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.

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 programming language.

A **contract** is like a **class** in Java.

**Contract deployment** is like **class instantiation**.
Smart contracts today

```solidity
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 programming language.
A **contract** is like a class in Java.

Contract states declaration.
Smart contracts today

```solidity
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 programming language.
A **contract** is like a class in Java.

Contract states declaration.

**Transactions** are public functions that alter the contract states.
Smart contracts today

```solidity
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 programming 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

<table>
<thead>
<tr>
<th>mint</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>receiver</td>
<td>amount</td>
</tr>
<tr>
<td>0x1234</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>burn</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>amount</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>transfer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>receiver</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x02</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x01</td>
<td>0x03</td>
<td>200</td>
</tr>
<tr>
<td>0x01</td>
<td>0x04</td>
<td>120</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05</td>
<td>0x01</td>
<td>500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x06</td>
<td>0x01</td>
<td>120</td>
</tr>
<tr>
<td>0x07</td>
<td>0x01</td>
<td>400</td>
</tr>
</tbody>
</table>
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.

```
<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x02</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x01</td>
<td>0x03</td>
<td>200</td>
</tr>
<tr>
<td>0x01</td>
<td>0x04</td>
<td>120</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x05</td>
<td>0x01</td>
<td>500</td>
</tr>
<tr>
<td>0x06</td>
<td>0x01</td>
<td>120</td>
</tr>
<tr>
<td>0x07</td>
<td>0x01</td>
<td>400</td>
</tr>
</tbody>
</table>
```

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:

  \[
  \text{Sum of amount} = n ?
  \]
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.

```
<table>
<thead>
<tr>
<th>receiver</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td>100</td>
</tr>
</tbody>
</table>
```

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):

```plaintext
// 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: `recv_mint`, `recv_burn`, `recv_transfer`
Column names: `p`, `amount`, `from`, `to`, `n`
Column types: `address`, `int`
Example: Wallet

1. Declare relations (table schema):

```plaintext
// 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):

```plaintext
// 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:

- `mint(address p, int amount) public` returns Bool
- `burn(address p, int amount) public` returns Bool
- `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:

```java
// 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):

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

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

```plaintext
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:

\[ r1: \text{mint}(p,n):-\text{recv\_mint}(p,n),\text{msg\_Sender}(s),\text{owner}(s),n>0. \]

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:

$r1: \text{mint}(p,n) : \neg \text{recv}_\text{mint}(p,n), \text{msgSender}(s), \text{owner}(s), n > 0.$

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

- The message sender is contract owner.
- The transaction amount $n > 0$. 
Example: Wallet

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

It specifies the transaction processing logic:

\[ r1: \text{mint}(p,n): - \text{recv}_\text{mint}(p,n), \text{msgSender}(s), \text{owner}(s), n > 0. \]

- 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.

\[ r4: \text{totalSupply}(n) :- \text{allMint}(m), \text{allBurn}(b), n := m - b. \]

- Total supply is \( \text{allMint} - \text{allBurn} \)
- Sum of all mint transaction amounts.
- Sum of all burn transaction amounts.
Example: Wallet

**View rules**: rules other than transaction rules.

\[ r4: \text{totalSupply}(n):=\text{allMint}(m),\text{allBurn}(b), n:=m-b. \]

- **mint transaction amounts**: sum of all mint transaction amounts.
- **burn transaction amounts**: sum of all burn transaction amounts.

```
receiver | amount
---------|-------
0x1234   | 100
```

**totalSupply is allMint - allBurn**
Example: Wallet

**View rules:** rules other than transaction rules.

\[ r4: \text{totalSupply}(n):=\text{allMint}(m),\text{allBurn}(b),n:=m-b. \]

\[ r10: \text{allMint}(s) :- s = \text{sum } n: \text{mint}(_,n). \]
\[ r11: \text{allBurn}(s) :- s = \text{sum } n: \text{burn}(_,n). \]
Example: Wallet

```plaintext
// 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 rules are only triggered when a transaction is received.
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(θ, p, n) := mint(p, n).
.r7: transfer(p, θ, 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.
The chain of updates continue until no new tuples can be inserted.
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) :- balance0f(p, n), n < 0.
```

Safety means that violation relations are empty after every transaction commit.
Monitoring properties in run-time

Generates the following instrumentation block:

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

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

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th># Functions</th>
<th># Rules</th>
<th>Byte-code size (KB)</th>
<th>Transaction</th>
<th>Gas cost (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reference</td>
<td>DeCon</td>
<td>Transaction</td>
</tr>
<tr>
<td>Wallet</td>
<td>57</td>
<td>6</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>mint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>burn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>transfer</td>
</tr>
<tr>
<td>Crowdsale</td>
<td>70</td>
<td>5</td>
<td>11</td>
<td>4</td>
<td>3</td>
<td>invest</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>close</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>withdraw</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>claimRefund</td>
</tr>
<tr>
<td>SimpleAuction</td>
<td>139</td>
<td>3</td>
<td>13</td>
<td>2</td>
<td>4</td>
<td>bid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>withdraw</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>auctionEnd</td>
</tr>
<tr>
<td>ERC721</td>
<td>447</td>
<td>9</td>
<td>13</td>
<td>10</td>
<td>11</td>
<td>transferFrom approve</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>setApprovalForAll</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>transferForAll</td>
</tr>
<tr>
<td>ERC20</td>
<td>383</td>
<td>6</td>
<td>18</td>
<td>5</td>
<td>6</td>
<td>transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>approve</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>transferFrom</td>
</tr>
</tbody>
</table>

**median:** 14%
### Run-time verification overhead

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