Smart digital contracts: Introduction

Fritz Henglein
University of Copenhagen, henglein@diku.dk
Deon Digital, henglein@deondigital.com

OPLSS 2019
June 25th, 2019
Select references

- Peyton Jones, Eber, *Composing contracts: an adventure in financial engineering (functional pearl)*, ICFP 2001
- Henglein, *Blockchain deconstructed (abstract)*, 2nd Symp. DLT, 2018
Overview

- Tue, 9:00-10:15: Distributed ledger technology and `smart contracts’
- Wed, 14:00-15:15: Algebraic foundations for resource management
- Thu, 9:00-10:15: Compositional contracts and `smart’ contract management
- Thu, 2:00-3:15: Contract analysis (and other topics)
Fritz Henglein
Professor of Programming Languages and Systems
University of Copenhagen

Head of Research
Deon Digital AG

Areas of interest
• Programming language technology
• Theoretical computer science (algorithms, semantics, logic)
• Blockchain technology
• Contract management
• Financial technology
• Enterprise systems

Related background
• European Blockchain Consortium (ebcc.eu)
• Steering committee chair, Innovation network for Finance IT (CFIR.dk)
• Principal investigator, Functional technology for high-performance architectures (FUTHARK)

Academic background, affiliations, guest positions
Why blockchain?
Why blockchain?

Decentralize?

- No: Server-/data center hosted system (trusted/privileged system provider)
- Yes: Tamper-proof logging?

Tamper-proof logging?

- No: Point-to-point communicating systems: RPC, REST, micro services (trusted/privileged data managers)
- Yes: Store and transfer resources?

Store and transfer resources?

- No: Structured P2P storage systems with hash pointers
- Yes: Blockchain/distributed ledger system

Digital contracts: protocols for resource transfers

Automated contract managers/smart contracts: Guaranteeing transactionality (tit-for-tat, consideration)
Blockchain/distributed ledger system

A computer system characterized by

• organizational and technical **decentralization**;
• **tamper-proof recording** of digitally signed (real-world) events and their **evidence**;

**digital resource management:**
• digital **storage, transfer, transportation**, transformation of money, goods, and services

It provides

• consistent, nonrepudiable information across all principals (suppliers, partners, customers, regulators, etc.)
• guarantees against **forging** and **double spending**
Organizational and technical decentralization

• **Technical** decentralization: A distributed peer-to-peer system

• **Organizational** decentralization: No single or select group of organizations controls/has privileged rights to system
  • *Nonpermissioned* ("blockchain"): open and self-authenticating, anybody can host a node, be a user and have multiple/many anonymous identities
  • *Permissioned*: nodes and users are authenticated and identified; may be private, but cannot have multiple identities

• **Governance policy** for regulating membership, functionality, conflict resolution, etc.
Intermezzo: Extended REA accounting model

- **Resource (= asset):** Money, licenses, physical objects (e.g. trucks),...
- **Information:** Data, invoices,...
- **Agent:** Person, company, institution, autonomous device,...
- **Contract:** Specification of obligations, permissions and prohibitions
- **Event:**
  - Atomic event:
    - A transfers R to B
    - A transforms R to R'
    - A informs B of I
    - ...
  - Complex event: Set of events that satisfies a given (sub)contract

Andersen et al, Compositional specification of commercial contracts (2006)
Tamper-proof recording of events and their evidence

- **Event recording**: Events are stored (eventually, probabilistically) *consistently*, that is, every node in the distributed system gives the same answer when queried about them.

- **Tamper-proof**: Stored events cannot subsequently be altered or deleted.

- **Evidence**
  - for atomic events:
    - *signature* (by sending agent), plus
    - *supporting evidence* of event actually having happened, e.g. receipt signature, 3rd party *validation* (“payment has been performed” [*], “parcel has been delivered”), DNA samples (“this is the same timber as received”), GPS-/time-tagged pictures (“the parcel delivered in your driveway”), IoT device messages (“parcel has been scanned at this time and location”), etc
  - for complex events:
    - (mathematical) *proof* that a particular set of events constitutes a correct execution (by all involved parties) of a particular, mathematically well-specified contract.

