

# Satisfiability Modulo Theory (SMT)

Dr. Avinash Malik

September 12, 2022

## 1 Why *Satisfiability Modulo Theory* (SMT)

- Explicit model checking suffers from state space explosion
- Explicit model checking not very well suited for proof of equivalence between different programs, e.g, specification and optimised programs.
- SMT is well suited for any verification and constraint problems.
  - Solving sudoku and other games.
  - Used in package managers for dependence analysis
  - Used for checking program security see here
  - Used for verifying device drivers see here
- SMT good for any optimisation problem, see here

## 2 Our very first problem

- Consider the linear system

$$x + y \geq$$

$$x - y \geq$$

(1)

- What are *some* integer values of  $x$  and  $y$  that satisfy the
- Multiple ways of doing this, e.g., Gaussian elimination, by substituting, etc.
- I will show manual solving:

$$x + y \geq 10$$

$$\therefore x \geq 10 - y$$

$$\therefore 10 - y - y \geq 20$$

$$\therefore -2y \geq 10 \implies y \leq -5$$

$$\therefore x \geq 10 - (-5) \implies x \geq 15 \quad (2)$$

- We use SMT solver called Z3 from Microsoft as shown in Listing 1

```
1 #!/usr/bin/env python3
2
3 # Using the z3 SMT solver with python3 bindings
4 from z3 import IntSort, Solver, sat, And, Consts
5 # Importing the datatype Int, the Solver class, and the 'sat' variable.
6 # Finally, the And (logical And class)
7
8 def main():
9     # Declaring and defining the two variables.
```

```

10     x, y = Consts('x y', IntSort()) # IntSort just stands for Int (type)
11
12     s = Solver()                     # Initialising the solver
13
14     # Adding the equations into the solver
15     s.add(And(x + y >= 10, x - y >= 20))
16
17     # Show the state of the solver
18     print('Solver state: %s' % s)    # Just for debugging
19
20     # Solving for all free variables: x and y
21     ret = s.check()
22
23     # Check if there is some assignment for x and y
24     # that satisfy the equations
25     if ret == sat:
26         print('Result:')
27         print('x: %s' % s.model()[x])
28         print('y: %s' % s.model()[y])
29
30
31 # Calling the main function in python
32 if __name__ == '__main__':
33     main()

```

Figure 1: Example 1

```

1 Solver state: [And(x + y >= 10, x - y >= 20)]
2 Result:
3 x: 15
4 y: -5

```

Figure 2: Results for Ex

- See Z3 documentation
- See Z3 API

### 3 Hardware circuit equivalence

3.1 Consider a 1-bit adder circuit described by a designer.

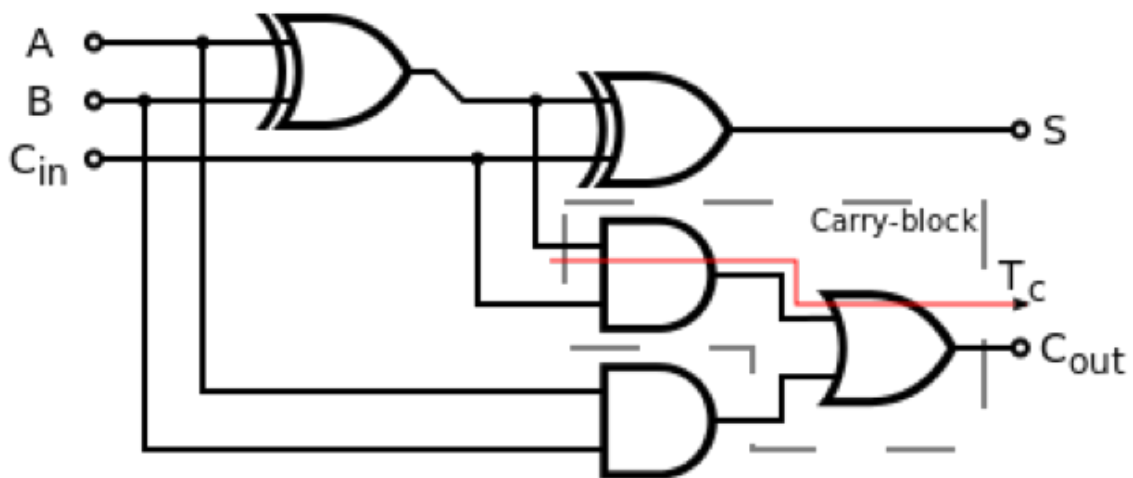
## Assignment Project Exam Help

<https://eduassistpro.github.io/>

- We have three inputs  $C_{in}$ , A, and B.
- We have two outputs, Sum and Carry.
- There are a number of gates that compute the Sum and the Carry.

