Implementation of Smart Contracts Using Hybrid Architectures with On and Off–Blockchain Components

Carlos Molina-Jimenez
Computer Laboratory
University of Cambridge, UK
carlos.molina@cl.cam.ac.uk

Ioannis Sfyrakis
School of Computing
Newcastle University, UK
ioannis.sfyrakis@ncl.ac.uk

Ellis Solaiman
School of Computing
Newcastle University, UK
ellis.solaiman@ncl.ac.uk

Irene Ng
Hat Community Foundation
Cambridge, UK
irene.ng@hatcommunity.org

Meng Weng Wong
CodeX, Stanford University
mengwong@stanford.edu
Legalese.com
mengwong@legalese.com

Alexis Chun
Visiting Fellow,
Singapore Management University
Legalese.com
alexis@legalese.com

Jon Crowcroft
Computer Laboratory
University of Cambridge, UK
jon.crowcroft@cl.cam.ac.uk

***Abstract*—Decentralised (on-blockchain) and centralised (off–blockchain) platforms are available for the implementation of smart contracts. However, none of the two alternatives can individually provide the services and quality of services (QoS) imposed on smart contracts involved in a large class of applications. The reason is that blockchain platforms suffer from scalability, performance, transaction costs and other limitations. Likewise, off–blockchain platforms are afflicted by drawbacks emerging from their dependence on single trusted third parties. We argue that in several applications, hybrid platforms composed from the integration of on and off–blockchain platforms are more adequate. Developers that informatively choose between the three alternatives are likely to implement smart contracts that deliver the expected QoS. Hybrid architectures are largely unexplored. To help cover the gap and as a proof of concept, in this paper we discuss the implementation of smart contracts on hybrid architectures. We show how a smart contract can be split and executed partially on an off–blockchain contract compliance checker and partially on the rinkeby ethereum network. To test the solution, we expose it to sequences of contractual operations generated mechanically by a contract validator tool.**

I. Introduction

This paper focuses on scenarios involving two or more commercial parties (as opposed to consumer or governmental entities) interacting digitally with each other. The parties are in a relationship regulated by some computer-readable and computer-executable formal specification that stipulates the operational aspects of the parties' business with each other. If the specification was written in natural language and signed by the parties, it would be considered a legal contract enforceable by a court. But since the specification is written in a formal language and executed digitally by the parties, it is realised primarily by technological means which obviate the need for conventional offline enforcement. As a running example, this paper uses a simple contract example based on selling of personal data. The example is translated from natural to formal language and executed as a smart contract.

A **smart contract** is an executable program (written in some programming language like, Java, C++, Solidity, Go, etc.) that is deployed to mediate contractual interactions between two or more parties. Its task is to detect (and if possible prevent) deviations from the agreed upon behaviour. To perform its task, the smart contract i) intercepts each business event generated by the parties, ii) analyses it to determine if it is contract compliant or not, iii) produces a verdict, and iv) records the outcome in an indelible log that is available for verification, for example, to sort out disputes. Notice that in some applications, the declaration of the verdict is directly and intricately associated to an action (e.g., collect payment) that is executed when the verdict is positive. In this paper we separate the two acts and focus on the verdict—the essence of smart contracts. We regard the action as an arbitrary reaction that can be immediately or eventually executed by the smart contract or by a different component.

We regard a smart contract as a piece of middleware expected to deliver a service with some QoS. Examples of QoS are: trust (who can be trusted with the deployment of the smart contract), transparency (can the contracting and third parties verify the verdicts), throughput (the number of operation that the smart contract can verify per second), response time (the time it takes to output a verdict), transaction fees (the monetary cost that the parties pay to the smart contract for processing each operation). Different applications (for example, a buyer–seller contract, property renting contract, etc.) will demand different QoS. The question that raises here is what architecture and technology to use to implement smart contracts that satisfy QoS requirements. Notice that in this paper we use the terms smart contracts and contract as synonymous.

Centralised (off–blockchain) and decentralised (on-blockchain) platforms are available for the implementation of smart contracts. However, we argue that some applications demand some QoS that can hardly be satisfied by none of the two alternatives individually.

Leading blockchain platforms Bitcoin [1], [2] and ethereum [3] are known to exhibit serious QoS limitations. For example, Bitcoin can only process 7 transaction per second—a poor throughput compared to visa's 2000 transaction per second [4]. Take another example, it takes Bitcoin about 10 min to publish a transaction in its block. off–blockchain platforms for smart contract implementation became available long before the Satoshi's seminal paper that launched Bitcoin [5]. Notable examples are [6]–[9]. However, their inherent centralisation prevents them from meeting the QoS requirentes demanded by some applications. For example, they cannot be used when the contracting parties are reluctant to place trust on trusted third parties (TTP).

The central argument of this paper is that in several applications, hybrid platforms composed from the integration of off and on–blockchain platforms are more adequate [10]. Unfortunately, the use of hybrid architectures in smart contract implementations is largely unexplored. This papers aims at helping covering the research gap.

The main contribution is the implementation of a smart contract on a hybrid architecture. At this stage we aim at proving the concept rather than at evaluating performance. We show how a smart contract can be split and executed partially on an off–blockchain contract compliance checker and partially on the rinkeby ethereum network. To test the solution, we expose it to sequences of contractual operations generated mechanically by a contract validator tool.

