Runtime Monitoring Processes Across Blockchains

Shaun Azzopardi[1] [0000−0002−2165−3698],
Joshua Ellul[2] [0000−0002−4796−5665], and
Gordon J. Pace[2] [0000−0003−0743−6272]

[1] University of Gothenburg, Sweden
shaun.azzopardi@gu.se
[2] University of Malta, Malta
{joshua.ellul,gordon.pace}@um.edu.mt

Abstract. Business processes have been long researched, with many tools, languages, and diagrammatic notations having been developed for automation. Recently, distributed ledger technology (of which Blockchain is one type) has been proposed for use in the monitoring of business process compliance. Such a set-up is attractive since it allows for immutability and thus a perfect record of the history of the business process regulated.
As blockchain platforms mature and their applications increase, one can observe that instead of having one blockchain as a 'one world computer' multiple blockchains will co-exist while possibly interacting. Existing work for business processes within the blockchain domain have focused on single isolated blockchain implementations. In this paper, we do away with this severely limiting assumption and propose a method to monitor business processes spanning different blockchains and other off-chain servers. We apply this work to business processes expressed in BPMN along with annotations proposed for a blockchain context. We further describe how we handle blockchain interoperability by synthesizing automatically off-chain monitors, acting as *notaries*, that handle message passing between blockchain systems, and how we employ hash-locking for cryptographically secure token swapping.

Keywords: blockchain· business processes· bpmn· monitoring· runtime verification.

1 Introduction

The challenge of documenting, managing and regulating business processes has long been studied. Business processes, by their very nature typically involve the interaction between different parties, for which different techniques to monitor and regulate such business processes have been proposed and adopted — by providing a centralised service which processes relevant events from involved parties, monitors the state of the business processes and enforces or otherwise regulates such interaction. Formal correctness of the monitoring service is an appropriate correctness criterion when these parties belong to a single entity, or all trust a particular service provider.

A challenge arises with decentralisation, when business processes span different entities which may not agree on a centralised party regulating their interaction. The

1. *This procurement process will regulate the purchasing and supplying process of items between a buyer and supplier, with some required minimum amount and of some maximum amount of items purchased and supplied.*
2. *A procurement process is initiated by the buyer upon placement of a deposit in escrow, amounting to the value of the pre-agreed minimum amount of items to be ordered.*
3. *The supplier similarly must then also put a performance guarantee in escrow, amounting to the pre-agreed maximum amount of items to be ordered.*
4. *The buyer and supplier then engage in an order-deliver loop.*
5. *The buyer orders items from the supplier, which uses a courier for delivery. The courier acquires a proof of delivery from the buyer and passes this on to the supplier for payment.*
6. *After at least one order-deliver iteration, the buyer can request termination of the contract. After which the buyer and the supplier can request their respective money held in escrow.*
7. *If the buyer's orders do not amount to at least the pre-agreed minimum then the difference will be taken from their deposit and given to the supplier.*
8. *If the supplier is not able to satisfy the orders (up to the pre-agreed maximum) then the value of the missed orders are transferred from the supplier's performance guarantee to the buyer.*
9. *After the money in escrow has been refunded the procurement process terminates.*

Fig. 1. A procurement business process

maritime logistics industry, for instance, faces this challenge due to the fact that their business processes range across a wide range of sectors, but also an international range of participating parties, implying different jurisdictions and legal frameworks, despite working within international regulations [**?**]. The result is a fragmented system, still dependent on physical sub-processes such as the use of *bills of lading* — legal documents providing evidence of contract of carriage, confirming receipt of goods, and resolving issues related to title of goods. This challenge of decentralised processes has been addressed in literature through the use of blockchain technology and smart contracts [**?**,**?**,**?**,**?**], which enable automated enforcement or monitoring of party interaction in a decentralised manner. In particular, work such as [**?**] showed how from a business process documented using a standard notation — Business Process Model and Notation (BPMN) [**?**] — smart contracts can be created to regulate the process.

