Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts

Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, Frank Leymann, Vladimir Yussupov

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

BibTeX:

```
@inproceedings{Falazi2019_SmartContractComposition,
  author    = {Ghareeb Falazi and Michael Hahn and Uwe Breitenb{\"u}cher and
                Frank Leymann and Vladimir Yussupov},
  title     = {{Process-Based Composition of Permissioned and Permissionless
                Blockchain Smart Contracts}},
  booktitle = {Proceedings of the 2019 IEEE 23rd International Enterprise
                Distributed Object Computing Conference (EDOC 2019)},
  publisher = {IEEE}
  year      = 2019,
  month     = oct
  pages     = {77--87},
  doi       = {10.1109/EDOC.2019.00019},
}
```

Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts

Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, Frank Leymann, Vladimir Yussupov
Institute for Architecture of Application Systems, University of Stuttgart, Stuttgart
{falazi, hahn, breitenbuecher, leymann, yussupov}@iaas.uni-stuttgart.de

***Abstract*—Blockchains are distributed systems that facilitate the interaction of autonomous entities with limited mutual trust. Many of them support transactional applications known as smart contracts, which access and modify the shared world state. Permissionless blockchains are completely decentralized and do not require mutual trust between interacting peers, but at the expense of having low performance and limited data confidentiality capabilities. On the other hand, permissioned blockchains solve these issues, but sacrifice complete decentralization and involve more trust assumptions. Therefore, there is no single blockchain system suitable for all use-cases. However, this becomes a serious integration challenge for enterprises that need to interact with multiple permissioned and permissionless blockchains in the same context. To facilitate this, we propose an approach that enables composing smart contract functions of various permissioned and permissionless blockchain systems by providing the ability to invoke them directly from business process models using a new task type. To keep this task blockchain-agnostic, we designed a generic technique to identify smart contract functions, as well as a generic metric to describe the degree-of-confidence in the finality of blockchain transactions. Thereby, the proposed approach extends our previous work, BlockME, which provides business modeling extensions only suitable for interacting with permissionless blockchains. To validate the practical feasibility of our approach, we provide a detailed system architecture and a prototypical implementation supporting multiple blockchains.**

I. Introduction

Blockchain systems have gained attraction in recent years in fields such as payment settlement, supply chains management, health care and digital identity. They can be described as distributes systems that allow independent parties to conduct collaborative processes even while having limited mutual trust. Early blockchain protocols, such as Bitcoin [1] and Ethereum [2], where permissionless, in the sense that participating in the protocol with any role is open for everyone. These systems favor absolute decentralization over privacy and performance, and thus are not suitable for demanding or competitive use-cases, such as the ones involving enterprises. Therefore, permissioned blockchains were introduced as an alternative that guarantees data confidentiality and ensures better performance at the expense of losing some degree of decentralization. However, due to their nature, permissioned blockchains are usually independent silos containing partners from a similar domain, which means that an enterprise could easily become involved in more than one permissioned blockchain. Furthermore, enterprises still need permissionless blockchains, e.g., to utilize cryptocurrencies or store immutable hashes of confidential data for auditing purposes.

A regular application program can be used to implement this sort of integration. Nonetheless, this would require the developers to have significant knowledge of the involved blockchain technologies, which do not follow established standards and are very different in terms of properties, behavior and exposed APIs. This has the potential of making the development process time-consuming and error-prone. Therefore, our aim here is to solve this integration challenge by allowing enterprises to compose the functionality provided by blockchain smart contracts, which are the transactional applications that these systems support, using business processes, and thus gain the ability to model the composition logic using well-established languages such as BPMN [3]. To this end we introduce the BlockME2 approach, which is based on our previous work [4], and has the following new contributions: (i) it proposes a new BPMN task type that is capable of modeling the invocation of a smart contract function and makes sure the associated blockchain transaction is durably committed; (ii) it specifies how this task can be transformed into standard-compliant BPMN, which can be executed on common process engines like Camunda [5]; (iii) it introduces a generic measure for describing the degree-of-trust that a certain blockchain transaction is durably committed, despite the fact that different blockchains decide differently on when a transaction is considered final; (iv) it also introduces a generic URI scheme for identifying smart contract functions of various permissioned and permissionless blockchains.

On the other hand, to prove the practical feasibility of the approach, we provide an architectural design and a prototypical implementation for the Blockchain Access Layer (BAL), which is the middleware component that we introduced to support the execution of BlockME2 models. Finally, we show how BlockME2 can be applied to a simple real-world scenario from the domain of car manufacturing and retail.

The remainder of the paper is structured as follows: in Section II, we present the background and motivation. In Section III, we provide a high-level overview of BlockME2 and describe the related previous approach. In Section IV, we explain how to measure the degree-of-confidence in the finality of blockchain transactions, while in Section V, we explain the actual approach in detail. Afterwards, in Section VI, we validate the feasibility of the approach, and in Section VII, we demonstrate a real-world scenario. We present the related work in Section VIII, and finally provide concluding remarks and discuss the planned future work in Section IX.

II. Background and Motivation

In this section, we introduce the required background knowledge and motivate this work based on an example.

A. *Permissionless Blockchains*

Permissionless (or public) blockchains are decentralized systems designed to run transactional programs that access a fully replicated immutable data store without the need to have full trust in any single participant or in a third-party. This kind of blockchains is open for participation by nature, and allows anyone to join the system by simply running a local instance of the corresponding peer-to-peer protocol. The first permissionless blockchain to be introduced was Bitcoin [1]. It is a decentralized payment network that introduces its own cryptocurrency and achieves probabilistic consensus on what transactions to execute next and in which order via a protocol based on Proof-of-Work (PoW) [6]. Furthermore, to ensure the integrity and non-repudiation of these transactions Bitcoin utilizes digital signatures. Therefore, users can be identified in the network using their public keys without the need for real-world identities, which makes them pseudonymous.

Apart from Bitcoin, other permissionless blockchains exist that aim at generalizing the scope of the supported transactional programs so that it includes use-cases beyond payment settlement. For example, Ethereum [2], another permissionless blockchain utilizing PoW to achieve consensus, introduced the notion of *smart contracts* [7] to the world of blockchains. Ethereum smart contracts are small applications written in a general-purpose, Turing-complete language, such as Solidity. They usually contain multiple functions and are stored immutably in the blockchain. The execution of smart contract functions is done by all nodes in the system and is guaranteed to be deterministic. Therefore, they are usually used in applications involving multiple parties and requiring trustworthy and decentralized execution of the business logic, such as multi-signature escrows, decentralized autonomous organizations, financial derivatives, name registries, and many more. A smart contract is isolated from other smart contracts by having its own storage that can only be accessed by its own functions. Furthermore, a subset of these functions is usually public allowing other smart contracts or even external applications to interact with them. Invoking smart contract functions that cause a change in the underlying storage always happens in the context of an atomic transaction. Therefore, the execution of a state-changing public function of a smart contract requires the formulation and submission of a *blockchain transaction*. The content of this data structure includes the address of the corresponding smart contract, the name of the function to be invoked, as well as the required parameters. After a peer issues such a transaction, it goes through the consensus mechanism, and if valid, it ends up in an agreed-upon batch of transactions, or a *block*. A block determines the relative order of all contained transactions that dictates which one should be executed before which. New blocks are broadcast throughout the network allowing all peers to execute them. Each transaction in a block runs atomically and in isolation.

