EasyChair Preprint

№ 1356

Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications

Christopher Klinkmüller, Alexander Ponomarev, An Binh Tran, Ingo Weber and Wil van der Aalst

August 1, 2019

Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications

Christopher Klinkmüller[1], Alexander Ponomarev[1], An Binh Tran[1], Ingo Weber[2,3], and Wil van der Aalst[4]

[1] Data61, CSIRO, Level 5, 13 Garden St, Eveleigh NSW 2015, Australia
{christopher.klinkmuller, alex.ponomarev, anbinh.tran}@data61.csiro.au
[2] Technische Universitaet Berlin, Germany, ⟨firstname⟩.⟨lastname⟩@tu-berlin.de
[3] University of New South Wales, NSW 2052, Australia
[4] RWTH Aachen University, Germany, wvdaalst@pads.rwth-aachen.de

Abstract. Blockchain technology has been gaining popularity as a platform for developing decentralized applications and executing cross-organisational processes. However, extracting data that allows analysing the process view from blockchains is surprisingly hard. Therefore, blockchain data are rarely used for process mining. In this paper, we propose a framework for alleviating that pain. The framework comprises three main parts: a manifest specifying how data is logged, an extractor for retrieving data (structured according to the XES standard), and a generator that produces logging code to support smart contract developers. Among others, we propose a convenient way to encode logging data in a compact form, to achieve relatively low cost and high throughput for on-chain logging. The proposal is evaluated with logs created from generated logging code, as well as with existing blockchain applications that do not make use of the proposed code generator.

Keywords: Process mining, blockchain, smart contracts, logging, XES

1 Introduction

Blockchain technology has been gaining popularity as a platform for developing *decentralized application*s (DApp) [14] that are, amongst others, used to execute cross-organisational processes [13,3,10,7]. In such cases, *process mining* [1] can assist developers in (i) understanding the actual usage of the DApp, (ii) comparing it to the intended usage, and (iii) adapting the DApp accordingly. A prerequisite for the application of process mining technology is the availability of *event data*, e.g., stored in the form of XES logs. Yet, extracting such data from DApps is surprisingly hard, as demonstrated by Di Ciccio et al. [2] on the attempt of extracting meaningful logs from the Caterpillar on-chain BPMS [5]. The challenges derive from a mismatch between the *logged data* and the event data required for analysis, e.g., minimising logged information keeps the cost and data volume manageable. Challenges also arise from the underlying technology itself, e.g., Ethereum's block timestamps refer to the time when mining started, not to the block production. Moreover, as the DApp's source code is by default

not shared, process participants are potentially left with cryptic information that is hard to decode.

To alleviate this pain, we propose a framework for extracting process event data from Ethereum-based DApps that utilize Ethereum's transaction log as a storage for logged data. The framework comprises three main parts:

- The *manifest* enables users to capture and share their view of how data logged by a DApp should be interpreted from a process perspective. It is input to all other parts and is processed without access to the source code. Thus, our framework eliminates the need to share DApp code. To support users in developing a manifest, our framework includes a *validator*, which checks if a particular manifest adheres to the rules outlined in this paper.
- The *extractor* retrieves logged data from the Ethereum transaction log, applies the rules from the manifest to transform the logged data into event data, and formats this data according to the XES standard [4]. As a consequence, the extracted data can readily be imported into process mining tools from academia and industry (e.g., ProM, Celonis, ProcessGold, Disco, Minit, QPR, Apromore, and RapidProM).
- The *generator* automatically creates logging functionality from the manifest. It further includes proposals for several optimisations, such as a means for encoding logged data in the compact form of a bitmap, which helps in achieving relatively low cost and high throughput for on-chain logging.

The proposal is evaluated with logs created from generated logging code, as well as with an existing DApp. It was created by developers other than the authors of this paper and thus demonstrates the universal applicability of the framework.

In the following, we first introduce relevant background information on process mining, blockchain and logging in Section 2. The approach is introduced in Section 3 and evaluated in Section 4, before Section 5 concludes.

2 Background

2.1 Process Mining and Process Event Data

Process Mining. The roots of process mining lie in the *Business Process Management* (BPM) discipline where it was introduced as a way to infer workflows and to effectively use the audit trails present in modern information systems. Evidence-based BPM powered by process mining helps to create a common ground for business process improvement and information systems development. The uptake of process mining is reflected by the growing number of commercial tools including Celonis, Disco, ProcessGold, Minit, myInvenio and QPR. Examples like Siemens where over 6,000 people are using process mining to improve operations in many areas attest the value of process mining for businesses.

