Introduction to CBMC: Part 1

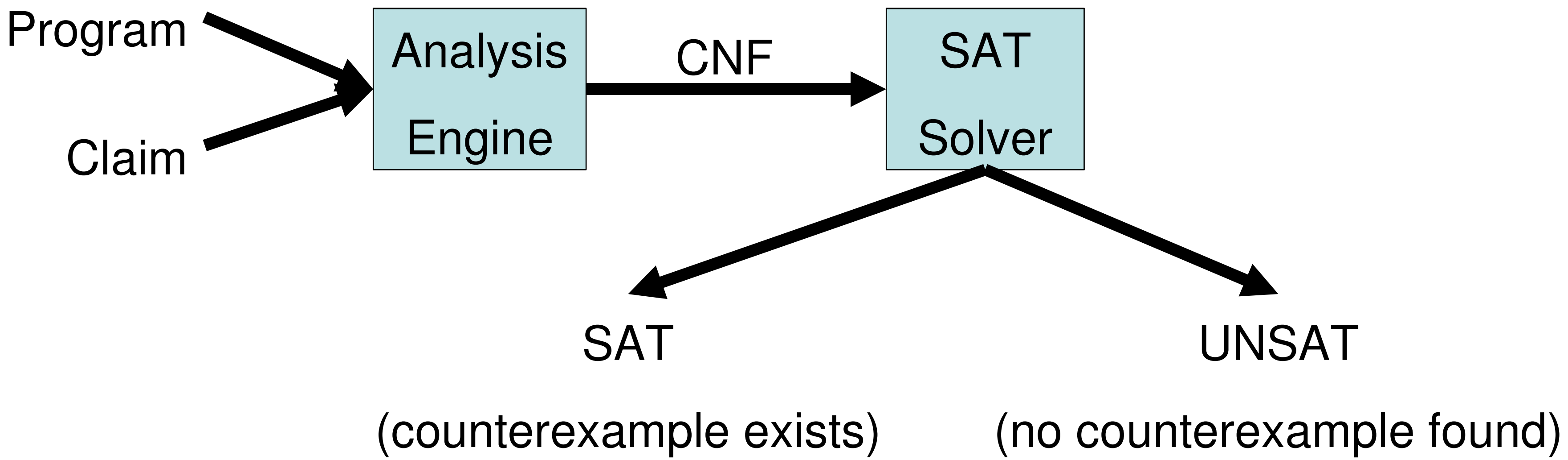
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Arie Gurfinkel, Sagar Chaki
October 2, 2007

Many slides are courtesy of
Daniel Kroening

Bug Catching with SAT-Solvers

Main Idea: Given a program and a claim use a SAT-solver to find whether there exists an execution that violates the claim.



Programs and Claims

- Arbitrary ANSI-C programs
    - With bitvector arithmetic, dynamic memory, pointers, …
- Simple Safety Claims
    - Array bound checks (i.e., buffer overflow)
    - Division by zero
    - Pointer checks (i.e., NULL pointer dereference)
    - Arithmetic overflow
    - User supplied assertions (i.e., `assert (i > j)`)
    - etc

Why use a SAT Solver?

- SAT Solvers are very efficient

- Analysis is completely automated

- Analysis as good as the underlying SAT solver

- Allows support for many features of a programming language
  - bitwise operations, pointer arithmetic, dynamic memory, type casts

A (very) simple example (1)

Program

```
int x;
int
y=8,z=0,w=0;
if (x)
  z = y – 1;
else
  w = y + 1;
assert (z ==
7 ||
          w ==
```

Constraints

y = 8,

z = x ? y – 1 : 0,

w = x ? 0 :y + 1,

z != 7,

w != 9

UNSAT

no counterexample

assertion always holds!