Add WeChat edu\_assist\_pro

3.2 Consider another 1-bit adder



- Again we have the same number of inputs and outputs.

- The second implementation is a whole lot better than the first one.
- The second implementation has fewer number of gates and hence is *optimised*.

### 3.3 We want to prove that the two implementations are equivalent

- The first implementation can be encoded in binary logic as shown in Equations 3 and 4.

$$\begin{aligned} Sumf &\Leftrightarrow \\ &((\neg A) \wedge (\neg B) \wedge C) \vee \\ &((\neg A) \wedge B \wedge (\neg C)) \vee \\ &(A \wedge (\neg B) \wedge (\neg C)) \vee \\ &(A \wedge B \wedge C) \end{aligned} \quad (3)$$

$$Carryf \Leftrightarrow (B \wedge C) \vee (A \wedge C) \vee (A \wedge B) \quad (4)$$

- The logical operators mean the following:
  1.  $\Leftrightarrow$  is logical equivalence.
  2.  $\neg$  is logical negation (or not).
  3.  $\wedge$  is logical conjunction (or and).
  4.  $\vee$  is logical disjunction (or operator).
  5.  $\oplus$  is logical XOR.

- The logic can be encoded into Z3 solver as follows:

```

1 # Note the captialisation of Boolean operators.
2 from z3 import And, Not, Or, solver
3
4 def functional(Sumf, Carryf):
5     # Note <=> is converted into ==
6     s.add(Sumf == Or(And(Not(A), B, Not(C)),
7                         And(A, Not(B), Not(C)),
8                         And(A, B, C)))
9
10
11     s.add(Carryf == Or(And(B, C), And(A, C), And(A, B)))
12

```

- The second implementation can be encoded into binary logic as shown in Equation 5

$$\begin{aligned} u &\Leftrightarrow A \oplus B \\ v &\Leftrightarrow (u \wedge C) \\ w &\Leftrightarrow (A \wedge B) \\ Si &\Leftrightarrow u \oplus C \\ Ci &\Leftrightarrow (w \vee v) \end{aligned} \quad (5)$$

```

1 from z3 import And, Not, Xor, BoolSort, Or, sat
2 from z3 import Solver, Consts
3 def implementation(Si, Ci, A, B, C, s):
4     u, v, w = Consts('u, v, w', BoolSort())
5     s.add(u == Xor(A, B))
6     s.add(v == And(u, C))
7     s.add(w == And(A, B))
8     s.add(Si == Xor(u, C))
9     s.add(Ci == Or(w, v))

```

- The two implementations are equivalent  $\Leftrightarrow$ , if and only if
  - For *all* inputs  $\text{Sumf} \Leftrightarrow \text{Si} \wedge \text{Carryf} \Leftrightarrow \text{Ci}$
  - Equivalently, there *exists no* input values for A, B, C, such that,  $(\text{Sumf} \oplus \text{Si}) \vee (\text{Carryf} \oplus \text{Ci})$  is **satisfied**.
  - The above is called a "mitre" circuit.
  - The SMT encoding is shown in Listing 3

```

1  #!/usr/bin/env python3
2
3  from z3 import And, Not, Xor, BoolSort, Or, sat
4  from z3 import Solver, Consts
5
6
7  def functional(sumf, carryf, a, b, c, s):
8      s.add(sumf == Or(And(Not(a), Not(b), c),
9                      And(Not(a), b, Not(c)),
10                     And(a, Not(b), Not(c)),
11                     And(a, b, c)))
12
13      s.add(carryf == Or(And(b, c), And(a, c), And(a, b)))
14
15
16  def implementation(Si, Ci, A, B, C, s):
17      u, v, w = Consts('u, v, w', BoolSort())
18      s.add(u == Xor(A, B))
19      s.add(v == And(u, C))
20      s.add(w == And(A, B))
21      s.add(Si == Xor(u, C))
22      s.add(Ci == Or(w, v))
23
24
25  def main():
26      A, B, Cin = Consts('A, B, Cin')
27      Sf, Cf = Consts('Sf, Cf', BoolSort())
28      s = Solver()
29      functional(Sf, Cf, A, B, Cin, s)
30      Si, Ci = Consts('Si, Ci', BoolSort())
31      implementation(Si, Ci, A, B, Cin, s)
32
33      # Now the "mitre" circuit
34      s.add(Or(Xor(Sf, Si), Xor(Cf, Ci)))
35
36      # Check if the circuits are equivalent
37      if s.check() == sat:
38          print('Circuits not equivalent')
39          print(s.model())          # print the values of A, B, C, etc.
40      else:
41          print('Circuits are equivalent')
42
43
44  if __name__ == '__main__':
45      main()

