SCEditor: A Graphical Editor Prototype for Smart Contract Design and Development

Yassine Ait Hsain[1], Naziha Laaz[2], Samir Mbarki[3]
Information Modeling and Communication Systems Team, EDPAGS Laboratory
Faculty of Science, Ibn Tofail University, Kenitra, Morocco[1,3]
ASYR RT, LaGeS Laboratory, Hassania School of Public Works, Casablanca, Morocco[2]

***Abstract*—In recent years, particularly with the Ethereum blockchain's advent, smart contracts have gained significant interest as a means of regulating exchanges among multiple parties via code. This surge has prompted the emergence of various smart contract (SC) programming languages, each possessing distinct philosophies, grammatical structures, and components. Consequently, developers are increasingly involved in SC programming. However, these languages are platform specific, implying that a transition to another platform necessitates the use of different languages. Additionally, developers require a certain level of control over SCs to address encountered bugs and ensure maintenance. To address these developer-centric challenges, this paper presents SCEditor, a novel Eclipse Sirius-based prototype editor designed for the visualization, design, and creation of SCs. The editor proposes a means of standardizing the usage of SC programming languages through the incorporation of graphical syntax and a metamodel conforming to Model-Driven Engineering (MDE) principles and SC construction rules to generate an abstract SC model. The efficacy of this editor is demonstrated through testing on a voting SC written in Vyper and Solidity languages. Furthermore, the editor holds potential for future exploitation in model transformation and code generation for various SC languages.**

***Keywords*—Blockchain; Metamodel; Model-driven Engineering (MDE); Smart Contract (SC); SC Programming**

I. Introduction

In 1994, SC started as a concept of formalizing and securing relationships over networks [1]. SCs are self-executing applications, representing a key technology of a decentralized system based on Blockchain platforms [2].

The integration of SCs across various sectors has been extensively explored and advocated for within academic literature. These contracts, enabled by blockchain technology, offer versatile applications with significant implications. They enhance security, trust, and efficiency in diverse domains such as healthcare [3], [4], banking and finance [5], IoT [6], secure data sharing [7], supply chain [8], business [9], [10], education [11], software development methodologies [12], [13], security risk management [14], [15], and legal frameworks [16].

Studies emphasize SCs' pivotal role in reshaping traditional business models [9], democratizing software development through accessible methodologies [12], [10], [13], fortifying security frameworks against vulnerabilities [14], [15], and promising enhanced security and efficiency in education systems [11]. Additionally, the evolution of SCs into self-enforcing entities, known as smart legal contracts, marks a transformative shift in contractual agreements [16].

This widespread impact affirms their integral role in shaping the future landscape of multiple domains. The substantial interest in SCs has led to the creation of various development languages aimed at facilitating their programming and mitigating potential issues related to their maintenance and control.

Solidity and Vyper are the two most popular and widely used languages in SC programming. Solidity is object-oriented and considered the primary language for Ethereum and other private Blockchain-based platforms. Vyper, on the other hand, has a clear and straightforward compilation code, it is a Pythonic programming language characterized by strong typing. Furthermore, Vyper purposefully includes less functionality than Solidity in an effort to make contracts more secure and simpler to audit [17].

Despite recent advancements in the SC programming languages, they still have a lot of problems to overcome, and several concerns continue to undermine their adoption. For example, Vyper does not support Inheritance, Inline Assembly, Function and Operator Overloading, Recurring Calls, etc [18], [19]. A significant challenge that developers encounter in the Ethereum platform is the dilemma of deploying code to a system that is immutable while the development platform itself continues to evolve. You cannot just easily upgrade or change your SCs. You must be ready to take proper actions to solve occurring problems, from migrating users, apps, and funds to deploying the SC.

To produce a SC with one of these two widely used languages, it will take time learning as well as coding. Also, developers need to have certain control over the SC, so in case of a hard fork. To tackle this problem effectively, developers can promptly and appropriately respond by taking necessary measures. Therefore, to enhance the definition and development of SCs, it is preferable to raise the level of abstraction and offer a contract modeling mechanism independently of any specific language [20].

In recent years, the exploration of Model-Driven Engineering (MDE) as a means to abstractly model smart contracts has garnered significant attention among researchers. While the State-Machine model, UML Class Diagram, and Business Process Model and Notation (BPMN) have been central to many of these studies, there remains a minority that explores the model-driven development process. In this paper, we introduce SCEditor, a prototype model-driven tool developed using Eclipse Sirius, and based on an abstract metamodel. It aims to standardize the modeling of smart contracts by leveraging MDE principles.

The primary objective of the proposed tool is to streamline development processes and make more efficient the design, visualisation and generation of SCs. This work also discusses the common SC programming languages, picking the most used ones, comparing them, and deducing common components to extract an abstract metamodel. The latter is used to instantiate models including structural and functional aspects, establishing a complete representation of SCs. The key benefit of the SCEditor is that offers a range of SC components in a user-friendly graphical syntax. This facilitates the creation of platform-independent SC models, simplifies platform migration, allows adaptation to various SC languages, and ensures ongoing control over them. To test the validity of the proposed editor, a voting SC application was designed, found in the documentation of both Vyper and Solidity.

The paper has the following structure: Section II reviews innovative and recent academic approaches for SC development based on Metamodeling. Section III gives an overview of SCs as well as MDA-based tools, whereas Section IV presents the proposed approach by describing the metamodel definition process and explaining in detail the implementation process of the SCEditor. Section V presents the chosen use case to verify the validity and effectiveness of the proposed method and emphasize the findings. In Section VI we discuss the obtained results and position our proposal against the studied approaches. Finally, Section VII summarizes the main findings and proposes some suggestions for future directions.

II. Related Work

Several studies have been conducted regarding SC programming and MDE. Most of them are based on state-machine or BPMN, and few on model-driven development process [20]. Most of the existing approaches focus on the behavioral aspect of SCs and employ both BPMN and UML statechart models for modeling business processes [21], [22], [23], [24].

During our research, we initially came across Lorikeet [23], which exploits BPMN models and fungible/non-fungible registry data schemas to create standardized ERC-20/ERC-721 compliant asset registry SCs. Lorikeet's BPMN Modeler is developed using the bpmn-js modeling library, which is licensed to bpmn.io, a division of Camunda. This led us to explore another tool called Caterpillar, also developed by the same author. Caterpillar makes use of Camunda, an open-source platform for workflow and decision automation. One notable advantage of Camunda is its upgradability and its adherence to industry standards such as the Case Management Model and Notation (CMMN) and the Decision Model and Notation (DMN), as defined by the Object Management Group (OMG).

Another approach based on BPMN entitled BlockME was presented by [21]. It focuses on creating a business process based on BPMN 2.0 which has the ability to integrate with the Blockchain Access Layer (BAL), which serves as a middleware facilitating communication between external applications and open blockchain systems, allowing for transaction exchange.

