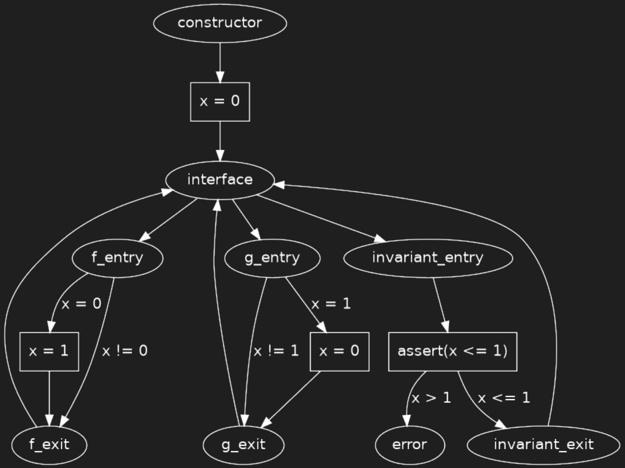
Fully automated inductive invariants inference for Solidity smart contracts

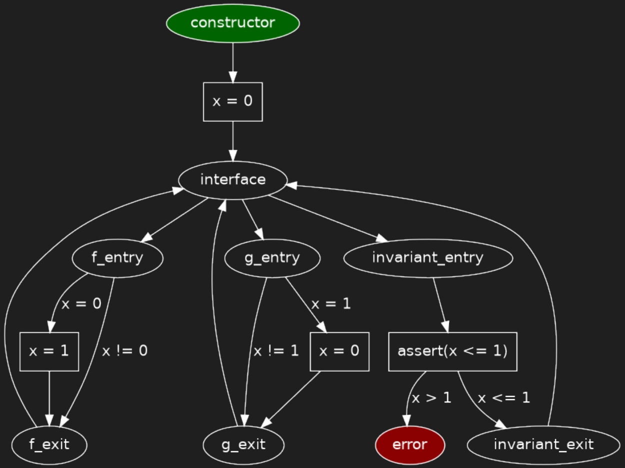
Leonardo Alt

Ethereum Foundation

leonardoalt
leo@ethereum.org
leonardoalt

```
pragma experimental SMTChecker;

contract StateMachine {
    uint x;

    function f() public {
        if (x == 0)
            x = 1;
    }

    function g() public {
        if (x == 1)
            x = 0;
    }

    function invariant() public {
        assert(x <= 1);
    }
}
```


constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x = 1
x != 0
x != 1
x = 0
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit


constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x = 1
x != 0
x != 1
x = 0
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

```
pragma experimental SMTChecker;

contract StateMachine {
    uint x;

    function f() public {
        if (x == 0)
            x = 1;
    }

    function g() public {
        if (x == 1)
            x = 0;
    }

    function invariant() public {
        assert(x <= 1);
    }
}
```

SMTChecker returns safe and the contract invariant $\mathbf{x} \leq \mathbf{1}$.

The invariant is the same as the assertion we wanted to prove.

```
pragma experimental SMTChecker;

contract StateMachine {
    uint x;

    function f() public {
        if (x == 0)
            x = 1;
    }

    function g() public {
        if (x == 1)
            x = 0;
    }

    function h() public {
        if (x == 7)
            x = 100;
    }

    function invariant() public {
        assert(x <= 7);
    }
}
```

SMTChecker again returns safe and the contract invariant $\mathbf{x \leq 1}$.

Why does it still return contract invariant $\mathbf{x} \leq \mathbf{1}$, even though we tried to prove $\mathbf{x} \leq \mathbf{7}$?

What is the difference between invariants $\mathbf{x} \leq \mathbf{1}$ and $\mathbf{x} \leq \mathbf{7}$?

