# Optimized Execution of Business Processes on Blockchain

Luciano García-Bañuelos[1], Alexander Ponomarev[2], Marlon Dumas[1], and Ingo Weber[2,3]

[1] University of Tartu, Estonia
{luciano.garcia, marlon.dumas}@ut.ee
[2] Data61, CSIRO, Sydney, Australia
{alex.ponomarev, ingo.weber}@data61.csiro.au
[3] School of Computer Science & Engineering, UNSW, Sydney, Australia

**Abstract.** Blockchain technology enables the execution of collaborative business processes involving untrusted parties without requiring a central authority. Specifically, a process model comprising tasks performed by multiple parties can be coordinated via smart contracts operating on the blockchain. The consensus mechanism governing the blockchain thereby guarantees that the process model is followed by each party. However, the cost required for blockchain use is highly dependent on the volume of data recorded and the frequency of data updates by smart contracts. This paper proposes an optimized method for executing business processes on top of commodity blockchain technology. Our optimization targets three areas specifically: initialization cost for process instances, task execution cost by means of a space-optimized data structure, and improved runtime components for maximized throughput. The method is empirically compared to a previously proposed baseline by replaying execution logs and measuring resource consumption and throughput.

## 1 Introduction

Blockchain technology enables an evolving set of parties to maintain a safe, permanent, and tamper-proof ledger of transactions without a central authority [1]. In this technology, transactions are not recorded centrally. Instead, each party maintains a local copy of the ledger. The ledger is a linked list of blocks, each comprising a set of transactions. Transactions are broadcasted and recorded by each participant in the blockchain network. When a new block is proposed, the participants in the network agree upon a single valid copy of this block according to a consensus mechanism. Once a block is collectively accepted, it is practically impossible to change it or remove it. Hence, a blockchain can be seen as a replicated append-only transactional data store, which can replace a centralized register of transactions maintained by a trusted authority. Blockchain platforms such as Ethereum[4] additionally offer the possibility of executing scripts on top

[4] https://www.ethereum.org/ – last accessed 4/3/2017

of a blockchain. These so-called *smart contracts* allow parties to encode business rules on the blockchain in a way that inherits from its tamper-proofness.

Blockchain technology opens manifold opportunities to redesign collaborative business processes such as supply chain and logistics processes [2]. Traditionally, such processes are executed by relying on trusted third-party providers such as Electronic Data Interchange (EDI) hubs or escrows. This centralized architecture creates entry barriers and hinders process innovation. Blockchain enables these processes to be executed in a distributed manner without delegating trust to central authorities nor requiring mutual trust between each pair of parties.

Previous work [3] demonstrated the feasibility of executing collaborative processes on a blockchain platform by transforming a collaborative process model into a smart contract serving as a template. From this template, instance-specific smart contracts are spawned to monitor or execute each instance of the process. The evaluation in [3] put into evidence the need to optimize resource usage. Indeed, the cost of using a blockchain platform is highly sensitive to the volume of data recorded and the frequency with which these data are updated by smart contracts. Moreover, the deployment of instance-specific contracts entails a major cost. In order to make blockchain technology a viable medium for executing collaborative processes, we need to minimize the number of contract creations, the code size, the data in the smart contracts, and the frequency of data writes.

This paper proposes an optimized method for executing business processes defined in the standard Business Process Model and Notation (BPMN) on top of commodity blockchain technology. Specifically, the paper presents a method for compiling a BPMN process model into a smart contract defined in the Solidity language – a language supported by Ethereum and other major blockchain platforms. The first idea of the method is to translate the BPMN process model into a minimized Petri net and to compile this Petri net into a Solidity smart contract that encodes the "firing" function of the Petri net using a space-optimized data structure. The second idea is to restrict the number of contract creations to the minimum needed to retain isolation properties. Furthermore, we optimized the runtime components to achieve high throughput rates. The scalability of this method is evaluated and compared to the method proposed in [3] by replaying artificial and real-life business process execution logs of varying sizes and measuring the amount of paid resources (called "gas" in Ethereum) spent to deploy and execute the smart contracts encoding the corresponding process models.

The next section introduces blockchain technology and prior work on blockchain-based process execution. Sec. 3 presents the translation of BPMN to Petri nets and to Solidity code. Sec. 4 discusses architectural and implementation optimizations. Sec. 5 presents the evaluation, and Sec. 6 draws conclusions.

## 2 Background and Related Work

### 2.1 Blockchain Technology

The term blockchain refers both to a network and a data structure. As a data structure, a blockchain is a linked list of blocks, each containing a set of transac-

tions. Each block is cryptographically chained to the previous one by including its hash value and a cryptographic signature, in such a way that it is impossible to alter an earlier block without re-creating the entire chain since that block. The data structure is replicated across a network of machines. Each machine holding the entire replica is called a *full node*. In *proof-of-work blockchains,* such as Bitcoin and Ethereum, some full nodes play the role of *miners*: they listen for announcements of new transactions, broadcast them, and try to create new blocks that include previously announced transactions. Block creation requires solving a computationally hard cryptographic puzzle. Miners race to find a block that links to the previous one and solves the puzzle. The winner is rewarded with an amount of new crypto-coins and the transaction fees of all included transactions.

The first generation of blockchains were limited to the above functionality with minor extensions. The second generation added the concept of *smart contracts*: scripts that are executed whenever a certain type of transaction occurs and which read and write from the blockchain. Smart contracts allow parties to enforce that whenever a certain transaction takes place, other transactions also take place. For example, a public registry for land titles can be implemented on a blockchain that records who owns which property at present. By attaching a smart contract to sales transactions, it is possible to enforce that when a sale takes place, the corresponding funds are transferred, the tax is paid, and the land title is transferred, all in a single action.

