Evaluation of a High-Level Metamodel for Developing Smart Contracts on the Ethereum Virtual Machine

Gislainy Crisostomo Velasco[1], **Marcos Alves Vieira**[1,2], **Sergio T. Carvalho**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Caixa Postal 131 — 74001-970 — Goiânia — GO -– Brasil

[2]Instituto Federal de Educação, Ciência e Tecnologia Goiano (IF Goiano)
76200-000 – Iporá – GO – Brasil

gislainycrisostomo@discente.ufg.br, marcos.vieira@ifgoiano.edu.br,

sergio@inf.ufg.br

Abstract. *Developers of smart contracts face challenges such as the immutability of contracts and asset storage, which make the activity complex and error-prone. To make contracts safer and more reliable, Model-Driven Engineering (MDE) offers an alternative approach with an emphasis on the High-Level Metamodel for Smart Contract (HLM-SC), which allows for the high-level declaration of elements within a contract. This paper evaluates the HLM-SC using the MQuaRE framework to verify its conceptual validity with 11 external evaluators. The results demonstrated the acceptance of the metamodel. Additionally, this paper presents a guide on how to use HLM-SC to facilitate its adoption by developers. Finally, it demonstrates the application of HLM-SC in a scenario related to the NFT industry.*

1. Introduction

Blockchain technology and smart contracts have enabled the creation of new decentralized applications that explore properties such as immutability, traceability, transparency, and privacy [Angelis and Ribeiro da Silva 2019, Khan et al. 2019]. However, the immutability of contracts and asset storage present significant challenges for developers, making the activity complex and error-prone [Jurgelaitis et al. 2022, Ferreira et al. 2020]. These challenges, in turn, represent barriers to entry for new developers in the field, as the learning curve is steep [Garamvölgyi et al. 2018, Jiao et al. 2020a].

Contracts developers face numerous challenges during the conception and implementation process, including the need for a more rigorous and secure approach [Zeng et al. 2022, Feist et al. 2019, Li et al. 2021, Dharanikota et al. 2021, Jiao et al. 2020b]. In the case of the Ethereum Virtual Machine (EVM), the contract development platform requires developers to have knowledge of its functioning mechanism and the high-level language used to write contracts. Additionally, it offers limited tools for validation and verification [Zeng et al. 2022, Dharanikota et al. 2021, Annenkov et al. 2020, Kaleem et al. 2021, Ferreira et al. 2020, Jiao et al. 2020a]. Understanding contracts by non-technical users also poses significant challenges [Qasse et al. 2021].

Model-Driven Engineering (MDE) has been considered an alternative approach to making smart contracts safer and more reliable [Chirtoaca et al. 2020]. For the

EVM, the High-Level Metamodel for Smart Contract (HLM-SC) has been proposed as a metamodel that enables the high-level declaration of elements in a contract, including complex structures, state variables, functions, and similar elements [Velasco 2023] [Velasco and Carvalho 2022]. With HLM-SC, both technical and non-technical developers can create contracts more efficiently, regardless of their previous knowledge in modeling.

This paper emphasizes the importance of evaluating the quality of HLM-SC using objective metrics, exploring its conceptual validity, and the relevance of a usage guide for the metamodel. External evaluation of HLM-SC is crucial, as it allows external experts to objectively and impartially analyze and validate the metamodel, leading to potential improvements in the metamodel and its practical application. Additionally, a usage guide can assist developers in better understanding the metamodel and using it appropriately, which can increase its adoption among developers. The objective of this paper is to present the evaluation of HLM-SC using the MQuaRE framework [Kudo et al. 2020a], along with a usage guide to its application in a scenario related to the NFT (Non-Fungible Tokens) industry.

The paper is organized as follows: Section 2 presents the concepts of smart contracts, MDE, and the tool used to evaluate the quality of the proposed metamodel (MQuaRE). Section 3 discusses related works. Section 4 provides details on HLM-SC, including its usage guide. In Section 5, an application in a use case related to the NFT industry is presented. In Section 6, the results of the quality evaluation of HLM-SC are presented. Finally, Section 7 offers concluding remarks.

2. Background

2.1. Smart Contracts

Smart contracts are computer protocols that can be implemented on blockchains, allowing parties to establish decentralized and trustworthy agreements [Buterin et al. 2014, Rajasekaran et al. 2022, Wang et al. 2019]. When the conditions of the agreement are met, these contracts are automatically executed without the need for a trusted third party [Buterin et al. 2014, Rajasekaran et al. 2022, Wang et al. 2019]. The immutability of smart contracts after deployment is a critical aspect, as it ensures the transparency and traceability of transactions [Cai et al. 2018].

