A High-Level Metamodel for Developing Smart Contracts on the Ethereum Virtual Machine

Gislainy Velasco[1], **Noeli Antonia Vaz**[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 Acadêmico de Ciências Exatas e Tecnológicas – Universidade Estadual de Goiás (UEG)
Caixa Postal 459 — 75132-903 — Anápolis — GO — Brasil

{gislainycrisostomo, noelivaz}@discente.ufg.br, sergiocarvalho@ufg.br

Abstract. *The development of smart contracts presents significant challenges compared to traditional software development, such as the immutability of the blockchain and the limitation of program size. These challenges can lead to human errors and the existence of vulnerabilities that may be exploited by malicious individuals, resulting in substantial financial losses. Contract developers face language and infrastructure constraints and insufficient information on interface patterns and implementation specifications. Existing proposals are often challenging to understand, with complex formal verifications requiring expertise in this approach. This article proposes using Model-Driven Engineering (MDE), employing a metamodel for contract development and code generation for the corresponding platform. The metamodel proposed in this study referred to as the High-Level Metamodel for Smart Contract (HLM-SC), is an abstraction applicable to contract development in various contexts. HLM-SC consists of a set of metaclasses allowing the declaration of essential elements for constructing a contract on the Ethereum Virtual Machine (EVM). A graphical tool has been developed to facilitate contract modeling following HLM-SC specifications. Additionally, the model generated from the tool is transformed into Solidity code. This approach aims to overcome developers' limitations, offering a more understandable and efficient approach to building smart contracts on the blockchain.*

1. Introduction

The emergence of smart contracts has facilitated the development of innovative decentralized applications that exploit attributes such as immutability, traceability, transparency, and privacy [Angelis and Ribeiro da Silva 2019, Khan et al. 2019]. The ability to create these applications is ascribed to blockchain technology, specifically focusing on the Ethereum Virtual Machine (EVM). The EVM facilitates the composition of high-level code instructions and guarantees their self-execution. Consequently, this phenomenon has given rise to novel methodologies in application development, employing contracts and capitalizing on the inherent characteristics of the blockchain.

The immutability inherent in blockchain technology holds significant implications for developing Decentralized Applications (DApps), necessitating software developers to embrace a novel programming paradigm. This challenge arises because any subsequent

modifications are impossible once a contract is deployed. Consequently, this characteristic mandates a more rigorous development process, requiring a distinct approach to the composition, validation, and implementation of applications [Zeng et al. 2022, Feist et al. 2019, Li et al. 2021, Dharanikota et al. 2021, Jiao et al. 2020].

The conception of a DApp requires all parties involved in the process to understand the selected blockchain's data flow and functional requirements. Consequently, contract developers face a series of challenges, making the activity complex and prone to errors for both experienced and novice developers alike [Jurgelaitis et al. 2022a, Ferreira et al. 2020]. Additionally, the steep learning curve can pose a significant barrier to onboarding new developers in the field [Garamvölgyi et al. 2018a, Jiao et al. 2020].

In addition to requiring developers to possess knowledge for creating a DApp, the EVM platform also necessitates understanding its state machine and the high-level language used for contract writing. However, commonly used programming languages, such as Solidity, have deficiencies that lead developers, especially beginners, to make errors [Garamvölgyi et al. 2018a]. According to Dharanikota et al. [Dharanikota et al. 2021], the language's semantics are obscure and only partially understood, even by experienced programmers. Understanding by non-technical users is a challenging task [Qasse et al. 2021]. The linkage between elements and available resources is not easily understandable, as there is no clear presentation of functionalities and their relationships in a visually accessible manner.

The development of smart contracts presents several challenges, from conception to implementation. Contract modeling, for instance, remains a constant concern in the field of contract development [Santiago et al. 2021, Chirtoaca et al. 2020, Guida and Daniel 2019, Hamdaqa et al. 2020, Ben Slama Souei et al. 2021]. To minimize human errors in creation, various authors have emphasized the importance of employing software engineering techniques, such as Model-Driven Engineering (MDE), as an alternative to aid in the contract development process [Ait Hsain et al. 2021]. MDE is a software development methodology that utilizes models as primary artifacts to represent system requirements, behaviors, and functionalities [Seidewitz 2003]. This approach enables technical and non-technical professionals to share knowledge and communicate more efficiently during software development. Models serve as abstractions of the system under study and can be employed to guide the specification, design, analysis, and implementation of software [Rodrigues da Silva 2015, Seidewitz 2003].

In this context, an alternative is the utilization of metamodels to specify the modeling language and guide the creation of models. A metamodel describes a model's elements, structure, constraints, and semantics, ensuring the consistency, correctness, and reusability of generated models. The use of metamodels is evident in some works employing the MDE approach [Kudo et al. 2020, Boubeta-Puig et al. 2021, Hamdaqa et al. 2022]. Employing metamodels allows for the standardization and comprehensibility of developed models and ensures the consistency, accuracy, and reusability of these models. The MDE approach utilizes techniques and tools to transform metamodels into executable code, including code generation, validation, and model transformations [Seidewitz 2003, Iovino et al. 2012]. Additionally, graphical tools can be created to assist in system modeling, enabling users to develop models visually without the need for advanced programming knowledge.

