Legal smart contracts in Ethereum Block chain: Linking the dots

Eleanna Kafeza
Zayed University
kafeza@zu.ac.ae

Syed Juned Ali
Data Science and Analytics Centre
IIIT Hyderabad
juned.ali@research.iiit.ac.in

Irene Kafeza
Kafeza Law Office
kafeza.e@gmail.com

Haseena AlKatheeri
Zayed University
Haseena.AlKatheeri@zu.ac.ae

***Abstract*—Block chain technology provides a decentralized and secure platform for executing transactions. Smart contracts in Ethereum have been proposed as the mechanism to automate legal contracts securely without the involvement of third parties. Yet, there are still several issues to be resolved especially regarding the updating of smart contracts in blockchain as well as the use of blockchain as part of a legal smart contracts system. In this work we propose a methodology and an architecture for building and deploying legal contracts in the blockchain. As the blockchain is immutable, we cannot update the code of the smart legal contracts, but in real life applications updating of contracts is a requirement that cannot be ignored. In this paper we address the problem of contract update by introducing a new versioning system that keeps track of the changes and links the different versions using a linked list. Moreover, we propose a system architecture where the user interface, the application logic and the blockchain are smoothly integrated in a manner that each part of the system contributes for producing a flexible and transparent execution. We show the applicability of our approach by implementing a system for the case of a rental agreement.** [1]

***Index Terms*—legal smart contracts, blockchain, smart contract modification, empirical study**

I. Introduction

Blockchain is becoming a disruptive technology for the legal field. The heart of this change lies to the fact that in blockchain traditional written rules with which people usually comply with, are no longer determined by the paper-based written agreements but by code in the form of smart contracts. There are several unique characteristics in this technology that label it as the next big thing in the law field. For example, it allows for trust-less transactions where parties can interact without the need of a third trusted party. As another example, smart contracts enable the "hard-coding" of legal contracts' obligations thus automatically imposing enforcement. Such property is of paramount value because it prevents possible breach of legal contracts before it can even occur.

The problem of autonomous contracting using machine intelligence is not new [1]. The possibility of legal smart contracts where the contract execution will be automatically enforced and monitor by an independent automated party has been attractive to the business environment. The definition of legal smart contracts was introduced by [2] where a smart contract is defined as a computerized transaction protocol that executes the terms of a contract. Moreover, this definition specifies the objectives of such smart contracts being to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. While related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs. This definition was introduced in 1997 and there have been several attempts to use technology to satisfy at least some of these terms. Legal XML aimed to create standards for the electronic exchange of legal data. In previous work [3] the authors map legal contracts modelled as workflows that expose different views to the participants. In some cases, there is a confusion between the term of legal smart contracts and smart contracts as introduced in the Ethereum platform.

Legal smart contracting creates a new broader opportunity for business interchanges that can be transparent and faster but at the same time, they are posing several challenges that need yet to be addressed. Currently, smart legal contracts can be viewed as a result of a process: legal contracts are written in a natural language, then translated to a business process and finally implemented by code in the blockchain. There are several steps then need still to be done to fully automate such a process. While at the same time, we cannot neglect that as mentioned in [4], technology is addressing only the technical part of the formation and the execution of a contract. In real life, contracts are part of a complex social network of relationships and in several cases, clauses are left unspecified and the parties' relationships have a dynamic nature that requires adjustments in the agreements. In [5], the author argues that there is a need for some mechanism that will allow the update of the contracts. Such modification can happen due to changes in the legal landscape for example while at the same time it has to ensure that terms of the contract will not change unilaterally.

In our work, we consider the above concerns about contract modification. Although the blockchain supports the immutability of the contract, we propose a versioning mechanism that will allow contract modification. As already mentioned the legal smart contract formation and execution is a process and we need to integrate several different technologies and application requirements at the same time. Example issues

[1]This research was supported by the Research Incentive Fund (RIF) Grant R18056 provided by Zayed University, UAE.

2473-3490/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDEW49219.2020.00-12

that need to be addressed are: a smart contract is written in code while the parties that are doing the legal agreement are interested to have access to the legal document as well. An additional application layer is required that will store the actual legal document. In Ethereum the contracts are written in Solidity and the compiler generates the bytecode. In this format the functions of the contract cannot be identified therefore a JSON representation of the could provide the functions and/or the event descriptions. The problem of the complexity of the programming languages and the ability to modify smart contracts has also been identified in the literature [6]. Similarly in the survey [7], programming of smart contract has been identified in the literature as a problem. The need for a road map regarding the use and evaluation of technologies in blockchain has been identified in [8] where the authors mention that existing empirical studies are in infancy.