Ethereum is a blockchain platform that pioneered the implementation of smart contracts, allowing for their writing in Solidity or Vyper programming languages [Buterin et al. 2014]. Other blockchain platforms, such as Tron, Cardano, and Polygon, also support smart contracts and are compatible with the Ethereum Virtual Machine (EVM), which enables code reuse across them [Grigg 2017]. The EVM is a virtual machine that executes contracts on different blockchain platforms, as long as they support the EVM. This means that smart contracts can be developed in a common language and executed on multiple blockchain platforms without the need to rewrite the code. As a result, interoperability between platforms increases, and efficiency in the development of smart contracts improves.

On the EVM platform, the essential concepts that make up smart contracts include interfaces, abstract contracts, concrete contracts, types, variables, complex data structures,

functions, and events. Interfaces represent the elements that a contract must implement, such as functions, events, or errors. They are critical in contract development because they specify all the necessary elements for the contract and can serve as a communication mechanism between contracts.

Abstract contracts are those that cannot be directly instantiated but require implementation by a concrete contract. They are used in cases where it is necessary to limit the scope, such as in utility contracts that only add functionality for control and security. On the other hand, concrete contracts are effectively implemented on the blockchain through a contract deployment transaction. They contain the business rules of a particular domain and can inherit other contracts, including concrete and abstract contracts or interfaces, allowing for code reuse and the implementation of additional functionalities.

Types are elements that are directly related to the programming language used for smart contract development. Each type of the language abstracts a concept from the real-world business rule, such as numbers, words, participant identification, and complex data structures. Variables are responsible for representing and storing information within the context of a smart contract. They serve as a source of truth for the contract and are used to make decisions and control the internal behavior of the contract. Changes to a state variable must be made by specific functions and controlled within the contract, ensuring the security and reliability of the stored information.

Functions contain and execute the business rules of a smart contract and have the ability to perform various actions, such as transferring monetary values, returning information to external applications, executing specific business rules, and modifying internal contract information. They are used to manage access to resources and have limitations, such as gas limit and execution time, which must be considered during development. Events, on the other hand, are used to notify decentralized applications when specific actions occur within the smart contract. They serve as a means of communication between the contract and external applications within the context of the EVM.

2.2. Model-Driven Engineering

The MDE approach uses models as the primary artifacts for software development and metamodels to guide their creation [Seidewitz 2003][Schmidt 2006]. Metamodels are formal descriptions of models that define their elements, structure, constraints, and semantics [Rodrigues da Silva 2015, Seidewitz 2003]. To create a metamodel, a meta-metamodel that describes the modeling language is required, and the MetaObject Facility (MOF) is the OMG standard specification for defining metamodels [Seidewitz 2003].

The MOF is structured as a four-layer architecture, with each lower element being an instance of a higher element. The layers are the M0 layer (models), the M1 layer (metamodels), the M2 layer (meta-metamodels), and the M3 layer (meta-meta-metamodels). Ecore, developed by the Eclipse Foundation, is an implementation of MOF and a widely used alternative for creating metamodels. The metamodel evaluated in this work is an instance of Ecore.

2.3. MQuaRE framework

MQuaRE (Metamodel Quality Requirements and Evaluation), proposed by [Kudo et al. 2020a], is an important tool for evaluating the quality of a metamodel

in various application areas. To ensure the quality of the metamodel, MQuaRE is divided into five characteristics: compliance, conceptual suitability, usability, maintainability, and portability. Each of these characteristics is further subdivided into sub-characteristics, which serve as a guide for evaluating the quality of the metamodel in all relevant aspects.

MQuaRE defines 19 MQRs (Metamodel Quality Requirements) and 23 quality measures associated with the mentioned characteristics, which should be considered during metamodel evaluation. To perform the evaluation, the framework suggests a process consisting of five phases: (1) establish the evaluation requirements, (2) specify the evaluation, (3) design the evaluation, (4) execute the evaluation, and (5) conclusion of the evaluation. Each phase is essential to ensure that the evaluation is conducted consistently and effectively.

3. Related Work

Several authors have employed techniques to model contracts at a high level based on MDE approach. However, in some cases, the quality of the presented tools has not been evaluated, with some validations based only on use cases. In this section, we present related works that address contract modeling on blockchain platforms and metamodel evaluation, highlighting their contributions and limitations.

The work of [Hamdaqa et al. 2020] proposed IContractML for contract modeling on blockchain platforms. In [Qasse et al. 2021], the authors used IContractML in conjunction with iContractBot for contract modeling using a chatbot. The work [Velasco and Carvalho 2022], in turn, proposed a metamodel for contract development on the EVM. However, in some cases, the quality of these tools has not been evaluated.

In the context of metamodel evaluation, the works of Kudo *et al.* in [Kudo et al. 2020a] and [Kudo et al. 2022] employ the MQuaRe framework, proposed by the same authors, to evaluate metamodels. Although these works are not directly related to contract development, they were important as a basis for applying the metamodel evaluation process proposed in this work.

This work differs from related works by using a quality evaluation method for metamodels, specifically for the metamodel proposed by [Velasco and Carvalho 2022]. The evaluation method used was MQuaRe, which evaluated the conceptual suitability of the metamodel in relation to the contract development concepts presented in this paper.

