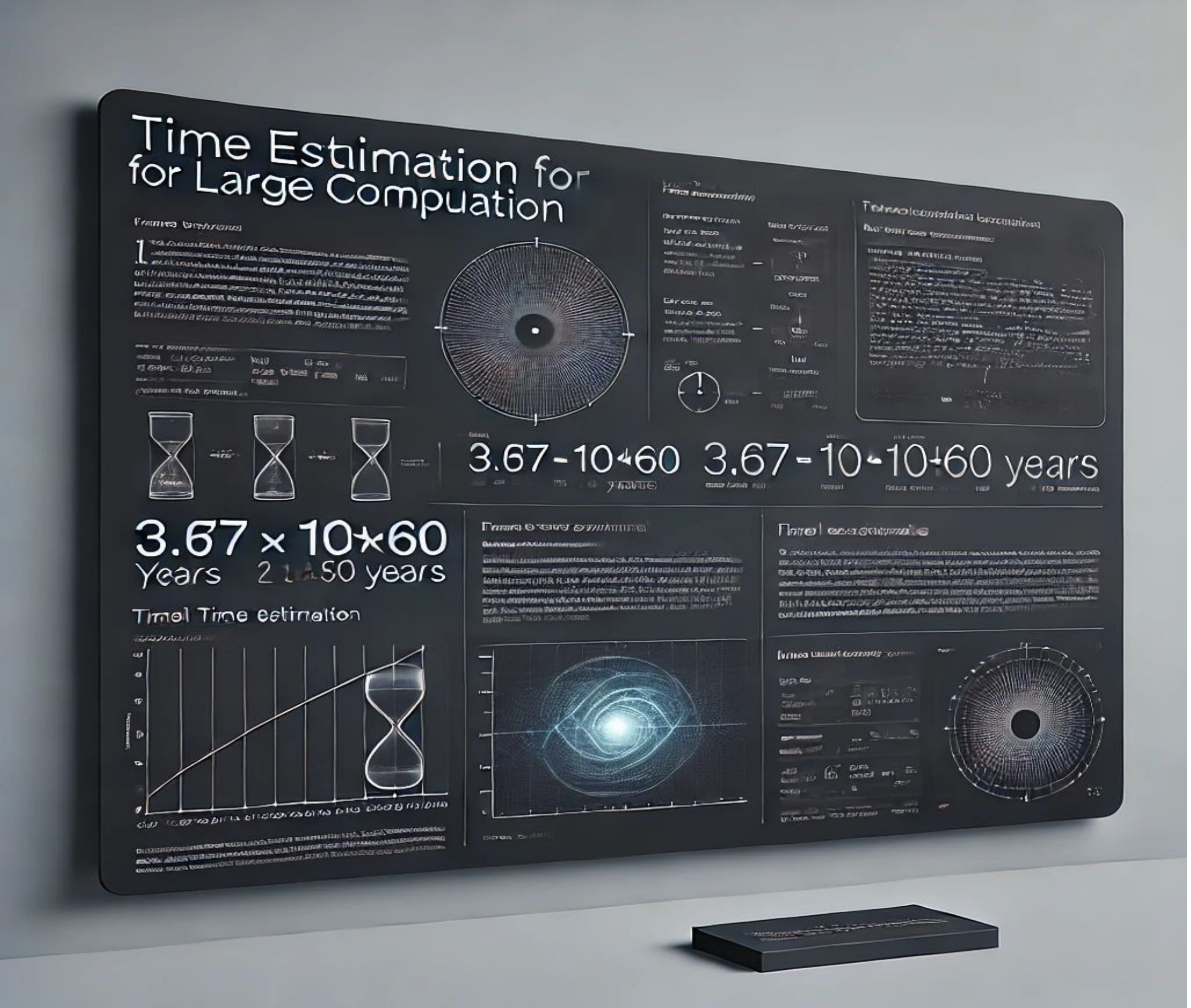
Internal Mechanisms

of Formal Verification

How long would it take to run ?

```
pragma solidity >=0.8.17;


contract Loop {
    function long() pure public {
        uint init = 0;
        uint end  = type(uint).max;
        for(uint x = init; x < end; x++) {
        }
    }
}
```



1 interaction = 1 nanosec

1 year = 31,536,000 sec

3.67 x 10^60 years

13.8 billion years

What is testing and why we need

- Process of evaluating software to find defects
- Verifies that software meets requirements
- Ensures software behaves as expected
- Includes various types: unit, integration, system, etc.

