A Framework for Compositional Model Checking

Yu-Yang Lin Nikos Tzevelekos

Model Checking

Model checking: verification technique where a model of a system is exhaustively and automatically checked against some specification

Some **limitations** of model checking:

1) **State Explosion Problem**

   Solution 1: *Bounded Model Checking*; monolithic, however.

   Solution 2: *Game Semantics* allows compositional verification to individually check components that do fit in memory

2) **Environment Problem:** external components (e.g. libraries, modules, system calls, remote procedure calls) typically have no model since code might not be available

   Solution: *Game Semantics* models the interaction between a program and its environment as a *sequence of moves* (trace)

Why open code matters

```
#in The DAO
def withdraw(user,m):
   if funds[user] >= m:
      user.send(m)
      funds[user] -= m
      assert(funds[user]>=0)
```

The DAO is a *Decentralized Autonomous Organization* (DAO) in the Ethereum platform

DAOs are a set of *smart contracts* (scripts) in the blockchain

The DAO had a bug, in their smart contract, analogous to the Python code above

Why open code matters

```
#in The DAO
def withdraw(user,m):
    if funds[user] >= m:
        user.send(m)
        funds[user] -= m
        assert(funds[user]>=0)
```

```
#in the attacker
def send(m):
    wallet.add(m)
    withdraw(self,1)
```

Recursive call drained The DAO for **over 3.6 million ether**

Price of ether dropped from **$20 to $13**

The Ethereum network was **hard-forked** to undo the “attack”

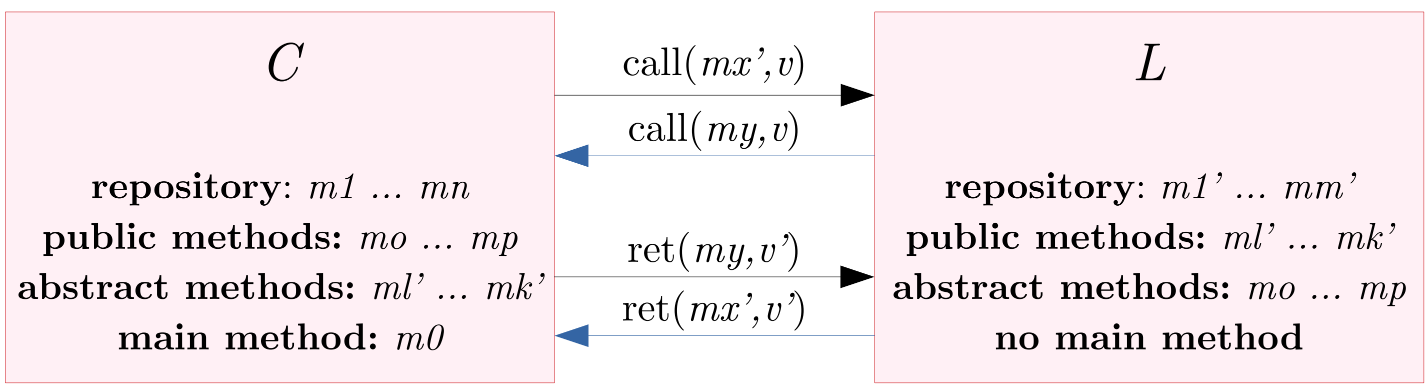
Members reject the hard-fork, claiming it violates the principles of a decentralized network, and continue on the original blockchain, now called **Ethereum Classic**

Our Approach

Combine *Bounded Model Checking* with *Game Semantics* to model check open code with free variables of arbitrary order

Focus on higher-order functions and higher-order store

Use the *Library*(L)-*Client*(C) paradigm



Goal 1: model check libraries independent of a client

Goal 2: compose the semantics of a library and a client to obtain the semantics of the whole program

A Syntax for Libraries

Terms are a lambda-calculus with higher-order store

$$
\begin{aligned}
M ::= \quad & \mathtt{assert}(M) \mid x \mid m \mid i \mid () \mid r := M \mid !r \mid M \oplus M \mid \langle M, M \rangle \\
& \mid \pi_1 M \mid \pi_2 M \mid xM \mid \mathtt{if}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M \\
& \mid \mathtt{let}\ x = M\ \mathtt{in}\ M \mid \mathtt{letrec}\ x = \lambda x.M\ \mathtt{in}\ M \mid \lambda x.M
\end{aligned}
$$