4. HLM-SC Metamodel

The HLM-SC (High-Level Metamodel for Smart Contract) incorporates essential parts of a contract, such as state variables, functions, constructors, complex data structures, events, errors, and modifiers. Comprising nine metaclasses, this metamodel presents a complete and well-structured approach to smart contract development.

4.1. The Metamodel

Figure 1 presents the HLM-SC Metamodel. The primary metaclass of the HLM-SC metamodel is *Contract*, which instantiates other metaclasses and models the smart contract. Although HLM-SC was originally designed for concrete contracts and not for abstract or interface contracts, adaptations can be made to represent them. Concrete smart contracts

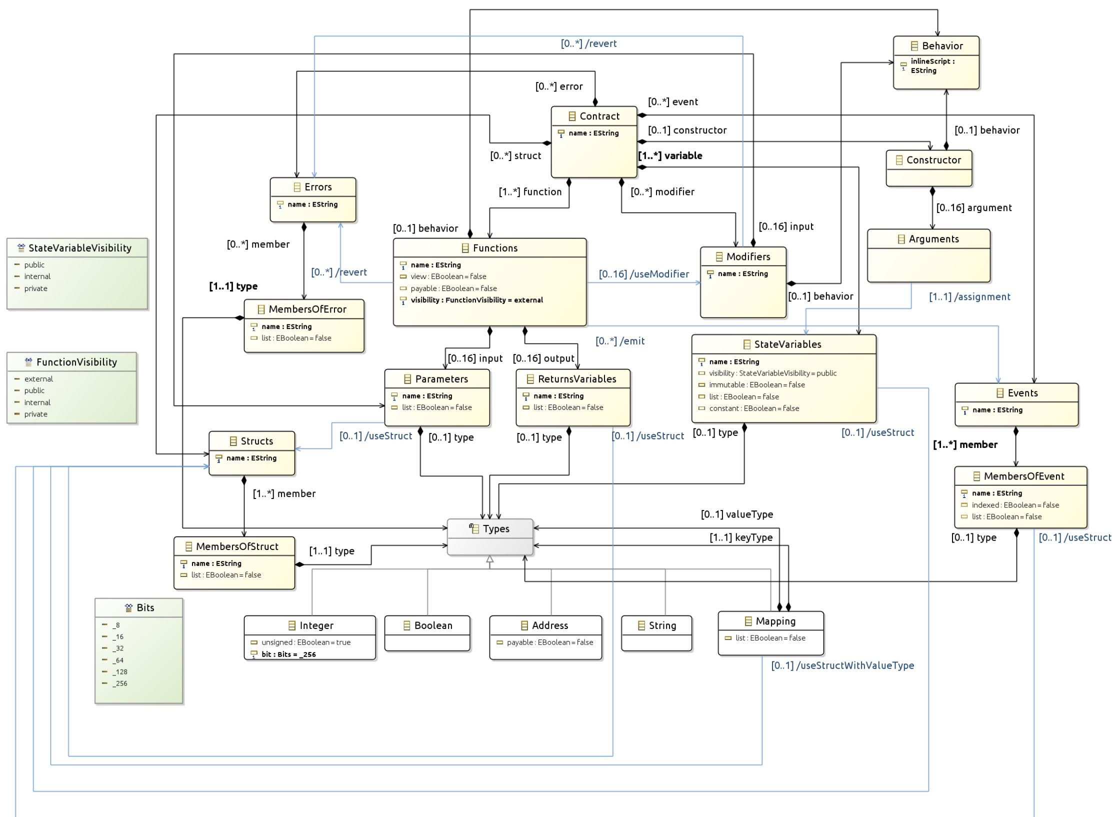

Figure 1. HLM-SC Metamodel.

are where the business logic is implemented during the development stage. The abstract metaclass *Types* represents the types supported in the Solidity language metamodel, which are used in other metaclasses through composition, including *Integer*, *Boolean*, *Address*, *String*, and *Mapping*.

The *Structs* metaclass represents more complex structures declared at the smart contract level, where each member is defined by composition with the abstract class *Types*. Structs can be used as parameters in functions, reducing the number of required inputs, and can also be used in function returns. The *State Variables* metaclass represents state variables in the smart contract, where their values are permanently stored and can be modified through RPC calls. Variables can have internal, public, or private visibility and are composed of a name, visibility, and type. The type can be a basic type, a key-value structure like *mapping*, or a data structure (*struct*).

The *Errors* and *Events* metaclasses in the metamodel have the purpose of enabling traceability and monitoring of activities in a smart contract. The *Errors* metaclass allows for the generation of more semantic and precise errors during the execution of an RPC call. On the other hand, the *Events* metaclass records a state change in a structured way within a smart contract. The event data is stored in the transaction that originated it and can be consumed by DApps (decentralized applications). However, a smart contract cannot directly access information emitted by events from other contracts.

