Smart Governance for Smart Contracts

Erik Weitenberg (TNO)
Maarten Everts (TNO)

Dutch Blockchain Coalition
connect and create
www.dutchblockchaincoalition.org

Contents

In the past few years, blockchains have become very popular, not least because of their ability to run *smart contracts*, small computer programs that are executed and checked "on the blockchain". This, combined with the immutability of a blockchain's history, has led to some interesting issues: if someone manages to abuse a bug in your smart contract, the resulting damage might be expensive and irreversible. In other words, some smart contracts could benefit from some oversight. In this document we explore some of the ways such oversight, or *governance*, can (or cannot!) be implemented in a handful of current blockchain technologies and introduce a novel way to implement a generalised form of governance for Ethereum smart contracts.

1 Introduction

1 Introduction

The rise of Bitcoin and Ethereum has kick-started a growing interest in blockchain and distributed ledger technologies. In particular the possibility of running a small computer program (a *smart contract*) on a public blockchain, as introduced by Ethereum, has sparked interest. On the one hand, these smart contracts provide peace of mind: a network of impartial peers check that the smart-contract code was executed correctly, and that nobody has modified the code since it first appeared, allowing smart contracts to take on the role of a trusted, incorruptible third party.

However, the "immutable" nature of smart contracts is also a weakness: if your smart contract turns out to contain a critical bug, it may be very difficult to fix (see also TNO's whitepaper "Will that smart contract really do what you expect it to do?[1]"). Bugs in smart contracts, especially those that handle money or assets of high value, can be extremely expensive: a bug in the smart contract governing The DAO[2] allowed an attacker to gain control over more than 50 million USD worth of funds. In this case this led to a very controversial platform intervention and resulted in a community split, triggering the creation of a competing Ethereum off-shoot called Ethereum Classic. Since then there have been numerous other examples of bugs in smart contracts causing the loss of multiple millions worth of crypto-currency.

However, allowing updates to a smart contract is far from trivial. Since typically on most blockchains smart contracts inherently cannot be changed after deployment, supporting updates requires upfront effort and technical trickery involving proxy contracts. Even if smart contracts can be updated, one must make sure that only the right person, under the right circumstances can update a smart contract – otherwise, anyone might update your smart contract to a malfunctional or malicious version. In particular when the smart contract has many different stakeholders, it is complex to decide who gets to do what. For example, you might require that all changes are approved by at least two out of three group members. How can a piece of code enforce such a restriction?

In this document, we discuss a number of *technical* approaches to the problem of such governance of smart contracts and we aim for it to be a starting point for the exploration of this topic. Note that the governance of the blockchains themselves is a topic we explicitly do not cover here.

[1] https://blockchain.tno.nl/blog/smart-contract

[2] https://en.wikipedia.org/wiki/The_DAO_%28organization%29

2 Smart contracts

2 Smart contracts

To aid in the exposition of the material to follow, a short introduction to common implementations of smart contracts is in order.

A smart contract is a (typically small) computer program that exposes various functions. If the program is a wallet, for example, one such function might be “send money to person X”. In permissionless blockchains, anyone can call any smart contract’s functions, though the function itself might refuse to work based on internal criteria (“I won’t send money because the person asking is not my owner”). In permissioned blockchains, there might additionally be policies external to the smart contract that govern the use of some or all functionality.

In many blockchain implementations, especially those based on Ethereum, it is not possible to change the code of a smart contract after deployment. If programmed to do so, a smart contract has the ability to modify its state variables or to destroy itself, but otherwise, it will exist in its initial form forever. Such state variables typically can take the form of variables, arrays, tables or unstructured data. In particular, it can usually contain pointers to other smart contracts, and use those pointers to call functions in other smart contracts.

In the following sections, we will discuss the technical details of several governance and upgrade implementations.

3 Smart contract governance in permissionless blockchains

3 Smart contract governance in permissionless blockchains

In this document, we distinguish between two categories of blockchains. The first category is that of permissionless blockchains. Well-known examples of these include Bitcoin and Ethereum. Their common feature is that anyone, perhaps given some necessary resources, can run a node in the blockchain, and that there is no concept of a privileged user or administrator.

