Distributed and Adversarial Resistant Workflow Execution on the Algorand Blockchain

Yibin Xu¹, Tijs Slaats¹, Boris Düdder¹, Søren Debois², and Haiqin Wu¹

 ¹ University of Copenhagen, Copenhagen, Denmark {yx,slaats,boris.d,hw@di.ku.dk}
 ² IT University of Copenhagen, Copenhagen, Denmark {debois}@itu.dk

Abstract. We provide a practical translation from the Dynamic Condition Response (DCR) process modelling language to the Transaction Execution Approval Language (TEAL) used by the Algorand blockchain. Compared to earlier implementations of business process notations on blockchains, particularly Ethereum, the present implementation is four orders of magnitude cheaper. This translation has the following immediate ramifications: (1) It allows decentralised execution of DCR-specified business processes in the absence of expensive intermediaries (lawyers, brokers) or counterparty risk. (2) It provides a possibly helpful highlevel language for implementing business processes on Algorand. (3) It demonstrates that despite the strict limitations on Algorand smart contracts, they are powerful enough to encode models of a modern process notation.

Keywords: Applications of blockchain \cdot Smart contracts \cdot Algorand \cdot Inter-institutional collaboration

1 Introduction

Blockchain technologies rose to prominence by realising decentralised financial systems and instruments [1, 5], then branched out into other domains such as supply chain management [16]. The main draw of blockchains is their ability to securely capture and track the ownership of resources [19], e.g., digital cash, real estate, and produce. Smart contracts [1] have added a second dimension to these use cases by allowing blockchains to control the valid movement of resources.

This development has drawn the interest of the Business Process Management (BPM) community [14], to which smart contracts harbor the promise of integrity-protected decentralised automation of process execution. In this community, a process is commonly defined as a structured, measured set of activities designed to produce a specific output for a particular customer or market [2]. In practice, e.g., products being traded and shipped in a supply chain and the treatment of a patient in a hospital, or a loan application process within a bank. The latter two of these examples are knowledge-intensive processes. A key approach to formalising knowledge-intensive processes is that of declarative process *notations* [15], which expresses the constraints a process must obey, as opposed to the exact sequencing of admissible activity executions, akin to the difference between an LTL formula and a Büchi automaton. In practice, declarative models tend to be more concise, and for processes subject to rules and regulations, easier to relate to those rules and regulations.

While process notations have been encoded into Solidity [1]), these are plagued by high costs and relatively low performance due to congestion of the Ethereum network and high gas prices [13]. Given the cost of transactions on the Ethereum blockchain in January 2022, creation of a declarative business process contract would cost roughly \$350 while the execution of a single event or activity would cost \$25.

In the current paper we address this weakness by exploring an encoding from the declarative Dynamic Condition Response (DCR) Graphs process notation to TEAL [5] contracts running on the Algorand blockchain. Transactions on Algorand are cheap, with a current cost, on the 14th of January 2022, of \$0.00136, and offer transaction finality in under 5 seconds. This encoding is not trivial however, as the efficiency and low cost of TEAL contracts carry limitations to the memory space and number of operations as a trade-off.

The contributions of this paper are:

- 1. We show how DCR Graphs can be efficiently stored in the limited memory space provided by TEAL and through pseudo-code show how their run-time semantics can be encoded without exceeding the operation limit;
- 2. we analyse the costs of storing and running DCR smart contracts on the Algorand blockchain based on the number of unique activities involved in the process;
- 3. we provide a prototype implementation of the encoding running on the Algorand testnet;
- 4. we discuss possible future extensions to the encoding that will allow capturing more complex and rich process descriptions.

In section 2, we proceed to discuss future work. In section 3, we shortly describe the primary attributes of the Algorand network. Section 4 introduces Dynamic Condition Response (DCR) graphs. In section 5, we show how the semantics of DCR Graphs can be encoded as smart contracts on the Algorand blockchain. Section 6 provides a financial analysis, show the maximum cost associated with our approach and how it related to earlier attempts at encoding DCR Graphs in Solidity. Finally Section 7 concludes and discusses future work.