In everyday business life, contracts execute in a dynamic environment that should allow for modifications. Contract modifications need to be done in writing. A contract modification can be Unilateral which means that the contracting officer makes the changes or bilateral which means that both parties are signing the changes. In our example, we have unilateral changes that are negotiated among the parties while changes lead to the contract modification. Several contract changes are possible but in most cases, there are a handful of changes that appear in most times. the contribution of our work is in several aspects:

- we provide a system architecture that allows the user to interact and update contracts while using the benefits of security and transparency provided by the blockchain.
- we propose a versioning mechanism, using link lists and data/logic separation, to handle the immutable nature of blockchain.
- we provide a roadmap on how different technologies can be integrated for building legal contracts applications, that can be applied to several other application fields as well.
- we provide a prototype for a rental agreement contract that shows the applicability of our approach.

The structure of this paper is as follows: Section II presents the related work, SectionIII describes the system architecture and the smart contracts modification mechanism. In Section IV the case study of the legal agreement is described. The smart contract lifecycle and the different feature of the legal business application are developed. The modification scenario is also presented. The road map for the implementation and several important highlights are described and discussed. Finally in Section V we are concluding and elaborate on future research directions.

II. Related Work

The problem of autonomous legal contracting has been addressed in the literature in the past. In [1], the authors describe a platform where agents subscribing in a given common platform are creating and executing contracts. The evolution of blockchain technology that enables transparency in contract execution eliminating third parties, has provided a new dimension to the problem of automated contracting. The authors in [9] provide an analysis of how concepts pertinent to legal contracts can influence certain aspects of their digital implementation through smart contracts, as inspired by recent developments in distributed ledger technology. They discuss how properties of imperative and declarative languages including the underlying architectures to support contract management and lifecycle apply to various aspects of legal contracts. They discuss how properties of imperative and declarative languages including the underlying architectures to support contract management and lifecycle apply to various aspects of legal contracts. Declarative smart contracts can better fit fundamental elements of legal contracts but fundamental problems can appear concerning the modification and termination of smart contracts. A declarative language may simplify the modification of blockchain smart contracts, but, compared to imperative counterparts, it may fall short of expectations in matters of computational complexity and associated costs. [9] also conclude that declarative and imperative approaches have the potential to complement each other for better opportunities. Smart contracts are defined as agreements existing in the form of software code implemented on the Blockchain platform, which ensures the autonomy and self-executive nature of Smart contract terms based on a predetermined set of factors. In [10] the authors analyze legal issues associated with the application of existing contract law provisions to so-called Smart contracts. While in [11] the authors examine smart contracts from the perspective of digital platforms and the Finnish contract law. They conclude that at least in some cases, smart contracts can create legally binding rights and obligations to their parties.

While computer code can enforce rules more efficiently than legal code, it also comes with a series of limitations, mostly because it is difficult to transpose the ambiguity and flexibility of legal rules into a formalized language which can be interpreted by a machine. In [12] the authors discuss the idea of ”Code is law” which refers to the idea that, with the advent of digital technology, code has progressively established itself as the predominant way to regulate the behaviour of Internet users. They describe the shift from the traditional notion of ”code is law” (i.e. code having the effect of law) to the new conception of ”law is code” (i.e. law being defined as code).

[13] discuss the legal applications of smart contracts. A software protocol performs an action (releases funds, sends information, makes a purchase, etc.) when certain conditions are met (payment is received, the outcome of an event is determined, etc.). The advantage of blockchain-based contracts is that they reduce the amount of human involvement required to create, execute and enforce a contract, thereby lowering its cost while raising the assurance of execution and enforcement processes.

Smart contracts are presented as a self-executing, autonomous alternative to traditional contracts that require enforcement by court involvement. The first experiences with smart contracts show that contracts involve more than con-

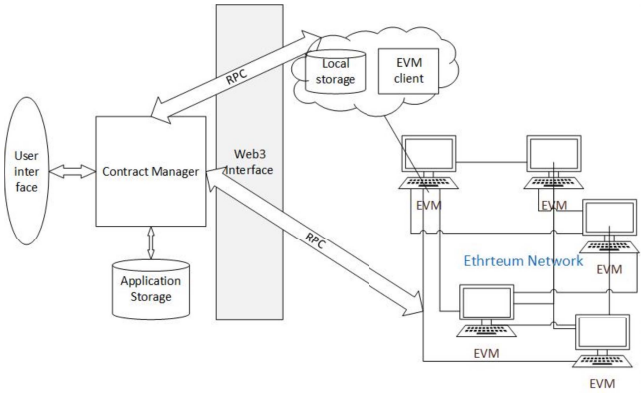


Fig. 1. System architecture for a smart contract application

ditional execution. [14] propose a method for formalizing contract law to make this as far as possible tractable for incorporation in smart contracts. Contract law can be viewed as a set of rules for resolving disputes while also protecting legitimate interests of parties.

