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

(Extended Version: 27 Jul 2018)

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

Meng Weng Wong
CodeX Center, Stanford University
mengwong@stanford.edu
and Legalese Pte. Ltd., Singapore
mengwong@legalese.com

Alexis Chun
Singapore Management University
and Legalese Pte. Ltd., Singapore
alexis@legalese.com

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

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

***Abstract*—Recently, decentralised (on-blockchain) platforms have emerged to complement centralised (off-blockchain) platforms for the implementation of automated, digital ("smart") contracts. However, neither alternative can individually satisfy the requirements of a large class of applications. On-blockchain platforms suffer from scalability, performance, transaction costs and other limitations. Off-blockchain platforms are afflicted by drawbacks due to their dependence on single trusted third parties. We argue that in several application areas, hybrid platforms composed from the integration of on- and off-blockchain platforms are more able to support smart contracts that deliver the desired quality of service (QoS). Hybrid architectures are largely unexplored. To help cover the gap, in this paper we discuss the implementation of smart contracts on hybrid architectures. As a proof of concept, 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 investigates scenarios involving two or more commercial parties interacting digitally with each other in a relationship regulated by some computer-readable formal specification that details the operational aspects of the parties' business with each other. If this specification were written in natural language and signed on paper by the parties, it would be considered a traditional legal contract enforceable by a court. However, a specification written in a formal language, intended for digital execution and performance by the parties, constitutes a new breed of contract. The fact that such contracts may be executed, performed, and enforced by technology alone promises to largely obviate the need for "offline" court enforcement. Hence the appellation 'smart' contract.

Approaches to automated contract execution pre-date today's on-blockchain Bitcoin and Ethereum contracts. For decades, financial institutions have executed trades digitally; Nick Szabo [1] used the term "smart contract" prior to Satoshi [2]; and purely mechanical vending machines have sold cold drinks and train tickets long enough for Lord Denning [3] to remark in 1970:

> The customer pays his money and gets a ticket. He cannot refuse it. He cannot get his money back. He may protest to the machine, even swear at it. But it will remain unmoved. He is committed beyond recall. He was committed at the very moment when he put his money into the machine. The contract was concluded at that time.

This quote is as relevant to the smart contracts of today as it was to the train tickets of 1970, with one key difference—in 1970 the contract would still have to be *performed* in the real world (the customer gets on the train, which moves him to his destination), whereas today a smart contract could be performed entirely digitally, without human involvement.

This paper uses the running example of an online data sale, which translates the operational essence of a traditional sale and purchase agreement toward smart-contract digital execution. A seller offers some digital content; a buyer pays; the seller delivers. The traditional, natural language version of such a contract might say *The data seller is obliged to make the purchased data available for retrieval by the buyer for a term of 3 days after payment is received*. Such clauses are clear candidates for formalisation and digital execution by means of smart contracts.

We define a **smart contract** as an executable program (written in a programming language like Java, C++, Solidity, Go, etc.) that is deployed to mediate contractual interactions between two or more parties. Its task is to prevent (or at least

signal) deviations from the agreed behaviour. To perform its task the smart contract (i) intercepts or observes each operation initiated by the parties, (ii) analyses it to determine if it is contract-compliant, (iii) produces a verdict, and (iv) records the outcome in an indelible log that is available for verification, for example, to sort out disputes. 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 here: what technology to use to implement smart contracts that satisfy the imposed requirements. Note that in this paper we use the terms smart contract and contract synonymously.

Centralised (off-blockchain) and decentralised (on-blockchain) platforms are available for the implementation of smart contracts. However, we argue that neither alternative can individually provide the QoS demanded by some applications.

Currently, leading examples of smart-contract blockchain platforms are Bitcoin [4], [5] and Ethereum [6]. Bitcoin has been criticised for throughput limitations: it can only process 7 transactions per second, compared to Visa's 2000 transaction per second [7]. And it takes Bitcoin about 10 minutes to publish a transaction in its block [8].

Off-blockchain platforms were available long before the Satoshi's seminal paper [9] that launched Bitcoin [10]–[13], [13]–[17]. These platforms rely on Trusted Third Parties (TTP) which may not deserve that trust.

The central argument of this paper is that in several application areas, hybrid platforms composed from the integration of off- and on-blockchain platforms are better [18] than either alone. To date, the use of hybrid architectures in smart contract implementations has been largely unexplored. This papers aims to help close 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 an Ethereum blockchain. 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: We present a contract example as a motivating scenario in Section II. We discuss different approaches to smart contract implementation in Section III. Our experience in the implementation of the hybrid architecture is discussed in Section IV. In Section VII we place our research in context. In Section VI we discuss open research questions and pending work. In Section VIII, we present concluding remarks.

