Collaboration among Adversaries

Madsen, Mads Frederik; Gaub, Mikkel; Høgnason, Tróndur; Kirkbro, Malthe Ettrup; Slaats, Tijs; Debois, Søren

Publication date:
2018

Document version
Publisher's PDF, also known as Version of record

Collaboration among Adversaries: Distributed Workflow Execution on a Blockchain

Mads Frederik Madsen
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S,
Denmark
mfrm@itu.dk

Mikkel Gaub
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S,
Denmark
mikg@itu.dk

Tróndur Høgnason
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S,
Denmark
thgn@itu.dk

Malthe Ettrup Kirkbro
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S,
Denmark
maek@itu.dk

Tijs Slaats
University of Copenhagen
Emil Holms Kanal 6
2300 Copenhagen S,
Denmark
slaats@di.ku.dk

Søren Debois
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S,
Denmark
debois@itu.dk

ABSTRACT

We study distributed declarative workflow execution in an adversarial setting. In this setting, parties to an agreed-upon workflow do not trust each other to follow that workflow, or suspect the other party might misrepresent proceedings at a later time. We demonstrate how distributed declarative workflow execution can be implemented as smart contracts, guaranteeing (I) enforcement of workflow semantics, and (II) an incontrovertible record of workflow execution history. Crucially, we achieve both properties *without* relying on a trusted third party.

The implementation is based on the Ethereum blockchain, inheriting the security properties (I) and (II) from the guarantees given by that chain. A recurring challenge for both the implementation and the analysis is the cost of operations on Ethereum: This cost must be minimised for honest parties, and an adversary must be prevented from inflicting extra cost on others.

1. INTRODUCTION

Mutually distrusting organisations must often collaborate, as illustrated by the following example. On the Danish labour market employer-employee disputes are resolved not by the parties themselves, but by the umbrella organisations for respectively Danish employers (abbreviated here "DE") and Danish unions (abbreviated here "DU")[1]. A dispute may be resolved through negotiations between the two parties, or if negotiations break down, in court.

[1]The actual Danish names are "Dansk Arbejdsgiverforening" (DE) and "Landsorganisationen" (DU).

Symposium on Foundations and Applications of Blockchain (FAB '18) March 9, 2018, Los Angeles, California, USA.

Given their conflicting interests, DU and DE are mutually distrusting collaborators. They follow an agreed-upon process when negotiating a dispute, a process which defines simple things like who proposes meeting dates, who submits which document to whom and how, etc.

However, depending on the strength of their respective cases, they may not have equal incentives to follow this process. If an employee has a strong claim to unpaid salary, DE may be less forthcoming in responding to meeting date proposals. Conversely, if an employer is planning legal but unpleasant mass firings, DU may similarly stall the process. Should a case go to court, either party's intransigence may have legal repercussions.

This reluctant collaboration is an example of a cross-organisational workflow between adversaries. System support for such a workflow must provide two key guarantees:

(I) **Workflow correctness.** The system must enforce the agreed-upon workflow, so that no party can obtain an advantage by acting out of turn or failing to fulfil an obligation to act.

(II) **Consensus on history.** The system must provide an incontrovertible record of execution, e.g., to decide in court which party did in fact violate the agreed upon workflow.

The usual way to achieve (I) and (II) is having participants agree on a trusted third party. This third party verifies that the actions taken are within the bounds of the agreement, and meticulously records the proceeding of the case. However, such a third party is not always practical: It may be difficult for the parties to agree on one, and it may be expensive to retain one, especially at large case volumes.

In this paper, we show how (I) and (II) may be achieved without a trusted third party by implementing an executable workflow specification as an Ethereum smart contract.

Our solution is based on recent advances in executable workflow specifications on the one hand and blockchain technologies on the other. A blockchain can be used as a mechanism to produce a trusted, immutable record of workflow execution. E.g. if DU and DE were to store the history of

their common processes on a blockchain, they could both trust this history to be correct with very high probability.

Executable workflow specifications. Agreeing on a record of workflow history is only a part of the puzzle. We must also enforce adherence to the agreed-upon workflow, i.e., the rules governing the exact order in which work can be done. Instead of encoding a workflow directly as a part of the source code of the system, it is typically modelled separately in a *workflow notation* such as BPMN [32], Workflow Nets [1], DECLARE [39], DCR graphs [7, 10], GSM [24], or CMMN [31]. In the best case, such a model is executed by an execution engine embedded in the overall system, enabling straightforward adaptation of work practices by changing the model rather than redeveloping the system itself.