A (very) simple example (2)

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y – 1;
else
  w = y + 1;
assert (z == 5 ||
w ==
```

Constraints

y = 8,

z = x ? y – 1 : 0,

w = x ? 0 :y + 1,

z != 5,

w != 9

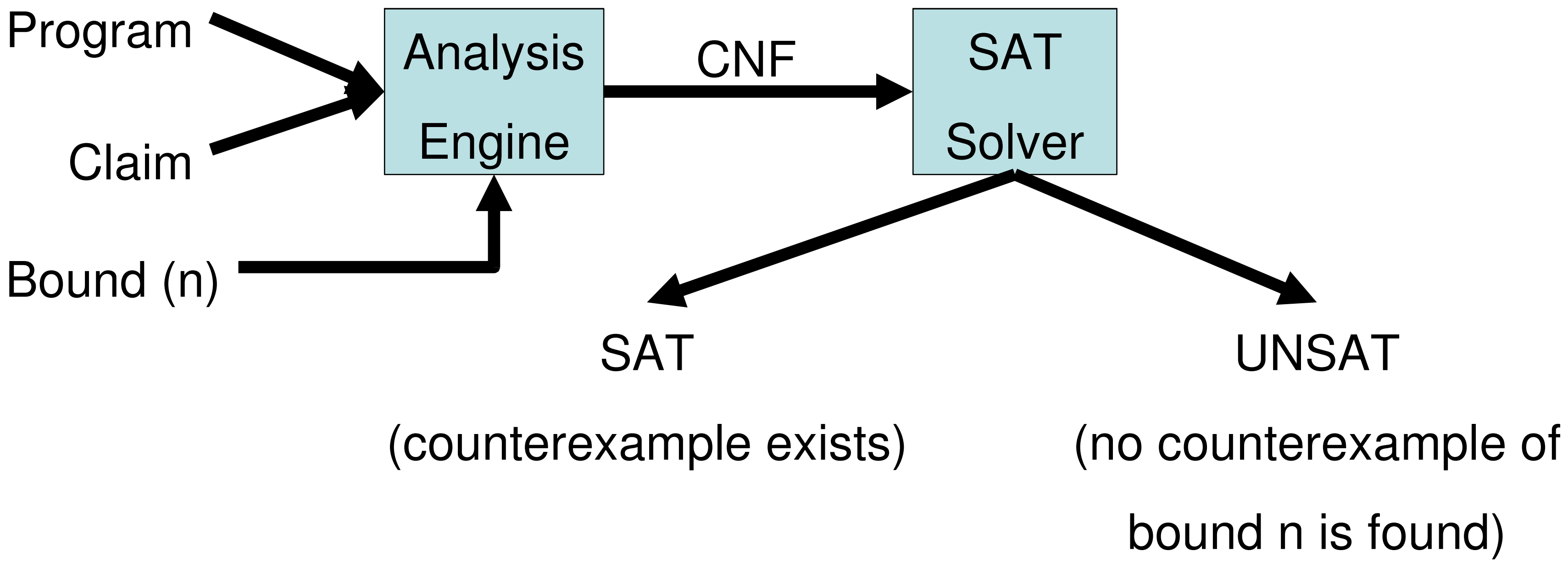
SAT

counterexample found!

y = 8, x = 1, w = 0, z = 7

What about loops?!

- SAT Solver can only explore finite length executions!
- Loops must be bounded (i.e., the analysis is incomplete)



CBMC: C Bounded Model Checker

- Developed at CMU by Daniel Kroening et al.
- Available at: http://www.cs.cmu.edu/~modelcheck/cbmc/
- Supported platafoms: Windows (requires VisualStudio's` CL), Linux
- Provides a command line and Eclipse-based interfaces

- Known to scale to programs with over 30K LOC
- Was used to find previously unknown bugs in MS Windows device drivers

CBMC: Supported Language Features

ANSI-C is a low level language, not meant for verification but for efficiency

Complex language features, such as

- Bit vector operators (shifting, and, or,…)
- Pointers, pointer arithmetic
- Dynamic memory allocation: malloc/free
- Dynamic data types: `char s[n]`
- Side effects
- `float/double`
- Non-determinism

DEMO

Using CBMC from Command Line

•To see the list of claims