The *Ethereum* [4] blockchain treats smart contracts as first-class elements. It supports a dedicated language for writing smart contracts, namely Solidity. Solidity code is translated into bytecode to be executed on the so-called *Ethereum Virtual Machine (EVM)*. When a contract is deployed through a designated transaction, the cost depends on the size of the deployed bytecode [5]. A Solidity smart contract offers methods that can be called via transactions. In the above example, the land title registry could offer a method to read current ownership of a title, and another one for transferring a title. When submitting a transaction that calls a smart contract method, the transaction has to be equipped with crypto-coins in the currency *Ether*, in the form of *gas*. This is done by specifying a gas limit (e.g. 2M gas) and gas price (e.g., $10^{-8}$ Ether / gas), and thus the transaction may use up to gas limit $\times$ price (2M $\times 10^{-8}$ Ether = 0.02 Ether). Ethereum's cost model is based on fixed gas consumption per operation [5], e.g., reading a variable costs 50 gas, writing a variable 5-20K gas, and a comparison statement 3 gas. Data write operations are significantly more expensive than read ones. Hence, when optimizing Solidity code towards cost, it is crucial to minimize data write operations on variables stored on the blockchain. Meanwhile, the size of the bytecode needs to be kept low to minimize deployment costs.

### 2.2 Related Work

In prior work [3], we proposed a method to translate a BPMN choreography model into a Solidity smart contract, which serves as a factory to create choreography instances. From this factory contract, instance contracts are created by providing the participants' public keys. In the above example, an instance could

be created to coordinate a property sale from a vendor to a buyer. Thereon, only they are authorized to execute restricted methods in the instance contract. Upon creation, the initial activity(ies) in the choreography is/are enabled. When an authorized party calls the method corresponding to an enabled activity, the calling transaction is verified. If successful, the method is executed and the instance state is updated, i.e. the executed activity is disabled and subsequent ones are enabled. The set of enabled activities is determined by analyzing the gateways between the activity that has just been completed, and subsequent ones.

The state of the process is captured by a set of Boolean variables, specifically one variable per task and one per incoming edge of each join gateway. In Solidity, Boolean variables are stored as 8-bit unsigned integers, with 0 meaning `false` and 255 meaning `true`.[5] Solidity words are 256 bits long. The Solidity compiler we use has an in-built optimization mechanism that concatenates up to 32 8-bit variables into a 256-bit word, and handles redirection and offsets appropriately. Nevertheless, at most 8 bits in the 256-bit word are actually required to store the information – the remaining are wasted. This waste increases the cost of deployment and write operations. In this paper, we seek to minimize the variables required to capture the process state so as to reduce execution cost (gas).

In a vision paper [6], the authors argue that the data-aware business process modeling paradigm is well suited to model business collaborations over blockchains. The paper advocates the use of the Business Artifact paradigm [7] as the basis for a domain-specific language for business collaborations over blockchains. This vision however is not underpinned by an implementation and does not consider optimization issues. Similarly [8] advocates the use of blockchain to coordinate collaborative business processes based on choreography models, but without considering optimization issues. Another related work [9] proposes a mapping from a domain specific language for "institutions" to Solidity. This work also remains on a high level, and does not indicate a working implementation nor it discusses optimization issues. A Master's thesis [10] proposes to compile smart contracts from the functional programming language Idris to EVM bytecode. According to the authors, the implementation has not been optimized.

## 3 From Process Models to Smart Contracts

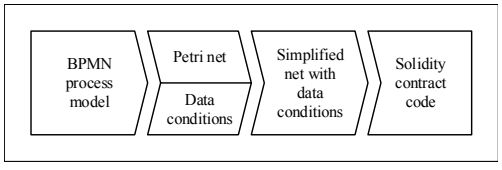



Fig. 1: Chain of transformations

The first and central component of the proposal is a method for transforming a given BPMN process model into a smart contract that can coordinate the execution of one process instance from start to end. Fig. 1 shows the main steps of this method. The method takes as input

[5] https://github.com/ethereum/EIPs/issues/93 – last accessed 20/3/2017

a BPMN process model. The model is first translated into a Petri net. An analysis algorithm is applied to determine, where applicable, the guards that constrain the execution of each task. Next, reduction rules are applied to the Petri net to eliminate invisible transitions and spurious places. The transitions in the reduced net are annotated with the guards gathered by the previous analysis. Finally, the reduced net is compiled into Solidity. Below, we discuss each step in turn.

### 3.1 From BPMN to Petri nets

The proposed method takes as input a BPMN process model consisting of the following types of nodes: tasks, plain and message events (including start and end events), exclusive decision gateways (both event-based and data-based ones), merge gateways (XOR-joins), parallel gateways (AND-splits), and synchronization gateways (AND-joins). Fig. 2 shows a running example of BPMN model. Each node is annotated with a short label (e.g. $A, B, g1 \ldots$) for ease of reference.

