Data Object Driven Implementation of Choreographies on the Blockchain

Simon Siegert and Tom Lichtenstein

Hasso-Plattner Institute, Prof.-Dr.-Helmert-Straße 2-3 14482 Potsdam, Germany
`{firstname.lastname}@student.hpi.de`

Abstract. Like the exchange of goods, the exchange of messages is a central element of every business. Choreography diagrams are a common way of representing the exchange of messages between different parties. However, the implementation of these choreographies becomes a challenge if there is no trust between the participants. To solve this problem, this paper proposes and evaluates a way to implement trustful and flexible choreographies on a blockchain focusing on the reusability of the exchanged data.

Keywords: Choreography · Blockchain · Design by Contract · Ethereum

1 Introduction

Communication is one of the most essential keys to running a successful business. With the growth of a company, the number of communication partners and messages exchanged is likely to increase. A tool to better manage and track the progress of messages is the choreography diagram. Generally, choreography diagrams abstract from the business processes internals and focus on the communication flow between all participants [4]. However, when it comes to implementing choreographies, one is likely to encounter difficulties if there is a lack of trust between the participants. Since different business entities communicate with each other in a choreography, the question arises where the choreography is executed. This results in a trust problem, as the party responsible for execution has a knowledge advantage over any of the remaining parties. In order to solve this problem, an implementation employing blockchain technology can be utilized. A blockchain is a distributed ledger technology that uses cryptographic algorithms to create a decentralized, immutable log of transactions. Due to their decentralized and trustworthy architecture, blockchains are ideally suited to serve as a neutral execution engine for choreographies.

Whilst choreographies have already been implemented on blockchains, we want to address the problem of current implementations not storing data in a format that can be trustworthily reused in other choreographies.

The objective of this paper is to provide an implementation approach for choreographies on the blockchain. The process flow is enforced by the data exchanged in the messages modeled in the choreography. The data is stored in a way similar to data objects from BPMN collaboration diagrams to enable semantically structured reuse of the data [4]. The attributes stored in the data objects can be manipulated according to rules that are modeled at design time. This paper does neither consider access right management regarding smart contracts nor the encryption of sensitive data.

2 Foundation

This paper proposes a new way of implementing choreographies on the blockchain. Therefore, we want to explain the terms choreography diagram and blockchain in the following section.

2.1 Choreography Diagram

BPMN choreography diagrams are an abstraction of the traditional BPMN collaboration diagram. While the latter addresses internal business processes as well as their communication, choreography diagrams focus only on communication between different business entities. Choreography tasks model the exchange of messages between participants. Gateways allow decisions and parallel behavior within the communication flow [4].

Figure 1 shows an example of a choreography modeling the interaction of the participants in a car rental process. The shown annotations are extensions utilized by our implementation and are not necessarily part of a choreography diagram.

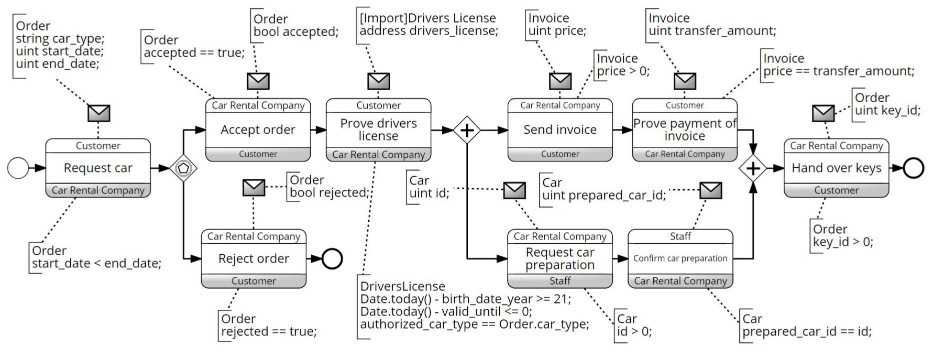

Fig. 1: Choreography diagram of a car rental process

2.2 Blockchain

As mentioned in section 1 a blockchain is a distributed ledger technology that uses cryptographic algorithms to create a decentralized, immutable log of transactions. Information is shared by transactions that get combined into blocks.

