Symbolic Execution

✔ Function: The goal of this function is to compute values based on a condition and return results depending on whether certain criteria are met.

✔ Path Formula for Path 1 (Branch 1 True, Branch 2 True):

Conditions like $(y_0 = x * 2) \wedge (y_0 > 10) \wedge (y_1 = y_0 + 5) \wedge (y_1 \% 3 == 0)$ represent the conditions along this path.

✔ SMT Solver Formula (Z3):

The SMT solver is used to check if this path is satisfiable and provides values for x, y_0, and y_1.

Code	Path Formula	SMT Solver Formula (Z3)	Output from get-model

int processValue(int x) {
int y = x * 2; // Assignment 1: $y_0 = x_0 * 2$
if (y > 10) { // Branch 1: Check y_0
y = y + 5; // Assignment 2: $y_1 = y_0 + 5$
if (y % 3 == 0) { // Branch 2: Check y_1
return 1;
} else {
return 2;
}
} else {
y = y - 10; // Assignment 3: $y_2 = y_0 - 10$
return 3;
}
}
	# Choose Path 1 as example: Branch 1 True, Branch 2 True, we have
	$(y_0 = x * 2) \wedge$
	$(y_0 > 10) \wedge$
	$(y_1 = y_0 + 5) \wedge$
	$(y_1 \% 3 == 0)$
		(declare-const x Int)
		(declare-const y_0 Int)
		(declare-const y_1 Int)
		(assert (= (* x 2) y_0))
		(assert (> y_0 10))
		(assert (= (+ y_0 5) y_1))
		(assert (= (mod y_1 3) 0))
		(check-sat)
		(get-model)
			sat
			(
			(define-fun y_0 () Int 16)
			(define-fun x () Int 8)
			(define-fun y_1 () Int 21)
			)

2 Loop Example: sumWithLimit Function

✔ Function: This function sums x for up to 3 iterations, exiting early if the sum exceeds 10.

✔ Path Formula for Iteration 3: The formula checks the sum at each step and determines if the sum exceeds 10.

Code	Path Formula	SMT Solver Formula (Z3)	Output from get-model

int sumWithLimit(int x) {
int y = 0;
for (int i = 0; i < 3; i++) { // Loop runs at most 3 times
y += x;
if (y > 10) {
break; // Early exit if y exceeds 10
}
}
return y;
}
 	
# Unroll with SSA. Assume a fixed bound if unsure how many iterations 	
int sumWithLimit(int x) {
int y_0 = 0; // Initial assignment
int i_0 = 0;
// Iteration 1
if (i_0 < 3) {
$y_1 = y_0 + x$; // $y_1 = x$
if (y_1 > 10) return y_1; // Exit early
$i_1 = i_0 + 1$; // $i_1 = 1$
// Iteration 2
if (i_1 < 3) {
$y_2 = y_1 + x$; // $y_2 = 2x$
if (y_2 > 10) return y_2;
$i_2 = i_1 + 1$; // $i_2 = 2$
// Iteration 3
if (i_2 < 3) {
$y_3 = y_2 + x$; // $y_3 = 3x$
if (y_3 > 10) return y_3;
$i_3 = i_2 + 1$; // $i_3 = 3$ (loop ends)
}
}
}
return y_3; // Returns 3x if loop completes
}
	# Path formula for iteration 3.
	$y_0 = 0 \wedge$
	$y_1 = y_0 + x \wedge$
	$y_2 = y_1 + x \wedge$
	$y_3 = y_2 + x \wedge$
	$i_0 = 0 \wedge$
	$i_1 = i_0 + 1 \wedge$
	$i_2 = i_1 + 1 \wedge$
	$i_3 = i_2 + 1 \wedge$
	$i_0 < 3 \wedge$
	$i_1 < 3 \wedge$
	$i_2 < 3 \wedge$ //
	always hold
	$y_3 > 10$
		# SMT formula
		(declare-const x Int)
		(declare-const y_0 Int)
		(declare-const y_1 Int)
		(declare-const y_2 Int)
		(declare-const y_3 Int)
		(declare-const i_0 Int)
		(declare-const i_1 Int)
		(declare-const i_2 Int)
		(declare-const i_3 Int)
		(assert (= 0 y_0))
		(assert (= 0 i_0))
		(assert (= (+ y_0 x) y_1))
		(assert (= (+ y_1 x) y_2))
		(assert (= (+ y_2 x) y_3))
		(assert (= (+ i_0 1) i_1))
		(assert (= (+ i_1 1) i_2))
		(assert (= (+ i_2 1) i_3))
		(assert (> y_3 10))
		(check-sat)
		(get-model)
			# Sample output
			sat
			(
			(define-fun x () Int 4)
			(define-fun y_3 () Int 12)
			(define-fun y_1 () Int 4)
			(define-fun i_1 () Int 1)
			(define-fun y_0 () Int 0)
			(define-fun i_3 () Int 3)
			(define-fun i_2 () Int 2)
			(define-fun y_2 () Int 8)
			(define-fun i_0 () Int 0)
			)

