IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 12, 2022, accepted March 19, 2022, date of publication March 25, 2022, date of current version March 31, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3162227

Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development

MANTAS JURGELAITIS, LINA ČEPONIENĖ, AND RITA BUTKIENĖ
Department of Information Systems, Faculty of Informatics, Kaunas University of Technology, 51368 Kaunas, Lithuania
Corresponding author: Mantas Jurgelaitis (mantas.jurgelaitis@ktu.lt)

ABSTRACT For the development of blockchain smart contracts, a structured approach based on the principles of the Model Driven Architecture can be beneficial and facilitate the implementation of smart contracts. This paper presents such an approach, which, in combination with Unified Modeling Language (UML) Class and State machine diagrams, allows the smart contract structure and behavior logic to be modeled in several abstraction layers. This paper delves into details on how the model-to-model transformations from the specified Blockchain Platform Independent Model (PIM) with specified state-like behavior can be used to produce a Solidity Platform Specific Model (PSM). Subsequently, we elaborate on how the Solidity PSM is used for Solidity smart contract code generation by employing model-to-text transformations. The paper also demonstrates the process of our proposed transformations and code generation using smart contract code examples from Solidity documentation. Based on the examples, a Blockchain PIM is specified and transformed to Solidity PSM, which is then used for Solidity smart contract code generation. The generated smart contract code is then compiled, deployed on the Ethereum blockchain JavaScript virtual machine, and compared to the original smart contract code in terms of Solidity code metrics, similarity scores, and execution costs. The evaluation results indicate that our approach could be successfully used to model and later generate smart contract code.

INDEX TERMS Blockchain, model driven architecture, model-driven development, smart contracts, solidity, state machine, unified modeling language.

I. INTRODUCTION

Blockchain technology enables the utilization of distributed networks for decentralizing data storage and promotes trust and transparency by employing common ledger and consensus algorithms. Currently, blockchain technology is being used to decentralize various business processes using programs deployed on a blockchain network called smart contracts [1]. These smart contracts allow automatic execution, control, and documentation of relevant events and actions [2]. Although blockchain technology and smart contracts are being integrated into various information systems, the adoption rate is still relatively slow [3]. The use of blockchain comes with a number of advantages, but its implementation is accompanied by development difficulties, such as dealing with the immutability of the smart contract, the immaturity of the technology, and its rapid change. Consequently, blockchain would be made more accessible and widely used by providing tools for the more straightforward implementation of blockchain solutions [3].

Unfortunately, there is no standard way to develop decentralized solutions for business processes, making the smart contract development process quite difficult and ambiguous. The solutions are prone to human errors since, unlike traditional software development, the software code deployed to the blockchain cannot be modified further. Due to the architectural differences with other technologies, especially the immutability of smart contracts, code copying and reusing have become prevalent in the development of blockchain solutions [4]. Continuing the trend, the community-based project EIP [5] aims to provide smart contracts standards as building blocks or code samples to deal with application-level conventions, such as token standards, name registries, wallet formats, etc. These standardized code samples could,

The associate editor coordinating the review of this manuscript and approving it for publication was Bing Li.

in turn, help developers create their custom solutions. EIP participants like OpenZeppelin [6] provide reusable Solidity components or implementation examples of collectively developed ERC standards. The provided standard code samples can be used in their current state or adapted for a specific application. This adaptation is currently performed manually, requiring the developer to invest time in code understanding, modification, and testing. The process could be alleviated using the principles of model-driven development (MDD) and code generation [7]–[9].

The principles of model-driven development are elaborated in the Model Driven Architecture (MDA) framework proposed and standardized by OMG [10]. MDA provides guidelines for structuring the development process by utilizing models at different levels of abstraction. MDA encompasses three different models: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). These models are transformed in the sequence provided and ultimately used to generate platform specific source code [11]. Combining the MDA principles and the smart contract development practices of using standard code samples, a structured development approach can be provided, which can facilitate the implementation of smart contracts and decrease the number of errors arising during the manual development. Furthermore, according to MDA, PIM can be developed to be generic enough to include fundamental principles of blockchain technology and allow the transition to PSMs for different implementation platforms when required.

The official standard for modeling in MDA is the Unified Modeling Language (UML) [12], which encompasses several types of diagrams for specifying the structure and behavior of the system under development. UML can be successfully used for smart contract modeling, which can increase the readability, comprehensibility, and verifiability of the developed software. The most persuasive example of the usefulness of UML in smart contract development can be the inclusion of the sol2uml tool in the Etherscan environment [13], which enables reverse engineering of the deployed smart contract into the UML class diagram to help the developer understand the structure of the analyzed smart contract. Although structural UML diagrams, like class diagrams, can facilitate the generation of smart contract boilerplate code for selected implementation language, the behavioral UML diagrams, like sequence, activity, state machine diagrams, can be even more useful in the generation of complete executable smart contract code.

Another important aspect for facilitating smart contract development is the application of reusable architectural solutions, called design patterns, which assist developers in solving common architectural problems. Various smart contract design patterns exist [14] and can be used in manual smart contract development and incorporated at a higher level of abstraction – UML models. Using the smart contract design patterns together with MDA principles, structural boilerplate code could be generated and behavioral code samples adapted and integrated into the software code. One such pattern, the state machine pattern [15], enables a smart contract to go through states by exposing different functionality based on a specific state. Similar software patterns are quite common in software engineering, and proposals for utilizing state machine for smart contract development purposes already exist [16]–[18], together with approaches that use a state machine to generate smart contract code [19], [20].

For smart contracts, the concept of state is an important construct [15] and in the development of smart contracts, state machines can be useful in many aspects. Moreover, state machines are already used in the verification of smart contract behavior against functional requirements [17], detecting security issues [16], avoiding deadlock scenarios, detecting vulnerabilities, and facilitating the specification of smart contract properties [21]. The state machine can be represented and modeled using various notations, amongst which are UML state machine diagrams. The UML state machine diagram can define event-driven behavior by specifying different states and transitions between them. The UML state machine diagram can be part of the overall smart contract specification in PIM or PSM according to MDA principles [22]. Thus, UML models, code samples, and software design patterns can be combined to produce code and facilitate the process of smart contract development.

Our research aims to demonstrate that the principles of MDA and UML diagrams can be successfully applied to facilitate the development of smart contracts for blockchain. In this paper, we present the algorithms for transformation from the Platform Independent Model (PIM) to the Ethereum Solidity Platform Specific Model (PSM) and Solidity smart contract code generation from the specified PSM. Our approach in combination with UML allows the smart contract structure and behavior logic to be modeled in a few different perspectives and combined to generate code. The algorithms developed are part of a broader methodology under development [23] that encompasses the whole MDA process of transformations from CIM to PIM, then to PSM, and finally to source code for a chosen blockchain platform. Adherence to the MDA principles allows extending the approach with different technologies at the PSM level. In our previous work [24], we demonstrated the approach to generate the Go chaincode source code from PSM for Hyperledger Fabric.

The goal of this paper is to place a greater focus on the Ethereum platform and propose a Solidity code generation solution based on MDA and a set of UML diagrams. Our approach is based on MDA for enabling the developers to focus on the structural and behavioral design of smart contracts, rather than on technical details of the specific blockchain platform. Additionally, we aim to facilitate common understanding of smart contract elements amongst different blockchain platform developers by providing guidelines for development of Blockchain Platform Independent Model. A greater focus is put on specification of smart contract behavior using UML state machine. The main

contribution of the paper is the outlined algorithms for model transformations and generating executable smart contracts. The generated code encompasses not only the structural elements in terms of boilerplate code, but also additional behavioral logic, like Solidity conditional statements, state declarations, modifiers, and Solidity specific state machine design patterns. This facilitates the development of smart contract, which is more resilient, as the contents of the contract are revised in graphical form more than once during the transformations and furthermore the contract can be straightforwardly transferred to a different platform if required.

The proposed algorithms were implemented in Eclipse ATL [25] and Acceleo [26] tools for model-to-model (M2M) and model-to-text (M2T) transformations and used to generate three different smart contracts, based on the examples provided in Solidity documentation. The generated code was successfully deployed on a sandbox blockchain in a JavaScript virtual machine environment and compared with the original smart contract counterparts.

