Towards Model-Driven Smart Contract Systems – Code Generation and Improving Expressivity of Smart Contract Modeling

Marek Skotnica, Jan Klicpera and Robert Pergl

Faculty of Information Technology,
Czech Technical University, Prague, Czech Republic
marek.skotnica@fit.cvut.cz
ORCiD: 0000-0002-8811-3389
ORCiD: 0000-0003-2980-4400

Abstract. Public blockchains are increasingly important in industries such as finance, supply-chain management, and governance. In the last two years, there has been increased usage of blockchain for decentralized finance (DeFi). The usage of DeFi mainly consists of cryptocurrency lending and providing liquidity for decentralized exchanges. However, the considerable volume of reports shows large financial losses during network congestion, increasing transaction prices, programming errors, and hacker attacks. One survey suggested that only 40% of people working with DeFi smart contracts understand their source code.
To address the issues, this paper proposes a model-driven approach to create blockchain smart contracts based on a visual domain-specific language called DasContract. An improved design of the DasContract language is presented, and a code generation process into a blockchain smart contract is described. The proposed approach is demonstrated on a proof-of-concept model of a decentralized mortgage process where the contract is designed, generated, and simulated in a blockchain environment.

Key words: Process Modeling, Blockchain, Smart Contract, DasContract

1 Introduction

The blockchain smart contracts promise a revolution in many industries that so far resisted the digital revolution. They may play a significant role in finance, supply chain management, voting, and many other industries. However, the promises of the blockchain technology were not yet materialized. According to the Gartner, mainstream adoption is expected in 2030 [15].

The public blockchains' primary use is currently the decentralized finance (DeFi) that is used for decentralized exchanges and crypto loans. However, there are many problems DeFi is currently facing. The first one is network congestion, which increases the transaction processing time and price (that reaches up to 12 USD per transaction [39]). Another issue is the smart contract programming

errors. In one of the many incidents, an input error led to a token price plunging 25% [26]. Furthermore, according to the CoinGeco survey [7], only 40% of the DeFi users can read and understand the smart contract's source code.

To address these challenges, a visual language for modeling smart contracts, DasContract, has been proposed in [35]. It argues that by using a visual domain-specific language (DSL), the readability of smart contracts would be more comfortable, and using model-driven engineering (MDE) approach generates the smart contracts source code. However, a complete method to generate the source code was not provided, and the included case study was very elementary.

Therefore, this paper aims to bridge the gap and build on top of the preliminary research. First, the DSL initially provided by the DasContract paper [35] was completely redesigned to support an automatic and blockchain implementation-independent code generation suitable for a limited execution environment provided by the blockchain technology. Second, a straightforward way to generate smart contract algorithms with non-fungible tokens was described together with an open-source implementation algorithm. Finally, to demonstrate the functionality of the proposed approach, a decentralized mortgage case study was created. A model of a mortgage process running in blockchain was modeled, Solidity smart contract was generated, and finally, a simulation of the functionality was performed.

The paper is organized as follows: In Section 2, the research method and the research question are formulated. In Section 3, the underlying scientific foundations are briefly discussed. An approach to generate blockchain smart contracts from a DasContract language is introduced in Section 4. The proposed approach is demonstrated on a Mortrage case study in Section 5. The related research is discussed in Section 6. Finally, in Section 7, the current results are summarized, and further research is proposed.

2 Research Approach

This research applies the design science (DS) approach of Hevner [19, 18], which is shown in Figure 1. In the first cycle, a relevance of the problem is discussed in Section 1. The second cycle is described in Section 4. And finally, the relevant grounding into the existing knowledge base is in Section 3, Section 4, and Section 7.

The research question is: **How to generate blockchain smart contracts from a high-level modelling language in a deterministic way so the probability of creating error-free smart contracts is increased?**

3 Theoretical Background

3.1 Blockchain

First conceptualized in 2008 [25] by Satoshi Nakamato, blockchain is a technology that provides a cryptographically secure, decentralized method to store data.

