From Institutions to Code: Towards Automated Generation of Smart Contracts

Christopher K. Frantz
Otago Polytechnic
Dunedin, New Zealand
Email: `christopher.frantz@op.ac.nz`

Mariusz Nowostawski
Norwegian University of Science and Technology
Gjøvik, Norway
Email: `mariusz.nowostawski@ntnu.no`

***Abstract*—Blockchain technology has emerged as a solution to consistency problems in peer to peer networks. By now, it has matured as a solution to a range of use cases in which it can effectively provide the notion of third party trust without the need for a trusted (physical) third party, which makes it an attractive coordination mechanism for distributed systems. To promote the wide adoption of this technology, we yet lack mechanisms that make the specification and interpretation of smart contracts accessible to a broader audience. In this work, we propose a modeling approach that supports the semi-automated translation of human-readable contract representations into computational equivalents in order to enable the codification of laws into verifiable and enforceable computational structures that reside within a public blockchain. We identify smart contract components that correspond to real world institutions, and propose a mapping that we operationalize using a domain-specific language in order to support the contract modeling process. We explore this capability based on selected examples and plot out directions for future research on smart contracts.**

Keywords–blockchain; autonomy; distributed autonomous institutions; public ledger; smart contracts; Ethereum; Solidity; institutions; institutional grammar; code generation; model-driven development; domain-specific language; Bitcoin

I. Introduction

To date the most successful example of blockchain deployments is Bitcoin. Bitcoin uses public-private key cryptography, consensus rules and incentive systems to achieve consensus in an open distributed peer-to-peer system. Even though discussions are generally associated with Bitcoin [1] as the currently most prominent instance of blockchain technology, the implications of blockchain technology reach much farther, beyond digital currencies.

Whereas earlier blockchain applications concentrated on the management of distributed state, the most modern blockchain deployments, such as Ethereum [2], not only allow the distributed management of state, but also the execution of procedural instructions in the form of *smart contracts*. Smart contracts reflect a notion of dynamically specified and instantiated contracts that live on the blockchain, outside the unilateral control of a single participant. The ability to express logic within an open, trusted and verifiable peer-to-peer system offers opportunities for individual users as well as corporations to delegate parts of their activities into a public blockchain. Examples include decision-making based on voting, crowd funding, assets management or workflow management. However, the open nature of these mechanisms offers opportunities that reach beyond the conventional integration of information systems or coordination of human actors. Smart contracts can be likewise be used by artificial entities that autonomously engage in contractual commitments, or create contracts to offer services to an open audience. This further extends to the potential use of the blockchain as coordination infrastructure for collective adaptive systems (CAS), a) affording the structural heterogeneity of participating entities, b) accommodating their globally distributed operation, and c) delegate life cycle management of contracts to the collective itself, instead of relying on humans to provide and maintain the infrastructure.

However, when interacting in an open environment, contractual specifications should be accessible to all engaging entities, whether artificial or human. While the blockchain assures deterministic execution and consistent state representations, the codified contracts as de facto coordination protocols, still require careful design and implementation, and cannot guard against badly written or insecure contracts, an aspect recently witnessed in the massive theft of funds from the Ethereum's most successful *Decentralized Autonomous Institution* (DAO) [3]. As with any programming task, the human-readable contract and associated obligations need to be codified, and then subsequently verified, to ensure that the machine-readable representation conforms to the specified behaviour. We believe that the complexity of codifying smart contracts, and the necessity to do so correctly (since they are publicly accessible), limits the mainstream adoption and acceptance of this technology.

To leverage broader understanding of the functionality and to make the creation of contracts accessible for the general use, we propose a mechanism that semi-automates contract generation by translating institutional formalization from a human-readable behaviour specification to a contractual structure in the form of smart contracts. In Section II we briefly highlight the technological underpinnings of the blockchain application Ethereum and its notion of executable smart contracts. We propose a component mapping for a widely accepted institution representation onto structural elements of smart contracts, before exemplifying this functionality for various coordination problems in Section III. Finally, in Section IV we discuss implications and the potential of the proposed approach.