III. System Model

A. Overview

Our objective is to build a system for the design and execution of legal smart contracts that can be modified based upon unilateral party requests. For the system design, we took into consideration that programming in the blockchain is a challenging task. Several technologies are involved and a solid software methodology has to be in place for combining them effectively. Currently, there is no such methodology and building applications for blockchain is an overwhelming task for developers and an impossible task for non-developers, dealt in a case by case basis. Our proposal for domain-specific applications is to base them on pre-existing templates that can significantly contribute to the development and will help developers to adjust or improve code while users can focus on the application logic instead of the coding issues. Traditional web applications are usually three-tier applications where the presentation tier displays the web page, the business tier which includes the business logic and the data tier that stores the data. In blockchain applications the communications and the demands are different. The presentation tier is still responsible for the interaction with the user. The business tier still includes the business logic and it is where the legal contract is defined as a template while the data tier stores the related information. The blockchain can be considered as another tier where the legal contract is deployed and executed.

When designing blockchain applications there is a choice regarding what to include in each tier. A two-tier architecture model would have the presentation and the blockchain tier. If such a model is selected, data storage and programming complexity issues need to be addressed. On the other hand, using a four-tier approach, the complexity of hardcoding the whole contract in the blockchain is reduced and contract creation is moved to the business tier that allows for more flexibility while it is easier to follow and update applications for the user. At the same time, the blockchain tier is used to provide its main characteristics, transparency, security and automated execution.

Our proposed solution provides the versioning of contracts in the form of a doubly linked list. Each smart contract has a previous and a next pointer that points to the next and the previous version of the smart contract. Each version handles different evolved requirements of a business process or solution. For example, the base version of a rental agreement shall require the tenant to pay rent, but an updated version of a smart contract might require to pay and some maintenance fee as well. The versioning helps in keeping track of evolution history of requirements as well as allows us to switch between versions to go back to the old requirements as well. Every new version of a contract is deployed again over the blockchain because of the immutability of the code of a smart contract on the blockchain.

B. System Architecture

Figure 1 shows our proposed architecture. The business tier includes the contract manager. The contract manager is the module where the business logic resides. At the data tier (database) all information necessary for the functionality of the business logic is stored. For example, a pdf version of the contract can be stored at the data tier. At the fourth tier is the Ethereum Network, the blockchain where the executable form of the contract is stored and accessed.

1) User interface: presentation tier: The user through the application provided User Interface interacts with the contract manager for various requests i.e. creating, modifying and deploying a smart contract. The user interface allows for a satisfactory user experience minimizing the complexity that arises when dealing directly with the blockchain. For the case of our example for the rental agreement, the user uses the web browser to access the capabilities provided by the contract manager.

2) Contracts Manager: business and data tier: The blockchain module represents the Ethereum blockchain where smart contracts are written in Solidity. Solidity is translated to EVM which is a bytecode language that executes on the Ethereum Virtual Machine (EVM) and an interface is created at the same time. The interface is used to deploy the contract using the Web3 library. When a contract is deployed, the EVM is the machine that runs the bytecode, hence the contract gets an address and the bytecode is pushed to the blockchain. The contract manager handles the application logic related and communicates with the storage to handle the necessary information of the smart contracts. For the interaction with the smart contracts deployed on the blockchain, the Application Binary Interface (ABI) and the address of the smart contract that we get after the contract is deployed needs to be provided.

3) Web3 Interface, blockchain tier: The Web3 Interface allows us to create and deploy the smart contracts used in our application to the blockchain tier. We use the python

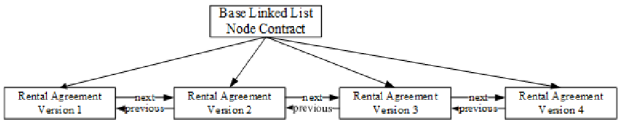


Fig. 2. Linking smart contract versions

wrapper of the Javascript library web3.js to interact with the blockchain. Moreover, another type of storage is necessary at this tier. An example would be storage that allows creating a content-addressable, peer-to-peer storing and sharing of hypermedia in a distributed file system (i.e. InterPlanatary FileSystem (IPFS)).