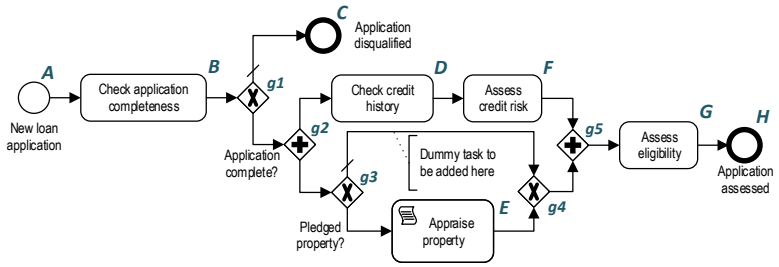



Fig. 2: Loan assessment process in BPMN notation

To simplify subsequent steps, we pre-process the BPMN model to materialize every *skip flow* as a dummy "skip" task. A skip flow is a sequence flow from an XOR-split to a XOR-join gateway such as the one between *g3* and *g4* in Fig. 2. Moreover, if the BPMN model has multiple end events, we transform it into an equivalent BPMN model with a single end event using the transformation defined for this purpose in [11]. In the case of the model in Fig. 2, this transformation adds an XOR-join at the end of the process that merges the incoming flows of the two end events, and connects them to a single end event. Conversely, if the process model has multiple start events, we merge them into a single one.

The pre-processed BPMN model is then translated into a Petri net using the transformation defined in [12]. This transformation can turn any BPMN process model (without OR-joins) into a Petri net.[6] The transformation rules in [12] corresponding to the subset of BPMN considered in this paper are presented in Fig. 3. Fig. 4 depicts the Petri net derived from the running example. The tasks and events in the BPMN model are encoded as labeled transitions (A, B, ...). Additional transitions without labels (herein called $\tau$ transitions) are introduced by the transformation to encode gateways as per the rules in Fig. 3, and to capture the dummy tasks introduced to materialize skip flows.

[6] The transformation cannot handle escalation and signal events and non-interrupting boundary events, but these constructs are beyond the scope of this paper.

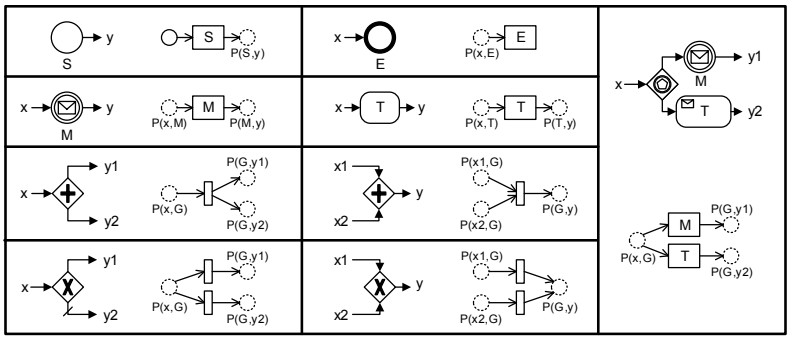



Fig. 3: Mapping of BPMN elements into Petri nets

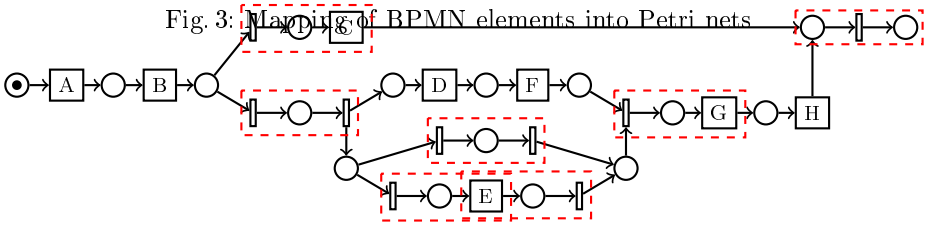
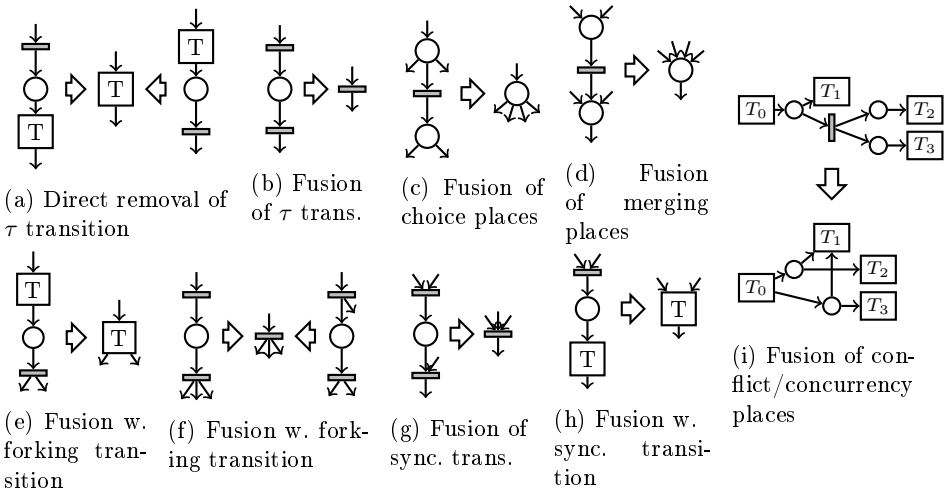



Fig. 4: Petri net derived from the BPMN model in Fig. 2

The transformation in [12] produces so-called *workflow nets*. A workflow net has one source place (start), one sink place (end), and every transition is on a path from the start to the end. Two well-known behavioral correctness properties of workflow nets are (i) *Soundness:* starting from the marking with one token in the start place and no other token elsewhere (the *initial marking*), it is always possible to reach the marking with one token in the end place and no other token elsewhere; and (ii) *Safeness:* starting from the initial marking, it is not possible to reach a marking where a place holds more than one token. These properties can be checked using existing tools [12]. Herein we restrict ourselves to workflow nets fulfilling these correctness properties. The latter property allows us to capture the current marking of the net by associating a boolean to each place (is there a token in this place or not?), thus enabling us to encode a marking as a bit array.