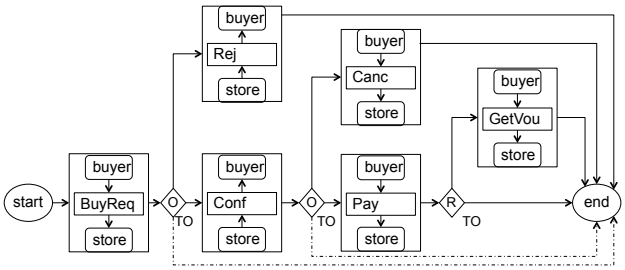

Fig. 1. A contract between a buyer and store for trading personal data.

II. Motivating scenario

Alice sells 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 [19]. Bob (the "buyer") buys data from Alice (the "seller" or "store"). The contract that governs their relationship includes the following clauses.

1) *The buyer (Bob) has the **right** to place a **buy request** with the store to buy an item.*
2) *The store (Alice) 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, within 5 days of submitting payment.*
5) *If, within 3 days of receiving the voucher, the buyer presents the voucher to the store, then the store **must deliver** the requested item.*

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 to honour the contract. We have highlighted the operations in bold. Though the clauses are relatively simple, they are realistic enough to illustrate our arguments.

III. Implementation alternatives

A close examination of the example reveals that the clauses describe the set of legal execution paths that the interaction between the two parties can follow. As such, the contract written in English—pseudocode—can be converted into a smart contract—a formalism—that can be enforced mechanically. To show how this can be done, we convert the English text contract into a systematic notation. Fig. 1 shows a graphical view of the contract example.

The operations in the English contract have been mapped to messages sent by one party to another. For example, the execution of the operation **buy request** corresponds to the **BuyReq** message sent by the buyer to the store. Similarly,

the execution of the operation **reject** corresponds to the **Rej** message sent by the store to the buyer. The diamonds represent exclusive splits in the execution path and have been labeled with *O* (Obligation) and *R* (Right). *TO* stands for Time Out and is used for defaults. Failure to execute and obligatory operations results in abnormal contract end (represented by dashed lines) with disputes to be sorted off line.

Fig. 1 reveals that the contract example can be modelled and implemented as a finite state machine (FSM). The challenge for the developer is to select a suitable architecture and technology for implementation. As discussed in [18], there are several approaches to smart contract implementations:

- Centralised: The smart contract is deployed on a Trusted Third Party. This approach is also known as off-blockchain implementation since there is no blockchain involved.
- Decentralised: The smart contract is deployed on a blockchain platform such as Ethereum. This approach is also known as on-blockchain.
- Hybrid: The contract is split and deployed partly off- and partly on-blockchain. Some clauses are enforced off-blockchain; others are enforced on-blockchain. The partition is based on several criteria including blockchain cost, performance, consensus latency, smart contract languages and privacy. See [18] [20] and [21].

IV. Hybrid Architecture

As explained in [18], the alternatives discussed in Section III offer different QoS attributes (for example, scalability, privacy, consensus latency, transaction fees) that render them suitable for some applications but unsuitable for others. There exist applications whose requirements are only met by the hybrid approach. In this section, we demonstrate a hybrid implementation.

Complexity inevitably emerges from the interaction between the off-blockchain and on-blockchain components. Several architectures are possible, such as master–slave or peer–to–peer. Alternatively, we can place them in a parallel-pipe relationship where an off-blockchain smart contract is deployed by one of the contractual parties to mirror the work of the on-blockchain smart contract, say to double-check its outputs. Other deployment alternatives are discussed in [22]. As discussed in Section IV-D, interaction intricacies demand systematic scrutiny to prevent buggy smart contracts.

The central idea of the hybrid approach is to divide the contractual operations into off-blockchain operations and on-blockchain operations. Off-blockchain operations are evaluated for contract compliance by a centralised smart contract deployed on a trusted third party. In contrast, on-blockchain operations are evaluated by a decentralised smart contract deployed on a blockchain.

Let us assume henceforth that Alice and Bob have agreed to use a hybrid architecture where the operation `Pay` will be enforced on-blockchain and all other operations, off-blockchain. An abstract view of the corresponding hybrid architecture is shown in Fig. 2.

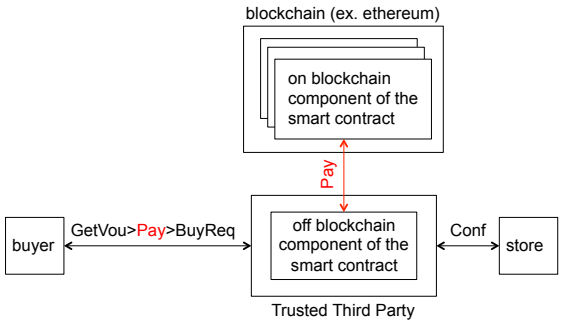

Fig. 2. Smart contract split into on- and off-blockchain enforcement.