```
cbmc --show-claims -I include file.c
```

•To check a single claim

```
cbmc --unwind n --claim x -I include file.c
```

•For help

- cbmc --help

Introduction to CBMC: Part 2

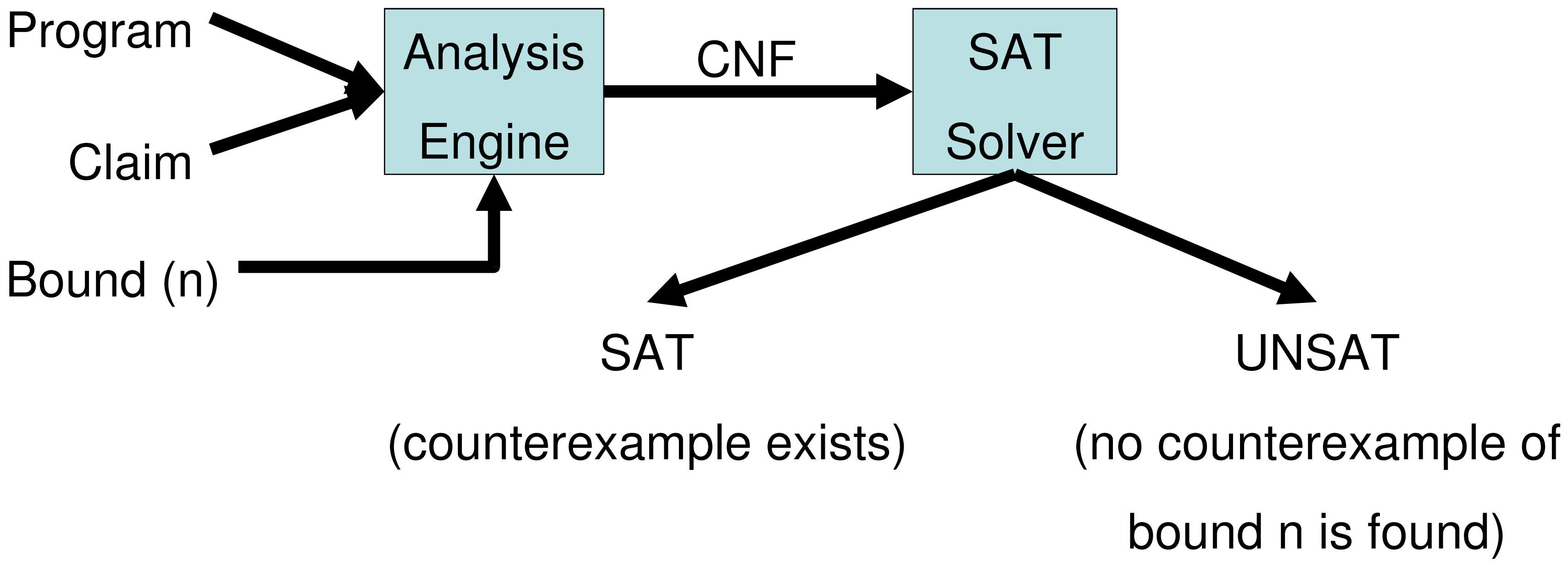
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Arie Gurfinkel, Sagar Chaki
October 2, 2007

Many slides are courtesy of
Daniel Kroening

What about loops?!

- SAT Solver can only explore finite length executions!
- Loops must be bounded (i.e., the analysis is incomplete)



How does it work

Transform a programs into a set of equations

2. Simplify control flow
3. Unwind all of the loops
4. Convert into Single Static Assignment (SSA)
5. Convert into equations
6. Bit-blast
7. Solve with a SAT Solver
8. Convert SAT assignment into a counterexample

Control Flow Simplifications

- All side effect are removal
  - `e.g., j=i++` becomes `j=i;i=i+1`
- Control Flow is made explicit
  - `continue`, `break` replaced by `goto`
- All loops are simplified into one form
  - `for`, `do while` replaced by `while`

Loop Unwinding

- All loops are unwound
  - can use different unwinding bounds for different loops
  - to check whether unwinding is sufficient special “unwinding assertion” claims are added
- If a program satisfies all of its claims and all unwinding assertions then it is correct!
- Same for backward `goto` jumps and recursive functions

Loop Unwinding

