The Distributed Computing Column

BY

Stefan Schmid

University of Vienna
Währinger Strasse 29, AT - 1090 Vienna, Austria

How to add concurrency to smart contracts*

Thomas Dickerson†, Paul Gazzillo‡, Maurice Herlihy†, Eric Koskinen‡

†Brown University
‡Stevens Institute of Technology

Abstract

Modern cryptocurrency systems, such as the Ethereum project, permit complex financial transactions through scripts called *smart contracts*. These smart contracts are executed many, many times, always without real concurrency. First, all smart contracts are serially executed by *miners* before appending them to the blockchain. Later, those contracts are serially re-executed by *validators* to verify that the smart contracts were executed correctly by miners.

Serial execution limits system throughput and fails to exploit today's concurrent multicore and cluster architectures. Nevertheless, serial execution appears to be required: contracts share state, and contract programming languages have a serial semantics.

This paper presents a novel way to permit miners and validators to execute smart contracts in parallel, based on techniques adapted from software transactional memory. Miners execute smart contracts speculatively in parallel, allowing non-conflicting contracts to proceed concurrently, and "discovering" a serializable concurrent schedule for a block's transactions. This schedule is captured and encoded as a deterministic *fork-join* program used by validators to re-execute the miner's parallel schedule deterministically but concurrently.

1 Introduction

Cryptocurrencies such as Bitcoin [17] or Ethereum [9] are very much in the news. Each is an instance of a *distributed ledger*: a publicly-readable tamper-proof record of a sequence of events. Simplifying somewhat, early distributed

*A full technical version of this paper can be found at `https://doi.org/10.1145/3087801.3087835`. This work is supported in part by National Science Foundation CCF Awards #1421126, #003991, and #1618542.

ledgers, such as Bitcoin's, work like this: *clients* send *transactions*[1] to *miners*, who package the transactions into *blocks*. Miners repeatedly *propose* new blocks to be applied to the ledger, and follow a global consensus protocol to agree on which blocks are chosen. Each block contains a cryptographic hash of the previous block, making it difficult to tamper with the ledger. The resulting distributed data structure, called a *blockchain*, defines the sequence of transactions that constitutes the distributed ledger[2].

Modern blockchain systems often interpose an additional software layer between clients and the blockchain. Client requests are directed to scripts, called *smart contracts*, that perform the logic needed to provide a complex service, such as managing state, enforcing governance, or checking credentials. Smart contracts can take many forms, but here we will use (a simplified form of) the Ethereum model [9].

A smart contract resembles an object in a programming language. It manages long-lived *state*, which is encoded in the blockchain. The state is manipulated by a set of *functions*, analogous to *methods* in many programming languages. Functions can be called either directly by clients or indirectly by other smart contracts. Smart contract languages are typically Turing-complete. To ensure that function calls terminate, the client is charged for each computational step in a function call. If the charge exceeds what the client is willing to pay, the computation is terminated and rolled back.

When and where is smart contract code executed? There are two distinct circumstances. Each smart contract is first executed by one or more *miners*, nodes that repeatedly propose new blocks to append to the blockchain. When a miner creates a block, it selects a sequence of user requests and executes the associated smart contract code for each Ethereum transaction in sequence, transforming the old contract state into a new state. It then records both the sequence of transactions and the new state in the block, and proposes it for inclusion in the blockchain.

Later, when the block has been appended to the blockchain, each smart contract is repeatedly re-executed by *validators*: nodes that reconstruct (and check) the current blockchain state. As a validator acquires each successive block, it replays each of the transactions' contract codes to check that the block's initial and final states match. Each miner validates blocks proposed by other miners, and older block are validated by newly-joined miners, or by clients querying the contract state. Code executions for validation vastly exceed code executions for mining.

Existing smart contract designs limit throughput because they admit no con-

[1]Following blockchain terminology, a transaction is a payment or set of payments, not an atomic unit of synchronization as in databases or transactional memory.

[2]This description omits many important issues, such as incentives, forking, and fork resolution.

currency. When a miner creates a block, it assembles a sequence of transactions, and computes a tentative new state by executing those transactions' smart contracts serially, in the order they occur in the block. A miner cannot simply execute these contracts in parallel, because they may perform conflicting accesses to shared data, and an arbitrary interleaving could produce an inconsistent final state. For Bitcoin transactions, it is easy to tell in advance when two transaction conflict, because input and output data are statically declared. For smart contracts, by contrast, it is impossible to tell in advance whether two contract executions will conflict, because the contract language is Turing-complete.

