Declarative 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

ABSTRACT

This paper presents *DeCon*, a declarative programming language for implementing smart contracts and specifying contract-level properties. Driven by the observation that smart contract operations and contract-level properties can be naturally expressed as relational constraints, DeCon models each smart contract as a set of relational tables that store transaction records. This relational representation of smart contracts enables convenient specification of contract properties, facilitates run-time monitoring of potential property violations, and brings clarity to contract debugging via data provenance. Specifically, a DeCon program consists of a set of declarative rules and violation query rules over the relational representation, describing the smart contract implementation and contract-level properties, respectively. We have developed a tool that can compile DeCon programs into executable Solidity programs, with instrumentation for run-time property monitoring. Our case studies demonstrate that DeCon can implement realistic smart contracts such as ERC20 and ERC721 digital tokens. Our evaluation results reveal the marginal overhead of DeCon compared to the open-source reference implementation, incurring 14% median gas overhead for execution, and another 16% median gas overhead for run-time verification.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

Smart contracts, Declarative programming, Run-time verification

ACM Reference Format:

Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. 2022. Declarative Smart Contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549121

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549121

1 INTRODUCTION

Smart contracts are programs stored and executed on blockchains. They have been used in a wide range of blockchain-enabled distributed applications to manage digital assets, including auctions [35], financial contracts [23], elections [47], trading platforms [50], and permission management [19]. Unfortunately, today's smart contracts are error-prone, and this has led to significant financial losses resulting from attacks such as Dice2win [51], King of Ether [2], Parity Multisig Bug [61], Accidental [24] and DAO [1, 58].

Over the past few years, different analysis and verification techniques have been proposed for known vulnerabilities of smart contracts, such as re-entrancy attacks and transaction-order dependency [17, 32, 52, 56, 63]. However, when it comes to high-level properties specific to individual smart contracts, programmers typically have to rely on hand-written assertions [10], which is hard to maintain and error-prone. For example, given a smart contract that manages digital tokens, one may want to ensure that all account balances add up to the total supply of tokens. To monitor this property during run-time, one has to instrument the code to maintain a state that keeps track of the sum of all account balances, and add assertions about their equivalence wherever either account balances or token supplies are updated. There are third-party tools that support high-level property specification and verification for Solidity, e.g., temporal logic [52] and formula with extended operators [44]. However, counter-examples are returned in the form of Ethereum bytecode traces or transaction sequences, which may not be easy for programmers to understand and localize bugs in the original implementation.

To make smart contracts easier to analyze and verify, this paper presents *DeCon*, a declarative programming language for smart contract implementation and property specification. DeCon is based on Datalog [15], a declarative logic programming language that syntactically is a subset of Prolog. Datalog frees programmers from low-level implementation details, e.g., data structures, algorithms, etc., and allows them to reason about the contract on the specification level via inference rules [22]. In addition, such relational representation serves as a high-level abstraction of the contract, which enables efficient formal analysis and verification [59, 64].

A typical smart contract provides two kinds of interfaces: *transactions* and *views*. Transactions are function calls that alter the contract states, e.g., a token transfer that updates both sender and recipient balances. Views are read-only functions that return particular states of the contract, e.g., the balance of an account.

Smart contract properties and operations can be naturally mapped to relational logic. For example, transactions, the main element in smart contracts, can be modeled as relational tables, where the table schema contains transaction parameters, e.g., sender, recipient, and amount. Similarly, the balance of each account can be expressed as sum aggregation on transaction records and looking up an account balance can be expressed as a constraint on the address column of the balance table.

Given this relational view of transactions, committing a transaction can be interpreted as appending a new row to the corresponding table. The commit and abortion logic of a pending transaction is specified by Datalog-based declarative rules. Views can then be specified as declarative queries on these tables. For example, an account balance is the total income of the account subtracted by its total expense, each of which is a query on relevant transaction records.

Contract properties are also specified as inference rules. They are interpreted as property violation queries, a special kind of views, and are expected to be always empty during correct executions. For example, if a smart contract forbids overspending, then a query on accounts with negative balances should always be empty. Such unification of implementation and property specification language saves programmers' effort to learn another language to formally specify properties.

DeCon complies declarative specifications into executable Solidity [11] programs that run on blockchains, e.g., Ethereum, and monitor the specified properties at run-time. When a property (violation) view is derived non-empty after executing a pending transaction, the transaction is aborted. Such automatic code generation not only saves implementation effort, but also eliminates the gap between the program specification and implementation, providing a stronger guarantee of the verification result.

The key insight to generate efficient executable code from declarative specifications is that smart contract transactions are executed in sequence. In other words, new rows are appended to the transaction tables one at a time. Therefore, DeCon borrows the idea of incremental view maintenance in databases [34] to generate efficient update procedures. On committing a new transaction, instead of evaluating the queries on the whole tables, only the differences in query results are computed and applied to existing views.

In addition, DeCon is easy to debug with data provenance [25]. Provenance is a mechanism for explaining how certain tuples or facts are derived, right down to the input values. In an imperative language like Solidity [11], dependency information is difficult to be captured automatically (through data-flow analysis). In contrast, inference rules in DeCon give explicit dependency information, where each tuple can be directly attributed to one rule, thus providing more clarity to the execution process.

The key contributions of the paper are as follows:

- We design DeCon, a declarative language that unifies smart contract implementation and specification. We demonstrate its expressiveness via case studies on representative smart contracts and their high-level correctness properties.
- We design an algorithm to compile these high-level specifications into executable Solidity programs, with instrumentation for run-time verification.
- We implement and experimentally evaluate DeCon. Our evaluation shows that the generated executable code has comparable efficiency with the equivalent open-source implementation of the same contract (14% median gas overhead), and the overhead of run-time verification is moderate (16% median gas overhead). The prototype implementation and evaluation benchmarks are open-sourced [5] for future studies and comparisons.

The rest of this paper is organized as follows. Section 2 motivates DeCon using a Wallet example. The declarative smart contract language is presented in Section 3. Section 4 demonstrates the translation of declarative rules into an executable Solidity program. The expressiveness of DeCon is presented in Section 5 using two case studies. Section 6 experimentally evaluates DeCon. Section 7 discusses related work, and Section 8 concludes the paper.

2 ILLUSTRATIVE EXAMPLE

In this section, we show how to use DeCon to implement a smart contract, specify its properties, and debug via provenance using a Wallet smart contract that manages digital tokens.

2.1 Contract Implementation

A smart contract offers two kinds of interfaces: *transactions* and *views*. Transactions are the function calls that update the contract states. On the other hand, views are read-only functions that return one or more contract states.

In declarative smart contracts, transaction records are the only states. Transactions are modeled as relational tables. A new row is appended to the table when a new transaction is committed, with column entries storing the transaction parameters. Transaction rules, i.e., the condition on which a new transaction can be committed, are specified as declarative rules. Finally, views are specified as declarative queries over the transaction tables.

We use the Wallet example, shown in Listing 1, to explain how relational tables and declarative rules can be specified. The Wallet contract manages token transactions between Ethereum addresses, where the contract owner can mint or burn tokens to addresses, and different addresses can transfer tokens to each other.

