Bounded Software Model Checking

An Introduction to **CBMC**

SCC.363 Security and Risk

School of Computing and Communications, Lancaster University

16 February 2024

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

Is a given Boolean formula F satisfiable?

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

Is a given Boolean formula F satisfiable?

```
bool x0, x1, x2, x3;
bool F = (x0 && x1 || x2) && (x3 && x0 || x2) && !(x3 || x0);
```

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

Is a given Boolean formula F satisfiable?

```
bool x0, x1, x2, x3;
bool F = (x0 && x1 || x2) && (x3 && x0 || x2) && !(x3 || x0);
```

[Recall: mathematical notation for logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not).]

$$F = (x_0 \wedge x_1 \vee x_2) \wedge (x_3 \wedge x_0 \vee x_2) \wedge \neg(x_3 \vee x_0)$$

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

Is a given Boolean formula F satisfiable?

```
bool x0, x1, x2, x3;
bool F = (x0 && x1 || x2) && (x3 && x0 || x2) && !(x3 || x0);
```

[Recall: mathematical notation for logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not).]

$$F \;=\; (x_0 \wedge x_1 \vee x_2) \wedge (x_3 \wedge x_0 \vee x_2) \wedge \neg(x_3 \vee x_0)$$

Problem: does there exist an assignment of truth values to the Boolean variables x_0, x_1, x_2, x_3 that makes the formula *True*?

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

Is a given Boolean formula F satisfiable?

```
bool x0, x1, x2, x3;
bool F = (x0 && x1 || x2) && (x3 && x0 || x2) && !(x3 || x0);
```

[Recall: mathematical notation for logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not).]

$$F \;=\; (x_0 \wedge x_1 \vee x_2) \wedge (x_3 \wedge x_0 \vee x_2) \wedge \neg(x_3 \vee x_0)$$

Problem: does there exist an assignment of truth values to the Boolean variables x_0, x_1, x_2, x_3 that makes the formula *True*?

Yes! $x_0 =$ *False*, $x_1 =$ *True*, $x_2 =$ *True*, $x_3 =$ *False*.

So F is *satisfiable* (otherwise *unsatisfiable*).

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

Is a given Boolean formula F satisfiable?

```
bool x0, x1, x2, x3;
bool F = (x0 && x1 || x2) && (x3 && x0 || x2) && !(x3 || x0);
```

[Recall: mathematical notation for logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not).]

$$F \;=\; (x_0 \wedge x_1 \vee x_2) \wedge (x_3 \wedge x_0 \vee x_2) \wedge \neg(x_3 \vee x_0)$$

Problem: does there exist an assignment of truth values to the Boolean variables x_0, x_1, x_2, x_3 that makes the formula *True*?

Yes! $x_0 =$ *False*, $x_1 =$ *True*, $x_2 =$ *True*, $x_3 =$ *False*.

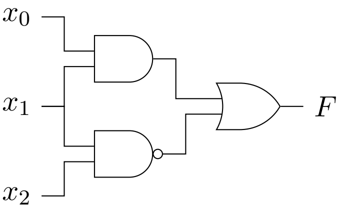
So F is *satisfiable* (otherwise *unsatisfiable*).

[The **SAT problem** was the first example of an **NP-Complete** problem (*Cook*, 1971).]

Boolean Satisfiability (SAT) Problem

Many problems (e.g. in circuit design) can be reduced to SAT.

[An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).]



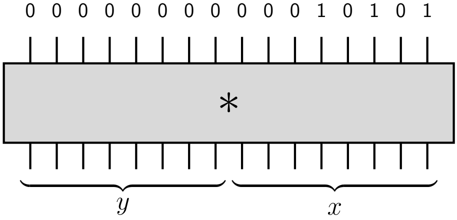
$$F = (x_0 \wedge x_1) \vee \neg(x_1 \wedge x_2)$$

Boolean Satisfiability (SAT) Problem

Many problems (e.g. in circuit design) can be reduced to SAT.

[An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).]

