Institute of Architecture of Application Systems

Modeling and Execution of Blockchain-aware Business Processes

Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{falazi, hahn, breitenbuecher, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@Article{Falazi2019BlockchainAwareBP,
  author    = {Falazi, Ghareeb and Hahn, Michael and Breitenb{\"u}cher, Uwe and
               Leymann, Frank},
  title     = {Modeling and execution of blockchain-aware business processes},
  journal   = {SICS Software-Intensive Cyber-Physical Systems},
  publisher = {Springer Berlin Heidelberg},
  pages     = {1--12},
  day       = {06},
  month     = {February},
  year      = {2019},
  issn      = {2524-8529},
  doi       = {10.1007/s00450-019-00399-5},
  url       = {http://link.springer.com/article/10.1007/s00450-019-00399-5},
}
```


University of Stuttgart
Germany

Noname manuscript No.
(will be inserted by the editor)

Modeling and Execution of Blockchain-aware Business Processes

Ghareeb Falazi · Michael Hahn · Uwe Breitenbücher · Frank Leymann

Received: date / Accepted: date

Abstract The blockchain is an emerging technology that allows multiple parties to agree on a common state without the need for trusted intermediaries. Moreover, business process technology streamlines the automation of inter- and intra-organizational processes while cutting-down on costs. With the new business opportunities provided by blockchains, it becomes vital to combine both technologies to allow the modeling and execution of blockchain-based interactions within business processes. However, the existing business process modeling languages lack support to intuitively model the various interactions with blockchains. In this paper we address this issue by proposing a business process modeling extension that captures the particularities of blockchains. We also show how to transform the proposed constructs into standard-compliant models, and we present an integration architecture that allows external applications, to communicate with the blockchains. Finally, we validate our approach by providing a prototypical implementation that proves its practical feasibility.

Keywords Business Process Management · Blockchain Technology · Blockchain-aware Business Processes

Acknowledgements This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program "Services Computing", and by SmartOrchestra (01MD16001F).

1 Introduction

The blockchain is an emerging technology that allows multiple parties to share a common state without the need for establishing trust among them, or for trusted intermediaries. It has a disruptive potential in the fields of finance[17], supply-chains[6, 8], health-care[12], and others. On the other hand, the automation of business processes is a well-established approach that streamlines inter- and intra-organizational processes, and cuts-down on costs. With the new business opportunities provided by the blockchain technology, it becomes vital for automated business processes, in the form of executable process models, to allow and support the modeling and execution of blockchain-based interactions. However, the standardized modeling languages for business processes lack the ability to intuitively model the various interactions between an organization and a blockchain due to its unique specificities. In Sect. 2, we highlight the properties specific to the blockchain technology that make interacting with blockchain-based systems differ from other data management systems. Moreover, we present in Sect. 3 an integration architecture that abstracts the common functionality of public blockchains and thus allows external applications, such as process engines, to communicate with them. Afterwards, in Sect. 4 we introduce the `BlockME`-method for modeling and executing blockchain-aware business processes, to this end we present a modeling extension that captures the particularities of blockchains, and provides the means to model the various interactions with them. In Sect. 5, we motivate our modeling extension, and show in Section 6 how this extension can be transformed to a standard business process modeling language. Finally, in Sect. 7 we validate our approach by introducing a system architecture that realizes the `BlockMe` providing a prototypical implementation that proves its practical feasibility.

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
E-mail: [lastname]@iaas.uni-stuttgart.de

2 Background and Motivation

Blockchain technology is a disruptive invention [10] that promises to revolutionize various domains as it allows multiple parties that do not fully trust each other nor third-parties to agree on the value of a shared state, thus providing a fully-decentralized solution to the distributed consensus problem. The nature of the shared state differs from one domain to another. In the most basic form, which is found in cryptocurrencies such as Bitcoin [13], a state represent a set of balances of all participants in the system. In other cases, such as Ethereum [20], the state could represent a set of small programs, commonly known as smart contracts, along with their local storage. A change in this state is known as a transaction. In Bitcoin this refers to the money transfer between accounts, whereas in Ethereum it refers to changing the value of the local storage of smart contracts due to their code execution.

The invention of blockchains allowed to introduce new business opportunities in many fields [18], such as finance[17], health-care[12], and supply-chains[6, 8]. This makes the ability to bridge the gap between existing software solutions organizations use and the innovative blockchain technology of high importance. Moreover, as many organizations use process engines to automate their business processes, it comes vital that existing business process models gain the ability to communicate with public blockchains that permit engaging in transactions with unforeseen business partners and customers.

However, public blockchains have some inherent properties that make using them by regular applications, in general, challenging. This suggests the need for specialized modeling and execution support when working with blockchains. From a business perspective, one of the fundamental problems with that regard is the durability of blockchain transactions since they are not durably persisted as soon as they reach the network, but rather an arbitrary time thereafter.

The lifecycle of a blockchain transaction, as shown in Fig. 1 is not simple. The reason behind this is the globally distributed nature of blockchain networks, as well as the specificities of the Proof-of-Work (PoW) consensus protocol that is used by most major public blockchains to guarantee the validity of blockchain transactions without the need for a central authority. When a blockchain transaction *tx* is *Created*, it is submitted to one peer in the network which validates it against a set of rules, such as its authenticity, the correctness of its structure and the absence of contradicting transactions in the network. If found to be *Valid*, the peer, also known as a node, broadcasts the transaction to the blockchain network which contains, along with the regular peers,

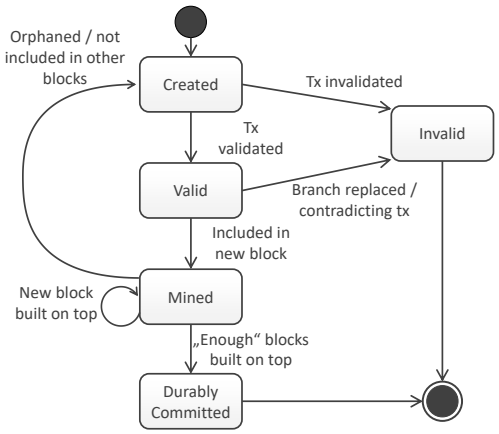

Figure 1 A simplified state-diagram showing the states a blockchain transaction can take, and the transitions between them.