Relations and interfaces. Lines 1 to 14 declare the relations, with schema in the parenthesis, and, optionally, primary key indices in the bracket (e.g., `balanceOf` on line 8). Primary keys uniquely identify a row in the table. For instance, `balanceOf` records the balance of each account, and thus has a unique account column. Without explicit specification, all columns are treated as primary keys. Relation `totalSupply` (line 7) is a singleton relation, a kind of relation that contains only one row and is annotated by a star symbol.

Given these relation declarations, transaction and view interfaces are generated. First, transaction interfaces are generated from relations with `recv_` prefix, where the input parameters define the schema and a Boolean return value indicates the success of the transaction. For example, relation `recv_mint` is translated into the following interface in Solidity, the target executable language.

```
function mint(address p, int amount) returns (bool);
```

Second, view functions are generated from the relations that appear in the public interface annotations (line 9). The input parameters

```
1  // Transaction event triggers
2  .decl recv_mint(p:address, amount:int)
3  .decl recv_burn(p:address, amount:int)
4  .decl recv_transfer(from:address,to:address,n:int)
5  
6  // Views
7  .decl *totalSupply(n:int)
8  .decl balanceOf(p:address, n:int)[0]
9  .public totalSupply,balanceOf
10 
11 // Transaction rules
12 .decl mint(p: address, amount: int)
13 .decl burn(p: address, amount: int)
14 .decl transfer(from: address, to: address, n: int)
15 r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),
16                n>0.
17 r2: burn(p,n):-recv_burn(p,n),msgSender(s),owner(s),
18                balanceOf(p,m), n<=m.
19 r3: transfer(s,r,n) :- recv_transfer(s,r,n),
20                        balanceOf(s,m),m>=n, n>0.
21 
22 // View rules
23 r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
24 r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.
25 
26 // Auxiliary relations and rules ...
27 .decl totalMint(p: address, n: int)[0]
28 .decl totalBurn(p: address, n: int)[0]
29 r6: transfer(0,p,n) :- mint(p,n).
30 r7: transfer(p,0,n) :- burn(p,n).
31 r8: totalOut(p,s):-transfer(p,_,_),
32                    s=sum n:transfer(p,_,n).
33 r9: totalIn(p,s):-transfer(_,p,_),
34                   s=sum n:transfer(_,p,n).
35 .decl *allMint(n: int)
36 .decl *allBurn(n: int)
37 r10: allMint(s) :- s = sum n: mint(_,n).
38 r11: allBurn(s) :- s = sum n: burn(_,n).
```

Listing 1: Wallet smart contract

are the primary keys, and the output is the remaining values. Note that since a singleton relation, e.g., `totalSupply`, has no primary keys, it becomes a function without parameters. If all columns are primary keys, then the function returns a Boolean value indicating the existence of the row. For example, `balanceOf(p:address, n:int)[0]` is translated into the following function interface.

```
    function balanceOf(address p) returns (int);
```

Rules and functions. The rest of the program shows the rules that process transactions and define the views. Each rule is of the form `<head> :- <body>`, interpreted as follows. For all valuation of the variables that satisfy all constraints in the body, generate a row as specified in the head. For example, `r1` on line 15 says that a `mint` transaction can only be sent by the contract owner, and the amount should always greater than 0. This rule is compiled into the following Solidity code (with simplification).

```
function mint(address p, int n) (returns bool) {
    bool ret = false;
    if (msg.sender == owner && n>0) {
        // call functions to update dependent views...
        ret = true;
    }
    return ret;
}
```

When a `mint` transaction is committed, `r5` will be triggered through a chain of rules (`r1->r6->r9->r5`). It specifies the balance of an account `p`, as the total income `totalIn(p,i)` subtracted by the total expense `totalOut(p,o)`, with `totalIn` and `totalOut` further defined by `r8` and `r9`, respectively. This rule is compiled into two Solidity functions, each updates `balanceOf[p]` when either `totalIn` or `totalOut` is updated.

```
function updateBalanceOfOnTotalIn(address p, int i) {
    int o = totalOut[p];
    balanceOf[p] = i-o;
}
function updateBalanceOfOnTotalOut(address p, int o) {
    int i = totalIn[p];
    balanceOf[p] = i-o;
}
```

To get the balance of a given account, one could call `balanceOf`, a view function that takes the account address as a parameter, and returns an integer value as the account balance. In DeCon, relational tables are stored in maps, mapping primary keys to values in the remaining columns. This view function is generated as follows.

```
function balanceOf(address p) public view returns (int)
     {
    // Read the row by primary key p
    BalanceOfTuple memory balanceOfTuple = balanceOf[p];
    // Return the value
    return balanceOfTuple.n;
}
```

2.2 Specification and Run-Time Verification

In DeCon, properties are specified the same way as views, but with additional annotation. For example, in the Wallet contract, one may want to make sure that all account balances are always non-negative, which can be specified as follows.

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

Rule `r14` specifies the violation instance of the property: for each row in `balanceOf` table with `n<0`, insert a row `(p,n)` in `negativeBalance` table. During the execution of the transaction, the `negativeBalance` table is incrementally updated when its dependent relations are updated, the same as other views.

The keyword `.violation` annotates that every row in the table is a property violation instance. A property is satisfied if and only if its corresponding *violation* table is empty. Given such annotations, DeCon instruments the program to check the emptiness of all violation tables before each transaction is committed.

Note that properties are monitored on the granularity of transactions. As we show in Section 4, due to the underlying update procedure, transient violations could occur during the execution of a transaction, but disappear at the end. Therefore, instead of aborting right after a violation tuple is derived, a transaction is only aborted if, at the end of its execution, any violation table remains non-empty. Such interpretation allows programmers to reason at the transaction level, without worrying about the underlying update procedure.

The violation checking procedure is generated and performed at the end of each transaction. In this example, the `negativeBalance` violation is checked as follows.

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

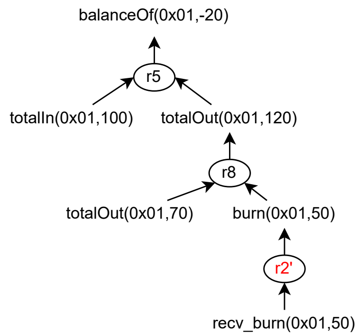


Figure 1: Provenance of a violation of negative balance

2.3 Debugging via Provenance

Data provenance is a feature of declarative programs that records the data flow from input to output and enables rule-wise debugging. It allows counter-example traces to be presented in the context of the original specification, instead of the low-level EVM instructions, thus making the debugging process more intuitive.

Suppose the original program has an incorrect `r2`, which misses a predicate to check that the account has enough balance to be burnt. The incorrect version of `r2` is shown as `r2'` in the following.

```
r2': burn(p,n):-recv_burn(p,n),msgSender(s),owner(s).
```

An account with a balance of n would have a negative balance if more than n tokens are burnt. Suppose during execution, the account `0x01` is detected to have a negative balance of −20. To understand why this violation happens, one could query the violation tuple's provenance tree, as shown in Figure 1. The provenance tree is read from top to bottom. On the top is a tuple `balanceOf(0x01,-20)` that triggers the violation in `negativeBalance`. Below shows that it is derived by `r5`, based on `totalIn(0x01,100)` and `totalOut(0x01,120)`, which are the total tokens received and sent by address `0x01`. The tuple `totalOut(0x01,120)` is further derived by `r8`. This back-tracing continues for another step until one finds the derivation of `r2'` is incorrect, which suggests that the condition `balanceOf(p,m),m>=n` should be added. With this provenance, programmers can debug contracts in a visual and interactive manner.

