Instrumented Code

While our intention is that you should not have to look at or understand instrumented code,scribble is still beta software and bugs will happen. In such cases it may be useful for you to understand how Scribble generates instrumented code. For this we will look at two examples.

Function Annotations

To understand function instrumentation lets look at the following (slightly contrived) example:

```

Copy contract Foo { int x; /// #if_succeeds {:msg "test"} x == old(x) + y; function add(int y) public { require(y > 0); x += y; } }

```

The instrumented code generated would look like this:

```

Copy contract Foo { event AssertionFailed(string message);

struct vars0 { int256 old_0; }

int internal x;

function add(int y) public { vars0 memory _v; _v.old_0 = x; _original_Foo_add(y); if ((!((x == (_v.old_0 + y))))) { emit AssertionFailed("0: test"); assert(false); } }

function _original_Foo_add(int y) private { require((y > 0)); x += y; } }

```

There are several important things to note:

1. Scribble interposes on functions by renaming them, making them private, and adding a new stub with the same signature as the original function. For example, note that the original code from theadd
2. function now lives in the new private function_original_Foo_add
3. (lines 20-23).
4. In the generated wrapperadd()
5. we invoke the renamed original function (line 13), and check theif_succeeds
6. annotation after the call (lines 14-17).
7. The values ofold
8. expressions are evaluated before the call (line 12) and require a temporary variable. However, instead of just adding multiple temporary variables, we group all of the temporary variables needed inside of a single struct (vars0
9.) defined on lines 4-6. This is done to reduce the extra stack space used by instrumented code.
10. Which custom property failed is signaled by emitting theAssertionFailed
11. event (line 15).
12.

Property Maps

When the above property is violated, a message with the string0: test (line 15) is emitted. Thetest part is the original user-readable label in the annotation.0 is an internally assigned id. Scribble assigns such an id for each annotation we write. It is possible to hint Scribble to also output a metadata with map from ids to the original property. See more about the metadata format in sectionJSON output specification .

There are two ways to get the instrumentation metadata with Scribble:

- Using--output-mode json
- (or-m json
-) command line option as part of JSON output ininstrumentationMetadata
- key.
- Using--instrumentation-metadata-file path/to/meta-file.json
- command line option to get it in a standalone JSON file at specified path.
-

Contract Annotations

For contract invariants there are several additional changes. Let's consider this slightly different example:

```

Copy /// #invariant {:msg "foo"} x>=1; contract Foo { int x; constructor() public { x = 1; }

function add(uint x) public { x += 1; } }

```

Now the emitted code is quite a bit more involved:

```

Copy /// Utility contract holding a stack counter contract __scribble_ReentrancyUtils { bool __scribble_out_of_contract = true; }

contract Foo is __scribble_ReentrancyUtils { event AssertionFailed(string message);

struct vars1 { bool __scribble_check_invs_at_end; }

int internal x;

constructor() public { __scribble_out_of_contract = false; x = 1; __scribble_check_state_invariants(); __scribble_out_of_contract = true; }

function add(uint x) public { vars1 memory _v; _v.__scribble_check_invs_at_end = __scribble_out_of_contract; __scribble_out_of_contract = false; _original_Foo_add(x); if (_v.__scribble_check_invs_at_end) __scribble_check_state_invariants(); __scribble_out_of_contract = _v.__scribble_check_invs_at_end; }

function _original_Foo_add(uint x) private { x += 1; }

/// Check only the current contract's state invariants function __scribble_Foo_check_state_invariants_internal() internal { if ((!((x >= 1)))) { emit AssertionFailed("0: foo"); assert(false); } }

/// Check the state invariant for the current contract and all its bases function __scribble_check_state_invariants() virtual internal { __scribble_Foo_check_state_invariants_internal(); } }

```

Again there are several changes:

1. A new helper contract -__scribble_ReentrancyUtils
2. (lines 2-4) was added. This contract was also added as a base contract ofFoo
3. . (line 6).__scribble_ReentrancyUtils
4. contains a single boolean property that is used for tracking whether the current stack frame is the root frame of an external call. This is useful as public functions can be called both internally and externally, but we only want to check contract invariants in them when they are called externally.
5. A new function__scribble_Foo_check_state_invariants_internal
6. was added (lines 36-41) which actually checks whether the invariant is true. It is only called indirectly, by__scribble_check_state_invariants
7. which is itself called by the wrapped functions.
8. The publicadd
9. function (lines 22-33) was wrapped similarly to functions in the previous section. The wrapper (lines 22-29) invokes the__scribble_check_state_invariants
10. function at the end of execution (line 27) to check whether the invariant is preserved by the call toadd
11. .
12. The check on line 27 is done conditionally. This is a twist to make sure we are not checking contract invariants inside of public functions when the public functions are called internally. Ifadd
13. was declaredexternal
14. then the call on line 27 would be unconditional.
15. All external, public functions in the contract (and the constructor) receive additional code at their beginning and end that manipulates the__scribble_out_of_contract
16. state variable defined in the contract. This is used to keep track of whether we are just entering the contract or not. The logic is particularly involved for public functions. Note that in the public functionadd
17. , whether we check the invariants at the end depends on the value of__scribble_out_of_contract
18. at the start of the function (which was stored in_v.__scrible_check_invs_at_end
19. on line 24). This reflects the intuition that we want to check contract invariants only at the end of external calls, not internal calls (for more details on contract invariants seeContract Invariants
20.).
21. Note that the contract invariants are also checked at the end of the constructor (line 18).
22.

Previous Assertion Checking Modes Next Debugging Instrumented Code Last updated2 years ago