ERC20: Correctness via Linearizability and Interference Freedom of the Underlying Smart Contract

Rudrapatna K. Shyamasundar[*]
Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Powai, Mumbai 400076, India

Keywords: Distributed Programs, ERC20 Tokens, Smart Contract, Linearization, Consensus, Interference Free.

Abstract: ERC20 is a standard for the creation of a specific type of tokens called ERC20 tokens, one of the most widely adopted tokens on Ethereum. ERC20 tokens are transferred through dedicated transactions among Ethereum addresses, and managed by smart contracts. Nondeterministic behaviour has been observed on the smart contracts that manage ERC20 tokens resulting in owners losing significant amounts while using it. In this paper, we first discuss issues of nondeterministic behaviour in the ERC20 smart contract, and the standard general remedies that have been proposed in the literature to avoid nondeterministic behaviour in ERC20. Then, through the notion of linearizability, it is shown that as ERC20 permits unbounded concurrency, the linearizability of the ERC20 smart contract is undecidable - thus, demonstrating the subtle complexity of ERC20 and the strong synchronization requirements of ERC20. Finally, treating ERC20 smart contract as a set of asynchronous interacting processes executing on a blockchain, we describe an approach that is common in classical programming language specification, and show how a set of constraints on the traces of ERC20 executions based on interference freedom property for concurrent execution on the blockchain overcomes the nondeterministic behaviour; we shall further sketch how such an execution can be implemented in Solidity. Furthermore, we discuss how the two approaches of linerarization and interference freedom mutually benefit each other and assist in arriving at constraints that leads to wait-free implementation of smart contracts.

1 INTRODUCTION

Blockchain platforms, need their own currency, a sort of tradable tokens, for interoperable services. Tokens are blockchain-based assets which can be exchanged across users of a blockchain platform. Tokens typically need to follow a standard so that it is possible to write tools, such as liquidity pools and wallets, that work with tradable tokens. For instance, in the Ethereum platform, tokens can represent virtually anything such as (i) financial assets (shares in a company), (ii) fiat currency (eg., INR), (iii) standard unit of gold, (iv) reputation points in an online platform, (v) lottery tickets etc. Naturally, such a powerful feature must be handled through a robust standard. Ethereum Request for Comment (ERC)20 defines a standard for the creation of a specific type, called ERC20 tokens, one of the most widely adopted tokens on Ethereum. ERC20 tokens are transferred through dedicated transactions among Ethereum addresses, and are managed by smart contracts. It is a standard for fungible tokens. That is, they satisfy a property that makes each token to be exactly the same (in type and value) as another token - that is, tokens are mutually exchangeable. Thus, ERC20 is a standard way to implement basic features of all tokens making them compatible with common Ethereum software such as Ethereum Wallets etc. In this paper, we shall first discuss issues, and attack scenarios on ERC20 smart contracts, and provide a brief overview of existing mitigation techniques to forbid unwanted behaviours of ERC20. Secondly, we shall demonstrate the subtle difficulties of ERC20 smart contract by showing that linearization of ERC20 methods as it stands is undecidable; this result shows the subtle strong synchronization requirements of ERC20. Finally, we envisage a novel way of realizing robustness of ERC20 smart contract using techniques of interference freedom used concurrent program verification. An interpretation of linearization and interference freedom assists in arriving at appropriate constraints so that the smart contracts when executed on a blockchain is able

[*]The work was carried out in the Centre for Blockchain Research sponsored by Ripple Labs.

Shyamasundar, R.
ERC20: Correctness via Linearizability and Interference Freedom of the Underlying Smart Contract.
DOI: 10.5220/0012145800003555
In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*, pages 557-566
ISBN: 978-989-758-666-8; ISSN: 2184-7711

to realize the intended sequential behaviour[1].

Rest of the paper is organized as follows: Section 2 highlights the ERC20 standard with descriptions of its interface, attack scenarios of ERC20 in Section 3. In Section 4, we discuss informal and formal approaches to mitigate attacks in the use of ERC20. Section 5 shows that the linearization of ERC20 is undecidable. Finally, we discuss how establishing interference freedom of concurrent procedures will lead to arriving at the intended sequential behaviour without resorting to locks in Section 6 followed by conclusions in Section 7.

2 ERC20 TOKEN STANDARD

A large number of Ethereum development standards focus on token interfaces that help in ensuring composable nature of smart contracts. ERC20 is the earliest token standard by Ethereum. We shall briefly describe the interface[2] below.