The remainder of the paper is structured as follows. In the second section, the related work is overviewed. In the third section, a proposal is outlined, Blockchain PIM to Solidity PSM transformation, and Solidity smart contract code generation algorithms are presented. The fourth section presents the results of the application of the algorithm implementation to specific smart contract specifications and the generated code evaluation metrics. Lastly, conclusions and future work are outlined.

II. RELATED WORK

The area of applying model-driven development techniques for smart contract code generation has been under active development during recent years [8], [27], [28]. Model-driven approaches are applicable in smart contract development, as they provide a valuable abstraction [27], [29]. This abstraction not only facilitates the understanding of software, but also helps to introduce validation/verification at earlier stages. Additionally, the use of models at different abstraction levels enables the transformation of the same model into source code for multiple platforms, which can reduce development times and costs [11], [18]. One of the key arguments for using MDD and code generation approaches in smart contract development is blockchain immutability and the need to verify smart contracts before deployment. Once deployed, the smart contract code cannot be updated [8], [30].

Smart contract development is a relatively novel field, so no unified approach exists. In recent years, model-driven development approaches have become more prominent for smart contract development, as they provide systemized design practices. The scientific community aims to provide a unified software development lifecycle, as in terms of smart contract development, Software Development Life Cycle (SDLC) phases are often unlinked [18].

The focus on the systemized process of smart contract development is placed in [27]. The authors propose the Smart Contract Engineering (SCE) approach, which provides the developer with the guidelines for designing, verifying, automatically generating, and testing the smart contract code. In this approach, ATL model transformations are proposed for transformations between formal models and for generating smart contract source code. Whereas the authors describe the method in detail, the presented case study is focused only on the formal verification of smart contract code using an event-based approach.

Another proposed approach for a structured development process for DApps (decentralized applications, which encompass smart contracts) is presented in [29]. The authors developed an Agile development process called ABCDE (Agile Block Chain DApp Engineering) for implementing DApps for blockchain. The approach is tailored for the Ethereum platform, Solidity programming language. UML class, use case, and sequence diagrams are used to describe smart contract and system actor interactions. Moreover, UML class and sequence diagrams are extended by using specific Solidity stereotypes. ABCDE requires the developer to outline system goals, determine actors, write user stories, develop use cases, divide the functionality between the smart contract and application as well as to specify the smart contract structure. Additionally, ABCDE provides a security checklist to evaluate security and gas consumption costs during the development of the DApp.

In addition to the proposed structured approaches, numerous less broad proposals exist that do not cover the entirety of SDLC but try to incorporate different existing notations for modeling the smart contract structure and behavior. The proposals encompass the usage of UML class diagrams [31], [32], UML sequence diagrams [29], UML state diagrams [20], UML deployment diagram [33], finite state machines (FSM) [28], BPMN [30], [34], [35], and even custom domain specific languages (DSL) [36]–[38] in smart contract development. In our research, we focus on transformation from models to smart contract code, and for this reason, we have analyzed in more detail several of the aforementioned proposals [30], [34], [35], [37], [39] that place a heavier focus on the usage of models for smart contract code generation purposes.

iContractML [37] provides a DSL for modeling the smart contract structure and the generation of smart contracts for multiple platforms, including Ethereum, Microsoft Azure, and Hyperledger Composer. Authors use Acceleo MTL (Model to Text Transformation Language) for code generation purposes and provide transformation templates for each analyzed blockchain implementation platform.

An approach for auto-generating smart contracts is presented in [39] that uses OWL (Web Ontology Language) ontology specifications and SWRL (Semantic Web Rule Language) rules to specify transaction-focused systems. Domain specific ontologies and semantic rules are used for defining the additional smart contract constraints and generating smart contracts in the Go programming language.

The model-driven approach based on DSL DasContract is presented in [35]. DSL is used not only to define smart

contract structure, but also to apply an extended subset of BPMN for smart contract behavior specification. The algorithm for generating smart contract code tailored for non-fungible tokens is presented using the specified DSL model as an input.

Catterpillar [34] is a BMPN execution engine that encompasses a set of tools for compiling, deployment, and monitoring of business processes on the blockchain. Catterpillar provides a BPM platform that emphasizes custom workflow execution on the Ethereum blockchain.

Another BPMN based solution is Lorikeet [30]. The Lorikeet tool encompasses business process management and provides an asset registry generator for creating smart contracts based on registry models. Both business processes and asset registries are integrated for the decentralization of fungible or non-fungible asset management.

Other approaches employ various UML diagrams for code generation. The usage of UML class diagrams [31] limits the generation of smart contracts by providing only boilerplate source code. Other existing proposals try to incorporate behavioral diagrams, such as UML state machine diagrams [20], for smart contract code generation. As state machines are similar, we present in more detail the proposed methods that incorporate UML state machines [20], transition systems [21], FSMs [40], and Petri Nets [41] in smart contract behavior specification for code generation purposes.

Transition systems (discrete systems consisting of states and transitions) are used to generate Solidity smart contracts in the VeriSolid tool [21]. VerySolid allows to define smart contract variable declarations, expressions, and event, return, if, loop, and compound statements. The VeriSolid generator can generate smart contract constructor, fallback functions, and introduce an additional state to denote the moment the smart contract is transitioning between the states.

The toolset for finite state machine synthesis called EFSM [40] is used to create finite state machines from LTL traces and a set of test scenarios. The generated FSMs are then augmented with state and action definitions and further used to generate Solidity code [19]. Although the proposed solution demonstrates the suitable approach for using formal specifications in smart contract development, the authors suggest that a more expressive formal system should be developed in the future.

In [42], the MDA framework is used as the basis for the structured development method of smart contracts. Based on MDA, the author establishes three models at different levels of abstraction. In the first model (CIM), ADICO (attribute, deontic, aim, condition, or else) statements are employed. On the basis of ADICO statements, a set of transitions for the FSM is created in the second model (PIM). In the third model (PSM), the FSM is extended by the contract variables and transitions; specific design patterns for access control, locking, transition counter, and timed transitions can be applied.

A graphical modeling tool is presented in [41] that allows modeling Petri Nets for smart contract generation purposes. Smart contract workflows can be modeled in Petri Net Markup Language, simulated, and formally verified before translation into smart contract code. Petri Net transitions are extended with guarded commands that represent business logic. The framework then maps transitions, places, and guarded commands to global variables and functions that implement the execution logic of transitions. The presented approach capabilities are demonstrated by generating Solidity smart contract code, but the authors state that the same principles can be applied to support other blockchain platforms.

In [20], a model-driven engineering approach is presented to generate Solidity smart contracts from UML state machine diagrams, placing greater emphasis on IoT and cyber-physical systems (CPS). CPSs are considered specific and complicated, as they are dynamically composed of smart devices and interconnected edge and cloud services, introducing additional dependability issues and cyber security risks. The authors choose to express the state of cyber-physical systems by employing UML state machine diagrams. UML elements are mapped to Solidity language constructs for code generation and verification purposes. Like FSM-based approaches, this approach enables the definition of states and transitions with guards and supports composite and history states. The provided proof of concept Solidity smart contract is generated for the Ethereum platform.

Our approach combines a clearly defined MDA framework and application of UML throughout the design process to ensure a smoother transition between different levels of abstraction via model transformation techniques. We utilize UML State machine diagram for defining smart contract behavior, ultimately resulting in generated smart contract code which includes not only the structure of smart contract, but also behavior for a chosen blockchain platform. Thus our approach is closely related to [20], [27], [29] in terms of application of model-driven development, usage of UML, and employment of state machines. Similarly to [42], MDA is used to define three models at different levels of abstraction that are ultimately used to generate code for different blockchain platforms. However, in our proposed solution for PIM and PSM, we opt out to use UML notation, as recommended by OMG [10], instead of relying on FSMs to model smart contract behavior. In our approach, the structure of the smart contract is specified using a UML class diagram, and the behavior can be specified using state machine diagrams. Similarly to [20], we also employ a UML state machine diagram to model the simple states, transitions, and guard constraints of the PIM smart contract.

