Ethereum Visualizer - Conceptual Approach

Dragana Saric, Paul Kneringer, Petar Petrov

Vienna, Austria

Abstract

Blockchain technology offers a wide range of possible applications with one being the opportunity to manage inter-organizational processes. Smart contracts ensure that process tasks are coordinated and respected by each involved party. Because process logic is often not modelled explicitly but only present in the system, we require better ways to extract the knowledge from smart contracts. Today different approaches exist that translate business process models into Solidity code - a language developed specifically for implementing smart contracts. However, scientific research has yet to address the possibility of translating Solidity code into Business Process Modeling Notation (BPMN) models. This paper proposes a conceptual approach to generate a process model from Solidity code based on an existing Java reverse engineering framework. Furthermore, we describe two possible concepts based on the output of existing compilers for smart contracts that could serve as a basis for developing similar approaches in the future.

Keywords: Ethereum, Blockchain, Reverse Engineering, Business Process Management

1. INTRODUCTION

Blockchain technology has the potential to change the way business processes are executed and managed. It opens the possibility for inter-organizational processes to be executed on blockchain systems. Therefor, hey are deployed on peer-to-peer networks without central authorities that do not require mutual trust between each pair of participants. Researchers have already addressed the possibility of generating smart contracts in Solidity code in order to execute processes on the blockchain [1], but there is still a gap in scientific research when it comes to visualization of Solidity code using BPMN representation. This approach could be beneficial for the analysis and inspection of the models deployed on the blockchain as Solidity code is not suitable for human interpretation. Abstraction of the source code and graphical representation of the models would enable a better understanding, easier identification of errors and provide a holistic view on the deployed processes.

In this paper we propose a method that generates BPMN models form Solidity source code that is executed on the Ethereum Virtual Machine. We developed one main conceptual approach to reverse engineer Solidity code to BPMN models that is based on a similar Java reverse engineering tool called MARBLE [2]. Additionally, we include two concepts that use the output of existing tools like Caterpillar [3] and Solgraph [4] as input and could be used to generate BPMN models in the future.

In this paragraph we will describe the structure of this paper. In section 2 we describe the current state of research on blockchain technology and its connection to Business Process Management (BPM). Section 3 displays existing approaches connected with our solution. Section 4 presents the developed concepts and section 5 is concerned with our final solution. A discussion about important findings and limitations of the approaches will be provided in section 6. Finally, section 7 includes a conclusion and possible directions for future research.

2. BACKGROUND AND RELATED WORK

In this section the main aspects of blockchain technology are described and the connection between BPM and blockchain systems is displayed.

2.1. Blockchain Technology and Ethereum

Blockchain technology offers a possibility for parties to operate in peer-to-peer networks without the need of intermediary or central authorities that manage the information generated by transactions. As the name suggests, blockchain is a structured chain of blocks that can contain information about transactions. Each new transaction has to be approved by the rest of the parties in the network. Once it is validated, transition is added as a new block to the already existing chain structure. New blocks are signed using cryptographic methods. Each block has its unique hash value which is derived from the transaction content and the hash value of the previous blocks. This enables the immutability concept of the blockchain because if a transaction is altered, the hash value will also change, hence the chain will break. Every change is recorded on the blockchain. If needed the full transaction history can be accessed and investigated which is an important concept for business process applications. Distributed ledgers are updated in real time as events and transactions occur. This is done in order to ensure that each network participant has up-to-date records of transactions which lowers the possibility of fraud.

Another important concept when it comes to executing processes on the blockchain are smart contracts as they enable enforcement of rules from business processes. Smart contracts run on top of the blockchain and can be defined as "self-enforcing agreements, i.e. contracts, implemented through a computer program whose execution enforces the terms of the contract" [5]. Ethereum is a decentralized ledger protocol that supports the Turing-complete programming language Solidity which is used for writing smart contracts.

The idea of executing business processes on the blockchain has already been introduced and the possible areas of application have already been detected [6]. Researchers have already addressed the possible opportunities and challenges that arise for Business Process Management and blockchain technology [1]. One of the main advantages for the execution of business processes on the blockchain is the unchangeable nature of blocks - parties involved can review a history of messages and pinpoint the errors as all of the state-changing messages are stored on the blockchain. It can ensure that only authorized parties can send messages that will be accepted. Furthermore, encryption ensures that only the public data is visible whereas the rest is readable only for authorized parties. Control flow and business logic of processes can to a large extent be compiled to the smart contracts which enforce the correct execution of the processes [7]. In the same paper, the authors propose an approach to facilitate the execution of collaborative processes with the use of blockchain technology.

2.2. Reverse Engineering

This section provides a description of reverse engineering as a discipline and a brief overview of relevant existing approaches.