In study [22], the authors have specified a visual domain-specific language (DSL) obtained from a UML class diagram, and this method uses a collection of BPMN and DEMO (DMN) models for the design of the process. The proposed approach presents a metamodel for designing a SC, which is simply a class diagram that incorporates various SC concepts.

A Model Driven Architecture (MDA) based approach was proposed by [25], to define legal SCs. It consists of the definition of the UML class diagram which describes legal SC components like legal states, data sources, action, etc. The added value of this approach is the comparison of current modeling languages for the creation of legal SCs in light of the suggested unified model.

iContractML 2.0 [26] is a framework that allows the creation of SCs using MDE. The proposed tool focuses on using a reference UML class diagram to model SCs graphically. However, the tool does not fully support the functional aspect, as it provides templates only for some of the commonly used basic functions.

The study used the Model-driven Architecture [27], employing the UML class and state-chart diagrams to model the structural and functional aspects of SCs. The modeling process involved a series of transformations, including model-to-model (M2M) and model-to-text (M2T) transformations, which converted the Solidity PSM model into code.

The study in [28] introduces a model-driven framework that automates the storage of domain-specific data on the Ethereum blockchain platform. It achieves this by utilizing Ecore model instances. The framework generates Solidity SCs by employing model-to-model and model-to-text transformations through the use of Acceleo. To evaluate the approach, the persons-movies dataset was utilized within the Ganache environment.

III. Background and Motivation

A. Smart Contract Programming Languages

Ethereum, the blockchain-based platform, is considered as a reliable technology with integrity characterized by the decentralized execution of processing in SC format [29]. It has become the most popular platform for both deploying and storing SCs in a public distributed database [30]. It was not until the advent of Ethereum that the concept of SCs was implemented, even though it had been around for years. The first idea was born in 1994 by Nick Szabo [31]. A SC is an autonomous computer application that runs without the need for validation by stakeholders. SCs are exploited to make transactions. Therefore, if a transaction attempts to perform more transactions than the gas spent allows, an exception is thrown, and the transaction is cancelled. Of course, the gas represents the fee that is paid before executing the SC by the caller with the Ethereum currency ETH.

For SCs development, Solidity is frequently employed, being the foremost choice due to its widespread popularity. In addition to Solidity, several other SC programming languages are utilized, such as Vyper, Mandala and Obsidian [32]. Each of these languages takes different approaches to enhance existing development tools.

As shown in Table I, we have compiled a comprehensive list of most of the SCs programming languages, organized them based on their respective dates of creation, and evaluated them against the following set of characteristics:

- Paradigm: This is a technique used to group programming languages based on their characteristics. Multiple paradigms can be used to categorize languages.
- Level: Programming languages can be distinguished into two levels: High and Low. The major distinction between them is that high-level languages are simpler for programmers to comprehend, interpret, and compile than low-level languages. Unlike humans, machines can easily and quickly understand low-level languages.
- Targeted platform: This defines the platform on which the language runs.

TABLE I. SC LANGUAGES DESCRIPTION

SC Programming languages	Paradigm	Level	Targeted Platform
LLL [33]	functional	low-level	Ethereum
Serpent [34]	procedural	low-level	Ethereum
Solidity [35]	object-oriented	high-level	Ethereum
Vyper [36]	procedural	low-level	Ethereum
Bamboo [37]	procedural	high-level	Ethereum
Obsidian [38]	state-oriented	-	Hyperledger Fabric
Rholang [39]	concurrent	-	RChain cooperative
Michelson [40]	stack-based	high-level	Tezos blockchain
Plutus [41]	functional	-	Cardano blockchain
Sophia [42]	functional	-	Æternity blockchain
Mandala [43]	-	high-level	-
Flint [44]	contract-oriented	high-level	Ethereum
Scilla [45]	functional	intermediate-level	Zilliqa

The choice of SC programming languages is limited to two, a deliberate decision prompted by various factors. Some languages are still in the development process, others do not provide well-informed documentation, and some are no longer used by developers. The selection process focused on identifying the most widely utilized and well-documented programming languages, to afford the maximum characteristics of comparison. Consequently, the chosen languages are Solidity and Vyper.

Through an analysis of both shared and distinct features, we delineate the disparities between the selected languages at two distinct levels: conceptual and syntactical. At a conceptual level, a prominent disparity between Solidity and Vyper lies in their handling of root element. Notably, a single Solidity file can encompass multiple SCs, whereas with Vyper, each SC necessitates its own separate file. In terms of syntax, Solidity draws inspiration from JavaScript, while Vyper is inspired by Python syntax (Indentation, constructor, etc).

B. MDA Based Tools

1) The MDA Architecture: MDA, which stands for Model Driven Architecture, is a specific vision of MDE defined by the OMG group. MDE is a powerful approach that exploits models as central artifacts to streamline the software and systems development process. It is considered a methodology for improving the quality, efficiency, and maintainability of software and systems by focusing on abstract representations of system structure, behavior, and functionality. When used effectively, MDE can lead to improved productivity, higher-quality software, and greater adaptability to changing requirements and technological platforms. Therefore, MDA can be seen as a subset of MDE, which represents an architecture for designing, visualizing, developing, transforming, and storing software models that the machine can understand, and develop independently of the implementation technologies by separating technical constraints from functional ones [46].

As depicted in Fig. 1, the OMG group proposes the MDA and turns it into a realizable engineering framework for use in the system/software design process. This approach advocates the exploitation of models as operational elements participating in the production and implementation of software. Consequently, the model serves as input for a transformation engine, which generates some or all of the desired software. Models are instances of the concepts defined by the metamodel, and a metamodel, in turn, defines the structure and rules that govern the construction of models in a particular domain. Metaclasses, on the other hand, are used to define the types of elements in a metamodel. The MDA approach requires the production of computation-independent models (CIM) which are transformed into platform-independent models (PIM) and subsequently into platform-specific models (PSM). Several tools and graphical editors have been created based on the principles of MDA to design models and facilitate the implementation of model transformations. One notable example is Eclipse Sirius.

2) Eclipse sirius: Sirius, developed by the Eclipse Foundation, is an open-source project that offers a flexible workbench for model-based architecture engineering. This workbench can be customized to suit specific requirements and needs. Sirius gives developers the ability to create a fully rich graphical editor containing all the components needed to design the domain model, from tables, trees, nodes, edges, colors, shapes, etc. It also offers the possibility to deploy the editor on the Web [47].

By leveraging Sirius, developers can implement the MDA principles by creating graphical editors. These editors can provide a visual representation of the models, allowing users to create, edit, and validate them. Additionally, Sirius can be combined with other OMG standards to define transformation rules and generate code from the models, aligning with the code generation aspect of MDA.