3 LANGUAGE

A DeCon contract consists of three elements: relations, rules, and relation annotations. A *relation* declaration specifies the name of a relational table and its schema. Each relational table can store either transaction records, with the transaction parameters being the column values, or views, the summary information of these transaction records. A *rule* specifies either the conditions on which a new transaction gets approved or the derivation of a view from the transaction records. Finally, *relation annotations* specify whether a relational table is a public view or a violation. Public views are compiled into public interfaces that take the relation's primary keys as parameters and return the remaining values in the matching row. Violations will be monitored during run-time, and a transaction is reverted if the violation relation is non-empty after the transaction execution.

$$\begin{array}{rcl}
(Type)\ T & := & int \mid uint \mid bool \mid address \\
(Schema)\ S & := & c1 : T1, c2 : T2, \ldots \\
(Primary\ keys)\ K & := & k1, k2, \ldots \\
(Reserved\ relation)\ RS & & \\
(Singleton\ relation)\ SG & := & .decl\ *r(Schema) \\
(Simple\ relation)\ SP & := & .decl\ r(Schema)[K] \\
(Transaction\ relation)\ TR & := & .decl\ recv_[r](Schema) \\
(Relation)\ R & := & RS \mid SG \mid SP \\
(Annotation)\ A & := & .public\ R \mid .violation\ R
\end{array}$$

Figure 2: Syntax of relation declarations and annotations

3.1 Relation Declarations and Annotations

The formal syntax of relation declarations and annotations is defined in Figure 2.

Schema. Schema of a relation is specified as a list of $c_i : T_i$, where c_i is the column name for the i-th column, and T_i is the data type.

Primary keys. Primary keys K are a list of indices in the relation schema. Specifying Primary keys is optional. If a simple relation is specified without primary keys, then all columns are treated as primary keys. Primary keys uniquely identify a row in each table. On inserting a new row, if an existing row has the same primary key, the existing row is replaced by the new row.

Singleton relations are relations with only one row, which are annotated with $*$ in the specification. When a new row is inserted into a singleton relation, it replaces the existing row.

Transaction relations are relations with prefix `recv_`. As explained in the next section, these relations are treated as event triggers when used in a rule, and are compiled into smart contract interfaces that handle incoming transaction requests.

Reserved relations. The following relations are reserved to handle smart contract-specific constructs:

- `msgSender(a:address)` stores the address of message sender.
- `msgValue(v:uint)` stores the values of Ethers sent along a message.
- `send(to:address, n:uint32)` triggers a transaction that sends n Ethers to another account.
- `constructor(*)` is translated into the constructor function, with schema being function parameters.

3.2 Rules

As shown in Figure 3, we distinguish two kinds of rules: transaction rules and view rules. A *transaction rule* contains a transaction relation in its body. Transaction relations are relations with a prefix *recv_* in names. These rules are only fired on receiving the corresponding transaction request, and the transaction is approved if the rest of the constraints in the rule body are satisfied. On the other hand, a *view rule* does not contain any transaction relations. It is evaluated whenever one of the relations in the body is updated.

Syntax restrictions. DeCon does not support recursions. That is, no dependency loop exists between any two relations. The dependency relationship in DeCon is defined as follows.

Definition 3.1 (Relation dependency)**.** Relation R_a is dependent on relation R_b, if there exists a view rule where R_a is in the head

$$
\begin{aligned}
&(\mathit{Variable})\ x\\
&(\mathit{Aggregation})\ \mathit{Agg} := \ \mathit{sum} \mid \mathit{max} \mid \mathit{min} \mid \mathit{count}\\
&(\mathit{Function})\ F := \ + \mid - \mid \times \mid \div\\
&(\mathit{Condition})\ C := \ > \mid < \mid >= \mid <= \mid == \mid != \\
&(\mathit{Transaction\ relation})\ TR := \ \mathit{recv\_}[r]\\
&(\mathit{Other\ relation})\ R\\
&(\mathit{Head\ literal})\ h := \ R(\bar{X})\\
&(\mathit{Body\ literal})\ b := \ R(\bar{X}) \mid C(\bar{X}) \mid y = F(\bar{X})\\
&\qquad\qquad \mid y = \mathit{Agg}\ x : R(\bar{X})\\
&(\mathit{Transaction\ Rule})\ Tx := \ h\ :-\ TR(\bar{X}), b_1, ..., b_n\\
&(\mathit{View\ Rule})\ V := \ h\ :-\ b_1, ..., b_n
\end{aligned}
$$

Figure 3: Syntax of rules

and R_b is in the body, or a transaction rule where R_a is in the head, and R_b is a transaction relation (with a prefix `recv_`) in the body.

Rule semantics. A rule is evaluated as follows. For each variables valuation π that satisfies the *rule constraint*, generate the head tuple with all variables assigned to its corresponding values in π. A variable valuation is a mapping from the set of variable names V to the variable domain D ($\pi : V \rightarrow D$). Rule constraint is a conjunction of all body literal constraints. As described in Figure 3, there are four kinds of body literals. For literals in the form of relational tuples $R(\bar{X})$, the constraint is satisfied if row $\bar{X}$ exists in the relational table R. Other kinds of literals (i.e., conditions, functions, and aggregations) are directly interpreted as constraints on the variables.

Consider `r5` in the Wallet example (listing 1).

```
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.
```

This rule is interpreted as follows: "for all values of variable `p,o,i` such that there exists a tuple `totalOut(p,o)` and `totalIn(p,i)`, derive the head tuple `balanceOf(p,s)`, where `s = i-o`".

Aggregation literal $\mathit{Agg}\ x : R(\bar{X})$ computes the aggregate for all rows in relation R that satisfy the rule constraint. For example, in the Wallet example (listing 1), line 31 shows a rule with aggregation.

```
r8:totalOut(p,s) :- transfer(p,_,_),
                    s=sum n:transfer(p,_,n).
```

For each unique value of `p` in the first column of `transfer` table, this rule computes the sum of the third column for rows in `transfer` table that has the value `p` in the first column. In other words, this rule groups the table by the first column, and then computes the sum of the third column within each group.

Limitations in expressiveness. DeCon forbids recursions in order to keep the gas consumption predictable and affordable. In fact, recursion is not recommended by the Solidity documentation for stack space issues [7].

In addition, there are functions that lie outside of relational logic, e.g. cryptographic operations, randomized functions, etc. Such functions can be implemented in DeCon by linking the contract with external libraries. However, analyzing such functions is challenging and would need substantial future research. DeCon also does not support contract inheritance, dynamic dispatching, and checking interfaces of another contract. These features can be incorporated into DeCon in future compiler designs.

4 COMPILATION TO SOLIDITY

DeCon translates a set of declarative rules into an executable Solidity program that (1) processes transactions following the conditions in transaction rules, (2) updates views incrementally as new transactions are committed, and (3) monitors property violations.

The compilation process involves three major steps.

(1) Abstract update functions. First, each rule is translated into a set of abstract update functions, each of which performs incremental updates to the head relation when one of the body relations is updated. These functions are abstract in that they do not implement concrete data structures. For example, in the Wallet contract, the following rule processes `mint` transactions.