Process mining is widely used to diagnose and address compliance and performance problems. There are three main types: (i) *process discovery*, (ii) *conformance checking*, and (iii) *model enhancement*. Starting from raw event data

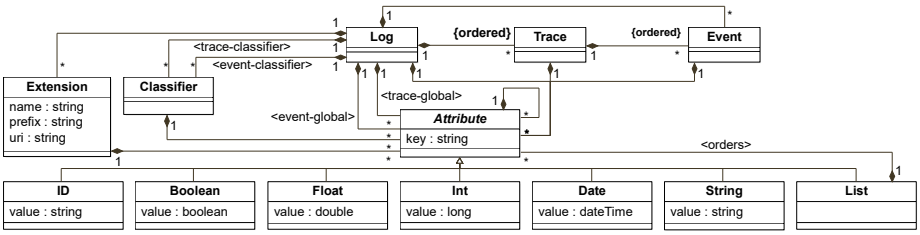


Fig. 1: XES meta-model (cf. [4])

process discovery creates process models that reflect reality and that include all or only the most frequent behavior. Conformance checking combines modeled and observed behavior. By replaying event data on a process model (modeled or automatically learned) one can diagnose and quantify deviations, e.g., to find root causes for non-compliance. Model enhancement is used to improve or extend a process model using event data. Process mining can also be used in an online setting and to predict compliance or performance problems before they occur. There are hundreds of process discovery, conformance checking, and model enhancement techniques that rely on model representations like Petri nets, directly-follows graphs, transition systems, process trees, BPMN and statecharts.

Event Data. Event data is represented as an event log which provides a view on a process from a particular angle. Each event in an event log refers to (i) a particular *process instance* (ii) an *activity*, and (iii) a *timestamp*. There may be various other attributes referring to costs, risks, resources, locations, etc. The XES standard [4] defines a format for storing such event logs. Due to its widespread use and tooling support, it is a suitable target format for blockchain logged data, enabling analysts to examine DApps using process mining.

Fig. 1 shows the XES meta-model as specified in [4]. The meta-model is oriented towards the general notion of *logs*, *traces*, and *events*. A log represents a process and consists of a sequence of traces which record information about individual process instances. Each trace contains a sequence of events referring to activities executed within the process instance. Logs, traces and events are described by attributes that have a *key*, a *type*, and a *value*. Attributes can also be nested, i.e., an attribute can contain other attributes. The XES standard does not prescribe terms for the keys and is thus free of domain semantics. However, to assign meanings to attribute keys, *extensions* can be included that define the meaning and the type associated with specific keys. Moreover, *global* values can be specified for any attribute at the event or trace level. Setting the global value v for the event (trace) attribute a means that if the value of attribute a is not specified for an event (trace), it implicitly takes value v. Finally, event (trace) *classifiers* comprise one or more attributes and each event (trace) with the same combination of values for these attributes belongs to the same class.

2.2 Ethereum: A Blockchain System

Blockchain. A blockchain is an append-only store of transactions, distributed across computational nodes and structured as a linked list of blocks, each con-

taining a set of transactions [14]. Blockchain was introduced as the technology behind Bitcoin [8]. Its concepts have been generalized to distributed ledger systems that verify and store any transactions without coins or tokens [11], without relying on any central trusted authority like traditional banking or payment systems. Instead, all participants in the network can reach agreement on the states of transactional data to achieve trust.

A smart contract is a user-defined program that is deployed and executed on a blockchain system [9,14], which can express triggers, conditions and business logic [13] to enable complex programmable transactions. Smart contracts can be deployed and invoked through transactions, and are executed across the blockchain network by all connected nodes. The signature of the transaction sender authorizes the data payload of a transaction to create or execute a smart contract. Trust in the correct execution of smart contracts extends directly from regular transactions, since (i) they are deployed as data in a transaction and are thus immutable; (ii) all their inputs are through transactions and the current state; (iii) their code is deterministic; and (iv) the results of transactions are captured in the state and receipt trees, which are part of the consensus.[5]

Ethereum. Ethereum is a specific blockchain system that allows users to deploy and execute smart contracts. We focus on this system as it is the longest-running blockchain with expressive smart contract capabilities. It provides an interface to store information in the transaction log. In general, smart contracts can only write information to the log, but not retrieve information from it. However, applications connected to an Ethereum node can query the log for information. This enables the implementation of an event-driven DApp architecture where smart contracts share information and applications react to published information.

Smart contracts for Ethereum are typically written in *Solidity*. This language provides write access to the transaction log via so called *events*. Events are specified through their signature including the event's name and a list of typed parameters (as of Solidity version 0.5.x only fixed-length types can be used), but no return type. Events also do not have an implementation. Instead, when an event is emitted, the event's signature and parameter values are automatically written to the transaction log as a structured *log entry*. There is also a low-level interface that allows developers to flexibly define the structure of log entries, but the burden for retrieving those entries is increased. In practice this interface is rarely used as revealed by our analysis of 21,205 different real-world smart contracts which we downloaded from Etherscan[6], covering a period of 10 months starting in June 2018. Within these smart contracts, we found more than 300,000 event emissions, but only 127 calls to the low-level interface. Hence, we decided to focus on extracting log entries whose structure follows that of the Solidity events and leave the full support for the low-level interface to future work.