Although the principles presented in our approach resemble the ideas from the related works, they differ from our approach in several key aspects. Although MDD is used extensively in [27] and [29], the models are not used to directly produce software artifacts applicable in the development of smart contracts. Meanwhile, proposals like [30], [34], [35] employ a modelling language (BPMN), but instead of generating code as in our approach, they rather develop a BPM engine, which executes processes and uses

the custom generated smart contract registries to store data on a blockchain. In [37], [39] the modeling principles are also used, and code is generated, but the new modeling notations are proposed and employed to produce smart contract code, instead of relying on generalized modelling language like UML, as our process does. Furthermore, a partially MDA-based proposal in [42] supports Solidity code generation, which does not employ model transformations from PIM to PSM, and thus the capability to extend the method to support another platform according to MDA principles is not available. In comparison to [42], our proposed method produces not only structural code elements (like variables, function headers), and state machine pattern specific elements (like *State* enumeration, the *timedTransition* modifier), but also contract events and contract constructor, modifiers for recurring guards, as well as *atState* modifier, thus supporting a more extensive code generation.

In the upcoming section, we will detail our proposal and explain the details of the proposed solution to generate Solidity smart contract code.

III. MDA BASED SOLIDITY SMART CONTRACT DEVELOPMENT

MDA is based on transformations between models, which enable moving from one abstraction level to another. This paper focuses on transformations from PIM to PSM for Ethereum Solidity and from Solidity PSM to smart contract Solidity code. The general process encompassing our proposed transformations is presented in Fig. 1. The process consists of three main steps: the development of Blockchain PIM, the transformation from PIM to Solidity PSM, and the transformation from Solidity PSM to Solidity smart contract code. For the implementation of transformations, metamodels for the PIM, Solidity PSM, and Solidity code are defined. Furthermore, algorithms are provided to perform both transformations. The final result of the transformations is the smart contract code, which can be compiled and executed on the Ethereum blockchain.

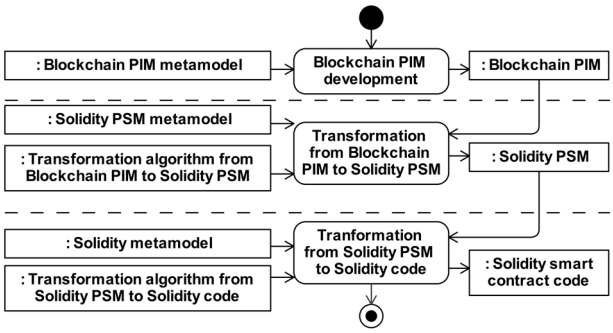

FIGURE 1. The MDA based process of Solidity smart contract code generation.

In MDA, PIM is used to capture essential details of the system under development without elaborating the specifics of the implementation technology platform of choice. In our approach, Blockchain PIM (hereafter referred to as PIM) is used to capture the smart contract details while remaining independent of the blockchain implementation platform. Our PIM is specifically intended for blockchain and smart contract development and captures common structural and behavioral features that can be further translated into different blockchain platform implementations. Following the MDA principles, our proposed PIM must be specified using UML. The PIM metamodel is based entirely on the UML metamodel [12] and only includes two smart contract specific elements: an additional Datatype (address) and an additional stereotype, which denotes the financial context (≪pay≫). The proposed PIM encompasses several types of UML diagrams: Class diagram for smart contract structure specification, State Machine diagram for smart contract behavior specification. In the PIM, the Class directly contained in the PIM package is treated as a smart contract. It can have properties, operations, and owned classifiers that define the smart contract structure or enumerations. In order to describe a smart contract behavior, the State Machine can be specified, which has to be owned by the smart contract class as its Behaviored Classifier representation. The developed PIM is further used as an input for the transformation to PSM. Since we focus on the Ethereum platform, in the next section, we present the Ethereum Solidity PSM metamodel and elaborate on the process of how the transformation to Solidity PSM is performed.

A. TRANSFORMATION FROM BLOCKCHAIN PLATFORM INDEPENDENT MODEL TO SOLIDITY PLATFORM SPECIFIC MODEL

As the proposed Blockchain PIM defines the input for the PIM to PSM transformation, the Solidity PSM is the output. Therefore, we have defined the Solidity PSM metamodel, which is based on the UML metamodel, and extended it with Solidity specific stereotypes (Fig. 2). The introduced stereotypes extend the application of different UML elements, like Class, Property, Operation, Parameter, etc., by defining Solidity specific concepts as stereotypes, e.g., contract, struct, enum, variable, function, etc. Furthermore, the Solidity PSM metamodel includes a set of additional Solidity Datatypes, which will be further mapped to Solidity code (uint, string, bool, address, etc.). Solidity PSM consists of UML Class and State Machine diagrams, the very same types of diagrams as used in Blockchain PIM. PSM Class diagrams represent the structural view of the model and, during transformation, are enriched not only with Solidity specific types but also with additional information extracted from PIM State Machines. Both PIM and PSM smart contracts are specified using State Machines for capturing relevant States, Transitions, Transition Guards, Call Events, and Pseudostates. PSM State Machines differ from PIM State Machines since, during the transformation from PIM to PSM, some of the State Machine elements are modified: any recurring guards Constraints of Transitions are extracted and transformed to function modifiers, any Time Events used in Triggers are substituted by Guard Constraints, and any Time Expressions *now* used in

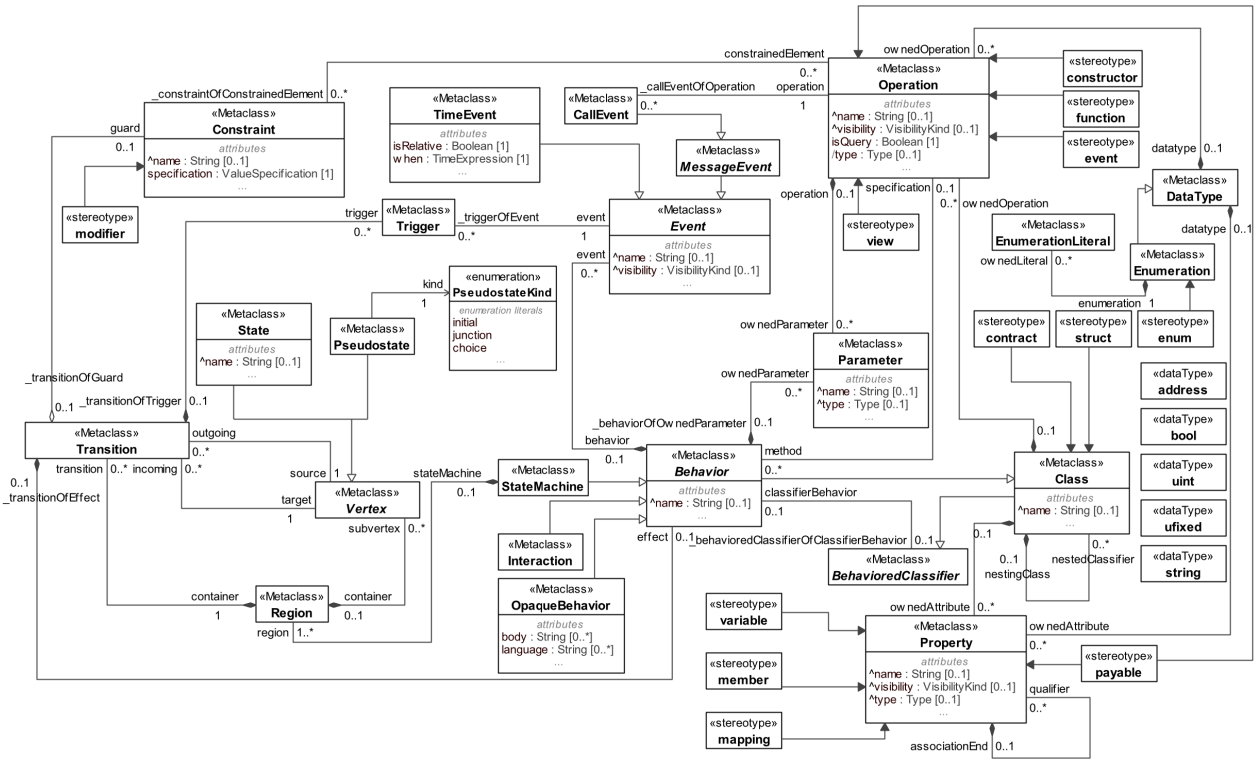

FIGURE 2. Solidity PSM metamodel (based on UML metamodel).

transition guards are substituted by Solidity global variable *block.timestamp*.

