From the Graphical Representation to the Smart Contract Language: A Use Case in the Construction Industry

Xuling Ye and Markus König

Department of Civil and Environmental Engineering, Ruhr-University Bochum, Germany
E-mail: xuling.ye@rub.de, koenig@inf.bi.rub.de

Abstract –

With the growing popularity of blockchain technology in the construction industry, smart contracts are becoming increasingly common. A smart contract is a self-executing contract, which contains if-then rules that automatically execute certain processes when certain conditions are met. Such smart contracts serve as programmable blockchain applications. Using blockchain-enabled smart contracts, many processes like construction contracting and payments can be automated. Since research on blockchain-enabled smart contracts in the construction industry is still theoretical, researchers usually assume that users (e.g. clients, contractors) can directly program the conditions in a smart contract. However, it is difficult for stakeholders to program smart contracts themselves due to a lack of knowledge. The smart contracts developed by programmers might not fully represent stakeholders' ideas.

Therefore, this paper proposes an approach that illustrates how graphical workflow notations (e.g. BPMN, YAWL) can be translated into smart contract programming languages (e.g. Solidity, Vyper). In this way, non-programmers can also design and generate their own smart contracts. To test the feasibility of this approach, an illustrative example is presented for generating smart contracts displaying automated the reporting, checking and payment process of construction works. In particular, the smart contracts in this example are translated from YAWL graphical representations into Solidity smart contract languages. Finally, improvements and further developments of the approach are discussed in several aspects.

Keywords –

process modeling; smart contracts; blockchain; construction industry

1 Introduction

Smart contracts can be used in the construction industry for process automation, for processes like construction, contracting, or payment. However, the applications and uses of smart contracts are still at a conceptual level. Moreover, smart contract languages are programming languages that are difficult to read, understand, or write for the stakeholders of construction projects, as they are usually not programmers. In the current situation of using smart contracts for system development or construction projects in the construction industry, two group of actors need to be involved: construction stakeholders (e.g. clients and contractors) and software programmers. The stakeholders need to decide system functionality, capture requirements, implement smart contracts and analyze the execution of the smart contracts for the system development. On the other hand, the programmers need to interpret requirements from stakeholders, develop and deploy smart contracts, and evaluate the execution of smart contracts. Implementation of proper smart contracts is a challenge for the non-programmers, which is related to a high effort in terms of time and resources. Otherwise, the programmers' interpretation could be wrong, leading to stakeholders having to deal with the consequences of these codes, which stakeholders may not be able to understand. The construction industry is not the only area with these problems. In the finance area, 60% decentralized finance (DeFi) users cannot read or understand the source code of smart contracts [12].

It is worthy and necessary to develop an approach that allows for the automatic translation of human-readable texts or graphics into smart contracts. Compared to pure text-based languages, graphical representations are more intuitive which can simplify complex contexts, enabling non-programmers to read, understand, verify and design them. This study proposes an approach to translate from graphical representations (e.g. BPMN or YAWL) to smart contract languages (e.g. Solidity) with a detailed explanation of translation and checking steps based on the YAWL graphical representation and the Solidity smart contract language. As a graphical workflow in XML-based format for business processes, YAWL (Yet Another Workflow Language) is the language that not only has proper formal semantics to check properties for

academic purposes, but also supports the control-flow patterns for business processes in practice [8]. Solidity is currently the most well-known and most common-used smart contract language, a language designed for the Ethereum blockchain according to the characteristics of smart contracts. The approach is tested via a payment case in the delivery and acceptance process of certain construction works by translating from the YAWL representation to Solidity.

2 Related work

2.1 Graphical workflow

A workflow can be defined as "a collection of tasks organized to accomplish some business process (e.g., processing purchase orders over the phone, provisioning telephone service, processing insurance claims)" [6]. To express the information, knowledge or systems of a business process in a structure by a consistent set of rules, different process modeling approaches have been proposed, including the Business Process Model and Notation (BPMN), the Web Services Business Process Execution Language (BPEL), the Event-driven Process Chain (EPC), the Yet Another Workflow Language (YAWL) and Petri nets [8].