a special kind of nodes called miners. When a miner receives a new transaction, apart from validating it like other nodes, it also places it with other transactions into a larger data structure know as a block. When the miner fills up enough transactions in a block, it starts running a very complex mathematical puzzle that takes an arbitrary time to solve, an act which is called mining. The first miner in the network which can prove solving the problem gets to append its block *b* to the end of the distributed blockchain, i.e., it updates its local version of the chain and broadcasts *b* to other nodes which do the same. When a transaction is finally included in a mined block, we say it has moved to the *Mined* state, however, its journey does not end there. Enough blocks need to be appended after *b* before it and its transactions are considered to be *Durably Committed*. Due to various reasons, a newly mined block suffers from the risk of being replaced by other blocks, a risk which greatly decreases when the block is old enough [13] in the chain. When a block is replaced with others, some of its transactions may return to the beginning of their lifecycle in which they need to be validated again, and might even become invalid if the replacing blocks contain a contradicting transaction.

The non-triviality of states a blockchain transaction goes through raises the need for tools and methods that support the modeling and execution of business processes that intend to communicate with blockchain networks [11]. To this end, we present the Blockchain-aware Modeling and Execution (`BlockME`)-method which supports the modeling and execution of blockchain-aware business processes by introducing a modeling extension that captures the various aspects of blockchain transaction durability issues, and gives the modeler the ability to intuitively handle blockchain communications without the need to have a deep knowledge and experience in the domain. Furthermore, we introduce the Blockchain

Access Layer, an extensible software component that unifies access to various public blockchain systems, and thus supports the execution of business process models that need to communicate with public blockchains.

3 The Blockchain Access Layer (BAL)

In this section, we present the concept and design goals of the Blockchain Access Layer (BAL), which provides a technology-agnostic asynchronous access to certain blockchain operations in a way that facilitates handling the uncertainty inherent to blockchains.

We have designed this component with four goals in mind: First, the BAL should **support handling blockchain uncertainty**; as we have seen earlier, blockchain transactions have a period of uncertainty before we consider them durably committed. During this period, a seemingly committed transaction might return to an uncomitted state or even get invalidated. The BAL should be aware of this uncertainty and provide means to handle it. Second, the BAL should play the role of **a technology-agnostic unification layer**. The reason is that the blockchain technology is still in its shaping phase, and many variations of it exist each with their implementations [21]. Thus, the BAL should provide a minimal and unified set of abstract blockchain operations applicable to the group of blockchain technologies we plan to address. Third, the BAL should **support extensibility**, meaning that the it should be designed in a way which allows new types of blockchains to communicate with it, so that its coverage of the domain can gradually enhance. Finally, the BAL should **provide an asynchronous API**; the probability that a mined transaction becomes durably persisted in the blockchain increases with additional blocks built on top of it [13, 15, 20], this means that if we want to achieve a certain degree of certainty, we should wait for a corresponding period of time before considering the transaction durable. This period can reach hours, e. g., in the case of Bitcoin. Thus, the BAL should allow external applications, such as process engines, to access its operations through an asynchronous API, and not force them to block waiting for the result.

As mentioned in Sect. 2, we focus in our work on public blockchains, and it turns out that even in this subset of the domain, the variation of attributes and capabilities is high. For example, some blockchains support smart contracts [20], while some support Multi-input/Multi-output (MIMO) transactions [13], and others do not even have a linear blockchain, but rather a graph of blocks [15]. Nonetheless, we identified that most public blockchains use some sort of a native cryptocurrency which gives the peers the motive to maintain the system functioning properly. This currency is transferable from one account to another through transactions. Thus, at the moment, the BAL only addresses the blockchain operations related to transactions that transfer a value. To this end, we identified three groups of relevant operations: (i) an operation to issue a blockchain transaction (*submitTransaction*), (ii) two operations to receive a blockchain transaction (*receiveTransaction* and *receiveTransactions*), and (iii) two operations for monitoring the state of an existing transaction (*detectOrphanedTransaction* and *ensureTransactionState*).

As we intend to provide an asynchronous API to BAL clients, each operation is provided as a one-shot subscription, i. e., whenever a client subscribes for some operation, the BAL waits for the corresponding situation in the blockchain to take place, and then notifies the client through a callback message and cancels the subscription. Moreover, all of these operations provide parameters that allow us to tackle the issue of transaction durability by specifying how many blocks need to be appended after the relevant transaction before the operation triggers a callback. Furthermore, clients have the ability to manually cancel an existing subscription.

According to this, each operation has two exposed functions in the API: *subscribe_<operationName>* and *unsubscribe_<operationName>*. One exception is the *receiveTransactions* operation, which is a durable subscription that makes the BAL send a callback message each time a new transaction with the desired properties is detected in the blockchain. Unsubscription from this operation happens only manually with an API call. Moreover, to enhance the resilience of this layer, canceling an already-canceled subscription does not cause any errors. Further architectural and implementation-related details are provided in Sect. 7.

4 Blockchain-aware Modeling Extension

In the following, we present our approach to model blockchain-aware processes that capture the semantics of blockchains and the potential trade-offs encountered when interacting with them. First, we present the overall method of the approach, then we introduce the proposed extension constructs and discuss their semantics.

4.1 Blockchain-aware Modeling and Execution Method

In this section, we introduce the Blockchain-aware Modeling and Execution (`BlockME`) method shown in Fig. 2 that gives a comprehensible overview of our approach.

First, a process modeling language that supports the proposed extensions, such as Business Process Model

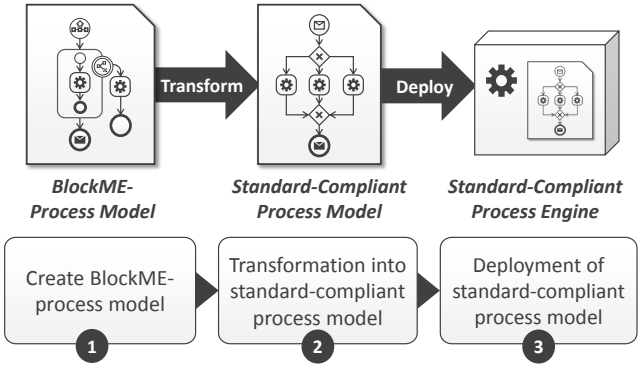

Figure 2 Overview of the BlockME-method (based on [5])

and Notation (BPMN) [14], is used to specify a process model with blockchain-aware modeling constructs, i. e., a BlockME-*process model*. Considering that we neither intend to develop a new process modeling language, nor to build a special blockchain-aware process engine, in the second step, we enable the transformation of the aforementioned BlockME-process models into standard-compliant and executable process models, such as BPMN 2.0 or Business Process Execution Language (BPEL). To achieve this, we employ an explicit step that follows a set of transformation rules. These rules ensure that all the blockchain-aware modeling constructs introduced by our BlockME extensions are transformed into natively supported constructs of the corresponding process modeling language. By transforming the blockchain-aware model into a standard-compliant model, we benefit from the portability of the native language while still making use of the intuitive and concise constructs of our extension.