As pointed out already, the purpose of token standards like ERC20 is to allow a spectrum of token implementations that are inter-operable across applications, like wallets, decentralized exchanges etc. For this purpose, an interface (similar to that in Java), is defined so that any code that needs to use the token contract can use the same definitions as in the interface and remain compatible with all the token contracts that use it, whether it is a wallet, a Dapp, or a different contract such as a liquidity pool. A brief on the main interface functions are given below[3]:

1. `function totalSupply() external view returns (uint256)`: Returns the amount of tokens in existence.
   - This is an external function, i.e., it can be called only from outside the contract. It returns the total supply of tokens in the contract using the unsigned 256 bits - the native word size of the EVM. It is a view function, and hence, does not change the state; thus, it can be executed on a single node. As the function does not generate a transaction, it does not consume gas. Note that anyone can view anyone's account balance (*there are no secrets on the blockchain*).
   - A question may arise as to what if the contract creator returns a smaller total supply count than the real value, making each token appear more valuable than it actually is. This is not possible as a characteristic of the blockchain is that anything that happens on the blockchain can be verified by every node. To confirm this, every contract's source code and storage must be available on every node. While it is not mandatory to publish the Solidity code for the contract, nobody would take the contract seriously unless the source code and the object code on which it was compiled is published; assuming the code has been published, we can rule out the possible cheating as mentioned above.
2. `function balanceOf(address account) external view returns (uint256);` `balanceOf` returns the balance of an account.
   - Note that Ethereum accounts are identified in Solidity using the address type, which holds 160 bits. Similar to `totalSupply()`, this is also an external function in view only mode and thus, can be used by anyone.
3. `function transfer(address recipient, uint256 amount) external returns (bool);` `transfer` transfers the specified number of tokens from the caller-user to a different receiver-user (and reduces its balance of tokens by that quantity) if available. `transfer` function is called to transfer tokens from the sender's account to an account of another user; it returns a boolean value, that is always true. The call `transfer` fails, for instance, when there are not enough tokens, and in such a case the contract reverts the call. Note that the owner (sender) is transferring tokens to another address (user/customer). The execution involves a change of state, creating a transaction and thus, costs gas. It also emits an event, `transfer`, to inform everybody on the blockchain that the event `transfer` has happened.
   - `transfer` has two types of outputs depending on the type of callers:
   (a) Users that call the function directly from a user interface: Here, user submits a transaction and does not wait for a response, which could take an indefinite amount of time. The user can ascertain execution of the transaction by looking for the transaction receipt (which is identified by the corresponding transaction

[1]The importance of analysing ERC20 like smart contracts can be seen by the very recent tweet: *Scam Sniffer (@realScamSniffer) tweeted at 3:41 am on Mon, May 01, 2023: someone lost $2.28m worth of USDC by ERC20 Permit phishing 6 hours ago https://t.co/x17Gw8w1Bw https://t.co/4jmyeVT1Ym (https://twitter.com/realScamSniffer/status/1652797793587851264?t=k7NtBkmdxiJjDYjIYq7OhQ&s=03)*

[2]https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

[3]https://ethereum.org/en/developers/tutorials/erc20-annotated-code/

hash) or by looking for the `transfer` event.

   (b) Other contracts which call the function as part of the overall transaction: Such contracts get the result immediately, as they are running in the same transaction, and thus, can use the *function return value*.