As Lohmann et al. [8] pointed out, academics prefer languages such as Petri nets, which have proper formal semantics to check properties on corresponding models. However, practitioners prefer languages such as BPEL, EPC and BPMN, which usually lack proper formal semantics. As an exception compared to the above languages, YAWL originated in academia but has actually been used in practice [1]. YAWL supports the most common control-flow patterns found in current workflow practices, allowing most workflow languages to map to YAWL without losing control flow details, even languages with high-level structures (such as cancellation regions or OR-joins) [8,17].

Graphical workflows are also used in the construction industry to describe data flows and processes, with Integration Definition for Function Modeling (IDEF), extended EPC, or BPMN being commonly used [3].

2.2 Smart contract

A smart contract, which was first proposed by Szabo in 1994 [14], is a self-executing digital agreement. After Buterin et al. [4] implemented the smart contract concept as a blockchain application to write, verify and enforce transaction conditions that allows non-currency data stored on the Ethereum blockchain platform in 2014, smart contracts become popular in research and practice. Up to 2020, there are already more than 4,119 papers related to smart contract languages, and 24 different blockchain platforms are being developed with 101 smart contract languages (e.g. Solidity, Vyper) in total [16].

More and more researchers in the construction industry study the possibilities of using smart contracts. For example, Penzes [11] pointed out that smart contracts in construction can be applied in three directions: 1) payment and project management; 2) procurement and supply chain management; and 3) BIM and smart asset management. Badi et al. [2] investigated the smart contract adoption in the construction industry by a survey among UK construction practitioners, and conclude that smart contracts can be useful in payment, construction contract and in general for organization.

2.3 Research on smart contract generation

The existing research on smart contract generation mainly focuses on using either text-based languages or graphical-based languages to translate smart contracts. The text-based languages can be natural languages [10, 15] and other specification languages [7], while the graphical-based languages can be BPMN [9, 12], Petri nets [19], Unified Modeling Language (UML) [5] or Finite State Machine (FSM) [13].

Tateishi et al. [15] proposed an approach that translates from controlled natural languages to a domain-specific language for smart contract (DSL4SC), then to state chart and finally to executable smart contracts with four defined actions: order, ship, arrive and pay by using a predefined template. Monteiro et al. [10] introduced a prototype method by translating natural language processing (i.e. plain text (TXT) or structured text with markups (XML or HTML)) to smart contract code. A specification language called SPESC designed by He et al. [7] defined contract, party, term and type to assist smart contract generation. However, a direct translation from text-based languages to smart contract languages is error-prone, which is not visual and the logic error could be difficult to discover.

Many other researchers focus on translating graphical workflows to smart contract languages [5, 9, 12, 13, 19]. For example, López-Pintado et al. [9] introduced and implemented a blockchain-based BPMN execution engine called Caterpillar to generate smart contracts by a BPMN-to-Solidity complier. Since BPMN does not have a logic checking mechanism like Petri nets, the generated smart contracts from BPMN could be error-prone without an additional verification step. Skotnica et al. [12] proposed a model-driven approach to generate smart contracts based on a visual domain-specific language called DasContract. It is a complex approach where the users not only need to define data structures in a programming way and draw BPMN graphics, but also define a user form, which could be difficult to use by non-programmers. Zupan et al. [19] presented a method and a prototype tool based on Petri nets for smart contract

generation. Using Petri nets to model smart contracts is a suitable method to minimize logical errors. However, Petri nets cannot specify roles or tasks so well like BPMN or YAWL for business process. Lohmann et al. [5] proposed to generate smart contracts through UML class diagram. Nevertheless, the design of UML class diagrams requires some basic programming background. Another approach designed by Suvorov and Ulyantsev [13] is using FSM synthesis for automatic smart contract generation. Using FSM is good for showing the states and actions of the smart contracts, but lack role specifications.

3 Methodology

3.1 Overview

It is assumed that a workflow is created in a BPMN or YAWL software tool and exported as XML format. The XML files serve as an import for the proposed smart contract language generator system. The overview of the approach is shown in Figure 1.

