Hints

Providing hints to backend tools with annotations The main use case for scribble properties is to instrument them as actual assertions in the code and allow a backend tool (such as a fuzzer or symbolic execution engine) to check the instrumetned code. A lot of backend tools explore parts of the possible input state space of the contract, trying to find a trace violating the properties. Since this state space is usuallyhuge , backend tools can only explore a small fraction of the input space, and can sometimes waste time on uninteresting inputs. In such case it can be helpful for the user to provide guidance to the backend tools.

In scribblev0.5.7 we added two new annotations that provide guidance to backend tools. Note that these are not properties to be checked, and are thus not part of the "specification" of the contract. Also while we tried to keep these annotations as general as possible, not all tools may be able to make use of them. So far we have been using them with our fuzzer (Harvey), but most tools that areguided by coverage should be able to use these annotations.

Limiting the input space - #require

The first annotation limits the set of inputs that a tool may try for a given contract. Its looks and behaves exactly as Solidity'srequire() statement:

```

Copy /// #require {:msg "description"}

```

A#require annotation can be inserted before a function or before any statement inside of a function.

When inserted before a function, it can talk about the arguments/returns of the function and is instrumented as arequire() statement at the beginning of the function.

When inserted before a statement, it can talk about any local variables in scope, as well as function argument/returns and state variables. It is instrumented as arequire() inserted before the target statement.

For example, consider the below contract that has an 'upgradable' model for interest rate:

```

Copy interface IInterestRateModel { function getInterstRate() external view return (uint256); }

contract Pool { IInterestRateModel intrestRateModel;

function upgradeInterestRateModel(IInterestRateModel newModel) onlyOwner external { this.interestRateModel = newModel; } }

```

The interest rate is modeled by a contract implementingIInterestRateModel and can be upgraded by callingPool.upgradeInterestRateModel . In our experience our fuzzer can often trigger the upgrade path with a random address, thus breaking the contract (the fuzzer is allowed to try things as the owner as well). Once this invalid upgrade is performed, and the contract is in an invalid state a lot of followup false positives are triggered.

We'd like to limit the fuzzer from triggering an invalid upgrade. We can do so with the following#require annotation:

```

Copy function upgradeInterestRateModel(IInterestRateModel newModel) onlyOwner external { /// #require(newModel ==

*); this.interestRateModel = newModel; }*

```

Since this is translated as arequire() , if the fuzzer tries to use an invalid address here, the transaction reverting will not be treated as a property violation.

Suggesting Inputs - #try

In certain cases the backend tool may need to guess/compute a very specific set of inputs to make progress. In such cases it may be useful for the user to suggest input values for the backend tool to try. We support this with the synonymous annotation#try :

```

Copy /// #try {:msg "description"}

```

try

can be inserted before a function or before a given statement. It allows the user to write a boolean expression over the inputs, that aims to help the backend tool. For example consider the following contrivedMultiTokenPool contract, that holds funds on behalf of a user for multiple different tokens. The pool provides a function for withdrawing funds from just one token -withrdawSingleToken .

```

Copy contract MultiTokenPool { function withdrawSingleToken(address token) { require(isValidToken(token)); ... } }

```

If the underlying tool attempts random bits as arguments towithdrawSingleToken , without realizing that a valid token address is needed, then its unlikely that it will be able to cover this function. In such cases a user can suggest specific inputs for the tool to try with the following#try annotation:

```

Copy function withdrawSingleToken(address token) { /// #try token ==; /// #try token == ; require(isValidToken(token)); ... }

```

The above annotation gets translated as the followingif s:

```

Copy

function withdrawSingleToken(address token) { if (token ==) { // dummy statement }

if (token == <Token2 addr) { // dummy statement }

require(isValidToken(token)); ... }

```

The addedif s are designed to have no actual effect on the exectuion of the program, but still to not be optimized away by the compiler.

The idea behind them is that if the underlying tool is trying to maximize coverage by exploring more paths, then it will attempt to reach the bodies of theif statemets. Reaching those bodies requires solving the relatively easy constraintstoken == andtoken == . Solving these constraints allows the tool to try exactly the inputs that we wanted it to try.

Note that unlike the#require annotation,#try does not limit the backend tool toonly use the inputs we suggested - the annotation allows any other possible input as well. Previous Macros Next Expressions Last updated2 years ago

On this page * Limiting the input space - #require * Suggesting Inputs - #try

Was this helpful?