The algorithm for the transformation of PIM to PSM is presented in Fig. 3. In the first step of the algorithm, the main PIM Class (which is owned directly by the PIM package) is transformed into the Ethereum Solidity smart contract by creating a copy of the PIM Class in PSM and applying the ≪contract≫ stereotype. Similarly, nested PIM classifiers are transformed into PSM classifiers. Based on the nested classifier type, either the PIM Class is transformed into a PSM Class with a ≪struct≫ stereotype applied, and each Property of the nested Class is transformed into the PSM ≪struct≫ Class Property with a ≪member≫ stereotype, or the PIM Enumeration with its literals is transformed into the PSM Enumeration with the ≪enum≫ stereotype. Then, each PIM Property is transformed into a PSM Property with applied stereotype ≪variable≫ having an appropriate Solidity PSM datatype. Moreover, if any Property is found in PIM that describes a set of Elements, a corresponding PSM Property with stereotype ≪mapping≫ is created. The ≪mapping≫ PSM Property is created to include two qualifiers: a type and a key. Furthermore, the PIM Operations are transformed into PSM Operations with a specified ≪function≫ stereotype. Once the PIM Class diagram elements are transformed, the analysis of the specified PIM State Machine is initiated.

Provided the State Machine was specified in PIM, during transformation, a new Enumeration named *State* with a PSM stereotype ≪enum≫ is created, which lists all States (as Enumeration literals) that are outlined in the State Machine diagram (Fig. 3). Furthermore, PIM State Machine Transitions that have specified effects are collected, and using their OpaqueBehavior, a PSM contract Operation with either ≪event≫ or ≪constructor≫ stereotype is created. If the OpaqueBehavior has stereotypes, Attributes, or Parameters, these are also transformed and applied to PSM Operation. The selection of relevant stereotype depends on the transition source: the OpaqueBehavior of the Transition from initial Pseudostate is transformed into a ≪constructor≫ Operation; all other Transition OpaqueBehaviors are transformed into PSM contract ≪event≫ Operations.

Afterward, all PIM State Machine Transitions are collected and analyzed (Get the list of all transitions step in Fig. 3). During the Transition analysis, if determined that PIM Operation has been used only once per Transition, or the Operation usage is recurring, but the source of the Transition is the same for all recurring Transitions, a common Solidity state machine design pattern can be applied. Then a new PSM *atState* Constraint with a stereotype ≪modifier≫ is created, and the Dependency relationship between the PSM ≪function≫ Operation and the Transition source PSM State ≪enum≫ Enumeration literal is created.

In addition, if the collected PIM Transitions list contains at least one Transition with a specified Time Event Trigger, a PSM Constraint ≪modifier≫ named *timedTransitions* is created (Fig. 3), which checks whether a specific point in time was reached by comparing with the current timestamp. As a Time Event from UML State Machine cannot be directly transformed to any Solidity specific element because

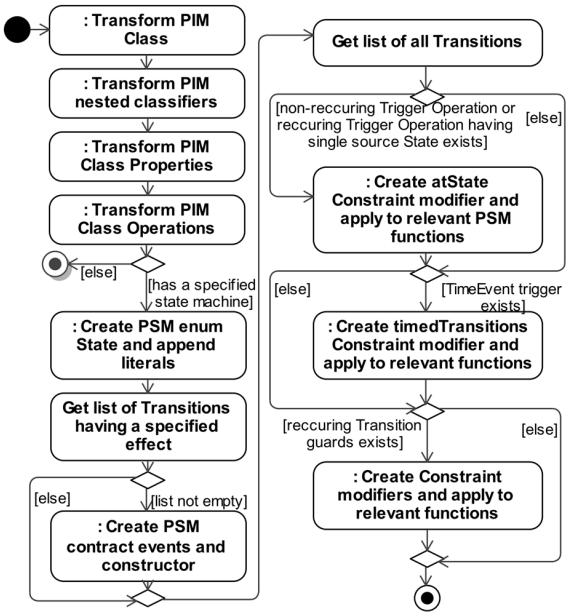

FIGURE 3. Blockchain PIM to Solidity PSM transformation algorithm.

Solidity does not support time-based transitions between states, any PIM Transition Time Events are transformed to PSM guard Constraints and the transitional logic is relocated to the PSM Constraint ≪modifier≫ *timedTransitions*. For this ≪modifier≫ implementation, an additional PSM Property ≪variable≫ named *creationTime* is required, which is used to calculate the point in time when the smart contract in a specific State is supposed to transition to a different State. Each PIM Time Event Transition is then transformed to a PSM guard Constraint, which compares the current time and the time (relative or absolute) when the transition should occur. The relative time (specified on the UML state machine transition as *after(relativeTime)*) is calculated based on the Time Event usage: the first Time Event usage can be described as *creationTime* + *relativeTime*, but for all other following Time Event Transitions, the time needs to be calculated by adding up all preceding Transitions Time Event *relativeTime* values. Once the ≪modifier≫ *timedTransitions* is created, it is retroactively applied to every smart contract PSM Operation with a ≪function≫ stereotype.

During the last step of the algorithm (Fig. 3), the list of all PIM Transitions is analyzed again and Transition guards Constraints that are used more than once are extracted and transformed into a PSM Constraint with stereotype ≪modifier≫. Instead of recurring guards from PIM Transitions, the Constraint ≪modifier≫ is applied to relevant ≪function≫ Operations in Solidity PSM.

After the transformation from PIM to PSM, the PSM ≪function≫ Operations can be extended by selecting OpaqueBehaviors from the curated library of code samples. Each OpaqueBehavior in the library is specified by a body that includes the software code and can also have Properties, Parameters, and additional Operations that need to be appended to the smart contract. Currently, the library includes code samples from the Solidity documentation [43]. Later, it is also planned to include the code samples from ERC standard implementations [5], and a set of Interactions specified for a specific ≪function≫ Operation and would be further transformed into Solidity code (the analogous possibility of generating the Hyperledger Go smart contract code from the sequence diagram is demonstrated in [24]). The proposed idea of a curated library and the possibility to extend PSMs with OpaqueBehaviors would allow developers to specify and reuse relevant smart contract implementation code samples.

The result of the transformation is Solidity PSM, which encompasses Class and State Machine diagrams. In contrast to PIM, the Solidity PSM additionally includes the elements that were extracted from the State Machine: ≪modifier≫ Constraints, ≪constructor≫ Operation, ≪event≫ Operations, the ≪enum≫ State and its literals.

B. TRANSFORMATION FROM SOLIDITY PLATFORM SPECIFIC MODEL TO SOLIDITY CODE

The proposed algorithm for generating the Solidity smart contract code uses the previously transformed Solidity PSM as an input, and by reading the model produces the smart contract Solidity file. For this transformation, the Ethereum Solidity metamodel is outlined in Fig. 4, which defines the main supported elements of the generated Solidity v0.8.0 source code (as our approach is intended for smart contract code generation only, the Solidity constructs related to library or interface development are not supported). The PSM model elements are mapped to relevant Solidity constructs during the transformation.

The smart contract boilerplate code is produced relatively straightforwardly from the PSM Elements specified in the Class diagram, such as Classes, Properties, Operations, and Constraints. Nevertheless, the full potential of our approach can be achieved when the Solidity PSM has a specified State Machine diagram, and Operations are extended with code samples by describing behavior either by specifying an OpaqueBehavior or by Sequence diagram as an Interaction. During the M2T transformation, each PSM Operation with the stereotype ≪function≫ is transformed into the Solidity Function and extended to include the logic specified in the PSM State Machine diagram. Additionally, if OpaqueBehavior was provided to the PSM ≪function≫ Operation, the OpaqueBehavior body is extended with the State Machine logic and appended to the Solidity Function code. Any sequence diagram specifying the PSM Operation as an Interaction can also be transformed into source code, but the idea of transformation of the sequence diagram into smart contract code is presented in [24] and is not elaborated further in this paper.