Invariant: $\mathbf{x} \leq \mathbf{1}$

Invariant: $\mathbf{x} \leq \mathbf{7}$

Invariant: $\mathbf{x} \leq \mathbf{1}$

```
x <= 1;
f():
if (x == 0)
    x = 1;
x <= 1;
```

Invariant: $\mathbf{x} \leq \mathbf{7}$

Invariant: $\mathbf{x} \leq \mathbf{1}$

```
x <= 1;
f():
if (x == 0)
    x = 1;
x <= 1;
```

Invariant: $\mathbf{x} \leq \mathbf{7}$

```
x <= 7;
f():
if (x == 0)
    x = 1;
x <= 7;
```

Invariant: $\mathbf{x} \leq \mathbf{1}$

```
x <= 1;
f():
if (x == 0)
    x = 1;
x <= 1;

x <= 1;
g():
if (x == 1)
    x = 0;
x <= 1;
```

Invariant: $\mathbf{x} \leq \mathbf{7}$

```
x <= 7;
f():
if (x == 0)
    x = 1;
x <= 7;
```

Invariant: $\mathbf{x \leq 1}$

```
x <= 1;
f():
if (x == 0)
    x = 1;
x <= 1;

x <= 1;
g():
if (x == 1)
    x = 0;
x <= 1;
```

Invariant: $\mathbf{x \leq 7}$

```
x <= 7;
f():
if (x == 0)
    x = 1;
x <= 7;

x <= 7;
g():
if (x == 1)
    x = 0;
x <= 7;
```

Invariant: $x \leq 1$

```
x <= 1;
f():
if (x == 0)
    x = 1;
x <= 1;

x <= 1;
g():
if (x == 1)
    x = 0;
x <= 1;

x <= 1;
h():
if (x == 7)
    x = 100;
x <= 1;
```

Invariant: $x \leq 7$

```
x <= 7;
f():
if (x == 0)
    x = 1;
x <= 7;

x <= 7;
g():
if (x == 1)
    x = 0;
x <= 7;
```

Invariant: $\mathbf{x} \leq \mathbf{1}$

```
x <= 1;
f():
if (x == 0)
    x = 1;
x <= 1;

x <= 1;
g():
if (x == 1)
    x = 0;
x <= 1;

x <= 1;
h():
if (x == 7)
    x = 100;
x <= 1;
```

Invariant: $\mathbf{x} \leq \mathbf{7}$

```
x <= 7;
f():
if (x == 0)
    x = 1;
x <= 7;

x <= 7;
g():
if (x == 1)
    x = 0;
x <= 7;

x <= 7;
h():
if (x == 7)
    x = 100;
x ? 7;
```

$\mathbf{x} \leq \mathbf{1}$ is Inductive!

$$\mathbf{x} \leq \mathbf{1} \wedge \text{local behavior} \implies \mathbf{x} \leq \mathbf{1}$$

Inductive invariants

can summarize a relevant piece of code without relying on prior information

Inductive invariants

are particularly useful to summarize the behavior of loops

Loop invariants

```
y = 0;
while (y < x)
    ++x;
assert(x == y);
```

SMTChecker returns safe and the loop invariant $y \leq x$.

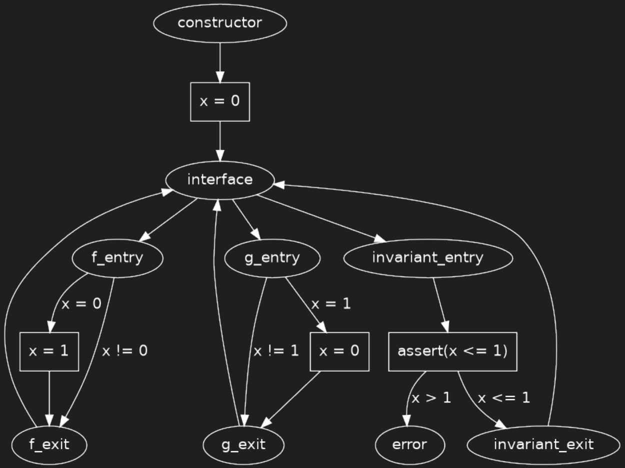
- $y \leq x$ is the core property of the the loop
- After the loop, its condition is false: $y \geq x$
- Which leads to $y \leq x \wedge y \geq x \implies y = x$.

