A Concurrent Perspective on Smart Contracts

Ilya Sergey[1(✉)] and Aquinas Hobor[2]

[1] University College London, London, UK
i.sergey@ucl.ac.uk
[2] Yale-NUS College and School of Computing, National University of Singapore, Singapore, Singapore
hobor@comp.nus.edu.sg

Abstract. In this paper, we explore remarkable similarities between multi-transactional behaviors of smart contracts in cryptocurrencies such as Ethereum and classical problems of shared-memory concurrency. We examine two real-world examples from the Ethereum blockchain and analyzing how they are vulnerable to bugs that are closely reminiscent to those that often occur in traditional concurrent programs. We then elaborate on the relation between observable contract behaviors and well-studied concurrency topics, such as atomicity, interference, synchronization, and resource ownership. The described *contracts-as-concurrent-objects* analogy provides deeper understanding of potential threats for smart contracts, indicate better engineering practices, and enable applications of existing state-of-the-art formal verification techniques.

1 Introduction

Smart contracts are programs that are stored on a blockchain, a distributed Byzantine-fault-tolerant database. Smart contracts can be triggered by blockchain transactions and read and write data on their blockchain [38]. Although smart contracts are run and verified in a distributed fashion, their semantics suggest that one can think of them as of *sequential* programs, despite the existence of a number of complex interaction patterns including *e.g.*, reentrancy and recursive calls. This mental model simplifies both formal and informal reasoning about contracts, enabling immediate reuse of existing general-purpose frameworks for program verification [5,16,31,32] that can be employed to verify smart contracts written in *e.g.* Solidity [15] with only minor adjustments.

Although all computations on a blockchain are deterministic,[1] a certain amount *non-determinism* still occurs due to races between transactions themselves (*i.e.* which transactions are chosen for a given block by the miners). We will show in that non-determinism can be exploited by adversarial parties and makes reasoning about contract behavior particularly subtle, reminiscent to known challenges involved in conventional concurrent programming.

[1] This requirement stems from the way the underlying Byzantine distributed ledger consensus protocol enables all involved parties to agree on transaction outcomes.

M. Brenner et al. (Eds.): FC 2017 Workshops 2017, LNCS 10323, pp. 478–493, 2017.
https://doi.org/10.1007/978-3-319-70278-0_30

In this paper we outline a model of smart contracts that emphasizes the properties of their *concurrent* executions. Such executions can span *multiple* blockchain transactions (within the same block or in multiple blocks) and thereby violate desired safety properties that cannot be stated using only the contract's implementation and local state—precisely what the existing verification methodologies focus on [5,32]. To facilitate the reuse of the common programming intuition, we propose the following analogy:

Accounts using smart contracts in a blockchain
are like
threads using concurrent objects in shared memory.

Threads using concurrent objects in shared memory. By *concurrent objects* we mean the broad class of data structures that are employed to exchange data between and manage the interaction of multiple *threads* (processes) running concurrently [20]. Typical examples of concurrent objects are locks, queues, and atomic counters—typically used via popular libraries such as `java.util.concurrent`. At runtime, these concurrent objects are allocated in a block of *shared memory* that is accessible to the running threads. The behavior resulting from the threads accessing the objects simultaneously—*i.e. interference*—can be extremely unpredictable and thus extremely difficult to reason about.

Concurrent objects whose implementation does not utilize proper synchronization (*e.g.*, with *locks* or *barriers*) can manifest *data races*[2] under interference leading to a loss of memory integrity. Even for race-free objects the observed behavior under interference may be erroneous from the perspective of one or more clients. For example, a particular thread may not "foresee" the actions taken by the other threads with a shared object and thus may not expect for that object to change in all of the ways that it does change under interference.

Accounts using smart contracts in a blockchain. Smart contracts are analogous to concurrent objects. Instead of residing in a shared memory they live in the blockchain; instead of being used by threads they are invoked by *accounts* (users or other contracts). Like concurrent objects, they have internal mutable state, manage resources (*e.g.* funds), and can be accessed by multiple parties both within a block and in multiple blocks. Unlike traditional concurrent objects, a smart contract's methods are atomic due to the transactional model of computation. That is, a single call to a contract (or a chain of calls to a series of contracts calling each other), is executed *sequentially*—without interrupts—and either terminates after successfully updating the blockchain or aborts and rolls back to its previous configuration before the call.

