

SmartCML: A Visual Modeling Language to Enhance the Comprehensibility of Smart Contract Implementations

Simon Curty¹^[0000–0002–2868–9001]✉ and Hans-Georg Fill¹^[0000–0001–5076–5341]

University of Fribourg, Digitalization and Information Systems Group,
Bd de Pérolles 90, 1700 Fribourg, Switzerland
{simon.curty,hans-georg.fill}@unifr.ch
<https://www.unifr.ch/inf/digits/en/>

Abstract. One of the most notable capabilities of blockchain technology, exemplified by the Ethereum platform, is the decentralized execution of deterministic code, commonly referred to as smart contracts. This can be employed to develop business services that capitalize on the unique properties of blockchain technology, including the ability to maintain immutable, transparent, and persistent records on a distributed ledger. Nevertheless, even experts may find the process of writing smart contracts challenging. In addition to cost, optimization, and security concerns, it is essential to ensure that the smart contracts align with the business case and the associated rules. To address this issue, we propose *SmartCML*, a domain-specific visual modeling language to draft smart contracts, with the primary objective of facilitating the communication of the codified information among relevant stakeholders. The modeling language has been implemented using the ADOxx metamodeling platform. Smart contract models can then be transformed into fully functional code for the Ethereum virtual machine. The application of the modeling language is demonstrated with the help of two use cases.

Keywords: Blockchain · Smart contracts · Visual programming · Ethereum · Model-driven engineering · Domain-specific modeling language

1 Introduction

Blockchain technology is commonly related to cryptocurrencies and finance. This is largely due to the Bitcoin electronic cash system, conceived in 2008 [26], that popularized this technology. The combination of a decentralized ledger and cryptographic schemes enables the persistent and tamper-proof storage of transaction records without the necessity of a central controlling party [2,13]. These intrinsic properties of blockchains offer promising potential for the digital transformation of businesses, facilitating the emergence of novel business cases [20,32].

The advent of program code that is executed in a decentralized, transparent, and traceable manner has significantly enhanced the versatility and viability of

this technology for a multitude of business models. These so-called *smart contracts* were popularized by the Ethereum blockchain platform [3] and have since been adopted by many other blockchain implementations. Smart contracts allow for the realization of business services and novel decentralized business models that benefit from the aforementioned blockchain properties. Despite the potential for blockchain technology to disrupt existing business models and facilitate the development of innovative business cases, the adoption of this technology within organizational contexts is still hindered by a multitude of challenges.

Organizational barriers such as unavailability of financial and human resources, regulatory involvement, and lacking knowledge of the technology in the organization can prevent practical adoption [4]. The successful implementation of a blockchain-based business model requires the alignment of institutional, market, and technological factors, which can be achieved through a comprehensive approach that addresses the inherent complexity of these interrelated elements [21]. In addition, the complexity and relative immaturity of the technology’s ecosystem [15], as well as specific technical challenges [23], further impede its adoption. While ongoing technological advancements have mitigated some concerns, such as high energy consumption [11] and scalability issues [27], the implementation of blockchain-based applications and business services remains a demanding and complex endeavor. The intrinsic properties of blockchains present a double-edged sword for smart contract development. On the one hand, the persistent and transparent nature of the ledger, and by extension the code deployed, fosters security, data privacy, and functional issues. On the other hand, these very properties enable smart contracts in the first place. Previous research on methods to support technical smart contract development predominantly focus on the engineering challenges related to these issues. However, interrelation of business and technological factors in blockchain-based business cases remains sparsely researched [10]. This includes in particular the alignment of business services and their associated rules with smart contract implementations. To support this, we propose a modeling method to increase the comprehensibility of smart contracts and thereby facilitate communication among stakeholders and the alignment of the business case with the code. In particular, we present a domain-specific visual modeling language, named *SmartCML*, designed for use with Ethereum-compatible smart contracts. *SmartCML* is intended to serve two distinct functions. First, it can be used to facilitate communication regarding smart contracts for business cases. Second, it can be employed to generate code written in the Solidity language.

The remainder of the paper is structured as follows: In Section 2 we introduce the necessary foundations of blockchain technology and smart contracts, followed by an overview of previous modeling approaches that relate to this work. In Section 3, we present the extended domain-specific modeling language, including the requirements, metamodel and graphical notation. Section 3.4 presents a prototypical implementation of the language and code generator. In Section 4, we demonstrate the proposed language with two use cases. Finally, in Sections 5 and 6 we summarize our contribution and provide an outline for future research.

2 Foundations and Related Work

The following section presents a concise overview of blockchain technologies, with a particular focus on smart contracts. Further, it provides an analysis of previous research on visual modeling languages for the development of smart contracts.

2.1 Blockchain Technology

A blockchain consists of an electronic ledger that is organized in cryptographically linked blocks of digitally signed transactions between authorized parties. This ledger is stored decentralized on a peer-to-peer network. A consensus mechanism defines the rules on how transactions are validated, recorded, and propagated among the network nodes, thereby ensuring a consistent and valid state of the distributed ledger [2,13]. Depending on the specific blockchain implementation, transactions may include the transfer of funds, digital assets or, more generally, the storage of data. In some blockchain systems, transactions may also include executable code in the form of smart contracts to be stored on the ledger [1]. Subsequent invocation of such smart contracts results in the decentralized execution of deterministic code, which can be reenacted trustfully given the original parameters and environment. These smart contracts can be used to build business services, as part of decentralized applications, which leverage core blockchain properties, such as transparency of records and shared data access. These properties can be regarded as powerful disruptors to business models [6,20]. Blockchain-based smart contracts were popularized by the Ethereum blockchain technology [3] and its application on the publicly available network of the same name. Ethereum-compatible smart contracts are compiled into bytecode for the Ethereum Virtual Machine (EVM), which is the decentralized execution environment on the network nodes. Bytecode instructions are assigned a cost value that serves as a measure of the complexity of the code to be executed. In public networks, this cost is to be paid by the invoking party as part of transaction fees. Several other blockchains use the EVM as an execution environment¹, so that smart contracts written for Ethereum are portable. Smart contracts targeting the EVM are often written in Solidity, a high-level, statically typed programming language resembling JavaScript [30].