4. `function allowance(address owner, address spender) external view returns (uint256);`
   - Using `allowance`, anybody can query to see what is the allowance that one address (owner) lets another address (spender) spend.
5. `function approve(address spender, uint256 amount) external returns (bool);` `approve` creates an allowance for another user. Note that there is no need that the sender must have those very many tokens approved by him at that time (or even later).
   - `approve` is the function that makes the standard quite complex and enables users to misuse/abuse intentionally or unknowingly. Note that *in an asynchronous system, the owner of a resource can control the order of his own transactions but not control the order of transaction of other users*. Thus, the method call can be interpreted as an account (or variable) that can be modified by multiple accounts/users. The issue with such complex operations will become clear in the sequel.
6. `event Transfer(address indexed from, address indexed to, uint256 value);`
   - Emitted when ‘value‘ tokens are moved from one account (‘from‘) to another (‘to‘); note that the value could be zero. Further, the respective event is emitted when the state of the ERC20 contract changes.
7. `function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);` the call `transferFrom` is used by the spender to spend the allowance that has been earlier approved (granted) by other user/owner for it.
   - Functionally, the operation (i) transfers an amount less than or equal to the allowance that has been approved by some other owner/user, (ii) reduces the allowance by that amount, and (iii) the operation fails if the amount to be transferred is greater than the approved allowance (if that happens, it reverts).
   - One of the main differences between `transfer` and `transferFrom` is that in the former the owner decrements the number of tokens when he sends the tokens to some other user, whereas in the latter the grantee decrements the number of tokens he uses to send it to some other user. In a sense, it is like there are multiple owners for the address.
8. `event Approval(address indexed owner, address indexed spender, uint256 value);`
   - Emitted when the allowance of a ‘spender‘ for an ‘owner‘ is set by a call to `approve`. ‘value‘ is the new allowance. Again note that the emission of the event `transfer` takes place when the state of the ERC20 contract changes.
9. There are other user interface functions that are needed for the actual usage but are not relevant for our discussion here.

3 ATTACKS AND THEIR MITIGATION

In this section, we shall analyze issues of the possible attack scenarios and their mitigation using the above described interface specification, from a programmer perspective as well as from the synchronization requirement of the smart contract.

3.1 Basic Attack Scenario

Consider a scenario of users (say owners of accounts) $\{U_1, U_2, \cdots, U_n\}$. Let $A_{ij}\ \forall i \neq j$ be the `Allowance` approved by U_i to user $U_j, \forall i \neq j$. It is to be noted that users $\{U_1, U_2, \cdots, U_n\}$ are essentially asynchronous processes. Possible attack scenarios on ERC20 have been highlighted by several authors[4] and are illustrated below:

1. U_1 approves 80 tokens to U_2 (i.e., A_{12} is 80)
2. U_1 realises that he made a mistake and wants to revise the approval to only 50 tokens.
3. To correct the mistake, U_1 sends out an approval using `Approve` for 50 tokens to U_2.
4. Now there are two possibilities:
   (a) It is possible U_2 has already withdrawn x tokens, $0 \leq x \leq 80$, using `transferFrom` . In such a case, U_2 will get an additional 50 tokens over and above x.

[4]https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit\#.)

(b) If U_2 has not transferred the approved tokens from U_1 then the allowance A_{12} will get set to 50 tokens as wanted by U_1.

(c) Thus, U_2 could have used anywhere between x + 50 to 80+50 tokens

The basic reason for such an anomalous behaviour is that users/processes $\{U_1, U_2, \cdots, U_n\}$ are asynchronous and hence, leads to nondeterminism as the program does not use either locking or a synchronization construct. A naive programmer understands that the execution on blockchains like Ethereum using smart contracts is sequential (as intended in Solidity) and thus, does not realize the underlying nondeterminism due to interleaving. The authors of (Sergey and Hobor, 2017) capture this through the following analogy of the underlying artifacts:

> Accounts using smart contracts on a blockchain are like threads using concurrent objects in shared memory with correspondences of artifacts as below: *contract state to object state, call/send to context switching, Reentrancy to (Un)cooperative multitasking, Invariants to Atomicity and Nondeterminism to Data races.*

Before we look at possible approaches of making ERC20 deployment robust, let us understand the complexity of the problem from the illustrated attack scenario.

While one could have illustrated the attack using only two users, we introduced n users to understand a general scenario to expose the underlying complexity. From the above discussions, the following facts can be observed about the ERC20 interface specification:

1. `transfer` can be interpreted as follows: An user, say U_i (the owner-the token creator), transfers to other user accounts U_j, $j \neq i$ some number of his tokens from his total supply recorded in `TotalSupply`. Looking at the function in an isolated manner, the semantic action corresponds to U_i (owner) decreasing its `TotalSupply`[5] by x every time he sends out x tokens; note that except for U_i nobody else can write into its `TotalSupply`. That is, the only writer of TotalSupply is its owner.
2. `transferFrom` can be interpreted as follows: Here, user U_i, takes x number of tokens that are $\leq A_{ji}$ (allowed by Owner U_j to U_i), and transfers to some other user U_k, $i,k \neq j$. Thus, U_i writes into the storage, TotalSupply owned by U_j to set its count after usage.