In these blockchains, any concept of governance for smart contracts must be implemented in a technical manner: your smart contract is not going to disobey another user just because you would like it to. It must be explicitly programmed to work as you intend; any mistake in this programming might be exploited by anyone.

We will discuss a few smart contract-capable blockchains. This excludes Bitcoin; though it has scripts, these offer only limited expressibility and functionality (not Turing-complete[3]) and Bitcoin scripts are not popular for development of smart contracts. In addition, we consider some possible ways to implement governance on the smart contracts running on these blockchains.

3.1 Ethereum

Ethereum[4] (and by extension, Ethereum Classic) is the best-known smart contract platform at the time of writing. It enjoys broad developer support, and smart contract governance has been implemented on Ethereum in a number of ways[5].
In Ethereum, smart contracts cannot be changed after initial deployment, but deploying a new smart contract is of course possible. Such a new smart contract will have a different address, so other software, including smart contracts, needs to be made aware of the location of the replacement smart contract. A straight-forward approach is to use a small smart contract (a mostly empty shell) that stores and manages a pointer to the current smart contract.

Alternatively, such a shell smart contract can be turned into a *proxy smart contract*[6] by programming it in such a way that it forwards any request (i.e., function call) to the smart contract in its pointer (see Figure 1). If this secondary smart contract needs an update, the pointer can be changed by someone the primary smart contract is programmed to trust with this responsibility (more on this later). The expectation is that the proxy smart contract, being as simple as possible, will never need an update, and that its address can therefore be relied on to never change.

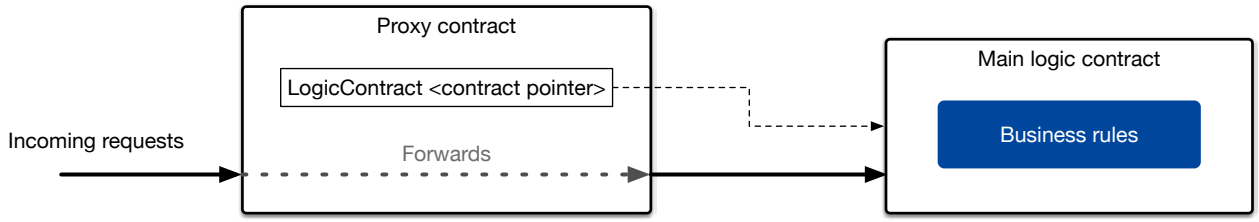
[3] See https://en.wikipedia.org/wiki/Turing_completeness

[4] https://ethereum.org/

[5] For an example, see uPort (https://github.com/uport-project/uport-identity), which is an identity framework that uses proxy contracts to act as a "user account" that can be revoked or given different access.

[6] See for example this blogpost (https://blog.gnosis.pm/solidity-delegateproxy-contracts-e09957d0f201) about the different kinds of proxies and their corresponding gas cost.

Figure 1- Smart contract proxy forwarding requests



Over the last few years, several such upgrade strategies have been explored.[7] A good example of a proxy smart contract implementation is provided by Zeppelin OS[8], which is a collection of community-vetted smart contract building blocks. In addition, Jack Tanner has summarised several proxy upgrade patterns on the Indorse blog[9], and there are several Ethereum Improvement Proposals[10] that attempts to standardise proxy contracts, and two more[11] that propose a consistent location to store the logic contract pointer.

An important, but tricky aspect of a smart contract proxies is the question of how to deal with the storage (i.e., state) of the smart contract, which on Ethereum is closely related to the way forwarding of smart contract requests is implemented. One approach is to let the secondary contract maintain its own state and require updates to somehow transfer and copy this state correctly to the replacement smart contract. Alternatively, using Ethereum's "delegatecall" functionality[12], it is also possible to keep all state at the proxy smart contract and let the secondary smart contract's code control this. The advantage of this approach is that the data does not have to be moved to a replacement smart contract when the secondary smart contract is updated. A disadvantage is that it is not feasible to change the data structure of the smart contract after its initial deployment.

