

A close-up, high-angle photograph of several interlocking metal gears. The gears are rendered in shades of light blue and grey, with soft lighting that creates highlights and shadows on their teeth and surfaces. The focus is sharp on the central gear, while the others are slightly blurred, creating a sense of depth and mechanical complexity.

# Program Analysis

Instructor: Martin Vechev

Lecture 8

# Rough Lecture Outline

(subject to change depending on how fast we move)

- ~~1. Introduction~~
- ~~2. Abstract Interpretation: Background~~
- ~~3. Numerical domains I: Intervals~~
- ~~4. Numerical domains II: Intervals/Octagons~~
- ~~5. Numerical domains III: Octagons/Pentagons~~
- ~~6. Applications of Numerical Domains: HPC/GPU~~
- ~~7. Applications of Numerical Domains: Semantic Differencing of Programs~~
8. SMT theories & Symbolic Execution
9. Predicate Abstraction and Concurrency
10. Program Synthesis
11. Race Detection: algorithms and complexity
12. Statistical Synthesis from “Big Code”
13. Probabilistic Program Analysis

# First order (SMT) theories

Restrictions on first order logic and the interpretation of formulas

These theories allow us to capture structures which are used by programs (e.g., arrays, integers, etc) and enable reasoning about them

Validity in first order logic (FOL) is undecidable, while validity in particular first order theories is (sometimes) decidable.

# First order (SMT) theories

- In every first order logical theory, the formulas in that theory are constructed with a specific set of function, predicate and constant symbols. This is the **signature** of the theory, called  $\Sigma$ .
- A first order formula in the theory is then built from the elements of  $\Sigma$  together with variables, logical connectives such as  $\wedge, \vee, \rightarrow, \neg$  and quantifiers  $\forall, \exists$
- Each theory comes with a set of **axioms** (FOL formulas), called  $A$ , which only contain elements from the signature. The predicates and functions in  $\Sigma$  have no meaning – their meaning is provided by the axioms in  $A$ .

# First order (SMT) theories

- A formula  $F$  in the theory is **valid** if all interpretations that satisfy the axioms in  $A$ , also satisfy the formula  $F$ .
- Some theories are meant to be used with a **particular interpretation**. For instance, in the theory of integers, the formulas are interpreted over the integers.
- A fragment of a theory consists of a subset of the possible formulas expressible in the theory (e.g., quantifier-free fragment).
- A theory is **decidable** if for every formula in the theory we can automatically check whether the formula is valid or not. Similarly for fragments of a theory.

# Decidability

It is useful if the fragment is decidable as we often need to decide whether a formula is valid or if two formulas are equivalent (for instance when performing fixed point computation).

Decidability is mainly needed to achieve 100% **automation**.

In principle, even if the theory is not decidable, we may still get lucky and the underlying theorem prover (e.g. **Z3**, **Yices**) which checks validity may succeed, but is **not guaranteed** to do so.

# SMT theories: Decidability

Theory	Description	Full Fragment	No Quantifiers
$T_E$	Equality	NO	YES
$T_{PA}$	Peano arithmetic	NO	NO
$T_N$	Presburger arithmetic	YES	YES
$T_Z$	Linear Integers	YES	YES
$T_R$	Reals (with *)	YES	YES
$T_Q$	Rationals (without *)	YES	YES
$T_{RDS}$	Recursive Data Structures	NO	YES
$T_{RDS}^+$	Acyclic Recursive Data Structures	YES	YES
$T_A$	Arrays	NO	YES
$T_A^=$	Arrays with extensionality	NO	YES

(source: The Calculus of Computation, Manna and Bradley)

# Theory of Equality

Signature  $\Sigma_E$  :

- Any function (uninterpreted), predicate and constant
- The predicate = which is **interpreted** (meaning defined via axioms)

Axioms  $A_E$  :