2.2 Visual Modeling Languages for Smart Contracts

The programming of smart contracts is subject to challenges that are not encountered in such a profound way in traditional development. This is partly due to the nature of blockchain technology and its idiosyncrasies, but also due to the intricacies of specific implementations, programming languages, and development practices [23]. This complexity renders smart contracts vulnerable to problems that are difficult to remedy once the contract is deployed. These include

¹ e.g., [Avalanche](#) and [Polygon](#)

security vulnerabilities that can result in the loss of funds, or poorly optimized code that leads unnecessary transaction costs [23]. In order to support the development of smart contracts and to address the challenges associated with them, several approaches have been discussed in the academic community, including visual modeling languages [18]. In the following, we highlight selected previous visual modeling languages for developing smart contracts and the subsequent generation of code, that are most relevant to the approach presented here. For a comprehensive overview of model-driven approaches to distributed application development, we refer to a recent literature review [10].

Previous visual modeling approaches for smart contracts are based either on an existing language, propose a domain-specific language or rely on a mixture of the two, for example, in the form of domain-specific extensions or profiles. An approach by Jurgelaitis et al. can be classified within the first group [22]. It utilizes UML class diagrams to model the structure of smart contracts. The behavioral logic is then modeled with UML state charts. The UML diagrams serve as input for the generation of executable Solidity code. The code generation of this approach is based on model transformations using the Eclipse ATL platform, with the main goal of facilitating the implementation process. Another approach relies on Petri nets to model smart contracts [35]. Here, the focus lies on the prevention of security issues at design time. This is achieved by first modeling contracts platform-independent and then simulating the resulting Petri net workflows in order to detect vulnerabilities such as deadlocks. Solidity template code can subsequently be generated from the workflows. The generated code may then serve as secure basis for further developments.

Designing a domain-specific language promises flexibility in terms of the representation of smart contract or blockchain platform-specific features. As such, several domain-specific languages for smart contract development have been proposed over the years. The modeling language iContractML aims to facilitate the development of language-independent smart contracts [17]. It features a visual notation for representing smart contracts in terms of participants, transactions, and assets at a high abstraction level. From the high-level models, the structure of smart contract code and partial behavioral logic can be derived using a template and transformation rule-based code generation paradigm. It allows to target multiple languages and platforms. Tan et al. presented a tool specifically targeting Solidity [29]. The modeling process relies on a combination of form-based definition of the structure of Solidity contracts and their implementation as action graphs. These are defined visually in a notation resembling flow-charts. From the definitions, Solidity code can be generated. A particular focus thereby lies on the estimation and optimization of execution costs. Based on Google's Blockly framework, SmartBuilder is a tool for visually programming smart contracts for the Hyperledger Fabric blockchain platform [24]. Code control structures and statements are represented as building blocks that can be combined together in a drag-and-drop fashion to compose smart contracts. This approach is intended to aid in learning smart contract coding.

Partially relying on Blockly as well, is the language Das Contract [28], in which Blockly is used to specify the behavior of a contract. Das Contract pertains to the last group of modeling approaches. For the conceptual representations of smart contracts it reverts to modified DEMO and BPMN models. Another approach combines UML class diagrams with a domain-specific language [19]. Thereby, language elements of the target smart contract language are mapped to UML class diagram elements. These diagrams are further complemented by models of operations in a domain-specific language. The models are then transformed into a platform-independent target language.

The modeling language presented in this work shares similarities with existing approaches, adopting several design decisions and ideas. However, previous work has mostly focused on facilitating the development activity in terms of model-based code generation or aims to make writing smart contracts more accessible to non-experts. Our approach instead seeks to enhance the comprehensibility of the implementation to facilitate its alignment with the business case. Therein, a focus lies on abstractions on the algorithmic level and explicit modeling of information access.

3 Domain-specific Language Design

The *SmartCML* language has been designed and developed in accordance with the macro process as outlined by Frank and considering principles of modeling method engineering [16,34]. The methodology comprises seven cyclical phases (micro processes), which serve as guidelines for designing a domain-specific modeling language (DSML). These phases are summarized as follows:

1. *Clarification of scope and purpose*: visual modeling of Ethereum-compatible smart contracts to enhance the comprehensibility of behavioral logic.
2. *Analysis of generic requirements*: These requirements are, in essence, applicable to every DSML. We revert to the catalog of generic requirements as outlined by Frank [16] and adopt them accordingly to our purposes as outlined in Section 3.1.
3. *Analysis of specific requirements*, that is, requirements that apply to the modeling artifact in particular. We will present these in Section 3.1.
4. *Language specification*: The metamodel will be presented in Section 3.2 in semi-formal notation. Formal specifications may be added later, e.g. using FDMM [14].
5. *Design and documentation of graphical notation*: The graphical notation of the visual smart contract modeling language will be shown in Section 3.3.
6. *Development of modeling tool*: The prototypical implementation of the language using the ADOxx metamodeling platform and the supplementary code generation are discussed in Section 3.4.
7. *Evaluation and refinement*: In accordance with the macro process, the modeling language and its constituent parts were subject to continuous evaluation and refinement cycles in alignment with the collected requirements. In Section 4 we show the applicability of the language for visually modeling smart contracts by means of two exemplary use cases.

