[Th 2] Symbolic execution

The objective of the symbolic execution is to verify how the program will behave respect with to a set of inputs that is potentially infinite.

For example, SAGE (Microsoft) tries to find vulnerabilities in software production.

In general, we have to face a lot of problems, for example, a defect that makes the system crash when a null pointer is dereferenced or non-initialized.

In general symbolic execution has some theoretical limitations:

DYNAMIC ANALYSIS

Let's imagine having a function f such that does not call any function.

```
void f(int x, int y)
{    int t=0;
     if(x>y)
        t=x;
     else
        t=y;
     if(t<x)
        [X]
}
```

Now in this case, if we use a black-box approach like using the fuzzier we will focus on the results of the inputs. So we will test a lot of possible inputs.

In this case, we are using a dynamic analysis.

A dynamic analysis that is based on tests has some limitations:

```
void f(int x)
{    int y=x+3;
     if(y==13)
        [X]
}
```

In this specific case the only way to enter in the if clause is to have x=10. Now if we use a fuzzier we will try 2^{32} inputs (an int is 32 bits) to find a unique integer.

STATIC ANALYSIS (all branches together)

In static analysis, we map variable names to symbolic values to perform a theoretical analysis.

```
void f(int x, int y)
{    int t=0;
     if(x>y)
        t=x;
     else
        t=y;
     if(t<x)
        [X]
}
```

In this case, we assign to x and y symbolic value that we know is between min_int and max_int.

Looking at the two different branches (if and else) we can figure out that in the last if we can have a value for t such that:

- if x > y then t = x
- if t=y then t >= x

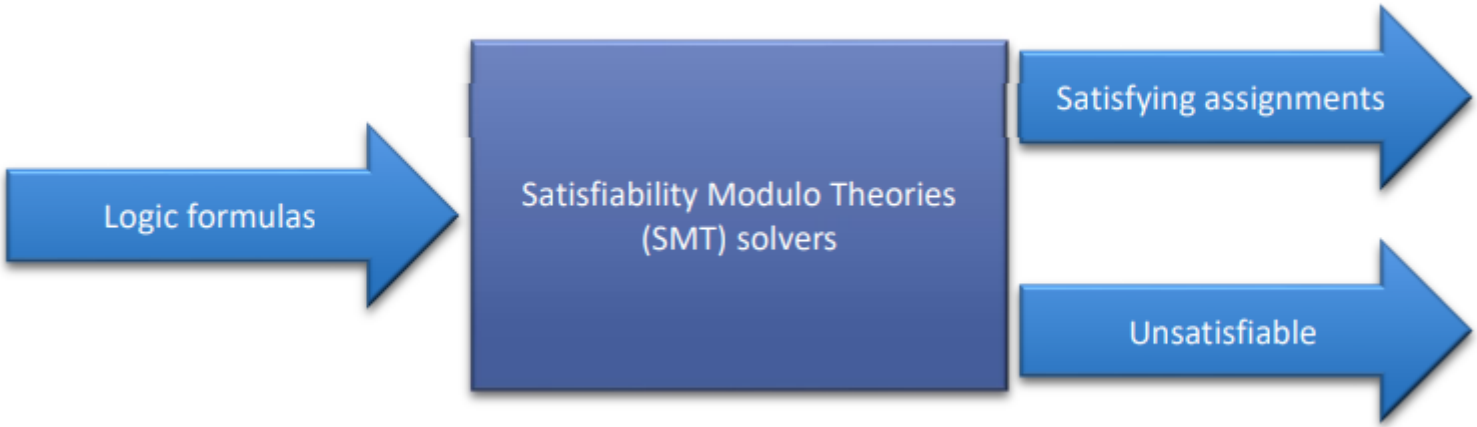
So is not possible to enter in the if clause for any possible value of x and y.

Symbolic execution can be encoded into an algorithm but how we can extract these formulas in a mechanical way?

How we can solve these formulas in a mechanical way?

