TRiC: Terms, RIghts and Conditions Semantic Descriptors for Smart Contracts

Luis-Daniel Ibáñez (0000-0001-6993-0001) and Elena Simperl (0000-0003-1722-947X)

University of Southampton
[l.d.ibanez|e.simperl]@soton.ac.uk

Abstract. Smart Contracts have emerged as a novel way to automate the execution of contracts in a decentralised and secure environment, minimising the risk of breach or non-compliance. However, recent research points out that the same measures that secure Smart Contracts against disruption with the purpose of breach makes altering the terms, rights and conditions of contracts difficult. The same research proposes a set of standards inspired in paper-contract law that Smart Contract platforms should implement to enable Smart Contract Undo and Alteration. This paper is about preliminary work on describing terms, rights, and conditions of Smart Contracts as RDF documents linked to them, levering Semantic Web tools enabling: (i) Definition and checking of complex rights and conditions (ii) Separation of the terms of the contract from its execution logic. (iii) Querying and Updating via SPARQL (iv) Alteration of terms that were not initially considered as modifiable.

Keywords: Smart Contracts, Semantic Web, Smart Contract Update

1 Introduction

Distributed Ledger Technologies (DLTs) have emerged as a novel way to implement decentralised, disintermediated and tamper-free *transactions* of value. After their success as a mean to implement *digital currencies* [9] that do not require a bank or intermediate to secure transactions made with them, efforts were focused to generalise such an approach to the state transitions of programs written in Turing-Complete languages. Such generalisation would enable the secure, decentralised and disintermediated execution of arbitrarily complex interactions between agents. Following the analogy of a *contract* between agents, programs executed in such an environment are known as *Smart Contracts*. Researchers have already started to study the applicability of Smart Contracts for online identity and reputation [10], define interactions between IoT devices [4] and re-imagining several types of financial services and contracts [6].

A recent work by Marino and Juels [7] highlights an important shortcoming of Smart Contracts when used to replace paper-based contracts deposited with a legal intermediary: the improved security and tamper-free properties obstruct the application of desirable undoes and alterations in response to unforeseen or changing circumstances, raising the following issue: *How to undo or alter terms of Smart Contracts?* In the same paper, they propose a set of standards to bring existing tools for paper-based contracts to the Smart Contract realm, and show how to implement them as part of their code for the particular case of the Ethereum platform [1] and its Solidity language [2].

We argue that *metadata* about the terms, rights and conditions (that we abbreviate as *TRiCs*) of Smart Contracts should be separated from their logic in the same way runtime parameters are separated from the logic of traditional programs. We propose to encode TRiCs as RDF documents *linked* to Smart Contracts that we call *TRiC descriptors*. Such a decoupling and the use of RDF and related Semantic Web technologies enables the following desirable properties:

– Improve readability by avoiding boilerplate code
– Reduce the number of re-factorisation and re-deployment cycles. Re-deployment can be expensive in some Smart Contract platforms (e.g., Ethereum)
– Reasoning capabilities that enable the definition and checking of complex rights and conditions
– Query and Update of TRiCs via SPARQL, opening the door to mashing and integration with other contracts, their TRiCs, and the Web of Data

In this paper we describe preliminary work towards the definition and implementation of TRiC descriptors. Section 2 provides a brief overview of Smart Contracts and the definitions we will be using throughout the paper. Section 3 gives an overview of the requirements defined by [7] for undoing and altering Smart Contracts, and describes our running example. Section 4 details the TRiC descriptor proposed approach. Finally, section 5 concludes the paper and provides an overview of the research questions stemming from the future realisation of our approach.

2 Smart Contracts Preliminaries

The first definition of Smart Contract was given by Nick Szabo in [8].

Definition 1 (Smart Contract (from [8])). *Smart contracts are a combination of protocols, users interfaces, and promises expressed via those interfaces, to formalize and secure relationships over public networks.*

Smart Contracts enable better ways to formalize digital relationships than paper-based contracts, reducing costs imposed by either principals or third parties. The advent of Distributed Ledger Technologies made possible the development of platforms to code and execute decentralised applications,*i.e.*, parties wanting to execute a program do not need to trust each other or an external partner to execute it. Smart Contracts are a natural use case for these platforms, and many of them were designed with them in mind. We adopt the definition of Smart Contract given in the White Paper of the Ethereum platform [1].

