Bus Inf Syst Eng
https://doi.org/10.1007/s12599-017-0507-z

RESEARCH PAPER

Automated Execution of Financial Contracts on Blockchains

Benjamin Egelund-Müller · Martin Elsman · Fritz Henglein · Omri Ross

Received: 10 January 2017 / Accepted: 16 August 2017

Abstract The paper investigates financial contract management on distributed ledgers and provides a working solution implemented on the Ethereum blockchain. The system is based on a domain-specific language for financial contracts that is capable of expressing complex multi-party derivatives and is conducive to automated execution. The authors propose an architecture for separating contractual terms from contract execution: a contract evaluator encapsulates the syntax and semantics of financial contracts without actively performing contractual actions; such actions are handled by user-definable contract managers that administer strategies for the execution of contracts. Hosting contracts and contract managers on a distributed ledger, side-by-side with digital assets, facilitates automated settlement of commitments without the need for an intermediary. The paper discusses how the proposed technology may change the way financial institutions, regulators, and individuals interact in a financial system based on distributed ledgers.

Keywords Blockchain · Domain specific language · Financial services · Distributed ledger

1 Introduction

The pillars on which the financial industry has been based for the last century are being challenged. The disruptive nature of new technologies such as modern machine learning and blockchain technology are changing the rules that form the financial sector and the financial system as a whole. Schneider et al. (2016) estimate savings from blockchain-based technologies to be in the region of tens of billion of US dollars annually across the financial sector with $11–12 billion in annual savings on the settlement of cash securities alone. In this paper, we demonstrate how a financial contract management system built upon a generalized distributed ledger can automate the execution of contracts, including clearing and settlement, thus potentially inducing drastic changes in the financial industry.

Essentially, a *distributed ledger* or *blockchain*[1] offers participants the opportunity to establish distributed consensus on a set of shared facts without assuming mutual trust. It does so by implementing a single coherent logbook of events shared amongst a set of non-trusting participants, which acts as a *single point of truth*. Critically, no privileged parties are required to maintain the ledger.

In its basic form, a distributed ledger provides a *fixed* protocol for adding new events to a log of events. In Bitcoin (Nakamoto 2009) the built-in protocol ensures that a Bitcoin transfer can only occur from an authenticated owner, whose transaction history sums to a positive balance, where the amount transferred is at most that balance and has not already been spent. Bitcoin thus enforces a specific *contract* amongst an open-ended number of

Accepted after two revisions by the editors of the special issue.

B. Egelund-Müller · M. Elsman · F. Henglein · O. Ross (✉)
Department of Computer Science, University of Copenhagen,
Universitetsparken 5, 2100 Copenhagen, Denmark
e-mail: omriross@gmail.com

[1] The term *blockchain* arises from the technique of sequencing blocks of atomic payments between pseudonymous participants into tamper-resistant verified (implicit) asset balances, which underlies Bitcoin, an unstructured peer-to-peer system with its own virtual currency. We use the term more generally here for peer-to-peer systems without central control, but varying performance, privacy and authentication mechanisms, and for the applications conceived for and made possible by such technology.

Published online: 27 November 2017

participants. It has no mechanism for *user-definable* contracts, though. We call such a system a *level*-1 distributed ledger.

While financial contracts are conventionally written in natural language, in recent years the financial industry has moved towards expressing financial agreements using formal *contract languages*, which serve as precise notations for expressing financial agreements among parties; see Arnold et al. (1995), Jones et al. (2000), Jones and Eber (2003), Andersen et al. (2006), Henglein et al. (2009), Frankau et al. (2009), Hvitved (2010), Hvitved et al. (2011), Hvitved et al. (2012), Andersen et al. (2014), Schuldenzucker (2014), and Bahr et al. (2015). These domain-specific languages (DSLs) have a clearly defined syntax and semantics, which rigorously specify valid contracts[2] and their proper execution. Unlike natural legal language, such formal languages are constructed for precision and automatic processing.

Integrating a contract language and its semantics into a distributed ledger has the primary advantage of extending the notion of single, verifiable truth to include not just *ex-post* events, but also possible *ex-ante* events contractually expected to happen. In particular, it becomes possible to objectively and indisputably monitor whether some party is violating the protocol established by a contract. We call such an integrated system a *level*-2 distributed ledger system.

In general, a contract obliges or permits its parties to perform certain actions, but usually neither fixes all their details nor does it perform the actions itself. For example, a lease leaves both landlord and tenant with the monthly option to give notice or not and it provides flexibility as to when exactly to pay the rent; the lease itself certainly does not perform the rental payments.

Simple financial contracts may not provide much leeway for individual execution strategies; nonetheless, it is important to distinguish between the terms of a contract, specific strategies for executing it (of which there may be infinitely many, considering both possible actions and their timing), and their automated execution. The parties may have their own *contract strategies* for *what* exactly to do and *when* to do it; these must *comply* with the given contract, but are not part of it.

A *level*-3 distributed ledger system is a level-2 system that, additionally, supports user-definable automated *contract managers*, which effect, clear and settle contractually required transfers *automatically*. In other words, a contract manager carries out a contract strategy, where the signatories of a contract jointly authorize a contract manager to perform actions on their behalf that are guaranteed to satisfy the terms of the contract. The contract manager is fully specified by source code authoritatively executed by the distributed ledger, which thus guarantees not only consensus on which past events have happened and which contract has been entered into by whom, but also how and when all parties' obligations will be performed.

As we shall see, level-3 systems are realisable by generalized distributed ledger systems with

- a programming language with clear semantics for implementing (replicated) state machines whose execution state is kept on the ledger; and
- digital assets that reside on the ledger (that is, their ownership is effectively determined by the ledger state) and that can be managed by properly authorized programs executing on the ledger. Any real-world asset can be recorded on a ledger in a legally binding fashion if supported by suitable legislation with attendant real-world enforcement mechanisms.[3]

Our main contribution is an implemented framework capable of hosting level-2 and level-3 managed contracts. We have adapted the contract language of Bahr et al. (2015), which is capable of expressing a large number of over-the-counter (OTC) financial contracts, and implemented an execution engine for it on top of Ethereum, a modern distributed ledger system (Wood 2016).[4] The architecture of the implementation separates the *contract evaluator*, which defines the semantics of the contract language, from *contract managers*, which evaluate contracts using the contract evaluator and execute their obligations in accordance with a contract strategy. Contract managers also link the abstract names in the contracts they manage with actual parties, who are uniquely identified by a public key, and with *feeds*, which establish links to *observable values* in the real world. Such observable values include, for instance, end-of-day quotes for stocks published by trusted feed providers.

Ethereum is able to execute level-3 ledger applications through so-called *smart contracts*, (distributed) state machines that are typically specified in Solidity, a class-

[2] The term *contract* is polysemic within the context of this paper, as it ranges over (1) *legal contracts*, natural language descriptions of legally binding rights and obligations, including financial contracts, (2) *formal contracts*, rights and obligations described in formal syntax with an unambiguous semantics facilitating mathematical reasoning, which are not necessarily legally binding, (3) *smart contracts*, programs irrevocably executed on a distributed ledger.

[3] This is rather common already and thus hardly controversial: ownership of real estate, shares, bonds, and (bank account) money is in many countries legally determined by the state of various (nondistributed) ledgers. In Denmark, for example, the legally *definitive* ledgers for these assets are in *electronic* form rather than paper form.

[4] The source code of our implementation, including technical documentation, is available at https://github.com/bem7/ledger-contracts/.

based object-oriented programming language. A key innovation of our work is the way in which contracts and contract managers are expressed and executed on Ethereum. We implement contract managers as Solidity *smart contracts* for registering and executing contracts. As smart contracts are passive (i.e., no code is executed on the ledger unless an authorized end-user triggers it), contract managers expose a single method for executing a registered contract, which any user who has an interest in contractual progress (e.g., to receive transfers) may call. Since smart contract execution is replicated on all the nodes that make up the network, Ethereum requires users to provide *gas* paid for in *Ether*, the native currency in Ethereum, to execute smart contracts. Given our model for executing contracts, this cost suitably falls on the party that triggers execution.

Moreover, in Ethereum, any assets canonically available on the ledger may be held by a smart contract itself, which, as we shall see, opens up for contract managers that *automatically* can handle a range of possible coordination mechanisms, including short-time escrow insurance (e.g., for cross-currency swaps) and margin accounts (e.g., for traditional options).

The result is a significantly disintermediated financial system. With assets recorded together with automated logic for entering and executing contracts, a distributed ledger can become the fabric of an entire financial (sub)system with the potential of disintermediating contract parties by automating and eliminating third parties involved in reconciling, clearing, settling, archiving, and auditing. Figures 1 and 2 illustrate such a paradigmatic shift. With a distributed ledger, (some) intermediaries can be eliminated; many trades can be settled in real time; regulators gain real-time access to all relevant details of contracts and can perform systemic analyses and stress tests across multiple parties and industry sectors. This shift may, possibly dramatically, lower the costs of implementing European Market Infrastructure Regulation, European Capital Regulation and similar regulation such as the Dodd-Frank Wall Street Reform and Consumer Protection Act introduced recently to improve financial system transparency and robustness.

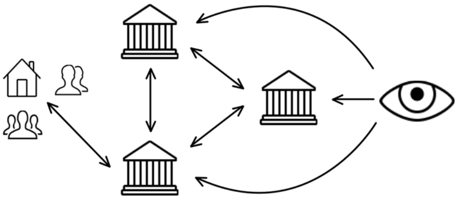


Fig. 1 Today's financial system is made up of a small group of large institutions that communicate bilaterally. Regulatory authorities ensure consistency of the system through audits of the institutions. Individuals, companies and smaller financial service providers access the financial system by partnering with a large institution

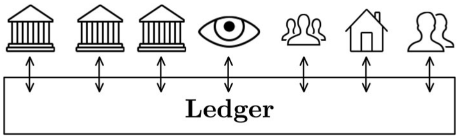


Fig. 2 Using a distributed ledger, the overhead of bilateral communication is eliminated as all parties enjoy direct access to the financial system. The ledger manages contracts and automatically settles them in accordance with participants' strategies for doing so. Access scales to an arbitrary number of participants as consensus protocols keep the ledger consistent

This paper considers primarily how to accurately describe and execute financial contracts within a distributed ledger. We acknowledge that the domain is also subject to a multitude of legal considerations, including how the judicial system, which is used to considerable degrees of interpretive leeway, assesses fully formalized contracts. While we comment on such considerations throughout this paper, an extensive analysis of legal implications is outside the scope of our present work. Section 6 discusses future work concerning legal considerations, and relates our results to work by Clack et al. (2016).

In Sect. 2, we describe in depth how the multi-party contract language of Bahr et al. (2015) can be adapted to work on distributed ledgers and how contract evaluation can be implemented on top of a modern distributed ledger system such as Ethereum. Moreover, in Sect. 3, we introduce the concept of contract managers and explore their applications. In particular, we demonstrate how contract managers link the abstract notions of transfers and observables to concrete assets and feeds on the Ethereum distributed ledger. Section 4 discusses the choice of appropriate ledger for a financial contract system. Section 5 discuss related work and Sect. 6 concludes the paper with an extensive research agenda.

2 The Contract Language

In this section, we present an informal and example-driven walkthrough of our language for expressing financial contracts and discuss the considerations required to implement it on a distributed ledger.

Figure 3 shows a simple 3-month USD-DKK cross-currency forward with notional $1000 and strike 7, written in our adapted version of the language designed and implemented by Bahr et al. (2015). Notice how the contract is composed of simpler contracts, joined by different constructors such as `translate`, `scale`, and `both`. The compositional approach facilitates a large variety of

```
translate(days(90),
  scale(1000,
    both(
      transfer(USD, X, Y),
      scale(7, transfer(DKK, Y, X))
)))
```

Fig. 3 FX forward contract

different contracts using just a handful of well-chosen constructors. The most basic constructors are those that are not compositional (i.e., that do not encapsulate another contract). Our language has two such constructors: `zero`, which denotes the empty contract, and `transfer`, which denotes the transfer of a single unit of a given currency from one party to another. In the example forward contract, the `translate` constructor is used to offset the enclosed contract three months into the future, the `scale` constructor is used to multiply transfers induced by a factor of 1000 and later by a factor of 7, and the `both` constructor specifies that both of its subcontracts must be executed.

Figure 4 illustrates *contract evaluation* or *contract reduction*, the evolution from a contract's original form via intermediate expressions to `zero`, representing its completion. It is specified by a small number of computer-interpretable rules that dictate with mathematical precision how any contract is evaluated. This eliminates ambiguities inherent in natural language and minimizes the potential for disputes amongst the contract parties.

In addition to the reduction semantics, the language also has a *type system* that enables parties to check – before commiting to an agreement – whether a contract is well-defined. Amongst other properties, the type system is able to verify that a contract is *causal*, meaning it contains no time absurdities. In the context of contracts on a ledger, the presence of a type system would make it impossible to create a contract where a constructor (such as `transfer`) is used incorrectly, or where a sequence of events could put the contract in a state that, e.g., requires a party to transfer an amount of funds today that depends on tomorrow's exchange rate. The ability to check these properties with mathematical certainty before signing a contract is a major benefit of using a formal language to express contracts.

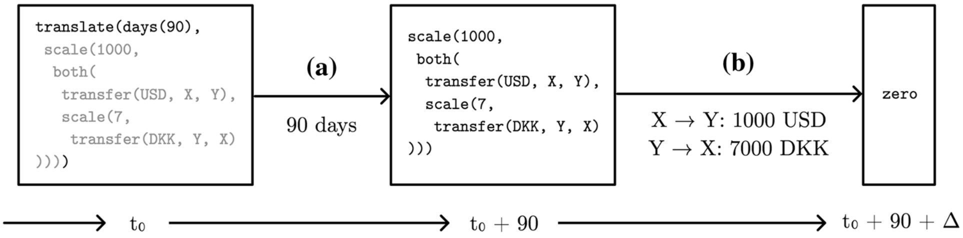
We consider the language of Bahr et al. (2015) well-suited for use in distributed ledgers. First, in addition to the reduction semantics and type system outlined above, the language has a natural cashflow semantics, which means that the result of evaluating a contract is a set of transfers that ought to take place, a semantics that fit well with a distributed ledger. Second, the language is multi-party and not biased toward any party specifically. This property contrasts with the language by Jones et al. (2000), which implicitly takes the viewpoint of the owner versus a single counterparty. As contracts recorded on a distributed ledger should look the same to all parties, this property is essential. Finally, the language is powerful enough to express a wide range of common contracts.

To use the language of Bahr et al. (2015) for representing and evaluating contracts on a distributed ledger, a number of adaptations had to be made. Figure 5 shows the full syntax of the adapted contract language. The adaptations are predominantly related to the handling of *feeds*. We will go into greater detail on the properties of feeds later, but for now, it suffices to say that we use the notion of feeds as sources of information that a contract operates over. Such information can be anything from decisions made by a contract's parties, to events relating to other contracts, to "real world" data such as the development in interest or exchange rates.

$$
\begin{array}{rll}
c ::= & \mathsf{zero} \mid \mathsf{scale}(e, c_1) \mid \mathsf{both}(c_1, c_2) \mid & \text{contracts} \\
& \mathsf{transfer}(a, p_1, p_2) \mid \mathsf{translate}(t, c_1) \mid & \\
& \mathbf{if}\ e\ \mathbf{within}\ t_1\ \mathbf{of}\ t_2 & \\
& \mathbf{then}\ \lambda x.\ c_1\ \mathbf{else}\ \lambda x.\ c_2 & \\
e ::= & x \mid i \mid b \mid \mathsf{obs}(f, a_1, \ldots, a_n, t) \mid & \text{expressions} \\
& op(e_1, \ldots, e_n) \mid \mathsf{foldt}(\lambda x.\ e_1, e_2, t) & \\
t ::= & \mathbf{now} \mid u(n) \mid t + t \mid t - t & \text{time} \\
u ::= & \mathbf{hours} \mid \mathbf{days} \mid \mathbf{months} \mid \ \ldots & \text{time unit} \\
a ::= & \mathbf{this} \mid i \mid l \mid p \mid a & \text{feed arg} \\
op ::= & + \mid - \mid \times \mid \mathsf{div} \mid = \mid \mathsf{if} \mid \ldots & \text{operators}
\end{array}
$$

where $n \in \mathbb{N}, i \in \mathbb{Z}, b \in \mathbb{B}, l \in \mathsf{Label}, p, q \in \mathsf{Party}$, $a \in \mathsf{Asset}, f \in \mathsf{Feed}, x \in \mathsf{Var}$

Fig. 5 Syntax of the contract language



```
translate(days(90),
 scale(1000,
  both(
   transfer(USD, X, Y),
   scale(7,
    transfer(DKK, Y, X)
))))
```

```
scale(1000,
 both(
  transfer(USD, X, Y),
  scale(7,
   transfer(DKK, Y, X)
)))
```

```
zero
```

Fig. 4 Example of how a contract evaluator gradually reduces the contract in Fig. 3 to `zero`. The contract takes effect at time t_0, the reduction (**a**) is applied at the first evaluation after the period of 90 days has passed, and the reduction (**b**) is applied after time Δ when the caller notifies the contract evaluator that the specified transfer has been settled succesfully

Specifically, the changes we have made include a multi-source paradigm for feeds, parameterization of the obs expression, and an amended model for the handling of time. The changes to the handling of observables reflect a paradigm where feed information is fetched directly from different sources within a ledger. The new model for handling time serves the dual purpose of enabling arbitrarily small units of time (limited only by the ledger's internal representation of time) as well as more flexible querying of feeds, which is necessary to bound time range within which a given observable is expected.

2.1 Examples

We now present a number of examples showcasing the kinds of contracts that can be written in the language. The first example is shown in Fig. 6 and represents an American option contract over a USD-DKK foreign exchange rate with a notional of $1000 and a strike of 7. This contract introduces the `if within`-contract constructor and the `obs` expression. In brief, `if within` evaluates to one contract if its expression becomes true at some point within the given period of time and another contract if not; `obs` is an expression that evaluates to an observable value at a given point in time. Both of these constructs are tied closely to the language's notion of time. The contract language uses a notion of relative time centered around the keyword `now`, which denotes different times depending on the scope in which it is used. For example, in `if ... within days(90) of now`, it refers to the start time of the contract, while in `obs(Decision, ..., now)` it iteratively refers to every time within the period spanned over by the surrounding `if within`-contract.

Our next example is the barrier option contract shown in Fig. 7. This contract extends the previous example with a restriction that the option can only be exercised if a feed of foreign exchange fixings reports a USD-DKK exchange rate above 7.5 within 90 days of the contract's start time. Notice the use of a nested `if within` constructor. The outer `if within`, which checks fixing rates against the barrier value, spans a 90-day period from the start time of the contract. The inner `if within`, which checks for a decision to exercise, should span the same 90 days period, thus limiting the period of time in which the option can be exercised. This is where the `\x` of the outer `if within` comes into play. Suppose the fixing rate goes above the barrier after 10 days. Then the value of `x` will be set to 10 days. With that information, the scope of the inner `if within`-contract can be limited to `within days(90) of -x`, that is, to within 90 days of 10 days ago.

```
if obs(Fixings, USDDKK, now) >= 7.5
within days(90) of now
then \x ->
 if obs(Decisions, X, exercises, this, now)
 within days(90) of -x
 then \y -> scale(
   1000,
   both(
     transfer(USD, Y, X),
     scale(7, transfer(DKK, X, Y))
   ))
 else \y -> zero
else \x -> zero
```

Fig. 7 FX up-and-in barrier option contract

Next, consider the credit default swap (CDS) contract shown in Fig. 8, which demonstrates the usefulness of the multi-party feature as well as the versatility of observables. In the contract, a party `Y` immediately owes $100 to another party `Z`, in return for which `Z` commits to paying $1000 to `Y` if a third party `X` defaults on the contract identified by the number `314`, managed by `ContractManager`, within 1 year of the start time of the CDS. This example depends upon the manager of the underlying contract acting as a feed of defaults. In a real-world example, the actual tracking of defaults would likely be handled in a more elaborate manner; yet this serves as a useful example of how observables enable us to construct contracts that depend on other contracts. We discuss future work on the handling of defaults in Sect. 6.

As a final example, the contract shown in Fig. 9 replaces an old contract with a new one. In this case, we apply the idea shown in the credit default swap example to have a new contract take effect at precisely the time another contract is terminated. The example demonstrates that contract managers need not expose functionality for updating or replacing contracts. By observing a contract manager and allowing a contract to be terminated, such functionality comes for free through the contract language.

```
if obs(Decisions, X, exercises, this, now)
within days(90) of now
then \x -> scale(
  1000,
  both(
    transfer(USD, X, Y),
    scale(7, transfer(DKK, Y, X))
  ))
else \x -> zero
```

Fig. 6 FX American option contract

```
both(
 scale(100, transfer(USD, Y, Z)),
 if obs(ContractManager, X, defaults, 314, now)
 within years(1) of now
 then \x -> scale(1000, transfer(USD, Z, Y))
 else \x -> zero
)
```

Fig. 8 Credit default swap contract

```
if obs(ContractManager, terminated, 314, now)
within weeks(1) of now
then \x -> ...
else \x -> zero
)
```

Fig. 9 Replacing a previous contract with a new one

2.2 Implementation

We use the term *contract evaluator* to describe the logic that allows for the creation, type checking, and evaluation of contracts. Our contract evaluator is implemented in the Solidity programming language as a so-called *smart contract* within the Ethereum ledger. A *smart contract* is a program that lives within, operates on, and whose execution is verified by the distributed ledger.[5] Despite *smart contract* being a misleading term for *state machine*, particularly within the context of financial contracts, we shall stick with it in the remainder of the paper because of its prevalence in contemporary blockchain discourse.

Distributed ledgers are passive. If no transactions are made, nothing happens on a ledger. This passiveness has the implication that contracts cannot be evaluated continuously, but only when someone explicitly triggers their evaluation. As a consequence there will be time gaps between a contract being evaluated. Hence, the implementation of contract evaluation must ensure that different evaluation times do not result in different contract evaluations. For example, evaluating an option contract on its maturity date as opposed to every day up to maturity should lead to exactly the same cashflows. We name this property the *consistency principle*, which is crucial to contract evaluation.

The greatest threat to the consistency principle are observables. The contract evaluator has no way of checking if an observable has been tampered with, for instance, by changing the timestamp of a decision to an earlier time. We address feeds in more depth shortly, but for now, it suffices to say that for the way we handle observables, contract parties should carefully audit feeds before settling on them for use in their contracts.

Given that contracts are not evaluated automatically, how do we ensure they will be evaluated at all? We reason that a party who stands to gain from a contract being evaluated will trigger its evaluation, and if no party stands to gain from a contract being evaluated, no one is any worse off if the contract is not evaluated. Notice that contract parties can always independently simulate the execution of a contract since the state of the ledger is known and the source code of the contract evaluator is public. Thus, parties can assess the results of contract evaluation *before* triggering it on the distributed ledger. Likewise, parties that need to evaluate periodically the state of their portfolio (due for instance to regulation) can do so without suffering the latency of transacting with the distributed ledger, or in the case of Ethereum, having to pay for the required *gas* (the cost of smart contract execution in Ethereum, as mentioned in Sect. 1).

The language of Bahr et al. (2015) operates with days as the smallest unit of time, meaning that all times within a day are equal – evaluating a contract multiple times in a day does not make sense. Conversely, Ethereum operates over seconds and other ledgers might well operate on even smaller units of time. We keep the discrete time of Bahr et al. (2015), but introduce the notion of a custom *time delta*. The time delta is the smallest unit of time within a contract and is part of a contract's metadata. A contract's time delta is not critical to the meaning of the contract, but has implications for the granularity of its evaluation and settlement, as well as how often observables can be queried. For example, a barrier contract that depends on live quotes requires a significantly lower time delta than one that only reads daily fixings.

3 Contract Managers

With a ready contract evaluator implementated within Ethereum, we now discuss automated execution of contracts. In Sect. 1 we introduced a taxonomy of *contracts* and *contract strategies*. The difference between the two is best illustrated by an example. Consider a vanilla American option such as the one shown in Fig. 6. The rules for how it may be evaluated are well-defined: The option can be exercised or not, and such a decision to exercise may be made at any time in the specified interval of time. Whether or when to exercise the option is *not* defined in the contract; rather, it is part of the *contract strategy* its owner applies. While a classical result by Merton (1973) concludes that American call options should never be exercised before maturity (the end of the time interval), a recent study by Jensen and Pedersen (2016) shows that many practitioners benefit from exercising such options before maturity. The multitude of stipulated ways and times of performing actions to fulfill a contract and specific strategies for performing actions employed by each contract party is the essential difference between contracts *per se* and contract strategies.

In our implementation, the contract language and its semantics are represented by a *contract evaluator* and contract strategies are encapsulated by *contract managers*. They are implemented in an object-oriented manner with

[5] A smart contract (and indeed the entire ledger) can be thought of as a *state machine* whose state and execution is replicated across a peer-to-peer network, supported by a suitable protocol to ensure observable consensus on the state.

the two being represented by different smart contracts interacting with each other.

A contract manager contains a reference to a contract evaluator, which can be analyzed, audited, and verified in isolation. The contract evaluator provides all functionality related to the contract language and its semantics, including the syntax of the language, which it exposes through functionality for constructing a contract's abstract syntax tree. It additionally provides functionality for statically checking that a contract is well-defined and causal.

A contract manager employs the contract evaluator to manipulate contracts in accordance with a particular, completely open contract strategy, to which all contract parties subscribe. At the practical level, this includes a number of bookkeeping tasks, such as storing contracts, handling the signing of contracts, allowing for contract cancellation or transfer, and fetching observable data from feeds. A contract manager encapsulates a particular contract strategy on top of this functionality. Contract managers may rely on human input to varying degrees: A simple manager might defer all actions to the parties involved, another might automatically take action in cases where a rational choice exists and only defer more complicated actions to the parties, while yet another manager might employ advanced models to take automatically all actions necessary for the successful settlement of a contract. As the logic governing a contract manager is openly available to everyone, users can in principle[6] audit its code and learn precisely in which cases actions will be based on human input and in which cases the manager will take action automatically.

We envision a future in which financial regulators publish a universal contract evaluator, encapsulating a safe domain-specific contract language capable of representing any financial contract with a mathematically certified contract semantics. Then the contract evaluator itself would be guaranteed to operate correctly, which would greatly reduce the risk of unintended behaviour. Compare this to a recent study by Luu et al. (2016), which found 8,833 out of 19,366 scanned smart contracts expressed in Ethereum's general-purpose native bytecode language to have at least one known vulnerability. Having a certified contract semantics would counter concerns of a vulnerable contract evaluator. This universal contract evaluator could be used in a wide range of contract managers that encapsulate different strategies. Actors in the financial system seeking to enter into a contract might develop a bespoke contract manager for their specific use case or choose between a wide range of off-the-shelf managers. Off-the-shelf contract managers could range from for-profit managers offered by financial service providers to crowd-based or even fully automated contract managers developed by the open-source community.

It is important to stress that contract managers should not be viewed as opaque, human-controlled intermediaries; they are completely specified mechanical computations and thus, in principle, completely and reliably predictable. As an example of a fully automated contract manager, consider a manager that implements the logic of a margin account to limit the risk of defaults. Upon signing a contract, the different parties deposit an initial margin with the manager. The manager continuously adjusts margin requirements for the different parties as the contract evolves. If a party is not able to maintain the margin, instead of defaulting, the manager simply transfers the funds to the counterparty and marks the contract as settled, or finds a new counterparty for the contract in the market.

A financial institution can launch a contract manager that largely takes action based on input from the institution itself. In this case, one could argue that the financial institution becomes an intermediary. While that may be the case, the contract managers themselves will remain transparent: Customers will be able to inspect their functionality and infer exactly what actions the financial institution may take.

A key component in the set up are feeds. A *feed* is a smart contract that obeys a standard protocol for the exchange of timestamped information. Due to the flexibility of smart contracts, these may be backed by a variety of mechanisms. A simple example is that of exchange rates published through a bespoke smart contract by a single entity (a so-called *oracle*), such as an exchange operator. Another example are crowd-based data feeds that use a mechanism such as SchellingCoin, as described by Buterin (2014), to provide information about the outside world based on a wisdom-of-crowds algorithm. Lastly, feeds may arise from other contract managers on the ledger that publish information about agreements being signed or parties defaulting on a contract. In our implementation, contract managers query feeds for information and pass this information along to the contract evaluator. We consider interaction with feeds, albeit largely standardized, an element of the contract strategy and therefore place responsibility for providing this functionality within contract managers rather than directly within the contract evaluator.

As mentioned previously, contract evaluation should follow the consistency principle, which states that the time and frequency of contracts being evaluated should not affect the cashflows induced. Since feeds are queried by contract managers during contract evaluation, the consistency principle could be violated by a corrupted feed that

[6] If coded in a Turing-complete programming language, contract manager code analysis ultimately incurs arbitrarily high computational cost, though.

uses false timestamps for its events. We do not have a solution for this problem and delegate responsibility for auditing feeds to the parties that agree to use them. As feeds will themselves be smart contracts, the logic governing them will be publicly visible, which should ease this process. But particularly for oracles, trust in the institutions that input real-world observations is required.

As contracts are evaluated, they induce cashflows. As we have already mentioned, a benefit arising from placing a full financial system on a shared ledger is that contract managers may access and transfer funds directly, without an intermediary. We use so-called *tokens*[7] to represent units of value on the ledger. Tokens can be any kind of fiat value, such as a currency, equity in a company, or ownership of a bond, but may also (through a trusted proxy) mirror commodities such as gold or Bitcoin. Presently no fiat currencies are available on a distributed ledger. However, several national banks are exploring issuing a Central Bank Digital Currency, including the Bank of England (2015).

In our system, parties to a contract may grant a contract manager permission to move tokens on their behalf, thus enabling the contract manager to automatically settle transfers required by a contract. If a party has insufficient funds, the contract manager may automatically declare the party defaulted through some appropriate mechanism. We consider such mechanisms for handling defaults in an automated way a very interesting area of future research and discuss them further in Sect. 6. Notice that such actions lie on the strategy side of a contract – parties preferring to settle their contracts manually could simply pick a contract manager that offers this option.

In some sense, a contract mananger acts as a trusted third party to whom financial contract parties, by their cryptographic signatures, issue powers of attorney to act on their behalf. These powers include granting control over their (digital) assets. That should give all parties pause prior to signing a financial contract with a contract manager, satisfying the cautionary aspect of contracts, possibly even more so than conventional contracts, which are usually informal and interpreted in the context of judicial practice and convention. Further, assuming contract managers are implemented in a Turing-complete programming language, analyzing their properties is hard, both in practice as described by Luu et al. (2016) and as exploited in The Dao Hack (Buterin 2016), as well as in theory, since the problem is known to be computationally undecidable (Rice 1953). This further sharpens the demand and inclination for caution, as is desired under any binding agreement.

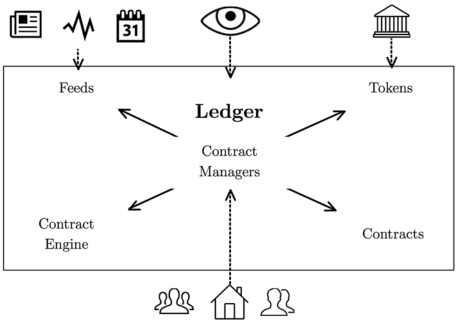


Fig. 10 Contract managers reside within the ledger, acting as a nexus that stores contracts, evaluates them using the contract evaluator, pulling the data necessary to complete evaluation from the appropriate feeds, and executing transfers by interacting with tokens. Users interact with contract managers to create, sign, evaluate and execute contracts. Feeds are populated by trusted external parties and tokens are created by institutions such as central banks. Regulators can inspect the ledger to get a full picture of the financial system

1. The parties inspect different options and agree on a contract manager that evaluates contracts by a logic they consider sound.
2. The parties construct or select a contract.
3. The parties grant the contract manager permission to move funds on their behalf.
4. One party registers the contract with the contract manager.
5. The contract manager checks that the contract is well-defined and that the requisite permissions have been granted for it to evaluate the contract.
6. All parties sign the contract through the manager; the contract is now in effect.
7. The parties repeatedly call the contract manager's evaluate function.
8. The contract is gradually reduced and cashflows automatically settle as the contract manager moves funds on behalf of the parties.
9. The contract eventually reduces to the zero contract.

Fig. 11 Step-by-step walk-through of the mechanics involved as a number of parties enter into a contract

Figure 10 summarizes the architecture of the implementation we have described. To further clarify how the different parts of the implementation play together, Fig. 11 offers a step-by-step example of how a number of parties enter into a contract through our implementation. Figure 12 provides a higher-level view of the relationship between the ledger, the contract evaluator (contract engine), contract managers, and contracts. At the lowest level, we find a generalized distributed ledger that has a Turing-complete programming language for expressing smart contracts. Built on top of this distributed ledger, we have the contract evaluator, which is implemented in the underlying distributed ledger's smart contract programming language. It

[7] A token is a smart contract that acts as an account manager, keeping track of different actors' balances of a specific unit of value. This tracking is often represented as a mapping from users to amounts. These smart contracts conventionally have functions for transferring funds and allowing other actors (usually other smart contracts) to transfer funds on the permitter's behalf.

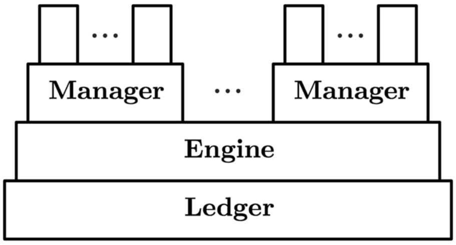


Fig. 12 The hierarchy of the implementation. At the top there are contracts, which are written by users and submitted to a contract manager. Contract managers may be created by, for example, financial service providers or the open-source community. They store, evaluate and execute contracts, all using the same contract evaluator (contract engine). The contract evaluator could, for instance, be created by regulators to evaluate contracts in accordance with a legally-binding semantics. Contracts, contract managers and the contract evaluator operate within a single distributed ledger

includes functionality for parsing a financial contract domain-specific language (DSL), and for evaluating and reducing financial contracts written in this DSL. At the highest level, we have contract managers. A contract manager is itself a smart contract, and so is also implemented in the underlying distributed ledger's smart contract programming language. It uses the contract evaluator to evaluate financial contracts expressed in the DSL and submitted by the users. It further interprets the result returned by the contract evaluator to take action in accordance with the contract strategy it encapsulates.

4 Choice of Distributed Ledger

Ethereum is a natural pick for a proof-of-concept implementation as it is widely used, stable and backed by a large community. Its object-oriented architecture, where different self-governing smart contracts interact in a standardized way, fits conceptually with our model of dividing responsibilities between contract evaluators, contract managers, feeds, and tokens (see Sect. 3).

In a more general sense, however, we consider the choice of platform to be of secondary importance. For a serious attempt at implementing a financial system on a distributed ledger, Ethereum is perhaps not the ideal choice, for a number of reasons. First, the computational cost (i.e., the required *gas*) of running a full-fledged contract evaluator is high.[8] Second, Ethereum's performance in terms of transactions per second is presently unlikely to meet the requirements of a global financial transaction system. Finally, transactions are accessible to anyone who has a copy of the ledger, which is a barrier to users who require keeping their contracts private, not just their identities.

We find that the common properties of a ledger implementation, such as its handling of privacy, its consensus mechanism, and its identity verification processes are orthogonal to the requirements of the architecture presented in Sect. 3. Thus, it should be a straightforward exercise to port our implementation to other widely-known generalized distributed ledgers, such as Hyperledger Fabric (Hyperledger Project 2016) and Corda (Hearn 2016), and those that come in the future.

We have identified features that we believe any ledger that aspires to host a financial system should offer:

Private transactions It should be possible to keep transactions private to the involved parties, with the possible exception of designated auditors, such as institutions responsible for financial stability. Corda, amongst other ledgers, offers such privacy without compromising consistency.

Authentication. For contracts that stipulate obligations beyond transfers that can be guaranteed by the ledger itself, it is necessary to authenticate parties by linking them to legal entities in the real world. Ledger-based know-your-customer (KYC) services, as described by Moyano and Ross (2017), could become a relevant service for authenticating users.

High performance. The ledger should be able to process large quantities of transactions quickly. As ever larger parts of the world economy are digitized, performance requirements will only increase.

It is worth reflecting on how a distributed ledger-based financial system compares to the existing system of back-office processes that are maintained by financial institutions.

In our view, the greatest benefit of the existing system is the relative ease with which manual overrides or corrections can be conducted – in the event of an unintended action, a bug, a hacking, or similar, the state change can often be reversed swiftly due to the fact that the process owner has full control. Such error mitigation is also possible on a distributed ledger, but the tools currently available are crude – on Ethereum, for example, errors not considered during the initial design of a smart contract can only be mitigated with a *hard fork*, which is the deployment of a new version of the ledger software that explicitly circumvents the error. Such *hard forks* are slow and controversial actions, as discussed by Buterin (2016). Another potential disadvantage of distributed ledger technology is the present prevalence of distributed ledgers that share and verify all transactions on all nodes in the network, giving rise to potential issues relating to performance and privacy. But as stated above, some distributed ledgers are already

[8] Our implementation runs on a private test-net where the cost of gas is not a concern.

moving away from the replicated state machine model implemented by an open, unstructured peer-to-peer system, the architecture of the original blockchain system, Bitcoin.

This contrasts to the numerous benefits offered by distributed ledger technology, which this paper has already touched upon in Sect. 1. These include, but are hardly not limited to: transparency of the mechanisms governing settlement and clearing; near-instantaneous settlement; streamlined auditing due to guaranteed consistency; higher reliability following from removal of single points of failure; and, lowered barriers to entry and thus increased potential for competition due to standardization and greater accessibility. Further, the potential in terms of cost-cutting are substantial: The only direct costs of distributed ledgers are for the hardware and electricity necessary to operate the nodes of the network. Santander Innoventures and Oliver Wyman (2015) estimate potential savings versus existing systems to be in the range of $15–20 billion annually by 2022. Such savings are liable to eventually benefit consumers.

5 Related Work

Various attempts exist of smart contracts that encapsulate a single financial contract. Notably, R3 (2016) offers an example of an interest-rate swap. Such ‘hard-coded’ contracts, however, mix both contract semantics and contract strategies in one single container, and hence do not offer a clear separation of rights/obligations (contract semantics) and assumed actions (contract strategies). Nor is it easily possible to author novel contracts.

Mortensen (2016) has implemented the equivalent of a contract evaluator for the Corda ledger with a contract language based on the work of Jones et al. (2000).

6 Conclusion and Future Work

Our research can be extended in many directions, not only to lift technical limitations, but also to establish a sound financial and economic model based on proper jurisprudence. Here we outline some of the tasks yet to be completed and questions that remain open for investigation. We list them in increasing order of abstraction.

First, the current implementation still has a rich roadmap, including the development of a varied sample portfolio with an accompanying test suite, implementation of a proper contract language parser (it currently requires users to input the abstract syntax tree of a contract), improvements in the handling and communication of errors, verification of the implementation, and if it is to ever leave the comforts of a test-net, significant performance tuning.

Second, the architecture presented in Sect. 3, upon which the implementation is based, is an obvious candidate for further refinements. Amongst possible pursuits is an investigation into a push-based model for feeds (which would guarantee proper timestamping of events), models for how to transfer a party’s rights and obligations for a contract to a third party (e.g., by inferring which parties would be affected by the transfer and only require these parties’ approval), a way to easily reverse contract evaluation as well as actions induced by it (e.g., for reversing transfers caused by corrupted feeds), and the implementation of prototypes on other ledgers than Ethereum, which could prove or disprove the general applicability of the model.

Third, the architecture presented in this paper places all the central pieces on the ledger. However, we do not discount the possibility that better models exist. We view two other models particularly as worthy of investigation. The first is a certificate approach in which contracts are stored, evaluated, and settled off-ledger and consistency is ensured through certificates that are verified through the ledger. The second is a compile-to-specialized approach in which contract, evaluator and manager are compiled off-ledger to a specialized and high-performance smart contract, which is then launched on the ledger. This could be done by verified software that is regulated and guaranteed by authorities.

Fourth, a different kind of pursuit would be to investigate new kinds of contract managers and the extent to which they can automatically do tasks that are currently manually or centrally maintained. We are particularly interested in models for how contract managers might prevent, ensure against or otherwise handle defaults. Conventionally institutions have assessed the credit-worthiness of individuals and set fees accordingly. This process might be managed in a decentralized, wisdom-of-crowds manner or even be fully automated. We have considered a schema in which the effectuation of a contract depends upon the auctioning of a credit default swap over that same contract. The price at which the CDS is bought would indicate the party’s credit-worthiness. This whole process could be automatically undertaken by a contract manager.

Furthermore, from a legal standpoint, the ideas presented in this paper would require a robust legal framework that clearly explains both the legal meaning of the contract language constructors and reduction rules, as well as the legal concessions granted to any given contract manager. It is a possibly insurmountable socio-political undertaking to create an understanding that accepts the mathematical semantics of contracts and contract managers as legally normative, and the actions taken by a contract manager as legally binding, particularly in the case of fully-automated contract managers, for which no real-world entities are legally responsible. Until it is available, it may be necessary to store legal contracts within the ledger, side-by-side with their implementation equivalents – this is the approach taken by R3 Corda (Hearn 2016). The overhead of such paired contracts could partially be alleviated by parameterizing legal contracts with values stored in a contract manager, in the style of *smart contract templates* as described by Clack et al. (2016). This raises the question, however, whether the formal contract code is to be considered legally binding and the associated text just nonbinding information, or whether the text is legally binding and the code a possibly-correct-possibly-incorrect implementation in cases where they are not consistent with each other.

Finally, we want to mention the opportunities that arise from placing an entire financial system on a single ledger. For example, regulators face issues assessing the risk of financial instability (European Commission 2016). If all transactions are logged on a shared ledger, regulators could develop simpler or partly automated models for regulating the financial industry and conduct systemic risk analysis in real time. Tax authorities could more easily find cases of tax evasion or fraud, or perhaps be able to automate (partially or completely) the correct reporting and collection of taxes such as VAT. Economists could compute key economic indicators such as GDP and inflation numbers based on up-to-the-minute transaction data on the ledger. We look forward to further research on these topics.

Acknowledgements We thank Sofus Mortensen for suggesting the topic of financial contracts on distributed ledgers.

References

Andersen J, Elsborg E, Henglein F, Simonsen JG, Stefansen C (2006) Compositional specification of commercial contracts. Int J Softw Tools Technol Transf 8(6):485–516

Andersen J, Bahr P, Henglein F, Hvitved T (2014) Domain-specific languages for enterprise systems. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation. Technologies for mastering change, vol 8802. LNCS. Springer, Berlin, pp 73–95

Arnold B, Van Deursen A, Res M (1995) An algebraic specification of a language for describing financial products. In: ICSE-17 workshop on formal methods application in software engineering. pp 6–13

Bahr P, Berthold J, Elsman M (2015) Certified symbolic management of financial multi-party contracts. In: Proceedings of the 20th ACM SIGPLAN international conference on functional programming, ACM. pp 315–327

Bank of England (2015) Digital currencies. https://goo.gl/BvHbRU. Accessed 5 Nov 2017

Buterin V (2014) SchellingCoin: a minimal-trust universal data feed. https://goo.gl/w2aJwu. Accessed 5 Nov 2017

Buterin V (2016) Critical update re: DAO vulnerability. https://goo.gl/Ojh8i1. Accessed 5 Nov 2017

Clack CD, Bakshi VA, Braine L (2016) Smart contract templates: foundations, design landscape and research directions. arXiv:1608.00771

European Commission (2016) Financial stability: new EU rules on central clearing for certain credit derivative contracts. https://goo.gl/676Hz8. Accessed 5 Nov 2017

Frankau S, Spinellis D, Nassuphis N, Burgard C (2009) Commercial uses: going functional on exotic trades. J Funct Program 19(1):27–45

Hearn M (2016) Corda: a distributed ledger. https://goo.gl/KzAmDR. Accessed 5 Nov 2017

Henglein F, Stefansen C, Simonsen J, Larsen K (2009) Poets: process-oriented enterprise transaction systems. J Logic Algebraic Program 78(5):381–401

Hvitved T (2010) A survey of formal languages for contracts. In: Fourth workshop on formal languages and analysis of contract–oriented software. pp 29–32

Hvitved T, Bahr P, Andersen J (2011) Domain-specific languages for enterprise systems. Department of Computer Science, University of Copenhagen, Tech. rep

Hvitved T, Klaedtke F, Zalinescu E (2012) A trace-based model for multiparty contracts. J Logic Algebraic Program 81(2):72–98

Hyperledger Project (2016) Hyperledger fabric: protocol specification. https://goo.gl/Z2PDHd. Accessed 5 Nov 2017

Jensen MV, Pedersen LH (2016) Early option exercise: never say never. J Financ Econ 121(2):278–299

Jones SP, Eber JM (2003) How to write a financial contract. In: Gibbons J, de Moor O (eds) The fun of programming. Palgrave Macmillan

Jones SP, Eber JM, Seward J (2000) Composing contracts: an adventure in financial engineering (functional pearl). In: Proceedings of the 20th ACM SIGPLAN international conference on functional programming, ACM. pp 280–292

Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, ACM, New York, NY, USA, CCS '16. pp 254–269. https://doi.org/10.1145/2976749.2978309.

Merton RC (1973) Theory of rational option pricing. Bell J Econ Manag Sci 4(1):141–183

Mortensen S (2016) Universal contracts. https://goo.gl/u64skF. Accessed 5 Nov 2017

Moyano JP, Ross O (2017) KYC optimization using distributed ledger technology. Bus Inf Syst Eng. https://doi.org/10.1007/s12599-017-0504-2

Nakamoto S (2009) Bitcoin: a peer-to-peer electronic cash system. https://goo.gl/wXWfP. Accessed 5 Nov 2017

R3 (2016) IRS demo. https://goo.gl/miGCVa. Accessed 5 Nov 2017

Rice HG (1953) Classes of recursively enumerable sets and their decision problems. Trans Am Math Soc 74:358–366

Santander Innoventures, Oliver Wyman (2015) The fintech 2.0 paper: rebooting financial services. https://goo.gl/xMXtks. Accessed 5 Nov 2017

Schneider J, Blostein A, Lee B, Kent S, Groer I, Beardsley E (2016) Blockchain-putting theory into practice. http://www.finyear.com/attachment/690548/. Accessed 25 Nov 2017

Schuldenzucker S (2014) Decomposing contracts. Master's thesis, University of Bonn

Wood G (2016) Ethereum: a secure decentralised generalised transaction ledger. https://goo.gl/0R6Slw, accessed 5 November 2017