UNIVERSITÀ DI TRENTO
Department of Information Engineering and Computer Science

Automated Reasoning and Formal Verification

Laboratory 3

Gabriele Masina
gabriele.masina@unitn.it
https://github.com/masinag/arfv2025

Università di Trento

March 19, 2025

These slides are derived from those by Giuseppe Spallitta for FM lab 2021-2024.

Outline

SMT-LIB file: options

Options can enable some additional functionalities, such as:

Command	Description
`(set-option :produce-models true)`	Generation of models
`(set-option :produce-unsat-cores true)`	Extraction of UNSAT cores
`(set-option :produce-proofs true)`	Building UNSAT proof
`(set-logic <logic>)`	Set background logic

While solving the exercises we will see the most popular options and their effects.

SMT-LIB file: declaration

In this section we declare variables and functions used to describe the problem.

- Variable declaration: `(declare-const <name> <type>)`
- Types supported by SMT-LIB are:
  - `Bool`
  - `Int`
  - `Real`
  - `(_ BitVec <size>)`
  - `(Array <index-type> <elem-type>)`

SMT-LIB file: declaration — defining functions

- We can declare **uninterpreted** functions as:

```
(declare-fun <name> ([input types]) <type>)
```

E.g. `(declare-fun foo (Int Int) Int)`

SMT-LIB file: declaration — defining functions

- We can declare **uninterpreted** functions as:

  ```
  (declare-fun <name> ([input types]) <type>)
  ```

  E.g. `(declare-fun foo (Int Int) Int)`
- We can define **interpreted** functions as:

  ```
  (define-fun <name> ([input types]) <type> <func>)
  ```

  E.g. `(define-fun max ((x Int) (y Int)) Int (ite (> x y) x y))`

SMT-LIB file: declaration — defining functions

- We can declare **uninterpreted** functions as:

```
(declare-fun <name> ([input types]) <type>)
```

  E.g. `(declare-fun foo (Int Int) Int)`

- We can define **interpreted** functions as:

```
(define-fun <name> ([input types]) <type> <func>)
```

  E.g. `(define-fun max ((x Int) (y Int)) Int (ite (> x y) x y))`

SMT-LIB file: declaration — defining functions

- We can declare **uninterpreted** functions as:

  ```
  (declare-fun <name> ([input types]) <type>)
  ```

  E.g. `(declare-fun foo (Int Int) Int)`
- We can define **interpreted** functions as:

  ```
  (define-fun <name> ([input types]) <type> <func>)
  ```

  E.g. `(define-fun max ((x Int) (y Int)) Int (ite (> x y) x y))`

Shortcuts

- `(declare-const <name> cd<type>)` is a shortcut for `(declare-fun <name> () <type>)`
- `(define-const <name> <type> <value>)` defines an *interpreted constant*:
  E.g., `(define-const sqrt2 Real 1.4142)`

SMT-LIB file: assertions

- Once defined the variables, we have to determine the constraints that rule the satisfiability of the problem in the form of assertions.
- The declaration of assertions can be done in the following way:

```
(assert <condition>)
```

- Conditions can be basic (e.g., $x = 5$) or nested (e.g., $(x = 2 \vee x = 5) \wedge y = 3$).

Warning

In SMT-LIB operators always use a prefix notation!

SMT-LIB assertions: Propositional Logic

Boolean operators are available to use:

Logical Operation	**Representation**
Negation	`(not <formula>)`
OR	`(or  <formula1> <formula2>)`
AND	`(and <formula1> <formula2>)`
Implies	`(=>  <formula1> <formula2>)`
XOR	`(xor <formula1> <formula2>)`
IFF (if and only if)	`(=   <formula1> <formula2>)`

Warning

The operators `and`, `or` can be used with *two or more* arguments.

SMT-LIB assertions: Equality

- The equality operator is defined as:

```
(= <var1> <var2>)
```

- The inequality operator is defined as:

```
(not (= <var1> <var2>))
```

The equality operator can be used with variables of any type.

SMT-LIB assertions: Arithmetic

SMT-LIB standardizes syntax for arithmetic over integers (`Int`) and reals (`Real`).

Operation	**Representation**	
Addition	`(+ <var1> <var2> ...)`	
Subtraction	`(- <var1> <var2>)`	
Multiplication	`(* <var1> <var2> ...)`	
Real division	`(/ <var1> <var2>)`	
Integer division	`(div <var1> <var2>)`	
Remainder (only `Int`)	`(mod <var1> <var2>)`	
Greater than (or equal)	`(> <var1> <var2>)`	`(>= <var1> <var2>)`
Less than (or equal)	`(< <var1> <var2>)`	`(<= <var1> <var2>)`

Warning

The operators `*`, `+` can be used with two *or more* arguments.