The conceptual schema of the data from the transaction log is shown in Fig. 2. A *log entry* represents an emitted event. We use the term log entry instead of event to avoid confusion with XES events (see Section 2.1). A log entry is

[5] Summary adapted from [12].
[6] http://etherscan.io

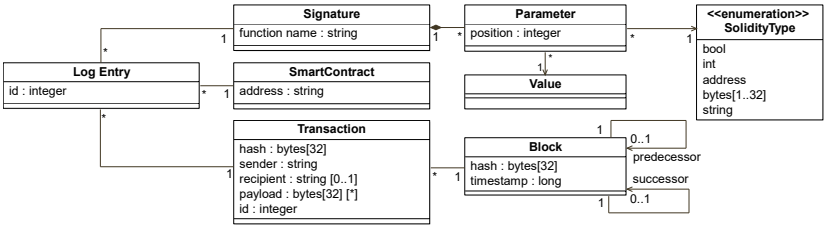


Fig. 2: EVM logging meta-model

associated with its *signature*, the *smart contract* that emitted the log entry and the *transaction* from which the log entry originated. The *id* of a log entry is only unique within the transaction and a smart contract is identified by its *address*. The signature contains the *function name* and a list of *parameters* defined by their *position* and *type*. Moreover, the transaction log contains the *value* for each parameter. For the transaction, we can retrieve its *hash*, the *payload*, the *sender* of the transaction, and the *recipient*, if available. Similar to the log entry, the *id* of a transaction is unique within the *block* that included the transaction. For such a block, we can load the *hash* and the *timestamp* as well as the *predecessor* and *successor* (which do not exist for the first and the latest block, respectively).

3 Approach

A high-level overview of our framework for extracting event data from Ethereum's transaction log is presented in Fig. 3. The *extractor* is a rule-based transformation algorithm that converts a set of log entries from the transaction log into files containing XES logs. The transformation rules can be flexibly adapted via the *manifest* which e.g., specifies which smart contracts to consider, how to filter events, which timestamps to include and where to find the concept (activity) name. The extractor can generate XES logs from any smart contract, given a fitting manifest, and given the required information has been logged in the first place. The *validator* supports the creation of a manifest by checking if it follows the rules of the manifest specification. Moreover, from a given manifest the *generator* automatically produces Solidity code for logging, which can then be integrated into smart contracts. This is particularly useful when using the feature mentioned in the introduction: compact encoding of data in a bitmap (details in Section 3.2). Following, we outline the different elements in detail.

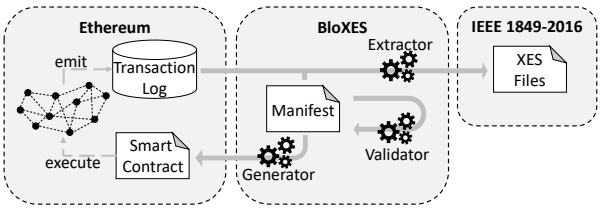


Fig. 3: High-level overview of the components

Algorithm 1: Extraction algorithm

```
   Input: manifest
   Output: xesFiles
 1 logs = {}
 2 foreach smartContractMapping in manifest do
 3 |   logEntries ← SelectLogEntries(scm)
 4 |   foreach logEntry in logEntries do
 5 |   |   foreach logEntryMapping in smartContractMapping do
 6 |   |   |   if logEntryMapping.signature = logEntry.signature then
 7 |   |   |   |   forall elementMapping in logEntryMapping do
 8 |   |   |   |   |   attributeMappings = elementMapping.mappings
 9 |   |   |   |   |   attributes ← Extract(entry, attributeMappings)
10 |   |   |   |   |   if isEventMapping(elementMapping) then
11 |   |   |   |   |   |   AddEventAttributes(attributes, logs)
12 |   |   |   |   |   else if isTraceMapping(elementMapping) then
13 |   |   |   |   |   |   AddTraceAttributes(attributes, logs)
14 xesFiles ← CreateXesFiles(smartContractMapping, xesFiles)
```