3.1 Requirements

The macro process distinguishes between *generic* and *specific* requirements. Thereby, generic requirements may relate to appropriate conceptual representation of the target domain, levels of abstraction, or pragmatics such as comprehensibility and ease of use. We have specified 6 generic requirements (\mathbf{GR}_{1-6}) that are adopted from the catalog proposed by Frank [16]:

The visual smart contract language contains concepts that are familiar and recognizable to smart contract experts (\mathbf{GR}_1). The visual notation serves to differentiate between discrete concepts and is readily comprehensible to both experts and non-experts alike (\mathbf{GR}_2). The concepts of the language allow the modeling of smart contracts in such a way that common features of the Solidity language can be represented adequately (\mathbf{GR}_3). The modeling language includes all the essential concepts so that models can convey all the necessary information to be transformed into Solidity as the target representation (\mathbf{GR}_4). Additional concepts can be added via an extension mechanism to accommodate future smart contract features (\mathbf{GR}_5). In order to prevent the model from becoming overloaded and to ensure the correct interpretation, the language provides different levels of abstraction where appropriate (\mathbf{GR}_6).

We further derived six specific requirements (\mathbf{SR}_{1-6}), which detail the modeling artifacts' capabilities and features. The requirements have been formulated based on features of previous approaches and use case scenarios. The latter category encompasses tasks for which the method is considered to be applicable, as well as concrete smart contracts that one should be able to replicate in terms of functionality through the use of the language. The fundamental use scenario of the language is for the visual modeling of an Ethereum-compatible smart contract by an expert, with the objective of conveying implementation details to other stakeholders for the purpose of aligning the implementation and the business case. The particular specifications were as follows:

- \mathbf{SR}_1 Visual smart contracts can be transformed to executable Solidity code as target representation. The produced code is fully consistent with the behavioural logic and contract structure as defined in the corresponding visual contract model. Ethereum has been selected as the target platform due to its status as the dominant blockchain technology with smart contract capabilities in both academic and industrial contexts [10]. Moreover, selecting Solidity, which compiles to EVM bytecode, permits compatibility with blockchains that rely on the EVM as an execution environment in general.
- \mathbf{SR}_2 Common programming language structures and features, namely if-else conditions, while loops, parameterized functions, function calls, arithmetic and Boolean operations, and variable assignments can be modeled or represented equivalently. The rationale for this requirement is that the language must be sufficiently expressive to enable the representation of general algorithms. Furthermore, these concepts are intuitively known to programmers.
- \mathbf{SR}_3 Concepts related to Solidity-specific language features are included, such that these features can be modeled or derived from the model. In particular, this includes the emission of events and errors, conditional transaction

guards, data location of reference types and multiple return values. The mapping and array data structures, as well as user-defined structs shall be supported. This selection is informed by an analysis of over 400.000 Solidity smart contracts on their use of language features and structures. The languages have been obtained from the thousand most popular public GitHub repositories that contain Solidity code. Subsequently, these have been parsed and analyzed on a language grammar level.

- SR₄** Native solidity types, builtin functions, and custom complex composite types, such as structs and arrays of structs, can be defined by the user. These definitions are used in visual smart contracts and can be shared among them. This requirement relates to **GR₄** and **GR₆** in particular. Solidity is a statically typed language, and ideally, the required information for deriving correct types is present within smart contract models.
- SR₅** Interactions that result in a change of the contract state are explicitly denoted. Any write operation that commits data to the contract state storage is permanently recorded in the context of a transaction on the blockchain. As a result, such interactions are subject to transaction fees and the committed information becomes immutable.
- SR₆** The notation of the modeling language is suitable for drafting visual smart contracts with pen and paper. The rationale here is that a strict requirement for a tool would compromise one of the core goals of the modeling language, which is to facilitate communication, since the method could not be used as easily in workshops, on whiteboards, and so on.

3.2 Metamodel

Based on the formulated requirements the metamodel of *SmartCML* was specified in seven steps (**MM₁₋₇**), as shown in Figure 1. The language is inherently flow-based and follows a similar paradigm as process languages and transition systems. The fundamental concept is based on the representation of smart contracts as a set of interfaces, with each interface accompanied by a graph of operations that can be regarded as a transition system. In the following, a detailed examination of the individual components of the metamodel will be presented. For the sake of clarity, terms in *italics* relate to their respective metamodel class.