Furthermore, a final deployment step could be necessary if the model contains an instantiating task that operates based on a blockchain-based event. The reason behind this is that the underlying layer responsible for triggering the instantiation of such a model needs to know the endpoint to send the instantiation message to, and this endpoint is only known after the model is deployed to the Business Process Management System (BPMS). Thus, the proposed deployment step involves informing the underlying layer with the desired endpoint details after it handles the actual model deployment. In the following, we present the proposed BlockME-artifacts and discuss their operational semantics.

4.2 Receiving a Blockchain Transaction

For detecting a new blockchain transaction addressed to a certain blockchain account, we present a new task type called *ReceiveTransactionTask*. The task can be used either as a regular task, or as a model-instantiating task

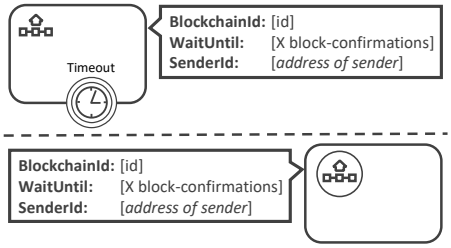

Figure 3 Graphical notation of the *ReceiveTransactionTask* (top), and the instantiating version of it (bottom).

and it has the following attributes: (i) *BlockchainId* that identifies the underlying blockchain network we are considering, (ii) *WaitUntil* which sets a minimum number of block-confirmations the new transaction should receive, and finally, (iii) an optional *SenderId* that identifies the transaction's sender. Notice that the *WaitUntil* attribute attached to this task and other BlockMEconstructs is responsible for handling the durability issue of blockchain transactions (cf. Sect. 2) by allowing modelers to decide the degree of trust they want to have in the finality of a transaction before operating on it, which differs greatly based on the importance of the transactions and the operations to be taken afterwards. The **operational semantics** for the *ReceiveTransactionTask* are as follows. In the case of an instantiating task, i.e., a task that is used to create a new instance of the process model, the task is activated only when the underlying BAL detects that a new blockchain transaction *tx*, which is addressed to a certain account, has been persisted in a block *b*, and that a sufficient number of additional blocks, as specified by the *WaitUntil* attribute, has been added to the blockchain on top of *b*. We refer to the number of blocks added on top of *b* as the number of block-confirmations received. Moreover, if the optional *SenderId* attribute is provided, then the task only gets activated if *tx* originates from the corresponding account. Furthermore, the specification of the exact blockchain address we expect to receive transaction at is done through a configuration file that initializes the underlying BAL.

On the other hand, in the case of a non-instantiating task the execution flow is blocked until the aforementioned conditions are satisfied and then continues regularly. Furthermore, the modeler can attach a timer boundary event to the task and in the case the timer is triggered before the conditions are satisfied, the execution continues with an alternative flow. Finally, if executed successfully, both of these tasks receive the details of the transaction *tx* in a message from the BAL.

Figure 3 shows a **visual representation** of both possible forms of this task. A task in BPMN is depicted as a rounded rectangle with an icon on the top left representing the type of the task, i.e., its semantics. In this

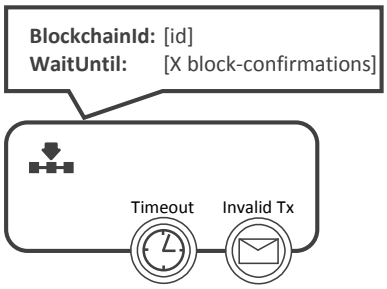

Figure 4 Graphical notation of the *SubmitTransactionTask*.

case, an icon showing an arrow going out of the blockchain is used. The icon has a white fill which indicates a "catching" task. In the case of an instantiating task, a circle is drawn around the icon making it look like a *start event*. Moreover, the optional interrupting timer event can be added to the task's boundary.

4.3 Submitting a Blockchain Transaction

The proposed extension supports submitting a transaction to the blockchain through a new *SubmitTransactionTask*. This task has two attributes: (i) *BlockchainId* that identifies the underlying blockchain network we are considering, and (ii) *WaitUntil* which sets minimum number of block-confirmations the submitted transaction should receive. Furthermore, the task has the following **operational semantics**. When it becomes activated, this task sends an unsigned blockchain transaction *tx* to the underlying BAL, and blocks until the BAL detects receiving the number of block-confirmations specified in the *WaitUntil* attribute. Afterwards, details about the submission, such as the transaction hash, and the block number in which it was included are received, and the flow continues normally. However, a submitted transaction can be invalid due to, e.g., insufficient funds, or an incorrect target address, and in such a case an error message is sent from the BAL causing an alternative flow to be taken instead of the normal flow. Furthermore, as for the *ReceiveTransactionTask*, the modeler may attach a timer boundary event to the task. In this case, another alternative flow is taken, if the timer is triggered before receiving the required number of block-confirmations.

Setting a timeout for the submission of a blockchain transaction is recommended because waiting for block-confirmations might take much more time than expected, and thus stall the execution of the process. The reason behind this is that a transaction might be reported as valid by the receiving node, but still does not end-up in a block due to very low transaction fees. Miners in a PoW-based blockchain network are free to choose the valid transactions they include into new blocks, and

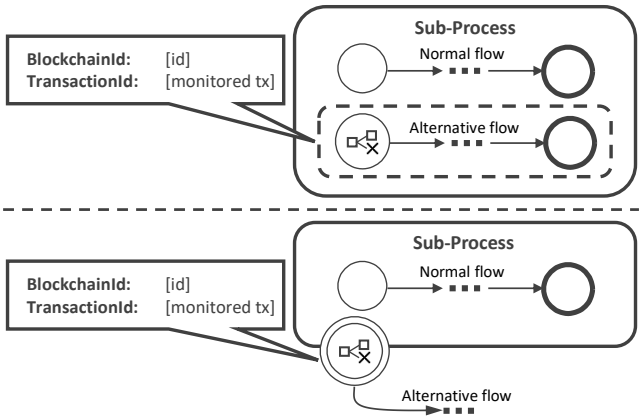

Figure 5 Graphical notation of the *OrphanedTransactionEvent* as a start event of an event sub-process (top), or as a boundary event on the border of a sub-process (bottom).