"Reverse engineering is the way of analysing a subject system to create representations of the system at a higher level of abstraction" [8]. The starting point for reverse engineering processes is the lowest level of abstraction. Therefore, the main focus of the research is at the code level. Code is a good, and in some cases the only source of information about the system, especially when the system documentation is lacking. Reverse engineering includes two approaches: static and dynamic. Static methods describes software artefacts and their relationships by analysing source code. Dynamic methods include the run-time behavious and therefore require the analysis of dynamic components like loggers, triggers, etc. In our research we focus on static approaches as we are only looking at the Solidity code and not the behaviour of the models at execution time.

When it comes to reverse engineering Ethereum, most of the research focuses on reverse engineering already deployed contracts in order to inspect contract code [9] [10]. A method proposed by [9] generates a high-level pseudocode from a smart contract that is suitable for manual analysis. This research focuses on reverse engineering opaque contracts that have no publicly available source code, in order to enable inspection.

In our research we also analysed on Java reverse engineering methods, as Solidity is an object-oriented language and was influenced by C++ and JavaScript. Approaches like Rational Rose or NetBeans generate a UML class diagrams from object-oriented source code such as Java or C++ [11]. Others focus on reverse engineering of Java applications to Petri Net structures [12] or on extracting design patterns from Java source code [13].

Another approach, MARBLE, enables business process mining from legacy systems [2]. It is divided into four different levels of abstraction with three transformations in between them. We will go into more details about MARBLE and MARBLE-like transformations in section 5 where we present our solution based on this framework.

3. EXISTING RELEVANT APPROACHES

In this section we describe existing approaches that are connected to our research. We focus on existing compilers and visualization tools that are relevant for the two concepts we developed for the visualization of Ethereum.

3.1. Caterpillar

One of the concepts we discuss in this paper is based on the output of the Caterpillar tool [3]. Caterpillar is an open-source Business Process Management Systems compiler that supports the creation of BPMN instances and the execution of instances on the Ethereum blockchain. The workflow routing is performed by smart contracts. Caterpillar consists of four different modules called Compilation Tool, Execution Engine, Event Monitor and Work Item Manager. The Compilation Tool module is responsible for the mapping from BPMN to Solidity code. It takes the BPMN model in standard XML format as input and generates a smart contract in Solidity which contains workflow logic of the process. These smart contracts then contain variables to encode the state of process instance and scripts to update the state when an event occurs or a task is completed. The operations to create instances of a deployed process model, to deploy a process model and to control and record the execution of enabled tasks are provided by the Execution Engine. Whenever a new transaction, connected to the process model, is approved on the blockchain the Event Monitor generates notifications in order to keep other components updated. The Work Item Manager module is used to handle user tasks. We have been able to run the Caterpillar tool and we will discuss the conceptual solution of Solidity to BPMN model transformation based on this in section 4.1.

3.2. Tools for the Visualization of Source Code

In this subsection we describe a few tools which we found appropriate for our visualization task. Subsequently, we will present the generated solutions. We are going to mention advantages and downsides of every tool respectively. We can distinguish two main groups of visualization tools - static and dynamic.

3.2.1. Static Visualization

The most widely used tool to visualize source code is Graphviz [14]. It is an open source graph visualization software that accepts strings as input and outputs a specific image corresponding to it. The format of the strings that Graphviz accepts is strictly defined by the DOT language. Graphviz supports many different features like: choosing node colours, node layout, fonts, custom shapes etc. [15] Everything is defined in the aforementioned string, which is fed as an input to the program and generates an image as output. The opportunities that Graphviz provides for the purpose of visualization of Ethereum are great. It enables us to configure the output of Solidity code (after going through some preprocessing and mapping it to DOT) to match the criteria we need in order to have the right BPMN model representation at the end.

3.2.2. Tools for the Visualization of BPMN

Another approach that can be taken is not to translate the Solidity to some third language and then try to output the BPMN elements, but instead translate it to XML, which is the default notation that BPMN files accept. The .bpmn file extension is not so widely used but if we take a look inside such a file, we can see that it is basically in XML format. After we have the xml, there exist multiple tools online that can produce a diagram out of a .bpmn file. One of them is the really easy to use web-based tool bpmn.io [16]. It is released by Camunda as a lightweight web version for viewing and editing BPMN [17].

The only downside of the Graphviz and the bpmn.io tool is that they only provide the opportunity to output the graph statically. This means that even when small changes are made, the entire graph is re-rendered. This is critical in the context of huge graphs and therefore also for our case - large processes with the corresponding models[14]. In our context, it may be needed to track the stages of the process. This can be done with the help of tokens that are used like in Petri net token games. Therefore, we might consider some dynamic graph rendering tools as another option.