Smart contracts address concerns of power and control of the computation in such a multiparty system by ensuring a decentralised computation engine on which the smart contract executes. However, over these past years multiple blockchains have been adopted by different groups of players working in parallel with centralised systems. Consider a procurement process described in Fig. **??** (and graphically in Fig. **??**), in which some of the parties may already be using smart contracts to decentralise their processes. For instance, the courier process may be using a blockchain system, whilst the buyer and supplier may be using a different blockchain to manage the escrow agreement, possibly also performing payment using a cryptocurrency on yet a different blockchain. The problem is that whilst adoption of decentralisation solutions between parties with direct common interests may be logistically possible, decentralising further may not be so. For instance, couriers with no link to the supplier's market do not stand to gain by integrating with the respective solution. No matter how theoretically attractive *'one blockchain to rule them all'* may sound, it is unlikely to be achievable in practice.

Theoretically, monitoring systems distributed across different locations is not a new idea [**?**], however one which still has many challenges [**?**]. The issues we wish to tackle in the blockchain case is how a business process can be decomposed across different locations, and how the decomposed sub-process monitors can communicate together across blockchains. This interaction certainly requires blockchain-to-blockchain communication techniques, which have been explored generally in [**?**].

The main contributions proposed herein include: (i) the proposal of macro business process modelling annotations that allow for the specification of location of parts of a process that spans across different systems (identifying a specific blockchain or off-chain server); and (ii) an approach that makes use of this information to synthesize automatically monitoring and regulatory smart contracts residing across different on-chain and off-chain locations, along with an appropriate inter-chain communication infrastructure. We will use the procurement use case as a motivating example.

The paper is organised as follows. In Section **??** we layout the theoretical and practical challenges to cross-chain interopability for business processes. In Section **??** we describe our solution. In Section **??** we compare this approach to related work and in Section **??** we describe remaining challenges, while we conclude in Section **??**.

2 Challenges of Full Decentralisation for Monitoring Business Processes

Business process diagrams have long been recognised as useful tools that enable for the analysis and automation of business logic. Their execution has also been explored for the purpose of monitoring and automating parts of processes (e.g. [**?**,**?**,**?**]).

Work in business process management has by and large relied on central authorities that manage, monitor, and enforce business processes. Centralisation however can be problematic — parties would not want the data to reside (centralised) under the control of another party, and may even find it difficult to agree on a third party, a central authority, to take this role. To address this issue of centralised trust, *blockchains* have been proposed to act as immutable, tamperproof and transparent records of both the business process (through appropriate smart contracts) and its history [**?**,**?**,**?**,**?**,**?**,**?**,**?**].

Blockchains provide an append-only transaction database along with computational logic which is stored and executed across all the different nodes whilst at the same time provides tamperproof guarantees. This is in contrast to the traditional centralised databases and programs, where there is only one trusted central copy of the database and software (which may be distributed across a single entity's infrastructure). Blockchains instead use *consensus algorithms* to ensure that any proposed transaction is agreed upon between the nodes before it is accepted. An attractive feature of blockchains is the immutability of its transaction history ensured through cryptography. Some blockchains also allow for the deployment of programs, commonly referred to as *smart contracts*, which provide guarantees with respect to the logic executed.

Blockchains typically also use a native cryptocurrency or token.[3] For example, Ethereum natively supports the *Ether* cryptocurrency, which functions similar to tradi-

[3] The terms are used in different ways in literature. We will use the two terms loosely here.

tional real-world monetary value through exchanges. Such cryptocurrencies are essential to the sustainable operation of many types of blockchain systems, e.g., Ethereum version 1.0 depends on *miners* who do computationally heavy work to secure and operate the blockchain who get rewarded with Ether, while transactions cost some amount of Ether (referred to as *gas*). Ether can be transferred to different account holders (or rather accounts) on the same blockchain, e.g. for payments, which may not only belong to a human owner but could be directly owned by a smart contract. Given the flexibility of the programming infrastructure which Ethereum provides, other specific-purpose tokens can be created by coding their logic using smart contracts.

In the verification of business processes through blockchain, one typically implements and encodes processes as one or more monitoring smart contracts, allowing the progress of participants to be recorded on-chain [**?**]. Such a decentralised model works well for many use cases, and is of high utility when the parties are mutually untrusting.