when the load on the network is high, they will probably give priority to transactions with higher transaction fees as they are more beneficial for them. Although a transaction with low fees tx_{low} could be dropped from the network due to the limited memory of nodes. This is not always the case, tx_{low} might eventually be mined and persisted in a block if it is still in the memory-pool of a miner when the load on the network decreases. Thus, triggering the aforementioned timer does not guarantee that tx_{low} will be aborted. An approach to address this situation is submitting a contradicting transaction with sufficient fees tx_{high} that will most likely be prioritized higher by the miners, and when included in a block will cause tx_{low} to be invalidated and aborted.

The *SubmitTransactionTask* has the **visual representation** shown in Fig. 4. Like the *ReceiveTransactionTask*, it has the regular shape of a task with an optional timer boundary event attached. The task can also have a message boundary event which allows receiving potential error messages from the BAL[1]. Moreover, the icon of the task shows an arrow pointing towards the blockchain with a solid black fill that indicates the semantics of a "throwing" task.

4.4 Handling Orphaned Blockchain Transactions

A transaction is orphaned if it is part of a mined block that has been replaced (potentially along with other blocks) by a longer, alternative chain. In a PoW-based blockchain, the longest valid chain is the manifestation of the majority decision over what the common state is [13], and thus represents the "one true" chain. However, due to geographical distance, network partitioning

[1] We use a message catch event icon when expecting to receive an error from the BAL because it is of an external nature.

and other reasons, two or more parts of the blockchain network can temporarily see multiple versions of the blockchain differing in one or more blocks located at the end. Nonetheless, when the these partitions finally exchange their versions of the reality, only the longest chain survives, and the other ones get discarded. At this point, we call all transactions contained only in the discarded blocks, orphaned transactions. These transactions are returned to the memory-pool of the peers aware of them, and go again into the verification process and wait to be put in a new block. However, an orphaned transaction **could** become invalid after being orphaned as another contradicting transaction might have replaced it in the longer chain (cf. Fig. 1). Due to this potential risk, it is advisable to capture the event of a transaction becoming orphaned.

The BlockME extension supports the detection of an orphaned transaction through the *OrphanedTransactionEvent*. This event has two attributes: (i) the *BlockchainId* attribute which is present in all of the artifacts proposed by the extension, and (ii) a *TransactionId* attribute which uniquely identifies the transaction that we are interested in monitoring. We assume that the transaction id is known to the process either from a previously executed transaction submission or reception, or directly sent to the process from another party.

The event can be used either as an interrupting boundary event attached to a sub-process, or as an interrupting start event in an event sub-process. In both cases it has the following overall **operational semantics**. If the transaction being monitored is found to be orphaned during the execution of the sub-process to which the event is attached ($P_{primary}$), then the sub-process is interrupted, i. e., the normal execution flow is canceled and the modeled alternative flow going out of the boundary event or specified in the event sub-process is executed. However, it is worth mentioning that in the case of a boundary event, the alternative flow executes outside the scope of $P_{primary}$, whereas in the case of an event sub-process, the alternative flow runs inside it.

The visual representation of both potential forms of the *OrphanedTransactionEvent* is shown in Fig. 5. It shows the event as a circle with an icon inside that represents the aforementioned case of a blockchain forking, and a branch being invalidated. Similar to the *TransactionReceiveTask*, the icon is drawn with a white fill to indicate that the event is of "catching" type. In the case of an event sub-process, the border of the circle is a single solid line to indicate that it is a start event of an interrupting event sub-process, whereas in the case of a boundary event, the border is a solid double line which indicates that it is an interrupting boundary event.

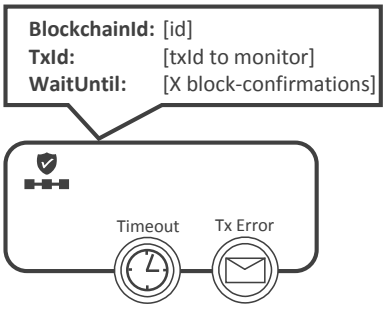

Figure 6 Graphical notation of *EnsureTransactionStateTask*.

4.5 Ensuring the State of a Specific Transaction

In certain scenarios, it is useful to ensure that a given transaction has at least a certain number of block-confirmations. We have seen this possibility in the *SubmitTransactionTask* and the *ReceiveTransactionTask*. However, having a task which gives modelers the ability to ensure a desired transaction state in a way which is decoupled from the acts of submitting or receiving a transaction would be beneficial. Therefore, the BlockME extension supports ensuring a transaction's state through the *EnsureTransactionStateTask*. For example, this task can be used together with the *OrphanedTransactionEvent* as part of the alternative flow originating from it to ensure that the orphaned transaction got re-validated before proceeding, or it can be used to safe-guard certain parts of the model which require a high level of certainty that a transaction is durably persisted.

The **operational semantics** of this task are as follows. When the task is activated it waits until the specified transaction (*TxId*) receives at least the number of block-confirmations indicated by the *WaitUntil* attribute, and then the normal flow continues. However, if while waiting for the required block-confirmations, it is detected that the transaction is invalidated after being orphaned, the execution continues with an alternative flow. The same alternative flow is also triggered if the specified transaction id cannot be found in the blockchain. Another alternative flow can be activated if a timer boundary event is attached by the modeler to the task and the timer gets fired before enough block-confirmations are detected. Finally, the **visual representation** of this task, which is shown in Fig. 6, is consistent with the appearance of other elements of our BlockME extension. The icon of the task represents a blockchain in an ensured state, and is depicted with a solid black fill to give the semantics of a "throwing" task.