- MM₁** The language elements are assigned to two distinct *Model Types*, namely the *Definitions Map* and the *Visual Smart Contract*. The former is comprised of shared definitions of types and builtins. The latter models a single contract in terms of structure, state, behavioral logic, and communication interfaces. A *Visual Smart Contract* model may reference elements of one or several definition maps and definition maps can be shared among models. This segregation relates in particular to **GR₆**.
- MM₂** A *Definitions Map* model defines *Builtin Functions* that are part of the execution environment, and data types. A type is either a *Struct*, *Mapping*, *Array* or *Basic Type* (**GR₄**, **SR₄**). The former three reference further types. E.g., a mapping has a key and a value type. Elements are defined by the

- MM₃** *Boundaries* define the bounds of a contract in terms of communication. An *Interface* defines a scope of execution with defined input and output. The *Function Interface* models a contract function, while the *Proxy Interface* is used to relate other interfaces or *Builtin Functions*. Further, *Emitters* allow for the signaling of *Events* and *Failures* during executions (**GR₃**, **SR₂**, **SR₃**).
- MM₄** An *Action* represents a discrete unit of work within a flow. This may take the form of an arithmetic or Boolean *Operation*, a branching *Condition*, the *Declaration* of an instance, a *Function Call* or the *Emission* of a signal (**SR₂**, **SR₃**). The *Code Block* element accommodates the inclusion of logic that cannot be modeled otherwise or whose explicit representation would be inappropriate for the targeted abstraction level (**GR₃**, **GR₅**, **GR₆**). One potential usage example is the incorporation of EVM assembly code.
- MM₅** *Actions* and *Boundaries* are *Flow Elements*, i.e., are within a flow. Each such element has a number of *Ports* that define the available outgoing relations. The *Flow* relation allows to specify a subsequent *Flow Element* from an outgoing *Flow Port*. Elements may differ in what ports they have. All *Actions* have an *Initial* and a *Final Port*, while a *Condition* additionally has a *True* and *False Port*. Each *Port* has a priority that defines in which order operations are to be executed. The sequence in which ports are activated and the action is performed is as follows: initial, action execution, mid or call, true, false, and final. This allows to model branching logic with well-defined sequences of actions while accommodating for some flexibility in modeling (**GR₁**, **GR₄**, **SR₂**).
- MM₆** The *Storage* class represents a state variable of a contract. To be available within an action, state variables must explicitly be accessed with the *Access* relation. Furthermore, the relation indicates the nature of the access, whether it is a read or write operation. As a result, contract functions that access state variables or modify them can be readily identified (**GR₄**, **SR₃**, **SR₅**).
- MM₇** The *Call* relation delineates the invocation of a function from the outgoing port. This may be in the form of an internal function call; in which case the target element is a *Function Interface* in the same model. Alternatively, the call may target a *Proxy Interface*, referencing either a *Builtin Function* or some external function (**GR₁**, **GR₃**, **SR₂**).

3.3 Graphical Notation

The concrete subclasses of the *Element* and *Relation* metamodel classes are available as modeling elements in one of the model types. As such, these have a designated graphical representation. In Table 1, a summary of the elements of the *Visual Smart Contract* model type is shown. The design of the graphical notation concerns mainly the requirements **GR₂** and **SR₆**. The visual language is based on simple geometric shapes where related elements share design elements. Each *Action* type is denoted with a distinctly decorated circle and has an annotation that designates what the action represents. For example, a *Condition* could have the annotation «if» or «while», depending on whether the element translates to an if-else statement or a while loop. *Boundary* elements, that is,

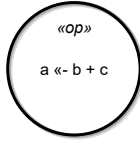
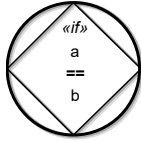
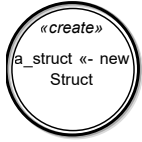
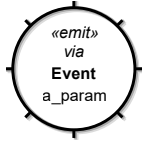

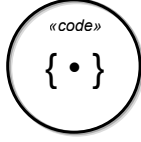
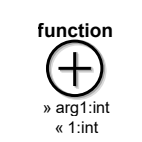
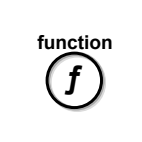


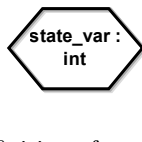
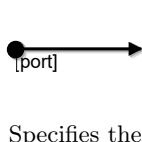
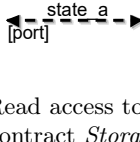
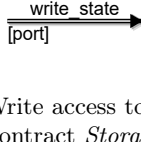
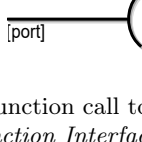
Operation  Arithmetic or Boolean operation with assignment	Condition  Branching control structure: if-else, while	Declaration  Declaration of an instance variable, e.g., for a struct	Emission  Emission of data via a boundary element, e.g., an event
Function Call  Invocation of an internal, external or builtin function	Code Block  Inclusion of additional code such as EVM assembly	Function Interface  Definition of a parameterized contract function	Proxy Interface  Reference to a builtin function or an external function
Event  Definition of an event that carries outgoing data	Failure  Definition of a failure that when triggered reverts a transaction	Storage  Definition of a typed smart contract state variable	Flow Relation  Specifies the subsequent action or boundary from the outgoing port
Access Relation (read)  Read access to a contract <i>Storage</i> element	Access Relation (write)  Write access to a contract <i>Storage</i> element	Call Relation  Function call to a <i>Function Interface</i> or a <i>Proxy Interface</i>	

Table 1: Graphical notation of the modeling elements that comprise a *SmartCML Visual Smart Contract* model.

Interfaces and *Emitters*, are uniquely named as they represent a distinct definition in the smart contract, e.g., of an error. Further, *Boundaries* display what

data is transmitted in the form of parameters and return values. Each relation shows the port from which it is triggered. The visualization of the *Access* relation is based on the access mode, and additionally denotes the variables read from, or written to state, represented by the *Storage* element. The straightforward shapes and the subtle visual design elements enable models to be drafted by hand (**SR₆**). The descriptive elements, such as the annotations, and the distinct visual representation facilitates comprehensibility and the differentiation of the concepts (**GR₂**).

