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 in SMT solver

- Consider the linear system of inequalities in Equation 1:

$$\begin{aligned} x + y &\geq 10 \\ x - y &\geq 20 \end{aligned} \tag{1}$$

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

$$\begin{aligned} x + y &\geq 10 \\ \therefore x &\geq 10 - y \\ \therefore 10 - y - y &\geq 20 \\ \therefore -2y \geq 10 &\implies y \leq -5 \\ \therefore x \geq 10 - (-5) &\implies x \geq 15 \end{aligned} \tag{2}$$

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

```
#!/usr/bin/env python3

# Using the z3 SMT solver with python3 bindings
from z3 import IntSort, Solver, sat, And, Consts
# Importing the datatype Int, the Solver class, and the 'sat' variable.
# Finally, the And (logical And class)

def main():
        # Declaring and defining the two variables.
```

```
        x, y = Consts('x y', IntSort())  # IntSort just stands for Int (type)

        s = Solver()                # Initialising the solver

        # Adding the equations into the solver
        s.add(And(x + y >= 10, x - y >= 20))

        # Show the state of the solver
        print('Solver state: %s' % s)                    # Just for debugging

        # Solving for all free variables: x and y
        ret = s.check()

        # Check if there is some assignment for x and y
        # that satisfy the equations
        if ret == sat:
                print('Result:')
                print('x: %s' % s.model()[x])
                print('y: %s' % s.model()[y])


# Calling the main function in python
if __name__ == '__main__':
        main()
```

Figure 1: Example 1


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


Figure 2: Results for Example 1

- See Z3 documentation
- See Z3 API

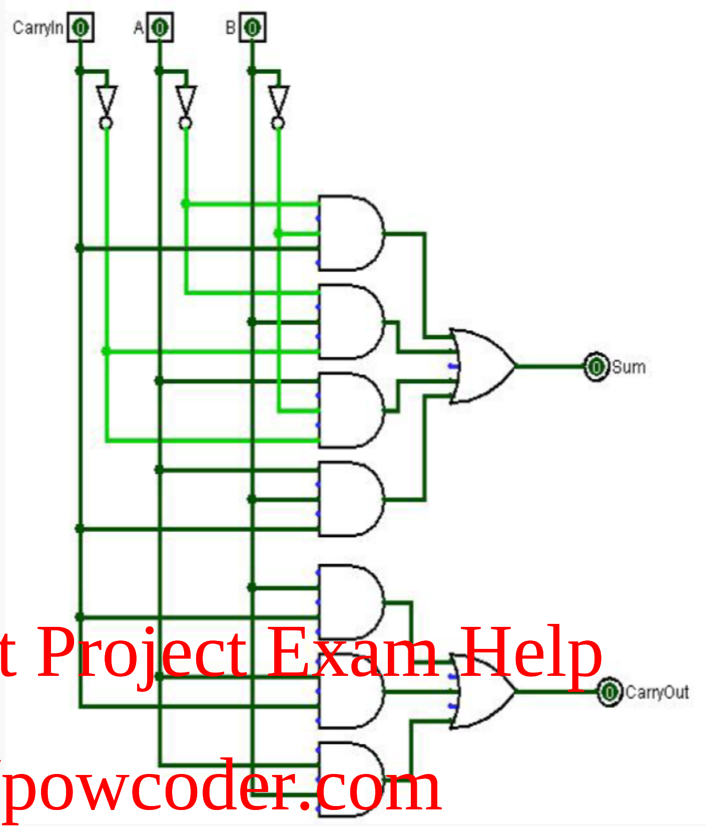
3 Hardware circuit equivalence

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

The expressions for the sum and carry lead to the following unified implementation:

$$Sum = \overline{A}\cdot\overline{B}\cdot C_{in} + \overline{A}\cdot B\cdot\overline{C_{in}} + A\cdot\overline{B}\cdot\overline{C_{in}} + A\cdot B\cdot C_{in}$$