From the perspective of specifiers and developers, Sirius enables the capability to define workbenches that incorporate various editors such as diagrams, tables, or trees. It allows for seamless integration and deployment of these customized environments within Eclipse IDE or RCP applications. Additionally, existing environments can be further customized through specialization and extension. From an end-user perspective, Sirius provides feature-rich and specialized modeling editors that facilitate the design of models. It ensures synchronization between different editors, enabling a cohesive and efficient modeling experience.

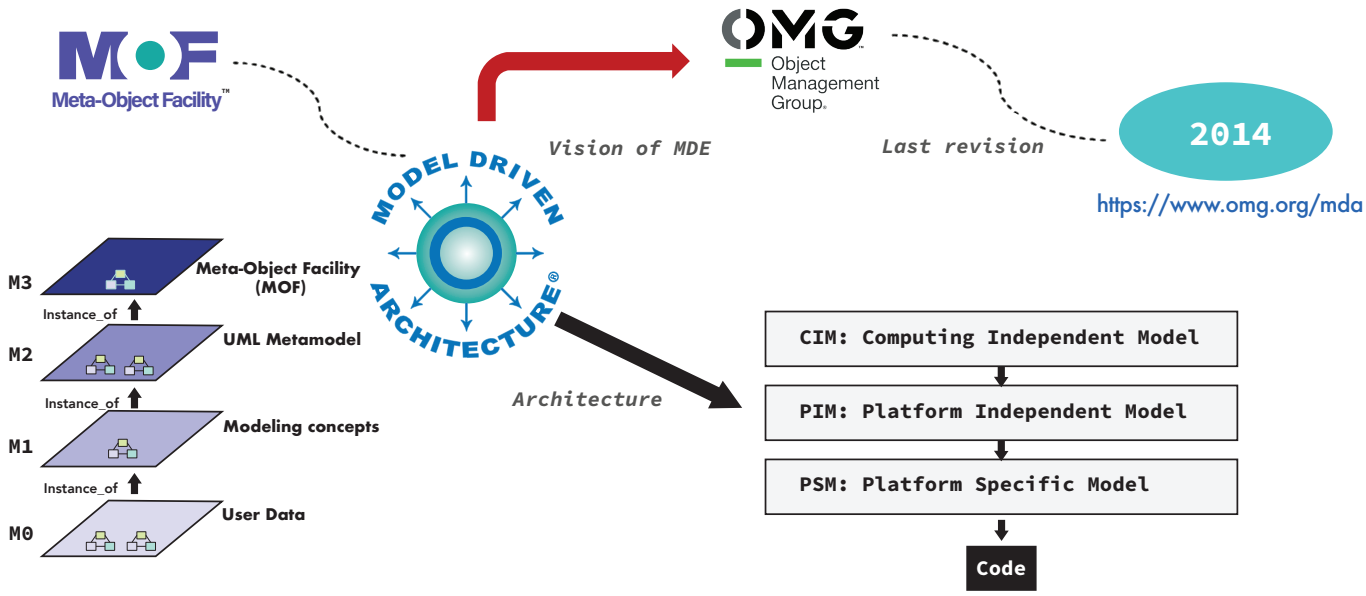

Fig. 1. The MDA framework.

IV. The Proposed Approach

In this section, we explain the proposed approach for the elaboration of the SCEditor. To begin with, we started with the creation of an abstract representation of the SC components, then, suggested a graphical representation for each one of them. Finally, we used the metamodel in the Sirius Eclipse project to elaborate a prototype of the SCEditor.

A. Metamodel Definition

The creation of the metamodel was done using Eclipse Modeling Framework (EMF) [48], which is an open-source technology in model-based software development. It is a comprehensive abstraction for describing, creating, and working with structured data.

The construction of the proposed metamodel was based on gathering a set of concepts used in the composition of SCs. These concepts were collected from both the literature describing SCs and the studied languages. There were continuous updates to the metamodel as we encountered new references, and while creating model instances that covered most of the SC elements.

In our endeavor to present a comprehensive yet digestible depiction of a the metamodel, we employed segmentation, dividing it into distinct structural and functional components. This segmentation aimed to enhance clarity and facilitate a more focused analysis. However, this division led to the exclusion of some relationships bridging the structural and functional aspects.

Fig. 2 illustrates the structural aspect of the metamodel. This segmented representation aims to provide a detailed breakdown of the structural aspect, showcasing the various metaclasses and their interrelations within the metamodel. it's important to note that due to the complexity and size constraints, certain interrelationships with the functional aspects have been omitted to maintain readability and visual clarity.

Fig. 3 presents the functional aspect of the segmented metamodel, focusing on the dynamic behavior and interactions between system components. This depiction emphasizes the operational aspects, illustrating Statements, Expressions and interactions between the identified elements within the function Body. The functional representation aims to elucidate the operational flow and functionalities. While this view provides insights into the functional dynamics, it may lack some structural context crucial for a complete understanding of the system.

The classes illustrated in Fig. 2 and Fig. 3 encompass both the structural and functional facets of the SC. Within the structural realm, delineating the program's foundational architecture, reside classes such as "Struct," "Interface," "Variable," "Modifier, "Function," "Event, and "Type" Conversely, the functional aspect characterizes the SC's operational behavior, incorporating the abstract classes "Statement" and "Expression" and their respective derived subclasses. Elaboration on the intricate relationships among these classes is provided in the next section.

B. SCEditor

SCEditor is a graphical editor for SCs design, visualization, and generation. The proposed editor is a high-level tool that provides a set of SC elements in a comprehensive graphical representation, making it easy for the users to create platform-independent models of SCs, facilitate platform migration, enable adaptation to different SC languages, and maintain a certain control over them through time.

SCEditor is based on the eclipse project "Sirius". It relies on creating model instances of a given SC conform to the