3.4 Implementation

The proposed domain-specific modeling language has been developed and implemented as a prototype using the ADOxx metamodeling platform [12]. ADOxx was selected for its maturity, acceptance in both academic and industrial contexts, and suitability for the prototyping of modeling methods. It has previously been successfully employed for the implementation of modeling languages with varying purposes [6,25]. In order to implement a metamodel for a custom modeling language, it is necessary to extend the ADOxx metamodel in the development toolkit and export it as a library. In order to facilitate this process, the ADOxx metamodel provides a set of pre-defined classes, relations, and attribute types that can be leveraged to simplify implementation. ADOxx further offers the capability to specify model types for which specific elements and relations are available (**MM₁**). The *Flow*, *Access*, and *Call* relations (**MM₅₋₇**) are realized as relation classes. However, ADOxx does not natively support the concept of ports. To circumvent this limitation, outgoing ports are imitated by an attribute of the relation classes. The references to *Definitions* (**MM₂**) are realized with attributes of the *Interref* type, which allows to link instances of elements across models. The linked *Type Definitions* are then leveraged by a type-checking system to verify that the typed variables are being utilized in a consistent manner. The typing system is implemented with the internal scripting language and constitutes an integral part of the modeling library. The code generator, on the other hand, is implemented as a separate Node.js application that takes ADOxx models exported as XML as input. The XML models are transformed into a syntax tree that conforms to the Solidity grammar, and the code is then generated from this syntax tree. The *SmartCML* modeling library for ADOxx and the Solidity code generator are openly available [7].

4 Exemplary Use Cases

In accordance with the macro process, the modeling language was subject to continuous evaluation and refinement cycles. Among other measures, the continuous analysis of use cases contributed to the refinement of the method. The application of the modeling language for the visual design of smart contracts is demonstrated through the analysis of two use cases.

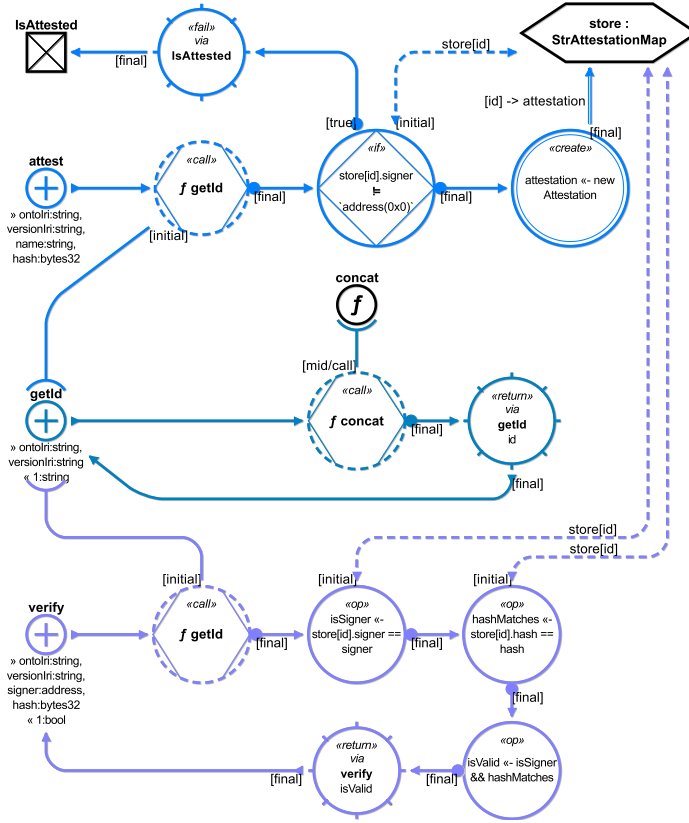


Fig. 2: *Visual Smart Contract* model for recording and verifying attestations of OWL ontologies.

4.1 Ontology Attestation

The decentralized attestation of information is a common topic of discussion with regard to the potential applications of blockchain technology. The persistence of records ensures the immutability of previous attestations, without the need to employ additional technological measures. This particular use case revolves around the proposal to utilize blockchain technology and smart contracts for the purpose of attesting to the provenance of OWL ontologies. As illustrated in Figure 2, the visual smart contract aligns with an architectural framework for this specific purpose that has been previously proposed [8].

This visual smart contract contains three *Function Interfaces* that correspond to the contract functions *attest*, *verify*, and the helper function *getId*. A specific version of an OWL ontology is identified by the ontology IRI and a version IRI. Accordingly, attestations are mapped to ontologies using these identifiers. The *getId* function constructs a key from the identifiers by calling a builtin function *concat*, that concatenates strings. To record a new attestation, the *attest*

function takes as input the identifiers, a name, and a hash of the ontology. A unique key is obtained through an internal invocation of *getId*. Subsequently, it is checked whether an attestation with the same key already exists by reading from the state storage mapping. The mapping maps the string keys to structs that contain the attestation information such as the address of the signer. If the signer is already set for this key, the ontology is already attested and an error *IsAttested* is emitted, and the transaction reverted as a result. Otherwise, a new attestation struct is created and stored in the mapping. Listing 1 shows the code that is generated from this part of the model. Similarly, given an ontology version's identifiers, its hash, and the address of the supposed signer, it can be verified whether this version has been attested by the given signer.

```
function attest(string calldata ontoIri, string calldata versionIri, string
    calldata name, bytes32 hash) public {
    string memory id = getId(ontoIri, versionIri);
    if (store[id].signer != address(0x0)) {
        revert IsAttested();
    }
    Attestation memory attestation = Attestation(msg.sender, name, block.
        timestamp, hash);
    store[id] = attestation;
}
```

Listing 1: Generated solidity code that corresponds to the *attest* function interface flow.