2 Related work

Previous approaches towards process-aware blockchains [8,9,12,18] have focused on providing translations from process models into existing smart contracts languages, particularly, by translating flow-based BPMN diagrams to Solidity. A recent work [10] proposed to reduce the cost of redeployment of the smart contracts when changing the process model by a specially designed interpreter of

 $\mathbf{2}$

BPMN process models based on dynamic data structures. [11] presented a model for dynamic binding of actors to roles in collaborative processes and an associated binding policy specification language. We differ from these works by first of all, taking a declarative approach to process modelling and second of all developing a native smart contract language for processes that is directly embedded in the blockchain.

Inspired by institutional grammars, [4] proposed a high-level declarative language that focuses on business contracts, however, no implementation is provided. A high-level vision of the business artifact paradigm towards modelling business processes on a distributed ledger was given in [7]. [17] proposed a lean architecture enabling lightweight and full-featured on-chain implementations of a decentralised process execution system.

3 Algorand blockchain

Algorand [5] is a late-generation blockchain with a series of features, including high scalability and a fork-free consensus protocol based on Proof-of-Stake. Its smart contract layer (ASC1) aims to reduce the security risk of smart contracts, and adopts a non-Turing complete programming model, which natively supports transactional atomic sets and user-defined assets. These characteristics make it an intriguing smart contract platform to study.

A smart contract language called TEAL [5] is used in Algorand. TEAL is a bytecode-based stack language and is processed by the Algorand Virtual Machine (AVM), with an official programming interface for Python (called PyTeal). In addition to standard arithmetic-logical operators, TEAL also includes operators for calculating and indexing all transactions in the current atomic group, as well as IDs and fields for accessing them. When launching a transaction involving a script, the user can specify a series of parameters. The script includes cryptographic operators that calculate the hash value and verify the signature.

Applications are stateful smart contracts created with Algorand. They are given an Application ID when they are launched. Application Transactions are used to communicate with these contracts. The primary Application Transaction provides additional data that the stateful smart contract's TEAL code can pass and process.

Per transaction call, any application can check the global state of up to two other smart contracts. This is accomplished by including the application IDs of the additional stateful smart contracts in the transaction call to the stateful smart contract. This is known as the Application Array in TEAL. Currently, the developer must know how many additional applications are expected to be sent into the contract call before writing the smart contract code in TEAL.

Figure 1 shows the architecture of the stateful smart contract in Algorand. Each transaction has an Application array, which indicates what smart contract (up to two smart contracts) the transaction will be sent to; an Accounts Array (up to four accounts), which indicates what accounts have opt-in to the smart contract; an Assets Array (up to two assets), which indicates the assets that will be sent to the smart contract; an Arguments Array (up to 255 arguments), which indicates the arguments passed to the smart contract. The maximum length of the stack and scratch space is 1000 and 255 respectively.



Fig. 1. Stateful Smart Contract Transaction Call Architecture.

3.1 Limitations

4

For each smart contract, we have the following limitations:

- 1. 64 Key/Value pairs in the global state.
- 2. 64 Key/Value pairs in total in the local state of the accounts. There can be four accounts opted in to the smart contract, each of it has 16 pairs.
- 3. Max key + Value length=128 bytes
- 4. A stateless smart contract only returns the result of the execution but will not store states in the blockchain. A stateful smart contract can create and update the states stored in the blockchain.
- 5. The program (a smart contract consists of an approval program and a clean program) costs no more than 20000 operations in stateless mode. It allows at most 700 operations for each the approval and clear program of a stateful contract. In other words, it allows 700 operations for each execution of the stateful contract. Each operation costs 1, some cryptographic operations are more costly but not used in our prototypical implementation.

Note that the pairs in the local state of the accounts are read-only for other people. People do not need to Opt-in to the smart contract in order to execute the smart contract.

4 DCR Graph

DCR Graphs [6] are a formal declarative notation for describing processes. The base notation focuses on control-flow, i.e., the allowed sequencing of activities. The nodes of a DCR Graph are the executable elements, called events, which can be labelled by a labelling function. The labelling function allows multiple events to share the same label, thereby allowing process activities to occur more than once in a graph, under different constraints depending on their context.