$$Carry = B\cdot C + A\cdot C + A\cdot B$$

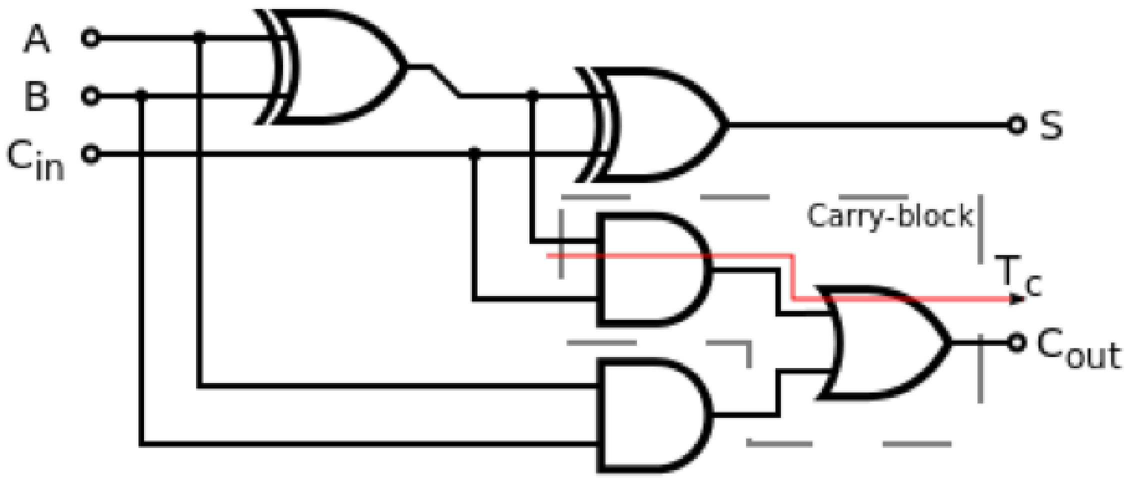


This implementation requires only two levels of logic (ignoring the inverters as customary).

Is there an alternative design that requires fewer AND/OR gates? If so, how many levels does it require?

- We have three inputs `Cin`, `A`, and `B`.
- We have two outputs, `Sum` and `Carry`.
- There are a number of gates that compute the Sum and the Carry.

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.

$$\begin{aligned}
Sumf \Leftrightarrow & \\
& ((\neg A) \land (\neg B) \land C) \lor \\
& ((\neg A) \land B \land (\neg C)) \lor \\
& (A \land (\neg B) \land (\neg C)) \lor \\
& (A \land B \land C) && (3) \\
Carryf \Leftrightarrow & (B \land C) \lor (A \land C) \lor (A \land B) && (4)
\end{aligned}$$

- The logical operators mean the following:
  1. $\Leftrightarrow$ is logical equivalence.
  2. $\neg$ is logical negation (or not).
  3. $\land$ is logical conjunction (or and).
  4. $\lor$ is logical disjunction (or operator).
  5. $\oplus$ is logical XOR.
- The logic can be encoded into Z3 solver as follows:

```python
# Note the captialisation of Boolean operators.
from z3 import And, Not, Or, solver

def functional(Sumf, Carryf, A, B, C, s):
    # Note <=> is converted into ==
    s.add(Sumf == Or(And(Not(A), Not(B), C),
                     And(Not(A), B, Not(C)),
                     And(A, Not(B), Not(C)),
                     And(A, B, C)))

    s.add(Carryf == Or(And(B, C), And(A, C), And(A, B)))

```


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

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

```python
from z3 import And, Not, Xor, BoolSort, Or, sat
from z3 import Solver, Consts
def implementation(Si, Ci, A, B, C, s):
    u, v, w = Consts('u, v, w', BoolSort())
    s.add(u == Xor(A, B))
    s.add(v == And(u, C))
    s.add(w == And(A, B))
    s.add(Si == Xor(u, C))
    s.add(Ci == Or(w, v))
```

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

```
#!/usr/bin/env python3

from z3 import And, Not, Xor, BoolSort, Or, sat
from z3 import Solver, Consts


def functional(sumf, carryf, a, b, c, s):
    s.add(sumf == Or(And(Not(a), Not(b), c),
                     And(Not(a), b, Not(c)),
                     And(a, Not(b), Not(c)),
                     And(a, b, c)))

    s.add(carryf == Or(And(b, c), And(a, c), And(a, b)))


def implementation(Si, Ci, A, B, C, s):
    u, v, w = Consts('u, v, w', BoolSort())
    s.add(u == Xor(A, B))
    s.add(v == And(u, C))
    s.add(w == And(A, B))
    s.add(Si == Xor(u, C))
    s.add(Ci == Or(w, v))


def main():
    A, B, Cin = Consts('A, B, Cin', BoolSort())
    Sf, Cf = Consts('Sf, Cf', BoolSort())
    s = Solver()
    functional(Sf, Cf, A, B, Cin, s)
    Si, Ci = Consts('Si, Ci', BoolSort())
    implementation(Si, Ci, A, B, Cin, s)

    # Now the "mitre" circuit
    s.add(Or(Xor(Sf, Si), Xor(Cf, Ci)))

    # Check if the circuits are equivalent
    if s.check() == sat:
        print('Circuits not equivalent')
        print(s.model())        # print the values of A, B, C, etc.
    else:
        print('Circuits are equivalent')


if __name__ == '__main__':
    main()
```