Traditionally, workflow notations have been flow-based, describing processes in a style similar to transition systems, representing precisely the steps that one may go through to satisfy the goals of the process. Such notations work well for strict production processes with little variation, but when applying them to knowledge intensive processes [11], which usually allow a large degree of flexibility and many different paths towards the goals of the process, the models tend to become overly complex and unreadable [39].

Declarative process notations [39, 19, 31, 38] address this deficiency by capturing not explicit flow but rather the constraints and goals of a process, letting the system deduce the allowed paths to the goal. As shown in [20], the declarative approach is highly relevant in the case of DU and DE, whose processes are strongly knowledge intensive.

A declarative process model may be implemented as a *smart contract* [36, 40]: a blockchain where blocks represent not only a common history, but also contracts in the form of executable code. For example, DU and DE have agreed that DU will always propose meetings first; encoding this rule in a smart contract, we can ensure that any attempts to add new events in violation of this rule are rejected.

Contributions. We show that a declarative workflow engine can be employed in an adversarial setting by embedding it on a blockchain as smart contracts. We demonstrate how this approach can be implemented in practice on the Ethereum [40] blockchain, using the smart contract language Solidity and the process execution semantics of DCR graphs [7, 10]. This implementation guarantees (I) correctness with regard to the agreed-upon workflow and (II) the recording of an incontrovertible history. Of course, these guarantees extend no further than the security of the underlying Ethereum blockchain technology, i.e., we assume no adversary can construct Ethereum blocks faster than the honest nodes.

Cost is an issue: Both the cost of participation in the workflow, and the possibility of attacks that inflict cost on honest nodes. In particular, to minimise cost caused by the Ethereum smart contract model (where each computational operation incurs a micro-fee), our cost-effective implementation required both counter-intuitive contract design as well as other non-trivial performance enhancements.

1.1 Related Work

In [41, 16, 30] the authors propose encoding workflows as a smart contract on a blockchain. An implementation of these ideas was given in Caterpillar [28]. In these works, workflows are modelled by BPMN diagrams [32]. This choice of notation clearly separates it from the present work: rather than structured, flow-based processes, we apply the approach to declarative process notations, thereby providing support for knowledge-intensive processes.

In [15], the authors introduce a high-level language, inspired by institutional grammars, that can be compiled into Solidity code. The notation has a declarative feel to it, but describes business contracts rather than workflows. Moreover, the authors do not provide an implementation.

In [23], the authors argue for the suitability of the business artefact paradigm towards modelling business processes on a distributed ledger. The paper lays out their vision, but does not go into detail on neither exact syntax or semantics, nor the exact guarantees offered by smart contracts.

In [20], the DU and DE case was studied in the context of declarative workflow specifications, but relying on a trusted third party for their collaboration. We use this collaboration as a running example; there is otherwise no special relation between the present and this older work.

Both of the properties (I) and (II) are closely related to classical security properties. It was demonstrated in [5] that a workflow notation may encompass security policy specifications. Enforcing distributed adherence to a workflow definition is related to enforcing distributed adherence to a security policy, e.g., [4, 33]. Achieving consensus on the history of a distributed workflow execution is reminiscent of distributed monitoring, e.g., [43, 25].

2. ETHEREUM

Ethereum [42, 40] is a blockchain extended with user-created code and arbitrary data encapsulated in smart contracts. When a transaction is included in a block, part of the verification of that block comprises running the code specified by the transaction, mutating the state of the contract accordingly.

This code is executed on the Ethereum Virtual Machine (EVM), in which each operation has an associated cost denoted in *Gas*. Once the sum of Gas has been calculated for an execution, it is paid for in the Ethereum cryptocurrency *Ether* by the user calling the code, at an Ether/Gas rate specified by that user. This rate allows miners to prioritise those calls paying the most.

The EVM is in principle Turing-complete [42]. However, all computations are in practice finite, limited by the amount of Gas that a caller is willing to spend.

Ethereum allows one to verify the existence of specific source code on the blockchain, whether it has been run, and whether a run was completed successfully or not. Moreover, Ethereum certifies that code was executed as specified, and that only authorised parties execute contract calls [42, 40]. This means that when implementing workflows as smart contracts, any participant can be certain that the source code is unchanged and that every execution is validated with respect to both the contract logic and execution rights.

Like the Bitcoin blockchain, the Ethereum blockchain relies on mining being hard to ensure that the probability of an attacker overtaking the main chain, rewriting history, is low. However, whereas the Bitcoin blockchain and variants has seen work on analysing under what circumstances and with what probabilities that might happen [3, 27, 35, 26, 6, 2, 37, 18] we are unaware of similar analyses for Ethereum.