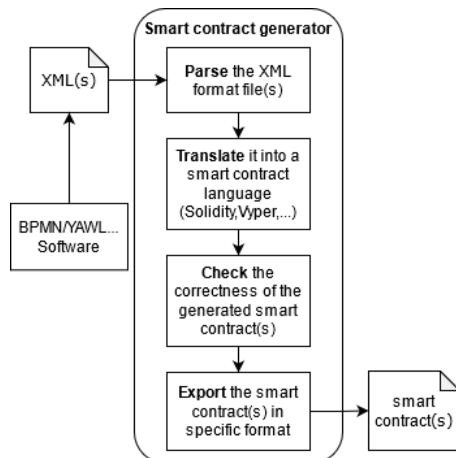

Figure 1. Overview of methodology

An XML-format file is generated via a BPMN or YAWL software tool, and imported into the proposed smart contract language generator system. Such a system needs functionalities for parsing, translating and checking a workflow. Finally, the generated smart contract is exported into the selected format for deployment. Even through the XML structure of YAWL and BPMN are different, the principle of the generation step is the same. YAWL is a graphical representation which not only allows logic checking, but is also easy to understand by non-programmers. Meanwhile, YAWL allows the mapping from most workflow languages without losing the details of the control flow. Therefore, YAWL is used as a graphical representation for the detailed explanation of translation and checking steps in the following two subsections.

3.2 Translation step

After parsing the data from an XML format file, the obtained information is translated into a certain smart contract language. To understand the obtained information, the structure of the XML format file should be explored. Therefore, the XML structure and the format explanation of YAWL and its roles are shown as Figure 2.

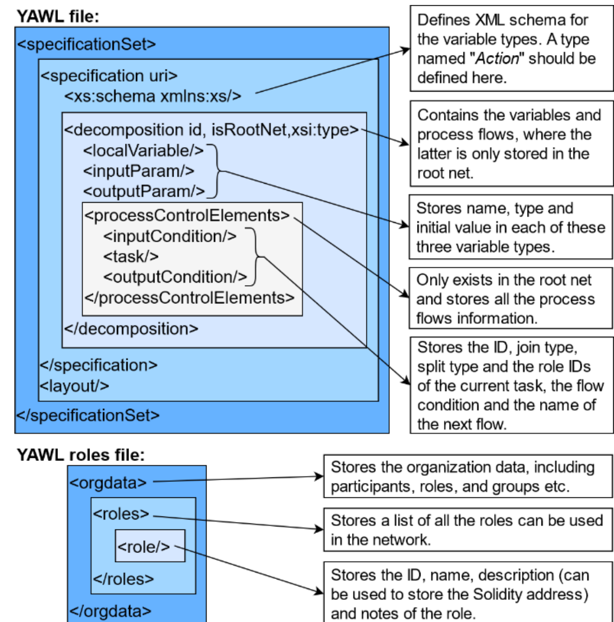

Figure 2 The explanation of the YAWL and the YAWL roles files

In the YAWL file, the stored information can be divided into two parts: specification and layout. The specification stores all the functional information of each task specified in the YAWL, where the layout stores all the geometrical data of these tasks. To aid the automatic translation from YAWL to smart contracts, an action type is designed in this method. As shown in Figure 2, the type named "*Action*" is defined in the XML schema (xs:schema) of the YAWL file for distinguishing different actions in tasks. The actions can be divided into three types, namely "view", "modify" and "pay". These three actions are defined as variables in the root net. Each task should declare one and only one variable in the "*Action*" type with one of the three variables to specify the action of this task. In this way, tasks can be automatically translated with the specified action variable. The "view" task specifies that this task is to view some information and no data should be changed in this task. The "modify" task indicates that the value of one or more state variables will be assigned or modified. The "pay" task executes automated payment action with a specific payment amount from one user address to

another user address.

There are two kinds of deposition in the specification, distinguished by with or without the *isRootNet* attribute. The former is the root net that stores the information of local variables and process flows, where the latter kind is a task storing input variables, output variables, and an "*Action*" type variable. The process flow information is stored in the element called *processControlElements* with the flow information of all tasks. The start and end tasks are tagged as *inputCondition* and *outputCondition* elements, respectively. Each task element stores not only the join type, split type and role identifier(s) of the current task, but also the flow condition and the name of the next task. All roles used in the YAWL file are defined and stored in the YAWL roles file, including the identifier, the name, the description (which can be used to store the Solidity address) and the notes of each role.

According to the explanation of the YAWL and its roles files shown in Figure 2, the information of local variables, roles, tasks with role identifier(s) and variables, and process flows can be extracted. The mapping relationship of the extracted YAWL information and the components of Solidity smart contract language is shown in Figure 3.