Figure 3: Hardware Equivalence Encoding

```
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.

```
/* Written by the programmer */
int power3_func(int i){
  int outl;
  outl = i;
  for (int ii = 0; ii < 2; ++ii) {
    outl *= i;
  }
  return outl;
}
/* Optimised by the compiler */
int power3_impl(int i){
  int outr = ((i * i) * i);
  return outr;
}
```


Figure 5: Hand written and optimised programs

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

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

```
/* Original form */
int power3_func(int i){
  int outl;
  outl = i;
  for (int ii = 0; ii < 2; ++ii) {
    outl *= i;
  }
  return outl;
}

/* SSA form */
int power3_func_ssa(int i){
  int outl;
  int o1 = i;
  /* Loop unrolled */
  int o2 = o1 * i;                /* iteration 1 */
```

```
    outl = o2 * i;                /* iteration 2 */
    return outl;
}

```

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

$$(o1 == i) \wedge (o2 == o1 * i) \wedge (outl == o2 * i) \tag{6}$$

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

$$(outr == ((i * i) * i) \tag{7}$$

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

- The SMT encoding is shown in Listing 7

```
#!/usr/bin/env python3
# The standard imports
from z3 import Solver, sat, And, Consts


def mul():
    # Importing the type Int
    from z3 import IntSort      # IntSort is typedef for Int type
    s = Solver()                # Importing the solver
    # Declaring the variables we need
    i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', IntSort())
    # Encode outl
    s.add(And(o1 == i, o2 == (o1 * i), outl == (o2 * i)))

    # Encode outr
    s.add(outr == (i * (i * i)))

    # Encode the condition that there exists no i, such that outl !=
    # outr
    s.add(outl != outr)

    if s.check() == sat:
        print('Codes not equivalent, example:')
        print(s.model())
    else:
        print('Codes are equivalent')


if __name__ == '__main__':
    mul()

```


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

```
/* Written by the programmer */
float add3_func(float i){
  float outl;
  outl = i;
  for (int ii = 0; ii < 2; ++ii) {
    outl += i;
  }
  return outl;
}
/* Optimised by the compiler */
float add3_impl(float i){
  float outr = ((i + i) + i);
  return outr;
}
```

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


```
#!/usr/bin/env python3
from z3 import Solver, sat, And, Consts


def add():
    # Importing the Real type, to simulate floats
    from z3 import RealSort
    s = Solver()

    # Declaring variables as Reals
    i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', RealSort())
    # Make add3_func_ssa format
    s.add(And(o1 == i, o2 == (o1 + i), outl == (o2 + i)))

    # Make outr
    s.add(outr == (i + (i + i)))


    # Add the equivalence statement
    s.add(outl != outr)
    if s.check() == sat:
        print('Codes not equivalent, example:')
        print(s.model())
    else:
        print('Codes are equivalent')


if __name__ == '__main__':
    add()
```

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

```
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

```
template <typename T>
T add3_func(T i){
  T outl;
  outl = i;
  for (int ii = 0; ii < 2; ++ii) {
    outl += i;
  }
  return outl;
}
template <typename T>
/* Optimised by the compiler */
T add3_impl(T i){
  T outr = ((i + i) + i);
  return outr;
}
```


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

```python
#!/usr/bin/env python3
from z3 import Solver, sat, And, Consts


def general():
    from z3 import DeclareSort

    # Declare the new type T
    T = DeclareSort('T')

    s = Solver()

    # Declare the variables of type T
    i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)

    # FIXME: Here we need to fill in the logic

    # Same as before, checking if for some i, outl and outr are
    # different.
    s.add(outl != outr)
    if s.check() == sat:
        print('Codes not equivalent, example:')
        print(s.model())
    else:
        print('Codes are equivalent')


if __name__ == '__main__':
    general()
```