3. DCR GRAPHS

In this Section, we recall DCR Graphs, a vehicle for specifying admissible sequences of event executions. A DCR Graph specifies an "agreed-upon" workflow, where the events are the activities of the workflow. A DCR Graph comprises *events* (nodes) and *relations* between events (edges); events have state which is recorded in a *marking*. Relations indicate how executability of one event may depend on the states of others, and how execution changes such states.

DEFINITION 1 (DCR GRAPH [19]). *A* DCR Graph *is a tuple* $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ *where*

- E *is a finite set of* events*, the nodes of the graph.*
- R *is the edges of the graph. Edges are partitioned into five kinds:* conditions $(\rightarrow\bullet)$*,* responses $(\bullet\rightarrow)$*,* milestones $(\rightarrow\diamond)$*,* inclusions $(\rightarrow+)$*, and* exclusions $(\rightarrow\%)$*.*
- M *is the* marking *of the graph, a triple* $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *of sets of events, respectively the previously executed* (Ex)*, the currently pending* (Re)*, and the currently included* (In) *events.*

When G is a DCR Graph, we write, e.g., $\mathsf{E}(G)$ for the set of events of G, as well as, e.g., $\mathsf{Ex}(G)$ for the executed events in the marking of G.

We give in Figure 1 an excerpt of the workflow of DU and DE reported in [20]. The events are nodes in the graph; the marking of each event is shown graphically: Hold Meeting is pending, viz. the blue exclamation mark; both Accept events are excluded viz. the dashed border.

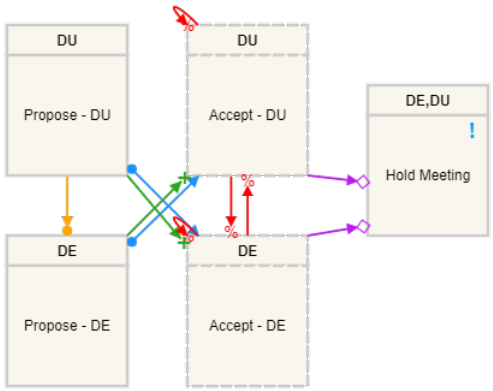

Figure 1: The DU/DE example—a DCR model of a cross-organisational workflow

By default, every activity may execute any number of times. We regulate the sequencing of such activity executions by adding relations between activities. There are five such relations: Three which mutate the state of some events when another executes, and two which constrain the ability of one event to execute depending on the state of others.

3.1 Execution of Events

To specify what happens when an event executes, we have the *response*, *inclusion*, and *exclusion* relations.

First, the *response*. When either DE or DU proposes a date, the other is required to eventually accept one. The blue *responses* $(\bullet\rightarrow)$ from Propose - DU to Accept - DE and Propose - DE to Accept - DU model this requirement: Executing the first event makes the second event *pending*.

The red *exclusions* $(\rightarrow\%)$ temporarily remove events from the process. This can be both an event removing itself after being executed, as is the case for each instance of Accept, or an event removing another event, exemplified by Accept - DU removing Accept - DE and vice versa. We say that such a removed event is *excluded*, indicated by a dashed border, as seen in the two Accept events.

Exclusions are dynamic and may be reverted: When DU or DE proposes new dates, the other is expected to accept these dates again. This is modelled through the green *inclusions* $(\rightarrow+)$ from Propose - DU to Accept - DE and Propose - DE to Accept - DU. Because Accept - DU and Accept - DE are excluded (dashed border), either requires its including event to happen before it can itself happen.

We formalise the notion of executing an event.

Notation. For a binary relation $\rightarrow \subseteq X \times Y$ and set Z, we write "$\rightarrow Z$" for the set $\{x \in X \mid \exists z \in Z.\ x \rightarrow z\}$, and similarly for "$X \rightarrow$". For singletons we usually omit the curly braces, writing $\rightarrow e$ rather than $\rightarrow\{e\}$.

DEFINITION 2 (EXECUTION [19]). *Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ be a DCR Graph with marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. If we* execute *$e$ in G, we obtain the* resulting *DCR graph $(\mathsf{E}, \mathsf{R}, \mathsf{M}')$ with $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ defined as follows.*

1. $\mathsf{Ex}' = \mathsf{Ex} \cup e$
2. $\mathsf{Re}' = (\mathsf{Re} \setminus e) \cup (e\bullet\rightarrow)$
3. $\mathsf{In}' = (\mathsf{In} \setminus (e\rightarrow\%)) \cup (e\rightarrow+)$

That is, to execute an event e one must: (1) add e to the set Ex of executed events; (2) update the currently required responses Re by first removing e, then adding any responses required by e; and (3) update the currently included events by first removing all those excluded by e, then adding all those included by e.