```
void f(...) {
   ...
   while(cond) {
      Body;
   }
   Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Assertion inserted after last iteration: violated if program runs longer than bound permits

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

      }
    }
  }
  Remainder;
}
```

Unwinding assertion

- while() loops are unwound iteratively
- Break / continue replaced by goto
- Assertion inserted after last iteration: violated if program runs longer than bound permits
- Positive correctness result!

Example: Sufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 2)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 2) {
    j = j + 1;
    if(j <= 2) {
      j = j + 1;
      if(j <= 2) {
        j = j + 1;
        assert(!(j <= 2));
      }
    }
  }
  Remainder;
}
```

Example: Insufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 10)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
      }
    }
  }
  Remainder;
}
```

Transforming Loop-Free Programs Into Equations (1)

Easy to transform when every variable is only assigned once!

Program

```
x = a;
y = x + 1;
z = y - 1;
```

Constraints

x = a &&

y = x + 1 &&

z = y – 1 &&

Transforming Loop-Free Programs Into Equations (2)

When a variable is assigned multiple times,

use a new variable for the RHS of each assignment

Program		SSA Program

`x=x+y;`
`x=x*2;`
`a[i]=100;`
	$\xrightarrow{\rho}$	x_1=x_0+y_0;
		x_2=x_1*2;
		a_1[i_0]=100;

What about conditionals?

Program

```
if (v)
   x = y;
else
   x = z;


w = x;
```

ρ

SSA Program

if (v_0)

   x_0 = y_0;

else

   x_1 = z_0;

w_1 = x??;

What should 'x' be?

What about conditionals?

Program

```
if (v)
   x = y;
else
   x = z;

w = x;
```

SSA Program

$\texttt{if } (v_0)$

$\quad x_0 = y_0;$

$\texttt{else}$

$\quad x_1 = z_0;$

$x_2 = v_0 \; ? \; x_0 : x_1;$

$w_1 = x_2$

ρ

For each join point, add new variables with selectors

Adding Unbounded Arrays

$$v_\alpha[a] = e \quad \xrightarrow{\rho} \quad v_\alpha = \lambda i : \begin{cases} \rho(e) & : \ i = \rho(a) \\ v_{\alpha-1}[i] & : \ \text{otherwise} \end{cases}$$

Arrays are updated "whole array" at a time

A[1] = 5;	$A_1 = \lambda\ i : i == 1\ ?\ 5 : A_0[i]$
A[2] = 10;	$A_2 = \lambda\ i : i == 2\ ?\ 10 : A_1[i]$
A[k] = 20;	$A_3 = \lambda\ i : i == k\ ?\ 20 : A_2[i]$

Examples:

$A_2[2] == 10$ $\qquad A_2[1]==5$ $\qquad A_2[3] == A_0[3]$

$A_3[2] == (k==2\ ?\ 20 : 10)$

Uses only as much space as there are uses of the array!

Example