Figure 13: SMT encoding with generic sort (type)

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

```cpp

template <typename T>
// We will not define the function f.
// We will let SMT define the function for us!
T f (T, T);

template <typename T>
T add3_func(T i){
  T outl;
  outl = i;
  for (int ii = 0; ii < 2; ++ii) {
    outl = f(outl, i);                // Replaced the operator with f
  }
  return outl;
}

template <typename T>
```

```
/* Optimised by the compiler */
T add3_impl(T i){
  T outr = f(f(i, i), i);          // Replaced the operator with f
  return outr;
}
```

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

```
#!/usr/bin/env python3
from z3 import Solver, sat, And, Consts


def general():
    from z3 import Function, ForAll, DeclareSort

    # Declaring the new type T
    T = DeclareSort('T')        # A new Type "T"

    s = Solver()

    # Declaring the variables we need for type T
    i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)

    # Declaring the new function "f" of type signature: (T, T) -> T
    f = Function('f', T, T, T)

    # Make outl, replacing the operator with f everywhere
    s.add(And(o1 == i, o2 == f(o1, i), outl == f(o2, i)))

    # Make outr, replacing the operators with f everywhere
    s.add(outr == f(i, f(i, i)))

    s.add(outl != outr)

    # Print what is in the solver
    print('Solver state: %s' % s)
    print('\n')

    if s.check() == sat:
        # Print the model if something is wrong
        print('Codes not equivalent, example trace:')
        print(s.model())
    else:
        # Else everything is A OK!
```

```
        print('Codes are equivalent')


def main():
    general()


if __name__ == '__main__':
    main()

```

Figure 15: First generic encoding in SMT

```
Solver state: [And(o1 == i, o2 == f(o1, i), outl == f(o2, i)),
 outr == f(i, f(i, i)),
 outl != outr]


Codes not equivalent, example trace:
[i = T!val!0,
 outr = T!val!3,
 outl = T!val!2,
 o2 = T!val!1,
 o1 = T!val!0,
 f = [(T!val!1, T!val!0) -> T!val!2,
      (T!val!0, T!val!1) -> T!val!3,
      else -> T!val!1]]
```


Figure 16: Results for Listing 15

1. The encoding does **not** work
   - The trace states the following:
     - Given i=0, outl=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


```
template <typename T>
T f (T a, T b) {
  if (a == 1 && b == 0)
    return 2;
  else if (a == 0 && b == 1)
    return 3;
  else return 1;
}
```

Figure 17: SMT defined function f in C++

2. What is incorrect about function f?
   - It is **not** commutative.
     - Returned values from cases if(a==0 && b==1) and (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

```
#!/usr/bin/env python3
from z3 import Solver, sat, And, Consts


def general():
    from z3 import Function, ForAll, DeclareSort

    # Declaring the new type T
    T = DeclareSort('T')        # A new Type "T"

    s = Solver()

    # Declaring the variables we need for type T
    i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)

    # Declaring the new function "f" of type signature: (T, T) -> T
    f = Function('f', T, T, T)

    # Adding the commutativity constraint
    x, y = Consts('x y', T)
    s.add(ForAll([x, y], f(x, y) == f(y, x)))

    # Make outl, replacing the operator with f everywhere
    s.add(And(o1 == i, o2 == f(o1, i), outl == f(o2, i)))

    # Make outr, replacing the operators with f everywhere
    s.add(outr == f(i, f(i, i)))

    s.add(outl != outr)

    # Print what is in the solver
    print('Solver state: %s' % s)
    print('\n')

    if s.check() == sat:
        # Print the model if something is wrong
        print('Codes not equivalent, example trace:')
        print(s.model())
    else:
        # Else everything is A OK!
        print('Codes are equivalent')


def main():
    general()


if __name__ == '__main__':
    main()
```


Figure 18: Second and correct SMT generic functional equivalence encoding

```
Solver state: [ForAll([x, y], f(x, y) == f(y, x)),
 And(o1 == i, o2 == f(o1, i), outl == f(o2, i)),
 outr == f(i, f(i, i)),
 outl != outr]

```

```

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?

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = a;
for i in range(2):
    b = b*a;

c = (a*a*a);

# are they equal?
print(c == b)
```