$$
\frac{M : \mathtt{int}}{\mathtt{assert}(M) : \mathtt{unit}} \qquad \frac{}{() : \mathtt{unit}} \qquad \frac{}{i : \mathtt{int}} \qquad \frac{x \in \mathtt{Vars}_\theta}{x : \theta} \qquad \frac{m \in \mathtt{Meths}_{\theta,\theta'}}{m : \theta \to \theta'}
$$

$$
\frac{M : \mathtt{int} \qquad M_0, M_1 : \theta}{\mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_0 : \theta} \qquad \frac{r \in \mathtt{Refs}_\theta}{!r : \theta} \qquad \frac{r \in \mathtt{Refs}_\theta \qquad M : \theta}{r := M : \mathtt{unit}} \qquad \frac{M' : \theta \to \theta' \qquad M : \theta}{M'\,M : \theta'}
$$

Libraries consist of a sequence of *method declarations*

- may themselves depend on *unknown/abstract* methods provided by the environment

Bounded Operational Semantics

Configurations of the form (M, R, S, k)

- **Bound** k on nested method application

M: term to evaluate
R: method repository
S: store
k: bound

Example rules:

$$(E[\mathtt{assert}(i)], R, S, k) \to (E[()], R, S, k) \quad (i \neq 0)$$

$$(E[!r], R, S, k) \to (E[S(r)], R, S, k)$$

$$(E[\mathtt{if}\ 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_0], R, S, k) \to (E[M_0], R, S, k)$$
$$(E[\mathtt{if}\ i\ \mathtt{then}\ M_1\ \mathtt{else}\ M_0], R, S, k) \to (E[M_1], R, S, k) \quad (i \neq 0)$$

$$(E[mv], R, S, k) \to (E[(\!|M\{v/x\}|\!)], R, S, k-1) \quad \text{where } R(m) = \lambda x.M$$
$$(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k+1)$$

$$E ::= \bullet \mid \mathtt{assert}(E) \mid r := E \mid E \oplus M \mid v \oplus E \mid \langle E, M\rangle \mid \langle v, E\rangle \mid \pi_j E \mid mE$$
$$\mid \mathtt{let}\ x = E\ \mathtt{in}\ M \mid \mathtt{if}\ E\ \mathtt{then}\ M\ \mathtt{else}\ M \mid (\!|E|\!)$$

Bounded Games

We present game semantics in ***operational form***

- i.e. a *trace semantics* for *open terms*

Traces: sequences of moves of the form $\texttt{call}(m,v)/\texttt{ret}(m,v)$

The semantics is **bounded** for both players:

- For P we bound nested method calls with bound k
- For O we bound *chattering*, which is when O keeps playing at the same *level* of the game, with bound l

$$\underbrace{(\mathcal{E}, M, R, \mathcal{P}, \mathcal{A}, S, k, -)}_{P\text{-configuration}} \qquad \underbrace{(\mathcal{E}, -, R, \mathcal{P}, \mathcal{A}, S, k, l)}_{O\text{-configuration}}$$

M, R, S, k as before, $\mathcal{E}$ is a call stack,
$\mathcal{P}$ and $\mathcal{A}$ are the method names of P and O

Back to The DAO Attack

Consider the following library:

```
public withdraw;
abstract send;

funds := 50;
withdraw = λm.
 if   !funds >= m
 then send(m);
      funds := !funds - m;
      assert(!funds >= 0)
 else skip
```

where $A;B$ is syntax sugar for $\mathrm{let}\ \_ = A\ \mathit{in}\ B$

We start from an opponent configuration (with $k,l{=}2$):

$$C_0 = (2,\text{-},R,\{withdraw\},\{send\},\{(funds{:=}50)\},2,2)_o$$

where $R(withdraw) = \lambda m.\ \dots$ and $dom(R) = \{withdraw\}$

Back to The DAO Attack

$$C_0 \xrightarrow{withdraw(42)?} (1 :: withdraw :: 2, withdraw(42), S, 2, -)_p$$
$$\to^* (1 :: withdraw :: 2, E[send(42)], S, 1, -)_p$$
$$\xrightarrow{send(42)?} (send :: E :: \ldots, -, S, 1, 1)_o$$
$$\xrightarrow{withdraw(42)?} (0 :: withdraw :: \ldots, withdraw(42), S, 1, -)_p$$
$$\to^* (0 :: withdraw :: \ldots, E'[send(42)], S, 0, -)_p$$
$$\xrightarrow{send(42)?} (send :: E' :: \ldots, -, S, 0, 0)_o$$
$$\xrightarrow{send(())!} (0 :: withdraw :: \ldots, E'[()], S, 0, -)_p$$
$$\to^* (0 :: withdraw :: \ldots, (), S[funds \mapsto 8], 0, -)_p$$
$$\xrightarrow{withdraw(())!} (send :: E :: \ldots, -, S[funds \mapsto 8], 0, 0)_o$$
$$\xrightarrow{send(())!} (1 :: withdraw :: \ldots, E[()], R, \mathcal{P}, \mathcal{A}, S[funds \mapsto 8], 1, -)_p$$
$$\to^* (1 :: withdraw :: \ldots, E[assert(-34 \geq 0)], S[funds \mapsto -34], 1, -)_p$$

```
public withdraw;
abstract send;

funds := 50;
withdraw = λm.
 if   !funds >= m
 then send(m);
      funds := !funds - m;
      assert(!funds >= 0)
 else skip
```