SMT SOLVERS

The idea of using an STM (Satisfiability Modulo Theories) solver is to solve these formulas:



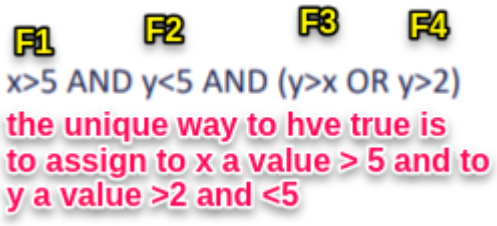
They take as input a logic formula and return an output that represents the solution or says if the formula is resolvable.

Theories usually have as domain real numbers, integers, lists, or non-interpreted formulas.
The simplest SMT problems are in NP-Complete so for this reason, many solvers after a lot of execution time return an answer that is "I don't know".

SMT solvers work on SAT solvers that use constrained databases. Each constraint is a CNF (conjunction of clauses in disjunction) formula.

In general, a SAT solver assigns to a CNF variable a truth value and derives the other clause variable values. In this way is able to add to the formula constraints that will reduce the entire formula complexity.

Example:



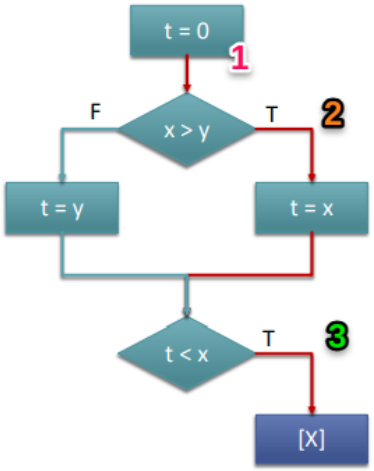
the complexity can be reduced if we add the constraints in which we have:

- NOT (F1 AND F2 and F3)

So in this specific case, we are considering all branches together

STATIC ANALYSIS (One Branch at a time)

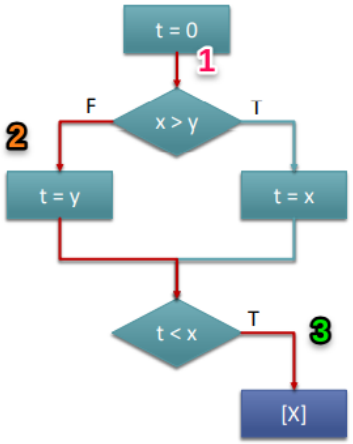
In this specific case, **we try to create a formula that represents the entire program but considers one branch at a time.**
The basic idea is to construct a subformula for each possible branch to understand if we reach unsatisfiable constraints.



X	Y	T	Constraints
x	y	0	true 1
x	y	x	x>y 2
x	y	x	x>y, x<x 3

- Unsatisfiable constraints

this branch will never take to the instruction [X]



X	Y	T	Constraints
x	y	0	true 1
x	y	y	x<=y 2
x	y	y	x<=y, y<x 3

- Unsatisfiable constraints
- We have explored all possible paths → we establish that the [X] instruction will never be reached

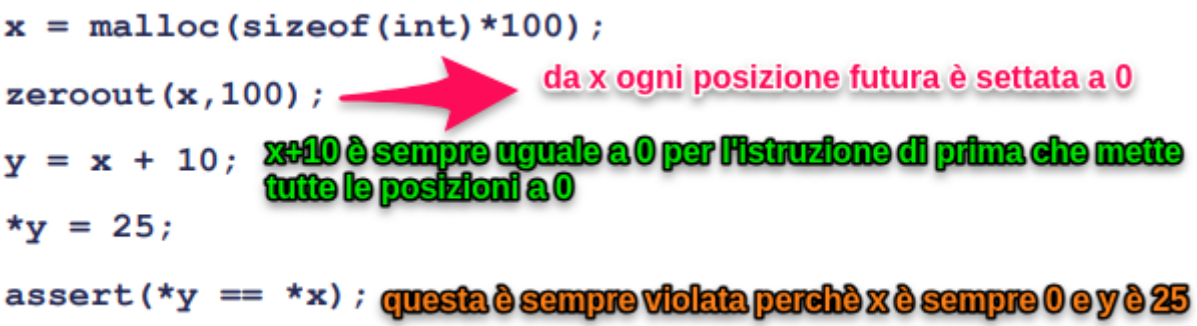
So in this case we have established that is not possible in any case to reach [X]