Despite their benefits, permissionless blockchains suffer from several drawbacks inherited from their fully replicated and open nature: (i) They have **low performance** in terms of the ability to process a high throughput of transaction requests. The reason is that, on the one hand, PoW progresses in periodic phases, which have a limit in terms of how frequent they are, and on the other hand, the number of transactions that can be included in each phase is also limited. For example, Bitcoin is currently able to process only about 7 tx/s. Approaches to enhance performance, such as altering PoW parameters, using other consensus mechanisms such as Proof-of-Stake, or sharding the network into smaller sub-networks, all have the side effect of reducing some desirable property of the system, which is usually the degree of decentralization [8]. (ii) Data stored in permissionless blockchains is publicly accessible as it is fully replicated among all peers. Therefore, without implementing additional measures on top of the original protocol, this kind of blockchains suffers from **confidentiality issues**. (iii) Public blockchain transactions **never reach absolute finality**, because the design of PoW allows forking the chain of agreed-upon transactions due to factors such as network latency. When the forks are amended, certain transactions committed in the dropped forks might be revoked. These drawbacks hinder the utilization of public blockchains by large businesses since they prevent their integration into enterprise-grade systems that usually require secure, high performance applications that guarantee reliable and consistent transactions. Therefore, permissioned blockchains were introduced with the main goal of avoiding these shortcomings.

B. *Permissioned Blockchains*

Permissioned blockchains are blockchain systems in which the participation in some or all roles is restricted to a set of users. This includes systems that implement total control over all user roles, such as Hyperledger Fabric [9], and systems that only control which nodes are allowed to participate in the consensus process while leaving other roles open, such as XRP Ledger [10] and Chain [11]. Permissioned blockchains are best suited for competing enterprises that are, nonetheless, willing to engage in collaborative processes without employing third-parties, such as notaries, or centralized settlement networks [12]. The reason is that, besides being mostly general-purpose and supporting smart contracts, permissioned blockchains provide enhancements over their permissionless counterparts that facilitate enterprise-grade use-cases: (i) Since the participation in the consensus protocol is limited to a specific group of users that requires explicit system reconfiguration to be modified, permissioned blockchains are able to use Byzantine Fault Tolerant (BFT) protocols, which are a better alternative in terms of transaction latency and throughput [13]. (ii) Furthermore, permissioned blockchains are generally better in terms of confidentiality since sensitive transactions can be isolated from public access. (iii) Finally, most permissioned blockchain systems achieve transaction finality and other desirable transactional properties which their permissionless counterparts lack [14].

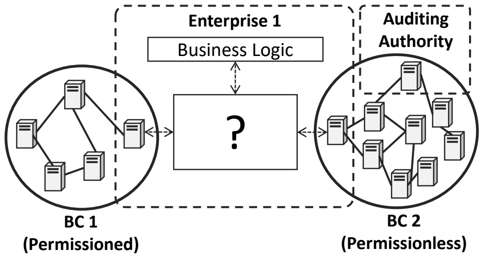


Fig. 1. The problem solved by the approach: How can we allow Enterprise 1 to utilize two different blockchain technologies?

However, these interesting enhancements come at the price of reducing decentralization due to depending on an administrative entity that decides user roles, and on a predefined set of nodes to perform transaction validation.

C. Motivation

It is often necessary for enterprises to run processes that span multiple systems. This also applies to blockchains: an enterprise could participate in multiple consortia of companies each with its own permissioned blockchain, while at the same time accepting payments using a cryptocurrency that is publicly traded on a permissionless blockchain. Another reason why processes spanning multiple blockchains are needed is that permissioned blockchains require participants to have a certain level of trust that the system administrators will not perform malicious acts, such as transaction censorship. Moreover, external entities responsible for performing audits, as well as end-users, cannot be sure that the participants of a permissioned blockchain system did not collide in reverting certain parts of the blockchain history for their own benefits [15]. This can be addressed, for example, by storing values that represent digests of the permissioned blockchain's state in a publicly accessible permissionless blockchain, which makes them immutable. An auditing entity, can then easily compare these values with the state of the permissioned blockchain and thus ensure that no malicious alterations were performed.

Figure 1 shows an example that demonstrates such a case. Here, an enterprise (Enterprise 1) is a member of a permissioned blockchain system BC1, and a permissionless blockchain BC2. By being a member of a permissioned blockchain system, we mean that the enterprise operates an *authorized* node, which is permitted to communicate with the rest of the system, read the committed state, and invoke smart contract function calls that change it. In contrast, membership in a permissionless blockchain does not require special authorization: the enterprise only needs to run an instance of the corresponding P2P protocol. In this example, Enterprise 1 uses BC1 to conduct business with a consortium of other participants. For example, assuming that Enterprise 1 is a car manufacturer, BC1 could be a blockchain handling the supply chain of car parts. Furthermore, Enterprise 1 uses BC2 to periodically record immutable digests of the confidential interactions it makes within BC1, which could be necessary for the auditing process conducted by external authorities

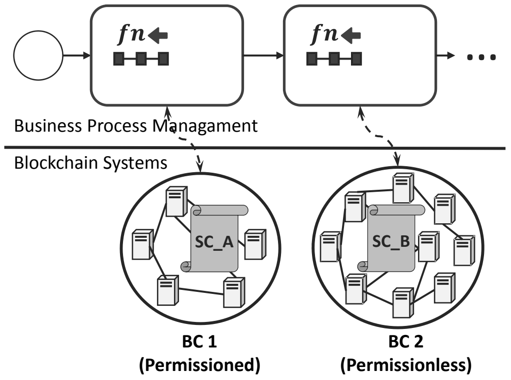


Fig. 2. Using business processes to model the interaction with permissioned and permissionless blockchain smart contracts.

responsible for, e.g., administering tax laws. Apart from that, Enterprise 1 has its own business logic which influences how and when it interacts with other enterprises via smart contracts.

For these scenarios, an application program, which acts as a node in all relevant blockchain networks, is usually used to implement how the enterprise interacts with the various systems and how this is influenced by its own business logic. If this program is implemented in an ad-hoc manner, extending the logic to further cases, or altering it according to changing requirements and urgent needs becomes difficult. Therefore, in this work we try to answer the question: *"How can we compose the functionality of permissioned and permissionless blockchain systems in a flexible and re-usable way that allows enterprises to benefit from their individual strengths while properly handling their specificities?"*

III. Approach Overview and Previous Work

In this section, we present an overview of our approach, which is process-based and allows enterprises to compose smart contracts of permissioned and permissionless blockchain systems in order to achieve the scalability, security and robustness offered by the former, and the high level of decentralization offered by the latter, while at the same time correctly handling their special interaction models.

A. Approach Overview

The approach we present allows an enterprise, being a member of one or more permissioned blockchain systems and potentially also one or more permissionless blockchain systems, to inter-operate the smart contracts it uses in these systems and at the same time incorporate its local business logic in a unified, flexible process model as shown in Fig. 2. To allow an enterprise, like Enterprise 1, to orchestrate its interactions with the smart contracts of various permissioned and permissionless blockchain systems while at the same time taking its own business logic into consideration, we propose a process-based approach. In this approach, a process model, developed in a language like BPMN [3], allows the enterprise to visually describe when and under which conditions it invokes functions of smart contracts residing in the blockchain systems it is connected to. It also models the business rules that govern these invocations. Such a process model is then instantiated and executed on a Business Process Management System (BPMS) which would translate the modeled constructs into actual messages and API calls.