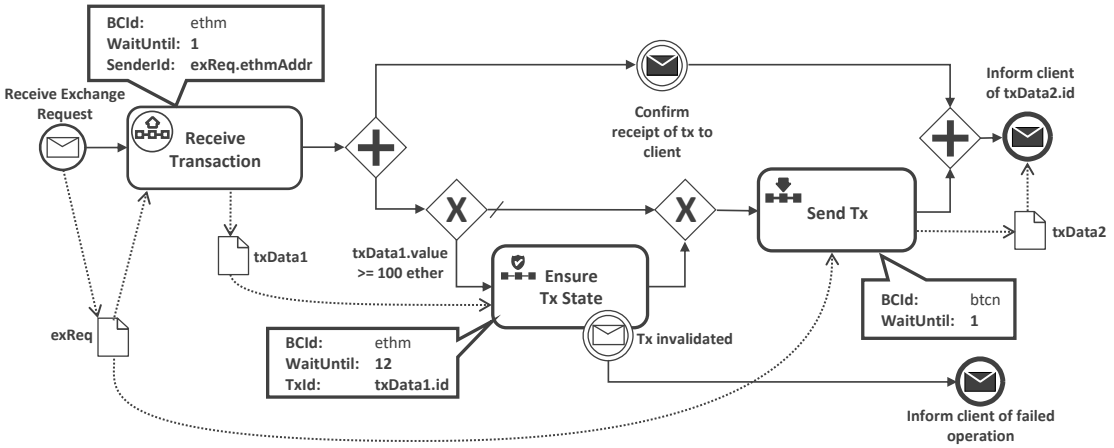

Figure 7 A scenario showing a potential usage for the *EnsureTransactionStateTask*.

5 Case Study

In this section we demonstrate the usage of the aforementioned extension artifacts by presenting a practical use-case from the domain of cryptocurrency exchanges. An exchange is an online service that allows customers to trade one type of cryptocurrencies for another. The case study we show in Fig. 7 represents a simplified version of the business process of an exchange service that transfers bitcoins in return for ethers. The process is initiated when a client sends an exchange request to the service containing their addresses in both Ethereum (source) and Bitcoin (target). Afterwards, the service expects the client to issue a transaction transferring ethers to its publicly known Ethereum address. This is expressed with a *ReceiveTransactionTask* that waits for only the minimum number of block-confirmations so that it can inform the client of the received transaction as soon as possible with a "throwing" event.

On parallel the service determines the risk level of the operation by checking the number of transferred ethers. If the risk is high, the service waits for 12 block-confirmations to be recorded for the received transaction using the *EnsureTransactionStateTask*, and then, issues a transaction transferring the corresponding amount of bitcoins to the client using the *SubmitTransactionTask* with a minimum number of block-confirmations.

On the other hand, if the risk is low, sending bitcoins to the client is done immediately. In both cases, the service sends the address of the submitted bitcoin transaction back to the client. Finally, if during the wait for block-confirmations (in case of risky transactions) the service detects that the transaction is orphaned and invalidated, it notifies the client about the failed operation by sending a message.

6 Transformation of BlockME-Artifacts into Standard-compliant Fragments

As mentioned earlier, we aim at producing standard-compliant models that support communication with blockchain-based systems, thus, in this section we show how to transform all BlockME-constructs into fragments compliant with BPMN 2.0. We have chosen BPMN 2.0 as the target of our transformation as (i) it supports the modeling of executable business processes, (ii) it is one of the mostly used business process modeling and execution languages, and (iii) tools exist [3] that allow transforming BPMN models into other languages, such as BPEL. Finally, (iv) we have used a BPMN-based notation to visualize the BlockME-extensions in the first place, which makes the transformation more comprehensible if BPMN is also the target language.

The set of transformation rules we present in the following aims at completely replacing the introduced BlockME-constructs with standard-compliant BPMN process fragments while capturing the exact same operational semantics through specifying respective interactions with the BAL. Moreover, we generally aim at encapsulating the transformed fragments into a subprocess as this enhances comprehension and reduces cluttering in the target model.

The BAL, presented in Sect. 3, provides an asynchronous API. For this reason, each time a blockchain-related operation is requested, at least two messages are exchanged, one originating from the process to call an operation through subscribing for an asynchronous callback, and the other originating from the BAL containing the result of the operation (callback). For some cases, additional messages can be part of the conversation, such as manual unsubscription messages sent from the process, or error messages sent from the BAL.

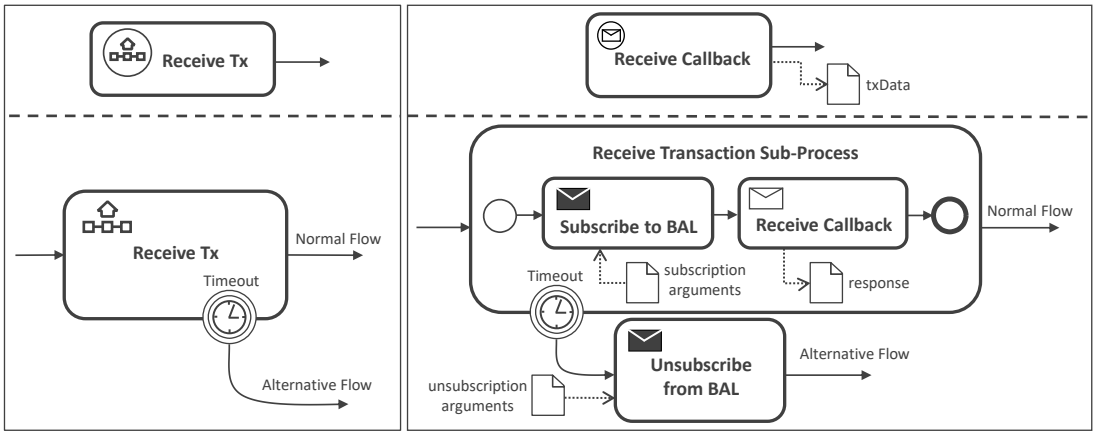

Figure 8 The transformation rule of the *ReceiveTransactionTask* for the instantiating (top), and non-instantiating case (bottom).