The blocks form an unambiguous sequence by referring to their corresponding predecessor. The concatenation of the information leads to the immutability of the stored information. This is done with cryptographic signatures provided by hash algorithms. A consensus algorithm ensures that only valid and verified transactions are included in the blockchain. On certain blockchain solutions, it is also possible to execute small programs, so-called smart contracts. These enable a trustworthy execution environment in exchange for a certain fee. In this paper we focus on the blockchain Ethereum using the compiler language *Solidity 0.5.3*[1], which we use to develop a process engine for choreographies.

3 Data Object Driven Implementation

The data object driven implementation of a choreography is inspired by the message behavior of a collaboration. Each message exchange can generate or transfer a data object in a specific state symbolizing the information of the message. If this behavior gets transferred to a choreography diagram, it can be assumed that a message in a choreography addresses an underlying data object. For this approach we use these data object as the basis to store the choreography state. We are aware, that the Choreography definition does not contain BPMN-data objects [4].

In our approach each data object has attributes. The attributes can be manipulated by messages. Every message addresses exactly one data object. The flow is enabled by the state of the data objects. Participants observe the states of the choreography-specific data objects. If the state is suitable, the next message can be sent. Therefore data objects store the state of the choreography implicitly by the state of their attributes.

As these attributes are not included in the modeling standard, we have to extend the choreography notation for this approach. For each message we have to add information about the data object and the attributes the message wants to update. Therefore we use annotations in order to be compliant with the standard. Additionally we add declarative descriptions to every choreography task. Those descriptions model the conditions that have to be fulfilled to execute this choreography task. A choreography diagram completely modeled according to our approach can be found in Figure 1.

3.1 Prevent undesired state changes

To keep the choreography in a valid state, we can extend the notation with declarative descriptions. State changes have to satisfy the modeled conditions. To prevent incomplete state changes we use the pattern *Design by Contract* [3]. The pattern uses preconditions and postconditions to validate the changes in state by a software component in a program. If the software component receives an input that does not match with the preconditions, the component will not

[1] `https://solidity.readthedocs.io/en/v0.5.3/`

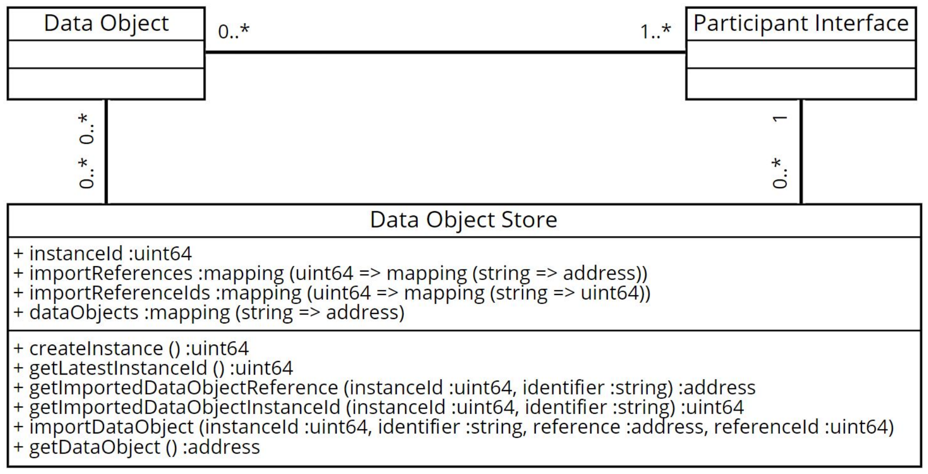
Data Object
0..*
1..*
Participant Interface
0..*
0..*
1
0..*
Data Object Store
+ instanceId :uint64
+ importReferences :mapping (uint64 => mapping (string => address))
+ importReferenceIds :mapping (uint64 => mapping (string => uint64))
+ dataObjects :mapping (string => address)
+ createInstance () :uint64
+ getLatestInstanceId () :uint64
+ getImportedDataObjectReference (instanceId :uint64, identifier :string) :address
+ getImportedDataObjectInstanceId (instanceId :uint64, identifier :string) :uint64
+ importDataObject (instanceId :uint64, identifier :string, reference :address, referenceId :uint64)
+ getDataObject () :address

Fig. 2: Data structure of a data object driven approach

be executed. Undesired state changes that do not match the postconditions will be reverted. This prevents state changes that should not be possible in the first place.