II. From Institutions to Contracts

Up to now, we have used the term 'contract' largely intuitively without describing a contract's structure and semantics in the technological sense. We first discuss technical details of the Ethereum platform, before moving towards the structure and capabilities of *smart contracts*.

A. Ethereum Virtual Machine

Smart contracts are executed on a specially designed blockchain virtual machine, called Ethereum Virtual Machine (EVM) [2]. The EVM is a stack-based virtual machine that operates on bytecode language. All Ethereum nodes share the same EVM specification, and the code is executed in a distributed fashion across all the nodes that validate the transactions. Program execution is always bound in time and in space by the up-front provision of the consumable *gas*, a unit of execution cost that is incurred by each opcode instruction, and by each byte of storage utilized by the contract. Payments underlie the same principle as in Bitcoin, but instead rely on the denomination *ether*.

While Ethereum's bytecode language is designed towards efficient distribution and execution, a set of additional high-level languages have emerged in order to simplify the creation of contracts. Those include Solidity, Serpent, and LLL, with Solidity [4] being the de facto standard for contract development. Solidity's syntax is derived from Javascript, but accommodates conceptual differences such as static typing.

B. Smart Contracts

Contracts correspond to classes in object-oriented languages and can hold typed state variables. Apart from general-purpose types, such as string, integers, static or dynamic arrays, found in traditional programming languages, a central type in Solidity is the *address* that identifies users (EOAs) and other contracts' locations.

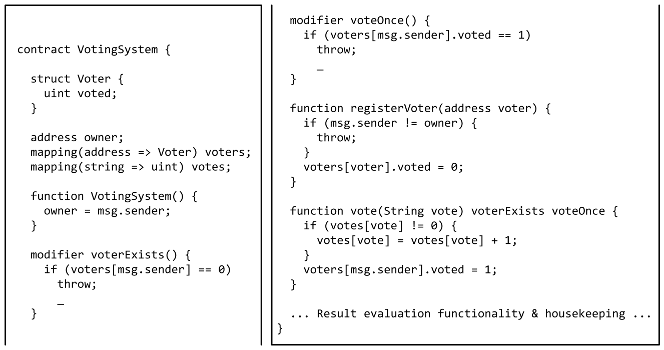
In addition, contracts can contain functions that are externally invoked. Function modifiers can further be attached to one or more functions in order to perform preliminary checks (e.g. for data validation) in a declarative manner, thus reflecting characteristics of aspect-oriented programming [5]. The EVM further supports events for the purpose of notification or implementation of callbacks. Note that though the EVM requires the execution of code on all connected nodes, it does not support any form of parallelization as part of the code. In addition to functions, Solidity supports structs as well as enumerations. Within contracts code can access properties of the invoking message, such as sender address, remaining funds for function execution (*gas*), various crytographic functions as well as a predefined *selfdestruct()* function for the destruction of contracts. It is worth noting that in Ethereum the termination of an object's existence can only be achieved by the object itself. The public nature of the blockchain makes a careful development and life cycle management thus essential. The complete language documentation can be found in the Solidity specification [4].

To instantiate new contracts, they are compiled into EVM bytecode and users must obtain sufficient amount of ether to fund the execution of the contract based on the estimated complexity of the submitted code. Transactions can then invoke functionality on the contract's address. For this purpose, Ethereum knows two types of transactions, consisting of the essential attributes:

- receiving address,
- included ether amount,
- a byte array containing the payload, and
- a signature of the sender account's private key.

If the receiving address is omitted, the transaction payload must be a new contract which is added to the network. Before executing the transaction, the EVM will establish the authenticity of transaction requests based on the sender's signature, and it will validate the attached ether needed to sponsor EVM code execution.