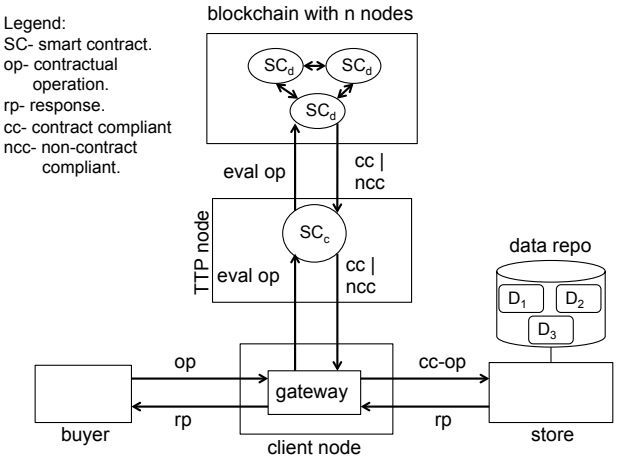

Fig. 3. A hybrid architecture for smart contracts: conceptual view.

Fig. 3 applies the concepts of Fig. 2 to the contract example of Section II. D_1, D_2 and D_3 are pieces of personal data that Alice is willing to sell, presumably under different conditions of price, privacy and so on.

The hybrid architecture can be implemented using a range of technologies. To realise the on-blockchain, decentralised component, we use the Rinkeby testnet of the Ethereum blockchain [6]. To realise the off-blockchain centralised component, we use the latest version of the Contract Compliance Checker (CCC) developed by University of Newcastle [23]. The integration follows a master–slave relationship between the centralised and decentralised smart contract components where the former is "in charge". The on-blockchain smart contract reads input events from the off-blockchain contract, treating it as an oracle. The off-blockchain code is able to read on-blockchain events—the chain itself.

A. Contract compliance checker

We use the contract compliance checker [17], [23] 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. Given a contract FSM, it grants and removes

rights, **obligations** and **prohibitions** to 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) 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. Similarly, when the seller executes the operation *Conf*, the seller's application sends the event *Conf* to the CCC for evaluation.

Drools is a declarative, Turing-complete language designed for writing business rules [24]. The contract loaded to the CCC is capable of evaluating contractual operations issued by business partners as RESTful requests against its rules. Rules give 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 parties, 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* which grants access to the seller's data. For example, when the buyer wishes to access the seller's data: (i) the buyer 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 contract clauses and responds with either *cc* or *ncc* to open or close the gateway, respectively, and (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. Note that, to keep the figure simple, the arrows show only the direction followed by operations initiated by the buyer. Operations initiated by the seller follow a similar procedure but right to left.

C. Ethereum

We chose the Ethereum platform [6] to implement the decentralised contract enforcer for the following reasons: It is currently one of the most mature blockchains. It supports Solidity—a Turing–complete language [25] 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 [26]. In addition, Ethereum offers developers online compilers of Solidity code [27]. 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 experiments using Ethereum tokens instead of "real" ether money [28], [29]. We run our experiments in Rinkeby as it is the most stable testnet. To conduct transactions, we created our own ERC20 tokens [30].

D. Execution sequences for testing the hybrid architecture

A feature of on-blockchain contracts is that because of their decentralisation and openness, they are generally immutable after deployment. Therefore, we suggest that smart contracts should be thoroughly validated (for example, using conventional model checking tools) to uncover potential logical inconsistencies in their clauses (omissions, contradictions, duplications, etc.) [31]. In addition, we suggest that the actual implementation be systematically tested before deployment. These tasks demand the assistance of software tools, such as the *contraval* tool that we have developed [32], specifically for model checking and testing contracts [32], [33]. It is based on the standard Promela language and the Spin model checker. It supports epromela (an extension of Promela) that provides constructs for intuitively expressing and manipulating contractual concepts such rights, obligations and role players.

In this work, 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. 6. We define an **execution sequence** 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*.

Our proof-of-concept proceeded as follows:

1) We converted the clauses of the contract example into a formal model written in epromela. We 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 problems (clause duplications, omissions, etc.) [31]).
3) We augmented the contractual model with an LTL formula and exposed it to Spin, and instructed Spin to produce counterexamples containing execution sequences of interest.
4) We ran a Python parser (called *parser-filtering.py* that we have implemented, over the Spin counterexamples to extract the execution sequences.