Faced with these challenges, this paper aims to present a solution that utilizes MDE to minimize human errors during the development process, provide a better understanding of artifacts, and enhance communication among stakeholders. A model-driven approach is introduced, enabling developers to create high-level contracts using metamodels and other MDE techniques.

The structure of the paper is organized as follows: in Section 2, related works on the addressed theme are concisely presented. Subsequently, in Section 3, the metamodel is introduced along with an overview, while Section 4 provides detailed presentations of the main metaclasses. Section 5 outlines the transformation flow from the metamodel to the Solidity code. Section 6 introduces the graphical tool for metamodel manipulation. The paper concludes in Section 7, summarizing its contributions and providing suggestions for future work.

2. Related Works

Several authors have employed techniques for modeling contracts at a high level [Ben Slama Souei et al. 2021, Chirtoaca et al. 2020, Guida and Daniel 2019, Hamdaqa et al. 2020, Santiago et al. 2021]. The main identified approaches include Business Process Model and Notation (BPMN) [Corradini et al. 2022], Unified Modeling Language (UML), block programming [Guida and Daniel 2019, Achour et al. 2021, Garamvölgyi et al. 2018b, Jurgelaitis et al. 2022b], and metamodel [Hamdaqa et al. 2020, Ben Slama Souei et al. 2021, Jurgelaitis et al. 2022b].

In the work of Hamdapa et al. [Hamdaqa et al. 2020], IContractML is proposed for contract modeling, accompanied by a graphical tool for manipulation using Sirius. However, the metamodel has some limitations in constructing smart contracts as it is specific to a particular domain. It has a reference model in which it is only possible to model contracts according to the available components (*Participants*, *Assets*, *Transactions*, *Relationship*), and their interactions, which does not allow for the generic modeling of contracts as proposed in this paper. Additionally, this work proposes a tool that enables high-level modeling of smart contracts for the EVM in a generic manner.

Jurgelaitis et al. [Jurgelaitis et al. 2022b] address the development of contracts on Ethereum based on the principles of Model-Driven Architecture (MDA) and UML models. They validated their proposal by implementing three examples of contracts from the Solidity language documentation, in contrast to our work, which utilizes a metamodel to achieve the same goal of producing implementation code for Ethereum contracts.

3. Metamodel

The High-Level Metamodel for Smart Contract (HLM-SC) incorporates essential elements for contract development. The basic structure of a contract in Solidity inspires its modeling. Figure 1 depicts the main metaclasses of HLM-SC, encompassing state variables, functions, constructors, complex data structures, events, errors, and modifiers. The primary class in HLM-SC is named *Contract*, representing a contract as an abstraction of immutable and traceable code.

State variables are crucial elements in contracts as they enable the representation of contextual properties. They are defined within the contract scope and can be initialized