```
int main() {
  int x, y;
  y=8;
  if(x)
    y--;
  else
    y++;

  assert
    (y==7 ||
    y==9);
}
```

$\xrightarrow{\rho}$

```
int main() {
  int x, y;
  y1=8;
  if(x0)
    y2=y1-1;
  else
    y3=y1+1;
  y4= x0 ?y2:y3;
  assert
    (y4==7 ||
    y4==9);
}
```

$\longrightarrow$

$$
\begin{array}{ll}
(& y_1 = 8 \\
\wedge & y_2 = y_1 - 1 \\
\wedge & y_3 = y_1 + 1 \\
\wedge & y_4 = x_0 \,?\, y_2 : y_3 \;) \\
\Longrightarrow & (y_4 = 7 \vee y_4 = 9)
\end{array}
$$

Pointers

While unwinding, record right hand side of assignments to pointers

This results in very precise points-to information

- Separate for each pointer
- Separate for each <u>instance</u> of each program location

Dereferencing operations are expanded into
case-split on pointer object (not: offset)

- Generate assertions on offset and on type

Pointer data type assumed to be part of bit-vector logic

- Consists of pair <object, offset>

Pointer Typecast Example

```
void *p;
int i;
int c;
int main (void) {
  int input1, intput2, z;
  p = input1 ? (void*)&i : (void*) &c
  if (input2)
     z = *(int*)p;
  else
     z = *(char*)p; }
```

Dynamic Objects

Dynamic Objects:

- `malloc`/`free`
- Local variables of functions

Auxiliary variables for each dynamically allocated object:

- Size (number of elements)
- Active bit
- Type

`malloc` sets size (from parameter) and sets active bit

`free` asserts that active bit is set and clears bit

Same for local variables: active bit is cleared upon leaving the function

Deciding Bit-Vector Logic with SAT

Pro: all operators modeled with their precise semantics

Arithmetic operators are flattened into circuits

- Not efficient for multiplication, division
- Fixed-point for `float`/`double`

Unbounded arrays

- Use uninterpreted functions to reduce to equality logic
- Similar implementation in UCLID
- But: <u>Contents</u> of array are interpreted

Problem: SAT solver happy with first satisfying assignment that is found. <u>Might not look nice</u>.

Example

```
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a; a = b; b = temp;
  }
  if (b > c) {
    temp = b; b = c; c = temp;
  }
  if (a < b) {
    temp = a; a = b; b = temp;
  }
  assert (a<=b && b<=c);
}
```

CBMC

```
State 1-3
  a=-8193 (11111111111111111101111111111
  b=-402 (1111111111111111111111100110111
  c=-2080380800 (100000111111111111110
  temp=0 (000000000000000000000000000000000

State 4 file sort.c line 10
  temp=-402 (11111111111111111111111001

State 5 file sort.c line 11
  b=-2080380800 (100000111111111111110

State 6 file sort.c line 12
  c=-402 (1111111111111111111111100110111

Failed assertion: assertion f.
sort.c line 19
```

Problem (I)

- Reason: SAT solver performs DPLL backtracking search
- Very first satisfying assignment that is found is reported
- Strange values artifact from bit-level encoding
- Hard to read
- Would like nicer values

Problem (II)

- Might not get shortest counterexample!
- Not all statements that are in the formula actually get executed
- There is a variable for each statement that decides if it is executed or not (conjunction of `if`-guards)
- Counterexample trace only contains assignments that are actually executed
- The SAT solver picks some…

Example

```
void f (int a, int b,
        int c)
{
  if(a)
  {
    a=0;
    b=1;
  }

  assert(c);
}
```

CBMC

```
{-1}  b_1#2 == (a_1#0?b_1#1:b_1#
{-2}  a_1#2 == (a_1#0?a_1#1:a_1#
{-3}  b_1#1 == 1
{-4}  a_1#1 == 0
{-5}  \guard#1 == a_1#0
{-6}  \guard#0 == T
|--------------------
{1}  c_1#0
```

from SSA

assign-ments

Example

```
void f (int a, int b,
        int c)
{
  if(a)
  {
    a=0;
    b=1;
  }

  assert(c);
}
```

CBMC

```
State 1-3
  a=1  (00000000000000000000000000000001)
  b=0  (00000000000000000000000000000000)
  c=0  (00000000000000000000000000000000)

State 4 file length.c line 5
  a=0  (00000000000000000000000000000000)