### 3.2 Petri net reduction

The Petri nets produced by the transformation in [12] contain many $\tau$ transitions. If we consider each transition as an execution step (and thus a transaction on the blockchain), the number of steps required to execute this Petri net is unnecessarily high. It is well-known that Petri nets with $\tau$ transitions can be reduced into smaller equivalent nets [13] under certain notions of equivalence. Here, we use the reduction rules presented in Fig. 5. Rules (a), (b), and (e)-(h) are fusions of series of transitions, whereas rules (c) and (d) are fusions of series of places. Rule(i) deals with $\tau$ transitions created by combinations of decision gateways and AND-splits. It can be proved that each of these reduction rules produces a Petri net that is *weak trace equivalence* to the original one, i.e. it generates the same traces (modulo $\tau$ transitions) as the original one.

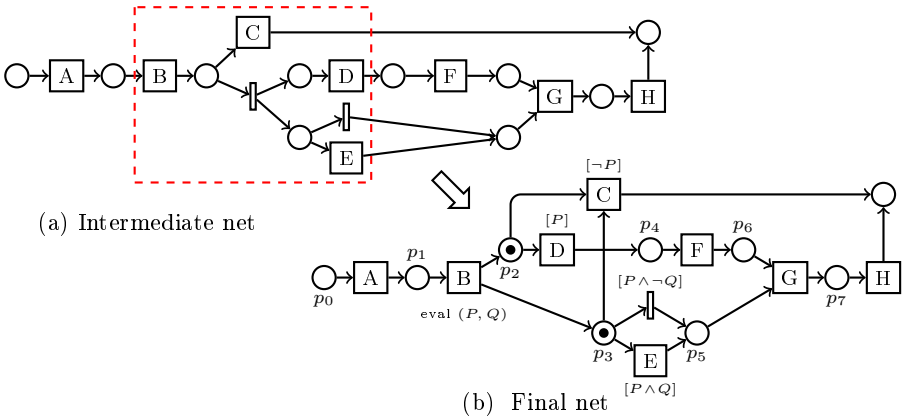



(a) Direct removal of $\tau$ transition (b) Fusion of $\tau$ trans. (c) Fusion of choice places (d) Fusion of merging places (i) Fusion of conflict/concurrency places

(e) Fusion w. forking transition (f) Fusion w. forking transition (g) Fusion of sync. trans. (h) Fusion w. sync. transition

Fig. 5: Petri net reduction rules

The red-dashed boxes in Fig. 4 show where the reduction rules can be applied. After these reductions, we get the net shown in Fig. 6a. At this point, we can still apply rule (i), which leads to the Petri net in Fig. 6b.




(a) Intermediate net

(b) Final net

Fig. 6: Reduced Petri net corresponding to the BPMN model in Fig. 2

### 3.3 Data conditions collection

Some of the $\tau$ transitions generated by the BPMN-to-Petri net transformation correspond to conditions attached to decision gateways in the BPMN model. Since some of these $\tau$ transitions are removed by the reduction rules, we need to collect them back from the original model and re-attach them to transitions in the reduced net, so that they are later propagated to the generated code.

Algorithm 1 collects the conditions along every path between two consecutive tasks in a BPMN model, and puts them together into a conjunction. Its output is one conjunctive condition – herein called a *guard* – per task in the original BPMN model. When given the start event as input, the algorithm applies a classical recursive depth-first traversal. It uses two auxiliary functions: (i) SUCCESSORSOF, which returns the direct successors of a node; and (ii) COND, which returns the condition attached to a flow. Without loss of generality, we assume that every outgoing flow of a decision gateway is labeled with a condition (for a default flow, the condition is the negation of the conjunction of conditions of its sibling flows). We also assume that any other flow in the BPMN model is labeled with condition *true* – these *true* labels can be inserted via pre-processing.

**Algorithm 1** Algorithm for collection of data conditions

```
1: global guards: Map⟨Node ↦ Cond⟩ = ∅, visited: Set⟨Node⟩ = ∅
2: procedure COLLECTCONDITIONS(curr: Node, predicate: Cond)
3:     guards[curr] ← predicate
4:     visited ← visited ∪ { current }
5:     for each succ ∈ SUCCESSORSOF(curr) : succ ∉ visited do
6:         if curr is a Gateway then
7:             COLLECTCONDITIONS(succ, predicate ∧ COND(curr, succ))
8:         else
9:             COLLECTCONDITIONS(succ, true)
```

We illustrate the algorithm assuming it traverses the nodes in the model of Fig. 2 in the following order: $[A, B, g1, g2, g3, E, \ldots]$. First, procedure COLLECTCONDITIONS sets guards $= \{(A, true)\}$ in line 3 and proceeds until it calls itself recursively (line 9) with the only successor node of $A$, namely $B$. Note that predicate is reset to *true* in this recursive call. Something similar happens in the second step, where guard is updated to $\{(A, true), (B, true)\}$. Again, the procedure is recursively called in line 9, now with node $g1$. This time guards is updated to $\{(A, true), (B, true), (g1, true)\}$ and since $g1$ is a gateway, the algorithm reaches line 7. There, the procedure is recursively called with succ $= g2$ and predicate $= (true \wedge P)$, or simply $P$, where $P$ represents the condition "Application complete?". Since the traversal follows the sequence $[A, B, g1, g2, g3, E, \ldots]$, it will eventually reach node $E$. When that happens, guards will have the value $\{(A, true), (B, true), (g1, true), (g2, P), (g3, P), (E, P \wedge Q)\}$, where $Q$ represents the condition "Pledged property?". Intuitively, the algorithm propagates and combines the conditions $P$ and $Q$ while traversing the path between nodes $B$ to $E$. When the algorithm traverses $E$, the recursive call is done in line 9, where predicate is set to *true*, i.e. the predicate associated with $E$ is not propagated further.