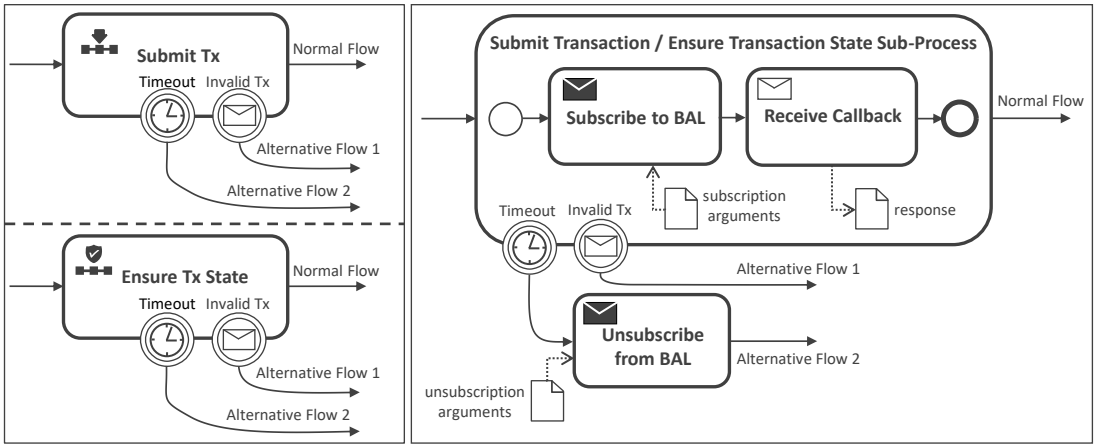

Figure 9 The transformation rule of the *SubmitTransactionTask* and *EnsureTransactionStateTask*.

Figure 8 describes the transformation rules for the *ReceiveTransactionTask*. A **non-instantiating *ReceiveTransactionTask*** (bottom) is transformed into a sub-process with two tasks; a send task that subscribes to the *receiveTransaction* operation at the BAL, and a receive task that waits for the resulting message sent back from the BAL when the desired transaction is detected in the blockchain. Along with the functional attributes required for the operation itself, such as the *BlockchainId* and the *SenderId*, the first message also contains a *subscriptionId* which is solely used for the purpose of message correlation, i. e., allowing the process engine to route the callback message to the correct process model instance. For this reason, the BAL includes the same *subscriptionId* inside the callback message, which also contains the details of the received transaction. Moreover, in order to inform the BAL of the endpoint of the process to which the BAL should send the callback message, an endpoint URL is also part of the subscription message. Furthermore, timer boundary events attached to the *ReceiveTransactionTask*, are copied and attached to the resulting sub-process. However, if such a timer event is triggered on the level of the process, we need to make sure to unsubscribe from the *receiveTransactionOperation* by sending an unsubscription message to the BAL before we continue with the alternative flow. Such an unsubscription message is not required for the normal flow, as the BAL unsubscribes automatically after sending the callback.

An **instantiating *ReceiveTransactionTask*** (top) is transformed into a single BPMN task, namely, an instantiating receive task. This task receives a message from the BAL containing the details of the received transaction. In this case, the subscription has to be done beforehand, e. g., during the deployment of the related process model, and also the unsubscription has to be done, e. g., when the model is about to be undeployed.

The transformation rules of ***SubmitTransactionTask*** and ***EnsureTransactionStateTask*** (Fig. 9) are very similar to the aforementioned case of the non-instantiating *ReceiveTransactionTask*. Apart from the obvious fact that we subscribe to different operations in the BAL, the only difference these two transformation rules have is the possibility to receive an error message

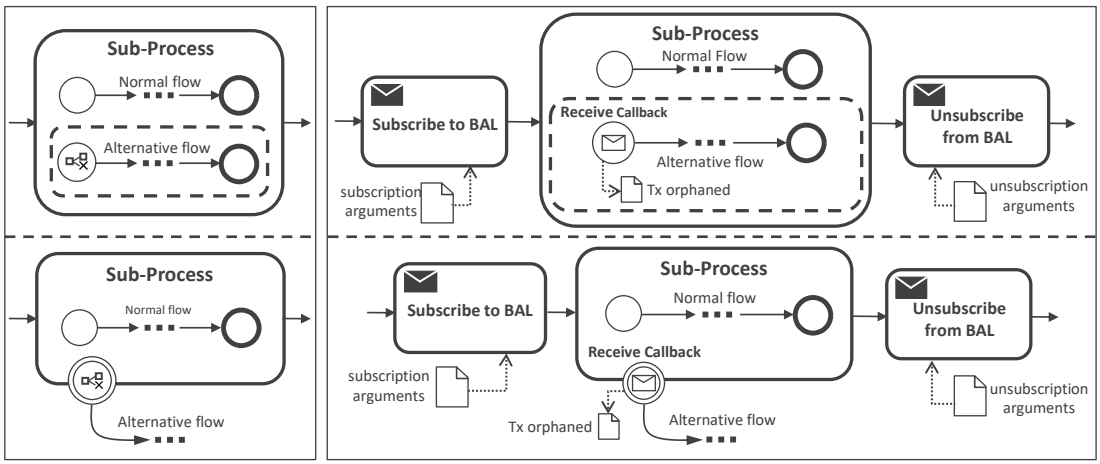

Figure 10 The transformation rule of the *OrphanedTransactionEvent* when used in an event sub-process (top), or as a boundary event (bottom).

as a callback from the BAL. To this end, we attach an interrupting message boundary event to the resulting sub-process which receives the potential error message, and activates the modeled outgoing flow (*Alternative Flow 1*). The body of the error message contains the specific reason behind it which can be used to trigger specialized compensating actions along this execution flow. An explicit unsubscription message is not required in this case, as the BAL automatically unsubscribes after sending an error callback.

On the other hand, the transformation of the ***OrphanedTransactionEvent***, which is shown in Fig. 10, looks different. As the *OrphanedTransactionEvent* is used to annotate an existing sub-process P (either as a boundary event or as part of an event sub-process), we use the same sub-process in the transformed fragment. Furthermore, depending on the way the *OrphanedTransactionEvent* is used, we either annotate $P_{primary}$ with an interrupting message boundary event, or add an interrupting event sub-process with a message start event to $P_{primary}$. These message events are triggered if a callback is sent from the BAL informing us that the transaction under consideration has been orphaned, and in which case, an alternative execution flow is taken.

In order to allow the BAL to send the potential callback message in a timely manner, i. e., starting from the point-in-time in which $P_{primary}$ is activated until it is completed, we use a pair of subscribe/unsubscribe send message tasks exactly before and after P, as shown in Fig. 10. Moreover, if a callback is sent from the BAL, there is no need for an unsubscription message, as the BAL does the unsubscription automatically in that case. Nonetheless, when using the *OrphanedTransactionEvent* in an event sub-process (top), and if the callback is triggered, which fires the inner sub-process inside $P_{primary}$, the normal flow going out of $P_{primary}$ is then taken after the inner sub-process finishes. This results in sending an unnecessary unsubscription message. However, this does not cause a problem to the BAL as it is designed to treat unsubscription requests idempotently, i. e., trying to unsubscribe from an already finished subscription does not result in an error.