One often overlooked issue is that **a business process may be fragmented over different organisations**, each of which may have adopted different blockchain standards and platforms and may be unwilling to undertake modifications to support integration with other parties. A group of organisations may agree to regulate their processes on one blockchain system. However there may still be pre-existing services on which the processes depend which may not be on that blockchain. The result is that of **business process islands** with no direct communication channel between them, since existing blockchain systems do not allow for external communication calls to be initiated.

Another significant issue is that (Ethereum) **transactions can be expensive**, and their mining can be **untimely** (which can adversely affect process execution). To remedy this parties may agree to only model sensitive parts of the business process on a public blockchain (e.g. payments and exchange of proofs), potentially leaving other parts for cheaper (and faster) private blockchains. Some questions that then arise are: (i) how do we model that different participants to a process are on different blockchains; and (ii) how to handle interaction between participants on different blockchains?

Consider the use case presented in Fig. **??**, the business process may require the parties to put some money in escrow (Clause 2). This would ideally be done publicly where the escrow manager's logic is known, such that the parties do not need to trust yet another party to hold their escrowed amounts. Ethereum provides a perfect solution to this. However, doing other parts of a business process publicly does not add any value, e.g. how the buyer decides if to order or terminate the contract is irrelevant to the collaborative context. More so, there likely are aspects of the processes that are confidential which the parties would not want to reveal to prying eyes. Therefore different parties may be then agree to only execute the escrow logic publicly, while recording their own sub-processes internally in a private blockchain.

It is important to highlight that the escrow manager cannot act independently of the other parties, but instead reacts to the events they trigger (e.g. upon the buyer triggering termination, Clause 6, the logic in Clauses 6–9 must be enacted). Since the parties' processes may not be on the same blockchain as the escrow manager (or as each other), to fully automate the process there is a need to bridge the gap between blockchains.

Cross-chain messaging can be challenging. Sending a message directly from one blockchain to another is not generally possible, since blockchains act as closed systems.

Instead cross-chain messages need to be delivered by a messenger, usually called a *notary* [**?**]. This, however, adds a layer of centralisation, requiring trust by both parties that the notary is acting as expected. Finding ways to mitigate the required trust in this notary is essential to prevent conflict in cross-chain collaborative process execution.

Interaction between blockchains can be more complex than simple message sending. In the context of blockchains, two parties may want to transfer or swap tokens, e.g. the ownership of a token corresponding to the delivered good may be swapped by the supplier to the buyer in exchange for a token proving delivery. The use of a notary for this kind of logic may not be suitable for this sensitive behaviour (tokens can have monetary value). Instead a provably secure method would be ideal, for example employing notions from cryptography to identify uniquely the send and intended recipient.

3 Monitoring Business Processes across Blockchains

Our proposed approach allows the description of a decentralised business process, that may be scattered across different blockchains or servers. From this, we can automatically generate monitors as smart contracts corresponding to different participants and their subprocesses, and appropriate off-chain monitors (*notaries*) that handle cross-chain interaction between monitors on different blockchains.

In particular, we will focus on business processes specified in BPMN [**?**], augmented with location annotations. We shall then generate smart contracts for each on-chain sub-process, and assumed to be based on Ethereum instances, written in the Solidity smart contract language. A system overview is presented in Fig. **??**[4].

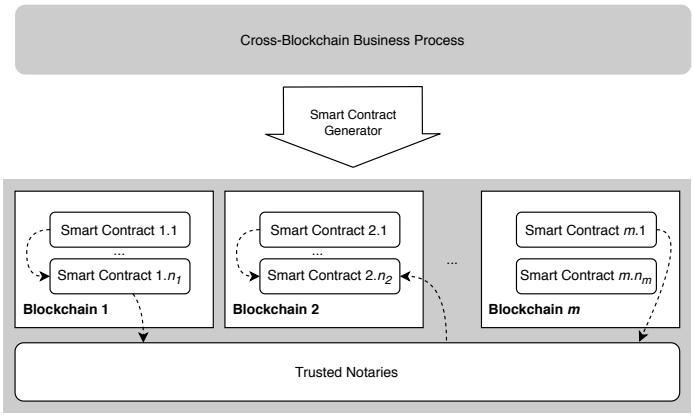

Fig. 2. System overview.