To apply the design by contract pattern to our approach we extend every choreography task with preconditions and postconditions as shown in the annotations of the tasks in our car rental choreography example in Figure 1. To invoke a sequential flow in the choreography we connect both tasks by using the postconditions of the current task as the preconditions of the next task. Thereby we want to ensure that the previous task has been executed.

In order to keep the execution order conform to the choreography, we have to make sure, that the conjunction of all postconditions of the predecessor can only be achieved by this particular task. This is essential for every choreography, because transitive postconditions such as $x < 10$ in one task and $x < 100$ in a sequential following task will lead to an illegal state where both tasks can be executed. Another constraint of this approach is that every task can only be executed once. This constraint coheres with the fact that the precondition of a task only depends on the postconditions of the predecessor. Once the postcondition of a task is fulfilled the resulting state can stay unchanged for the rest of the choreography. This leads to the problem that the precondition of the following task is always fulfilled and the task can be executed again later in the choreography creating undesired state. To prevent this, the execution number of every tasks is limited to one. As a result, loops can not be represented in this approach.

The first task in a choreography has no precondition, so it can be triggered at any time. If the previous element of a task is a gateway, we adjust the precondition accordingly to satisfy the flow conditions. Inclusive gateways are not supported yet.

Exclusive Gateway. In a choreography all affected participants have to know the necessary information for the exclusive gateway decision. To allow an exclusive behavior, we add conditions to the paths that pertain to one or multiple attributes that have been manipulated earlier by one of the affected participants. Those conditions have to exclude each other, so there is no way that two conditions are satisfied at the same time in the same choreography instance. To join the exclusive paths, the precondition of the task after the exclusive join gateway is the disjunction of all post-conditions of the tasks before the exclusive join gateway.

Parallel Gateway. A parallel process can only be enforced if the parallel paths do not depend on the same data objects. The parallel split has no specific rules to be considered. All following tasks take the postcondition of the task in front of the split as their precondition. To be compliant with the parallel join, the precondition of the predecessor of the parallel join gateway is the conjunction of all post-conditions of the tasks that the parallel join gateway takes as input.

Event-based Gateway. To enforce event-based split gateways, the preconditions of the following tasks correspond to the postcondition of the previous task and the negation of the postcondition of all other tasks that have the event-based gateway as their predecessor. This ensures that only one execution path is followed. In our approach the event-based join has the same conditions as the exclusive gateway join.

3.2 Data Object Decoupling

Data objects are the central aspect of the data object driven choreography implementation. To reuse data objects in another choreography, they must not be coupled to the choreography in which they were created. We achieve this decoupling by creating an abstraction layer above the data objects. To do this, we create an interface contract for each participant. These interface contracts implement the access rules according to the design by contract pattern as described in subsection 3.1. The participants of the choreography, as in our car rental example are the customer, the staff as well as the car rental company itself, use their interface contracts to modify the state of the data objects by using functions exposed by their specific interface which enforces the process-specific rules as shown in Figure 2. This makes it possible to reuse data objects in the same choreography, since the interface contracts contain references to the data objects.

3.3 Runtime Data Object Injection

The reusability of the generated data objects is one of the key properties of this approach. A data object is decoupled from the choreography that instantiated it, so it can be used in other choreographies as well. To keep the state of the data object conform to the state of the choreography that it originally belongs to, we have to limit the access rights of reused data objects to read only. Therefore we have to distinguish between choreography-specific and *external data objects*, which we also refer to as *imported data objects*. The former store the choreography state and can be manipulated, whilst imported data objects are handled as additional sources of information for the choreography instance. We also want to support the import of data objects at run time, to allow lazy imports of data objects, that might not exist at the deployment time of the choreography, or in a specific path of the choreography. To indicate a imported data object, we prefix the data objects name in the choreography annotations with a *[Import]* tag as shown in the Drivers License object in Figure 1.