Miners are rewarded for each block they successfully append to the blockchain, so they have a strong incentive to increase throughput by parallelizing smart contract executions. We propose to allow miners to execute contract codes in parallel by adapting techniques from Software Transactional Memory (STM) [11]: treating each invocation as a speculative atomic action. Data conflicts, detected at runtime, are resolved by delaying or rolling back some conflicting invocations. Treating smart contract invocations as speculative atomic actions dynamically "discovers" a *serializable* concurrent schedule, producing the same final state as a serial schedule where the contract functions were executed in some one-at-a-time order.

But what about later validators? Existing STM systems are *non-deterministic*: if a later validator simply mimics the miner by re-running the same mix of speculative transactions, it may produce a different serialization order and a different final state, causing validation to fail incorrectly. Treating contract invocations as speculative transactions improves miners' throughput, but fails to support deterministic re-execution as required by validators.

Notice, however, that the miner has already "discovered" a serializable concurrent schedule for those transactions. We propose a novel scheme where the miner records that successful schedule, along with the final state, allowing later validators to replay that same schedule in a concurrent but deterministic way. Deterministic replay avoids many of the the miner's original synchronization costs, such as conflict detection and roll-back. Over time, parallel validation would be a significant benefit because validators perform the vast majority of contract executions. Naturally, the validator must be able to check that the proposed schedule really is serializable.

In the remainder of this paper, we show first how techniques from *transactional boosting* [10] can be adapted to permit non-conflicting smart contracts to execute concurrently. We then show how miners running smart contracts concurrently in this manner can instrument their transactions to capture a *fork-join* [1] schedule to be executed by validators, deterministically, verifiably, and in parallel.

```
1  contract Ballot {
2    mapping(address => Voter) public voters;
3    // more state definitions
4    function vote(uint proposal) {
5      Voter sender = voters[msg.sender];
6      if (sender.voted)
7        throw;
8      sender.voted = true;
9      sender.vote = proposal;
10     proposals[proposal].voteCount += sender.weight;
11   }
12   // more operation definitions
13 }
```

Figure 1: Part of the Ballot contract

2 Blockchains and Smart Contracts

In Bitcoin and similar systems, transactions typically have a simple structure, distributing the balances from a set of input accounts to a set of newly-created output accounts. In Blockchains such as Ethereum, however, each block also includes an explicit *state* capturing the cumulative effect of transactions in prior blocks. A Transaction is expressed as executable code, often called a *smart contract*, that modifies that state. Ethereum blocks thus contain both transactions' smart contracts and the final state produced by executing those contacts.

The contracts themselves are stored in the blockchain as byte-code instructions for the Ethereum virtual machine (EVM). Several higher-level languages exist for writing smart contracts. Here, we describe smart contracts as expressed in the Solidity language [18].

Listing 1 is part of the source code for an example smart contract that implements a ballot box [19]. The owner initializes the contract with a list of proposals and gives the right to vote to a set of Ethereum addresses. Voters cast their votes for a particular proposal, which they may do only once. Alternatively, voters may delegate their vote. The contract keyword declares the smart contract (Line 1).

The contract's persistent state is recorded in *state variables*. For Ballot, the persistent state includes fields of scalar type such as the owner (omitted for lack of space). State variables such as voters (declared on Line 2) can also use the built-in Solidity type mapping which, in this case, associates each voter's address with a Voter data structure (declaration omitted for brevity). The keys in this mapping

are of built-in type address, which uniquely identifies Ethereum accounts (clients or other contracts). These state variables are the persistent state of the contract.

Line 4 declares contract *function*, vote, to cast a vote for the given proposal. Within a function there are transient *memory* and *stack* areas such as sender. The function vote first recovers the Voter data from the contract's state by indexing into the voters mapping using the sender's address msg.sender. The msg variable is a global variable containing data about the contract's current invocation. Next, the sender.vote flag is checked to prevent multiple votes. Note that sequential execution is critical: if this code were naïvely run in parallel, it would be vulnerable to a race condition permitting double voting. Ethereum contract functions can be aborted at any time via **throw**, as seen here when a voter is detected attempting to vote twice. The **throw** statement causes the contract's transient state and tentative storage changes to be discarded. Finally, this Ballot contract also provides functions to register voters, delegate one's vote, and compute the winning proposal. The complete Ballot example is available elsewhere[3].