A close examination of Fig. 1 will reveal that it encodes six alternative paths from contract *start* to contract *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}
```

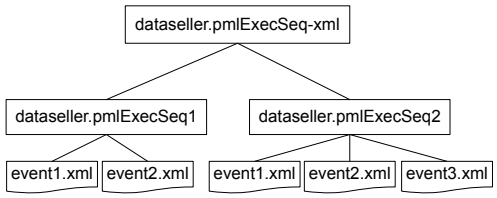

Fig. 4. Folder with execution sequences of the contract example.

Seq1, Seq2 and Seq3 result in normal contract completion. However, Seq4 and Seq5 result in abnormal contract completion. In Seq4 the seller fails to meet its obligation (to 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 has 5 days to claim a voucher, failure to execute `GetVou` does not result in abnormal contract completion because `GetVou` is a right, rather than an obligation. Seq6 is particularly problematic. It will be analysed separately in Section IV-G.

To ease sequence manipulation, we programmed the Python parser to store the execution sequences in N subfolders, one for each sequence. In our experiments with the contract example, we used the folders shown in Fig. 4.

Each subfolder *dataseller.pmlExecSeqi* (only two are shown in the figure) contains M files, *event1.xml*, *event2.xml*, etc., one file for each event included in the sequence. In our experiments, the subfolder *dataseller.pmlExecSeq1* is related to Seq1 and consequently, containes two files: event1.xml and event2.xml that correspond, respectively, to the *BuyReq* and *Rej* events.

We use XML-like tags to enrich the events with additional information, The contents of files event1.xml and event2.xml are shown in the following code, left and right, respectively.

```
<event>
<origin>buyer</origin>
<respond>store</respond>
<type>BuyReq</type>
<status>success</status>
</event>
```

```
<event>
<origin>store</origin>
<respond>buyer</respond>
<type>Rej</type>
<status>success</status>
</event>
```

The *type* tag indicates the type of the event. For example, the execution of the contractual operation *BuyReq* produces an event of type *BuyReq*, similarly, the execution of the contractual operation *Rej* produces an event of type *Rej*. The *origin* tag indicates the party that originated the event (the buyer in the example of the left), similarly, *respond* indicates the responding party—the store. *status* indicates the outcome of the execution, since we are not accounting for exceptional outcomes, the *status* is *success*.

E. On-Blockchain Deployment

The technology that we use in the integration is shown in Fig. 6. 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 deploy it on a Mac computer (regarded as a TTP node) as explained in Section IV-A. On the Mac we also deploy an ethereum client connected to the Rinkeby Ethereum network (see Fig. 6).

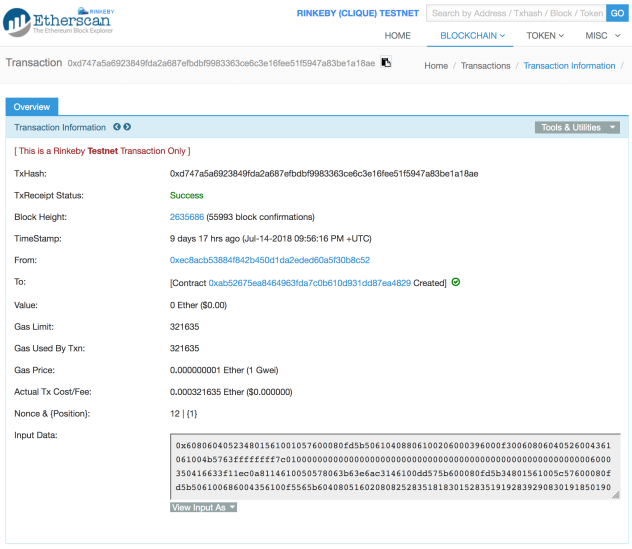

Fig. 5. Transaction that deployed the collectPayment.sol contract.

collectPayment.sol corresponds to the SC_d and is encoded in Solidity language [25]. There are several alternatives such as the `web3j` library to deploy the `collectPayment.sol` contract. For simplicity, we opted for metamask [34]: 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. 5 to deploy the `collectPayment.sol` contract on the Rinkeby test network [35]. We used Ethereum tokens to pay for gas.

F. Smart contracts code

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 contract
     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 the buyer if the buyer has the right
# ('cos it has fulfilled his payment oblig) to get it.
# It removes the 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;
 }
}
```

Since our focus here is to demonstrate the hybrid architecture, the *collectPayment.sol* contract is simple: it only receives string messages (not 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. 3 and acts as a web client to the CCC. 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 execution sequences encoded in the contract example and previously stored in the *dataseller.pmlExecSeq-xml* folder (see Fig. 4.

As shown in Fig. 6, the CCC relies on the `web3j` library [36] to communicate with the Ethereum client. Among other services, `web3j` includes a command line application that mechanically generates wrapper code from a smart contract specified in 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 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()` method of the `dataseller.drl` contract to forward the `Pay` operation to the `collectPayment.sol` contract. Intuitively, the statement calls the `submitPayment` function of the `collectPayment.sol` contract. The aim of this

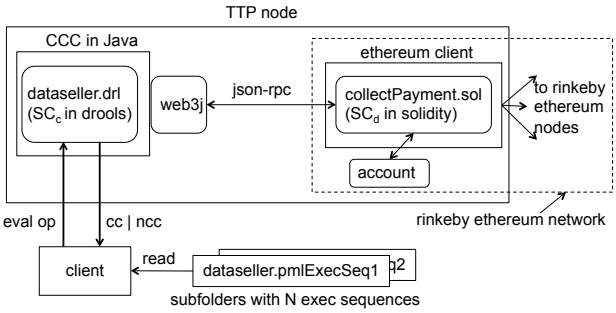

Fig. 6. Hybrid architecture for smart contract: technology view.