3.1 The Extractor and the Manifest

The extraction algorithm, Algorithm 1, takes the manifest as input and first initializes an empty set of logs (line 1). This set can be viewed as the root of a *log hierarchy* where each child is a log that, according to the XES standard (see Fig. 1), contains traces and events. Next, the algorithm iterates over the *smart contract mapping*s from the manifest (lines 2-13) and for each such mapping *selects the log entries* from the transaction log (line 3). Information is extracted from each log entry (lines 4-13) by applying the *log entry mappings* whose signature is equal to that of the entry (lines 5-6); the signature can thus be seen as the head of a mapping rule. Two signatures match if they have the same function name, the same number and types of parameters in the same order. For all matching log entry mappings, the algorithm maps the log entry to XES elements (lines 7-13). As one log entry might contain information for multiple traces or events, there can be multiple *element mappings* in a log entry mapping. For each element mapping (line 7-13) the algorithm extracts the attributes from the log entry according to the respective *attribute mappings* (lines 8-9). If the element mapping is a *trace mapping* (*event mapping*) the algorithm *adds the attributes* to a new or an existing trace (event) in the log hierarchy (lines 10-13). Lastly, the algorithm *creates the XES files* from the logs (line 14) and returns the files. Below, we describe the steps of the algorithm and explain how the steps can be configured using the manifest. Further, we discuss exception handling. We also present a consolidated manifest meta-model and details of the validator.

Selecting Log Entries. For a smart contract mapping, log entries are selected based on two criteria. First, the log entries must have been written by a transaction that is included in a block from a specified block range $[fromBlock, toBlock]$. If no block range is defined, the log entries of the 1000 most current blocks are retrieved. Second, the log entries must have been emitted by a smart contract

Table 1: Support for casting solidity into XES types ('+' = cast supported; '!' = cast supported, runtime exception possible; '-' = cast not supported)

Solidity Types	XES Types						
	int	float	date	string	boolean	id	list
int	!	!	!	+	-	-	-
string	-	-	-	+	+	!	-
address	-	-	-	+	-	-	-
byte	+	+	-	+	+	-	-
bytes	-	-	-	+	+	-	-
boolean	-	-	-	+	+	-	-
array	-	-	-	-	-	-	!

whose address is in a set of predefined *addresses* and there must be at least one address. Note that by specifying multiple addresses, a developer can apply the same transformation rules to different smart contracts. Finally, log entries are retrieved in the order in which they were written into the transaction log and the created XES elements follow this ordering.

Extracting Attributes. For every attribute mapping, the developer needs to specify the attribute's *name* and a *value builder*. A value builder is a function that (i) takes a log entry, (ii) returns a value, and (iii) is applied to each log entry the element mapping is executed for. A value builder is configured by specifying its *function name* and the return *type*, an XES type that becomes the type of the attribute. Moreover, the builder's *parameters*, a list of key-value pairs, must be set. *Static parameters* have a fixed value, whereas *value builder parameters* specify another value builder. We provide the following value builders:

- A *static value* is a fixed value that is assigned to the attribute. We support static values for all XES types except lists.
- *Parameter casts* access a log entry parameter or attributes of the associated block or transaction identified by their name and cast the value of the respective Solidity type into the XES type. Table 1 lists the supported type casts. Some type casts might result in a runtime exception, if the parameter value violates the range of allowed values, e.g., only string values that represent UUIDs can be cast into ID attributes [4].
- A *string concatenation* returns a formatted strings that joins the values of value builders that return a string, int or id.
- *Value dictionaries* map the value returned by another value builder to attribute values. Value maps can be specified for arbitrary XES type combinations and must include a default value.
- *Bit mappings* are used when data from the smart contracts is compressed before being written into the transaction log. This is typically achieved by assigning bit ranges of a single log entry parameter to certain variables. We support the decompression of such bit ranges. To this end, the value of a specified bit range of length l is at runtime mapped to an integer value p from the interval $[0, 2^l)$. Then, p is converted into a meaningful value based on a value array with 2^l elements from which we return the pth value.

Table 2: XES extension for identity attributes (name: "Identity", prefix: "ident", URI: "https://www.data61.csiro.au/ident.xesext")

Key	Definition	Type	Cardinality	Element
pid	Identifies a particular process	string	0-1	event, trace
piid	Identifies a particular process instance	string	0-1	event, trace
eid	Identifies a particular event	string	0-1	event

Appending Attributes to the Log Hierarchy. The processing of an element mapping for a log entry results in a set of attributes which belongs to the same XES element. We determine the identity of this element, add the attribute set to it, and integrate it into the log hierarchy in the following way. Each attribute set is considered to contain identity attributes. In particular, we defined three identity attributes within our custom *ident* XES extension (see Table 2). The *ident:pid* attribute identifies the process that an event or a trace belongs to and each distinct pid-value results in a separate log element. The *ident:piid* attribute determines the identity of a trace element within a certain log element, i.e., the global identity of a trace element is determined by its pid and piid-values. Finally, the *ident:eid* attribute establishes the identity of an event within a trace, i.e., the global identity of an event is given by the pid, piid, and eid-values. If the extracted attribute set does not include any of the identity attributes, we add those attributes and set their values to a default value.

