```

Supported Features: Structures

- Structures are handled as aggregate data structures like arrays

```
p = &y
if(choice ==1)
{
        p->a[1] = 1;
        p->b = ‘c’;
}
else
{
        p->a[1] = 1;
        p->b = ‘c’;
}
```

```
p2 = &y1
\gaurd1 = (choice1 == 1)
y2 = y1 WITH [.a:=y1.a WITH [1l = 1]]
y3 = y2 WITH [.b = ‘c’]
y4 = y1
y5 = y4 WITH [.a = y4.a WITH [1l = 2]]
y6 = y5 WITH [.b = ‘c’]
y7  = \guard1 ? y3 : y6
```

Supported Features: Unions

- Unions are allowed in CBMC , but access to elements of the unions is not allowed across fields.
- Access to a field is permitted if only it was the field last updated.

```
union myunion
{
 char a[2];
 int b;
};

union myunion mu;
mu.a[0]= 0;
mu.a[1] = 3;
k  = mu.b;
assert(k ==3);
```

```
{-12} mu!0@1#2 ==
byte_update_little_endian(mu!0@1#1, 0l, 0, char)
{-13} mu!0@1#3 ==
byte_update_little_endian(mu!0@1#2, 1l, 3, char)
{-14} k!0@1#2 == byte_extract_little_endian(mu!0@1#3,
0l, signed int)
|-------------------------
{1} k!0@1#2 == 3
```

Trace for main.assertion.1:

---
 mu={ .a={ 0, 3 } } ({ 00000000, 00000011 })

State 21 file /home/prateek/workspace/workshop/examples/union.c line 17
function main thread 0
---
 k=0 (00000000000000000000000000000000)

State 22 file /home/prateek/workspace/workshop/examples/union.c line 18
function main thread 0
---
 k=768 (00000000000000000000001100000000)

Violated property:
 file /home/prateek/workspace/workshop/examples/union.c line 19 function main
 assertion k ==3
 k == 3

mu={ .a={ 0, 3 } } ({ 00000000, 00000011 })

State 21 file /home/prateek/workspace/workshop/examples/union.c line 17 function main thread 0

---

k=0 (00000000000000000000000000000000)

State 22 file /home/prateek/workspace/workshop/examples/union.c line 18 function main thread 0

---

k=536871680 (00100000000000000000001100000000)

Violated property:

file /home/prateek/workspace/workshop/examples/union.c line 19 function main

assertion k ==768

k == 768

Function Calls

- Support functions by in-lining.
- Preserves the locality of the parameters and the non-static local variables by renaming.
- Supports Recursion by finite unwinding

Other Supported Features

Test case generation:

CBMC can be used to automatically generate test cases following a certain code coverage criterion.

1. MC/DC
2. Decision Coverage
3. Branch
4. Path

Test Case Generation contd…

```
int foo(int a,int b , int c)
{
 int d;
 if (a > 12 && b < 45)
       {
       if (c > 4)
              {
              d =1;
              }
       else
              {
              d = 2;
       }}
 else
       d = 3;
 return d;

}
```

```
./cbmc simple_test_case.c --cover mcdc --function foo
```

```
** 11 of 11 covered (100.0%)
** Used 5 iterations
Test suite:
a=13, b=45, c=-2147483644
a=-2147483636, b=45, c=-2147483644
a=-2147483636, b=-2147483634, c=-2147483644
a=13, b=-2147483634, c=5
a=13, b=-2147483634, c=-2147483644
```

Supported features

- CBMC has support for concurrency. Supports various memory models: SC, PSO, TSO

```
void* foo(void *arg)
{
 __CPROVER_atomic_begin();
 ++i;
 __CPROVER_atomic_end();
 return 0;
}
```

```
int i=0;
int main()
{
 pthread_t th1, th2;
 pthread_create(&th1, 0, foo, 0);
 pthread_create(&th2, 0, foo, 0);
 pthread_join(th1, 0);
 pthread_join(th2, 0);
 assert(i==1);
assert(i==2);
}
```

```
./cbmc --mm sc thread_example.c --trace
```

```
** Results:
[main.assertion.1] assertion i==1: FAILURE
[main.assertion.2] assertion i==2: SUCCESS

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

References

- CBMC – C Bounded Model Checker, Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2014 –Daniel Kroening et al
- CBMC Ver 1.0 Slides by Daniel Kroenig
- CBMC CMU Website: www.cprover.org/cbmc
- Introduction to CBMC – Arie Gurfinkel, December 5, 2011
- Rui Goncalo : Automated test Generation using CBMC
- Ajith K J: VBMC - A VHDL bounded Model Checker, Feb 20, 2017

Thank you.