Logic and Mechanized Reasoning

Using SMT Solvers

Marijn J.H. Heule

Carnegie Mellon University

SMT-LIB

Example: Magic Squares

Application: Verification

SMT-LIB

Example: Magic Squares

Application: Verification

SMT-LIB: Introduction

Consists of five blocks:

- theory `(set-logic ...)`, e.g. QF_UF and QF_LIA
- variables, functions, and types `(declare-const ...)`
- a list of constraints `(assert ...)`
- solving the problem `(check-sat)`
- termination the solver `(exit)`

SMT-LIB: Introduction

Consists of five blocks:

- theory `(set-logic ...)`, e.g. QF_UF and QF_LIA
- variables, functions, and types `(declare-const ...)`
- a list of constraints `(assert ...)`
- solving the problem `(check-sat)`
- termination the solver `(exit)`

Variable and functions:

- `(declare-const name type)`
- `(declare-fun name (inputTypes) outputType)`
- `(define-fun name (inputTypes) outputType (body))`

SMT-LIB: QF_UF example

Example

Does there exist a satisfying assignment for $p \wedge \neg p$?

```
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat) ; should be UNSAT
(exit)
```

SMT-LIB: QF_LIA example

Example

Does there exist an integer x that is larger than an integer y?

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> x y))
(check-sat) ; should be SAT
(get-model)
(exit)
```

SMT-LIB

Example: Magic Squares

Application: Verification

Magic Squares: Introduction

A $n \times n$ square is called a magic square if each number from 1 to n^2 occurs uniquely and the sum of all rows, columns, and diagonals is the same: $(n^3 + n)/2$

1	9	12	20	23	→ 65
17	25	3	6	14	→ 65
8	11	19	22	5	→ 65
24	2	10	13	16	→ 65
15	18	21	4	7	→ 65
↓ 65	↓ 65	↓ 65	↓ 65	↓ 65	

↙ 65 ↘ 65

Magic Squares: Linear Arithmetic for 3×3 Magic Square

```
(set-logic QF_LIA)
(declare-const m_0_0 Int)
(declare-const m_0_1 Int)
...
(declare-const m_2_2 Int)
(assert (and (> m_0_0 0) (<= m_0_0 9)))
(assert (and (> m_0_1 0) (<= m_0_1 9)))
...
(assert (and (> m_2_2 0) (<= m_2_2 9)))
(assert (distinct m_0_0 m_0_1 m_0_2 m_1_0
                  m_1_1 m_1_2 m_2_0 m_2_1 m_2_2))
(assert (= 15 (+ m_0_0 m_0_1 m_0_2)))
(assert (= 15 (+ m_1_0 m_1_1 m_1_2)))
...
(assert (= 15 (+ m_2_0 m_1_1 m_0_2)))
(check-sat)
(get-model)
(exit)
```

Magic Squares: Bitvectors for 3×3 Magic Square

```
(set-logic QF_BV)
(declare-const m_0_0 (_ BitVec 16))
(declare-const m_0_1 (_ BitVec 16))
...
(declare-const m_2_2 (_ BitVec 16))
(assert (and (bvugt m_0_0 #x0000) (bvule m_0_0 #x0009)))
(assert (and (bvugt m_0_1 #x0000) (bvule m_0_1 #x0009)))
...
(assert (and (bvugt m_2_2 #x0000) (bvule m_2_2 #x0009)))
(assert (distinct m_0_0 m_0_1 m_0_2 m_1_0
                  m_1_1 m_1_2 m_2_0 m_2_1 m_2_2))
(assert (= #x000f (bvadd m_0_0 m_0_1 m_0_2)))
(assert (= #x000f (bvadd m_1_0 m_1_1 m_1_2)))
...
(assert (= #x000f (bvadd m_2_0 m_1_1 m_0_2)))
(check-sat)
(get-model)
(exit)
```

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

QF_LIA: the solver applies (exponentially) many SAT calls

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

QF_LIA: the solver applies (exponentially) many SAT calls

When using QF_BV, the solver applies bitblasting: every bit in each bitvector is turned into a propositional variable. Each constraint, such as `(> m_2_2 0)` is turned into many clauses.

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

QF_LIA: the solver applies (exponentially) many SAT calls

When using QF_BV, the solver applies bitblasting: every bit in each bitvector is turned into a propositional variable. Each constraint, such as `(> m_2_2 0)` is turned into many clauses.

QF_BV: the solver applies a single SAT call

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

QF_LIA: the solver applies (exponentially) many SAT calls

When using QF_BV, the solver applies bitblasting: every bit in each bitvector is turned into a propositional variable. Each constraint, such as `(> m_2_2 0)` is turned into many clauses.

QF_BV: the solver applies a single SAT call

Compare: $n \geq 5$ is hard for QF_LIA, $n \leq 10$ is easy for QF_BV

Magic Squares: Demo

```
SAT with assignment:
m_2_2 ↦ 2
m_2_1 ↦ 9
m_2_0 ↦ 4
m_1_2 ↦ 7
m_1_1 ↦ 5
m_1_0 ↦ 3
m_0_2 ↦ 6
m_0_1 ↦ 1
m_0_0 ↦ 8

Square:
8 1 6
3 5 7
4 9 2
```

SMT-LIB

Example: Magic Squares

Application: Verification

Verification: Equivalence Checking

SAT and SMT solvers are crucial for verification tasks

- Equivalence checking
- Bounded model checking

Equivalence checking:

- Are two hardware/software designs functionally equivalent?
- Does any input to both produces the same output?
- Typically one is unoptimized and the other is optimized

Verification: Example of the Power of 3