Table 1. Informal description of correspondence between BPMN symbols and their translation to Solidity code.

BPMN Symbols	As Solidity Code
Start events	A function callable by the process owner.
Throwing message events	Emits a message in the form of a Solidity event, and/or calls any corresponding catching events functions.
Catching message events	A function only callable by the message owner.
End events	Finalises the monitoring, with no function being callable after.
Pools	Correspond to one whole smart contract.
Activities	Functions that owners use to signal the activity has succeeded, or that may include some Solidity logic according to its annotations in the diagram.
Sequence flow	Handled implicitly by the smart contract. The smart contract can be queried for the next possible elements.
Message flow	Implemented through function calls when target and source of flow are on same blockchain, or through emitting events and notary listening otherwise.
Exclusive Gateway	Conditions on outgoing flows are parameters to corresponding function, with if-then-else logic used to continue the flow.
Event-based Gateway	Activated automatically depending on the received events.
Parallel Gateway	Execution forks in parallel directions, through keeping an array of the next elements in sequence.

3.1 Business Process Monitoring through Smart Contracts

Modelling and execution of business processes on blockchains is not new, with multiple approaches and tools being developed for this purpose [**?**,**?**,**?**,**?**]. Such approaches focused on processes that are bound to execute within a single blockchain. We build on such previous work and extend the state-of-the-art by proposing cross-blockchain and system BPMN. We assume basic knowledge of BPMN (see Fig. **??** as an example), for the standard specification see [**?**], and of Solidity, for its documentation see [**?**]. We discuss how salient BPMN aspects are facilitated through Solidity smart contracts.

In our context a BPMN diagram will be a set of pools with message flows between them, with each pool belonging to a different party to the process. We distinguish two aspects of a BPMN diagram: (i) the behaviour within the pool; and (ii) the collaborative behaviour between pools. For each pool, we generate smart contracts that monitor for the flow of the enclosed process. This is done by keeping track of the business flow through the process' elements, from a start element to an end one. Since there may be parallel flows we allow for multiple possible 'next' elements. This could be problematic with conditional gateways, but we provide for their immediate triggering. This is reflected in the below code snippet, where `Elements` is the type of elements in the process, and `next` at each point in time marks set of next elements:

```
1 enum Elements {Start, SendDeposit, Order, ..., End}
2
```

[4] Find a prototype here: `https://github.com/shaunazzopardi/bpmn-to-solidity`.

```
3 mapping(Elements => bool) next;
```

Appropriate activation functions for each element are synthesized, which can only be called successfully when the corresponding element is next in the flow. These functions have appropriate access control that allows them to only be triggered by the respective participant. The following code snippet illustrates a function that is called by the buyer party to activate the start event (with the only flow being to *SendDeposit*):

```
function trigger_Start() {
    require(msg.sender == buyer);

    if (nextBusinessFlowPoint[Elements.Start]) {
        nextBusinessFlowPoint[Elements.Start] = false;
        nextBusinessFlowPoint[Elements.SendDeposit] = true;
    }
}
```

This access control also takes care of points in the process that depend on triggering by another participant (either from a different smart contract or a different blockchain). Table. **??** summarises the translation for a select number of BPMN symbols. We discuss in some detail how cross-chain communication works next.

3.2 Communication across Blockchains

Different participant pools can communicate with each other in the context of a larger business process. Existing solutions assume that this communication happens on the same blockchain, but we do not make that assumption here. For example, a group of participants may not be interested in every detail of each other's business process, but only that the synchronising behaviour between them is correct. Another motivation is that public blockchains can be expensive, thus it is more efficient to carry out only the critical part of the business process on a public blockchain while leaving the rest for each party's private blockchains. To show which blockchain a pool is targeted for we use a text annotation that includes an identifier for a blockchain. For example, a pool annotated with `Ethereum` is aimed for the Ethereum blockchain.

