Introduction to CBMC

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

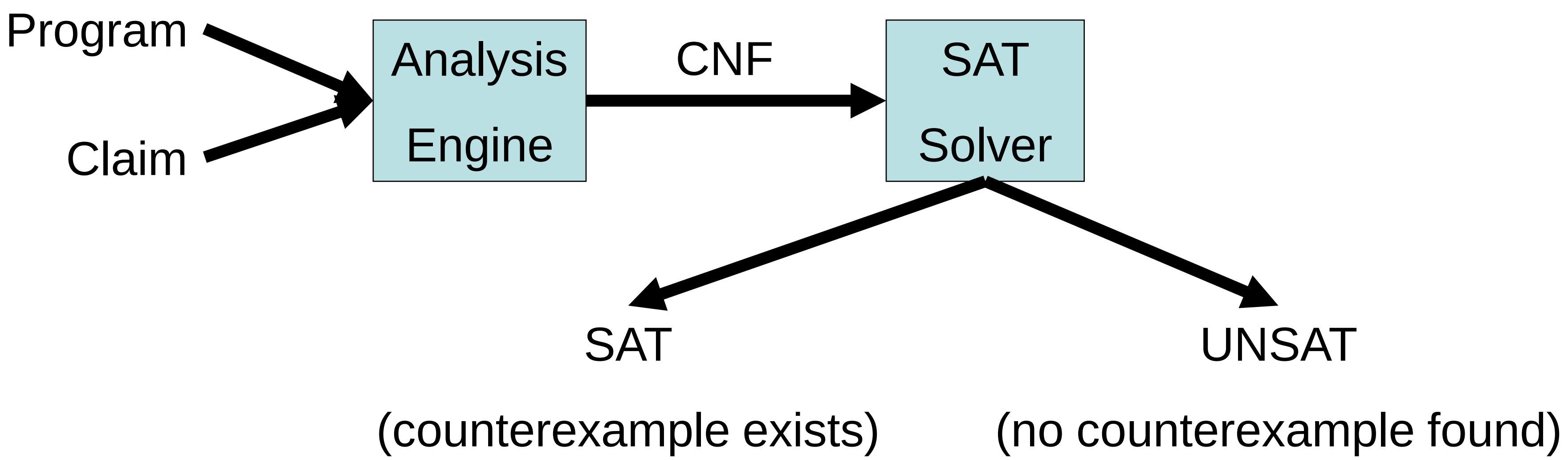
Arie Gurfinkel
December 5, 2011

based on slides by
Daniel Kroening

© 2011 Carnegie Mellon University

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.



© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

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 == 9)
```

Constraints

$y = 8,$

$z = x\ ?\ y – 1 : 0,$

$w = x\ ?\ 0 : y + 1,$

$z \ != 7,$

$w \ != 9$

UNSAT

no counterexample

assertion always holds!

© 2011 Carnegie Mellon University

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 == 9)
```