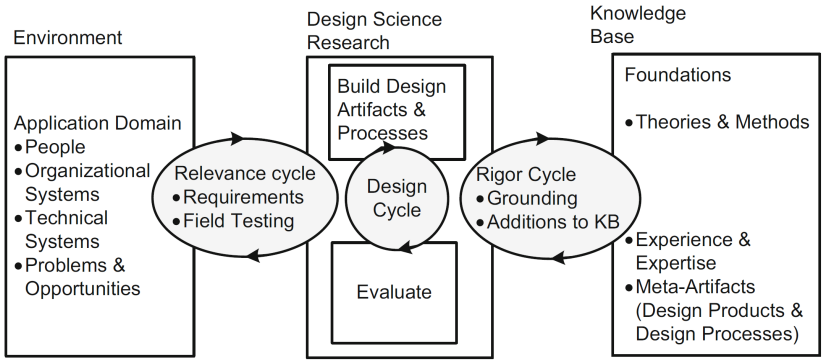

Fig. 1. Design Science Research Cycles [18]

The key aspects of a blockchain systems are: 1)Decentralization The data in a blockchain is distributed and managed by a cluster of computers, without any central authority. 2)Transparency All the data in a blockchain is completely visible to any computer in the blockchain.3) Immutability Once a data block is verified and added to the blockchain, it is not possible to edit or remove it.

3.2 Smart Contracts

A smart contract is a computer protocol that allows creating digital contracts between multiple parties without a need for a trusted third-party intermediary. Once deployed, the smart contract will automatically evaluate the commitments that may occur to the parties over time, based on the agreed terms [36].

Ethereum [13, 14] is a smart contract implementation used in this paper that is described as: "*an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system's state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs.*" [3]

As further described in [3], the smart contracts are deployed onto the blockchain in the form of a specialized bytecode. This bytecode then runs on each Ethereum node in a limited execution environment, called Ethereum Virtual Machine (EVM). Since creating smart contracts directly in the bytecode would be too impractical, multiple specialized high-level languages have been created, together with the compilers needed to convert them into EVM bytecode. The most popular high-level language for creating EVM-based smart contracts is Solidity.

3.3 Cryptographic Tokens

As described by Voshmgir [38]: "*Cryptographic tokens represent programmable assets or access rights, managed by a smart contract and an underlying distributed ledger. They are accessible only by the person who has the private key for that address and can only be signed using this private key.*"

On the Ethereum platform, Ethereum Improvement Proposals (EIPs) [1] are used to standardize the general structure of tokens. This ensures that the smart contracts running on Ethereum can interact with tokens defined by other smart contracts.

One of these standards is the ERC-20, providing a typical list of rules on how the tokens should be structured. These tokens are fungible, meaning that tokens of the same type are interchangeable. That makes them easy to trade, as there is no need to differentiate the tokens. To represent unique assets, however, the tokens must be non-fungible – every token is unique and non-interchangeable, even if they are of the same type. In order to facilitate these types of tokens, the ERC-721 standard was created. [38]

3.4 BPMN

BPMN is a standard notation for business process modeling under the Object Management Group (OMG) [27]. There are three levels of BPMN modeling based on the goal. This paper uses the DasContract language that builds on BPMN level 3 to express machine-executable process models.

3.5 Unified Modeling Language (UML)

Unified Modeling Language (UML) [32] is a standardized modeling language enabling developers to specify, visualize, construct and document artifacts of a software system. [37] In this paper, a UML Class diagram is used to describe a meta-model of a DasContract language and also to represent a DasContract data model.

3.6 DasContract

The DasContract was first introduced in [35] and it builds on top of the existing knowledge about Business Process Management (BPM) [4] and Enterprise Engineering discipline [8]. DasContract provides a visual domain-specific language to describe blockchain smart contracts that is based on an extended combination of DEMO modeling language [8], BPMN [6], and UML [28].

The main goal of the DasContract is to provide a way to conclude digital contracts for people, companies, and governments without the need for an intermediary. The contract conditions are agreed on upfront and inserted into a blockchain smart contract that ensures the contract's immutability and execution according to the agreed conditions.

The proposed concept architecture of the DasContract is shown on Figure 2. First, a contract between parties is formalized using the DasContract language and a legal text, then a smart contract is generated, and finally, the people, companies, and legal authorities interact according to the smart contract logic.