Business Process Management (BPM) provides us well-established concepts, standards and tools for the specification, execution and automation of processes within and across organizations through the notion of business process models [16, 17]. Therefore, adopting business process models within our approach results in advantages on multiple levels. This includes, for example, the use of standardized modeling languages supporting complex error and compensation handling, mature tool-support through standardized process engines and modeling tools, broad application, i. e., many companies are already familiar with BPM tools and modeling languages, and straight-forward interactions and integration with other systems through services and API calls.

B. Previous Work: Blockchain-aware Business Process Modeling and Execution (BlockME)

In our previous work [4], we developed the BlockME method, which aimed at allowing existing business processes to conduct cryptocurrency-related operations with permissionless blockchains, like Ethereum and Bitcoin, by providing support at the modeling level and at the execution level. The major obstacle BlockME tried to tackle is the lack of transaction finality of most permissionless blockchain systems. As a result of this undesirable property, applications issuing permissionless blockchain transactions need to wait for several blocks to be confirmed and appended to the blockchain data structure on top of a transaction *tx* before being relatively confident that *tx* is final and will not be revoked due to blockchain forking. Apart from handling blockchain uncertainty, the approach facilitates sending and receiving cryptocurrency transfers. Overall, the following operations are supported: (i) *submitTransaction* operation, which allows submitting a blockchain transaction that transfers an amount of cryptocurrency from one account to another on the same system, and ensures that the transaction receives a certain number of *block confirmations*; (ii) *receiveTransaction* operation, which allows monitoring a specific blockchain address and issuing callbacks when it receives cryptocurrency transfers via transactions with a defined number of block confirmations; (iii) *detectOrphanedTransaction* and *ensureTransactionState* operations, which both monitor the state of a given blockchain transaction. The former of the two triggers a callback when it detects that the transaction resides in an dropped, or *orphaned*, blockchain fork, which puts it under the risk to be invalidated if a contradicting transaction exists in the replacing fork. On the other hand, the latter operation triggers a callback only when the monitored transaction receives a specific number of block confirmations, making it relatively safe to be considered durably committed. The BlockME method supports these operations at both the modeling and the execution levels.

To realize this, the method is split into three phases: In the first phase, the modeling phase, we introduced an extension to the BPMN language that abstracts the possible interactions and complex events of permissionless blockchains into easy-to-understand constructs that have the familiar look and feel of BPMN. Specifically, the proposed constructs aim at providing a visual representation of the aforementioned operations and their parameters allowing the modeler to develop a blockchain-aware process model that is easy to understand and free of potentially unmanageable clutter resulting from trying to handle blockchain uncertainty. Table I provides a brief overview of the proposed constructs. In the second phase, the resulting model is transformed into a standard-compliant BPMN model that is directly executable on process engines like Camunda [5]. To this end, we provided a set of clear transformation rules to apply. In the last phase, the execution phase, the process engine communicates with a specialized plug-in based middleware layer called the Blockchain Access Layer (BAL) that provides external applications a unified, asynchronous API that allows them to communicate with various permissionless blockchain systems. The BAL is extended with a new adapter for each blockchain system we want to support. These adapters perform translation between the high level BAL API being used by the process engine and other applications, and the APIs of individual blockchains. Table I also shows which BAL API operations support which BPMN constructs. Notice that for each high level operation, two BAL API operations are present: one to subscribe for future notifications when the corresponding conditions are met, and the other to unsubscribe from such notifications. This is due to the asynchronous nature of the operations under consideration.

BlockME suffers from two major drawbacks that prevent using it as-is to solve the new research question introduced in this paper: (i) it is only designed for permissionless blockchain systems that have a linear blockchain data structure. Therefore, the confidence that a given transaction is durably committed is measured in the number of block confirmations. For BFT-based permissioned blockchains, and for graph-based blockchains in general, a different measure is needed; (ii) it only supports cryptocurrency-related transactions that transfer value from one account to another on the same permissionless blockchain system. Therefore, the invocation of permissioned or permissionless smart contracts is not supported. In the next sections, we describe the required modifications and extensions of BlockME to support cross-blockchain composition of smart contracts. To have a clear differentiation between the old and the new approaches, we call the new one *BlockME2*.

IV. Unified Measure for Transaction Finality

In this section, we try to find a measure for transaction finality that is suitable for any kind of blockchains.

A. The Problem of Blockchain Transactions Finality

As we briefly mentioned in Section II-A, transactions of permissionless blockchains as well as certain permissioned blockchains never reach absolute finality: In such systems, the

TABLE I
OVERVIEW OF THE BPMN EXTENSION CONSTRUCTS OF BLOCKME AND THE CORRESPONDING BAL API OPERATIONS

BPMN Extension	BAL API
SubmitTransactionTask	subscribe_submitTransaction unsubscribe_submitTransaction
ReceiveTransactionTask	subscribe_receiveTransaction unsubscribe_receiveTransaction
EnsureTransactionStateTask	subscribe_ensureTransactionState unsubscribe_ensureTransactionState
OrphanedTransactionEvent	subscribe_detectOrphanedTransaction unsubscribe_detectOrphanedTransaction

creation of a new block of transactions is open to any node. Although the created block has to fulfill certain criteria, like the validity of its contents and the expensive calculation of a suitable *nonce* according to PoW, multiple nodes could come up with a valid block at relatively the same time. This results in splitting-up the network into partitions, each of which accepting a different block as the latest in the chain. Effectively, this splits the blockchain data structure into branches. On certain conditions, these branches can grow independently for some time, or can even produce further branches. To eventually come at a consistent data structure, each system has its rule of selecting the *one-true branch*. For example, Bitcoin selects the chain with the largest accumulative PoW (basically, the longest chain) as being the only valid chain. When a winning branch is chosen, transactions that only exist on the discarded branches, *orphaned transactions*, move to the memory pools of the nodes that are aware of them and are broadcast again to the remaining nodes. Normally, these transactions will be included in the valid chain after some node "mines" them into a block again. However, if the main chain already includes transactions that contradict with them, the orphaned transactions are considered invalid, and will be discarded.

This happens, for example, if the sender of the transaction is an adversary trying to implement a *double-spending attack*, in which they submit a transaction transferring an asset to peer A, and try to send the same asset in another transaction to a different peer B. The aim of this attack is tricking A into thinking that they actually received the ownership of the asset and, in return, send some valuable product back to the attacker. To make the attack succeed, the adversary tries to create a different branch in the blockchain starting from a block that precedes the one including the transaction to A. This new branch would contain the transaction to B. Furthermore, the attacker tries to convince the rest of the network that this branch is the *one-true branch* by making it the longest through mining new blocks. The success of the attack boils down to the ability of the attacker to outpace the rest of the network in generating new blocks. However, if the attacker does not control more than half of the computing power of the network, the probability that the attack succeeds diminishes exponentially over time, i.e., when further blocks are added on top of it [1]. Therefore, a recipient should not immediately consider the transaction final and rather wait for some blocks to be appended, i.e., block confirmations, before processing it. Nonetheless, the usage of block confirmations as a measure of confidence in the finality of transactions has some drawbacks, which we will discuss next.