SMT-LIB assertion: Bit Vectors

Bit-Vectors are fixed-size sequences of bits representing numbers: `(_ BitVec <size>)`

Operation	Representation	
Addition	`(bvadd  <var1> <var2>)`	
Subtraction	`(bvsub  <var1> <var2>)`	
Multiplication	`(bvmul  <var1> <var2>)`	
Division	`(bvudiv <var1> <var2>)`	
Remainder	`(bvurem <var1> <var2>)`	
Greater than (or equal)	`(bvugt  <var1> <var2>)`	`(bvuge <var1> <var2>)`
Less than (or equal)	`(bvult  <var1> <var2>)`	`(bvule <var1> <var2>)`

Warning

The `u` in the operators stands for **unsigned**. By changing it into an `s`, you get the corresponding **signed** operators. (Beware that the range of admitted values changes!)

SMT-LIB assertion: Arrays

Arrays map an index type to an element type: `(Array <index-type> <elem-type>)` (similarly to Python `dict` type)

Operation	**Representation**
Selecting an element	`(select <array> <index>)`
Updating an element	`(store <array> <index> <value>)`

SMT-LIB file: Action

The bottom part of the file tells the solver the action to perform.

Action	**Description**
`(check-sat)`	Check the satisfiability of the problem.
`(get-model)`	If SAT, get the value for all the variables.
`(get-value (x y))`	If SAT, get the value for the variables `x` and `y`.
`(get-unsat-core)`	If UNSAT, get an unsatisfiable core.
`(get-proof)`	If UNSAT, get proof of unsatisfiability.
`(check-allsat ())`	Get all (T-SAT) truth assignments.
`(check-allsat (a b))`	Get all (T-SAT) truth assignments on atoms `a` and `b`.
`(exit)`	End the file.

Encoding step-by-step

The procedure to encode an SMT problem is identical to what we have done for SAT:

- Identify the variables that can describe the problem.
- Define the assertions to constrain the models and check the satisfiability.

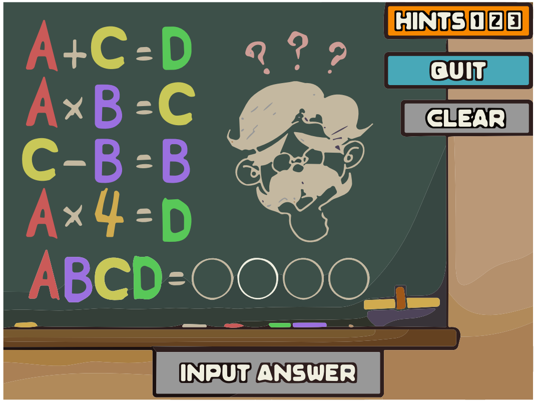
The major difference is the expressive power of SMT with respect to standard SAT.

Outline

First encodings

Exercise 3.1: Guess the Code

A, B, C, and D are single-digit numbers. Solve the following equations :



$$A + C = D$$
$$A \times B = C$$
$$C - B = B$$
$$A \times 4 = D$$
$$ABCD = \bigcirc\bigcirc\bigcirc\bigcirc$$

First encodings: variables

- We require 4 constants: A, B, C, and D
- Since they are single-digit numbers, we set them as `Int`.

First encodings: assertions

- We must encode the 4 equations that are written on the blackboard, using the basic arithmetical operators.
- Moreover, we must ensure that all the digits are different: we can use the command

  ```
  (assert (distinct A B C D))
  ```

  to easily encode it. If you don't remember it during the exam don't worry, you can encode it by hand...

First encodings: output

- Once we add the final action, we can feed it to the SMT solver.
  $\Rightarrow$ The solver returns SAT
- If we want to know the values of the variables, we have to add some options and some additional actions.

Outline

Solving Geometric Problems

Exercise 3.2: intersecting lines

Given two points in the Euclidean space $A = (1, \frac{3}{2})$ and $B = (\frac{1}{2}, 7)$, find:

- the equation of the line passing through them,
- the values x_i and y_i where the line intersects the x and y axes, respectively.

Solving geometric problems: Variables

- We can set 4 variables of type `Real` to store the coordinates of each point: (xa, ya) and (xb, yb).
- We need also to define a function variable (we will call it f) with arity 1 so that we can have an analytical representation of the line.
- A line is represented by the formula:

$$f(x) = mx + q$$

  Thus we need other two variables.
  Notice that f is an **interpreted function** (we know its behavior).

Solving geometric problems: Assertions

- We start defining 4 assertions to set the value of the coordinates and one assertion to define the line equation:

  ```
  (define-fun f ((x Real)) Real (+ (* m x) q))
  ```

- Let's force the line to pass through the two points:

  ```
  (assert (= (f xa) ya))
  (assert (= (f xb) yb))
  ```

  m and q will be calculated by the solver.