- Improves software quality and reliability
- Identifies and prevents bugs early
- Enhances user satisfaction and trust
- Reduces long-term costs and maintenance
- Ensures compliance with standards and regulations

Invariants

An **invariant** is a condition or property that remains constant and unchanging throughout the execution of a system or during specific operations. It helps maintain the integrity and expected behavior of a system.

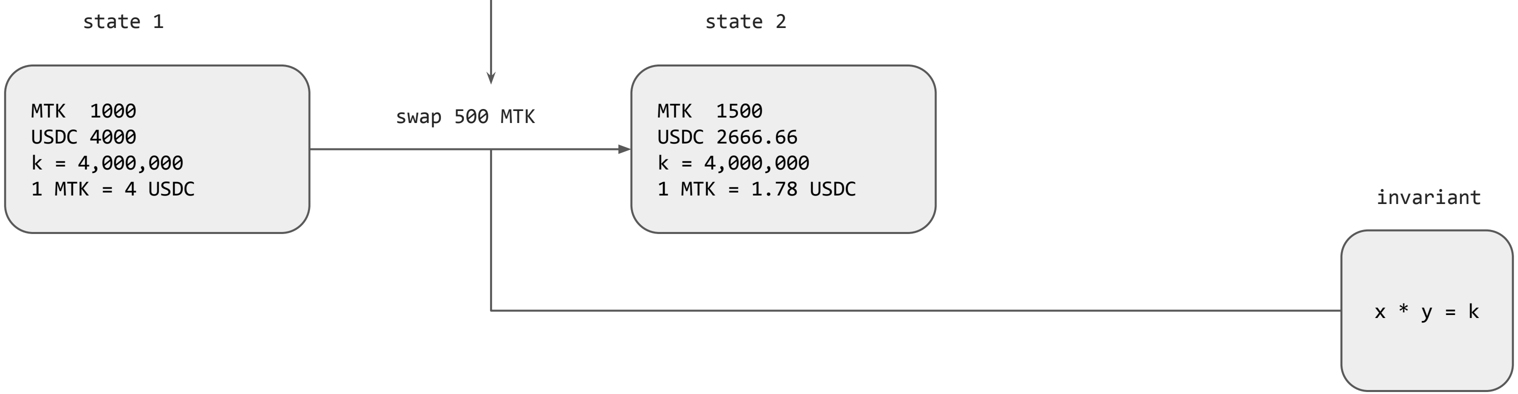
1. **Liquidity Pool Invariant**
   - In AMMs like Uniswap, the product of the quantities of two tokens in a pool remains constant. For tokens A and B, the invariant is x⋅y=kx \cdot y = kx⋅y=k, where xxx and yyy are the quantities, and kkk is a constant.
2. **Collateralization Ratio Invariant**
   - In lending protocols like MakerDAO, the value of collateral must always exceed a specific ratio relative to the value of the loan, ensuring that collateral is sufficient to cover the loan.
3. **Total Supply Invariant**
   - In ERC20 tokens, the total supply of tokens remains constant unless explicitly changed. The sum of all account balances equals the total supply.
4. **Transaction Fee Invariant**
   - In some DeFi protocols, the transaction fee is a fixed percentage of the transaction amount. This invariant ensures that the fee structure remains consistent and predictable.
5. **Governance Voting Invariant**
   - In decentralized governance systems, the total number of votes cast must equal the number of participants who voted. This invariant ensures accurate representation and counting of votes.

State space and Invariants

```
function swap(
        address recipient,
        bool zeroForOne,
        int256 amountSpecified,
        uint160 sqrtPriceLimitX96,
        bytes calldata data
    ) external returns (int256 amount0,
                        int256 amount1)
```

$$2^{160+1+256+160+256} = 2^{833}$$

Testing the invariant for all possible state space (2^833 combinations) would take by far more than the age of the universe.