One may e.g. model a *multiplier circuit* as a Boolean formula and ask if there are inputs to the circuit that result in the number 21 (10101 in binary) along the output wires (i.e. check if 21 is *prime*).

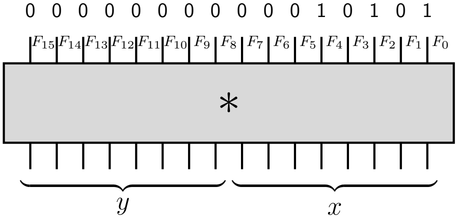


Boolean Satisfiability (SAT) Problem

Many problems (e.g. in circuit design) can be reduced to SAT.

[An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).]

One may e.g. model a *multiplier circuit* as a Boolean formula and ask if there are inputs to the circuit that result in the number 21 (10101 in binary) along the output wires (i.e. check if 21 is *prime*).

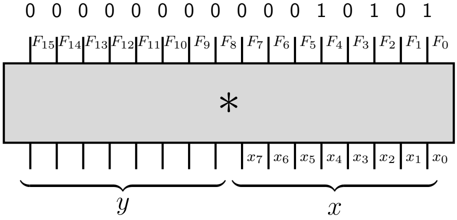


Boolean Satisfiability (SAT) Problem

Many problems (e.g. in circuit design) can be reduced to SAT.

[An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).]

One may e.g. model a *multiplier circuit* as a Boolean formula and ask if there are inputs to the circuit that result in the number 21 (10101 in binary) along the output wires (i.e. check if 21 is *prime*).

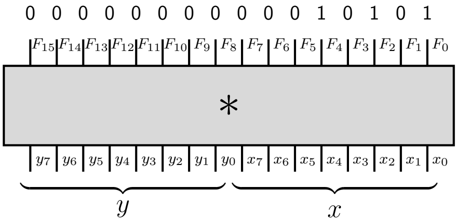


Boolean Satisfiability (SAT) Problem

Many problems (e.g. in circuit design) can be reduced to SAT.

[An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires 1 = True, 0 = False).]

One may e.g. model a *multiplier circuit* as a Boolean formula and ask if there are inputs to the circuit that result in the number 21 (10101 in binary) along the output wires (i.e. check if 21 is *prime*).

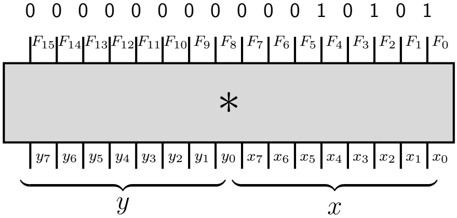


Boolean Satisfiability (SAT) Problem

Many problems (e.g. in circuit design) can be reduced to SAT.

[An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).]

One may e.g. model a *multiplier circuit* as a Boolean formula and ask if there are inputs to the circuit that result in the number 21 $(10101$ in binary) along the output wires (i.e. check if 21 is *prime*).



$$F = \neg F_{15} \wedge \neg F_{14} \wedge \neg F_{13} \wedge \neg F_{12} \wedge \neg F_{11} \wedge \neg F_{10} \wedge \neg F_9 \wedge \neg F_8 \wedge \neg F_7 \wedge \neg F_6 \wedge F_4 \wedge \neg F_3 \wedge F_2 \wedge \neg F_1 \wedge F_0$$

SAT Solvers

Boolean formula $F \longrightarrow$ $\boxed{\textbf{SAT solver}}$ $\longrightarrow$ SAT/UNSAT

Modern SAT solvers are very efficient and can solve SAT problems with *millions* of Boolean variables! Enormous progress made in the past 25 years, spurred on by the SAT competition.

http://www.satcompetition.org/

Most modern SAT solvers are based on the **DPLL** algorithm or its refinements such as **CDCL** (time complexity is exponential).

DPLL is based on *backtracking* and works on Boolean formulas in *Conjunctive Normal Form* (CNF).

Conjunctive Normal Form

A ***literal*** is a Boolean variable x_i, or its negation $\neg x_i$.

A ***clause*** is a disjunction of literals (e.g. $x_1 \vee \neg x_3 \vee x_2$).