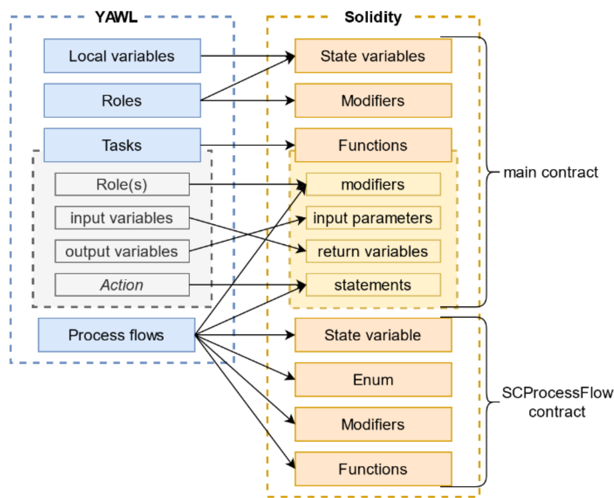

Figure 3 The component mappings from YAWL to Solidity

Solidity is an object-oriented language, which has similar components like all modern programming languages and other smart contract languages. All local variables defined in the YAWL file can be translated as state variables in the Solidity main contract. The roles extracted from the YAWL roles file will be defined as state variables under address type and be declared as modifiers at the same time in the Solidity main contract. A Solidity modifier is used to restrict the executing conditions of a function. When a modifier is called in a function, the function can be executed only when the conditions stated in the modifier are satisfied. The declared modifiers for roles are used to restrict that only certain roles can execute certain functions. The role(s) and variables of each task will be translated as modifiers, input parameters, return variables and statements based on the variable in *Action* type of each function in the Solidity main contract. Since the logic of tasks in YAWL and functions in Solidity are different, the output variables of tasks are the input parameters of functions. The process flows are translated into state variables, an enumeration, modifiers and functions in the Solidity *SCProcessFlow* contract, which is a parent contract of the main contract, to restrict the executing order of functions in the main contract.

3.3 Checking step

The aim of the checking step is to verify the correctness of the generated smart contracts. In this step, certain rules should be defined for verifying the translated smart contracts. Eight checking rules of a Solidity contract are shown as Figure 4. A translated Solidity contract has five basic components: parent contract(s), enumerations, state variables, modifiers and functions.

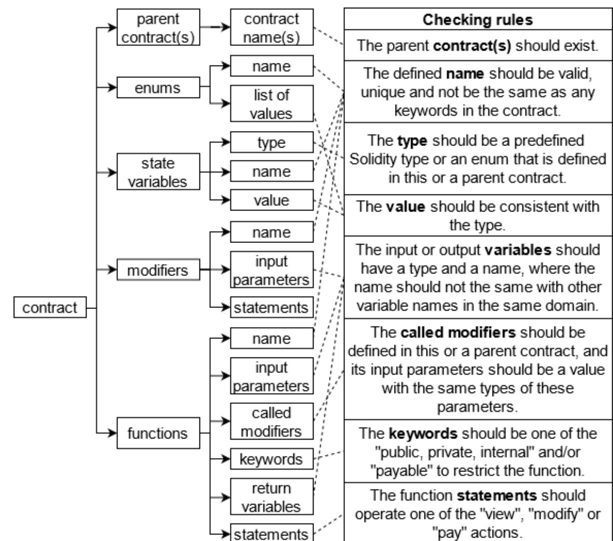

Figure 4 Checking rules of the Solidity smart contract language

When a contract is defined based on one or more parent contracts, the existence of the parent contract(s) should be checked. All the defined component names should be unique and valid. A unique name is not the same as any other names or keywords in the contract. A valid name does not contain any special characters except "_", or start with a digit. The used types should be preset or defined. The type of value should be consistent with the type of the variable. A declaration of a variable should

be in form "type name;" or "type name = value;". The modifiers in a function should be validly defined. As mentioned in the previous section, functions translated from YAWL tasks should handle one of the "view", "modify" and "pay" actions. A "view" function must not change any values of state variables. A "modify" function, on the other hand, should have at least one state variable modified in this function. In the "pay" function, a "payable" keyword must be declared. Moreover, the payer address must be payable, the payer's wallet must have at least the payment amount, and the payee address must be declared.