B. Why not Using Block Confirmations?

There are three major reasons that make block confirmations a problematic system-agnostic measure for transaction finality confidence to be used by a higher abstraction layer: (i) different PoW-based blockchains have different parameters, such as the block generation interval, which results in a different recommended number of block confirmations. For example, 6 block confirmations are suggested by the standard Bitcoin client [18], whereas 12 are suggested by standard Ethereum clients [19, 20]; (ii) some blockchain technologies, such as the Tangle [21], do not use a linear chain data structure, but rather a graph. These systems measure the confidence that a transaction is valid and final in ways different than block confirmations; (iii) finally, certain permissioned blockchains, such as Hyperledger Fabric [9], use a consensus mechanism that reaches absolute finality with the first block. Therefore, waiting for more than one block confirmation is an overkill and would result in an unnecessary delay. For these reasons, we propose a different measure of transaction finality confidence.

C. Defining a new Measure

Instead of using the number of block confirmations to measure the confidence we have in the finality of a blockchain transaction *tx*, we propose using the probability that an attacker will fail to overwrite *tx* even that they control a certain percentage of the network's resources. In order to formally define this probability, first, let us define the following event:

Definition 1 (Failed Attack Event): The event that an attack involving an adversary controlling a proportion q of the voting power of a blockchain system s and trying to replace a committed transaction tx with a different transaction tx' in the common data store fails. □

Where the *common data store* refers to the data store recognized as the source of truth by the users that follow the standard protocol of s , i.e., the *honest users*. For example, the chain with the largest accumulative PoW is considered as the agreed-upon shared data store in Bitcoin. Furthermore, the *voting power* of a user refers to the degree in which the user can influence the consensus protocol. For example, the voting power in PoW is measured by the computational power, or the hash-rate, controlled by the corresponding user. Based on this event we define the following random variable:

Definition 2 (Failed Attack Duration Random Variable (X_{fail})): A random variable describing the duration of a failed attack event. □

Finally, let us define the measure we are looking for:

Definition 3 (Degree of Confidence in the Finality of Transactions of a Blockchain System s ($DoC_{s,T}$)): The probability that an attack on s fails within a given duration T, i.e., $P[X_{fail} \leq T]$ □

We notice that the previous definition is generic and technology-agnostic, which makes it applicable to a wide range of blockchian systems. However, we cannot give a general formula for $DoC_{s,T}$ that is applicable for all intended systems since they differ in many ways. Nonetheless, we know that this value typically increases quickly over time, since otherwise, s would be vulnerable to double-spending attacks (if the probability is constantly low), or not usable (if it takes a long time to sufficiently increase). Next, we see how $DoC_{s,T}$ is formulated in a number of permissionless and permissioned blockchain systems.

D. The Formulation of $DoC_{s,T}$ in Various Blockchain Systems

$DoC_{s,T}$, as explained in Definition 3, is usually formulated by the creators of every blockchain system that uses a consensus protocol guaranteeing only a probabilistic model of transaction finality, such as PoW and Tangle, since it is an important factor in proving that these systems are resistant to resource-bounded attacks. For example, S. Nakamoto [1], the creator of **Bitcoin**, formulated this probability as:

$$P[X_{fail} \leq T] = \sum_{k=0}^{z-1} \frac{\lambda^k e^{-\lambda}}{k!}(1 - (\frac{q}{1-q})^{z-k}) \qquad (1)$$

Where z represents the current *depth* (the distance from the top of the blockchain) of the block that contains tx, and since the protocol is designed so that a new block is mined at almost every 10 minutes, z can be estimated by $z \approx 600 * T$. Moreover, λ is defined as $\lambda = z\frac{q}{1-q}$. Providing that an attacker does not control more than half of the computing power of the network, i.e, $q < 0.5$, this probability increases over time.

In the case of **Ethereum**, the same formula is used. However, since the designated block interval is smaller than that of Bitcoin (12s vs 600s), i.e., $z \approx 12 * T$, Ethereum suffers from a higher *stale rate*, which refers to nodes working on mining a block of some height n when such a block is already added to the blockchain due to network latency, which effectively results in wasted computing power on the honest side of an attack scenario. Furthermore, because this interval is small, the cost of the attack decreases since the required energy is smaller. This means that attackers can invest more on buying mining equipment. Both issues affect the relative distribution of computing power between the honest nodes and the attacker "racing" with them, which is denoted as q in the previous equation. V. Buterin, the creator of Ethereum, suggests to use a $10\times$ increase of the attacker's computing power due to the aforementioned factors [22]. This explains why Ethereum suggests to wait for 12 block confirmations (which achieves $DoC_{ethereum,0.2} = 0.9998$ at $q = 0.2$), while Bitcoin suggests 6 (which also achieves $DoC_{bitcoin,0.1} = 0.9998$, but at $q = 0.1$). Moreover, a formula for the graph-based blockchain, **Tangle** is also defined in the literature [21, Equ. 11].

In the case of permissioned blockchain systems that guarantee the finality of transactions as soon as the consensus protocol is successfully executed, such as **Hyperledger Fabric** [9], **Chain** [11] and **Quorum** [23], defining a formula for this measure is straight forward; since a committed transaction cannot be replaced, a regular attack will definitely fail, i.e., $\forall T > 0, P[X_{fail} \leq T] = 1$. Nonetheless, censorship attacks can occur *before* the transaction is committed, but we do not consider them here.

V. BlockME2: A Process-based Approach for Cross-blockchain Smart Contract Composition

In this section, we explain the BlockME2 approach that allows enterprises to compose the functionality provided by various permissioned and permissionless blockchain smart contracts without worrying about the specificities of each one of these systems.

A. Applying the DoC Measure

In Section IV, we defined a measure capable of evaluating how much we are confident that a given committed transaction will not be revoked as a result of an attack. Here, we show how we apply this measure to BlockME2 both at the modeling level, and the execution level: all previously defined BPMN constructs at the modeling level (see Table I) had a parameter called *WaitUntil* that accepted the number of block confirmations as its value. To apply the new measure, we replace this parameter in these constructs with a new one called *Confidence* that takes as a value a real-number between 0 and 100 which represents the DoC as a percentage. We also add this parameter to the new construct we define later in this paper. On the other hand, the actual formula for the measure is implemented at the BAL (execution) level. However, this is not done in the technology-agnostic layer of BAL, but rather at the level of each individual adapter, which means that developers of new adapters are responsible for defining and implementing the formula that represents the probability described in Definition 3 for the corresponding system.

B. Invoking Blockchain Smart Contracts

Since blockchain smart contracts embed the transactional programs supported by such systems, invoking them is crucial for external applications trying to exploit their functionality. In this section, we explain how BlockME2 supports the invocation of smart contract functions of permissioned and permissionless blockchains both by introducing a new BPMN task, and by adding a corresponding asynchronous operation to the BAL. However, the ability to uniquely identify the specific smart contract function we want to invoke is a clear prerequisite for the success of this goal. Therefore, we first discuss how this can be achieved.