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. In production, it could be replaced by a function that returns actual Ethers representing the voucher for the buyer, or by any other function.

G. 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. We 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 as explained above.
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 execution 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 execution sequence.
6) When the client sends the last event of the execution sequence, it proceeds to the $dataseller.pmlExeSeq_2$, followed by the $dataseller.pmlExeSeq_3$ and so on, till $dataseller.pmlExeSeq_N$. 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 execution sequence `seq: BuyReq, Conf, Pay, GetVou` which includes the `Pay` operation is more prob-

lematic because it involves the `collectPayment.sol` contract. `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.
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:
   - **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.
   - **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 confirmation from the `collectPayment.sol` contract. Consequently, the `dataseller.drl` contract declares `GetVou` non–contract compliant — as far as the `dataseller.drl` contract is aware, the buyer does not have the right to get a voucher.

This problematic situation 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 `true` output of the third last line, in this execution the `dataseller.drl` contract declares the `GetVou` operation contract compliant.

```
/* In this run of the execution sequence
 *  BuyReq, Conf, Pay, GetVou the dataseller.drl contract
 * declares the GetVou operation contract compliance: true
 */

 -------- Begin Request to CCC service ----------
BusinessEvent{originator='buyer', responder='store',
             type='BuyReq', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


-------- Begin Request to CCC service ----------
BusinessEvent{originator='store', responder='buyer',
             type='Conf', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


 -------- Begin Request to CCC service ----------

BusinessEvent{originator='buyer', responder='store',
             type='Pay', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


 --------- Begin Request to CCC service ----------
BusinessEvent{originator='buyer', responder='store',
             type='GetVou', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------
```

In this execution, the `dataseller.drl` contract declares the `GetVou` operation non contract compliant (see (`false` in the third last line).

```
 /* In this run of the execution sequence
 *  BuyReq, Conf, Pay, GetVou the dataseller.drl contract
 * declares the GetVou operation non contract compliance:
 * false
 */

 -------- Begin Request to CCC service ----------
BusinessEvent{originator='buyer', responder='store',
             type='BuyReq', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


-------- Begin Request to CCC service ----------
BusinessEvent{originator='store', responder='buyer',
             type='Conf', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


 -------- Begin Request to CCC service ----------

BusinessEvent{originator='buyer', responder='store',
             type='Pay', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------
```

```
 --------- Begin Request to CCC service ----------
BusinessEvent{originator='buyer', responder='store',
              type='GetVou', status='success'}
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
 <result>
    <contractCompliant>false</contractCompliant>
</result>
-------- 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 (for example, with the *contraval* tool) and excluded from the model by the developer. But model checking is not enough. The error that we are analyzing materialises 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 to the problem; 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. Such measures need not rely on human labour; static analysis and formal methods should work.

H. Code and repeatability of experiments

The code used in the implementation of Fig. 6 is available from the *conch* Git repository. The `epromela` model and the ancillary code used to generate the execution sequences are available from the *contraval* Git repository. Interested in readers should be able to download both and replicate the experiments discussed in this paper.

V. Executions with abnormal completions

For the sake of simplicity, the discussion in Section IV-D about the contract example assumes that each contractual operation always succeeds. This is the desirable outcome; however, in practice an operation might fail for business or technical reasons. This problem is discussed in [17].

To account for potential exceptional execution outcomes of contractual operations, we use the execution model shown in Fig. 7.

The *OR exec* indicates that there is either a right or obligation to execute either $operation_A$ or $operation_B$ before a deadline $timeout$). The timeout arrow leads to the execution of another operation or to the end of the contract. In the simplest case, $operation_B$ is absent.

Imagine that the $operation_A$ is initiated by $party_S$. The output of the execution can be either $success$ or $bizfail$ (business failure). An execution that completes in $bizfail$ is normally retried until it either succeeds or a number of attempts (N) is exhausted. In our execution model, a $bizfail$ outcome that has not exceeded after N attempts leads to the *OR exec* where execution of $operation_A$, $operation_B$ or $timeout$ can take place. It is worth clarifying that the initiator of $operation_A$ and $operation_B$ is not necessarily the same contracting party.

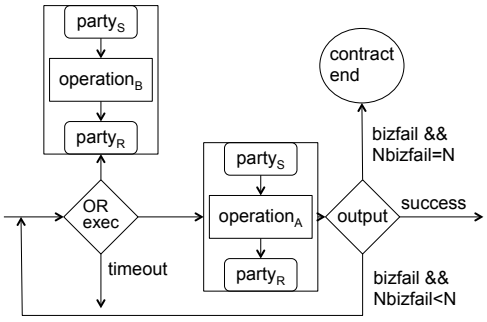

Fig. 7. Execution model of contractual operations.

We argue that realistic smart contracts need to account for exceptional outcomes and follow execution models similar to the one shown in Fig. 7. There are several alternatives. For example, a different model results from placing the head of the `bizfail && Nbizfails <N` to the right of the `OR exec` diamond. We have build an `epromela` model of the contract example following the execution model of Fig. 7. The *contraval* tool helped us to reveal that the model encodes 246 execution sequences. We have extracted some of them to illustrate our arguments.