However, the original paper does not provide a straightforward method for translating the DEMO model to the executable BPMN and does not provide

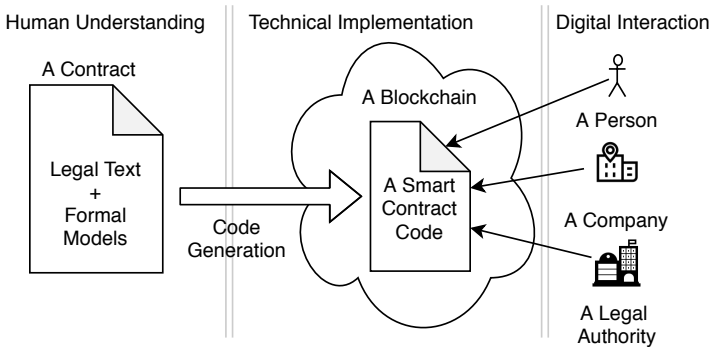

Fig. 2. A proposed concept architecture of DasContract [35].

code generation algorithms. This leaves a gap that this paper aims to fill by introducing a new DSL language (shown in Figure 3) and describes how the smart contract code should be generated.

4 Towards Generating Smart Contracts from DasContract Language

This section introduces a way to generate blockchain smart contracts using a Model-Driven Engineering approach (MDE) [5]. To achieve this, a complete redesign of the original DasContract metamodel [35] was done, and a way of automatically generating blockchain smart contracts is described. Ethereum Solidity language is used as a demonstration; however, the principles described should be transferable to other blockchain implementations.

The new DasContract DSL metamodel is shown on fig. 3. The original DasContract metamodel was based on DEMO models. Although the execution of DEMO models is described in [34] and the translation of DEMO to BPMN in [24], the approach creates models that contain all possible execution paths according to the DEMO transaction axiom. This is not desired in a blockchain smart contract because only the necessary execution paths should be modeled to prevent unwanted behavior. However, the DEMO models still should be produced before modeling the DasContract to achieve a better ontological understanding of the domain.

As a basis for the execution behavior, an extended subset of BPMN 2.0 level 3 is used. However, compared to the BPM systems that interpret the model, an approach that generates code with the model behavior is used. This approach is used due to blockchain performance and storage limitations. Each line of code executed in the blockchain costs money, and the storage costs are also at a premium.

The algorithm described in this paper is implemented in C# programming language and published on Github under an MIT license [33].