The main algorithm for transforming the Solidity PSM to the Solidity source code is presented in Fig. 5. The algorithm creates the Solidity file by reading the Solidity PSM and appends the code to this file line by line. During the Transform Contract Class step, the PSM ≪contract≫ Class and its ≪variable≫ Properties are transformed to the Solidity

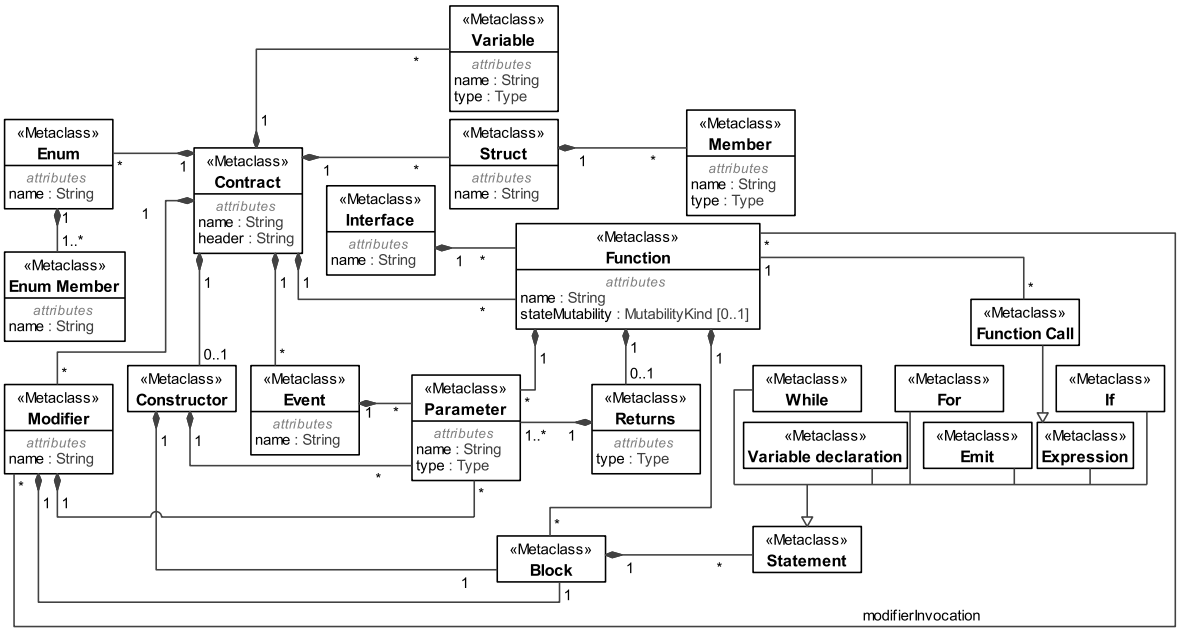

FIGURE 4. Ethereum Solidity smart contract metamodel: key elements used in transformation.

Contract and its Variable declarations. Similarly, during the Transform Struct Class step, the PSM ≪struct≫ Class is transformed to a Solidity Struct type and its Members. In the Transform Enumeration step, the PSM Enumeration and its literals are transformed to Solidity Enum and its members. Correspondingly, the PSM ≪modifier≫ Constraints and their specifications are transformed into Solidity Modifiers and their Blocks of Statements.

Once all PSM ≪contract≫, ≪variable≫, ≪struct≫, ≪member≫, ≪enum≫, ≪modifier≫ Elements are transformed, the algorithm checks for the Operation usage in the specified PSM State Machine diagram. If such usage is not detected, the Operation is considered as a nonextendable Operation. During the Transform nonextendable Operation step (Fig. 5), based on the PSM Operation stereotype, the Operation is transformed into Solidity Constructor, Event, or Function definition. Additionally, any specified Parameters are appended to code, and in case ≪modifier≫ Constraints were applied to ≪function≫ Operation, Solidity Modifier invocations are appended to Function code as well. Furthermore, if Operation Behavior was specified either by OpaqueBehavior or an Interaction, the OpaqueBehavior body or the code generated from the Interaction is appended to the Solidity Function code.

When all the PSM Operations not used in the State machine are transformed, the Transform PSM State Machine step (Fig. 5) starts. During this step, the ≪function≫ Operations specified in State Machine Transition Trigger Call Events (considered as extendable Operations) are transformed to Solidity Function definitions and extended with the information specified in the State Machine. This transformation is performed by selecting all Transitions Triggers and checking for Call Event Operation usage. Once found, all Transitions with the relevant Call Event Operation are selected for further analysis, and the Solidity Function name, Parameters, and Function Modifier invocations are appended to the code. The last step is Transform extendable Operation (Fig. 5), further detailed in Fig. 6.

Each extendable Operation transformation (Fig. 6) mainly consists of Transition sources and Transition targets transformations. In the first step, Transition sources are transformed, which results in Solidity Conditional Statements appended to Function code. Additionally, if Operation has a specified OpaqueBehavior, the part of its body before the return statement is appended to the code. Afterwards, the Transition targets are transformed, resulting in a Block of Statements (encompassing all necessary Conditional Statements, Variable *state* declarations, and emit Statements) that is appended to the Function code. And finally, if Operation has a specified OpaqueBehavior, its return statement is appended. The steps of Transforming transitions sources and Targets are elaborated in more detail in Fig. 7 and Fig. 8.

During the transformation of Transition sources (Fig. 7), the Solidity Function code is extended to include the Conditional Statements, which check whether the smart contract is in a specific State, and a specific Function can be executed. Since the Transitions with specific guard Constraints can be specified in two ways, one using the junction Pseudostate and the other by specifying multiple Transitions between the same source Vertices, the algorithm covers both alternatives by collecting and further analyzing all Transition sources from the selected list of Transitions for a specific Operation. Afterwards, by determining each Transition source type, a Conditional Statement is appended (except when the source is an initial Pseudostate, then this step is skipped). For the Transition Trigger Call Event Operation that is reused multiple times in the State Machine, or has multiple sources, either specified by using junction Pseudostate, or specified by