The guards gathered by the algorithm are attached to the corresponding transitions in the reduced net. In Fig. 6b the collected guards are shown as labels above each transition. To avoid cluttering, *true* guards are not shown. The $\tau$ transition in the net in Fig. 6b corresponds to the skip task that was inserted in the BPMN model in Fig. 2, hence this $\tau$ transition has a guard.

For each transition in the reduced Petri net, we can now determine the conditions that need to be evaluated after it fires. To do so, we first compute the set of transitions that are reachable after traversing a single place from each transition, and then analyze the guards associated to such transitions. In our example, we can reach transitions $\{C, D, E, \tau\}$ by traversing one place starting from transition $B$. Hence, conditions $P$ and $Q$ need to be evaluated after task $B$ is executed. This is represented by attaching a label $eval(P, Q)$ to transition $B$.

### 3.4 From reduced Petri net to Solidity

In this step, we generate a Solidity smart contract that simulates the token game of the Petri net. The smart contract uses two integer variables stored on the blockchain: one to encode the current marking and the other to encode the value of the predicates attached to transitions in the reduced net. Variable marking is a bit array with one bit per place. This bit is set to zero when the place does not have a token, or to one otherwise. To minimize space, the marking is encoded as a 256-bits unsigned integer, which is the default word size in the EVM.

Consider the reduced Petri net in Figure 6b. Let us use the order indicated by the subscripts of the labels associated to the places of the net. The initial marking (i.e. the one with a token in $p_0$) is encoded as integer 1 (i.e. $2^0$). Hence, we initialize variable marking with value 1 when an instance smart contract is created. This marking enables transition $A$. The firing of $A$ removes the token from $p_0$ and puts a token in $p_1$. Token removal is implemented via bitwise operations: marking = marking & uint($\sim$1);. Similarly, the addition of a token in $p_1$ (i.e. $2^1$ hence 2) is implemented via bitwise operations: marking = marking | 2;.

Variable predicates stores the current values of the conditions attached to the Petri net transitions. This variable is also an unsigned integer representing a bit array. As before, we first fix order the set of conditions in the process model, and associate one bit in the array per condition. For safety, particularly in the presence of looping behavior, the evaluation of predicates is reset before storing the new value associated with the conditions that a given transition computes. For instance, transition $B$ first clears the bits associated with conditions $P$ and $Q$ (i.e. $2^0$ and $2^1$, respectively), and then stores the new values accordingly.

When possible, an additional space optimization is achieved by merging variables marking and predicates into a single unsigned integer variable. The latter is possible if the number of places plus the number of predicates is at most 256. Note that this is not a restriction of our approach: If more space is needed to represent a process model, multiple 256-bit variables can be used.

Algorithm 2 sketches the functions generated for each transition in the reduced Petri net. Item 1 sketches the code for transitions associated to user tasks, while Item 2 does so for transitions associated to script tasks and $\tau$ transitions with predicates. For $\tau$ transitions without predicates, no function is generated, as these transitions only relay tokens (and this is done by the step function).

In summary, the code generated from the Petri net consists of a contract with the two variables marking and predicates, the functions generated as per Algorithm 2 and the step function. This smart contract offers one public function

per user task (i.e. per task that requires external activation). This function calls the internal step function, which fires all enabled transitions until it gets to a point where a new set of user tasks are enabled (or the instance has completed).

**Algorithm 2** Sketch of code generated for each transition in the reduced net

1. For each transition associated to a user task, generate a public function with the following code:
   - If task is enabled (i.e. check marking and predicates), then
     (a) Execute the Solidity code associated with the task
     (b) If applicable, compute all predicates associated with this task and store the results in a local bit set, tmpPreds
     (c) Call step function with new marking and tmpPreds, to execute all the internal functions that could become enabled
     (d) Return TRUE to indicate the successful execution of the task
   - Return FALSE to indicate that the task is not enabled
2. For each transition associated with a script task or $\tau$ transition that updates predicates, generate an internal function with the following code:
   (a) Execute the Solidity code associated with the task
   (b) If applicable, compute all predicates associated with this task and store the results in a local bit set, tmpPreds
   (c) Return the new marking and tmpPreds (back to the step function)

An excerpt of the smart contract generated for the running example is given in Listing 1.1. The excerpt includes the code corresponding to transitions $B$, $E$ and the $\tau$ transition. Transition $B$ corresponds to task CheckApplication. The corresponding function is shown in lines 4-17 in Listing 1.1. Since this is a user task, the function is called explicitly by an external actor, potentially with some data being passed as input parameters of the call (see line 4). In line 5, the function checks if the marking is such that $p_2$ holds a token, i.e., if the current call is valid in that it *conforms* to the current state of the process instance. If so, the function executes the script task (line 6 is a placeholder for the script). Then the function evaluates predicates $P$ and $Q$ (lines 8-9). Note that the function does not immediately updates variable predicates but stores the result in a local variable tmpPred, which we initialized in line 7. In this way, we defer updating variable predicates as much as possible (cf. line 39) to save gas (predicates is a contract variable stored in the blockchain and writing to it costs 5000 gas). For the same reason, the new marking is computed in line 11 but the actual update to the respective contract variable marking is deferred (cf. line 39).