[5]For the sake of simplicity let us assume `TotalSupply` also denotes the total number of tokens it has.

3. Looking at (1)-(2) above, we can see the storage of U_i need to be written by all the other users. If we treat each of the accounts U_i as assets, one can see that each account in principle can be written by other accounts/users who are not its' owners.
4. Overlaying the use of `Approve` as highlighted along with (1)-(3) provides a complete picture of the execution of ERC 20 usage in a given deployed context.

Thus, to establish the intended sequential specification of ERC20 deployment, we not only have to show that the attack scenario relative to `Approve` is not possible but also show that execution traces arising from (1)-(2) above also do not introduce any new attack scenarios.

4 OVERCOMING ATTACKS NAIVELY

Here, we briefly provide an overview of efforts in the literature for a robust ERC20 deployment.

4.1 Naive Programming Tricks to Achieve Robustness

Some of the tricks a programmer (a clairvoyant!) can do to overcome data races/nondeterminism are:

1. Restrict token transfers to only smart contracts (or trusted account holders!) whose source code and object code are public from which one can assure that the logic needed for the transfers shown in the attack scenario are not possible.
2. Set the `Allowance` to zero after every `transferFrom`. Note that such a constraint imposes the following restrictions: (a) Forbids the possibility of several transfers of the approved tokens up to the limit of `approve`, and (b) Either one uses the tokens approved at once or loses those that have not been used. It is easy to see that it changes the interface itself.
3. In[6] additional interface functions `increaseAllowance(address spender, uint256 addedValue) −> bool` and `decreaseAllowance(address spender, uint256 subtractedValue) −> bool` are introduced, where the latter is interpreted as follows: Atomically decrease the allowance granted to the spender by the caller. Even

[6]https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20-decreaseAllowance-address-uint256.

with an enhancement, the same nondeterminism persists as one can see from the command sequence: `Approve(80, ...);` `decreaseAllowance(40, ...)`; the number of tokens utilized could vary from 40 (when the user has not used any of the tokens given as allowance) and 80 (when the user has already utilized 80 tokens) assuming `decreaseAllowance()` does not do anything when the number is already zero. Thus, ambiguity persists in spite of atomicity being enforced on these two new functions. This is due to strong synchronization requirements as discussed in Section 5.

Assuming that the blockchain platform is a public ledger and supports viewing of the source codes, if the strategies (i.e., assumptions/commitments) indicated are indeed possible, it shall not have the attack vector indicated above.

4.2 Approaches Using Formal Analysis

In this section, we shall briefly describe two of the formal approaches that have been used to tackle such scenarios.

One of the very interesting attempts has been a proposal for adaptation of the classic proof-carrying code (pcc) (Dickerson et al., 2018) to overcome the property of *immutability* of the smart contract on the blockchain through the specification invariant specification of the methods used in the smart contract. Referring the adaptation as proof-carrying smart contract (pcsc), the authors illustrate a part of ERC20 specification in terms of the invariants of the methods of ERC20. The basic ideas of their proposal is briefed below:

- The first idea is that of setting the `Allowance` to zero after every `transferFrom` as already discussed in the previous subsection.
- The second modification can be understood in two phases: the first is to show how the data races could be overcome and the second phase is to mimic the functionality of load-linked (LL) and store-conditional (SC) instructions. LL loads a value from memory, and SC writes new value to the same location, if and only if it has not been written since the matching LL.

One of the aims of the proposal is to find ways to overcome the issue of immutability of smart contracts once it is on the blockchain. One of their proposals is to specify through invariants so that implementational changes can be catered to keeping the specification invariant. In summary, the proposal consists of modifying the program after exploring an abstract specification of ERC20, and proposes a specification/implementation using notione like *linearization, etc.,* (Herlihy and Shavit, 2012). Note that the suggested ideas change in some sense the intended behaviour of ERC20 itself. While such an analysis is extremely useful for smart contract programmers, it is difficult to say how such approaches can be adopted in practice as it involves an understanding of rich mathematical formalism of derivation of invariant formally. We shall revert to the use of notions like *lineralizability* (Herlihy and Shavit, 2012)) that can be applied to analyse smart contracts to assure the preservation of sequential specification of smart contracts in the sequel.