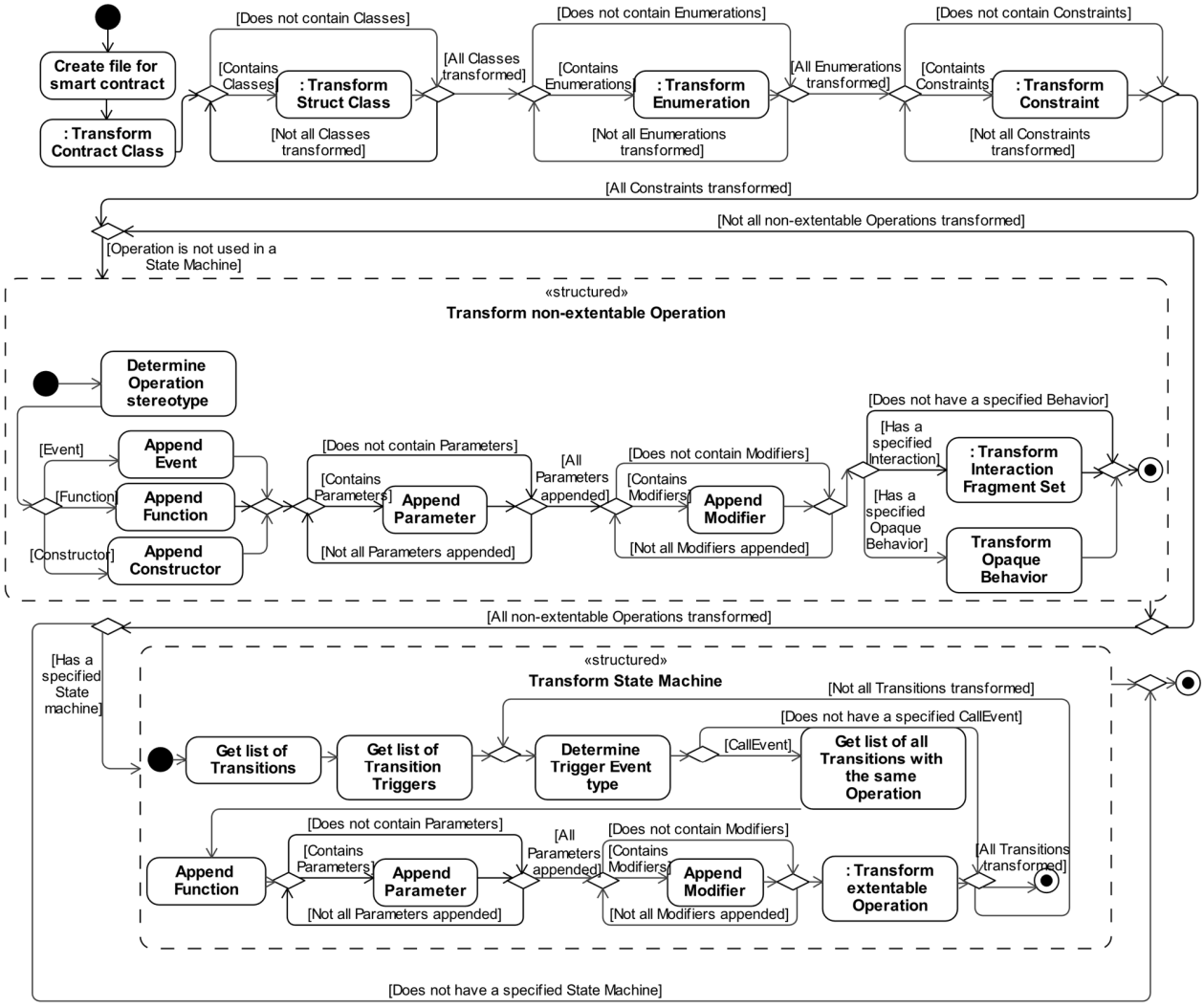

FIGURE 5. Solidity PSM to Solidity smart contract code transformation algorithm.

multiple Transitions, a Conditional Expression *require* is generated which lists the multiple State sources inside separated by an *or* operator for the relevant Function. Additionally, regardless of the Transition source, each Transition is checked for a specified Transition guard Constraint. If such Constraint exists, Conditional expression is also appended, this time using the *and* operator.

Similarly, the Transition targets are transformed (Fig. 8) into *state* Variable declarations inside a particular Function source code, but this time by analyzing all Transition targets from selected the list of Transitions for a specific PSM Operation. The Transition targets are collected once again since the Transitions can be specified in multiple ways, either by having multiple transitions using the same Call Event Operation or by using the junction or choice Pseudostates. During the transformation to code, the Transition set is checked if more than one Transition source Vertex exists, and if it does, additional *if* Conditional Statement is appended, which checks whether the smart contract is in a specific *State* and whether expression transformed from specified guard Constraint is true. Then the Transition list is checked for Transition targets, and by determining the Transition target type, if the type is a State, a Transition effect (if specified) is transformed to *emit* Statement, and a *state* Variable declaration (if the target does not match the source) is appended. Additionally, if the target of a Transition is of a choice or junction Pseudostate, additional outgoing Transitions and their targets are determined. Based on the collected set of Transitions, additional nested if/else Conditional Expressions based on the Transition guard Constraints are appended. Once each Conditional Expression is appended, the *emit* Statement is appended. Finally, if the source and target of the Transition are different, the Variable *state* declaration expression is appended.

C. IMPLEMENTATION OF THE TRANSFORMATIONS

For the necessary transformation implementation, a combination of Eclipse Modeling Tools was used. Eclipse ATL was used to develop M2M transformations, and Eclipse Acceleo

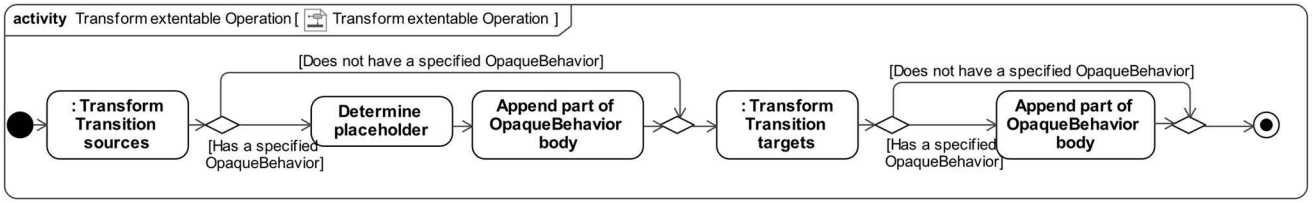

FIGURE 6. Extendable operation to the solidity function code transformation algorithm.

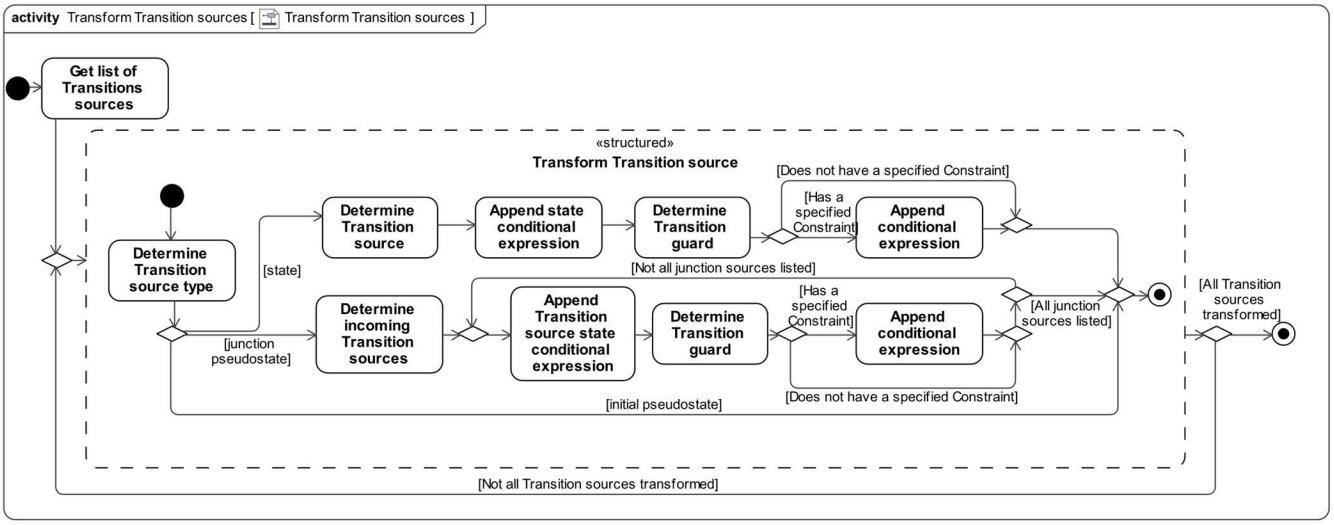

FIGURE 7. Transform transition sources step in extendable operation transformation algorithm.

tool was used for the implementation of M2T transformations. Additionally, the Magicdraw 19.0 CASE tool was used to specify the required Blockchain PIM and later review and extend the generated Solidity PSM, which supports the UML 2.5 version.

The specified Blockchain PIM has to be exported as Eclipse compliant (UML 2 (v5.0)) XMI format file and used as an input for Eclipse ATL during M2M transformations. The Blockchain PIM to Solidity PSM transformation algorithm was developed using the Eclipse ATL tool [25] in the ATL transformation language. The implemented transformation uses the Blockchain PIM metamodel and the Solidity PSM metamodel to map relevant model elements. The result of PIM to PSM transformation is a Solidity PSM XMI file, which encompasses UML Class, State Machine diagrams, and UML Elements with relevant PSM stereotypes. The generated XMI format file can be imported to the Magicdraw CASE tool, using which the generated PSM can be reviewed and extended by the developer if necessary. Afterwards, like Blockchain PIM previously, Solidity PSM needs to be exported as Eclipse supported XMI format file and is used as an input for the Eclipse Acceleo tool during the M2T transformation. The proposed Solidity PSM to code transformation algorithms were implemented in the Eclipse Acceleo tool [26] using the MOFM2T templating language. Acceleo uses the Solidity PSM metamodel to map relevant model elements and executes the transformations during which the specific MOFM2T templates are applied for Solidity code production. The Eclipse Acceleo uses the Solidity PSM XMI format file as an input and, as a result, produces the.sol file, which contains the generated smart contract code in the Solidity programming language. The generated smart contract code needs to be compiled using a Solidity compiler and afterward can be deployed on a network.