Execution Model: Miners and Validators. When a miner prepares a block for inclusion in the blockchain, it starts with the ledger state as of the chain's most recent block. The miner selects a sequence of new transactions, records them in the new block, and executes them, one at a time, to compute the new block's state. The miner then participates in a consensus protocol to decide whether this new block will be appended to the blockchain.

To ensure that each transaction terminates in a reasonable number of steps, each call to contract bytecode comes with an explicit limit on the number of virtual machine steps that a call can take. (In Ethereum, these steps are measured in "gas" and clients pay a fee to the miner that successfully appends that transaction's block to the blockchain.)

After a block has been successfully appended to the blockchain, that block's transactions are sequentially re-executed *by every node in the network* to check that the block's state transition was computed honestly and correctly. (Smart contract transactions are deterministic, so each re-execution yields the same results as the original.) These *validator* nodes do not receive fees for re-execution.

To summarize, a transaction is executed in two contexts: once by miners before attempting to append a block to the blockchain, and many times afterward by validators checking that each block in the blockchain is honest. In both contexts, each block's transactions are executed sequentially in block order.

[3] `http://solidity.readthedocs.io/en/develop/solidity-by-example.html`

3 Speculative Smart Contracts

This section discusses how miners can execute contract codes concurrently. Concurrency for validators is addressed in the next section.

Smart contract semantics is *sequential*: each miner has a single thread of control that executes one EVM instruction at a time. The miner executes each of the block's contracts in sequence. One contract can call another contract's functions, causing control to pass from the first contract code to the second, and back again. (Indeed, misuse of this control structure has been the source of well-known security breaches [6].) Clearly, even sequential smart contracts must be written with care, and introducing explicit concurrency to contract programming languages would only make the situation worse. We conclude that concurrent smart contract executions must be *serializable*: indistinguishable, except for execution time, from a sequential execution.

There are several obstacles to running contracts in parallel. First, smart contract codes read and modify shared storage, so it is essential to ensure that concurrent contract code executions do not result in inconsistent storage states. Second, smart contract languages are Turing-complete, and therefore it is impossible in general to determine statically whether contracts have data conflicts.

We propose that miners execute contract codes as *speculative actions*. A miner schedules multiple concurrent contracts to run in parallel. Contracts' data structures are instrumented to detect synchronization conflicts at run-time, in much the same way as mechanisms like transactional boosting [10]. If one speculative contract execution conflicts with another, the conflict is resolved either by delaying one contract until the other completes, or by rolling back and restarting one of the conflicting executions. When a speculative action completes successfully, it is said to *commit*, and otherwise it *aborts*.

Storage Operations. We assume that, as in Solidity, state variables are restricted to predefined types such as scalars, structures, enumerations, arrays, and mappings. A *storage operation* is a primitive operation on a state variable. For example, binding a key to a value in a mapping, or reading from a variable or an array are storage operations. Two storage operations *commute* if executing them in either order yields the same result values and the same storage state. For example, in the address-to-Voter Ballot mapping in Listing 1, binding Alice's address to a vote of 42 commutes with binding Bob's address to a vote of 17, but does not commute when deleting Alice's vote. An *inverse* for a storage operation is another operation that undoes its effects. For example, the inverse of assigning to a variable is restoring its prior value, and the inverse of adding a new key-value pair to a mapping is to remove that binding, and so on. The virtual machine system can provide all storage operations with inverses.

The virtual machine is in charge of managing concurrency for state variables such as mappings and arrays. Speculation is controlled by two run-time mechanisms, invisible to the programmer, and managed by the virtual machine: *abstract locks*, and *inverse logs*.

Each storage operation has an associated abstract lock. The rule for assigning abstract locks to operations is simple: if two storage operations map to distinct abstract locks, then they must commute. Before a thread can execute a storage operation, it must acquire the associated abstract lock. The thread is delayed while that lock is held by another thread[4]. Once the lock is acquired, the thread records an *inverse operation* in a log, and proceeds with the operation.

If the action commits, its abstract locks are released and its log is discarded. If the action aborts, the inverse log is replayed, most recent operation first, to undo the effects of that speculative action. When the replay is complete, the action's abstract locks are released.