- $\forall x. x = x$
- $\forall x.y. x = y \rightarrow y = x$
- $\forall x.y.z. x = y \wedge y = z \rightarrow x = z$
- $\forall x_1 \dots x_n, y_1 \dots y_n. x_1 = y_1 \wedge \dots \wedge x_n = y_n \rightarrow f(x_1 \dots x_n) = f(y_1 \dots y_n)$  **functions**
- $\forall x_1 \dots x_n, y_1 \dots y_n. x_1 = y_1 \wedge \dots \wedge x_n = y_n \rightarrow p(x_1 \dots x_n) = p(y_1 \dots y_n)$  **predicates**

$T_E$  is undecidable

QFF  $T_E$  is decidable



# Theory of Equality: Example

$$a = b \wedge b = c \rightarrow g(f(a), b) = g(f(c), a)$$

Is this valid? If so, prove it

# Theory of Peano Arithmetic

Signature  $\Sigma_{PA}$  :

- Constants: 0, 1
- Binary Functions: + , \*
- Predicate: =

Axioms  $A_{PA}$  :

- $\forall x. \neg (x + 1 = 0)$
- $\forall x. x + 0 = x$
- $\forall x. x * 0 = 0$
- $\forall x.y. x + 1 = y + 1 \rightarrow x = y$
- $\forall x.y. x + (y + 1) = (x + y) + 1$
- $\forall x.y. x * (y + 1) = x * y + x$
- $F[0] \wedge (\forall x. F[x] \rightarrow F[x+1]) \rightarrow \forall x. F[x]$  (induction)

Interpreted over the **natural numbers** with the predicates and functions having the usual meaning (e.g., + over natural numbers).

# Theory of Peano Arithmetic

Is the formula  $3 * x + 2 = 2 * y$  in  $T_{PA}$  ?

# Theory of Peano Arithmetic

Is the formula  $3 * x + 2 = 2 * y$  in  $T_{PA}$  ?

Yes!

Can be written as:

$$(1 + 1 + 1) * x + 1 + 1 = y + y$$

**Exercise:** How would we express inequality  $>$  ?

# Theory of Peano Arithmetic

What about this one?

$$\forall x.y.z \quad x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \rightarrow x^n + y^n \neq z^n : n > 2 \wedge n \in \mathbb{Z}$$

# Theory of Peano Arithmetic

What about this one?

$$\forall x.y.z \quad x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \rightarrow x^n + y^n \neq z^n : n > 2 \wedge n \in \mathbb{Z}$$

$T_{PA}$  is undecidable

QFF  $T_{PA}$  is also undecidable

# Theory of Presburger Arithmetic

Signature  $\Sigma_N$  :

- Constants: 0, 1
- Binary Functions: +
- Predicate: =

NO MULTIPLICATION

Axioms  $A_N$  :

- $\forall x. \neg (x + 1 = 0)$
- $\forall x. x + 0 = x$
- $\forall x.y. x + 1 = y + 1 \rightarrow x = y$
- $\forall x.y. x + (y + 1) = (x + y) + 1$
- $F[0] \wedge (\forall x. F[x] \rightarrow F[x+1]) \rightarrow \forall x. F[x]$

(induction)

Interpreted over the **natural numbers** with the predicates and functions having the usual meaning (e.g., + over natural numbers).

$T_N$  is decidable

# Handling the Integers $\mathbb{Z}$ ?

In principle, it is possible to take a formula involving integers (and not just natural numbers) and subtraction and re-write it into a  $T_N$  formula.

However, a cleaner way is to directly focus on a theory which allows us to handle integers and subtraction.



# Theory of Integers

Signature  $\Sigma_Z$  :

- Constants: ..., -3, -2, 1, 0, 1, 2, 3, ...
- Unary functions: ..., -3\*, -2\*, 2\*, 3\*, ...
- Binary Functions: + and -
- Predicate: = and >

Interpreted over the **integers** with the predicates and functions having the usual meaning (e.g., + over the integers).

$T_Z$  is simply a convenient way to reason about addition over the integers.

$T_Z$  is decidable

# Theory of Integers

$$\forall x, y, z. \quad x > z \wedge y \geq 0 \rightarrow x + y > z$$

Is this formula valid?