To mitigate this disadvantage, the ZeppelinOS[13] explored several patterns using "delegatecall" and for example introduced a pattern called *Eternal Storage*. In this pattern, instead of regular struct-based storage, the proxy's storage is a string-to-bytes-mapping. Instead of a variable x storing for example an address, the smart contract would store an entry storage["x"] in the mapping and convert its content from a byte array to an address and back as necessary. By destructuring the storage like this, the smart contract writers only have to account for code upgrades, at the cost of extra gas needed for the conversions.

All-in-all, there is already a good body of work exploring upgradeable smart contracts for the Ethereum platform. The flexibility the solutions

[7] This blog post (https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns) contains some examples, also of what not to do, when designing upgradeable contracts.

[8] https://github.com/zeppelinos/zos/tree/master/packages/lib/contracts/upgradeability

[9] https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c

[10] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-897.md and https:/github.com/ethereum/EIPs/issues/2535

[11] https://eips.ethereum.org/EIPS/eip-1822 and https://eips.ethereum.org/EIPS/eip-1967.
The second one also implements an early form of governance by storing the proxy's owner's address.

[12] See https://solidity.readthedocs.io/en/v0.5.12/units-and-global-variables.html#members-of-address-types

[13] https://blog.zeppelinos.org/proxy-patterns/

provide comes at the cost of high complexity and requiring upfront effort. In addition, it appears the rules governing under which circumstances upgrades actually can take place is something that is not explored as much, which is why we explore this below in Section 5.

3.2 Tezos

Tezos[14], another public smart contract platform, offers the standard 'proxy smart contract' method, as well as a slightly expanded upgrade functionality[15].

Tezos smart contracts are written in a programming language called Michelson. Unlike Ethereum's smart contracts, these smart contracts do not distinguish between regular remote function calls and the delegatecall variant. Nevertheless, functionality equivalent to a delegatecall can be achieved, so proxy-based upgrade patterns can be used.

A drawback is that, since Michelson smart contracts are formally verified, the proxy smart contract needs to know the exact specification of the logic smart contract's interface, whereas in Ethereum you can sometimes get away with knowledge of only a partial ABI[16]. It is possible to work around this by having multiple layers of proxy smart contracts that translate commands from an older interface to a newer one, but this may nevertheless hamper the introduction of new functionality.

In addition, Michelson allows functions to be passed as parameters[17] and kept in storage. Therefore, a smart contract could allow upgrades to some or all functionality by storing its functions in state variables, which as opposed to 'normal' functions can be updated. Then, it could offer an upgrade command that under appropriate conditions replaces a stored function with a new one.

Though this offers a lot of flexibility, care should be taken to check whether the user submitting the new function is authorised to do so. In addition, this does not allow for changes to the smart contract's storage structure (though workarounds exist), or the number of methods or their signature.

3.3 EOS

EOS is a smart contract platform that is public and world-readable, but requires prospective smart contract users to make accounts on the blockchain. Smart contracts' usage can also be restricted to accounts; these are not unlike Ethereum's wallet addresses, but allow for slightly more fine-grained control.

One of its features[18] is truly upgradeable smart contracts. This means that a smart contract developer can submit a transaction that changes the code of a smart contract, leaving its storage and other properties intact. Note that if the new smart contract code needs data to be in a different format, the smart contract should perform the conversion by itself.

[14] https://tezos.com/

[15] Source: Beerend Gerats, Erik Jansen, Jan-Jaap Korpershoek, Arne de Roode (personal communication).

[16] Application Binary Interface: the place in a smart contract's bytecode where the contract's state variables are stored, and the locations in the bytecode of its functions. To use another contract's functions, you need to know their location, but you don't necessarily need to know the location of every function, hence 'partial'.