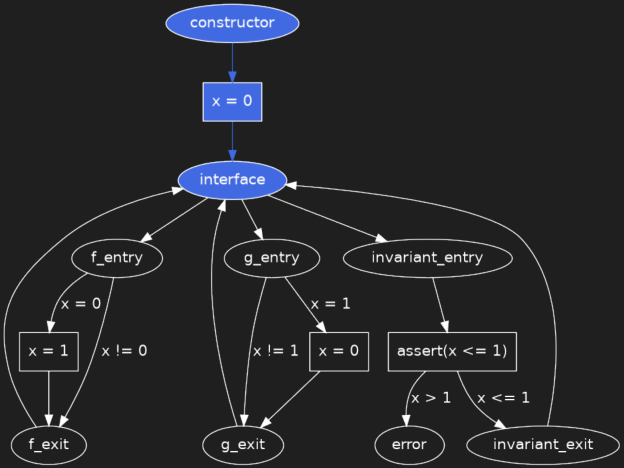
Inductive invariants

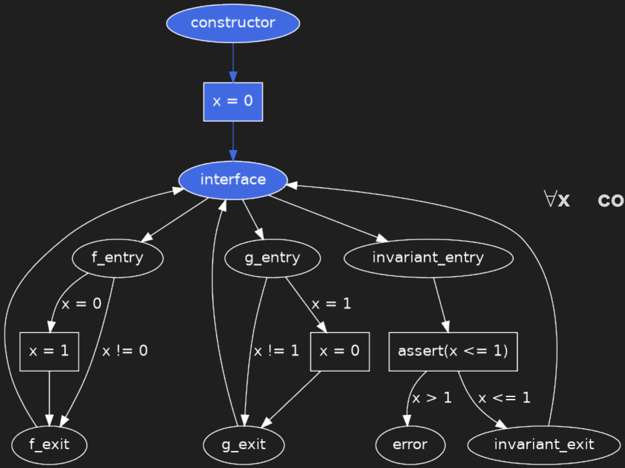
can also be applied to recursive programs, as the inductive hypothesis to be proven.

How can we use inductive invariants for smart contract verification?

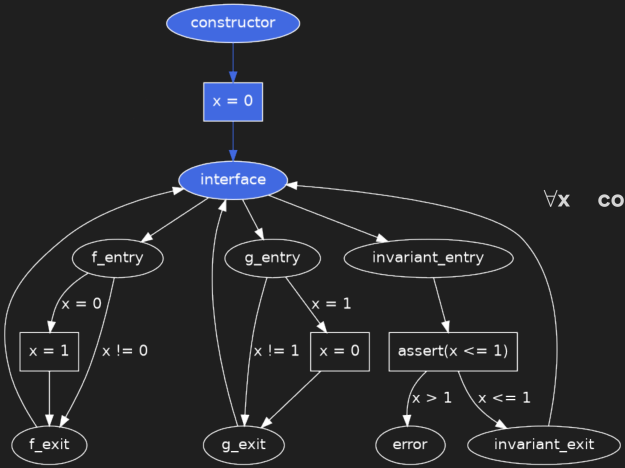
The lifecycle of a smart contract can also be seen as a control-flow containing a loop


constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x != 0
x != 1
x = 1
x = 1
x = 0
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

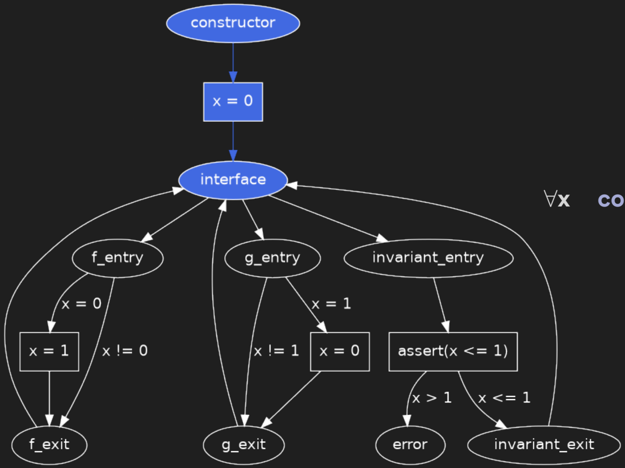

constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x = 1
x != 0
x != 1
x = 0
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit



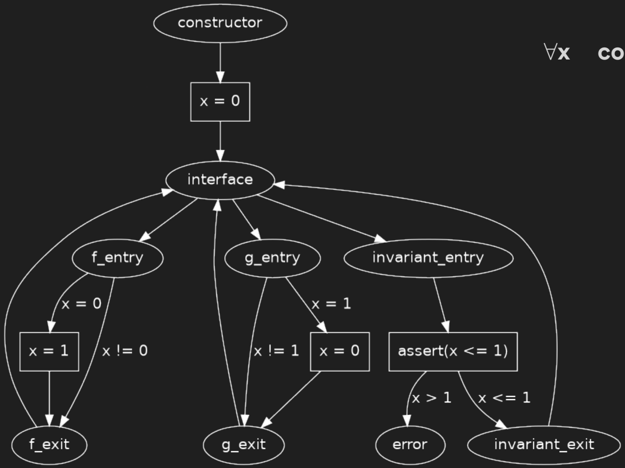
$$\forall x \quad \mathbf{constructor}(x) \wedge x = 0 \implies \mathbf{interface}(x)$$



$$\forall x \quad \mathbf{constructor}(x) \land x = 0 \implies \mathbf{interface}(x)$$



$$\forall x \quad \text{constructor}(x) \land x = 0 \implies \text{interface}(x)$$



$$\forall x \quad \text{constructor}(x)$$

$$\forall x \quad \text{constructor}(x) \wedge x = 0 \implies \text{interface}(x)$$

$$\forall x \quad \text{interface}(x) \implies f(x)$$

$$\forall x \quad \text{interface}(x) \implies g(x)$$

$$\forall x \quad \text{interface}(x) \implies \text{invariant}(x)$$

$$\forall x \quad f_{entry}(x) \wedge x = 0 \implies f_{body}(x)$$

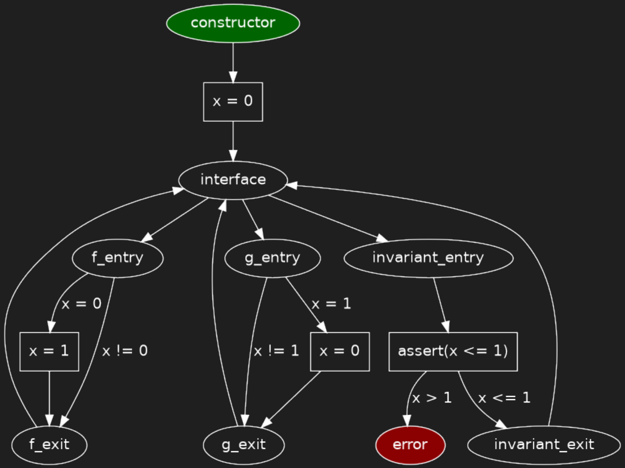
$$\forall x \quad f_{body}(x) \wedge x' = 1 \implies f_{exit}(x')$$