A Boolean formula is in ***Conjunctive Normal Form*** (CNF) if it is a conjunction of one or more clauses, e.g.

$$(x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_4 \vee \neg x_1 \vee x_3) \wedge (\neg x_5)\,.$$

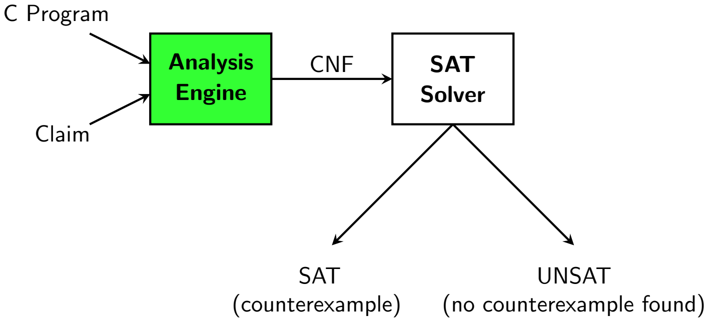
[Any Boolean formula F can be brought to CNF by applying some transformations:

- eliminating double negations (i.e. $\neg\neg P$ becomes P),
- De Morgan's laws (e.g. $\neg(P \wedge Q) = \neg P \vee \neg Q$),
- distributive laws (e.g. $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$, etc.)]

An Application of SAT Solvers: Model Checking

CBMC: the **C** **B**ounded **M**odel **C**hecker
(https://www.cprover.org/cbmc/)

Main idea: Given a C program and a claim, use a SAT solver to check if there is an execution that violates the claim.



CBMC: Bug [Catching 😇/ Finding 😈] with a SAT Solver

Developed by D. Kroening and others at CMU in 2003.

Given a C program, **CBMC** can automatically check simple safety claims, some of which are important to security:

- array bound checks,
- division by zero,
- arithmetic overflow,
- pointer checks (NULL pointer dereference),
- user-supplied assertions.

CBMC expects there to be a program entry point, i.e. a **`main()`**.

It allows the user to make *assertions* using **`assert()`** and to create *assumptions* using **`__CPROVER_assume()`**.

Using CBMC

Claims made by decorating code with assumptions and assertions.

- **assert**(e)
  aborts execution when e is false; no-op otherwise.
  `void assert(_Bool e) { if (!e) exit(); }`
- **__CPROVER_assume**(e)
  "ignores" execution when e is false; no-op otherwise.
  Program traces are restricted to those satisfying the assumption.

To find *counterexamples* to claims in a program `prog.c` we run:

```
$ cbmc --trace prog.c
```

Using **CBMC**

Claims made by decorating code with assumptions and assertions.

- **`assert`**`(e)`
  aborts execution when e *is* $false$*;* $no\text{-}op$ *otherwise.*
  `void assert(_Bool e) { if (!e) exit(); }`

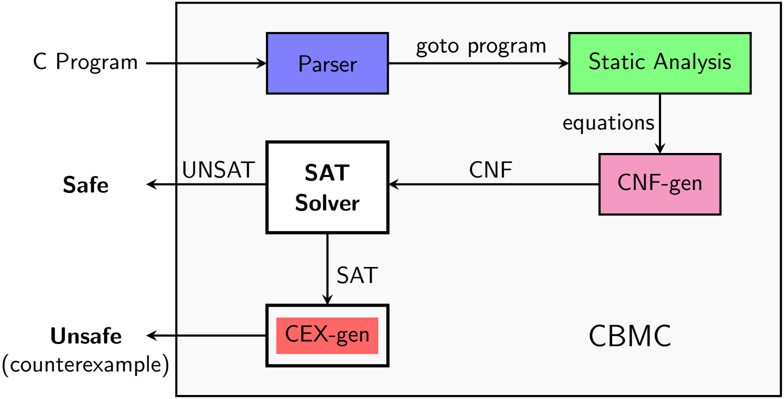
- **`__CPROVER_assume`**`(e)`
  "ignores" execution when e *is* $false$*;* $no\text{-}op$ *otherwise.*
  Program traces are restricted to those satisfying the assumption.