```
// Execution sequences encoded in Fig 1 under the execution
// model of Fig. 4.
// (S)=  execution completed in success
// (BF)= execution completed in business failure
// RejConfTo=Rej or Conf timeout: store failed to execute
//           either rej or Conf by the 3day deadline.
// PayCancTo=Pay or Canc timeout: buyer failed to execute
//           either Pay or Canc by the 7day deadline.
 seq1:   {BuyReq, Rej}
 seq130: {BuyReq(S), Rej(BF), Conf(BF), Conf(S), Pay(BF),
         Canc(BF), Pay(S), GetVou(S)}
 seq150: {BuyReq(S), Rej(BF), Conf(BF), Conf(S), Pay(BF),
         Canc(BF), PayCancTO}
```

Some sequences, like Se1, are visible to the naked eye. Some can be found only by software verification tools. Within this category fall sequences (for example, `seq130` and `seq150`) that include business failures, retries and timeouts. `Seq130` allows the buyer to get his vouchers in spite of the failure of some operations. However, in `seq150` the buyer fails to get his voucher due to the occurrence of the `PayCancTO` timeout. All these sequences are available from the *examples/datasellercontract_TOandBizFailures* folder of the Git *Contraval* repository. Instructions and code to repeat the experiment to generate them are also provided.

VI. 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 a performance evaluation of some QoS requirements to demonstrate that the hybrid approach can meet them, and

equally importantly, to demonstrate in which situations the hybrid approach is better than pure off- and on-blockchain platforms. For example, we are planning to evaluate the throughput of the channel that communicates the off- and on-blockchain components. In pursuit of this aim, we are planning a more demanding contract example that includes several on-blockchain operations in addition to the single `Pay` operation of the current example. Another pending challenge is the exploration of different interaction models between the off- and on-blockchain components. For example, the `Pay` operation can be sent by the buyer directly to the Ethereum smart contract instead of sending it indirectly through the off-blockchain smart contract. There are also different deployment alternatives for the off-blockchain smart contracts [22]. For instance, it can be deployed within the buyer's or store's premises instead of on a trusted third party.

Our testing does not account for potential failures of the execution of contractual operations. For instance, it assumes that the pay operation always succeeds and ignores the possibility of technical failures (for example, the Ethereum network is unreachable) and business failures (for example, delivery address not found). We are planning to explore the behaviour of the hybrid architectures under the exposure of the execution sequences shown in Section V.

Another issue that deserves additional research is the analysis of the logics implicit in the English text of the contract as the logics impacts the implementation complexity and completness 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. Also, the inclusion of a timeout default converts and softens an obligation to respond, to a permission to respond. Finally, in the contract example for instance, the buyer's right to obtain a voucher from the store could be strengthened to an store's obligation to deliver the voucher if the buyer claims it. For the sake of readability the contract example is written in what is known as *denormalised* form which correspond to the popular intuitions about contract deontics [37]. These are issues that we are currently exploring within the context of our work on programmatic contract drafting.

Programmatic contract drafting is another open research area. In this regard, we are currently exploring the notion of the Ricardian Contract, [38] where systems build and fill templates both for formal-language contracts intended for digital execution (whether on- or off-blockchain), and for natural-language, human-readable versions of the contracts. These contracts in natural languages describe the same operational core but are intended to be signed on paper and legally binding. The natural language contracts also handle exceptions that cannot be handled within the on-blockchain contracts; for example, scenarios involving security holes in the on-blockchain contracts, or forks of the blockchain platform itself. Natural language generation systems offer the potential for efficient production and filling of such dual contract templates. Together, formal verification of programmatic smart contracts and natural language generation of human-readable contracts promise to create useful synergies: one product of these ideas is a natural language contract which has been mathematically proven to be bug-free.

VII. Related work

Research on smart contracts was pioneered by Minsky in the mid 80s [10] and followed by Marshall [11]. Though some of the contract tools exhibit some decentralised features [39], those systems took mainly centralised approaches. Within this category falls [40] and [14]. To the same category belongs the model for enforcing contractual agreements suggested by IBM [15] and the Heimdahl engine [41] aimed at monitoring state obligations (for example, *the store is obliged to maintain the data repository accessible on business days*). Directly related to our work is the Contract Compliant Checker reported in [17] [42] which also took a centralised approach to gain simplicity at the expense of all the drawbacks that TTPs inevitably introduce.