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

This rule is translated into the following abstract update function.

```
1    on insert recv_mint(p,n) {
2      search owner where  {
3        address s = owner;
4        search msgSender where s==msg.sender {
5          if(n>0) {
6            insert mint(p,n)
7          }}}}
```

This update function is triggered when a `mint` transaction is received, as indicated by the event trigger tuple `recv_mint(p,n)`. The remaining two relational literals, `owner(s)` and `msgSender(s)`, are translated into nested search statements (line 2 and line 4). A search statement has the form `(search R where C do S)`, where `R` is the relational table, `C` is the set of constraints on rows, and `S` is the statement to execute for each row that satisfies the constraints in `C`. The condition literal (`n>0`) is translated into an `if` statement (line 5). If all prior conditions are satisfied, we arrive at line 6, where the rule head is inserted.

(2) Data structures. These abstract functions are then translated into concrete Solidity statements, where the `search` statements become efficient join algorithms on concrete data structures, and update functions for dependent views are called after an `insert` statement.

(3) Instrumentation. In the last step, the Solidity program is instrumented to monitor property violations, and abort the transaction if any violation has been detected by the end of transaction execution.

4.1 Abstract Update Function Generation

There are two kinds of updates that could trigger a rule: tuple insertion and tuple deletion. We use *Insert(e)* and *Delete(e)* to denote the update trigger on inserting and deleting a tuple e, respectively. Note that both a tuple and a literal have the form $R(\bar{X})$. It is called a tuple when $\bar{X}$ has concrete values, and is called literal in a rule, where $\bar{X}$ is symbolic. We use literal and tuple interchangeably in the following discussion of update triggers.

Given a rule r, let $B(r)$ be the set of all relational literals in r's body, and e be the transaction relation in r if r is a transaction rule, the set of update triggers $T(r)$ are defined as:

$$T(r) := \begin{cases} \bigcup_{l \in B(r)} \{Insert(l), Delete(l)\} & r \text{ is View} \\ \{Insert(e)\} & r \text{ is Tx rule} \end{cases} \quad (1)$$

If r is a view rule, then it can be triggered by updates of any relation in its body. Otherwise, r is a transaction rule triggered only when receiving a transaction request.

Algorithm 1 UpdateFunction(r, t). Given a rule r, and a trigger t, returns an update object.

(1) Initialize the set of grounded variables $G := t.variables$.
(2) Literals other than the trigger $L := \{r.body \setminus t\}$.
(3) Update procedure $S :=$ Update$(r.head, L, t, G)$.
(4) Return (on t do S)

For each rule r, and for each update triggers in $T(r)$, an abstract update function is generated by UpdateFunction(r, t), presented in algorithm 1. It first initializes the set of grounded variables by variables in the trigger literal. Grounded variables are variables that are constrained to a constant value. Variables in a trigger literal are considered grounded because the update function is always triggered by the insertion or deletion of a concrete tuple. In step(3), update procedure S is generated by a sub-routine Update, which is presented in algorithm 2. Finally, it returns the abstract update function in the form of (`on t do S`), where t is the update trigger and S is the update procedure.

Algorithm 2 Update(h, L, t, G). Given a rule head h, a list of body literals L, an update trigger t, and the set of grounded variables G, return statements that perform the incremental update.

match L:
  case Nil => **match** t
    case Insert => **return** Insert(h)
    case Delete => **return** Delete(h)
  case $head :: tail$ =>
    Add grounded variables $G' := G \cup \{x | x \in head\}$
    Inner statements $S :=$ Update(h, tail, t, G')
    match head:
      case $R(\bar{X})$ =>
        Derive constraints $C :=$ Constraint$(R(\bar{X}), G)$
        return (Search R where C do S)
      case $C(\bar{X})$ => **return** (If C Then S)
      case $y = F(\bar{X})$ => **return** $(y = F(\bar{X}) :: S)$
      case $y = Agg\ x : R(\bar{X})$ =>
        return $(y = Agg\ x : R(\bar{X}) :: S)$

As shown in Algorithm 2, Update(h, L, t, G) performs recursion on L, the list of literals in the rule body, with every recursion translating one literal to a layer of code block, nested within the code block generated by the previous literals.

In particular, it performs pattern matching on input L, a list of literals to be translated. If L is empty, which means all body literals have been translated, an update statement that is consistent with the update trigger is returned. Otherwise, L has the form $head :: tail$. It first adds all variables in *head* into the set of grounded variables, and then generates the inner code blocks S by recursively calling itself on *tail* and the updated set of grounded variables G'. Depending on the form of *head*, the current layer of code block is generated in different ways. By the syntax of the language in Section 3, *head* could take one of the following forms:

- A relational literal $R(\bar{X})$. Given the set of grounded variables G, the search constraints for rows in R is generated as follows.

$$Constraints(R(\bar{X}), G) := \bigwedge \{(R[i] == v) | v \in G, v \in \bar{X}, i = \bar{X}.\text{indexOf}(v)\}$$

  where $R[i] == v$ means filtering rows in table R whose i-th column equals to v.
- A condition literal $C(\bar{X})$, in which case, the condition is directly used in the same way as an If condition, with the inner code block S placed within the If statement.
- A function or aggregation. In either case, the literal is directly translated into an assignment statement, followed by the inner code block S.

Aggregations. The evaluation results of aggregation functions are maintained incrementally. Sums are incremented by n when a row with aggregate value n is inserted, and decremented by n when a row is deleted. Similarly, counts are incremented by 1 on row insertion and decremented by 1 on row deletion. Maximums and minimums are slightly different. When a new row is inserted with value n, if n is greater than the current maximum, the maximum is updated to n. When the current maximum row is deleted, the maximum is updated as the second maximum value. Thus, it requires maintaining a sorted list of values. Minimum is maintained in a similar fashion.

4.2 Concrete Data Structures and Instructions

Given the abstract functions generated from each rule, the next step is to generate concrete and efficient data structures and search algorithms in the Solidity language.

Data structures. Each relational table `R`, except singleton relations, is translated into a mapping from its primary keys to a structure that stores the rest of the column values:

```
struct RTuple {
    bool valid;
    T1 field1;
    T2 field2;
    ...
};
mapping(k1 => k2 => ... => kn => RTuple) R;
```

By default, hash-maps in Solidity map all keys to zero. Therefore, a valid bit (`valid`) is introduced to indicate the existence of a tuple. Columns other than primary keys are the structure members. If all columns are primary keys, its structure only contains a valid bit.

Singleton relations are directly stored in a structure with columns being the structure members.

Join index. Join index is built for each search statement in the abstract update program. Given a search statement `Search R where C do S` in the abstract update program, if all primary keys of `R` are constrained to constant values, no join index is generated. The matching entry can be directly looked up by primary keys.

On the other hand, if, in some rules, not all primary keys of `R` are constrained to constant values, a join index is built as a map from the constrained keys to a list of unconstrained keys.

Suppose relation `R1(k1,k2,v1)` has primary keys `k1` and `k2`. As described above, table `R1` is stored as a map from primary keys to remaining values (`mapping(k1=>k2=>R1Tuple)`). Given a search statement `Search R1 where R1[0]==k1 do S`, where only one primary key `k1` is constrained, the join index for `R1` is built as the following.

```
struct R1KeyTuple {
    bool valid;
    T2 k2;
}
mapping(k1 => R1KeyTuple[]) R1Index;
```

where `R1Index` maps `k1` to a list of `R1KeyTuple`, which stores the value of the other primary key `k2`. During the join execution, to iterate all rows in `R1` that satisfy `R1[0] == k1`, it first looks up all `k2` in `R1KeyTuple[k1]`, and then for each `k2`, get the value in `R1[k1][k2]`.

