Smart-Contract Value-Transfer Protocols on a Distributed Mobile Application Platform

Patrick Dai[1], Neil Mahi[1], Jordan Earls[1], Alex Norta[2]

[1] Qtum Foundation, Singapore
foundation@qtum.org
[2] Large-Scale Systems Group, Tallinn University of Technology,
Akadeemia tee 15A, 12816 Tallinn, Estonia
alex.norta.phd@ieee.org

Abstract. Blockchain-enabled smart contracts that employ proof-of-stake validation for transactions, promise significant performance advantages compared to proof-of-work solutions. For broad industry adoption, other important requirements must be met in addition. For example, stable backwards-compatible smart-contract systems must automate cross-organizational information-logistics orchestration with lite mobile wallets that support simple payment verification (SPV) techniques. The currently leading smart-contract solution Ethereum, uses computationally expensive proof-of-work validation, is expected to hard-fork multiple times in the future and requires downloading the entire blockchain. Consequently, Ethereum smart contracts have limited utility and lack formal semantics, which is a security issue. This whitepaper fills the gap in the state of the art by presenting the Qtum smart-contract framework that aims for sociotechnical application suitability, the adoption of formal-semantics language expressiveness, and the provision of smart-contract template libraries for rapid best-practice industry deployment. We discuss the Qtum utility advantages compared to the Ethereum alternative and present Qtum smart-contract future development plans for industry-cases applications.

Key words: smart contract, business network model, DAPP, mobile, information logistics, cross-organizational, peer-to-peer, distributed system, e-governance, Qtum framework

1 Introduction

Orchestration and choreography protocols that facilitate, verify and enact with computing means a negotiated agreement between consenting parties, are termed smart contracts. The latter initially find application in diverse domains such as, e.g., financial-technology [6], Internet-of-Things (IoT) applications [33], digital-signing solutions [11]. An essential aspect of smart contracts is a decentralized validation of transactions, initially by means of so-called proof-of-work (PoW) [42]. The core technology that enables smart contracts is a public distributed ledger termed the blockchain, which records transaction events without requir-

ing a trusted central authority. Blockchain technology spreads in popularity with the inception of Bitcoin [23], a peer-to-peer (P2P) cryptocurrency and payment system that comprises a limited set of operations on the protocol layer. Bitcoins use PoW for transaction validation that is computationally expensive and electricity intensive.

In contrast to Bitcoins, many smart-contract systems are equipped with the Turing-complete language Solidity[1] that resembles JavaScript syntax and targets for enactment, e.g., the Ethereum Virtual [44] machine. Ethereum is the defacto leading smart-contract system despite being plagued by several deficiencies. First, proof-of-work transaction validation diminishes scalability to the point where Ethereum is considered to not be feasible for most industry applications. Second, in a recent crowdfunding casestudy, the Ethereum affiliated Solidity smart contract was hacked[2] because of security flaws resulting from a lack in the state of the art with respect to tools for formal verifications [3]. The security flaw resulted in a loss of ca. $50 million. Consequently, Ethereum performed a hardfork resulting in a schism yielding two separate Ethereum versions[3]. Yet another Ethereum hardfork[4] was caused by a denial of service attack, and more hardforks must be expected[5] for realizing proof-of-stake [2] transaction validation and blockchain sharding [20].

More reasons limit widespread Ethereum industry adoption [8]. For example, an inability to automate cross-organizational information-logistics, lacking privacy protecting differentiations between external- versus related internal private contracts, secure and stable virtual machines for blockchains with better performing proof-of-stake [2] transaction validation, formally verifiable smart-contract languages, lite wallets that do not require downloading the entire blockchain, and mobile-device solutions for smart contracts with simple payment verification (SPV) [14]. The latter means that clients merely download block headers when they connect to an arbitrary full node [23].

While Qtum uses the Ethereum Virtual Machine (EVM) for a current lack of more suitable alternatives, according to [19], the EVM has deficiencies such as earlier experienced attacks against mishandled exceptions and against dependencies such as for transaction-ordering, timestamps, and so on. It is also desirable for a smart-contract system to achieve industry-scalability with employing sidechains [10] and unspent transaction outputs (UTXO) [10], achieving compatibility to other blockchain systems such as Bitcoins [23], or Colored coins [36]. Furthermore, an adoption of features from the Bitcoin Lightning Network [35] yields scalability via bidirectional micropayment channels.

[1] http://solidity.readthedocs.io/en/develop/
[2] https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/
[3] https://bitcoinsmagazine.com/articles/ethereum-classic-hard-forks-diffuses-difficulty-bomb-1484350622/
[4] https://cointelegraph.com/news/ethereum-hard-fork-no-4-has-arrived-as-dos-attacks-intensify
[5] https://forum.daohub.org/t/whats-up-with-casper-proof-of-stake-and-sharding/6309

While smart-contract systems such as Ethereum attract attention, a widespread industry adoption does not exist for the above discussed reasons. This whitepaper addresses the gap by specifying the Qtum[6] framework for smart-contract systems that answers the question of how to develop a smart-contract solution to satisfy critical customer requirements for enabling cross-organizational information logistics to reduce costs and time? To establish a separation of concerns, we pose the following sub-questions. What differentiating technological performance advantages do Qtum smart-contract solutions provide? What are critical smart-contract requirements the Qtum framework satisfies? What are the unique features of cross-organizational information logistics automation the Qtum framework aims to support?

The remainder of this whitepaper is structured as follows. First, Section 2 focuses on concrete advantages of the Qtum framework for achieving technologically performance increases in comparison to related solutions. Section 3 gives functional- and quality goals in combination with involved stakeholders for sociotechnically organized smart-contract systems. Section 4 shows how the running case is supported by the Qtum-framework value-transfer protocol. Finally, Section 5 concludes this whitepaper together with discussing limitations, open issues and future development work.

2 Qtum Performance Advantage

One of the primary goals of Qtum is to build the first UTXO-based smart-contract system with a proof-of-stake (PoS) [37] consensus model. The latter means the creator of the next block is chosen based on the held wealth in cryptocurrency. Thus, blocks are usually forged, or minted instead of being mined, there are block rewards in addition to transaction fees and forgers receive a percentage of "interest" for the amount of funds they stake.

Qtum is compatible with the Bitcoin- and Ethereum ecosystems and aims at producing a variation of Bitcoin with Ethereum Virtual Machine (EVM) compatibility. Note that differently to Ethereum, the Qtum EVM is constantly backwards compatible. Pursuing a pragmatic design approach, Qtum employs industry use cases with a strategy comprising mobile devices. The latter allows Qtum promoting blockchain technology to a wide array of Internet users and thereby, decentralizing PoS transaction validation.

The remainder is structured as follows. Section 2.1 compares the advantages of Bitcoin UTXO versus the Ethereum account model. Next, Section 2.2 discusses the consensus platform for the Qtum blockchain. Section 2.3 shows the integration of Qtum contracts into the EVM. Finally, Section 2.4 describes the payment model for Qtum operations.

[6] https://qtum.org/

2.1 UTXO Versus Account Model

In the UTXO model, transactions use as input unspent Bitcoins that are destroyed and as transaction outputs, new UTXOs are created. Unspent transaction outputs are created as change and returned to the spender [1]. In this way, a certain volume of Bitcoins are transferred among different private key owners, and new UTXOs are spent and created in the transaction chain. The UTXO of a Bitcoin transaction is unlocked by the private key that is used to sign a modified version of a transaction. In the Bitcoin network, miners generate Bitcoins with a process called a coinbase transaction, which does not contain any inputs. Bitcoin uses a scripting language for transactions with a limited set of operations[7]. In the Bitcoin network, the scripting system processes data by stacks (Main Stack and Alt Stack), which is an abstract data type following the LIFO principle of Last-In, First-Out.

In the Bitcoin client, the developers use `isStandard()` function [1] to summarize the scripting types. Bitcoin clients support: P2PKH (Pay to Public Key Hash), P2PK (Pay to Public Key), MultiSignature (less than 15 private key signatures), P2SH (Pay to Script Hash), and `OP_RETURN`. With these five standard scripting types, Bitcoin clients can process complex payment logics. Besides that, a non-standard script can be created and executed if miners agree to encapsulate such a non-standard transaction.

For example, using P2PKH for the process of script creation and execution, we assume paying 0.01BTC for bread in a bakery with the imaginary Bitcoin address "Bread Address". The output of this transaction is:

```
OP_DUP OP_HASH160 <Bread Public Key Hash> OP_EQUAL OP_CHECKSIG
```

The operation `OP_DUP` duplicates the top item in the stack. `OP_HASH160` returns a Bitcoin address as top item. To establishes ownership of a bitcoin, a Bitcoin address is required in addition with a digital key and a digital signature. `OP_EQUAL` yields `TRUE (1)` if the top two items are exactly equal and otherwise `FALSE (0)`. Finally, `OP_CHECKSIG` produces a public key and signature together with a validation for the signature pertaining to hashed data of a transaction, returning `TRUE` if a match occurs.

The unlock script according to the lock script is:

```
<Bread Signature> <Bread Public Key>
```

The combined script with the above two:

```
<Bread Signature> <Bread Public Key> OP_DUP OP_HASH160
<Bread Public Key Hash> OP_EQUAL OP_CHECKSIG
```

Only when the unlock script and the lock script have a matching predefined condition, is the execution of the script combination true. It means, the Bread Signature must be signed by matching the private key of a valid Bread Address signature and then the result is true.

Unfortunately, the scripting language of Bitcoin is not Turing-complete, e.g., there is no loop function. The Bitcoin scripting language is not a commonly used programming language. The limitations mitigate the security risks by preventing

[7] https://en.bitcoin.it/wiki/Script

the occurrence of complex payment conditions, e.g., generating infinite loops, or other complicated logic loopholes.

In the UTXO model, it is possible to transparently trace back the history of each transaction through the public ledger. The UTXO model has parallel processing capability to initialize transactions among multiple addresses indicating the extensibility. Additionally, the UTXO model supports privacy in that users can use Change Address as the output of a UTXO. The target of Qtum is to implement smart contracts based on the innovative design of the UTXO model.

Versus the UTXO model, Ethereum is an account based system[8]. More precisely, each account experiences direct value- and information transfers with state transitions. An Ethereum account address of 20 bytes comprises a nounce as a counter for assuring one-time processing for a transaction, the balance of the main internal crypto fuel for paying transaction fees called Ether, an optional contract code and default-empty account storage.

The two types of Ether accounts are on the one hand, private-key controlled external and on the other hand, contract-code controlled. The former code-void account type creates and signs transactions for message transfer. The latter activates code after receiving a message for reading and writing internal storage, creating contracts, or sending other messages.

In Ethereum, balance management resembles a bank account in the real world. Every newly generated block potentially influences the global status of other accounts. Every account has its own balance, storage and code-space base for calling other accounts or addresses, and stores respective execution results. In the existing Ethereum account system, users perform P2P transactions via client remote procedure calls. Although sending messages to more accounts via smart contracts is possible, these internal transactions are only visible in the balance of each account and tracking them on the public ledger of Ethereum is a challenge.

Based on the discussion above, we consider the Ethereum account model to be a scalability bottleneck and see clear advantages of the Bitcoin-network UTXO model. **Since the latter enhances the network effect we wish to offer, an essential design decision for the pending Qtum release is the adoption of the UTXO model.**

2.2 Consensus Management

There are ongoing discussions about consensus and which platform meets the needs of respective project requirements. The consensus topics most widely discussed are: PoW [41], PoS [2], Dynamic PoS[9], and Byzantine Fault Tolerance [7] as discussed by HyperLedger. The nature of consensus is about achieving data consistency with distributed algorithms. Available options are, e.g., the Fischer Lynch and Paterson theorem [5] that states consensus cannot be reached without 100% agreement amongst nodes.

[8] https://github.com/ethereum/wiki/wiki/White-Paper
[9] http://tinyurl.com/zxgayfr

In the Bitcoin network, miners participate in the verification process by hash collision through PoW. When the hash value of a miner is able to calculate and meet a certain condition, the miner may claim to the network that a new block is mined:

$$Hash(BlockHeader) \leq \frac{M}{D}$$

For the amount of miners `M` and the mining difficulty `D`, the `Hash()` represents the SHA256 power with value range [0, M], and D. The SHA256 algorithm used by Bitcoin enables every node to verify each block quickly, if the number of miners is high versus the mining difficulty.

The 80 byte BlockHeader varies with each different Nonce. The overall difficulty level of mining adjusts dynamically according to the total hash power of the blockchain network. When two or more miners solve a block at the same time, a small fork happens in the network. This is the point where the blockchain needs to make a decision as to which block it should accept, or reject. In the Bitcoin network, the chain is legitimate that has the most proven work attached.

Most PoS blockchains can source their heritage back to PeerCoin[10] that is based on an earlier version of Bitcoin Core. There are different PoW algorithms such as Scrypt[11], X11[12], Groestl[13], Equihash [4], etc. The purpose of launching a new algorithm is to prevent the accumulation of computing power by one entity and ensure that Application Specific Integrated Circuits (ASIC) can not be introduced into the economy. Qtum Core chooses PoS based on the latest Bitcoin source code for basic consensus formation.

In a traditional PoS transaction, the generation of a new block must meet the following condition:

$$ProofHash < coins \times age \times target$$

In `ProofHash`, the stake modifier [40] computes together with unspent outputs and the current time. With this method, one malicious attacker can start a double-spending attack by accumulating large amounts of coin age. Another problem caused by coin age is that nodes are online intermittently after rewarding instead of being continuously online. Therefore, in the improved version of PoS agreement, coin age removal encourages more nodes to be online simultaneously.

The original PoS implementation suffers from several security issues due to possible coin age attacks, and other types of attacks [16]. Qtum agrees with the security analysis of the Blackcoin team [40] and adopts PoS 3.0 [14] into the latest Qtum Core. PoS 3.0 theoretically rewards investors that *stake* their coins longer, while giving no incentive to coin holders who leave their wallets offline.

[10] https://peercoin.net/
[11] https://litecoin.info/Scrypt
[12] http://cryptorials.io/glossary/x11/
[13] http://www.groestlcoin.org/about-groestlcoin/
[14] http://blackcoin.co/

2.3 Qtum Contract and EVM Integration

The EVM is stack-based with a 256-bit machine word. Smart contracts that run on Ethereum use this virtual machine for their execution. The EVM is designed for the blockchain of Ethereum and thus, assumes that all value transfer use an account-based method. Qtum is based on the blockchain design of Bitcoin and uses the UTXO-based model. Thus, Qtum has an account abstraction layer that translates the UTXO-based model to an account-based interface for the EVM. Note that an abstraction layer in computing is instrumental for hiding the implementation details of particular functionality to establish a separation of concerns for facilitating interoperability and platform independence.

EVM Integration: All transactions in Qtum use the Bitcoin Scripting Language, just like Bitcoin. In Qtum however, there exist three new opcodes.

- `OP_EXEC`: This opcode triggers special processing of a transaction (explained below) and executes specific input EVM bytecode.
- `OP_EXEC_ASSIGN`: This opcode also triggers special processing such as `OP_EXEC`. This opcode has as input a contract address and data for the contract. Next follows the execution of contract bytecode while passing in the given data (given as `CALLERDATA` in EVM). This opcode optionally transfers money to a smart contract.
- `OP_TXHASH`: This opcode is used to reconcile an odd part of the accounting abstraction layer and pushes the transaction ID hash of a currently executed transaction.

Traditionally, scripts are only executed when attempting to spend an output. For example, while the script is on the blockchain, with a standard public key hash transaction, no validation or execution takes place. Execution and validation does not happen until a transaction input references the output. At this point, the transaction is only valid if the input script (`ScriptSig`) does provide valid data to the output script that causes the latter to return non-zero.

Qtum however, must accommodate smart contracts that execute immediately when merged into the blockchain. As depicted in Figure 1 , Qtum achieves this by the special processing of transaction output scripts (`ScriptPubKey`) that contain either `OP_EXEC`, or `OP_EXEC_ASSIGN`. When one of these opcodes is detected in a script, it is executed by all nodes of the network after the transaction is placed into a block. In this mode, the actual Bitcoin Script Language serves less as a scripting language and instead carries data to the EVM. The latter changes state within its own state database, upon execution by either of the opcodes, similar to a Ethereum contract.

For easy use of Qtum smart contracts, we have to authenticate the data sent to a smart contract as well as its creator stemming from a particular `pubkeyhash` address. In order to prevent the UTXO set of the Qtum blockchain from becoming too large, `OP_EXEC` and `OP_EXEC_ASSIGN` transaction outputs are also spendable. `OP_EXEC_ASSIGN` outputs are spent by contracts when their code sends money to another contract, or to a `pubkeyhash` address. `OP_EXEC` outputs

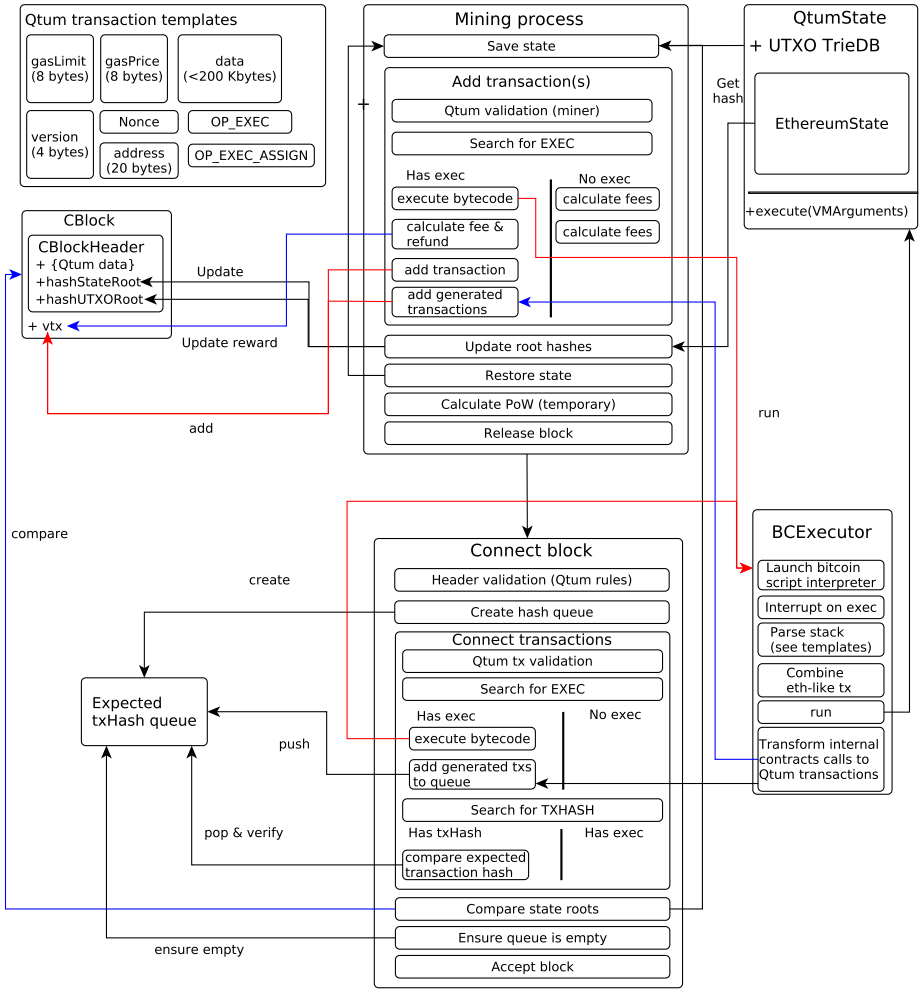

Fig. 1. Qtum transaction processing.

are spent whenever the contract uses the `suicide` operation to remove itself from the blockchain.

Qtum Account Abstraction Layer The EVM is designed to function on an account-based blockchain. Qtum however, being based on bitcoin, uses a UTXO-based blockchain and contains an Account Abstraction Layer (AAL) that allows the EVM to function on the Qtum blockchain without significant modifications to the virtual machine and existing Ethereum contracts.

The EVM account model is simple to use for smart-contract programmers. Operations exist that check the balance of the current contract and other contracts on the blockchain, and there are operations for sending money (attached

to data) to other contracts. Although these actions seem fairly basic and minimalistic, they are not trivial to apply within the UTXO-based Qtum blockchain. Thus, the AAL implementation of these operations may be more complex than expected.

A Qtum-blockchain deployed smart contract is assigned and callable by its address and comprises a newly deployed contract balance set to zero. There is currently no protocol in Qtum that allows a contract to be deployed with a non-zero balance. In order to send funds to a contract, a transaction uses the `OP_EXEC_ASSIGN` opcode.

The example output script below sends money to a contract:

```
1; the version of the VM
10000; gas limit for the transaction
100; gas price in Qtum satoshis
0xF012; data to send the contract
(usually using the Solidity ABI)
0x1452b22265803b201ac1f8bb25840cb70afe3303;
ripemd-160 hash of contract txid
OP_EXEC_ASSIGN
```

The simple script above hands over transaction processing to the `OP_EXEC_ASSIGN` opcode. Assuming no `out-of-gas`, or other exceptions occur, the value amount given to the contract is `OutputValue`. The exact details of the `gas` mechanism we discuss below. By adding this output to the blockchain, the output enters the domain of the contract owned UTXO set. This output value is reflected in the balance of the contract as the sum of spendable outputs.

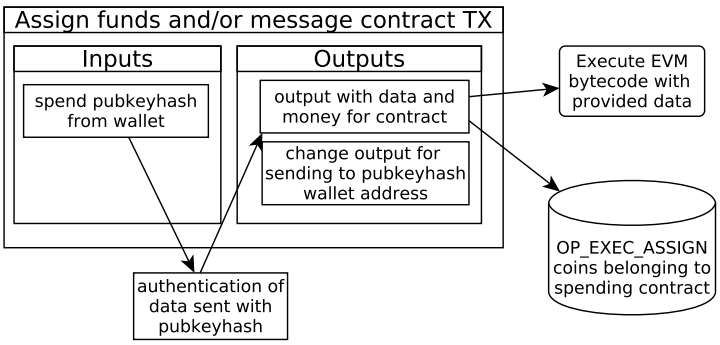

Fig. 2. Assign Funds and/or a message contract TX.

Although Figure 2 shows sending funds to a contract from a standard public key hash output, the method for sending money from one contract to another is nearly identical. When the contract sends funds to another contract or public key hash address, the former spends one of its owned outputs. The sending

contract involves Expected Contract Transactions for the fund sending. These transactions are special in that they must exist in a block to be valid for the Qtum network. Expected Contract Transactions are generated by miners while verifying and executing transactions, rather than being generated by consumers. As such, they are not broadcast on the P2P network.

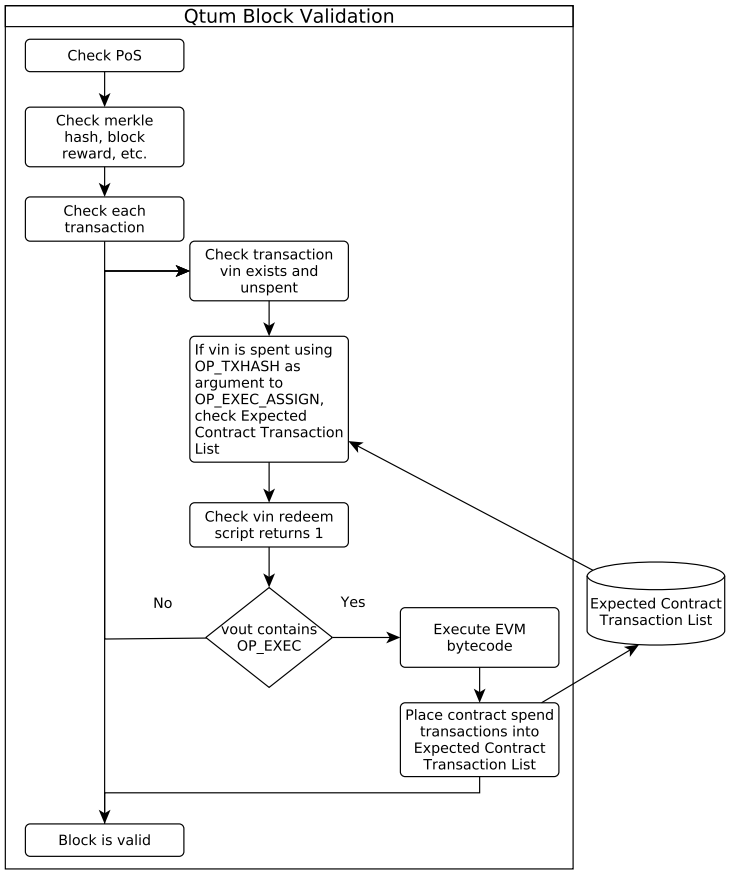

Fig. 3. Qtum block validation showing the Expected Contract Transaction List.

The primary mechanism to perform Expected Contract Transactions is the new opcode, `OP_TXHASH` that is part of Figure 3. Internally, both `OP_EXEC` and `OP_EXEC_ASSIGN` have two different modes. Upon their execution as part of the output script processing, the EVM is executed. When the opcodes are executed as part of input script processing, however, the EVM is not executed to avoid double execution. Instead, the `OP_EXEC` and `OP_EXEC_ASSIGN` opcodes behave similar to no-ops and return either 1 or 0, i.e., spendable or not spendable respectively, based on a given transaction hash. This is why `OP_TXHASH` is so

important to the functioning of this concept. Briefly, `OP_TXHASH` is a new opcode added which pushes the current spending transaction's SHA256 hash onto the Bitcoin Script stack. The `OP_EXEC` and `OP_EXEC_ASSIGN` opcodes check the Expected Contract Transaction List during a spend attempt.

After the transaction passes (usually from `OP_TXHASH`) to the opcodes that exist in the Expected Contract Transaction List, the result is 1, or spendable. Otherwise, the return is 0, or not spendable. In this way, `OP_EXEC` and `OP_EXEC_ASSIGN` using `vouts` are only spendable when a contract and thus, the Account Abstraction Layer, requires that the `vout` is spendable, i.e., while the contract attempts sending money. This results in a secure and sound way of allowing contract funds to be spent only by a respective contract in alignment with a normal UTXO transaction.

A specific scenario occurs if a contract has more than one output that can be spent. Each node may pick different outputs and thus, use completely different transactions for spending `OP_EXEC_ASSIGN` transactions. This is resolved in Qtum by a consensus-critical coin picking algorithm. The latter is similar to the standard coin picking algorithm used within a user wallet. However, Qtum significantly simplifies the algorithm to avoid the risk of denial of service (DoS) attack vectors and to realize simple consensus rules. With this consensus-critical coin picking algorithm, there is now no possibility for other nodes to pick different coins to be spent by a contract. Any miner/node who picks different outputs must fork away from the main Qtum network, and their blocks are rendered invalid.

When an EVM contract in Figure 4 sends money either to a `pubkeyhash` address, or to another contract, this event constructs a new transaction. The consensus-critical coin-picking algorithm chooses the best owned outputs of the contract pool. These outputs are spent as inputs with the input script (`ScriptSig`) comprising a single `OP_TXHASH` opcode. The outputs are thus, the destination for the funds, and a change output (if required) to send the remaining funds of the transaction back to the contract. This transaction hash is added to the Expected Contract Transaction List and then the transaction itself is added to the block immediately after the contract execution transaction. Once this constructed transaction is validated and executed, a confirmation check of the Expected Contract Transaction List follows. Next, this transaction hash is removed from the Expected Contract Transaction List. Using this model, it is impossible to spoof transactions for spending them by providing a hardcoded hash as input script, instead of using `OP_TXHASH`.

The above described abstraction layer renders the EVM contracts oblivious to coin picking and specific outputs. Instead, the EVM contracts know only that they and other contracts have a balance so that money can be sent to these contracts as well as outside of the contract system to `pubkeyhash` addresses. Consequently, contract compatibility between Qtum and Ethereum is strong and very few modifications are required to port an Ethereum contract to the Qtum blockchain.

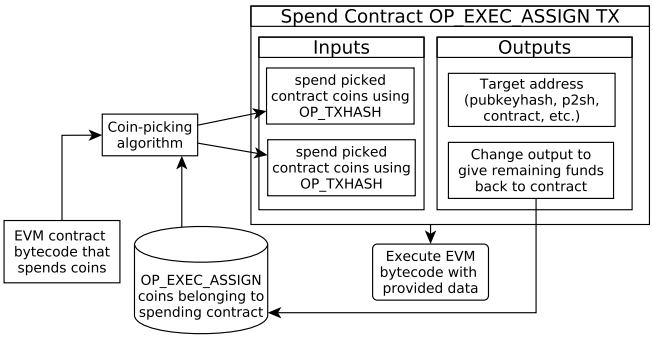

Fig. 4. Spend contract `OP_EXEC_ASSIGN` transaction.

Added Standard Transaction Types: The following are the standard transaction types that we add to Qtum. They are documented here as Bitcoin script templates: Deploying a new contract to the blockchain requires an output script as follows:

```
1; the version of the VM
[Gas limit]
[Gas price]
[Contract EVM bytecode]
OP_EXEC
```

Sending funds to an already deployed contract on the blockchain requires the script below:

```
1; the version of the VM
[Gas limit]
[Gas price]
[Data to send to the contract]
[rip-emd160 hash of contract transaction id]
OP_EXEC_ASSIGN
```

Note there are no standard transaction types for spending as that requires the Expected Contract Transaction List. Thus, these spending transactions are neither broadcast nor valid on the P2P network.

2.4 Gas Model

A problem Qtum faces with adding Turing-completeness to the Bitcoin blockchain is relying on only the size of a transaction, which is not reasonable for determining the fee paid to miners. The reason is that a transaction may infinitely loop and halt the entire blockchain for transaction-processing miners. As Figure 5 shows, the Qtum project adopts the concept of `gas` from Ethereum. In the `gas` concept, each EVM opcode executed has a price and each transaction has

an amount of `gas` to spend. Post-transaction remaining `gas` is refunded to the sender.

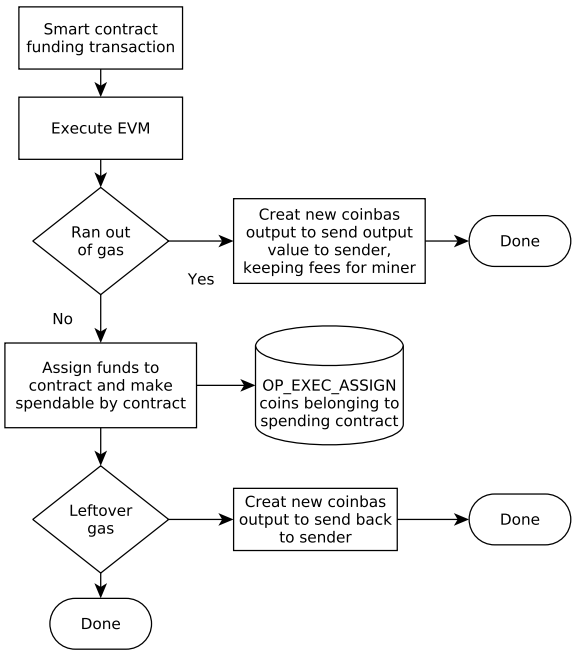

Fig. 5. Gas refund model.

When the `gas` required for contract execution exceeds the amount of `gas` available to a transaction, then the actions of a transaction and state changes are reverted. Thus, any modified permanent storage is reverted to its original state including any spending of contract funds so that the latter are not spent. Despite a reversion, all `gas` of a transaction is consumed and given to the processing miner since the computing resources have already been spent.

Although Qtum uses the `gas` model from Ethereum, we expect the `gas` schedule, i.e., the `gas` price of each EVM opcode, to significantly differ from Ethereum. The exact values are determined by comparing existing prices in Ethereum with the amount of processing and blockchain resources required for each opcode to Qtum.

When creating a contract funding-, or deployment transaction, the user specifies two specific items for `gas`. The `GasLimit` determines the amount of consumable `gas` by a contract execution. The second item is the `GasPrice` to set the exact price of each unit of `gas` in Qtum Satoshis. The latter are currently a smaller unit of the Bitcoin currency that the blockchain records. The maximum Qtum expenditure of a contract execution equates the multiplication of `GasLimit` by `GasPrice`. If this maximum expenditure exceeds the transaction fee provided by

the transaction then the latter is invalid and can not be mined, or processed. The remaining transaction fee after subtracting this maximum expenditure is the Transaction Size Fee and analogous to the standard Bitcoin fee model.

To determine the appropriate priority of a transaction, miners consider two variables. First, the transaction size fee must match the total size of a transaction, i.e., usually determined by a minimum amount of coins per kilobyte formula. The second variable is the `GasPrice` of a contract execution. In combination, PoS miners choose the most important and profitable transactions to process and include in a block. Consequently, there exists a free-market fee model with miners and users optimizing for the best fee that suits their transaction speed and the price they are willing to pay.

Refunds: Using the UTXO model, funds sent to miners as transaction fees are non-negotiable. It is impossible for a miner to partially refund a fee if the transaction is easier for the miner to process than expected. Still, for the `gas`-model to be useful, a method must exist to refund funds back to the sender. Moreover, it must be possible to roll back the state of a transaction that runs out of `gas` and return `gas`-fees to miners.

Refunding `gas`-fees in Qtum is enabled by creating new outputs as part of the coinbase transaction of a miner. We add a new block validation consensus rule to ensure refunding outputs are required to exist in the coinbase transaction. Otherwise, miners may choose to not refund `gas`. The refund is given back to the sender of a transaction fund by copying the output script. For security reasons, this script is currently a standard `pay-to-pubkeyhash`, or `pay-to-scripthash` script. We plan to lift the restriction after further security studies.

For reference, the `OP_EXEC_ASSIGN` has the following format to assign contract funds:
Inputs: (in push order)

- Transaction hash for spending [optional]
- version number (VM version to use, currently just 1)
- `gas` limit (maximum amount of `gas` that can be used by this exec)
- `gas` price (How much qtum each `gas` unit is)
- data (data to be passed to this smart contract)
- smart contract address

Outputs: (in pop order)

- Spendable (if the funds are currently spendable)

Consequently, we give an example `EXEC_ASSIGN` below:
1
10000
100
0xABCD1234...
3d655b14393b55a4dec8ba043bb286afa96af485
`EXEC_ASSIGN`

If the VM execution results in an `out-of-gas` exception, this `vout` is spent by the next transaction in the block using the redeem script `OP_TXHASH`.

The generated `vout` for this transaction is a `pubkeyhash` script taken from the `vin[0].prevout` script. In this early version of Qtum, only `pubkeyhash` senders are allowed for VM funding transactions. Although other forms can be accepted into blocks to result in VM execution, the `msg.sender` in the EVM is "0" and any `out-of-gas`, or `gas`-refund needed results in the contract keeping the funds.

Partial Refund Model: Pertaining to the `gas` model, it is also necessary to refund the unspent portion for several reasons. On the one hand, users can spend a large amount of funds to ensure their contract is executed properly. Still, the unused `gas` returns as a Qtum refund.

The return address for `gas` is expressed on the blockchain as a `vin[0].prevout` script of the sending transaction. Gas is sent to a contract by using the standard bitcoin transaction-fee mechanism. Thus, the new fee model slightly augments this to make the transaction fee:

```
gas_fee = gas_limit * gas_price
txfee = vin - vout
tx_relay_fee =txfee - gas_fee
refund = gas_fee - used_gas
```

A proposal exists for enabling miners to evaluate both the `tx_relay_fee` and the `gas_price` under a single "credit_price" value for determining transaction priority.

During contract execution, `gas` tokens are subtracted from the total fee, i.e., multiplying by `gas_price`. After completing the contract execution, the remainder of this `gas_fee` must be returned to the given `gas` return script by adding an output to the coinbase transaction the miners use to retrieve their block reward. The coinbase added `vout` is a `pubkeyhash` from `vin[0].prevout`. In order to receive a `gas` refund, this must be a spent `pubkeyhash vout`. Otherwise, the `gas` refund remains with the miner in an `out-of-gas` condition and the funds sent will remain with the contract.

Note that it is currently only possible to have one EVM contract execution per transaction. Thus, never does the case arise where two contract executions attempt to share the transaction fee. This scenario may be enabled after solving existing problems with multiple EVM executions per transaction. The current design support multiple contract executions per transaction.

Important GAS Edge Cases: Miners must be cautious with `contract-gas`, and `fund-return` scripts. If the latter script output causes a block to exceed the maximum size then the contract transaction can not be put into this block. Instead, the `gas-return` script execution must take place again in the next mined block. Miners must ensure sufficient capacity exists in the candidate block for the `gas-return` script before attempting to execute the contract. Not following this rule results in a contract requiring repeated execution, if the refund script

does not fit into the current block. If there are no `gas` funds to return, no `vout` requirement exists for returning the funds.

It is consensus critical that the transaction fee includes the `gas_fee`. A transaction is invalid when adding it to a block results in a negative `gas` refund, or when the `gas_fee` is lower than the transaction fee.

No transaction output script is valid that has more than one `OP_EXEC`, or `OP_EXEC_ASSIGN` opcode. While this limits scripting abilities, it is preferable to potential recursion- and multiple-execution problems. Consequently, static analysis suffices to determine if a script is invalid.

After very blockchain-oriented Qtum technicalities, we next describe conceptually the management of smart-contract lifecycles. Note that the conceptual presentation in the sequel is backed by scientific literature [12, 13, 24, 18, 26, 27, 32].

3 Smart-Contract Management

As stipulated above, we assume that lifecycle management is essential for securing smart contracts in that a proper vetting of potential collaborating parties takes place before enactment. We consider a real-life case from a failed seafood delivery[15] where a business-transaction conflict emerges from an underspecified conventional contract (CC). An EU company (buyer) orders 12 920 kg of cuttlefish from a South-Asian company (seller). In the CC, the liability of the product quality rests with the seller until the carrier obtains the goods. The underspecification pertains to the quality of the goods that is not specified in the CC and the buyer does not check the goods before transferral to the shipping company (carrier).

The smart-contract alternative resolves the underspecification conflict that exist in the CC. Thus, in Section 3.1 the presented Qtum-framework goal model reflects the properties of a smart-contract lifecycle that is fully formalized in [18, 26, 27, 32]. Next, Section 3.2 gives a small lifecycle example for the seafood-shipment case.

3.1 Lifecycle-Management Goals

For discussing goals, we use the following approach. The Agent-Oriented Modeling (AOM) method [38] is a socio-technical requirements-engineering approach that takes into account that humans who may belong to organizations use technology to collaborate for solving problems. In this section, we use the AOM goal-model type to capture important socio-technical behavioral features for the Qtum smart-contract system that supports the running case. Goal models enhance the communication between technical and non-technical stakeholders to increase the understanding of the problem domain. Note that AOM goal models are also instrumental [39] for novel agile software development techniques.

[15] http://cisgw3.law.pace.edu/cases/090324s4.html

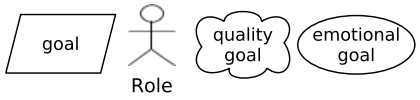

Fig. 6. Modeling elements for AOM goal models.

A goal model comprises three main elements depicted as in Figure 6. Functional requirements we refer to as goals and are depicted as parallelograms, roles we depict as sticky men, and non-functional requirements. The latter has two variants, namely quality goals for software-related non-functions requirements depicted as clouds, and human-related emotional goals depicted as ellipses. The goal model starts with a central root value proposition that is not atomic. Consequently, the value proposition is decomposed in a tree-hierarchy into sub-goals where each sub-goal represents an aspect for achieving its parent goal [21] and the lowest sub-goal must be atomic. Goals may have assigned roles, quality- and emotional goals that are inherited to lower-level goals.

Value Proposition of the Qtum Framework: The root of the goal for the Qtum-framework we depict in Figure 7 and it is the value proposition of cross-organizational information- and value-transfer logistics automation. We split the complex value proposition into goals for smart-contract lifecycle management [26, 27, 32], i.e., *setup*, *rollout*, *enactment*, *rollback*, *termination*. These refined goals we further explore in Section 3.2.

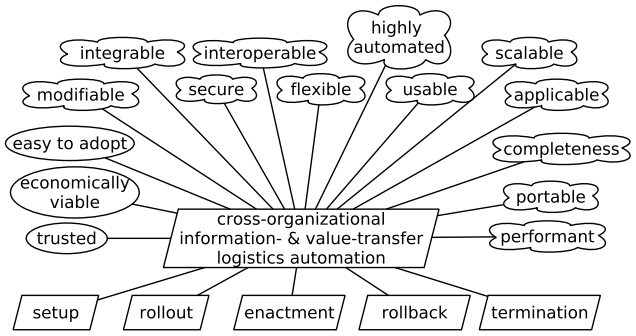

Fig. 7. Qtum value proposition with lifeycle-management refinement [26, 27, 18].

An essential emotional goal for industry adoption in Figure 7 is *trust* in the sociotechnical Qtum-system [34] to reliably perform the intended behavior. In this case, trust pertains to the dependencies among humans who use technology to achieve goals. We consider *economically viable* and *easy to adopt* as additional emotional goals that influence widespread industry diffusion. The former means

that using the Qtum-system results in economic return on investment, while the latter means the personal barrier of entry for working with Qtum is low.

There are quality goals affiliated with the value proposition that affect all refining parts of the Qtum-system. These quality goals we derive from a reference architecture [28] for cross-organizational business-process aware collaboration. The quality goals below are structured in accordance with [9, 17]. The following quality goals are not discernible during system execution time.

Modifiable means the Qtum-system changes and adapts during its lifecycle to the business context. Additionally, it harmonizes cross-organizationally heterogeneous system environments comprising regularly updating commercial software. *Integrable* systems consists of separately developed and integrated components for which the interface protocols between the components must match. Hence, integrability between the components of Qtum must be assured.

Next, we specify the quality goals for Qtum that are discernible during runtime. *Interoperable* means Qtum must interoperate at runtime with systems supporting business functions such as planning, logistics, production, external partner systems, and so on. Dynamic interoperability challenges are business-, conceptual-, and technical heterogeneity. *Secure* refers to resisting unauthorized attempts at usage and denial of service while providing services to trusted users with good reputation. To address security, trust- and reputation problems, several strategies are possible for Qtum. A blockchain-supported authentication service checks collaborating parties, monitors, inspects and logs network events. The communication of a system may be encrypted, and so on. *Highly automated* collaboration requires systems must cover the entire smart-contract lifecycle. Hence, Qtum must provide for possibilities of a high degree of meaningful collaboration automation that processes tedious and repetitive work while allowing humans to focus on the remaining creative action. *Flexible* collaboration is a highly dynamic process enacting activities by diverse partners exchanging heterogeneous data [25]. Hence, Qtum must enable diverse cross-organizational collaboration scenarios harmonizing heterogeneous concepts and technologies. *Usable* means Qtum must be easy to use for cross-organizational information-logistics automation and decomposes into three areas. Error avoidance must anticipate and prevent commonly occurring collaboration errors. Error handling is system support for a user to recover from errors. Learnability refers to the required learning time of users to master the Qtum-system.

Finally, there exist quality goals that are architecture specific. *Completeness* is the quality of Qtum comprising the set of components for smart-contract lifecycle management. *Scalable* refers to the ability of Qtum to combine more than two collaborating parties into one configuration. *Applicable* means that Qtum is instrumental for automating cross-organizational information logistics and value transfers. *Portable* means Qtum supports information logistics independent of the industrial domain and collaboration heterogeneity with respect to business-, conceptual-, and technological system infrastructure. Note this also includes mobile devices. *Performant* means the computational and communicational strain is low for information-logistics automation. Hence, it is important to ensure that

all phases of a smart-contract lifecycle are carried out within a desirable response time and without an exponential need for computing power.

3.2 Lifecycle Management Example

The goal model of Section 3.1 we map into Figure 8 for projecting in the running seafood case. The modeling notation in Figure 8 is the business process model and notation BPMN [22] and the complete lifecycle is formalized in [26, 27, 18]. The green circle denotes a start of the lifecycle and the red circle the lifecycle end. The rectangles with plus signs are so-called subprocesses that correspond to lifecycle stages in Section 3.1. A subprocess is a compound activity that hides lower-level business-process details.

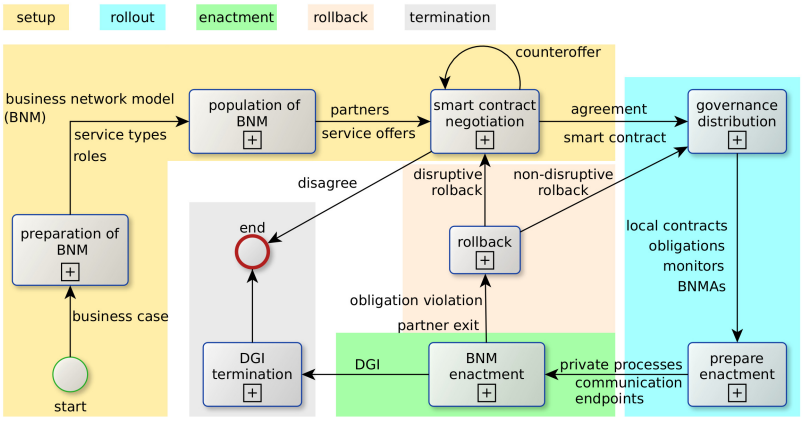

Fig. 8. Qtum smart-contract lifecycle management.

The starting point of each smart-contract lifecycle in Figure 8 is the business case of seafood transportation that requires cross-organizational information logistics automation. Assuming there exists a collaboration hub [29] that serves as a preparation platform for the inception of smart contracts, a designer creates a template for a business-network model (BNM) into which service types are inserted together with roles.

The BNM template enters the population phase. The roles affiliated to respective service types are filled with organizations that collaborate in the smart contract, i.e., *bank*2, *seller*, *fridge*1, *carrier*, *fridge*2, *buyer* and *bank*1. Note that it is possible for several candidate organizations to compete for filling a specific role. In order to reinforce the desire to fill a role, potential partner organizations must match a service offer into the service type a role is affiliated with. A service consumer can evaluate the proposal and decide if a service offer is acceptable.

When all roles are filled and service-types match with acceptable service offers, the smart-contract negotiation commences. We assume no party from the

running seafood-delivery case has a desire to disagree and bring the setup phase to a sudden end. Instead, the buyer provides a counter offer that introduces obligations pertaining to the temperature inside containers where the seafood is stored. We assume the shipment containers are equipped with Internet-of-Things (IoT) [15] sensors that inform the shipper, seller and buyer in realtime when a temperature-threshold violation occurs. The counteroffer of the *buyer* defines in this case that either a price reduction follows in accordance with the reduced quality of the seafood. If the temperature change results in the seafood not being fit for consumption any more, the *buyer* has the right to reject purchasing the shipment on arrival.

The counteroffer is accepted by all other parties and a consensus occurs, which is the prerequisite for a contract establishment. The smart contract is a coordinating agent from which a distributed governance infrastructure (DGI) must be deduced. Thus, each party of the running case receives a local contract copy from which a set of respective obligations are deduced. For example, an obligation for the *carrier* is that the temperature inside of a seafood-shipping container must never be above 20C. The obligations are observed by monitors and assigned business-network model agents (BNMA) that connect to IoT-sensors.

Next, all collaborating parties can assign their respective private processes [12] into an emerging (DGI). For example, we assume peer-to-peer payment by Bitcoins that the *buyer* first must purchase with Euros. That purchase and payment via *bank1* involves a process that comprises compliance- and reporting steps as government impose regulations on the use of crypto-currencies. For enabling information exchanges between *bank1* and *bank2* of the *seller*, the communication endpoints must be established. That way, the management of seller compliance data is automated.

Assuming a temperature-threshold obligation violation occurs in the domain of *fridge1*, an assigned BNMA escalates the event and the *buyer* checks the breach severity. If the temperature breach lasts for a period of time resulting in diminished seafood quality that still allows a successful sale for a lower price that the *buyer* tolerates, a response may be the latter requests further cooling by a different company that slips into the role *fridge1*. Assuming the seafood is severely spoiled and can not be sold in the target country, the *buyer* triggers a disruptive rollback that collapses the transaction. If the seafood shipment arrives with the buyer in the agreed upon state and the payment to the *seller* via *bank2* is complete, then the termination stage dissolves the DGI and releases all collaborating parties.

We next give the relationships between detailed collaboration elements that the lifecycle management of Figure 8 coordinates.

4 Value-Transfer Protocol

An integral part of the Qtum-framework is the notion of a value-transfer protocol (VTP) that orchestrates cross-organizational information logistics and value

transfers, in line with the value proposition Figure 7 depicts. Consequently, Section 4.1 describes the relationship of process types that form a VTP. Section 4.2 discusses the need for a specific smart-contract language with the utility to specify VTPs. Finally, Section 4.3 discusses the features of a VTP-supporting language versus Solidity that Ethereum uses.

4.1 Cross-Organizational Processes

The VTP comprises three different types of collaborating processes. Figure 9 shows a simplified BNM in BPMN notation for the seafood delivery that Section 3 introduces. The BNM assumes that a sequence of subprocesses are placeholders for service types [12, 13] with labels that indicate the roles of organizations.

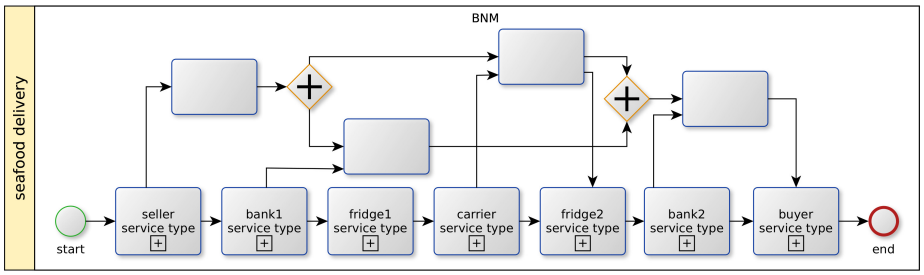

Fig. 9. Qtum BNM.

We assume that the BNM also comprises tasks connecting service-type subprocesses for establishing choreography control flow. For simplicity, Figure 9 depicts unlabeled choreographing tasks together with an AND-split and -join. The BNM commences with the seafood seller informing the bank to prepare for an international currency transaction and next, the seafood is cooled before a carrier performs shipping to the destination. In the destination country, the seafood is cooled again while a local bank processes the currency transaction between both countries. Finally, the buyer receives the seafood for local sales.

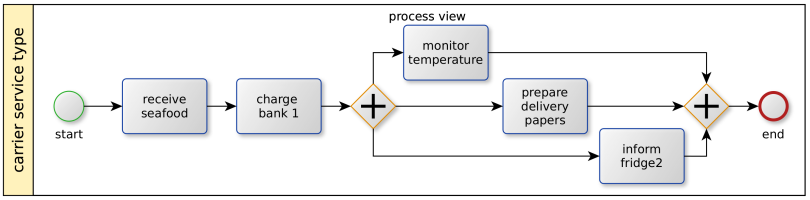

Fig. 10. Externalized service-type process view.

For the carrier subprocess of the BNM, the assumption is that several candidate organizations exist for filling the role of seafood carrier. Figure 10 depicts a

simplified example for a lower-level refinement in the form of a service-type process view [12, 13]. The simplified process in Figure 10 assumes a carrier receives the seafood from the fridge in the source country and charges the bank of the seller. Next, three parallel branches require that temperature monitoring, preparing delivery papers and informing the cooling company in the target company take place simultaneously. Only a candidate organization can become a service-providing carrier that promises adhering to this simplified process. Note that a collaboration hub [30] can offer the service-type process views for matching the latter with corresponding service-offering organizations.

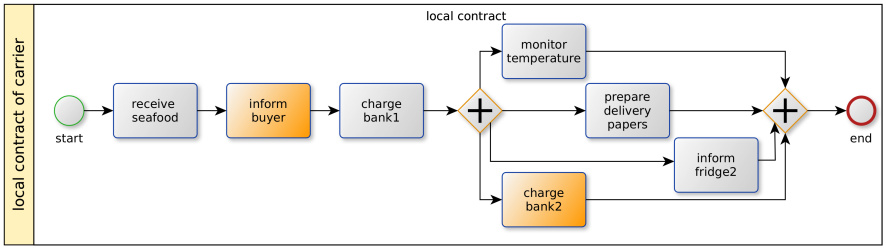

Fig. 11. Local carrier contract.

As a third VTP-element, Figure 11 shows the local contract the carrier uses internally. Note that differently to the service-type process view in Figure 10, the local contract comprises two additional tasks with the labels `inform buyer` and `charge bank2`. Thus, the local contract is a subclass of the service-type process view with respect to enactment behavior [12, 13], i.e., all tasks of the process view are experienced externally while the carrier has the option to insert hidden additional steps in a privacy-assuring way that constitute a competitive advantage, or are not of interest for external display, and so on.

4.2 Qtum Smart-Contract Language

To support the VTP scenario of Section 4.1, the current smart-contract lingua franca Solidity does not have the required utility level with respect to contained concepts and properties. Instead, it is the objective to develop a Qtum smart contract language (QSCL) and compiler that has comparatively better utility for VTP management.

High-level QSCL concepts and properties, Figure 12 depicts. The VTP scenario of Section 4.1 resembles the eSourcing framework for which a dedicated language exists, the eSourcing Markup Language (eSML) [31] that is currently specified for the semantic-web domain. We intend to map the concepts and properties of eSML into the blockchain domain for creating QSCL together with a language compiler for a novel Qtum virtual machine.

Briefly, while we refer the reader to [31] for further details, the properties in Figure 12 we organize along conceptual interrogatives. One QSCL-instance

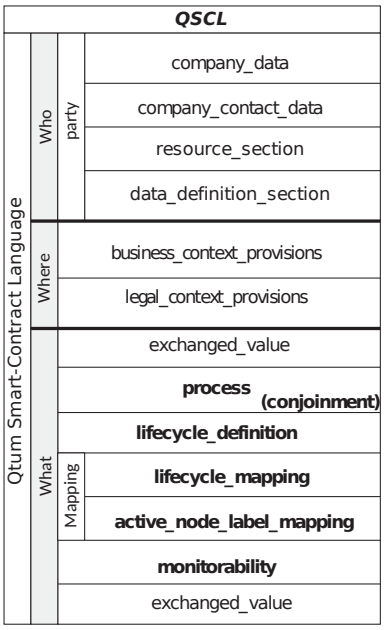

Fig. 12. Properties and concepts of the future Qtum smart contract language [31].

resembles a BNM definition (Figure 9). The `Who` concept of QSCL comprises constructs for defining the contracting parties uniquely together with involved resources and data definitions. The `Where` concept specifies business-context- and also legal-context provisions in which a specific smart contract holds. The `What` concept allows for defining exchanged values and the service-type process views (Figure 10) together with lifecycle definitions for such process views and also for elementary tasks respectively. Thus, in the `What` part of a QSCL instance, several service-type process views can be defined comparable to Figure 9. Finally, `conjoinment` constructs are specifically defined exchange channels for cross-organizational data flow. `Monitorability` constructs allow for a flexible definition of dedicated task-monitoring that either uses a polling-, or a messaging principle.

4.3 Comparative Discussion

Using the smart-contract ontology [31], we examine informally the suitability of existing Solidity versus QSCL that we construct for the Qtum-framework. As a general observation, Solidity is a language with a focus on mostly low-level blockchain-manipulation commands with JavaScript-resembling syntax. Still, it is possible to import third-party APIs and perform external function calls. So-called external functions in Solidity are part of a smart-contract interface that can be called from other contracts and via transactions.

Because of the Turing-completeness of Solidity, it is in principle possible to define cumbersome supports for all concepts and properties of the smart-

contract ontology that QSCL embodies. However, concepts such as pattern-based design, process awareness, matching of processes, etc., are not adopted in any way in Solidity. With respect to inventing cumbersome workarounds, a recent conference-paper publication [43] uses Solidity to demonstrate the feasibility of untrusted business-process monitoring and execution in smart contracts.

One must stress that Solidity has historically not been backed by formal verification means, differently to the design inception of QSCL [31]. Without such formally verifiable expressiveness, it is not possible to know ahead of enactment if a contract is correct and free of security issues. A Solidity-related security incident[16] has triggered only very recently the development and application of verification tools such as Why[17], Solidifier[18], or Casper[19] that likely leads to a shift from proof-of-work towards proof-of-stake for Ethereum all together.

5 Conclusions

This whitepaper presents the Qtum-framework for a novel smart-contract blockchain-technology solution. We show the specific Qtum transaction-processing implementation that uses proof-of-stake validation. Furthermore, Qtum integrates the Ethereum virtual machine (EVM) together with the Bitcoin unspent transaction output protocol. Note that the Qtum EVM constantly remains backwards compatible. Additionally, the Qtum-framework recognizes that smart-contract lifecycle management is important for supporting proper security vetting of collaborating parties. To support Qtum lifecycle management, the current lingua franca Solidity lacks suitability. Consequently, the emerging Qtum-framework requires a novel smart-contract language with enhanced utility.

The adoption of proof-of-stake into Qtum constitutes a considerable saving of computational effort over the not scaling Ethereum alternative that still uses proof-of-work. While Ethereum plans to also adopt proof-of-stake, it is unclear when such a new version will be released. Also the use of unspent transaction outputs is more scalable in comparison to the account management of Ethereum. In combination with simple payment verification, Qtum already develops a smart-contract mobile-device solution. While the not scaling Ethereum solution does not allow for mobile solutions, Qtum aims to achieve a democratized and highly distributed proof-of-stake transaction validation with its mobile strategy.

The Qtum-framework has a clear understanding of quality criteria that future developments must satisfy. With respect to functional requirements, Qtum plans to develop an application layer for smart-contract lifecycle management. Most importantly, such lifecycle management is important for vetting collaborating parties to reduce security breaches such as those Ethereum recently experienced, resulting in multiple hardforks of the latter.

[16] https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/
[17] http://why3.lri.fr/
[18] https://hack.ether.camp/idea/solidifier—formal-verification-of-solidity-programs
[19] http://www.coindesk.com/ethereum-casper-proof-stake-rewrite-rules-blockchain/

The value-transfer protocol for information logistics in Qtum comprises a business-network model for choreographing several collaborating organizations. The latter can provide services with local contracts that must match with the specified runtime behavior of service-type process views in the business-network model. With a multi-layered smart-contract management layer, collaborating parties protect the privacy of their business secrets that pose a competitive advantage by hiding extension steps in local contracts.

In summary, the Qtum-framework recognizes that smart contracts are sociotechnical artifacts that must also take into account essential quality requirements for achieving widespread user adoption. Ongoing real-life industry projects with Qtum applications result in continuous empirical requirements harvesting. The mobile strategy in support of highly distributed proof-of-stake transaction processing aims at a significant advancement in the state of the art. Still, Qtum also recognizes that smart-contract lifecycle management requires application-layer development with sophisticated front-end user experience that current solutions do not pay attention to sufficiently.

References

1. A.M Antonopoulos. Mastering bitcoins, 2014.
2. I. Bentov, A. Gabizon, and A. Mizrahi. *Cryptocurrencies Without Proof of Work*, pages 142–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
3. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.
4. A. Biryukov and D. Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. *Proceedings of NDSSâĂŹ16, 21–24 February 2016, San Diego, CA, USA. ISBN 1-891562-41-X*, 2016.
5. B. Bisping, P.D. Brodmann, T. Jungnickel, C. Rickmann, H. Seidler, A. Stüber, A. Wilhelm-Weidner, K. Peters, and U. Nestmann. Mechanical verification of a constructive proof for flp. In *International Conference on Interactive Theorem Proving*, pages 107–122. Springer, 2016.
6. O. Bussmann. *The Future of Finance: FinTech, Tech Disruption, and Orchestrating Innovation*, pages 473–486. Springer International Publishing, Cham, 2017.
7. C. Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.
8. K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.
9. L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.
10. K. Croman, C. Decker, I. Eyal, A.E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer. *On Scaling Decentralized Blockchains*, pages 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
11. N. Emmadi and H. Narumanchi. Reinforcing immutability of permissioned blockchains with keyless signatures' infrastructure. In *Proceedings of the 18th In-*

ternational Conference on Distributed Computing and Networking, ICDCN '17, pages 46:1–46:6, New York, NY, USA, 2017. ACM.
12. R. Eshuis, A. Norta, O. Kopp, and E. Pitkanen. Service outsourcing with process views. *IEEE Transactions on Services Computing*, 99(PrePrints):1, 2013.
13. R. Eshuis, A. Norta, and R. Roulaux. Evolving process views. *Information and Software Technology*, 80:20 – 35, 2016.
14. D. Frey, M.X. Makkes, P.L. Roman, F. Taïani, and S. Voulgaris. Bringing secure bitcoin transactions to your smartphone. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, ARM 2016, pages 3:1–3:6, New York, NY, USA, 2016. ACM.
15. J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013.
16. A. Kiayias, I. Konstantinou, A. Russell, B. David, and R. Oliynykov. A provably secure proof-of-stake blockchain protocol, 2016.
17. G. Kotonya and I. Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.
18. L. Kutvonen, A. Norta, and S. Ruohomaa. Inter-enterprise business transaction management in open service ecosystems. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 31–40. IEEE, 2012.
19. L. Luu, D.H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, 2016.
20. L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.
21. J. Marshall. Agent-based modelling of emotional goals in digital media design projects. *International Journal of People-Oriented Programming (IJPOP)*, 3(1):44–59, 2014.
22. Business Process Model. Notation (bpmn) version 2.0. *Object Management Group specification*, 2011. `http://www.bpmn.org`.
23. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.
24. N.C. Narendra, A. Norta, M. Mahunnah, L. Ma, and F.M. Maggi. Sound conflict management and resolution for virtual-enterprise collaborations. *Service Oriented Computing and Applications*, 10(3):233–251, 2016.
25. A. Norta. *Exploring Dynamic Inter-Organizational Business Process Collaboration*. PhD thesis, Technology University Eindhoven, Department of Information Systems, 2007.
26. A. Norta. *Creation of Smart-Contracting Collaborations for Decentralized Autonomous Organizations*, pages 3–17. Springer International Publishing, Cham, 2015.
27. A. Norta. *Establishing Distributed Governance Infrastructures for Enacting Cross-Organization Collaborations*, pages 24–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
28. A. Norta, P. Grefen, and N.C Narendra. A reference architecture for managing dynamic inter-organizational business processes. *Data & Knowledge Engineering*, 91(0):52 – 89, 2014.

29. A. Norta and L. Kutvonen. A cloud hub for brokering business processes as a service: A "rendezvous" platform that supports semi-automated background checked partner discovery for cross-enterprise collaboration. In *SRII Global Conference (SRII), 2012 Annual*, pages 293–302, July 2012.
30. A. Norta and L. Kutvonen. A cloud hub for brokering business processes as a service: A "rendezvous" platform that supports semi-automated background checked partner discovery for cross-enterprise collaboration. *Annual SRII Global Conference*, 0:293–302, 2012.
31. A. Norta, L. Ma, Y. Duan, A. Rull, M. Kõlvart, and K. Taveter. eContractual choreography-language properties towards cross-organizational business collaboration. *Journal of Internet Services and Applications*, 6(1):1–23, 2015.
32. A. Norta, A. B. Othman, and K. Taveter. Conflict-resolution lifecycles for governed decentralized autonomous organization collaboration. In *Proceedings of the 2015 2Nd International Conference on Electronic Governance and Open Society: Challenges in Eurasia*, EGOSE '15, pages 244–257, New York, NY, USA, 2015. ACM.
33. Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. *Towards a Novel Privacy-Preserving Access Control Model Based on Blockchain Technology in IoT*, pages 523–533. Springer International Publishing, Cham, 2017.
34. E. Paja, A.K. Chopra, and P. Giorgini. Trust-based specification of sociotechnical systems. *Data & Knowledge Engineering*, 87:339 – 353, 2013.
35. J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2015.
36. M. Rosenfeld. Overview of colored coins. *White paper, bitcoil. co. il*, 2012.
37. P. Serguei. A probabilistic analysis of the nxt forging algorithm. *Ledger*, 1:69–83, 2016.
38. L. Sterling and K. Taveter. *The art of agent-oriented modeling.* MIT Press, 2009.
39. T. Tenso, A. Norta, and I. Vorontsova. Evaluating a novel agile requirements engineering method: A case study. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering - Volume 1: ENASE,*, pages 156–163, 2016.
40. P Vasin. BlackcoinâĂŹs proof-of-stake protocol v2, 2014.
41. M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.
42. M. Vukolić. *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*, pages 112–125. Springer International Publishing, Cham, 2016.
43. I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. *Untrusted Business Process Monitoring and Execution Using Blockchain*, pages 329–347. Springer International Publishing, Cham, 2016.
44. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.