Listing 1.1: Excerpt of Solidity contract

```
1  contract BPMNContract {
2    uint marking = 1;
3    uint predicates = 0;
4    function CheckApplication( -input params- ) returns (bool) {
5      if (marking & 2 == 2) { // is there a token in place p1?
6        // Task B's script goes here, e.g. copy value of input params to contract variables
7        uint tmpPreds = 0;
8        if ( -eval P- ) tmpPreds |= 1; // is loan application complete?
9        if ( -eval Q- ) tmpPreds |= 2; // is the property pledged?
10       step(
11         marking & uint(~2) | 12,        // New marking
12         predicates & uint(~3) | tmpPreds // New evaluation for "predicates"
13       );
14       return true;
15     }
16     return false;
17   }
18   function AppraiseProperty(uint tmpMarking) internal returns (uint) {
```

```
19      // Task E's script goes here
20      return tmpMarking & uint(~8) | 32;
21    }
22    function step(uint tmpMarking, uint tmpPredicates) internal {
23      if (tmpMarking == 0) { marking = 0; return; } // Reached a process end event!
24      bool done = false;
25      while (!done) {
26          // does p3 have a token and does P ∧ Q hold?
27          if (tmpMarking & 8 == 8 && tmpPredicates & 3 == 3) {
28            tmpMarking = AppraiseProperty(tmpMarking);
29            continue;
30          }
31          // does p3 have a token and does P ∧ ¬Q hold?
32          if (tmpMarking & 8 == 8 && tmpPredicates & 3 == 2) {
33            tmpMarking = tmpMarking & uint(~8) | 32;
34            continue;
35          }
36          ...
37          done = true;
38      }
39      marking = tmpMarking; predicates = tmpPredicates;
40    }   ... }
```

After executing $B$, if condition $P$ holds the execution proceeds with the possibility of executing $E$ or the $\tau$ transition. $E$ is a script task and can be executed immediately after $B$, if condition $Q$ holds, without any further interaction with external actors. For this reason, the Solidity function associated with task $E$ is declared as internal. In the Solidity contracts that we create, all internal functions are tested for enablement, and if positive, executed. Specifically, the last instructions in any public function of the smart contract call a generic step function (cf. lines 22-40 in Listing 1.1). This function iterates over the set of internal functions, and executes the first activated one it finds, if any. For instance, after executing $B$ there are tokens in $p_2$ and $p_3$. If $P \wedge Q$ holds, then the step function reaches line 28, where it calls function AppraiseProperty corresponding to transition $E$. This function executes the task's script in line 19 and updates marking in 20. After this, the control returns to line 29 in the step function, which restarts the while loop. Once all the enabled internal functions are executed, we exit the while loop. In line 39, the step function finally updates the contract variables.

## 4 Architecture and Implementation Optimization

In this section, we describe the improvements we have made in terms of architecture and implementation, relative to our earlier work on this topic [3].

**Architecture Optimization.** As introduced in Section 2, in [3] we proposed an architecture wherein a process model is mapped to a "factory" smart contract. For each instantiation, this factory contract creates an "instance" smart contract with the code necessary to coordinate the process instance. The instance contract is bound to a set of participants, determined at instantiation time. While this ensures isolation between different groups of participants, it is wasteful if the same group of participants repeatedly executes instances of the same model. In the latter case, the code encapsulating the coordination logic is redeployed for each instance, and contract deployment is particularly expensive.

To avoid this cost, we give the option to combine the factory and instance smart contracts into one, i.e., one smart contract that can handle running multiple instances in parallel. Instead of creating one bitvector per instance, we maintain an extensible array of bitvectors, each encoding the state of a process instance. On deployment, the array is empty. Creating a process instance assigns an instance ID and creates a new bitvector, which is appended to the array. This option is applicable when one group of actors repetitively executes instances of the same process. In situations where the actors differ across process instances, the option with separate factory and instance contracts should be used.

**Implementation Optimization.** During initial throughput experiments, we discovered that our original trigger implementation was a bottleneck on throughput. Our hypothesis was that we should be able to optimize the performance of the trigger to the point where it is no longer a bottleneck, i.e., a single trigger can handle at least as much throughput as the blockchain itself. To test this hypothesis, we improved our trigger by: (1) switching to asynchronous, non-blocking handling of concurrent requests, to achieve a high degree of parallelism in an environment that lacks full multi-threading. (2) Using the inter-process communications (IPC) channel to communicate with the blockchain software, *geth*, which runs the full blockchain node for a given trigger. During a small experiment we found that IPC can be 25× faster than the previously used HTTP connection. IPC requires the trigger and *geth* to run on the same machine, HTTP allows more flexible deployment architectures – but the performance advantage was too significant to ignore it during performance optimization. (3) Switching to asynchronous interaction with *geth*, which is a prerequisite for using IPC. The above changes required an almost complete rewrite of the code. The resulting throughput performance results are presented in the next section.

## 5 Evaluation

The goal of the proposed method is to lower the cost, measured in *gas*, for executing collaborative business processes when executed as smart contracts on the Ethereum blockchain. Thus, we evaluate costs with our improvements comparatively against the previous version. The second question we investigate is that of throughput: is the approach sufficiently scalable to handle real workloads?

