Declarative Smart Contracts

A declarative domain-specific language for smart contracts

Haoxian Chen
University of Pennsylvania
USA
hxchen@seas.upenn.edu

Gerald Whitters
University of Pennsylvania
USA
whitters@seas.upenn.edu

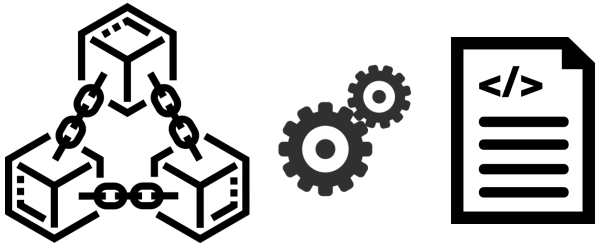
Mohammad Javad Amiri
University of Pennsylvania
USA
mjamiri@seas.upenn.edu

Yuepeng Wang
Simon Fraser University
Canada
yuepeng@sfu.ca

Boon Thau Loo
University of Pennsylvania
USA
boonloo@seas.upenn.edu

Smart contracts

- are programs **stored** and **executed** on blockchains.



- typical applications: tokens (digital money), auctions, financing, etc.

Smart contracts

- **Billions** $ worth of tokens being traded everyday [1].
- Bugs in smart contracts have cost significant financial loss [2,3].
- Important to ensure smart contract correctness.

#	Token	Price	Change (%)	Volume (24H)
1	Tether USD (USDT)	$0.9986 0.000035 Btc 0.000511 Eth	-0.24%	$47,358,862,799.00
2	BNB (BNB)	$442.8586 0.015315 Btc 0.226753 Eth	1.87%	$3,362,542,087.00
3	USD Coin (USDC)	$0.9997 0.000035 Btc 0.000512 Eth	-0.13%	$5,452,121,640.00
4	HEX (HEX)	$0.1154 0.000004 Btc 0.000059 Eth	-0.67%	$17,576,171.00
5	Binance USD (BUSD)	$0.9993 0.000035 Btc 0.000512 Eth	-0.28%	$5,724,586,616.00
6	Wrapped BTC (WBTC)	$28,996.00 1.002739 Btc 14.846598 Eth	-3.46%	$323,563,692.00
7	stETH (stETH)	$1,930.14 0.066748 Btc 0.988275 Eth	-4.59%	$29,347,677.00

[1] Etherscan. ERC-20 Top Tokens. https://etherscan.io/tokens
[2] David Siegel. 2016. Understanding The DAO Attack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/
[3] Parity Technologies. 2017. Parity Security Alert. https://www.parity.io/blog/security-alert-2/

Smart contracts today

```
contract Wallet {
  address private _owner;
  mapping(address => int) private _balanceOf;
  int private _totalSupply;

  function mint(address account, int amount)
     public {
    require(msg.sender == _owner);
    require(account != address(0));
    _totalSupply += amount;
    _balanceOf[account] += amount;
  }

  function balanceOf(address account)
      public view returns(int) {
    return _balanceOf[account];
  }

  // Other functions ...
}
```

Solidity: an object-oriented programing language.
A **contract** is like a **class** in Java.
Contract deployment is like **class instantiation**.

Smart contracts today

```
contract Wallet {
  address private _owner;
  mapping(address => int) private _balanceOf;
  int private _totalSupply;

  function mint(address account, int amount)
     public {
    require(msg.sender == _owner);
    require(account != address(0));
    _totalSupply += amount;
    _balanceOf[account] += amount;
  }

  function balanceOf(address account)
      public view returns(int) {
    return _balanceOf[account];
  }

  // Other functions ...
}
```

Solidity: an object-oriented programing language.
A **contract** is like a class in Java.

Contract states declaration.

Smart contracts today

```
contract Wallet {
  address private _owner;
  mapping(address => int) private _balanceOf;
  int private _totalSupply;

  function mint(address account, int amount)
     public {
    require(msg.sender == _owner);
    require(account != address(0));
    _totalSupply += amount;
    _balanceOf[account] += amount;
  }

  function balanceOf(address account)
      public view returns(int) {
    return _balanceOf[account];
  }

  // Other functions ...
}
```

Solidity: an object-oriented programing language.
A **contract** is like a class in Java.

Contract states declaration.

Transactions are public functions that alter the contract states.

Smart contracts today

```
contract Wallet {
  address private _owner;
  mapping(address => int) private _balanceOf;
  int private _totalSupply;

  function mint(address account, int amount)
     public {
    require(msg.sender == _owner);
    require(account != address(0));
    _totalSupply += amount;
    _balanceOf[account] += amount;
  }

  function balanceOf(address account)
      public view returns(int) {
    return _balanceOf[account];
  }

  // Other functions ...
}
```

Solidity: an object-oriented programing language.
A **contract** is like a class in Java.