3.3. Dynamic Visualization

There exist many different tools that support dynamic graph visualization and exploration. Most of them, however, do not provide a way to change the node structure and instead are focused on graph analysis and not so much on the appearance. We need something that is able to output at least most, if not all, of the BPMN elements without having to change the source code of the specific tool. Therefore, we have examined multiple options available which provide the functionality needed for this task. The tool we would like to mention here is called Cytoscape.js [18]. It is an open-source graph library written in JS [18] and offers some functionality that may suit the needs of the requirements we have. It supplies the tools to create interactive data visualizations by using the widely used JSON format as an input among others. One of the benefits we will obtain by using this tool is the better information that we receive by clicking or hovering specific nodes. It is responsive and gives information about outgoing and incoming edges (in our case transactions), as well as their number. The biggest benefit, however, is the opportunity to dynamically highlight one of the nodes. By using this functionality one can represent the stage of the process which is being processed at the moment. This happens instantly and does not require re-rendering of the entire image as it is the case for static methods.

3.4. Existing Solidity Visualization Tools

Until now we have mentioned models which go from source code to graphical representation. In this section we will narrow it down to Solidity visualization. We will display approaches that already exist in scientific research and will subsequently show how they can be used for our case. We have found two interesting projects on GitHub in this context Solgraph [4] and Surya [19]. However, they do provide similar functionalities and we will only give an explanation about one of them in the following lines.

The tool we chose to describe and later use in our proposal is Solgraph. It generates a DOT graph that visualizes function control flow of a Solidity contract and highlights potential security vulnerabilities. It is a good starting point for our research, as it shows the structure of the code. However, it does not support the dynamic aspects of a BPMN model and many of the relationships between the elements are lost. Nevertheless, this approach provides useful information for us, because it captures the structure and inherent logic of smart contracts. Furthermore, Solgraph uses the DOT representation, which was described in a previous chapter.

After comparing all these different tools and approaches we believe that for the purpose of our research it is best to stick to static representation in order to understand the code. The dynamic representation will be left for further research. Therefore, in the next section we will examine the functionalities of Graphviz and bpmn.io and will define requirements for the development of the solution.

4. BPMN VISUALIZATION - SOLUTION PROPOSALS

4.1. Proposal #1 - Reverse Engineering: Caterpillar

Our first approach is strongly connected to the Caterpillar tool mentioned above. The idea is to take the output of the tool as an input for our solution, parse it and output XML files [20] which adhere to the BPMN standards. The solution will be easy to achieve, because it is an exact reverse engineering on the Caterpillar tool, which parses and translates all the XML tags into Solidity source code. In our case, the opposite is required. After we have the code we need to reverse it back to XML. XML is simple to understand and work with. It provides distinguishable elements and sections, because of its syntax which is comprised of opening and closing tags $\langle tag \rangle Content \langle / tag \rangle$ and uses parent/child structure between the tags.

We are going to use the example provided by Caterpillar, which is a .bpmn file. To receive the Solidity code that is generated by Caterpillar, we executed the program with a sample BPMN model. Afterwards, we examined it and checked how all elements are mapped between the two different languages. Next, we will present our findings by showing a simple example in order to prove that the mapping between all elements is consistent. The function for Exclusive Gateway is defined in Solidity like this:

```
function ExclusiveGateway_0ga7p17(uint localTokens) internal returns (uint) {
        if (applicantEligible)     return localTokens & uint(~2048) | 4096;
        else    return localTokens & uint(~2048) | 8192; }
```

Currently, by following the Caterpillar logic, the token is in the position of *ExclusiveGateway_0ga7p17* and at position 2048. We can observe the different elements and conditions that are involved here. For example, if applicant Eligible is True, it will remove the token from its current position and move it to position 4096, if it is False - to 8192. In BPMN visualization this check is not necessary. We only need the positions that are available, without needing to check the conditional statements. The reason for that is, the scope is set on static model and not on the different possible outcomes of a specific run of the process. This XOR Gateway maps to the BPMN in the following way:

```
<bpmn:sequenceFlow id="SequenceFlow_1pxsdl6" sourceRef="Task_15lfaes"
targetRef="ExclusiveGateway_0ga7p17" />
<bpmn:exclusiveGateway id="ExclusiveGateway_0ga7p17" default="SequenceFlow_04nl5rk">
        <bpmn:incoming>SequenceFlow_1pxsdl6</bpmn:incoming>
        <bpmn:outgoing>SequenceFlow_0jigqn5</bpmn:outgoing>
        <bpmn:outgoing>SequenceFlow_04nl5rk</bpmn:outgoing>
</bpmn:exclusiveGateway>
<bpmn:sequenceFlow id="SequenceFlow_0jigqn5" sourceRef="ExclusiveGateway_0ga7p17"
targetRef="EndEvent_19xiayo">
        <bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">applicantEligible
        </bpmn:conditionExpression>
</bpmn:sequenceFlow>
<bpmn:sequenceFlow id="SequenceFlow_04nl5rk" sourceRef="ExclusiveGateway_0ga7p17"
targetRef="EndEvent_1ubxmre" />
```

