Introduction to CBMC: Part 1

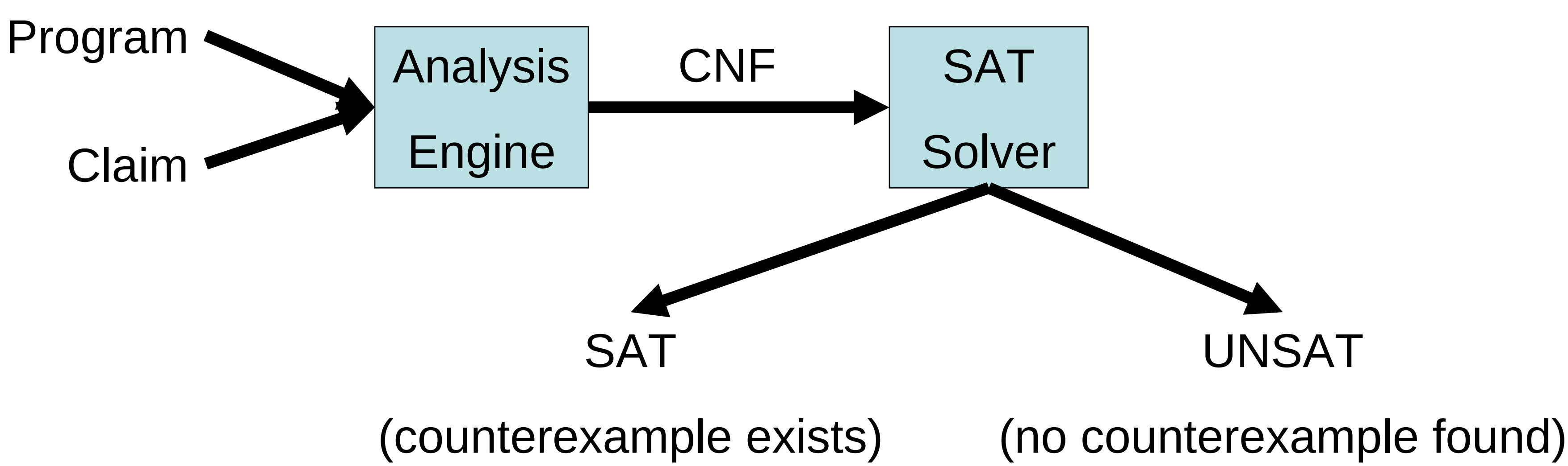
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

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.