Update dependent views. An insert or delete statement in the abstract update function is translated into two sets of Solidity instructions. The first set updates the corresponding data structure, and the second set calls the update functions for the dependent relations (Definition 3.1).

Inserting a relational tuple t_1 directly updates the map, as well as the join index if one exists. If a tuple t_0 with the same primary keys exists, all dependent views are updated by first calling deletion updates on t_0, and then the insertion updates on t_1. Insertion update refers to functions triggered by tuple insertion, and deletion update refers to functions triggered by tuple deletion. Otherwise, insertion updates are directly called. Since a Solidity mapping maps all keys to value zero by default, a tuple exists if its valid bit is set to true.

Deleting a relational tuple resets its valid bit to false. Then deletion updates are called for all dependent relations.

In this way, when a new transaction is committed, all dependent views are updated through this chain of update propagation. Since there is no recursion, i.e., dependency loop between relations, allowed in DeCon, update propagation is guaranteed to terminate.

Logging. Committed transactions are logged as Solidity Events [12], a more gas efficient storage than global memory, but can only be read offline. These events constitute all states of a DeCon contract, which enable offline analysis for further insights and potential bugs.

4.3 Run-Time Verification

Properties are specified as declarative rules that derive violation instances. Such relations are annotated with the keyword `violation`.

As introduced in Section 2.2, transient violations that occur during the transaction execution are not counted. To see why transient violations can occur, consider again the Wallet contract in Section 2, and a property that requires all account balances to add up to the total supply. The property can be specified as shown in Listing 2.

```
.violation unequalTotalSupply
r12: totalBalance(s) :- sum n: balanceOf(_,n).
r13: unequalTotalSupply(n,m):-totalSupply(n),
                              totalBalance(m),n!=m.
```

Listing 2: All account balances add up to total supply.

During the execution of a `mint` transaction, the `totalSupply` and the `totalBalance` are updated in sequence, which leads to a violation when one is updated before another, but the violation disappears when both are updated.

Given this notion of transient violations, instead of aborting the transaction right after a violation tuple is derived, the checking procedure is deferred to the end of transaction. If any violation view is non-empty, the transaction is aborted. Note that a Solidity mapping does not record its domain. Hence, a separate array of mapping keys are maintained and iterated for valid violation tuples.

4.4 Provenance Generation

To debug a violation, programmers can use data provenance to visualize the derivation process of a violation tuple. As shown in Figure 1, provenance is a directed graph with two kinds of vertices: tuples and rules. Edges from a tuple vertex to a rule vertex denote tuple reads, and edges from a rule vertex to a tuple vertex denote tuple derivations.

To generate this provenance graph during the rule evaluation procedure, two kinds of additional records are logged: tuple read `Read(tuple, rid)` and tuple derivation `Write(rid,tuple)`, where `rid` is a unique identifier for each rule. The `Read(tuple,rid)` is interpreted as an edge from `tuple` to the rule indexed by `rid`, and, conversely, `Write(rid,tuple)` is an edge from rule `rid` to `tuple`.

Note that in Solidity, a failed transaction reverts all instructions, including logging. When a transaction is reverted due to a property violation, the provenance logs would also be reverted. Therefore, to generate provenance for a violation tuple, the transaction needs to be executed in a local debugging environment instead of the deployment blockchain. This practice also saves storage space on the public blockchain.

4.5 Optimizations

To improve gas and storage efficiency, two optimizations have been applied to the generated codes.

Join order. Body literals in a rule are sorted by their iteration cost in an increasing order. First are reserved relations and singleton relations, since they need no iteration. Second are the relations whose primary keys have all appeared in proceeding literals. These literals can be searched via a direct mapping look-up, thus requiring no iterations either. Next are the rest of the relations, which are translated into loops. Finally come condition and function literals.

Storage space. Storage space on a blockchain is precious due to the high synchronization cost. Deriving relations on-demand, that is, delaying evaluating an inference rule until it is used, can save storage space, but may incur performance overhead. To achieve a balanced trade-off between time and space, DeCon only proactively derives and stores relations annotated as public views or violations, as well as relations that are read during their derivation. Other relations are derived on-demand. For example, in the Wallet contract in Section 2, relation `mint` only serves as an update trigger for dependent rules, which is never queried during the update of public views or violations. Therefore, when a `mint` tuple is generated by `r1`, it only triggers the update for dependent rules, but it is not written to the persistent storage.

5 CASE STUDIES

This section demonstrates the expressiveness of DeCon and the explainability of data provenance via case studies on two popular smart contracts: ERC20 and ERC721. For the sake of brevity, only a subset of rules is discussed. All contracts are available online [4].

5.1 ERC20

ERC20 [29] is a token standard for fungible tokens. Similar to the Wallet contract in Section 2, it also supports token transfers between users. In addition, it has an allowance mechanism, where users can allow other users to transfer their tokens, up to a certain amount called `allowance`. The allowance mechanism can be specified as follows.

```
r1: transferFrom(sender,receiver,spender,n) :-
        recv transferFrom(sender, receiver, n),
        /* Sender has enough balance. */
        balanceOf(sender,m), m>=n,
        /* Operator has enough allowance. */
        msgSender(spender),
        allowance(sender,spender,l),l>=n.
```

On receiving a `transferFrom` transaction, in addition to checking that the `sender` has enough balance (`m>=n`), the rule also requires the `spender` to have enough allowance to spend tokens on `sender`'s behalf (`l>=n`). The relation `transferFrom` represents transactions where the `spender` sends `n` tokens from the `sender` to the `receiver`.

The allowance of a `spender` on an account is specified as follows.

```
r2: spentTotal(o,s,m) :- transferFrom(o,_,s,_),
                          m = sum n: transferFrom(o,_,s,n).
r3: allowance(o,s,n) :- allowanceTotal(o,s,m),
                         spentTotal(o,s,l), n := m-l.
```

The relation `spentTotal` accounts the amount of tokens `m` that spender `s` has spent on behalf of the sender `o`. And `allowance` is derived by subtracting the total spending from the total allowance, an amount approved by the sender (defined in another rule).

Given the definition of `allowance` and the `spentTotal`, we can specify a property that a spender never overspends as the following:

```
.violation overSpent
overSpent(o,s,n,m) :- allowanceTotal(o,s,n),
                      spentTotal(o,s,m), m>n.
```

DeCon then generates instrumentation to monitor this property at run-time.

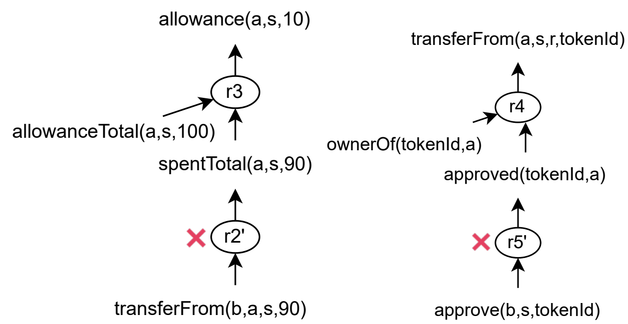
Explain allowance changes via data provenance. Suppose the programmer made a mistake in specifying `spentTotal`:

```
r2': spentTotal(r,s,m):-transferFrom(_,r,s,_),
            m = sum n: transferFrom(_,r,s,n).
```

The error in `r2'` is that the `transferFrom` table is grouped by the receiver (`r`) and spender (`s`) column, instead of the sender (`o`) and spender (`s`) column (as in `r2`).