```

Figure 3: Hardware Equivalence Encoding

```

1  Circuits are equivalent

```

Figure 4: Results for Hardware Equivalence in Listing 3

## 4 Software program code equivalence

### 4.1 Equivalence of two C++ programs Example 1

- Consider the two pieces of C++ programs in Listing 5
- Function `power3_func` is written by the programmer
- Function `power3_impl` is the optimised code translated by the compiler
- We want to show that for all input values  $i$ , the output is the same for both programs
- Logically we want to show that  $\forall i \in \mathbb{Z}, outl == outr$
- Equivalently, we want to show that:  $\neg(\exists i \in \mathbb{Z}, outl \neq outr)$
- We want to do this at **compile** time, without running the program.
- This is called proving compiler correctness.

```
1  /* Written by the programmer */
2  int power3_func(int i){
3      int outl;
4      outl = i;
5      for (int ii = 0; ii < 2; ++ii) {
6          outl *= i;
7      }
8      return outl;
9  }
10 /* Optimised by the compiler */
11 int power3_impl(int i){
12     int outr = ((i * i) * i);
13     return outr;
14 }
```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Figure 5: Hand written and opti

Add WeChat edu\_assist\_pro

#### 4.1.1 Step-1 change programs into single static assignment (SSA) form

- Every variable is assigned *only* once
- Requires unrolling the loop
- One assignment for each loop iteration
- See Listing 6

```
1  /* Original form */
2  int power3_func(int i){
3      int outl;
4      outl = i;
5      for (int ii = 0; ii < 2; ++ii) {
6          outl *= i;
7      }
8      return outl;
9  }
10
11 /* SSA form */
12 int power3_func_ssa(int i){
13     int outl;
14     int o1 = i;
15     /* Loop unrolled */
16     int o2 = o1 * i;          /* iteration 1 */
17 }
```

```

17     outl = o2 * i;           /* iteration 2 */
18     return outl;
19 }
20

```

Figure 6: SSA representation of the program

#### 4.1.2 Step-2 model the SSA form of the program into logic

- We have the logic from `power3_func_ssa` program as shown in Equation 6

$$(o1 == i) \wedge (o2 == o1 * i) \wedge (outl == o2 * i) \quad (6)$$

- The logic from `power3_impl` is shown in Equation 7

$$(outr == ((i * i) * i)) \quad (7)$$

#### 4.1.3 Step-3 encoding the logical formulas in SMT (Z3)

- The SMT encoding is shown in Listing 7

```

1  #!/usr/bin/env python3
2  # The standard imports
3  from z3 import Solver, sat, And, Consts
4
5
6  def mul():
7      # Importing the type Int
8      from z3 import IntSort
9      s = Solver()
10     # Declaring the variables
11     i, o1, o2, outl, outr = Consts
12     # Encode outl
13     s.add(And(o1 == i, o2 == (o1 * i), outl == (o2 * i)))
14
15     # Encode outr
16     s.add(outr == (i * (i * i)))
17
18     # Encode the condition that there exists no i, such that outl !=
19     # outr
20     s.add(outl != outr)
21
22     if s.check() == sat:
23         print('Codes not equivalent, example:')
24         print(s.model())
25     else:
26         print('Codes are equivalent')
27
28
29 if __name__ == '__main__':
30     mul()
31

```

Figure 7: SMT encoding of the software program equivalence, Example 1

```

1  Codes are equivalent

```

Figure 8: Results for Listing 7

## 4.2 Equivalence of two C++ programs Example 2

- Similar to Example 1, see Listing 9
  - Using addition instead of multiplication
  - Using `float` type instead of `int` type
  - We want to prove that `add3_func` and `add3_impl` outputs are the same for every input `i`.

```
1  /* Written by the programmer */
2  float add3_func(float i){
3      float outl;
4      outl = i;
5      for (int ii = 0; ii < 2; ++ii) {
6          outl += i;
7      }
8      return outl;
9  }
10 /* Optimised by the compiler */
11 float add3_impl(float i){
12     float outr = ((i + i) + i);
13     return outr;
14 }
```