Definition 2 (Smart Contract (from [1])). *A Smart Contract is a computer program code that is capable of facilitating, executing, and enforcing the negotiation or performance of an agreement (i.e. contract) using blockchain (Distributed Ledger) technology.*

We will also adopt the programming language of Ethereum, Solidity [2], for our code examples. Further required definitions are below.

Definition 3 (Smart Contract Platform). *A Smart Contract Platform (In short,* Platform*) is the infrastructure and machinery required to store and execute Smart Contracts.*

We abstract from the particular implementation of the platform, but we assume it implements the following affordances: (i) Has in place a system for agents to get *pseudonyms* and send messages under these pseudonyms to trigger, halt or alter contracts. Note that we allow an agent to have as many pseudonyms as it wants (ii) Each pseudonym has an *account* that holds cryptocurrency that may be transferred to contracts to trigger their functions (iii) A function (or a Smart Contract) that allows several pseudonyms to agree in a certain action, *e.g.*, invoke a function of a Smart Contract with the approval of all of them. Every time we say that pseudonyms or agents *agree*, we assume they did it through this function.

Definition 4 (Signatories). *Given a Smart Contract, we call* signatories *to the set of pseudonyms that have the right to alter it.*

Definition 5 (Variable terms). *Given a Smart Contract, we call its* variable terms *to the subset of its variables agreed by all signatories that can be altered.*

Definition 6 (Functional terms). *Given a Smart Contract, we call its* functional terms *to the subset of its functions agreed by all signatories that can be altered*[1]*.*

Example 1.1 shows a simplified Smart Contract for a Crowdraise[2]. We will use it as a running example for the remainder of the paper. The example defines a funding goal and two beneficiaries. The *contribute* function increments the amount raised in one unit every time is called. The *payable* keyword enables the handling by the platform of the transfer of *msg.value* (*i.e.*, the amount specified by the caller) units of cryptocurrency from the account of the caller of the function to the Smart Contract. Once the contract is deployed in the platform, agents wanting to collaborate can transfer funds to the contract by calling the *contribute* function. The withdrawal function checks that the funding goal has been achieved and that the caller is the first beneficiary, before sending half of the raised funds to each beneficiary[3]. We assume that both beneficiaries are also signatories of the Smart Contract.

[1] Functional and variable terms are referred in [7] as *Variable-Captured* and *Function-Captured* terms but not defined

[2] Loosely based on the example in `https://www.ethereum.org/crowdsale`

[3] For the sake of brevity, we omit the definition of the *FundTransfer* function

Example 1.1. Simplified Smart Contract for Crowdraising

```
contract Crowdraise {
  uint amountRaised;
  uint fundingGoal = 500;
  address[] benefs = {beneficiary1, beneficiary2}

  function contribute() payable
  {
    amountRaised = amountRaised + msg.value;   }

 function withdrawal()
 {
   if (amountRaised >= fundingGoal &&
             msg.sender == beneficiary1) {
      FundTransfer(beneficiary1, amountRaised/2);
      FundTransfer(beneficiary2, amountRaised/2);
   } }
}
```

Note that a simplistic way[4] to see a Smart Contract like our example is as a cryptographic safe box that contains value and only unlocks it if certain conditions are met. Note also that the Smart Contract executes a specific piece of code (one of its functions) whenever a message or transaction invokes it.

After a Smart Contract is deployed, it might be necessary to alter or undo some terms. Smart Contracts in platforms like Ethereum cannot be modified after being deployed, and redeployment of contracts may incur in high cryptocurrency fees. Based on our running example, we aim at providing a solution for the following alterations: 1) Modify the *fundingGoal* variable term 2) Temporarily stop receiving contributions or *disabling* the *contribute* function. 3) Change who has the right to call the withdrawal function 4) Change how the funds are transferred (e.g. 1/4th for one beneficiary and 3/4th for the other), *i.e.*, modify the *withdrawal* functional term.

3 Undo and Alteration of Smart Contracts

The work in [7] classifies undo and alteration of Smart Contracts according to the agent that solicits it. *By Right* means that one or more signatories have the right to undo or alter the Smart Contract unilaterally. *By Agreement* means that all signatories agree on undo or altering the contract. *By Court* means that a court mandated the undo or alteration. The implementation of all types can be summarized as the execution of the following steps:

[4] Though used in the live version of Ethereum's white paper https://github.com/ethereum/wiki/wiki/White-Paper#ethereum

1. Check the rights of the soliciting signatory, court or that the agreement is valid.
2. Check that further termination or alteration conditions beyond pseudonym rights are met
3. Halt the execution of the Smart Contract
4. If altering the Smart Contract, delete/add/edit terms
5. Compensate partial performance of all terms of the Smart Contract if undoing, if altering, compensate deleted/added/edited terms
6. If altering, start execution of modified Smart Contract

In this paper we focus on steps 1, 2 and 4, mapping to the following research questions: *How to model and check arbitrarily complex rights and termination conditions?* and *How to delete/add/edit terms while avoiding re-deployment.* In [7], these are implemented *into* the code of the Smart Contract using Solidity constructs available at their time. Based on their work, we rewrote them currently available Solidity constructs. Example 1.2 shows the final result.