The notion of "atomicity for free" is deceptive, however, as concurrent behavior can still be observed *at the level of the blockchain*:

[2] That is, unsynchronized concurrent accesses by different threads to a single memory location when at least one of those accesses is a write.

- The order of the transactions included to a block is not determined at the moment of a transaction execution, and, thus, the outcome can largely depend on the ordering with respect to other transactions [27].
- Several programming tasks require the contract logic to be spread across several blockchain transactions (*e.g.*, when contracts "communicate" with the world outside of the blockchain), enabling true concurrent behavior.
- Calling other contracts can be considered to be a kind of *cooperative multitasking*. By cooperative multitasking we mean that multiple threads can run but do not get interrupted unless they explicitly "yield". That is, a call from contract A to contract B can be considered to be a yield from contract A's perspective, with contract B yielding when it returns. The key point for smart contracts is that **contract B can run code that was unanticipated by contract A's designer**, which makes the situation much closer to a concurrent setting than a typical sequential one.[3] In particular, contract B can modify state that contract A may assume is unchanged during the call. This is the essence of The DAO bug [9], in which contract B made a call back into contract A to modify A's local state before returning [27]. However, reentrancy is not the only way this kind of error can manifest, since:
- It is not difficult to imagine a scenario in which a certain contract is used as a *service* for other parties (users and contracts), managing the access to a shared resource and, in some sense, serving as a concurrent library. As multi-contract transactions are becoming more ubiquitous, various interference patterns can be observed and, thus, should be accounted for.

Our goals and motivation. Luckily, the research in concurrent and distributed programming conducted in the past three decades provides a large body of theoretical and applied frameworks to code, specify, reason about, and formally verify concurrent objects and their implementations. The goal of this paper is thus twofold. First, we are going to provide a brief overview of some known concurrency issues that can occur in smart contracts, characterizing the problems in terms of more traditional concurrency abstractions. Second, we are aiming to build an intuition for "good" and "bad" contract behaviors that can be identified and verified/detected correspondingly, using existing formal methods developed for reasoning about concurrency.

2 Deployed Examples of *Concurrentesque* Behavior

Here we discuss two contracts that have been deployed on the Ethereum blockchain that each illustrate different aspects of concurrent-type behavior. The BlockKing contract, like many others on the Ethereum blockchain today, implements a simple gambling game [2]. Although BlockKing is not heavily used, we study it because it showcases a potential use of the Oraclize service [4], which is a service that allows contracts to communicate with the world outside of the blockchain and thus invites true concurrency. Since the early adopters of the

[3] A better term would be "uncooperative multitasking" under the circumstances.

Oraclize service wrote it as a demonstration of the service and has made its source code freely available, it is likely that many other contracts that wish to use Oraclize will mirror it in their implementations.

The second example we discuss is the widely-studied bug in the DAO contract [1]. The DAO established an owner-managed venture capital fund with more than 18,000 investors; at its height it attracted more than 14% of all Ether coins in existence at that time. The subsequent attack on it cost investors approximately 3.6 million Ether, which at that time was worth approximately USD 50 million. The DAO employed what we call "uncooperative multitasking", in that when the DAO sent money to a recipient then that recipient was able to run code that interfered (via reentrancy) with the DAO's contract state that the DAO assumed would not change during the call.

2.1 The BlockKing Contract

The gamble in BlockKing works as follows. At any given time there is a designated "Block King" (initially the writer of the contract). When money is sent to the contract by a sender s, a random number j is generated between 1 and 9. If the current block number modulo 10 is equal to j then s becomes the new Block King. Afterwards, the Block King gets sent a percentage of the money in the contract (from 50% to 90% depending on various parameters), and the writer of the contract gets sent the balance.

Generation of good quality random numbers is often difficult in deterministic systems, especially in a context in which all data is publicly stored—and in which there are financial incentives for attackers. Accordingly, BlockKing utilizes the services of a trusted party, Wolfram Alpha, to generate its random numbers using the Oraclize service. Assuming Oraclize is well-behaved, this strategy for random number selection should be very difficult for attackers to predict.

The code for BlockKing is 365 lines long, but the lines of particular interest are given in Fig. 1; line numbers here refer to the actual source code of the contract as given by Etherscan [2]. The `enter` function is called when money is sent to the contract. It sets some contract variables (lines 299–301) and then sends a query to the Oraclize service (line 303).