The *Modifiers* and *Functions* are two important metaclasses in HLM-SC for build-

ing a smart contract. Modifiers are used as a special kind of function that can be included in the code of another function to control access or prevent reentrancy attacks. In the metamodel, a piece of code can be added and incorporated into the modifier through composition with the *Behavior* metaclass.

Functions, in turn, are responsible for checking conditions and performing necessary actions. They can be of two types: *view*, which do not change the contract state and have zero cost for the call, and *modifying*, which change the contract state and have a cost involved. In the metamodel, functions can receive input parameters and return a sequence of variables, in addition to having public, internal, or private visibility. Functions can also have behavior added to them through composition with the *Behavior* metaclass.

The *Constructor* is a metaclass that enables the initialization of state variables when the smart contract is instantiated. It is similar to a constructor of a class in object-oriented modeling. In the metamodel, input arguments can be added to the constructor, and all of them must be used to initialize a state variable. Additionally, behavior can be added to the constructor through composition with the *Behavior* metaclass.

4.2. Usage Guide

This subsection presents a guide to assist in smart contract modeling on the EVM using the HLM-SC. The process of contract modeling using the HLM-SC starts with the instantiation of the *Contract* metaclass and the definition of a symbolic name, which is used as the contract identifier. If the contract presents complex data structures, it becomes necessary to define these structures using the *Struct* metaclass, as they can be used by other elements such as state variables, function parameters, and events.

State variables are defined in sequence using the *StateVariables* metaclass. Variables with internal or private visibility must be declared starting with _ (underscore), while public variables must start with letters. Variable names should be descriptive of their purpose, especially in the case of public variables since *view* functions are automatically created for them. Additionally, each variable must have a unique name in the context of the contract, *i.e.,* it cannot have the same name as other elements such as functions, events, errors, and others. During this step, it is essential to assign a type to each variable through the relationship with the *Types* metaclass. It is possible to use the structures defined in the previous step as a type.

If there is a need to initialize state variables during the contract deployment, it is recommended to declare the *Constructor*. It is important to note that the constructor is capable of initializing only state variables, and state variables of type *mapping* cannot be automatically initialized by it.

Errors are declared in sequence using the *Errors* metaclass. It is recommended that the names and arguments of each error be semantic and appropriate to the context of the error, as errors are a mechanism for DApps to understand the reason for a function failure. It is important to emphasize that defining effective errors contributes to the improvement of contract comprehension as well as its usability.

When there is a need to trace interactions and modifications with the contract, the *Events* metaclass is used, which must be defined in sequence. It is important that each event is related to its context. Additionally, it is recommended that when there is a need

to use the EVM's filtering mechanism, indexed events should be declared first to ensure a more efficient filtering process.

The modifiers metaclass must be declared before the functions but after defining all state variables that can be used within the modifier's scope. The order is important so that functions can use modifiers correctly. Next, at least one function must be declared using the *Functions* metaclass. It is important to emphasize that functions can use the errors, events, and modifiers previously defined in the contract, allowing for greater flexibility in their implementation.

The careful definition of each of these elements is essential to ensure that the contract is efficient and easy to understand. The sequence of the eight steps described in this paper can facilitate the modeling of the contract using the HLM-SC metamodel, ensuring that all relevant elements are considered and that the contract meets the user's needs.

5. Use Case Scenario

This section describes a use case scenario of a company that aims to offer a system for buying and selling NFTs, applying the usage guide of the proposed metamodel. The objective of this use case is to provide an effective way for users to put their NFTs up for sale. The NFTs can be traded using the native currency of the blockchain and can also be exchanged for other fungible tokens (ERC-20)[1]. To serve users who release collectibles, the use case aims to provide an efficient way to list multiple NFTs from the same contract with the same settings. The user who places an NFT sell order should be able to update or cancel the order. Due to cost and blockchain interaction concerns, it is desired to have only one contract that serves the sale of NFTs from different contracts and users. It will only be possible to purchase a single NFT per transaction. The payment method can be chosen exclusively through the native or ERC-20 currency. For each token sold, a service fee will be charged and transferred to an address owned by the use case. Additionally, when there are royalties on the NFT during its sale on the secondary market, it is required that the creator's percentage be passed on. The elements necessary to build the *Purchase* contract (Figure 2) according to the requirements of the scenario are described below.

NFTs are listed for sale with support for both native currency and ERC-20 tokens through the *Order* structure. When trading with ERC-20 tokens, it is not possible to send the quantity of tokens in the transaction content. Therefore, there is a need for a new *Currency* structure. In the HLM-SC metamodel, a *Struct* does not relate to other *Structs*. Therefore, a state variable is used to represent ERC-20 prices (*_orderToPrices*).