4.2 Decentralized Auctions

In order to demonstrate the viability of proposed languages for the visual modeling of more complex smart contracts, we revert to the use case of decentralized auctions [5]. The smart contract that corresponds to the model illustrated in Figure 3, allows the posting of auctions for items with a fixed duration. In this context, the blockchain fulfills the function of executing the auction process and recording of the involved bids. This enables the reenactment of the auction process in the event of disputes without the necessity of a trusted third party [5]. It should be noted that, in this illustration, the smart contract does not serve as a payment channel; however, it could be adapted to include this capability.

Auctions are stored as structs in an array, represented as the *auctions Storage* element. Publishing a new auction simply involves appending a new auction struct to the array, represented by the first *Declaration* action and subsequent write *Access* relation to the *Storage* element. The index of the new array element is then returned and serves as future identifier for the auction. The visualization allows to easily identify each access to the state storage. The distinction between read and write access further allows to identify at a glance which functions may alter the contract state, and thus require to be invoked in a transaction context. That is, posting, bidding and closing an auction may change the state and is thus recorded on the blockchain.

Furthermore, the boundary elements serve as clearly identifiable points of data exchange that may be further processed by applications. The emitters

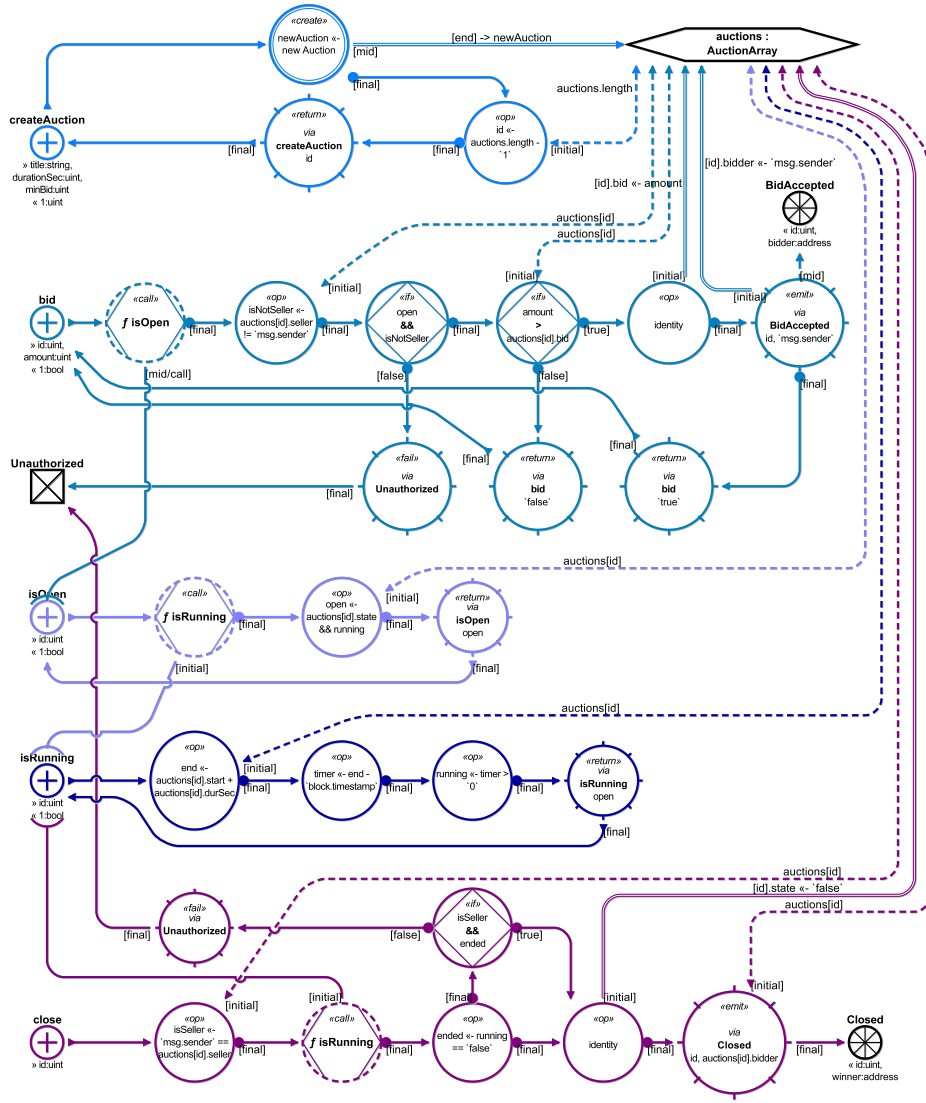


Fig. 3: Visual Smart Contract model for decentralized, timed auctions.

Unauthorized, *BidAccepted*, *Closed* are shared across flows. That is, these elements, same as the *Storage* are defined on the contract scope. This is indicated by the black coloring of these elements.

