# C07 – Symbolic execution

Program Verification

FMI · Denisa Diaconescu · Spring 2025

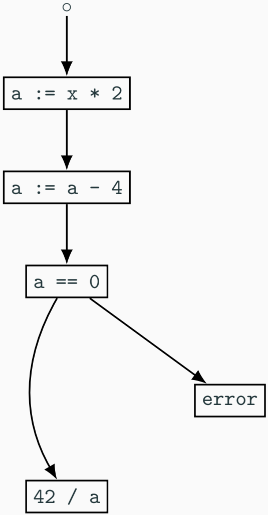
# Symbolic execution

# Symbolic execution

- Symbolic execution is widely used in practice.
- Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:
  - Networks servers
  - File systems
  - Device drivers
  - Unix utilities
  - Computer vision code
  - ...

# Control Flow Graph

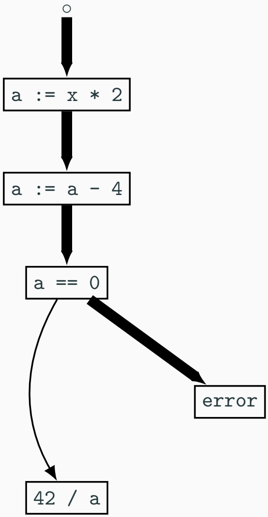
```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



# Path Feasibility

A path is feasible if there exists an input $\mathcal{I}$ to the program that covers the path (when program is executed with $\mathcal{I}$ as input, the path is taken).

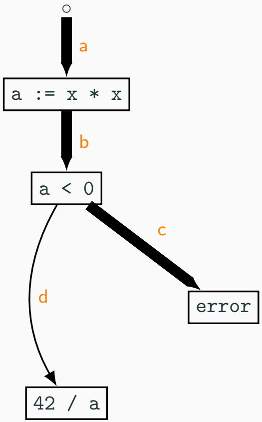
```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



# Infeasible path

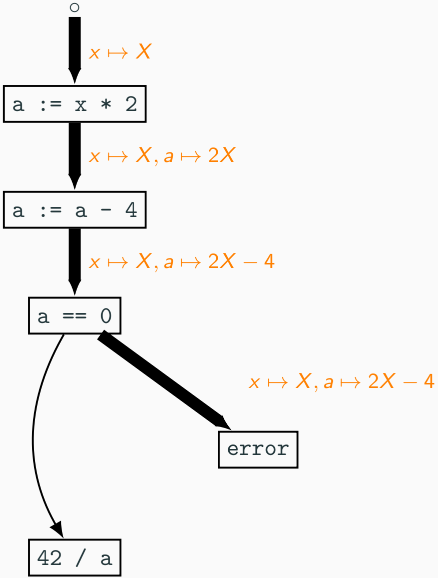
A path is infeasible if there exists no input $\mathcal{I}$ that covers the path.

```
int f(int x) {
  int a = x * x;
  if (a < 0)
    error("Too small!");
  return 42 / a;
}
```



# Symbols

```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



# Symbolic execution

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

- symbolic store and
- path constraint

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

# Symbolic store

The value of variables at any moment in time are given by a function

$$\sigma_s : Var \to Sym$$

- *Var* is the set of variables
- *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.

**Example**
Suppose the symbolic store is $\sigma_s$ : `x` $\mapsto$ `x0`, `y` $\mapsto$ `y0`.

Then `z = x + y` will produce the new symbolic store

$$\sigma_s' \ : \ \texttt{x} \mapsto \texttt{x0},\ \texttt{y} \mapsto \texttt{y0},\ \texttt{z} \mapsto \texttt{x0 + y0}$$

We literally keep the symbolic expression `x0 + y0`.

# Path constraint

- Path constraint is a condition on the input symbols such that if a path is feasible its path-constraint is satisfiable.
- 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**
Suppose the symbolic store is $\sigma_s$ `:  x` $\mapsto$ `x0, y` $\mapsto$ `y0`.

Suppose the path constraint is `pct = x0 > 10`.

Let us evaluate `if(x > y + 1) {5: ...}`

At label 5, we will get the symbolic store $\sigma_s$. It does not change!

But, at label 5, we will get an updated path constraint:

$$\texttt{pct = x0 > 10} \land \texttt{x0 > y0 + 1}$$

# Constraint solver

A constraint solver is a tool that finds satisfying assignments for a *constraint*, if it is satisfiable.

A solution of the constraint is a set of assignments, one for each free variable that makes the constraint satisfiable.

# 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

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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 `x0` and `y0`

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

$\sigma_s$ : x ↦ x0,
y ↦ y0

pct : true

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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 ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : true

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

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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:

$\sigma_s$ : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 = 2*y0

This is the result if x != z:

$\sigma_s$ : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 ≠ 2*y0

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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:

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

pct : $x0 = 2*y0$

This is the result if x != z:

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

pct : $x0 \neq 2*y0$

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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:

$\sigma_s$ : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 = 2*y0 ∧ x0 > y0+10

This is the result if x ≤ y + 10:

$\sigma_s$ : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 = 2*y0 ∧ x0 ≤ y0+10

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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` ↦ `x0`, `y` ↦ `y0`, `z` ↦ `2*y0`

pct : `x0 = 2*y0` $\wedge$ `x0 > y0+10`

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

For instance, `x0 = 40, y0 = 20` is a satisfying assignment. That is, running the program with those concrete inputs triggers the error.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Test generation

- We can use symbolic execution to generate tests for each feasible path
- For example, a symbolic execution tool Klee

# Test generation - example