C/L-Compositionality

Intuitively: For any client (C) that imports library (L), the semantics of the linked program can be obtained by composing the semantics of L and C

This requires a correspondence between *semantic composition* and *syntactic composition* for any terminating configuration

Formally:

For any library L and compatible client C:

- *there exists a bound k such that $L;C$ with k terminates with χ, iff*
- *there exist traces $\tau \in [\![L]\!]_{k1,l1}$ and $\tau^{\perp} \in [\![C]\!]_{k2,l2}$, such that $[\![L]\!]_{k1,l1}$ terminates with χ by playing the moves in τ*

where χ is a terminal configuration holding a term v or $\mathrm{assert}(0)$.

Lemma A.1. *Given $\rho \asymp \rho'$ where ρ is an L-configuration and ρ' is a C-configuration, it is the case that $(\rho \oslash \rho') \sim (\rho \curlywedge \rho')$.*

From Concrete to Symbolic

Model checking: we place our semantics in a symbolic setting

Two approaches considered for Bounded Model Checking:

- **CBMC approach:** translating all paths in the program into a single SAT formula with joins
- **Bounded Symbolic Execution:** symbolically explore every possible path up to a given depth, keeping track of a path condition formula for each path explored

For our semantics, symbolic execution is more fitting

Symbolic Execution

Add symbolic environment and path condition, and check for reachability of keyword `fail`

Symbolic branching on assertions:

$$(E[\texttt{assert}(0)], R, \sigma, pc, k) \xrightarrow{sym} (\texttt{fail}, \sigma, pc)$$

$$(E[\texttt{assert}(x)], R, \sigma, pc, k) \xrightarrow{sym} (\texttt{fail}, \sigma, pc \wedge (\sigma(x) = 0))$$

$$(E[\texttt{assert}(i)], R, \sigma, pc, k) \xrightarrow{sym} (E[()], R, \sigma, pc, k) \quad \text{where } i \neq 0$$

$$(E[\texttt{assert}(x)], R, \sigma, pc, k) \xrightarrow{sym} (E[()], R, \sigma, pc \wedge (\sigma(x) \neq 0), k)$$

Updating the symbolic environment:

$$(E[!r], R, \sigma, pc, k) \xrightarrow{sym} (E[\sigma(r)], R, \sigma, pc, k) \quad \text{where } x \text{ is fresh}$$

$$(E[r := v], R, \sigma, pc, k) \xrightarrow{sym} (E[()], R, \sigma[r \mapsto \sigma(v)], pc, k)$$

Symbolic branching on conditionals:

$$(E[\texttt{if } 0 \texttt{ then } M_1 \texttt{ else } M_0], R, \sigma, pc, k) \xrightarrow{sym} (E[M_0], R, \sigma, pc, k)$$

$$(E[\texttt{if } i \texttt{ then } M_1 \texttt{ else } M_0], R, \sigma, pc, k) \xrightarrow{sym} (E[M_1], R, \sigma, pc, k) \quad \text{where } i \neq 0$$

$$(E[\texttt{if } x \texttt{ then } M_1 \texttt{ else } M_0], R, \sigma, pc, k) \xrightarrow{sym} (E[M_0], R, \sigma, pc \wedge (\sigma(x) = 0), k)$$

$$(E[\texttt{if } x \texttt{ then } M_1 \texttt{ else } M_0], R, \sigma, pc, k) \xrightarrow{sym} (E[M_1], R, \sigma, pc \wedge (\sigma(x) \neq 0), k)$$

Symbolic Games

Symbolic games: games where moves involve symbolic values, and a symbolic environment and path condition are used to model each path

Obtain symbolic games by:

- Extending game configurations with a symbolic environment (σ) and a path condition (pc)
- Transforming concrete moves into *symbolic moves* by allowing players to play symbolic values (free variables)
- Using symbolic execution as internal moves