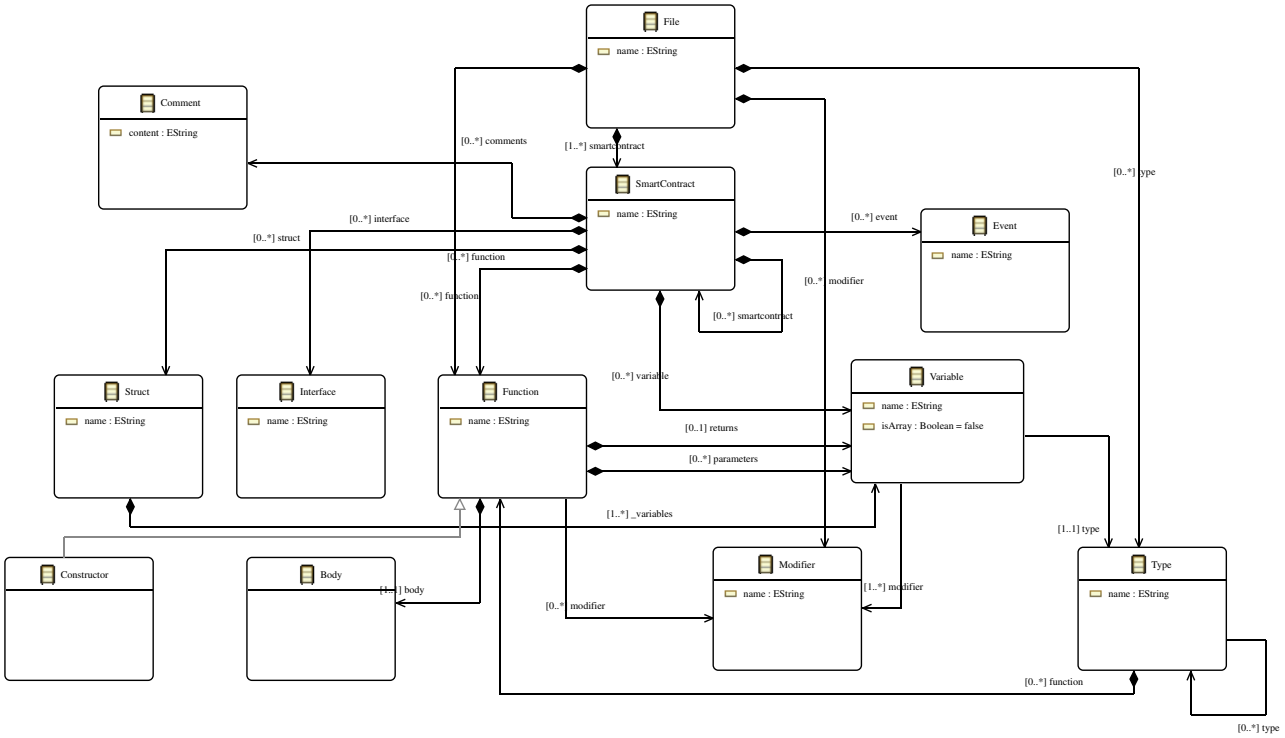

Fig. 2. Metamodel strcutural metaclasses.

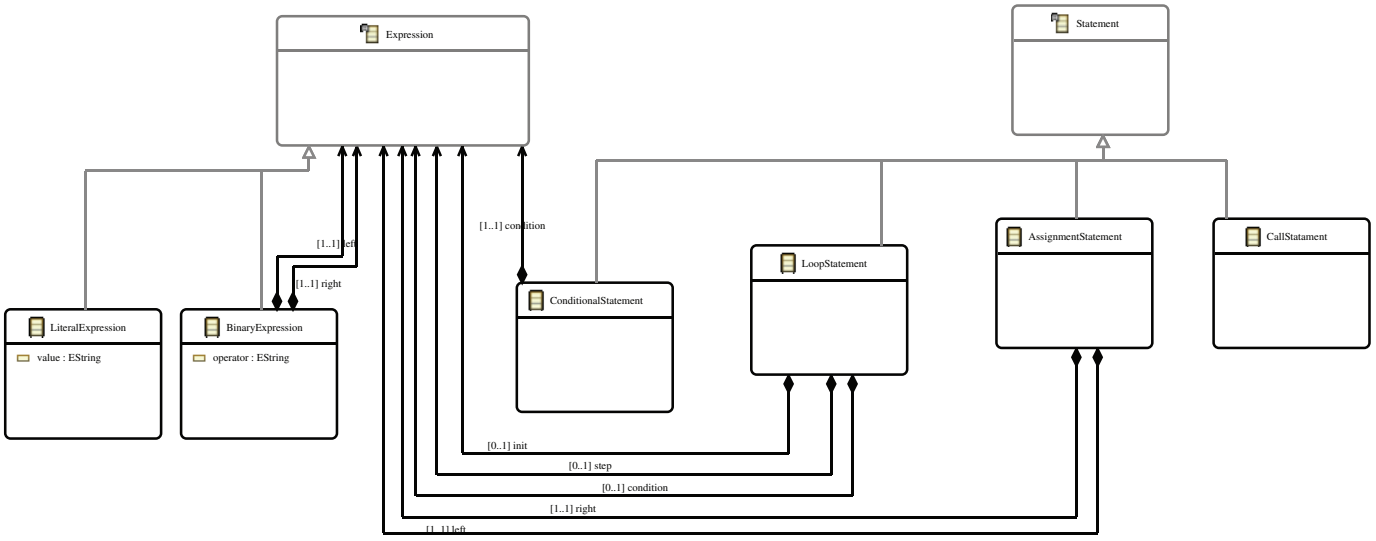

Fig. 3. Metamodel functional metaclasses.

SC metamodel. The creation of the model consists of using the elements provided in the palette of the editor. The editor relies on two diagrams: the Smart Contract Diagram (SCD) which contains elements needed for the representation of the structural aspect, and the Function Diagram (FD) composed of elements describing the functional aspect of the SC metamodel.

1) Smart contract diagram: The structural and static aspect plays an integral role in storing, managing, and manipulating

data and transactions within a SC. In this section we present SCD which is a graphical representation comprising various structural nodes. it describes the static aspect of a given SC. Its visual representation aids in comprehension, enabling developers to grasp the composition and structure of the SC.

At the outset, the design of the SC model begins with the creation of the SCD, the latter represents the top layer of the model. These elements are categorized and found in the palette (see Fig. 4(a)).

The SCEditor comprises two palettes: one dedicated to the SCD (see Fig. 4(a)), and the other specifically designed for the FD (see Fig. 4(b)), the latter will be elaborated upon in the following section. These palettes encompass tools divided into four categories:

- Contract Tools: It contains the main elements of a SC, such as Contract, Function, and Struct.
- Statement Tools: It contains all statement elements like AssignmentStatement, CallStatement, etc.
- Expression Tools: It contains all expression elements, like LiteralExpression, ValueExpression, etc.
- Function Tools: It contains the variables used as Parameter or Return variables.

Table II presents the components of the SCD, accompanied by their respective palette icons and graphical representations.

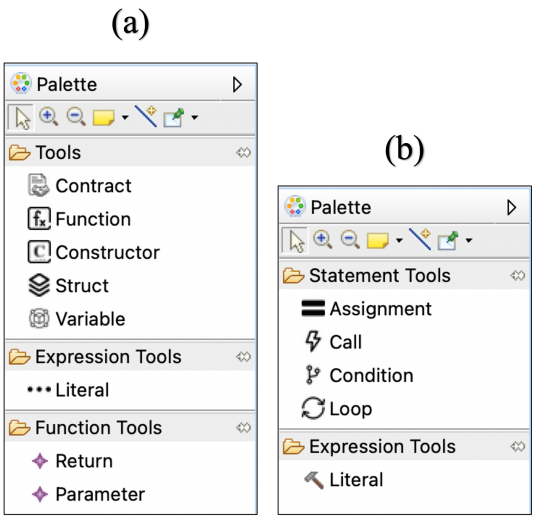

Fig. 4. SCD and FD palettes.

