Undergraduate Project-I Report

Rachita Chhaparia
Guide: Dr. Sandeep Shukla

November 2015

1 Goal

The goal of the project was to learn about theoretical and practical aspects of SMT Solvers like z3 as well as to understand and implement some of their applications.

2 SMT Solvers

2.1 Introduction

Satisfiability is a decision problem of determining if there exists a model for a formula which makes it true.
Boolean Satisfiability (SAT) problem, decided by a SAT solver, is the problem of checking the satisfiability of a boolean formula.
Satisfiabilty Modulo Theories (SMT) problem [1], performed by an SMT solver, is the problem of checking the satisfiability of a first order logic formula with respect to a combination of background theories such as, arithmetic, arrays, bit-vectors and uninterpreted functions.
An SMT instance is hence a generalization of a SAT instance, where the Boolean variables are replaced by predicates belonging to the theories mentioned above. It helps in creating verification systems which can reason at a higher level of abstraction. This is more useful for systems used today as they are also modeled at a higher level than the Boolean level.
Example of a SAT instance solved by a SAT solver:

$$(p \vee q) \wedge (\neg p \vee \neg q) \tag{1}$$

Satisfiable, Model = $[p = TRUE, q = FALSE]$
Example of an SMT instance solved by an SMT solver (p, q declared as integers):

$$(p \geq 0) \wedge (p + q \leq 5) \tag{2}$$

Satisfiable, Model = $[p = 0, q = 5]$

2.2 Theories

Examples of built-in theories of various SMT solvers:

Theory	Signature	Domain	Satisfiability
Theory $T_{\in}$ of Equality	Empty	-	undecidable(general), decidable(conjunction of literals)
Theory T_Z of Integers	$\sum_Z^x : (0, 1, +, -, \leq)$	Z	decidable
Theory T_Z of Integers	$\sum_Z^x : (0, 1, +, -, \leq, X)$	Z	undecidable
Theory T_R of Reals	$\sum_R^x : (0, 1, +, -, \leq)$	R	decidable
Theory T_R of Reals	$\sum_R^x : (0, 1, +, -, \leq, X)$	R	decidable

Other theories supported are Theory of Arrays, Theory of Inductive Datatypes (eg, lists), Theory of Bitvectors.
Theories can also be combined to produce an SMT instance of the following kind: ($x \in Z$)

$$(x \geq 0) \wedge (f(x) == 0) \tag{3}$$

3 Z3

Z3[2] is an SMT solver developed at Microsoft Research, which is mainly used for software/hardware testing and verification, constraint solving, security, analysis of hybrid systems, etc.
Z3 can support the theories of uninterpreted functions, linear arithmetic (integer and real), arrays, bit-vectors, algebraic data-types (eg. lists, trees etc.), polynomial arithmetic (fails in some cases), division. It can also solve some problems involving quantifiers, but, cannot prove inductive facts.

3.1 Some Applications performed

I implemented the following applications of Z3 using the pyhton API of Z3, Z3Py , after going through the tutorials available on: `http://cpl0.net/~argp/papers/z3py-guide.pdf`

3.1.1 Constraint Solving

Z3 can be used to model and solve some constraint satisfaction problems like Sudoku, eight queens problem. The basic idea is to model the constraints in first order logic using the required theories. The Sudoku problem and the eight queens problem were solved in the following manner: (using Z3Py)

```
#Solves the sudoku example input1 where x_i_j  represents the (i,j)
    th entry in the sudoku
from z3 import *
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ] for i in
    range(9) ]
cells = [ And(1 <= X[i][j], X[i][j] <= 9) for i in range(9) for j
    in range(9) ]
col = [ Distinct([ X[i][j] for i in range(9) ]) for j in range(9) ]
```

```
row= [ Distinct([X[i][j] for j in range(9) ] ) for i in range(9) ]
grid = [ Distinct([ X[3*i1 + i][3*j1 + j] for i in range(3) for j
    in range(3) ]) for i1 in range(3) for j1 in range(3) ]
input1 = ((0,0,0,0,9,4,0,3,0), #an input instance
(0,0,0,5,1,0,0,0,7),
(0,8,9,0,0,0,0,4,0),
(0,0,0,0,0,0,2,0,8),
(0,6,0,2,0,1,0,5,0),
(1,0,2,0,0,0,0,0,0),
(0,7,0,0,0,0,5,2,0),
(9,0,0,0,6,5,0,0,0),
(0,4,0,9,7,0,0,0,0))
sudoku = cells+ row + col + grid
check = [ If(input1[i][j]==0, True, X[i][j]==input1[i][j]) for i in
    range(9) for j in range(9)]
solve(check+sudoku)
```