To achieve a flexible and decoupled data structure, we can not include the external data object at deployment time. We extend our current contract architecture with another contract, which we will refer to as *data object store*. The data object store acts as a storage for all imported data object references. A data object can be referenced by the address of the contract that instantiated it. An address in *Solidity 0.5.3*[2] that refers to a smart contract can be casted into the according contract in order to reuse it. To cast an address into the desired smart contract only an interface with the required function definitions is needed. If the defined functions do not match with the original smart contract, the execution throws an error and reverts the changes. An advantage of storing references in the form of addresses instead of contract instances is that we do not have to know the specific data object types of the required imports in the data object store.

To store the references to the data objects we use a mapping from a string to the data object contract address, instead of storing empty references to the imports required by the choreography. We choose strings as identifiers for the data objects to improve the readability of the contract. A mapping is a flexible data structure in Solidity that can store up to 2^{256} data objects using the *keccak256 hashing algorithm*[3]. This enables the use of the same data object store contract instance for multiple choreography instances if the identifiers are unique.

The data object store is referenced by every participant of the corresponding choreography in a way that each participant can access all imported data objects. To import a data object, a participant casts a data object to its contract address and uses the reference to the data object store to invoke the import message with the address and the corresponding identifier as parameters. All participants access this data object with the data object store reference, the identifier and the data object interface. This allows you to import external data objects into your choreography.

[2] `https://solidity.readthedocs.io/en/v0.5.3/` 04.02.2019

[3] `https://solidity.readthedocs.io/en/v0.5.3/types.html` 04.02.2019

4 Process-Instantiation Optimization

Most choreographies, similar to business processes, are executed frequently. Therefore it is essential to keep the costs for new instances as low as possible. One of the main cost points of smart contracts is deployment. In the following section we describe a way to minimize these costs in our approach.

To avoid frequent redeployment, we have to find a way to store several instances of the choreography in only one smart contract, i.e. to reuse it. As described before, the state of the choreography is implicitly modeled in the data objects. In solving the problem of multiple instances, Object-oriented programming has proved helpful. Object-oriented programming dictates that every object has a state, a behavior and an identity. By modeling as a solidity smart contract, our data objects already have a state and a behavior, which are produced by the attributes and the available functions. However, it is not intended for a contract to contain several instances, except by deploying it again, which we want to avoid. Therefore, we need to store multiple instances within one data object. To achieve this, we introduce a centrally managed instance ID, which we call an instance ID. The instance ID is passed to the data object with every function call. The data object internally maintains a mapping in which the respective attribute for the corresponding instance ID is stored and read.

Since the data objects only provide their functionality with an instance ID, we need a unit which takes over the administration of the instance IDs. The solution we have chosen is to use the previously described data object store. This manages the instance IDs internally and offers a function which creates a new identifier for a choreography.

In the data object store, the IDs are represented as unit64 integers. If a new instance is to be created, an internal counter gets incremented by one. Furthermore, the data object store offers a function to get the latest instance ID. This enables interface contracts to enumerate all available instances of a choreography.

5 Related Work

This paper addresses research areas in the context of BPMN 2.0 choreography processes and blockchain technologies as well as the implementation of tamper-proof collaborations on the blockchain. In this chapter we provide a brief overview of the ideas that are associated with the data object driven approach.

Because of their distributed storage technology, blockchains have already been discovered as tamper-proof alternatives to execute and monitor processes and collaborations. In terms of business processes, Caterpillar is an open-source Business Process Management System using the Ethereum blockchain to create an execute instances of process models [2]. With regard to choreographies, there are already several approaches that use block chains to address the lack of trust in collaborative processes. Ingo Weber et al. proposed a way to model and execute choreography processes on the blockchain using smart contracts [5]. Weber et al.

introduce two main ideas to implement choreographies on blockchains. The first one focuses on process monitoring, by storing the role assignment and the process execution status of each involved participant. Every data exchange between the parties is driven by a monitor in the form of a smart contract that ensures the correct execution order and conformance to the choreography. The second one extends the first approach by addressing the coordination of a choreography additionally to the other tasks. All choreography instances are generated by a factory contract.