The `oraclize_query` function raises an event visible in the "real world" before returning to its caller, which then exits (line 304). In the real world the Oraclize servers monitor the event logs, service the request (in this case by contacting the Wolfram Alpha web service), and then make a fresh call into the originating contract at a designated callback point (line 306 in BlockKing). Between the event and its callback, many things can occur, in the sense that the the blockchain can advance several blocks between the call to `oraclize_query` and the resumption of control at `__callback`. During this time the state of the blockchain, and even of the BlockKing contract itself, can have changed drastically. In other words, *this is true concurrent behavior on the blockchain.*

What can go wrong? Suppose that multiple gamblers wish to try their luck in a short period of time (even within the same block). The contract makes no attempt to track this behavior. Accordingly, each new contestant will overwrite

```
293  function enter() {
294   // 100 finney = .05 ether minimum payment otherwise refund payment and stop contract
295   if (msg.value < 50 finney) {
296    msg.sender.send(msg.value);
297    return;
298   }
299   warrior = msg.sender;
300   warriorGold = msg.value;
301   warriorBlock = block.number;
302   bytes32 myid =
303    oraclize_query(0,"WolframAlpha","random number between 1 and 9");
304  }
305
306  function __callback(bytes32 myid, string result) {
307   if (msg.sender != oraclize_cbAddress()) throw;
308   randomNumber = uint(bytes(result)[0]) - 48;
309   process_payment();
310  }
311
312  function process_payment() {

     ...

339   if (singleDigitBlock == randomNumber) {
340    rewardPercent = 50;
341    // If the payment was more than .999 ether then increase reward percentage
342    if (warriorGold > 999 finney) {
343     rewardPercent = 75;
344    }
345    king = warrior;
346    kingBlock = warriorBlock;
347   }
```

Fig. 1. BlockKing code fragments [2].

the previous one's data (the critical `warriorBlock` and `warrior` variables) in lines 299–301. When the callbacks do eventually occur, the last contestant in the batch will enjoy multiple chances to win the throne curtesy of the earlier contestants in that batch who payed for the other callbacks! The culprit is lines 339–347 from the `process_payment` function, called as the last line of the `__callback` function in line 309.

Each time the `process_payment` function is called the least significant digit of `warriorBlock` is computed and stored into the variable `singleDigitBlock`.[4] Each time the `process_payment` function is called by `__callback` he has a new chance to match the random number in line 339. If the numbers do match, then that final contestant is crowned on line 345.

2.2 The DAO Contract

The source code for the DAO is 1,239 lines and markedly more complex than BlockKing [23]. Since much has already been written about this bug (*e.g.* [9,27]), we present in Fig. 2 only the key lines. The problem is the order of line 1012, which (via a series of further function calls) sends Ether to `msg.sender`, and line 1014, which zeros out the balance of `msg.sender`'s account.

In a sequential program, reordering two independent operations has no effect on the ultimate behavior of the program. However, in a concurrent program

[4] For reasons that seem rather strange to us, this modulus is computed very inefficiently in lines 315–338 of the contract, which we elide to save space.

```
1010   // Burn DAO Tokens
1011   Transfer(msg.sender, 0, balances[msg.sender]);
1012   withdrawRewardFor(msg.sender); // be nice, and get his rewards
1013   totalSupply -= balances[msg.sender];
1014   balances[msg.sender] = 0;
1015   paidOut[msg.sender] = 0;
1016   return true;
1017 }
```

Fig. 2. DAO code fragment [23].

the effect of a sequentially-harmless reorder can have significant effect since the order in which operations occur can affect how the threads interfere. In the DAO, sending the Ether in line 1012 "yields" control, in some multitasking sense, to any arbitrary (and thus potentially malicious) contract located at `msg.sender`.

Unfortunately, the DAO internal state still indicates that the account is funded since its account balance has not yet been zeroed out in line 1014. Accordingly, a malicious `msg.sender` can initiate a second withdrawal by calling back into the DAO contract, which will in turn send a second payment when control reaches line 1012 again. In fact, the malicious `msg.sender` can then initiate a third, fourth, *etc.* withdrawal, all of which will result in payment. Only at the end is his account zeroed out, after being paid many multiples of its original balance.

Previous analyses of this bug have indicated that the problem is due to recursion or unintended reentrancy. In a narrow sense this is true, but in a wider sense what is going on is that sequential code is running in what is in many senses a concurrent environment.