5 Discussion

Domain-specific languages are particularly well-suited for addressing the complexities inherent to the adoption of blockchain technology. This is due to the fact that intrinsic properties can be captured in great detail within the context of such languages. Thus, several modeling methods have been proposed that aim to address the many inherent challenges. Common objectives of modeling methods are related to security concerns [31] or model-driven development [9,33], while methods to support the alignment of blockchain technology and business aspects remains sparsely researched [10]. This work aims to leverage the advantages of model-driven engineering methods for the development of smart contracts, while providing a visualization of the implementation that facilitates communication of the behavioral logic. As such, the presented *SmartCML* may serve as a tool to discuss smart contract implementations among stakeholders. To achieve this, the modeling language has been rigorously designed following the *macro process* [16]. The demonstration underlines the languages applicability for visually modeling smart contract that can be transformed into executable Solidity code, without the need of any further manual programming ($\mathbf{GR}_{1,3,4}$, \mathbf{SR}_{1-4}). An emphasis in the language’s design has been placed on the explicit modeling of smart contract states and which interaction cause a state change (\mathbf{SR}_5). The immediate practical benefit is the straightforward identification of interaction in a transaction context and all consequences thereof. Moreover, this offers the opportunity to further link business models with smart contract implementations regarding information spaces, e.g., for ensuring fulfillment of regulatory requirements. The graphical notation is based on simple shapes, which allows for drafting models by hand (\mathbf{SR}_6), thus facilitating use on whiteboards.

6 Conclusion and Future Work

In this paper we presented *SmartCML*, a domain-specific modeling language for the visual programming of Ethereum-compatible smart contracts. A focus has been put on a simple graphical notation that aims to facilitate the comprehensibility of the implementation and explicit representation of state changing interactions. The visual smart contract model can be transformed to readily deployable and executable smart contract code. Opportunities for future work include the integration with business modeling approaches, e.g., e³value, model-based formal verification methods, and the specification of modeling procedures. Further evaluations, for example in the form of expert interviews or ontological analysis, are required to validate the language beyond the initial feasibility demonstration.

Acknowledgments. This work was supported by the Swiss National Science Foundation project Domain-Specific Conceptual Modeling for Distributed Ledger Technologies [196889].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'reilly Media (2018)
2. Banafa, A.: Introduction to Blockchain Technology. River Publishers, New York (Jul 2023). <https://doi.org/10.1201/9781003426264>
3. Buterin, V.: A Next-Generation Smart Contract and Decentralized Application Platform (2013), <https://ethereum.org/en/whitepaper/>
4. Clohessy, T., Acton, T., Rogers, N.: Blockchain Adoption: Technological, Organisational and Environmental Considerations. In: Treiblmaier, H., Beck, R. (eds.) Business Transformation through Blockchain, pp. 47–76. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-319-98911-2_2
5. Curty, S., Fill, H.G.: Exploring the Systematic Design of Blockchain-based Applications Using Integrated Modeling Standards. In: Bork, D., Barat, S., Asprion, P.M., Marcelletti, A., Morichetta, A., Schneider, B., Kulkarni, V., Breu, R., Zech, P. (eds.) Proceedings of the PoEM 2022 Workshops and Models at Work Co-Located with Practice of Enterprise Modelling 2022, London, United Kingdom, November 23–25, 2022. CEUR Workshop Proceedings, vol. 3298. CEUR-WS.org (2022)
6. Curty, S., Fill, H.G.: A Domain-Specific e3value Extension for Analyzing Blockchain-Based Value Networks. In: Almeida, J.P.A., Kaczmarek-Heß, M., Koschmider, A., Proper, H.A. (eds.) The Practice of Enterprise Modeling, vol. 497, pp. 74–90. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-48583-1_5
7. Curty, S., Fill, H.G.: SmartCML ADOxx Application Library and Code Generator (PoEM 2024) (Oct 2024). <https://doi.org/10.5281/zenodo.13899102>
8. Curty, S., Fill, H.G., Gonçalves, R.S., Musen, M.A.: An Architecture for Attesting to the Provenance of Ontologies Using Blockchain Technologies. In: Shishkov, B. (ed.) Business Modeling and Software Design. pp. 182–199. Lecture Notes in Business Information Processing, Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-11510-3_11
9. Curty, S., Härer, F., Fill, H.G.: Blockchain application development using model-driven engineering and low-code platforms: A survey. In: Enterprise, Business-Process and Information Systems Modeling. pp. 205–220. Springer International Publishing, Cham (2022)
10. Curty, S., Härer, F., Fill, H.G.: Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: A structured literature review. Software and Systems Modeling (Jun 2023). <https://doi.org/10.1007/s10270-023-01109-1>
11. Fernando, Y., Saravannan, R.: Blockchain Technology: Energy Efficiency and Ethical Compliance. Journal of Governance and Integrity 4(2), 88–95 (Mar 2021). <https://doi.org/10.15282/jgi.4.2.2021.5872>
12. Fill, H.G., Karagiannis, D.: On the Conceptualisation of Modelling Methods Using the ADOxx Meta Modelling Platform. Enterprise Modelling and Information Systems Architectures (EMISAJ) 8(1), 4–25 (2013). <https://doi.org/10.18417/emisa.8.1.1>
13. Fill, H.G., Meier, A. (eds.): Blockchain: Grundlagen, Anwendungsszenarien und Nutzungspotenziale. Edition HMD, Springer Fachmedien, Wiesbaden (2020). <https://doi.org/10.1007/978-3-658-28006-2>
14. Fill, H., Redmond, T., Karagiannis, D.: Formalizing meta models with FDMM: the adoxx case. In: Enterprise Information Systems - 14th International Conference,