The first thing that must be mentioned is that the actual edges are represented between $\langle bpmn : sequeceFlow \rangle$ tags, which contain the attributes sourceRef and targetRef. These represent the actual process stages that the current element points to. Therefore, the first stage of the process mapping the elements will be to assign token locations from Solidity to element names. By following this example, we can substitute the position 2048 with its class name - $ExclusiveGateway\_0ga7p17$. Also the target positions must be assigned the respective values, using their class names. After having a list of locations in human readable format with their relationships, the mapping to BPMN will be simple. The main things that must be taken into consideration are the BPMN syntax and requirements. As it can be seen on the example, each element has an id, which should be taken from the aforementioned list of locations. The elements which are not sequence flows have slightly different structure: They do not require information about incoming and outgoing edges. After the list of element locations is extracted, these sequence flows can be taken from there. Another thing to note here is that all elements may contain child elements in the example: conditionExpression. This is, as mentioned above, the conditional statement that is defined by the Solidity function. Here this means that if the candidate was eligible, the edge should point to *EndEvent*_19*xiayo* (formerly position 4096). However, as noted above, it is not critical to include this in the BPMN, as we are only aiming to extract the basic structure of the code.

After describing this example we need to mention the only real downside of this approach: There is manual work involved in order to render the BPMN image. Each BPMN file consists of two major components: $\langle bpmn : process\rangle$ and $\langle bpmndi : BPMNDiagram\rangle$. We are only able to generate the components which are inside the $\langle bpmn : process\rangle$ tag and all of the drawing and rendering is done in the other part. We may have all the components but we are not able to draw them automatically. The problem with it is that we need to specify coordinates for every single element and this can be troublesome. Here is the XOR Gateway we used above as an example in the drawing part:

```
<bpmndi:BPMNShape id="ExclusiveGateway_0ga7p17_di" bpmnElement="ExclusiveGateway_0ga7p17"
isMarkerVisible="true">
    <dc:Bounds x="961" y="135" width="50" height="50" />
    <bpmndi:BPMNLabel>
      <dc:Bounds x="941" y="185" width="90" height="0" />
    </bpmndi:BPMNLabel>
</bpmndi:BPMNShape>
```

The drawing can be done manually by referencing the XML file and using the bpmn.io tool to place the different elements into the required positions. In order to skip this manual step we are proposing another approach in the next section. Following the previously described logic, this approach maps the components directly to the DOT language in order to enable automated rendering with the Graphviz tool.

4.2. Proposal #2 - Using DOT Language and Solgraph

In our second solution we will be using the tool Solgraph, which was described before. It provides the basic functionality that we need, thus we will use it as a foundation, build on it and adjust it to our needs. The most important feature that we will use Solgraph for, is the extraction of different elements from Solidity code. The tool was designed for a different purpose but this functionality can be really helpful, provided the necessary adjustments. We have again used the Caterpillar output as an input to the Solgraph and are again displaying a few simple examples, with the aim of explaining the mapping logic. The first thing to note for this approach is that we need to define different BPMN elements according to the DOT language. An example with Exclusive Gateway follows:

```
ExclusiveGateway_06dboho ---> ExclusiveGateway_06dboho [ shape=diamond,label="X"]
```

Appendix A includes a graphical representation of the mapping.

As we can see, Solgraph outputs the different elements without providing parameters for visualization options, but instead prints them out with the DOT default shape. We can easily change that by parsing the name of the object and specifying some pre-defined string for each different BPMN element. In this example this is done by parsing everything before the underscore and the unique ID and defining the shape and the label that should be shown to the viewer. An entire list of all the possible elements must be made and the respective shapes must be specified in DOT [15]. It is not as pretty as the Camundas bpmn.io tool, but it is sufficient to display the main elements of the model.

The next thing to note is that for each external element (non-automatic element, which requires external action), the Caterpillar tool is defining a func_start and func_callback functions in the Solidity code. We will not go into more detail but the idea is that both function are actually referring to the same element, which confuses the Solgraph tool, as it takes them as two different elements. Therefore, we must parse the elements which have _start and _callback by using their ids and combine them into one. An example with the Appraise Property Activity element follows:

```
Appraise_Property_start [color=gray] + Appraise_Property_callback  ----->
Appraise_Property [shape=Mrecord,label="\n Appraise \n Property \n\n"]
```

Appendix B shows the graphical representation of the mapping.