[17] https://tezos.stackexchange.com/a/628

[18] https://developers.eos.io/eosio-home/docs

Since EOS requires users to have accounts to interact with smart contracts, it is possible to restrict the upgrade functionality to one person or, by way of a multi-signature[19], to a group of people. In general, for certain actions, one can assign a list of people that can sign a transaction performing that action, and specify a threshold number of people that have to sign such a transaction. It is also possible to empty[20] the list of people who have such permissions, effectively making a smart contract immutable.

3.4 Kadena

Kadena[21] is a blockchain that offers both a public, proof-of-work-based version and a permissioned enterprise-focused one. Unlike for example Ethereum, Kadena allows contract code to be updated without having to replace the entire contract by a new one. Kadena's smart contracts are written in a specialized, lisp-inspired smart contract language called Pact. This language together with the Kadena platform allows[22] for both key-based and programmable governance for smart contracts. This platform-supported smart contract governance is supported by tooling and in this way Kadena sets itself apart from the more ad-hoc and "hacky" upgrade strategies in the Ethereum community as discussed above.

[19] https://developers.eos.io/eosio-nodeos/docs/accounts-and-permissions

[20] https://eosio.stackexchange.com/questions/4069/how-do-i-set-my-smart-contract-account-to-immutable

[21] https://kadena.io/en/

[22] https://medium.com/kadena-io/pact-solving-smart-contract-governance-and-upgradeability-976aac3bbb31

4 Smart contract governance in permissioned blockchains

4 Smart contract governance in permissioned blockchains

The second category of blockchains we will consider is that of permissioned blockchains. These are blockchains where technical restrictions allow only a certain group of entities to process transactions on the blockchain. Typically, the number of parties involved is small, making ad-hoc, in person governance easier than for permissioned blockchains. Nevertheless, for such blockchains it can still be useful to consider governance of a specific smart contract, in particular when the number of parties involved becomes larger and the impact of "manual" platform interventions becomes more invasive and costly.

4.1 Ethereum-based permissioned blockchains

Ethereum is open-source, and several projects have been started with the goal of creating a permissioned blockchain based on Ethereum. Examples include Hyperledger Burrow, Quorum, and (to an extent) VeChain. The advantage is obvious: it is the oldest smart contract platform, and most blockchain developers are adept at the programming language used to write its smart contracts.

This also means that the technical approaches to blockchain governance discussed above for Ethereum are directly applicable to these blockchains. Therefore, we will forgo an explicit discussion of this category of permissioned blockchains.

4.2 Hyperledger Fabric

Hyperledger Fabric also supports smart contracts ('chaincode[23]'), and in addition, allows deployed smart contracts to be upgraded or uninstalled. As a permissioned blockchain, Fabric allows administrators to regulate actions performed on the blockchain by having them checked against complex policies[24] like "must be signed by an admin of organisation A and two other admins, or 11 of 20 organisation B admins." This also applies[25] to transactions that create or update a smart contract. By default, a majority of the organisations in the sub-blockchain ('channel[26]') must have their administrators approve the smart contract; in addition, anyone who wants to use the smart contract must have approved it.

To effect an upgrade of a particular smart contract, the upgrader should install the new version of the smart contract on its peers. This will yield a package identifier. The upgrader can then create a new version of the 'chaincode definition' for the smart contract, which contains the package identifier along with a sequence number that indicates how often the smart contract has been upgraded. After submitting the new definition, the upgrader must wait for the other channel members to vote on the proposal; if enough members have approved it, the new definition can be committed to the channel, which finalizes the upgrade. The upgraded smart contract can contain initialization code that updates the smart contract's data (or creates new data).

[23] https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html

[24] https://hyperledger-fabric.readthedocs.io/en/latest/access_control.html

[25] https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html

[26] Fabric supports 'sub-blockchains' called channels, in which only a subset of the blockchain's users participate. A smart contract can then be confined to a certain channel.

4.3 R3 Corda

Corda, another permissioned smart-contract framework, also supports smart contract upgrades, although the upgrade[27] is slightly different. In Corda, smart contracts leave certain data on the blockchain ('state'), and smart contracts are used to transform this state (similar to Bitcoin's approach of "unspent transaction outputs"). In particular, multiple smart contracts might be suitable for working with a particular state, as opposed to Ethereum, where state is part of the smart contract and thus must be explicitly extracted if needed elsewhere.