Listing 1: Sudoku solver

Output:

```
x_1_1=7, x_1_2=1, x_1_3=5, x_1_4=8, x_1_5=9, x_1_6=4, x_1_7=6, x_1_8=3, x_1_9=2,
x_2_1=2, x_2_2=3, x_2_3=4, x_2_4=5, x_2_5=1, x_2_6=6, x_2_7=8, x_2_8=9, x_2_9=7,
x_3_1=6, x_3_2=8, x_3_3=9, x_3_4=7, x_3_5=2, x_3_6=3, x_3_7=1, x_3_8=4, x_3_9=5,
x_4_1=4, x_4_2=9, x_4_3=3, x_4_4=6, x_4_5=5, x_4_6=7, x_4_7=2, x_4_8=1, x_4_9=8,
x_5_1=8, x_5_2=6, x_5_3=7, x_5_4=2, x_5_5=3, x_5_6=1, x_5_7=9, x_5_8=5, x_5_9=4,
x_6_1=1, x_6_2=5, x_6_3=2, x_6_4=4, x_6_5=8, x_6_6=9, x_6_7=7, x_6_8=6, x_6_9=3,
x_7_1=3, x_7_2=7, x_7_3=6, x_7_4=1, x_7_5=4, x_7_6=8, x_7_7=5, x_7_8=2, x_7_9=9,
x_8_1=9, x_8_2=2, x_8_3=8, x_8_4=3, x_8_5=6, x_8_6=5, x_8_7=4, x_8_8=7, x_8_9=1,
x_9_1=5, x_9_2=4, x_9_3=1, x_9_4=9, x_9_5=7, x_9_6=2, x_9_7=3, x_9_8=8, x_9_9=6,
```

The constraints modeled in the Sudoku problem ensure that there are distinct numbers in the range[1,9] across all columns, rows, grids.

```
#Provides a solution for the 8 queens problem s.t. Q_i_0 represents
    the row number ranging from
#1-8 and Q_i_1 represents the col number ranging from 1-8, of the
    ith queen
from z3 import *
Q=[ [Int("Q_%s_%s" %(i, j)) for j in range(2) ] for i in range(8) ]

cells=[ And(Q[i][j]>=1, Q[i][j]<=8 ) for i in range(8) for j in
    range(2) ]
diag = [ If(i==i1, True, And(Q[i][0]-Q[i1][0]!=Q[i][1]-Q[i1][1], Q[
    i][0]-Q[i1][0]!=Q[i1][1]-Q[i][1]) )for i in range(8) for i1 in
    range(8) ]
row_col=[ Distinct([Q[i][j] for i in range(8)] ) for j in range(2)
    ]
queen = row_col+cells+diag
solve(queen)
```

Listing 2: Eight Queens Problem

```
Output:

x_1_0=5, x_1_1=1,
x_2_0=3, x_2_1=4,
x_3_0=7, x_3_1=2,
x_4_0=2, x_4_1=8,
x_5_0=4, x_5_1=7,
x_6_0=1, x_6_1=3,
x_7_0=6, x_7_1=6,
x_8_0=0, x_8_1=0,
```

The constraints modeled in the Queen problem are that no two queens lie in the same row, column or diagonal.

3.1.2 Program Verification

Z3 is also used for program verification in various ways. Here, I will use it for static verification of some programs. The program is first converted to symbolic form and tested for validity i.e. the negation of the symbolic form of the program is proved unsatisfiable by Z3.
The process of converting a program to symbolic form differs depending on the program but, in general reduces to finding some invariants in the program. Factorial and bubblesort programs were verified in the following manner:

```
from z3 import *
def factorial( n): #the verifier
  fact = 1
  for j in (1,n):
      fact = fact*j
  return fact;

i = Function('i',IntSort(), IntSort())
fact = Function('fact',IntSort(), IntSort())
k = Int('k')
n = Int('n')
s = Solver()
s.add(And(k>1,ForAll(k, fact(k)==fact(k-1)*i(k)))) #invariants
s.add(And(k>1,ForAll(k, i(k)==i(k-1)+1)))
s.add(fact(1)==1) #initial conditions
s.add(i(1)==1)
s.add(factorial(n)!=fact(n)) #negation of program
print s.check()
```

Listing 3: Verification of Factorial Program

```
Output:
unsat
```

The invariants in the Factorial program were identified in terms of the recurrence relations $fact(k) == fact(k-1)*i(k)$ and $i(k) == i(k-1)+1$. Initial conditions