3 Interference and Synchronization

Having showed that concurrent-type behavior exists and causes problems in real contracts on the Blockchain, we will now examine other ways that our *concurrent-objects-as-contracts* viewpoint can help us understand how contracts can behave on the blockchain.

3.1 Atomic Updates in Shared-Memory Concurrency

Figure 3 depicts a canonical example (presented in a Java 8-like pseudocode) of a wrongly used concurrent object, which is supposed to implement an "atomic" counter with methods `get` and `set`. The implementation of the concurrent counter on the left is obviously *thread-safe* (*i.e.*, *data race*-free), thanks to the use of `synchronized` primitives [17]. What is problematic, though, is how an instance of the `Counter` class is used in the multithreaded client code on the right.

Specifically, with two threads running in parallel and their operations interleaving, the call to `incr()` within `thread2`'s body could happen, for instance, between the assignment to `a` and the call `c.set(a + 1)` within the `incr()` call of

```
class Counter {
  private int x = 0;

  /** Return current value */
  synchronized int get() {
    return x;
  }

  /** Set x to be v */
  synchronized int set(int v) {
    int t = x;
    x = v;
    return t;
  }
}
```

```
final Counter c = new Counter();

void incr() {
  int a = c.get();
  int b = c.set(a + 1);
  assert (a == b);
}

// In the main method
Runnable thread1 = () ->
  { incr(); }

Runnable thread2 = () ->
  { incr(); }

thread1.run(); thread2.run();
```

Fig. 3. A concurrent counter (left) and its two-thread client application (right).

`thread1`. This would invalidate the condition in the following `assert` statement, making the overall program fail *non-deterministically* for a certain execution!

The issue arises because the implementation of `incr()` on top of `Counter` does not provide the *atomicity guarantees*, expected by the client code. Specifically, the code on the right is implemented in the assumption that there will be *no interference* between the statements of `incr()`, hence the counter `c` is going to be incremented by `1`, and `a` and `b` will be the same by the end of its execution. Indeed, this is not always the case in the presence of concurrently running `thread2`, and not only `a` and `b` will be different, the later call to `c.set()` will also "overwrite" the result of the earlier one.

A better designed implementation of `Counter` could have instead provided an *atomic* implementation of `incr()`, implemented via a version of *fetch-and-increment* operation [20, §5.6], via explicit locking, or by means of Java's `synchronized` keyword. However, given the only two methods, `get` and `set`, the implementation of `Counter` has synchronization properties of an atomic register whose *consensus number* [20, §5.1] (*i.e.*, the number of concurrent threads that can unambiguously agree on the outcomes of `get` and `set`) is exactly 1. Therefore, it is fundamentally impossible to implement an atomic incrementation of `c` by using only `get` and `set`, and without relying on some additional synchronization, by giving priorities to certain preordained threads.

Perhaps a bit surprisingly, even though the implementation of `Counter` from Fig. 3 is not flawed by itself, its weak atomicity properties render it quite useless in the presence of an unbounded number of threads, making it virtually impossible to make any *stable* (*i.e.*, resilient with respect to concurrent changes) assumptions about its internal state.

3.2 Atomic Updates in Concurrent Blockchain Transactions

The left part of Fig. 4 shows a smart contract, implemented in Solidity [15], with functionality and methods reminiscent to those of an atomic concurrent counter. The function `get` allows one to query the contract for the current balance, associated with some fixed address `id`, whereas the `set` function allows one to update balance with the new balance, taken from the message via `msg.value`, sending back the old amount and returning it as a result.

```
contract Counter {
  address public id;
  uint private balance;

  function get() returns (uint) {
    return balance;
  }

  function set() returns (uint) {
    uint t = balance;
    balance = msg.value;
    msg.sender.send(t);
    return t;
  }}
```

```
// ...
// Same code as in Counter

function testAndSet(uint expected)
  returns (uint) {
  uint t = balance;
  if (t == expected) {
    balance = msg.value;
    msg.sender.send(t);
    return t;
  } else {
    throw;
  }
}
```

Fig. 4. A counter contract (left) and a synchronizing `testAndSet` method (right).

Since the bodies of both `get` and `set` are going to be executed sequentially in the course of some transactions, neither there is any need to synchronized them, nor there is any explicit way to do so in Solidity. However, it is not difficult to observe that as an implementation of the simplest possible storage (*e.g.*, for some `id`-related funds), used by multiple different parties to update it's balance, the `Counter` contract is as useless as its Java counterpart from Fig. 3.