Constraints

y = 8,
z = x ? y – 1 : 0,
w = x ? 0 :y + 1,
z != 5,
w != 9

SAT

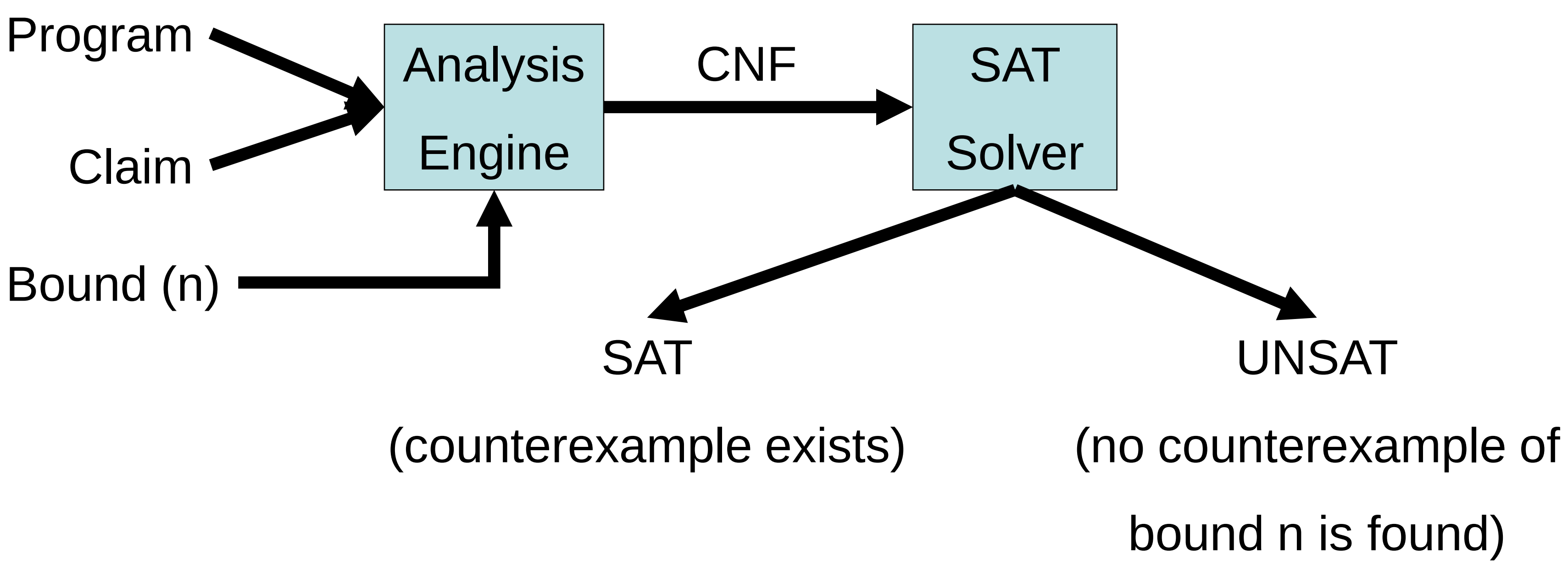
counterexample found!

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

© 2011 Carnegie Mellon University

What about loops?!

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



© 2011 Carnegie Mellon University

CBMC: C Bounded Model Checker

- Developed at CMU by Daniel Kroening et al.
- Available at: http://www.cprover.org/cbmc
  - On Ubuntu: `apt-get install 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

© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

DEMO

© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

How does it work

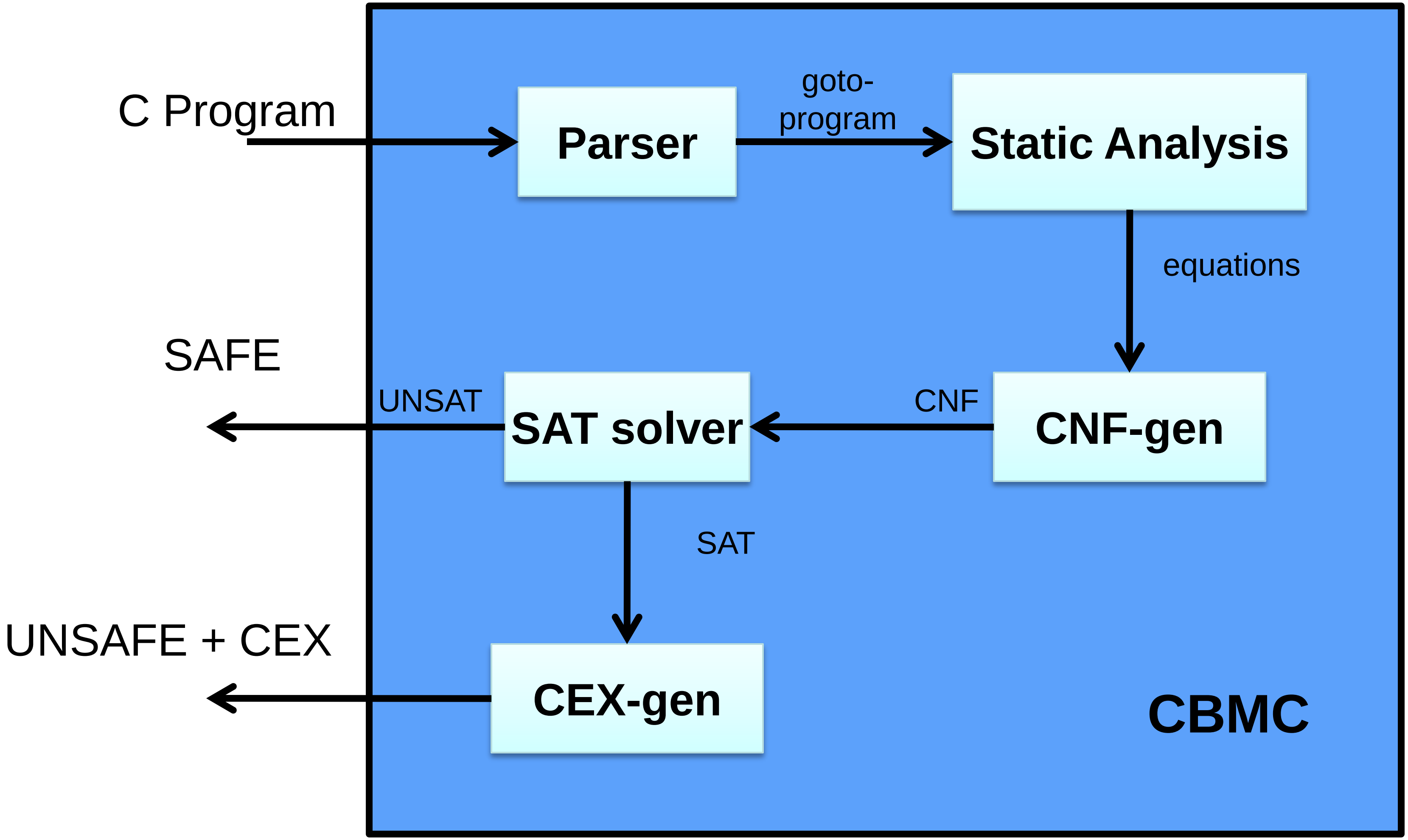
Transform a programs into a set of equations

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

© 2011 Carnegie Mellon University

CBMC: Bounded Model Checker for C

A tool by D. Kroening/Oxford



© 2011 Carnegie Mellon University

Control Flow Simplifications

- All side effect are removed
  - `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`

© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

Loop Unwinding

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

while() loops are unwound iteratively

Break / continue replaced by goto

© 2011 Carnegie Mellon University

Loop Unwinding

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

- while() loops are unwound iteratively
- Break / continue replaced by goto

© 2011 Carnegie Mellon University

Loop Unwinding

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

while() loops are unwound iteratively

Break / continue replaced by goto

© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

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!

© 2011 Carnegie Mellon University

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;
}
```


© 2011 Carnegie Mellon University

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;
}
```


© 2011 Carnegie Mellon University

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 &&

© 2011 Carnegie Mellon University

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_1=x_0+y_0;
`x=x*2;`	ρ	x_2=x_1*2;
`a[i]=100;`		a_1[i_0]=100;

© 2011 Carnegie Mellon University

What about conditionals?