We allow for different types of interaction: (i) message flows; and (ii) token transfers and swaps. Flows between participants are appropriately tagged when they are of the latter type. When the participants use one blockchain this interaction is not problematic. Participants being on different blockchains requires specialised approaches trusted by each participant. Here we present our proposal for this interoperability.

Message Passing In our approach cross-chain message flows are facilitated through a trusted notary. A notary in this context is an off-chain monitor that acts as an intermediary between on-chain monitors. We implement a notary as a Python service.

Essentially the notary establishes a connection to each smart contract on the respective blockchains. Through this connection it subscribes for events intended for cross-chain communication. Upon event triggering it calls the appropriate function in the intended recipient's smart contract passing the message payload.

The involved parties can inspect this code to ensure it will behave as expected. However, once deployed to a traditional server the trustless environment is lost and each

party must inherently trust that the web service corresponds to the code they agreed to. To tackle this issue of trust the parties may agree to place the notary code on a trusted server owned and operated by both of them. However, ideally the processes are designed in such a way that off-chain logic need not be trusted. Another approach is for each party to monitor separately each other's smart contract to ensure each message is received unmodified (when using a blockchain visible to both parties).

Token Transfers and Atomic Swapping One may want to encode some blockchain-specific logic such as token transfers for payments, allowing them to be directly synthesized from a BPMN diagram. We allow this by tagging inter-pool message flows not just by a message name but also by annotations describing token transfers or swaps.

For transfers we simply specify the type of the token to be sent, e.g. when sending ether we simply insert: *[Send: ETH]*, in a message flow label. When instead we wish to swap tokens we also specify the type of token to be received, e.g. if we want to swap some ether with some bitcoin we would write: *[Send: ETH, Receive: BTC]*. Here we use the annotations solely to be able to create the necessary monitoring infrastructure for the swapping and transfer, leaving the parties to agree on amounts and other case-specific validation code. In the future we envision these to also be encoded in the diagram.

For message passing our solution was simply to use a trusted notary that passes messages from one blockchain to another. Logic surrounding token transfers and swaps can however be more critical, since these may have real-world value.

Implementation-wise we simply assume that the source party will transfer the tokens to an address belonging to the target party, and that the target party will perform appropriate validation. The alternative here is simply to have a notary act as a middle-man, but this does not add any utility but only creates a possible point of failure. The following code snippet illustrates the function in the target escrow smart contract that a buyer calls to deposit ether. The code is only slightly different for custom tokens.

```
uint depositAmount;
function receive_deposit() public{
    require(msg.sender == buyer);
    require(msg.value == depositAmount);

    if(next[Elements.SendDeposit]){
        next[Elements.SendDeposit] = false;
        next[Elements.Order] = true;
    }
}
```

To handle cross-chain swaps instead we use a cryptographically secure trustless protocol — *hash-locking* [**?**]. Consider two parties, each owning tokens on different blockchains, and that wish to swap them. Party A initiates the atomic swap by locking their tokens in a smart contract with a certain hash, corresponding to a secret only they know. The tokens can be withdrawn only upon presenting the secret. Party B then ensures the tokens have been locked, and then lock their counterpart tokens on their blockchain with the same hash. Party A then uses their exclusive knowledge of the secret to unlock Party B's token, upon which Party B is informed of the secret and unlocks the tokens Party A locked initially. To ensure Party A cannot just withdraw both tokens a time lock is enforced, see [**?**] for more detail. The smart contracts we

produce have this ability to hash lock tokens, while a notary is used to monitor at which stage the swap is at, and to notify parties about locks and unlocks.