Figure 9: Hand written and optimised programs

### 4.2.1 Encoding the equivalence logical formulas in SMT (Z3)

- Same as before, with minor changes, see Listing 10

```
1  #!/usr/bin/env python3
2  from z3 import Solver, sat, And, C
3
4
5  def add():
6      # Importing the Real type, to simulate floats
7      from z3 import RealSort
8      s = Solver()
9
10     # Declaring variables as Reals
11     i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', RealSort())
12     # Make add3_func_ssa format
13     s.add(And(o1 == i, o2 == (o1 + i), outl == (o2 + i)))
14
15     # Make outr
16     s.add(outr == (i + (i + i)))
17
18
19     # Add the equivalence statement
20     s.add(outl != outr)
21     if s.check() == sat:
22         print('Codes not equivalent, example:')
23         print(s.model())
24     else:
25         print('Codes are equivalent')
26
27
28 if __name__ == '__main__':
29     add()
```

Figure 10: SMT encoding for software program equivalence, Example 2



1 Codes are equivalent

Figure 11: Results for Listing 10

- There are some challenges and questions:
  1. For *every* function `*`, `+`, etc, we need to encode it into SMT and prove its correctness.
  2. For every type `int`, `float`, `double`, etc we need to encode it into SMT and prove its correctness.
  3. Can we do better?
  4. Can we make a general statement about correctness of the above program *irrespective* of types and operators?

### 4.3 Generic software program equivalence

- Have a general type, `int`, `float`, `unsigned char`, etc.
- Have a general function, `*`, `+`, etc.

#### 4.3.1 Generalising the types via Sorts

- Consider the *generic* C++ program for `add3_func` and `add3_impl` in Listing 12

```
1 template <typename T>
2 T add3_func(T){
3     T outl;
4     outl = i;
5     for (int ii = 0; ii < 2; ++ii) {
6         outl += i;
7     }
8     return outl;
9 }
10 template <typename T>
11 /* Optimised by the compiler */
12 T add3_impl(T i){
13     T outr = ((i + i) + i);
14     return outr;
15 }
```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu\_assist\_pro

Figure 12: Generic types in C++

- The type signature of `add3_func` is: `T add3_func(T)`
- The type signature of `add3_impl` is: `T add3_impl(T)`
- The type signature of `operator+` is: `T operator+ (T, T)`
- The type signature of `operator+=` is: `T operator+= (T, T)`
- The type signature of `operator=` is: `T operator= (T, T)`
- The return type is `T`
- The inputs are also of type `T`
- SMT *steals* this idea of C++ templates to implement generic types
- All variables are now declared with type `T`
- The *partial* encoding is given in Listing 13 with type `T`

- We are missing the logic of the SMT program in Listing 13

```

1  #!/usr/bin/env python3
2  from z3 import Solver, sat, And, Consts
3
4
5  def general():
6      from z3 import DeclareSort
7
8      # Declare the new type T
9      T = DeclareSort('T')
10
11     s = Solver()
12
13     # Declare the variables of type T
14     i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)
15
16     # FIXME: Here we need to fill in the logic
17
18     # Same as before, checking if for some i, outl and outr are
19     # different.
20     s.add(outl != outr)
21     if s.check() == sat:
22         print('Codes not equivalent, example:')
23         print(s.model())
24     else:
25         print('Codes are equivalent')
26
27 if __name__ == '__main__':
28     general()
29
30
31

```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Figure 13: SMT encoding with gen

Add WeChat edu\_assist\_pro

#### 4.3.2 Generalising the operators via uninterpreted functions

- We want a generic function  $f$ , which captures **properties** of all  $*$ ,  $+$ , etc operators.
- We *simply* replace the operator with some **random** function "f"
- See the C++ code in Listing 14

```

1
2  template <typename T>
3  // We will not define the function f.
4  // We will let SMT define the function for us!
5  T f (T, T);
6
7  template <typename T>
8  T add3_func(T i){
9      T outl;
10     outl = i;
11     for (int ii = 0; ii < 2; ++ii) {
12         outl = f(outl, i);           // Replaced the operator with f
13     }
14     return outl;
15 }
16
17 template <typename T>

```

```

18 /* Optimised by the compiler */
19 T add3_impl(T i){
20     T outr = f(f(i, i), i);          // Replaced the operator with f
21     return outr;
22 }

```

Figure 14: Generic function and types in C++

- Just like defining a generic function in C++, we define a generic function  $f$  in SMT.
- Moreover, we replace all uses of  $+$ ,  $*$ , etc, with just  $f$  as we have done in the C++ code.
- This function  $f$  is called an *uninterpreted function* in SMT
- Function "f" has no definition, and hence, no semantics.
- The logic with the generic function  $f$  is shown in Equation 8

$$\begin{aligned}
 (o1 == i) \wedge (o2 == f(o1, i)) \wedge (outl == f(o2, i)) \\
 (outr == f(f(i, i), i))
 \end{aligned} \tag{8}$$