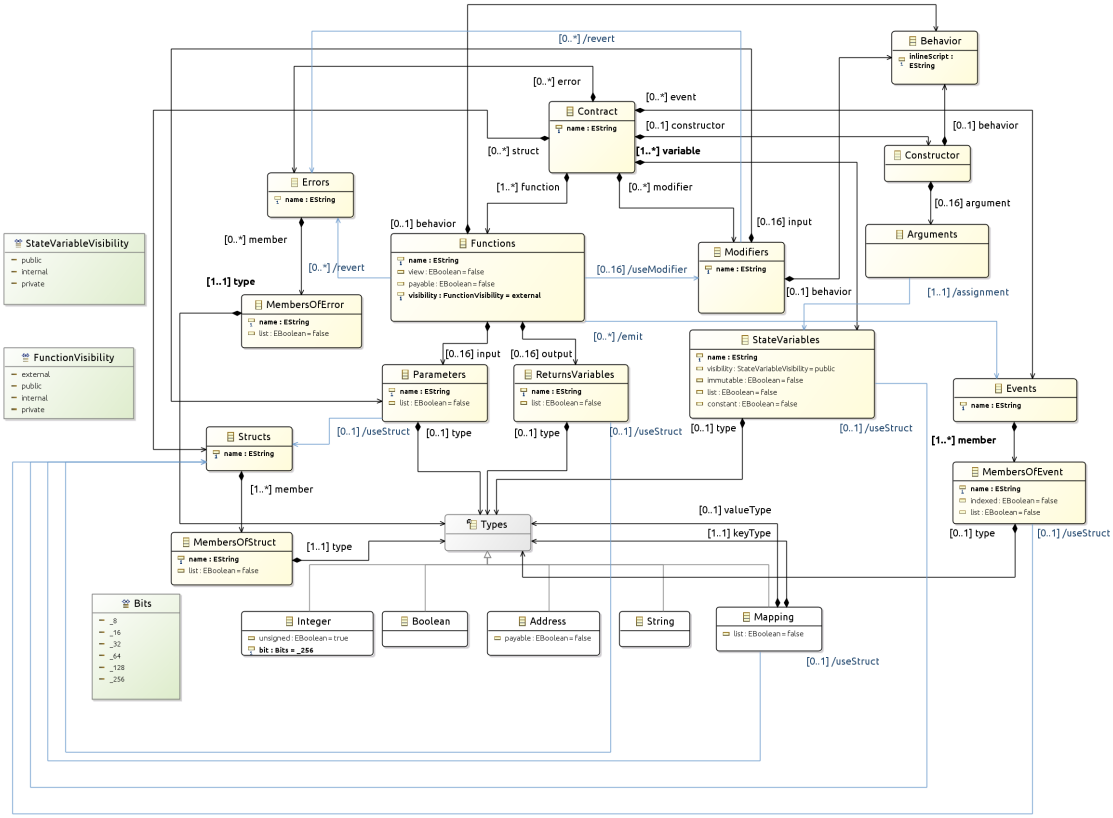

Figure 1. Proposed metamodel for modeling smart contracts in the EVM.

when the contract is instantiated or through specific functions. Variables are declared alongside their basic type and assume an initial value, which varies based on the type. For instance, an integer has a default value of zero. The type of a variable may be associated with a complex structure. Modifications to variables can only be carried out through transactions, ensuring the security and validation of business rules. Depending on their visibility, variable values can be externally visible, allowing verification and validation by external callers.

Functions enable the representation of agreed-upon rules among the involved parties. They can exhibit simple behaviors, such as meeting the conditions defined in state variables (deterministic), or be dependent on received inputs. When a contract rule is fulfilled, notification can be made through events, indicating that a specific action has occurred, allowing interested DApps to execute their rules. These events may have a different structure from state variables and are often employed for auditing purposes. In failure, the function can trigger custom errors defined within the contract scope.

4. The metaclasses of HLM-SC

This section details the metaclasses of HLM-SC (Figure 1). All metaclasses adhere to the following properties: the name must start with a letter or _ (underscore) and should not contain spaces[1]; names must be unique within the scope they are used, and no reserved keywords (*address*, *function*, *int*, *contract*, *modifier*, *struct*, *bool*) should be used.

[1]Names may follow the *Lower Camel Case* convention, for example, lowerCamelCase.

4.1. Types

The abstract metaclass *Types* represents the supported types in the metamodel. These types are used in the composition of other metaclasses. The metamodel supports the following types:

- *Integer*: Represents integers. The *unsigned* property represents only positive numbers, and *bit* represents the variable size, ranging from 8 to 256 bits.
- *Boolean*: Represents true or false values.
- *Address*: A special type representing cryptographic addresses. These addresses can correspond to contract addresses or user account addresses. The *payable* property represents addresses capable of receiving the native cryptocurrency of the blockchain.
- *String*: Used to store literal values. Its content consists only of ASCII characters.
- *Mapping*: A data structure in the form of a table with a key and value. *Mappings* are used for quick lookups. However, it is not possible to iterate over a *Mapping*. In such cases, using an array for that purpose is advisable. All values assume the default value of the used type and can use user-defined types, such as *Structs*. Additionally, they can only be used as state variables and are not supported as function input or output parameters.

4.2. Contract

The main metaclass of HLM-SC is *Contract*, from which other metaclasses are instantiated to model the contract. The contract must have a symbolic name used to identify it during development, particularly in inheritance cases.

The contract represented in HLM-SC is a concrete contract with business logic implemented during the development stage. Abstract contracts or interfaces can also be represented using HLM-SC. However, during the metamodel-to-text transformation stage, the generated code must be adapted to become an abstract contract.

4.3. Structs

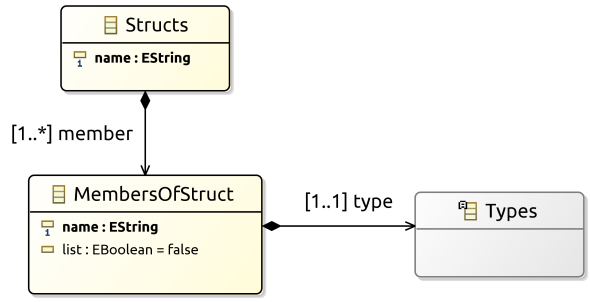

Figure 2. The metaclass representing *Structs* and their relationships.

Structs (Figure 2) are more complex structures declared at the contract level. They can be used as a type in the declaration of a state variable, within the scope of a function, or as an event member. To be considered a *struct*, it must have at least one member, defined through composition with the abstract class *Types*. Each member must have a unique name, defined as a list.

4.4. State Variables

State variables (Figure 3) represent the contract's state, and their values are permanently stored. Immutable or constant variables (represented by the attributes *immutable* and *constant*, respectively) cannot be altered after initialization. On the other hand, other types of variables can be modified after initialization through Remote Procedure Call (RPC) invocations.