When a spender account `s` wants to transfer `20` tokens from account `a` to account `b`, by submitting a transaction `transferFrom(a,b,s,20)`, it is reverted. DeCon explains that it is because the condition `l >= n` in `r1` is false, which means the spender `s` does not have sufficient allowance to transfer tokens on `a`'s behalf.

To understand why the `spender` only has `10` allowance to `a`'s account, one could get the provenance of the tuple `allowance(a,s,10)`, as shown in Figure 4a. On top of the provenance tree, `allowance(a,s,10)` is derived by `r3`, from the fact that the total allowance is `100` (`allowanceTotal(a,s,100)`), and that `a` has spent `90` already (`spentTotal(a,s,90)`). To see why `spentTotal(a,s,90)` is derived, the programmer continues expanding its provenance tree. A bug is revealed at this step, where a transaction from address `b` to `a` is accounted for `s`'s allowance on address `a`, which points to the bug in `r2'`.



(a) allowance(a,s,10) **(b) transferFrom(a,s,r,tokenId)**

Figure 4: Provenance tree for tuples.

5.2 ERC721

ERC721 [28] is a smart contract standard for non-fungible tokens (NFTs). A main transaction for ERC721 tokens is `transfer`, which records the transfer of a token from `sender` to `recipient` at a particular `time`. The transaction time is included to specify the following views.

First is the view function `ownerOf`. Given the `transfer` relation, the owner of a token is defined as follows.

```
latestTransfer(tokenId,s,r,t):-transfer(tokenId,s,r,t),
    t = max i: transfer(tokenId,_,_,i).
ownerOf(tokenId, p):-latestTransfer(tokenId,_,p,_),p!=0.
```

The first rule selects the latest transfer record for `tokenId`, and the next rule specifies that if the recipient of the latest transfer is non-zero, it is the owner of the token.

Next is the `exist` relation. A token exists if it is minted and is not burnt. In ERC721 contracts, burning a token emits a transfer record from its owner to zero address. Hence, `exist` is defined as follows.

```
exists(tokenId, true) :-
    latestTransfer(tokenId,_,to,_), to!=0.
```

The rule checks that a token's latest transfer recipient is a non-zero address, which means it is not burnt.

To ensure every existing token has an owner, we could specify the following property.

```
.violation tokenNoOwner
tokenNoOwner(tokenId) :-
    ownerOf(tokenId,0),exists(tokenId,true).
```

This rule defines a property violation as an entry in the `ownerOf` table, where owner address is `0` and `tokenId` exists.

Explain an unexpected token transfer via data provenance. Suppose the owner wants to understand why one of her tokens has been transferred away in a transaction `transferFrom(a,s,r,tokenId)`, where a is the operator, s is the sender, and r is the receiver, she expands the provenance tree for the transaction, which is shown in Figure 4b. On top of the provenance tree is a transferFrom tranaction, approved by the following rule:

```
r4: transferFrom(operator, sender, receiver, tokenId) :-
        recv transferFrom(operator, sender, receiver,
             tokenId),
        /* Sender owns the token. */
        ownerOf(tokenId, sender),
        /* Operator is approved to move the token. */
        msgSender(operator), approved(tokenId,operator).
```

where approved(tokenId,operator) means that the token tokenId has been approved to use by operator. This approval is set by the token owner. Suspicious about the approved(tokenId,a) tuple, the owner continues to expand the provenance tree, and finds that it is derived from the following rule.

```
r5': approved(tokenId,operator) :-
                    approve(_,operator,tokenId).
```

and the tuple approve(b,s,tokenId), which means account b, a previous owner, has approved this token to operator s before transferring this token to a. Here, she finds the bug; r5 does not check that the address that approves the token should be the token owner. The rule should have been updated as follows instead.

```
r5: approved(tokenId,operator):-ownerOf(tokenId,owner),
        approve(owner,operator,tokenId).
```

6 EVALUATION

We implement a prototype compiler [5] for DeCon in Scala that generates Solidity programs with instrumentation for run-time verification. We first evaluate the compiler by comparing its output, without instrumentation, with reference contract written in Solidity. Next, we evaluate the overhead of run-time verification on these contracts and their properties.

6.1 Overhead to Reference Implementations

Reference smart contracts. We collect five reference smart contract implementations from public repositories and prior research. Wallet is the example shown in Section 2. CrowdSale is from prior research paper [52]. SimpleAuction is from Solidity documentation [9]. ERC20 (fungible tokens) and ERC721 (non-fungible tokens) are two of the most popular smart contracts deployed on Ethereum [1], and we use the implementation from the OpenZepplin library [6].

Declarative smart contract implementation. We implement declarative counter-parts for all reference contracts with the same interfaces and functionalities without instrumentation for run-time verification or provenance. These contracts consist of 10 to 18 rules (column *#Rules* in Table 1).

Although DeCon can specify all the high-level logic of the these contracts, we note that the generated Solidity code has the following difference from the reference implementations. First, the reference CrowdSale contract is implemented as two separate contracts. As DeCon does not yet support contract composition, the compiler outputs a stand-alone smart contract with all the functionalities. For the ERC721 contract, there is a safeTransferFrom interface, which wraps the transferFrom function with a check: if the recipient is also a smart contract, it should implement the onERC721Received interface. The current implementation of DeCon does not yet support such checking procedure, which relies on calling the built-in functions of Solidity, so this interface is omitted.

Measurement metrics. We measure two metrics: (1) the size of EVM byte-code deployed on the blockchain; and (2) the gas cost for each transaction. EVM byte-code is generated by the Truffle [13] compiler. To measure gas cost, we first deploy the smart contract on Truffle's local blockchain, and then populate the smart contract states by sending transactions from N test accounts, which results in N entries in the contract states. Then we call each transaction interface again and record gas cost reported by Truffle. We find that N (10 to 1000) does not impact gas cost. This is because all contracts use hash-maps to store contract states. If the hash-collision rate is low, the number of instructions is constant to the size of the hash-map, and thus the gas cost remains constant. Therefore, we report the gas cost measured with $N = 10$.

Results. As shown in Table 1, the median gas overhead to reference implementation is 14% across 16 transactions, with 3 of them have even lower gas cost between −28% to −12%. In the extreme case, the withdraw transaction from SimpleAuction shows 101% gas overhead.

We identify two sources of extra gas cost: (1) long function invocation chain, and (2) inefficient use of data structures. For example, in the Wallet example (Section 2), mint transaction updates the variable totalMint, which further updates totalSupply, thus adding extra cost than directly incrementing totalSupply as done by the reference implementation. For data structures, relational tables are directly maintained as arrays of tuples, with extra information like valid bits and timestamps. Such extra information takes up additional space than their counterparts in Solidity implementations. Mitigating such overhead borrowing ideas in SQL execution plan optimization would be an interesting direction for future research.