5 LINEARIZATION AND ERC20

Subtle correctness criteria for concurrent systems is adherence to established sequential specifications. This demands that each concurrent execution of operations corresponds to some serial sequence of the same operations permitted by the specification at the level of abstraction described by the specification of the operations. In the context of distributed programs, the concept of *linearization* proposed in (Herlihy and Shavit, 2012) has been the foundation for such a criteria. This is briefed below.

The standard correctness requirement for concurrent implementations of abstract data structures packaged into a concurrent library is the concept of *linearizability* (Herlihy and Shavit, 2012; Bouajjani et al., 2013), that requires every method or operation to be atomic (behave as if it were executed in one indivisible step) for establishing the sequential specification of each of the methods under concurrent execution. The notion of linearizability identifies the so-called *linearization points* of each operation. These are program points where the entire effect of a method call logically takes place. However, these linearization points are quite complicated as they often depend on a non-local boolean conditions and plausibly even reside in other concurrently executing threads. This makes a brute force search for the linearization points infeasible. A little more formal definition is given below.

Definition 1: Linearizability (Herlihy and Shavit, 2012; Bouajjani et al., 2013) is a formalisation of the concept of atomicity. It demands that every execution history consisting of calls to the methods is equivalent (up to reordering of events) to a legal, sequential history that preserves the order of non-overlapping methods in the original history. We say that a history is sequential if none of its methods overlap in time; moreover, it is legal if each method satisfies its specification[7].

Before we see whether the general ERC20 is linearizable, let us see the general possible execution traces from the ERC20 functional specification.

5.1 General Trace Behaviour of ERC20

From the discussion of the ERC20 functional specification given in section 2, it can be seen that functions like `transfer` or `transferFrom` revert when there are not enough tokens for the transaction. In view of this, we can see that the execution traces are nothing but a regular expression over all the calls specified in ERC20. Based upon the parameters, we can distinguish calls like `transfer(i,j)`, `transferFrom (i,j,k)` and so on. Thus, we can say

1. The execution is just a regular expression over the method signatures with parameters as given above.
2. While the contract is on the blockchain, there is no restriction on the number of instances of the different methods including parallel invocation.
3. Also all the methods described in ERC20 keeping in view the sequential execution are realizable as loop-free programs.

From the above possible execution traces, let us ask the following questions about the usage of methods that lead to transactions like `Approve`, `Allowance`, `transfer`, and `transferFrom` in a concurrent way:

1. If a method `Approve ()` is called (in a sense overlapping call) while one call is pending or executing still, what should be the execution semantics?
2. It is possible to have a scenario of having user U_j trying to utilize A_{ij} through an appropriate method call to `transferFrom()`, corresponding to writing by U_j to record the balance of allowance in U_i (i.e., the balance in A_{ij}). Further, user U_k, say $k, i \neq j$ could also be sending some coins to U_i using a method call `transferFrom`. This corresponds to *overlapping writes by distinct users to the same locations*.
3. Apart from other symmetrical combinations of scenario depicted in (2), scenarios corresponding to combinations of `transfer` and `transferFrom` from the users are possible, reflecting multiple users attempting to write into the same location.

[7]Informally, in the context of linearization, a begin-call and end-call of a method can be treated to be happening at some time point between the beginning and the end of the method call.

From the definition of `transferFrom`, it should be clear that nondeterminism arises as multiple writers concurrently write onto the same locations. The illustrations shown above depict overlapping executions–that is the crux of its nondeterministic behaviour. As already mentioned, linearization is one of the main techniques that is used to realize deterministic implementation of distributed programs. We shall discuss such an application on ERC20 below and analyse its subtle features.

5.2 Is ERC20 Linearizable?

If it is possible to show that ERC20 is linearizable, then we would have arrived at an implementation that would not have nondeterministic behaviour or races as we can realize an implementation using wait-free programs and CAS (compare and swap) registers. That is, we could have a spectrum implementations for programs that satisfy linearization, that are race-free and free of nondeterministic behaviour.

From (Bouajjani et al., 2013), we have the following theorem:

Theorem 1: The linearizability problem for unbounded concurrent computations with regular specifications is undecidable (cf. (Bouajjani et al., 2013)).

From this theorem we can conclude,

Theorem 2: The linearizability of ERC20 is undecidable.

Proof: The proof follows from trace characteristics given in section 5.1 noting that there no a priori bound on the number of threads. Note that the full simulation used in (Bouajjani et al., 2013) follows easily.

Now the question is: Is it possible to enforce a set of constraints (perhaps minimal in a qualitative sense) without deviating too much from the intended specification of ERC20 that makes it linearizable.