3.2 Enabled Events

Not all events in a graph are necessarily allowed to execute. To specify which events are in fact executable we have *conditions* and *milestones*. A condition indicates that when the source is included but not executed, the target cannot execute. For example, by convention, DU is always the first to propose dates. This is modelled by the condition relation$(\rightarrow\bullet)$ between Propose - DU and Propose - DE.

When dates have been proposed but not yet accepted, the meeting cannot be held. The milestone relations $(\rightarrow\diamond)$ from Accept - DU and Accept - DE to Hold Meeting ensure this: a milestone indicates that whenever the source is included and pending, the target cannot execute. In the diagram, the Accept events are not yet pending. This is intentional: DU and DE may skip proposing dates and hold ad hoc meetings.

Unlike the condition relation, an event constrained by a milestone can become blocked again. In our example, if a date was accepted but later new dates are proposed, accepting dates becomes pending again, blocking Hold Meeting.

We give formal meaning to these relations.

DEFINITION 3 (ENABLED EVENTS [19]). *Suppose $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ is a DCR Graph with marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. We say that an event $e \in \mathsf{E}$ is* enabled *and write $e \in \mathsf{enabled}(G)$ iff (a) $e \in \mathsf{In}$, (b) $\mathsf{In} \cap (\rightarrow\bullet e) \subseteq \mathsf{Ex}$, and (c) $\mathsf{In} \cap (\rightarrow\diamond e) \subseteq \mathsf{E}\setminus\mathsf{Re}$.*

That is, enabled events (a) are included, (b) have their included conditions already executed, and (c) have no included pending milestones. The enabled events for the DCR Graph in Figure 1 are Propose - DU and Hold Meeting.

General DCR Graphs have labelled events, allowing distinct events to exhibit the same observable activity, a detail we have elided in the current paper. In the general case, DCR Graphs express the union of regular and ω-regular languages [10].

3.3 Distributed DCR Graphs

Distributed implementations of DCR Graphs were studied in [22] and [9]. In both cases, the core idea is that workflows are partitioned in subsets of events, with each participant owning a particular subset. The owner of an event is responsible for maintaining the marking (M, Definition 1) of that event. Moreover, only the owner of an event can execute it (Definition 2).

Since executing one event may modify others via exclusion, inclusion and response arrows (Definition 2), whenever a party executes an event, it may have to notify owners of affected events. E.g., in the DU/DE example (Figure 1), events are naturally owned by either DU or DE as indicated at the top of each event. The event Propose - DU is owned by DU and so can only be executed by DU; however, executing this event includes the event Accept - DE, and so DU must notify DE whenever it executes Propose - DU, in order that DE may toggle the state of Accept - DE to included.

Similarly, before executing an event, the owner must verify that the event is enabled (see Definition 3). Whether an event is enabled is a function of the marking of other events via condition or milestone relations, hence the owner may have to query owners of such other events. E.g., in the DU/DE example, DE cannot execute Propose - DE before querying DU about the state of Propose - DU because of the condition relation from the latter to the former.

As queries for enabledness may interleave with effects of an execution, distributed implementations of DCR Graphs generally need some form of concurrency control [9].

4. DISTRIBUTED DCR GRAPHS AS ETHEREUM SMART CONTRACTS

In this section, we consider in the abstract an implementation of distributed DCR Graphs as Ethereum smart contracts. We shall see how such an implementation achieves the goals (I) and (II) of Section 1 *provided* an adversary has no feasible attack on the Ethereum blockchain.

The naive implementation of DCR Graphs as Ethereum smart contracts is to simply implement a contract comprising a DCR Graph (Definition 1) represented as an Ethereum data structure, and calls for computing execution and enabled events (Definitions 2 and 3). Only the owner of an event has access rights to execute that event. This appealingly simple idea turns out to mask considerable pitfalls, in particular regarding who bears the cost of executing that call. In this section, we analyse this situation.

4.1 Cost of Relations

Previous treatments of distributed DCR graphs [9, 22] do not emphasise ownership of *relations*. Adding a relation to a DCR Graph induces additional computation in either enabledness (condition, milestone) or effect of execution (inclusion, exclusion, response). On Ethereum, additional computation translates directly to additional cost, so an adversary can inflict cost on honest parties if he can add new relations. For example, adding 100 distinct conditions $A_i \rightarrow\bullet\ X$ to some event X would increase the cost of computing enabledness of X by 100 additional checks whether each A_i is executed or excluded.