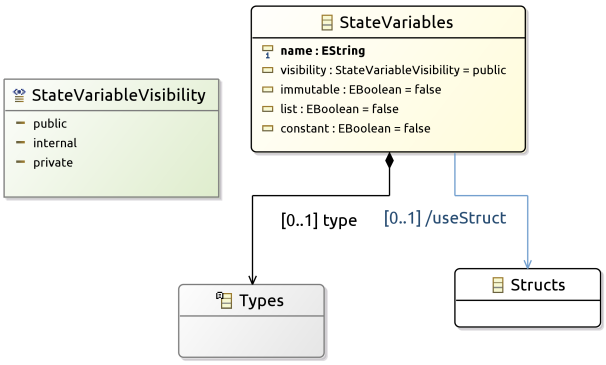

Figure 3. The metaclass representing *State Variables* and their relationships.

The variables within a contract consist of a unique identifier (name), visibility, and type. The variable's visibility can be defined as internal, public, or private. The type can be basic (*integer*, *address*, *string*) or a key-value structure, such as *mapping*, or even a data structure (*struct*) specified within the contract's scope.

One uses composition with the abstract class *Types* to define variable types. Data structures, on the other hand, are specified through a relationship with the *Structs* element. The variable type will be exclusively determined by composition with *Types* or by the relationship with *Structs*, but never both simultaneously. Additionally, it is possible to represent a variable as a list through the *list* property. In the case of *mapping*, only the value can be represented as a list.

4.5. Errors

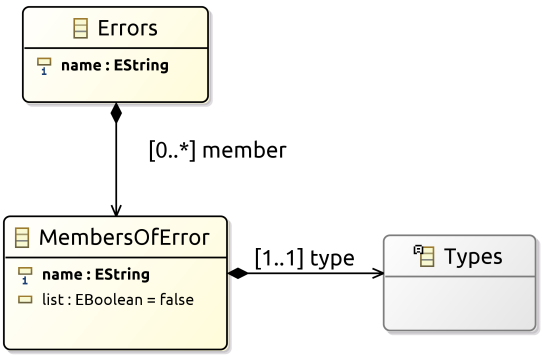

Figure 4. The metaclass representing *Errors* and their relationships.

Errors in contracts (Figure 4) are a more sophisticated way of notifying callers about the occurrence of an action, providing more precise and detailed information rather

than a simple message. Additionally, regarding gas costs, errors are more efficient than messages. In HLM-SC, errors can be employed by both functions and modifiers. When an error occurs during an RPC call, the caller can identify the context of the error.

4.6. Events

Events (Figure 5) record information that can be consumed for traceability purposes. A structured recording of a state change in a contract characterizes an event. An event is defined within the contract scope, with a name and at least one member. An event member must have a name whose type follows the same characteristics as a type in a state variable. Additionally, it is possible to index up to three members of an event, enabling more efficient searches within a contract.

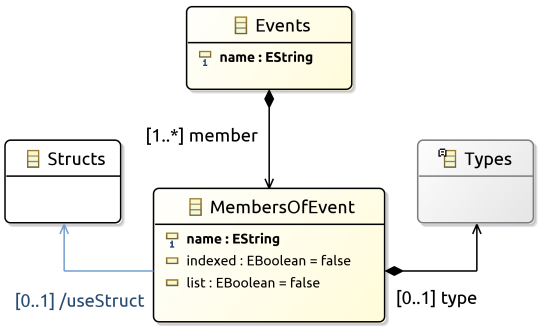

Figure 5. The metaclass representing *Events* and their relationships.

4.7. Modifiers

Modifiers (Figure 6) are used as a special function. Their execution always occurs before the function's logic, and the developer chooses which part of the modifier the function's logic will be embedded through the special character _ (underscore).

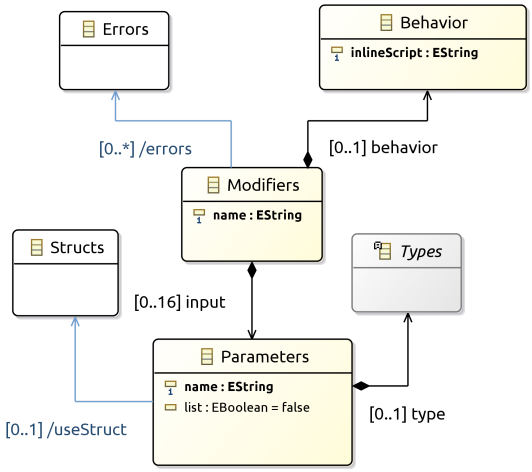

Figure 6. The metaclass representing *Modifiers* and their relationships.

In HLM-SC, adding a code snippet incorporated into the modifier through composition with the metaclass *Behavior* is possible. The code needs to be written in the Solidity language.

4.8. Functions

Functions (Figure 7) are crucial in building a contract, responsible for verifying conditions and executing when true. Otherwise, their execution fails, and the contract state remains unchanged. The execution of a function can fail using customizable errors defined within the contract scope using the *revert* keyword. A successfully executed function can emit various events using the *emit* keyword. In HLM-SC, these two mentioned elements are represented by the derivation relationships *revert* and *emit*, respectively.

