Program verification

Program Analysis with bounded model checking : SAT/SMT

Laure Gonnord David Monniaux

September 20, 2016

Plan

Program Analysis with bounded model checking: finding bugs
- Boolean Program analysis
- Numerical program analysis

Bonus: SAT and SMT solving
- SAT-solving
- SMT-solving

Plan

Program Analysis with bounded model checking: finding bugs
- Boolean Program analysis
- Numerical program analysis

Bonus: SAT and SMT solving
- SAT-solving
- SMT-solving

Model of program : boolean transition systems

Let us suppose we have a Boolean formula τ defining $\rightarrow$:

- variables $b_1, \ldots, b_m$ for the m bits of state before executing the computation step
- variables $b'_1, \ldots, b'_m$ for the m bits of state after executing the computation step

Unfolding

$\mathbf{b}^{(k)} = (b_1^{(k)}, \ldots, b_m^{(k)})$ value of the variables at step k.

$\tau[\mathbf{b}^{(k)}/\mathbf{b}, \mathbf{b}^{(k+1)}/\mathbf{b}']$ links $b_1^{(k)}, \ldots, b_m^{(k)}$ to $b_1^{(k+1)}, \ldots, b_m^{(k+1)}$.

Reminder: $\tau[a/b] = \tau$ where a is replaced by b

Unfolding the formula for n steps:

$$\tau[\mathbf{b}^{(0)}/\mathbf{b}, \mathbf{b}^{(1)}/\mathbf{b}'] \wedge \cdots \wedge \tau[\mathbf{b}^{(n-1)}/\mathbf{b}, \mathbf{b}^{(n)}/\mathbf{b}']$$

$\mathbf{b}^{(0)}, \ldots, \mathbf{b}^{(n)}$ is the **execution trace**: value of all bits in the system at every step.

Looking for bugs

We impose that executions:

- Start in an initial state $\mathbf{b}^{(0)}$ such that $I[\mathbf{b}^{(0)}/\mathbf{b}]$ is true.
- Ends in a final state $\mathbf{b}^{(n)}$ such that $F[\mathbf{b}^{(n)}/\mathbf{b}]$ is true.
- F defines the **negation of a desirable property** or . . . a kind of **bug**.

$$I[\mathbf{b}^{(0)}/\mathbf{b}] \wedge \tau[\mathbf{b}^{(0)}/\mathbf{b}, \mathbf{b}^{(1)}/\mathbf{b}'] \wedge \cdots \wedge \tau[\mathbf{b}^{(n-1)}/\mathbf{b}, \mathbf{b}^{(n)}/\mathbf{b}'] \wedge F[\mathbf{b}^{(n)}/\mathbf{b}]$$

defines traces of exact length n starting in a state defined by I and ending in a state defined by F.

Solving the problem

We have a Boolean formula with free variables $b_1^{(0)}, \ldots, b_m^{(0)}, \ldots, b_1^{(n)}, \ldots, b_m^{(n)}$.

A solution of this formula is an **assignment** to these variables that makes the formula true.

Example: formula $((a \vee b) \wedge \neg b) \vee c$ has solutions

Solving the problem

We have a Boolean formula with free variables $b_1^{(0)}, \ldots, b_m^{(0)}, \ldots, b_1^{(n)}, \ldots, b_m^{(n)}$.

A solution of this formula is an **assignment** to these variables that makes the formula true.

Example: formula $((a \vee b) \wedge \neg b) \vee c$ has solutions

- $(a, b, c) = (\text{true}, \text{false}, \text{false})$

Solving the problem

We have a Boolean formula with free variables $b_1^{(0)}, \ldots, b_m^{(0)}, \ldots, b_1^{(n)}, \ldots, b_m^{(n)}$.

A solution of this formula is an **assignment** to these variables that makes the formula true.

Example: formula $((a \vee b) \wedge \neg b) \vee c$ has solutions

- $(a, b, c) = (\text{true}, \text{false}, \text{false})$
- $(a, b, c) = (\text{false}, \text{false}, \text{true})$

Try increasing values of n.

