

# CERTORA

Move fast and break nothing

# TYPES OF PROPERTIES

# INTRODUCTION

To make the process of property creation easier, we can look at the system from different points of view. In our case, these will be our types of properties.

**We split properties into 5 main types:**





- Valid States
- State Transitions
- Variable Transitions
- High-Level Properties
- Unit Tests

💡 Thought of the day: in order to create a good property, you need to think like a good property.

# VALID STATES

Systems can often be thought of as state machines. The states define the possible values that the system's variables can take. We call these Valid States.

**For example, a system can be in one of the following states:**

-  **doesn't exist**
-  **was created**
-  **is active**
-  **is finished**

# VALID STATES

Developers often rely on the valid states implicitly or explicitly in the system's workflow. Therefore, unintentional behavior can occur if the system can break out of the intended valid states, potentially resulting in devastating bugs.

Usually, there can be only **one valid state** at any given time. Thus, we also check that a system must always be in exactly one of its valid states.



# VALID STATES EXAMPLE - ONE STATE AT A TIME

```
definition meetingPending(bytes32 meetingId) returns bool =
    getMeetingId(meetingId) != 0 &&
    getStartTime(meetingId) != 0 &&
    getEndTime(meetingId) != 0 &&
    getNumOfParticipents(meetingId) == 0 &&
    getStatus(meetingId) == 1;

definition meetingStarted(bytes32 meetingId) returns bool =
    getMeetingId(meetingId) != 0 &&
    getStartTime(meetingId) != 0 &&
    getEndTime(meetingId) != 0 &&
    getNumOfParticipents(meetingId) == 0 &&
    getStatus(meetingId) == 2;

invariant oneStateAtATime(bytes32 meetingId)
    meetingPending(meetingId) && !meetingStarted(meetingId)
    || !meetingPending(meetingId) && meetingStarted(meetingId)
```

- 💡 This is a simplified version of the example code. There are 5 valid states in the system. The inv will be fully correct only if we extend it to include all combinations of the 5 states.

# STATE TRANSITIONS

We also verify the correctness of transitions between valid states. Firstly, we confirm that the **valid states change according to their correct order** in the state machine. Then, we verify that the **transitions only occur under the right conditions**, like calls to specific functions or time elapsing.

```
rule checkUninitializedToPending(method f, uint256 meetingId){
  env e;
  calldataarg args;
  uint stateBefore = getStateById(e, meetingId);
  f(e, args);
  uint stateAfter = getStateById(e, meetingId);

  assert (stateBefore == 0 => (stateAfter == 1 || stateAfter == 0));
  assert ((stateBefore == 0 && stateAfter == 1) => f.selector == scheduleMeeting(uint256, uint256, uint256).selector);
}
```



# VARIABLE TRANSITIONS

Additionally, we can verify the change and validity of variables. Some variables are designed to change in a certain way through the entire system's life cycle (**monotonicity**), while others can change in any direction.

As a simple example, If we had a variable that counts the number of transactions that have ever took place in a system, it would've had to change in a non-decreasing manner throughout the system's life.

# VARIABLE TRANSITIONS EXAMPLE

We can check that after calling [deposit](#), the balance of all users and the total balance of the system shouldn't decrease.

```
rule depositIncreaseOnly(uint256 amount, env e){
    uint256 userBalanceBefore = getUserBalance(e.msg.sender);
    uint256 systemTotalBefore = getTotalSupply();

    deposit(amount);

    uint256 userBalanceAfter = getUserBalance(e.msg.sender);
    uint256 systemTotalAfter = getTotalSupply();

    assert userBalanceBefore <= userBalanceAfter, "user's balance was decreased";
    assert systemTotalBefore <= systemTotalAfter, "systems's balance was decreased";
}
```

# HIGH-LEVEL PROPERTIES

Probably the most powerful type of properties is the **high-level** property.

It doesn't cover any tangible part of the system like the aforementioned types (state, variable, or transition). However, it does try to cover the whole system from the users' point of view.

## HIGH-LEVEL PROPERTIES EXAMPLE

Let's take **Bank** as an example. We can think of the system as an interaction between a client and the bank itself.

A high-level property for a client would be:

**“If a client makes any operation within the bank system (currency conversion, transfer between accounts, etc.), the total balance of all clients' accounts should remain the same”.**

This property makes sure that **no assets are disappearing or being created out of nowhere** (unintentionally). We often call this property solvency.

## HIGH-LEVEL PROPERTIES EXAMPLE

This is a simplified example based on a [simple bank system](#):

The balance of any single (arbitrary) user should be no more than the total funds of the bank.

```
invariant oneIsNotMoreThanAll(address user)  
    getUserFunds(user) <= getTotalFunds()
```

getUserFunds() should equal totalFunds only if a single user legitimately owns the entire funds in the bank. In any other case, getUserFunds() must be smaller than totalFunds.

# UNIT TEST

Like in QA, unit tests target specific functions (or a part of them) individually to check if they work as intended.

**Example 1:** A transfer within a system should increase the recipient's balance by a specified amount and decrease the sender's balance by the same amount.

**Example 2:** In ERC20, if `increasedAllowance()` was called, the allowance of spender by owner increased exactly by the specified amount.

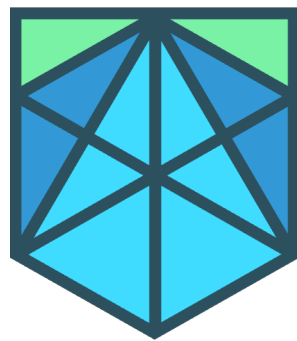
# UNIT TEST

In the example below, based on [OpenZeppelin's ERC20](#), we check that allowance was increased correctly.

```
rule increaseAllowanceUnitTest(address spender, uint256 amount){
  env e;

  uint256 allowanceBefore = allowance(e, e.msg.sender, spender);
  increaseAllowance(e, spender, amount);
  uint256 allowanceAfter = allowance(e, e.msg.sender, spender);

  assert allowanceAfter == allowanceBefore + amount,
    "the allowance of the owner to the spender changed by a wrong amount";
}
```



THANK YOU