Figure 1 depicts a contract extract that represents a simple voting system. Upon creation (Constructor *VotingSystem()*) the contract records the creator's address (state variable *owner*). It further maintains a set of mappings that capture voters and the corresponding votes. Voters need to be registered by the contract owner in order to participate (Function *registerVoter()*). Voters can then vote by invoking the function *vote(String vote)* that takes their choice as parameter. The call to *vote()* is preempted by two checks on corresponding modifiers, that is *voterExists()* and *voteOnce()*. The remaining evaluation functionality (e.g. determining the highest vote) is omitted in this example. We further omitted infrastructural functions that are not of direct relevance at this stage.



```
contract VotingSystem {

  struct Voter {
    uint voted;
  }

  address owner;
  mapping(address => Voter) voters;
  mapping(string => uint) votes;

  function VotingSystem() {
    owner = msg.sender;
  }

  modifier voterExists() {
    if (voters[msg.sender] == 0)
      throw;
      _
  }

  modifier voteOnce() {
    if (voters[msg.sender].voted == 1)
      throw;
      _
  }

  function registerVoter(address voter) {
    if (msg.sender != owner) {
      throw;
    }
    voters[voter].voted = 0;
  }

  function vote(String vote) voterExists voteOnce {
    if (votes[vote] != 0) {
      votes[vote] = votes[vote] + 1;
    }
    voters[msg.sender].voted = 1;
  }

  ... Result evaluation functionality & housekeeping ...
}
```

Figure 1. Example Contract for Voting (compact representation)

Note that contracts do not require the representation of contractual obligations in the strict sense, but can, similar to objects, be composed of elementary contracts that serve specific purposes, such as state-centric or logic-centric operations. They can furthermore inherit properties from other contracts.

The use of high-level languages makes it relatively easy to specify and encode smart contracts. However, it is worth highlighting that the potential use of smart contracts as mediators in open systems makes them critical infrastructure. This can apply to a single organization's functioning, or to governments and countries, if those were to adopt blockchain technology and smart contracts to provide sensitive public services, for example land ownership registers. If contracts are ill-specified or exploitable, the implications can be far-reaching. Furthermore, at the current stage few approaches are available that support the systematic modeling of contracts in order to resemble real-world institutions. In this work we intend to bridge this gap by proposing a possible mapping between institutional specifications made in human-readable language and the machine-readable encoding in the form of Solidity contracts.

C. Institutional Grammar

For this purpose we borrow a concept from the area of institutional analysis [6] that deals with the systematic analysis of functions of human institutions from a social and economic perspective. In this context Crawford and Ostrom have proposed a *Grammar of Institutions* [7] that captures essential institutional characteristics. Using this representation, it is possible to effectively decompose institutions into simple rule-based statements that capture the essence of the institution's function. Statements are constructed from different components (abbreviated as *ADICO*) that include:

- ***A**ttributes* – describe an actor's characteristics or attributes.
- ***D**eontic* – describes the nature of the statement as an obligation, permission or prohibition.
- *A**I**m* – describes the action or outcome that this statement regulates.
- ***C**onditions* – describe the contextual conditions under which this statement holds. If not specified, the institutional statement holds under any circumstance.
- ***O**r else* – describes consequences associated with nonconformance.

Based on those individual components, we can specify institutional statements of varying character that include one or more of the above-mentioned components. For example a prescription along with consequences for non-compliance can be described as `People` **`(A)`** `must` **`(D)`** `vote` **`(I)`** `every four years` **`(C)`**`, or else they face a fine` **`(O)`**.

This grammar has found application in the context of policy-coding [8] as well as agent-based modeling [9].

We borrow the structure of Nested ADICO (nADICO) [10], a refined variant of the original institutional grammar, to decompose complex institutional functions into a simple set of prescriptions that can be linked using the logical operators `and`, `or`, and `xor` to express conjunctions, inclusive disjunctions and exclusive disjunctions.