### 5.1 Datasets

We draw on four datasets (i.e., logs and process models) described in Table 1. Three datasets are taken from our earlier work [3], the *supply chain*, *incident management*, and *insurance claim* processes, for which we obtained process models from

| Process | Tasks | Gateways | Trace type | Traces |
|---|---|---|---|---|
| Invoicing | 40 | 18 | Conforming | 5,316 |
| Supply chain | 10 | 2 | Conforming | 5 |
| | | | Not conforming | 57 |
| Incident mgmt. | 9 | 6 | Conforming | 4 |
| | | | Not conforming | 120 |
| Insurance claim | 13 | 8 | Conforming | 17 |
| | | | Not conforming | 262 |

Table 1: Datasets used in the evaluation.

the literature and generated the set of conforming traces. Through random manipulation, we generated sets of non-conforming traces from the conforming ones. The fourth dataset is stemming from a real-world invoicing process, which we received in the form of an event log with 65,905 events. This log was provided to us by the Minit process mining platform[7]. Given this log, we discovered a business process model using the Structured BPMN Miner [14], which showed a high level of conformance ($> 99\%$). After filtering out non-conforming traces, we ended up with dataset that contains 5,316 traces, out of which 49 traces are distinct. The traces are based on 21 distinct event types, including one for instance creation, and have an average length of 11.6 events.

### 5.2 Methodology and Setup

We translated the process models into Solidity code, using the previous version of the translator from [3] – referred to as *default* – and the newly implemented translator proposed in this paper – referred to as *optimized*. For the optimized version, we distinguish between the two architectures, i.e., the previous architecture that deploys a new contract for each instance, and the architecture that runs all process instances in a single contract. We refer to these options at *Opt-CF* ("CF" for control flow) and *Opt-Full*, respectively. Then we compiled the Solidity code for these smart contracts into EVM bytecode and deployed them on a private Ethereum blockchain.

To assess gas cost and correctness on conformance checking, we replayed the log traces against all three versions of the contracts and recorded the results. We hereby relied on the log replayer and trigger components from [3], with the trigger improvements discussed in Section 4. The replayer iterates through the traces in a log and sends the events, one by one, via a RESTful Web service call to the trigger. The trigger accepts the service call, packages the content into a blockchain transaction and submits it. Once it observes a block that includes the transaction, it replies to the replayer with meta-data that includes block number, consumed gas, transaction outcome (accepted or failed, i.e., non-conforming), and whether the transaction completed this process instance successfully. The replayer has been modified to cater for concurrent replay of thousands of traces.

Experiments were run using a desktop PC with an Intel i5-4570 quadcore CPU without hyperthreading. Ethereum mining for our private blockchain was set to use one core. The log replayer and the trigger ran on the same machine, interacting via the network interface with one another. For comparability, we used the same software versions as in the experiments reported in [3], and a similar blockchain state as when they were run in February–March 2016. For Ethereum mining we used the open-source software geth[8], version v1.5.4-stable.

### 5.3 Gas Costs and Correctness of Conformance Checking

For each trace, we recorded the gas required for initialization of a new process instance (deploying an instance contract or creating a new bitvector, depending

[7] http://www.minitlabs.com/ – last accessed 13/3/2017
[8] https://github.com/ethereum/go-ethereum/wiki/geth – last accessed 20/3/2017

on the architecture), the sum of the gas required to perform all the required contract function invocations, the number of rejected transactions due to non-conformance and the successful completion of the process instance.

The results of this experiment are shown in Table 2. The base requirement was to maintain 100% conformance checking correctness with the new translator, which we achieved. Our hypothesis was that the optimized translator leads to *strictly monotonic improvements* in cost on the process instance level. We tested this hypothesis by pairwise comparison of the gas consumption per trace, and confirmed it: all traces for all models incurred less cost in *Opt-CF*. In addition to these statistics, we report the absolute costs as averages.

| Process | Tested Traces | Variant | Avg. Cost Instant. | Avg. Cost Exec. | Savings (%) |
|---|---|---|---|---|---|
| Invoicing | 5316 | Default | 1,089,000 | 33,619 | – |
| | | Opt-CF | 807,123 | 26,093 | -24.97 |
| | | Opt-Full | 54,639 | 26,904 | -75.46 |
| Supply chain | 62 | Default | 304,084 | 25,564 | – |
| | | Opt-CF | 298,564 | 24,744 | -2.48 |
| | | Opt-Full | 54,248 | 25,409 | -42.98 |
| Incident mgmt. | 124 | Default | 365,207 | 26,961 | – |
| | | Opt-CF | 345,743 | 24,153 | -7.04 |
| | | Opt-Full | 54,499 | 25,711 | -57.96 |
| Insurance claim | 279 | Default | 439,143 | 27,310 | – |
| | | Opt-CF | 391,510 | 25,453 | -8.59 |
| | | Opt-Full | 54,395 | 26,169 | -41.14 |

Table 2: Gas cost experiment results

As can be seen from the table, the savings for *Opt-CF* over *default* are small for the simple *supply chain* process with only two gateways – cf. Table 1 – whereas they are considerably larger for the complex *invoicing* process with 18 gateways. Considering the reduction rules we applied, this can be expected. The other major observation is that *Opt-Full* yields massive savings over *Opt-CF*. When considering the absolute cost of deploying a contract vs. the average cost for executing a single transaction and the resulting relative savings, it is clear that the improved initialization is preferable whenever the respective architecture is applicable. As discussed in Section 4, this cost reduction also results in a loss of flexibility, and thus the choice requires a careful, case-specific tradeoff.

### 5.4 Throughput Experiment

To comparatively test scalability of the approach, we analyze the throughput using the three variants of contracts, *default, Opt-CF,* and *Opt-Full*. To this end, we used the largest of the four datasets, *invoicing*, where we ordered all the events in this log chronologically and replayed all 5,316 traces at a high frequency. The three variants were tested in separate campaigns. To ensure conformance, the events within a single trace were replayed sequentially. Ethereum's miners keeps a transaction pool, where pending transactions wait to be processed.