The state of a graph is described by a marking, indicating for each event whether it (1) has been previously executed, (2) is currently pending, and (3) is currently included. The evolution of the graph is described by its edges, the relations between events. Through the relations, an event can constrain another event or have an effect on it. There are two possible constraints: the condition $(\rightarrow \bullet)$ captures that an event can not be executed unless another event has been executed some time (not necessarily immediately) before it, the milestone $(\rightarrow \diamond)$ captures that an event cannot be executed while another event is pending. There are three effect relations: the exclusion $(\rightarrow \%)$ removes an event from the process and disables any constraints it may place on other events, the inclusion relation $(\rightarrow +)$ includes an event back into the process, re-enabling any constraints it may have had, and the response relation makes another event pending $(\bullet \rightarrow)$. Pending events are obligations and must be satisfied by either executing or excluding them before a process can be considered to be in an accepting state.

Definition 1. A DCR Graph is a tuple $(E, M, L, \ell, \rightarrow \bullet, \bullet \rightarrow, \rightarrow \diamond, \rightarrow +, \rightarrow \%)$, where

- -E is the set of events
- $-M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \in \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E)$ is the marking of the graph
- -L is the set of labels
- $-\ell: E \to L$ is the labelling function
- $-\phi \subseteq E \times E \text{ for } \phi \in \{ \rightarrow \bullet, \bullet \rightarrow, \rightarrow \diamond, \rightarrow +, \rightarrow \% \}$ are respectively the condition, response, milestone, inclusion, and exclusion relations between events

For DCR Graph G with events E and event $e \in E$, we write $(\rightarrow \bullet e)$ for the set $\{e' \in E \mid e' \rightarrow \bullet e\}$, write $(e \bullet \rightarrow)$ for the set $\{e' \in E \mid e \bullet \rightarrow e'\}$ and similarly for $(e \rightarrow +), (e \rightarrow \%)$ and $(\rightarrow \diamond e)$.

An event of a DCR graph is *enabled* when (a) it is included, (b) there are no included conditions that have not been executed, and (c) there are no pending and included milestones.

Definition 2 (Enabled events). Let $G = (E, M, L, \ell, \rightarrow \bullet, \bullet \rightarrow, \rightarrow \diamond, \rightarrow +, \rightarrow \%)$ be a DCR Graph, with marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. An event $e \in E$ is enabled, written $e \in \mathsf{enabled}(G)$, iff (a) $e \in \mathsf{In}$ and (b) $\mathsf{In} \cap (\rightarrow \bullet e) \subseteq \mathsf{Ex}$ and (c) ($\mathsf{Re} \cap \mathsf{In}$) $\cap (\rightarrow \diamond e) = \emptyset$.

If an event is enabled then it can be executed. Executing an event e updates the marking of the graph by (a) adding it to the set of executed events, (b) removing it from the set of pending events and adding its responses $(e \bullet \rightarrow)$ to the set of pending events, and (c) respectively removing its exclusions $(e \to \%)$ from and adding its inclusions $(e \to +)$ to the set of included events.

6 Y. Xu, T. Slaats, B. Düdder, S. Debois, H. Wu

Definition 3 (Execution). Let $G = (E, M, L, \ell, \rightarrow \bullet, \bullet \rightarrow, \rightarrow \diamond, \rightarrow +, \rightarrow \%)$ be a DCR Graph, with marking M = (Ex, Re, In). When $e \in enabled(G)$, the result of executing e, written execute(G, e) is a new DCR Graph G' with the same events, labels, labelling function, and relations, but a new marking M' = (Ex', Re', In'), where (a) $Ex' = Ex \cup \{e\}$ (b) $Re' = (Re \setminus \{e\}) \cup (e \bullet \rightarrow)$, and (c) $In' = (In \setminus (e \rightarrow \%)) \cup (e \rightarrow +)$.

We define the language of a DCR Graph as all finite and infinite sequences of such executions, where all pending responses are eventually executed or excluded.

