Bounded Model Checking (BMC)

Automated Program Verification (APV)
Fall 2019

Prof. Arie Gurfinkel

UNIVERSITY OF
WATERLOO

SAT-based Model Checking

Main idea

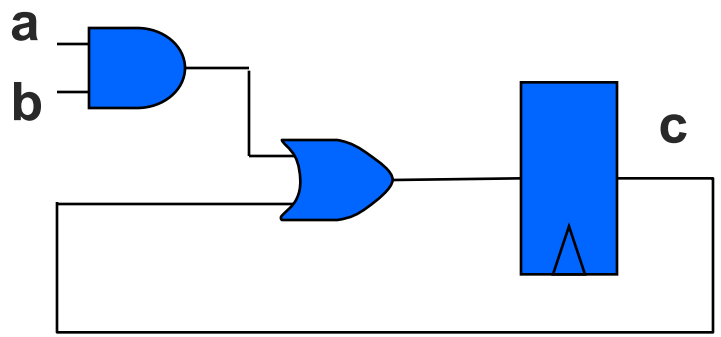
Translate the model and the specification to propositional formulas

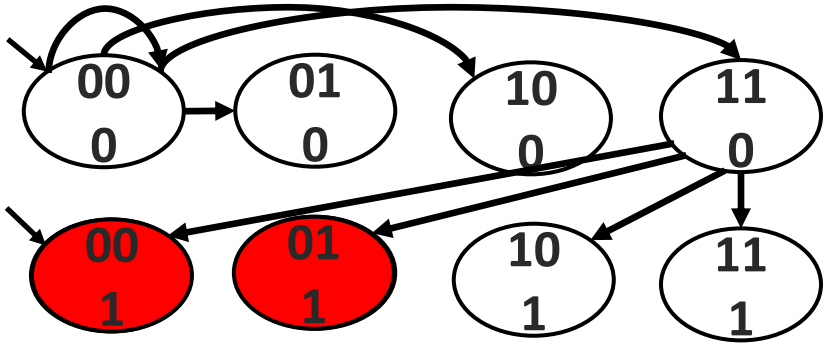
(p, ¬p, p∨q, p∧q, p→q…)

Reduce the model checking problem to satisfiability of propositional formulas

Use efficient tools (SAT solvers) for solving the satisfiability problem

Modeling with Propositional Formulas





Finite-State System is modeled as (V, INIT, T):

- **V – finite set of Boolean variables**
  - **Boolean variables: a b c ➔ 8 states: 000,001,…**
- **INIT(V) – describes the set of initial states**
  - **INIT = ¬a ∧ ¬b**
- **T(V,V') – describes the set of transitions**
  - **T(a,b,c,a',b',c') = (c' ↔ (a ∧ b) ∨ c)**

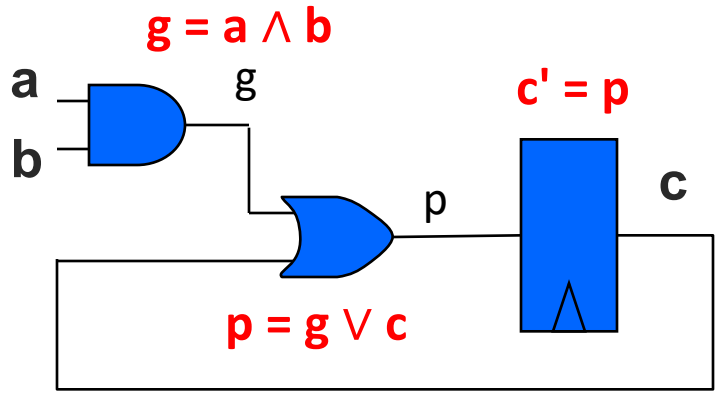
state = valuation to variables

note: $c = c_t$ and $c' = c_{t+1}$

Property:

- **p(V) - describes the set of states satisfying p**
  - **p = a ∨ ¬c (Bad = ¬p = ¬a ∧ c)**

Modeling in CNF (Tseitin encoding)



$T(a,b,c,g,p,a',b',c') =$

$g \leftrightarrow a \wedge b,$

$p \leftrightarrow g \vee c,$

$c' \leftrightarrow p$

Each circuit element is a constraint

Bounded model checking (BMC) for checking AGp

Given

- A finite transition system M= (V, INIT(V), T(V,V'))
- A safety property AG p, where p = p(V)
- A bound k

Determine

- Does M contain a counterexample to p of *k transitions (or fewer)* ?

*** BMC can handle all of LTL formulas**

A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, DAC'99

Bounded model checking for checking AGp

Unwind the model for k levels, i.e., construct all computations of length k

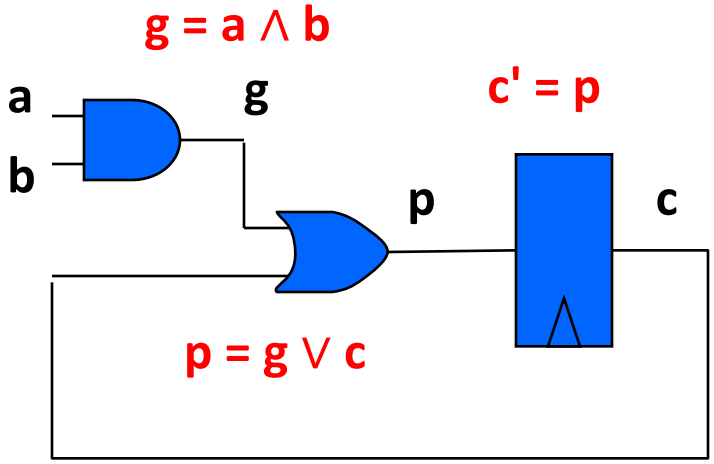
If a state satisfying $\neg p$ is encountered, then produce a counterexample

The method is suitable for **falsification**, not verification

Can be translated to a SAT problem

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k



T(a,b,c,a',b',c') =

$g \leftrightarrow a \wedge b$,

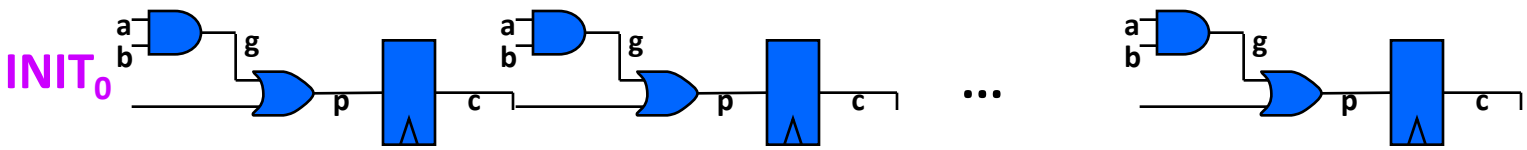
$p \leftrightarrow g \vee c$,

$c' \leftrightarrow p$

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k

$$f_{M,k} = INIT_0 \wedge T_0 \wedge T_1 \wedge \ldots \wedge T_{k-1}$$



$a_0,b_0,c_0,$ $a_1,b_1,c_1,$ $a_{k-1},b_{k-1},c_{k-1},$ $a_k,b_k,c_k,$

g_0,p_0 g_1,p_1 g_{k-1},p_{k-1} g_k,p_k

$INIT_0 = INIT(V_0)$

$T_i = T(V_i,V_{i+1})$

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k

Construct a formula $f_{\varphi,k}$ expressing that $\varphi = \mathbf{EF}\neg\mathbf{p}$ holds within k computation steps