3 Example with Arrays: analyzeArray Function

✔ Function: This function checks for negative values in an array and exits early if found. If the sum exceeds 20, it returns 1, otherwise 0.

✔ Symbolic Variables: arr[0], arr[1], arr[2] are replaced with symbolic variables like A_0, A_1, A_2.

```
int analyzeArray(int arr[3]) {
int sum = 0;
for (int i = 0; i < 3; i++) {
if (arr[i] < 0) { // Early exit for negative values
return -1;
}
sum += arr[i];
}
if (sum > 20) {
return 1;
} else {
return 0;
}
}
# Replace arr[0], arr[1], arr[2] with symbolic variables say A₀, A₁, A₂
```

4 Bitvector Example: sumThree Function

✔ Function: This function sums three 32-bit integers and checks if the result exceeds 100.

✔ SMT Formula for Bitvector Handling:

✔ The symbolic variables a, b, and c are treated as 32-bit bitvectors, and their sum is computed.

# Verification field also prefers bit vectors. More from memory perspective 	
int sumThree(int a, int b, int c) {
int total = a + b + c;
if (total > 100) {
return 1;
} else {
return 0;
}
}
int sumThree(int a_0, int b_0, int c_0)
{
int $total_0 = a_0 + b_0 + c_0$; // Bitvector addition (32-bit)
if ($total_0$ > 100) { // Bitvector comparison
return 1;
} else {
return 0;
}
}
	# SMT formula
	(declare-const a (_ BitVec 32)) ; 32-bit symbolic variables
	(declare-const b (_ BitVec 32))
	(declare-const c (_ BitVec 32))
	; Compute total = a + b + c (32-bit addition with potential overflow)
	(define-fun total () (_ BitVec 32) (bvadd a (bvadd b c)))
	; Path constraint: total > 100 (interpreted as unsigned/signed comparison)
	(assert (bvsgt total (_ bv100 32))) ; Signed comparison
	(check-sat)
	(get-model) ;
		# Sample output
		sat
		(
		(define-fun a () (_ BitVec 32)
		#x00000065)
		(define-fun total () (_ BitVec 32)
		(bvadd a b c))
		(define-fun b () (_ BitVec 32)
		#x00000000)
		(define-fun c () (_ BitVec 32)
		#x00000000)
		)

Propositional Logic

1 What is Propositional Logic?

Propositional logic deals with atomic propositions (statements that are either true or false) and their combinations using logical connectives (AND, OR, NOT, IMPLIES).

2 Syntax of Propositional Logic:

Atomic Propositions:

These are basic statements that are either true or false.

Example:

P: "It is raining."

Q: "The door is open."

Connectives:

¬P: NOT P (negation)

P∧Q: P AND Q

P∨Q: P OR Q

P→Q: P IMPLIES Q (equivalent to ¬P ∨ Q)

⊕: XOR (exclusive or)

↔: Biconditional (if and only if, necessary and sufficient)

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$	$P \rightarrow Q$
T	T	T	T	F	T
T	F	F	T	F	F
F	T	F	T	T	T
F	F	F	F	T	T

3 Examples:

1. P → Q: "If it is raining, then the ground is wet."
2. (A ∧ D) → I: "If the alarm rings and the door is open, then there is an intruder."
   Where: A = "Alarm rings" D = "Door is open", I = "Intruder"

3. ¬(C ∧ I): "You cannot have both cake and ice cream."
   Where: C = "Cake", I = "Ice cream"

4 Calculus:

✔ Rules of Inference:

A function that takes premises, analyzes their syntax, and returns a conclusion.