1) Addressing Smart Contract Functions: Smart contracts and their functions are identified differently based on the corresponding blockchain system they are deployed on. For example, **Ethereum** smart contracts are either identified by their *blockchain address*, which is a unique fixed-length hexadecimal value [2], or by a *human readable name* taking the form of a domain name registered using the Ethereum Name Service (ENS) [24]. A second example is **Hyperledger Sawtooth** [25], a permissioned blockchain systems that defines families of transaction programs. A *transaction family* can directly include fixed functions, like the IntegerKey transaction family [26], or can allow the deployment of user-defined smart contracts that include their own functions, like the Seth transaction family [27]. A third and a final example is **Hyperledger Fabric**, which divides the system into *channels*, in which peers install *chaincodes*. Each chaincode consists of one or more *smart contracts* that include invocable functions.

We see that in all cases, smart contract functions are addressable by specifying one or more hierarchical levels and then the function itself. Therefore, we propose to treat these functions as resources and use the Unified Resource Identifier (URI) web standard [28] to identify them. To this end, we present a new resource identifier scheme called, `scip` (which stands for "Smart Contract Invocation Protocol") with the following ABNF [29] syntax:

```
syntax = "scip://"  bcid  "/"  *(pseg  "/")  func  "?"  [pars]
         ":"  rtype
bcid   = delid
pseg   = delid
func   = id
pars   = pname  "="  ptype  *("&"  pname  "="  ptype)
pname  = id
ptype  = id
rtype  = id
delid  = 1*(ALPHA/DIGIT/"$"/"_"/"-"/"+"/".")
id     = (ALPHA/"$"/"_")  *(ALPHA/DIGIT/"$"/"_")
```

In which `bcid` refers to the identifier of the blockchain system where the smart contract function is located. Recall that an enterprise could be connected to multiple blockchian systems (even of the same type, e. g., Fabric). We assume that an identifier, which is unique within the domain of the enterprise, is associated with each of these systems, and is used here. Furthermore, `pseg` refers to a single segment in the path that spans the various levels of hierarchy needed to reach the smart contract function. Although in general, at least one path segment is necessary, which would refer to the smart contract itself, the syntax allows for an empty path to address the special case in which we refer to a system-wide function not contained in any smart contract. For example, one could query the current height of a linear blockchain using a system-wide function like `getHeight`. However, we do not predefine such special functions at this level as they are technology-specific. Additionally, `func` refers to the name of the function we want to address. Finally, `pars` and `rtype` allow concretely defining the exact function signature we want to address. The former indicates the order, name and type of all potential input parameters, whereas the latter defines the type of the returned value or the keyword `void` if no value is returned. Input and output parameter types are system-specific and are not predefined here. Below is a list of some valid smart contract function identifiers for Ethereum, Fabric, and Sawtooth:

```
scip://eth1/0xdf068aC89E6d5fa88520faace0267047e47102c2
     /getOwner?:address
scip://fabric1/channel2/chaincode2/myFunc2?newVal=int:void
scip://sawtooth2/seth/0xfa3622e1/set?key=uint&value=uint:void
```

2) Invoking Smart Contract Functions: BlockME2 supports invoking smart contract functions by introducing a new task type, namely, the **InvokeSCFunctionTask**. This task, which is visually represented in Fig. 3, has the following attributes: (i) **Function**, which identifies the smart contract function we want to call using the previously introduced `scip` scheme; (ii) **Parameters**, which is a key-value list containing parameter names and their values to be passed to the function, and (iii) **Confidence**, is the DoC (see. Section IV) that we want to achieve for the blockchain transaction issued to invoke the function in case it is not read-only. This value is represented as a percentage. Furthermore, this task has the following **operational semantics**: when it is reached, the execution stops and the smart contract function is located and invoked using the specified parameters. In case the function causes one or more write operations to the underlying blockchain, a transaction is issued and the execution remains halted until it achieves the required confidence. Afterwards, the transaction details are received and the execution continues regularly. However, if the invoked function is read-only, no transaction is needed, and the execution continues immediately after receiving the returned value. Moreover, errors that could occur while trying to invoke the function, e.g., a malformed `scip`, insufficient permissions, or an exception thrown by the function itself, are caught by a boundary message catch event and cause the execution to take an alternative flow. Similarly, if a user-defined timeout is reached before finishing the execution, another alternative flow is taken starting from the attached timer boundary event.

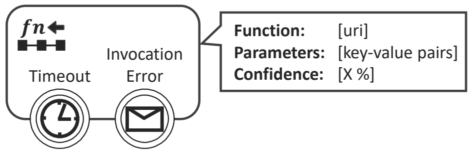


Fig. 3. Visual representation of the *InvokeSCFunctionTask* and its parameters.

On the other hand, to support the execution of this task, the BAL introduces the *invokeSCFunction* asynchronous operation. Under the hood, the BAL analyzes the `scip` URI passed to it, and extracts the `bcid` fragment in order to identify the suitable blockchain adapter. Next, it forwards the request to this adapter, which knows exactly how smart contract functions are invoked and how the DoC is measured for the specific blockchain system it is handling. Finally, when the required conditions are satisfied, the BAL notifies the process engine running the business model, and passes the result to it.

To ensure compatibility with common BPMS, we provide a transformation rule, demonstrated in Fig. 4, that allows transforming the *InvokeSCFunctionTask* into a standard compliant fragment. Accordingly, the task is transformed into a sub-process with two tasks; a send task that triggers the *invokeSCFunction* operation at the BAL, and a receive task that waits for the resulting message sent back from the BAL when the specified smart contract function is successfully invoked. Along with the functional attributes required for the operation itself, the request message contains an endpoint url to which the callback message should be sent and a correlation identifier that allows the process engine to route it to the correct

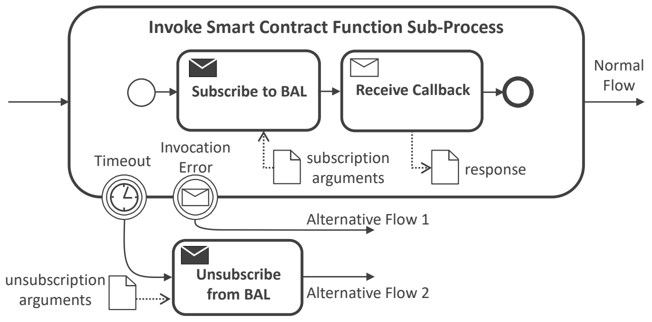


Fig. 4. Visual representation of the transformed *InvokeSCFunctionTask*.

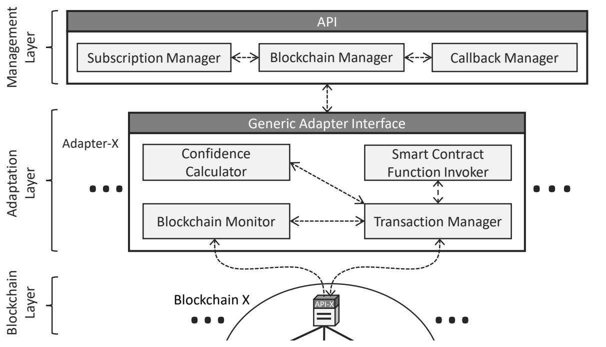


Fig. 5. The architecture of the Blockchain Access Layer (BAL)

model instance. The callback message itself also contains the correlation identifier in addition to details about the transaction that caused the invocation if the called function is not read-only, or contains a concrete value otherwise. Furthermore, timer boundary events attached to the *InvokeSCFunctionTask*, are copied and attached to the resulting sub-process. However, if such a timer event is triggered, we need to make sure to unsubscribe from the *invokeSCFunction* operation 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. Finally, to allow the resulting segment to receive potential error messages, we attach a boundary message catch event to the sub-process. When such a message is received, an alternative flow is activated. The body of the message contains the specific reason behind it, which can be used to trigger specialized compensating actions along this new execution path. An explicit unsubscription message is not required, since the BAL automatically unsubscribes after sending an error callback. A message catch event is used instead of an error catch event to capture this kind of errors, because they are of an external nature, and are not triggered by a an error end event inside the associated sub-process.