$$\forall x \quad f_{entry}(x) \wedge x = 1 \implies f_{exit}(x)$$

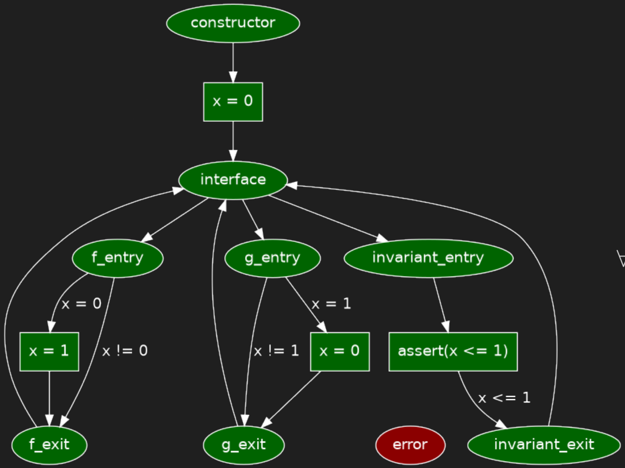
$$\forall x \quad f_{exit}(x) \implies \text{interface}(x)$$

$$\forall x \quad \text{invariant}(x) \wedge x > 1 \implies \text{error}(x)$$

$$\forall x \quad \text{invariant}(x) \wedge x <= 1 \implies \text{interface}(x)$$


constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x != 0
x != 1
x = 1
x = 0
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

error(x)?



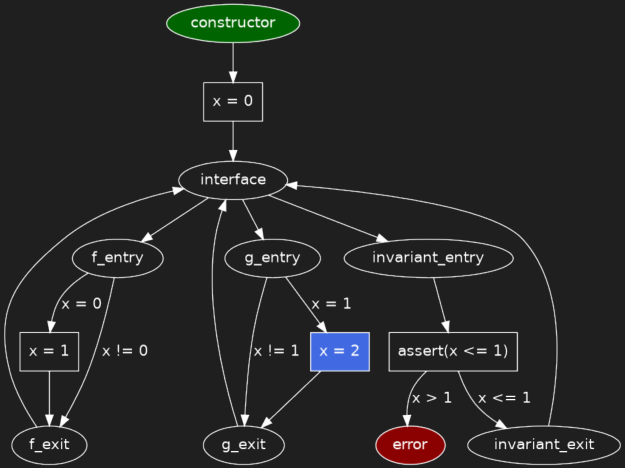
$\forall$**x.error(x)** is unreachable

- Existential positive Least Fixed-Point logic (E+LFP) matches Hoare logic

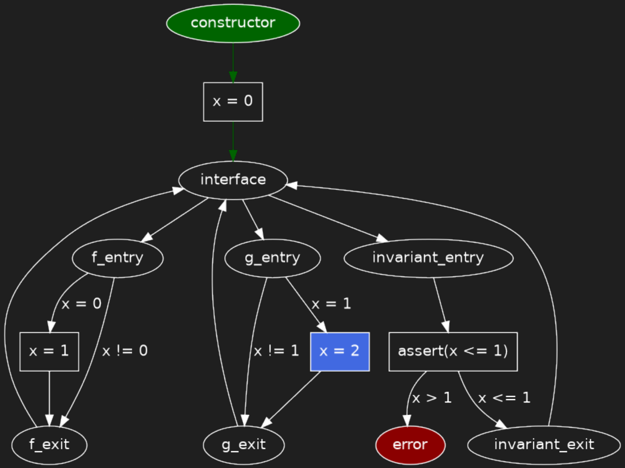
  Blass, A., Gurevich, Y.: Existential fixed-point logic. In: Computation Theory and Logic, In Memory of Dieter Rödding. pp. 20-36 (1987)

- E+LFP solved by CHCs satisfiability

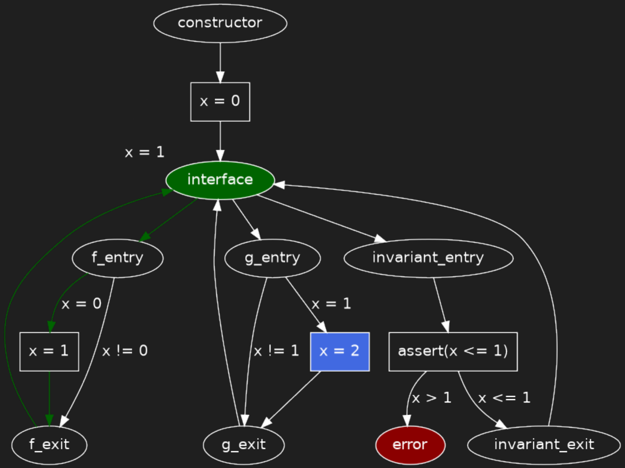
  Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II. pp. 24-51 (2015)


constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x != 0
x = 1
x != 1
x = 2
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