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


```
[[ True  True]
 [ True  True]]
```


Figure 21: Results for Listing 20

- However, matrix multiplication is **not** a commutative operator.
  - $A * B \neq B * A$

- Hence, our previous proof does not apply to the matrix multiplication operator.
- 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

```
#!/usr/bin/env python3
from z3 import Solver, sat, And, Consts


def general():
    from z3 import Function, ForAll, DeclareSort

    # Declaring the new type T
    T = DeclareSort('T')        # A new Type "T"
```

```

    s = Solver()

    # Declaring the variables we need for type T
    i, o1, o2, outl, outr = Consts('i o1 o2 outl outr', T)

    # Declaring the new function "f" of type signature: (T, T) -> T
    f = Function('f', T, T, T)

    # Adding the associativity constraint
    x, y, z = Consts('x y z', T)
    s.add(ForAll([x, y, z], f(f(x, y), z) == f(x, f(y, z))))

    # Make outl, replacing the operator with f everywhere
    s.add(And(o1 == i, o2 == f(o1, i), outl == f(o2, i)))

    # Make outr, replacing the operators with f everywhere
    s.add(outr == f(i, f(i, i)))

    s.add(outl != outr)

    # Print what is in the solver
    print('Solver state: %s' % s)
    print('\n')

    if s.check() == sat:
        # Print the model if something is wrong
        print('Codes not equivalent, example trace:')
        print(s.model())
    else:
        # Else everything is A OK!
        print('Codes are equivalent')


def main():
    general()


if __name__ == '__main__':
    main()
```


Figure 22: SMT encoding of software program equivalence with associativity

```
Solver state: [ForAll([x, y, z], f(f(x, y), z) == f(x, f(y, z))),
 And(o1 == i, o2 == f(o1, i), outl == f(o2, i)),
 outr == f(i, f(i, i)),
 outl != outr]


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

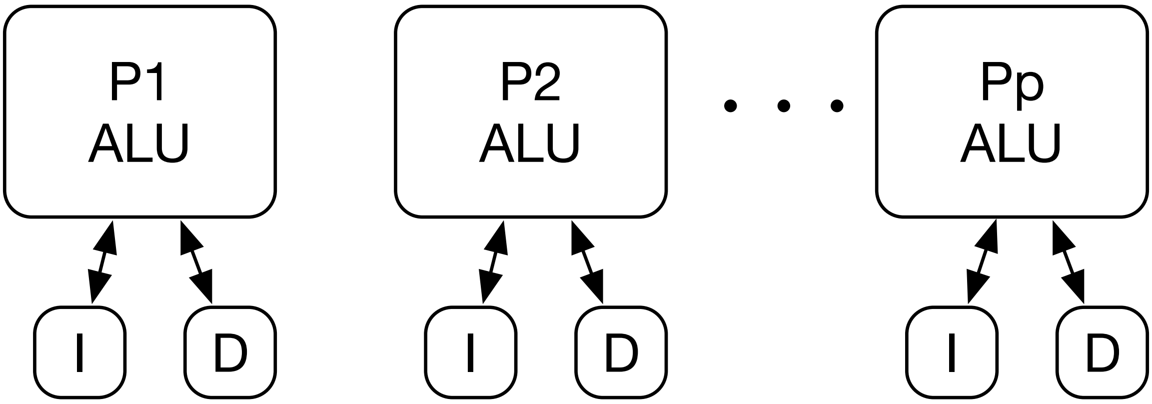


Figure 24: The processor architecture

- 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 (mathematically) describe the 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.

$$\begin{array}{ccc} & P1 & P2 \\ T1 & 1 & 0 \\ T2 & 0 & 1 \\ T3 & 1 & 0 \end{array} \tag{9}$$

- 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; $P1$ has two tasks allocated to it. Hence, total execution time of all tasks on $P1$ is: $1 \times X + 0 \times X + 1 \times X == (1 + 0 + 1) \times X == 2 \times X$.

- Formally, $E[j] = (\sum_{\forall i \in T} I[i][j]) \times X, \forall j \in P$, 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.