To find *counterexamples* to claims in a program `prog.c` we run:

```
$ cbmc --trace prog.c
```

Demo: *Let's try out* **CBMC** *to see if we can factorise integers.*

CBMC: How Does It Work?



Control Flow Simplification

- All side effects are removed

  e.g. `j=i++;` *is transformed into* `j=i; i=i+1;`

- Control flow is made *explicit*

  `continue` and **`break`** are replaced by **`goto`**

- All loops are simplified into *one form*

  e.g. **`for`**, **`do`**, **`while`** are replaced by just **`while`**

Transforming Loop-Free Programs Into Equations

It is trivial to translate a program into a set of equations if each variable is only assigned once!

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

This program is directly transformed into

$$x = a \land y = x + 1 \land z = y - 1\,.$$

Transforming Loop-Free Programs Into Equations

Static Single Assignment (SSA) form.

- Every variable is assigned *exactly once*.
- Every variable is defined *before it is used*.

When a variable is assigned multiple times, we use a new variable for each assignment.

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

```
x1 = x0 + y0;
x2 = x1*2;
a1[i0] = 100;
```

Transforming Loop-Free Programs Into Equations

Static Single Assignment (SSA) form.

- Every variable is assigned *exactly once*.
- Every variable is defined *before it is used*.

When a variable is assigned multiple times, we use a new variable for each assignment.

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

```
x1 = x0 + y0;
x2 = x1*2;
a1[i0] = 100;
```

What about conditionals?

Transforming Loop-Free Programs Into Equations

Converting conditionals to SSA.

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

```
if(v0)
  x0 = y0;
else
  x1 =  z0;
w1 = x?? // which x?
```

Transforming Loop-Free Programs Into Equations

Converting conditionals to SSA.

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

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

For each joint point add new variables with *selectors*.

Loop-Free Example

Starting from the following C code:

```c
int y;
int x;
x=x+y;
if(x!=1)
  x=2;
else
  x++;
assert(x<=3);
```

Loop-Free Example

Simplify control flow and remove side effects

```
int y;
int x;
x=x+y;
if(x!=1)
  x=2;
else
  x=x+1;
assert(x<=3);
```

Loop-Free Example

Convert to SSA (Static Single Assignment form)

```
x1 = x0+y0;
if(x1 != 1)
  x2 = 2;
else
  x3 = x1 + 1;
x4 = (x1 != 1) ? x2 : x3;
assert(x4<=3);
```

Loop-Free Example

Convert to SSA (Static Single Assignment form)

```
x1 = x0+y0;
if(x1 != 1)
  x2 = 2;
else
  x3 = x1 + 1;
x4 = (x1 != 1) ? x2 : x3;
assert(x4<=3);
```

Generate constraints (if SAT, then assertion is false):

$$
\begin{aligned}
& x_1 = x_0 + y_0 \wedge x_2 = 2 \wedge x_3 = x_1 + 1 \\
\wedge\ & ((x_1 \neq 1 \wedge x_4 = x_2) \vee (x_1 = 1 \wedge x_4 = x_3)) \quad \text{[selector]} \\
\wedge\ & \neg(x_4 \leq 3) \quad \text{[negated assertion]}
\end{aligned}
$$

Loop Unwinding

- All loops are *unwound*:
  - *can use different unwinding bounds for different loops*,
  - *can check whether unwinding is <u>sufficient</u> using a special <u>unwinding assertion</u>*.
- If a program satisfies all of its claims *and* all unwinding assertions, then it is *correct*.
- Recursive functions and backward **`goto`** are similar (use *inlining*).

Loop Unwinding

`while` loops are unwound *iterative*ly.

(**`break`**/**`continue`** replaced by **`goto`**.)

```c
void f(...) {
  ... // some code
  while(cond){
    Body;
  }
  Remainder;
}
```

Loop Unwinding

while loops are unwound *iteratively*.
(**break**/**continue** replaced by **goto**.)