#### 4.3.3 Encoding the generic equivalence formula in SMT (Z3)

- The generic encoding in SMT is shown in Listing 15

```

1  #!/usr/bin/env python3
2  from z3 import Solver, sat, And, Consts
3
4
5  def general():
6      from z3 import Function, F
7
8      # Declaring the new type T
9      T = DeclareSort('T')          # A new Type "T"
10
11      s = Solver()
12
13      # Declaring the variables we need for type T
14      i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)
15
16      # Declaring the new function "f" of type signature: (T, T) -> T
17      f = Function('f', T, T, T)
18
19      # Make outl, replacing the operator with f everywhere
20      s.add(And(o1 == i, o2 == f(o1, i), outl == f(o2, i)))
21
22      # Make outr, replacing the operators with f everywhere
23      s.add(outr == f(i, f(i, i)))
24
25      s.add(outl != outr)
26
27      # Print what is in the solver
28      print('Solver state: %s' % s)
29      print('\n')
30
31      if s.check() == sat:
32          # Print the model if something is wrong
33          print('Codes not equivalent, example trace:')
34          print(s.model())
35      else:
36          # Else everything is A OK!

```

```

37         print('Codes are equivalent')
38
39
40 def main():
41     general()
42
43
44 if __name__ == '__main__':
45     main()
46

```

Figure 15: First generic encoding in SMT

```

1 Solver state: [And(o1 == i, o2 == f(o1, i), outl == f(o2, i)),
2   outr == f(i, f(i, i)),
3   outl != outr]
4
5
6 Codes not equivalent, example trace:
7 [i = T!val!0,
8   outr = T!val!3,
9   outl = T!val!2,
10  o2 = T!val!1,
11  o1 = T!val!0,
12  f = [(T!val!1, T!val!0) -> T!val!2,
13        (T!val!0, T!val!1) -> T!val!3,
14        else -> T!val!1]

```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

1. The encoding does

- The trace states the following:
  - Given  $i=0$ ,  $outr=2$
  - Given  $i=0$ ,  $outr=3$
  - The problem is the function  $f$
  - The function  $f$  defined by SMT in C++ is given in Listing 17

```

1 template <typename T>
2 T f (T a, T b) {
3     if (a == 1 && b == 0)
4         return 2;
5     else if (a == 0 && b == 1)
6         return 3;
7     else return 1;
8 }

```

Figure 17: SMT defined function  $f$  in C++

2. What is incorrect about function  $f$ ?

- It is **not** commutative.
  - Returned values from cases  $\text{if}(a==0 \ \&\& \ b==1)$  and  $\text{if}(a==1 \ \&\& \ b==0)$  should be the same!
  - Note that  $+$ ,  $*$ , etc are all commutative.
  - Example:  $2*3 == 6 == 3*2$ ,  $2+3 == 5 == 3+2$ .
- We can enforce this *property* using the following logic:

- $\forall x, y \in T, f(x, y) == f(y, x)$
- The correct SMT encoding is shown in Listing 18

```

1  #!/usr/bin/env python3
2  from z3 import Solver, sat, And, Consts
3
4
5  def general():
6      from z3 import Function, ForAll, DeclareSort
7
8      # Declaring the new type T
9      T = DeclareSort('T')          # A new Type "T"
10
11     s = Solver()
12
13     # Declaring the variables we need for type T
14     i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)
15
16     # Declaring the new function "f" of type signature: (T, T) -> T
17     f = Function('f', T, T, T)
18
19     # Adding the commutativity constraint
20     x, y = Consts('x y', T)
21     s.add(ForAll([x, y], f(x, y) == f(y, x)))
22
23     # Make outl, replacing the operator with f everywhere
24     s.add(And(o1 == i, o2 == f(o1, i), outl == f(o2, i)))
25
26     # Make outr, replacing the operators with f everywhere
27     s.add(outr == f(i, f(i, i)))
28
29     s.add(outl != outr)
30
31     # Print what is in the so
32     print('Solver state: %s' % s)
33     print('\n')
34
35     if s.check() == sat:
36         # Print the model if something is wrong
37         print('Codes not equivalent, example trace:')
38         print(s.model())
39     else:
40         # Else everything is A OK!
41         print('Codes are equivalent')
42
43
44 def main():
45     general()
46
47
48 if __name__ == '__main__':
49     main()
50