Aspect	Unit Test	Fuzz Test	Static Test	Formal Verification
Focus	Tests individual functions	Tests with random inputs	Analyzes code without execution	Proves correctness mathematically
Automation	Highly automated	Automated but requires input generation	Automated	Partially automated, requires manual setup
Coverage	Specific code paths	Unpredictable coverage	Broad but non-executable aspects	Comprehensive but theoretical
Complexity	Simple	Moderate	Simple to moderate	High
Bugs Detected	Functional bugs	Edge cases, security flaws	Syntax, style, certain logical errors	Logical correctness, security proofs
Execution	Code execution required	Code execution required	No execution required	No execution required
Use Case	Everyday development	Security-critical software	Early-stage development	Mission-critical systems
Resources Required	Low	Moderate	Low to moderate	High (time, expertise)

Proving the absence of bugs

Formal verification tools use *mathematical* representations
to check whether a given invariant holds
for **all** input values and <u>states</u> of the system.

Dafny automatic verification

```
method exec1_forward(x: int) returns (y: int)
requires 10 <= x
ensures  25 <= y {          // 10 <= x
    var a := x + 3;         // 10 <= x && a == x + 3
    var b := 12;            // 10 <= x && a == x + 3 && b == 12
    y := a + b;
}

// 10 <= x && a == x + 3 && b == 12 && y == a + b ==> 25 <= y
```

Implication

The implication statement is only a false implication if the "if" statement is true and the "then" statement is false. If we start with a false statement we can imply anything, because the full statement will never be fulfilled.

⊃ or > = 'If ... then ...' = Implication

p: it's raining
q: I have an umbrella

```
p ⊃ q
T T T - if it's raining     then  I have an umbrella        ⊃: T [since both statements are true]
T F F - if it's raining     then  I don't have an umbrella  ⊃: F [the 'if' is true and 'then' is false]
F T T - if it's not raining then  I have an umbrella        ⊃: T [the 'if' is false, so we imply anything]
F T F - if it's not raining then  I don’t have an umbrella  ⊃: T [the 'if' is false, so we imply anything]
```

Dafny's Internal Verification Tools

1. SMT Solver (Z3)
- Translates program and specs into logical formulas
- Feeds formulas to Z3 SMT solver
- Z3 searches for spec-violating counterexamples
- Verification successful if no counterexample found

2. Static Analysis
- Gathers info about variables and relationships
- Tracks assignments, propagates constants, identifies invariants

3. Symbolic Execution
- Executes program with symbolic variable values
- Reasons about all possible execution paths

4. Verification Condition Generation
- Generates verification conditions (VCs) at each program point
- VCs encode conditions for program correctness

5. Axiom Instantiation
- Uses built-in axioms and theories (e.g., arithmetic)
- Instantiates axioms as needed during proof process

6. Proof Obligation Discharge
- Attempts to prove each verification condition
- Combines facts, applies logic rules, uses SMT solver

Lean4 semi automatic verification [tactics]

```
example {a b : ℝ} (h1: a - 5*b = 4) (h2: b + 2 = 3) : a = 9 :=
calc
    a = a                   := by ring
    _ = a - 5*b + 5*b := by ring
    _ = 4         + 5*b := by rw[h1]
    _ = -6 + 5 * (b+2):= by ring
    _ = -6 + 5 * 3      := by rw[h2]
    _ = -6 + 15           := by ring
    _ = 9                   := by ring
```

https://blog.trailofbits.com/2024/03/22/why-fuzzing-over-formal-verification/

When to Consider Formal Verification

- **Invariant-Driven Development:** Ensure you're following this approach.
- **Thorough Fuzz Testing:** Many invariants have been tested with fuzzing.
- **Targeted Application:** You know which remaining invariants/components need formal methods.
- **Code Maturity:** All other issues affecting code maturity are resolved.

Key Insight:

- **Focus on Invariants:** Writing good invariants is 80% of the work.
  The tools for verification are secondary.
- **Start with Fuzzing:** Begin with the easiest and most effective technique.
  Use formal methods only when necessary.

Certora

```
rule transferSpec {
	address sender; address recip; uint amount;
	env e;
	require e.msg.sender == sender;

	mathint balance_sender_before = balanceOf(sender);
	mathint balance_recip_before = balanceOf(recip);

	transfer(e, recip, amount);

	mathint balance_sender_after = balanceOf(sender);
	mathint balance_recip_after = balanceOf(recip);

	require sender != recip;

	assert balance_sender_after == balance_sender_before - amount,
	"transfer must decrease sender's balance by amount";
	assert balance_recip_after == balance_recip_before + amount,
	"transfer must increase recipient's balance by amount";
}
```