How about this one?

$$\forall x, y. \quad x > 0 \wedge (x = 2y \vee x = 2y + 1) \rightarrow x - y > 0$$

# Theory of Integers

Here are some fragments of the theory of integers we looked at:

- Polyhedra
- Octagon
- Pentagon
- Interval

# Theory of Arrays

Signature  $\Sigma_A$  :

- Functions:
  - $\text{read}(\_, \_)$ , written for simplicity as  $\_ [\_]$ , e.g.,  $a[i]$
  - $\text{write}(\_, \_, \_)$  : e.g.,  $\text{write}(a, v, i)$  denotes the array  $a'$  where  $a'[v] = i$  and all other entries are the same as  $a$ .
- Predicate: =

Axioms  $A_A$  :

- Same as  $A_E$
- $\forall a.i.j. \ i = j \rightarrow a[i] = a[j]$
- $\forall a.v.i.j. \ i = j \rightarrow \text{write}(a, i, v)[j] = v$
- $\forall a.v.i.j. \ i \neq j \rightarrow \text{write}(a, i, v)[j] = a[j]$

Captures common operations on arrays such as reads and updates.

$T_A$  is undecidable  
QFF  $T_A$  is decidable

# Theory of Arrays

`a[i] = e → write(a,i,e) = a`

Is this formula valid?

# Theory of Arrays

$a[i] = e \rightarrow \forall j. \text{write}(a, i, e)[j] = a[j]$

What about this one?

# Combining Theories: The Nelson-Oppen method

In practice, the formulas arising in programs span multiple theories, not a single theory. For instance, reasoning about arrays of integers spans both theory of arrays and theory of integers. Note however how the signature of every theory has the equality predicate  $=$ .

Thus, to combine two (quantifier-free) theories  $T_1$  and  $T_2$  where  $\Sigma_1 \cap \Sigma_2 = \{=\}$ , that is, only  $=$  is shared, the combined theory of  $T_1$  and  $T_2$  is:

- $\Sigma_1 \cup \Sigma_2$
- $A_1 \cup A_2$

Further, if  $T_1$  and  $T_2$  are both decidable (+ some other requirements), then the combined theory is also decidable.

# Clients of SMT solvers

SMT solvers are widely used in many analyses, including synthesis (as in the project), verification, program analysis (discussed in later lectures) and symbolic execution.

Next, we will discuss one such client analysis: symbolic execution.



# Symbolic Execution: Applications

**Symbolic execution** is widely used in practice. Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:

- Network servers
- File systems
- Device drivers
- Unix utilities
- Computer vision code
- ...

# Symbolic Execution: Tools

- Stanford's KLEE:
  - <http://klee.llvm.org/>
- NASA's Java PathFinder:
  - <http://javapathfinder.sourceforge.net/>
- Microsoft Research's SAFE
- UC Berkeley's CUTE
- EPFL's S2E
  - <http://dslab.epfl.ch/proj/s2e>

# Symbolic Execution

At any point during program execution, symbolic execution keeps two formulas:

**symbolic store** and a **path constraint**

Therefore, at any point in time the **symbolic state** is described as the **conjunction** of these two formulas.

# Symbolic Store

- The values of variables at any moment in time are given by a function  $\sigma_s \in \text{SymStore} = \text{Var} \rightarrow \text{Sym}$ 
  - Var is the set of variables as before
  - Sym is a set of symbolic values
  - $\sigma_s$  is called a **symbolic store**
- Example:  $\sigma_s : x \mapsto x0, y \mapsto y0$

# Semantics

- Arithmetic expression evaluation simply manipulates the symbolic values.
- Let  $\sigma_s : x \mapsto x_0, y \mapsto y_0$
- Then,  $z = x + y$  will produce the symbolic store:  
 $x \mapsto x_0, y \mapsto y_0, z \mapsto x_0 + y_0$

That is, we literally keep the **symbolic expression**  $x_0 + y_0$

# Path Constraint

- The analysis keeps a path constraint (**pct**) which records the **history of all branches taken so far**. The path constraint is simply a formula.
- The formula is typically in a decidable logical fragment without quantifiers
- At the start of the analysis, the path constraint is **true**
- Evaluation of **conditionals** affects the path constraint , but not the symbolic store.