Using this structure, we can express majority-based voting as a set of declarative rules, represented as combined institutional statements:

- *People **(A)** must **(D)** vote **(I)** only once **(C)*** `AND`
- *People **(A)** may **(D)** vote **(I)** if they are registered voters **(C)*** `AND`
- *People **(A)** must not **(D)** vote **(I)** after the deadline **(C)**.*

In principle, we could further specify obligations associated with the institutional functionality itself, such as

*The contract **(A)** must **(D)** notify voters about the outcome **(I)** once the deadline is reached **(C)**.*

D. Mapping Grammar Components to Contract Structure

Exploring the structural elements of contracts and the institutional grammar, we can identify the essential domain-specific constructs, and propose a mapping that allows us to simplify the generation of contracts. For example, the conceptual equivalent to ADICO's *Attributes* component is Solidity's *struct*. In analogy, *aims* effectively represent *functions* and *events* (which we will explore further below), whereas the combination of *Deontic* and corresponding *Conditions* component are reflected in *function modifiers* that introduce declarative checks that preempt function execution. Table 1 summarizes the proposed ADICO-Solidity mapping.

TABLE I. MAPPING OF ADICO COMPONENTS TO SOLIDITY DECLARATIONS

ADICO Component	**Solidity Construct**
Attributes	Structs
Deontic	Function modifiers
Aim	Functions, Events
Conditions	Function modifiers
Or else	*throw* statements/alternative control flow

This mapping of core constructs provides the foundation to translate institutional specifications into Solidity contracts. However, in addition to the high-level mapping, we can refine ADICO's component structure. We introduce an open set of properties attached to the *Attributes* component (i.e. actor properties), which translate to Solidity struct members. We further refine *Aims* (which represent the action an institutional statement controls) by allowing the specification of object and target associated with a given action, both of which correspond to parameters for Solidity functions. A special case are events that are triggered based on the fulfilment of encoded conditions (e.g. reaching a deadline). A further refinement involves the potential annotation of attributes' property data types in order to improve code generation. As an operationalization of ADICO, *Attributes* can be explicitly specified, and we further permit the specification of objects associated with actions as well as potential targets of an action. Additionally, multiple conditions can in principle be combined into a single statement. As discussed in Section II-C, multiple ADICO statements can be combined using logical connectors.

To clarify this mapping and exemplify its use, we have developed a domain-specific language (DSL) using Scala [11] that automates this process using a templating approach. The syntax for the ADICO-Solidity DSL is shown in EBNF notation in Figure 2.

```
actor       ⇐ <String literal> ;                                  (* Attributes actor *)
prop        ⇐ property(<String literal>[, <String literal>]) ; (* Attributes property and
                                                                        optional type definition *)
deontic     ⇐ “may” | “must” | “must not” ;                       (* Deontic value *)
action      ⇐ <String literal> ;                                  (* Aim action *)
obj         ⇐ object(<String literal>[, <String literal>]) ;   (* Aim object *)
tgt         ⇐ target(<String literal>[, <String literal>]) ;   (* Aim target *)
lOperator   ⇐ “AND” | “OR” | “XOR” ;                              (* Logical operators *)
condition   ⇐ IF(<Boolean expression>) |
                (condition lOperator condition) ;                 (* Individual condition or
                                                                        condition combination *)
conditions  ⇐ condition {lOperator condition} ;                   (* Conditions component *)
adico       ⇐ ADICO(
                A(actor[{, prop}]),
                D(deontic),
                I(action[, object][, target])
                [, C(conditions)]
                [, O(adico)]) ;                                   (* Individual ADICO statement *)
statement   ⇐ (adico {lOperator adico}) ;                         (* Complete statement *)
```

Figure 2. ADICO-Solidity DSL Syntax

The associated algorithm involves the following steps:

1) It initially iterates over all institutional statements associated with a given contract and aggregates all properties associated with an attribute concept to model a struct from those.
2) All conditions are extracted from statements to form individual modifiers. The deontic component (e.g. must, must not, may) inverts given conditional statements. *Or else* statements are translated as consequences for violating conditions in function modifiers, using the *throw* primitive by default. Conditions that are combined by disjunctions are represented by a single modifier construct that delegates the implementation of the semantics to the developer.
3) Functions are generated based on aim name and properties that are used to construct the function signature. Associated function modifiers (generated in Step 2) are attached to the generated function.
4) Events are introduced for statements whose aims' actions use the keyword 'notify'.