The publication of the Bitcoin white paper [2] in 2008 motivated the development of several platforms for supporting the implementation of decentralised smart contracts. Platforms in [4], [6] and [5] are some of the most representative. A good summary of the features offered by these and other platforms can be found in [43]. Though they differ in language expressive power, fees and other features they are convenient for implementing decentralised smart contracts.

An early example of a permissioned distributed ledger that is similar in functionality to the Hyperledger [5] of current blockchains is *Business to Business Objects (B2Bobjs)* [44]. *B2Bobjs* is a component middleware implemented at Newcastle University in the early 2000s and used for the enforcement of decentralised contracts [9]. As such, it offers consensus services (based on voting initiated by a proposer of a state change) and storage for recording non-repudiable and indelible records of the operations executed by the contracting parties. *B2Bobjs* is permissioned (as opposed to public) in the sense that only authenticated parties are granted access to the object.

Our hybrid approach addresses problems that neither the centralised nor decentralised approach can address separately. It was inspired by the arguments presented in [18], though the original idea emerged by the off-blockchain payment channel discussed in [4], [45]. The concept of logic-based smart contracts discussed in [46] 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 — here, Ethereum's Solidity. On- and off-blockchain enforcement of contractual operations is also discussed in [47]. Though an architecture is presented, no technical details about its implementation or functionality are given. Another conceptual design directly related to our work is private contracts executed in the Enigma [21] architecture. As in our work, a private

contract is a conventional business contract with 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. 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 (SPC) protocol [48], [49] that guarantees privacy. In that collaborative architecture, the blockchain is in charge. It is responsible for guaranteeing that the contractual operations are honoured and for delegating tasks to the Enigma nodes as needed. The integration of the SPC protocol ensures that the smart contract running in the Ethereum blockchain never accesses raw data that might compromise privacy. Unlike our TTP, the Enigma nodes charge computation and storage fees, as Ethereum and Bitcoin do. The cost that the Enigma architecture pays for privacy protection is complexity.

The logical correctness of smart contracts is discussed in several papers [50]–[53]. In [54] the author use Petri Nets for validating the correctness of business process expressed in BPMN notation and executed in Ethereum. They mechanically convert BPMN notation into Petri Nets, verify soundness and safeness properties, optimise the resulting Petri Net and convert it mechanically into Solidity. Formal systems for reasoning about the evolution of contract executions have also been suggested. Examples of questions of interest are to determine the current obligation or state of a party at time t and predicting weather a given contract will complete by time t. To address these issues, the authors in [55] suggest the use of timed calculus to reason about deontic modalities and conditions with time constraints. A system for programmatic analysis of contracts written in natural languages (normative texts) to extract contractual commitments (what parties are and are not expected to do) are discussed in [56].

The idea of interconnecting smart contracts to enable them to collaborate with each other is also discussed in [57]. These authors draw a similarity between blockchains and the Internet. They speculate that in the future, we will have islands of blockchain systems interconnected by gateways.

VIII. 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 to communicate with the standard APIs that current blockchains offer.

We have presented the approach as a pragmatic solution to the current problems that afflict off- and on-blockchain platforms. However, we believe that these ideas will become useful in the development of smart contract applications in the near future. We envision cross-smart contract applications that will involve several smart contracts running on independent platforms. The architecture that we have implemented is in line with this vision. Though the current implementation includes only two components, it can be generalised to include an 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 and testing of smart contracts.

Acknowledgements

Carlos Molina-Jimenez is currently collaborating with the HAT Community Foundation under the support of Grant RG90413 NRAG/536. Ioannis Sfyrakis was partly supported by the EU Horizon 2020 project PrismaCloud (https://prismacloud.eu) under 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] N. Szabo, "Smart contracts: Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, Sep. 1997.

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

[3] T. v Shoe Lane Parking Ltd [1970] EWCA Civ 2, http://www.bailii.org/ew/cases/EWCA/Civ/1970/2.html, 1970.

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

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

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

[7] 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.

[8] K. Noyen, D. Volland, D. Worner, and E. Fleisch, "When money learns to fly: Towards sensing as a service applications using bitcoin," https://arxiv.org/pdf/1409.5841.pdf, Sep. 2014.

[9] S. Shrivastava, "An overview of the tapas architecture," http://tapas.sourceforge.net/deliverables/D5Extra.pdf, Jan 2005, supplement Delivery of the TAPAS (Trusted and QoS-Aware Provision of Application Services) IST Project No: IST-2001-34069.