The code snippet below illustrates functions related to swapping, for example the event `swap_initiated_Pay` is used by the buyer to initiate a swap where they lock an amount of ether with a certain hash. Appropriate events are emitted both for the notary to pass on the message to the other party (line 6) and by the notary to notify the buyer that the swap has been reciprocated (lines 10–13).

```
1  event SwapInitiatedPay(bytes32 indexed contractId, bytes32 indexed hash);
2  function swap_initiated_Pay(bytes32 _hash, uint _amount) payable public{
3      require(msg.sender == buyer);
4      require(msg.value >= _amount);
5      bytes32 contractId = hashlock(_amount, _hash);
6      emit SwapInitiatedPay(contractId, _hash);
7  }
8
9  event SwapReciprocatedPay(bytes32 indexed _contractId);
10 function swap_reciprocated_Pay(bytes32 _contractId) public{
11     require(msg.sender == notary);
12     emit SwapReciprocatedPay(_contractId);
13 }
```

Off-Chain Processes We also allow for processes to be deployed off-chain (as NodeJS servers). This may be beneficial when the required logic is too expensive to be performed on a public blockchain. The code of these servers follows closely that of the smart contracts, with the only exception being that these servers can communicate with and listen to the blockchain without the need of the notary.

4 Case Study

Table 2. Total gas costs of running the business process when all sub-processes are: (i) on the same blockchain; and (ii) different blockchains.

Configuration	Process	Transaction Costs (in gas)	Execution Costs (in gas)	Total
Same blockchain	Buyer	1244352	3316585	4560937
	Escrow	959764	2778631	3738395
	Supplier	1192312	3448979	4641291
	Courier	923832	2606714	3530546
	Total #1	**4427980**	**12311111**	**16739091**
All different blockchains	Buyer	1415148	3674380	5089528
	Escrow	964568	2794075	3758643
	Supplier	1476248	3878837	5355085
	Courier	1009084	3015735	4024819
	Total #2	**4865048**	**13363027**	**18228075**
	Cost of Distribution (Total #2 - Total #1)	437068	1051916	1488984

To illustrate our approach, Fig. **??** shows a BPMN business process designed for the procurement contract illustrated in Fig. **??**. For brevity we only model a process

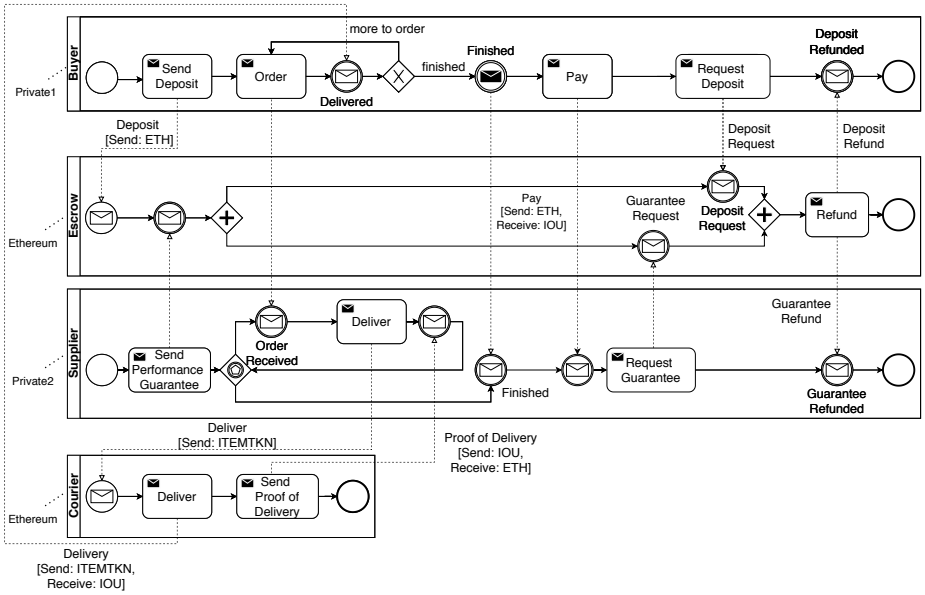

Fig. 3. Business process diagram for procurement contract (Fig. **??**).

that is compliant with the contract, and ignore possible misbehaviour. The parties to the process are: (i) a buyer; (ii) an escrow handler; (iii) a supplier; and (iv) a courier.