Based on these principles we can now decompose intuitive, yet complex institutional operations into simple rules that capture the peculiarities of concepts such as voting. Beyond the introduced conceptual mapping, the translation from ADICO statements to Solidity constructs operates on a syntactic level and requires developers to add the actual semantics associated with the specified functionality. However, the provision of a contract skeleton offers two important benefits:

- It separates the specification task from implementation and thus provides a safeguard against omissions of crucial functionality as part of the implementation process.
- It allows modellers to specify institutional constructs in a declarative form that is (a) accessible to individuals of varying, potentially even non-technical background, and (b) allows the modeling on a fixed intermediate level of abstraction, which is helpful when conceptualising heterogeneous systems that are characterised by open environments and thus changing interaction partners.

In addition to supporting the systematic construction of institutions from the bottom up, we can address challenges that are specific to the smart contract implementation in Ethereum in order to avoid pitfalls, such as maintaining control about deployed contracts as well as to deal with malformed contract invocations. In consequence, we can encode those directly as part of the contract generation. We will explore the introduced concept based on a set of examples.

III. Examples

A. Example 1: Voting with Result Notification

The following statements model a simple voting mechanism that permits registered voters to cast their vote. This specification further notifies another contract once a threshold of voting participation is reached. The individual statements decompose individual permissible actions and associated constraints: Voters need to be registered (Statement 1), may only vote once (Statement 2), but may only vote prior to a given deadline (Statement 1). Statement 3 describes the notification of external accounts (i.e. another contract) upon having collected a given number of votes. This statement further exemplifies varying levels of detail regarding attribute and condition specifications (e.g. optional provision of type information) and the use of syntactic sugar to generate machine-parseable conditions (see *Conditions* component in Statement 3). However, naturally this comes at the expense of manually refining contracts during implementation (e.g. types, machine-parseable condition specifications).

```
Adico(
  A("Voters"),
  D(may),
  I("cast", object("vote", "string"), target("candidate")),
  C(IF("voter", "is" , "registered") AND
    IF("vote", "before", "deadline"))) AND
Adico(
  A("Voters"),
  D(may),
  I("cast", object("vote", "string"), target("candidate")),
  C("only", "once")) AND
Adico(
  A("System"),
  D(must),
  I("notify", object("vote count"), target("contract",
    "address")),
  C(IF("votes.length", Operator.>, "100")))
```

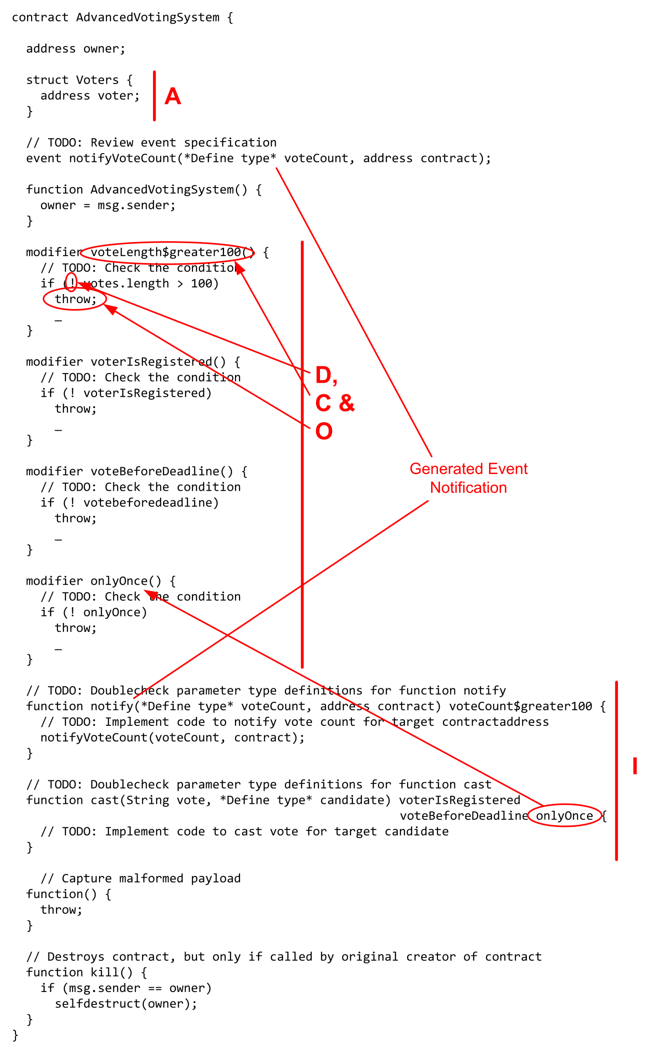
Figure 3 shows the corresponding generated contract, including the generated event as well as associated function. Similar to all other constructs, the specification of events can be of limited accuracy, for example it omits type specifications. As mentioned before, the contract implementation requires manual refinement in order to reflect the required semantics. To aid this process, all relevant constructs are annotated with TODO labels. At this stage the generated contract skeleton represents an interface that captures the contract's purpose on an abstraction level defined by the institutional statements, without exposing implementation details.