Definition 4 (Language of a DCR Graph). Let $G = (E, M, L, \ell, \rightarrow \bullet, \bullet \rightarrow, \rightarrow \bullet, \rightarrow +, \rightarrow \%)$ be a DCR Graph. A run of G is a finite or infinite sequence of events e_0, e_1, \ldots such that $e_i \in \mathsf{enabled}(G_i)$, $\mathsf{execute}(G_i, e_i) = G_{i+1}$, and $G_0 = G$. We call a run accepting iff for each G_i with marking $M_i = (\mathsf{Ex}_i, \mathsf{Re}_i, \mathsf{In}_i)$ and $e \in \mathsf{Re}_i \cap \mathsf{In}_i$ there exists a $j \geq i$ such that $e_j = e$ or $e \notin \mathsf{Re}_j \cap \mathsf{In}_j$.

The language $lang(G) \subseteq L^{\infty}$ of G is the set of finite and infinite sequences of labels $l_0 l_1 \cdots$ such that there is an accepting run e_0, e_1, \ldots where $\ell(e_i) = l_i$.



Fig. 2. DCR Graph of a mortgage application process adapted from [3].

As an example, Fig. 2 shows a simplified version of a loan application process encountered in industry [3] modelled as a DCR Graph. The labels of the events contain not only the name of the activity, but also the roles who are allowed to execute them. The loan application should always be assessed by the case worker, shown by the red text and exclamation mark, which denote that the event is an *initial response*. To reach this goal, the case worker must first collect documents and the customer must submit a budget, shown by the condition relations from these two events. In addition, a statistical or on-site appraisal must have been performed. Both are a condition to assess loan application, but they also mutually exclude each other, meaning that if one is executed, the other is excluded and will not block other events from executing. Submit budget also has a response relation towards the assessment, meaning that a loan application must always be assessed (again) after the customer submits a (new) budget. Finally IT may determine that the neighbourhood of the property requires an on-site appraisal. It then excludes the statistical appraisal event and includes the on-site appraisal event, which will re-enable on-site as a condition for the assessment, even if it was previously excluded by a statistical appraisal.

In the initial marking, irregular neighbourhood and assess loan application are blocked as having unsatisfied conditions. Other events are enabled as they are included and have no blocking conditions or milestones. The graph is in a non-accepting state as the assess loan application is included and a pending response.

Executing Collect documents and Submit budget will mark these events as executed. Doing a Statistical appraisal will mark itself as executed and exclude On-site appraisal, meaning that we can execute Assess loan application, which will remove the pending response and bring the graph into an accepting state.

Note that we can still execute Submit budget if new information is provided by the customer, which requires Assess loan application to be executed again.

5 Distributed DCR Graphs as Algorand smart contracts

In this section, we transform the DCR Graph into a stateful smart contract in TEAL. We eliminate the labels of the events and only keep the relationships and the IDs of the events. This design is for maintaining the anonymity of the DCR Graphs in the blockchain and saving space for more events.

For each DCR Graph, we maintain three global key/value pairs:

- GC, which records the address of the graph creator as a Byte32 String.
- -MK, which indicates the marking of the graph as a Byte16 String;
- -TEN, the total event number as an unsigned 64-bit integer.³

Each four bits of MK represents the status of an event, with the first bit describing if the event is included or excluded. The structure of the status:

- Excluded: $(xxx0)_2$;
- Included: $(xxx1)_2$;
- Pending: $(xx1x)_2$;
- executed: $(x1xx)_2$.

where x represents either 1 or 0. The number $i, i \in [0, TEN \times 5)$ bit refers to a status of the number int(i/5) event. Only CG can add events or add the relationships between the events.

We maintain two key/value pairs E and E_links for each event E. The key/value pair E indicates the account address which can execute E; E_links indicates the links between E and other events (which event's status needs to be changed after the execution of E and which events are preventing E from execution) as a Byte32 String. Each five bits of the Value represents the links of an event. The structure of the links:

- Include: $(xxxx1)_2$;

 $^{^{3}}$ Note that integer in TEAL is automatically a unit 64 integer.

- Y. Xu, T. Slaats, B. Düdder, S. Debois, H. Wu
- Exclude: $(xxx1x)_2$;

8

- Milestone: $(xx1xx)_2$;
- Condition: $(x1xxx)_2$;
- Response: $(1xxxx)_2$.