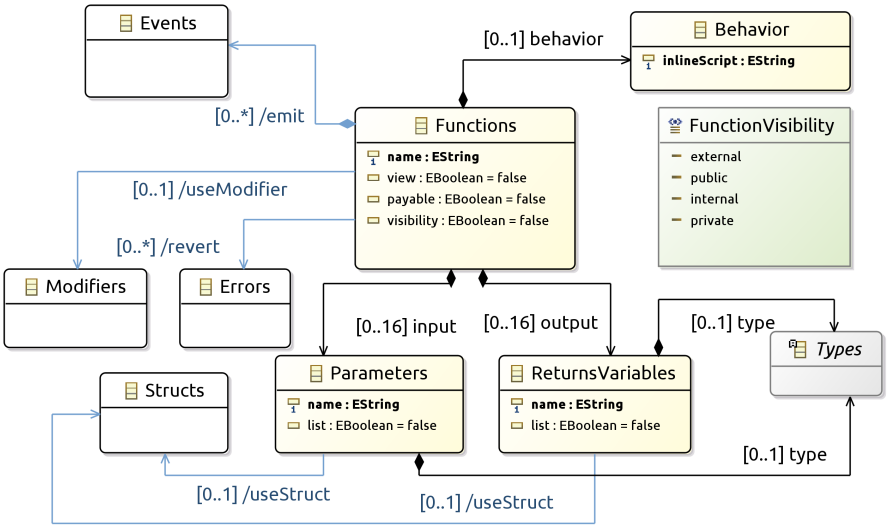

Figure 7. The metaclass representing *Functions* and their relationships.

Contract functions can be divided into two groups: view and mutative. View functions are represented by the *view* attribute. If the *view* attribute is not used, the function is considered mutative and may receive currency transfers, represented by the *payable* attribute.

Functions can receive input parameters and return a sequence of variables. Every input parameter must be used within the function's scope. The visibility of the function can take on external, public, internal, and private visibility scopes. Lastly, functions can add behavior through composition with the *Behavior* class.

4.9. Constructor

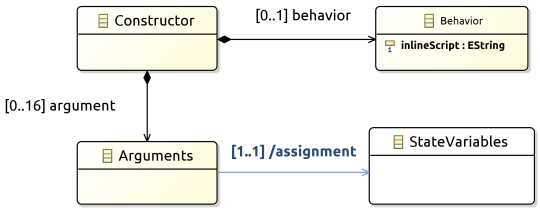

Figure 8. The metaclass representing *Constructor*.

The *Constructor* (Figure 8) is responsible for initializing the state variables of the contract in HLM-SC. Each constructor parameter represents a state variable that will be

initialized with the value passed as an argument. Additionally, behavior can be added when the contract is instantiated through composition with the *Behavior* class.

5. Transformation of the metamodel into Solidity code

The MDE approach enables the transformation of elements, such as metamodels or models, into other artifacts, such as text or other models. This mechanism allows the definition of high-level elements that can be converted into various artifacts. To facilitate this activity, specialized tools, such as *Acceleo*[2], are employed. *Acceleo* is a model-to-text transformation tool that implements the *MOFM2T* specification[3] defined by the Object Management Group (OMG).

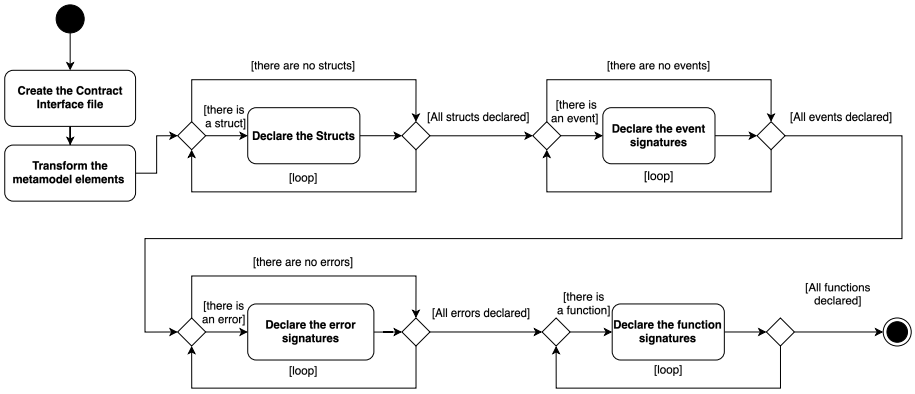

Figure 9. Flowchart for transforming HLM-SC into a Solidity interface.

To illustrate the transformation process, two flowcharts were created. The first flowchart (Figure 9) depicts the creation of the contract interface, while the second one (Figure 10) represents the definition of the contract, encompassing the implementation logic of the contract itself.

In our context, the definition of the contract interface has been added. When a contract interacts with another, it is sufficient to know only the signature. Thus, the code generation process begins with the creation of the interface (Figure 9), which contains the declaration of structures and signatures (functions, events, and errors). By convention, the interface starts with an uppercase 'I', followed by the contract name, indicating it is an interface. For example, if the contract is named *MyContract*, the corresponding interface would be named *IMyContract*.