[*] In blockchain systems: Payment validation = recording on blockchain
Blockchain/DL systems as digital twins

Tamper-proof

Physical world

History in Blockchain

Physical evidence framework

Physical assets and physical events

Alternative state 1

Alternative state 2

Illustration by Boris Düdder
Digital resource management

• System keeps track of **ownership** and **location** of resources
• Guarantees that digitally represented resources can only be **transferred** and **transported**, *never* erroneously or maliciously duplicated
• Can be decomposed into
  • resource preservation
  • credit limit enforcement
Resource preservation and credit limit enforcement

- **Resource preservation**: *Transfers* keep the sum of all resources invariant:
  - A transfers 50 ETH to B: The *sum* of all ETH is the same. Atomically, after event, A has 50 ETH less; B has 50 ETH more.
  - *We allow negative numbers to be transferred and as account balances!*

- **Credit limit enforcement**: A transfer is only *valid* (and *effected*) if the *credit limits* of each agent are respected. For above transfer of 50 ETH:
  - If A owns 60 ETH and has credit limit 0: Valid. (A owns 10 ETH after transfer.)
  - If A owns 30 ETH and has credit limit 0: Invalid. (No transfer. A still owns 30 ETH.)
  - If A owns 30 ETH and has credit limit 20: Valid. (A "owns" -20 ETH after transfer.)

- No-double-spend guarantee = all agents have credit limit 0.
- UTxO = account where complete balance must be transferred.

Henglein, Blockchain deconstructed (2018), 2nd Symp. on DLT
Why adaptive credit limits instead of just zero credit limit?

• **Full-reserve monetary system**: One agent (the central bank) has no credit limit, all others have credit limit 0.

• **Fractional-reserve monetary system**: A designated set of agents (“banks”) have a dynamic non-0 credit limit, all others have credit limit 0.
  - Banks’ dynamic credit limits depend on other assets they own, including expected future repayments as part of the contracts (loans, etc.) they have made.

• **Demand-driven production of physical assets**: A car manufacturer has no credit limit (they produce cars on demand), all others have credit limit 0.
  - A car transfer by a car manufacturer need not be validated (“do they even own it?”); they can produce a new one.
Commutativity theorem, part 1

• **Theorem**: If every agent has an infinite credit limit, then all resource transfers are valid and can be executed in arbitrary order. (Each order results in the same state of ownership.)

• **Corollary**: Contract execution involving \( k \) agents with infinite credit limit requires only consensus by the \( k \) agents (on the particular execution); no event needs to be validated by a 3d party. Usually \( k=2 \).

• Loosely: The Internet with TLS for authentication, reliable message delivery, and hashpointers for tamper-proof recording of authenticated message exchanges is a permissioned blockchain/DL system if there are no credit limits.
Commutative theorem, part 2

• If some agents have finite credit limits, outside validation of their resource transfers is required.
  • Point-to-point communication between the $k$ agents only is insufficient.
  • Some information about resource transfers must be "leaked" to other nodes for validation.
Total event ordering

• Total event ordering: Distributed consensus by all non-Byzantine (= correctly executing) nodes on a particular total order (= sequence) of events arriving at any one non-Byzantine node
  • Very hard problem (impossibility results, tricky algorithms, lower bounds on rounds/complexity, need for randomization, synchronous communication etc.)

• Total event ordering is sufficient for resource transfer validation:
  • agree on total order of events, including resource transfers;
  • validate a transfer if, in that order, all agents’ credit limits are satisfied (= no double-spend).