The blockchain technology is not limited to choreography implementations only. Stephan Haarmann proposed an approach to execute *Decision Models* on the blockchain Ethereum. [1]

6 Evaluation

To evaluate the feasibility of this approach, we compare the varying costs of the data object driven approach with a reference approach that uses only a single contract for the whole choreography. The implementation and behavior of the reference contract is inspired by the monitor approach described in section 5. A direct comparison with the original monitor approach is not possible because the implementation was not provided. For simplification reasons we do not include the factory contract described in the approach. In our reference implementation of the Weber et al. approach we store the state of each task in a variable. If all predecessors of a task are in a finished state, the task can be executed. If a task receives a payload it will store it in a corresponding variable for a decent comparison of the execution costs.

The key aspect of this evaluation is the accruing cost of the data object driven approach compared to the reference approach. We consider the costs of deployment, instantiation and execution of the choreographies. In order to compare the two implementations, we define two scenarios that form the basis of the evaluation.

6.1 Scenarios

This section describes the scenarios used for the evaluation of the approaches.

Minimal Choreography To illustrate the impact of the contract overhead of the data object driven implementation we create a minimal choreography. The choreography consists of two choreography tasks that model the communication between two participants. There are no gateways involved. Each participant has to execute one task.

Use case scenario To analyze the cost behavior in a real world use case scenario we modeled the choreography process of a car rental company that can be seen in Figure 1. The scenario consists of nine choreography tasks and three gateways. Parallel executions are included. The scenario models the communication between three participants.

6.2 Costs

In the execution environment Ethereum costs are measured in gas. A defined amount of gas is required for each atomic operation of the execution of a smart contract. The gas has to be payed in Ether which is the cryptocurrency of the Ethereum blockchain. The amount of Ether that has to be payed for one gas is defined by the gas price.

Instantiation costs The diagrams shown in Figure 3 compare the instantiation costs of both implementations regarding the first instantiation that includes the deployment of the contracts and all following instantiations.

Execution costs The diagrams shown in Figure 4 compare the execution costs of both implementations regarding the average task cost and the execution cost of the whole choreography.

6.3 Discussion

Splitting the choreography into multiple contracts and adding contracts for each participant as well as a storage system that dispatches all the requests adds a certain amount of costs to the deployment (c_{dpl}) of the data object driven approach. Despite being about three times more expensive to deploy, the data object driven approach also requires the resolution of references to other smart contracts in order to access the data objects, which increases the execution costs (c_{exec}) as well. The only advantage of the data object driven approach in terms of costs is the creation of new instances (c_{inst}). Whilst the reference approach has to deploy a new smart contract for each instance, the data object store can simply increase the instance number. This enormous difference between both approaches leads to the fact that the data object driven approach can be cheaper than the reference approach. According to our scenario results, the data object driven approach gets cheaper after 20 executions of the minimal scenario as well as after 13 executions of the car rental company scenario. Therefore, the data object driven approach can be feasible in financial terms when it gets executed often and does not change. The following equation describes the calculation of the total costs (c_{total}) after i executions with $i > 0$.

$$c_{total} = c_{dpl} + c_{exec} + (i - 1) \times (c_{inst} + c_{exec}) \tag{1}$$

The data object driven approach still has some drawbacks though. Transitive postconditions can lead to undesired state changes and have to be avoided at design time. Furthermore loops can not be realized in the data object driven approach. Nevertheless the data object driven approach offers a flexible way of implementing choreographies that can securely reuse information from previous or other choreography executions on the blockchain.