The Figure 10 illustrates the contract creation process. The flow begins with generating the contract code file, utilizing the previously defined interface. State variables, the contract constructor, modifiers, and functions are declared. State variables are declared according to their type and are not initialized. The constructor, modifiers, and functions are declared with pseudo-implementation. For modifiers and functions, a list of errors that may occur during their execution is specified. Functions also declare the events that can be emitted during their processing.

After this process, two files are made available for implementation on the

[2]`https://www.eclipse.org/acceleo/`
[3]`https://www.omg.org/spec/MOFM2T`

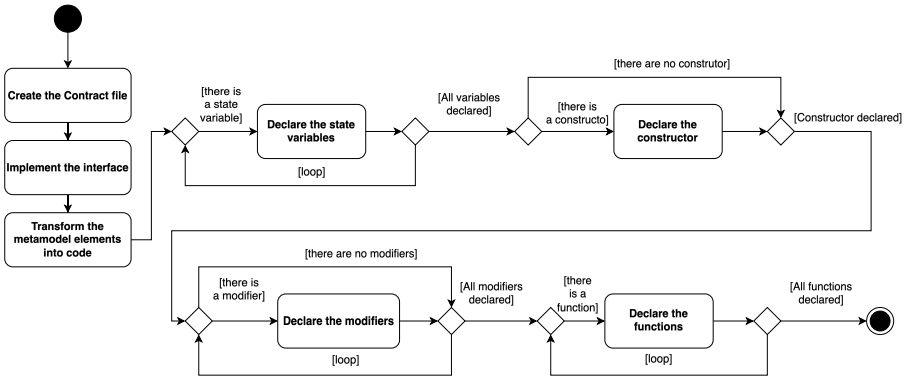

Figure 10. Flowchart for transforming HLM-SC into a Solidity contract.

blockchain. If there is a need for subsequent modifications, it is possible to modify both files.

6. Graphical Tool

The graphical tool, named Smart Contract Modeling Tool (SCMTool), allows for contract modeling simply by dragging and dropping components. SCMTool was developed using Sirius, a comprehensive framework that provides resources for building a graphical modeling environment.

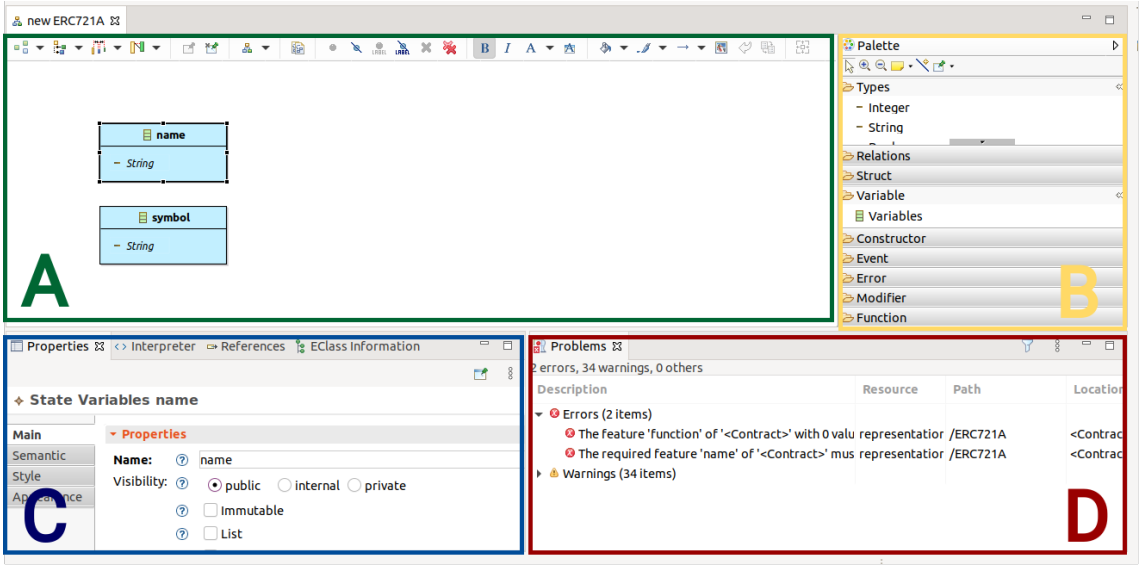

Figure 11. SCMTool.

Figure 11 illustrates the SCMTool and its main elements. The contract is modeled in the central area (A), where users can drag the components available in the palette (B). The palette contains all the components that make up the contract. When a rectangular shape is selected, it is possible to modify its properties (C). If the modeled contract does not comply with the HLM-SC specifications, errors are presented in this area (D). The SCMTool has validation rules according to the HLM-SC specifications, ensuring that it is only possible to drag a component onto one of the rectangular shapes when there is a match with the HLM-SC specification.