To implement rights checking, we use a special function called *modifier*[5], that can be used to check conditions before executing other functions. In our example, we use the *rightscheck* modifier to check if *beneficiary1* is the pseudonym calling the function, stopping the execution otherwise. To modify the *fundingGoal* variable, we use the *setFundingGoal* function (a simple setter method), together with the *rightscheck* modifier. For disabling/enabling the *contribute* function, we use a combination of a *halt* boolean variable that is switched through the *fliphalt* function, and the *haltcheck* modifier, that simply aborts execution if *halt* is set to *true*. Note that with this pattern, the deletion of a function term is implemented as "disabling forever". Finally, for altering the *withdrawal*, we declare the address of a so-called *satellite* contract that can be set like any other variable term, the withdrawal function in the *master* contract is simply a wrapper to the function in the satellite contract. Note that the Satellite contract pattern naturally induces a *link* between both Smart Contracts.

We note several drawbacks of this approach: (i) It requires prognostication of which terms will be altered to place the appropriate modifier calls or setter methods. What if we need to add a new beneficiary? Or disable the withdrawal function following a court mandate? If the setter method was omitted before deploying, refactorization and redeployment is the only solution. (ii) It increases the number of lines of code not related to the logic of the contract. This becomes more evident if the contract requires complex rules, in our running example, what happens if each function needs to be called by a different beneficiary? What if there are other raising campaigns running in parallel and the right to withdraw funds from this campaign depends on the results of the others? (iii) It is not straight forward to alter the rights and conditions themselves. In our example, what happens if after a certain time both beneficiaries agree that both need to approve withdrawal of funds?

We argue that terms, rights and conditions in Smart Contracts are similar to *runtime parameters* in traditional programming. As such, we propose to separate

[5] https://solidity.readthedocs.io/en/develop/contracts.html#function-modifiers

Example 1.2. CrowdFunding Smart Contract with in-place code for alteration management

```
contract Crowdraise {

  uint amountRaised;
  uint fundingGoal = 500;
  address[] benefs = {beneficiary1, beneficiary2}

  bool halt = false;
  address satelliteContract;

  modifier rightscheck(){
    if (msg.sender != beneficiary1) throw;
    _;  }

  modifier haltcheck(){
    if (halt) throw;
    _;  }

  function setFundingGoal() {
    rightscheck
    amountRaised = msg.value;   }

  function flipHalt() {
     rightscheck
     halt = !halt;     }

  function setSatellite(){
    rightscheck
    satelliteContract = msg.value;    }

  function contribute() payable{
    haltcheck
    amountRaised = amountRaised + msg.value;   }

  function withdrawal(){
     Satellite sat = Satellite(satelliteAddress);
     sat.withdrawal();   }

  /* This function gets moved to a satellite contract
  function withdrawal(){
   if (amountRaised >= fundingGoal) {
        FundTransfer(beneficiary1, amountRaised/2);
        FundTransfer(beneficiary2, amountRaised/2);   }
  }
  */
}
```

them from their logic, in a similar way to configuration files. We propose to use RDF to encode these configuration files, in order to leverage the power of Semantic Web tools like inferencing and linking to external sources for enabling complex rights and conditions.

4 TRiC Descriptors

In this section, we describe our approach of Terms, RIghts and Conditions (TRiC) descriptors. A TRiC descriptor is an RDF document that encodes that describes the terms, rights and conditions of a Smart Contract. We assume that the platform is extended with the following capabilities: (i) A domain name and an URI minting machine under bespoke domain name. Each pseudonym and each Smart Contract in the platform have an IRI under the platform's domain name. We also assign an IRI to each variable and each function of each deployed Smart Contract (ii) Storage of RDF-Documents in a Distributed Ledger. Updates in these documents are treated as transactions in a Distributed Ledger, therefore, guaranteeing that they are tamper-free. (iii) A Graph Store to load RDF-Documents and execute SPARQL queries

Definition 7 (TRiC descriptor). *Given a Smart Contract S, its TRiC descriptor is an RDF document that contains data about its signatories, functional and variable terms, and rights and conditions*

A minimal TRiC descriptor contains one RDF triple stating a single signatory. A TRiC descriptor can be arbitrarily large, depending on the complexity of the rights and conditions that it encodes. The only modification that our approach requires to the Smart Contract code in example 1.1 is the addition of a link to a TRiC descriptor. This should be done in a way that guarantees it can be changed afterwards, for example, with a special variable that has a default setter that can be triggered upon agreement of all signatories. Compare this with the amount of code that had to be introduced in example 1.2.