As the use case must have only one sales contract, control is done through a state variable that represents open orders. For this purpose, two state variables are created: *nextOrderId*, and *_orderIdToOrder*. To represent the sold tokens, *_orderIdToTokensSold* is used, which is a *mapping*. In addition to these variables, two more are created to represent the service charged by the platform: *marketplaceFee* and *marketplaceAddress*. When a token has been previously sold by the marketplace, the *_nftSoldPrimaryMarket* variable is updated, which is used to transfer royalties in the secondary market.

[1]ERC-20 tokens are contracts with an unlimited number of tokens pegged to other currencies such as dollars, euros, and ether.

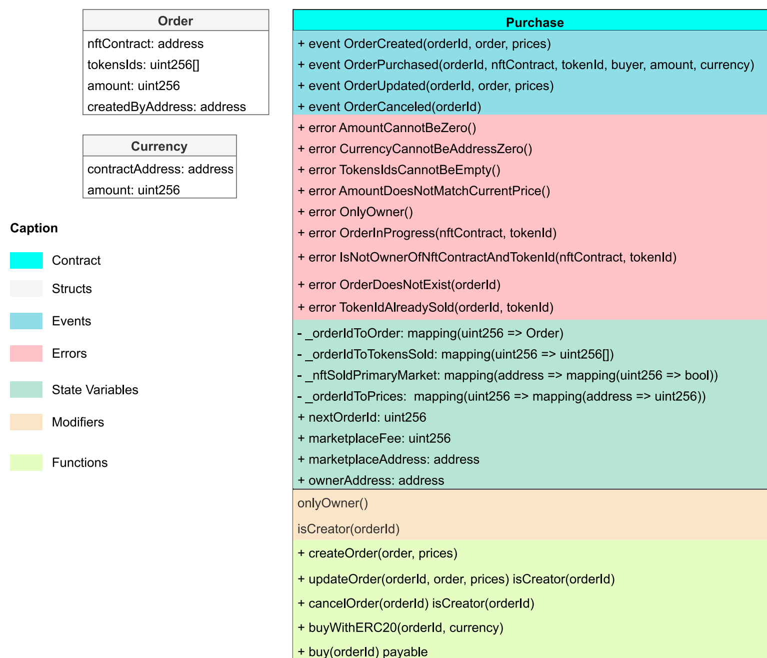

Figure 2. Illustrate the structs, events, errors, variables, modifiers, and functions of the *Purchase* contract.

A new order is created through the *createOrder* function, which receives the *Order* and a list of *Currency* as inputs. All parameters are validated, and in case of success, the *OrderCreated* event is triggered with the assigned order identifier. When there is an inconsistency, the transaction fails, and the order is not created. The function can fail with errors such as *AmountCannotBeZero*, *CurrencyCannotBeAddressZero*, *TokensIdsCannotBeEmpty*, and *IsNotOwnerOfNftContractAndTokenId*.

After creation, the order can be modified or canceled by the creator. Given this context, there is the *isCreator* modifier that validates permissions to execute the action. The *updateOrder* function can modify all data of a particular order and triggers the *OrderUpdated* event. In case of cancellation, the *cancelOrder* function is used, which triggers the *OrderCanceled* event. There are other management functions for updating information, and each function has an associated event. To validate if the caller has permission to make changes, the *onlyOwner* modifier is used.

For purchasing with native currency or with ERC-20 tokens, two new functions are used. The purchase with native currency, the *buy* function is used, which is a *payable* function with *orderId* as the input. In the other case, the *buyWithERC20* function has the *orderId* and *currency* as inputs. In both cases, if the caller meets all requirements, the *OrderPurchased* event is triggered. In case of any failure, an error is used, such as *TokenIdAlreadySold*, *OrderDoesNotExist*, and *AmountDoesNotMatchCurrentPrice*. Finally, when the contract is implemented, the service fee and receiving wallet address are initialized. Additionally, the *ownerAddress* variable receives the address of the responsible party for the transaction.

6. Quality evaluation

The evaluation method used in this study is an adaptation of MQuaRE [Kudo et al. 2020a]. In Subsection 6.1, the evaluation planning is presented according to the MQuaRE usage guide. In Subsection 6.2, the execution process is presented. In Subsection 6.3, the evaluation results are presented for each evaluator, as well as the profile of each evaluator and the evaluation discussions according to the purpose. Finally, in Subsection 6.4, threats to validity are presented.

6.1. Planning

The purpose of the HLM-SC evaluation is to identify improvement points and verify the acceptance of the metamodel. In this scope, the conceptual completeness and conceptual correctness subcharacteristics were selected. These were chosen due to their relationship with the research objectives, allowing for the evaluation of whether the metamodel contains the contract concepts correctly and completely. Table 1 presents the selected Metamodel Quality Requirements (MQR), including the relationship between the MQRs, selected characteristics and subcharacteristics, and measures.