4 Illustrative example

An example is illustrated via YAWL to graphically model a workflow, where the workflow is transferred to the Solidity smart contract language. To figure out whether using YAWL can explain the whole process for automated reporting, checking, and payment of construction works, an example is designed in the YAWL graphical representation and shown in Figure 5. The similar workflow was presented as BPMN in the related work [18].

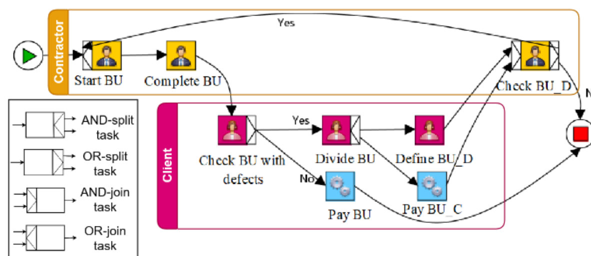

Figure 5. YAWL graphical representation of the example

The example illustrates the delivery and acceptance processes between clients and contractors for a group of construction works, which is defined as a billing unit (BU) with a total payment amount and a completion date. BUs are defined and captured in the billing plan during the contract negotiations. After that, the work can be carried out. As shown in Figure 5, eight tasks and two roles are defined in the example. After certain construction works are defined as a BU, the contractor can start to execute the task "Start BU" and the corresponding construction works. After it is completed, the contractor reports the completion of this BU via executing the "Complete BU" task. The client checks the BU and determines if there are defects regarding the construction works and executes the "Check BU with defects" task. If the BU work has no defects, the task "Pay BU" should be automatically executed, where the client will pay the full payment amount of the BU to the contractor. Otherwise, this BU will be divided into two parts via a "Divide BU" task. The BU_C is the checked and accepted part, which will be automatically paid with a predefined payment amount via "Pay BU_C" task. The BU_D is the part with defects, which will be redefined by client as a new BU with a new total payment amount and a new completion date in the "Define BU_D". The contractor can then decide to accept this task or not via "Check BU_D" task. The corresponding local variables and parameters of each task are also shown in the Figure 5.

4.1 Variables

The local variables and declared actions in *Action* type for the example are defined as Figure 6. The initial values of variables can be either assigned at the beginning or later by users. In this case, the bu_id, planedPayment and plannedCompletionDate variables are assigned in the beginning. The values of "modify" and "view" variables should be the same as their names, where the value of "pay" variable should be the name of payee address variable.

Name	Type	Scope	Initial Value
bu_id	string	Local	BUID_50b842e6
plannedPayment	unsignedInt	Local	14
plannedCompletionDate	unsignedInt	Local	20210711
actualStartDate	unsignedInt	Local	0
actualCompletionDate	unsignedInt	Local	0
reducedPayment	unsignedInt	Local	0
isAgreedStart	boolean	Local	☐
isDefect	boolean	Local	☐
modify	Action	Local	modify
view	Action	Local	view
pay	Action	Local	Contractor

Local variables

Action Type variables

Figure 6 Defined local variables and three variables with "*Action*" type

The detailed input and output variables of each task are presented in Figure 7. In a YAWL task, the values of input variables should be assigned before this task, and the values of output variables should be assigned in this task by users. An *Action* type variable is declared in each task as an input variable to tag the action type. In this example, five "modify" tasks, one "view" task and two "pay" tasks are defined. In the "Define BU_D" task, the values of variables bu_id, plannedPayment and plannedComplemetionDate are reassigned.