It is out of the scope of this paper to discuss the most appropriate vocabularies to describe Smart Contracts and model rights. For the latter, we expect that work on general ontology-based access control like [3] could be adapted for this purpose. For the former, [5] presents MiniBlockVoc, a minimal vocabulary for Distributed Ledgers, including a *Smart Contract* class and property for declaring *signatories*. Following the requirements of our running example, we extend MiniBlockVoc with the following properties:

- A class *Term* and a property *hasTerm* with domain *Smart Contract* and range *Term*
- A Class *variableTerm*, having the property *value*, that represents the value of a variable term
- A Class functional term, having the properties *enabled*, with range *xsd:boolean*, representing if the function is enabled or not; *authorizedCaller* with range *Member*, representing Members with the right to call the function; and

replacedBy, with range *functionalTerm* that represents when a functional term has been replaced by other term

Figure 1 shows a diagram of our extension to MiniBlockVoc.

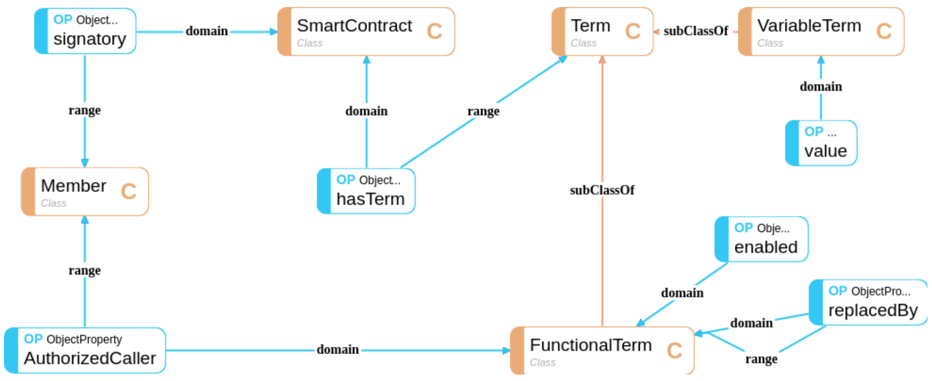

Fig. 1. Extension of the MiniBlockVoc vocabulary [5] for including terms and authorized callers

Example 1.3 shows the TRiC descriptor corresponding to our running example. Lines 4-5 state that *beneficiary1* and *beneficiary2* (here shortened to b1 and b2) are the signatories of the contract linked to this descriptor. Lines 7-9 state that *fundingGoal* is a variable term and sets its value (600) different from declaration. Lines 11-13 declare *contribute* as a function, the *active* property set to false indicates that it has been disabled. Lines 15-18 state that *withdrawal* is an active function and that its authorized caller is the agent identified with the pseudonym *b1* of this platform. Finally, line 20 states that the *withdrawal* function has been replaced by the *new-withdrawal* function on the *alt-contract* contract.

When a function of the Smart Contract is invoked, the platform executes the following algorithm:

1. Dereference the descriptor and load it into its Graph Store
2. Ask if the function is active or not. If so, continue, else, return.
3. Ask if the caller of the function has the right to do so according to the descriptor. If so, continue, else, return.
4. Override the values of all variables with new values according to the descriptor.
5. Ask if the function has been *replaced* by another one. If so, delegate the call to it. If not, execute the function as described in the contract.

To modify any term, right or condition, signatories agree[6] on a SPARQL Update query to be applied to the TRiC descriptor. In our running example, if signatories want to re-enable the *contribute* function, they agree in executing the SPARQL Update shown in listing 1.4.