We continue the discussion as follows: In Section II we discuss a motivating example of a contract that we use as running example in this paper. We explore different approaches to smart contract implementation in Section III. The implementation of the the hybrid architecture is discussed in Section IV. In Section V raise some research questions that need attention. In Section VI we discuss work related to ours. In Section VII, we present some concluding remarks.

II. Motivating scenario

Let us take the example of an individual (e.g., Alice) interested in selling personal data that she has aggregated from different sources (domestic sensors, social networks, shopping, etc.) and stored in a repository, as envisioned in the HAT project [11]. Let us imagine that a data buyer (e.g., Bob) has agreed with Alice on buying the data under a contract that includes the following clauses. In the contracts Bob and Alice play the roles of the buyer and store, respectively.

1) *The buyer has the* ***right*** *to place a* ***buy request*** *with the store to buy an item.*
2) *The store has the* ***obligation*** *to respond with either* ***confirmation*** *or* ***rejection*** *within 3 days of receiving the request.*
    a) *No response from the store within 3 days will be treated as a rejection.*
3) *The buyer has the* ***obligation*** *to either* ***pay*** *or* ***cancel*** *the request within 7 days of receiving a confirmation.*
    a) *No response from the buyer within 7 days will be treated as a cancellation.*
4) *The buyer has the* ***right*** *to* ***get a voucher*** *from the store, withing 5 days of submitting payment.*

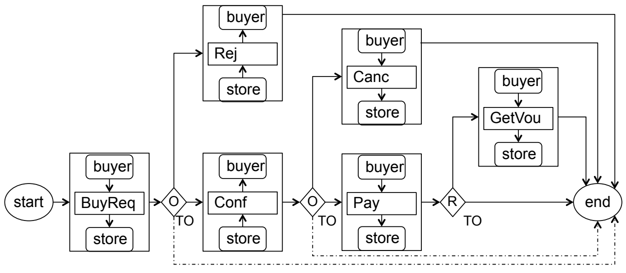


Fig. 1. A contract between a buyer and store for trading personal data.

The clauses include contractual operations (for example, **buy request**, **reject** and **confirmation**) that the parties have the right or obligation to execute under strict time constrains. Though the clauses are relatively simple, they are realistic enough to illustrate our arguments.

III. Implementations alternatives

The contract written in English can be converted into a formal notation such as the graphical view shown in Fig. 1, and from there into smart contract code.

The operations written originally in English have been mapped to messages sent by a party to its counterpart. For example, the **BuyReq** message sent by the buyer to the store corresponds to the execution of the operation **buy request** by the buyer. The diamonds represent exclusive splits in the execution and have been labeled with O (Obligation) and R (Right). TO stands for Time Out and represents contractual deadlines. Failure to execute and obligatory operations results in abnormal contract end (represented by dashed lines) with disputes to be sorted off line.

Fig. 1 can be regarded as a FSM modelled and implemented in a Turing complete language. There are several architectures and technologies that can be used [10]:

- Centralised: The smart contract is deployed on a TTP. Since there is no blockchain involved, this approach is also known as off–blockchain.
- Descentralised: The smart contract is deployed on a blockchain platform, for example on ethereum. This approach is also known as on–blockchain.
- Hybrid: The contract is split and deployed partially off and on–blockchain, i.e., some clauses are enforced off–blockchain whereas others are enforced on–blockchain.

IV. Implementation of a hybrid architecture

Motivated by the arguments presented in [10] about the potential of hybrid architectures in meeting certain QoS, we

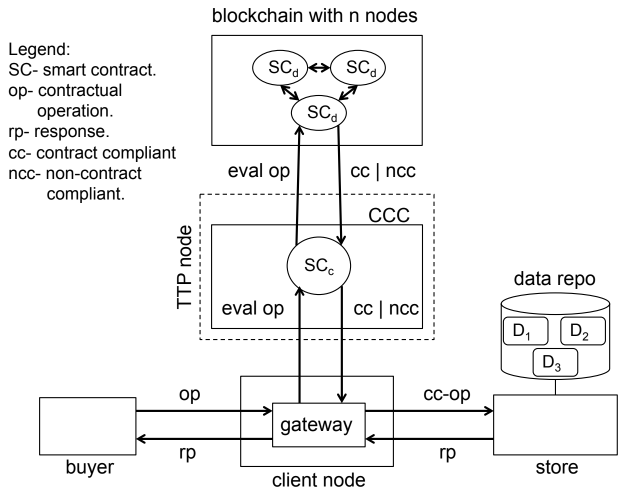


Fig. 2. A hybrid architecture for smart contracts: conceptual view.

have decided to implement one and use it to run the contract example. The buyer and store represent the buyer and store of the contract example. D_1, D_2 and D_3 are pieces of personal data that the store is selling to the buyer.

Let us assume from here on that the buyer and store of the contract example, have agreed on using a hybrid architecture where the operation `Pay` is to be enforced by the on–blockchain component of the smart contract and the rest by the off–blockchain component.

The interaction between the on and off–blockchain components can can be based on several models such as peer–to–peer and master–slave [12]. In pursuit of simplicity, in this paper we follow a master–slave interaction model where the off-blockchain smart contract is the master.

A conceptual view of the hybrid architecture shown in Fig. 2. It can be implemented on the basis of several alternative technologies. In this work, we use the ethereum blockchain [3] to deploy the decentralised smart contract (SC_d) component and the Contract Compliance Cheker (CCC) developed by University of Newcastle [13] to deploy the centralised smart contract (SC_c) component.