• Total event ordering is not necessary for resource transfer validation:
  • Most resource transfers commute with each other:
    transfer(A,B,R); transfer(C,D,Q) = transfer(C,D,Q); transfer(A,B,R) if \( \{A,B\} \cap \{C,D\} = \emptyset \)
  • Total event ordering is the main bottleneck in large distributed systems, especially asynchronous ones; nonetheless it is used in most blockchain/DL systems (Bitcoin, Ethereum, Fabric, etc.).
Contract-oriented business architecture with intermediaries

- Resource manager/settlement layer (blockchain/DL system)

- Strategy
  - Contract
  - Strategy
  - Contract
  - Strategy

- Company A
- Company B
- Company C

- Trusted third parties:
  - Exchanges, brokers, clearing houses, IT service and platform providers, government institutions, etc.
Standard blockchain architecture (without digital contracts)

- **Private** *front-end program* (wallet management, trading strategy, etc)
- **Public** *smart contracts* (programs tied to particular blockchain system)
- **Public** *information* (may be ‘off chain’, e.g. in IPFS)
- **Public** *resource manager* (*single* blockchain system)

If decentralized:
- Blockchain/DL system
- Distributed application (“dapp”)
# A Crash Slide on Blockchain and Smart Contracts

<table>
<thead>
<tr>
<th>SMART TERM</th>
<th>WHAT IT ACTUALLY MEANS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blockchain</strong></td>
<td><strong>Distributed Append-Only Transaction Log (Ledger)</strong></td>
</tr>
<tr>
<td><strong>Smart Contract (Code)</strong></td>
<td><strong>Class</strong> (in Java-like language)</td>
</tr>
<tr>
<td><strong>Smart Contract (Executing)</strong></td>
<td><strong>Process</strong> (Object [= Class Instance])</td>
</tr>
<tr>
<td><strong>Object Messages</strong></td>
<td><strong>Ordinary Messages</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Linear Resource Transfers</strong></td>
</tr>
</tbody>
</table>
The price of expressiveness: unpredictability

- Smart contract: any program, written in **Turing-complete, feature-rich** programming language (Solidity, Kotlin, Go, …)
  - + : Expressive, familiar
  - - : **Very undecidable properties** (*), even with *full access to the source code*

- Smart contracts without support for reasoning for *qualitative and quantitative safety* are dangerous
  - Move money, not just bits
  - Invite honeypotting

(*) Undecidable and statically hard and/or cumbersome to analyze
Example: Ethereum Vulnerabilities

LUU, CHU, OCLICK, SAXENA, HOBOR, MAKING SMART CONTRACTS SMARTER (2016)

• Transaction-order dependence: Messages may have different effect depending on their order of arrival
  • Who controls the process scheduler (= message sequencer)? Some miner: Front-running

• Time-stamp dependence: Smart contracts may have different executions depending on the time stamp on a transaction block
  • Who controls the time stamping of transaction blocks? Some miner: Clock manipulation

• Exception handling, gas management fragility: Subtle differences in exception semantics, limited run-time stack
  • Provoking out-of-stack and gas exhaustion exceptions: Any user

• Programming language complexities:
  • Exception handling subtleties (send vs. call)
  • Reentrancy vulnerability (DAO hack)
  • Implicit method forwarding (multi-sig exploit)
Example: Reentrancy vulnerability

```
contract SendBalance {
    mapping (address => uint) userBalances;
    bool withdrawn = false;
    function getBalance(address u) constant returns (uint) {
        return userBalances[u];
    }
    function addToBalance() {
        userBalances[msg.sender] += msg.value;
    }
    function withdrawBalance() {
        if (!msg.sender.call.value(
            userBalances[msg.sender]())(()) { throw; }
        userBalances[msg.sender] = 0;
    }
}
```

Figure 7: An example of the reentrancy bug. The contract implements a simple bank account.

LUU, CHU, OLICKEL, SAXENA, HOBOR, MAKING SMART CONTRACTS SMARTER (2016)
Contract-oriented IT architecture with digital contracts and contract managers (generic smart contracts)

- ...

- Confidential *contract managers*:
  - *Smart contracts* for escrow, trade finance, public funding, etc.
  - Secure cryptographic *exchanges* and *auction systems*
  - *Contract execution* monitoring (encompassing procure-to-pay)
Contract-oriented IT architecture with digital contracts and contract managers (generic smart contracts)

• ...

• Confidential contract managers:
  • Intelligent contract management (monitoring, arbitration, escrow, collateral management, etc.)
  • Transactional resource management of multiple resource managers
Consequences

• Separation of **contract life cycle management** from **contracts**
  • Contracts **portable** (CSL), quantitatively **analyzable, domain-oriented** (‘zero programming’)
  • Contract life cycle managers **generic** (any contract), in **any implementation language** (Kotlin, Go, Java, Haskell,...), **instrumentable, changeable** (adding escrow, collateral management, etc., without changing contracts)

• Separation of **resource management** from **contract management**
  • Increased scalability 1: Event log **per contract**, no/few dependencies across contracts
  • Increased scalability 2: Aggressive partitioning of agents and resources (sharding, channels, etc.)
  • Increased **privacy** (contract and contract execution not disclosed to resource manager)

• Precise, mathematical **semantics**
  • Mathematical guarantees, formal verification, static analysis (no hacks possible)
Observations

• Popular blockchain and DL systems employ replicated state machine architecture (unstructured P2P system): High redundancy, high message traffic

• Total event order consensus: Distributed consensus on a specific total order of transactions is sufficient, but not necessary for resource transfer validation
  • Many possibilities of distribution/parallelization by *partitioning*, increasingly recognized (sharding, state channels)

• Hash pointer graphs have multiple uses.
  • For tamper-proof, *confidential* recording of events making up execution of *specific* contract. (Note: Not confidential, but public, in Ethereum and similar blockchain systems)
  • For tamper-proof, *public* recording of total order of validated resource transfers.
Open problems

• Fully exploit resource transfer commutativity
  • Current blockchain/DL systems solve an unnecessarily hard problem: distributed consensus on total order of all events across all contracts.
  • Exploit that (certain) subsets of transfers can be validated and effected independent of each other.

• Ideas for specialized distributed consensus protocols for scalability:
  • Hierarchical clearing and settlement, as in banking systems with real-time gross settlement via central bank (hierarchical “sharding” by partitioning of agent accounts)
  • Time- and resource-sensitive validation (bigger transfers require more time for validation)
  • Insurance (applying transaction fees to cover losses due to overdrafts detected too late)
  • ...
Plan X

• Programmable platform for distributed storage and applications

• Data model: Raw data (bits), *immutable* location-independent references (value references), mutable references (pointers), and sequences/stream of such
  • Key aspect: Value references, guaranteed immutable and location independent

• Architectural components:
  • XMLStore: Programmable, compositional save/load-architecture for storing and retrieving immutable and mutable data
  • MergeLang: User-definable updating of mutable data, based on 3-way merging
  • Distributed applications: XMLStore with exec-interface for executing code stored as structured data in XMLStore
  • Distributed garbage collection, deduplication, etc: Systems components for memory deallocation, optimization, etc.

• Conceived of in 2001 at ITU. Projects in 2001-2004 at DIKU and ITU.
  • 20 B.S. and M.S. projects, multiple faculty, ~1 GB of code and test data
XMLStore components

XMLStore constructors

Async write

Caching

Kademlia router

Multicast router

FileSystem XMLStore

In-memory XMLStore

SSD disk controller

Magnetic disk controller

Basic XMLStores
Example: Structured P2P store by composing XMLStore components

- Kademlia router
  - Caching
    - Async write
      - FileSystem XMLStore
  - In-memory XMLStore
  - Caching
  - Caching
    - Async write
      - SSD disk controller
  - Caching
    - Async write
      - Magnetic disk controller
  - Caching
    - Async write
      - Magnetic disk controller
Resource ownership consensus by 3-way merging

• Maintain account balance in replicated updatable reference
• Formulate updating as applying an update function to balance
• Merge multiple updates into joint update function during replica synchronization
• Observation: Almost all resource updates commute -> update joining is (almost) commutative
To be continued...

henglein@diku.dk
henglein@deondigital.com