In general, we have to consider that there are exponentially many paths but we can use various strategies for reducing computational cost:

- **check the path conditions at every step**
- **never go down again on a path that was already excluded.**

EXAMPLE OF STATIC ANALYSIS: MODELING THE HEAP

Imagine doing a statical analysis of this code:



So in order to perform an analysis of our function we can convert it in this way:

Another better model for the heap:

- the entire heap is represented as a big array

```
x = malloc(sizeof(int)*100);
zeroout(x,100);
y = x + 10*sizeof(int);
MEM[y] = 25;
assert(MEM[y] == MEM[x]);
```

x e y sono int quindi mi muovo di 4 bytes

Mem can give us the access to the memory in pos y

We can easily understand that the difficutly here is to perfrom a static analysis of the malloc function (it is very complex)
So in order to perform an analysis of our function we can assume that the malloc function can be replaced with this easy function (only for the analysis) :

```
POS = 1;
int malloc(int n)
{    rv = POS;
     POS += n;
     return rv;
}
```

- In many practical cases, we can model «malloc» as if it just kept the last free memory position
- No freeing, no protected memory spaces
- In general, a very important step is modeling library functions
  - Again, tradeoff performance/precision

The objective is to model data structure to extract theories.

Theory of arrays

Let's consider a certain array **a**.

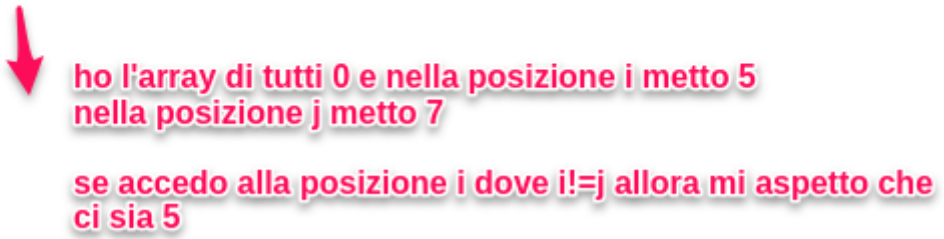
When we write **a{ i -> e }:**

- we refer to a new array in which we substitute the value of the i-th position with the new value e

Now following this notation:

- $$a\{i \to e\}[k] = \begin{cases} a[k] & if\ k \neq i \\ e & if\ k = i \end{cases}$$
- we are saying that when i create the new array and we acceded to the k position of it we will find:
  - e if k=i (that is the new value)
  - a[k] if k!=i (so we accessed on an unmodified element)

- Example:
  - Assume «Zero» is the zeroed out array
  - $$Zero\{i \to 5\}\{j \to 7\}[k] = 5 \leftrightarrow \begin{cases} true & if\ k = i\ and\ i \neq j \\ false & otherwise \end{cases}$$



CONCOLIC EXECUTION

We start with a sample input for the function and execute the function under trace. At each point the execution passes through a conditional, we save the conditional encountered in the form of relations between symbolic variables.

In concolic execution, we do:

1. execution with concrete inputs
2. while the execution
    1. we record the symbolic values of variables derived from inputs
    2. if possible we maintain path constraints of the executed path
3. after the execution is done we negate an if condition
4. re-execute with new concrete inputs
    1. to perform different paths
5. back to point 3

DYNAMIC VS STATIC ANALYSIS

DYNAMIC

For a dynamic analysis, we need to choose test input.

It is able to find bugs and vulnerabilities but cannot prove that they are absent.

STATIC

Can consider all possible inputs and can find bugs and vulnerabilities. In some cases can also prove the absence of bugs.