Contract, Struct, and Function elements are represented as containers. The reason behind is that these elements can contain other components. “Contract” represents the container that will include all the necessary elements to design the SC structure and logic. It is composed of one or many Struct, Function, or Variable. The “Struct” element defines a custom type and contains one or many “Variable”. The latter constitutes a value stored in the SC storage. Finally, the representation of the “Variable” element is defined by a node.

When representing the logic of the SC, we faced many challenges. The first one was the creation a comprehensive

TABLE II. SCD ELEMENTS

Element	Palette icon	Graphical Representation
Contract		Container
Struct		Container
Variable		Node
Function		Container

set of components (Statement, Expression) to illustrate the functional aspect of the SC model. The second one was related to the representation of these components which was excessively growing as the body of the function expanded, leading to an overcrowded diagram. To solve these issues, a “Function Diagram” was defined, thus having only the visualization of the functions on the SCD, and when explored, it leads to a specific FD representation.

2) Function diagram: The Function Diagram is a visual representation that depicts the structure and relationships of statements within a “Function” element.

The FD serves as a valuable tool to express the logic of a function and to improve the readability of the SCD. The numerous elements required for the representation of the function body is mitigated by encapsulating them in the “Function” node.

To create an FD, we first need to declare a “Function” element in the SCD, then, by double-clicking the latter we navigate into the FD workbench that contains a dedicated palette shown in Fig. 4(b). This palette is composed of “Statement” and “Expression” child metaclasses required to design the function body. These elements are described in Table III with their graphical representation.

When choosing a type of “Statement”, the editor will create a container composed of the nodes required for the composition of the selected element. For example, the creation of an “AssignmentStatement” implicates the construction of two nested nodes, the first is the left “Expression” that will be assigned the value of the second one, which is the right “Expression”. The CallStatement allows the call of any defined function in the “File” element. As for the “LoopStatement”, it is used when an iteration is needed. The construction of this element implicates the creation of three nodes: “initial” which describes the initial state of the iteration, “condition” which indicates the condition to stop the iteration, and “step” which represents the iteration step. For expressing a series of “Statement” conditioned by a “Condition” node, we use “ConditionalStatement”. Finally, we have the “Expression” child metaclasses which consist of “LiteralExpression” that contains an expression value, and, “BinaryExpression” which is used to compare, increment, or decrement a certain expression.

It is important to note that besides the nodes contained in “LoopStatment” and “ConditionalStatement”, these elements need to have a body that contains their logic. Similarly, we can define an FD to describe these statements’ bodies.

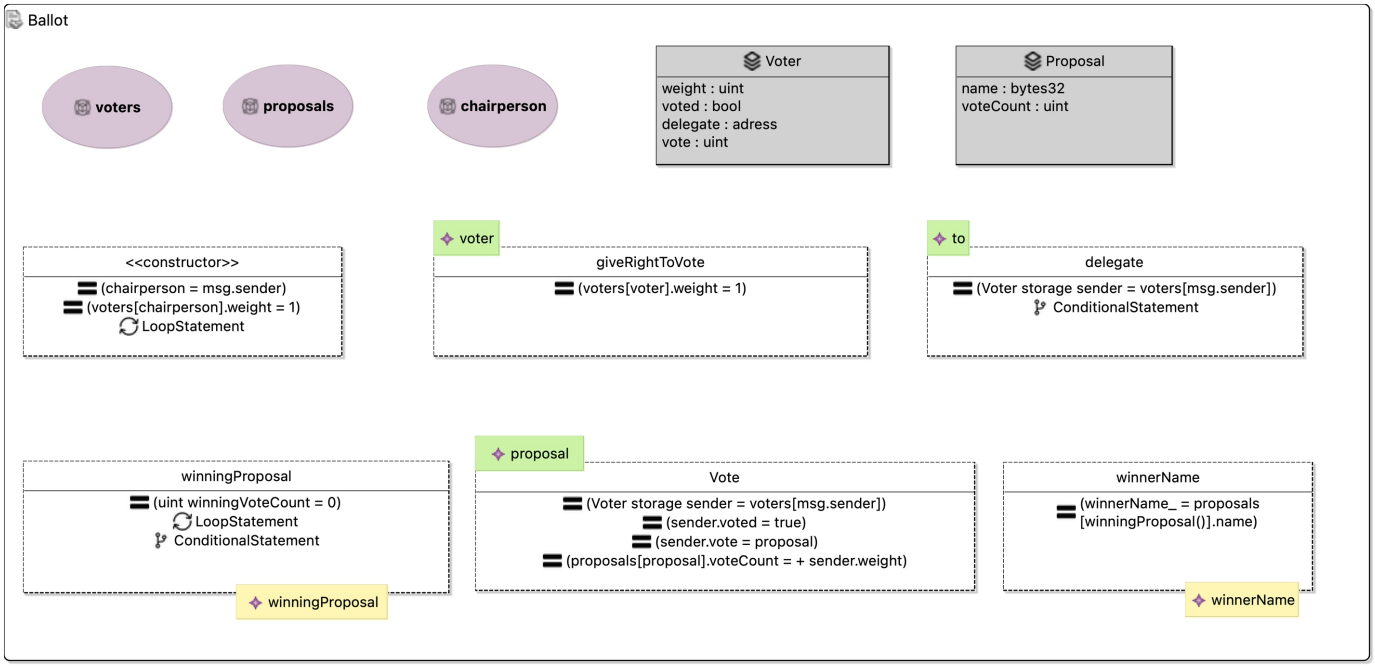

Fig. 5. SCD representation of the Ballot SC.

TABLE III. FD ELEMENTS

Element	Palette icon	Graphical representation
AssignmentStatement		Container
CallStatement		Container
LoopStatement		Container
ConditionalStatement		Container
LiteralExpression		Container

V. RESULTS

A. Case Study

Voting is a delicate, precise, and open procedure process, it requires a certain degree of security and privacy. The issue with most of the voting applications is that they have several design problems [49]. They are centralized by design and proprietary, which implies that the code base, database, system outputs, and monitoring tools are all under the simultaneous control of one supplier. Such centralized systems struggle to gain the credibility needed by voters and election organizers due to the absence of an open-source, independently verifiable output.

Given this, blockchain technologies can be very helpful for this process as they offer open-source, peer-reviewed software that is ubiquitous, secure, and efficient, preserving the ballots' confidentiality, enabling free, impartial audits of the results, lowering the level of confidence required from the organizers [49], [50].

In this light, we choose to work with a basic example of a voting application to validate the applicability of our approach. The case study represents a voting SC that will be used as a reference to create a model based on the metamodel. The latter is found in the documentation of the studied languages [51], [52]. The SC will create one contract per ballot, and then the chairperson - who is the creator of the contract - will grant the right to vote to each individual by his address. These individuals will then choose to vote themselves or delegate their vote to another person they trust. Finally, after the voting process is done, we will get the proposal with the largest number of votes.