$f_{\varphi,k} = V_{i=0,..k} (\neg p_i)$ [Sometimes $f_{\varphi,k} = \neg p_k$]

$p_i = p(V_i)$

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k

Construct a formula $f_{\varphi,k}$ expressing that $\varphi = \mathbf{EF}\neg\mathbf{p}$ holds within k computation steps

Check whether $f = f_{M,k} \wedge f_{\varphi,k}$ is satisfiable

If f is satisfiable then $M \not\models \mathbf{AGp}$

The satisfying assignment is a **counterexample**

BMC for checking AG p with SAT

Unfold the model k times:

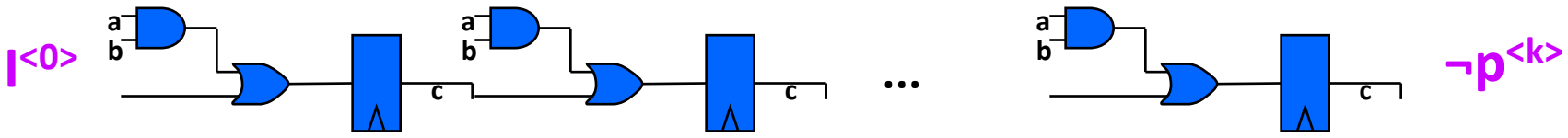
- $U = T^{<0>} \wedge T^{<1>} \wedge \ldots \wedge T^{<k-1>}$

Biere, et al. TACAS99

$I^{<0>} = I(V_0)$

$T^{<i>} = T(V_i, V_{i+1})$

$p^{<k>} = p(V_k)$



- Use SAT solver to check satisfiability of

  $I^{<0>} \wedge U \wedge \neg p^{<k>}$

- If satisfiable: the satisfying assignment describes a counterexample of length k
- If unsatisfiable: property has no counterexample of length k

Example – shift register

Shift register of 3 bits: <x, y, z>

Transition relation:

$T(x,y,z,x',y',z') = x' \leftrightarrow y \wedge y' \leftrightarrow z \wedge z'=1$

|____|

error

Initial condition:

$INIT(x,y,z) = x=0 \vee y=0 \vee z=0$

Specification: $AG (x=0 \vee y=0 \vee z=0)$

Propositional formula for k=2

$$f_{M,2} = (x_0=0 \vee y_0=0 \vee z_0=0) \wedge$$
$$(x_1 \leftrightarrow y_0 \wedge y_1 \leftrightarrow z_0 \wedge z_1=1) \wedge$$
$$(x_2 \leftrightarrow y_1 \wedge y_2 \leftrightarrow z_1 \wedge z_2=1)$$

$$\text{INIT} = x=0 \vee y=0 \vee z=0$$

$$T = x' \leftrightarrow y \wedge y' \leftrightarrow z \wedge z'=1$$

$$f_{\varphi,2} = V_{i=0,..2} (x_i=1 \wedge y_i=1 \wedge z_i=1)$$

$$P = x=0 \vee y=0 \vee z=0$$

Satisfying assignment: 101 011 111

This is a counterexample!

A remark

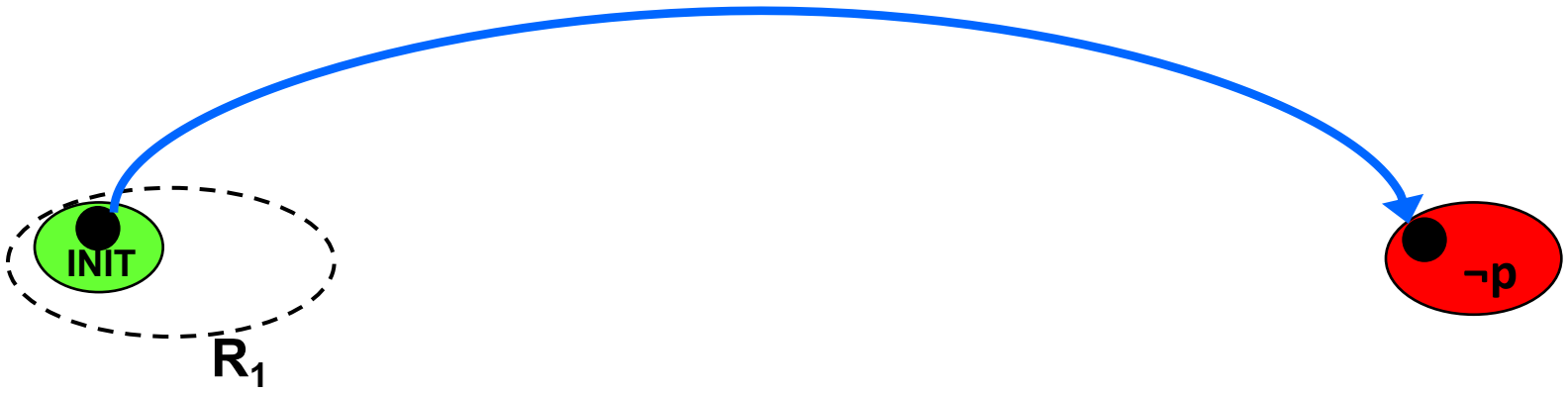
In order to describe a computation of length k by a propositional formula we need k+1 copies of the state variables.

With BDDs we use only two copies: for current and next states.