The above question has been answered affirmatively in (Guerraoui et al., 2019) while assessing the consensus number (Herlihy and Shavit, 2012) of crypto-currency. As it is quite interesting, we shall provide a brief discussion the notion of consensus and consensus number below.

Definition 2: Consider an object, each provided with a thread with one method, say DECIDE(T value). Now N threads operating on the object concurrently have to return a value such that:

Consistent: all threads decide the same value, and

Valid: the common decision is some thread's input.

We say that a class of objects C solves N-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers.

Definition 3: Consensus number of a class C is the largest N for which N-thread consensus is solvable. If no such n exists, consensus number is said to be infinite.
Note: In a system with N or more concurrent threads it is impossible to construct a wait-free or lock-free implementation of an object with consensus number N from an object with a lower consensus number (Fischer et al., 1985). A register with CompareAndSet() and get() methods has an infinite consensus number.

The authors of (Guerraoui et al., 2019) have carved out a k-shared asset transfer problem (very informally, it allows multiple reading and writing from shared accounts each of which have a bounded set of owners – see the informal interpretation of `transferFrom` given earlier) from ERC20 and provide a wait-free implementation for the same; the main restriction placed by the authors is that of statically fixing the set of owners for each account and hence, bounding the number of threads that can be there at run time a priori. The wait free implementation provided in (Guerraoui et al., 2019) result correlates with the results in (Bouajjani et al., 2013) that shows that linearization with a fixed set of threads is solvable. Another interesting aspect of this study has been the carrying out, a subtle analysis of strong synchronization requirement of ERC20 in (Alpos et al., 2021). It is useful to recall the classic result from Fischer, Lynch, and Paterson (Fischer et al., 1985) that establishes the impossibility of wait-free implementation of consensus from atomic registers. That is, consensus requires a higher level of synchronization than atomic registers and the consensus object is universal, in the sense that any shared object described by a sequential specification can be wait-free implemented from consensus objects and atomic registers (Herlihy and Shavit, 2012). Thus, it follows that consensus plays a crucial role to reason about the synchronization power of all shared objects (which admit a sequential specification) among a number of processes. Such a role is captured through the concept of consensus number to express the synchronization power of shared objects. Returning to (Alpos et al., 2021), the authors show that ERC20 supports a richer set of methods compared to standard cryptocurrencies, and thus, results in strictly stronger synchronization requirements. Informally speaking, the authors establish dependency of the lower bound and the upper bound on the consensus number of the stakeholders $U_1, \cdots U_n$, on the object's state that can be modified by method invocations; this is interpreted as that number of threads cannot be bounded a priori.

In fact, the result captured in our Theorem 2 above, demonstrates the same in a succinct straight forward manner.

In distributed/concurrent program analysis, another general approach to establish correctness is the notion of interference freedom. We shall show the application of this concept for ERC20.

6 ERC20 AND INTERFERENCE FREEDOM

The notion of interference freedom in the concurrent execution of programs (Owicki and Gries, 1976) plays a vital role in the establishment of correctness in concurrent/distributed programs. We shall define the notions in a semi-formal way below.
Definition 4: Let P be the pre-condition, Q be the post-condition and and S be the program. Then the triple {P} S {Q} denotes the well-known Hoare's triple with the interpretaion: If P is true before execution of program S, then Q holds if and when execution of S terminates. The triple represents the partial-correctness of S relative to P and Q.
Definition 5: Given {P} S {Q} and another statement T with pre-condition pre(T), the statement T is said to be non-interfering with {P} S {Q} if the following two conditions holds:

- {Q ∧ pre(T)} T {Q}, and
- {pre(S') ∧ pre(T)} T {pre(S')}, where S' is an await or assignment statement within S but not within an *await* block.

Continuing with our earlier discussion, it is quite clear that to realize a robust deterministic specification for ERC20, we need to ensure interference-freedom as defined in Definition 5 for the various methods that are defined in ERC20. For instance, execution scenarios like (a) overlapping of calls `Approve(50)` and `Approve (40)`, or (b) an `Approve` call in lieu of the earlier `Approve`. That is, interpretation of combining concurrent/overlapping `Approve` operations[8] must be unambiguous; scenario (b) corresponds to withdrawal (preemption) of the first `Approve` when the second is issued. Such a preemptive operation is not realizable in an asynchronous setting as shown in (Berry et al., 1993) as preemption is essentially a non-monotonic operation. Thus, for correctness purposes, it is necessary to enforce constraints on the underlying read/write operations (which is indeed the approach highlighted in (Dickerson et al., 2018)). From the results in (Alpos et al., 2021), we can see that simple atomicity at the register level is not sufficient.