(Premise 1), (Premise 2) ⊢ (Conclusion)

"Given Premise 1 and Premise 2, we can derive Conclusion."

✔ Example:

- Implication Elimination (Modus Ponens):
  P → Q, P ⊢ Q
  If P → Q (if P, then Q) and P (P is true), then Q follows.
- Biconditional Introduction:
  (P → Q), (Q → P) ⊢ (P ↔ Q)
  If P → Q and Q → P, then P ↔ Q (P if and only if Q).
- Conjunction Introduction:
  P, Q ⊢ (P ∧ Q)
  If both P and Q are true, then P ∧ Q is true (both P and Q).
- Conjunction Elimination:
  (P ∧ Q) ⊢ P or (P ∧ Q) ⊢ Q
  From P ∧ Q, we can infer P or Q individually.
- Constructive Dilemma:
  (P → Q), (R → S), (P ∨ R) ⊢ (Q ∨ S)
  If P → Q, R → S, and P ∨ R, then Q ∨ S follows.
- Hypothetical Syllogism:
  P → Q, Q → R ⊢ P → R
  If P → Q and Q → R, then P → R.
- Absorption:
  P → Q ⊢ P → (P ∧ Q)
  If P → Q, then P → (P ∧ Q) is true (proof by absorption).

✔ Rules of Replacement:

- Associativity:
  (P ∨ (Q ∨ R)) ⇔ ((P ∨ Q) ∨ R)
  The order of grouping in an OR operation doesn't affect the result. This also applies to conjunctions (AND operation).
- Commutativity:
  (P ∨ Q) ⇔ (Q ∨ P)
  The order of the operands in OR (and similarly for AND) doesn't change the result. For instance, P ∨ Q is the same as Q ∨ P.
- Distributivity:
  (P ∨ (Q ∧ R)) ⇔ ((P ∨ Q) ∧ (P ∨ R))
  Distributivity shows how AND distributes over OR and vice versa. For example, (P ∧ (Q ∨ R)) can be rewritten as ((P ∧ Q) ∨ (P ∧ R)).
- De Morgan's Laws:
  ¬(P ∧ Q) ⇔ (¬P ∨ ¬Q)
  These laws provide a way to distribute the negation across conjunctions and disjunctions. For example, the negation of P ∧ Q is equivalent to ¬P ∨ ¬Q, and the negation of P ∨ Q is equivalent to ¬P ∧ ¬Q.
- Material Implication:
  P → Q ⇔ ¬P ∨ Q

This rule replaces an implication (P → Q) with a disjunction (¬P ∨ Q). In other words, "P implies Q" is logically equivalent to "Either P is false, or Q is true."

- Exportation:
  ((P ∧ Q) → R) ⇔ (P → (Q → R))

$$\frac{(P \land Q) \to R}{P \to (Q \to R)}, \quad \frac{P \to (Q \to R)}{(P \land Q) \to R}$$

This rule transforms implications. The first form shows how P ∧ Q → R can be re-written as P → (Q → R), which helps simplify the expression.

- Tautology:
  P ∧ P ⇔ P
  P ∨ P ⇔ P

Tautology states that P ∧ P is just P, and similarly, P ∨ P is just P. These are self-evident truths.

Predicate Logic (First Order Logic)

What is Predicate Logic? Extends propositional logic with Quantifiers (∀, ∃), Variables (x, y, z), Predicates (e.g., x > 5, x is a parent of y)

✔ Syntax:

Variables: Represent objects (e.g., x, y, z)

Predicates: Describe properties (e.g., P(x), Q(x, y))

Quantifiers:

∀x P(x): "For all x, P(x) is true"

∃x P(x): "There exists an x such that P(x) is true"

✔ Example:

Student(x) means "x is a student."

∀x (Student(x) → Studies(x)): "All students study."

Even numbers: "Every even number greater than 2 is the sum of two primes."

⚖️ Rules:

Universal Generalization: If something is true for one, it's true for all.