BMC for checking φ=AGp

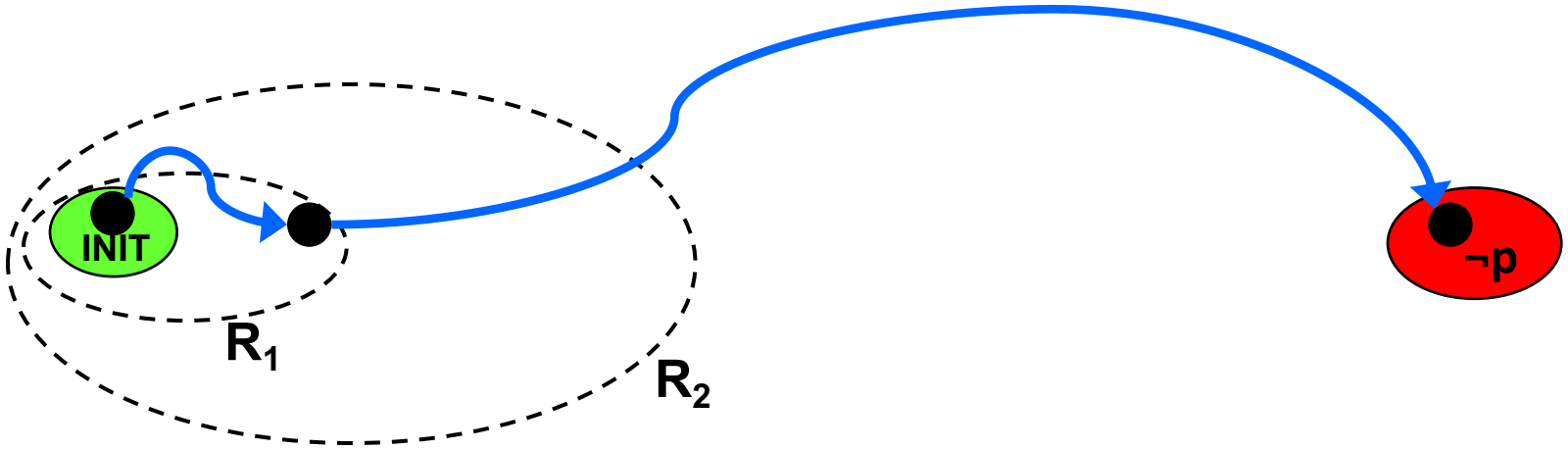
1. k=1
2. Build a propositional formula f_M^k describing all prefixes of length k of paths of M from an initial state
3. Build a propositional formula f_φ^k describing all prefixes of length k of paths satisfying F¬p
4. If $(f_M^k \wedge f_\varphi^k)$ is **satisfiable**,
   return the satisfying assignment as a **counterexample**
5. Otherwise, increase k and return to 2.

Bounded Model Checking



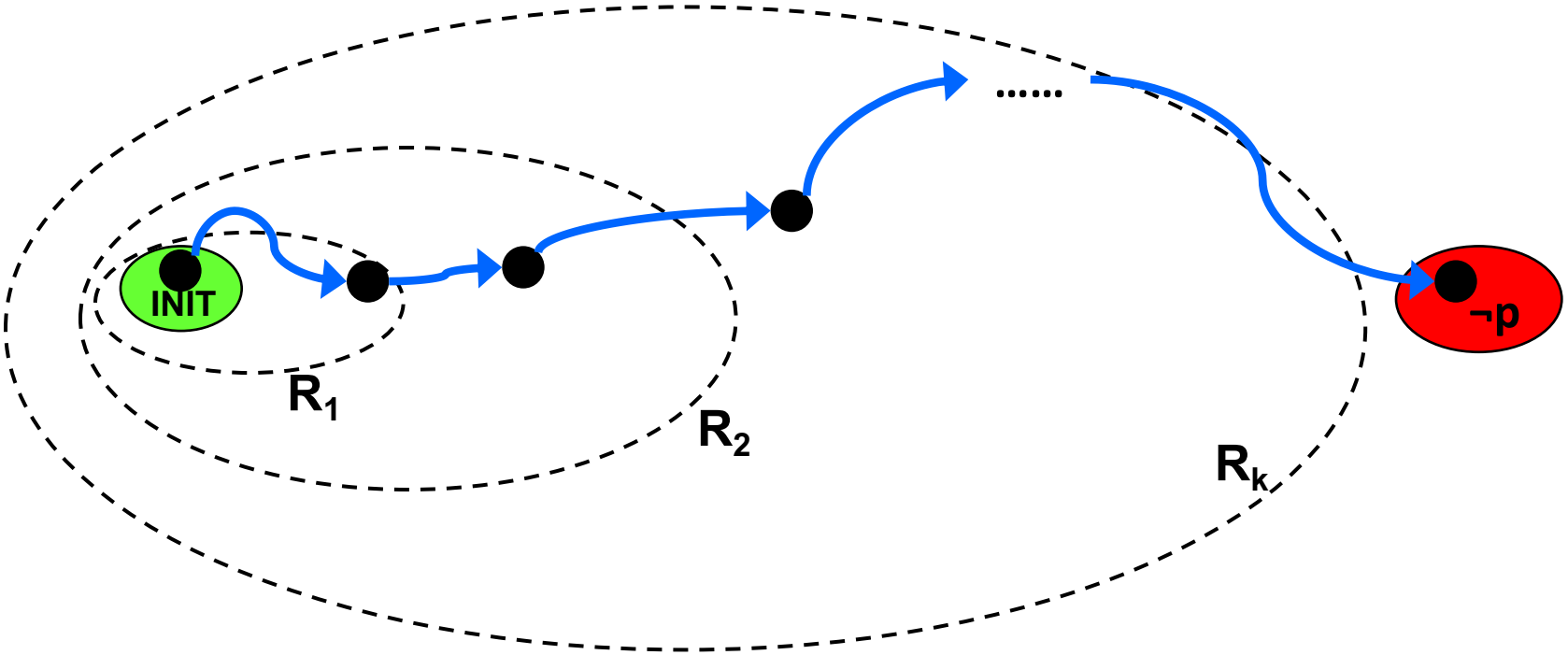
$$INIT(V^0) \wedge T(V^0,V^1) \wedge \neg p(V^1)$$

Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge T(V^1,V^2) \wedge \neg p(V^2)$$

Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge \ldots \wedge T(V^{k-1},V^k) \wedge \neg p(V^k)$$

BMC for checking AFp (φ=EG¬p)

Is there an infinite path in M

- From an initial state
- all of its states satisfying **¬p**
- Over k+1 states ?

If exists, there must also exist a lasso

BMC for checking AFp ($\varphi = EG\neg p$)

An infinite path in M, from an initial state, over k+1 states, all satisfying $\neg p$:

$$f_M^k(V_0,\ldots,V_k) = INIT(V_0) \wedge \bigwedge_{i=0,\cdots k-1} T(V_i,V_{i+1}) \wedge \bigvee_{i=0,\cdots k-1} (V_k = V_i)$$

- $V_k = V_i$ means bitwise equality: $\bigwedge_{j=0,\ldots n} (v_{kj} \leftrightarrow v_{ij})$

$$f_\varphi^k(V_0,\ldots,V_k) = \bigwedge_{i=0,\cdots k} \neg p(V_i)$$

Remark: BMC can handle all of LTL formulas

Bounded model checking

Can handle all of **LTL** formulas

Can be used for **verification** by choosing k which is large enough

- Need bound on length of the shortest counterexample.
  - *diameter* bound. The diameter is the maximum length of the shortest path between any two states.