One major limiting factor for throughput is the gas limit per block: the sum of consumed gas by all transactions in a block cannot exceed this limit, which is set through a voting mechanism by the miners in the network. To be consistent with the rest of the experimental setup, we used the block gas limit from March 2016 at approx. 4.7M gas, although the miner in its default setting has the option to increase that limit slowly by small increments. Given the absolute gas cost in Table 2, it becomes clear that this is fairly limiting: for *Opt-CF*, instance contract creation for the invoicing dataset costs approx. 807K gas, and thus no more than 5 instances can be created within a single block; for *default*, this number drops

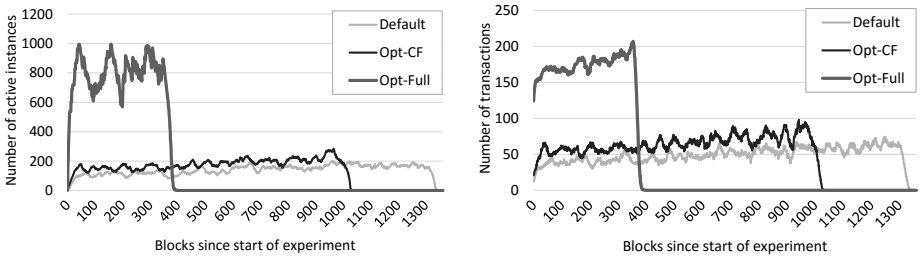



Fig. 7: Throughput results. Left: # of active instances. Right: # of transactions per block, smoothed over a 20-block time window.

to 4. Regular message calls cost on average 26.1K / 33.6K gas, respectively for *optimized* / *default*, and thus a single block can contain around 180 / 140 such transactions at most. These numbers would decrease further when using a public blockchain where we are not the only user of the network.

Block limit is a major consideration. However, block frequency can vary: on the public Ethereum blockchain, mining difficulty is controlled by a formula that aims at a median inter-block time of 13-14s. As we have demonstrated in [3], for a private blockchain we can increase block frequency to as little as a second. Therefore, when reporting results below *we use blocks as a unit of relative time.*

Fig. 7 shows the process instance backlog and transactions per block. Note that each datapoint in the right figure is averaged over 20 blocks for smoothing. The main observation is that *Opt-Full* completed all 5,316 instances after 403 blocks, *Opt-CF* needed 1,053 blocks, and for *default* it took 1,362 blocks. This underlines the cost results above: due to the network-controlled gas limit per block, the reduced cost results in significant increases in throughput.

## 6 Conclusion

This paper presented a method to compile a BPMN process model into a Solidity smart contract, which can be deployed on the Ethereum platform and used to enforce the correct execution of process instances. The method minimizes gas consumption by encoding the current state of the process model as a space-optimized data structure (i.e. a bit array with a minimized number of bits), reducing the number of operations required to execute a process step, and reducing initialization cost where possible. The experimental evaluation showed that the method significantly reduces gas consumption and achieves considerably higher throughput relative to a previous baseline.

The presented method is a building block towards a blockchain-based collaborative business process execution engine. However, it has several limitations, including: (i) it focuses on encoding control-flow relations and data condition evaluation, leaving aside issues such as how parties in a collaboration are bound to a process instance and access control issues; (ii) it focuses on a "core subset" of the BPMN notation, excluding timer events, subprocesses and boundary events for example. Addressing these limitations is a direction for future work.

**Acknowledgements.** This research was started at the Dagstuhl seminar #16191 – *Fresh Approaches to Business Process Modeling.* The research is partly supported by the Estonian Research Council. (grant IUT20-55).

## References

1. UK Government Chief Scientific Adviser: Distributed ledger technology: Beyond block chain. Technical report, UK Government Office of Science (2016)
2. Milani, F., García-Bañuelos, L., Dumas, M.: Blockchain and business process improvement. BPTrends newsletter (October 2016)
3. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: Proc. of BPM, Springer (2016) 329–347
4. Buterin, V.: Ethereum white paper: A next-generation smart contract and decentralized application platform. First version (2014) Latest version: https://github.com/ethereum/wiki/wiki/White-Paper – last accessed 29/11/2016.
5. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. homestead revision (23 June 2016) https://github.com/ethereum/yellowpaper.
6. Hull, R., Batra, V.S., Chen, Y.M., Deutsch, A., Heath III, F.F.T., Vianu, V.: Towards a shared ledger business collaboration language based on data-aware processes. In: Proc. of ICSOC, Springer (2016)
7. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Syst. J. **42**(3) (July 2003) 428–445
8. Norta, A.: Creation of smart-contracting collaborations for decentralized autonomous organizations. In: Proc. of BIR, Springer (2015) 3–17
9. Frantz, C.K., Nowostawski, M.: From institutions to code: Towards automated generation of smart contracts. In: Workshop on Engineering Collective Adaptive Systems (eCAS), co-located with SASO, Augsburg. (2016)
10. Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. Master's thesis, Dept. of CS&E, Chalmers University of Technology & University of Gothenburg, Sweden (2015)
11. Kiepuszewski, B., ter Hofstede, A.H.M., van der Aalst, W.M.P.: Fundamentals of control flow in workflows. Acta Inf. **39**(3) (2003) 143–209
12. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology **50**(12) (2008) 1281–1294
13. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4) (April 1989) 541–580
14. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Bruno, G.: Automated discovery of structured process models: Discover structured vs. discover and structure. In: Proc. of ER, Springer (2016) 313–329