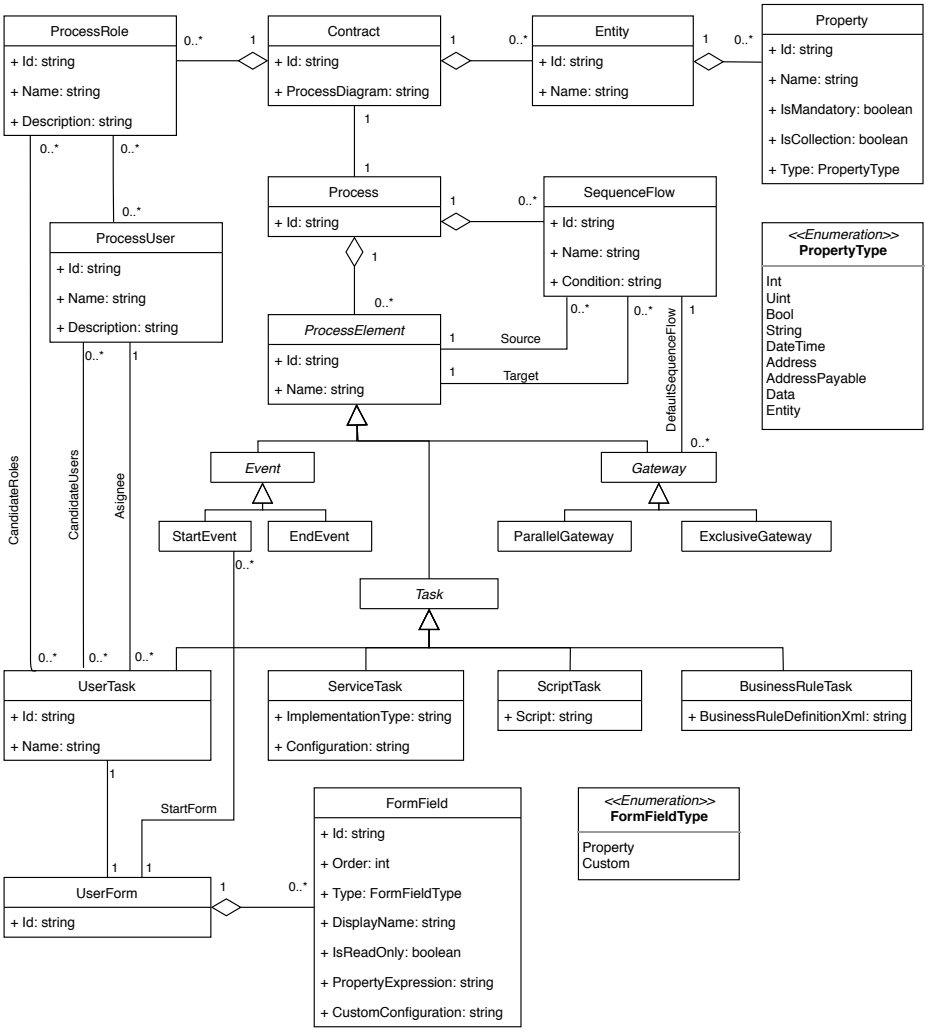

Fig. 3. A new DasContract DSL metamodel

The DasContract DSL consists of the following models: *Data Model* Specifies data structures used inside of the smart contract. These structures can be referenced inside of the process model. *Process Model* Specifies a contract's business process as an extended subset of the BPMN 2.0 language. *Forms Model* Specifies a user form required to fill by the user in a process user task. The forms provide a way to interact with smart contracts.

4.1 Data Model

The data model's implementation is straightforward because the blockchain smart contract languages provide generous native support to specify data structures. The supported concepts are entities, properties, entity properties, and arrays. Each property can also be marked as mandatory. The data types of properties are Int, UInt, Bool, String, DateTime, Address, AddressPayable, Data, Entity. The types Address and AddressPayable are blockchain specific and support cryptocurrency or token payments.

Example Let's have an entity *Payment* with four properties – two addresses identifying the sender and receiver, a numeric defining the amount sent and a boolean indicating whether the payment was on schedule. The generated code is shown in Listing 1.

```
struct Payment {
       uint256 amount;
       bool onSchedule;
       address sender;
       address receiver;
}
```

Listing 1. An example of a generated data structure.

4.2 Process Model

The process model uses an extended subset of the BPMN 2.0 level 3 notation. The formal specification of a BPMN execution is already well researched, and there are many different formalizations such as [9, 21]. In this paper, the algorithms are based on Petri-net based formalization described in [10].

The blockchain smart contracts are more similar to a programmable database rather than a desktop, web, or console application. Due to this fact, not all of the BPMN concepts can be implemented or make sense. Therefore only a limited subset is implemented: user task, script task, XOR gateway, parallel gateway, start event, and end event. The blockchain-specific activities were added: payment task. The payment task is a task attribute that can be added to both user and script task to add support to work with cryptocurrency or tokens.

Process Flow Exclusive gateways are converted into a sequence of if-else statements, advancing the execution based on the statement's result. Parallel gateways are more complex, as they can not only create multiple execution branches but are also used for synchronizing multiple flows (branches) into a single flow. If a gateway has multiple incoming flows, then a counter is defined, keeping track of the number of flows that have reached the gateway. The outgoing flows are triggered once the counter matches the number of incoming flows. An example can be seen in Listing 2, that contains the logic of parallel gateways generated based on Figure 4.

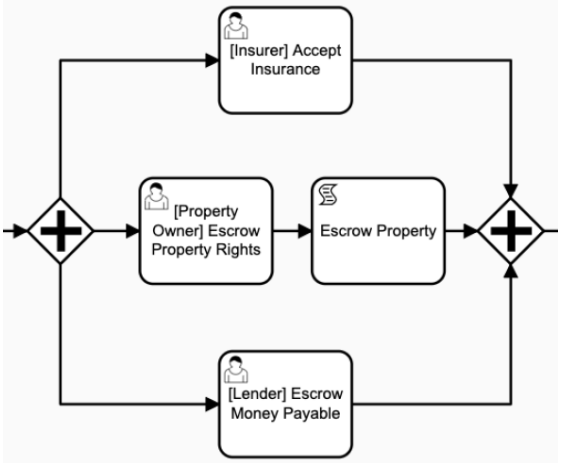

Fig. 4. A contract snippet used to demonstrate a gateway conversion in Listing 2 and user task conversion in Listing 3

```
int Gateway_2Incoming = 0;

function Gateway_1Logic() internal {
       ActiveStates["EscrowPropertyRights"] = true;
       ActiveStates["AcceptInsurance"] = true;
       ActiveStates["EscrowMoneyPayable"] = true;
}

function Gateway_2Logic() internal {
       if(Gateway_2Incoming==3){
              ActiveStates["ValidateContractPayable"] = true;
              ValidateContractPayable();
              Gateway_2Incoming = 0;
       }
}
```

Listing 2. Gateway logic generated based on the model snippet in Figure 4.