```

Figure 18: Second and correct SMT generic functional equivalence encoding

```

1  Solver state: [ForAll([x, y], f(x, y) == f(y, x)),
2  And(o1 == i, o2 == f(o1, i), outl == f(o2, i)),
3  outr == f(i, f(i, i)),
4  outl != outr]
5

```

```

6
7 Codes are equivalent

```

Figure 19: Results for Listing 18

3. Hence, the functional and optimised code are equivalent for all commutative operators and of any type.

#### 4.3.4 Relaxing the commutativity constraint

- Consider the program with matrices in python – in Listing 20
- Are the specification and implementation equivalent?

```

1 import numpy as np
2 a = np.array([[1, 2], [3, 4]])
3 b = a;
4 for i in range(2):
5     b = b*a;
6
7 c = (a*a*a);
8
9 # are they equal?
10 print(c == b)

```

Figure 20: Specification and (optimised) implementation, Example 3

```

1 [[ True  True]
2  [ True  True]]

```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

- However, matrix multiplication is **not** a commutative operation
  - $A * B \neq B * A$
- Hence, our previous proof does not apply to the matrix multiplication
- So can we do better?
- What is the common property shared between addition, and multiplication for `int` and matrix?
  - It is associativity, i.e.,  $\forall x, y, z \in T, f(f(x, y), z) == f(x, f(y, z))$
  - We need to replace the commutativity constraint with the associativity constraint in the SMT encoding.
- Using associativity makes it more general.
- The proof applies to more operations of any type.
- The associative SMT encoding is given in Listing 22

```

1 #!/usr/bin/env python3
2 from z3 import Solver, sat, And, Consts
3
4
5 def general():
6     from z3 import Function, ForAll, DeclareSort
7
8     # Declaring the new type T
9     T = DeclareSort('T')          # A new Type "T"

```

```

10
11 s = Solver()
12
13 # Declaring the variables we need for type T
14 i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)
15
16 # Declaring the new function "f" of type signature: (T, T) -> T
17 f = Function('f', T, T, T)
18
19 # Adding the associativity constraint
20 x, y, z = Consts('x y z', T)
21 s.add(ForAll([x, y, z], f(f(x, y), z) == f(x, f(y, z))))
22
23 # Make outl, replacing the operator with f everywhere
24 s.add(And(o1 == i, o2 == f(o1, i), outl == f(o2, i)))
25
26 # Make outr, replacing the operators with f everywhere
27 s.add(outr == f(i, f(i, i)))
28
29 s.add(outl != outr)
30
31 # Print what is in the solver
32 print('Solver state: %s' % s)
33 print('\n')
34
35 if s.check() == sat:
36     # Print the model if something is wrong
37     print('Codes not equivalent, example trace:')
38     print(s.model())
39 else:
40     # Else everything is A 0
41     print('Codes are e
42
43
44 def main():
45     general()
46
47
48 if __name__ == '__main__':
49     main()

```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu\_assist\_pro

Figure 22: SMT encoding of software program equivalence with associativity

```

1 Solver state: [ForAll([x, y, z], f(f(x, y), z) == f(x, f(y, z))),
2 And(o1 == i, o2 == f(o1, i), outl == f(o2, i)),
3 outr == f(i, f(i, i)),
4 outl != outr]
5
6
7 Codes are equivalent

```

Figure 23: Result of Listing 22

## 5 Modelling the task allocation problem

- In this section we solve an **optimisation** problem using SMT.
- Consider Figure 24