Small example

$m = 5$ bits $b_1, \ldots, b_m$.
Initial states: true...
Transition relation τ:

$$b_2' = b_1 \wedge b_3' = b_2 \wedge b_4' = b_3 \wedge b_5' = b_4 \wedge b_1' = \neg b_1$$

Small example

$m = 5$ bits $b_1, \ldots, b_m$.
Initial states: true...
Transition relation τ:

$$b_2' = b_1 \wedge b_3' = b_2 \wedge b_4' = b_3 \wedge b_5' = b_4 \wedge b_1' = \neg b_1$$

In other words: b_1 alternates between 0 and 1, and bits $b_1 \to b_2 \to b_3 \to b_4 \to b_5$.
$0, 1, 0, 1, 0, 1 \ldots$ moving.

"Buggy" final states: $b_4 \wedge b_5$.
Are they reachable within $n = 10$ steps?

Solution

For $n = 0$: formula is $b_4^{(0)} \wedge b_5^{(0)}$, solution $(b_1^{(0)}, b_2^{(0)}, b_3^{(0)}, b_4^{(0)}, b_5^{(0)}) = (\text{false}, \text{false}, \text{false}, \text{true}, \text{true})$.

Don't forget the initial states!

Solution

For $n = 0$: formula is $b_4^{(0)} \wedge b_5^{(0)}$, solution $(b_1^{(0)}, b_2^{(0)}, b_3^{(0)}, b_4^{(0)}, b_5^{(0)}) = (\text{false}, \text{false}, \text{false}, \text{true}, \text{true})$.

Don't forget the initial states!

And for $n = 10$... solve

$$\tau[\mathbf{b}^{(0)}/\mathbf{b}, \mathbf{b}^{(1)}/\mathbf{b}'] \wedge \cdots \wedge \tau[\mathbf{b}^{(9)}/\mathbf{b}, \mathbf{b}^{(10)}/\mathbf{b}'] \wedge b_4^{(10)} = b_5^{(10)}$$

Our intuition is that this formula has no solution... but how to automate this? **SAT Solving**

Plan

Program Analysis with bounded model checking: finding bugs

- Boolean Program analysis
- Numerical program analysis

Bonus: SAT and SMT solving

- SAT-solving
- SMT-solving

From program to SMT

```
if (x < 5) {
   y = x+3;
} else {
   y = x+2;
}
```

From program to SMT

```
if (x < 5) {
   y = x+3;
} else {
   y = x+2;
}
```

$$(x < 5 \wedge y' = x + 3) \vee (x \geq 5 \wedge y' = x + 2)$$

SMT-solvers can handle this kind of formula.

Conversion from program to SMT

- Conversion looks like compilation into SSA-form (intermediate representation in compilers e.g. modern gcc and LLVM).
- Loops are unrolled into nested if-then-else.
- If-then-else's in source code introduce $\vee$ in formulas (or "if then else" Boolean operators).
- $\vee$ introduce exponential complexity in SAT/SMT solving.
- Thus cost may be exponential in loop unrolling.

Plan

Program Analysis with bounded model checking: finding bugs
- Boolean Program analysis
- Numerical program analysis

Bonus: SAT and SMT solving
- SAT-solving
- SMT-solving

Plan

Program Analysis with bounded model checking: finding bugs
- Boolean Program analysis
- Numerical program analysis

Bonus: SAT and SMT solving
- SAT-solving
- SMT-solving

The problem

Given a Boolean formula:

- Test whether it has solutions.
- If it has solutions, give one.

Known to be NP-complete!

SAT in CNF

We have a formula F with nested $\wedge, \vee, \neg \ldots$
but most solvers accept only formulas in **conjunctive normal form** (CNF).

CNF is conjunction of disjunction of literals. Ex:

$$(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z)$$

Tseiting encoding

Take as example: $((a \vee b) \wedge \neg b) \vee c$

Add variables $x = a \vee b$, $y = x \wedge \neg b$, $z = y \vee c$.

$x = a \vee b$ is equivalent to

$$(a \Rightarrow x) \wedge (b \Rightarrow x) \wedge (x \Rightarrow (a \vee b))$$

thus to

$$(\neg a \vee x) \wedge (\neg b \vee x) \wedge (\neg x \vee a \vee b)$$

Constraint solving: unit propagation

Each conjunct is a **constraint** on variables. We must satisfy them all!

Work with **partial assignments**: give values to some of the variables.
"Try $a = \text{false}, b = \text{false}$ and see what happens."

We must satisfy $\neg x \vee a \vee b$: if $a = b = \text{false}$, then $\neg x$ must be true, thus $x = \text{false}$. **Unit propagation.**

Practical example

Is there a mine at the light blue square?