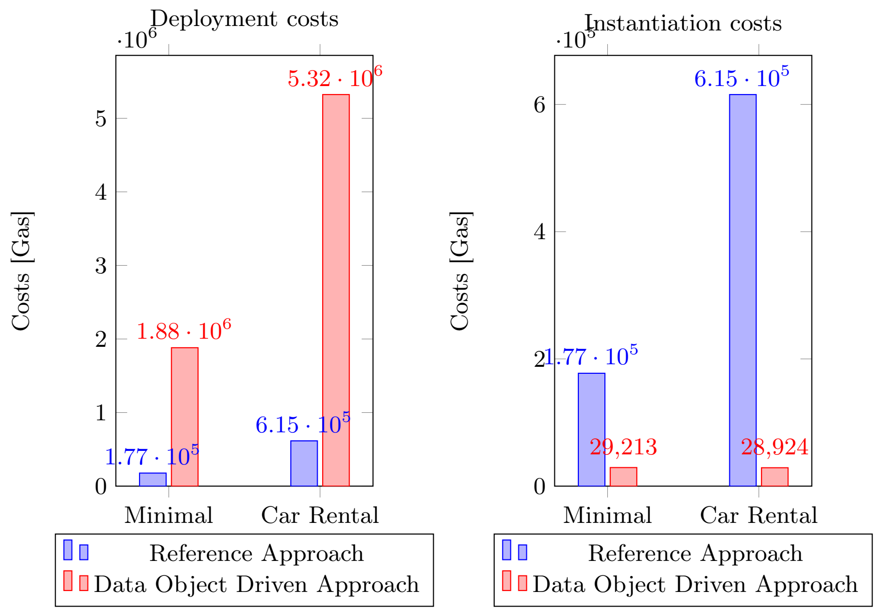

Fig. 3: Comparison of deployment and instantiation costs for both approaches

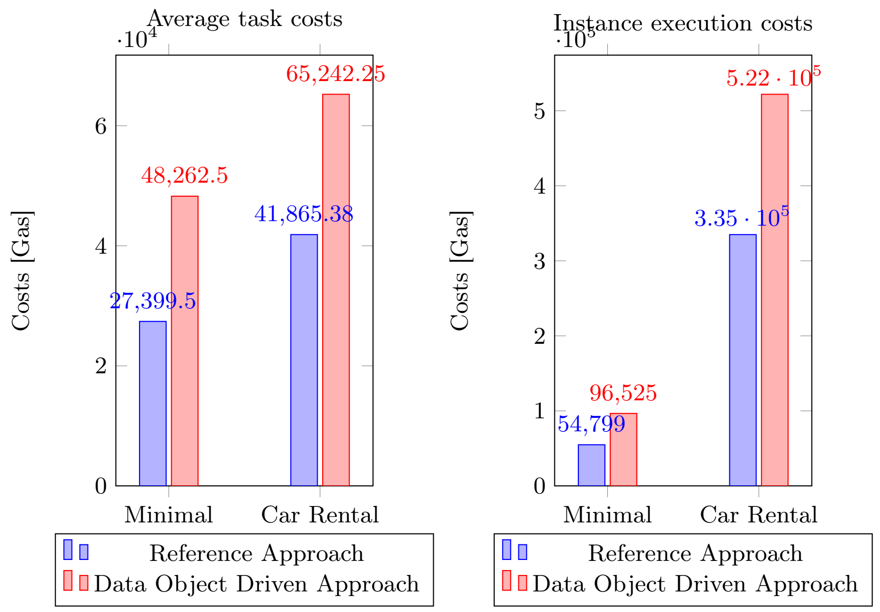

Fig. 4: Comparison of task and instance execution costs for both approaches

7 Conclusion

This paper introduces and evaluates a new approach to model and execute choreographies on blockchain technologies. Therefore, we identify the underlying data objects that are manipulated by the messages modeled in the choreography tasks. The concrete manipulation is subject to the rules that are added as BPMN 2.0 conform annotations in the choreography diagram and represent the postconditions of the tasks [4]. With the help of these rules the process flow is ensured in combination with the design by contract pattern. The main aspect of this approach is the possibility of reusing the data objects and the information they contain. In order to make this as flexible as possible, we also make it possible to import external data objects at runtime. An instance management concept is introduced to reduce instantiation costs. We compare this approach with the one published by Ingo Weber et al. and evaluate the costs of both implementations [5]. This shows the practical and financial feasibility of the data object driven implementation of choreographies on the blockchain.

However, some aspects of process implementations on the blockchain have not been considered in this paper:

Access rights management, which plays an important role as a blockchain is publicly accessible, is not considered. Furthermore, the data object driven approach does not offer the possibility to encrypt sensitive data. In the future, this approach can be enriched with further research in these areas.

References

1. Stephan Haarmann, Kimon Batoulis, Adriatik Nikaj, and Mathias Weske. Dmn decision execution on the ethereum blockchain. In *International Conference on Advanced Information Systems Engineering*, pages 327–341. Springer, 2018.
2. Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, and Ingo Weber. Caterpillar: A blockchain-based business process management system. In *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain*, 2017.
3. Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.
4. OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.
5. Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *International Conference on Business Process Management*, pages 329–347. Springer, 2016.