Existential Generalization: If something is true for some, it exists in the domain.

✔ Propositional vs Predicate Logic:

Propositional Logic: Deals with simple true/false values.

Predicate Logic: Models more complex relationships with variables and quantifiers.

📌 Predicate Logic helps us reason about relationships between variables using quantifiers and predicates. It's more powerful than propositional logic and is used in mathematics and logic for detailed reasoning.

✔ Applications of Predicate Logic:

Puzzle Solving: Solving puzzles like Sudoku and N-Queens using logical reasoning.

Hardware Verification: Check circuit/chip designs to ensure they work as expected, including efforts in quantum computing.

Dynamic Execution: Generate test cases for all possible program paths to ensure full coverage during testing.

Program Verification: Prove code correctness, such as ensuring there's no division by zero or other runtime errors.

Program Verification with SAT/SMT

Program Verification ensures a program meets its specifications (e.g., no division by zero, array bounds safety). SAT/SMT solvers help automate this process.

How SAT/SMT Helps:

Encoding Programs:

Treat inputs as symbolic variables.

Use symbolic execution to check possible execution paths.

Verification Conditions:

Encode conditions (e.g., no division by zero) into formulas for SAT/SMT solvers to verify.

Division by Zero

```
int divide(int a, int b) {
  if (b != 0) return a / (b + 1);
  else return b / a;
}
```

```
(declare-const a Int)
(declare-const b Int)

(define-fun branch1 () Bool (and (not (= b 0)) (= (+ b 1) 0)))
(define-fun branch2 () Bool (and (= b 0) (= a 0)))

(assert (or branch1 branch2))
(check-sat)
(get-model)
```

Array Bounds Check

```
int access(int[] arr, int idx) {
  return arr[idx];
}
```

```
(declare-const idx Int)
(declare-const arr_size Int)

(assert (>= idx 0))
(assert (< idx arr_size))
(check-sat)
```

Buffer Overflow Check

#include <string.h>
void processInput(const char* input) {
  char buffer[16];
  strcpy(buffer, input);
}
	(declare-const N Int)
	(declare-const input (Array Int (_ BitVec 8)))
	
	(define-fun no_null_in_buffer () Bool
	  (forall ((i Int)) (=> (and (>= i 0) (< i 16))
	  (not (= (select input i) #x00))))
	)
	
	(assert (> N 16))
	(assert (and no_null_in_buffer (> N 16)))
	(check-sat)
	(get-model)

Loop Invariant

✔ A loop invariant is a condition that remains true before and after every iteration of a loop. It acts as a "checkpoint" to help verify the correctness of the loop.

✔ Importance of Loop Invariants

Critical Systems: In systems like flight control, invariants help ensure that the loop behaves correctly. For example, Boeing and Airbus use formal methods, including invariants, to verify that flight control loops behave correctly under all conditions.

Failure Impact: Failing to maintain loop invariants can lead to issues such as sensor faults, control failures, or safety risks.

✔ Properties of Loop Invariants

Initialization:

The invariant must be true before the loop starts.

Example: Before starting the loop to sum numbers, the sum is zero.

Maintenance (Preservation):

If the invariant holds at the start of an iteration, it must remain true after the iteration.

Example: As the sum increases, it must always equal the sum of the numbers processed so far.

Termination:

When the loop ends, the invariant, together with the loop's exit condition, should imply the desired postcondition.

Example: After the loop finishes, the sum should equal the sum of numbers from 1 to n.

✔ How to Derive Loop Invariants

1. Understand the Goal of the Loop:

   Identify what the loop should guarantee when it finishes (the postcondition).

   Example: In summing numbers, the goal is for the sum to equal 1 + 2 + ... + n after the loop.
2. Work Backwards from the Postcondition:

   Derive intermediate relationships. For example, after processing the first i numbers, the sum should be 1 + 2 + ... + i.
3. Express the Invariant Clearly:

   Write the invariant that maintains consistency at each iteration.

   Example: For summing numbers, the invariant could be that after the i-th iteration, the sum equals 1 + 2 + ... + i.
4. Verify Initialization:

   Ensure the invariant holds before the loop starts.
5. Check Maintenance:

   After each iteration, ensure the invariant is still true.