Results in configurations:

$$(\mathcal{E}, M, R, \mathcal{P}, \mathcal{A}, \sigma, pc, k, -)_p \qquad (\mathcal{E}, -, R, \mathcal{P}, \mathcal{A}, \sigma, pc, k, l)_o$$

p-configuration $\qquad$ o-configuration $\qquad$ o-configuration

Symbolic DAO Attack

$$C_0 \xrightarrow{withdraw(x)?} (\ldots, withdraw(x), \{(funds := 50)\}, \top, 2, -)_p$$

$$\rightarrow^* (\ldots, send(x), \{(funds := 50)\}, (x \leq 50), 1, -)_p$$

$$\xrightarrow{send(x)?} (\ldots, -, \{(funds := 50)\}, (x \leq 50), 1, 1)_o$$

$$\xrightarrow{withdraw(y)?} (\ldots, withdraw(y), \{(funds := 50)\}, (x \leq 50), 1, -)_p$$

$$\rightarrow^* (\ldots, send(y), \{(funds := 50 - y)\}, (x < 50) \wedge (y \leq 50), 0, -)_p$$

$$\xrightarrow{send(y)?} (\ldots, -, \{(funds := 50 - y)\}, (x \leq 50) \wedge (y \leq 50), 0, 0)_o$$

$$\xrightarrow{send(())!} (\ldots, \{(funds := 50 - y - x)\}, (x \leq 50) \wedge (y \leq 50), 1, 0)_p$$

$$\rightarrow^* (\ldots, \texttt{assert}(!funds >= 0), \{(funds := 50 - y - x)\}, (x \leq 50) \wedge (y \leq 50), 1, 0)_p$$

$$\rightarrow (\texttt{fail}, (x \leq 50) \wedge (y \leq 50) \wedge \neg(50 - y - x \geq 0))$$

$$pc = (x \leq 50) \wedge (y \leq 50) \wedge \neg(50 - y - x \geq 0)$$

$$\{(x \mapsto 1), (y \mapsto 50)\} \vDash (1 \leq 50) \wedge (50 \leq 50) \wedge \neg(-1 \geq 0)$$

Soundness and Correctness

Sound Errors: Model Checking a library will find an assertion violation if and only if the error is reachable by executing the counter example on the linked library-client system

- i.e. produces no false positives

Formally:

(I) **Soundness:** *For any L the following are equivalent:*

1. $L \xrightarrow{\tau}\!\!\!\rightarrow_G (\chi, \sigma, pc)$ *and* $\exists \alpha . \alpha \vDash pc \wedge \sigma^\circ$

2. $L \xrightarrow{\tau\{\alpha\}}\!\!\!\rightarrow \chi\{\alpha\}$

where $\chi' \neq \texttt{nil}$ *and* $\chi\{\alpha\}$ *is the equivalent concrete configuration.*

(II) **Correctness:** *For any L the following are equivalent:*

1. $L \twoheadrightarrow \chi$ *with bounds* k, l,

2. $\exists C . L ; C \twoheadrightarrow \chi$ *with bound* k'

where χ *is a terminal configuration holding a term* v *or* $\texttt{assert}(0)$.

(III) **Sound Errors (I.1) ↔ (II.2):** corollary from (I) and (II)

Further Directions

Extend properties checked

- Currently limited reachability, i.e. safety
- Liveness and other temporal properties

Specification-driven BMC

- Instead of full symbolic execution for clients, we drive path exploration to greatly reduce the number of paths explored

General compositionality

- Currently limited to library and client that close each other
- Library-library composition would allow compositionality at the level of functions

Thank You

Implementing this...

To model check a library L:

1) $[\![L]\!]^{k,l}$ produces a transition system starting from an opponent configuration with final configurations of the form (χ, τ, σ, pc), where χ can be a value (v), an assertion violation ($fail$) or a bound exception (nil)

2) For each final configuration of the form $(fail, \tau, \sigma, pc)$, find a model:

$$M \vdash (\sigma^o \wedge pc)$$

3) If a model is found, τ contains a counterexample in the form of a trace of moves that causes the library to reach an assertion violation

$[\![L]\!]^{k,l}$ performs form of bounded symbolic execution for programs with higher-order store and free variables of arbitrary order.

Model Checking Clients and Linked Libraries: We write $[\![C]\!]^{k,l,m0}$ for the transition system starting from a proponent configuration holding a term M_0. We can then compose the separate library and client semantics to obtain the semantics of the linked program $(L;C)$.