- Let's find the intersection with the axes:

  ```
  (assert (= (f 0) yi))
  (assert (= (f xi) 0))
  ```

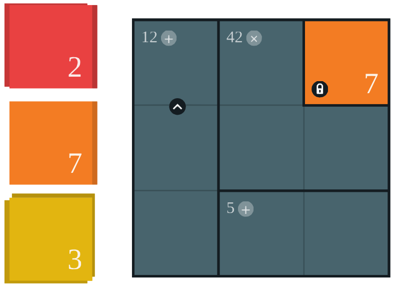
Solving geometric problems: results

- Now we can run MathSAT, obtaining a valid solution.
- The problem can be easily adapted to different sets of points: if we change the coordinates, we will obtain a different line.
- You can also extend this code to generalize this exercise in case you want to determine the intersection of two lines.

Mistery Grid

Exercise 3.3: Mistery Grid

To solve a MysteryGrid puzzle, place all of the tiles on the grid so that:



- Every row and every column contains exactly one of each type of tile.
- **Cages** (regions surrounded by a heavy border) have a target and an operation. The tiles you place in a cage must make the target number using the operation.
- Inequality clues between cells must be respected by the tiles you place.
- Tiles with a lock are already placed on the grid and cannot be moved.

Mistery Grid: variables

- We need to define 9 variables, labelled `x_ij`, where i is the row and j is the column.
- Each variable is of type `Int` and can assume one of the tiles values $\{2, 7, 3\}$.

Mistery Grid: assertions

- We must place exactly one tile in each cell of the grid.

  ```
  (assert (or (= x_ij red) (= x_ij orange) (= x_ij yellow)))
  ```

  Notice: that the “at most” condition is implicit in the fact that a variable can assume only one value.
- For each row and each column, we must ensure that all the tiles are different. We can use the `distinct` command.
- We must respect the constraints given by the cages.
- We must respect the constraints given by the inequalities.
- We must respect the constraints given by the locks.

Mistery Grid: output

- Once we add the final action, we can feed it to the SMT solver.
  $\Rightarrow$ The solver returns SAT
- Let's check the model to see if it is correct.

Cracking the Code

Exercise 3.3: Cracking the Code

You are a hacker trying to break into a high-security system. You have to crack a password that is a 5-digit number. You know, from an anonymous source, that:

- The 1st and last digits differ, as do the 2nd and 3rd.
- The 2nd digit is twice the 1st, and the 4th is one less than the last.
- No digit appears more than twice (e.g., 12322 is invalid).
- The password cannot be sorted (e.g., 12279, 84321 are invalid).
- The 1st and last digits are odd; the others are even.
- The digits' sum equals the 4th digit plus twice the 3rd.

Crack the password using an SMT solver.

Since you have only one guess you must be extremely careful. Is it the only solution?

Cracking the Code: variables

- We need to define 5 `Int` variables, labelled `a,b,c,d,e` for each digit.
- We must ensure to assign only digits from 0 to 9:

```
(assert (and (>= a 0) (<= a 9)))
```

Cracking the Code: assertions

- The first two constraints can be encoded straightforwardly.
- The condition that no digit appears more than twice is more complex:

```
(assert (=> (= a b) (and (not (= a c)) (not (= a d)) (not (= a e)))))
(assert (=> (= a c) (and (not (= a b)) (not (= a d)) (not (= a e)))))
...
```

- To ensure that the password cannot be sorted (both ascending and descending):

```
(assert (not (and (<= a b) (<= b c) (<= c d) (<= d e))))
(assert (not (and (>= a b) (>= b c) (>= c d) (>= d e))))
```

- Finally the last two conditions are easy to encode. Remember that we can exploit the `mod` operator to check if a number is odd or even.

Cracking the Code: output

- Once we add the final action, we can feed it to the SMT solver.
  $\Rightarrow$ The solver returns the model

  ```
  ( (a 1) (b 2) (c 6) (d 2) (e 3) )
  ```

- We can block the model with

  ```
  (assert (not (and (= a 1) (= b 2) (= c 6) (= d 2) (= e 3))))
  ```

  and check if the solver returns SAT again.
  $\Rightarrow$ The solver returns the model

  ```
  ( (a 1) (b 2) (c 8) (d 4) (e 5) )
  ```

Outline

Working as a receptionist

Consider the problem of previous lab:

Exercise 2.4: receptionist

You are a receptionist in a prestigious hotel and you are waiting for 5 new guests. There are 5 available rooms, but you don't know their preferences about the room they want to book until the last moment:

- Guest A would like to choose room 1 or 2.
- Guest B would like to choose a room with an even number.
- Guest C would like the first room.
- Guest D has the same behavior as user B.
- Guest E would like one of the external rooms.