[8]In the ERC20 standard, this is left unspecified.

In the following, we shall articulate conditions under which the depicted nondeterminism scenario can be overcome via techniques used in classical concurrent programming language specification and discuss how they can be enforced in languages like Solidity.

6.1 Deriving Required Constraints for Non-Interference

Concurrent programming languages have been designed emphasizing on correctness in particular non-interference. For instance, one of the early well designed languages, Ada, was quite disciplined from a concurrent perspective; here integrity was realized through *mutuallly exclusive access of shared resources*. For this reason, Ada rendezvous was considered inefficient as it did not permit concurrency even when the operations were non-interfering. There have been lot of research to realize efficiency/performance without foregoing correctness. For instance (Shyamasundar and Thatcher, 1989) explored a language structure to realize data integrity without unnecessary mutual exclusion. Thus, in the context ERC20 specification that assume a sequential specification, we shall approach the technique explored in (Shyamasundar and Thatcher, 1989): permit methods to operate in parallel, multiple incarnations or overlapping only if they do not interfere. In the following, we apply the following restrictions using that approach and arrive at the following set of restrictions:

1. Consider the restriction: user U_i cannot make a call to method `Approve` for U_j unless A_{ij} is zero. That is, the user to whom U_i has given an allowance awaits full utilization by it before issuing another approval for allowance. Note that utilization may happen through several calls and thus, does not restrict the allowance usage through one call only and thus, there is no issue of under-utilizing the approved number of tokens. It is to further note that the full utilization of the allowance is observable on the blockchain. Note that the described semantics is valid for Solidity.
2. Method calls of form either `transfer` or `transferFrom` for each user is expected to happen in a non-interfering manner. As `transferFrom` semantically corresponds to multiple processes reading and writing into other locations of other processes. Thus for avoiding data races/nondeterminism, it is necessary to permit only multiple reads or writes that do not interfere with each other. Some of the interference cases due to multiple writings and reading/writing are given below:
   (a) $Approve_i(k,x)$ and $Approve_i(n,y)$ interfere when $i = n$ (this is the illustrated case above).
   (b) $Transfer_i(j,x), Transfer_k(j,y)$ interfere assuming i and k are distinct as i and k are writing into the common location of j.
   (c) $transferFrom_i(j,k,x), transferFrom_k(m,n,y)$ interfere when $k = n$ or $j = m$
   (d) $Transfer_i(j,x)$ and $transferFrom_j(k,m,y)$ interfere when $k = j$.
   (e) other combinations of constraints not given for lack of space.
3. It must be pointed out that several non-conflicting concurrent operations like `transfer` or `transferFrom` are still permitted and thus, constraints are general than mutual exclusion.

Realizing Constraints in Solidity: The above constraints can be realized in the programming language Solidity by transforming the original method calls to *guarded execution of method calls* where the guard is nothing but the constraints to be enforced. Guards satisfying the constraints can be realized using `require`, `assert` and `revert` constructs of Solidity; in fact, this is the underlying principle used in the implementation of *exceptions* as well as safeMath library in a robust manner in Solidity. Such a technique has been the basis in the work explored in (Shyamasundar, 2022) to derive run-time monitors for a spectrum of vulnerabilities of Solidity through classical declarations (that enforces certain constraints for the compiler) in Solidity, where language Solidity with declarations is referred to as $Solidity^+$. Using the constraints of the methods, highlighted above, as guards, we can implement ERC20 in an interference-free manner – thus overcoming its nondeterministic behaviour. We shall not go into further details for lack of space.

6.2 Discussion

The approach of interference-freedom highlighted above allows us to see how the constraints overcome concurrent execution of interfering programs. In fact, some of the approaches of mitigation as highlighted in section 4, and 4.2 follow from the analysis given in section 6.1. In other words, by placing restrictions, we shall find a way of bounding concurrency and hence, derive a wait-free implementation of the methods. This shall not only capture the sequential specification intended in Solidity but also enables one to arrive at an efficient implementation that do not use locks. We are of the opinion that such a methodology will provide further directions in analysing smart contracts and arriving at proof carrying smart contracts.