The owner of a smart contract that needs to be upgraded has to advise all its users that an upgrade is incoming, and to stop using the old version of the smart contract. If any state belonging to the old smart contract needs to be changed into a different format for use with the new smart contract version, the parties must decide whether they will allow this: since Corda allows state to be shared between smart contracts, the old version of the state might still be needed for another smart contract. After all parties have agreed to the update, they will sign a special kind of upgrade transaction; when everyone has signed it, it is recorded in the ledger and the smart contract is replaced by its newer version.

4.4 Proxy smart contracts in general

We note that most blockchains support interaction between smart contracts, and thus, proxy smart contract-based governance methods like in Ethereum can be implemented to an extent.
For example, in the *Stratis* blockchain[28], smart contracts are written in C# and, similarly to Ethereum smart contracts, can invoke methods on other smart contracts. The same goes for the South-Korean *Aergo* blockchain[29].
Since these approaches to governance were discussed as part of Ethereum, we will not treat them separately.

[27] https://docs.corda.net/head/contract-upgrade.html

[28] https://stratisplatform.atlassian.net/wiki/spaces/WIKI/pages/1025174/Documentation

[29] https://docs.aergo.io/en/1.0/

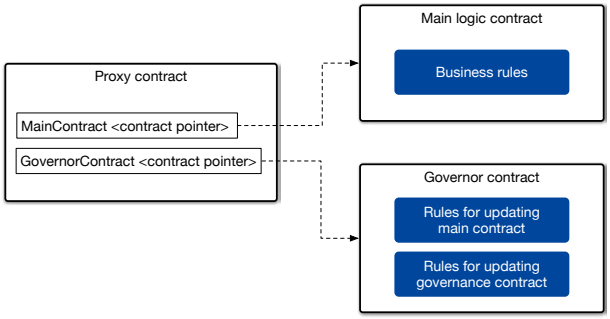
5 Bootstrapping governance

5 Bootstrapping governance

As alluded to above, if one allows for upgrading smart contracts, the actual rules describing under what circumstances this is possible, are important to get right. If this power is given to a single person, then why bother doing this on an expensive, distributed blockchain system? In addition, what type of governance to use really depends on the stakes at play, for example the amount of value that the smart contract manages. Furthermore, it is often not possible oversee at the start who should have permission to govern a certain smart contract. As an example, suppose you start a company, and you control certain aspects of your company and its funds in a wallet-type smart contract. You are registered as its owner, and therefore only you can for example spend the funds inside the wallet, or update (or replace) the wallet smart contract in case it turns out to have a security vulnerability. This works well, but at some point, a friend joins your company as a fellow founder. Now, you would like both of you to have authority over the funds, but you would rather not share the private key of the account that owns the wallet. What to do?

We propose, and have experimented with, an approach we call smart contract governance *bootstrapping*. The general idea of this approach is, starting from the proxy idea (see above), to not only have a (changeable) pointer to a logic contract, but to also have a changeable pointer to a *governor*. This governor has code describing under what circumstances the logic code can be replaced *and* under what circumstances the governer *itself* can be replaced. See also Figure 2. This approach allows us to start the wallet out with a very simple governor that allows for example only its creator to replace the logic or to replace the governor. When your friend joins, you create a new governor smart contract that allows both of you to do these things, and you ask the old governor to replace the wallet's governor pointer with a pointer to this new smart contract.