Using such k is often **not practical** due to the size of the model

- Worst case diameter is exponential. Obtaining better bounds is sometimes possible, but generally intractable.

Bounded Model Checking

Terminates

- with a counterexample or
- with time- or memory-out

=> The method is suitable for **falsification**, not verification

Can be used for **verification** by choosing k which is large enough

- Need bound on length of the shortest counterexample.
  - *diameter* bound. The diameter is the maximum length of the shortest path between any two states.

Using such k is often **not practical**

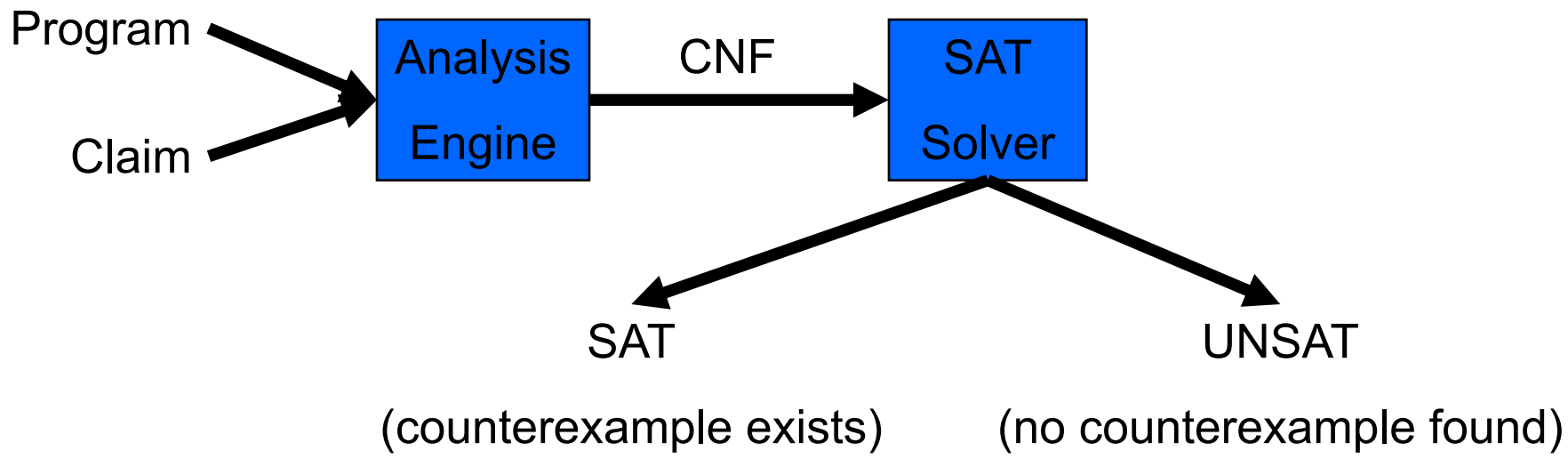
- Worst case diameter is exponential. Obtaining better bounds is sometimes possible, but generally intractable.

Bounded Model Checker for C

CBMC

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 == 9)
```

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 == 9)
```

Constraints

$$y = 8,$$
$$z = x \ ? \ y - 1 : 0,$$
$$w = x \ ? \ 0 : y + 1,$$
$$z \neq 5,$$
$$w \neq 9$$

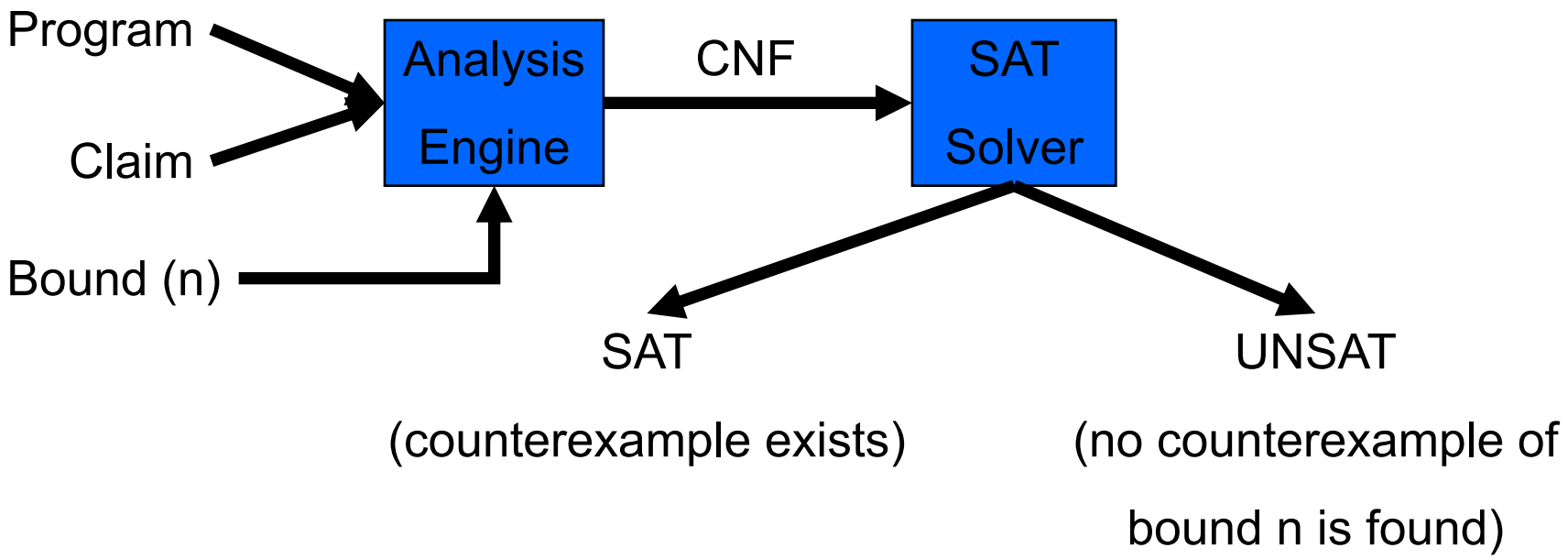
SAT

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 and Ed Clarke
- Available at: http://www.cprover.org/cbmc
  - On Ubuntu: `apt-get install cbmc`
  - `with source code`
- Supported platforms: Windows, Linux, OSX
- Has a command line, Eclipse CDT, and Visual Studio interfaces

- Scales to programs with over 30K LOC
- Found 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

