

Bounded model checking, SAT-solving, Sudoku and Mines

David Monniaux

CNRS / VERIMAG

14 février 2011



Plan

Model-checking basics

Boolean programs

SAT-solving

SMT-solving



BDD-based model-checking

- ▶ Compute the **set of reachable states**.
- ▶ Explicit state : memory use proportional to $|S|$, S set of reachable states. . .but $|S| = 2^n$ where n is the memory size !
- ▶ Implicit state : compression of the set of states, but BDDs can still be exponential, choice of order of variables. . .**Memory explosion**.
- ▶ Set of states reachable in **any number of steps** from initialization.
- ▶ Reachability is **PSPACE-complete**.

The **state explosion** problem.



Bounded model-checking

- ▶ Looks for a path **of given length** n from the initial states to the undesirable states.
- ▶ Is **NP-complete** or worse (depending on kinds of data), due exponential number of paths.
- ▶ Efficient SAT-based search algorithms(DPLL etc.). **Do not blow up memory.**
- ▶ Time explosion with increased n .

Can **search for bugs** but not prove their absence!



Distinguish two kinds of applications

- ▶ Find bugs.
- ▶ **Prove their absence.** Must be **sound** : should not say “no bugs” when bugs are present.

Plan

Model-checking basics

Boolean programs

SAT-solving

SMT-solving



Transition relations

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

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

Unfolding

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

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

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

Unfolding the formula for n steps :

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

$\mathbf{b}^{(0)}, \dots, \mathbf{b}^{(n)}$ 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 an initial 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**.

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

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
 $b_1^{(0)}, \dots, b_m^{(0)}, \dots, b_1^{(n)}, \dots, b_m^{(n)}$.

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
 $b_1^{(0)}, \dots, b_m^{(0)}, \dots, b_1^{(n)}, \dots, b_m^{(n)}$.

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 $b_1^{(0)}, \dots, b_m^{(0)}, \dots, b_1^{(n)}, \dots, b_m^{(n)}$.

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, \dots, 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, \dots, 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

$b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4 \rightarrow b_5$.

0, 1, 0, 1, 0, 1... 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 $n = 10 \dots$ solve

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

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

Plan

Model-checking basics

Boolean programs

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 $\wedge, \vee, \neg \dots$
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 pink 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 : $F(b_1, \dots, b_m)$ and $G(b_1, \dots, b_m)$ equivalent iff $\forall b_1, \dots, b_m \ F(b_1, \dots, b_m) = G(b_1, \dots, b_m)$.

How ?

Mass industrial use of SAT solving

In **EDA (electronic design automation)**.

Prove that two circuit designs (without memory) are equivalent : $F(b_1, \dots, b_m)$ and $G(b_1, \dots, b_m)$ equivalent iff $\forall b_1, \dots, b_m \ F(b_1, \dots, b_m) = G(b_1, \dots, b_m)$.

How ?

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

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

For circuits with memory

Unroll execution traces, as seen before !

Plan

Model-checking basics

Boolean programs

SAT-solving

SMT-solving



How about numbers?

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

How can we prove unsat :

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

Bit-blasting

Assume **int** is 32-bit.

Expand $0 \leq x$, $z = x + y \dots$ 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 :

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

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

Formula 1 becomes

$$((a \wedge b) \vee (g \wedge h)) \wedge c \wedge d \wedge e \wedge f \quad (2)$$

DPLL(T) suite

$$((a \wedge b) \vee (g \wedge h)) \wedge c \wedge d \wedge e \wedge f \quad (3)$$

Solution : every variable to true !

But this means $b \triangleq x \leq 2$ and $g \triangleq g \geq 4$ 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.