VI. System Architecture and Prototypical Validation

We validate the feasibility of the approach by providing a refined architecture for the BAL and a prototypical implementation for it. Figure 5 shows the layered architecture of the BAL. At the top, we find the management layer, which is technology-agnostic and responsible for managing the various interactions that take place with external applications via the exposed asynchronous API. To this end, the *Subscription Manager* is responsible for correlating request messages received from external applications with their responses. Furthermore, the *Callback Manager* is responsible for sending back the result of each requested operation to the endpoint specified by the external application. Finally, the *Blockchain Manager*, coordinates the work of the previous sub-components, and provides the high-level logic for all supported operations.

On the other hand, technology-specific tasks are handled by the second layer, which contains a collection of adapter instances exposing the same generic interface, but implementing their functionality differently based on the underlying blockchain. Specifically, adapters contain the following sub-components: (i) a *Confidence Calculator* that is capable of applying the formula suitable for calculating the DoC (see Definition 3) of a transaction corresponding to the supported blockchain system; (ii) a *Smart Contract Function Invoker* that manages the invocations of smart contract functions and, if necessary, requests issuing a transaction to the underlying blockchain; (iii) a *Blockchain Monitor* which detects changes in the blockchain and thus is capable of monitoring the reception of monetary transactions, or triggering the (re-)evaluation of the DoC of a given transaction. (iv) A *Transaction Manager* which is responsible for sending requests to the underlying blockchain system in the form of transactions, or in the form of read-only queries. By interacting with the *Blockchain Monitor*, and the *Confidence Calculator*, it is also capable of determining when the DoC of a transaction becomes acceptable for a certain request from the management layer.

We realized the BAL as a Java web application that exposes a RESTful API to its client applications, such as BPMS. When the application starts, it loads a configuration file containing a definition of the connected blockchain systems. Among other things, these definitions contain the local identifier to be associated with each network, as well as the information on how to access the corresponding nodes, and the needed credentials. The top layer of the BAL uses this configuration file to determine the types and number of instances of the required adapters. When these adapters are created, the top layer starts routing requests to them as previously described. Our prototype, which is publicly accessible at https://github.com/ghareeb-falazi/BlockchainAccessLayer/releases/tag/smart-contract-composition contains the implementation of Bitcoin, Ethereum and Hyperledger Fabric adapters, and in the future we plan to support further systems such as Hyperledger Sawtooth and EOS.

VII. Case Study

Figure 6 shows how the BlockME2 approach can be used to address the following simplified scenario: a car manufacturer is participating in two Fabric-based permissioned blockchains. The first represents a consortium of car part providers and

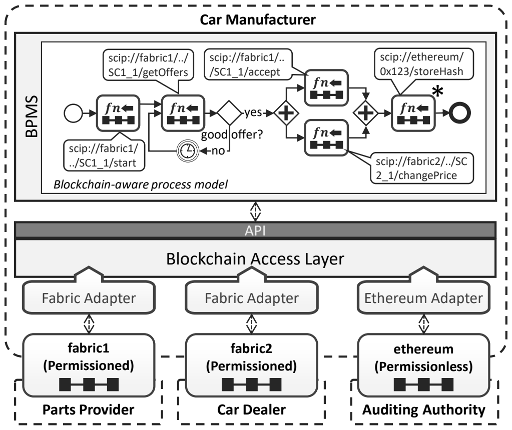
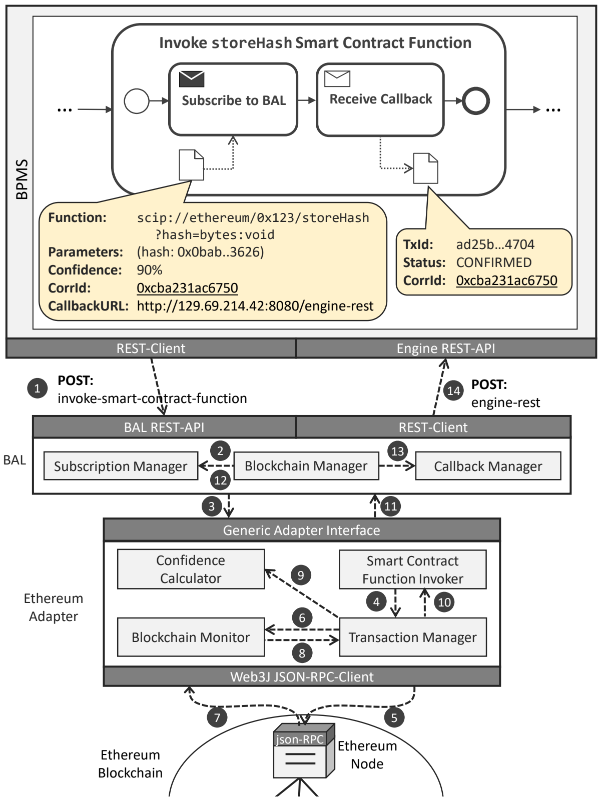

Car Manufacturer
BPMS
scip://fabric1/../SC1_1/getOffers
scip://fabric1/../SC1_1/accept
scip://ethereum/0x123/storeHash
scip://fabric1/../SC1_1/start
yes
good offer?
no
scip://fabric2/../SC2_1/changePrice
Blockchain-aware process model
API
Blockchain Access Layer
Fabric Adapter
Fabric Adapter
Ethereum Adapter
fabric1 (Permissioned)
fabric2 (Permissioned)
ethereum (Permissionless)
Parts Provider
Car Dealer
Auditing Authority

Fig. 6. A car manufacturer using the BlockME2 method to compose functionality from two permissioned blockchains and a permissionless blockchain.



Function:	scip://ethereum/0x123/storeHash ?hash=bytes:void
Parameters:	(hash: 0x0bab..3626)
Confidence:	90%
CorrId:	0xcba231ac6750
CallbackURL:	http://129.69.214.42:8080/engine-rest

TxId:	ad25b...4704
Status:	CONFIRMED
CorrId:	0xcba231ac6750

Fig. 7. The transformation and invocation procedure of an example BlockME2 *InvokeSCFunctionTask*.

car manufacturers, whereas the second represents another consortium also involving car manufacturers but with retail car dealers. The example addresses one of the possible business use-cases in which the manufacturer announces a request for offers for a set of needed car parts by invoking the `start` function of the smart contract SC1_1 located on the fabric1 network. The manufacturer then periodically retrieves a list of the offers made by invoking the read-only function `getOffers` of the same smart contract. If an offer is made with favorable criteria, the loop breaks and the offer is accepted by invoking a third function of the same smart contract, `accept`. On parallel, the pricing of some of the cars affected by this agreement is decreased for the corresponding car dealers by invoking the function `changePrice` in the SC2_1 contract of the fabric2 blockchain. At the end, the manufacturer stores a hashed summary of the involved transactions in a third smart contract on the Ethereum permissionless blockchains. This facilitates auditing by allowing an external authority to compare these immutable hashes with the values stored in the two permissioned blockchains.