Practical example

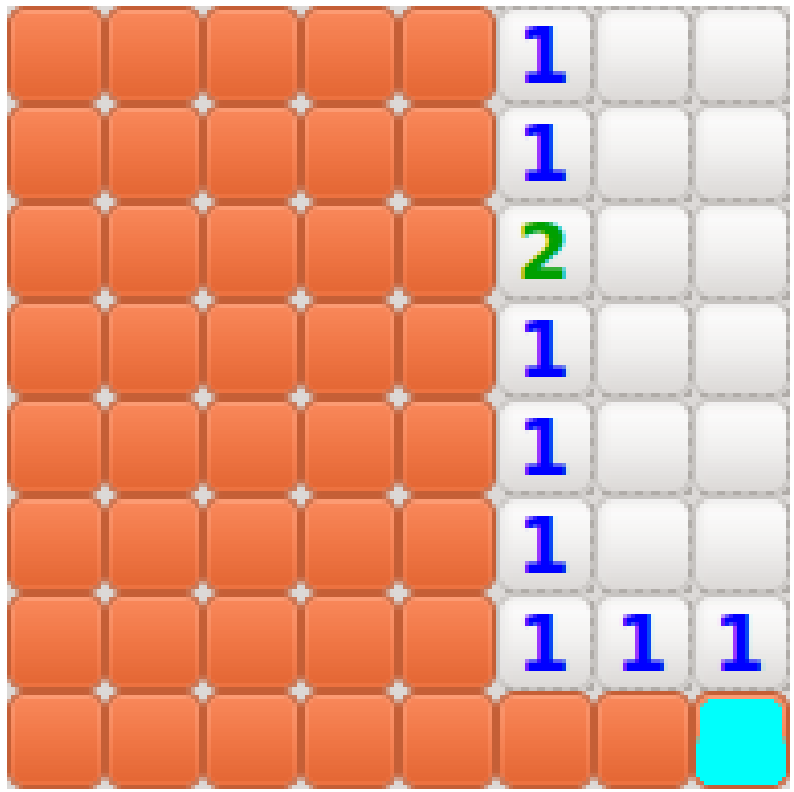
The pink square imposes a **constraint**: total number of mines in neighbourhood is exactly 1. Since there is already one mine, the light blue square cannot contain one.

Unit propagation

- You have already made a number of **choices** on some variables.
- These choices, through constraints, **impose** values to other variables.
- Valid for SAT as well as for Mines or Sudoku.

Branching

Sometimes, unit propagation does not suffice. One must "work on a hypothesis".
Is there a mine on the light blue square?



Method

Search with backtracking:

- Make a "work hypothesis".
- Propagate the consequences (unit propagation).
- If encountering an absurdity, **backtrack**.

Method

Search with backtracking:

- Make a “work hypothesis”.
- Propagate the consequences (unit propagation).
- If encountering an absurdity, **backtrack**.

When solving Sudoku

- If the problem is “easy”, unit propagation suffices.
- The more difficult the problem is, the more one has to make choices and backtrack.

DPLL algorithm

- Unit propagation.
- Make choices: select a variable, assign it to true or false.
- If reaching a contradiction: the last assignment was absurd; backtrack and replace last assignment by its negation.

Time complexity?

DPLL algorithm

- Unit propagation.
- Make choices: select a variable, assign it to true or false.
- If reaching a contradiction: the last assignment was absurd; backtrack and replace last assignment by its negation.

Time complexity? Still exponential, of course!

DPLL with learning

$$(a \vee b \vee c \vee \neg d) \wedge (\neg a \vee \neg b \vee \neg c \vee \neg d) \wedge (\neg b \vee c)$$

Choose $d = \text{true}$. Constraints become:

$$(a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (\neg b \vee c)$$

Choose $a = \text{true}$. Constraints become:

$$(\neg b \vee \neg c) \wedge (\neg b \vee c)$$

Choose $b = \text{true}$. The formula becomes $(\neg c) \wedge c$.
Contradiction!
Thus $F \wedge a \wedge b$ unsat. In other words, $F \Rightarrow (\neg a \vee \neg b)$.

Thus I can add $\neg a \vee \neg b$ to the original problem without changing solutions!

Modern SAT solving

DIMACS standardized file format.

Advanced technology.