Quality Requirements	Characteristics	Subcharacteristics	Measures
MQR02 - The metamodel must cover the concepts found in its specifications.	Conceptual Suitability	Conceptual Completeness	Conceptual Coverage
MQR03 - The metamodel must represent the concepts found in its specifications correctly.	Conceptual Suitability	Conceptual Correctness	Conceptual correcteness

Table 1. Relation between quality requirements, measures, characteristics and subcharacteristics selected. Adapted from Kudo *et al.* [Kudo et al. 2020b].

To assist in the evaluation of the selected MQRs, the MQuaRE tool presents artifacts that should be used in this process. According to the scope of this evaluation, the necessary artifacts are the metamodel specification and the metamodel implementation. The following is a detailed description of each artifact used for the evaluation:

- Metamodel specification: It includes concepts related to contract development. The fundamental concepts include contracts (C1), interfaces (C2), abstract (C3) and concrete (C4) contracts, types (C5), numbers (C6), string (C7), participant identification (C8), complex data structures (C9), variables (C10), functions (C11), and events (C12).
- Metamodel implementation: It includes an explanatory guide of the metaclasses, relations, and attributes of the metamodel represented in Ecore format in both text and video formats. Additionally, to facilitate the visualization of the metamodel, a complementary video was made available that presents a visual interpretation of the metamodel, especially for those who are not familiar with MDE concepts and their interrelationships.

In addition to the presented artifacts, external evaluators were provided with complementary video materials to assist in their understanding. These include an explanation of the structure of the Solidity language. This measure was taken considering the evaluators lack of familiarity with the Solidity language, and it aims to ensure that they have the necessary knowledge to understand the metamodel's implementation.

MQR	Measures	Description of the measure	Measurement function
MQR02	CCp-1 - Conceptual coverage	What proportion of the specified concepts has been modeled?	X = 1 - A / B A = Number of missing concepts. B = Number of concepts described in the metamodel specifications. 0 <= X <= 1. The closer to 1, the more complete.
MQR03	CCr-1 - Conceptual correctness	What proportion of metamodel concepts are modeled correctly?	X = 1 - A / B A = Number of incorrectly modeled concepts incorrectly. B = Number of concepts considered in the evaluation. 0 <= X <= 1. The closer to 1, the more correct.

Table 2. Quality measures, description, measurement, interpretation. Adapted from [Kudo et al. 2020b].

The selection of quality measures was carried out according to the previously selected MQRs. Table 2 presents the quality measures, their descriptions, the measurement functions, and the interpretation of the measured value. The target value for the measures is 1, and the acceptable tolerance value is 0.8. The target value of 1 indicates that the software artifact fully satisfies the corresponding quality measure, while the acceptable tolerance value of 0.8 allows for some level of imperfection while still considering the artifact to be of acceptable quality.

Finally, the decision criteria for the evaluation and the formulas used to calculate the scores of the characteristics and subcharacteristics are defined. For each subcharacteristic, the measurement function suggested by the authors of MQuaRE was used.

6.2. Execution

After planning the evaluation, the selection of external evaluators was carried out based on specific criteria, taking into account that they should have knowledge in programming or blockchain. This choice was motivated by the fact that the metamodel, in its current state, requires technical knowledge for its use. Therefore, the selection of evaluators with these skills was strategic, as the research objective is to facilitate contract development for software developers who possess such technical knowledge.

A Google Forms[2] questionnaire was created to collect evaluation information. In addition to questions related to MQRs, the form included sections to collect information about the evaluators profile and level of knowledge about the topic. The presentation of the material and the evaluation dynamics were presented to the evaluators according to their availability. The preference was through video calls, but in some cases, it was only done through text. This process began in early February 2023, and the phase of collecting responses from external evaluators lasted for three weeks.

6.3. Results and discussion

The evaluation process involved 11 evaluators, all with previous programming experience, with an average of over 5 years of experience (82%). The evaluators familiarity

[2]Google Forms is a Google application that allows the creation of dynamic forms. For more information, please visit: `https://docs.google.com/forms/`.

with blockchain concepts was significant (73%), however, their knowledge that went beyond basic (intermediate or advanced) was lower (37%). The vast majority of evaluators (73%) had no prior knowledge of the Solidity language, with only one evaluator having intermediate knowledge in this language. Only a portion of the evaluators knew the MDE approach (45.5%), and only one of them had intermediate knowledge in the area.

Regarding the evaluation of the specified concepts (MQR02), some evaluators (45.5%) stated that the concept of abstract contracts (C3) was not modeled. Additionally, other evaluators (27.3%) indicated that the concept of interfaces (C2) was not applied in the modeling. Only one evaluator considered that the contract concept (C1) was not applied, but this same evaluator considered that the concepts of interfaces (C2), abstract contracts (C3), and concrete contracts (C4) were modeled. Furthermore, one evaluator did not consider the application of the participant identification concept (C7), while another evaluator did not consider the application of the complex data structures concept (C8).