As we can see, Solgraph can be really helpful with the task to capture all the different elements of a BPMN process model after defining the mentioned parsing rules. But in order to reproduce the entire structure of the model, a more challenging part is the definition of the relationships between the elements.

Unfortunately, Solgraph uses its own approach to parse the relationships in contrast to the one provided by Caterpillar. It correctly identifies some of them, but fails to identify others. The solution here lies in the step function, which is created by Caterpillar and contains the process stage transitions logic. In order to correctly identify the relations, an entire module needs be built that parses the elements from the step function and creates the edges between the elements. This is, however, left for further research, as we move on to our third and most promising proposal in the next section.

5. SOLIDITY TO BPMN PARSER - A MARBLE EXTENSION

In this chapter we will describe an approach of Solidity to BPMN model transformation that is built on an already existing software solution called MARBLE (Modernization Approach for Recovering Business processes from LEgacy Systems) [2]. The business process archeology [21] tool was designed to extract business processes as graphical models from Java source code. We introduce a solution that enables the MARBLE tool to take Solidity source code as input and generate BPMN models by means of static code analysis. This paper only provides the principles and logic of an addition to the MARBLE software. In order to use the parsing algorithm, a technical implementation and integration into the MARBLE tool is necessary.

The MARBLE tool was created by Prez-Castillo et. al. [21] and is based on Architecture-Driven Modernization (ADM), an initiative of the Object Management Group (OMG) defining standards for the modernization of legacy systems. The goal of the ADM initiative was to develop standard metamodels that represent the information that is involved in a modernization process. In the context of ADM, modernization is understood as the preservation of business value that is inherent in existing software systems [22]. MARBLE utilizes the Knowledge Discovery Metamodel (KDM), which is one of the core metamodels of the ADM standard, in order to store information in a language independent format. KDM defines several packages that can be used to capture entire legacy systems on four different layers of abstraction. Although the scope of ADM and KDM exceed the analysis and transformation of source code, some of its principles can be used for model transformation as we require it for our prototype. For the Java to BPMN model transformation, MARBLE only uses the Code and the Action package within the Program Element layer of the KDM specification [21]. The Code package provides information about the named items of the source code and structural relationships while the Action package refers to data-flow and behavioural components. Program Element layer is the second layer of abstraction of the KDM and aims at the provision of a language-independent intermediate representation that is based on common constructs of programming languages [21]. MARBLE only uses these parts of the KDM specification because source code is the sole artefact used for the transformations, hence no more packages are necessary.

The business process archeology tool uses four different kind of models on four different layers of abstraction. It defines the models that are used on each layer and the transitions between them. The layers are enumerated from L0 to L3. Layer 0 (L0) represents the software artefacts in the real world (e.g. source code). Layer 1 (L1) models are Platform Specific Models (PSM) because they represent the software with regards to their specific technologies or platforms [2]. On layer 2 (L2) the previously mentioned KDM is used to provide a common Platform Independent Model (PIM) framework. All L1 models are integrated in the technological-independent KDM. Layer 3 (L3) is the highest level of abstraction and displays the knowledge extracted from the KDM as a business process model. The notation used for L3 business process models in MARBLE is the Business Process Model and Notation (BPMN). The business processes are retrieved by applying pattern matching onto the structure of the L2 KDM representation.

The transformation from L0 to L1 models is executed by extracting PSM models from software artefacts according to metamodels. This can be done by means of standard reverse engineering methods like static and dynamic analysis, subsystem decomposition, etc. [2] For the Java to BPMN transformation, MARBLE transforms the source code into Abstract Syntax Trees (AST) conforming to the Java metamodel. The L1 to L2 model transformation is performed by using the OMG Query/Views/Transformations (QVT) standard [21]. The BPMN models on L3 are retrieved by applying business process pattern [23] matching onto the structure of the L2 KDM representation.

In the following chapter we will describe our approach on L0 to L1 transformation for Solidity source code. We use the Solidity compiler solc in order to create an AST structure from the source code and transform it to conform to the AST structure that was created from Java source code. Similar constructs like classes in Java and contracts in Solidity result in the same AST elements. Table 1 in Appendix C provides a coherent view on the mapping of constructs that was done between the different models. This is done in order to ensure compatibility of our parser to the MARBLE tool and to enable further processing of the generated inherent process knowledge. An implementation of this proposed method could be added to the business archeology tool with little effort, because the MARBLE tool software does not have to be altered.