Activities Functions are also used to encapsulate the logic of activities. Unlike the internal gateway functions, these functions are publicly visible. User activities allow accepting parameters and store them inside of the contract using property binding logic. The generator also allows restricting access to function execution based on the executor's address. This is done using roles defined using square brackets inside the name of the activity (see Figure 4). The addresses to each role are assigned at runtime, meaning that anyone can execute an activity with an unassigned role. The first execution assigns the role, reserving the remaining activities of the given role to that address.

```
modifier isEscrowPropertyRightsAuthorized{
       if(addressMapping["Property Owner"] == address(0x0)){
              addressMapping["Property Owner"] = msg.sender;
       }
       require(msg.sender==addressMapping["Property Owner"]);
       _;
}

modifier isEscrowPropertyRightsState{
       require(isStateActive("EscrowPropertyRights")==true);
       _;
}

function EscrowPropertyRights(bool Agree) isEscrowPropertyRightsState
    isEscrowPropertyRightsAuthorized public {
       ActiveStates["EscrowPropertyRights"] = false;
       escrowagreement.agreeToEscrow = Agree;
       ActiveStates["EscrowProperty"] = true;
       EscrowProperty();
}
```

Listing 3. User task logic generated based on the model snippet in Figure 4.

An example of a converted user activity can be seen in Listing 3, generated based on the *Escrow Property Rights* activity in Figure 4. The generated function contains two modifiers, checking whether the contract is in the valid state and whether the executor (*Property Owner* role) is authorized. The function has one input parameter that has been defined using the editor. This parameter is then stored inside the contract according to the defined property binding logic. An address of the property owner is read from the *msg.sender* property that provides a sender's public address verified by his private key while sending the transaction to the blockchain.

Token Activities Described in Section 3.3, tokens play a significant role in the smart contract ecosystem by allowing to express ownership of an asset. By complying with the token standards, these tokens can also interact with other smart contracts. An activity that would allow us to create/send/receive tokens would significantly improve the generated smart contracts' capabilities. For example, it would allow sending voting ballots (in the form of a token) to the voters.

4.3 Forms Model

The forms model defines a form that is shown to a user and allows interaction with a contract. Such logic is implemented in two places called on-chain and off-chain.

On-chain code is run inside the blockchain. Our algorithm is represented as parameters that are passed into a function generated from a user task. An example is shown on Listing 3 - the method *EscrowPropertyRights* contains a parameter Agree that will be provided by the user while calling the method.

Off-chain code is run inside a cryptocurrency wallet to provide the user with comfortable user experience while interacting with a smart contract. However, the Ethereum Solidity does not support off-chain code, and therefore we leave this aspect for further research.

4.4 Design, Compilation and Execution

For the modeling of the new DasContract models, a new visual modeling environment was created. The visual modeling reduces the modeler's errors and reduces the time required to produce a model. The modeling environment is available on Github under an MIT license [11].

Compilation to the Solidity code assumes a valid DasContract model created in the modeling environment and does not check for other errors. The last check is done during the Solidity code compilation (usually in the Remix environment) and allows the modeler to fix issues in custom script tasks and user task validation logic.

4.5 Limitations

There are the following limitations to this approach, and they should be addressed in future research. First, the script tasks and a validation logic of user tasks still need to be expressed in Solidity language. An implementation-independent DSL for such expressions should be added to the design of the language. Second, the language only contains support for receiving tokens. Support to issue both fungible and non-fungible tokens would be required. Third, it is not clear how would people authenticate themselves and prove their roles outside of the standard smart contract wallet. A way to support decentralized identity standards such as W3C DID should be explored. Finally, according to Gartner [15], the public blockchains are still immature for mass adoption mostly because of low transaction processing capability and high cost and, therefore, only minimal applications such as initial coin offerings, escrows, and decentralized finance are currently possible.

5 Case Study: Mortgage

This section provides a case study to demonstrate the capabilities of the approach proposed in the previous section. A case of a decentralized mortgage that supports non-fungible tokens (ERC-721) to represent the property ownership was created. This case study designs the process in a decentralized way

that means that the property token is held in the smart contract as collateral. A scenario where the lenders can request a default of the loan is modeled. The complete DasContract model and the generated source code are published on Github under an MIT license [33].