In addition to the translation of individual statement components into Solidity code artefacts, the generated code can also consider aspect that are specific to contract implementations in Solidity, so as to safeguard the users and developers from common pitfalls. This includes the specification of a default method that captures transactions with invalid payload to avoid unnecessary expenses on the part of the invoking party. Another provision is to control the ability to manage the life cycle of a contract by restricting the ability to destroy a contract to the creating owner (see owner initialization in constructor (*AdvancedVotingSystem()*) and function *kill()*).

B. Example 2: Escrow Service

As a second example we present a simple escrow service that performs a trust role typically endowed to third parties specialized in the mediation of transactions. With Ethereum we can codify this functionality with transparency for both involved parties. We will use this example to explore statements of higher complexity, such as the consideration of consequences.

The following statements characterise an escrow service that requires the payment of funds prior to a deadline, or otherwise releases the traded object (*objectOfInterest*) and returns eventual (partial) funds to the original buyer as shown in the first statement. This exploits the nesting capability of the institutional grammar (combined nested consequences) in



```
contract AdvancedVotingSystem {

  address owner;

  struct Voters {
    address voter;
  }

  // TODO: Review event specification
  event notifyVoteCount(*Define type* voteCount, address contract);

  function AdvancedVotingSystem() {
    owner = msg.sender;
  }

  modifier voteLength$greater100() {
    // TODO: Check the condition
    if (! votes.length > 100)
      throw;
      _
  }

  modifier voterIsRegistered() {
    // TODO: Check the condition
    if (! voterIsRegistered)
      throw;
      _
  }

  modifier voteBeforeDeadline() {
    // TODO: Check the condition
    if (! votebeforedeadline)
      throw;
      _
  }

  modifier onlyOnce() {
    // TODO: Check the condition
    if (! onlyOnce)
      throw;
      _
  }

  // TODO: Doublecheck parameter type definitions for function notify
  function notify(*Define type* voteCount, address contract) voteCount$greater100 {
    // TODO: Implement code to notify vote count for target contractaddress
    notifyVoteCount(voteCount, contract);
  }

  // TODO: Doublecheck parameter type definitions for function cast
  function cast(String vote, *Define type* candidate) voterIsRegistered
                                                      voteBeforeDeadline onlyOnce {
    // TODO: Implement code to cast vote for target candidate
  }

    // Capture malformed payload
  function() {
    throw;
  }

  // Destroys contract, but only if called by original creator of contract
  function kill() {
    if (msg.sender == owner)
      selfdestruct(owner);
  }
}
```

Figure 3. Voting Example with Result Notification

order to reflect complex institutional interdependencies, but, in this case, to automate the generation of control flow.

The remaining statements indicate the release of object and payment of seller if sufficient funds are deployed. For the purpose of brevity we omit further essential aspects (that are secondary in this context) such as prior registration of buyer and seller from the statement specification as well as life cycle management functions.

```
Adico(
  A("buyer"),
  D(must),
  I("pay", object("funds")),
  C("before", "deadline"),
  O(Adico(
      A("system"),
      D(must),
      I("release", object("objectOfInterest"),
        target("seller", "address")))
   AND
    Adico(
      A("system"),
      D(must),
      I("send", object("funds"),
        target("buyer", "address")))
      )
  ) AND
Adico(
  A("system"),
  D(must),
  I("send", object("funds"),
    target("seller", "address")),
  C(IF("msg.value", Operator.>=, "price"))) AND
Adico(
  A("system"),
  D(must),
  I("release", object("objectOfInterest"),
    target("buyer", "address")),
  C(IF("msg.value", Operator.>=, "price")))
```

The translation of these statements results in a contract skeleton shown in Figure 4.