The selected SC highlights numerous features of the SC languages, implying a test of the proposed metamodel's validity. To make it short, we focused only on code segments implemented in Solidity [51] and Vyper [52]. This decision was made due to existence of most elements required for constructing a SC, encompassing SC and variable declarations, structures, constructors, and more.

Analyzing code from both Solidity and Vyper reveals common elements such as variables, loops, conditional statements, and functions. Both languages share fundamental constructs for control flow and data types, despite syntax variations.The common elements found in codes are as follows:

- **Ballot contract**: It refers to the voting SC. In Vyper, it is represented by the filename itself.
- **Voter structure**: It contains information about the voter defined by the following variables:
  - **weight**: It is accumulated by delegation.
  - **voted**: True/false, depends if the person has voted or not.
  - **delegate**: The address of the person to whom the right to vote has been delegated.
  - **vote**: Index of the voted proposal.
- **Proposal structure**: it can be created by users. It has the following variables:

- **name**: Name of the proposal.
- **voteCount**: Number of votes.

- **chairperson variable**: The creator of the contract.
- **giveRightToVote** function: It gives a voter the right to vote, it can be called only by ‘chairperson‘.
- **delegate function**: It delegates vote to the voter ‘to‘.
- **winningProposal function**: It returns the proposal with the largest number of votes.

B. Design and Implementation

In this section, we will provide an example of an SCD, an FD, and the resulting model.

At the outset of creating a new diagram, the workbench automatically positions itself within the root file which contains one or multiple SCs. The illustrated palette in Fig. 4(a), provides the necessary elements to create a Contract, Struct, Function, or Variable.

The user initiates by selecting the desired item and clicking on the workspace, triggering the display of a graphical representation of the corresponding element. Subsequently, the user is asked to enter the properties of the created element. Once completed, the user can proceed to define the description of the functions, this can be achieved by double-clicking the function element on the SCD. This action generates new workspace and palette enabling the user to define the body of the function

Fig. 5 illustrates the SCD of the case study. First, we find the name of the SC on the top left of the container. The three purple elliptic nodes represent the voters, proposals, and chairperson variables. Right next to them, we can find the Voter and Proposal Struct illustrated by a gray container containing the dedicated variables. The constructor and functions of the SC are represented by dashed white containers including in the body the logic defined by the Function Diagram. The green-bordered nodes found on top of the functions represent the input parameters, while the yellow-bordered ones represent the return value of the function.

For example, the “winningProposal” function shown in Fig. 5, has five statements varying from AssignmentStatment, LoopStatement, and ConditonalStatement. These statements are ordered according to the execution order declared in the Function Diagram in Fig. 6.

Fig. 7 represents an ecore viewpoint of the resulting model. The instance output is an XML Metadata Interchange (XMI) format, which is an OMG standard for the representation of object-oriented information in XML format. It is important to highlight how easy it is for the user to use the XMI model for other Model-driven related purposes such as M2M or M2T transformations.

The components depicted Fig. 7(a) directly correlate to the specific elements established within the SCD. Furthermore, Fig. 7(b) serves as an illustration of the structure of the Voter “Struct” element. Additionally, Fig. 7(c) elucidates the composition of the “winningProposal” function, showcasing its internal body structure and the variable returned by this function.

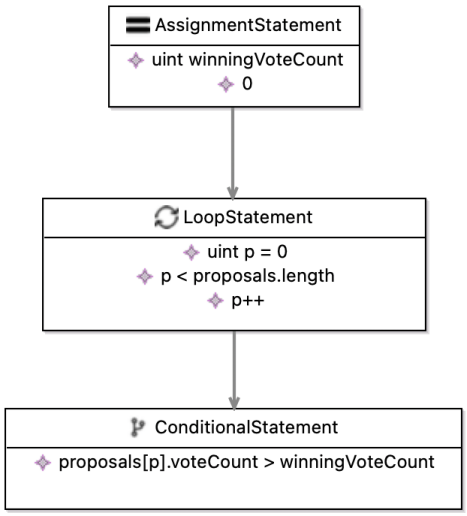

Fig. 6. FD representation of the winningProposal function.

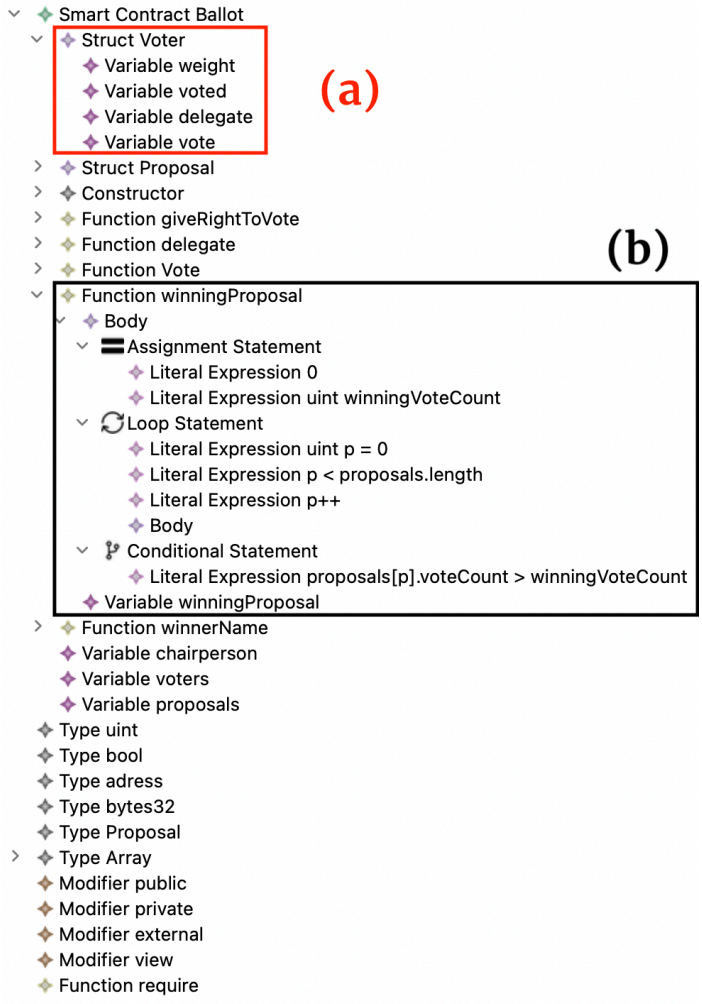

Fig. 7. Ballot model (a) with voter struct (b) and winningProposal function (b) in ecore viewpoint.