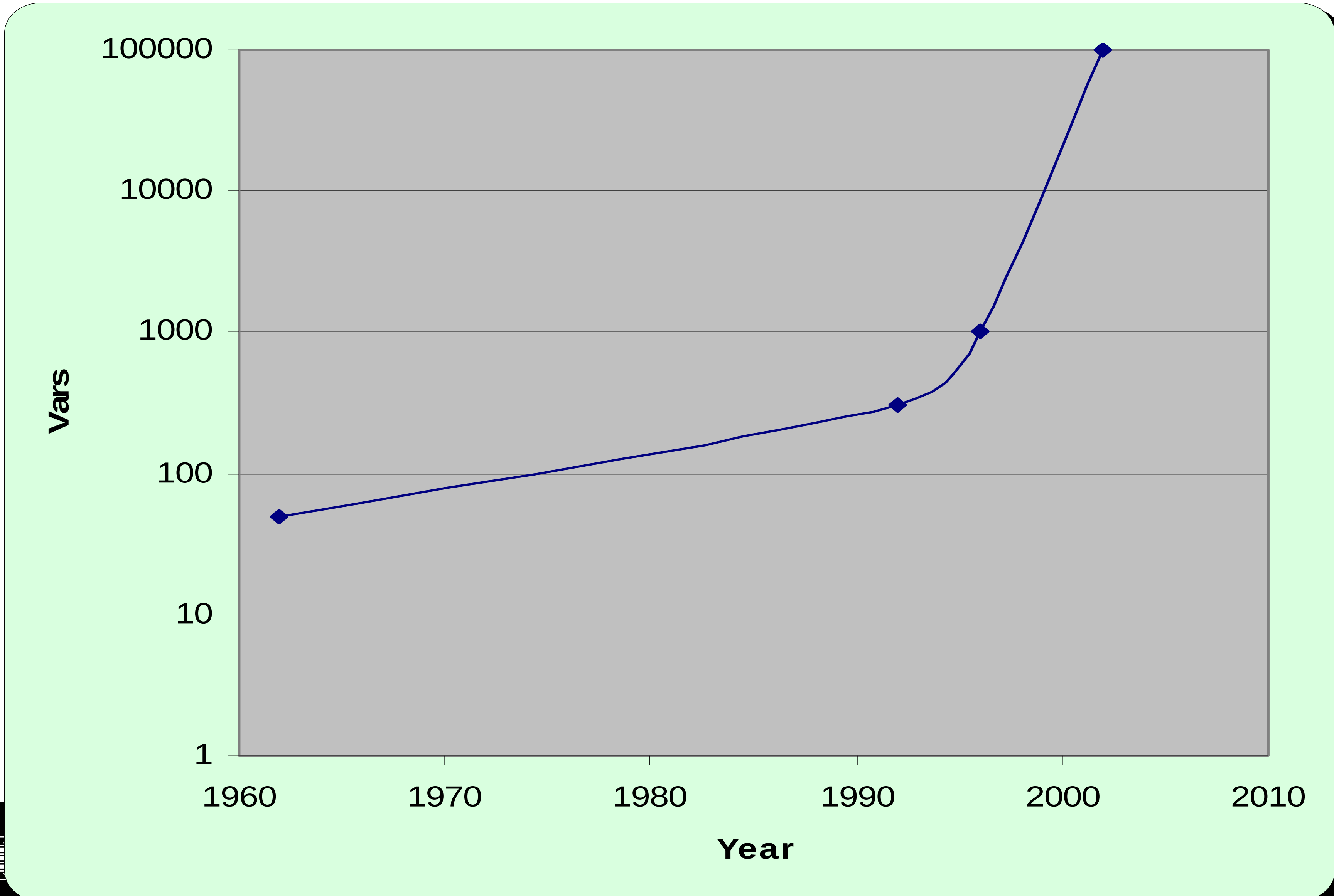
Programs and Claims

- 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

SAT made some progress…



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 != 5,

w != 9

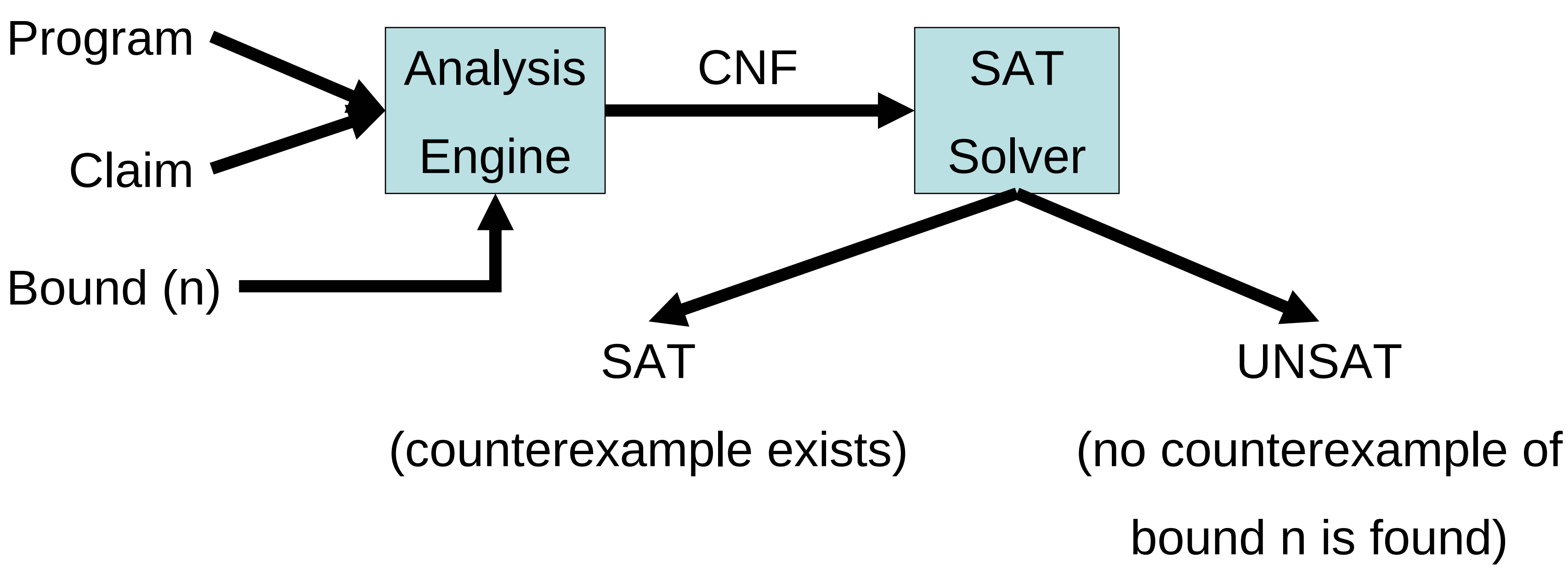
SAT

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 platfoms: 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

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

How does it work

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

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`

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

```
x=x+y;
x=x*2;
a[i]=100;
```

ρ

SSA Program

$x_1 = x_0 + y_0;$

$x_2 = x_1 * 2;$

$a_1[i_0] = 100;$

What about conditionals?

Program

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

w = x;
```

ρ

SSA Program

$\texttt{if } (v_0)$

$\quad x_0 = y_0;$

$\texttt{else}$

$\quad x_1 = z_0;$

$w_1 = x??;$

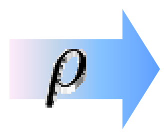
What should ‘x’ be?

What about conditionals?

Program

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

w = x;
```



SSA Program

$\texttt{if}\ (v_0)$

$\quad x_0 = y_0;$

$\texttt{else}$

$\quad x_1 = z_0;$

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

$w_1 = x_2$

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] == ?? A_2[1]==?? A_2[3] == ??

y=A_3[2] =>??

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);
}
```

→

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

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>

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

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

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;
}
```

Convert Bit Vector Logic Into Propositional Logic