- Solvers zChaff, MiniSAT etc.
- Many improvements, heuristics...
- Very clever implementation techniques.
- **Mass industrial use.**

Mass industrial use of SAT solving

In **EDA (electronic design automation)**.
Prove that two circuit designs (without memory) are equivalent: $F(b_1, \ldots, b_m)$ and $G(b_1, \ldots, b_m)$ equivalent iff $\forall b_1, \ldots, b_m \; F(b_1, \ldots, b_m) = G(b_1, \ldots, b_m)$.
How?

Mass industrial use of SAT solving

In **EDA (electronic design automation)**.
Prove that two circuit designs (without memory) are equivalent: $F(b_1, \ldots, b_m)$ and $G(b_1, \ldots, b_m)$ equivalent iff $\forall b_1, \ldots, b_m \; F(b_1, \ldots, b_m) = G(b_1, \ldots, b_m)$.
How?

Show that $F(b_1, \ldots, b_m) \neq G(b_1, \ldots, b_m)$ is unsat.

NB: $(x_1, x_2, x_3, x_{m'}) \neq (y_1, y_2, y_3, y_{m'})$ iff $(x_1 \oplus y_1) \vee \cdots \vee (x_{m'} \oplus y_{m'})$, where $\oplus$ is exclusive-or (XOR).

For circuits with memory

Unroll execution traces, as seen before!

Plan

Program Analysis with bounded model checking: finding bugs
- Boolean Program analysis
- Numerical program analysis

Bonus: SAT and SMT solving
- SAT-solving
- SMT-solving

How about numbers?

```
int x, y, z;
assume(0 >= x && x <= 2 || x >= 4 && x <= 6);
assume(0 >= y && y <= 5);
z = x + y;
assert(x + y <= 12);
```

How can we prove unsat:

$$((0 \leq x \leq 2) \vee (4 \leq x \leq 6)) \wedge (0 \leq y \leq 5) \wedge (z = x+y) \wedge (x+y > 12)$$

Bit-blasting

Assume **int** is 32-bit.
Expand $0 \geq x$, $z = x + y$... into Boolean gates using adders, comparators, etc.
Obtain a pure Boolean circuit.
(Same as C-to-hardware compilation.)

Apply SAT-solving!

DPLL(T)

Add Boolean variables:

$$((0 \leq x \leq 2) \vee (4 \leq x \leq 6)) \wedge (0 \leq y \leq 5) \wedge (x = x+y) \wedge (x+y > 12) \tag{1}$$

with $a \stackrel{\triangle}{=} 0 \geq x$, $b \stackrel{\triangle}{=} x \leq 5$, $c \stackrel{\triangle}{=} 0 \geq y$, $d \stackrel{\triangle}{=} y \leq 5$,
$e \stackrel{\triangle}{=} x = x + y$, $f \stackrel{\triangle}{=} x + y > 12$, $g \stackrel{\triangle}{=} x \geq 4$, $h \stackrel{\triangle}{=} x \leq 6$.

Formula 1 becomes

$$((a \wedge b) \vee (g \wedge h)) \wedge c \wedge d \wedge e \wedge f \tag{2}$$

DPLL(T) suite

$$((a \wedge b) \vee (g \wedge h)) \wedge c \wedge d \wedge e \wedge f \tag{3}$$

Solution: every variable to true!

But this means $b \stackrel{\triangle}{=} x \leq 2$ and $g \stackrel{\triangle}{=} g \geq 4$ both true! This is **impossible** with respect to integer arithmetic!

Add $\neg(b \wedge g)$ $(\neg b \vee \neg g)$ to the Boolean constraints, and start again.

DPLL(T) explained

- Find a Boolean solution using DPLL.
- Check if feasible with respect to arithmetic.
- If not, add a Boolean constraint and restart.

In practice: DPLL interleaved with arithmetic (theory) solving.

Linear arithmetic over reals/rationals: simplex algorithm.

Tools

Industrial:

- Microsoft Z3
- SRI Yices

Academic:

- VeriT (Nancy)
- MathSAT
- OpenSMT
- CVC3...

Example of tools

CBMC `http://www.cprover.org/cbmc/`
Bounded Model Checking for ANSI-C

Another example: Sage

- Fuzzing = throw random data at programs and see if they crash.
- Better fuzzing: solve SMT-formulas for obtaining inputs that exerce certain program paths.
- Combines binary analysis (disassembling) and SMT-solving.
- Patrice Godefroid @Microsoft Research.
- Industrial use for detecting security bugs in file format and protocol parsers.
- (Not sound: can fail to detect problems.)