We then use these attributes to add the entire attribute set to the log hierarchy. First, we select the log element with the respective pid-value. If such a log element does not exist, we create a new one with the respective pid-value, add it to the hierarchy, and select it. Next, we look for the trace element with the respective piid-value within the selected log element. Again, if we cannot find such an element, we create a new one, add it to the log element, and select it. If the element mapping is a trace mapping, we append the attribute set to the trace. Otherwise, we select or create an event element within the selected trace element based on the specified eid-value and append the attributes to this element. Following this strategy, we can integrate log entries from different smart contracts within the same log hierarchy. If no ident-attributes were specified, the algorithm generates one log element, containing one trace element and all log entries are mapped to individual event elements under this trace element.

Creating XES Files. The last step is to create XES files. Here, we create one XES file per log in the log hierarchy and set the file's name to the log's pid-value. To fully support the XES standard [4] and to relieve developers of having to edit the generated XES files, we allow users to specify XES extensions, global values and classifiers within the manifest. Each of these elements can be bound to a range of pids. If pids are specified for an element, the element is only included in the XES files corresponding to one of those pids. Otherwise, the element is added to all XES files. The inclusion of those features requires developers to adhere to the constraints that they impose on XES attributes. We discuss those constraints in more detail in the context of the validator (see below).

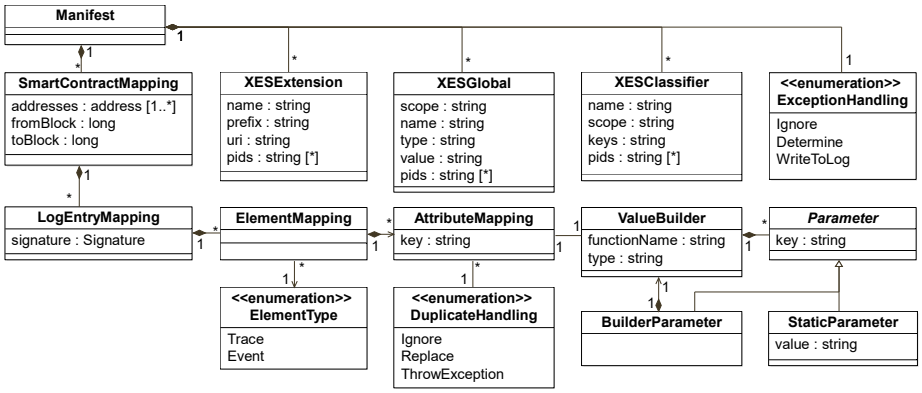


Fig. 4: Manifest meta-model

Runtime Exceptions. Some of the operations can result in runtime exceptions. The first exception type can occur when casting Solidity to XES types and was already discussed above. The second type refers to situations where the value of a certain XES attribute is set multiple times. While we restrict developers to set the value for an attribute only once within an element mapping, the problem can occur when adding attributes to existing elements. To circumvent this problem, the user can specify one *duplicate handling* strategy per attribute mapping. There are three different strategies: (i) *throw an exception*, (ii) *replace* the old value, and (iii) *ignore* the new value. Lastly, an *extension exception* is thrown in cases where an XES extension attribute is added to the log hierarchy, but the extension is not defined for the respective pid. Hence, extensions should only be restricted to certain pid-values, if the pid values are known in advance.

Developers can select one of three *exception handling* strategies for the entire manifest: (i) *determine* the algorithm (default option), (ii) *ignore* exceptions, and (iii) *write* exceptions to the XES logs. When the last option is selected, exceptions are converted into attributes. In case of a type cast exception and an extension exception, the algorithm creates a list with the key "error" and adds it to the attribute set. The exception information is added as a separate string attribute to the list's values section and the attribute's key is set to the key of the attribute that caused the exception. If there are multiple exceptions when processing one attribute mapping, the exceptions are grouped in the same error list. In case of a duplicate handling exception, we also create an error list which contains a string attribute per exception. In contrast to the other exceptions, we add the list to the attribute for which the value is set multiple times.

The Manifest. A consolidated view of the manifest which has been introduced throughout the introduction of the extraction algorithm is presented in Fig. 4.

The Validator. The validation of a manifest is a two-step process. First, we check whether the manifest's structure adheres to the meta-model from Fig. 4. To this end, we ensure that (i) all manifest elements are structured according to their meta-model type, (ii) the relationships between the model elements adhere to those specified in the meta-model, (iii) all required elements, relationships

and attributes are present, and (iv) that specified values have a valid format, e.g., that the specified Solidity types or XES types are known.

Second, we investigate the definition and usage of attributes as well as log entry parameters. To this end, we check that all combinations of attribute keys and types specified in attribute mappings, extensions and global values are consistent. We also verify that for each attribute that contains a prefix there is an extension that defines this attribute. Following the XES standard [4], we validate that there is a global value for all attribute keys used within classifier definitions. Further, we check the validity of the value builder specifications including their parameters. If a static value is specified, its string representation from the manifest must be castable into the specified type. For parameter casts, we check that they reference a valid parameter or attribute and that the type cast is supported (see Table 1). For all other value builders, we recursively verify that the return type of its child value builders is compatible with the expected input types, e.g., the concatenation value builder only allows the use of *xs:string*, *xs:id*, and *xs:int*.