IV. EVALUATION OF IMPLEMENTED TRANSFORMATIONS FOR SMART CONTRACT DEVELOPMENT

This section demonstrates how the proposed approach can be employed to model smart contract behavior and generate Solidity source code. For this demonstration, three different smart contract examples from the Solidity documentation were selected. The selected smart contracts were modeled according to the principles of our approach, and then the Solidity code was generated using the algorithms implemented in the Eclipse ATL and Acceleo tools. The generated Solidity source code was evaluated and compared with the original counterparts from Solidity documentation in terms of Solidity code metrics, similarity scores, and execution costs.

A. THE PROCESS OF EVALUATION

To demonstrate the capabilities of our approach, three different smart contract examples *(SimpleAuction* [44], *Purchase* [45], and *StateMachine*[46]) from the Solidity documentation were selected, which have state machine-like constructs. The smart contracts were specified as Blockchain PIMs, depicting the structure and behavior of the

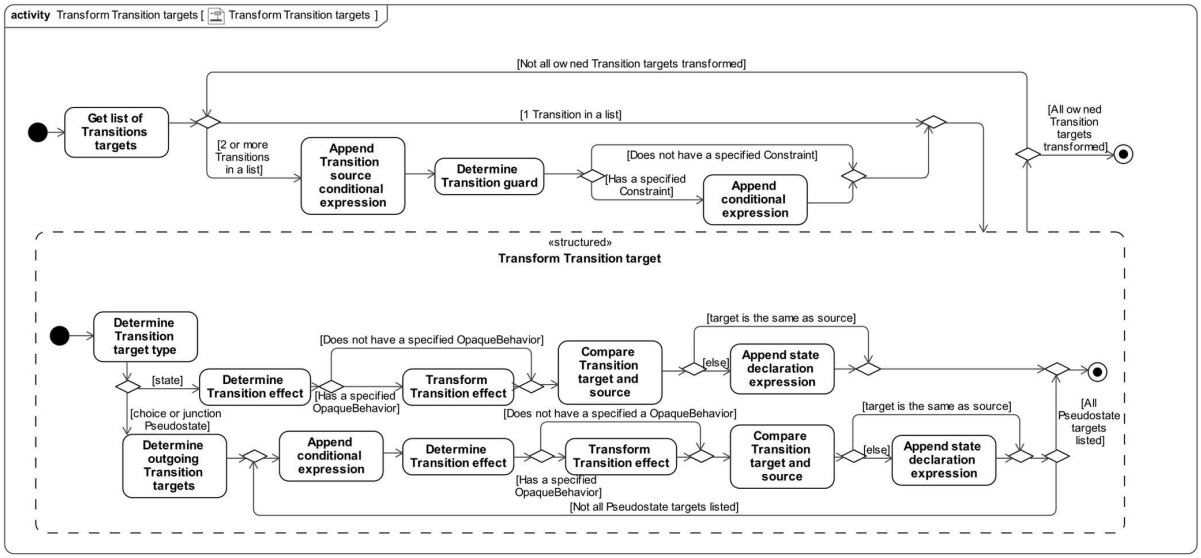

FIGURE 8. Transform transition targets step in extendable operation transformation algorithm.

SimpleAuction, Purchase, and *StateMachine* smart contracts. In the next step, Solidity PSMs were generated for each smart contract. The generated *SimpleAuction, Purchase*, and *StateMachine* PSMs were later imported into the Magicdraw CASE tool, extended by OpaqueBehavior specifications, and once again exported as XMI format files. The XMI files were used in the Eclipse Acceleo tool to generate Solidity *SimpleAuction, Purchase,* and *StateMachine* smart contract code files. The generated smart contract files are available at.github.com/m-jurgelaitis/generated-smart-contracts.

For each generated smart contract code file, Solidity code metrics were calculated using the Visual Studio Code tool and the VSCode Solidity Metrics plugin [47]. Normalized source lines of code (nSLOC) count and complexity score metrics (custom score inferred from complexity introducing code statements, like branches, calls, etc.) were used to compare the generated and the original smart contracts. Additionally, for each pair of smart contracts (original and generated), similarity scores were calculated using the SmartEmbed tool [48], which estimates similarity based on the semantic distance between two fragments of code. SmartEmbed parsed each Solidity file into an Abstract Syntax Tree (AST) and calculated numerical vectors, which were later used to estimate similarity scores. In addition to the SmartEmbed similarity score, the cosine similarity was calculated for each smart contract pair, using the Solidity smart contract numerical vectors.

Lastly, both original and generated smart contracts were compiled using the Remix IDE and deployed onto the JavaScript Virtual Machine running on a Firefox browser to calculate and compare smart contract execution costs.

B. SIMPLEAUCTION SMART CONTRACT CODE GENERATION

The first smart contract example used to demonstrate transformations is a simple auction example provided in the

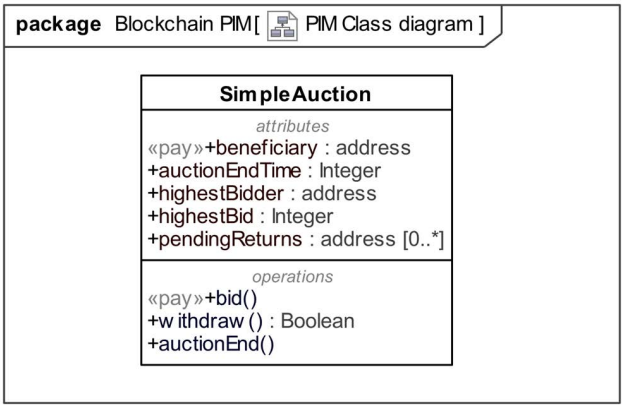

FIGURE 9. Blockchain PIM *SimpleAuction* smart contract: Class diagram.

Solidity documentation [44]. The *SimpleAuction* smart contract overviews the basic principles of how state machines can model smart contract behavior. Even though the original *SimpleAuction* smart contract does not have any specified states, it has a bool variable *ended* for tracking the status of the auction, which in our approach was specified using the PIM State Machine diagram.

The *SimpleAuction* smart contract structure was specified using a Class diagram (Fig. 9), and the behavior was modeled using the State Machine diagram (Fig. 10). The *SimpleAuction* Class encompasses three Operations and five Attributes. The original attribute *ended* was omitted and, instead, a state machine was specified, encompassing *Open* and *Ended* states. Additionally, the PIM state machine for *SimpleAuction* encompasses two Operations as Call Event Transitions, two Transition guard Constraints, and three effect OpaqueBehaviors.

The diagrams resulting of the transformation from Blockchain PIM to Solidity PSM are presented in Fig. 11 and Fig. 12. One of the important differences between the PIM Class diagram (Fig. 9) and PSM Class diagram (Fig. 11)

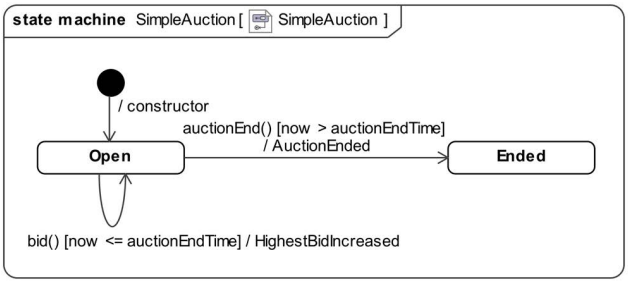

FIGURE 10. Blockchain PIM *SimpleAuction* smart contract: State machine diagram.