```
from z3 import Solver, sat, Or, If, Real


def reduce(sV, Ej):
    def max(x, y):
        return If(x > y, x, y)

    if len(Ej) == 0:
        return sV
    else:
        # Return the max from first and rest
        return max(Ej[0], reduce(sV, Ej[1:]))


def main(P, T, X):
    """P is the number of processors
    T is the number of tasks
    X is the execution time of each task on any processor
    """
    assert(X >= 0)

    # Initialise the solver
    s = Solver()

    # Making the 1/0 Reals
    Iijs = [[Real('I_%s_%s' % (i, j))
             for j in range(P)] for i in range(T)]

    # Adding the constraint that they can only be 1 or 0
    [s.add(Or(Iijs[i][j] == 1, Iijs[i][j] == 0))
     for i in range(T) for j in range(P)]

    # Now, making sure that the allocation of task is only on one
    # processor.
    [s.add(1 == sum(Iijs[i])) for i in range(T)]

    # Next compute the total execution time for each processor
    Ej = [Real('E_%s' % j) for j in range(P)]

    # Adding the constraint for the total execution time
    for j in range(P):
        V = 0
        for i in range(T):
            V += Iijs[i][j]
        s.add(Ej[j] == V*X)

    # Now the total makespan
    makespan = Real('makespan')
    s.add(makespan == reduce(0, Ej))
    ret = s.check()
    if ret == sat:
```

```
        model = s.model()

        # The makespan
        print('Result makespan %s\n' % (model[makespan]))

        # The allocations
        print('Allocations: \n')
        for i in range(T):
            row = [str(model[Iijs[i][j]]) for j in range(P)]
            print('\t'.join(row))
    else:
        print('No satisfaction found. No model!')


if __name__ == '__main__':
    P = 4
    T = 5
    X = 100
    main(P, T, X)
```

Figure 25: SMT encoding of the task allocation problem

```
Result makespan 200

Allocations:

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


Figure 26: Results for Listing 25

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.

```
from z3 import Solver, sat, Or, If, Real


def reduce(sV, Ej):
    def max(x, y):
        return If(x > y, x, y)

    if len(Ej) == 0:
        return sV
    else:
        return max(Ej[0], reduce(sV, Ej[1:]))


Iijs = None
def main(P, T, X):
    """P is the number of processors
    T is the number of tasks
    X is the execution time of each task on any processor
    """
    assert(X >= 0)

    # Initialise the solver
    s = Solver()

    # Making the 1/0 Reals
    global Iijs
    Iijs = [[Real('I_%s_%s' % (i, j))
             for j in range(P)] for i in range(T)]

    # Adding the constraint that they can only be 1 or 0
    [s.add(Or(Iijs[i][j] == 1, Iijs[i][j] == 0))
     for i in range(T) for j in range(P)]

    # Now, making sure that the allocation of task is only on one
    # processor.
    [s.add(1 == sum(Iijs[i])) for i in range(T)]

    # Next compute the total execution time for each processor
    Ej = [Real('E_%s' % j) for j in range(P)]

    # Adding the constraint for the total execution time
    for j in range(P):
        V = 0
        for i in range(T):
            V += Iijs[i][j]
        s.add(Ej[j] == V*X)

    # Now the total makespan
    makespan = Real('makespan')
    s.add(makespan == reduce(0, Ej))

    return s, makespan
```

```python
def binary_search(lb, ub, s, makespan, epsilon=1e-6):
    if (ub - lb <= epsilon):
        global Iijs
        s.check()
        return s.model()[makespan], Iijs, s.model()
    else:
        half = lb + ((ub - lb)/2.0)
        s.push()
        s.add(makespan >= lb, makespan <= half)
        ret = s.check()
        s.pop()
        if ret == sat:
            return binary_search(lb, half, s, makespan, epsilon)
        else:
            return binary_search(half, ub, s, makespan, epsilon)


if __name__ == '__main__':
    P = 4
    T = 5
    X = 100
    s, makespan = main(P, T, X)

    # Now do a binary search for the optimal makespan between lower and
    # upper bounds

    UB = (X*T)
    LB = (UB/P)

    optimal_makespan, Iijs, model = binary_search(LB, UB, s, makespan)
    print('Optimal makespan %s \n' % optimal_makespan)

    # The allocations
    print('Allocations: \n')
    for i in range(T):
        row = [str(model[Iijs[i][j]]) for j in range(P)]
        print('\t'.join(row))
```


Figure 27: SMT encoding of the task allocation problem

```
Optimal makespan 200

Allocations:

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

Figure 28: Result of executing Listing 27