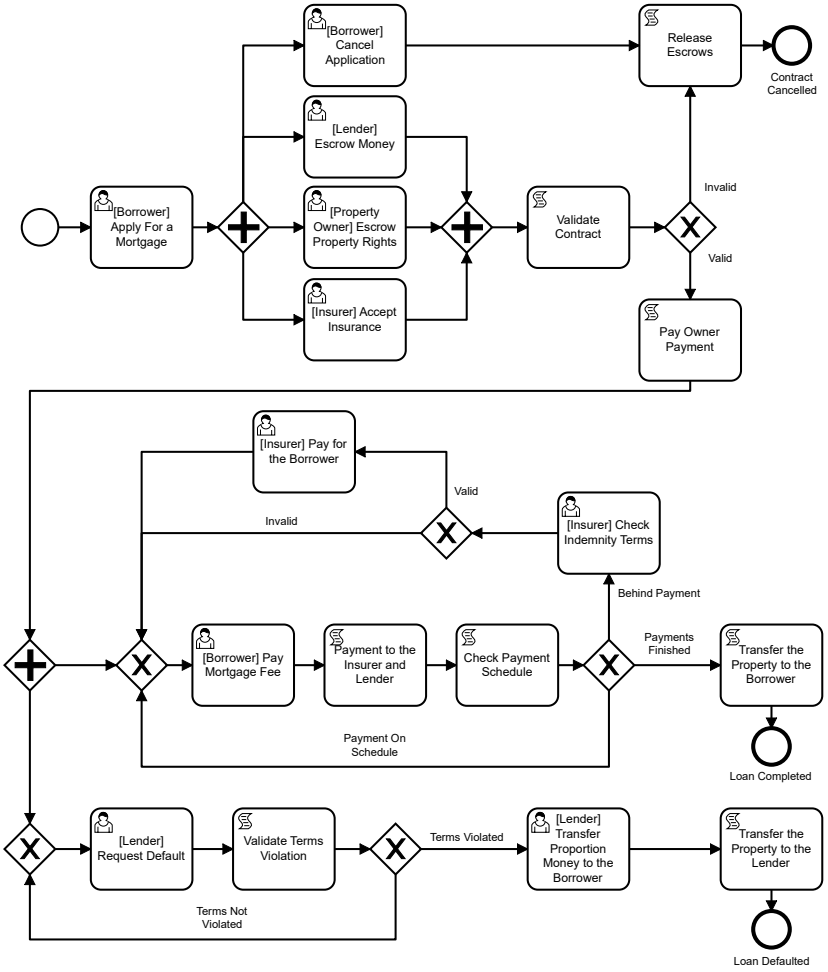

Fig. 5. Mortgage Process Model

The following steps were taken while conducting this case study: 1) A DasContract model was created in a visual editor. 2) An Ethereum Solidity smart contract code was created by an algorithm described in Section 4. 3) A simulation of the generated smart contract was performed in the Remix [2] test environment.

The process model of the proposed process can be seen in Figure 5. The process begins with a borrower applying for a mortgage. Three other parties – insurer, property owner, lender – must then confirm their involvement by carrying out their specific action. If any of them declines, or the borrower changes their mind, then the contract is deemed invalid, and any escrows that have been already transferred will be released back to their previous holders.

When all parties agree and the contract is successfully validated, then the full payment to the owner is automatically released, also ending their involvement in the contract. In the next phase of the contract, the borrower is tasked to carry out to periodically pay the mortgage fees. Those fees are then automatically distributed to the insurer and lender. The payment schedule is checked afterward, resulting in three possible scenarios. First, the payment is on schedule; the contracts state will return to "waiting" for another payment. Second, The payments are behind schedule; in this case, the insurer checks the indemnity terms, paying for the borrower if they are met. Third, the payments are finished, in which case the property is automatically transferred to the borrower, ending the contract.

Before the payments are fully finished, the lender is also at any time allowed to request for borrower's default to check whether the borrower has not violated terms. If the terms have indeed been violated, then the lender will pay proportion money defined in the terms to the borrower. The property will then be transferred to the lender, ending the contract.

The described process was simulated using the Remix IDE. The test environment allows to perform transactions from various addresses, enabling to simulate people interacting with the smart contract. The transaction was successfully run by the borrower (left sidebar), returning status information about the transaction (right window).