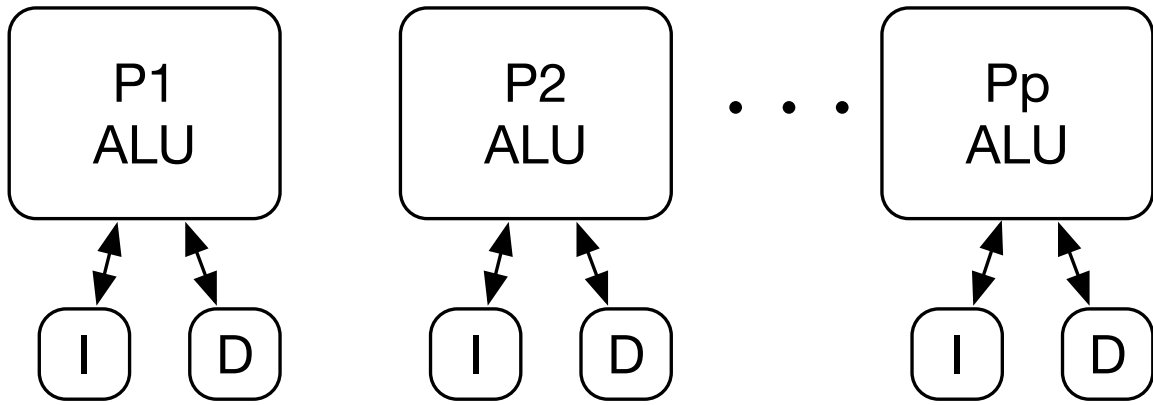


Figure 24: The processor architecture

- There are  $P$  processors, each with their own ALU, instruction, and data cache.
- The processors do not communicate with each other at all.
- We are given  $T$  independent tasks, that do not communicate with each other.
- Each task takes  $X$  time units to execute on *any* processor.
- We want to:
  1. Allocate the  $T$  tasks onto the processors, such that each task is allocated only on a single processor.
  2. We want to do an optimal allocation, such that all tasks run to completion in the *shortest* time.

### 5.1 SMT solution to the task allocation problem

- We will first formally (in SMT) encode the task allocation problem.
- Then we will encode it into SMT
- Then we will add optimisation

#### 5.1.1 Mathematical model of the task allocation problem.

- Let  $I = T \times P$  be a matrix be a ones and zeros.
- $I[i][j] = 1$  if task  $i \in T$  is allocated to processor  $j \in P$ , else it is 0.
- An example  $I$  matrix with 3 tasks and 2 processors is shown in Equation 9.
- Rows represent the tasks and columns represent the processors in Equation 9.

	$P1$	$P2$	
$T1$	1	0	
$T2$	0	1	
$T3$	1	0	(9)

- Formally, the matrix  $I$  is represented as  $I[i][j] \in \{0, 1\}, \forall i \in T, \forall j \in P$ .
- Since, a given task can only be assigned to a single processor; the sum of all values in a *row* should be 1.
- Formally,  $\sum I[i] == 1, \forall i \in T$ .
- Since, each task takes  $X$  units of time on *any* processor. The total execution time for *each* processor is given by the sum of the column for that processor multiplied by  $X$ .



- For example, in Equation 9;  $P_1$  has two tasks allocated to it. Hence, total execution time of all tasks on  $P_1$  is:  $1 \times X + 0 \times X + 1 \times X == (1 + 0 + 1) \times X == 2 \times X$ .
- Formally,  $E[j] = (\sum_{i \in T} I[i][j]) \times X, \forall j \in P$ , where  $E$  is a vector of execution times for *each* processor.
- Total time taken for completion *makespan* =  $\max(E_j), \forall j \in P$ .

### 5.1.2 Encoding the allocation problem in SMT

- Listing 25 shows the task allocation problem encoded in SMT.

```

1  from z3 import Solver, sat, Or, If, Real
2
3
4  def reduce(sV, Ej):
5      def max(x, y):
6          return If(x > y, x, y)
7
8      if len(Ej) == 0:
9          return sV
10     else:
11         # Return the max from first and rest
12         return max(Ej[0], reduce(sV, Ej[1:]))
13
14
15 def main(P, T, X):
16     """P is the number of processors
17     T is the number of tasks
18     X is the execution time of each task on any processor
19     """
20     assert(X >= 0)
21
22     # Initialise the solver
23     s = Solver()
24
25     # Making the 1/0 Reals
26     Iijs = [[Real('I_%s_%s' % (i, j))
27              for j in range(P)] for i in range(T)]
28
29     # Adding the constraint that they can only be 1 or 0
30     [s.add(Or(Iijs[i][j] == 1, Iijs[i][j] == 0))
31      for i in range(T) for j in range(P)]
32
33     # Now, making sure that the allocation of task is only on one
34     # processor.
35     [s.add(1 == sum(Iijs[i])) for i in range(T)]
36
37     # Next compute the total execution time for each processor
38     Ej = [Real('E_%s' % j) for j in range(P)]
39
40     # Adding the constraint for the total execution time
41     for j in range(P):
42         V = 0
43         for i in range(T):
44             V += Iijs[i][j]
45         s.add(Ej[j] == V*X)
46
47     # Now the total makespan
48     makespan = Real('makespan')
49     s.add(makespan == reduce(0, Ej))
50     ret = s.check()
51     if ret == sat:

```

```

52     model = s.model()
53
54     # The makespan
55     print('Result makespan %s\n' % (model[makespan]))
56
57     # The allocations
58     print('Allocations: \n')
59     for i in range(T):
60         row = [str(model[Iijs[i][j]]) for j in range(P)]
61         print('\t'.join(row))
62     else:
63         print('No satisfaction found. No model!')
64
65
66 if __name__ == '__main__':
67     P = 4
68     T = 5
69     X = 100
70     main(P, T, X)