Include, Exclude, Response are out-links, meaning that after execution of E, the relevant event will be included, excluded or pended. Milestone and Condition are in-links, meaning E may not be executed if the relevant event is pended or have not executed. For example, given two events A and B with relations $A \rightarrow \bullet B$ and $B \rightarrow \% A$, B_links will indicate both $A \rightarrow \bullet B$ and $B \rightarrow \% A$. $B_links = (0100000010)_2$ assuming A indexed 1 and B indexed 2.

Given the limitations discussed in Section 3.1, our approach can have 61 events in maximum because we have 128 pairs in total and we use two key/value pairs for each event and three pairs for the graph.

We were provided with a breakdown of a database containing 22787 DCR models created by academic and commercial users; we report a summary in Figure 3. Note that the statistics show that the average number of events within a graph generated in the site is 23 and 92.5% of the graphs have an event number below 61. Therefore, the 61 event limit appears to be a promising start for practical usage.



Fig. 3. The statistic from https://www.dcrgraphs.net/.

Figure 4 shows an example of the architecture, and 1_LINK indicates the link K/V pair of event 1.

Algorithms 1, 2, 3, and 4 show the pseudo-code of the operations of adding an event, adding an relationship, executing an event, and updating the status of the events, respectively. In Algorithm 1, we add an event by creating two key/value

pairs representing the executor and the links to other events. In Algorithm 2, we add an relationship between two events by updating their links. The executor of an event can execute the event in Algorithm 3, the Algorithm will check in-links of the event to see if it is executable and then update MK via the out-links. In Algorithm 4, CG can update the status of an event.

In the codes, we are not using any operations that require a cost of more than 1. Therefore the four Algorithms are all within 700 operations when there are 61 events in maximum.

An example implementation corresponding to the graph shown in Figure 2 is in the Algorand testnet, APP-ID:59565714. The link to the Github repository for the source code is https://github.com/XU-YIBIN/DCR-Algorand.

MK: Bytes16(ABDEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA					
1_LINK: Bytes32(ABCEIQABDEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA					
Milestone: 00100					

Fig. 4. The structures of the Key/Value pairs.

6 Financial Analysis

When a smart contract is deployed to the Algorand blockchain, it is given an app ID, which is a unique identifier. Furthermore, each smart contract has its own Algorand address, which is created from this unique ID. The address allows the smart contract to function as an escrow account. In order for the smart contract

$\fbox{Algorithm 1 Add an event}$

1:	$\label{eq:Global states: GC: Graph creator, MK: Marking, TEN: Total event number} \end{tabular}$						
2:	procedure ADD AN EVENT $(TXS : Transaction \ sender, \ EC : executor)$						
Require: $TXS == GC$ and $TEN < 61$							
3:	$TEN \leftarrow TEN + 1$						
4:	Set a K/V pair A :						
	Key: <i>TEN_links</i> , Value: <i>Byte</i> 32(<i>Null</i>)						
5:	Set a Scratch value total_ops, //scratch value is run-time value.						
6:	$total_ops \leftarrow (TEN - 1) \times 2 + 3 + 1$						
	// because we have used three key/value pairs at the beginning and we are currently						
	adding one more.						
7:	if $total_ops > 64$ then						
8:	Set A as a local state in the account $[(total_ops - 64)/16 - 1]$						
9:	else Set A as a global state.						
10:	$total_ops \leftarrow total_ops + 1$						
11:	Set a K/V pair B :						
	key: TEN , Value: EC						
12:	if $total_ops > 64$ then						
13:	Set B as a local state in the account $[(total_ops - 64)/16 - 1]$						
14:	else Set B as a global state.						