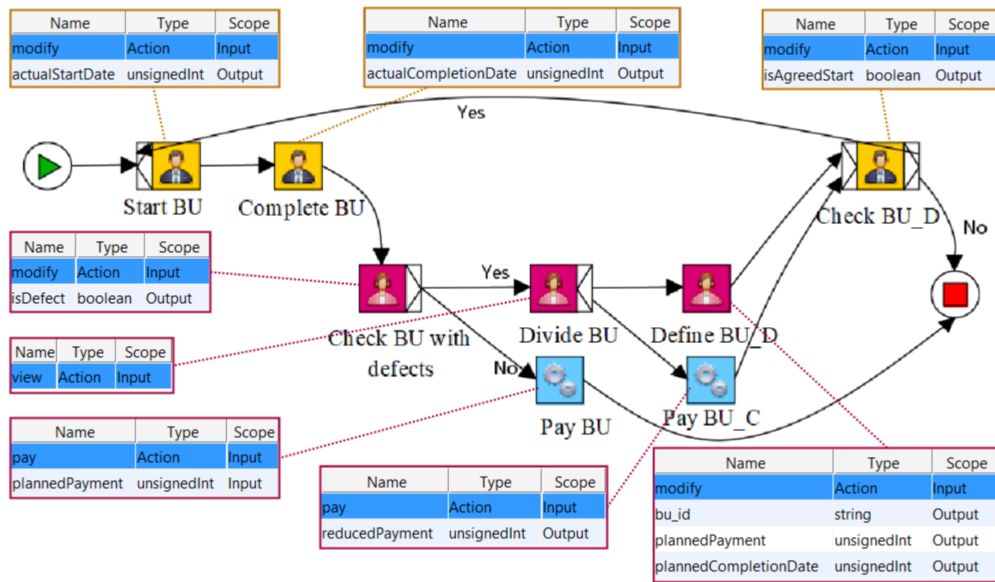

Figure 7 The declared input and output variables of each task in the example

4.2 Implementation

To test the feasibility of the method, a tool of smart contract generation is developed. The implemented smart contract generator and the corresponding results of this example are shown in Figure 8. This generator can be divided into two parts: A) parse & translate, and B) check & export. In subparts A1 and A2, YAWL graphical representation and YAWL roles should first be loaded. After clicking the translation button, the information in the YAWL and YAWL roles files is translated into the Solidity smart contract language according to the translation step, and the generated smart contracts are shown in subpart A3. The generated file structure is shown as a tree table on the left side of A3, and the detailed codes of the corresponding smart contracts are shown on the right side of A3. In part B, a tree table and error message(s) are generated after checking the generated smart contracts, and then the whole contract folder can be exported.