   For example, adding the current number i to the sum ensures the invariant remains true.
6. Verify Termination:

   Combine the invariant with the loop's exit condition to guarantee the desired postcondition.

   Example: After the loop terminates, the sum should be the total of numbers from 1 to n.

Hoare Logic

✔ Hoare Logic is a formal system used to verify the correctness of computer programs. It allows reasoning about the program's behavior using preconditions and postconditions.

✔ Hoare Triple is represented as {P} C {Q}, where:

P is the precondition: a logical assertion that must hold before the program runs.

C is the program (or command) to be verified.

Q is the postcondition: a logical assertion that must hold after the program runs, assuming it terminates.

Logic Rules:

1. Assignment Rule: {Q[e/x]} x := e {Q}
   {y + 1 = 5} x := y + 1 {x = 5}
2. Sequence Rule: {P} C1 {R}, {R} C2 {Q} → {P} C1; C2 {Q}
   {x = 3} y := x + 2; z := y * 2 {z = 10}
3. Conditional: {P ∧ e ≠ 0} C1 {Q}, {P ∧ e = 0} C2 {Q} → {P} if e then C1 else C2 {Q}
   {true} if (x > 0) y := x else y := -x {y ≥ 0}
4. Loop: {I ∧ e ≠ 0} C {I}, {I} while e do C {I ∧ e = 0}
   while (i < n) { sum += a[i]; i++; }

```
bool login(char* pwd) {
if (strcmp(pwd, "secret") == 0) return true;
else return false;
}
Hoare Triple:
{true} login(pwd) {return_value = true ↔ pwd = "secret"}.
```

// Pre: 0 ≤ index < array_size
void safe_write(int* array, int index, int value) {
array[index] = value;
}
// Post: array[index] == value ∧ array unchanged elsewhere
	# Z3
	(declare-const array (Array Int Int))
	(declare-const index Int)
	(declare-const value Int)
	(declare-const array_size Int)
	; Precondition
	(assert (and (>= index 0) (< index array_size)))
	; Postcondition
	(define-fun post () Bool
	(and (= (select (store array index value) index) value) ; Updated index
	(forall ((i Int))
	(=> (not (= i index))
	(= (select (store array index value) i) (select array i))))) ;
	Unchanged elsewhere
	)
	(assert (not post))
	(check-sat) ; unsat → postcondition holds

int n = 5; // Input
int i = n;
int result = 1;
// Compute factorial: result = n!
while (i > 1) {
result *= i;
i--;
}
 	
// {result == 120}
	// Precondition: requires { n >= 0 } (same as the spec).
	// Postcondition: ensures { result = fact(n) } (must match the spec).
	let compute_factorial (n: int) : (result: int)
	requires { n >= 0 } // Precondition: n is non-negative
	ensures { result = fact(n) } // Postcondition: Result matches math definition
	=

module FactorialExample
use int.Int
use int.Fact
let rec function fact (n: int) : int
requires { n >= 0 }
ensures { result = fact(n) }
=
if n = 0 then 1 else n * fact(n - 1)
	let ref i = n in
	let ref result = 1 in
	while i > 1 do
	invariant { result * fact(i) = fact(n) }
	// Loop invariant
	invariant { i >= 0 } // Safety invariant
	variant { i } // Termination variant
	result <- result * i;
	i <- i - 1;
	done;
	result

Temporal Logic

✔ Basic Temporal Operators

Path Quantifiers:

A: "All paths" (universally quantified)

E: "Exists a path" (existentially quantified)

Temporal Operators:

G: "Globally" (always)

F: "Future" (eventually)

X: "Next" (in the next state)

U: "Until"

W: "Weak until"

R: "Release"

✔ Basic State Formulas:

1. AG φ: In all paths, φ is always true.

Example: AG(temperature < 100) — Temperature is always below 100 in all possible futures.

2. EG φ: There exists a path where φ is always true.

Example: EG(connection_active) — There's at least one future where the connection stays active forever.

3. AF φ: In all paths, φ eventually becomes true.

Example: AF(process_terminates) — The process will eventually terminate in all possible futures.

4. EF φ: There exists a path where φ eventually becomes true.

Example: EF(resource_available) — It's possible to reach a state where the resource becomes available.