The limitations of the case study are the following. First, the mortgage is paid in a cryptocurrency, which is a very volatile asset. This can be addressed using a stablecoins [23]. The second issue is that the case requires a cadastre of properties being represented by a non-fungible token that is recognized in a country's legal context. This can be addressed by a legal binding of the token to property and mapping to a real cadastre record using a blockchain oracle solution such as [12]. Third, the insurer is represented by a human oracle and is not enforced by a code to pay. Therefore, the terms of insurance would be needed to sign in a separate legal contract. This is already a part of the DasContract architecture described at Figure 2.

6 Related Research

A very similar research is done by the Caterpillar project [30, 31] that is creating a BPM engine in the Solidity language. The main difference between the approaches is that the DasContract generates the logic of the model into the smart contract where the Caterpillar interprets it. The goal of the DasContract is to provide a decentralized way to conduct contracts between people, compa-

nies, and governments in a blockchain implementation independent way. The Caterpillar is currently bound to the Solidity programming language.

Other approaches provide a graphic environment to visually compose smart contract code based on Blockly [17] such as [16]. There is also another approach that proposes a domain specific language for definition of financial smart contracts called Marlowe [22]. The approach presented in this paper does not represent the script part in the visual format; it instead focuses on modeling data, processes, and forms that is not supported by Blockly-based approaches.

7 Conclusions

The paper introduced an approach to creating a model-driven blockchain smart contracts. The potential benefits are: 1) An increase readability of the blockchain smart contracts by providing a visual DSL that allows contract simulation. 2)The simulation also contributes to eliminating smart contract errors as the behavior can be examined before publishing the contract. 3)The expressivity was improved by a complete redesign of the DasContract DSL and providing the capability to support non-fungible tokens. To demonstrate the proposed approach's functionality, a decentralized mortgage case study was presented in Section 5. The mortgage case was modeled; a Solidity code was generated and simulated in the Remix environment.

The major limitation of this approach remains the immaturity of available blockchain implementations. According to the Gartner [15], the blockchain technology is very immature to support most of the potential use cases, and there is still a tremendous amount of research, implementation, and adoption to be done.

Further research is following: 1) Support for generation of smart contracts on different platforms. 2) Case studies and usability studies to improve the proposed language design. 3) Explore a possible extension with more BPMN symbols and concepts. 4) Explore a possible extension of the proposed model by DMN [29] OMG standard. 5) Propose a method to devise DasContract models from DEMO. 6) Propose a case study of a smart contract in financial domain aligned with CC-CP ontology [20].

Acknowledgement This research has been supported by CTU SGS grant No. SGS20/209/OHK3/3T/18

References

1. Ethereum improvement proposals, `https://eips.ethereum.org/`
2. Remix - ethereum ide, `https://ethereum.org/en/foundation/`
3. Antonopoulos, A., Wood, G.: Mastering Ethereum: Building Smart Contracts and DApps (2018), `https://github.com/ethereumbook/ethereumbook/blob/develop/book.asciidoc`
4. BPM: What is BPM (2012), `http://bpm.com/what-is-bpm`

5. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice: Second Edition. Morgan & Claypool Publishers, 2nd edn. (2017)
6. CAMUNDA: Bpmn 2.0 symbol reference, https://camunda.com/bpmn/reference/
7. CoinGeco: Coingecko yield farming survey 2020, https://www.coingecko.com/buzz/yield-farming-survey-2020?locale=en, [online], [cit. 2020-09-22]
8. Dietz, J.L.G., Mulder, H.B.F.: Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation. The Enterprise Engineering Series, Springer International Publishing (2020), https://www.springer.com/gp/book/9783030388539
9. Dijkman, R., Van Gorp, P.: Bpmn 2.0 execution semantics formalized as graph rewrite rules. In: Mendling, J., Weidlich, M., Weske, M. (eds.) Business Process Modeling Notation. pp. 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of bpmn process models using petri nets. Queensland University of Technology, Tech. Rep pp. 1–30 (2007)
11. Drozdík, M., e.a.: Dascontract editor github repository (2020), https://github.com/drozdik-m/das-contract-editor
12. Ellis, S., Juels, A., Nazarov, S.: Chainlink: A decentralized oracle network (2017), https://link.smartcontract.com/whitepaper
13. Ethereum Foundation: About the ethereum foundation, https://ethereum.org/en/foundation/
14. Ethereum Foundation: Ethereum whitepaper, https://ethereum.org/en/whitepaper/
15. Garther: The reality of blockchain, https://www.gartner.com/smarterwithgartner/the-reality-of-blockchain/, [online], [cit. 2019-01-29]
16. Github: Solidity for blockly, https://github.com/promethe42/blockly-solidity, [online], [cit. 2020-08-10]
17. Google: Blockly, https://developers.google.com/blockly/, [online], [cit. 2020-08-10]
18. Hevner, A.: A Three Cycle View of Design Science Research. Scandinavian Journal of Information Systems 19(2) (Jan 2007), http://aisel.aisnet.org/sjis/vol19/iss2/4
19. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design Science in Information Systems Research. MIS Q. 28(1), 75–105 (Mar 2004)
20. Huňka, F., Kervel, S.: A Generic DEMO Model for Co-creation and Co-production as a Basis for a Truthful and Appropriate REA Model Representation, pp. 203–218 (08 2019)
21. Lam, V.S.: A precise execution semantics for bpmn. IAENG International Journal of Computer Science 39(1), 20–33 (2012)
22. Lamela Seijas, P., Thompson, S.: Marlowe: Financial Contracts on Blockchain: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV, pp. 356–375 (11 2018)
23. Mita, M., Ito, K., Ohsawa, S., Tanaka, H.: What is stablecoin?: A survey on price stabilization mechanisms for decentralized payment systems. In: 2019 8th International Congress on Advanced Applied Informatics (IIAI-AAI). pp. 60–66. IEEE (2019)

24. Mráz, O., Náplava, P., Pergl, R., Skotnica, M.: Converting DEMO PSI Transaction Pattern into BPMN: A Complete Method. In: Aveiro, D., Pergl, R., Guizzardi, G., Almeida, J.P., Magalhães, R., Lekkerkerk, H. (eds.) Advances in Enterprise Engineering XI: 7th Enterprise Engineering Working Conference, EEWC 2017, Antwerp, Belgium, May 8-12, 2017, Proceedings, pp. 85–98. Springer International Publishing, Cham (2017), http://dx.doi.org/10.1007/978-3-319-57955-9_7
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf
26. News, B.: Defi's smart contract risks: Cream finance's input error led to cream token plunging 25%, https://blockchain.news/news/defi-smart-contract-risks-cream-finance-input-error-token-plunge, [online], [cit. 2020-09-22]
27. OMG: Business Process Model and Notation (BPMN), Version 2.0 (Jan 2011), http://www.omg.org/spec/BPMN/2.0
28. OMG: Unified Modeling Language, version 2.5 (Mar 2015), http://www.omg.org/spec/UML/2.5
29. OMG: Decision Model and Notation (DMN), Version 1.2 (Jan 2019), https://www.omg.org/spec/DMN/1.2/
30. Pintado, O.: Caterpillar: A Blockchain-Based Business Process Management System (2017)
31. Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: CATERPILLAR: A Business Process Execution Engine on the Ethereum Blockchain (2018)
32. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)
33. Skotnica, M., e.a.: Dascontract 1.0 github repository (2020), https://github.com/CCMiResearch/DasContract/tree/v1.0
34. Skotnica, M., van Kervel, S.J.H., Pergl, R.: A DEMO Machine - A Formal Foundation for Execution of DEMO Models. In: Aveiro, D., Pergl, R., Guizzardi, G., Almeida, J.P., Magalhaes, R., Lekkerkerk, H. (eds.) Advances in Enterprise Engineering XI. pp. 18–32. Springer International Publishing, Cham (2017)
35. Skotnica, M., Pergl, R.: Das contract - a visual domain specific language for modeling blockchain smart contracts. In: Aveiro, D., Guizzardi, G., Borbinha, J. (eds.) Advances in Enterprise Engineering XIII. pp. 149–166. Springer International Publishing, Cham (2020)
36. Szabo, N.: Smart contracts, https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html
37. Techopedia: Definition - what does unified modeling language (uml) mean?, https://www.techopedia.com/definition/3243/unified-modeling-language-uml, [online], [cit. 2019-01-29]
38. Voshmgir, S.: Token Economy: How Blockchains and Smart Contracts Revolutionize the Economy. Shermin Voshmgir (2019)
39. YChart: Ethereum average transaction fee (2020), https://ycharts.com/indicators/ethereum_average_transaction_fee