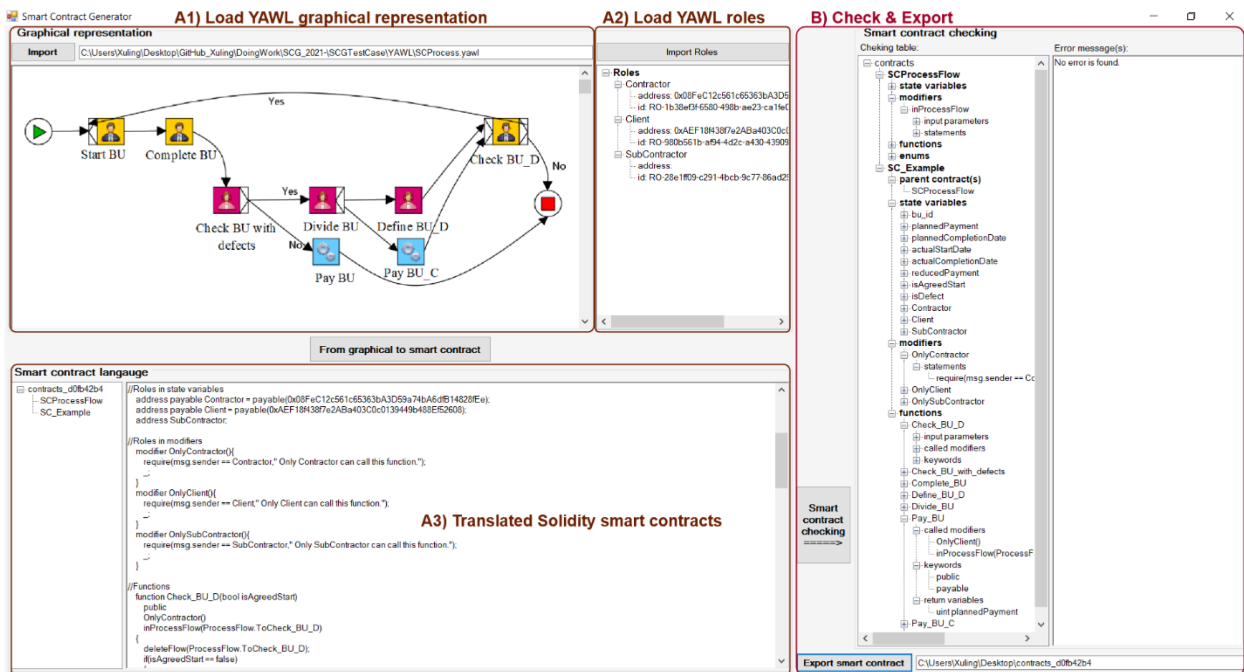

Figure 8 User Interface of smart contract language generator

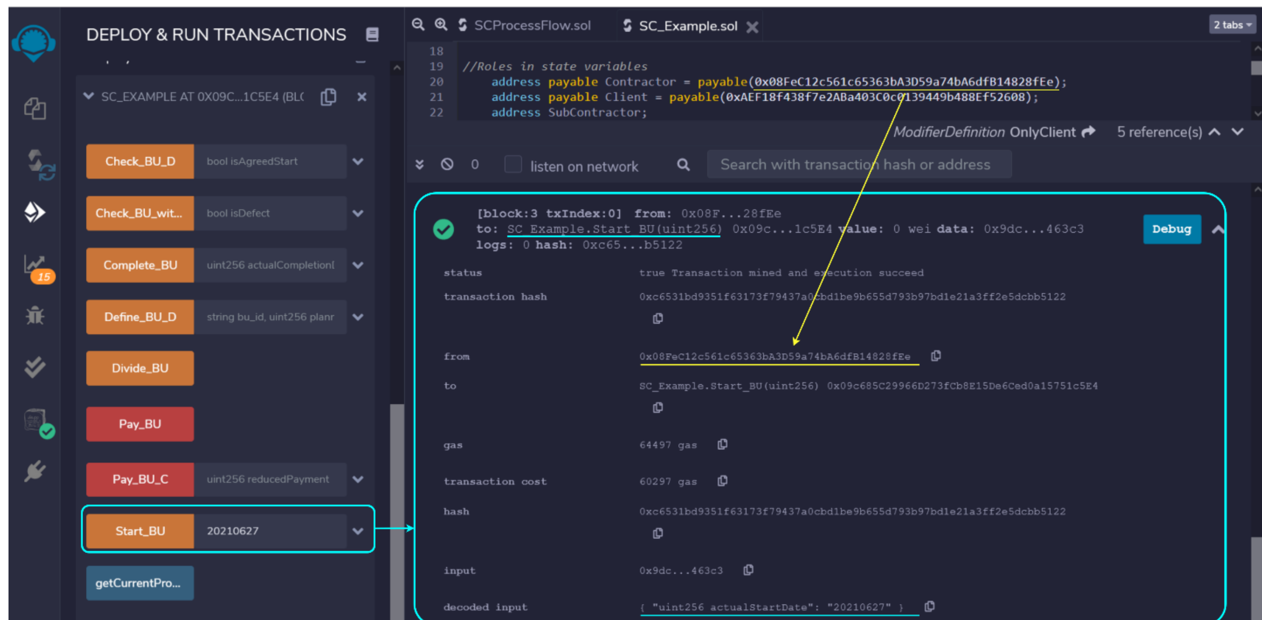

Figure 9 The test result in the Ethereum Remix after successfully executing the BU_stated function

The generated Solidity smart contract files can be simply deployed and tested via the Ethereum Remix in the browser. The test result shown in Figure 9 is after successfully executing the Start_BU function defined in the SC_Example contract. The start date is successfully set as "20210627" by the contractor at the address "0x08FeC12c561c65363bA3D59a74bA6dfB14828fEe".

5 Conclusion and future work

Smart contracts can automate many construction processes, such as contracting, delivery, acceptance and payment. However, as smart contracts are programming codes, they are difficult to be read, design, program, and verify by non-programmers. This paper proposes a framework for automatically generating smart contracts from graphical representations following four steps of parsing, translation, checking and exporting. The YAWL graphical representation and Solidity smart contract language are used to further explain the translation and checking steps. An illustrative example is presented through generating Solidity smart contracts from YAWL for automated the reporting, checking and payment process of construction works between clients and contractors. The corresponding implemented smart contract generator is illustrated and the execution result of the generated smart contracts in the Ethereum Remix is also displayed.

Future studies should consider a more detailed and practical development of payment functions. For example, in addition to using the currency in the blockchain platform, linking with banks via smart contracts for automatic payments could be more suitable for the current situation. Moreover, more components of the Solidity smart contract language (e.g., mappings, events, abstract contracts), more smart contract checking rules, other graphical representations (e.g., BPMN), and other smart contract languages should be further taken into account.