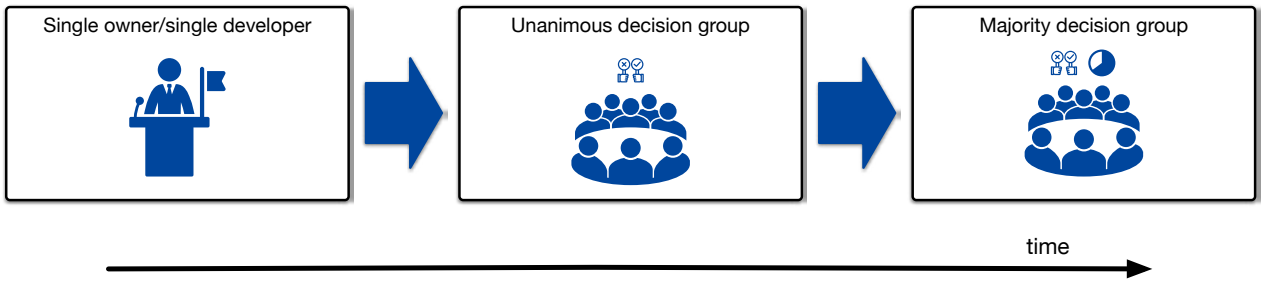
Figure 2- Smart contract governance proxy with logic and governor pointers.



You can keep repeating this procedure as the situation demands. For example, if you later hire a board of directors, you might replace the governor with a multi-signature governor that requires a majority of the board to agree to any changes and expenditures. Maybe you'll hire an accountant and create a governor that allows the accountant to spend any funds, but needs a majority vote from the board to do anything else. This flexibility mirrors the flexibility of real growing organisations, that adapt their governance structure to new needs (see Figure 3).[30]

[30] There are many of examples of such multi-party governance contracts: super-administrators (https://ethereum.stackexchange.com/questions/26726/how-can-i-set-owners-in-constructor-when-create-contract), multi-signature schemes (https://github.com/christianlundkvist/simple-multisig/blob/master/contracts/SimpleMultiSig.sol) and voting schemes (https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.html) are well-known concepts.

Figure 3- Governance can change over time



We developed and experimented with this bootstrapping approach for governance of smart contracts on the Ethereum platform within the Techruption Community[31]. More (technical) information can be found in a blogpost[32], which includes links to the open source code we published on GitHub.

[31] https://www.techruption.org/

[32] https://www.techruption.org/future-proofing-smart-contract-governance-using-governor-bootstrapping/

6 Concluding remarks & future developments

6 Concluding remarks & future developments

Smart contracts have great potential to take on the role of incorruptible third parties in distributed settings. However, their "immutability" after deployment also means that fixing bugs in the smart contract's code or updating them in any way is non-trivial. Therefore, it's important to properly think about *and* implement governance for smart contracts. Furthermore, proper governance is not something you can tack on after the fact; it is something you need to think about and deal with upfront.

In this short paper we have surveyed and discussed the current state of the art of smart contract governance. Existing approaches range from highly regulated, especially in permissioned blockchains, to quite ad-hoc, especially in permissionless ones. From the abundance of approaches to governance, we conclude that many forms of governance can indeed be implemented in smart contracts, and many blockchains have introduced implementations of varying maturity and versatility. Implementing smart contract governance is possible at the application-level, as is often done on the Ethereum platform, but it has the downside of adding complexity, resulting in more possibilities for high-impact bugs.

As an experiment in one of the ways to approach the problem, we have also introduced a generalised approach to governance of smart contracts, which we call *governance bootstrapping*. This approach can allow a smart contract's governance structure to grow alongside the organisation it supports, and to adjust to its changing interests and needs. Nevertheless, there is still room for experimentation with the various approaches to governance, taking clues from existing, traditional forms of governance. Future work could validate these approaches in existing organisations and use-cases to determine strengths and shortcomings.

Another promising direction for future developments is smart contract governance supported by the DLT platform, as exemplified in e.g. Kadena. As noted above, many DLT platforms implement smart contract governance at the application level, whereas there are only a few platforms in which governance is part of the infrastructure. This leads to an interesting trade-off between trust in the application and trust in the platform, which should be carefully evaluated.

Colophon

Dr. Erik Weitenberg
Research Scientist at TNO, Groningen

Dr. Maarten H. Everts
Senior Research Scientist at TNO, Groningen

Core Partners

Support Partners

Knowledge Partners

Dutch
Blockchain
Coalition

connect and create

info@dutchblockchaincoalition.org

www.dutchblockchaincoalition.org