A. Contract compliant checker

We use the contract compliance checker [9], [13] because it offers several features that can ease integration with a blockchain platform. The CCC is an open source tool designed for the enforcing of smart contracts. It is a Java application composed of several files, RESTful interfaces, and a database. At its core lies a FSM that grants and removes **rights**, **obligations** and **prohibitions** to the contracting parties as the execution of the contract progresses. To enforce a smart contract with the CCC, the developer (i) writes the contract in the Drools language and stores it in a *.drl* file (for example *dataseller.drl*), (ii) loads (copies) the *drl* file into the *configuration/drools/upload* folder, and (iii) deploys and instantiates the CCC as a web server (for example on a TTP node) that waits for the arrival of events representing the contractual operation. An **event** is a notification about the execution of a contractual operation by a contractual partner. For example when the buyer of Fig 1, executes the operation *BuyReq* the event *BuyReq* is generated by the buyer's application and sent to the CCC for evaluation.

Drools is a declarative Turing complete language designed for writing business rules [14]. The contract loaded to the CCC is capable of evaluating contractual operations issued by business partners as RESTful requests against its rules. Rules respond with RESTful responses that can be the outcome of an evaluation of an operation (*contract compliant* or *non contract compliant*) or an arbitrary message such as a request to execute an operation on a blockchain.

B. Client node

The client node is an ordinary node and not necessarily the same as the TTP shown in the figure. It is responsible for hosting the *gateway*. Contractual operations (*op*) are initiated by the business partners, such as *BuyReq*, and *Pay*. The SC_c contract determines if a given operation is contract compliant (*cc*) or non contract compliant (*ncc*). The SC_c is in control of the *gateway* that grants access to the store's data. For example, when the buyer wishes to access the stores's data, the buyer i) issues the corresponding operation against the gateway, ii) the gateway forwards the operation to the SC_c, iii) the SC_c evaluates the operation in accordance with its business rules that encode the contractual clauses and responds with either *cc* or *ncc* to open or close the gateway, respectively, iv) the opening of the gateway allows the buyer's operation to reach the data repository and retrieve the response (*rp*) that travels to the buyer. Notice that, to keep the figure simple, the arrows show only the direction followed by operations initiated by the buyer. Operations initiated by the store follow a similar procedure but right to left.

C. Ethereum

We have chosen the ethereum platform [3] for implementing the decentralised contract enforcer for the following reasons: It is currently one of the most mature blockchains. It supports Solidity—a Turing–complete language that designers can use for encoding stateful smart contracts of arbitrary complexity. For complex contracts, ethereum is more convenient than Bitcoin which supports only an opcode stack-based script language. In addition, ethereum offers developers on line compilers of Solidity code [15]. Equally importantly, ethereum provides, in addition to the main ethereum network (Mainnet), four experimental networks (Ropsten, Kovan, Sokol and Rinkeby) that developers can use for experimenting with their ideas using ethereum tokens instead of real ether money [16], [17]. We run our experiments in Rinkeby as it is the most stable. To pay for transactions we used ether tokens requested from faucet [18].

D. Execution sequences for testing the hybrid architecture

A particularity of smart contracts deployed on–blockchain is that because of their descentralisation and openess, they are hard to amend after deployment. Therefore, we suggest that smart contracts are thoroughly validated (for example, using conventional model checking tools) to uncover potential logical inconsistencies of their clauses (omissions, contradictions, duplications, etc.). In addition, we suggest that the actual implementation is systematically tested before deployment.

In hybrid architectures the risk of implementing buggy is exacerbated by the interaction between the on and off–blockchain components. For instance, besides its simplicity, the master–slave interaction model that we use in the architecture already exhibits intricate behaviour that demands systematic scrutiny to uncover potential inconsistencies. To face the challenge, in this paper we use *contraval*—a contract validator that we have developed specifically for model checking and testing contracts [19], [20]. It is based on the standard Promela language and the Spin model checker. It supports the epromela (an extension of Promela) that provides constructs for intuitively expressing and manipulating contractual concepts such rights, obligations and role players.

We use *contraval* for model checking the contract example and, more importantly, for generating the execution sequences that we use for testing the hybrid architecture of Fig. 4. We define an **execution sequence** (or sequences for brevity) as a set of one or more contractual operations that the contractual parties need to execute to progress the smart contract from the *start* to the *end*.

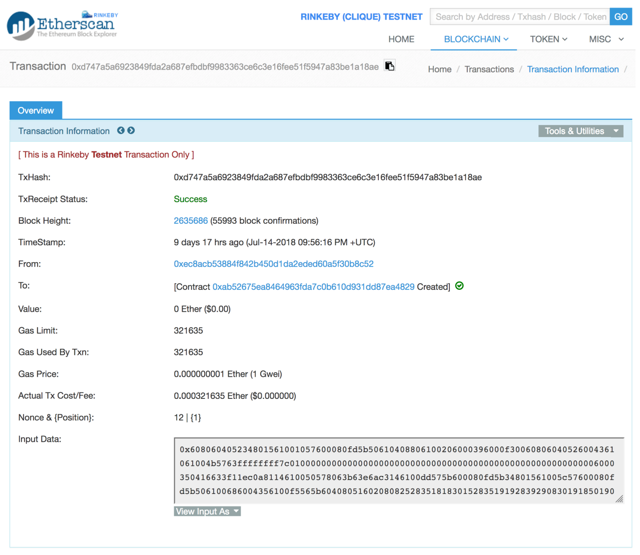
To generate the sequences we proceeded as follows:

1) We converted the clauses of the contract example into a formal model written in epromela. We arbitrarily called it *dataseller.pml*.
2) We model checked the contractual model with Spin to verify conventional correctness requirements (deadlocks, missing messages, etc.) and typical contractual issues (clause duplications, omissions, etc.).
3) We exposed the model checked model augmented with a LTL formula to Spin and instructed Spin to produce counter examples containing the sequences of interest.
4) We run a Python parser (called *parser-filtering.py* that we have implemented, over the Spin counter examples to extract the execution sequences.

A close look at Fig. 1 will reveal that it encodes six alternative paths (sequences) from *start* to *end*.

```
// Execution sequences encoded in Fig 1.
// RejConfTo=Rej or Conf timeout,
// PayCancTo=Pay or Canc timeout
 seq1: {BuyReq, Rej}
 seq2: {BuyReq, Conf, Canc}
 seq3: {BuyReq, Conf, Pay}
 seq4: {BuyReq, RejConfTo}
 seq5: {BuyReq, Conf, PayCancTo}
 seq6: {BuyReq, Conf, Pay, GetVou}
```

Seq1, Seq2 and Seq3 result in normal contract completion. However, Seq4 and Seq5 result in abnormal contract completion. In Seq4 the store fails to meet it obligation (execute either *Rej* or *Conf*) before the 3 day deadline. Similarly, in Seq5, the buyer fails to execute either *Pay* or *Canc* before the 7 day deadline. Observe that although the buyer's has a deadline of 5 days to claim a voucher, failure to execute


Etherscan
RINKEBY
The Ethereum Block Explorer
RINKEBY (CLIQUE) TESTNET
Search by Address / Txhash / Block / Token
GO
HOME
BLOCKCHAIN
TOKEN
MISC
Home / Transactions / Transaction Information /
Overview
Tools & Utilities
View Input As

Transaction 0xd747a5a6923849fda2a687efbdbf9983363ce6c3e16fee51f5947a83be1a18ae

Transaction Information

[This is a Rinkeby **Testnet** Transaction Only]

TxHash:	0xd747a5a6923849fda2a687efbdbf9983363ce6c3e16fee51f5947a83be1a18ae
TxReceipt Status:	Success
Block Height:	2635686 (55993 block confirmations)
TimeStamp:	9 days 17 hrs ago (Jul-14-2018 09:56:16 PM +UTC)
From:	0xec8acb53884f842b450d1da2eded60a5f30b8c52
To:	[Contract 0xab52675ea8464963fda7c0b610d931dd87ea4829 Created] ✓
Value:	0 Ether ($0.00)
Gas Limit:	321635
Gas Used By Txn:	321635
Gas Price:	0.000000001 Ether (1 Gwei)
Actual Tx Cost/Fee:	0.000321635 Ether ($0.000000)
Nonce & {Position}:	12 | {1}
Input Data:	

```
0x608060405234801561001057600080fd5b506104088061002060003960 00f30060806040526004361
061004b5763ffffffff7c0100000000000000000000000000000000000000000000000000000000006000
350416633f11ec0a8114610050578063b63e6ac3146100dd575b600080fd5b34801561005c57600080f
d5b506100686004356100f5565b604080516020808252835181830152835191928392908301918501 90
```

Fig. 3. Transaction that deployed the collectPayment.sol contract.

`GetVou` does not result in abnormal contract completion because `GetVou` is a right, rather than an obligation. Seq6 is particularly problematic. It will be analised separately in Section IV-F.

To ease sequence manipulation, we have programmed the Python parser to store the sequences in a folder with N subfolders where each of them represents an sequence. Each subfolder contains M files (one for each event). For instance subfolder *dataseller.pml.ExecSeq1* is related to Seq1 and contains two files: `event1.xml` and `event2.xml` which store messages, `BuyReq` and `Rej`, respectively.

E. Smart contracts code and deployment

Fig. 4 shows the technology that we have used in the implementation of the architecture. We have split the contract example into two parts: *dataseller.drl* and *collectPay.sol*.

dataseller.drl: corresponds to the SC_c and is encoded in drools. We deployed it on a Mac computer (regarded as a TTP) as a web server as explained in Section IV-A. On the Mac we also deployed the ethereum client shown in the figure.

collectPayment.sol corresponds to the SC_d and is encoded in Solidity. There are several alternatives such as the `web3j` library to deploy the `collectPayment.sol` contract. For simplicity, we opted for metamask [21]: a plugin that allows developers to perform operations against Ethereum applications (including contract deployment) from their browsers, without deploying a full geth Ethereum node. We deployed metamask on Firefox and, before instantiating the CCC, we executed the transaction shown in Fig. 3 to deploy the `collectPayment.sol` contract on the Rinkeby test network [22].