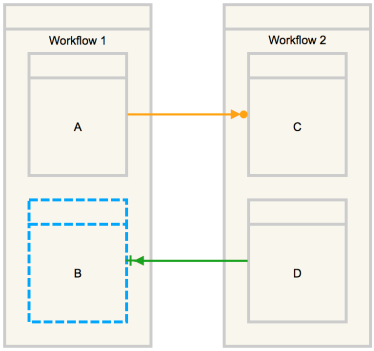

Figure 2: Inter-workflow relations

For *conditions and milestones*, each such relation induces computational cost at the owner of the *target* event. For example, in Figure 2, Workflow 2 must consult Workflow 1 to learn the state of A before it can execute event C. In general, adding an incoming relation such as $A \rightarrow\bullet\ C$ increases the cost of computing enabledness of its target C. To avoid cost-inflicting attacks, only the owner of the target C should be allowed to add incoming relations to it.

However, because executing D requires an update of the state of B, if that update is to be performed by the owner of B, there is again an opportunity for an adversary to inflict cost. In that case, we must again require that only the owner of B and D jointly may add relations.

We summarise where adding relations incurs cost Table 1.

Added relation	cost on A	cost on B
$A \rightarrow\bullet\ B$		✓
$A \rightarrow\diamond\ B$		✓
$A \rightarrow+\ B$	✓	
$A \rightarrow\%\ B$	✓	
$A \bullet\rightarrow B$	✓	

Table 1: Incurred cost of added relations

4.2 Correctness

In general, adding relations to a workflow can make that workflow both more and less restrictive [8, 10]. For example, in Figure 2, the condition relation (top) means that Workflow 2 must wait for Workflow 1 to execute A before it can execute activity C. If we imagine we have just added that condition, the new combined workflow has less behaviour than the old one, but no new behaviour. Thus, an adversary who can add relations can mount a potential denial-of-service attack by adding enough relations that the resulting combined workflow has *no* behaviour left.

Conversely, adding inclusions and exclusions can make a workflow less restrictive [8]. Without the inclusion relation

(bottom) in Figure 2, Workflow 1 can never execute activity B. If we imagine we have just added that inclusion, the new combined workflow has *more* behaviour than the original one, since the new one admits the sequence DB which the old one did not. This means that an adversary who can add relations can violate correctness of the original workflow. E.g. if the activity B were "pay out lump sum", the adversary has succesfully orchestrated a payout in violation of the original workflow policy.

Assume a correct implementation of (1) the computation of enabled events and (2) the effect of executing event in Solidity. Assume moreover that this implementation is used for implementing the distributed workflow in such a way that each party to the workflow can execute only the events they own, and only when these events are enabled. In this case, running this implementation on Ethereum, we get an implementation of the workflow which *automatically* achieves the goals of workflow correctness (I) and consensus on history (II) *provided* the adversary cannot produce valid blocks fast enough to outpace the Ethereum miner network.

Note that in workflows with more than two participants, we do not preclude colluding actors *within* the bounds of concurrent workflow semantics. In such a workflow, two or more participants could mount a denial-of-service attack against other participants by coordinating executions of activities on the same block, thereby skipping states in which specific activities were enabled. This is an inherent consequence of allowing concurrent executions of activities in DCR-graphs, and *not* a violation workflow correctness (I).

5. COST REDUCTIONS

Our practical experiments have revealed two major insights about executing DCR Graphs on Ethereum:

1. It is indeed feasible to implement distributed workflows in an adversarial setting on the Ethereum blockchain.
2. However, to keep costs manageable, our implementation must take some counter-intuitive design decisions, including implementing only one contract and implementing set operations as bitvectors.

DCR Graphs as presented in Section 3 are simple enough that the core data structures (relations and markings, Definition 1) as well as operations on them (execution and enabledness, Definitions 2 and 3) are straightforward to implement in contemporary programming languages.

A naive implementation implements DCR Graphs straightforwardly as an Ethereum contract for each workflow instance, representing marking and relations straightforwardly using standard data structures. This naive implementation has two shortcomings:

1. The Gas costs of Ethereum are dominated by the price of creating a smart contract, which is an order of magnitude more expensive than other operations [42].
2. The cost of computing enabledness respectively execution grows linearly with the number of incoming respectively outgoing relations.

5.1 Relations

To reduce the impact of additional relations on the cost of computing enabledness and execution, we exploit that the core EVM datatype is a 256-bit value, noting that the core operations of DCR Graphs (Definitions 3 and 2) are all simple set-manipulations and can be implemented efficiently as *bit vectors*.