5.1. L0 to L1 Transfromation

As Prez-Castillo et al. [24] state in their work, the first model transformation from layer L0 to L1 takes source code written in an object-oriented language as input. A syntactic parser transforms it into abstract syntax tree models conforming to the Java metamodel. The authors mention that this parser was developed for Java source code but can be implemented for other object-oriented languages. In order to build a parser that enables the transformation of Solidity source code to a representation that can be used for the following analysis with the MARBLE tool we need to understand the structure of the intermediate abstract syntax tree representation. In this chapter, we will explain how the parser present in the MARBLE business process archeology tool translates the constructs of Java source code to elements of the abstract syntax tree. Subsequently, we will introduce an algorithm for the transformation of Solidity source code to a similar structure.

5.2. Analysing the Structure of MARBLE-conform Abstract Syntax Trees

The parser works in a top-down manner, starting from the Java source code files to the classes, methods, statements and finally to a detailed fine-grade decomposition of these statements. The abstract syntax tree as intermediate code representation is concerned with the structure of the source code and removes irrelevant parts for the subsequent analysis. In the following paragraph we will describe the major elements of the AST data structure and describe the way the layer 0 to layer 1 transformation is realized in the MARBLE tool. All required information was gathered by feeding different kinds of Solidity smart contracts with many different constructs into the MARBEL tool and analyse the resulting outputs.

For every source code file that is analysed, the parser creates a CompilationUnit element in the AST. Then, the parser adds a PackageDeclaration element to the CompilationUnit, storing the package that the file is located in. Additionally, an ImportDeclaration element is added for every package that is imported in the file. As a next step, the parser creates a TypeDeclaration element and inserts a ClassOrInterfaceDeclaration element for every class in the file. These elements contain ClassOrInterfaceBody definitions that represent all content of the Java classes. A ClassOrInterfaceBodyDeclaration element with a fitting subelement for every method and property that is present in the class are added. Methods result in a MethodDeclaration subelement and properties in a FieldDeclaration subelement. A MethodDeclaration element contains a ResultType (return-type), a MethodDeclarator that stores the parameter values and a Block element that contains all statements of the method in the form of BlockStatement elements. According to the source code, BlockStatements contain either a LocalVariableDeclaration or a Statement element. Statement elements are specialized into different kind of structural elements e.g. ForStatement, ReturnStatement, IfStatement, Expression, and many more. The respective Statements are further broken down to a detailed level until the statement is atomic and hence completely separated in its basic elements (more details on the different statements in the following chapter). The resulting tree structure is then used by the MARBLE process archeology tool for the subsequent transformations.

5.3. From Solidity to MARBLE-conform Abstract Syntax Trees

In order to enable Solidity source code to BPMN model transformation, we require a parsing algorithm that creates a MARBLE-conform abstract syntax tree. In this section we will describe this algorithm in textual form and therefore enable future researchers to implement a functioning parser that could be integrated into the MARBLE tool in further consequence. We use the Solidity compiler solc (https://github.com/ethereum/solidity/releases) in order to create an abstract syntax tree from Solidity source code (command: solc –ast-json). The output is provided in JavaScript Object Notation (JSON) format. This representation can not be processed by the MARBLE tool and needs to be transformed into a similar AST data format as the Java code in the chapter before. In order to enable an easy integration of our Solidity parser into the MARBLE tool, it is necessary for us to use the same labels and names for elements that were used for Java code transformations. We do so, because the constructs in Solidity and Java do have commonalities, although they do not map one to one (e.g. class and contract). We added a table (Table 1) describing the mapping that was done (e.g. ClassOrInterface element maps to a single Contract element although they are not exactly the same).

- **Files and Contracts:** For every Solidity file (JSON attribute nodeType with value SourceUnit) that is analysed by our parsing algorithm, a CompilationUnit element in the resulting AST structure is created. Every import definition in Solidity results in an ImportDeclaration element that is created in order to store the name of the imported resource. These ImportDeclaration elements are then added to their respective CompilationUnit element. As a next step, the parser adds a TypeDeclaration element to the CompilationUnit element. After that, the parser iterates over all child-nodes of the JSON base object and searches for the attribute nodeType with the value ContractDefinition. For every element found the parser creates a ClassOrInterfaceDeclaration element in the resulting AST.

- **Structs:** Because Solidity enables the usage of structs and Java does not, we need to represent a struct in an understandable way for the MARBLE tool, while remaining the basic functionality of the element. Consequently, we treat structs that are present in Solidity code like classes with public properties (struct-like class). Therefore, every JSON node with the nodeType element set to StructDefinition results in a ClassOrInterfaceDeclaration element in the MARBLE AST. These elements have ClassOrInterfaceBody child-nodes that represent the body definitions of the classes or structs. They are built up by elements called ClassOrInterfaceBodyDeclaration containing a fitting element for every possible source code construct. For every JSON element with the nodeType set to StructDefinition, the parser iterates over its list of members and adds the ones with the nodeType set to VariableDeclaration as ClassOrInterfaceBodyDeclaration/FieldDeclaration elements to its ClassOrInterfaceBody. These FieldDeclaration elements represent the data type and value of the variables that were stored in the struct. Because structs were only transformed to separate classes yet, the parser needs to add the relationships between the current class and the newly struct-like class to the AST. Therefore, a ClassOrInterfaceBodyDeclaration element is added, representing a property. This element has the subelements FieldDeclaration/Type which contain a ReferenceType element with a ClassOrInterfaceType reference pointing to the respective struct-like class.