C. Smart contracts modification in Ethereum

Smart Contracts on Ethereum are responsible for defining the logic of execution of the business process. One of the main characteristics of blockchain technology is immutability. Each transaction is permanent and cannot be changed thus trust and integrity can be achieved. The drawback is that the contract itself cannot be updated since any update is considered as tampering with the information. Yet in real-life applications and especially for legal contracts, contract modification is a routine process. There are several different types of contract modifications. One case is when the contract logic changes but the data remains the same and the other case is when several changes can happen to the contract. We are adopting a versioning system regarding contract modification representation and a data separation implementation approach to allow for efficient contract logic modification.

```
pragma solidity ^0.5.0;
contract DataStorage {

    mapping (address => mapping( string => string )) keyValuePairs;
}
```

Fig. 3. A Minimal Data Storage Contract

1) Data and logic separation: Since we are interested to provide a flexible mechanism for updating not only the data but the logic of the contract as well, we need to apply a separation principle where the data and the contract logic can change independently. For any contract, we consider the change in the functional logic of the smart contract as an update in the smart contract. In several cases, the logic of execution changes but the data remains the same. Therefore, we need to provide a common source of data that can be used by several different version of contracts where the logic of execution is the only modification. We address this issue by creating a common smart contract for the data layer. Note that a smart contract, in this case, is not a legal contract, is a coding tool for separating data from the rest of the contract. As a next step, we can create multiple updated legal contracts that will import the data from the ”data store” smart contract. Each one of them needs only to contain the address to the data contract and thus it can use the data. Whenever we create a new smart contract or update a smart contract, we take the data from the *data store* smart contract and set the attribute values of the updated version of the smart contract. The updated version of the smart contract now uses this data to execute the logic. The values are assigned by our application layer, where we fetch the data of the storage contract and assign values to the new updated smart contract. Figure 3 shows a minimal data storage contract that stores the data of the smart contracts. This smart contract maps the address of the smart contract with a map. This map stores the key-value pairs of the attributes and values of the smart contract. For a new version of a smart contract to get all the data of the previous contract, the new version should know the address of the previous smart contract.

2) Contract versioning system: Having resolved the data/logic separation issue, the next problem to address is how to keep track of the contract modifications. Taking into consideration that a contract cannot be changed, we adopt a versioning mechanism for supporting smart contracts updates. Figure 2 presents our versioning system. Each smart contract is a derived contract of a base *Base* smart contract. The *node* smart contract defines a *node* in a doubly-linked list. We are interested after the creation of the initial contract to keep track of all changed contracts. This line of contracts modification becomes an evidence line that proves the changes. Therefore each smart contract is derived from the *node* smart contract. The links store the address of the deployed smart contracts. The contract manager sets the next and previous pointers of the smart contracts whenever a new version of a smart contract is deployed.

Note that the links provide the address of the previous and the next version of the contract. These addresses can be used to get the data from the data storage mapping contract.

With versioning, we can get the address of the next (or previous) version of the smart contract, but to interact with it, we would require the ABI of the next (or previous) smart contract. To handle this issue, we store the ABI file mapped with the address of the smart contract in InterPlanatary FileSystem (IPFS). We explain the versioning implementation in a detailed manner in the case study section. InterPlanetary File System (IPFS) is a protocol and network designed to create a content-addressable, peer-to-peer method of storing and sharing hypermedia in a distributed file system. In this way, using the address we can get the ABI of the contract from IPFS and then we can use the address and ABI to interact with the smart contract. In this work, we use a smart contract for storage.

IV. Rental Agreement Smart contract management Case study

We demonstrated the application of our methodology for designing and implementing modified legal contracts with a case study. A rental between a landlord and a tenant was used as the legal contract. The presentation tier is the interaction with the user, the contract management tier handles the rental agreement creation, execution and modification and the contracts are deployed in the blockchain tier.

Our case shows that the business tier holds the application logic. A legal contract that deals with the stakeholders, i.e landlord and tenant with the clauses of the agreement between the two parties, and with a version ability for modification is defined at this stage. The business logic tier acts as a tool that provides the manifestation of execution of legal contracts translated into smart contracts. While managing the smart contract, the application essentially manages clauses of the legal contract.

Our application allows the user to upload and deploy multiple instances of smart contracts which are a digital manifestation of legal contracts. Landlords can upload available properties and users can agree to the available contracts deployed by the landlords. Once the user agrees to the agreement, the application manages the transactions and clauses to validate the legal contract. Each smart contract is linked to a pdf of the legal contract. This allows the user to go through the contract in the English language before confirming the agreement. Our objective is to use the different tiers to implement different parts of the application applying our design architecture and to use versioning as an alternative of information update in the blockchain.

A. *Smart contract Life cycle*

1) Stakeholders: A person needs to login to our application to be able to perform actions and use our application. The actions are user-specific so that requires the user to login.

- Landlord - The landlord can upload and deploy a contract over the blockchain. A smart contract has user-specific tasks. For example, only a landlord can terminate a contract. Only a landlord can modify the contract. We explain the modification aspect of the contract in detail in the later sections.
- Tenant - A tenant and only the tenant can confirm the agreement to the smart contract. Only a tenant can pay the rent once they have confirmed the agreement to the contract. A tenant has to confirm the agreement again if the landlord modifies the contract.