The operation on funds as a central economic resource is a first-order entity in Ethereum. This allows us to encode the operation of currency transactions, since those are tied to individual accounts and are part of any transaction message, which can be accessed via *msg.value* and sent by a call to the corresponding target address (e.g. *buyer.send(amount)*). This way we can nearly automate the generation of the modifier that captures the consequences attached to failing to pay before a given deadline (Modifier *payBeforeDeadline()*).

```
contract EscrowSystem {

  address owner;

  function EscrowSystem() {
    owner = msg.sender;
  }

  ... Seller and buyer registration ...

  // TODO: Doublecheck parameter type definitions for function release
  function release(*Define type* objectOfInterest, address buyer) {
    // Implement code to release objectOfInterest to target buyer
  }

  modifier payBeforeDeadline() {
    // TODO: Check the condition
    if (! payBeforeDeadline) {
      release(objectOfInterest, buyer);
      buyer.send(funds);
    }
    else {
      _
    }
  }

  // TODO: Doublecheck parameter type definitions for function pay
  function pay(*Define type* funds) payBeforeDeadline
                             buyerIsRegistered sellerIsRegistered {
    // TODO: Implement code to pay funds
  }

  modifier msgValue$greaterEqualsPrice() {
    // TODO: Check the condition
    if (! msg.value >= price)
      throw;
      _
  }

  // TODO: Doublecheck parameter type definitions for function release
  function release(*Define type* objectOfInterest) msgValue$greaterEqualsPrice {
    // TODO: Implement code to release objectOfInterest
  }

  // TODO: Doublecheck parameter type definitions for function send
  function send(*Define type* funds) msgValue$greaterEqualsPrice {
    seller.send(funds);
  }

  ... life cycle management functions ...

}
```

Figure 4. Escrow Service with Expiry

Similarly, we can further automate the code generation for the modifier that checks for sufficient payment. However, as with the previous cases, the generated contract is far from executable and requires revision by a developer. It nevertheless captures essential institutional characteristics and prevents their omission. A future direction is the systematic consideration of

Solidity-specific functionality in order to provide code stubs that reach beyond the specification of the contract interface.

IV. Summary, Discussion & Outlook

In this paper we have outlined the potential of blockchain technology to coordinate interaction between independent entities, such as humans, agents, etc. The central challenge associated with a broader use is the unambiguous and correct specification of smart contracts. In this context, the main contribution is the introduction of a process that automates the translation of institutional constructs into codified machine-readable contractual rules. To achieve that, we have used a two step process. First, we express human-readable institutional rules, regulations, and laws in terms of ADICO statements that capture high-level institutional semantics. As a second step, those sentences are mapped and transcribed into Solidity, the smart contract language of the Ethereum platform. The ability to generate those smart contracts easily by human participants with varying level of technological background makes this technology accessible to groups that would otherwise not participate in the use of the blockchain as a coordination tool. On the other hand, the notion of smart contracts represents an expressive modelling tool that can coordinate system interaction on a high level of abstraction, while maintaining a representation that is conceptually and syntactically accessible to humans.

However, our approach to aid the translation of interpretable contracts into codified institutional representations does not exist in isolation. Pitt et al. [12] describe an approach that is inspired by Ostrom's institutional framework [6] and uses event calculus to represent nested institutional rule sets in self-organising systems. Similarly, da Silva Figueiredo et al. [13] propose a generic description language that formalises the norm specifications comprehensively, similar to the nADICO structure used in this work. At the current stage the described approach exclusively concentrates on the operational level of institutional rules; structural institutional regress in the form of Ostrom's meta-rules and constitutional rules (that shape operational rules) is not considered. Modelling the coordination of independent entities is an essential challenge in the context of collective adaptive and self-organising systems. To enabling the use of the proposed approach in the area of distributed systems in general, and collective adaptive systems in particular, this work minimizes any assumptions about the interacting individuals (e.g. agents, humans, contracts, services), other than requiring them to have access to shared smart contracts. At the current stage we are not aware of existing approaches that support the generation of contracts as coordination mechanisms in blockchain technologies.