Program

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

w = x;
```

ρ

SSA Program

$\texttt{if } (v_0)$

$\quad x_0 = y_0;$

$\texttt{else}$

$\quad x_1 = z_0;$

$w_1 = x??;$

What should ‘x’ be?

© 2011 Carnegie Mellon University

What about conditionals?

Program

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


w = x;
```

$\xrightarrow{\rho}$

SSA Program

if (v_0)

   $x_0 = y_0$;

else

   $x_1 = z_0$;

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

$w_1 = x_2$

For each join point, add new variables with selectors

© 2011 Carnegie Mellon University

Adding Unbounded Arrays

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

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: $A_2[2] == 10$ $A_2[1]==5$ $A_2[3] == A_0[3]$

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

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

© 2011 Carnegie Mellon University

Example

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

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

ρ

```
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);
}
```

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

© 2011 Carnegie Mellon University

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 instance 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>

© 2011 Carnegie Mellon University

Pointer Typecast Example

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


© 2011 Carnegie Mellon University

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

© 2011 Carnegie Mellon University

Modeling with CBMC

© 2006 Carnegie Mellon University

From Programming to Modeling

Extend C programming language with 3 modeling features

Assertions

- assert(e) – aborts an execution when e is false, no-op otherwise

```
void assert (_Bool b) { if (!b)  exit(); }
```

Non-determinism

- nondet_int() – returns a non-deterministic integer value

```
int nondet_int () { int x; return x; }
```

Assumptions

- assume(e) – "ignores" execution when e is false, no-op otherwise

```
void assume (_Bool e) { while (!e) ;  }
```


© 2011 Carnegie Mellon University

Example

```
int x, y;
void main (void)
{
  x = nondet_int ();

  assume (x > 10);
  y = x + 1;

  assert (y > x);
}
```

possible overflow

assertion fails

© 2011 Carnegie Mellon University

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; }
```

```
void return_foo ()
{ is_foo_taken = 0; }
```


© 2011 Carnegie Mellon University

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);
  …
}
```


© 2011 Carnegie Mellon University

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);
  …}
```


© 2011 Carnegie Mellon University

Dangers of unrestricted assumptions

Assumptions can lead to vacuous satisfaction

This program is passed by CMBMC!

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

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

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


© 2011 Carnegie Mellon University

Example: Prophecy variables

```
int x, y, v;
void main (void)
{
  v = nondet_int ();
  x = v;

  x = x + 1;
  y = nondet_int ();
  assume (v == y);