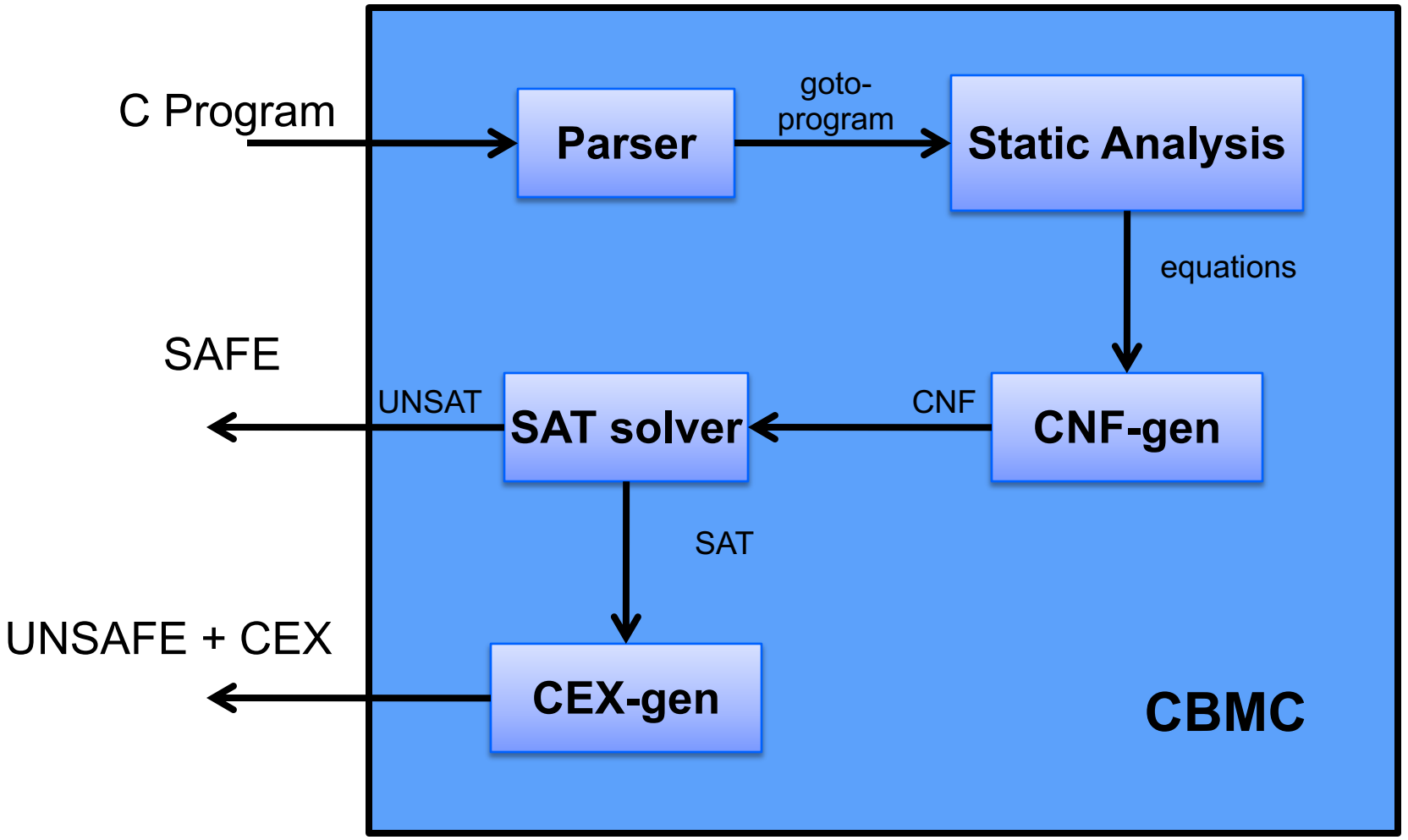
How does it work

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

CBMC: Bounded Model Checker for C

A tool by D. Kroening/Oxford and Ed Clarke/CMU



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`

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

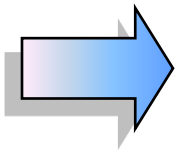
```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
        }
      }
    }
  }
  Remainder;
}
```

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;
```

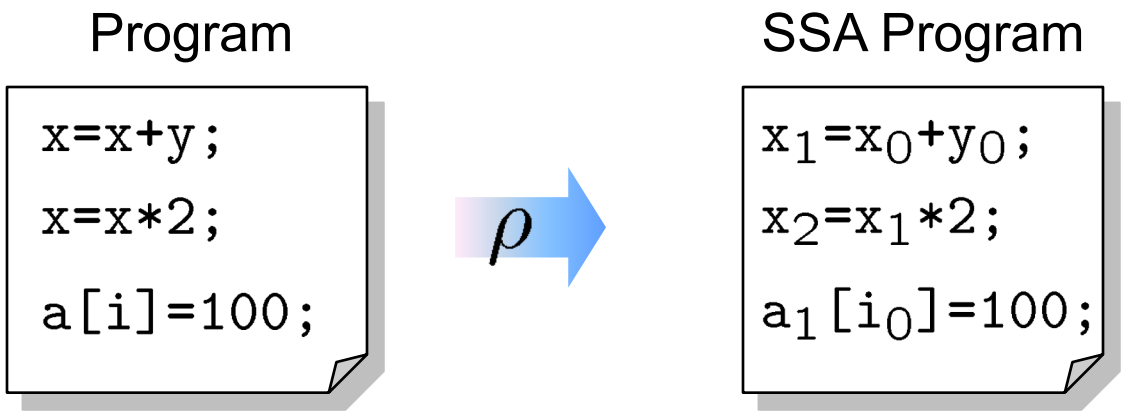


Constraints

```
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



What about conditionals?

Program

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


w = x;
```

ρ

SSA Program

```
if (v0)
  x0 = y0;
else
  x1 = z0;


w1 = x??;
```

What should ‘x’ be?

What about conditionals?

Program

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


w = x;
```

SSA Program

```
if (v0)
  x0 = y0;
else
  x1 = z0;
x2 = v0 ? x0 : x1;
w1 = x2
```

For each join point, add new variables with selectors

Adding Unbounded Arrays

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

Arrays are updated “whole array” at a time

A[1] = 5; A_1=λ i : i == 1 ? 5 : A_0[i]

A[2] = 10; A_2=λ i : i == 2 ? 10 : A_1[i]