- ICEIS 2012, Wroclaw, Poland, June 28 - July 1, 2012, Revised Selected Papers. Lecture Notes in Business Information Processing, vol. 141, pp. 429–451. Springer (2012). https://doi.org/10.1007/978-3-642-40654-6_26
15. Flovik, S., Moudnib, R.A., Vassilakopoulou, P.: Determinants of Blockchain Technology Introduction in Organizations: An Empirical Study among Experienced Practitioners. *Procedia Computer Science* **181**, 664–670 (2021). <https://doi.org/10.1016/j.procs.2021.01.216>
 16. Frank, U.: Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines. In: *Domain Engineering: Product Lines, Conceptual Models, and Languages*, pp. 133–157. Springer (May 2013). https://doi.org/10.1007/978-3-642-36654-3_6
 17. Hamdaqa, M., Met, L.A.P., Qasse, I.A.: iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology* **144**, 106762 (2022). <https://doi.org/10.1016/j.infsof.2021.106762>
 18. Härer, F., Fill, H.G.: A Comparison of Approaches for Visualizing Blockchains and Smart Contracts. *Jusletter IT Weblaw*, ISSN 1664-848X **21 February 2019** (Feb 2019). <https://doi.org/10.5281/zenodo.2585575>
 19. Heckel, R., Erum, Z., Rahmi, N., Pul, A.: Visual Smart Contracts for DAML. In: *Graph Transformation*. vol. 13349, pp. 137–154. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-09843-7_8
 20. Iansiti, M., Lakhani, K.R.: The Truth About Blockchain. *Harvard business review* **95**(1), 118–127 (2017)
 21. Janssen, M., Weerakkody, V., Ismagilova, E., Sivarajah, U., Irani, Z.: A framework for analysing blockchain technology adoption: Integrating institutional, market and technical factors. *International Journal of Information Management* **50**, 302–309 (Feb 2020). <https://doi.org/10.1016/j.ijinfomgt.2019.08.012>
 22. Jurgelaitis, M., Ceponiene, L., Butkiene, R.: Solidity code generation from UML state machines in model-driven smart contract development. *IEEE Access* **10**, 33465–33481 (2022). <https://doi.org/10.1109/ACCESS.2022.3162227>
 23. Kannengiesser, N., Lins, S., Sander, C., Winter, K., Frey, H., Sunyaev, A.: Challenges and common solutions in smart contract development. *IEEE Transactions on Software Engineering* (2021). <https://doi.org/10.1109/TSE.2021.3116808>
 24. Merlec, M.M., Lee, Y.K., In, H.P.: SmartBuilder: A block-based visual programming framework for smart contract development. In: *2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021*. pp. 90–94. IEEE (2021). <https://doi.org/10.1109/Blockchain53845.2021.00023>
 25. Muff, F., Fill, H.G.: A Domain-Specific Visual Modeling Language for Augmented Reality Applications Using WebXR. In: Almeida, J.P.A., Borbinha, J., Guizzardi, G., Link, S., Zdravkovic, J. (eds.) *Conceptual Modeling*. pp. 334–353. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-47262-6_18
 26. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), <https://web.archive.org/web/20140320135003/https://bitcoin.org/bitcoin.pdf>, [last accessed: 2024-07-24]
 27. Nguyen, C.T., Hoang, D.T., Nguyen, D.N., Niyato, D., Nguyen, H.T., Dutkiewicz, E.: Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities. *IEEE Access* **7**, 85727–85745 (2019). <https://doi.org/10.1109/ACCESS.2019.2925010>
 28. Skotnica, M., Pergl, R.: Das contract - A visual domain specific language for modeling blockchain smart contracts. In: *Advances in Enterprise Engineering*

- XIII - 9th Enterprise Engineering Working Conference, EEWC 2019, Lisbon, Portugal, May 20-24, 2019, Revised Papers. Lecture Notes in Business Information Processing, vol. 374, pp. 149–166. Springer (2019). https://doi.org/10.1007/978-3-030-37933-9_10
29. Tan, S., Bhowmick, S.S., Chua, H.E., Xiao, X.: LATTE: Visual construction of smart contracts. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, Online Conference [Portland, OR, USA], June 14-19, 2020. pp. 2713–2716. ACM (2020). <https://doi.org/10.1145/3318464.3384687>
 30. The Solidity Team: Solidity — Solidity 0.8.26 documentation. <https://docs.soliditylang.org/en/v0.8.26/>, [last accessed: 2024-07-24]
 31. Tolmach, P., Li, Y., Lin, S., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. *ACM Comput. Surv.* **54**(7), 148:1–148:38 (2022). <https://doi.org/10.1145/3464421>
 32. Treiblmaier, H., Clohessy, T. (eds.): Blockchain and Distributed Ledger Technology Use Cases: Applications and Lessons Learned. Progress in IS, Springer International Publishing, Cham (2020). <https://doi.org/10.1007/978-3-030-44337-5>
 33. Varela-Vaca, Á.J., Quintero, A.M.R.: Smart contract languages: A multivocal mapping study. *ACM Comput. Surv.* **54**(1), 3:1–3:38 (2021). <https://doi.org/10.1145/3423166>
 34. Visic, N., Fill, H., Buchmann, R.A., Karagiannis, D.: A domain-specific language for modeling method definition: From requirements to grammar. In: IEEE RCIS 2015. pp. 286–297. IEEE (2015). <https://doi.org/10.1109/RCIS.2015.7128889>
 35. Zupan, N., Kasinathan, P., Cuellar, J., Sauer, M.: Secure Smart Contract Generation Based on Petri Nets. In: Blockchain Technology for Industry 4.0, pp. 73–98. Springer Singapore, Singapore (2020). https://doi.org/10.1007/978-981-15-1137-0_4