2) Life Cycle: Life Cycle for rent payment to contract termination

- User logs in as a landlord
- Uploading contract
- Deploying a contract
- User logs in as a tenant
- Tenant confirms the agreement to the contract and pays the deposit as prescribed by the landlord in the legal contract
- Tenant pays the rent and the ethers worth of rent value are transferred from the tenant's account to the landlord's.
- Tenant pays the rent for next months
- Landlord can modify the legal contract and uploads new contracts and deploys it.
- Tenant can either confirm the modified contract or can reject it. If the tenant rejects the contract the previous contract is terminated, else the new modified contract is initiated and continues to execute over the blockchain.
- The tenant can cancel the contract midway before the prescribed period with giving a certain amount of fine.
- The landlord pays the tenant full or half deposit back depending on the time of contract termination by the tenant

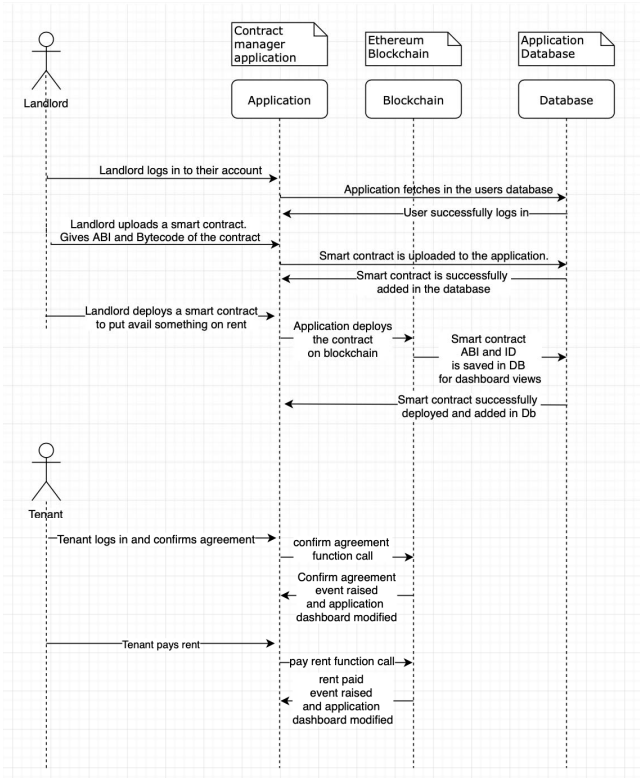


Fig. 4. Sequence of actions to deploy smart contract by landlord and pay rent by tenant

3) Features of the application:

- 1. **User Specific Dashboard for contract management** - A user can deploy multiple contracts and as the contract progresses, the application allows the users with respective actions to take. For example, a tenant gets the option to confirm the agreement before they have agreed to the contract, but after the confirmation of the contract, they get an option to pay the rent or terminate the contract. The dashboard shows all the contracts that are currently deployed by the user, i.e the logged-in user is the landlord of, and also shows all the contracts that the user is a tenant of. The dashboard also shows all the previous contracts that were deployed and provides an option to see the transaction history of the contract.
- **Contract Modification** - Our application allows for the smart contract modification by allowing the landlord to upload and deploy a new contract. We then link the new modified contract as the new version of the previous contract. Modification of the new contract could mean

TABLE I
TECHNOLOGIES USED IN OUR SOLUTION

Technology Used	*Purpose*
Solidity	Programming language to write smart contracts
IPFS	InterPlanatary File System used to store ABI of contracts
Python	Defining the Application logic of our Decentralised Rental agreement application
Web3py	Python library used to interact with Ethereum blockchain node and execute transactions using python code.
MetaMask	Chrome extension that provides User interface to connect to or host an Ethereum blockchain node
Ganache	Application that creates a blockchain node on localhost that can be used for testing decentralised applications.
Django	Python Framework to build the web based decentralised rental agreement application.
MySql	Database used for storing data on the Django application.

adding new clauses to the contract or changing the parameters like rent, the penalty for early contract termination, contract length etc. This allows the user to deploy a new contract as a new version of the previous contract, keeping the previous transactions intact. For example, say a house named H is under a contract between a landlord L and tenant T. After the contract is deployed a modification is required in the contract. Using our application, a landlord can deploy a new contract as a modification of the previous contract and the previous transactions will be linked to the H, even though the new contract does not have the old transactions related to the H between L and T. We define functions in the solidity contracts to link the various versions of the contract and store the address of the next and previous version as an attribute of the smart contract. The pointers of the smart contracts are updated when a new version is deployed.