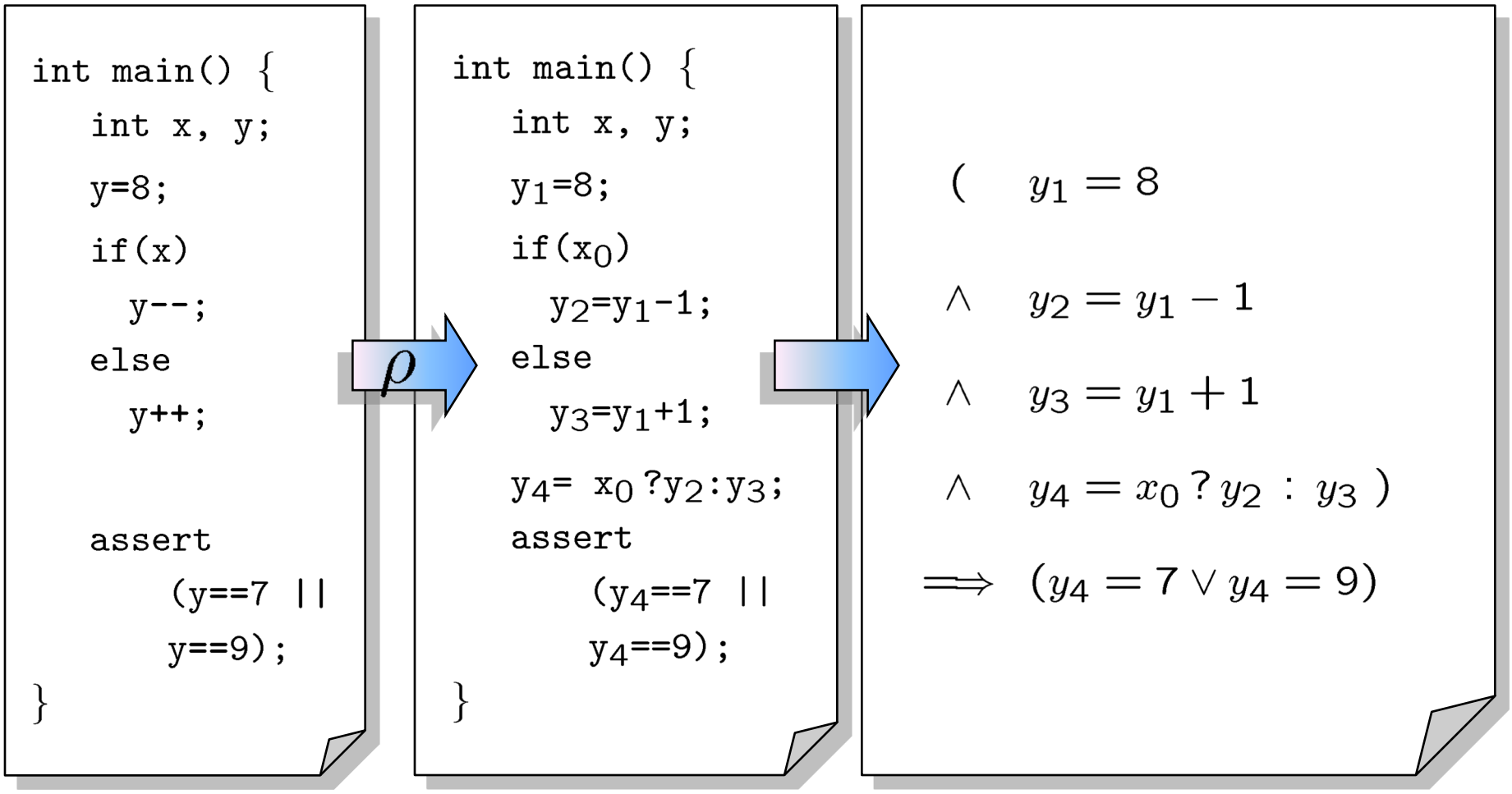
A[k] = 20; A_3=λ i : i == k ? 20 : A_2[i]

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!

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);
}
(y1 = 8
∧ y2 = y1 − 1
∧ y3 = y1 + 1
∧ y4 = x0 ? y2 : y3)
⟹ (y4 = 7 ∨ y4 = 9)

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>

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

Modeling with CBMC

UNIVERSITY OF
WATERLOO

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) ;  }
```

Example

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

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

  assert (y > x);
}
```

possible overflow

assertion fails

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; }
```

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

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);
```

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

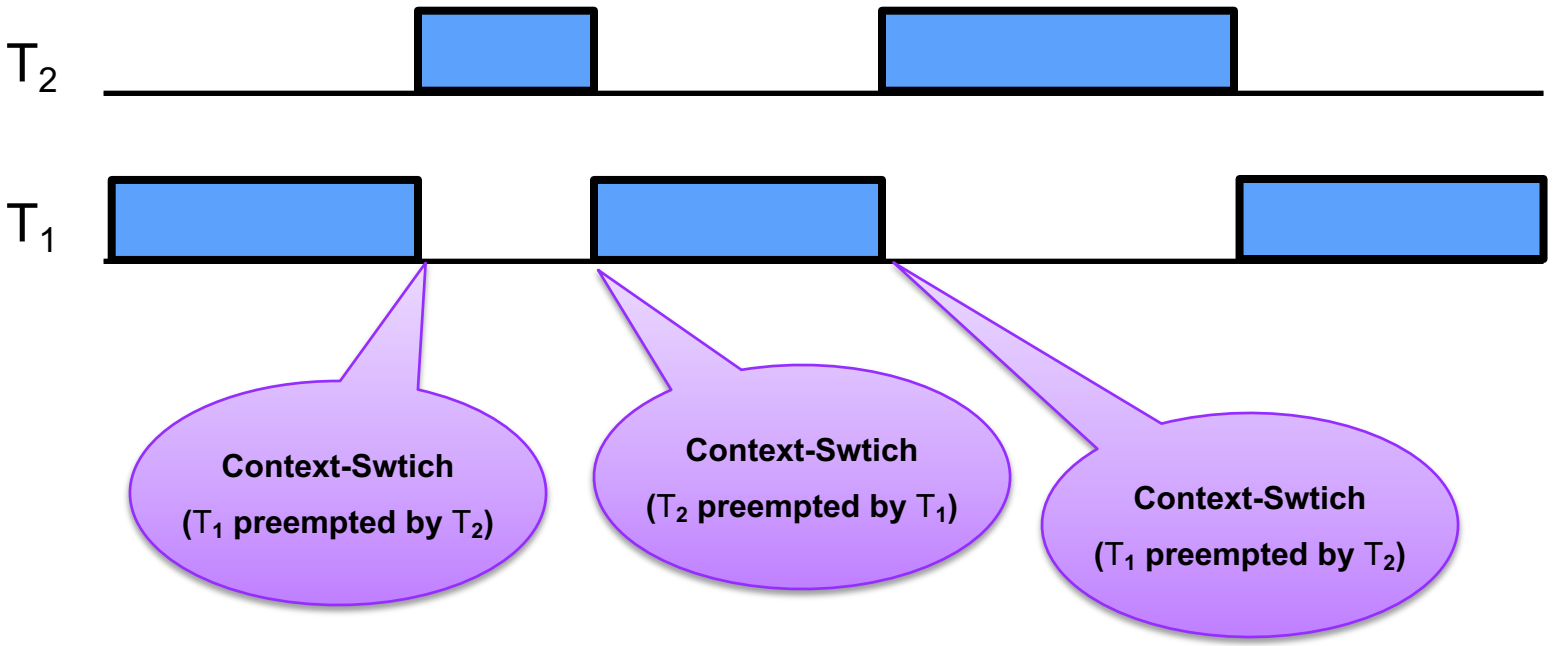
Context-Bounded Analysis with CBMC

UNIVERSITY OF WATERLOO

Akash Lal and Tom Reps. “Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis”