After the scenario is modeled using a composition of the introduced *InvokeSCFunctionTask*, the resulting model is transformed into standard-compliant BPMN and deployed onto a BPMS operated by the car manufacturer. Then the BPMS communicates with a BAL instance, which performs the actual smart contract function invocations and returns the results back asynchronously. As an example, Fig. 7 demonstrates these steps in details for one of the smart contract invocations of this scenario, namely the last call in Fig. 6, which is annotated with an asterisk (*). At the top of this figure, we see that the *InvokeSCFunctionTask* is transformed into a sub-process called “Invoke `storeHash` Smart Contract Function” with two inner tasks according to the rule presented in Fig. 4. The figure also shows a data object passed to the inputs of the “Subscribe to BAL” send task. This data object contains the input parameters necessary to invoke the *subscribe_invokeSCFunction* operation of the BAL API. These parameters include: (i) **Function**, which specifies the Ethereum smart contract function to be invoked using a `scip` URI. In this case, a function called `storeHash`, which is located in a smart contract with the address `0x123` and has a single input and no outputs, is indicated; (ii) **Parameters**, which defines a set of key-value pairs corresponding to the input parameters indicated in the previous `scip`; in this case, a single entry is present with a value that corresponds to the digest of the previous blockchain transactions involved in the current process instance; (iii) **Confidence**, which specifies the DoC (see Section IV) that the resulting transaction should achieve before the BAL sends a callback message to the process engine. In this case, the value is 90 %, which corresponds to about 4 block confirmations in the case of Ethereum; (iv) **CorrId**, which is a randomly generated correlation identifier that permits the process engine to correctly deliver the final resulting message to the designated process instance; (v) **CallbackURL**, which is used to inform the BAL where to send the callback message. In this case, it specifies an endpoint located at: `http://129.69.214.42:8080/engine-rest`, which the process engine reserves for this purpose.

A standard-compliant business process engine, such as Camunda [5], is used to execute the transformed model, which includes this BPMN fragment. When the execution reaches the “Subscribe to BAL” send task, the following steps take place: (1) the engine issues a POST HTTP call containing a message with the previous variables to the endpoint that corresponds to the *subscribe_invokeSCFunction* operation of the BAL. Then, the *Blockchain Manager* takes control, and (2) registers the request at the *Subscription Manager*, and then (3) analyzes the provided `scip` in order to route the message to the adapter responsible for providing access to the designated blockchain, in this case Ethereum. (4) Afterwards, the *Smart Contract Function Invoker*, detects that the execution of this function requires issuing a blockchain transaction. Therefore, it delegates the task to the *Transaction Manager*. (5) Then, the manager formulates the corresponding transaction and submits it to an Ethereum node via a blockchain-specific JSON-RPC call. (6) In the next step, it triggers the *Blockchain Monitor* to start monitoring the status of the submitted transaction. The monitor then subscribes for updates from the connected Ethereum node. (7) When the node detects that the transaction is mined into a block, or that a new block is added on top of it, it sends a callback to the *Blockchain Monitor*. (8) Then the monitor informs the *Transaction Manager* of the update. The manager, then, needs to decide whether the required **Confidence** level is reached. (9) To this end, it consults with the *Confidence Calculator*, which implements the suitable DoC equation for Ethereum (see Eq. (1)), and supplies it with the necessary inputs. If the calculator indicates that the required DoC is not reached yet, steps (7), (8), and (9) are repeated until it does. (10) Then, the *Transaction Manager* informs the function invoker of the successful execution of the blockchain transaction and supplies it with its identifier. (11) The invoker, then, forwards this information to the *Blockchain Manager*, which (12) retrieves the corresponding **CorrId** and **CallbackURL** from the *Subscription Manager*, and (13) requests the *Callback Manager* to send a callback message to the process engine via the specified endpoint. (14) This is done also via an HTTP POST call that contains, among other things, the same correlation identifier sent in the original request message. Then, the engine uses this identifier to select the correct process instance and forwards the received message to it. Finally, the "Receive Callback" task emits this message as a data object, which can be further used by other activities. Similar steps are done for each of the smart contract function invocations involved in the previous example.

VIII. Related Work

Blockchain interoperability, which is abstractly defined as the ability of blockchain systems to exchange data, is a widely approached topic. Buterin [30] lists three major schemes in which interoperability can be achieved: Notary schemes, such as Interledger [31], involve a consortium of trusted intermediaries that prove to one chain that certain events took place on another chain. On the other hand, relay schemes, like Polkadot [32] or Cosomos [33], build a dedicated permissioned blockchain that plays the role of a client of other blockchain systems and facilitates their interactions. Finally, hash-locking schemes, like the Lightning Network [34], facilitate off-chain atomic swaps of tokens among heterogeneous blockchains without trusting third parties. These approaches make certain trade-offs achieving only two out of the following three properties: portability (applicability to a wide range of existing and future blockchain systems), decentralization (not depending on trusted third-parties) and functionality (is only the exchange of assets supported or also the execution of cross-chain smart contracts?). Since our approach does not provide a global blockchain interoperability solution, but rather allows an enterprise to compose the functionality of existing blockchains it is directly connected to, such trade-offs are not applicable, and the problem can be approached more directly.

On the other hand, Guida and Daniel [35] look at blockchains and smart contracts from a service-oriented viewpoint and propose to introduce smart contract descriptors and a corresponding descriptor registry together with a composition paradigm for smart contracts, which is prototypically realized for the Ethereum blockchain. However, the focus of their approach is on the development, reuse and composition of smart contracts on the level of a single blockchain, while our approach targets the composition of smart contracts within and across different blockchains by utilizing business process models as a composition paradigm. For smart contract development and reuse itself, we fully agree with the authors arguments, that BPMN-based solutions are not beneficial and that a visual development environment, as presented by them, is more helpful in supporting developers.

Other works such as Auberger and Kloppmann [36] or Schmidt et al. [37] introduce connectors to communicate with blockchains from business processes or other IT-systems. However, these approaches delegate the task of how to correctly interact with a blockchain to the application layer, which requires the involvement of blockchain experts. To abstract away blockchain-specific issues, we introduce the BAL that handles such concerns internally, and exposes a comprehensible interface to external applications eliminating the need for a blockchain expert on the other end. Furthermore, on top of the BAL, the overall approach allows enterprises to compose the functionality provided by blockchain smart contracts without worrying about their specificities by utilizing corresponding BlockME2 tasks within their process models.

Another way of combining BPM and blockchains is the use of blockchains as the underlying infrastructure to execute modeled business processes involving different partners in an immutable and trustworthy manner [38–40]. While such approaches introduce advantages regarding the execution of collaborative business processes (i. e., process choreographies) in a truly distributed manner within untrusted environments or settings, it forces collaborating partners to agree on a single blockchain technology upfront since, by default, smart contracts of one blockchain system cannot invoke smart contracts of another blockchain. As a result, this hinders the interaction with different blockchains in a business process and therefore does not allow to realize multi-blockchain scenarios as the one described in Fig. 6. We look at blockchains as external systems that allow to exchange transactions with unforeseen business partners and customers. Therefore, our approach allows to specify arbitrary interactions with different blockchain systems to invoke smart contracts and issue blockchain transactions as part of business process models to establish trust between partners, while still enabling the specification of classical business logic that does not require an additional level of trust as provided through a blockchain.