4) Technologies used and Integration: Table I mentions all the different technologies used in our solution. The purpose of each technology is also provided in the table. The solidity programming language is used to write smart contracts and IPFS is used to store data related to rental agreements on the application. We use the python programming language to model business logic. The python application provides an interface for the end-user to create and deploy smart contracts. It also allows the user to update the smart contract with evolving requirements, for example modifying rent amount or adding a function that needs to be triggered to handle a newly introduced clause in the contract. The contract manager interacts with the smart contracts and InterPlanatary FileSystem (IPFS) which is used to store information required to interact with the deployed smart contract, for example, Application Binary Interface (ABI). The logic to store the files on IPFS is also provided by our application written in python. Web3py is a python library which provides an API that provides an interface to interact with a blockchain node. With Web3py library, contract manager interacts with the various versions of the smart contracts. We allow metamask as well as ganache runtime environment for the deployment of the smart contracts. The ganache runtime environment allows the user to test their smart contracts and the metamask can be used by the user in production to deploy the contract to the main Ethereum blockchain.

```
pragma solidity ^0.5.0;
contract BaseRental {
    /* This declares a new complex type which will hold the paid rents*/
    struct PaidRent {
        uint Monthid; /* The paid rent id*/
        uint value; /* The amount of rent that is paid*/
    }
    PaidRent[] public paidrents;
    uint public createdTimestamp;
    uint public rent;
    /* Combination of zip code and house number*/
    string public house;
    address payable public landlord, tenant;
    uint creationTime, contractTime;
    enum State {Created, Started, Terminated}
    State public state;
    /*Address of the next contract linked*/
    address next;
    /*Address of the previous contract linked*/
    address previous;
    constructor (uint _rent, string memory _house, uint _contractTime ) public payable{
        rent = _rent;
        house = _house;
        contractTime = _contractTime;
        landlord = msg.sender;
        creationTime = now;
        state = State.Created;
    }
    event agreementConfirmed();
    event paidRent();
    event contractTerminated();
    /* Confirm the lease agreement as tenant*/
    function confirmAgreement() public payable { /*confirmAgreement logic*/ }
    function payRent() public payable { /*Pay Rent Logic */}
    function terminateContract() public payable{ /*Terminate Contract Logic */ }
    function getNext() public returns (address addr){ return next; }
    function getPrev() public returns (address addr){ return previous;}
    function setNext(address _next){ next = _next; }
    function setPrev(address _previous){previous = _previous;}
}
```

Fig. 5. Base contract snippet for rental agreement

```
pragma solidity ^0.5.0;
contract RentalAgreement is BaseRental {
    /* This declares a new complex type which will hold the paid rents*/
    uint nextBillingDate;
    uint monthCounter;
    constructor (uint _rent, uint _deposit, uint _contractTime,
    uint _discount, uint _fine,
    string memory _house) public payable  {
        rent = _rent;
        deposit = _deposit;
        house = _house;
        discount = _discount;
        fine = _fine;
        contractTime = _contractTime;
        landlord = msg.sender;
        createdTimestamp = block.timestamp;
        state = State.Created;
    }
    /* Events for DApps to listen to */
    event agreementConfirmed();
    event paidRent();
    event contractTerminated();
    /* Confirm the lease agreement as tenant*/
    function confirmAgreement() public payable { /*Updated confirmAgreement logic*/ }
    function payRent() public payable { /*Updated Pay Rent Logic */ }
    function terminateContract() public payable{ /*Updated Terminate Contract Logic */ }
    function aNewFunction() public payable {/* A new function to do something advanced*/}
}
```

Fig. 6. Updated contract for rental agreement

B. Evolving Rental Agreement Case study: Implementation Details

Figure 5 and Figure 6 show the base and updated contract for rental agreement. The base contract shows the attributes necessary for a basic rental smart contract and the logic necessary for the functions like paying rent, confirming the agreement and terminating the contract. The updated smart contract involves a new function that can be used to add functionality to the smart contract logic or the existing functions can also be updated. The base contract also involves the previous and next contract version addresses. This address is used to fetch the data associated with the contract from the data storage contract.

We used Django inbuilt authentication module with a modified user model. The database table definition required for our application was -

- Contract (landlord, tenant, version, state, abi) - this table denotes the contract with landlord, tenant attributes, the version denotes the version of the contract which may be different if it is a modified contract and the state denotes the state of the contract, which may be one from active, inactive and terminated. The active state is the one when the version of the contract between the stakeholders is executing and is active. The passive state is when the current version is no longer active and a modified version of the contract is inactive state and terminated state is when the contract has ended. There is an attribute abi which is application binary interface, and this allows the application to interact with the smart contract deployed over the blockchain.
- User(name, email, password, public key) - denotes the user of the application. Public key is used to fetch the balance of the user from the blockchain and also to make a user specific dashboard.

