COSE419: Software Verification

Lecture 5 — Problem Solving using SMT Solver

Hakjoo Oh
2024 Spring

The Z3 SMT Solver

- A popular SMT solver from Microsoft Research:

  `https://github.com/Z3Prover/z3`

- Supported theories:
  - Propositional Logic
  - Theory of Equality
  - Uninterpreted Functions
  - Arithmetic
  - Arrays
  - Bit-vectors, ...
- References
  - Programming Z3
    `https://z3prover.github.io/papers/programmingz3.html`
  - Z3 API in Python
    `http://ericpony.github.io/z3py-tutorial/guide-examples.htm`

SMT Module

```
open Smt

let check_sat f =
  let _ = print_endline ("\n" ^ Fmla.to_string f) in
  let (v, model_opt) = Solver.check_satisfiability [f] in
  let _ = print_endline (Solver.string_of_satisfiability v) in
    match model_opt with
    | Some model -> print_endline (Model.to_string model)
    | None -> ()
```

Propositional Logic

```
let p = Expr.create_var (Expr.sort_of_bool ()) ~name:"p"
let q = Expr.create_var (Expr.sort_of_bool ()) ~name:"q"
let r = Expr.create_var (Expr.sort_of_bool ()) ~name:"r"
let f1 = Fmla.create_and [
  Fmla.create_imply (Fmla.create_exp p) (Fmla.create_exp q);
  Fmla.create_iff r (Fmla.create_not (Fmla.create_exp q));
  Fmla.create_or [Fmla.create_not (Fmla.create_exp p);
           (Fmla.create_exp r)]
]

let _ = check_sat f1
```

```
(and (=> p q) (= r (not q)) (or (not p) r))
SAT
(define-fun r () Bool
  false)
(define-fun q () Bool
  true)
(define-fun p () Bool
  false)
```

Example: Implementing SAT Solver

```
type var = string
type formula =
  | True
  | False
  | Var of var
  | Not of formula
  | And of formula * formula
  | Or of formula * formula
  | Imply of formula * formula
  | Iff of formula * formula

let rec trans : formula -> Fmla.t
=fun f -> (* TODO *)

let check_sat : formula -> bool * Model.t option
  let v, model_opt = Solver.check_satisfiability [trans f] in
    if Solver.is_sat v then
      match model_opt with
      | Some model -> (true, Some model)
      | None -> raise (Failure "check_sat")
    else (false, None)
```

Integer Arithmetic

```
let x = Expr.create_var (Expr.sort_of_int ()) ~name:"x"
let y = Expr.create_var (Expr.sort_of_int ()) ~name:"y"
let f2 = Fmla.create_and [
  Fmla.create_exp (Expr.create_gt x (Expr.of_int 2));
  Fmla.create_exp (Expr.create_lt y (Expr.of_int 10));
  Fmla.create_exp (Expr.create_eq (Expr.create_add x
      (Expr.create_mul (Expr.of_int 2)
      (Expr.of_int 7))
]
```

```
(and (> x 2) (< y 10) (= (+ x (* 2 y)) 7))
SAT
(define-fun y () Int
  0)
(define-fun x () Int
  7)
```

Real Arithmetic

```
let x = Expr.create_var (Expr.sort_of_real ()) ~name:"x"
let y = Expr.create_var (Expr.sort_of_real ()) ~name:"y"
let f3 = Fmla.create_and [
  Fmla.create_exp (
    Expr.create_gt (
      Expr.create_add (Expr.create_mul x x)
                      (Expr.create_mul y y))
      (Expr.of_int 3));
  Fmla.create_exp (
    Expr.create_lt (
      Expr.create_add (
        Expr.create_power x
          (Expr.of_int 3)) y)
    (Expr.of_int 5))
]
```

```
(let ((a!1 (< (+ (^ x (to_real 3)) y) (to_real 5))))
  (and (> (+ (* x x) (* y y)) (to_real 3)) a!1))
SAT
(define-fun y () Real
  (- 1.0))
(define-fun x () Real
  (/ 61.0 40.0))
```