Algorithm 2 Add a relationship

0
1: Global states: GC : $Graph$ creator, MK : $Marking$, TEN : $Total$ event number of the states
2: procedure ADD A RELATIONSHIP $(TXS : Transaction \ sender, \ E1 : Event1 \ I$
$E2: Event2 \ ID, \ RT: Relationship \ Type)$
Require: $TXS == GC$
3: Get a K/V pair $(E1_link,A)$.
//This pair may be from the global state or the local state, which was set wh
adding the event (using key: TEN_links).
4: Get a K/V pair $(E2_link,B)$.
//This pair may be from the global state or the local state.
5: Set a scratch value k depanding on RT (include $\rightarrow 0$, exclude $\rightarrow 1$, Milesto
$\rightarrow 2$, Condition $\rightarrow 3$, Response $\rightarrow 4$).
6: if $k = 2$ or $k = 3$ then
7: Set the $(E1-1) \times 5 + k$ -bit of B to 1.
8: else
9: Set the $(E2 - 1) \times 5 + k$ -bit of A to 1.
10: Update K/V pairs $(E1_link,A)$ and $(E2_link,B)$ using updated A and B.

Algorithm 3 Execute an event

1: Global states: GC: Graph creator, MK: Marking, TEN: Total event number 2: procedure EXECUTE AN EVENT $(TXS : Transaction \ sender, \ E1 : Event1 \ ID)$ 3: Get a K/V pair (E1,A). //This pair may from the global state or the local state. **Require:** A == TXSGet a K/V pair $(E1_link,B)$. 4: //This pair may be from the global state or the local state. for i=0 to $TXN \times 5$ do 5:Set a scratch value C_{ID} , $C_{ID} \leftarrow int(i/5) + 1$. 6: $//C_I D$ refers to the current event ID. 7:Set a scratch value $k, k \leftarrow i \mod 5$. 8: if k-st bit of B.Value=1 then 9: if k=2 then // The event C_ID milestone E1. 10:If the $(C_{ID} - 1) \times 4 + 1$ -st bit of MK is 1, then return false. 11: else if k=3 then //The event C_ID condition E1. 12:If the $(C_ID - 1) \times 4 + 2$ -st bit of MK is 0, then return false. 13:for i=0 to $TXN \times 5$ do 14: Set a scratch value C_{ID} , $C_{ID} \leftarrow int(i/5) + 1$. $//C_{ID}$ refers to the current event ID. 15:Set a scratch value $k, k \leftarrow i \mod 5$. 16:if k-st bit of B is 1 then 17:if k=0 then // The event C_ID should be included. 18: Set the $C_{ID} \times 4 + 0$ -bit of MK as 1. else if k=1 then //The event C_{ID} should be excluded. 19:20: Set the $C_{ID} \times 4 + 0$ -bit of MK as 0. 21: else if k=4 then //The event C_{-ID} should be pended. 22:Set the $C_{ID} \times 4 + 1$ -bit of MK as 1. 23:Set $(E1 - 1) \times 4 + 1$ -bit of *MK* as 0 //cancel the pending status. Set $(E1-1) \times 4 + 2$ -bit of *MK* as 1 //update the status as executed. 24:

Algorithm 4 Update the status of the event

Global states: GC : Graph creator, MK : Marking, TEN : Total event number
 procedure UPDATE STATUS(TXS : Transaction sender, E1 : Event1 ID, S : Status)

Require: TXS == GC

- 3: if S="include" then
- 4: Set $(E1-1) \times 4$ -bit of MK as 1.
- 5: else if S="exclude" then
- 6: Set $(E1-1) \times 4$ -bit of MK as 0.
- 7: else if S="pend" then
- 8: Set $((E1 1) \times 4 + 1)$ -bit of *MK* as 1.

12 Y. Xu, T. Slaats, B. Düdder, S. Debois, H. Wu

to run, there must be at least $Escrow_{overall}$ amount of microAlgos inside the smart contract, otherwise the transaction fails automatically.

$$Escrow_{global} = 100000 \times (1 + ExtraProgramPages) +(25000 + 3500) \times schema.NumUint (1) +(25000 + 25000) \times schema.NumByteSlice.$$

where Schema.NumUint refers to the global integer key-value pairs (the value size is UInt64bits); Schema.NumByteSlice refers to the global string key-value pairs. ExtraProgramPages is only needed when the compiled program exceeds 2KB, which we do not require.