To demonstrate the usage of these rules, a transformed version of the case study presented in Sect. 5 is accessible on Github.[2] Finally, we notice that, an expert could implement the interaction with the BAL directly using these rules without the need for the extension. However, this is inconvenient and error prone.

7 Validation

In this section, we prove the practical feasibility of our approach by showing an architecture in which it can be realized, as well as a prototypical implementation of key parts of this architecture, namely, the blockchain access layer and blockchain adapters.

7.1 System Architecture

Figure 11 shows the system architecture we propose to support the BlockME-method. The architecture, which is based on our previous work [5], is divided into three major layers: (i) Blockchain Layer. (ii) Blockchain Access Layer. (iii) Blockchain-aware Process Layer. The **Blockchain Layer** represents the set of blockchain networks our system intends to support. The figure shows two of these networks, namely, the Bitcoin network and the Ethereum network. In order to communicate with a blockchain network, an application needs to have access

[2] https://github.com/ghareeb-falazi/BlockME-UseCase

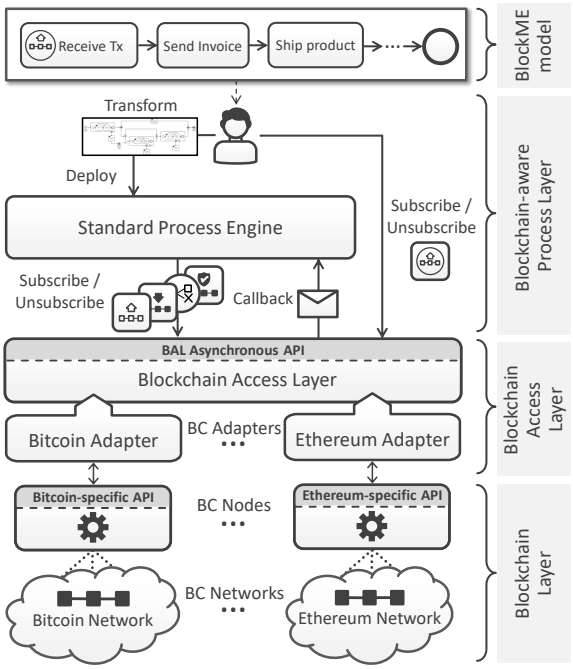

Figure 11 BlockME system architecture showing its three major layers (based on the architecture presented in [5]).

to one of the network's nodes. A blockchain node is a process that executes the peer-to-peer protocol specific to this blockchain. An example Ethereum node is *Geth*[3], and an example Bitcoin node is *bitcoind*[4]. A blockchain node usually runs in the same environment as the accessing application. Moreover, most kinds of blockchain nodes expose some sort of an API to other applications so that they can interact with the underlying blockchain network. However, these APIs are technology-specific although exposing some common functionality.

As part of the **Blockchain Access Layer**, adapters with technology-specific implementations are introduced in order to unify access to blockchain nodes. These adapters know the specificities of the corresponding blockchain technology and can process requests from higher levels and translate them to calls addressed to the underlying node's API. Furthermore, all adapters have a common interface to communicate with the higher layer. By having such a common interface, we guarantee the aforementioned extensibility requirement (cf. Sect. 3). We can extend our architecture to include new blockchains only by providing a new adapter that implements the same common interface, without the need to alter any other component. The BAL has also the responsibility of managing subscriptions by storing subscription ids and callback endpoints, and using them to correctly direct response messages. Furthermore, this layer provides an asynchronous API to service external applications, such as process engines, and supports the operations we previously described in Sect. 3.

Finally, the third layer of the architecture is the **Blockchain-aware Process Layer** which is responsible for executing BlockME-models. As we do not intend to build a new process engine nor to extend an existing one, the core of this layer is a standard-compliant engine. This approach allows us to benefit from proven features of standard engines, and promotes the portability of the extension by allowing adopters to continue using the engines they might already have. To allow this to happen, the BlockME-models have to be manually transformed so that all BlockME-specific constructs are replaced with standard-compliant fragments by following the rules we presented in Sect. 6. Furthermore, manual intervention is also necessary during the deployment of a process model onto a process engine if the given process model contains an instantiating *ReceiveTransactionTask* (cf. Sect. 4.2) which is responsible for instantiating a new instance of the process model if a transaction with specific properties is received. To this end, a durable subscription request for the process model has to be sent to the BAL by invoking the *subscribe_receiveTransactions* operation. A subscription always requires sending the details of the process model endpoint that the BAL needs to invoke when sending response messages. The endpoint, in this case, is only known after deploying the process model to a process engine, thus the user which is responsible for deployment is also responsible for subscribing the process model at the BAL. Moreover, the corresponding unsubscription operation of the BAL should be invoked when such a process model is about to be undeployed from the engine.

During the execution of a process model, the process engine makes calls to the asynchronous API exposed by the BAL in the form of subscription and unsubscription requests as specified in the underlying process models. The BAL responds accordingly with callback messages sent to the endpoints specified in each request.

7.2 Prototype

In this section we provide a short description of the prototypical implementation of the key component of the introduced architecture, namely the BAL. We realized this component as a layered Java 8 web application exposing, at its highest level, a RESTful HTTP API. On the other hand, at the lowest level of the BAL, we can find a set of blockchain-specific adapters each of which implementing the same Java interface which provides a unified set of operations to the higher level. One example

[3] https://github.com/ethereum/go-ethereum/wiki/geth
[4] https://github.com/bitcoin/bitcoin/

of these adapters is the *Ethereum Adapter*, which communicates with an underlying Ethereum node using a lightweight Java library called web3j[5] that implements a client for the JSON-RPC API all Ethereum nodes provide. The adapter utilizes a thread pool in order to execute the long running operations of monitoring the desired changes in the blockchain without blocking its clients. In between the HTTP API at the top and the adapters at the bottom operates the subscription manager. This manager is responsible for mapping externally provided subscription ids and endpoint URLs to the currently running operations at the adapters level, so that the resulting callbacks find their designated targets. The prototype is packaged using Maven, and is publicly available as an open-source project on Github[6].

8 Related Work

The ability of blockchains to remove the need for establishing trust between communicating partners have been utilized to support business processes in various aspects which we will discuss in the following.