The advantage of combining abstract locks with inverse logs is that the virtual machine can support very fine-grained concurrency. A more traditional implementation of speculative actions might associate locks with memory regions such as cache lines or pages, and keep track of old and versions of those regions for recovery. Such a coarse-grained approach could lead to many false conflicts, where operations that commute in a semantic sense are treated as conflicting because they access overlapping memory regions. In the next section, we will see how to use abstract locks to speed up verifiers.

When one smart contract calls another, the run-time system creates a *nested* speculative action, which can commit or abort independently of its parent. A nested speculative action inherits the abstract locks held by its parent, and it creates its own inverse log. If the nested action commits, any abstract locks it acquired are passed to its parent, and its inverse log is appended to its parent's log. If the nested action aborts, its inverse log is replayed to undo its effects, and any abstract locks it acquired are released. Aborting a child action does not abort the parent, but a child action's effects become permanent only when the parent commits. The abstract locking mechanism also detects and resolves deadlocks, which are expected to be rare.

The scheme described here is *eager*, acquiring locks, applying operations, and recording inverses. An alternative *lazy* implementation could buffer changes to a contract's storage, applying them only on commit.

A miner's incentive to perform speculative concurrent execution is the possibility of increased throughput, and hence a competitive advantage against other miners. Of course, the miner undertakes a risk that synchronization conflicts

[4]For ease of exposition, abstract locks are mutually exclusive, although it is not hard to accommodate shared and exclusive modes.

among contracts will cause some contracts to be rolled back and re-executed, possibly delaying block construction, and forcing the miner to re-execute code not compensated by client fees. Nevertheless, the experimental results reported below suggest that even a small degree of concurrent speculative execution pays off, even in the face of moderate data conflicts.

4 Concurrent Validation

The speculative techniques proposed above for miners are no help for validators. Here is the problem: miners use speculation to discover a concurrent schedule for a block's transactions, a schedule equivalent to some sequential schedule, except faster. That schedule is constructed non-deterministically, depending on the order in which threads acquired abstract locks. To check that the block's miner was honest, validators need to reconstruct the same (or an equivalent) schedule chosen by the miner.

Validators need a way to deterministically reproduce the miner's concurrent schedule. To this end, we extend abstract locks to track dependencies, that is, who passed which abstract locks to whom. Each speculative lock includes a *use counter* that keeps track of the number of times it has been released by a committing action during the construction of the current block. When a miner starts a block, it sets these counters to zero.

When a speculative action commits, it increments the counters for each of the locks it holds, and then it registers a *lock profile* with the VM recording the abstract locks and their counter values.

When all the actions have committed, it is possible to reconstruct their common schedule by comparing their lock profiles. For example, consider three committed speculative actions, A, B, and C. If A and B have no abstract locks in common, they can run concurrently. If an abstract lock has counter value 1 in A's profile and 2 in C's profile, then C must be scheduled after A.

A miner includes these profiles in the blockchain along with usual information. From this profile information, validators can construct a *fork-join* program that deterministically reproduces the miner's original, speculative schedule. By logging the locking schedule during parallel execution, the miner generates a happens-before graph of transactions according to the order in which they acquire locks and commit. The validators can then use this happens-before graph to generate a fork-join program, where each transaction is a task that joins on any other tasks that precede it in the happens-before graph.

The resulting fork-join program is not speculative, nor does it require inter-thread synchronization other than forks and joins. The validator is not required to match the miner's level of parallelism: using a work-stealing scheduler [1], the

validator can exploit whatever degree of parallelism it has available. The validator does not need abstract locks, dynamic conflict detection, or the ability to roll back speculative actions, because the fork-join structure ensures that conflicting actions never execute concurrently.

To check that the miner's proposed schedule is correct, the validator's virtual machine records a trace of the abstract locks each thread would have acquired, had it been executing speculatively. This trace is thread-local, requiring no expensive inter-thread synchronization. At the end of the execution, the validator's VM compares the traces it generated with the lock profiles provided by the miner. If they differ, the block is rejected.

Miners have an incentive to publish a block's fork-join schedule along with the block to induce other miners to build on that block. If a miner publishes an incorrect schedule, the error will be detected and that block rejected. A miner could publish a correct schedule equivalent to, but less parallel than the schedule it discovered, it would have no motive to do so because a less parallel schedule makes that block less attractive than competing blocks with more parallel schedules, and the miner will be rewarded only if the other miners choose to build on that block. Because fork-join schedules are published in the blockchain, their degree of parallelism is easily evaluated.