The following code contains two rules extracted from the *dataseller.drl* contract.

```
#dataseller.drl contract in drools
rule "Payment Received"
```

```
# Grants buyer the right to get a voucher when
# the buyer's obligation to pay is fulfilled.
   when
     $e: Event(type=="PAY", originator=="buyer",
     responder=="store", status=="success")
     eval(ropBuyer.matchesObligations(payment))
   then // Remove buyer's oblig to pay or cancel
     ropBuyer.removeObligation(payment, seller);
     ropBuyer.removeRight(cancelation, seller);
     bcEvent.submitPayment();//forward pay to ethe contr.
     ropBuyer.addRight(voucher,seller,0,0,120);//5 days
     CCCLogger.logTrace("* Payment result received -
     add right to GetVoucher ");
     CCCLogger.logTrace("* Payment rule triggered");
     responder.setContractCompliant(true);
end

rule "Get Voucher"
# Grants a voucher to buyer if the buyer has the right
# ('cos it has fulfilled his payment oblig) to get it.
# It removes buyer's right to get a voucher after given
# it to him or 5 days expiry.
   when
     $e: Event(type=="GETVOU", originator=="buyer",
     responder=="store", status=="success")
     eval(ropBuyer.matchesRights(voucher))
   then
     ropBuyer.removeRight(voucher, seller);
     bcEvent.getVoucher();
     CCCLogger.logTrace("* Get Voucher rule triggered");
     responder.setContractCompliant(true);
end
```

The following code is the *collectPayment.sol* contract.

```
///collectPayment.sol contract in Solidity
pragma solidity ^0.4.4;
contract collectPayment{
...

function submitPayment(uint pay) public constant
returns (string) {
/// func to submit payment. Returns:
/// "Payment received " + pay converted into str
 var s=uint2str(pay);
 var new_str=s.toSlice().concat("Received".toSlice());
 return new_str;
}

function getReceipt(uint trasactionNum) public constant
returns (string) {
/// func to get a receipt of a given Tx.
/// returns: "Receipt 4 Tx " + transactionNum
/// converted into str
 var s=uint2str(trasactionNum);
 var new_str="Receipt 4 Tx".toSlice().concat(s.toSlice());
 return new_str;
 }
}
```

We stress that since our focus at this stage is to demonstrate the building of the hybrid architecture, the *collectPayment.sol* contract is simple, it only receives string messages (instead of ethereum tokens or actual ethereum currency) from the *dataseller.drl* contract and replies with another string message.

The *client* corresponds to the *client node* of Fig. 2 and acts as a web client to the *dataseller.drl* contract. We use it to test the implementation of the contract example implemented by the combination of *dataseller.drl* and *collectPayment.sol*. In this order, we provide the client with all the sequences encoded in the contract example and previously generated by the *contraval* tool and stored in folders (see Section IV-D).

As shown in the figure, the CCC relies on the web3j library [23] to communicate with the ethereum client. Among other services, the web3j library includes a command line application that mechanically generates wrapper code from

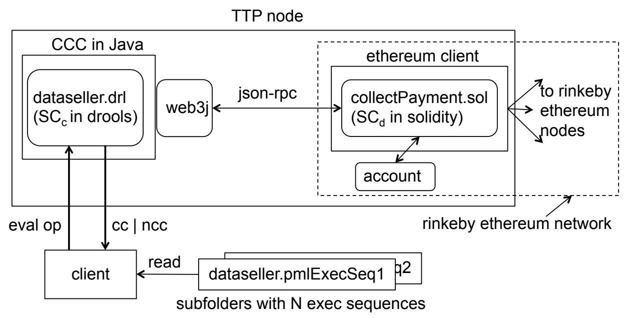


Fig. 4. Hybrid architecture for smart contract: technology view.

a smart contract specified using Solidity and compiled using the solc compiler. The CCC (a Java application) can use the generated wrapper code to communicate with the `collectPayment.sol` contract, through the json–rpc API provided by ethereum. In addition, the web3j library provides an API to for the CCC to unlock an ethereum client account by providing the path to the keystore file and the password.

In our implementation, the communication facilities provided by web3j are used by the `bcEvent.submitPayment()` statement of the `dataseller.drl` contract (line 12) to forward the `Pay` operation to the `collectPayment.sol` contract. The statement calls the `submitPayment` function of the `collectPayment.sol` contract. The aim of this example is to demonstrate how the `dataseller.drl` and `collectPayment.sol` contracts can communicate with each other. Another example of communication is `bcEvent.getVoucher()` of the `Get Voucher` rule. As it is, this statement calls the `getReceipt` function of the `collectPayment.sol` contract to receive a string. However, it can be replaced by a function that returns actual ethers representing voucher for the buyer, or by any other function. Naturally, more functions that interact with more rules from the `dataseller.drl` contract can be included to compose a practical application.

F. Determination of contract compliance

Let us examine the procedures followed by the *dataseller.drl* and *collectPay.sol* contracts to process the operations included in the contract example. Let us start with execution sequences that do not include the `Pay` operation.

1) We assume that the set of the N execution sequences to test the architecture are already available from a local folder.
2) We load the CCC with the *dataseller.drl* contract and instantiate it to listen for incoming events.
3) We instantiate the client. In response, it proceeds to read the $dataseller.pmlExeSeq_1$ folder to extract its sequence: `BuyReq, Rej`. Next the client sends the `BuyReq` event to the *dataseller.drl* contract formatted as a RESTful message.