# Path Constraint: Example

Let  $\sigma_s : x \mapsto x_0, y \mapsto y_0$

Let  $pct = x_0 > 10$

Lets evaluate:  $\text{if } (x > y + 1) \{ 5: \dots \}$

At label 5, we will get the symbolic store  $\sigma_s$ . It does not change. But we will get an **updated path constraint**:

$$pct = x_0 > 10 \wedge x_0 > y_0 + 1$$

# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

Can you find the inputs that make the program reach the ERROR?

Lets execute this example with classic symbolic execution



# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

The read() functions read a value from the input and because we don't know what those read values are, we set the values of  $x$  and  $y$  to fresh symbolic values called  $x_0$  and  $y_0$

pct is true because so far we have not executed any conditionals

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$

pct : true

# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto 2*y_0$

pct : true

Here, we simply executed the function twice() and added the new symbolic value for z.

# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

We forked the analysis into 2 paths: the true and the false path. So we **duplicate** the state of the analysis.

This is the result if  $x = z$ :

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : x_0 = 2*y_0$$

This is the result if  $x \neq z$ :

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : x_0 \neq 2*y_0$$

# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

We can avoid further exploring a path if we know the constraint `pct` is **unsatisfiable**. In this example, both `pct`'s are **satisfiable** so we need to keep exploring both paths.

This is the result if `x = z`:

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : x_0 = 2*y_0$$

This is the result if `x != z`:

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : x_0 \neq 2*y_0$$

# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

Lets explore the path when  $x == z$  is true.  
Once again we get 2 more paths.

This is the result if  $x > y + 10$ :

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : \begin{array}{l} x_0 = 2*y_0 \\ \wedge \\ x_0 > y_0+10 \end{array}$$

This is the result if  $x \leq y + 10$ :

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : \begin{array}{l} x_0 = 2*y_0 \\ \wedge \\ x_0 \leq y_0+10 \end{array}$$

# Symbolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

So the following path reaches “**ERROR**”.

This is the result if  $x > y + 10$ :

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto 2*y_0$

$pct : x_0 = 2*y_0$   
 $\wedge$   
 $x_0 > y_0 + 10$

We can now ask the SMT solver for a satisfying assignment to the pct formula.

For instance,  $x_0 = 40, y_0 = 20$  is a satisfying assignment. That is, running the program with those concrete inputs triggers the error.

# Handling Loops: a limitation

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < k; i++)  
        sum += i;  
    return sum;  
}
```

A serious limitation of symbolic execution is handling unbounded loops. Symbolic execution runs the program for a finite number of paths. But what if we do not know the bound on a loop ? The symbolic execution will keep running **forever** !

# Handling Loops: bound loops

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < 2; i++)  
        sum += i;  
    return sum;  
}
```

A **common solution in practice** is to provide some loop bound. In this example, we can bound  $k$ , to say 2. This is an example of an **under-approximation**. Practical symbolic analyzers usually under-approximate as most programs have unknown loop bounds.