DeCon consumes less gas in some transactions. In Wallet, the DeCon contract has less read / write to the global memory. In Crowfunding, the reference contract invokes an external call to another contract, whereas DeCon implements everything in one monolithic contract, thus eliminating the inter-contract transaction cost. In ERC721, DeCon has fewer condition checks because some conditions are specified as rules and therefore automatically maintained by the contract.

In terms of byte-code size, DeCon's compiler output is slightly greater than the reference programs, with a 2 KB (SimpleAuction) maximum increase. Note that on CrowSale, DeCon's output is smaller than the reference contract. This is because the reference implements two separate contracts, while the program generated by DeCon compiler has all functions implemented in one contract.

Contract features that are not yet supported. During the search of benchmarks, we find some contracts use features that are not yet supported by DeCon. For example, the voting contract from Solidity documentation [14] checks voting loop in a recursive manner. Although recursion can be naturally expressed in DeCon language, the execution of recursion functions requires non-trivial reasoning

[1]According to https://etherscan.io, at the time of writing this paper, there are about 502,000 ERC20 tokens and 50,000 ERC721 tokens on Ethereum.

Table 1: Overhead of Solidity programs generated by DeCon, compared to reference implementations. Column *#Rules* shows the number of rules in the declarative smart contracts.

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%**

Table 2: Run-time verification overhead. Column *Size* and *Gas* show the overhead in byte-code size (KB) and gas cost (K) respectively, compared to the DeCon contract without instrumentation.

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%**

to ensure termination and gas efficiency, and is therefore not yet supported by DeCon. In addition, certain functions that lie outside of relational logic, including checking interfaces of another contract (e.g. `safeTransferFrom` in ERC721), and cryptographic functions[8], are not yet supported, but they can be incorporated into DeCon via user-defined functions in the future.

6.2 Run-Time Verification Overhead

We measure run-time verification overhead by first specifying properties for each contract, which are generated as instrumentation in the output Solidity program. These instrumented programs are then compared to DeCon programs without instrumentation, on byte-code size and gas usage.

Contract properties are specified as follows. First, as shown in the example in Section 2, the Wallet contract is monitored for negative account balances. The Crowdsale contract allows participants to invest in a crowd funding project with a particular funding target. The property specifies that the total amount of raised fund should equal to all participants' investments. In SimpleAuction, bidders transfer their fund on every bid, and get refunds when the auction is ended. A property specifies that every bidder can claim refund at most once. In ERC721, the property specifies that all existing tokens should have a valid owner (non-zero address). In ERC20, all account balances should add up to the total supply of tokens.

Results. Table 2 shows the overhead of run-time verification. Byte-code sizes are increased by no more than 2 KB. Gas usage overhead varies across different transactions, with the median being 16%. Wallet and ERC721 contracts show small overhead, where transaction gas consumption increases by no more than 17% and 8%, respectively. Crowdsale and SimpleAuction contract come with larger overhead. The highest increase in their transaction gas usage are 50% and 60%. The ERC20 contract incurs the highest overhead, where the `transferFrom` transaction shows 109% increase.

7 RELATED WORK

In this section, we survey several lines of research that are related to our work.

Run-time verification. Similar to DeCon, Solythesis [44] also specifies properties as invariants and generates instrumentation for run-time monitoring. It applies to general smart contracts implemented in Solidity, whereas DeCon targets declarative contracts only. By restricting the scope on declarative contracts, both specification and monitoring can be performed in a more straightforward manner. Invariants become violation queries, where joins are analogous to existential quantifiers, and aggregations to universal quantifiers. Detection becomes query evaluation, which reuses the same procedure for contract execution.

SODA [27] is a framework for implementing generic attack detection algorithms. Unlike DeCon where the monitoring procedure is automatically generated from specification, the detection algorithms in SODA are implemented manually.

Sereum [54] monitors reentrancy attacks online via taint analysis. Azzopardi et al. [20] monitors contract execution against legal contract logic. These two work targets specific vulnerabilities and properties on Solidity smart contracts, whereas DeCon monitors user-specified properties on declarative contracts.

Static analysis and verification. Static analysis has been applied to detect generic vulnerabilities such as reentrancy attacks [33, 45], integer bugs [60, 62], trace vulnerability [49], and event-ordering bugs [42]. Securify [63] translates the EVM byte-code into stratified Datalog, and checks vulnerability patterns using off-the-shelf Datalog solvers.

Alt et al. [18] translate Solidity program into SMT formulas and use off-the-shelf SMT solver to verify contract properties. Zeus [41] leverages abstract interpretations and symbolic model checking to verify correctness and fairness of smart contracts.

Symbolic execution [16, 17, 26, 43, 46, 48, 52, 63] is another popular technique for smart contract verification. Oyente [46] detects generic predefined vulnerabilities including reentrancy, transaction order dependency, mishandled exceptions, etc. Verx [52], on the other hand, allows programmers to specify contract-specific properties in temporal logic.

Fuzzing has also been applied to smart contracts. For example, ContractFuzzer [38] tests smart contracts for security vulnerabilities. Echidna [31] generates tests that triggers assertion violations. ILF [36] and Harvey [65] focus on improving code coverage.

Unlike these work, DeCon monitors properties online, which incurs run-time overhead, but does not suffer from false-positives or false-negatives. In addition, DeCon targets declarative smart contracts, while these tools analyze Solidity or EVM byte-code. Although targeting different languages, the underlying verification techniques can also be applied to DeCon and benefit from its higher-level abstraction. We believe this is an exciting direction for future research.

Domain-specific languages for financial contracts. Scilla [57] is a intermediate-level language for smart contracts that offers type safety and support for verification. KEVM [37] defines the formal semantics of EVM, and has been used to verify contracts against ERC20 standards. These languages provides precise formal specification of smart contract down to the byte-code level, and are good for verifying low-level properties. In contrast, DeCon focuses on the high level abstraction of smart contracts and specification of contract-specific properties. Jones et al. [39] uses functional programming language to write financial contracts. BitML [21] is a high-level language for Bitcoin smart contracts. Based on process calculus, it translates contracts into Bitcoin transactions. DeCon, on the other hand, is based on relational logic and targets Ethereum smart contracts.

Datalog languages. DeCon shares similar syntax with general Datalog languages like Souffle [40], and is inspired by incremental evaluation techniques in systems like DDlog [55]. DeCon, however, is specific to Ethereum smart contracts in the following aspects. First, DeCon has a number of domain-specific language extensions necessary for capturing execution semantics in Smart Contracts (Section 3). Second, DeCon compiles Datalog to Solidity, with several domain-specific optimizations (Section 4.5). Finally, DeCon offers a property specification and run-time monitoring feature (Section 4.3), which is essential since smart contracts are managing a lot of digital assets.

Deontic logic for normative knowledge. Gabbay et al. [30] present a historical overview of deontic logic for normative knowledge. Based on similar principles, Prakken et al. [53] overview logic-based approaches for legal applications. DeCon is a logical system representing knowledge in the domain of smart contracts, which enables efficient communication and automatic reasoning.

8 CONCLUSION AND FUTURE WORK

We present DeCon, a declarative programming language for smart contract implementation and property specification. In DeCon, smart contracts are specified in a high-level and executable manner, thus providing opportunities for efficient analysis and verification, bringing clarity to transaction execution via data provenance. Contracts implemented in DeCon demonstrate comparable efficiency to open-source reference implementation. Furthermore, run-time verification adds moderate gas overhead.