[10] 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.

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

[12] 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.

[13] P. Gama and P. Ferreira, "Obligation policies: An enforcement platform," in *Proc. 6th IEEE Int'l Workshop on Policies for Distributed Systems and Networks (POLICY'05)*. IEEE Computer Society, 2005, pp. 203–212.

[14] O. Perrin and C. Godart, "An approach to implement contracts as trusted intermediaries," in *Proc. 1st IEEE Int'l Workshop on Electronic Contracting (WEC'04)*, 2004, pp. 71–78.

[15] H. Ludwig and M. Stolze, "Simple obligation and right model (SORM)-for the runtime management of electronic service contracts," in *Proc. 2nd Int'l Workshop on Web Services, e–Business, and the Semantic Web(WES'03), LNCS vol. 3095*, 2003, pp. 62–76.

[16] L. Xu, "A multi-party contract model," *ACM SIGecom Exchanges*, vol. 5, no. 1, pp. 13–23, July 2004.

[17] 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.

[18] 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.

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

[20] J. Eberhardt and S. Tai, "On or off the blockchain? insights on off-chaining computation and data," in *Proc. i6th European Conference on Service-Oriented and Cloud Computing (ESOCC'17)*, 2017.

[21] 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].

[22] C. Molina-Jimenez, S. Shrivastava, and S. Wheater, "An architecture for negotiation and enforcement of resource usage policies," in *Proc. IEEE Int'l Conf. on Service Oriented Computing & Applications (SOCA 2011)*, 2011.

[23] 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.

[24] 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.

[25] Ethereum, "Solidity," http://solidity.readthedocs.io/en/develop/index.html, Visited 23 Oct 2017 2017.

[26] Bitcoin Wiki, "Scrypt," https://en.bitcoin.it/wiki/Script, 2018.

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

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

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

[30] Maxnachamkin, "How to create your own ethereum token in an hour (erc20 + verified)," https://steemit.com/ethereum/@maxnachamkin/how-to-create-your-own-ethereum-token-in-an-hour-erc20-verified, Visited 17 Jul 2018 2018.

[31] C. Molina-Jimenez and S. Shrivastava, "Model checking correctness properties of a middleware service for contract compliance," in *Proc. 4th Int'l Workshop on Middleware for Service Oriented Computing (MW4SOC'09)*, 2009, pp. 13–18.

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

[33] 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.

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

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

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

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

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

[39] 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.

[40] G. Governatori, Z. Milosevic, and S. Sadiq, "Compliance checking between business processes and business contracts," in *Proc. 10th IEEE Int'l Enterprise Distributed Object Computing Conf. (EDOC'06)*. IEEE computer society, 2006, pp. 221–232.

[41] P. Gama, C. Ribeiro, and P. Ferreira, "Heimdhal: A history–based policy engine for grids," in *Proc. 6th IEEE Int'l Symp. on Cluster Computing and the Grid (CCGRID'06)*. IEEE CS, 2006, pp. 481–488.

[42] E. Solaiman, I. Sfyrakis, and C. Molina-Jimenez, "A state aware model and architecture for the monitoring and enforcement of electronic contracts," in *Proc. IEEE 18th Conference on Business Informatics (CBI'2016)*, 2016.

[43] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," https://arxiv.org/pdf/1703.06322.pdf, visited in Nov 2012 2017.

[44] N. Cook, S. Shrivastava, and S. Wheater, "Distributed object middleware to support dependable information sharing between organisations," in *Proc. Int'l Conf. on Dependable Systems and Networks (DSN'02)*, 2002, pp. 249–258.

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

[46] 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,.

[47] 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.

[48] A. C. Yao, "Protocols for secure computations (extended abstract)," in *Proc. 23rd Annual Symposium on Foundations of Computer Science, (SFCS'08)*, 1982.

[49] M. Andrychowicz, S. Dziembowski, and Ł. M. Daniel Malinowski, "Secure multiparty computations on bitcoin," in *Proc. IEEE Symposium on Security and Privacy*, 2014.

[50] C. Prybila, S. Schulte, C. Hochreiner, and I. Weber, "Runtime verification for business processes utilizing state machine based apprddoach," https://arxiv.org/pdf/1706.04404.pdf, Aug 2018, arXiv:1706.04404 [cs.SE].

[51] I. Sergey, A. Kumar, and A. Hobor, "Scilla: A smart contract intermediate-level language: Automata for smart contract implementation and verification," https://arxiv.org/abs/1801.00687, Jan 2018, arXiv:1801.00687 [cs.PL].

[52] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, and N. Swamy, "Formal verification of smart contracts (short paper)," in *PLAS16*, 2018.

[53] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," https://arxiv.org/pdf/1711.09327.pdf, Nov 2018, arXiv:1711.09327 [cs.CR].

[54] 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].

[55] M. E. Cambronero, L. Llana, and G. J. Pace, "A timed contract–calculus," Department of Computer Science, University of Malta, Tech. Rep. CS2017-02, Sep 2017.

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

[57] 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].