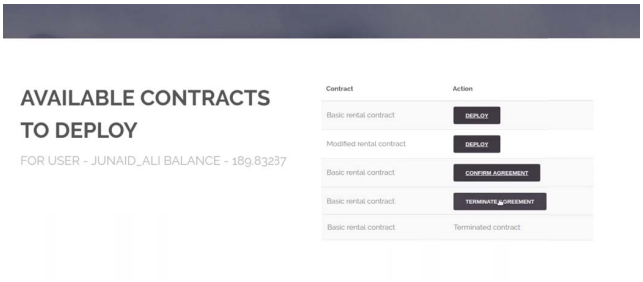


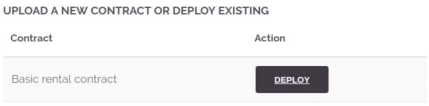
Fig. 7. Dashboard of User for Evolving Rental Agreement Manager Web Interface

1) Dashboard: The dashboard in Figure 7 shows a user logged in who can upload a new contract, deploy an uploaded contract, terminate a contract for which the logged-in user is the landlord of, and confirm the agreement of a contract that some other user is a landlord of. The snippet in Figure 8 shows the code that get executed to deploy a smart contract and execute the transactions on the already deployed smart contract.

```python
def execute_transaction(account,contract,function,*args,**kwargs):
    txn = contract.get_function_by_name(function)(*args).buildTransaction(kwargs)
    print("built transaction")
    signed = account.signTransaction(txn)
    print("signed transaction")
    tx_hash = w3.eth.sendRawTransaction(signed.rawTransaction)
    print("sent transaction")
    tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
    return tx_receipt
    # except Exception:
    #     print("Some error occured while executing transaction!!")
    # Wait for transaction to be mined...

def deploy_contract(account, contract_interface, *args, **kwargs):
    # Instantiate and deploy contract
    # acct = w3.eth.account.privateKeyToAccount(privateKey)
    contract = w3.eth.contract(
        abi=contract_interface['abi'],
        bytecode=contract_interface['bin']['object']
    )
    # Get transaction hash from deployed contract
    txn = contract.constructor(*args).buildTransaction(kwargs)
    signed = account.signTransaction(txn)
    # Get tx receipt to get contract address
    tx_hash = w3.eth.sendRawTransaction(signed.rawTransaction)
    tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
    return tx_receipt['contractAddress']
```

Fig. 8. Snippet for deploying contract and executing smart contract



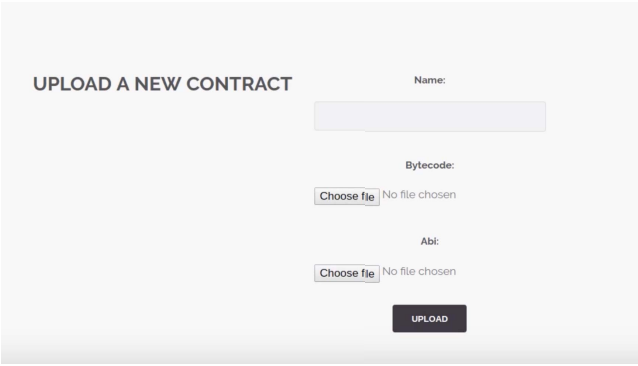


Fig. 9. Upload Contract Web Interface for landlord

2) Upload contract: Figure 9 shows how to upload a smart contract. The figure shows it requires an ABI file and the bytecode of the smart contract to upload the contract on the application. Once uploaded, this contract can be deployed on the blockchain.

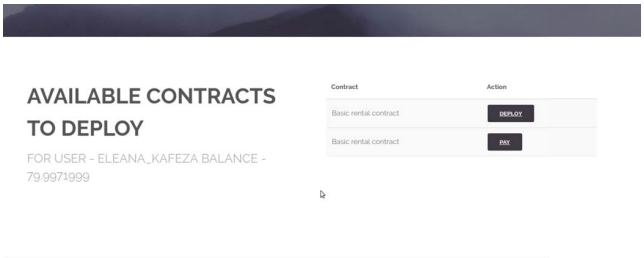


Fig. 10. Deploying Contract Web Interface for Landlord

3) Deploy contract: Figure 10 shows the functionality that allows the logged-in user to deploy an uploaded contract. Once the contract is deployed over the blockchain, the application can interact with the smart contract to execute the code in the smart contract, which is the execution logic of the smart contract. The execution logic encapsulates the legal clauses that need to be followed during the execution of a smart contract.