While the validator verifies that the generated XES documents adheres to the XES standard, it cannot be guaranteed that all log entries can be processed, as discussed above in the context of runtime exceptions.

3.2 The Generator

We support developers of new smart contracts by generating logging functionality from the manifest. First, we use the signatures specified in the log entry mappings to generate Solidity events. This step is straightforward, as the signature specification in the manifest corresponds to the representation of events in Solidity's *contract application binary interface specification*, or *ABI*, the interface specification of smart contracts which is shared with users that want to interact with the respective smart contracts.

Additionally, we implemented the generation of logging functionality for value dictionaries or bit mapping builders specified in the manifest. As mentioned in Section 2.2, smart contract developers can log various information via Solidity's event parameters. On the public Ethereum blockchain, there is a cost involved for emitting smart contract events, which is proportional to the size of event parameters values being emitted (for example, a parameter of type *string* may cost more to log than an *int* parameter, if the string value is longer than 32 bytes). Therefore, developers can choose to log smart contract event parameters of a smaller type, such as *int*, then define a value dictionary in the manifest to map the Solidity event parameter values to a corresponding description of the value. In such cases, the generator produces (i) the Solidity event signature, (ii) an *enum* of the dictionary values which are mapped to the event parameter, and (iii) a logging function which accepts an *enum* value, then emits the Solidity event with the corresponding parameter value. An example of Solidity code generated from a manifest value dictionary is provided in Listing 1.1.

Another pattern that developers can use to further reduce the smart contract event log size is to encode multiple pieces of information into one log entry parameter with a small type such as *int*. To do this, they can specify a *bitmap*

```
contract XESLogger {
  event GitCommit(uint authorId, bytes32 sha);
  enum Author {FIRST, SECOND, THIRD, FOURTH};
  uint[] enumValsAuthor = [1000, 2000, 3000, 4000];
  function logCommit(Author author, bytes32 sha) internal {
    uint authorId = enumValsAuthor[uint8(author)];
    emit GitCommit(authorId, sha);
  }
}
```

Listing 1.1: Solidity generated from manifest with value dictionary builder

value builder in the manifest, which maps a subset of consecutive bits in a log entry parameter to a range of values. This is essentially a generalization of the bitmapping strategy adopted in earlier works of one author [3,6]. To effectively encode multiple pieces of information, multiple bitmap value builders can be defined on separate bit ranges of the same parameter.

As an example, assume we define a manifest for extracting events from *ShirtProduced* log entries, which are emitted each time a new shirt is produced in a *textile factory*. Three attributes related to a shirt, *Size*, *Fabric* and *Quality* are extracted from the same parameter, *encodedAttributes*. The first 3 bits (offset 0 to 2) are used to encode eight size classifications, bits 3-5 encode six shirt fabric types, and bits 6-7 represent four quality classes. Table 3 shows the bit mapping definitions for this example. From the respective value builder, the generator can produce a Solidity logging function which takes *Quality*, *Fabric* and *Size* input parameters as enums, and emits an event whose value correctly encodes the above information according to the mapping defined (see Listing 1.2).

```
contract XESLogger {
  enum Quality {LOW, NORMAL, HIGH, EXCELLENT};
  enum Fabric {COTTON, LINEN, WOOL, SILK, POLYESTER, BLEND};
  enum Size {XXS, XS, S, M, L, XL, XXL, XXL};
  event ShirtProduced(uint256 shirtId, uint256
      encodedAttributes, uint256 batchId);
  function logShirtProduced(uint256 shirtId, Quality q,
      Fabric fabric, Size size, uint256 batchId) internal {
    uint256 qMask = uint256(q) * (2**(6));
    uint256 fabricMask = uint256(fabric) * (2**(3));
    uint256 sizeMask = uint256(size) * (2**(0));
    uint256 encodedAttributes = qMask | fabricMask |
        sizeMask;
    emit ShirtProduced(shirtId, encodedAttributes, batchId);
  }
}
```

Listing 1.2: Solidity generated from manifest with bit mapping builders

Table 3: Encode info in Solidity event parameter with bit mapping builders

Quality - Bit range: 6..7		**Fabric - Bit range: 3..5**		**Size - Bit range: 0..2**	
Key	**Value**	**Key**	**Value**	**Key**	**Value**
00	low	000	cotton	000	2x small
01	normal	001	linen	001	extra small
10	high	010	wool	010	small
11	excellent	011	silk	011	medium
		100	polyester	100	large
		101	blend	101	extra large
				110	2x large
				111	3x large

4 Implementation, Evaluation, and Discussion

4.1 Implementation

We implemented the framework. The extractor and generator have been written in JavaScript for node.js, and manifests are specified in JSON format. The extractor takes several parameters as command line inputs:

- `rpc`: the URL of the blockchain node to query through the RPC interface
- `m`: manifest specification
- `output`: folder for storing extracted XES files

The complexity of the manifest file strongly depends on the complexity of the analyzed contracts. The manifest for our case study below has approximately 230 lines; the one for the code generation evaluation has about 65 lines.