Contract states declaration.

Transactions are public functions that alter the contract states.

Views are public functions that do not alter contract states.

Why a new language?

- Existing smart contract verification work focus on generic, low-level properties.
  - e.g., re-entrancy attack (leads to losing money), integer overflow, etc.

- But not so much on **contract-specific**, **high-level properties.**
  - e.g., do account balances add up to total supply of tokens?

- We need a **high-level**, yet **executable** language.
  - Ease specification and implementation.

DeCon

We present **DeCon**, a declarative language for smart contracts

that brings the following benefits:

- Safety property **run-time verification**
- Executable **code generation**
- Debugging interface via **data provenance**

Why a **declarative** language?

Observation 1: smart contracts are managing **relational databases.**

Transaction records are stored as relational tables on block chain:
- every row is a transaction
- each column is a transaction parameter

mint

receiver	amount
0x1234	100

burn

account	amount

transfer

sender	receiver	amount

Why a **declarative** language?

Observation 2: smart contract operations and contract-level properties can be naturally expressed as **relational constraints**, e.g.:

- **Balance** is the sum of income subtracted by sum of expense.

transfer

sender	receiver	amount
0x01	0x02	100
…		
0x01	0x03	200
0x01	0x04	120
…		

transfer

sender	receiver	amount
…		
0x05	**0x01**	500
…		
0x06	**0x01**	120
0x07	**0x01**	400

Why a **declarative** language?

Observation 2: Smart contract operations and contract-level properties can be naturally expressed as **relational constraints**, e.g.:

- **Balance** is the sum of income subtracted by sum of expense.

transfer

sender	receiver	amount
0x01	0x02	100
…		
0x01	0x03	200
0x01	0x04	120
…		

transfer

sender	receiver	amount
…		
0x05	**0x01**	500
…		
0x06	**0x01**	120
0x07	**0x01**	400

Sum: income of 0x01

Sum: expense of 0x01

Why a **declarative** language?

Observation 2: Smart contract operations and contract-level properties can be naturally expressed as **relational constraints**, e.g.:

- **Property**: all account balances add up to total supply of tokens. It can be specified as the following query:

balance

account	amount
…	…

totalSupply

amount
n

Sum of amount = n ?

Why a **declarative** language?

Observation 1: smart contracts are managing relational **databases.**

Observation 2: smart contract operations and contract-level properties can be naturally expressed as relational constraints.

Smart contracts can be implemented declaratively, the same way as Database queries are specified in **Datalog**.

Declarative smart contracts

1. How to specify smart contracts in DeCon
2. Executable code generation (paper)
3. Data provenance (paper)
4. Evaluation

Example: Wallet

Wallet is a smart contract that manages digital tokens:

- Supports three kinds of transactions: mint, burn, and transfer.
- Each kind of transaction records are stored in a **relational table**.

mint

receiver	amount
0x1234	100

burn

account	amount

transfer

sender	receiver	amount

Each call of mint / burn / transfer function will append an entry to the corresponding table.

Example: Wallet

DeCon consists of two major components:

1. Declare relations (table schema)
2. Specify transactions and views (in rules)

Example: Wallet

1. Declare relations (table schema):

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)
```

table name

column names followed by types

Example: Wallet

1. Declare relations (table schema):

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)
```

Relations with "recv_" prefix are transaction event triggers.

Example: Wallet

1. Declare relations (table schema):

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)
```

⇩

Each relation declaration with "recv_" prefix is compiled into a transaction interface:

```
function mint(address p, int amount) public returns Bool
function burn(address p, int amount) public returns Bool
function transfer(address from, address to, int amount) public returns Bool
```

function arguments are the relation schema

returns a Bool indicating the success of the transaction.

Example: Wallet

Other special relation annotations:

```
// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,balanceOf
```

* annotates singleton relation, which only has one row.

The first field (p) is the primary key.

Declare public interfaces

Primary keys uniquely identify a row: inserting a row will update the row with the same primary key.

Example: Wallet

1. Declare relations (table schema):

```
// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,balanceOf
```

Public relations are compiled into smart contract **view** functions:

```
function totalSupply() public view returns int
function balanceOf(address p) public view returns int
```

function argument is the primary key(s)

return values are the remaining fields

Example: Wallet

DeCon consists of two major components:

1. Declare relations (table schema)
2. Specify transactions and views (in rules)

Example: Wallet

A **transaction rule** is a rule with transaction event trigger (“recv_” prefix)

It specifies the transaction processing logic:

```
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),n>0.
```

Receive a transaction to mint **n** tokens to address **p.**

1. Receive a function call.

Example: Wallet

A **transaction rule** is a rule with transaction event trigger ("recv_" prefix)

It specifies the transaction processing logic:

```
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),n>0.
```

The message sender is contract owner.

The transaction amount n > 0.

1. Receive a function call
2. Check parameters against internal states.

Example: Wallet

A **transaction rule** is a rule with transaction event trigger (“recv_” prefix)

It specifies the transaction processing logic:

```
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),n>0.
```

Add a row (p,n) into mint table.

1. Receive a function call
2. Check parameters against internal states.
3. If checks are OK. Commit the transaction by adding a new row to the relational table.

Example: Wallet

View rules: rules other than transaction rules.

totalSupply is allMint - allBurn

```
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
```

sum of all mint transaction amounts.

sum of all burn transaction amounts.

Example: Wallet

View rules: rules other than transaction rules.

totalSupply is allMint - allBurn

```
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
```

sum of all mint transaction amounts.

sum of all burn transaction amounts.

mint

receiver	amount
0x1234	100

burn

account	amount

Example: Wallet

View rules: rules other than transaction rules.

```
r4:  totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
```

```
r10: allMint(s) :- s = sum n: mint(_,n).
r11: allBurn(s) :- s = sum n: burn(_,n).
```

mint

receiver	amount
0x1234	100

burn

account	amount

Example: Wallet

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)

// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,balanceOf

// Transaction rules
.decl mint(p: address, amount: int)
.decl burn(p: address, amount: int)
.decl transfer(from: address, to: address, n: int)
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),
            n>0.
r2: burn(p,n):-recv_burn(p,n),msgSender(s),owner(s),
            balanceOf(p,m), n<=m.
r3: transfer(s,r,n) :- recv_transfer(s,r,n),
                   balanceOf(s,m),m>=n, n>0.
```

```
// View rules
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.

// Auxiliary relations and rules ...
.decl totalMint(p: address, n: int)[0]
.decl totalBurn(p: address, n: int)[0]
r6: transfer(0,p,n) :- mint(p,n).
r7: transfer(p,0,n) :- burn(p,n).
r8: totalOut(p,s):-transfer(p,_,_),
              s=sum n:transfer(p,_,n).
r9: totalIn(p,s):-transfer(_,p,_),
             s=sum n:transfer(_,p,n).
.decl *allMint(n: int)
.decl *allBurn(n: int)
r10: allMint(s) :- s = sum n: mint(_,n).
r11: allBurn(s) :- s = sum n: burn(_,n).
```

Example: Wallet

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)

// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,

// Transaction rules
.decl mint(p: addres
.decl burn(p: address, am        int)
.decl transfer(from: address, to: address, n: int)
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),
               n>0.
r2: burn(p,n):-recv_burn(p,n),msgSender(s),owner(s),
               balanceOf(p,m), n<=m.
r3: transfer(s,r,n) :- recv_transfer(s,r,n),
                       balanceOf(s,m),m>=n, n>0.
```

Transaction rules are only triggered when a transaction is received.

```
// View rules
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.

// Auxiliary relations and rules ...
.decl totalMint(p: address, n: int)[0]
.decl totalBurn(p: address, n: int)[0]
r6: transfer(0,p,n) :- mint(p,n).
r7: transfer(p,0,n) :- burn(p,n).
r8: totalOut(p,s):-transfer(p,_,_),
                   s=sum n:transfer(p,_,n).
r9: totalIn(p,s):-transfer(_,p,_),
                  s=sum n:transfer(_,p,n).
.decl *allMint(n: int)
.decl *allBurn(n: int)
r10: allMint(s) :- s = sum n: mint(_,n).
r11: allBurn(s) :- s = sum n: burn(_,n).
```

Example: Wallet

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)

// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,balanceOf

// Transaction rules
.decl mint(p: address, amount: int)
.decl burn(p: address, amount: int)
.decl transfer(from: address, to: address, n: int)
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),
              n>0.
r2: burn(p,n):-recv_burn(p,n),msgSender(s),owner(s),
              balanceOf(p,m), n<=m.
r3: transfer(s,r,n) :- recv_transfer(s,r,n),
                    balanceOf(s,m),m>=n, n>0.
```

```
// View rules
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.

// Auxiliary relations and rules ...
.decl totalMint(p: address, n: int)[0]
.decl totalBurn(p: address, n: int)[0]
r6: transfer(0,p,n) :- mint(p,n).
r7: transfer(p,0,n) :- burn(p,n).
r8: totalOut(p,s):-transfer(p,_,_),
                   s=sum n:transfer(p,_,n).
r9: totalIn(p,s):-transfer(_,p,_),
                  s=sum n:transfer(_,p,n).
.decl *allMint(n: int)
.decl *allBurn(n: int)
r10: allMint(s) :- s = sum n: mint(_,n).
r11: allBurn(s) :- s = sum n: burn(_,n).
```

Each rule's derivation result add entries to the relational table.