4) Confirm Agreement or Pay Rent: The application allows a tenant to pay rent to the landlord of the smart contract, for which this user confirmed the agreement. Once the rent is paid by the tenant, the ether (Ethereum currency) gets debited from the account of the tenant and gets credited in the account of the landlord.

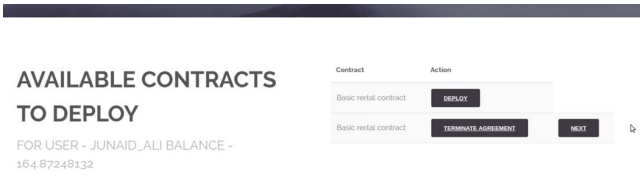


Fig. 11. Terminate or Modify Contract Web Interface for landlord

5) Terminate or Modify Contract: Figure 11 shows the functionality that allows the landlord to terminate contract or modify contract.

The application allows a user to modify a smart contract by taking a new smart contract, which is the modified version of the old smart contract and deploys the new contract. The application also links the old smart contract with the new one, so that all the previous transactions between the tenant and the landlord stay intact and the tenant can see the previous contracts.

The contract gets terminated when either the period of contract gets over or the tenant requests to terminate the contract before the agreed period. In case the contract is terminated by the tenant, half of the deposit is taken as a penalty from the tenant's account. The code to handle the case of timely or untimely termination is taken care of by the solidity code.

V. Conclusion

In this work, we examine the legal smart contract in the blockchain. We initially identify a set of problems that arise when adopting blockchain as a technology to implement legal agreements automation. Similar approaches can be followed in other applications as well. We present an architecture that can be followed when smart contract applications are developed that clarifies the modules, the communication and the technology to be used to design and implement the different parts of the smart contracts. Moreover, we propose a versioning mechanism using linked lists for the modification of smart contracts as well as we show how smart contracts can be used as a tool for data and logic separation. We present a case study of a rental agreement and we explain how such an agreement can be designed and executed in the blockchain. We propose the introduction of a platform for creating and handling smart contracts that can be responsible for the business logic and the use of blockchain for verifying transparency and security. We have implemented a system and presented our results. For the future work, we are examining to use more sophisticated techniques for implementing the versioning where the already executed part of the contract will not be able to change, as well as to increase the level of available modification for the user. We are also considering several other aspects for introducing trust to the system.

References

[1] I. Kafeza, E. Kafeza, and D. K. Chiu, "Legal issues in agents for electronic contracting," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. IEEE, 2005, pp. 134a–134a.

[2] N. Szabo, "Formalizing and securing relationships on public networks." 1997.

[3] D. K. Chiu, K. Karlapalem, Q. Li, and E. Kafeza, "Workflow view based e-contracts in a cross-organizational e-services environment," *Distributed and parallel databases*, vol. 12, no. 2-3, pp. 193–216, 2002.

[4] K. E. Levy, "Book-smart, not street-smart: blockchain-based smart contracts and the social workings of law," *Engaging Science, Technology, and Society*, vol. 3, pp. 1–15, 2017.

[5] M. Raskin, "The law of smart contracts," *SSRN Electronic Journal*, 2016.

[6] M. Alharby, A. Aldweesh, and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study of academic research (2018)," in *Proceedings of the 2018 International Conference on Cloud Computing, Big Data and Blockchain*, 2018.

[7] D. Macrinici, C. Cartofeanu, and S. Gao, "Smart contract applications within blockchain technology: A systematic mapping study," *Telematics and Informatics*, vol. 35, no. 8, pp. 2337–2354, 2018.

[8] R. M. Parizi, A. Dehghantanha *et al.*, "Smart contract programming languages on blockchains: An empirical evaluation of usability and security," in *International Conference on Blockchain*. Springer, 2018, pp. 75–91.

[9] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu, "On legal contracts, imperative and declarative smart contracts, and blockchain systems," *Artificial Intelligence and Law*, vol. 26, no. 4, pp. 377–409, 2018.

[10] A. Savelyev, "Contract law 2.0:'smart'contracts as the beginning of the end of classic contract law," *Information & Communications Technology Law*, vol. 26, no. 2, pp. 116–134, 2017.

[11] K. Lauslahti, J. Mattila, and T. Seppala, "Smart contracts–how will blockchain technology affect contractual practices?" 2017.

[12] P. De Filippi and S. Hassan, "Blockchain technology as a regulatory technology: From code is law to law is code," *arXiv preprint arXiv:1801.02507*, 2018.

[13] S. A. Abeyratne and R. P. Monfared, "Blockchain ready manufacturing supply chain using distributed ledger," 2016.

[14] E. Tjong Tjin Tai, "Formalizing contract law for smart contracts," 2017.