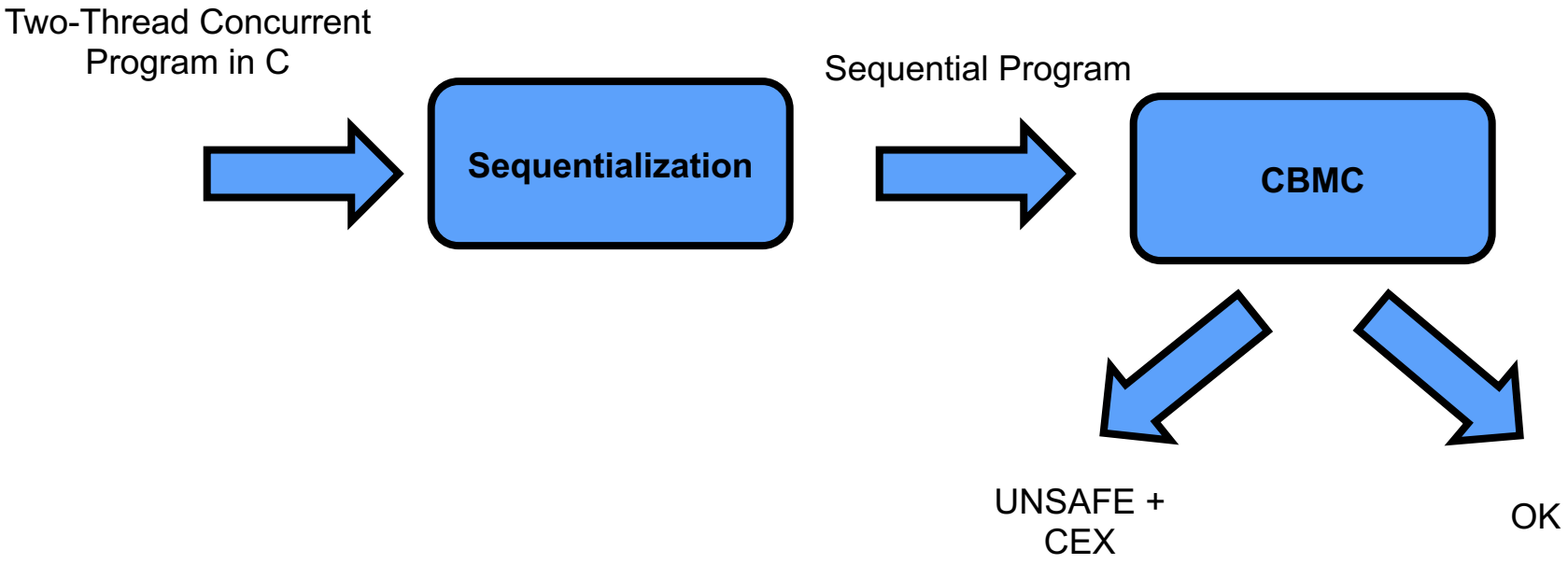
Context-Bounded Analysis (CBA)