The operation opt-in an account to the smart contract requires:

$$Escrow_{local} = 100000 + (25000 + 3500) \times schema.NumUint +(25000 + 25000) \times schema.NumByteSlice.$$
 (2)

schema.NumUint and schema.NumByteSlice refers to the local states.

We use one global integer key-value pair (the number of events), all other key-value pairs are String key-value pairs. ExtraProgramPages = 0. Then,

$$Escrow_{alobal} = 100000 + 28500 + 50000 \times \min(TSN \times 2 + 2, 63).$$
(3)

$$Escrow_{local_i,i \in [0,ceil((TSN \times 2 + 2 - 63)/16))}$$
(A)

$$= 100000 + 50000 \times (\min(TSN \times 2 + 2 - 63 - (i - 1) \times 16, 16)).$$
⁽⁴⁾

 $Escrow_{overall} = Escrow_{global} + Escrow_{local_{i,i} \in [0,ceil((TSN \times 2+2-63)/16))}.$ (5)

When TSN=61,

$$Escrow_{overall} = Escrow_{global} + Escrow_{local} \times 4 = 3278500 + 3450000 = 6728500.$$
(6)

As of January 14th, 2022, 1000000 microAlgos are worth \$1.36. Therefore, the maximum amount of Algo locked for deploying a DCR Graph has a value of \$9.35. Figure 5 show the relationship between the number of events in a contract and the amount of USD locked. Note that the escrow is locked in the account that starts the contract and the remaining of it is released when the smart contract is closed. There is a fee for executing the smart contract.

The fees for executing the smart contract is paid by the escrow account linked to the smart contract or can be set to be paid by the executor. Each execution below 1kB costs a fixed 1,000 microAlgos or 0.001 Algos. Larger transactions use a fee-per-byte ratio. One can also choose to use the fee-per-byte ratio to increase the probability of getting included into a new block, however at the current usage levels of the network this is unnecessary. For our implementation, both contract creation and event execution transactions remain below the 1kB limit. In addition to the locked escrow, the creation fee for a smart contract is the



Fig. 5. The relationship between the events and the USD escrowed.

same as a regular transaction, however since the DCR Graph is dynamically constructed through addEvent, addRelation, and updateStatus calls to the smart contract, creating the graph will require 1 + E + R + S transactions, where Eis the number of events, R is the number of relations, and S is the number of status updates that need to be made to set the marking of the events of the contract to their initial state. A comparison of the costs for contract creation and event execution in Algorand and Ethereum is shown in Table 1. We calculated these numbers based on the example graph used in [13] which contains 5 events, contains 11 relations, and requires 3 status changes to the marking ⁴. We observe a decrease in price by four orders of magnitude for both event execution and contract creation. If one includes the escrow, then contract creation is two orders of magnitude cheaper.

Finally, Algorand provides transaction finality in under 5 seconds, compared to approximately 3 minutes for Ethereum. While the latter is acceptable for many practical business processes, this is a notable improvement for more timecritical scenarios.

7 Conclusion

In this paper, we demonstrated how business processes can be executed on the Algorand blockchain through a translation from the declarative DCR process modelling language to TEAL smart contracts. We provided precise calculations of limitations on the size of process models and the cost of their execution. We showed that execution on the Algorand blockchain cuts costs by four orders of magnitude when compared to earlier implementations on Ethereum, bringing the use of public blockchains for business process execution back in the realm

⁴ [13] does not provide a generalised calculation of gas costs that can be used for a more thorough comparison.

14 Y. Xu, T. Slaats, B. Düdder, S. Debois, H. Wu

	Algorand	Ethereum	USD Ratio
Contract creation cost (excluding escrow)	0.02 Algo \$0.02720	717,709 gas \$349.88314	17494
Contract creation cost (including escrow)	0.7485 Algo \$1.01796	717,709 gas \$349.88314	467
Event execution	0.001 Algo \$0.00136	54,496 gas \$26.56690	19534

Table 1. Costs and escrow for DCR contract creation and event execution on Algorand and Ethereum based on the example used in [13]. USD prices based on exchange rates on the 14th of January 2022. The calculated dollar cost for Ethereum transactions use a gwei/gas ratio of 150. Event execution cost in Ethereum is given as the mean of the 5 executions reported in [13].

of reasonable possibilities. We implemented a prototype that demonstrates the feasibility of our approach and allows for future extensions.