For instance, imagine that two parties, unaware of each other try to increment the amount, stored by an instance of `Counter` by a certain value. Since the contract does not provide a way for them to do it in one operation, they will have to first query the amount via `get` and then try to change it via `set` function, following the same pattern as the implementation of `incr` from Fig. 3. Indeed, both these calls can be accomplished in a single transaction, which would make the execution sequential. However, because of the limited gas requirement,[5] it is ill-advised to call more than one external contract in the course of execution. Furthermore, the call to `get` can be performed by a client, external to the blockchain, which would mean that the consecutive calls to `get` and `set` will end up in *two different* transactions. If this is the case, those calls might interfere with other transactions, launched by multiple parties trying to modify `Counter`

[5] This is a standard way in Ethereum to ensure that execution of a contract terminates: by supplying it with a limited amount of "gas", used as a fuel for execution steps.

at the same time, making us face the familiar problem: the result of calling the function `set` cannot be predicted out of the local observations.

The cause of the described problem, both in the shared-memory and blockchain cases, is the lack of *strong synchronization primitives*, allowing one to simultaneously observe and manipulate with the counter in the presence of concurrent executions. One solution to the problem, which would make it possible to increment the counter atomically, is to enhance the counter with the `testAndSet` function (right part of Fig. 4). This function implements the check/update logic similar to the *compare-and-swap* primitive [20, §5.8], (known as `CMPXCHG`, on the Intel x86 and Itanium architectures), as a way to implement synchronization between multiple threads. The consensus number of `testAndSet` (and some other similar *Read-Modify-Write* primitives) is known to be ∞, hence it is strong enough to allow an arbitrary number of concurrent parties agree on the outcome of the operation.

Notes on formal reasoning and verification. The modern formal approaches for runtime concurrency verification, based on exploring dynamic execution traces and summarizing their properties, provide efficient tools for detecting the violations of atomicity assumptions, and the lack of synchronization [26]. For instance, by translating our contract to the corresponding shared-memory concurrent object, one would be able to use the existing tools to summarize its traces [13], thus, making it possible to observe undesired interaction patterns.

4 State Ownership and Permission Accounting

A different way to prohibit the unwelcome interference on a contract's state is to engineer a tailored permission accounting discipline, controlling the set of operations allowed for different parties.

Let us first notice that the problems exhibited by the two-thread example in Fig. 3 and preventing one from asserting anything about its state `x` could be avoided if we enforced a restricted access discipline: for instance, by stating that at any moment at most one thread can query/modify its state. This would grant the corresponding thread an exclusive *ownership* [30] over the object, thus, justifying any assertions made locally from this thread about the object's state.

The unique ownership is traditionally ensured in Ethereum's contracts by disallowing any other party, but a dedicated *owner*, make critical changes in the contract state. For instance, Fig. 5 (left) shows an altered version of the `Counter` contract, so no other party can interact with it but its "owner". The ownership discipline is enforced by Solidity's mechanism of `modifier`s, allowing one to provide custom dynamically checked pre-/postconditions for functions. In our example, the `byOwner` modifier will enforce that the functions `get` and `set` will be only invoked on behalf of a fixed party—the `owner` of the contract.

This is a rather crude solution to the interference problem, as it would mean to exclude any concurrent interaction at a contract whatsoever. It is quite illuminating, though, from a perspective on thinking of contracts as concurrent

```
contract Counter {
  address public owner;
  uint private balance;

  modifier byOwner() {
    if (msg.sender != owner) throw;
    _
  }

  function get() external byOwner
    returns (uint) {
    return balance;
  }

  function set() external byOwner
    returns (uint) {
    uint t = balance;
    balance = msg.value;
    msg.sender.send(t);
    return t;
  }}
```

```
// Same declarations as in Counter

mapping (address => bool) readers;

// Initialized with 0x0
address writer;

modifier canRead() {
  if (msg.sender != writer ||
      !readers[msg.sender]) throw;
   _
}

modifier canWrite() {
  if (msg.sender != writer) throw;
  _
}

function acquireReadLock() returns (bool) {
  if (writer == 0x0) {
    readers[msg.sender] = true;
  } else return false;
}

// ... Other synchronization primitives
```

Fig. 5. An exclusively-owned (left) and Read/Write-locked (right) contract.

objects, allowing us to immediately apply our analogy: *accounts are threads*. Indeed, by imposing a specific ownership discipline on a contract as shown in Fig. 5 is similar to enhancing its Java counterpart with an explicit check of `Thread.currentThread().getId()`.