Supposing the guests come one after the other, is there a moment where it is not possible to help every guest? How many guests can be sorted without problems?

Incremental SAT solving

We can use *push* and *pop* commands to save and restore the state of the solver.
This us to incrementally add new constraints, and check the satisfiability at each step.

SMT-LIB

```
; General constraints
(assert ...)
; Guest preferences
(push 1)
(assert (or A1 A2)) ; guestA
(check-sat)
(push 1)
(assert (or B2 B4)) ; guestB
(check-sat)
...
```

PySMT

```
# General constraints
solver.add_assertions(assertions)
# Guest preferences
solver.push()
solver.add_assertion(Or(A1, A2))
solver.solve()
solver.push()
solver.add_assertion(Or(B2, B4))
solver.solve()
...
```

UNSAT cores

An *unsatisfiable core* is a subset of the original set of constraints that is unsatisfiable.

SMT-LIB

```
(set-option :produce-unsat-cores
                                  true)
; General constraints
(assert ...)
; Guest preferences
(assert(!(or A1 A2) :named guestA))
(assert(!(or B2 B4) :named guestB))
...
(check-sat)
(get-unsat-core)
```

PySMT

```
s = Solver(name="msat",
    unsat_cores_mode="named")
# General constraints
s.add_assertions(assertions)
# Guest preferences
s.add_assertion(...,named="guestA")
s.add_assertion(...,named="guestB")
...
if not s.solve():
    print(s.get_named_unsat_core())
```

All-SAT

Sometimes we want to know all the possible solutions to a problem.
$\Rightarrow$ All-SAT is is the problem of finding all satisfying assignments to a given formula.
PySMT does not support All-SAT natively, but we can use the MathSAT API.

SMT-LIB

```
; General constraints
(assert ...)
; Guest preferences
(assert (or A1 A2))
(assert (or B2 B4))
(check-allsat ())
```

PySMT

```
import mathsat
def callback(model, converter):
    print([converter.back(v) for v in model])
    return 1 # continue, 0 to stop
solver = Solver(name="msat")
converter = solver.converter
solver.add_assertions(assertions)
mathsat.msat_all_sat(solver.msat_env(),
    [converter.convert(v) for v in vv],
    lambda model : callback(model, converter))
```

Outline

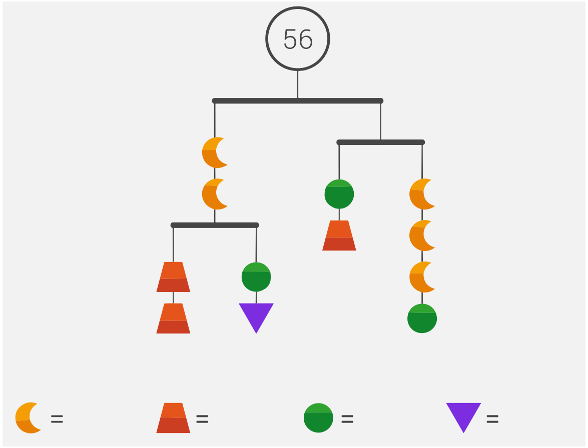
Homework

Homework 3.1: Math Olympics

Find 3 digits a, b, c, not necessarily distinct, with a $\neq$ 0 and c $\neq$ 0 such that both abc and cba are multiples of 4.

Homework

Homework 3.2: Balance Puzzle



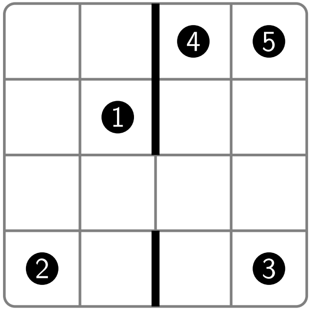
Solve it using an SMT solver (hint: use temporary variables to store intermediate results)

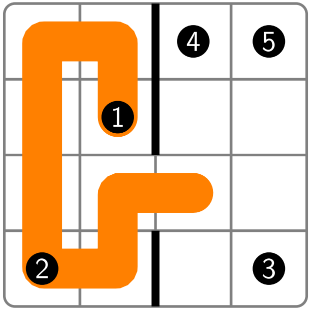
Questions:

1. What type should the variables have?
2. What happens if instead we impose the sum to be equal to 58?

Homework

Homework 3.3: Zip Puzzle





You have found a new puzzle to solve:

1. Draw a line from the first dot to the last, moving only horizontally or vertically.
2. Connect the dots in order
3. Fill every cell
4. You cannot cross the walls (the bold lines)

Use an SMT solver to solve it.