The results of the evaluation of the modeled concepts (MQR03) showed differences between the evaluators. According to one evaluator (E2), the interface concept (C2) was correctly modeled, while in the previous question, they considered that it was not applied. Only this evaluator (E2) considered that the concept of concrete contracts (C4) was not correctly modeled. The other concepts pointed out as not correctly modeled are consistent with the quality measure MQR02, which already indicated the absence of application in the metamodel.

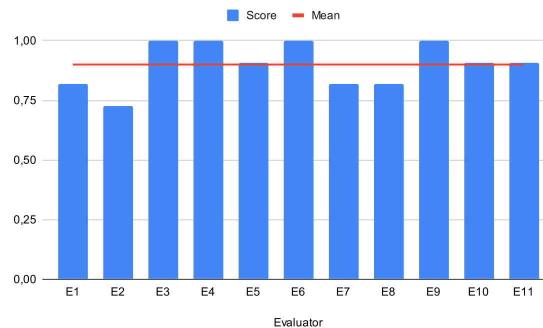

Figure 3. Distribution of evaluators scores for quality requirements MQR02 and MQR03.

According to the evaluation's objective (identification of improvement points), the evaluators pointed out some aspects that could be improved in the metamodel. One evaluator (E7) highlighted the difficulty in understanding the distinction between the contract and the concrete contract in the representation proposed by the metamodel. Therefore, it is essential to provide a clearer explanation of this aspect in the support materials to facilitate a better understanding of the metamodel's usage.

In addition, other evaluators (E4 and E9) identified the need for the provision of complementary materials that demonstrate the application of the metamodel, such as use cases. However, these materials were not included in the evaluation, considering the established objective and the necessary requirements provided by the used framework. It is essential to consider these suggestions for future improvements and updates of the metamodel to enhance its applicability and usefulness for contract developers. The pro-

vision of additional materials, such as use cases, can facilitate the understanding of the metamodel's usage and its potential benefits for the development of smart contracts.

Figure 3 shows the distribution of scores assigned by evaluators for quality requirements MQR02 and MQR03. The information includes the mean (0.908) of all scores. It was observed that the scores were similar for both quality requirements, which was expected. Considering that no weight was assigned to the concepts and that the application of the concepts in the modeling could result in correct modeling, the scores obtained indicate an overall acceptable level of quality for the metamodel.

The score obtained for MQR02 and MQR03 indicates conformity between the metamodel and the completeness and correctness of the incorporated concepts. However, it is important to note that there is still room for future improvements. It is important to emphasize that the evaluation of concepts is subjective and may vary according to the evaluators views and experience, which can influence their assessment.

Finally, based on the experimentation phase of the evaluation and the final score obtained by the metamodel, there are indications of acceptance of the metamodel. However, it is essential to monitor and improve the metamodel over time to ensure its effectiveness and efficiency. It is also important to consider the suggestions and feedback from the evaluators to make improvements and updates to the metamodel, in order to make it increasingly useful and applicable to contract developers. Continuous evaluation and improvement are necessary to maintain the metamodel's relevance and usefulness for the development of smart contracts, especially considering the rapid evolution and changes in the blockchain technology field.

6.4. Threats to validity

Threats to validity refer to factors that may compromise the accuracy of the results obtained in research. Regarding this evaluation, it is important to consider possible issues that may affect the validity of the results.

The sample used in the evaluation was not stratified, resulting in an unbalanced distribution between evaluators who have and do not have experience in contract development. Such an imbalance may impact the results of the evaluation, as each group may have a distinct perspective on the concepts incorporated into the metamodel.

Additionally, it is essential to implement a way of measuring to evaluate the weight of each specified concept that has been incorporated into the metamodel. Currently, the technique used is summation, but it does not ensure that any concept may be considered more important than others. This lack of weighting can result in undervaluation or overvaluation of the modeled concepts.

7. Conclusion

This paper evaluated the HLM-SC, a metamodel for modeling smart contracts based on the EVM platform. The HLM-SC is an abstraction that allows for the declaration of essential elements for building a contract, making it a primary artifact for contract modeling on the EVM platform. Additionally, a usage guide was proposed to facilitate the adoption of the HLM-SC by developers. The metamodel was evaluated based on quality criteria established by the evaluation method used, and the results showed indications of acceptance of the metamodel. However, further studies are needed to confirm its effectiveness

and efficiency in other contexts and to constantly monitor the metamodel to ensure its relevance and adequacy for new demands and decentralized applications.

References

Angelis, J. and Ribeiro da Silva, E. (2019). Blockchain adoption: A value driver perspective. *Business Horizons*, 62(3):307–314.

Annenkov, D., Nielsen, J. B., and Spitters, B. (2020). Concert: A smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN*, page 215–228. Association for Computing Machinery.

Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1.

Cai, W., Wang, Z., Ernst, J. B., Hong, Z., Feng, C., and Leung, V. C. M. (2018). Decentralized applications: The blockchain-empowered software system. *IEEE Access*, 6:53019–53033.