Blockchains can tackle the issue of lacking trust facing business process choreographies; although the integration of business processes across organizations could be beneficial, interacting partners need to establish trust among them, or they need to trust an external party to execute the joint process. This hinders the idea of developing a collaborative process in the first place. To this end Weber et al. [19] developed an approach that integrates the blockchain technology into choreographies, in a way that eliminates the need for a central authority, while still maintaining trust. Specifically, their approach allows generating immutable audit trails of the execution of choreographies, and further allows to channel all Business-to-Business (B2B) choreography messages through automatically generated smart contracts so that business rules are enforced by the decentralized blockchain system. However, their approach suffers from potentially high costs of data storage in public blockchains, and thus García-Bañuelos et al. [7] propose an optimized method for executing business processes on top of commodity blockchain technology focusing on the reduction of the aforementioned cost. Furthermore, López-Pintado et al. [9], propose building an entire BPMN-compatible BPMS based on the ethereum blockchain, making trustless execution of business process inherent in the engine itself. These two approaches look at blockchains as a means to support the execution of business processes, specifically process choreographies. However, our approach addresses blockchains in a broader sense. We look at blockchains as external systems that allow to exchange transactions with unforeseen business partners and customers. Furthermore, our approach does not require to alter existing BPMSs, but rather focuses on supporting standard-compliant ones.

Various kinds of connectors have also been introduced to allow exiting BPMSs to communicate with blockchains. Auberger and Kloppmann [4] introduce a connector that allows IBM Business Process Manager [1] to communicate with the permissioned blockchain, Hyperledger Fabric [2] and access its various capabilities. Moreover, Schmidt et al. [16], introduce a framework for business integration based on blockchains. At the bottom of the introduced architecture they propose a set of "Smart Adapters" that they use to connect various IT-systems to blockchains. These adapters support various kinds of public and permissioned blockchains, and expose an API that allows external applications to interact with them. However, both of these approaches lack the ability to address the durability issues of blockchain transactions. They delegate this task to the higher level of the architecture, i. e., the layer of external applications. However, this complicates the design of these applications, and requires the involvement of blockchain experts. On the other hand, the BAL, which we introduced here, can handle the aforementioned durability concerns internally, and exposes an comprehensible interface to external applications eliminating the need for a blockchain expert on the other end.

9 Conclusions and Outlook

In this paper we introduced the BlockME-method that supports the modeling and execution of blockchain-aware business processes. We proposed an extension to BPMN that captures the semantics of blockchain-based systems and assists modeling fine-grained decisions when handling the uncertainty of blockchain transactions. We further showed how to convert each of the proposed modeling extensions into standard-compliant BPMN 2.0 process fragments. Moreover, we designed the Blockchain Access Layer, an integration middleware that allows external applications to communicate with public blockchain systems while taking care of blockchain specificities. This layer supports the execution of blockchain-aware processes by providing asynchronous operations they can access. Finally, we proved the feasibility of the method by presenting a system architecture and a prototypical implementation of the BAL.

As future work, we intend to expand the approach both horizontally, by supporting further blockchain systems, and vertically, by introducing new capabilities such

[5] https://github.com/web3j/web3j

[6] https://github.com/ghareeb-falazi/blockchainaccesslayer

as handling smart contracts deployment and invocation. We further intend to develop a concept to express the durability of blockchain transaction with means other than the number of block-confirmations in order to support non-linear ledgers, such as the Tangle [15]. Finally, we plan to create tools for transforming and deploying BlockME-models without the need for a human-expert, thus fully automating the BlockME-method.

References

1. (2018) Digital business automation on cloud. URL https://www.bpm.ibmcloud.com/
2. (2018) Hyperledger Fabric. URL https://www.hyperledger.org/projects/fabric
3. (2018) Petals BPM. URL http://bpmneditor.petalslink.com/
4. Auberger L, Kloppmann M (2017) Combine business process management and blockchain. URL https://www.ibm.com/developerworks/library/mw-1705-auberger-bluemix/1705-auberger.html
5. Breitenbücher U, et al (2015) A situation-aware workflow modelling extension. In: Proc. of iiWAS, ACM Press, DOI 10.1145/2837185.2837248
6. Cecere L (2017) Seven use cases for Hyperledger in supply chain. URL http://www.supplychainshaman.com/big-data-supply-chains-2/10-use-cases-in-supply-chain-for-hyperledger/
7. García-Bañuelos L, et al (2017) Optimized execution of business processes on blockchain. In: Business Process Management, Springer International Publishing, Cham, pp 130–146
8. van Kralingen B (2018) IBM, Maersk joint blockchain venture to enhance global trade. URL https://www.ibm.com/blogs/think/2018/01/maersk-blockchain/
9. López-Pintado O, et al (2017) Caterpillar: A blockchain-based business process management system. In: Proc. of BPM Demo Track co-located BPM
10. Mattila J (2016) The blockchain phenomenon – the disruptive potential of distributed consensus architectures. ETLA Working Papers 38, The Research Institute of the Finnish Economy, URL https://ideas.repec.org/p/rif/wpaper/38.html
11. Mendling J, et al (2018) Blockchains for business process management - challenges and opportunities. ACM Transactions on Management Information Systems 9(1):4:1–4:16, DOI 10.1145/3183367, URL http://doi.acm.org/10.1145/3183367
12. Mettler M (2016) Blockchain technology in healthcare: the revolution starts here. In: 2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom), pp 1–3, DOI 10.1109/HealthCom.2016.7749510
13. Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system. White Paper
14. OMG (2011) Business Process Model and Notation (BPMN), Version 2.0
15. Popov S (2018) The Tangle. White Paper
16. Schmidt S, et al (2018) Unibright-the unified framework for blockchain based business integration. White Paper
17. Schwartz D, Youngs N, Britto A, et al (2014) The Ripple protocol consensus algorithm. White Paper
18. Underwood S (2016) Blockchain Beyond Bitcoin. Commun ACM 59(11):15–17, DOI 10.1145/2994581
19. Weber I, et al (2016) Untrusted business process monitoring and execution using blockchain. In: La Rosa M, Loos P, Pastor O (eds) Business Process Management, Springer International Publishing, Cham, pp 329–347
20. Wood G (2018) Ethereum: a secure decentralised generalised transaction ledger - Byzantium version. White Paper
21. Xu X, Weber I, Staples M, Zhu L, Bosch J, Bass L, Pautasso C, Rimba P (2017) A taxonomy of blockchain-based systems for architecture design. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp 243–252, DOI 10.1109/ICSA.2017.33

All links were last followed on 2019-02-04.