VI. Discussion

The primary objective of this research endeavors to construct a structured graphical representation for SCs, founded upon an abstract metamodel developed in adherence to MDE standards. The methodology adopted for this endeavor involves the construction of a metamodel, predicated on an in-depth analysis of programming languages used for SCs, aiming to offer a broader, more generalized, and abstract delineation. This approach facilitates a platform-independent depiction of SC definitions, emphasizing the comprehensive portrayal of both structural and behavioral facets intrinsic to SCs. The core focus of this representation lies in encapsulating the intricacies inherent in the structural and operational dimensions of SCs.

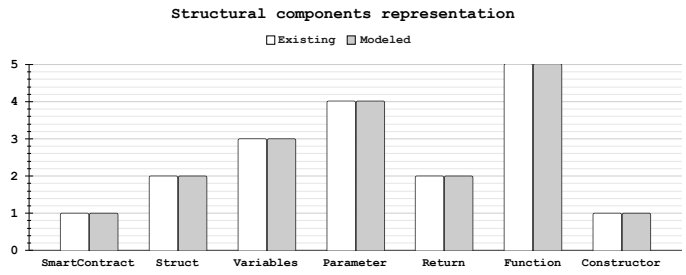

Fig. 8. Structural representation comparison between existing SC code and elements modeled within the SCEditor modeled elements.

Fig. 8 displays a bar graph contrasting existing elements in white with modeled components in gray. The white bars indicate the count of elements found within the codebase, including classes, structs, variables, parameters, returns, functions, and constructors. Conversely, the gray bars represent the number of these code elements visually depicted in the diagram. This visual comparison reveals that all structural components were successfully represented.

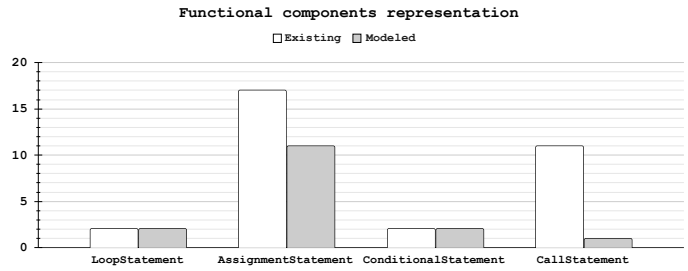

Fig. 9. Functional representation comparison between existing SC code and elements modeled within the SCEditor modeled elements.

In Fig. 9, it is evident that only 50% of the behavioral components were represented in the model, predominantly comprising function bodies, resulting in a cumulative representation of 68% for all components. This comparison underscores notable disparities; for instance, out of the 17 existing AssignmentStatements, only 11 were visually represented. Similarly, among the 11 CallStatements, merely one was visually modeled.

Several limitations were encountered concerning the representation of the behavioral aspect (FD). Specifically, the complexity of FD elements increased as the statements became progressively more intricate. Although the representation remained feasible, the growing number of components resulted in an overcrowded diagram. Regarding the structural aspect, users are required to define the types and built-in functions they intend to employ while constructing the SCD and FD. Additionally, the absence of test and exception handling functions (e.g., assert, require, etc.) is notable. Further work and validation are necessary to incorporate these functionalities at the abstract level alongside other features.

After examining various modeling approaches highlighted in the related work section, a key observation emerged regarding the predominant use of BPMN. Many methodologies for representing business processes rely on BPMN, an officially recognized standard by the OMG. However, while BPMN adeptly illustrates data flow and connections between data artifacts and activities, it isn't explicitly designed as a data flow language. Furthermore, this specification does not cover the operational simulation, monitoring, or deployment of business processes.

Other modeling approaches offer capabilities similar to those of our proposal [26], [27], [28]. It can be argued that these approaches are restricted in their scope as they do not include all the structural and dynamic components of the SC. However, they can only present the structural aspect of the SC leaving the functional one to the user in the manual definition.

In comparison with these approaches, the SCEditor presents a broad and abstract representation of the SC, as the user can define both the structural and functional aspects of the SC.

VII. Conclusion

This work presents the SCEditor, a prototype model-driven tool based on an abstract representation designed to standardize modeling SCs using MDE. The development of this graphical editor utilized Eclipse Sirius in conjunction with a metamodel definition formulated from derived rules originating from Solidity and Vyper languages.

The primary objectives of this proposed tool encompass streamlining and enhancing the efficiency of SC design and modeling processes, to meet the needs for developing large-scale SCs.Additionally, it aims to consolidate the similarities across various SC programming languages while identifying and addressing disparities between them, ultimately proposing a unified model.

We conducted a validation of our graphical editor by subjecting it to a voting SC example sourced from Solidity and Vyper documentations. The graphical editor effectively modeled a majority of the SC components, encompassing both its structural and functional elements. This abstract representation of SCs holds promise for future utilization in generating customized code for various blockchain programming languages

Our forthcoming efforts will center on model transformations to target additional blockchain platforms. This approach will enable us to propose more abstract models that adhere to the distinct rules of each platform. Subsequently, we intend to conduct a usability test aimed at addressing challenges associated with the abstraction of the metamodel.

REFERENCES

[1] N. Gabashvili, T. Gabashvili, and M. Kiknadze, "From paper contracts to smart contracts," *Sciences of Europe*, no. 107, pp. 124–127, 2022.

[2] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: Applications, challenges, and future trends," *Peer-to-peer Networking and Applications*, vol. 14, no. 5, pp. 2901–2925, 2021.

[3] M. Attaran, "Blockchain technology in healthcare: Challenges and opportunities," *International Journal of Healthcare Management*, vol. 15, no. 1, pp. 70–83, 2022.

[4] A. Rghioui, S. Bouchkaren, and A. Khannous, "Blockchain-based electronic healthcare information system optimized for developing countries." *IAENG International Journal of Computer Science*, vol. 49, no. 3, 2022.

[5] M. K. Al Kemyani, J. Al Raisi, A. R. T. Al Kindi, I. Y. Al Mughairi, and C. K. Tiwari, "Blockchain applications in accounting and finance: qualitative evidence from the banking sector," *Journal of Research in Business and Management*, vol. 10, no. 4, pp. 28–39, 2022.

[6] L. Elhaloui, M. Tabaa, S. Elfilali, and E. habib Benlahmar, "Promises, challenges and opportunities of integrating sdn and blockchain with iot applications: A survey."

[7] K. Yuan, Y. Yan, L. Shen, Q. Tang, and C. Jia, "Blockchain security research progress and hotspots." *IAENG International Journal of Computer Science*, vol. 49, no. 2, 2022.

[8] M. El Midaoui, E. B. Laoula, M. Qbadou, and K. Mansouri, "Logistics tracking system based on decentralized iot and blockchain platform," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 23, no. 1, pp. 421–430, 2021.

[9] Y. Sun, S. Jiang, W. Jia, and Y. Wang, "Blockchain as a cutting-edge technology impacting business: A systematic literature review perspective," *Telecommunications Policy*, vol. 46, no. 10, p. 102443, 2022.

[10] S. Curty, F. Härer, and H.-G. Fill, "Blockchain application development using model-driven engineering and low-code platforms: A survey," in *International Conference on Business Process Modeling, Development and Support*. Springer, 2022, pp. 205–220.

