Using Solidity's SMTChecker

Leonardo Alt

Ethereum Foundation

leonardoalt
leo@ethereum.org
leonardoalt

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

Can we find integer values for $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ such that $\phi(\mathbf{a}, \mathbf{b}, \mathbf{c}) \models \top$?

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

Can we find integer values for $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ such that $\phi(\mathbf{a}, \mathbf{b}, \mathbf{c}) \models \top$?

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

Can we find integer values for $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ such that $\phi(\mathbf{a}, \mathbf{b}, \mathbf{c}) \models \top$?

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

Can we find integer values for $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ such that $\phi(\mathbf{a}, \mathbf{b}, \mathbf{c}) \models \top$?

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

$$\texttt{Satisfiable:}\ \{\mathbf{a} := \mathbf{2}, \mathbf{b} := \mathbf{1}, \mathbf{c} := \mathbf{0}\}$$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

Satisfiable: $\{\mathbf{a} := \mathbf{2}, \mathbf{b} := \mathbf{1}, \mathbf{c} := \mathbf{0}\}$ $\{\mathbf{a} := \mathbf{200}, \mathbf{b} := \mathbf{100}, \mathbf{c} := \mathbf{0}\}$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{c}))$$

$$\texttt{Satisfiable:}\ \{\mathbf{a} := \mathbf{2}, \mathbf{b} := \mathbf{1}, \mathbf{c} := \mathbf{0}\}\ \ \{\mathbf{a} := \mathbf{200}, \mathbf{b} := \mathbf{100}, \mathbf{c} := \mathbf{0}\}$$
$$\{\mathbf{a} := \mathbf{1}, \mathbf{b} := \mathbf{1}, \mathbf{c} := \mathbf{1}\}$$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) < \mathbf{f}(\mathbf{c}))$$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) < \mathbf{f}(\mathbf{c}))$$

Unsatisfiable: $\emptyset$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) < \mathbf{f}(\mathbf{c}))$$

Unsatisfiable: $\emptyset$

SMT Solver

$$\phi := \forall \mathbf{x}(\mathbf{f}(\mathbf{x}) = \mathbf{x} * \mathbf{42}) \wedge (\mathbf{a} \geq \mathbf{b}) \wedge (\mathbf{b} \geq \mathbf{c}) \wedge (\mathbf{f}(\mathbf{a}) < \mathbf{f}(\mathbf{c}))$$

Unsatisfiable: $\emptyset$

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) >= f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$

$$(a \geq b) \wedge (b \geq c) \wedge$$

$$(f(a) < f(c))$$

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) >= f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$

$$(a \geq b) \wedge (b \geq c) \wedge$$

$$(f(a) < f(c))$$

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) >= f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$

$$(a \geq b) \wedge (b \geq c) \wedge$$

$$(f(a) < f(c))$$

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) >= f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$

$$(a \geq b) \wedge (b \geq c) \wedge$$

$$(f(a) < f(c))$$

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) >= f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$
$$(a \geq b) \wedge (b \geq c) \wedge$$
$$(f(a) < f(c))$$

Unsatisfiable: the assertion is safe!

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) == f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$

$$(a \geq b) \wedge (b \geq c) \wedge$$

$$(f(a) \neq f(c))$$

Program Verification

```
pragma experimental SMTChecker;
contract C
{
  uint c;
  function f(uint x) public pure
    returns (uint) {
      return x * 42;
  }
  function g(uint a, uint b) public {
    require(a >= b);
    require(b >= c);
    assert(f(a) == f(c));
  }
}
```

$$\forall x(f(x) = x * 42) \wedge$$

$$(a \geq b) \wedge (b \geq c) \wedge$$

$$(f(a) \neq f(c))$$

```
Warning: Assertion violation happens here
        assert(f(a) == f(c));
        ^------------------^
  for:  a = 1, b = 1, c = 0
```

Satisfiable: a counterexample is given.

SMTChecker

- SMT-based smart contract formal verification framework
- Built-in the compiler
- Encodes program logic into SMT statements
- Checks assertions, overflow/underfow, trivial conditions/unreachable code
- Sound but not complete
- Fast and light
- Gives useful counterexamples

Smart Contract Verification Frameworks

- EVM Formal Semantics: Eth-Isabelle, KEVM
- KLab: debugger for K proofs (KLab workshop tomorrow!)
- EVM bytecode Symbolic Execution: Oyente, Mythril, MAIAN
- Translation of Solidity to verifiable languages: Why3, F*, ZEUS

How to enable it

```
pragma experimental SMTChecker;
```

Formal Specification

- `require` – assumptions
- `assert` – verification targets

Formal Specification - Require

From Solidity docs:
The require function should be used to ensure valid condition on inputs and contract state variables, or to validate return values from calls to external contracts.

Formal Specification - Require

```
contract C
{
    uint a;
    function g() ...
    function h() ...
    function f(uint x) public {
        // Correctly filters values for a and x
        require(a == 0);
        require(x < 100);
        a += x;
        // Should be a target, not an assumption
        require(a < 100);
    }
}
```

Formal Specification - Assert

From Solidity docs:
The assert function should only be used to test for internal errors, and to check invariants. Properly functioning code should never reach a failing assert statement; if this happens there is a bug in your contract which you should fix.

Formal Specification - Assert

```
contract C
{
    uint a;
    function g() ...
    function h() ...
    function f(uint x) public {
        // The value of a is not necessarily 0,
        // unless the developer knows it's an invariant
        assert(a == 0);
        // Anyone can call this function with x >= 100
        // and break the assertion
        assert(x < 100);
        a += x;
        // Correctly placed as a verification target
        assert(a < 100);
    }
}
```

What about false positives

- Complex types and functions
- External function calls
- Contract state invariants

Helping the SMTChecker: require flood

+ constraints

– false positives

Thank you!

Questions?