BitVectors

```
let x = Expr.create_var (Expr.sort_of_bitvector 32) ~name:"x"
let y = Expr.create_var (Expr.sort_of_bitvector 32) ~name:"y"
let c2 = Expr.create_bv_numeral "2" 32
let c3 = Expr.create_bv_numeral "3" 32
let c24 = Expr.create_bv_numeral "24" 32
let f4 = Expr.create_eq (Expr.create_land x y) y
let f5 = Expr.create_eq (Expr.create_shl x c2) c3
let f6 = Expr.create_eq (Expr.create_shl x c2) c24
```

```
(= (bvand x y) y)
SAT
(define-fun y () (_ BitVec 32)
  #xfffffffe)
(define-fun x () (_ BitVec 32)
  #xfffffffe)

(= (bvshl x #x00000002) #x00000003)
UNSAT

(= (bvshl x #x00000002) #x00000018)
SAT
(define-fun x () (_ BitVec 32) #x00000006)
```

Problem 1: Program Equivalence

Consider the two code fragments:

```
if (!a&&!b) then h
else if (!a) then g else f
```

```
if (a) then f
else if (b) then g else h
```

The latter might have been generated from an optimizing compiler. We would like to prove that the two programs are equivalent.

Encoding in Propositional Logic

The if-then-else construct can be replaced by a PL formula as follows:

$$\text{if } x \text{ then } y \text{ else } z \equiv (x \wedge y) \vee (\neg x \wedge z)$$

The problem of checking the equivalence is to check the validity of the formula:

$$\begin{aligned} F : (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f) \\ \leftrightarrow a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h) \end{aligned}$$

If $\neg F$ is unsatisfiable, the two expressions are equivalent.

Implementation

```
type var = string
type exp =
  | Var of var
  | Not of exp
  | And of exp * exp
  | If of exp * exp * exp

let e1 = If (And (Not (Var "a"),
                  Not (Var "b")),
             Var "h",
             If (Not (Var "a"),
                 Var "g",
                 Var "f"))
let e2 = If (Var "a",
             Var "f",
             If (Var "b", Var "g", Var "h"))

let verify_equiv : exp -> exp -> bool
=fun e1 e2 -> (* TODO *)
```

Problem 2: Seat Assignment

Consider three persons A, B, and C who need to be seated in a row. There are three constraints:

- A does not want to sit next to C
- A does not want to sit in the leftmost chair
- B does not want to sit to the right of C

We would like to check if there is a seat assignment for the three persons that satisfies the above constraints.

Encoding in Propositional Logic

To encode the problem, let X_{ij} be boolean variables such that

$$X_{ij} \iff \text{person } i \text{ seats in chair } j$$

We need to encode two types of constraints.

- Valid assignments:
  - Every person is seated
  $$\bigwedge_i \bigvee_j X_{ij}$$
  - Every seat is occupied
  $$\bigwedge_j \bigvee_i X_{ij}$$
  - One person per seat
  $$\bigwedge_{i,j} (X_{ij} \rightarrow \bigwedge_{k \neq j} \neg X_{ik})$$

Encoding in Propositional Logic

- Problem constraints:
  - A does not want to sit next to C:

$$(X_{00} \rightarrow \neg X_{21}) \wedge (X_{01} \rightarrow (\neg X_{20} \wedge \neg X_{22})) \wedge (X_{02} \rightarrow \neg X_{21})$$

  - A does not want to sit in the leftmost chair

$$\neg X_{00}$$

  - B does not want to sit to the right of C

$$(X_{20} \rightarrow \neg X_{11}) \wedge (X_{21} \rightarrow \neg X_{12})$$

Implementation

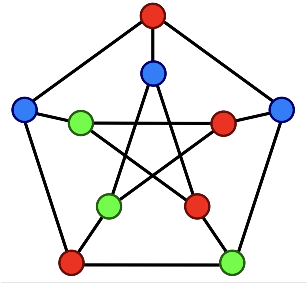
```
(* constraint *)
type const =
  | X of int * int
  | True
  | False
  | And of const list
  | Or of const list
  | Imply of const * const
  | Not of const

let encode : unit -> const
=fun () -> (* TODO *)

let trans : const -> Fmla.t
=fun _ -> (* TODO *)
```

Problem 3: Graph Coloring

Given:

- A graph $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$ and $E \subseteq V \times V$.
- A finite set $C = \{c_1, \dots, c_k\}$ of colors.

Can we assign each vertex $v \in V$ a color $\textbf{color}(v) \in C$ such that for every edge $(v, w) \in E$, $\textbf{color}(v) \neq \textbf{color}(w)$?