Applications to Revisions of ERC20: Some of the most popular revisions of token standards are:

1. ERC-721: is a standard interface for non-fungible tokens, like a deed for an artwork or a song,
2. ERC-777: allows people to build extra functionality on top of tokens such as a mixer contract for improved transaction privacy or an emergency recover function to bail you out if you lose your private keys,
3. ERC-1155: allows for more efficient trades and bundling of transactions – thus saving costs. This token standard allows for creating both utility tokens (such as $BNB or $BAT) and Non-Fungible Tokens like CryptoPunks, and
4. ERC-4626 - A tokenized vault standard designed to optimize and unify the technical parameters of yield-bearing vaults.

ERC777 is a refinement of ERC20 and provides a new feature called *hooks*, to simplify the sending process offering a single channel for sending tokens to any recipient. One of the main differences with respect to ERC20 is the mechanism of allowing processes to manage tokens on behalf of others. In ERC20, the approve method lets an account owner p define an amount of tokens that some process p' can spend from the account of p. In contrast, the operator in ERC777 allows process p' that has been approved by p to spend all the tokens owned by the approving process p.

It is clear that standards like ERC777, ERC721 etc., need to be further analyzed for robustness with respect to the intended interpretation. Our methods discussed above provide a sound approach for further investigation of the revisions and assist in arriving at robust deterministic behaviour of corresponding smart contracts.

7 CONCLUSIONS

We have discussed the issues of nondeterminism in ERC20, and analysed the complexity of synchronization requirements of ERC20. Using the approach of linearization , we have shown how the general ERC20 smart contract is not linearizable demonstrating the strong synchronization requirements of ERC20 in comparison to just simple smart contracts for cryptocurrency operations. Further, we have demonstrated through the approach of interference-freedom, how constraints can be derived for enforcing a deterministic behaviour on ERC20. We discussed a sketch of the implementation in Solidity using `guarded execution of method calls` - that is all the methods are preceded by appropriate guards (as in Dijkstra's guarded commands). In a sense, our application of linearization and the approach of interference-freedom have highlighted how overlapping execution of interfering programs leads to nondeterministic behaviour. The approach of interference-freedom highlighted allows us to see how appropriate constraints can be placed to overcome concurrent overlapped execution of interfering programs. As illustrated in 6.2, the approach provides assistance in arriving at constraints for which efficient wait-free implementations can be realized without loosing the intuition of the Solidity sequential specification of smart contracts. The approach further assists in analysing a plethora of token standards as most of the revisions are backward compatible and have additional features along with powerful methods.

In summary, the paper has analysed the complexity of ERC20 from different perspectives and the techniques proposed are also of general interest to smart contract designers to visualize deterministic execution of smart contracts on blockchain platforms.

REFERENCES

Alpos, O., Cachin, C., Marson, G., and Zanolini, L. (2021). On the synchronization power of token smart contracts. In *2021 IEEE 41st ICDCS*, pages 640–651.

Berry, G., Ramesh, S., and Shyamasundar, R. K. (1993). Communicating reactive processes. In *Proc. of the 20th ACM POPL*, page 85–98.

Bouajjani, A., Emmi, M., Enea, C., and Hamza, J. (2013). Verifying concurrent programs against sequential specifications. volume LNCS, 7792, pages 290–309. Springer Berlin Heidelberg.

Dickerson, T., Gazzillo, P., Herlihy, M., Saraph, V., and Koskinen, E. (2018). Proof-carrying smart contracts. In *Financial Cryptography and Data Security: FC 2018*, page 325–338. Springer-Verlag.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.

Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., and Seredinschi, D.-A. (2019). The consensus number of a cryptocurrency. In *Proc. 2019 ACM PODS*, page 307–316, New York, NY, USA.

Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Owicki, S. and Gries, D. (1976). An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340.

Sergey, I. and Hobor, A. (2017). A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security*, pages 478–493. Springer.

Shyamasundar, R. K. (2022). A framework of runtime monitoring for correct execution of smart contracts. In *Int. Conf. on Blockchains, LNCS (ICBC2022))*, pages 92–116. Springer Nature.

Shyamasundar, R. K. and Thatcher, J. W. (1989). Language constructs for specifying concurrency in cdl. *IEEE Trans. Software Engineering*, 15(8):977–993.