The escrow handler receives a deposit from the buyer and supplier, and refunds them when both parties have requested their respective deposit. This participant is annotated by the *Ethereum* tag, signalling that it is to be deployed on the Ethereum public blockchain. Since this will handle tokens of both parties they are both interested in it being carried out in a fully public and transparent manner, to ensure auditability.

The buyer repeatedly orders items from the supplier, and waiting for their delivery. The supplier, in turn, reacts to the buyer's orders by engaging the courier to deliver the ordered items and waiting to receive proof-of-delivery. When the buyer no longer wishes to order any further items they effect payment and request their deposit.

The courier process is started upon receiving from the supplier an amount of tokens representing the number items to be delivered. Upon delivery these are swapped with IOU tokens backed by the buyer, which they guarantee they can be swapped with ether. The courier swaps these IOUs as proof of delivery and for delivery payment with the supplier. In turn the supplier can swap these IOUs with the buyer for payment of the items. In the maritime context the proof of delivery required by the supplier corresponds to bills of lading (a delivered items receipt). In this context the supplier would be able to present all the collected bills of lading and demand payment from the buyer. Note how both the buyer and supplier are on different private blockchains, and thus the message flow from the buyer's *Order* activity must flow between these blockchains. Here, implicitly a notary is passing on this message between the blockchains.

The ease with which we can reconfigure the placement of different parts of the business process model indicates that our framework is also useful at the design and configuration level, whenever the location where the monitoring is to be deployed can be determined by the parties setting it up.

To determine the viability of our approach in this case we measured the gas costs associated with executing the process on Ethereum. In Table **??** we present the results in terms of the costs associated with the full execution of each sub-process, in terms of execution and transaction costs, for different chain placements of the sub-processes. Transaction costs relate to the cost of initiating a transaction, while execution costs are directly correlated with the computational intensity of the logic executed. We consider that all the sub-processes are deployed on the same blockchain (the theoretically lowest cost option), and that each is on a different blockchain (the theoretically highest cost option). Here we are assuming every smart contract is deployed to a public blockchain, however a party may use a private permissioned blockchain or simply an off-chain server, resulting in no need for gas payments (except for party communication).

From our experiments, the costs associated with our approach for a full business process is substantial. Consider that if we take the USD value of the total gas costs of the theoretically lowest cost (that all the processes are on the same blockchain) is around 120 USD[5]. Implementing the process across different blockchains increases this value by around 10 USD, we term this the *cost of distribution*.

From this we can conclude that performing the monitoring of a whole business process on a blockchain is not insignificant, and its attractiveness depends on the profit margin of the industry. However, our tests show that moving parts of the business process across blockchains does not cause inordinate substantial increases in costs. This can justify moving non-sensitive parts of the business process onto costless private blockchains, thus off-setting any monetary costs due to the distribution. This supports the utility of our approach to allow a BPMN model to be spread across different blockchains and/or off-chain servers. For example, if the supplier process is not on a public blockchain, then more than 30 USD can be shaved off the total cost. The decision of how to distribute a business process depends on many variables, including the level of trust between parties, the availability of private blockchains.

5 Related Work

This work is not the first to propose a blockchain-based solution to business process management. [**?**] proposes the use of a shared distributed ledger to log the delivery of sensor-equipped parcels, enabling service agreements to be monitored for and evaluated in a trustless manner. [**?**] proposes the use of a private permissioned blockchain between different parties using Hyperledger Fabric.

There are also a number of tools that facilitate the use of blockchain for similar purposes as our work. Weber et al. [**?**] employ the blockchain in two ways to support business process management: (i) to monitor message exchanges between processes

[5] When taking a conservative (at the time of writing) average, as of November 2020, gas cost of 16 Gwei per gas unit.

and ensure conformance while also facilitating payments and escrow; and (ii) to coordinate the whole collaborative process execution. There are other tools available that perform similar functions, e.g. Caterpillar [**?**], ChorChain [**?**], and Lorikeet [**?**]. [**?**] extend BPMN 2.0 choreography diagrams to utilise the power of smart contracts to enable a shared data model between processes and also the execution of some logic. Since we use process diagrams the notion of data models is already compatible with our approach, while we also allow the diagram to reference certain scripts. These works are very similar to what we offer with our approach. However we take a step further by dealing with a cross-chain distributed business processes and automatically generate off-chain code that facilitates cross-chain communication securely.