```

Figure 25: SMT encoding of the task allocation problem

```

1 Result makespan 200
2
3 Allocations:
4
5 1      0      0      0
6 0      0      1      0
7 0      1      0
8 1      0      0
9 0      1      0

```

Figure 26: Results for Li

### 5.1.3 Optimal task allocation

- Obviously the result in Listing 26 is not optimal.
- We can see that some tasks can be moved around in Listing 26 to get a shorter makespan.
- We will now design an optimal solution.
- In the **worst** case all tasks can get allocated to the same processor.
  - We will call this the upper bound
- Hence, upper bound  $UB = X \times T$
- In the **best** case (not always feasible), all tasks get allocated to a different processor.
  - We will call this the lower bound
- Hence, the lower bound  $LB = UB/P$
- The optimal makespan is somewhere between  $UB$  and  $LB$ , i.e.,  $optimal\_makespan \in [UB, LB]$ .
- Hence, we can perform a *binary search* between these two bounds to get the *optimal\_makespan*.

#### 5.1.4 SMT encoding for the *optimal* task allocation

- The SMT encoding for the optimal task allocation and the result are shown in Listings 27 and 28
- The optimal makespan problem is NP-hard.
- The execution time of the SMT solver will grow exponentially with increasing number of tasks or processors.

```
1 from z3 import Solver, sat, Or, If, Real
2
3
4 def reduce(sV, Ej):
5     def max(x, y):
6         return If(x > y, x, y)
7
8     if len(Ej) == 0:
9         return sV
10    else:
11        return max(Ej[0], reduce(sV, Ej[1:]))
12
13
14 Iijs = None
15 def main(P, T, X):
16     """P is the number of processors
17     T is the number of tasks
18     X is the execution time of each task on any processor
19     """
20     assert(X >= 0)
21
22     # Initialise the solver
23     s = Solver()
24
25     # Making the 1/0 Reals
26     global Iijs
27     Iijs = [[Real('I_%s_%s' % (i, j))
28              for j in range(P)] for i in range(T)]
29
30     # Adding the constraint that they can only be 1 or 0
31     [s.add(Or(Iijs[i][j] == 1, Iijs[i][j] == 0))
32      for i in range(T) for j in range(P)]
33
34     # Now, making sure that the allocation of task is only on one
35     # processor.
36     [s.add(1 == sum(Iijs[i])) for i in range(T)]
37
38     # Next compute the total execution time for each processor
39     Ej = [Real('E_%s' % j) for j in range(P)]
40
41     # Adding the constraint for the total execution time
42     for j in range(P):
43         V = 0
44         for i in range(T):
45             V += Iijs[i][j]
46         s.add(Ej[j] == V*X)
47
48     # Now the total makespan
49     makespan = Real('makespan')
50     s.add(makespan == reduce(0, Ej))
51
52     return s, makespan
53
54
```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu\_assist\_pro

```

55 def binary_search(lb, ub, s, makespan, epsilon=1e-6):
56     if (ub - lb <= epsilon):
57         global Iijs
58         s.check()
59         return s.model()[makespan], Iijs, s.model()
60     else:
61         half = lb + ((ub - lb)/2.0)
62         s.push()
63         s.add(makespan >= lb, makespan <= half)
64         ret = s.check()
65         s.pop()
66         if ret == sat:
67             return binary_search(lb, half, s, makespan, epsilon)
68         else:
69             return binary_search(half, ub, s, makespan, epsilon)
70
71
72 if __name__ == '__main__':
73     P = 4
74     T = 5
75     X = 100
76     s, makespan = main(P, T, X)
77
78     # Now do a binary search for the optimal makespan between lower and
79     # upper bounds
80
81     UB = (X*T)
82     LB = (UB/P)
83
84     optimal_makespan, Iijs, model = binary_search(LB, UB, s, makespan)
85     print('Optimal makes
86
87     # The allocations
88     print('Allocations
89     for i in range(T):
90         row = [str(model[Iijs[i][j]]) for j in range(P)]
91         print('\t'.join(row))

```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu\_assist\_pro

Figure 27: SMT encoding of the task allocation problem

```

1 Optimal makespan 200
2
3 Allocations:
4
5 0      0      1      0
6 1      0      0      0
7 0      0      0      1
8 0      1      0      0
9 0      0      1      0

```

Figure 28: Result of executing Listing 27