# Handling Loops: loop invariants

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < k; i++)  
        sum += i;  
    return sum;  
}
```

loop invariant



Another solution is to provide a **loop invariant**, but this technique is rarely used for large programs because it is **difficult to provide** such invariants **manually** and it can also lead to **over-approximation**. This is where a combination with static program analysis is useful (static analysis can infer loop invariants). We will not study this approach in our treatment, but we note that the approach is used in program verification.

# Constraint Solving: challenges

Constraint solving is fundamental to symbolic execution as a constraint solver is continuously invoked during analysis. Often, the main **roadblock to performance** of symbolic execution engines is the time spent in constraint solving. Therefore, it is important that:

1. The SMT solver supports as many decidable logical fragments as possible. Some tools use more than one SMT solver.
2. The SMT solver can solve large formulas quickly.
3. The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

# Key Optimization: Caching

The basic insight here is that often, the analysis will invoke the SMT solver with **similar formulas**. Therefore, the symbolic execution system can keep a **map (cache)** of formulas to a satisfying assignment for the formula.

Then, when the engine builds a new formula and would like to find a satisfying assignment for that formula, it can first access the cache, **before calling** the SMT solver.

# Key Optimization: Caching

Suppose the cache contains the mapping:

$$\begin{array}{ll} \text{Formula:} & \text{Solution:} \\ (x + y < 10) \wedge (x > 5) & \rightarrow \{x = 6, y = 3\} \end{array}$$

If we get a weaker formula as a query, say  $(x + y < 10)$ , then we can immediately reuse the solution already found in the cache, without calling the SMT solver.

If we get a stronger formula as a query, say  $(x + y < 10) \wedge (x > 5) \wedge (y \geq 0)$ , then we can quickly try the solution in the cache and see if it works, without calling the solver (in this example, it works).

# When Constraint Solving Fails

Despite best efforts, the program may be using constraints in a fragment which the SMT solver does not handle well.

For instance, suppose the SMT solver does not handle **non-linear constraints** well.

Let us consider a modification of our running example.

# Modified Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

Here, we changed the `twice()` function to contain a **non-linear** result.

Let us see what happens when we symbolically execute the program now...

# Modified Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

This is the result if  $x = z$ :

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto y_0 * y_0$

$pct : x_0 = y_0 * y_0$

Now, if we are to invoke the SMT solver with the pct formula, it would be **unable** to compute satisfying assignments, precluding us from knowing whether the path is feasible or not.

# Solution: Concolic Execution

**Concolic Execution:** combines **both** symbolic execution and concrete (normal) execution.

The basic idea is to have the concrete execution drive the symbolic execution.

Here, the program runs as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information.



# Concolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

The read() functions read a value from the input. Suppose we read  $x = 22$  and  $y = 7$ .

We will keep both the concrete store and the symbolic store and path constraint.

$\sigma : x \mapsto 22,$   
 $y \mapsto 7$

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$

pct : true

# Concolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

$\sigma : x \mapsto 22,$   
 $y \mapsto 7,$   
 $z \mapsto 14$

$\sigma_s : x \mapsto x0,$   
 $y \mapsto y0$   
 $z \mapsto 2*y0$

pct : true

The concrete execution will now take the 'else' branch of  $z == x$ .

# Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Hence, we get:

$\sigma : x \mapsto 22,$   
 $y \mapsto 7,$   
 $z \mapsto 14$

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto 2*y_0$

$\text{pct} : x_0 \neq 2*y_0$

# Concolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

At this point, concolic execution decides that it would like to explore the “true” branch of  $x == z$  and hence it needs to generate concrete inputs in order to explore it. Towards such inputs, it negates the pct constraint, obtaining:

pct :  $x_0 = 2 * y_0$

It then calls the SMT solver to find a satisfying assignment of that constraint. Let us suppose the SMT solver returns:

$x_0 \mapsto 2, y_0 \mapsto 1$

The concolic execution then runs the program with this input.

# Concolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

With the input  $x \mapsto 2, y \mapsto 1$  we reach this program point with the following information:

$\sigma : x \mapsto 2,$   
 $y \mapsto 1,$   
 $z \mapsto 2$

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto 2*y_0$

$\text{pct} : x_0 = 2*y_0$

Continuing further we get:

# Concolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

We reach the “else” branch of  $x > y + 10$

$$\sigma : \begin{array}{l} x \mapsto 2, \\ y \mapsto 1, \\ z \mapsto 2 \end{array}$$
$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$
$$\text{pct} : \begin{array}{l} x_0 = 2*y_0 \\ \wedge \\ x_0 \leq y_0+10 \end{array}$$

Again, concolic execution may want to explore the ‘true’ branch of  $x > y + 10$ .

# Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

We reach the “else” branch of  $x > y + 10$

$$\sigma : x \mapsto 2, \\ y \mapsto 1, \\ z \mapsto 2$$
$$\sigma_s : x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0$$
$$\text{pct} : x_0 = 2*y_0 \\ \wedge \\ x_0 \leq y_0+10$$

Concolic execution now negates the conjunct  $x_0 \leq y_0+10$  obtaining:

$$x_0 = 2*y_0 \quad \wedge \quad x_0 > y_0+10$$

A satisfying assignment is:  $x_0 \mapsto 30, y_0 \mapsto 15$

# Concolic Execution: Example

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

If we run the program with the input:

$x_0 \mapsto 30, y_0 \mapsto 15$

we will now reach the **ERROR** state.

As we can see from this example, by keeping the symbolic information, the concrete execution can use that information in order to obtain new inputs.



# Non-linear constraints

Let us return to the problem of **non-linear constraints**

# Non-linear constraints

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Let us again consider our example and see what concolic execution would do with non-linear constraints.

# Concolic Execution: Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

The read() functions read a value from the input. Suppose we read  $x = 22$  and  $y = 7$ .

$\sigma : x \mapsto 22,$   
 $y \mapsto 7$

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$

pct : true

# Concolic Execution: Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

$\sigma : x \mapsto 22,$   
 $y \mapsto 7,$   
 $z \mapsto 49$

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto y_0 * y_0$

pct : true

The concrete execution will now take the 'else' branch of  $x == z$ .

# Concolic Execution: Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

Hence, we get:

$\sigma : x \mapsto 22,$   
 $y \mapsto 7,$   
 $z \mapsto 49$

$\sigma_s : x \mapsto x_0,$   
 $y \mapsto y_0$   
 $z \mapsto y_0 * y_0$

$\text{pct} : x_0 \neq y_0 * y_0$

# Concolic Execution: Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x, y);
}
```