Future work. The currently generated smart contract skeletons require considerable manual input to make them executable. To further automate implementation steps, the DSL should provide richer inference mechanisms that automate the generation of state variables and associate those with function implementations (e.g. inferring struct properties from conditional statements).

Whether the practical use of DSLs is sufficient for non-programmers to express contractual statements and to agree on the contract semantics has to be explored further. One direction is the generation of validation code based on a more detailed intermediary representation of the ADICO rules and make use of advanced Solidity-specific concepts.

The most interesting aspect for future work will be the ability to achieve the reverse – that is, given a blockchain contract, is it possible for humans or autonomous entities to verify the actual contractual semantics and obligations? Is it possible to generate ADICO-like institutional statements based on the EVM machine code alone, without the availability of a high-level language such as Solidity? We believe that specifically this latter aspect will be central once (and if) wide-spread adoption is reached, ideally to an extent that the reuse of existing contracts becomes attractive. Addressing these questions will clarify whether institutional statements can serve as a vehicle to specify coordination mechanisms for collective adaptive systems – whether smart contracts are encoded by humans, or ultimately, by machines themselves.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[2] Ethereum Team. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper. (known as 'Ethereum White Paper'), Accessed on: 1st May 2016.

[3] Klint Finley. A $50 million hack just showed that the dao was all too human. http://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, 2016. Accessed on: 1st June 2016.

[4] Solidity. http://ethereum.github.io/solidity/. Accessed on: 1st May 2016.

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*, pages 220–242, Berlin, 1997. Springer.

[6] Elinor Ostrom. *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge University Press, New York (NY), 1990.

[7] Sue E.S. Crawford and Elinor Ostrom. A Grammar of Institutions. *The American Political Science Review*, 89(3):582–600, September 1995.

[8] Saba Siddiki, Christopher M. Weible, Xavier Basurto, and John Calanni. Dissecting Policy Designs: An Application of the Institutional Grammar Tool. *The Policy Studies Journal*, 39(1):79–103, 2011.

[9] A. Smajgl, L. Izquierdo, and M. G. A. Huigen. Rules, Knowledge and Complexity: How Agents Shape their Institutional Environment. *Journal of Modelling and Simulation of Systems*, 1(2):98–107, 2010.

[10] C. Frantz, M. K. Purvis, M. Nowostawski, and B. T. R. Savarimuthu. nADICO: A Nested Grammar of Institutions. In G. Boella, E. Elkind, B. T. R. Savarimuthu, F. Dignum, and M. K. Purvis, editors, *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Artificial Intelligence*, pages 429–436, Berlin, 2013. Springer.

[11] École Polytechnique Fédérale de Lausanne (EPFL). The scala programming language. http://www.scala-lang.org/. Accessed on: 1st May 2016.

[12] Jeremy Pitt, Julia Schaumeier, and Alexander Artikis. Coordination, conventions and the self-organisation of sustainable institutions. In David Kinny, Jane Yung-jen Hsu, Guido Governatori, and Aditya K. Ghose, editors, *Agents in Principle, Agents in Practice: 14th International Conference, PRIMA 2011, Wollongong, Australia, November 16-18, 2011. Proceedings*, pages 202–217, Berlin, 2011. Springer.

[13] Karen da Silva Figueiredo, Viviane Torres da Silva, and Christiano de Oliveira Braga. Modeling Norms in Multi-agent Systems with NormML. In Marina De Vos, Nicoletta Fornara, Jeremy V. Pitt, and George Vouros, editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems VI*, volume 6541 of *Lecture Notes in Computer Science*, pages 39–57. Springer, Berlin, 2011.