In future work, we intend to extend the prototype, lift current limitations and implement more advanced features of the DCR language. In particular, we will extend the current 61 event limit by creating multiple linked smart contracts that can read each others global states. This operation is supported in TEAL by indicating multiple smart contracts in the Application Array of the transactions.

Currently our implementation describes an event by only using an event ID. This is to preserve privacy and save memory space. When the graph is extended by using multiple smart contracts, we may use space to store more information on the events such as their name and a description.

DCR Graphs support various advanced features such as notions of (logical) time, data-constraints, replication, and more advanced assignments between events, roles, and users [13], we plan to add these to our encoding in the future which will allow for the description and execution of more complex processes.

References

- 1. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
- Davenport, T.H.: Process Innovation: Reengineering Work Through Information Technology. Harvard Business School Press, Boston, MA, USA (1993)
- Debois, S., Hildebrandt, T., Slaats, T.: Concurrency and asynchrony in declarative workflows. In: BPM 2016. LNCS, vol. 9253. Springer, Cham (2016)
- Frantz, C.K., Nowostawski, M.: From institutions to code: Towards automated generation of smart contracts. In: FAS*W. pp. 210–215. IEEE (2016)
- Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: SOSP 2017. pp. 51–68 (2017)

- Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: Honda, K., Mycroft, A. (eds.) PLACES 2010. EPTCS, vol. 69, pp. 59–73 (2010)
- Hull, R., Batra, V.S., Chen, Y.M., Deutsch, A., Heath III, F.F.T., Vianu, V.: Towards a shared ledger business collaboration language based on data-aware processes. In: ICSOC. pp. 18–36. Springer (2016)
- 8. Klinger, P., Bodendorf, F.: Blockchain-based cross-organizational execution framework for dynamic integration of process collaborations. In: WI (2020)
- Ladleif, J., Weske, M., Weber, I.: Modeling and enforcing blockchain-based choreographies. In: BPM. pp. 69–85. Springer (2019)
- López-Pintado, O., Dumas, M., García-Bañuelos, L., Weber, I.: Interpreted execution of business process models on blockchain. In: EDOC. pp. 206–215. IEEE (2019)
- López-Pintado, O., Dumas, M., García-Bañuelos, L., Weber, I.: Controlled flexibility in blockchain-based collaborative business processes. Information Systems p. 101622 (2020)
- López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: a business process execution engine on the ethereum blockchain. SPE 49(7), 1162–1193 (2019)
- Madsen, M.F., Gaub, M., Høgnason, T., Kirkbro, M.E., Slaats, T., Debois, S.: Collaboration among adversaries: distributed workflow execution on a blockchain. In: SCFAB 2018 (2018)
- Mendling, J., Weber, I., Aalst, W.V.D., Brocke, J.V., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C.D., Dumas, M., Dustdar, S., et al.: Blockchains for business process management-challenges and opportunities. ACM TMIS 9(1), 1–16 (2018)
- Pesic, M., Schonenberg, H., van der Aalst, W.M.: DECLARE: Full Support for Loosely-Structured Processes. In: EDOC 2007. pp. 287–287. IEEE (oct 2007)
- Saberi, S., Kouhizadeh, M., Sarkis, J., Shen, L.: Blockchain technology and its relationships to sustainable supply chain management. International Journal of Production Research 57(7), 2117–2135 (2019)
- Sturm, C., Szalanczi, J., Schönig, S., Jablonski, S.: A lean architecture for blockchain based decentralized process execution. In: Daniel, F., Sheng, Q.Z., Motahari, H. (eds.) BPM. pp. 361–373. Springer (2019)
- Tran, A.B., Lu, Q., Weber, I.: Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In: BPM. pp. 56–60 (2018)
- 19. Zakhary, V., Amiri, M.J., Maiyya, S., Agrawal, D., Abbadi, A.E.: Towards global asset management in blockchain systems (2019)