State 5 file length.c line 6
  b=1  (00000000000000000000000000000001)

Failed assertion: assertion
file length.c line 11
```

Basic Solution

Counterexample length typically considered to be most important

- e.g., SPIN iteratively searches for shorter counterexamples

Phase one: Minimize length

$$min \sum_{g \in G} l_g \cdot l_w$$

l_g: Truth value (0/1) of guard,
l_w: Weight = number of assignments

Phase two: Minimize values

Pseudo Boolean Solver (PBS)

Input:

- CNF constraints
- Pseudo Boolean constraints
  - 2x + 3y + 6z <= 7, where x, y, z are Boolean variables
- Pseudo Boolean objective function

Output:

- Decision (SAT/UNSAT)
- Optimizatioin (Minimize/Maximize an objective function)

Some implementations:

- PBS http://www.eecs.umich.edu/~faloul/Tools/pbs
- MiniSat+ (from MiniSat web page)

Example

```
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a; a = b; b = temp;
  }
  if (b > c) {
    temp = b; b = c; c = temp;
  }
  if (a < b) {
    temp = a; a = b; b = temp;
  }
  assert (a<=b && b<=c);
}
```

CBMC

Mini-mization

```
State 1-3
  a=0 (00000000000000000000000000000000
  b=0 (00000000000000000000000000000000)
  c=-1 (11111111111111111111111111111111)
  temp=0 (00000000000000000000000000000000

State 4 file sort.c line 10
  temp=0 (00000000000000000000000000000000

State 5 file sort.c line 11
  b=-1 (11111111111111111111111111111111)

State 6 file sort.c line 12
  c=0 (00000000000000000000000000000000)

Failed assertion: assertion f
sort.c line 19
```

Modeling with CBMC (1)

CBMC provides 2 modeling (not in ANSI-C) primitives

`xxx nondet_xxx ()`

Returns a non-deterministic value of type `xxx`

`int nondet_int (); char nondet_char ();`

Useful for modeling external input, unknown environment, library functions, etc.

Using nondet for modeling

Library spec:

"foo is given non-deterministically, but is taken until returned"

CMBC stub:

```
int nondet_int ();
int is_foo_taken = 0;
int grab_foo () {
  if (!is_foo_taken)
    is_foo_taken = nondet_int ();
  return is_foo_taken; }
```

```
int return_foo ()
{ is_foo_taken = 0
```

Modeling with CBMC (2)

The other modeling primitive

`__CPROVER_assume (expr)`

If the `expr` is fasle abort the program, otherwise continue executing

```
__CPROVER_assume (x>0 && y <= 10);
```

Assume-Guarantee Reasoning (1)

Is `foo` correct?

Check by splitting on the argument of `foo`

```
int foo (int* p) { ... }
void main(void) {
   ...
   foo(x);
   ...
   foo(y);
   ...
}
```

Assume-Guarantee Reasoning (2)

(A) Is `foo` correct assuming `p` is not NULL?

```
int foo (int* p) { __CPROVER_assume
(p!=NULL); … }
```

(G)Is `foo` guaranteed to be called with a non-NULL argument?

```
void main(void) {
  …
  assert (x!=NULL);//
foo(x);
  …
  assert (y!=NULL); //
foo(y);
}
```

Dangers of unrestricted assumptions

Assumptions can lead to vacuous satisfaction

```
if (x > 0) {
  __CPROVER_assume (x <
  assert (0); }
```

This program is passed by CMBMC!

Assume must either be checked with assert or used as an idiom:

```
x = nondet_int ();
y = nondet_int ();
__CPROVER_assume (x < y);
```

Checking user-specified claims

Assert, assume, and non-determinism + Programming can be used to specify many interesting claims

How to use CBMC to check whether

the loop

has an infinite execution?

```
dir=1;
while (x>0)
{ x = x + dir;
  if (x>10) dir = -1*di
  if (x<5) dir = -1*dir
}
```