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



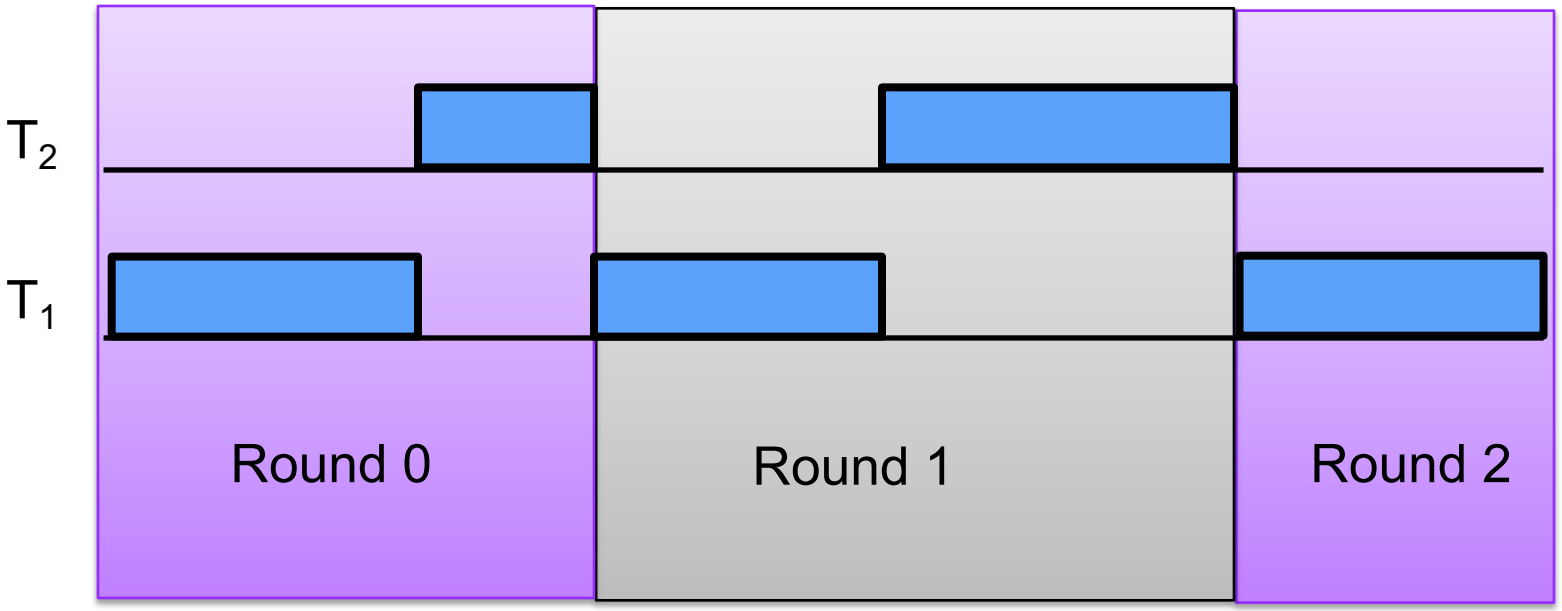
CBA via Sequentialization

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

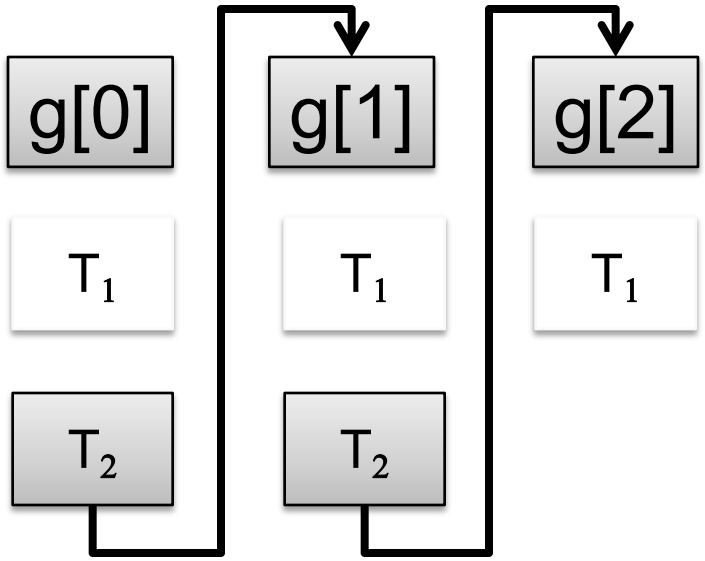


Key Idea

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



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

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

CBA Sequentialization


```
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);
```

CBA Sequentialization: Task Body 2/2

```
void T'_t ()
  Same as T_t, but each statement 'st' is replaced with:
     contextSwitch(); 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.

Checking user-specified claims

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

Use CBMC to check that this loop has a non-terminating execution

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

Symbolic Execution

Analysis of programs by tracking symbolic rather than actual values

- a form of Static Analysis

Symbolic reasoning is used to reason about *all* the inputs that take the same path through a program

Builds constraints that characterize

- conditions for executing paths
- effects of the execution on program state

Symbolic Execution

Uses symbolic values for input variables.

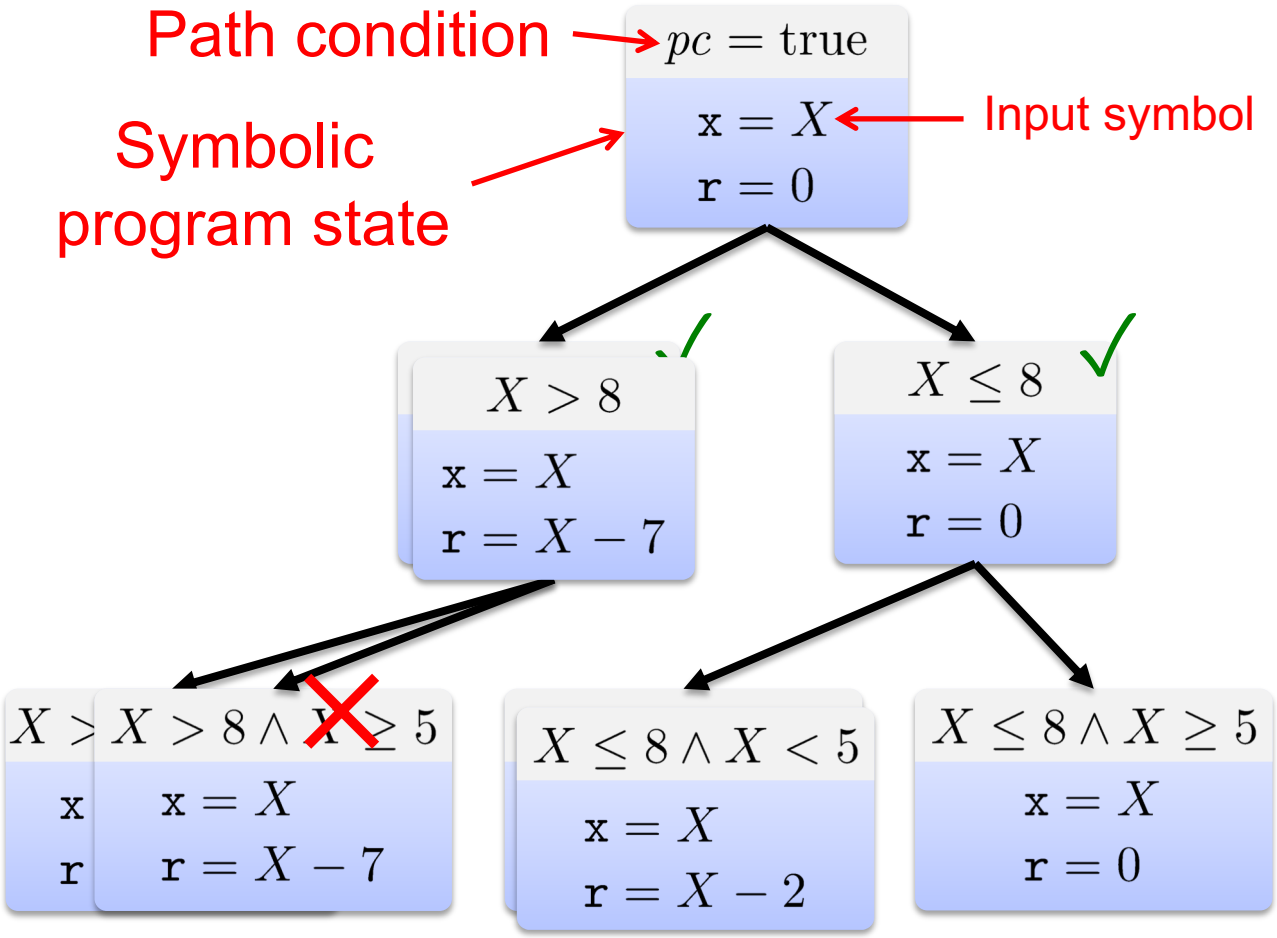
Builds constraints that characterize the conditions under which execution paths can be taken.

Collects **symbolic path conditions**

- a path condition for a path P is a formula PC such that PC is satisfiable if and only if P is executable

Uses theorem prover (**constraint solver**) to check if a path condition is satisfiable and the path can be taken.

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r
15 }
```



Satisfying assignments:

X = 9 X = 4 X = 7

Test cases:

proc(9) proc(4) proc(7)

Symbolic Execution

Verification Condition Generation

Dynamic State Merging

Query Count Estimation

[Kuznetsov, Kinder, Bucur, Candea, PLDI'12]

State joining
[Hansen et al., RV'09]

BMC slicing
[Ganai&Gupta, DAC'08]

Compositional SE / Summaries
[Godefroid, POPL'07]

Boogie
[Barnett et al., FMCO'05]

EXE (KLEE)
[Cadar et al., CCS'06]

F-Soft
[Ivancic et al., CAV'05]

DART (SAGE)
[Godefroid, PLDI'05]

CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG