Verdict: Blockchains for complex, real world smart contracts.

Nikk Wong
nikk@verdict.network
www.verdict.network

Abstract. Current smart contract languages are non-efficacious at modeling highly-complex real-world organizational structures, in part due to their inability to handle principal-agent issues. A highly opinionated smart contract framework could allow developers to easily build and manage these organizational structures by implementing permission-controlled restrictable account and contract types. Further, the framework could aide in the development, security, interoperability, and maintainability of decentralized applications (dApps) created on the framework, turning imperative 'tangled-web' smart contracts into simple, declarative relationships that are simple and easy to understand. The blockchains implementing this framework could boost the productivity and utility of their organizations, allowing new classes of smart contract and blockchain based organizations to emerge.

1.0 Preamble

The concept of the smart contract was coined by Nick Szabo in 1994, but the first widely used implementation was Ethereum's DAO in 2013[1]. Ethereum's smart contract language, dubbed *Solidity,* is a turing-complete programming language which enables the programmatic control of the transfer of funds and state changes on the network. Solidity is credited with allowing complex transactions to take place on the blockchain, leading to the creation of new types of organizations and decentralized applications (*dApps*).

Solidity is a generic language, and provides developers some (but not many) tools to manage the networks they're creating. This is done by choice, Solidity and it's assembly-like counterpart are intended to give developers unfettered access over control of the contract's execution. This allows for flexibility, yet does little to address higher-order relationships which may become tedious to define and maintain.

Principal-agent problems involving multiple parties are especially difficult to define and enforce in Solidity. ERC20 tokens in Solidity are almost always fungible (although a light non-fungible version currently exists), meaning that, similar to traditional fiat money, once money leaves a user's account (or jointly controlled contract, i.e. multisig), the issuer has little control over how it's used. Fungibility is an important characteristic of money in certain contexts. For example, it would be problematic if one USD note had characteristics which changed its denomination relative to an otherwise similar USD note.

However, many networks in the real world benefit from the implementation of a tradable, non-fungible asset. Imagine a gift card network wherein a participating vendor, *Vendor A*, issues a redeemable card to a customer. The card may contain a few clauses, such that the customer can exchange the card for goods at Vendor A, or partnering store, *Vendor B*, at any time. The card also may contain the clause that the card is not redeemable at participating store *Vendor C* (maybe

they're on bad terms, etc.). These arbitrary rules are decidable by the issuing vendor at time of issuance.

Non-fungibility is important in this hypothetical network. If there was no way to track the *type* of card and the rules associated with the card the customer currently had, vendors would have to treat all cards in the network generically, depriving the network of some important utility (the ability to spend at Vendors A & B, but not C).

Networks which allow parties to track the issuance and movement of a non-fungible currency are most notable for their ability to solve principal-agent problems. Vendor A can ensure that any stipulations they create will be adhered to by a redeeming customer—i.e. the customer will *never* redeem the issued card at Vendor C, thereby protecting Vendor A's interest.

Most organizations have a similar structure—a few different account types, each with their own interest to protect. Another example could be a school district network, in which parents are issued some proprietary currency, *SchoolDistrictToken,* for their children which are exchangeable for goods or services in the school throughout the year. By issuing proprietary tokens to their children, parents can erase a principal-agent conflict of interest—the tokens are only redeemable at the school, so students can *only* use the currency to buy goods at the school. They can't, for example, take the tokens and use them to buy candy down the street, (the tokens are not liquid as USD)—thereby conforming to the parent's interest and avoiding any principal-agent problems.

And, imagine

> *"Administrators have noticed that too many tokens are floating around and decide that newly issued tokens should expire at years end, meaning that parents whom purchase school tokens this year will have receive a stable currency for each token they still hold at year's end. The existing circulation is not affected."*

We don't have to stretch our imaginations far to imagine a network which has multiple parties which all wish to avoid principal-agent problems. For example, in our school district network, an administrator account may wish to restrict parents from trading the tokens they've received for USD, creating a secondary-market for the tokens, or other arbitrary rules. Students too, may wish to restrict how they spend the funds, maybe a student account wishes to restrict herself to a cheaper meal plan to save allowance (deciding avoiding principal-agent problems from her future self), etc. For a list of examples see Section 3 Examples.

The principal-agent problem is as old as society itself, and there are several ways that blockchains can be implemented to make sure every party in a network can protect their own interest. When implemented correctly, the types of networks that blockchains support will allow future networks to conduct highly complex and specialized transactions.

1.1 Multiparty principal-agent solutions are difficult to implement in Solidity

The problem, however, is that the network in which everyone can protect their own interest (presumably through the issuance of their own smart contract) is very difficult to do correctly in Solidity, and end up being tangential to building organizational business logic.

We'll illustrate this by example. Our school district token's aforementioned requirements were:

1. Administrators can restrict tokens from leaving the network (i.e. being swapped out for cash, base currency, etc).
2. Parents can restrict child spending.

This simple implementation is not trivial in Solidity. Novice developers will notice that requirement #1 is not enforceable via a vanilla contract or contract wallet, as these contract flavors can only apply their contractual restrictions for the period in which the funds actually reside in the contract. Contract wallets are quite useful for ensuring spending is properly permissioned. However, they are not a cureall. Once the funds are dispersed to the next owner, the contract no longer holds the funds and cannot restrict spending in any way. There are non-ideal ways to enforce transfers from contract wallet to contract wallet, essentially 'chaining' the logic, but those also suffer from issues.

Rules like #1 and #2 are best suited to be executed in Solidity's `transfer` call, (pseudocoded below, we're going to ignore conventions such as `accept`, `acceptAndCall`, etc. for brevity), in which we can run arbitrary rules applied from multiple parties to decide if a transfer should happen or not. A generic implementation looks like so:

```
SchoolDistrictTokenContract {
        …
        transfer(address _to, uint256 _value) {
                require(balanceOf[msg.sender] >= _value;
                require(balanceOf[_to] + _value >= balanceOf[_to]);
                balanceOf[msg.sender] -= _value;
                balanceOf[_to] += _value;
        }
}
```

As such, the transaction will be cleared as long as the sender has enough funds. This is not what we want. We need to make sure that each specific token's actions will allow or prevent the token from being spent. In Solidity, this can only be done if we use the non-fungible token standard, and a bit of contract trickery:

```
SchoolDistrictTokenContract {
        …
        transfer(address _to, uint256 _value, address _tokenId) {
                ...
                // as per https://github.com/ethereum/EIPs/issues/721
                // grab the token metaData which has been mapped in this contract
                Storage tokenMeta = tokenMetadata(_tokenId)
                // get the address of some attached validation contract and call it
                bool canTransfer = nameReg.call(bytes4(sha3("validateTransfer(uint256)")),
```

```
                    tokenMeta.someValidationContract);
                ...
            }
        }
```

In this scenario, the school district only issues NFTs and assigns each token an ID, which can be used to lookup the current restrictions on the token—the behavior is essentially 'attached' to the token.

The benefits of this technique are not without its tradeoffs, some of which are:

- Contract versioning must be done manually, and can potentially lead to security holes.
- It's difficult to gauge the restrictions on a NFT; NFTs with many restrictions may have ambiguous uses.
- Incorrectly written contracts anywhere down the line can lead to non-transferable, 'frozen' funds.

Further, blockchain applications in the real world are generally private but hold even deeper organizational complexity.

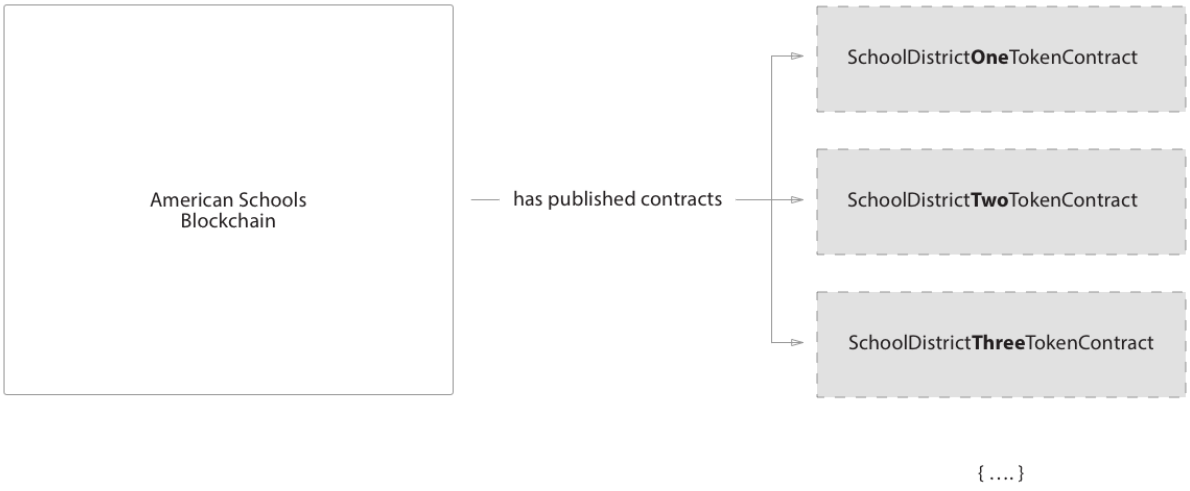

Fig 1—A hypothetical blockchain where organizations band together for mutual benefit.

We can imagine a scenario exists where different school districts operate on the same blockchain, to reap benefits such as interoperability, auditability, economies of scale in backing their tokens by fiat, etc. Though this will introduce benefits, it will also add more complexity.

Real world requirements also require factories which can create different types of accounts for different types of users, which have their own permissions, need to be revocable, etc.

Once we mirror real world requirements in Solidity our list of requirements soon become unmanageable and we end up writing more implementation logic than business logic. Further, that logic is hard to test, manage, and modify, leading to incidents like that in the DAO.

Smart contracts can be far more advanced than those we've seen thus far. Solidity is an incredibly powerful framework for creating smart contracts, but is impeded by a few problems:

- **Unopinionated**. Apart from the core APIs, there are few coding conventions in Solidity that are enforced within the community to enable the creation of next-generation contracts.
- **Procedural**. Contracts in Solidity allow for their state to be modified in unpredictable ways, leading to bugs which can (and do) lock funds.
- **Fragmented**. It's easy to spread business logic across multiple contracts, obfuscating meaning and creating more bugs.
- **Unversioned**. Versioning your contract requires the creation of duplicates and proxy contracts[2].

2.0 Introducing Verdict

Verdict is an opinionated framework for expressing complex, real world organizations in smart contracts. Verdict strives to provide utilities similar to other programming frameworks—abstraction, flexibility, security, and speed which make creating complex smart contracts easier. Verdict is **not** meant to be a replacement or competitor to Solidity. In many cases, Solidity provides APIs that make smart contract development easy.

Other times, Solidity is not enough—and Verdict shines in creating contracts that are naturally complex, requiring coordination between multiple account types, tokens that need extensive control measures, etc. Verdict attempts to solve principal-agent problems, by handling the life cycles associated with attaching and removing restrictions from tokens as they flow through the network.

Contracts in the natural world are an arrangement in which involved parties agree to act in a restricted manner. Verdict attempts to mirror these relationships in the digital world by introducing a new concept, the *RestrictionContract*. RestrictionContracts adhere to the same API as vanilla contracts, (just like how a token contract is just a contract in Solidity), but choose to implement one or more `restriction` typed functions in the contract's body.

2.0.1 Subissueable tokens, Restrictions, and non-fungibility

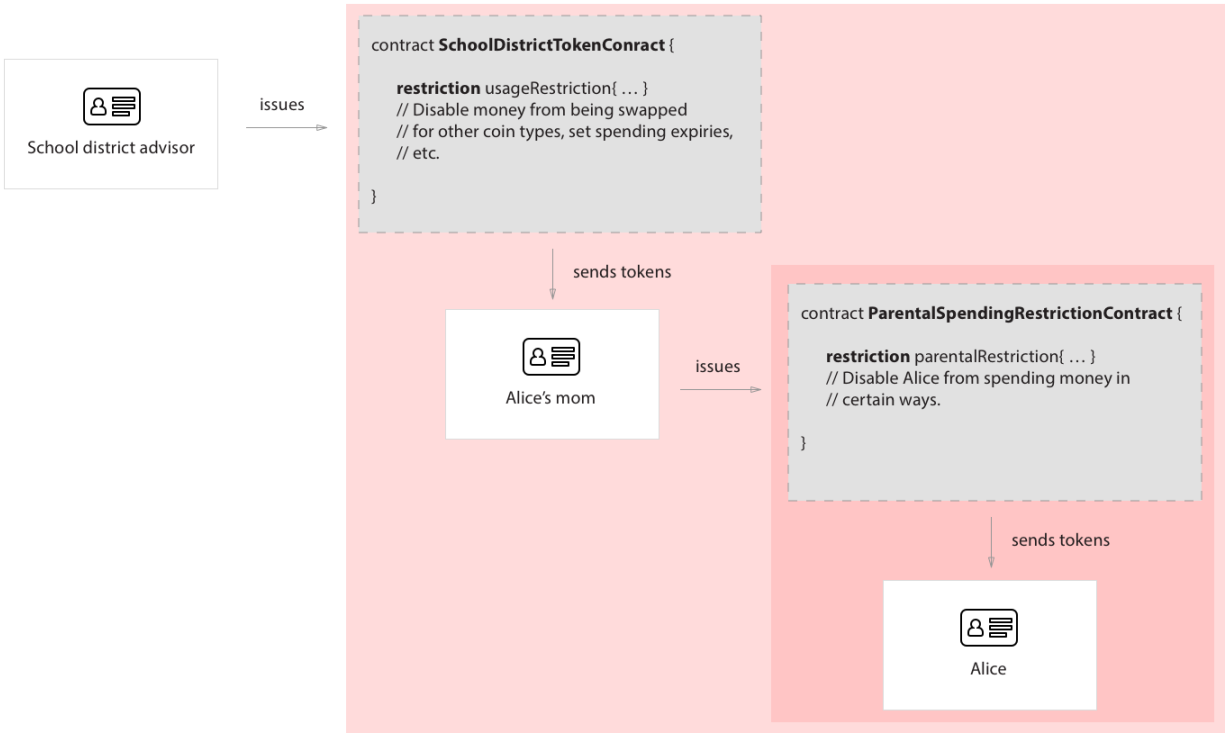

Fig 3—Relationships in Verdict. Alice is subject to the restrictions from both the SchoolDistrictTokenContract and ParentalSpendingRestrictionContract.

Fig. 4 demonstrates a series of two transactions in Verdict. The first transaction is from the School District Advisor account to the *Alice's Mom* (a parent) account. The second transaction is from the *Alice's Mom* account to the *Alice* (a student) account. The net effect is Alice receives an unspecified

quantity of tokens that have the restrictions from both the *SchoolDistrictTokenContract* and the *ParentalSpendingRestrictionContract* rules placed on them. This relationship is denoted in red—the SchoolDistrictTokenContract effectively encompasses the ParentalSpendingRestrictionContract. Alice will only be able to transfer her tokens if the return value of both calls to the respective restriction typed functions result in boolean `true` values. This is achievable because all tokens in Verdict are *non-fungible*—or non-equally interchangeable.

Permissioned accounts (discussed in #2.0.3) in Verdict can 'cleanse' tokens of their restrictions, meaning tokens can be recirculated without impediment. An example of such in practice would be a contract which waits for Alice to make a transfer back into the school in some way (maybe for lunch, or books, etc), at which point the token is instantly transferred back to the School District Advisor account who removes all restrictions (those imposed by Alice's mom) on the token and reissues the tokens to another parent. For an example, see section #2.1.

2.0.2 RestrictionTypes

There are certain issuable restrictions that are very common and are therefore built into the network. These can be thought of as the network's 'packages'. A non-exhaustive list of these restrictions may be composed of:

DiscountFor, DiscountFrom	*PremiumFor, PremiumTo*	*RefundableFor, RefundableTo*
OnlyToAddress	*OnlyAtTime*	*OnlyLimit*
OnlyToAccountType	*Expires*	*TransferTo, TransferFor*

For illustration purposes, the *Expires* restriction type may be imposed by the token administrator to make a token unspendable after a certain period of time.

2.0.3 Accounts

Verdict would suffer if everyone in the ecosystem could write and remove restrictions to tokens. For example, Alice should not have the permission to make a token *RevokableTo*—allowing her to withdraw a token that she previously transacted.

Verdict manages the life cycles associated with account creation (i.e. account factory contracts), modification, analysis, and revocation, allowing developers to work on their network's special characteristics rather than boilerplate. Internally, Verdict also manages the CRUD of permissions granted to account types or instances.

The ability to issue the aforementioned classes of restrictions on tokens can also be modified on a permission controlled basis. For example, it would be advantageous to the network if *students* and *administrators* don't have the same access to modify tokens in the network. For example, the restriction *ExpiresOn*—which can be used to make a token expire on a particular date, probably should only be grantable by the organization's administrators. Further, network administrators in

our hypothetical network may decide that parent accounts should only be granted permissions which allow them to decide how their children use their tokens, and nothing else.

———

The obvious benefit here is composability, which can be simple or complex as per the system's requirements. This allows networks to form and manage relationships in a more useful and flexible manner than was previously possible.

Verdict's innovation is its particular combination of the use of account life cycle management, non-fungible tokens, and opinionated contracts (like restriction contracts). These factors combined can fully mitigate principal-agent problems, causing tokens to transact freely with all parties' interests held in check. This specific flavor of contract development aims to be opinionated, yet flexible enough to easily model and facilitate the relationships that are likely to exist between individuals and organizations in the natural world.

2.1 Authority, Verdict's Contract Language

Authority is Verdict's contract programming language. Structurally, contracts written in Authority look and feel very similar to the Solidity syntax that blockchain developers are accustomed to, and porting contracts from Solidity to Authority is easy. A minimum viable token contract can be defined as so:

```
schoolDistrictTokenContract = {
   private state: { tokens<Token[]>: [] },
   mutation transfer: function (
        {message: Message, _state: State},
        {tokenId: byte32, to: address}): State {
       const token: Token = _state.tokens[tokenId]
       if (token.owner !== message.sender) throw('Incorrect user')
       _state.tokens[i].owner = to
       return _state
   }
}
```

Though the API has a similar flavor to that of Solidity's, under the hood Authority has a few key differences which make developing complex contracts easier:

- *Purely functional.* Only `mutation` typed functions can modify the state of a contract, which is only allowed by returning the entire modified contract `state` object. Testing contracts for behavior and side effects is trivial in Authority, which is paramount for contracts that handle financial transactions.
- *Versioned.* Under the hood, Authority tracks changes that are made to a contract's dependencies, via an internal proxying mechanism similar to manual solutions rolled in Solidity. If a contract doesn't trust a dependency, it has the option to opt in or out of updates.
- *Predictable and maintainable.* It's impossible to commit implicit state changes to contracts, since only explicit `mutation`s are able to change state. This enhances contract

interoperability, behavioral complexity, and maintainability by scores in comparison to current contract programming languages.

Authority is currently in development, and more examples of Authority's use as well as edge case handling will be discussed later.

2.2 Verdict Blockchains

Verdict offers several tools that developers can use to spin up networks with ease. Verdict blockchains can be privately or publicly deployed, or privately managed, in a sort of "Verdict-as-a-service" for or organizations. Most deployments of Verdict are expected to be private, inclusive networks—schools, churches, businesses, etc. These types of networks benefit in terms of scalability and security from private chains with member run nodes (at least until *LN* and other blockchain innovations become commonplace and fix the problems with today's chains).

The Verdict blockchain is inspired by and based off Ethereum's, in which state is a first-class citizen. The Public Verdict Network (*PVN)* is Verdict's shared blockchain, which anybody can publish to or build upon. Running an organization on the PVN vs a private Verdict network is beneficial for communities who would like to benefit from:

- The transparency granted by a large network.
- Decentralized and more difficult to cheat.
- Cheaper, requiring no setup fees.
- Quick to implement, with no maintenance required.

That being said, organizations will have to balance tradeoffs to find the implementation mechanism that is preferable for their specific needs.

2.3 Protocol Token

Verdict's cryptoeconomic protocol creates financial incentives that incentivizes members of the system to behave in a way that benefits the network. Each party in any given network has different needs and financial incentives—but protocol tokens align financial incentives and offset costs. The PVN makes use of Verdict Tokens (*VRD*), which are synonymous to ether in Ethereum. VRD tokens are used by members of a network to execute state-changing code on Verdict's distributed virtual machine, allowing users to facilitate transactions, create smart contracts, accounts, etc.

VRD Tokens will be deployed as ERC20 compliant tokens to the Ethereum blockchain with a fixed supply that will be issued to partnering dApps and future end users. When the Verdict blockchain and Authority languages have been released, VRD Tokens will be transferable to the Verdict blockchain for use in the Verdict ecosystem. Protocol tokens will have two uses: for market participants to pay transaction fees to nodes hosting the Verdict blockchain and for the use of Verdict hosted chains. The VRD tokens will not impose unnecessary costs on users, seek rent, etc. Verdict's blockchain will be publicly accessible and cheap to use.

3 Examples

The blockchain is evolving rapidly and replacing incumbent infrastructure at a mind bending pace. However, the dApps built with smart contracts have had lukewarm responses—lots of interesting proofs of concept, but none that have hit wide markets. The virality of a semi-joke app, CryptoKitties, was at the time of this writing the most viral use of an ERC20 token to date. With that said, dApps can hit wide markets when built on the right infrastructure.

Verdict shines in facilitating networks involving multiple account types (i.e. *parent, student, admin*) which are vulnerable to principal-agent problems. We explore how Verdict can aide in these situations with some example use cases:

Organization Type	Account Types	Use case
School	Student, Parent, Vendor	Discussed at large above, but also including: Issue groups of tokens that are spendable only at select parts of the school, i.e. parents are issued tokens which are individually allowances for books, lunch, school activities, etc.
Loyalty	Custodian, Store, Customer	Create a loyalty network in which participating vendors can issue tokens to customers who are free to spend them with any participating vendor—and, upon spending, retract the tokens to the original issuing vendor. Set discounts if tokens are spent at a particular vendor, at a particular time, etc.
HOA	Resident, Account supervisor	—Can elect to only disperse (or later revoke) funds to account supervisor under certain conditions. —
Charity	Charity, Vendors, Donors	—Donors can issue tokens which charity can only transfer to select vendors. —Donors can set to revoke tokens in case of audits.
Neighborhood group	Neighbor, Admin	
Family	Parents, children, vendors	
Youth group		
Fitness club	Club owners, Instructors, members	A reward system, in which members can redeem points for special perks, like massages, etc. Gym club owners issue 100 tokens per month to instructors who may place additional restrictions on them (ex. instructor Alice wants to reward Bob by giving him access to a specific gym amenity, and only that amenity) then issue them to members, based on their performance or other factors. The tokens are tracked as NFTs and expire so owners never expose themselves to too much liability, and instructors have an additional layer of programmatic control on how their members are rewarded for their efforts.
Trade group		
Due based organization	Organization member	Imagine an organization where you are required to do a bit of work every month as part of your membership. When you work, you're issued tokens. Theoretically, one member in the

		organization could work more than others and sell his tokens to other members who did not work. With Verdict's NFTs, you can restrict the transfer of token to a # of transfers (say 1 per month), allowing for some flexibility (maybe Alice is sick so Bob will work for her and give her his tokens) but mostly eliminating any secondary market and upholding the fidelity of the organization's expectations.

...etcetera. By starting work on these types of networks today, developers can pave the way for the transactional foundation for the organizations of tomorrow.

If you're interested in working on these (or any other) ideas, please send an email to ico@verdict.network.

3 Summary

We've described an opinionated smart contract framework based on a blockchain that enables the creation of complex contracts mirroring the relationships seen in the real world. In summary, contracts on Verdict will give rise to new ways for intra-organizational transactions that empower stakeholders to protect their interests, whilst being easy to develop and deploy for organization administrators.

Refererences

[1] *http://iqdupont.com/assets/documents/DUPONT-2017-Preprint-Algorithmic-Governance.pdf*

[2] The immutable nature of the blockchain means bug fixes or code updates to smart contracts which have already been committed are impossible. Consequently, Ethereum's blockchain is littered with abandoned contracts, which, when called, may produce unexpected behavior. In practice, contracts that need to be fixed or updated are simply committed again. To maintain fidelity, other contracts which had referenced the original, buggy contract must now also be updated to reference the new, non-buggy version, as do it's dependents, recursively. The obvious solution to this problem is for an implementing contract to avoid hardcoding a dependency contract's address and instead setting it as an updatable state variable. Empirically, this convention is not widely disseminated and thereby almost never followed, further complicating the Ethereum platform. Even worse, 'library contracts' (packages, essentially), must fight this reference problem by only exposing a *proxy contract's* address to the implementing contract, which internally maintains the content address of the library's most recent update, and thereby dynamically delegate the call.