State Variable Annotations

Properties that hold when state variables are written to State variable specifications allow us to describe how a smart contract is allowed to change state variables and under which circumstances.

This allows us to specify generically how all functions of the smart contract should be allowed to modify state. We highlight two use cases to demonstrate how this enables effective specification writing:

Access Control

You can write access control specifications usingState Specifications . For example, we can define properties that say "only the owner can change this". Checking this property will ensure that none of the functions in the smart contract allow a non-owner to change that variable.

We discuss this example in greater detail below

Integrity

UsingState Specifications we can also make sure that the integrity of the state is maintained by all the functions in the smart contract.

For example, we can say that some variables shouldn't be changed, which we can use to state that thetotalBalance of a non de-/inflationary token should stay constant. If we accidentally make the mint function of a contract public and unprotected, then this property would be violated.

Variable Updates

Theif_updated specification enables properties that will be checked at any point where a variable is updated. These properties follow the following structure and are placed above state variable declarations

```

Copy if_updated {:msg ""} ;

```

Example 1 Example 2 Example 3 The following property states that a variable shouldn't be updated.

~~~

Copy contract Example { /// #if_updated {:msg "the totalBalance is constant"} false; uint256 totalBalance; }

``` In this property we make sure that only the owner can change this variable.

This property ensures only the owner can mint tokens! ```

Copy contract Example is Ownable { /// #if_updated "only the owner can change this" msg.sender == owner; uint256 totalBalance; }

``` This property ensures that you can only increase the value of a variable.

~~~

Copy contract Example { /// #if_updated {:msg "the value increases"} old(counter) < counter; uint256 counter; }

```

The Ownable case study

Sometimes it's more convenient to reason about properties of the contract's state variable. For example, consider theOwnable contract below:

```

Copy contract Ownable { address owner; constructor() { owner = msg.sender; }

modifier onlyOwner { assert(msg.sender == owner); _; }

function transferOwnership(address newOwner) onlyOwner public { owner = newOwner; } }

```
```

You may want to express the following property:

The value ofowner may only be set at contract creation, OR changed by the current owner by callingtransfer

Note that we want this property to also hold for all contracts that inherit fromOwnable . You can do this by adding anif_succeeds annotation on all functions OTHER thantransferOwnership . However, this is fragile; what if we miss one?

We currently cannot express this as a contract-wide invariant, as those don't supportold() and don't (yet) support temporal predicates. Even if we specified this as a contract invariant, a function inside the contract can temporarily re-assignonlyOwner and then restore it before returning.

Fundamentally, we want our property to be checkedevery time owner is modified, which is precisely what you can do with the newif_updated annotation:

```

Copy /// #if_updated /// (msg.sig == 0x0 && owner == msg.sender) || /// ( /// msg.sig == && /// msg.sender == old(owner) && /// owner == newOwner /// ); address owner

```

There are several things to unpack here.

Theif_updated annotation appears right above the declaration of theowner variable. This property will ensure that the condition followingif_updated will be checked on every write to the owner variable. This condition has two clauses that describe the conditions where the owner variable is allowed to be updated.

Clause 1

The first clausemsg.sig == 0x0 && owner == msg.sender describes the case when we are in the constructor (in that casemsg.sig == 0x0), and states that in that case,after the assignmentowner must equal to themsg.sender (i.e., the contract creator).

Clause 2

The second clause specifies several things:a) we need to be in a call totransferOwnership (which is specified bymsg.sig ==)b) The caller must be the current owner (msg.sender == old(owner)). Note the use, ofold() to refer to the value ofowner before the assignmentc) The new value ofowner must be what was passed in totransferOwnership (owner == newOwner).

The two clauses together precisely specify the property that we had in mind in the beginning, without any manual annotations required at every location whereowner is modified

In Summary:

1. if_updated
2. annotations appaer in docstrings right before avariable declaration
3. In anif_updated
4. annotation you can useold()
5. to refer to the value of an expression before the assignment.
6. Scribble automatically finds all places where the variable is modified and inserts the check there.
7. Note that "variable modification" is not just limited to assignments. A variable may be also modified due todelete
8. operations, unary increment/decrement operations (++
9. /--
10.), etc.
11. Scribble can only instrument state variables that are not aliased. A state variable is aliased if it (or any part of it) is assigned to a storage pointer variable on the stack. For example in the below codearr
12. is aliased, and cannot be annotated:
13.

```

Copy contract Aliased { uint[] arr; function main() public { uint[] storage ptr = arr; } }

```

Previous Function Annotations Next Contract Invariants Last updated2 years ago

On this page * Variable Updates * The Ownable case study

Was this helpful?