Let us now try to push the analogy between accounts and threads a bit further by designing a version of a counter with more elaborated access rights. In particular, we are going to ensure that as long as there are accounts (aka "threads") "interested" in having its value immutable (as their internal logic might rely on its immutability), no other party may be allowed to modify it. Similarly, if at the moment there is exactly one party that holds a unique permission to modify the counter, no other parties may be allowed to read it. The solution to this synchronization problem is well-known in a concurrency community by the name *Read/Write lock* [6]. Its implementation requires keeping track of threads currently reading and writing to the shared object, so a thread should explicitly *acquire* the corresponding permission before performing a read/write operation, and then should *release* it upon finishing.

The right part of Fig. 5 shows the essential fragments of the Read/Write-locked contract implementation. The two new fields, `readers` and `writer` keep track of the currently active readers and writers. The new modifiers `canRead` and `canWrite` are to be used for the omitted `get` and `set` operations correspondingly. Finally, `acquireReadLock` allows its caller to acquire the lock as long as there is no active writer in the system, by registering it in the `readers` mapping.

As we can see, the accounts-as-threads is a rather powerful analogy, suggesting a number of solutions to possible synchronization problems that can be taken verbatime from the concurrency literature. The only drawback of the presented solution is the fact that it is rather monolithic: the contract now combines the

functionality of the data structure (*i.e.*, the counter) and that of a synchronization primitive (*i.e.*, a lock). We will discuss possible ways to improve the modularity of the implementation in Sect. 5.

Notes on formal reasoning and verification. Formal reasoning about permission accounting and separation of state access is a long studied topic in the shared-memory concurrency literature (see, *e.g.*, [8] for an overview). Formalisms, such as Concurrent Separation Logic and [30] Fractional/Counting permissions [6] provide a flexible way to define the abstract ownership discipline and verify that a particular implementation follows it faithfully. For instance, our Read/Write lock contract can be formally proven *safe* (*i.e.*, prohibiting concurrent write-modifications) using a formal model of permissions by Bornat *et al.* [6].

5 Discussion

5.1 Composing the Contracts

The locking contract "pattern", considered in Sect. 4, has a significant drawback: its design is *non-modular*. That is, the locking machinery is implemented by the contract itself rather than by a third-party library. This is at odds with good practices of software engineering, in which it is advised to implement synchronization primitives, such as ordinary and reentrant locks, as standalone libraries, which can be used for managing access client-specific resources.

But once the lock logic is factored out of the contract, the reasoning about the contract's behavior becomes significantly more difficult, as, in order to prove the preservation of its internal invariants, one needs to be aware of the properties of the extracted locking protocol, such as, *e.g.*, uniqueness of a writer, which are external to the contract. In other words, verification of a contract can no longer be conduced in an *isolated* manner and will require building a model that allows reasoning about a contract interacting with other, rigorously specified contracts. The idea of disentangling the logic of contracts is not inherent to our concurrent view and is paramount in the existing good practices of contract development. For instance, the same idea is advocated as a way to implement *upgradable* contracts in Ethereum through introducing and additional level of indirection [11]. Having a "contract factory", implemented as another contract, which can be invoked by any party, poses verification challenges similar to those of proving the safety properties of *higher-order* concurrent object (*i.e.*, an object, that is manipulating with other objects) [19].

The idea of compositional reasoning and verification of mutually-dependent and higher-order concurrent objects using concurrency logics has been a subject of a large research body in the past decade [12,33,34,37]. Most of those approaches focus on a notion of *protocol*, serving as an abstract interface of an object's behavior in the presence of concurrent updates, while hiding low-level implementation details (*i.e.*, the actual code). We believe, that by leveraging our analogy, we will be able to develop a method for modular verification of such multi-contract interactions.

5.2 Liveness Properties

With the introduction of locks and exclusive access, another concurrency-related issue arises: reasoning about *progress* and *liveness* properties of contract implementations. For instance, it is not difficult to imagine a situation, in which a particular account, registered as a "reader" in our example from Fig. 5, might never release the reader-lock, thus, blocking everyone else from being able to change the contract's state in the future. The liveness in this setting would mean that *eventually something good happens*, meaning that any party is properly incentivised to release the lock. In a concurrency vocabulary, such an assumption can be rephrased as *fairness* of the system scheduler, making it possible to reuse existing proof methods for modular reasoning about progress [25] and termination [18] in of single- and multi-contract executions.