5 Related Work

The notion of smart contracts can be traced back to an article by Nick Szabo in 1997 [20]. Bitcoin [17] includes a scripting language whose expressive power was limited to protect against non-terminating scripts. The Ethereum blockchain [9] is perhaps the most widely used smart contract platform, employing a combination of a Turing-complete virtual machine protected from non-termination by charging clients for contract running times. Solidity [18] is the most popular programming language for programming the Ethereum virtual machine.

Luu *et al.* [15] identify a number of security vulnerabilities and pitfalls in the Ethereum smart contract model. Luu *et al.* [16] also identify perverse incentives that cause rational miners sometimes to accept unvalidated blocks. Delmolino *et al.* [7] document common programming errors observed in smart contracts. The Hawk [14] smart contract system is designed to protect the privacy of participants.

As noted, many of the speculative mechanisms introduced here were adapted from *transactional boosting* [10], a technique for transforming thread-safe linearizable objects into highly-concurrent transactional objects. Boosting was originally developed to enhance the concurrency provided by software transactional memory systems [11] by exploiting type-specific information. Other techniques that exploit type-specific properties to enhance concurrency in STMs include *trans-*

actional predication [3] and *software transactional objects* [12].

There are other techniques for deterministically reproducing a prior concurrent execution. See Bocchino *et al.* [2] for a survey.

Cachin et al. discuss non-deterministic execution of smart contracts in the context of BFT-based permissioned blockchains [4].

6 Conclusion

We have shown, conceptually, that one can exploit multi-core architectures to increase smart contract processing throughput for both miners and validators. First, miners execute a block's contracts speculatively and in parallel, resulting in lower latency whenever the block's contracts lack data conflicts. Miners are incentivized to include in each block an encoding of the serializable parallel schedule that produced that block. Validators convert that schedule into a deterministic, parallel fork-join program that allows them to validate the block in parallel.

Elsewhere, we have published a more technical discussion showing concrete benchmarking results from a prototype system based on the concepts presented here. [8] Even with only three threads, a prototype implementation yields overall speedups of 1.39x for miners and 1.59x for validators on representative smart contracts. In that paper we also provide a more focused discussion of the correctness of this architecture.

Although our discussion has focused on "permissionless" systems where anyone can participate, the mechanisms proposed here would also be useful for "permissioned" systems, such as Hyperledger [13], where participants are controlled by an authority such as an organization or consortium. For example, in a permissioned blockchain based on Practical Byzantine Fault-Tolerance (PBF) [5], the leader might use speculative execution to discover a concurrent schedule for a block, while particpants in the PBFT protocol would use the concurrent schedule to validate the block before voting.

Future work includes adding support for multithreading to the Ethereum virtual machine, in much the same way as today's Java virtual machines. Our proposal for miners only is compatible with current smart contract systems such as Ethereum, but our overall proposal is not, because it requires including scheduling metadata in blocks and incentivizing miners to publish their parallel schedules. It may well be compatible with a future "soft fork" (backward compatible change), a subject for future research.

In addition to a multithreaded VM, we see room for advancement in programming language support for smart contracts. Designing a language that lends itself to finer-grained concurrency will increase the success of speculative execution thereby increasing throughput. It would also be useful for the language to pro-

vide better control of concurrency, helping the smart contract developer maximize throughput while avoiding concurrency pitfalls.

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[2] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[3] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: High-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM.

[4] Christian Cachin, Simon Schubert, and Marko Vukolic. Non-Determinism in Byzantine Fault-Tolerant Replication. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[6] DAO. Thedao smart contract. Retrieved 8 February 2017.

[7] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab*, pages 79–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[8] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 303–312, New York, NY, USA, 2017. ACM.

[9] Ethereum. `https://github.com/ethereum/`.

[10] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[12] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 31:1–31:16, New York, NY, USA, 2016. ACM.

[13] Hyperledger white paper. `http://www.the-blockchain.com/docs/Hyperledger%20Whitepaper.pdf`.

[14] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, 2015.

[15] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269, 2016.

[16] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[18] Solidity documentation. `http://solidity.readthedocs.io/en/latest/index.html`.

[19] Solidity documentation: Solidity by example. `http://solidity.readthedocs.io/en/develop/solidity-by-example.html`.

[20] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.