  assert (x == y + 1);
}
```

v is a *prophecy* variable

it guesses the future value of y

assume blocks executions with a wrong guess

syntactically: x is changed *before* y

semantically: x is changed *after* y

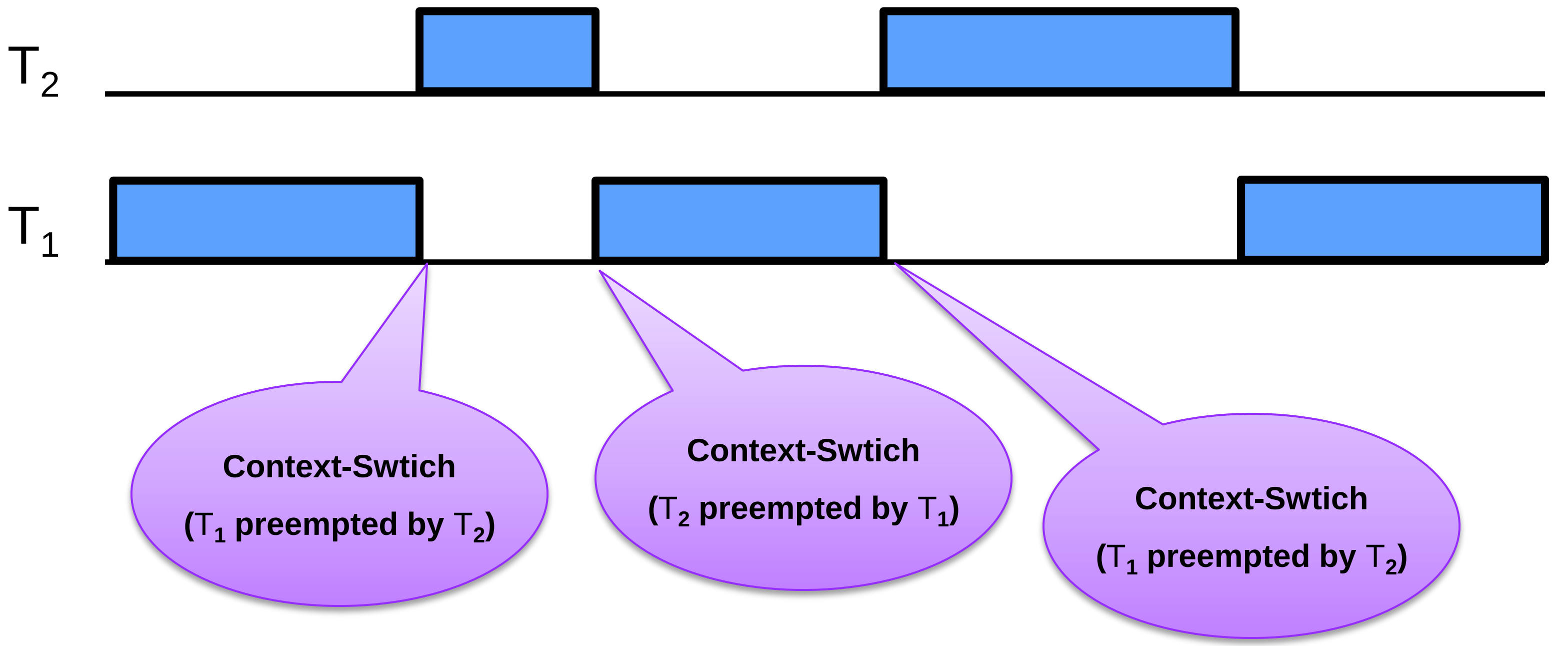
© 2011 Carnegie Mellon University

Context-Bounded Analysis with CBMC

© 2011 Carnegie Mellon University

Context-Bounded Analysis (CBA)

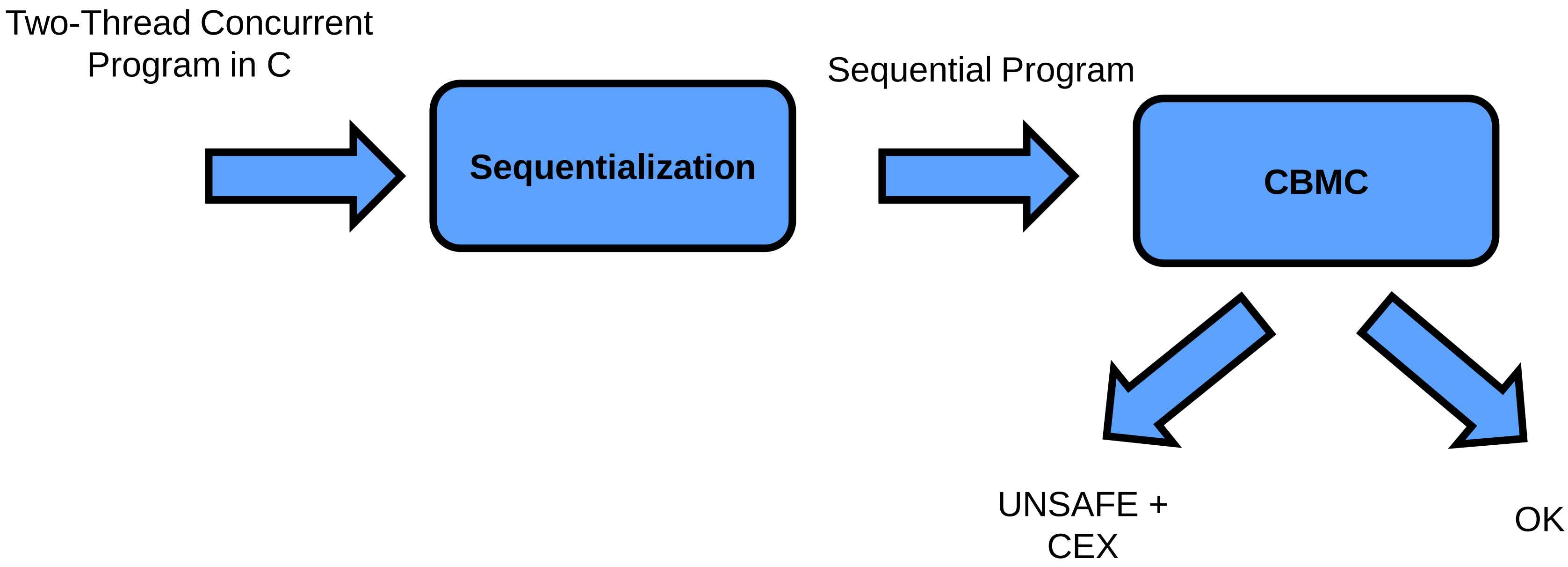
Explore all executions of TWO threads that have at most R context-switches (per thread)



© 2011 Carnegie Mellon University

CBA via Sequentialization

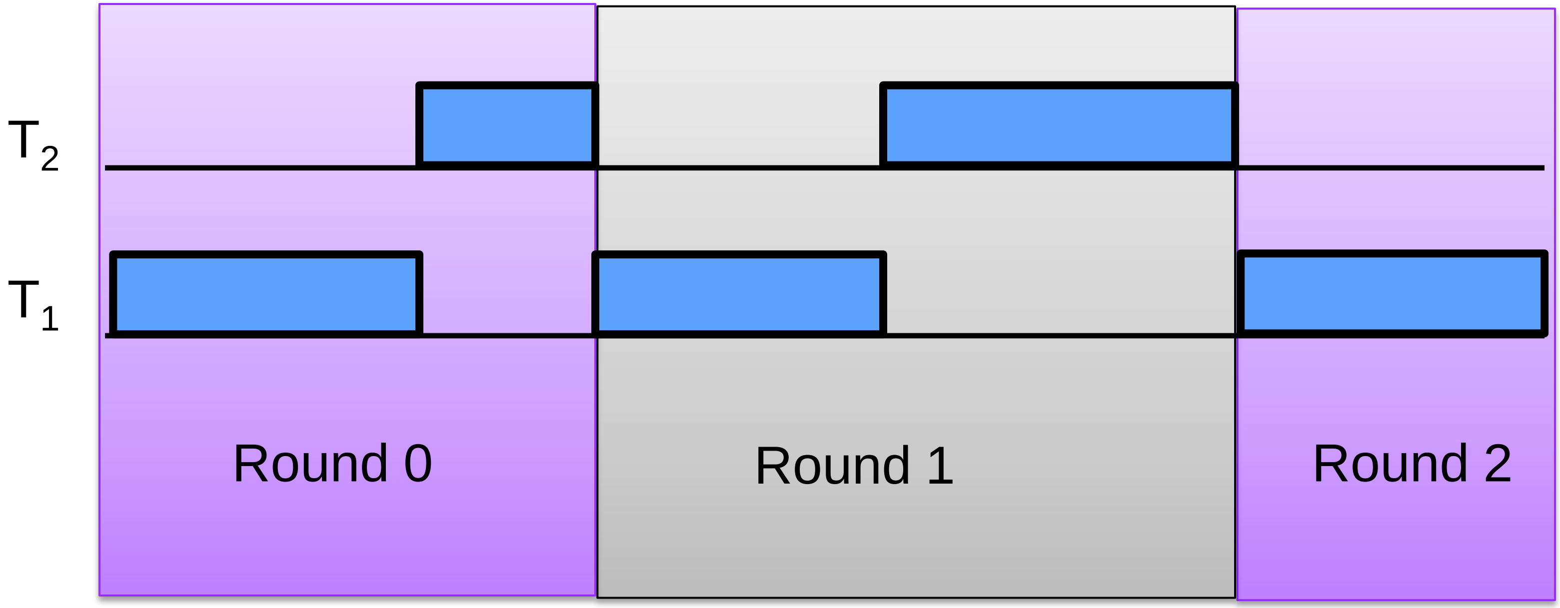
1. Reduce concurrent program P to a sequential (non-deterministic) program P’ such that “P has error” iff “P’ has error”
2. Check P’ with CBMC



© 2011 Carnegie Mellon University

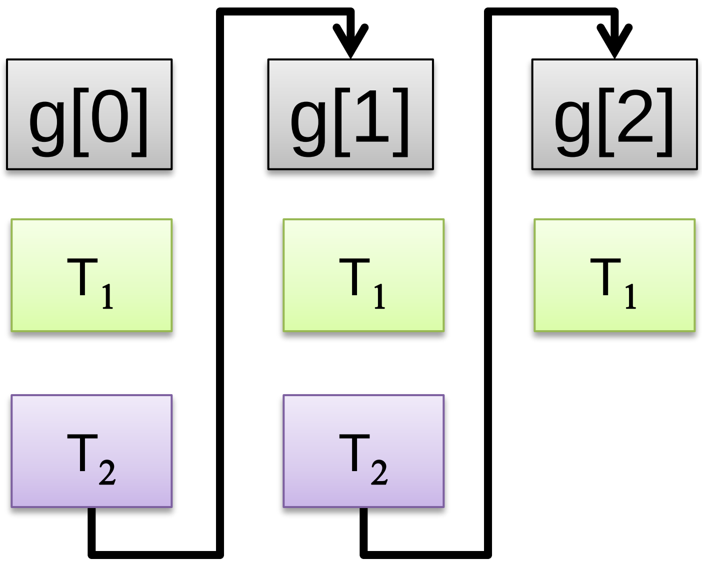
Key Idea

1. Divide execution into rounds based on context switches
2. Execute executions of each context separately, starting from a symbolic state
3. Run all parts of Thread 1 first, then all parts of Thread 2
4. Connect executions from Step 2 using assume-statements



© 2011 Carnegie Mellon University

Sequentialization in Pictures



Guess initial value of each global in each round

Execute task bodies

- T_1
- T_2

Check that initial value of round i+1 is the final value of round i

© 2011 Carnegie Mellon University

CBA Sequentialization in a Nutshel

Sequential Program for execution of R rounds (i.e., context switches):

1. for each global variable g, let g[r] be the value of g in round r
2. execute thread bodies sequentially
   - first thread 1, then thread 2
   - for global variables, use g[r] instead of g when running in round r
   - non-deterministically decide where to context switch
   - at a context switch jump to a new round (i.e., inc r)
3. check that initial value of round r+1 is the final value of round r
4. check user assertions

© 2011 Carnegie Mellon University

CBA Sequentialization 1/2