4) The `BuyReq` event triggers a rule of the *dataseller.drl* contract that determines if the corresponding `BuyReq` operation was contract compliant or non–contract compliant. The *dataseller.drl* contract sends its verdict back to the client.
5) The above procedure is repeated with the next event (`Rej`) of the sequence.
6) When the client sends the last event of the sequence, it proceeds to the $dataseller.pmlExeSeq_2$, followed by the $dataseller.pmlExeSeq_3$ and so on till $dataseller.pmlExeSeq_N$. Since all the sequences are legal, the *dataseller.drl* contract declares each event of each sequence to be contract compliant.
   However, the procedure changes when the *dataseller.drl* is presented with an event that is meant to be processed by the ethereum *collectPay.sol* contract, such as the *Pay* operation in the contract example. Let us discuss this situation separately.

The sequence `seq: BuyReq, Conf, Pay, GetVou` which includes the `Pay` operation is more problematic because it involves the `collectPayment.sol` contract. The *dataseller.drl* erratically declares *GetVou* either contract complince or non contract compliance. `BuyReq` and `Conf` are processed by the client and *dataseller.drl* contract as above. However, when the *dataseller.drl* receives the `Pay` event, the rule `Payment Received` (see the *dataseller.drl* code) does not evaluate it immediately for contract compliance but performs the following actions:

1) It creates a blockchain event object that the `dataseller.drl` contract uses for interacting with the `collectPayment.sol` contract.
2) It uses the wrapper code (provided by the web3j library) to call the `submitPayment` function of the `collectPayment.sol` contract by means of a json–rpc message. Basically, the message forwards the `Pay` operation from the `dataseller.drl` contract to the `collectPayment.sol` contract.
3) The result of the call to the `submitPayment` function is not necessarily notified immediately to the *dataseller.drl* contract. Consequently, two situations can develop:
   - **a) Pay confirmation precedes GetVou:** The `dataseller.drl` contract receives pay confirmation and grants the buyer the right to get a voucher. Consequently, when the `dataseller.drl` eventually receives the `GetVou` event from the buyer, the operation is declared contract compliant and the voucher is granted.
   - **b) GetVou precedes pay confirmation:** This situation might happen if we assume that the pay confirmation might take arbitrarily long. Because of this, the `dataseller.drl` contract receives the `GetVou` event from the buyer before receiving pay conformation from the `collectPayment.sol` contract. Consequently, the `dataseller.drl` contract declares `GetVou` non–contract compliance— as far as the `dataseller.drl` contract is aware of, the buyer does not have the right to get a voucher.

The materialization of situation `b)` is shown in the outputs produced by the client when it presents the `BuyReq, Conf, Pay, GetVou` to the `dataseller.drl` contract. The text has been slightly edited for readability. As shown by the `false` output of the third last line, in this execution the `dataseller.drl` contract declares the `GetVou` operation non contract compliant. The output produced by situation `a)` is is not shown but is similar, except that the third last line shows `true` instead of `false`.

```
/* b) In this run of the seq BuyReq, Conf, Pay, GetVou
 * the dataseller.drl contract declares GetVou operation
 * contract compliant: false
 */
-------- Begin Request to CCC service ----------
BusinessEvent{type='BuyReq'}
-------- End Request to CCC service ----------
-------- Begin Response from CCC service ----------
 <contractCompliant>true</contractCompliant>
 -------- End Response from CCC service ----------
 -------- Begin Request to CCC service ----------

BusinessEvent{type='Conf'}
-------- End Request to CCC service ----------
-------- Begin Response from CCC service ----------
 <contractCompliant>true</contractCompliant>
 -------- End Response from CCC service ----------

BusinessEvent{type='Pay'}
-------- End Request to CCC service ----------
-------- Begin Response from CCC service ----------
 <contractCompliant>true</contractCompliant>
 -------- End Response from CCC service ----------

 BusinessEvent{type='GetVou'}
-------- End Request to CCC service ----------
-------- Begin Response from CCC service ----------
 <contractCompliant>false</contractCompliant>
 -------- End Response from CCC service ----------
```

We stress that the problematic situation emerges from a legal sequence. The potential existence of illegal sequences such as those that include `GetVou` not preceded by `Pay` can be uncovered by model checking. This situation suggests that model cheking is not enough. The error that we are analyzing materialised at run time because of the interaction (about pay confirmation) between the `dataseller.drl` and `collectPayment.sol` contracts. In this work, we uncover it at testing time.

One can also argue that there are simple mechanisms to prevent the occurrence of the problematic situation (for example, queue the `GetVou` event) and to resolve it (for example, the buyer retries the `GetVou` operation until it is eventually declared contract compliant by the `dataseller.drl` contract. These are valid solutions, however, our main observation is that this is only an example of a large class of situations that might impact hybrid contracts unless adequate measures are taken to uncover them at design and testing time.

G. *Code and repeatability of experiments*

The code for the implementation of Fig. 4 and generating sequences is available from the *conch* [13] and *contraval* [20] Git repositories, respectively.

V. Future research directions

We are only starting the exploration of hybrid implementations of smart contracts—our research is at proof of concept stage. To consolidate our ideas, we are planning to conduct performance evaluation of some QoS requirements to demonstrate that our architecture can meet them. We will use more demanding contracts. The exploration of different interaction models (e.g., peer–to–peer) between on- and off–blockchain components is also pending. To save space, the sequences that we have used do not account for operations that might fail to complete successfully, e.g., after failing to reach a counterpart. We are planning to cover this issue in a separate paper as suggested in [9], [12].