$fact(1) == 1$ and $i(1) == 1$ are also established. The negation of the program $factorial(n)! = fact(n)$ is then tested for satisfiability. Since, the output is unsat, the program is valid i.e. $\forall n, factorial(n) = fact(n)$.

```
from z3 import *

a = Array('a', IntSort(), IntSort())
i = Int('i')
s = Solver()
k = Int('k')
k_ = Int('k')
i = Int('i')
n = Int('n')
bsort = Function('bsort',a.sort(),IntSort(),IntSort(),IntSort(),a.
    sort())
opt1 = Function('opt1',a.sort(),IntSort(),IntSort(),IntSort(),a.
    sort())
opt3 = Function('opt3',a.sort(),IntSort(),IntSort(),IntSort(),a.
    sort())
opt2 = Function('opt2',a.sort(),IntSort(),IntSort(),IntSort(),a.
    sort())
j = Int('j')
issorted = Function('issorted', a.sort(),IntSort(),BoolSort())
s.add(And(k<=1),ForAll(k,issorted(a,k)==True)) #invarinats
s.add(And(k>1),ForAll(k,issorted(a,k)==And(issorted(a,k-1),Select(a
    ,k)>=Select(a,k-1))))
s.add(And(i==0,ForAll([a,n,i,j],bsort(a,n,i,j)==a)))
s.add(And(j==i,ForAll([a,n,i,j],bsort(a,n,i,j)==bsort(a,n,i-1,n))))
s.add(And(i>=1,And(i<=n,And(j!=i,And(j<=n,And(Select(a,j)<Select(a,
    j-1)))))),ForAll([a,n,i,j],bsort(a,n,i,j)==opt3(  opt2(opt1(
    bsort(a,n,i,j-1),n,i,j),n,i,j),n,i,j),n,i,j)))
s.add(ForAll([a,n,i,j], opt1(a,n,i,j)== Store(a,Select(a,j),n+1)))
s.add(ForAll([a,n,i,j], opt2(a,n,i,j) == Store(a,Select(a,j-1),j)))
s.add(ForAll([a,n,i,j], opt3(a,n,i,j) == Store(a,Select(a,n+1),j-1)
    ))
s.add(issorted(bsort(a,n,n,n),n)==False) #negation of program
print s.check()
```

Listing 4: Verification of Bubblesort Program

```
Output:
unsat
```

The program of Bubblesort is converted to symbolic form. The *issorted* function ensures that the array remains sorted and the *bsort* function performs bubblesort on the array. The negation of the program in the form of the array remaining un-sorted after bubblesort, is returned unsat, thereby, establishing the validity of the Bubblesort program i.e. arrays of all lengths are sorted after undergoing bubblesort.

3.1.3 Program Testing and Analysis

Z3 is widely used in various ways for verifying and testing software to identify potential bugs or pieces of dead code. A general approach is to feed generated

test inputs to the program and analysing their behaviour on the same.
Random Testing involves generating random inputs and testing the program on them. This often leads to redundant checks and low code-coverage which may miss several possible potential bugs in the program.
Symbolic Execution, on the other hand, tests the program on symbolic inputs (not concrete) and generates path constraints. Whenever a conditional statement is executed, the path constraints get updated and are hence, encoded on the input to reach a particular program point. These constraints are then solved on a constraint solver for test generation. However, the constraints often fall in undecidable theories and do not get solved.
Another method, called **Concolic Testing** [3], addresses the challenges mentioned above by combining random testing and classical symbolic execution. It starts program execution with some random concrete input, concurrently running symbolic testing and collects symbolic path constraints upon encountering a conditional statement. These constraints are then solved by a constraint solver like Z3 to generate a concrete input to execute an alternate program path. This process is repeated until all program paths have been traversed (or depending on various heuristics).
Following are some programs on which Concolic Testing is performed:

```
1
2  void function( int x, int y){
3    z = 2x + 3y;
4    if(x > 5){
5      ..
6      if(z < 0){
7        ..
8      }
9
10     else if ( y!=3){
11       ..
12     }
13   }
14
15   else {
16     //error
17   }
18 }
```

Listing 5: Concolic Testing I: Error Detection

Analysis:

1. Take random concrete inputs: [x = 6, y = -3] and symbolic inputs (x_o, y_o)
2. Execute function(6,-3)
3. No error received, collect path constraints $x_o > 5, 2x_o + 3y_o \geq 0$
4. Use Z3 to solve $x_o > 5, 2x_o + 3y_o \geq 0$, Model: [x = 6, y = -4]
5. Execute function(6,-4)
6. No error received, collect path constraint $x_o \leq 5$,
7. Use Z3 to solve $x_o \leq 5$, Model: [x = 5]