```
var
  int round;                     // current round
  int g[R], i_g[R];              // global and initial global
  Bool saved_assert = 1;         // local assertions
```

```
void main ()
  initShared ();
  initGlobals();

  for t in [0,N) : // for each thread
       round = 0;
       T'_t();
  checkAssumptions ();
  checkAssertions ();
```

```
initShared ()
 for each global var g, g[0] = init_value (g);
```

```
initGlobals ()
  for r in [1,R): //for each round
    for each global g: g[r] = i_g[r] = nondet();
```

```
checkAssumtpions ()
  for r in [0,R-1):
    for each global g:
      assume (g[r] == i_g[r+1]);
```

```
checkAssertions ()
  assert (saved_assert);
```


© 2011 Carnegie Mellon University

CBA Sequentialization: Task Body 2/2

```
void T'_t ()
  Same as T_t, but each statement 'st' is replaced with:
     contextSwitch (t); st[g ← g[round]];
  and 'assert(e)' is replaced with:
     saved_assert = e;
```

```
void contextSwitch ()
  int oldRound;

  if (nondet ()) return;  // non-det do not context switch

  oldRound = round;
  round = nondet_int ();
  assume (oldRound < round <= R-1);
```

For more details, see

Akash Lal and Tom Reps. “Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis”, in Proceedings of Computer Aided Verification, 2008.

© 2011 Carnegie Mellon University

© 2011 Carnegie Mellon University