Ladleif et al. [**?**] is the only work we found that deals with multi-chain business processes. This work allows multi-chain choreographies, employing off-chain adapters as channels between smart contracts in different blockchains. Our approach on the other hand handles BPMN collaborations (as opposed to choreographies, i.e. the message flow between different participants). As opposed to simple cross-chain message flows our approach also adds richer interaction, including token transfers and atomic swaps.

Recent related work [**?**,**?**] considers the decomposition of global process compliance rules into rules that can be securely verified in a distributed way without requiring parties to disclose sensitive details. In our case decomposition of the business process is done simply according to the location annotations for pools.

In other work we took more formal approaches to general monitoring and analysis of smart contracts, using deontic logic [**?**] and automata-based behavioural specifications [**?**,**?**,**?**]. In that line of work we deal simply with the monitoring of one single smart contract. In this paper we do not simply monitor for compliance of the business logic, but also provide ways to carry it out across blockchains.

For a more detailed discussion and overview of general challenges for blockchain-based business process management see [**?**,**?**].

6 Remaining Challenges

From a security and trust point of view a weak point in our approach is the use of off-chain notaries to handle cross-chain communication. These act as oracles, i.e. trusted off-chain services that workaround the closed-world assumption of blockchains to keep them up to date with off-chain data sources — see [**?**] for a discussion and characterization of oracle patterns and costs. Since these notaries are off-chain we lose the guarantees ensured by a blockchain environment. For example, if the notary is deployed to a traditional server then there is no assurance of immutability of the code, and the owner of the server could potentially change the notary's code unilaterally. A challenge here is how to remove the need for trust, at least partially. One could explore set-ups involving the use of a common server, or cryptographic signatures of the application binary.

Our case study in Fig. **??** models the expected behaviour in a business process. However things can go wrong, for example one of the parties may not allow a swap to be carried out. One would want to include ways to recover from such misbehaviour, e.g. by allowing the swap to be re-attempted. In previous work we have explored this kind of recovery in the form of reparations [**?**] or compensations [**?**]. We have already

explored this [**?**,**?**] in the context of other formal specifications for verification on the blockchain. Applying this work in the context of our annotations however remains an open problem.

Assessing the suitability of our tool requires appropriate testing of the produced artifacts. These artifacts however are not meant to be deployed in a single environment, but across different environments, which can be on-chain or off-chain. On the blockchain side, one can make use of testnets or private blockchains. While for off-chain artifacts one can simply deploy them to local servers. We are currently working on an approach to automatically generate code that performs a test runs of business processes.

We intend to apply this business process management approach in the maritime context. For this we will need to extend our prototype for a larger fragment of BPMN, including also allowing multiple participants of certain type. This is required for example to allow dynamic onboarding of courier services, rather than simply assuming one static courier service. Another possible extension is to consider that different couriers may be used in different contexts or for different goods.

Implementation wise we are limited in that we only consider Solidity smart contracts. In a future iteration it would be ideal to be able to produce code tailored for different blockchains. Moreover, our text annotations could be enriched by identifying when message flows are carried out by the same notary or not. Different pairs of participants may trust to deploy notaries on different servers. We are also investigating code optimisations, e.g., we use loops which can be costly in Ethereum.

7 Conclusions

Blockchains have been successfully proposed and used as vehicles for business process monitoring and execution. Multiple approaches and tools already exist. In this paper we extend this work in a novel direction, where we do not assume all participants are located on the same blockchain. We further consider annotations to BPMN diagrams that express blockchain related notions, e.g. transfer or swapping of tokens, and annotations of participants with the blockchain they will be located on. Our approach, with an associated prototype, generates from BPMN a set of smart contracts (currently limited Solidity). To handle cross-chain interaction we also generate notaries to securely pass on messages between participants, while we employ the notion of hash-locking to handle cryptographically safe cross-chain swaps of tokens.