5. AX φ: In all paths, φ is true in the next state.

Example: AX(acknowledgment_sent) — In the next step/state, the acknowledgment will be sent regardless of what happens.

6. EX φ: There exists a path where φ is true in the next state.

Example: EX(task_scheduled) — It's possible that in the next step, the task will be scheduled.

✔ Until Operators:

1. A[φ U ψ]: In all paths, φ holds until ψ becomes true.

Example: A[waiting U served] — System keeps waiting until it is eventually served in all paths.

2. E[φ U ψ]: There exists a path where φ holds until ψ becomes true.

Example: E[processing U completed] — There's a possible path where processing continues until completion.

✔ Complex Nested Formulas:

1. AG(EF φ): From any state, it's always possible to eventually reach φ.

Example: AG(EF restart_possible) — System can always eventually be restarted from any state.

2. EF(AG φ): There exists a path to a state after which φ always holds.

Example: EF(AG stable_state) — System can reach a permanently stable state.

3. AF(EG φ): All paths lead to a state from which there's a path where φ always holds.

Example: AF(EG operational) — System eventually reaches a state from which it can stay operational.

4. EG(AF φ): There exists a path where φ is repeatedly satisfied.

Example: EG(AF process_completed) — There's a path where processes keep completing infinitely often.

✔ Combinations with Multiple Properties:

1. AG(φ → AF ψ): Always, if φ occurs, ψ will eventually follow.

Example: AG(request → AF response) — Every request is eventually followed by a response.

2. AG(φ → EF ψ): Always, if φ occurs, ψ is possible in the future.

Example: AG(error → EF recovered) — After any error, recovery is possible.

3. EF(φ ∧ EG ψ): Can reach a state where φ is true and ψ can hold forever after.

Example: EF(initialized ∧ EG running) — Can reach an initialized state and possibly run forever.

4. AG(φ → AX ψ): Always, if φ occurs, ψ follows in the next state.

Example: AG(commit → AX consistent) — After every commit, the next state is always consistent.

✔ Equivalences and Relationships:

Future and Until:

AF φ ≡ A[true U φ]

EF φ ≡ E[true U φ]

Globally and Future:

AG φ ≡ ¬EF ¬φ

EG φ ≡ ¬AF ¬φ

Real-World Temporal Logic Examples

1. Mutual Exclusion Properties

✔ No two processes in critical section simultaneously:

AG ¬(process1.critical ∧ process2.critical)

This ensures that no two processes are in the critical section at the same time.

✔ If a process requests, it will eventually enter the critical section:

AG(process.requesting → AF process.critical)

This guarantees that a requesting process will eventually gain access to the critical section.

2. Elevator System Properties

✔ The elevator will eventually service all requests:

AG(floor_request → AF floor_serviced)

This ensures that all floor requests will eventually be serviced by the elevator.

✔ The door is never open while the elevator is moving:

AG(door_open → ¬elevator_moving)

This ensures that the elevator door cannot be open while the elevator is moving.

✔ When the elevator is called, it eventually arrives:

AG(button_pressed → AF elevator_arrived)

This ensures that when a button is pressed, the elevator will eventually arrive at the requested floor.

3. File System Properties

✔ A file cannot be read and written simultaneously:

AG ¬(file.reading ∧ file.writing)

This ensures that a file cannot be in both reading and writing states at the same time.

✔ After closing a file, it's eventually available for opening:

AG(file.closed → AF file.available)

This ensures that once a file is closed, it will eventually become available for opening.

✔ If a write operation starts, it eventually completes:

AG(write_started → AF write_completed)

This ensures that any started write operation will eventually complete.

4. Network Protocol Properties

✔ If a message is sent, it's eventually received or an error occurs:

AG(message_sent → AF(message_received ∨ error_state))

This ensures that any sent message will either be received or result in an error.

✔ The connection is never in connected and disconnected states simultaneously:

AG ¬(connection.connected ∧ connection.disconnected)

This ensures that the connection cannot be both connected and disconnected at the same time.

✔ After a timeout, the system eventually retries or fails:

AG(timeout → AF(retry ∨ failure))

This ensures that after a timeout, the system will either retry or fail eventually.