Chirtoaca, D., Ellul, J., and Azzopardi, G. (2020). A framework for creating deployable smart contracts for non-fungible tokens on the ethereum blockchain. In *2020 IEEE DAPPS*, pages 100–105.

Dharanikota, S., Mukherjee, S., Bhardwaj, C., Rastogi, A., and Lal, A. (2021). Celestial: A smart contracts verification framework. In *2021 FMCAD*, pages 133–142.

Feist, J., Grieco, G., and Groce, A. (2019). Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd WETSEB*, pages 8–15.

Ferreira, J. F., Cruz, P., Durieux, T., and Abreu, R. (2020). Smartbugs: A framework to analyze solidity smart contracts. In *2020 35th IEEE/ACM ASE*, pages 1349–1352.

Garamvölgyi, P., Kocsis, I., Gehl, B., and Klenik, A. (2018). Towards model-driven engineering of smart contracts for cyber-physical systems. In *2018 48th Annual IEEE/IFIP DSN-W*, pages 134–139.

Grigg, I. (2017). Eos-an introduction. *White paper. https://whitepaperdatabase. com/eos-whitepaper*.

Hamdaqa, M., Metz, L. A. P., and Qasse, I. (2020). IContractML: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. In *Proceedings of the 12th SAM*, page 34–43. Association for Computing Machinery.

Jiao, J., Lin, S.-W., and Sun, J. (2020a). A generalized formal semantic framework for smart contracts. In *Fundamental Approaches to Software Engineering: 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, page 75–96, Berlin, Heidelberg. Springer-Verlag.

Jiao, J., Lin, S.-W., and Sun, J. (2020b). A generalized formal semantic framework for smart contracts. In *Fundamental Approaches to Software Engineering: 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, page 75–96, Berlin, Heidelberg. Springer-Verlag.

Jurgelaitis, M., čeponienė, L., and Butkienė, R. (2022). Solidity code generation from uml state machines in model-driven smart contract development. *IEEE Access*, 10:33465–33481.

Kaleem, M., Kasichainula, K., Karanjai, R., Xu, L., Gao, Z., Chen, L., and Shi, W. (2021). An event driven framework for smart contract execution. DEBS '21, page 78–89, New York, NY, USA. Association for Computing Machinery.

Khan, A. G., Zahid, A. H., Hussain, M., Farooq, M., Riaz, U., and Alam, T. M. (2019). A journey of web and blockchain towards the industry 4.0: An overview. In *2019 International Conference on Innovative Computing (ICIC)*, pages 1–7.

Kudo, T. N., Bulcão Neto, R. F., and Vincenzi, A. M. R. (2020a). Toward a metamodel quality evaluation framework: Requirements, model, measures, and process. In *Proceedings of the XXXIV SBES*, page 102–107. Association for Computing Machinery.

Kudo, T. N., Bulcão-Neto, R. d. F., Neto, V. V. G., and Vincenzi, A. M. R. (2022). Aligning requirements and testing through metamodeling and patterns: design and evaluation. *Requirements Engineering*, pages 1–19.

Kudo, T. N., Bulcão-Neto, R. F., and Vincenzi, A. M. R. (2020b). Metamodel quality requirements and evaluation (mquare). *arXiv preprint arXiv:2008.09459*.

Li, Z., Zhou, Y., Guo, S., and Xiao, B. (2021). Solsaviour: A defending framework for deployed defective smart contracts. In *Annual ACSAC*, page 748–760, New York, NY, USA. Association for Computing Machinery.

Qasse, I., Mishra, S., and Hamdaqa, M. (2021). iContractBot: A chatbot for smart contracts' specification and code generation. In *2021 IEEE/ACM 3rd BotSE*, pages 35–38.

Rajasekaran, A. S., Azees, M., and Al-Turjman, F. (2022). A comprehensive survey on blockchain technology. *Sustainable Energy Technologies and Assessments*, 52:102039.

Rodrigues da Silva, A. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems Structures*, 43:139–155.

Schmidt, D. C. (2006). Guest editorx27;s introduction: Model-driven engineering. *Computer*, 39(02):25–31.

Seidewitz, E. (2003). What models mean. *IEEE Software*, 20(5):26–32.

Velasco, G. (2023). A model-driven approach for smart contract development (in-progress). In *Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás*, Goiânia, GO, Brazil.

Velasco, G. and Carvalho, S. (2022). A model-driven approach to developing smart contracts on the ethereum virtual machine *(in portuguese)*. In *Proceedings of the X ERI-GO*, pages 106–117. SBC.

Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., and Wang, F.-Y. (2019). Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277.

Zeng, Q., He, J., Zhao, G., Li, S., Yang, J., Tang, H., and Luo, H. (2022). Ethergis: A vulnerability detection framework for ethereum smart contracts based on graph learning features. In *2022 IEEE 46th COMPSAC*, pages 1742–1749.