```c
int power3(int in)
{
   int i, out_a;
   out_a = in;
   for (i = 0; i < 2; i++)
     out_a = out_a * in;
  return out_a;
}
```

```c
int power3_new(int in)
{
   int out_b;

   out_b = (in * in) * in;

   return out_b;
}
```

$$\begin{aligned}\Gamma_a \equiv\ & (out0\_a = in0\_a) \wedge (out1\_a = out0\_a \times in0\_a) \wedge \\ & (out2\_a = out1\_a \times in0\_a) \\ \Gamma_b \equiv\ & out0\_b = (in0\_b \times in0\_b) \times in0\_b\end{aligned}$$

To show these programs are equivalent, we must show the following formula is valid:

$$in0\_a = in0\_b \wedge \Gamma_a \wedge \Gamma_b \implies out2\_a = out0\_b$$

Verification: Integers

```
(declare-const out0_a Int)
(declare-const out1_a Int)
(declare-const in0_a  Int)
(declare-const out2_a Int)
(declare-const out0_b Int)
(declare-const in0_b  Int)
(define-fun gamma_a () Bool
    (and (= out0_a in0_a)
        (and (= out1_a (* out0_a in0_a))
            (= out2_a (* out1_a in0_a)))))
(define-fun gamma_b () Bool
    (= out0_b (* (* in0_b in0_b) in0_b)))
(define-fun gamma_in () Bool
    (= in0_a in0_b))
(define-fun gamma_out () Bool
    (= out2_a out0_b ))
(assert (not (=> (and gamma_in gamma_a gamma_b) gamma_out)))
(check-sat)
```

Verification: Bitvectors

```
(declare-const out0_a (_ BitVec 128))
(declare-const out1_a (_ BitVec 128))
(declare-const in0_a  (_ BitVec 128))
(declare-const out2_a (_ BitVec 128))
(declare-const out0_b (_ BitVec 128))
(declare-const in0_b  (_ BitVec 128))

(define-fun gamma_a () Bool
    (and (= out0_a in0_a)
        (and (= out1_a (bvmul out0_a in0_a))
            (= out2_a (bvmul out1_a in0_a)))))
(define-fun gamma_b () Bool
    (= out0_b (bvmul (bvmul in0_b in0_b) in0_b)))
(define-fun gamma_in () Bool
    (= in0_a in0_b))
(define-fun gamma_out () Bool
    (= out2_a out0_b ))
(assert (not (=> (and gamma_in gamma_a gamma_b) gamma_out)))
(check-sat)
```

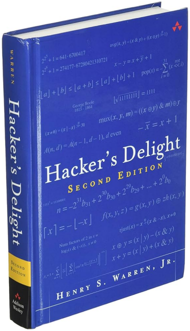
Verification: Uninterpreted Functions

```
(declare-const out0_a (_ BitVec 128))
(declare-const out1_a (_ BitVec 128))
(declare-const in0_a  (_ BitVec 128))
(declare-const out2_a (_ BitVec 128))
(declare-const out0_b (_ BitVec 128))
(declare-const in0_b  (_ BitVec 128))
(declare-fun f ((_ BitVec 128) (_ BitVec 128)) (_ BitVec 128))
(define-fun gamma_a () Bool
    (and (= out0_a in0_a)
        (and (= out1_a (f out0_a in0_a))
            (= out2_a (f out1_a in0_a)))))
(define-fun gamma_b () Bool
    (= out0_b (f (f in0_b in0_b) in0_b)))
(define-fun gamma_in () Bool
    (= in0_a in0_b))
(define-fun gamma_out () Bool
    (= out2_a out0_b ))
(assert (not (=> (and gamma_in gamma_a gamma_b) gamma_out)))
(check-sat)
```

Verification: Popcount

Popcount: count the number of 1's in a bitvector

```
int popCount32 (unsigned int x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
  return x; }
```



Verification: General Setup

```
(set-logic QF_BV)
(declare-const x (_ BitVec 32))

(define-fun fast ((x (_ BitVec 32))) (_ BitVec 32)
    ...

(define-fun slow ((x (_ BitVec 32))) (_ BitVec 32)
    ...

(assert (not (= (fast x) (slow x))))
(check-sat) ; expect UNSAT
(exit)
```

Verification: Specification

```
(define-fun slow ((x (_ BitVec 32))) (_ BitVec 32)
 (bvadd
   (ite (= #b1 ((_ extract  0  0) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract  1  1) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract  2  2) x)) #x00000001 #x00000000)
                ...
   (ite (= #b1 ((_ extract 30 30) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract 31 31) x)) #x00000001 #x00000000)))
```

Verification: Code conversion

```
int popCount32 (unsigned int x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
  return x; }

(define-fun line1 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvsub x (bvand (bvlshr x #x00000001) #x55555555)))

(define-fun line2 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvadd (bvand x #x33333333)
         (bvand (bvlshr x #x00000002) #x33333333)))

(define-fun line3 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvlshr (bvmul (bvand (bvadd (bvlshr x #x00000004)
        x) #x0f0f0f0f) #x01010101) #x00000018))

(define-fun fast ((x (_ BitVec 32))) (_ BitVec 32)
   (line3 (line2 (line1 x))))
```

Verification: Demo

```
#eval (do
 let out ← callZ3 popcount (verbose := true)
 :  IO Unit)
```

```
Solver replied:
   unsat
```