IX. Concluding Remarks and Future Work

In this work, we have presented BlockME2, an approach that facilitates the composition of smart contract functions of various permissioned and permissionless blockchain systems by allowing BPMS to invoke them via a new technology-agnostic BPMN task. To achieve this, we introduced a generic mechanism to address smart contract functions. Furthermore, we showed how it can be transformed into a standard-compliant BPMN activity that is deployable on common BPMS. Moreover, we defined a new metric that measures the degree of confidence that a certain blockchain transaction is durably committed, which is necessary for ensuring a certain amount of reliability when external applications utilize blockchain systems. Finally, we supported these new features at runtime by extending the BAL middleware with a new asynchronous operation and blockchain-specific adapters, and showed how BlockME2 can be used via a simple case study.

As a future work, we plan to introduce a new operation to BlockME2 that allows monitoring smart contract functions so that interesting situations can be detected. This would simplify the process model we presented in Section VII, for example. Moreover, we plan to define a new unified protocol for blockchain smart contract function invocation that would facilitate their interoperability and integration with existing systems. Finally, we plan to analyze and define the semantics of distributed transactions spanning multiple blockchains, and equip them with a suitable coordination protocol. This would allow to model and execute more robust business processes involving multiple blockchains.

Acknowledgments

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program "Services Computing".

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Whitepaper, 2008.
[2] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger - Byzantium version," Whitepaper, 2018.
[3] OMG, *Business Process Model and Notation (BPMN), Version 2.0*, Object Management Group Std., Rev. 2.0, Jan. 2011.
[4] G. Falazi, M. Hahn, U. Breitenbücher, and F. Leymann, "Modeling and execution of blockchain-aware business processes," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2-3, pp. 105–116, 2019.
[5] (2019) Camunda-workflow and decision automation platform. [Online]. Available: https://camunda.com/
[6] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in *CRYPTO 2003*. Springer, 2003, pp. 426–444.
[7] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, 1996.
[8] K. Croman, C. Decker, I. Eyal *et al.*, "On scaling decentralized blockchains," in *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2016, pp. 106–125.
[9] E. Androulaki, A. Barger, V. Bortnikov *et al.*, "Hyperledger Fabric: a distributed operating system for permissioned blockchains," in *Proc. of EuroSys Conference (EuroSys '18)*. ACM, 2018, pp. 30:1–30:15.
[10] B. Chase and E. MacBrough, "Analysis of the XRP Ledger consensus protocol," *arXiv preprint arXiv:1802.07242*, 2018.
[11] "Chain protocol whitepaper," Whitepaper, Chain Inc, 2019. [Online]. Available: https://chain.com/docs/1.2/protocol/papers/whitepaper
[12] M. Vukolić, "Rethinking permissioned blockchains," in *Proc. of ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC '17)*. ACM, 2017, pp. 3–7.
[13] C. Cachin and M. Vukolic, "Blockchain consensus protocols in the wild (keynote talk)," in *International Symposium on Distributed Computing (DISC 2017)*, 2017, pp. 1:1–1:16.
[14] G. Falazi, V. Khinchi, U. Breitenbücher, and F. Leymann, "Transactional properties of permissioned blockchains," *SICS Software-Intensive Cyber-Physical Systems*, 2019, to be published.
[15] K. Wst and A. Gervais, "Do you need a blockchain?" in *Proc. of 2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, Jun. 2018, pp. 45–54.
[16] M. Weske, *Business Process Management - Concepts, Languages, Architectures, 2^{nd} Edition*. Springer, 2012.
[17] F. Leymann and D. Roller, *Production Workflow - Concepts and Techniques*. PTR Prentice Hall, 2000.
[18] (2019) Bitcoin Core. [Online]. Available: https://bitcoincore.org/
[19] (2019) Go Ethereum. [Online]. Available: https://geth.ethereum.org/
[20] (2019) Parity Ethereum client. Parity Technologies. [Online]. Available: https://www.parity.io/ethereum/
[21] S. Popov, "The Tangle," Whitepaper, Apr. 2018.
[22] V. Buterin. (2015, Sep.) On slow and fast block times. [Online]. Available: https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/
[23] (2018) Quorum wiki. J.P. Morgan Chase & Co. [Online]. Available: https://github.com/jpmorganchase/quorum/wiki
[24] Ethereum name service. True Names LTD. [Online]. Available: https://ens.domains/
[25] (2019) Hyperledger Sawtooth documentation. [Online]. Available: https://sawtooth.hyperledger.org/docs/core/releases/latest/
[26] IntegerKey transaction family. [Online]. Available: https://sawtooth.hyperledger.org/docs/core/releases/latest/transaction_family_specifications/integerkey_transaction_family.html
[27] Hyperledger Sawtooth Seth documentation. [Online]. Available: https://sawtooth.hyperledger.org/docs/seth/releases/latest/
[28] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform resource identifier (URI): generic syntax*, Network Working Group Internet Standard RFC 3986, Jan. 2005. [Online]. Available: https://tools.ietf.org/html/rfc3986
[29] D. Crocker and P. Overell, *Augmented BNF for syntax specifications: ABNF*, Network Working Group Internet Standard, Rev. RFC 5234, Jan. 2008. [Online]. Available: https://tools.ietf.org/html/rfc5234
[30] V. Buterin. (2016) Chain interoperability. [Online]. Available: https://allquantor.at/blockchainbib/pdf/vitalik2016chain.pdf
[31] A. Hope-Bailie and S. Thomas, "Interledger: creating a standard for payments," in *Proc. of International Conference Companion on World Wide Web - (WWW '16 Companion)*. ACM Press, 2016.
[32] (2019) Polkadot: decentralized Web 3.0 blockchain interoperability platform. Polkadot. [Online]. Available: https://polkadot.network/
[33] (2019) Internet of blockchains - Cosmos network. Interchain Foundation. [Online]. Available: https://cosmos.network/
[34] (2019) Lightning Network. [Online]. Available: https://lightning.network/
[35] L. Guida and F. Daniel, "Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development," in *IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON 2019)*. IEEE, 2019, pp. 59–68.
[36] L. Auberger and M. Kloppmann, "Combine business process management and blockchain," May 2017.
[37] S. Schmidt, M. Jung, T. Schmidt *et al.*, "Unibright-the unified framework for blockchain based business integration," White Paper, Apr. 2018.
[38] J. Mendling, I. Weber, W. v. d. Aalst *et al.*, "Blockchains for business process management - challenges and opportunities," *ACM Transactions on Management Information Systems*, vol. 9, no. 1, Feb. 2018.
[39] L. García-Bañuelos, A. Ponomarev, M. Dumas *et al.*, "Optimized execution of business processes on blockchain," in *Business Process Management*. Cham: Springer, 2017, pp. 130–146.
[40] O. López-Pintado, L. García-Bañuelos, M. Dumas *et al.*, "Caterpillar: A blockchain-based business process management system," in *Proc. of BPM Demo Track co-located to BPM*, 2017.