- **Properties:** As a next step, all properties of the Solidity source code are added to the AST. Therefore, the parser iterates over all children of JSON objects with the nodeType property set to ContractDefinition. For all childnodes with nodeType set to VariableDeclaration, a ClassOrInterfaceBodyDeclaration/FieldDeclaration is inserted in the respective ClassOrInterfaceBody element.

- **Functions and Constructor:** In order to add all functions to the AST, the parser searches for nodes with the nodeType property set to FunctionDefinition. For every object found, the parser creates a ClassOrInterfaceBodyDeclaration and inserts one of the following elements into it. If the isConstructor property of the object is set to true, a ConstructorDeclaration element is inserted. Otherwise, a MethodDeclaration element is inserted. Both elements store information about the signature in an element called FormalParameter and information about all statements in a Block element. Additionally, the MethodDeclaration elements also store the datatype that is returned in an element called ResultType.

- **BlockStatements:** The BlockStatement element is used to store the structure and content of variable declaration, function calls, etc. BlockStatement elements are grouped in Block elements. They can contain different kind of elements, most importantly: Statements and LocalVariableDeclaration. Statement elements represent function calls and their childnodes further decompose a function call in a Prefix and a Suffix component. The Prefix describes the name of the function that is called. The Suffix describes all parameters that are passed. The LocalVariableDeclaration element describes the definition of a local variable.

5.4. Summary of the Approach

By applying and implementing the proposed model transformations, it will be possible to generate BPMN models from Solidity smart contracts using the MARBLE tool. If the developers stick to the defined transformations, no major changes in the archeology tool will be required. Keep in mind that the proposed method only aims to provide a basis for future implementations. We have only described the main constructs and mapping between them. A comparison between Solidity and Java constructs needs to be performed in order to validate the interchangeability of constructs which is a main premise of this work.

6. DISCUSSION

An important outcome of our Solidity to BPMN analysis was a thorough analysis and summary of current efforts taken in the field. This work represents the foundation for future research concerned with Solidity to BPMN transformation. The main limitation of our research is the focus on static artefacts as the execution of business processes makes them highly dynamic and dependent on the different process execution.

We propose different methods that enable the representation of Solidity smart contracts as BPMN models. Our main method, the MARBLE tool extension, is described and can be implemented and tested in future research. The major limitations of the MARBLE extension include the focus on theory with no technical implementation, the assumption made on the interchangeability between Solidity and Java constructs and the need for further testing once the constructs are implemented.

The future development of the approaches and concepts presented in this paper can have a significant impact on this research field. With further analysis and practical development, the solution presented in this paper can result in working tools that enable BPMN representation of the Solidity code which goes beyond any tool capabilities at the moment.

7. CONCLUSION

The aim of our research was to develop an approach for visualizing Solidity code. This approach is very beneficial as it enables graphical representation of the code which would enable easier analysis of the processes, identification of errors and holistic view on processes. We focused on generating BPMN models from Solidity code using the static methods and we developed one main approach and two valuable concepts suitable for further development. The first concept includes reverse engineering Caterpillar output but the main downside is the need for manual work in between. The second concept uses the DOT graph representation as a main technology and is based on the existing visualization tool Solgraph. And as a final solution, we presented a conceptual approach to reverse engineer Solidity code based on the similar MARBLE tool. This approach could be used in future research as a guideline for developing a working prototype that enables the visualization of Solidity code. Further research efforts should focus on extracting dynamic properties and combining them with the static reverse engineering which would enable to monitor behaviour of processes at the execution time as well.

[1] J. Mendling, I. Weber, W. V. D. Aalst, J. V. Brocke, C. Cabanillas, F. Daniel, S. Debois, C. D. Ciccio, M. Dumas, S. Dustdar, et al., Blockchains for business process management-challenges and opportunities, ACM Transactions on Management Information Systems (TMIS) 9 (2018) 4.

[2] R. Perez-Castillo, Marble: Modernization approach for recovering business processes from legacy information systems, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, pp. 671–676.

[3] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, A. Ponomarev, Caterpillar: A business process execution engine on the ethereum blockchain, arXiv preprint arXiv:1808.03517 (2018).

[4] R. Revere, Solgraph, https://github.com/raineorshine/solgraph, 2015. (Accessed: 22.12.2018).