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

Loop Unwinding

`while` loops are unwound *iterative*ly.
(**`break`**/**`continue`** replaced by **`goto`**.)

```c
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      while(cond){
        Body;
      }
    }
  }
  Remainder;
}
```

Loop Unwinding

while loops are unwound *iteratively*.
(**break**/**continue** replaced by **goto**.)

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

Loop Unwinding

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

```c
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      if(cond){
        Body;
        assert(!cond); //Unwinding assertion
      }
    }
  }
  Remainder;
}
```

Loop Unwinding

Assertion inserted after last iteration: violated if the program runs longer than bound permits. Positive correctness result!

```
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      if(cond){
        Body;
        assert(!cond); //Unwinding assertion
      }
    }
  }
  Remainder;
}
```

Example: Sufficient Loop Unwinding

unwind = 3

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

```
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;
}
```

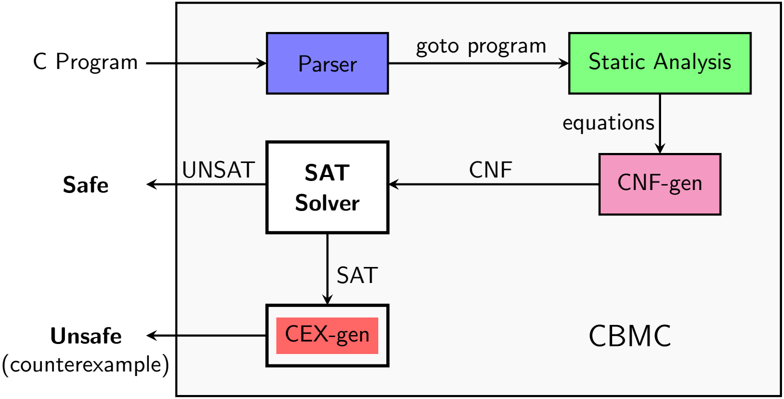
Example: Insufficient Loop Unwinding

`unwind = 3`

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

```
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;
}
```

CBMC: How Does It Work?



Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

The operations are performed on *bit vectors*.

In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening*/*bit blasting*.

Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

The operations are performed on *bit vectors*.

In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening*/*bit blasting*.

Intuitively, we can build Boolean circuits for the bit-vector operations; these can be described by Boolean formulas.

Unfortunately with a lot more variables.

CBMC: How Does It Work?

CBMC works by transforming a C program into a *set of equations*.

The main steps in **CBMC** are:

1. Simplify control flow
2. Unwind all 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: What is the Security Angle?

Some important *automatic* checks offered by **CBMC** (see command line options with `cbmc --help`):

- array bound checks,
- division by zero,
- arithmetic overflow,
- pointer checks (NULL pointer dereference).

User-supplied assertions to **CBMC** offer a very flexible tool for bug-finding.

Problems & Further Reading

See problem sheet on the course website.

Further Reading:

- **CBMC** tutorial: http://www.cprover.org/cprover-manual/cbmc/tutorial/
- **Edmund Clarke, et al.** "*Behavioral consistency of C and Verilog programs using bounded model checking.*" Proceedings 2003. Design Automation Conference. IEEE, 2003. CMU-CS-03-126.pdf

A more modern approach (using *SMT solvers* in **ESBMC**):

- **Lucas Cordeiro, et al.** "*SMT-based bounded model checking for embedded ANSI-C software.*" IEEE Transactions on Software Engineering 38.4 (2011): 957-974. https://core.ac.uk/download/pdf/59348834.pdf
- **ESBMC** (An **E**fficient **S**MT-based **B**ounded **M**odel **C**hecker): http://esbmc.org/

SMT (**S**atisfiability **M**odulo **T**heory) Solving:

- N. Bjørner, L. de Moura, L. Nachmanson, and C. Wintersteiger. "Programming Z3". https://theory.stanford.edu/ nikolaj/programmingz3.html
- **Z3** SMT Solver: https://www.microsoft.com/en-us/research/project/z3-3/

Acknowledgements: Partly based on material by D. Kroening and G. Parlato.