6 Related Work

Formal reasoning about smart contracts is an emerging and exciting topic, and suitable abstractions for describing a contract's behavior are a subject of active research. In this section, we relate our observations to the existing results in formalizing and verifying contract properties, outlining promising areas that would benefit from our concurrency analogy.

6.1 Verifying Contract Implementations

Since the DAO bug [9], the Ethereum community has been focusing on preventing similar errors, with the aid of general-purpose tools for program verification.

At the moment, contracts written in Solidity can be annotated with Hoare-style pre/postconditions and translated down to OCaml code [32], so they become amenable to verification using the Why3 tool, which uses automation to discharge the generated verification conditions [16]. This approach is efficient for verifying basic safety properties of Solidity programs, such as particular variables always being within certain array index boundaries, and preservation of general contract invariants (typically stated in a form if linear equations over values of `uint`-valued variables) at the method boundaries and before performing external contract calls—precisely what was violated by the DAO contract.

Bhargavan *et al.* have recently implemented a translation from a subset of Solidity (without loops and recursion) [5] into F*—a programming language and verification framework, based on dependent types [35]. They also provided a translator from EVM bytecode to F* programs. Both these approaches made it possible to use F* as a uniform tool for verification of contract properties, such as invariant preservation and absence of unhandled exceptions, which were encoded as an effect via F*'s support for indexed Hoare monad [36]. A similar approach to specify the behavior of contracts and based on dependent types has been adopted by Pettersson and Edström [31], who implemented a small effect-based contract DSL as a shallow embedding into Idris [7], with the executable code extracted to Serpent [14], a Python-style contract language.

Hirai has recently formalized the entire specification of Ethereum Virtual Machine [22] in Lem [28] with extraction to the Isabelle/HOL proof assistant, allowing mechanized verification of contracts, compiled to EVM bytecode, for a number of safety properties, including assertions on mutable state and the absence of potential reentrancy. Unlike the previous approaches, Hirai's formalization does not provide a syntactic way to construct and compose proofs (*e.g.*, via a Hoare-style program logics), and all reasoning about contract behavior is conducted out of the low-level execution semantics [38].

In contrast with these lines of work, which focus predominantly on *low-level* safety properties and invariant preservation, our observations hint a more high-level formalism for capturing the properties of a contract behavior and its communication patterns with the outside world. In particular, we consider communicating state-transition systems (STSs) [29] with abstract state as a suitable formalism for proving, *e.g.*, trace and liveness properties of contract executions using a toolset of established tools, such as TLA+ [24]. In order to connect such an abstract representation with low-level contract code, one will have to prove a *refinement* [3] between the high-level and the low-level representations, *i.e.*, between an STS and the code. In some sense, finding a suitable contract invariant and proving it via Why3 or F* may be considered as proving a refinement between a *one-state* transition system, such that the only state is what is described by the invariant, and an implementation that preserves it. However, we expect more complicate STSs will be required in order to reason about contracts with preemptive concurrency.

6.2 Reasoning About Global Contract Properties

The observation about some contracts being prone to unintentional or adversarial misuse due to the interference phenomenon has been made by Luu *et al.* [27]. They characterised the problem similar to what's exhibited by our counter example in Sect. 3 as *transaction-ordering dependency* (TOD), which under our concurrency analogy can be generalized as a problem of unrestricted interference. The solution to the TOD-problem, suggested by Luu *et al.*, required changing the semantics of Ethereum transactions, providing a primitive, similar to our `testAndSet` from Fig. 4. While the advantage of such an approach is the absence of the need to modify the already deployed contracts (only the client code interacting with them needs to be changed), it requires all involved users to upgrade their client-side applications, in order to account for the changes. In essence, Luu *et al.*'s solution targets a very specific concurrency pattern: strengthening synchronization, provided by atomic registers, by adding a blockchain-supported *read-modify-write* primitive. Realizing the nature of the problem, hinted by our analogy, might instead suggest alternative *contract-based* solutions, such as, *e.g.*, engineering a locking proxy contract. The disadvantage of this approach is, however, the need to foresee this behavior at the moment of designing and deploying a contract. That said, such an ability to model this behavior is precisely what, we believe, our analogy enables.