However, here we have a non-linear constraint  $x_0 \neq y_0 * y_0$ . If we would like to explore the true branch we negate the constraint, obtaining  $x_0 = y_0 * y_0$  but again we have a **non-linear constraint** !

In this case, concolic execution simplifies the constraint by plugging in the concrete values for  $y_0$  in this case, 7, obtaining the simplified constraint:

$$x_0 = 49$$

Hence, it now runs the program with the input

$$x \mapsto 49, \quad y \mapsto 7$$

# Concolic Execution: Example

```
int twice(int v) {
    return v * v;
}

void test(int x, int y) {
    z = twice(y);
    if (x == z) {
        if (x > y + 10)
            ERROR;
    }
}

int main() {
    x = read();
    y = read();
    test(x,y);
}
```

Running with the input

$x \mapsto 49, \quad y \mapsto 7$

will reach the error state.

However, notice that with these inputs, if we try to simplify non-linear constraints by plugging in concrete values (as concolic execution does), then concolic execution we will never reach the else branch of the `if (x > y + 10)` statement.

# Symbolic Execution vs. Abstract Interpretation


Is Symbolic Execution an instance of Abstract Interpretation?

For instance, is SE an abstract interpreter over the abstract domain of logical formula, where we do not perform joins?



# SE vs. Abstract Interpretation

Question on the A.I. Facebook group: <https://www.facebook.com/groups/abstract.interpretation/>


**Anitha Gollamudi** March 26 at 1:36pm


This has been eating me for sometime - is concolic execution static or dynamic analysis? I have heard it being pushed to static side... How do we draw a line between static and dynamic analysis?


Like · Comment · Share


---


✓ Seen by 52

**Gautam Krishna** Static.. March 26 at 1:38pm · Like

**Anitha Gollamudi** why? March 26 at 4:50pm · Edited · Like

**Roberto Giacobazzi** I think it is a kind of lower approximation made over the concrete domain in a (static) symbolic way! March 27 at 3:33am · Like

**Johannes Kinder** I don't think "static analysis" and "dynamic analysis" have a formal meaning - these terms usually only refer to whether a program is actually executed on a real system or only reasoned about. It's more useful to think in terms of under- and over-approximation: as Roberto correctly points out, concolic execution (which is a particular strategy of symbolically executing a program) is an under-approximation. But to answer your original question: tools that perform concolic execution (like KLEE, SAGE, S2E, etc.) typically execute the program under test, therefore you could say they perform a dynamic analysis. March 27 at 11:17am · Like · 1

**Anitha Gollamudi** That helps. March 28 at 8:25pm · Like

# Summary

- Symbolic Execution is a popular technique for analyzing large programs
  - completely automated, relies on SMT solvers
- To terminate, may need to bound loops
  - leads to under-approximation
- To handle non-linear constraints and external environment, mixes concrete and symbolic execution (called concolic execution)
  - also leads to under-approximation