Figure 12 illustrates the graphical components of the diploma certification contract. Rectangular shapes represent the main components of the HLM-SC, and edges depict relationships. In the SCMTool, each file represents an instance of the *Contract*. State variables are represented by rectangular shapes, such as *_certificate* (blue color). The *_certificate* variable utilizes the complex data structure represented by *Certificate* (green color). The shape labeled as *mint* represents a function (white color) that employs the *onlyAuthorizedAddress* modifier (orange color). This modifier throws a *NotAuthorized* error if the calling address is not authorized (red color). Upon success, the *mint* function emits the event represented by the *CertificateIssued* shape (yellow color). Finally, the constructor (purple color) initializes the *institutionName* state variable.

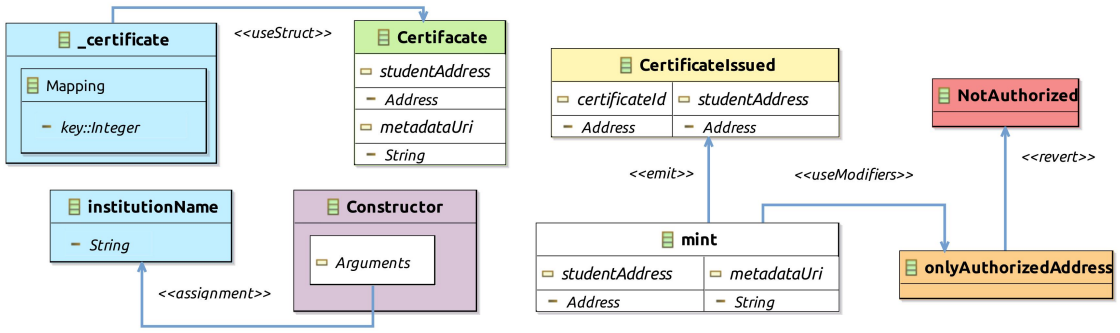

Figure 12. The main graphical components and their relationships.

The SCMTool employed the graphical representation of a Sirius diagram. The graphical elements utilized include nodes, containers, and edges. Containers can represent various types of graphical components and may contain others, unlike nodes, which represent only one component. Edges, in turn, represent the relationships between them. The components represent the metaclasses of HLM-SC, and the domain class defines their association.

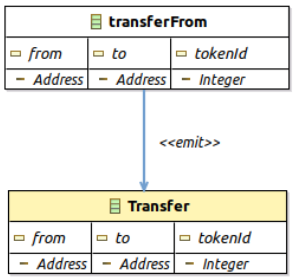

Figure 13. Example of the graphical representation of an event (*Transfer*) and a function (*transferFrom*).

A use of graphical representations to compose components is illustrated in Figure 13. The *Transfer* event is depicted by a yellow rectangular shape, modeled through a container. Within this container, additional containers representing the event members (*from*, *to*, and *tokenId*) can be added, each represented by white rectangular shapes. Within these containers, components denoting types (*address* and *string*), exemplified here through a

node, can be utilized. The white rectangular shape corresponds to a function (*transferFrom*) that has a relationship with the event, depicted by a blue edge with the keyword «*emit*».

One way to make the model editable for the user is by defining a tool palette to manipulate previously established graphical components. The absence of a tool palette would limit the model's functionality to visualization only, preventing any form of editing by the user. Components have been organized into groups based on their dependency relationships to facilitate the use of the tool palette. For instance, variable types used in various metaclasses have been grouped into a single category. Additionally, relationships between metaclasses have also been grouped, such as the «*useStruct*» relationship. Figure 14 illustrates, on the left, a snippet of the tool palette component development process, and on the right, how it is presented to the end user.

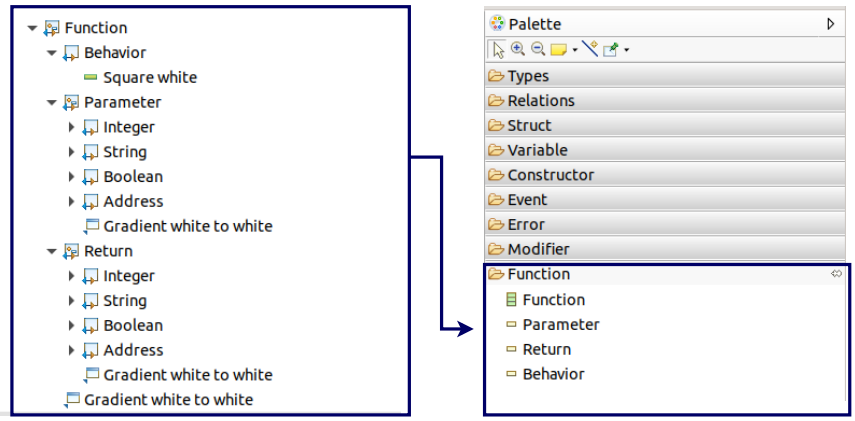

Figure 14. Example of a tool palette component during development (left) and its presentation to the end user (right).

After completing the contract modeling, it is possible to transform the model into code directly in the editor, following the workflow described in Section 5. The generated files are stored in the *src-gen* folder. If the user wishes to model another contract, initiating a new project using the SCMTool is necessary.