For the sake of readability the contract example that we have used is written in a *denormalised* form to correspond to popular intuitions about contract deontics [24]. However, this point deserves further exploration. It involves the analysis of the logics implicit in the English text of the contract as this logics impacts the implementation complexity and completeness of its smart contract equivalent. The issue is that there several ways of phrasing contractual clauses with subtle implications. For instance, prohibitions can be expressed as obligations.

Programmatic contract drafting up is another pending question. We are currently exploring Ricardian Contracts [25] to develop systems that fill, in parallel, templates for both formal–language contracts intended for digital execution (whether on or off–blockchain), and natural language versions of the contracts. These contracts in natural languages describe the same operational core but contain additional boilerplate and are intended to be signed on paper and legally binding. Natural language generation systems offer the potential for efficient production and filling of such dual contract templates. We will complement this research with formal verification of smart contracts.

VI. Related work

Research on smart contracts was pioneered by Minsky in the mid 80s [6] and followed by Marshall [7]. Though some of the contract tools exhibit some decentralised features [26], those systems took mainly centralised approaches. Within this category falls the contract compliance checker that we use in our implementation [9].

The publication of the Bitcoin paper [5] motivated the development of several platforms for supporting the implementation of decentralised smart contracts. Platforms in [1], [3] and [2] are some of the most representative. Though they differ in language expression power, fees and other features they are convenient for implementing decentralised smart contracts.

The hybrid approach that we suggest was inspired by the arguments presented in [10], though the original idea emerged by the off–blockchain payment channel [1], [27].

The concept of logic–based smart contracts discussed in [28] has some similarities with our hybrid approach. They suggest the use of logic–based languages in the implementation of smart contracts capable of performing on–blockchain and off–blockchain inference. The difficulty with this approach is lack of support of logic–based languages in current blockchain technologies. In our work, we rely on the native languages offered by the blockchain platforms, for example, ethereum's Solidity. On and off–blockchain enforcement of contractual operations is also discussed in [29], though an architecture is presented, no technical details about its implementation or functionality are discussed. Another conceptual design directly related to our work is private contracts executed in the Enigma [30] architecture. Like in our work, a private contract is a conventional business contract with contractual operations separated into on and off–blockchain categories. Similarly to our hybrid design, they use a blockchain platform (namely ethereum) to execute on–blockchain operations. However, unlike in our work, instead of using a TTP to execute off-blockchain operations, they use a set of distrusting Enigma nodes running a secure multi–party computation protocol that guarantees privacy. The Enigma nodes charge computation and storage fees.

The logical correctness of smart contracts is discussed in several papers. For instance, in [31] the author use PetriNets for validating the correctness of business process expressed in BPMN notation and executed as a smart contract in ethereum. They mechanically convert BPMN notation into PetriNets, verify soundness and safeness properties, optimise the resulting PetriNet and convert it mechanically into Solidity. Related to our work on Cartesian Contract is the system for programmatic analysis of contracts written in natural languages to extract contractual rights and obligations [32].

The idea of interconnecting smart contracts to enable them to collaborate with each other is also discussed in [33]. These authors draw a similarity between blockchain and the Internet. They speculate that in the future, we will have islands of blockchain systems interconnected by gateways.

VII. Concluding remarks

The aim of this paper has been to argue that there are good reasons to consider hybrid architectures composed of off and on–blockchain components as alternatives for the implementation of smart contracts with strict QoS requirements. As a proof of concept, we have demonstrated that hybrid architectures are implementable as long as the off–blockchain component provides standard APIs (such as RESTful) to communicate with the standard APIs that current blockchains offer such as json–rpc.

We have presented the approach as a pragmatic solution to the current problems that afflict both off and on–blockchain platforms. However, we believe that these ideas will become useful in the development of smart contract applications of the near future. We envision cross–smart contract applications that will involve several smart contracts running on different independent platforms. The architecture that we have implemented is in line with this futuristic view. Though the current implementation includes only two components, it can be generalised to include and arbitrary number of off–blockchain and on–blockchain components. This generalisation should be implementable provided that the components offer interfaces (gateways) to interact with each other and the developer devises mechanisms for coordinating their collaboration.

We have argued that the implementation of sound smart contracts is not trivial and that the inclusion of off and on–blockchain components makes the task even harder. To ease the task, we advise the use of software tools to mechanise the verification of the smart contract and testing of its implementation.

Acknowledgements

Carlos Molina-Jimenez is currently collaborating with the HAT Community Foundation, Grant RG90413 NRAG/536. Ioannis Sfyrakis was partly supported by EU Horizon2020 prismacloud.eu project, GA No.644962. Meng Weng Wong is a 2017–2018 Fellow at Stanford University's CodeX Center for Legal Informatics, and previously a 2016–2017 Fellow at Harvard University's Berkman Klein Center for Internet and Society, and a 2016 Fellow at Ca'Foscari University of Venice.

References

[1] A. Antonopoulos, *Mastering Bitcoin*, 2nd ed. O'Reilly, 2017.

[2] The Linux Foundation, "Hyperledger," www.hyperledger.org, Visited Nov 2017 2017.

[3] Ethereum, "A next-generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, Visited 23 Oct 2017 2017.

[4] T. McConaghy, R. Marques, A. Müller, D. D. Jonghe, T. T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto, "Bigchaindb: A scalable blockchain database," www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf, Visited 1 Nov 2017 2017.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://nakamotoinstitute.org/bitcoin/, Visited 13 Nov 2017 2008.