```
#include <climits>
#include "stdlib.h"
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    exit(-1);
  return 42 / a;
}
int main(int argc, char** argv) {
  int x = atoi(argv[1]);
  return f(x);
}
```

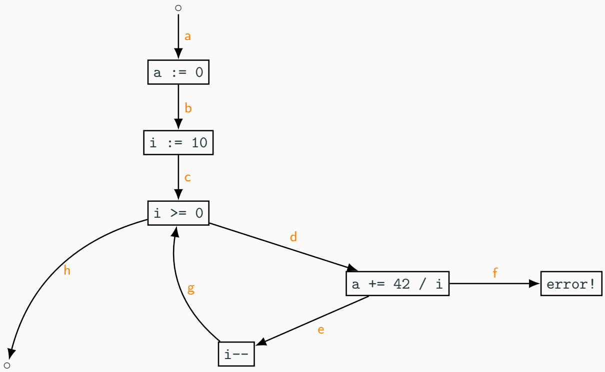
```
#include <climits>
#include "stdlib.h"
#include "klee/klee.h"
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    exit(-1);
  return 42 / a;
}
int main(int argc, char** argv) {
  int x;
  klee_make_symbolic(&x, sizeof(x), "x");
  return f(x);
}
```

Klee will generate two tests for `x = 0` and for `x = 2`.

# Path explosion

Path explosion refers to the fact that the number of control-flow paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations.

```
int a = 0;
for (int i = 10; i >= 0; i--) {
  a += 42 / i;
}
```



# 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 happens 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 the above 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.
- This technique is rarely used for large programs because it is difficult to provide such invariants manually.
- It can also lead to over-approximation.

# Constraint solving - challenges

- Constraint solving is fundamental to symbolic execution.
- An SMT solver is continuously invoked during analysis.
- Often, the main roadblock to performance of symbolic execution engines is the time spent in constraint solving.
- Important features:
  - The SMT solver supports as many decidable logical fragments as possible.
    - Some tools use more than one SMT solver.
  - The SMT solver can solve large formulas quickly.
  - The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

# Key optimization - caching

- The analyzer will invoke the SMT solver with similar formulas.
- The symbolic execution engine can keep a map (cache) of formulas to a satisfying assignment for the formulas.
- 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

**Example**
Suppose the cache contains the mapping:

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

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 example, the SMT solver does not handle non-linear constraints well.

# When constraint solving fails - 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...

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# When constraint solving fails - 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 ↦ x0,
y ↦ y0
z ↦ y0*y0

pct : x0 = y0*y0

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.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Concolic execution

# Concolic execution

$\boxed{\text{concolic} = \text{concrete} + \text{symbolic}}$

- Combines both symbolic execution and concrete (normal) execution.
- The basic idea is to have the concrete execution drive the symbolic execution.
- The programs run 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.

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

$\sigma_s : x \mapsto x0,$
$\quad y \mapsto y0$

pct : true

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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 ↦ 22,
y ↦ 7,
z ↦ 14

$\sigma_s$ : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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 the 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 : $x0 = 2*y0$

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

$$x0 \mapsto 2, \quad y0 \mapsto 1$$

The concolic execution then runs the program with this input.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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 ↦ 2, y ↦ 1 we reach this program point with the following information:

$\sigma$ : x ↦ 2,
y ↦ 1,
z ↦ 2

$\sigma_s$ : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0

Continuing further we get:

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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

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

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

**pct** $: x0 = 2*y0 \wedge x0 \leq y0+10$

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

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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

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

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

$$\text{pct} : x0 = 2*y0 \wedge x0 \leq y0+10$$

Concolic execution now negates the conjunct $x0 \leq y0+10$ obtaining:

$$x0 = 2*y0 \wedge x0 > y0+10$$

A satisfying assignment is: $x0 \mapsto 30,\ y0 \mapsto 15$

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# 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:

$$x0 \mapsto 30,\ y0 \mapsto 15$$

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.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Non-linear constraints - 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);
}
```

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

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Non-linear constraints - 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 $\mapsto$ x0,
y $\mapsto$ y0

pct : true

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Non-linear constraints - 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);
}
```

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

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

pct : true

The concrete execution will now take the ‘else’ branch of x == z.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Non-linear constraints - 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 $\mapsto$ x0,
y $\mapsto$ y0
z $\mapsto$ y0*y0

pct : x0 $\neq$ y0*y0

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Non-linear constraints - 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 `x0 ≠ y0*y0` . If we would like to explore the true branch we negate the constraint, obtaining `x0 = y0*y0` but again we have a non-linear constraint !

In this case, concolic execution simplifies the constraint by plugging in the concrete values for `y0` in this case, 7, obtaining the simplified constraint:

$$x0 = 49$$

Hence, it now runs the program with the input

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

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

# Non-linear constraints - 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.

*source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.*

## Symbolic execution

- Is a popular technique for analyzing 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 (concolic execution).

# References


- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emina Torlak.
- "Program Analysis Crash Course", Yegor Bugayenko