Example: Wallet

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address,to:address,n:int)

// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,balanceOf

// Transaction rules
.decl mint(p: address, amount: int)
.decl burn(p: address, amount: int)
.decl transfer(from: address, to: address, n: int)
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),
              n>0.
r2: burn(p,n):-recv_burn(p,n),msgSender(s),owner(s),
              balanceOf(p,m), n<=m.
r3: transfer(s,r,n) :- recv_transfer(s,r,n),
                     balanceOf(s,m),m>=n, n>0.
```

```
// View rules
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.

// Auxiliary relations and rules ...
.decl totalMint(p: address, n: int)[0]
.decl totalBurn(p: address, n: int)[0]
r6: transfer(0,p,n) :- mint(p,n).
r7: transfer(p,0,n) :- burn(p,n).
r8: totalOut(p,s):-transfer(p,_,_),
                 s=sum n:transfer(p,_,n).
r9: totalIn(p,s):-transfer(_,p,_),
                s=sum n:transfer(_,p,n).
.decl *allMint(n: int)
.decl *allBurn(n: int)
r10: allMint(s) :- s = sum n: mint(_,n).
r11: allBurn(s) :- s = sum n: burn(_,n).
```

The chain of updates continue until no new tuples can be inserted.

Views are updated when any relation in the body is updated.

Property specification

Properties are specified in the same way as views, but with a violation annotation.

```
.decl negativeBalance(p:address,n:int)[0]
.violation negativeBalance
r14: negativeBalance(p,n) :- balanceOf(p,n), n < 0.
```

Safety means that violation relations are empty after every transaction commit.

Monitoring properties in run-time

```
.decl negativeBalance(p:address,n:int)[0]
.violation negativeBalance
r14: negativeBalance(p,n) :- balanceOf(p,n), n < 0.
```

Generates the following instrumentation block:

```
function checkViolations() {
    if negativeBalance is not empty:
        revert("Negative balance.")
    // check other violations...
}
```

Evaluation

Measure overhead in two ways:

1. compared to reference Solidity implementation.
2. introduced by run-time verification.

Gas: a metric used by Ethereum smart contract to measure the execution cost. Reading or writing to memory consumes most gas.

Execution overhead

Contract	LOC	# Functions	# Rules	Byte-code size (KB)		Transaction	Gas cost (K)		
				Reference	DeCon		Reference	Compiled	Diff
Wallet	57	6	12	3	3	mint	36	62	70%
						burn	36	47	29%
						transfer	52	38	-26%
Crowdsale	70	5	11	4	3	invest	38	33	-12%
						close	38	47	25%
						withdraw	26	29	14%
						claimRefund	29	33	13%
SimpleAuction	139	3	13	2	4	bid	69	115	66%
						withdraw	24	47	101%
						auctionEnd	54	56	4%
ERC721	447	9	13	10	11	transferFrom	59	42	-28%
						approve	49	75	53%
						setApprovalForAll	27	27	2%
ERC20	383	6	18	5	6	transfer	52	55	6%
						approve	47	50	7%
						transferFrom	43	50	15%
								median:	**14%**

Run-time verification overhead

Contract	Property	Size	Transaction	Gas
Wallet	No negative balance	2	mint	14%
			burn	14%
			transfer	17%
Crowdsale	No missing funds	2	invest	50%
			close	24%
			withdraw	22%
			claimRefund	33%
Simple Auction	Refund once	2	bid	2%
			withdraw	60%
			auctionEnd	4%
ERC721	Every token has owner	1	transferFrom	5%
			approve	3%
			setApprovalForAll	8%
ERC20	Account balances add up to total supply	1	transfer	96%
			approve	13%
			transferFrom	109%
			median:	16%

Summary

- DeCon shows that smart contracts can be naturally expressed as relational queries.
- DeCon can:
  - automatically generate Solidity code from declarative rules.
  - verify safety properties during run-time.
  - support data-provenance for intuitive debugging.
- DeCon has moderate overhead over reference Solidity implementation.

Future work

- Static verification of DeCon contracts:

Could we exploit the high-level abstraction of DeCon to perform efficient static verification?

- Gas optimization.

Could the DeCon compiler generate more efficient code?

Checkout DeCon at:

https://github.com/HaoxianChen/declarative-smart-contracts