[5] R. Tonelli, G. Destefanis, M. Marchesi, M. Ortu, Smart contracts software metrics: a first study, arXiv preprint arXiv:1802.01517 (2018).

[6] M. Nofer, P. Gomber, O. Hinz, D. Schiereck, Blockchain, Business & Information Systems Engineering 59 (2017) 183–187.

[7] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, J. Mendling, Untrusted business process monitoring and execution using blockchain, in: International Conference on Business Process Management, Springer, pp. 329–347.

[8] G. CanforaHarman, M. Di Penta, New frontiers of reverse engineering, in: 2007 Future of Software Engineering, IEEE Computer Society, pp. 326–341.

[9] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, M. Bailey, Erays: Reverse engineering ethereum's opaque smart contracts, in: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 1371–1385.

[10] S. Bragagnolo, H. Rocha, M. Denker, S. Ducasse, Smartinspect: solidity smart contract inspector, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, pp. 9–18.

[11] A. Gade, S. Patil, S. Patil, D. Pore, Reverse engineering of object oriented system, International Journal of Scientific and Research Publications 3 (2013) 1–7.

[12] J. Fuhs, J. Cannady, An automated approach in reverse engineering java applications using petri nets, in: SoutheastCon, 2004. Proceedings. IEEE, IEEE, pp. 90–96.

[13] N. Shi, R. A. Olsson, Reverse engineering of design patterns from java source code, in: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on, IEEE, pp. 123–134.

[14] E. K. S. C. N. John Ellson, Emden R. Gansner, G. Woodhull, Graphviz and dynagraph static and dynamic graph drawing tools (2004).

[15] E. R. Gansner, E. Koutsofios, S. North, Drawing graphs with dot, https://www.graphviz.org/pdf/dotguide.pdf, 2015.

[16] Camunda, bpmn.io online tool, https://bpmn.io/, 2018. (Accessed: 10.01.2018).

[17] Camunda, Desktop tool, https://camunda.com/bpmn/tool/, 2018. (Accessed: 10.01.2018).

[18] C. Consortium, Cytoscape, http://js.cytoscape.org/, 2015.

[19] F. Bond, Surya, https://github.com/ConsenSys/surya, 2018. (Accessed: 28.12.2018).

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible markup language (xml), World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 16 (1998) 16.

[21] R. Pérez-Castillo, I. G.-R. de Guzmán, M. Piattini, Business process archeology using marble, Information and Software Technology 53 (2011) 1023–1044.

[22] J. Canovas, J. Molina, An architecture-driven modernization tool for calculating metrics, IEEE software 27 (2010) 37–43.

[23] R. Pérez-Castillo, I. G. R. de Guzmán, M. Piattini, On the use of patterns to recover business processes, in: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, pp. 165–166.

[24] R. Pérez-Castillo, J. A. Cruz-Lemus, I. G.-R. de Guzmán, M. Piattini, A family of case studies on business process mining using marble, Journal of Systems and Software 85 (2012) 1370–1385.

Appendix A. Exclusive Gateway Mapping

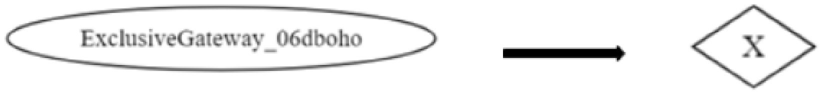

Figure A.1: Example of XOR mapping

Appendix B. Appraise Property Activity Mapping

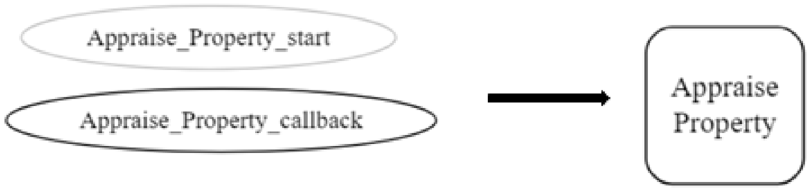

Figure B.2: Example of Appraise Property Activity element mapping

Appendix C. Comparative table

Java	**Solidity**	**SOLC AST**	**MARBLE AST**
.java files	.sol files	SourceUnit	CompilationUnit
Import	Import	N/A	ImportDeclaration
Class	Contract	ContractDefinition	ClassOrInterfaceDeclaration
N/A	Struct	StructDefinition	ClassOrInterfaceDeclaration
N/A	Struct variable	VariableDeclaration	FieldDeclaration
Method	Function	FunctionDefinition	MethodDeclaration
Constructor	Constructor	IsConstructor	ConstructorDeclaration
Property	Property	VariableDeclaration	FieldDeclaration

Table C.1: Comparison of structural elements in different models