Our initial experience with DeCon suggests a few exciting future directions. First, we find interesting contracts that require additional language features, including contract composition, recursion, user-defined functions, etc. Second, there are extreme cases where DeCon compiler generates contracts with non-negligible overhead to the reference hand-written code. DeCon compiler needs further optimization to generate more efficient executable code. Third, to save the overhead of run-time verification, we can leverage the high-level abstraction of DeCon programs to perform static verification.

9 DATA-AVAILABILITY STATEMENT

The software and scripts for reproducing the experiment results are available online [3].

ACKNOWLEDGEMENTS

We thank the anonymous reviewers of FSE'22 for their insightful feedback and suggestions. This work is supported by NSF CNS-2104882, NSF CNS-2107147, NSERC Discovery Grant, and Office of Naval Research grant on Security for resource limited networked Cyber-physical systems.

REFERENCES

[1] 2016. The DAO. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[2] 2016. King of the Ether Throne — Post-Mortem Investigation. https://www.kingoftheether.com/postmortem.html.

[3] 2022. Artifacts for reproducing experiment results. https://doi.org/10.5281/zenodo.7080252

[4] 2022. Declarative smart contract benchmarks. https://github.com/HaoxianChen/declarative-smart-contracts/tree/master/benchmarks.

[5] 2022. Declarative smart contracts. https://github.com/HaoxianChen/declarative-smart-contracts.

[6] 2022. Openzeppelin contracts. https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token.

[7] 2022. Recursions in Solidity. https://docs.soliditylang.org/en/v0.8.13/control-structures.html?highlight=recursion.

[8] 2022. Safe remote purchase. https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html#safe-remote-purchase.

[9] 2022. Simple auction. https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html#simple-open-auction.

[10] 2022. SMTChecker and Formal Verification. https://docs.soliditylang.org/en/v0.8.12/smtchecker.html#assert.

[11] 2022. Solidity. https://docs.soliditylang.org.

[12] 2022. Solidity events. https://docs.soliditylang.org/en/v0.8.13/abi-spec.html?highlight=events#events.

[13] 2022. Truffle. https://trufflesuite.com.

[14] 2022. Voting. https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html#voting.

[15] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. *Foundations of Databases: The Logical Level* (1st ed.). Pearson.

[16] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2019. SAFEVM: a safety verifier for Ethereum smart contracts. In *Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM SIGSOFT, 386–389.

[17] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. Ethir: A framework for high-level analysis of ethereum bytecode. In *Int. symposium on automated technology for verification and analysis (ATVA)*. Springer, 513–520.

[18] Leonardo Alt and Christian Reitwiessner. 2018. SMT-based verification of solidity smart contracts. In *Int. Symposium on Leveraging Applications of Formal Methods*. Springer, 376–388.

[19] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. Medrec: Using blockchain for medical data access and permission management. In *Int. Conf. on Open and Big Data (OBD)*. IEEE, 25–30.

[20] Shaun Azzopardi, Gordon J Pace, and Fernando Schapachnik. 2018. On observing contracts: deontic contracts meet smart contracts. In *Legal Knowledge and Information Systems*. IOS Press, 21–30.

[21] Massimo Bartoletti and Roberto Zunino. 2018. BitML: a calculus for Bitcoin smart contracts. In *CCS*. 83–100.

[22] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for smt-based static analysis. *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* 4 (2020), 1–31.

[23] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure derivative contracts for Ethereum. In *Int. Conf. on Financial Cryptography and Data Security (FC)*. Springer, 453–467.

[24] Ryan Browne. 2017. 'Accidental' bug may have frozen $ 280 million worth of digital coin ether in a cryptocurrency wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[25] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *International conference on database theory*. Springer.

[26] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. 2019. sCompile: Critical path identification and analysis for smart contracts. In *Int. Conf. on Formal Engineering Methods*. Springer, 286–304.

[27] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts.. In *NDSS*.

[28] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. 2022. EIP-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.

[29] Vitalik Buterin Fabian Vogelsteller. 2022. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[30] Dov Gabbay, Jeff Horty, Xavier Parent, Ron van der Meyden, and Leon van der Torre. 2013. Handbook of deontic logic and normative systems. (2013).

[31] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM SIGSOFT, 557–560.

[32] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and tools for the static analysis of ethereum smart contracts. In *Int. Conf. on Computer Aided Verification (CAV)*. Springer, 51–78.

[33] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Programming Languages (POPL)* 2 (2017), 1–28.

[34] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[35] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. 2017. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *Power & energy society innovative smart grid technologies Conf. (ISGT)*. IEEE, 1–5.

[36] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Conf. on Computer and Communications Security (CCS)*. ACM SIGSAC, 531–548.

[37] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *Computer Security Foundations Symp. (CSF)*. 204–217. https://doi.org/10.1109/CSF.2018.00022

[38] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Int. Conf. on Automated Software Engineering (ASE)*. IEEE/ACM, 259–269.

[39] SL Peyton Jones and Jean-Marc Eber. 2003. How to write a financial contract. (2003).

[40] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

[41] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Network and Distributed System Security Symposium (NDSS)*. 1–12.

[42] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Int. symposium on software testing and analysis (ISSTA)*. ACM SIGSOFT, 363–373.

[43] Johannes Krupp and Christian Rossow. 2018. {teEther}: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Security Symposium*. USENIX, 1317–1333.

[44] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[45] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Int. Conf. on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[46] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Conf. on computer and communications security (CCS)*. ACM SIGSAC, 254–269.

[47] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. 2017. A smart contract for boardroom voting with maximum voter privacy. In *Int. Conf. on financial cryptography and data security (FC)*. Springer, 357–375.

[48] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Int. Conf. on Automated Software Engineering (ASE)*. IEEE/ACM, 1186–1189.

[49] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *computer security applications Conf.* 653–663.

[50] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. 2017. Trading stocks on blocks-engineering decentralized markets. In *Int. Conf. on Design Science Research in Information System and Technology*. Springer, 474–478.

[51] Zhiniang Peng. 2018. Not A Fair Game – Fairness Analysis of Dice2win. https://blogs.360.net/post/Fairness_Analysis_of_Dice2win_EN.html.

[52] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *symposium on security and privacy (S&P)*. IEEE, 1661–1677.

[53] Henry Prakken and Giovanni Sartor. 2015. Law and logic: A review from an argumentation perspective. *Artificial intelligence* 227 (2015), 214–245.

[54] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Network and Distributed Systems Security (NDSS)*.

[55] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. *Datalog* 2 (2019), 4–5.

[56] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Conf. on Computer and Communications Security (CCS)*. ACM, 621–640.

[57] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[58] David Siegel. 2016. Understanding The DAO Attack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/. (2016).

[59] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*. Springer, 245–251.

[60] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *Symposium on Security and Privacy (S&P)*. IEEE, 1678–1694.

[61] Parity Technologies. 2017. Parity Security Alert. https://www.parity.io/blog/security-alert-2/. (2017).

[62] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Annual Computer Security Applications Conf.* 664–676.

[63] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Conf. on Computer and Communications Security (CCS)*. ACM SIGSAC, 67–82.

[64] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118.

[65] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESE/SFSE)*. ACM, 1398–1409.