7. Conclusion

This paper proposes using Model-Driven Engineering (MDE) for contract development using a metamodel, a graphical modeling tool, and a code generation mechanism for Solidity. With the adoption of this approach, it is anticipated that both technical and non-technical individuals can create contracts more easily, regardless of their prior modeling knowledge.

The High-Level Metamodel for Smart Contract (HLM-SC) is an abstraction applicable in various contexts for developing contracts on the Ethereum Virtual Machine (EVM) platform. It comprises a set of metaclasses that enable the declaration of essential elements for constructing a contract, such as state variables, functions, complex data structures, events, and custom errors. The HLM-SC was initially crafted based on the fundamental structure of a Solidity contract and subsequently refined and adapted to facilitate contract development across diverse application domains. Consequently, the HLM-SC adeptly represents information and business rules inherent in a contract in a

more precise and structured manner. The Smart Contract Modeling Tool (SCMTool) is a graphical tool designed to facilitate intuitive contract modeling by allowing users to drag and drop graphical components within the SCMTool for contract creation.

A potential avenue for future research involves integrating machine learning techniques to assess the presence of vulnerabilities and error verification in contracts, thereby enhancing the contract creation process with HLM-SC. Furthermore, external assessors could evaluate the SCMTool to assess its usability. This evaluation could lead to improvements in the user interface, subsequently fostering increased adoption and utilization within the contract development industry.

References

[Achour et al. 2021] Achour, I., Idoudi, H., and Ayed, S. (2021). Automatic Generation of Access Control for Permissionless Blockchains: Ethereum Use Case. In *2021 IEEE 30th WETICE*, pages 45–50.

[Ait Hsain et al. 2021] Ait Hsain, Y., Laaz, N., and Mbarki, S. (2021). Ethereum's Smart Contracts Construction and Development using Model Driven Engineering Technologies: a Review. *Procedia Computer Science*, 184:785–790. The 12th ANT / The 4th EDI40 / Affiliated Workshops.

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

[Ben Slama Souei et al. 2021] Ben Slama Souei, W., El Hog, C., Sliman, L., Ben Djemaa, R., and Ben Amor, I. A. (2021). Towards a Uniform Description Language for Smart Contract. In *2021 IEEE 30th WETICE*, pages 57–62.

[Boubeta-Puig et al. 2021] Boubeta-Puig, J., Rosa-Bilbao, J., and Mendling, J. (2021). CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain. *Expert Systems with Applications*, 184:115578.

[Chirtoaca et al. 2020] 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.

[Corradini et al. 2022] Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., and Tiezzi, F. (2022). Engineering Trustable and Auditable Choreography-Based Systems Using Blockchain. *ACM Trans. Manage. Inf. Syst.*, 13(3).

[Dharanikota et al. 2021] 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 et al. 2019] 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 et al. 2020] 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 et al. 2018a] Garamvölgyi, P., Kocsis, I., Gehl, B., and Klenik, A. (2018a). Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems. In *2018 48th Annual IEEE/IFIP DSN-W)*, pages 134–139.

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

[Guida and Daniel 2019] Guida, L. and Daniel, F. (2019). Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development. In *2019 IEEE DAPPCON*, pages 59–68.

[Hamdaqa et al. 2022] Hamdaqa, M., Met, L. A. P., and Qasse, I. (2022). iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology*, 144:106762.

[Hamdaqa et al. 2020] 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 SAM '20*, page 34–43. Association for Computing Machinery.

[Iovino et al. 2012] Iovino, L., Pierantonio, A., and Malavolta, I. (2012). On the Impact Significance of Metamodel Evolution in MDE. *J. Object Technol.*, 11(3):3–1.

[Jiao et al. 2020] Jiao, J., Lin, S.-W., and Sun, J. (2020). A Generalized Formal Semantic Framework for Smart Contracts. In *Fundamental Approaches to Software Engineering: 23rd FASE 2020/ETAPS 2020, Proceedings*, page 75–96. Springer-Verlag.

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

[Jurgelaitis et al. 2022b] Jurgelaitis, M., čeponienė, L., and Butkienė, R. (2022b). Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development. *IEEE Access*, 10:33465–33481.

[Khan et al. 2019] 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 et al. 2020] Kudo, T. N., Bulcão-Neto, R. F., and Vincenzi, A. M. R. (2020). Metamodel Quality Requirements and Evaluation (MQuaRE). *arXiv preprint arXiv:2008.09459*.

[Li et al. 2021] Li, Z., Zhou, Y., Guo, S., and Xiao, B. (2021). SolSaviour: A Defending Framework for Deployed Defective Smart Contracts. In *ACSAC '21*, page 748–760. Association for Computing Machinery.

[Qasse et al. 2021] 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.

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

[Santiago et al. 2021] Santiago, L., Abijaude, J., and Greve, F. (2021). A framework to generate smart contracts on the fly. In *Proceedings SBES '21*, SBES '21, page 410–415. Association for Computing Machinery.

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

[Zeng et al. 2022] 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.