7 Conclusion

We believe that our analogy between *smart contracts* and *concurrent objects* can provide new perspectives, stimulate research, and allow effective reuse of existing results, tools, and insights for understanding, debugging, and verifying complex contract behaviors in a distributed ledger. As any analogy, ours should not be taken verbatim: on the one hand, there are indeed issues in concurrency, which seem to be hardly observable in contract programming; on the other hand, smart contract implementers should also be careful about notions that do not have direct counterparts in the concurrency realm, such as gas-bounded executions and management of funds.

To conclude, we leave the reader with several speculations, inspired by our observations, but neither addressed nor disproved:

- A common concurrency challenge in non garbage-collected languages is to track the uniqueness of heap locations, which can be later reclaimed and repurposed—an issue dubbed *the ABA problem* [10]. With the lack of due caution, the ABA problem may lead to the violation of the object's state integrity. Can we imagine a similar scenario in a multi-contract setting?
- Continuing the analogy, if one sees a blockchain as a shared state, then the mining protocol defines the priorities for scheduling. Can we leverage the insights from efficient concurrent thread management in order to analyze and improve the existing distributed ledger implementations?
- *Linearizability* [21] (aka *atomicity*) is a standard notion of correctness for specifying high-level behavior of lock-free concurrent objects. What would be an equivalent de-facto notion of consistency for composite contracts with multi-transactional operations, such as BlockKing?

Acknowledgements. Sergey's research is supported by EPSRC grant EP/P009271/1. Hobor's research is funded by Yale-NUS College R-607-265-045-121.

References

1. The DAO. https://en.wikipedia.org/wiki/The_DAO_(organization)
2. BlockKing contract (2016). https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1
3. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS, pp. 165–175. IEEE Computer Society (1988)
4. Bertani, T.: Oraclize (2016). http://www.oraclize.it
5. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: short paper. In: PLAS, pp. 91–96. ACM (2016)
6. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: POPL, pp. 259–270. ACM (2005)
7. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP, pp. 133–144. ACM (2013)

8. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. ACM SIGLOG News **3**(3), 47–65 (2016)
9. Buterin, V.: Critical update re: DAO vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability
10. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In: ISORC, pp. 185–192. IEEE Computer Society (2010)
11. Dimitrova, E.: Writing upgradable contracts in Solidity. https://blog.colony.io/writing-upgradeable-contracts-in-solidity-6743f0eecc88. Accessed 3 Feb 2017
12. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_24
13. Emmi, M., Enea, C.: Symbolic abstract data type inference. In: POPL, pp. 513–525. ACM (2016)
14. Ethereum Foundation: The Serpent Contract-Oriented Programming Language. https://github.com/ethereum/serpent
15. Ethereum Foundation: The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity
16. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
17. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley, Boston (2006)
18. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL, pp. 16–28. ACM (2009)
19. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA, pp. 355–364 (2010)
20. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. M. Kaufmann, Burlington (2008)
21. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Prog. Lang. Syst. **12**(3), 463–492 (1990)
22. Hirai, Y.: Formalization of Ethereum Virtual Machine in Lem. https://github.com/pirapira/eth-isabelle. Accessed 3 Feb 2017
23. Jentzsch, C.: The DAO (2016). https://etherscan.io/address/0xffbd72d37d4e7f64939e70b2988aa8924fde48e3
24. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
25. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: POPL, pp. 385–399. ACM (2016)
26. Lin, Y., Dig, D.: CHECK-THEN-ACT misuse of java concurrent collections. In: ICST, pp. 164–173. IEEE Computer Society (2013)
27. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS, pp. 254–269. ACM (2016)
28. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. In: ICFP, pp. 175–188. ACM (2014)
29. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 290–310. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_16

30. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comp. Sci. **375**(1–3), 271–307 (2007)
31. Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Sweden (2016)
32. Reitwiessner, C.: Formal Verification for Solidity Contracts. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts. Accessed 3 Feb 2017
33. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI, pp. 77–87. ACM (2015)
34. Svendsen, K., Birkedal, L., Parkinson, M.: Modular reasoning about separation of concurrent data structures. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 169–188. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_11
35. Swamy, N.,Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP, pp. 266–278. ACM (2011)
36. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F*. In: POPL, pp. 256–270. ACM (2016)
37. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In: ICFP, pp. 377–390. ACM (2013)
38. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014). http://gavwood.com/paper.pdf