Our prototype for this reason assumes at most 256 events in a DCR Graph, an assumption that is both practically reasonable [29, 34], and straightforward to remove if necessary.

For such fixed-size bit vectors, we get an upper bound of the cost of executing an activity: execution is implemented as a static check of the legality of the execution, followed by 3 bitwise-operations between bit vectors representing relations. We give the implementation of the enabledness computation in Listing 1; we encourage the reader to compare that listing, in particular lines 16, 20–21, and 25–27 to the clauses (a)-(c) in Definition 3.

```
1  function canExecute(uint256 wfId, uint256 activity)
2    public constant returns (bool)
3  {
4    var workflow = workflows[wfId];
5    uint32 i;
6
7    // sender address must have execute rights
8    for (i = 0; i < workflow.authAccounts.length; i++)
9      if (workflow.authAccounts[i] == msg.sender)
10       break; // sender authorised
11
12   if (i == workflow.authAccounts.length)
13     return false; // sender not authorised
14
15   // activity must be included --- Def. 3(a)
16   if ((workflow.included & (1<<activity)) == 0)
17     return false;
18
19   // all included conditions executed --- Def. 3(b)
20   if(workflow.conditionsFrom[activity] &
21      (~workflow.executed & workflow.included) != 0)
22      return false;
23
24   // no included milestones pending --- Def. 3(c)
25   if(workflow.milestonesFrom[activity]
26      & (workflow.pending & workflow.included) != 0)
27      return false;
28
29   return true;
30 }
```

Listing 1: Enabled computation

Besides the optimisations we have mentioned so far, our prototype implementation uses additional tricks to minimise Gas costs, notably packaging call data to conserve storage space. We refer the interested reader to [17], which contains additional implementation detail.

As mentioned in Section 2, execution of Ethereum smart contracts is paid for by setting an exchange rate between Gas, the cost of execution instructions, and the cryptocurrency Ether. We compare the cost of the naive and optimised implementations in Table 2.

Note that the cost of executing some activities actually *increases* from naive the to the optimised implementation: the bit vector implementation give lower Gas cost on execution only when events have many relations. The DU/DE example is too small to exhibit this effect; however, practical workflows tend to have many more relations [21].

Event	Naive		Optimised	
	GAS	USD*	GAS	USD*
1. Initialisation**	2,185,061	14.582	717,709	4.790
2. Propose - DU	61,126	0.408	66,293	0.442
3. Propose - DE	62,592	0.418	52,615	0.351
4. Accept - DU	46,126	0.308	51,293	0.342
5. Accept - DE	46,226	0.308	52,615	0.351
6. Hold Meeting	37,353	0.249	49,665	0.331
Sum	2,392,258	16.273	990.190	6.608

* Prices in USD are computed from average Gas- and Ether prices at the time of writing [13, 14].

** Prices for the naive implementation includes contract creation and workflow creation; prices for the optimised implementation only workflow creation.

Table 2: Cost comparison, naive and optimised implementation.

5.2 Contract Creation & Access Control

The cost of creating an instance of the DU/DE example workflow is given in Table 2, column "Naive". Notice that creating the contract, "initialisation" is two orders of magnitude more expensive than subsequent event executions.

To reduce this cost, we propose a *mono-contract implementation*, that is, a single contract which hosts *all* workflows, and new workflows can be added at any point after contract creation. In this mono-contract implementation methods each take an index of the workflow to work on. In such a setting, the cost overhead for creating a contract is incurred only *once*[2]: As subsequent workflows are hosted by this single contract, the cost of creating a contract does not reoccur. The cost of constructing a new workflow is reduced substantially, see the column "optimised" in Table 2.

The mono-contract provides access control by accepting, on workflow creation, a list of public keys/addresses that are authorised to subsequently execute events; the implementation manually checks that the caller is authorised before executing an event. Because state in Ethereum can only be changed through contract calls, this mechanism provides complete mediation: there is no way to alter the state of running workflows without going through the contract operations. Returning to the code for the enabledness computation in Listing 1, we see access control computed in line 7–10, using the Ethereum provided `msg.sender` constant.

As mentioned, workflow creation is an order of magnitude cheaper in the optimised implementation. Moreover, in the naive implementation, workflow creation is two orders of magnitudes more expensive than event executions; in the optimised implementation only one.