[6] except if the alteration is by court

Example 1.3. TRiC descriptor corresponding to our running example

```
PREFIX plat:  <http://platform-domain.org/>
PREFIX mbv: <https://github.com/ldibanyez/miniblockvoc/MinimalBlockChain.owl>

plat:Crowdraise mbv:signatory platform/pseudonym/b1
plat:Crowdraise mbv:signatory platform/pseudonym/b2

plat:Crowdraise mbv:hasTerm plat:fundingGoal
plat:fundingGoal rdf:type mbv:variableTerm
plat:fundingGoal mbv:value 600

plat:Crowdraise mbv:function plat:contribute
plat:Crowdraise/contribute rdf:type mbv:functionalTerm
plat:Crowdraise/contribute mbv:enabled 'False'^xsd:boolean

plat:Crowdraise mbv:function plat:withdrawal
plat:withdrawal rdf:type function
plat:withdrawal mbv:active 'True'^xsd:boolean
plat:withdrawal mbv:authorizedCaller plat:pseudo/b1

plat:Crowdraise/withdrawal mbv:replacedBy
                          plat:alt-contract/new-withdrawal
```

5 Conclusion and Outlook

In this paper we presented preliminary work on TRiC descriptors, RDF documents describing Terms, RIghts and Conditions linked to Smart Contracts. TRiC descriptors enable a subset of the conditions proposed by Marino and Juels [7] for undo and alteration of Smart Contracts. The advantages of using TRiC descriptors over expressing TRiCs into Smart Contracts are numerous: (i) Separates the definition of rights and conditions from the Smart Contract logic, improving readability. (ii) Enables the querying, inference and update of terms, rights and conditions (via SPARQL) (iii) Allows the alteration of terms that were not identified as *modifiable* at deployment time

Our next steps are to develop an ontology for expressing the alteration requirements that could be shared among several Smart Contract platforms, and to implement TRiC descriptors into an existing Smart Contract platform to test their feasibility and performance. We believe that in this endeavour there are several challenges that need to be tackled:

– How to integrate a Graph Store into a Smart Contract Platform? Is it relevant for the TRiC descriptor context to execute SPARQL queries in a Smart Contract Platform as if they were code? If so, how to do it?
– How to efficiently store TRiC descriptors? Note that our approach implies replacing re-factorisation and re-deployment with data updates, therefore, op-

Example 1.4. SPARQL Update over TRiC descriptor

```
PREFIX plat:  <http://platform-domain.org/>
PREFIX mbv: <https://github.com/ldibanyez/miniblockvoc/MinimalBlockChain.owl>

DELETE { plat:Crowdraise/contribute mbv:enabled "false"^xsd:boolean }
INSERT { plat:Crowdraise/contribute mbv:enabled "true"^xsd:boolean }
WHERE
  { plat:Crowdraise/contribute mbv:enabled "false"^xsd:boolean }
```

timisation of said updates in a Distributed Ledger Environments environment is crucial

- How to check the validity of alterations made via TRiC descriptors from an execution and legal point of view? The question becomes more challenging in the presence of dependencies to other contracts.
- In our current definition, one TRiC descriptor is associated with one Smart Contract, is it possible to improve the approach to enable the re-use of descriptors (or parts of them) across several contracts?

References

1. A next-generation smart contract and decentralized application platform, https://github.com/ethereum/wiki/wiki/White-Paper
2. Solidity 0.4.10 documentation, https://solidity.readthedocs.io/en/develop/contracts.html
3. Buffa, M., Faron-Zucker, C.: Ontology-based access rights management. In: Advances in Knowledge Discovery and Management, vol. 2, pp. 49–61 (2012)
4. Christidis, K., Devetsikiotis, M.: Blockchains and smart contracts for the internet of things. IEEE Access 4, 2292–2303 (2016)
5. Ibáñez, L.D., Simperl, E., Gandon, F., Story, H.: Redecentralising the web with distributed ledgers. IEEE Intelligent Systems 32(1), 92–95 (2017)
6. MacDonald, T.J., Allen, D.W., Potts, J.: Blockchains and the boundaries of self-organized economies: Predictions for the future of banking. In: Banking Beyond Banks and Money, pp. 279–296. Springer (2016)
7. Marino, B., Juels, A.: Setting Standards for Altering and Undoing Smart Contracts. In: Rule Technologies: Research, Tools, and Applications. vol. 9718, pp. 151–166 (2016)
8. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997)
9. Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: A technical survey on decentralized digital currencies. IEEE Communications Surveys & Tutorials 18(3), 2084–2123 (2015)
10. Yasin, A., Liu, L.: An online identity and smart contract management system. In: 40th Annual Computer Software and Applications Conference (COMPSAC). vol. 2, pp. 192–198. IEEE (2016)