Encoding

$$X_{ij} \iff \text{vertex } v_i \text{ is assigned color } c_j$$

- Every vertex is assigned at least one color:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} X_{ij}$$

- Neighbors are not assigned the same color:

$$\bigwedge_{(i,j)\in E} \bigwedge_{t=1}^{k} \neg(X_{it} \wedge X_{jt})$$

- Every vertex is assigned not more than one color:

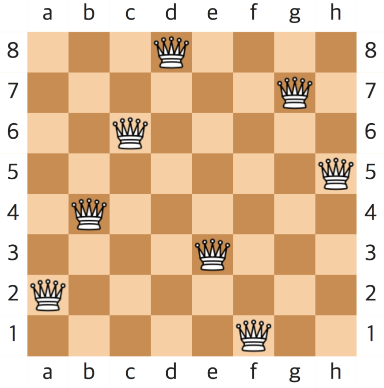
$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \bigwedge_{j'=1}^{k} j \neq j' \rightarrow \neg(X_{ij} \wedge X_{ij'})$$

Implementation

```
type node = int
type edge = node * node
type graph = node list * edge list
type color = int

let instance1 : graph * color list = (
  ([1; 2; 3], [(1, 2); (2, 3)]),
  [1; 2]
)

let coloring : graph * color list -> bool * (node * color) list option
=fun (graph, colors) -> (* TODO *)
```

Problem 4: Eight Queens

The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.



Encoding

Define boolean variables Q_i as follows:

$$Q_i : \text{the column position of the queen in row } i$$

- Each queen is in a column $\{1, \ldots, 8\}$:

$$\bigwedge_{i=1}^{8} 1 \leq Q_i \wedge Q_i \leq 8$$

- No queens share the same column:

$$\bigwedge_{i=1}^{8} \bigwedge_{j=1}^{8} (i \neq j \rightarrow Q_i \neq Q_j)$$

- No queens share the same diagonal:

$$\bigwedge_{i=1}^{8} \bigwedge_{j=1}^{i} (i \neq j \rightarrow Q_i - Q_j \neq i - j \wedge Q_i - Q_j \neq j - i)$$

Implementation

```
type solution = int list

type exp =
  | Q of int
  | Int of int
  | Sub of exp * exp

type const =
  | And of const list
  | Or of const list
  | Imply of const * const
  | Le of exp * exp
  | Neq of exp * exp

let encode : unit -> const (* TODO *)
let trans : const -> Fmla.t (* TODO *)
let model2solution : Model.t -> solution  (* TODO *)
```

Exercise: Finding All Solutions

There are multiple solutions to the eight queens problem. For example, the following can also be a solution:

```
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
```

How many different solutions can you find?

Problem 5: Sudoku

Insert the numbers in the $\mathbf{9} \times \mathbf{9}$ board so that each row, column, and $\mathbf{3} \times \mathbf{3}$ boxes must contain digits 1 through 9 exactly once.

	8	2			5			
			6			2		
6					1			
5								
			4		2			
								6
			8					5
		8			9			
			5			4	3	

Encoding in SMT formulas

$$X_{ij} : \text{number in position } (i, j), \text{ for } i, j \in [1, 9]$$

- Each cell contains a value in $\{1, \ldots, 9\}$:

$$\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} 1 \leq X_{ij} \leq 9$$

- Each row contains a digit at most once:

$$\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} \bigwedge_{k=0}^{8} (j \neq k \rightarrow X_{ij} \neq X_{ik})$$

- Each column contains a digit at most once:

$$\bigwedge_{j=0}^{8} \bigwedge_{i=0}^{8} \bigwedge_{k=0}^{8} (i \neq k \rightarrow X_{ij} \neq X_{kj})$$

Encoding in SMT formulas

- Each $\mathbf{3} \times \mathbf{3}$ square contains a digit at most once:

$$\bigwedge_{i_0=0}^{2} \bigwedge_{j_0=0}^{2} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{i'=0}^{2} \bigwedge_{j'=0}^{2} \big((i \neq i' \vee j \neq j') \rightarrow X_{3i_0+i,3j_0+j} \neq X_{3i_0+i',3j_0+j'}\big)$$

- Board configuration (stored in B, where 0 means empty):

$$\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} (B[i][j] \neq 0 \rightarrow B[i][j] = X_{ij})$$

Summary

Problem solving using an SMT solver:

1. Express the problem as a satisfiability problem in logic
2. Use the SMT solver to decide the satisfiability