8. Execute function(5,-4)
9. Error found!

```
1
2  void function( int x, int y){
3    z = 2x + 3y;
4    if(x > 5){
5      ..
6      if(z != 0){
7        ..
8      }
9
10     else if ( y<-7){
11       ..
12     }
13   }
14
15   else {
16     ..
17   }
18 }
```

Listing 6: Concolic Testing II: Dead Code Detection

Analysis:

1. Take random concrete inputs: [x = 3, y = -3] and symbolic inputs (x_o, y_o)
2. Execute function(3,-3)
3. No error received, collect path constraints $x_o > 5$
4. Use Z3 to solve $x_o > 5$, Model: [x = 6, y = -3]
5. Execute function(6,-3)
6. No error received, collect path constraint $x_o > 5, 2 * x_o + 3 * y_o == 0$,
7. Use Z3 to solve $x_o > 5, 2 * x_o + 3 * y_o == 0$, Model: [x = 6, y = -4]
8. Execute function(6,-4)
9. Collect path constraints $x_o > 5, 2 * x_o + 3 * y_o == 0, y_o < -7$ and use Z3 to solve, Model: no solution
10. Dead Code detected!

The benefit of Concolic testing over classical symbolic testing is that if the constraints are not solvable by a solver, then it can be simplified by replacing some of the symbolic expressions with concrete values.

```
1
2  void function( int x, int y , float z){
3    int v = x + y;
4    if(v > 5){
5      ..
6      if(z < 0){
7        ..
8      }
9
10     else if (x*x*x*y*z > 99){
11       //error
12     }
13   }
14
```

```
15    else {
16      ..
17    }
18  }
```

Listing 7: Concolic Testing III

Analysis

1. Take random concrete inputs: [x = 3, y = 3, z = -1.0] and symbolic inputs (x_o, y_o, z_o)
2. Execute function(3,3,-1)
3. No error received, collect path constraints $x_o + y_o > 5, z_o > 0$
4. Use Z3 to solve $x_o + y_o > 5, z_o > 0$, Model: [x = 6, y = 0, z = 1]
5. Execute function(6,0,1)
6. No error received, collect path constraint $x_o > 5, z_o > 0, x_o * x_o * x_o * y_o * z_o > 99$
7. Use Z3 to solve $x_o > 5, z_o > 0, x_o * x_o * x_o * y_o * z_o > 99$ Output: failed to solve
8. Replace $x * x * x * y * z = 100$ and execute function(6,1,1) taking the path in line 10.
9. Error detected!

3.1.4 Deadlock Detection for Polychronous Data-Flow Specifications

Polychronous data-flow specifications specify concurrent, multi-clocked models which operate relatively asynchronous to each other. The term 'clock' of a variable here refers to the set of instants at which the variable is present. Before the synthesis of executable code from Polychronous specifications, it needs to be ensured that it is deadlock free. Signal, a polychronous data-flow language allows the specification of multi-clocked specifications. However, the Signal compiler rejects apparent causal loops which do not cause deadlocks, thereby, rejecting many valid specifications. An approach, as described by Ngo et. al. [4] using dependency graphs and first order logic formulas reasoned by an SMT Solver, gives a more precise method to check for deadlock freedom.
The data dependencies among the signals in a program are represented in the form of dependency graph called Synchronous Data-flow Dependency Graph (SDDG), where each node is a signal or a clock and each edge is labeled by a clock constraint. Cycles in this graph are identified and checked if they represent a deadlock. A cycle stands for a deadlock if the product of clocks representing the edges involved is not always null. Hence, the product is fed to an SMT solver and checked for satisfiability. If it returns 'unsat', then the cycle does not stand for a deadlock. This explained through an example below:

```
input: integer x, integer c

y := x when c>=0
x := y when c>=0
```

A cycle in its SDDG is: $\hat{x} \wedge \hat{y}$, where Here b represents $c \geq 0$.
$\hat{y} = \hat{x} \wedge (\hat{x} \vee (\hat{x} \wedge \hat{c} \wedge \tilde{c} \in [0, \infty)))$ This when fed to Z3, returns sat, i.e. it is not null when signal x is present. Hence, this represents a deadlock.
The goal was to implement this deadlock detection method as part of SIGNAL toolbox. But, I coudn't start it and hence, for now, that remains as future work.

References

[1] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[2] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[3] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[4] Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier. "Precise deadlock detection for polychronous data-flow specifications". In: *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*. IEEE. 2014, pp. 1–6.