[6] N. H. Minsky and A. D. Lockman, "Ensuring integrity by adding obligations to privileges," in *Proc. 8th Int'l Conf. on Software Engineering*, 1985, pp. 92–102.

[7] L. F. Marshall, "Representing management policy using contract objects," in *Proc. IEEE First Int'l Workshop on Systems Management*, 1993, pp. 27–30.

[8] Z. Milosevic, A. Josang, T. Dimitrakos, and M. Patton, "Discretionary enforcement of electronic contracts," in *Proc. 6th IEEE Int'l Enterprise Distributed Object Computing Conf.(EDOC'02)*. IEEE CS Press, 2002, pp. 39–50.

[9] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "A model for checking contractual compliance of business interactions," *IEEE Trans. on Service Computing*, vol. PP, no. 99, 2011.

[10] C. Molina-Jimenez, E. Solaiman, I. Sfyrakis, I. Ng, and J. Crowcroft, "On and off-blockchain enforcement of smart contracts," in *Proc. Int'l Workshop on Future Perspective of Decentralized Applications (FPDAPP)*, 2018.

[11] "Hat: Hub-of-all-things," http://hubofallthings.com/home/, visited: 10 Feb 2016.

[12] C. Molina-Jimenez, I. Sfyrakis, E. Solaiman, I. Ng, and J. Crowcroft, "Implementation of smart contracts using on and off blockchain components (extended version)," Jul 2018, to appear in e-prints arxiv.org. Available from Research gate DOI: 10.13140/RG.2.2.34438.2720.

[13] C. Molina-Jimenez and I. Sfyrakis, "Deployment of the contract compliant checker: (user's guide)," https://github.com/carlos-molina/conch.git, Visited in Feb 2016.

[14] The JBoss Drools team, "Drools expert user guide," https://docs.jboss.org/drools/release/5.3.0.Final/drools-expert-docs/html/index.html, visited: 7 May 2018.

[15] Remix, "Remix solidity ide," https://remix.ethereum.org, Visited 17 Jun 2018 2017.

[16] Ethereum, "Ethereum: Comparison of the different testnets," https://ethereum.stackexchange.com/questions/27048/comparison-of-the-different-testnets, Visited 17 Jun 2018 2018.

[17] C. Svensson, "Transactions—webj 3.4.0 documentation," https://web3j.readthedocs.io/en/latest/transactions.html, Visited 17 Jul 2018 2018.

[18] Faucet, "Rinkeby authenticated faucet," https://faucet.rinkeby.io, Visited 30 Jul 2018 2018.

[19] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "On model checker based testing of electronic contracting systems," in *12th IEEE Int'l Conf. on Commerce and Enterprise Computing(CEC'10)*, 2010, pp. 88–95.

[20] C. Molina-Jimenez, "Deployment of contraval—a contract validator : (user's guide)," https://github.com/carlos-molina/contraval.git, 2012.

[21] Metamask support, "Metamask installation," https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn, Visited 24 Jul 2018 2018.

[22] "Collectpay.sol smart contract deployment transaction." https://rinkeby.etherscan.io/address/0xab52675ea8464963fda7c0b610d931dd87ea4829, Visited 24 July 2018 2018.

[23] C. Svensson, "web3j," https://web3j.readthedocs.io/en/latest/, Visited 17 Jul 2018 2018.

[24] T. Hvitved, "Contract formalisation and modular implementation of domain-specific languages," Ph.D. dissertation, Faculty of Science University of Copenhagen, Mar 2012.

[25] I. Grigg, "The ricardian contract," http://iang.org/papers/ricardian_contract.html, 2000, (Accessed on 07/26/2018).

[26] N. Minsky, "A model for the governance of federated healthcare information systems," in *IEEE Int'l Symposium on Policies for Distributed Systems and Networks (Policy'10)*, 2010, pp. 111–119.

[27] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," https://lightning.network/lightning-network-paper.pdf, Jan. 2016.

[28] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor, "Evaluation of logic–based smart contracts for blockchain systems," in *Proc. 10th Int'l Symposium RuleML'16: Rule Technologies: Research, Tools, and Applications, LNCS, Vol 9718*, 2018, pp. 167183,.

[29] X. Xu, C. Pautasso, V. Gramoli, and A. Ponomarev, "The blockchain as a software connector," in *Proc. 13th Working IEEE/IFIP Conf. on Software Architecture (WICSA). IEEE, apr 2016, pp. 182191*, 2016, pp. 182–191.

[30] G. Zyskind, O. Nathan, and A. S. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," https://arxiv.org/abs/1506.03471 (visitied in Mar 2018), Jan 2015, arXiv:1506.03471v1 [cs.CR].

[31] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber, "Optimized execution of business processes on blockchain," https://arxiv.org/pdf/1612.03152.pdf, Dec 2016, arXiv:1612.03152 [cs.SE].

[32] J. J. Camilleri, "Contracts and computation formal modelling and analysis for normative natural language," Ph.D. dissertation, Department of Computer Science and Engineering, 2017.

[33] T. Hardjono, A. Lipton, and A. Pentland, "Towards a design philosophy for interoperable blockchain systems," https://arxiv.org/pdf/1805.05934.pdf, May 2018, arXiv:1805.05934 [cs.CR].