6. IMPLEMENTATION

We have implemented a software tool which converts a DCR Graph to a Solidity smart contract. To show that our DCR engine can be used in practice, we have implemented a graphical user interface (GUI), where users can create workflows and execute activities on a deployed Ethereum contract. We host the source code of the contracts at https://github.com/DCReum/dcreum.github.io for perusal, and the GUI for anyone to use at https://dcreum.github.io. An Ethereum node and client are required to view and use the GUI. We recommend Parity alongside the Google Chrome extension Parity Ethereum Integration.

Multiple high-level languages compiling to EVM bytecode exist; our implementation was done in the statically typed object-oriented language Solidity. However, compared to main-stream programming languages, in EVM/Solidity we have to contend additionally with quirks of the Solidity interpreter. For example, Solidity limits the number of variables allowed in scope at one time, as these are always kept on the stack, and the EVM only allows access to the 16 topmost items [42]. Other limitations include externally available functions not being allowed structs or nested arrays as arguments or return type.

Ethereum execution causes a delay in event execution, as the network has to process and accept such an execution. We can have the acceptance of a transaction (event execution) prioritised by offering above-market Gas prices. Unless we send the request at almost exactly the time of new block propagation, in experiments run late spring 2017, our requests have been included in the next mined block when paying market Gas prices. However, even though a block is accepted, it may still find itself on a less difficult chain, and thus eventually discarded. In general, like in Bitcoin and other blockchain-based transactional systems, one must wait some number of blocks before one can reasonably assume that the transaction is permanently included.

The frequency of event executions is bounded by the (dynamic) Gas limit [42]. This limit is currently at 6,718,941, which for the DU/DE example (Figure 1) in theory would allow between 101 and 135 executions, depending on the exact activity executed. If we consider instead single-participant, non-concurrent executions, the limit is the mining time, which should average 12 seconds, although at the time of writing, the average for the last 5000 blocks is ca. 30 seconds. In general, it has been our experience that mining time varies between a few seconds and several minutes. We estimate we have seen an average of 1-2 executions per minute at market Gas prices.

7. CONCLUSION

We have demonstrated how to implement distributed declarative workflow execution in an adversarial setting, without the assistance of a trusted third party, by implementing DCR Graph declarative process models as Solidity contracts running on the Ethereum blockchain. Within the the security guarantees given by this blockchain, this implementation guarantees that the execution does follow the agreed-upon workflow—the DCR Graph—(I) and that the sequence of executions recorded on the blockchain is incontrovertibly the actual recorded history (II).

Cost is an issue, both because an adversary must be prevented from inflicting cost on an honest party, and because cost of contract execution is high enough that we must optimise. Particularly helpful optimisations are the mono-contract and bitvector representation of sets and relations.

We have demonstrated the economic feasibility of the implementation: see actual costs in Table 2. Moreover, we have discussed bounds on delay and frequency of event executions in Section 6, estimating that the Ethereum blockchain can likely sustain 1-2 execution per minute at market prices.

[2]This contract was created in the transaction [12] at a cost of 2,976,162 Gas/USD 8.73.

8. REFERENCES

[1] W. M. P. v. d. Aalst. Verification of Workflow Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, ICATPN '97, pages 407–426, London, UK, UK, 1997. Springer-Verlag.

[2] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A Sharded Smart Contracts Platform. *arXiv:1708.03778*, 2017.

[3] S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better—how to make bitcoin a better currency. In *Proc. of FC '12*, pages 399–414. Springer, 2012.

[4] D. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu. Monitoring usage-control policies in distributed systems. In *Proc. of TIME '11*, pages 88–95. IEEE, 2011.

[5] D. A. Basin, S. Debois, and T. T. Hildebrandt. In the Nick of Time: Proactive Prevention of Obligation Violations. In *Proc. of CSF '16*, pages 120–134. IEEE, 2016.

[6] E. Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, 2016.

[7] S. Debois and T. Hildebrandt. The DCR Workbench: Declarative Choreographies for Collaborative Processes. In *Behavioural Types: from Theory to Tools*, River Publishers Series in Automation, Control and Robotics, pages 99–124. River Publishers, June 2017.

[8] S. Debois, T. Hildebrandt, and T. Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. pages 143–160, 2015.

[9] S. Debois, T. T. Hildebrandt, and T. Slaats. Concurrency and Asynchrony in Declarative Workflows. In *Proc. of BPM '15*, volume 9253 of *LNCS*, pages 72–89. Springer, 2015.

[10] S. Debois, T. T. Hildebrandt, and T. Slaats. Replication, refinement & reachability: complexity in dynamic condition-response graphs. *Acta Informatica*, Sept. 2017.