Further testing should be conducted with construction stakeholders for feedback on the feasibility of the method and the developed tool proposed in this paper. Compared with the development of the entire smart contract generation system, the development of a plugin application or a library can be more extensible, more convenient, and more widely used. In addition to generating smart contracts from graphical representations, another direction, namely translating smart contracts to XML files or graphical representations, should also be investigated.

Acknowledgement

The study was conducted as part of the BIMcontracts research project funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) within the "Smart Data Economy" technology program (project number: 01MD19006B).

Reference

[1] Adams M., Hofstede A. H. ter, and La Rosa M. Open Source Software for Workflow Management: The Case of YAWL. *IEEE Softw.*, 28(3): 16–19, 2011.

[2] Badi S., Ochieng E., Nasaj M., and Papadaki M.

Technological, organisational and environmental determinants of smart contracts adoption: UK construction sector viewpoint. *Construction Management and Economics*, 39(1): 36–54, 2021.

[3] Borrmann A., König M., Koch C., and Beetz J. *Building information modeling: Technology foundations and industry practice.* Springer, Cham, Switzerland, 2018.

[4] Buterin V. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[5] G. A. Pierro. Smart-Graph: Graphical Representations for Smart Contract on the Ethereum Blockchain. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*: 708–714, Hawaii, US, 2021.

[6] Georgakopoulos D., Hornick M., and Sheth A. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distrib Parallel Databases*, 3(2): 119–153, 1995.

[7] He X., Qin B., Zhu Y., Chen X., and Liu Y. SPESC: A Specification Language for Smart Contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*: 132–137, Tokyo, Japan, 2018 - 2018.

[8] Lohmann N., Verbeek E., and Dijkman R. Petri Net Transformations for Business Processes – A Survey. In *Transactions on Petri Nets and Other Models of Concurrency II. Special Issue on Concurrency in Process-Aware Information Systems*. Springer, Berlin, Heidelberg: 46–63, 2009.

[9] López-Pintado O., García-Bañuelos L., Dumas M., Weber I., and Ponomarev A. Caterpillar: A business process execution engine on the Ethereum blockchain. *Softw: Pract Exper*, 49(7): 1162 - 1193, 2019.

[10] Monteiro E., Righi R., Kunst R., Da Costa C., and Singh D. Combining Natural Language Processing and Blockchain for Smart Contract Generation in the Accounting and Legal Field. In *Intelligent human computer interaction. 12th International Conference, IHCI 2020, Proceedings, Part I and II*: 307–321, Daegu, South Korea, 2021.

[11] Penzes B. *Blockchain Technology in the Construction Industry*. Institution of Civil Engineers (ICE), London, 2018.

[12] Skotnica, M., Klicpera, J., & Pergl, R. Towards Model-Driven Smart Contract Systems–Code Generation and Improving Expressivity of Smart Contract Modeling. In *EEWC Forum 2020*, (online) Bozen / Bolzano, Italy, 2020.

[13] Suvorov D. and Ulyantsev V. *Smart Contract Design Meets State Machine Synthesis: Case Studies*. arXiv.org, 2019.

[14] Szabo N. Smart contracts. *Virtual School*: 26, 1994.

[15] Tateishi T., Yoshihama S., Sato N., and Saito S. Automatic smart contract generation using controlled natural language and template. *IBM J. Res. & Dev.*, 63(2/3): 6:1-6:12, 2019.

[16] Varela-Vaca Á. J. and Quintero A. M. R. Smart Contract Languages: A Multivocal Mapping Study. *ACM Comput. Surv.*, 54(1): 1–38, 2021.

[17] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: on the expressive power of (Petri-net-based) workflow languages. *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*: 1–20, 2002.

[18] Ye X. and König M. Framework for Automated Billing in the Construction Industry Using BIM and Smart Contracts. In *Proceedings of the 18th International Conference on Computing in Civil and Building Engineering. ICCCBE 2020*. Springer, Cham: 824–838, 2021.

[19] Zupan N., Kasinathan P., Cuellar J., and Sauer M. Secure Smart Contract Generation Based on Petri Nets. In *Blockchain technology for industry 4.0. Secure, decentralized, distributed and trusted industry environment*. Springer, Singapore: 73–98, 2020.