4.2 Code Generation Evaluation

We tested the generator of logging code with a proof-of-concept demonstration of the *textile factory* example from Section 3.2. To this end, we first wrote the manifest specification, used it to generate logging code, then wrote a simple smart contract using this logging code and deployed it on a Geth Ethereum client using Ganache (a test mode that simulates an Ethereum blockchain in memory, but is otherwise identical in behavior). We then ran a test script against this Geth client, to 'produce' shirts according to a 'production schedule' which generated Solidity logs, and finally used the extractor to extract XES logs.

In this part of the evaluation, we tested the functional correctness of the generator and the extractor, which we found to be operating as per the design: the XES logs contained the same data in the same order as the production schedule. The tests of value and bit mapping functionality were successful as well: data was encoded as assumed, in 8 bits per shirt.

```
1 event Birth(address owner, uint256 kittyId, uint256
    matronId, uint256 sireId, uint256 genes);
2 event Transfer(address from, address to, uint256 tokenId);
3 event Pregnant(address owner, uint256 matronId, uint256
    sireId, uint256 cooldownEndBlock);
4 event AuctionCreated(uint256 tokenId, uint256 startingPrice,
     uint256 endingPrice, uint256 duration);
5 event AuctionSuccessful(uint256 tokenId, uint256 totalPrice,
     address winner);
6 event AuctionCancelled(uint256 tokenId);
```

Listing 1.3: Event definitions in the CryptoKittie contract

4.3 Case Study: CryptoKitties

CryptoKitties is "a game centered around breedable, collectible, and oh-so-adorable creatures we call CryptoKitties"[7]. While not Ethereum's most serious application, it is a well-known example of a DApp (that is primarily based on smart contracts), has been used heavily at times (likely due some of the kitties being sold for thousands of dollars), and has been in operation since December 2017. It also was developed without involvement of any of the paper's authors, making it a suitable candidate for demonstrating the framework's applicability.

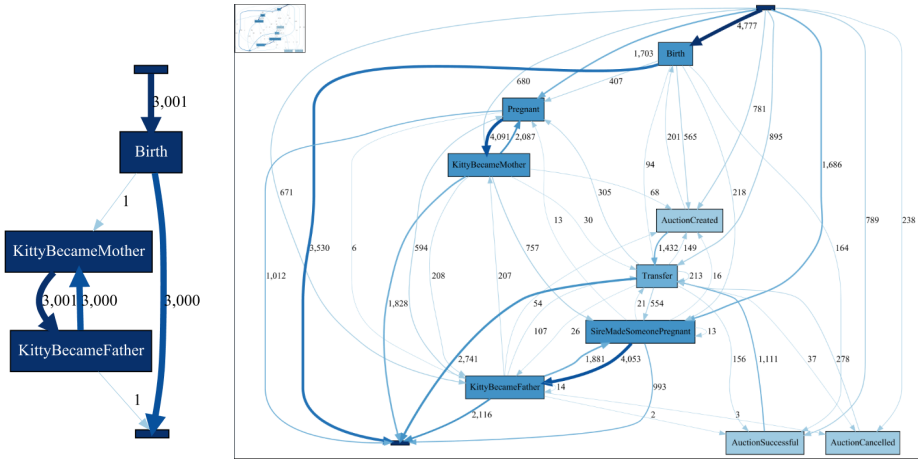
A CryptoKitty is the Ethereum version of a Tamagotchi. It is a digital asset owned by an Ethereum account and it can be traded. A CryptoKitty can breed new CryptoKitties. To this end, the owner can trigger the impregnation of a cat when having access to a second cat (either by owning it or by having the permission of the cat's owner). After a cat becomes pregnant, the owner must publish a "birth helper request" asking an independent account to trigger the birth in exchange of a certain amount of Ether. A CryptoKitty is represented by an identifier and its DNA, from which its features and appearance are derived. The source code of the CryptoKitties smart contracts is available on etherscan[8] and Listing 1.3 summarizes the event definitions from the source code.