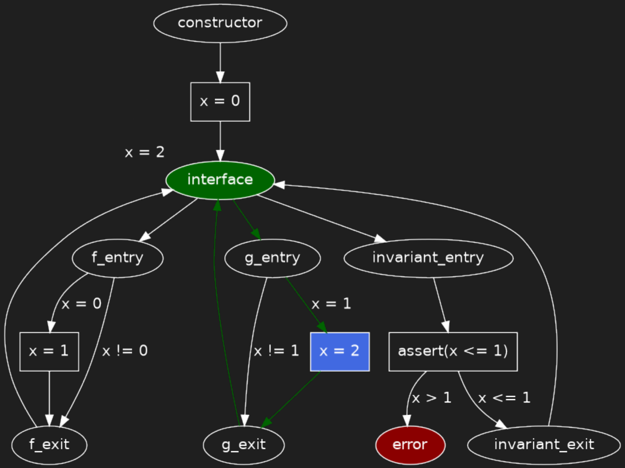
error(**x**)?


constructor
x = 0
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x = 1
x != 0
x != 1
x = 2
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

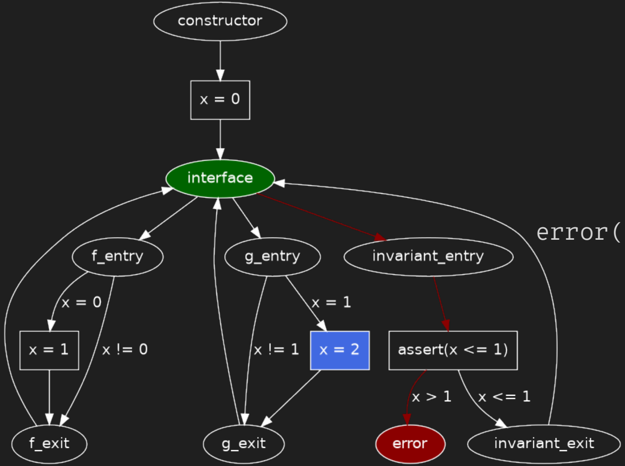
error(**x**)?


constructor
x = 0
x = 1
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x = 1
x != 0
x != 1
x = 2
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

error(**x**)?


constructor
x = 0
x = 2
interface
f_entry
g_entry
invariant_entry
x = 0
x = 1
x = 1
x != 0
x != 1
x = 2
assert(x <= 1)
x > 1
x <= 1
f_exit
g_exit
error
invariant_exit

error(x)?



error(x) is reachable!

The sequence that leads to the error is deployment, f(), g(), invariant().

Horn solvers

- Predicate abstraction
- Abstract interpretation
- Maximal inductive subsets
- Machine learning

Horn solvers

- SMT-based unbounded model checking - PDR/IC3
- Spacer - spacer.bitbucket.io
- Backwards reachability
- Quantifier-free SMT queries and interpolation to find predecessors and new lemmas

What's next

- Function calls!
  - Function summaries
  - No changes in the state of the caller contract
  - Synthesis of external functions
  - Multi-contract-unbounded-transactions properties
  - Maybe entire state?
- Nice looking counterexamples and invariants
- Better usability
- Simple formal spec language –
  github.com/ethereum/smart-contract-spec-lang

Final remarks

- SMT solvers are powerful and fast (hopefully as powerful as we sell them)
- Unbounded model checking with PDR
- Unbounded transaction properties and counterexamples
- Embedded in the Solidity compiler
- Contract inductive invariants can further help verification of bytecode (added lemmas)

Thank you!