[11] D. Hindarto, "Blockchain-based academic identity and transcript management in university enterprise architecture," *Sinkron: jurnal dan penelitian teknik informatika*, vol. 8, no. 4, pp. 2547–2559, 2023.

[12] M. Jurgelaitis, R. Butkienė *et al.*, "Solidity code generation from uml state machines in model-driven smart contract development," *IEEE Access*, vol. 10, pp. 33 465–33 481, 2022.

[13] M. Jurgelaitis, L. Čeponienė, K. Butkus, R. Butkienė, and V. Drungilas, "Mda-based approach for blockchain smart contract development," *Applied Sciences*, vol. 13, no. 1, p. 487, 2022.

[14] M. Iqbal, A. Kormiltsyn, V. Dwivedi, and R. Matulevičius, "Blockchain-based ontology driven reference framework for security risk management," *Data & Knowledge Engineering*, p. 102257, 2023.

[15] I. Al-Azzoni and N. Petrović, "On persisting emf data using blockchains," in *2022 9th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2022, pp. 1–5.

[16] B.-J. Butijn, "From legal contracts to smart contracts and back again: Towards an automated approach," 2022.

[17] B. Gramlich, "Smart contract languages: A thorough comparison," *ResearchGate Preprint*, 2020.

[18] R. Rahimian and J. Clark, "Tokenhook: Secure erc-20 smart contract," *arXiv preprint arXiv:2107.02997*, 2021.

[19] T. A. Valerievitch, T. I. Vladimirovitch, K. J. Alexandrovitch, B. D. Andreevitch *et al.*, "Overview of the languages for safe smart contract programming," *Proceedings of the Institute of System Programming RAS*, vol. 31, no. 3, pp. 157–176, 2019.

[20] Y. Ait Hsain, N. Laaz, and S. Mbarki, "Ethereum's smart contracts construction and development using model driven engineering technologies: a review," *Procedia Computer Science*, vol. 184, pp. 785–790, 2021.

[21] G. Falazi, M. Hahn, U. Breitenbücher, and F. Leymann, "Modeling and execution of blockchain-aware business processes," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2, pp. 105–116, 2019.

[22] M. Skotnica and R. Pergl, "Das contract-a visual domain specific language for modeling blockchain smart contracts," in *Enterprise Engineering Working Conference*. Springer, 2019, pp. 149–166.

[23] A. B. Tran, Q. Lu, and I. Weber, "Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management." in *BPM (Dissertation/Demos/Industry)*, 2018, pp. 56–60.

[24] O. López-Pintado, L. García-Bañuelos, M. Dumas, and I. Weber, "Caterpillar: A blockchain-based business process management system." *BPM (Demos)*, vol. 172, 2017.

[25] J. Ladleif and M. Weske, "A unifying model of legal smart contracts," in *International Conference on Conceptual Modeling*. Springer, 2019, pp. 323–337.

[26] M. Hamdaqa, L. A. P. Met, and I. Qasse, "icontractml 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," *Information and Software Technology*, vol. 144, p. 106762, 2022.

[27] M. Jurgelaitis, L. čeponienė, and R. Butkienė, "Solidity code generation from uml state machines in model-driven smart contract development," *IEEE Access*, vol. 10, pp. 33 465–33 481, 2022.

[28] I. Al-Azzoni and N. Petrovic, "On persisting emf data using blockchains."

[29] A. Ayman, S. Roy, A. Alipour, and A. Laszka, "Smart contract development from the perspective of developers: Topics and issues discussed on social media," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 405–422.

[30] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[31] J. Xu, H. Liu, and Q. Han, "Blockchain technology and smart contract for civil structural health monitoring system," *Computer-Aided Civil and Infrastructure Engineering*, vol. 36, no. 10, pp. 1288–1305, 2021.

[32] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A security comparison with solidity based on common vulnerabilities," in *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2020, pp. 107–111.

[33] LLL, "Documentation for the LLL compiler — LLL Compiler Documentation 0.1 documentation," n.d. [Online]. Available: https://lll-docs.readthedocs.io/en/latest/

[34] Serpent, "GitHub - ethereum/serpent," n.d. [Online]. Available: https://github.com/ethereum/serpent

[35] Solidity, "Solidity — solidity 0.8.11 documentation," n.d. [Online]. Available: https://docs.soliditylang.org/en/v0.8.11/

[36] Vyper, "Vyper — Vyper documentation," n.d. [Online]. Available: https://vyper.readthedocs.io/en/stable

[37] Bamboo, "GitHub - pirapira/bamboo: Bamboo see https://github.com/cornellblockchain/bamboo," n.d. [Online]. Available: https://github.com/pirapira/bamboo

[38] M. Coblenz, "Obsidian: a safer blockchain programming language," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 97–99.

[39] Rholang, "Documentation page for rholang - rchain network," n.d. [Online]. Available: https://rchain-community.github.io/

[40] M.-T. L. of Tezos, "Michelson-The Language of Tezos," n.d. [Online]. Available: https://www.michelson.org/

[41] T. P. Platform and Marlowe, "The Plutus Platform and Marlowe — The Plutus Platform and Marlowe 1.0.0 documentation," n.d. [Online]. Available: https://docs.cardano.org/projects/plutus/en/latest

[42] æternity Sophia Language, "æternity sophia language," n.d. [Online]. Available: https://docs.aeternity.com/aesophia/v7.0.1/

[43] M. Knecht, "Mandala: a smart contract programming language," *arXiv preprint arXiv:1911.11376*, 2019.

[44] F. Schrans, D. Hails, A. Harkness, S. Drossopoulou, and S. Eisenbach, "Flint for safer smart contracts," 2019.

[45] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a smart contract intermediate-level language," *arXiv preprint arXiv:1801.00687*, 2018.

[46] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, "Low-code development and model-driven engineering: Two sides of the same coin?" *Software and Systems Modeling*, vol. 21, no. 2, pp. 437–446, 2022.

[47] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: A rapid development of dsm graphical editor," in *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 2014, pp. 233–238.

[48] F. Budinsky, R. Ellersick, D. Steinberg, T. J. Grose, and E. Merks, *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[49] P. Noizat, "Blockchain electronic vote," in *Handbook of digital currency*. Elsevier, 2015, pp. 453–461.

[50] K. Curran, "E-voting on the blockchain," *The Journal of the British Blockchain Association*, vol. 1, no. 2, p. 4451, 2018.

[51] Solidity, "Solidity by example — solidity 0.8.13 documentation," n.d. [Online]. Available: https://docs.soliditylang.org/en/v0.8.13/solidity-by-example.html voting

[52] Vyper, "Vyper by example — vyper documentation," n.d. [Online]. Available: https://vyper.readthedocs.io/en/stable/vyper-by-example.html voting