We extracted two logs from these smart contracts with our framework and implementation[9]. The *genesis* log stems from the first 3000 blocks after creation of the smart contract at block 4605167. The *everyday* log is based on log entries from a random block range containing 13000 blocks, starting from block 6605100. In both cases, we only extracted information about the lifecycle process of the cats. Thus, we grouped all process instances in the same log and the ID of a cat is used as the process instance ID, i.e., the lifecycle of each cat is viewed as an independent process instance. Moreover, each log entry is mapped to individual events. Per auction and transfer-related log entry, we created one event in the trace of the cat represented by the tokenId. The birth and pregnant log entries

[7] https://www.cryptokitties.co/, accessed 30/5/2019

[8] https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d#code, Accessed: 17/05/2019

[9] Generated XES files and manifest available under https://doi.org/10.25919/5d242b0be3384



(a) Genesis (b) Everyday

Fig. 5: Directly-Follows Graphs (DFGs) generated using ProM from both logs

involve multiple cats (matronId, sireId, kittyId) and are hence mapped to events in each cat's lifecycle. Note that choosing the pid, piid and eid-values is critical and a general concern for process analytics, on blockchain data or otherwise. For example, in our case study, we take the viewpoint of an individual kitty, but this may not be suitable for analysing the complete population. To generate different views, our framework allows analysts to materialize their choice of identity attributes in the manifest, and for some applications multiple manifests with different choices might be required to obtain the desired views.

We mined the extracted event flows from both logs as depicted in Fig. 5. Fig. 5a shows that the developer started with two kitties initially, and bred them 3000 times for bootstrapping the game. The behavior during the everyday use (Fig. 5b) shows considerably more variation and includes all types of events.

While we could delve into a deep analysis of kitty behaviour, the purpose of the case study in this paper was to test if the proposed framework can be applied to existing smart contracts – which we deem to be the case. We successfully extracted event logs, stored them in the XES format, and loaded them for analysis in both ProM and Disco.

5 Conclusion and Discussion

In this paper, we addressed the problem of applying process mining to smart contracts and focused on extracting meaningful event data from blockchain systems, in particular from Ethereum transaction logs. Our proposed framework includes (i) the manifest specification for defining transformation rules that are automatically validated; (ii) the extractor that transforms log entries from a transaction log to XES logs; and (iii) the generator that produces high-level logging functionality for user-defined DApps. We showed that the generator produces logging functionality adhering to the log entries and data compression

rules from the manifest. Further, we successfully applied the extractor to logs created from generated code, as well as logs from a pre-existing, long running and heavily used DApp, demonstrating its broad applicability.

There are a few limitations that impact the applicability of our framework. First, we focused on Ethereum and disregarded other blockchain systems which might use different logging mechanisms, potentially requiring a generalization of our framework. Second, our framework only offers a certain set of functionality; e.g., there are currently five types of value builders; complex conditions for filtering attributes and elements are not supported; and we do not fully support the low-level logging interface. While this emphasizes the need for further generalization, we also plan to improve the extensibility of the framework and to release it as open source, enabling users to adapt it to their needs. Further, process mining can be used for many purposes; here we only used it for exploration of the data, to demonstrate the feasibility of our framework. In future work, we will apply the tool to more use cases and purposes.

References

1. Aalst, W.v.d.: Process Mining - Data Science in Action. Springer (2016)
2. Di Ciccio, C., Cecconi, A., Mendling, J., Felix, D., Haas, D., Lilek, D., Riel, F., Rumpl, A., Uhlig, P.: Blockchain-based traceability of inter-organisational business processes. In: BMSD (2018)
3. García-Bañuelos, L., Ponomarev, A., Dumas, M., Weber, I.: Optimized execution of business processes on blockchain. In: BPM (2017)
4. IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams (Nov 2016), IEEE Std 1849-2016
5. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: A business process execution engine on the Ethereum blockchain. Softw: Pract Exper pp. 1–32 (2019)
6. López-Pintado, O., Dumas, M., García-Bañuelos, L., Weber, I.: Dynamic role binding in blockchain-based collaborative business processes. In: CAISE (2019)
7. Mendling, J., Weber, I., Aalst, W.v.d., et al: Blockchains for business process management – challenges and opportunities. ACM TMIS **9**(1), 4:1–4:16 (2018)
8. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
9. Omohundro, S.: Cryptocurrencies, smart contracts, and artificial intelligence. AI Matters **1**(2), 19–21 (Dec 2014)
10. Prybila, C., Schulte, S., Hochreiner, C., Weber, I.: Runtime verification for business processes utilizing the Bitcoin blockchain. FGCS (Aug 2017)
11. Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: A technical survey on decentralized digital currencies. IEEE Commun. Surv. Tutor. **18**(3), 2084–2123 (2016)
12. Weber, I., Lu, Q., Tran, A.B., Deshmukh, A., Gorski, M., Strazds, M.: A platform architecture for multi-tenant blockchain-based systems. In: ICSA (2019)
13. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: Intl. Conf. Business Process Management (BPM) (2016)
14. Xu, X., Weber, I., Staples, M.: Architecture for Blockchain Applications. Springer (2019)