[11] C. Di Ciccio, A. Marrella, and A. Russo. Knowledge-intensive processes: characteristics, requirements and analysis of contemporary approaches. *J. on Data Semantics*, 4(1):29–57, 2015.

[12] Mono-contract creation transaction. https://etherscan.io/tx/0x003fb07eb74b4a2557dc48fa-8e7799e481f98e0c4ed0857ce646dbe3b9b90cda.

[13] Ethereum average gasprice chart. https://etherscan.io/chart/gasprice.

[14] Ethereum/us dollar (eth/usd) price chart. https://www.coingecko.com/en/price_charts/ethereum/usd.

[15] C. K. Frantz and M. Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *Proc. of FAS*W '16*, pages 210–215, Sept 2016.

[16] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber. Optimized execution of business processes on blockchain. In *Proc. of BPM '17*, pages 130–146. Springer, Cham, 2017.

[17] M. Gaub, M. E. Kirkbro, F. Madsen, and T. Högnason. Consensus in declarative process models using distributed smart-contracts. 2017.

[18] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. On the Security and Performance of Proof of Work Blockchains. In *Proc. of SIGSAC '16*, CCS '16, pages 3–16, New York, NY, USA, 2016. ACM.

[19] T. Hildebrandt and R. R. Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In *Post-proc. of PLACES '10*, volume 69 of *EPTCS*, pages 59–73, 2010.

[20] T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Designing a cross-organizational case management system using dynamic condition response graphs. In *Proc. of EDOC '11*, pages 161–170. IEEE, 2011.

[21] T. T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Nested Dynamic Condition Response Graphs. In F. Arbab and M. Sirjani, editors, *Proc. of FSEN '11*, volume 7141 of *Lecture Notes in Computer Science*, pages 343–350. Springer, Apr. 2011.

[22] T. T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Safe distribution of declarative processes. 7041:237–252, 2011.

[23] R. Hull, V. S. Batra, Y.-M. Chen, A. Deutsch, F. T. Heath, and V. Vianu. Towards a shared ledger business collaboration language based on data-aware processes. In *Proc. of ICSOC*, 2016.

[24] R. Hull, E. D. R. D. Masellis, F. Fournier, M. Gupta, F. Heath, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, and R. Vaculín. *A Formal Introduction to Business Artifacts with Guard-Stage-Milestone Lifecycles*. 2011.

[25] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proc. of CCS '00*, pages 190–199. ACM, 2000.

[26] A. Kiayias and G. Panagiotakos. On Trees, Chains and Fast Transactions in the Blockchain. *IACR Cryptology ePrint Archive*, 2016:545, 2016.

[27] Y. Lewenberg, Y. Sompolinsky, and A. Zohar. Inclusive block chain protocols. In *Proc. of FC '15*, pages 528–547. Springer, 2015.

[28] O. López-Pintado, L. García-Bañuelos, M. Dumas, and I. Weber. Caterpillar: A Blockchain-Based Business Process Management System. In *Demo Track, BPM '17*, 2017.

[29] M. Marquard, M. Shahzad, and T. Slaats. Web-based Modelling and Collaborative Simulation of Declarative Processes. In *Proc. of BPM '15*, pages 209–225, 2015.

[30] J. Mendling, I. Weber, et al. Blockchains for business process management-challenges and opportunities. *arXiv:1704.03610*, 2017.

[31] Object Management Group. Case Management Model and Notation. Technical Report formal/2014-05-05, Object Management Group, May 2014. Version 1.0.

[32] Object Management Group BPMN Technical Committee. *Business Process Model and Notation, Version 2.0*. 2013.

[33] A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Comm. of the ACM*, 49(9):39–44, 2006.

[34] T. Slaats, R. R. Mukkamala, T. T. Hildebrandt, and M. Marquard. Exformatics Declarative Case Management Workflows as DCR Graphs. In *Proc. of BPM '13*, pages 339–354, 2013.

[35] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

[36] N. Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), Sept. 1997.

[37] J. Teutsch and C. Reitwießner. A scalable verification solution for blockchains. 2017.

[38] R. Vaculín, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya. Declarative business artifact centric modeling of decision and knowledge intensive business processes. In *Proc. of EDOC '11*, pages 151–160, 2011.

[39] W. M. van Der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development*, 23(2):99–113, 2009.

[40] B. Vitalik. A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[41] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted business process monitoring and execution using blockchain. In *Proc. of BPM '16*, pages 329–347. Springer International Publishing, 2016.

[42] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[43] X. Zhang, J.-P. Seifert, and R. Sandhu. Security enforcement model for distributed usage control. In *Proc. of SUTC '08*, pages 10–18. IEEE, 2008.