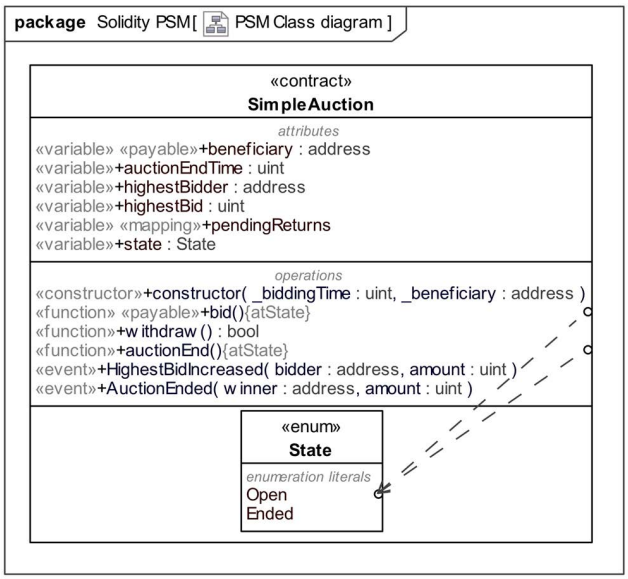

FIGURE 11. Solidity PSM *SimpleAuction* smart contract: Class diagram.

is that the PIM Property *pendingReturns* (a set of *address*) (Fig. 9) was transformed to PSM mapping variable (Fig. 11), having two properties: a key (*uint*) and a type (*address*). Although these two Properties are not visible in the diagram, they are recorded as qualifier Properties of the variable. Another difference is that the *bid()* function and the *beneficiary* variable in PSM have the ≪payable≫ stereotype applied. Moreover, based on the PIM State Machine (Fig. 10), the PSM smart contract Class (Fig. 11) has several newly created Elements, like *state* variable and *State* enum, which lists the defined *Open* and *Ended* states. Likewise, a constructor and two events (*HighesBidIncreased* and *AuctionEnded*) were created in PSM (Fig. 11), based on the specified effects in the PIM State Machine. Lastly, the *atState* modifier was created and applied to the *auctionEnd(),* and *bid()* functions according to their usage in PIM State Machine Transition Call Events (Fig. 10). The main difference between the PIM state machine (Fig. 10) and the PSM state machine (Fig. 12) is that the time expression *now* was transformed to global variable *block.timestamp*.

The generated Solidity source code for *SimpleAuction* is compared with the original smart contract code using calculated code metrics (Table 1), encompassing the AST node count, source units, and similarity scores. Based on the

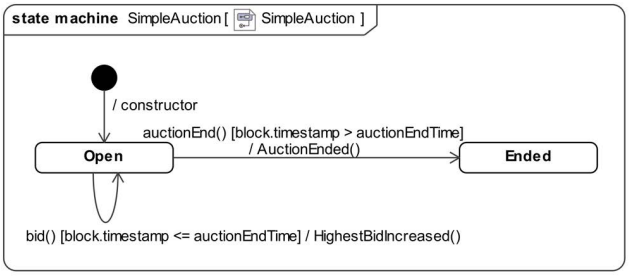

FIGURE 12. Solidity PSM *SimpleAuction* smart contract: State machine diagram.

results, the generated and original Solidity code differ mainly in terms of *State* enum usage (AST elements Enum Definition and Enum Value) and generated *atState* modifier (AST elements Modifier Definition, Block and Modifier Invocation, Expression Statement) that were introduced during transformations. Additionally, compared to the original one, more Variable Declaration (a variable declaration is included in a constructor). In terms of Source Units metrics, the contracts are almost identical, the nSLOC differs by one line of code, and the Complexity Score differs by two; this is also because of the inclusion and usage of enum and modifier.

For the more specific comparison between the original and generated smart contracts, the similarity scores were calculated, demonstrating that generated smart contracts are similar enough to the original, reaching approximately 90% similarity calculated using the SmartEmbed tool, and approx. 99% cosine similarity.

Additionally, the generated code was deployed on a JavaScript VM for the calculation of smart contract execution costs (Table 2). In the calculation of the *constructor+deployment* cost, constructor gas execution costs (if the constructor was defined) and smart contract deployment cost are included. The execution cost of the generated smart contract is reduced by 5% compared to the original in terms of *constructor+deployment* cost and is slightly increased for smart contract functions. The most notable difference of 3% is in the case of the *bid()* function, which now has additional modifier invocation compared to the original smart contract code.

C. PURCHASE SMART CONTRACT CODE GENERATION

The second example is the *Purchase* smart contract [45], which provides an implementation pattern that employs states and functions to transition between these states. This example demonstrates how the Solidity modifiers can be produced based on the specified recurring guard Constraints.

The *Purchase* Blockchain PIM, based on the Solidity documentation implementation code example, was manually specified to include three Properties and four Operations (Fig. 13). The Properties *seller* and *buyer,* and the Operation *confirmPurchase,* have the ≪pay≫ stereotype applied. Additionally, PIM State Machine representing the *Purchase* smart contract behavior was specified (Fig. 14), which encompasses *Created*, *Locked*, *Release,* and *Inactive* States, and four

TABLE 1. Comparison of *SimpleAuction* smart contract code metrics.

Source Units	Generated	Original
nSLOC	51	52
Complexity Score	30	28
AST Element	Generated	Original
Block	8	7
EmitStatement	2	2
EnumDefinition	1	0
EnumValue	2	0
EventDefinition	2	2
ExpressionStatement	15	13
FunctionCall	9	9
FunctionDefinition	4	4
Mapping	1	1
ModifierDefinition	1	0
ModifierInvocation	2	0
StateVariableDeclaration	6	6
VariableDeclaration	15	14
Similarity		
SmartEmbed Similarity Score	0.897	
Cosine Similarity	0.989	

TABLE 2. Comparison of *SimpleAuction* smart contracts code execution costs (gas).

Smart Contract Function	Generated	Original
constructor+deployment	606556	641548
bid()	71547	69331
withdraw()	23703	23636
auctionEnd()	60731	60651

Transitions between them. Each Transition has a Trigger Call Event Operation, a guard Constraint, and a specified effect OpaqueBehavior.

The diagrams that depict the outcomes of the Purchase PIM to PSM transformation are presented in Fig. 15 and Fig. 16. Like in the previous example, a new *state* variable was created in the PSM Class diagram (Fig. 15), accompanying the *State* enum, which lists four states (*Created, Locked, Inactive, Released*) specified in PIM State Machine (Fig. 14). Another important difference between PIM and PSM Class diagrams is that in PSM, the smart contract constructor, the Operations *Aborted*, *PurchaseConfirmed*, *ItemReceived*, *SellerRefunded*, and *atState* modifier are created, and the modifier is applied to every function of the PSM smart contract Class (Fig. 15). Furthermore, the *onlySeller* modifier for a guard Constraint which was used twice in the PIM State machine (Fig. 14), is applied to the PSM Class Operations *abort()* and *refundSeller()* (Fig. 15), which in PIM were specified on the relevant Transitions as Call Events. Since this modifier was automatically generated, it was first given a default name *mod1*, but later manually renamed to *onlySeller* for the sake of comprehension and readability.

While comparing PIM and PSM State Machines (Fig. 14 and Fig. 16), it can be noted that the recurring guard Constraints from PIM State Machine are omitted in the PSM State Machine (Fig. 16), as they resulted in *onlySeller* modifier, applied to relevant functions of the PSM smart contract Class (Fig. 15). Additionally, the generic expressions *transactionValue* and *transactionSender,* used in the guard Constraint specifications in PIM, were transformed

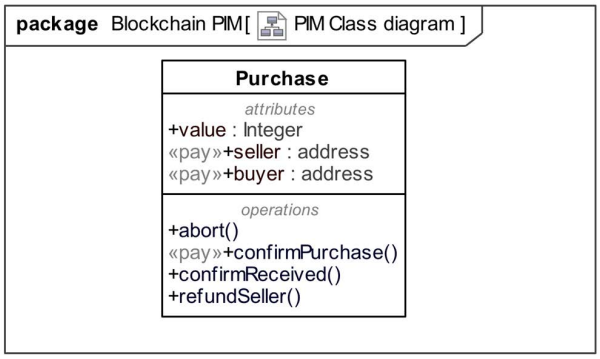

FIGURE 13. Blockchain PIM *Purchase* smart contract: Class diagram.

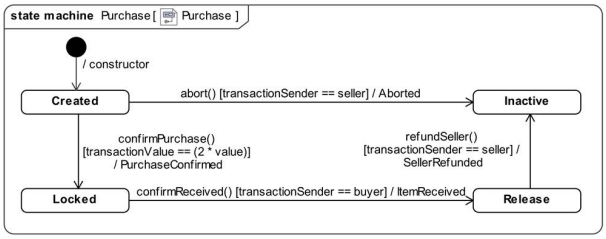

FIGURE 14. Blockchain PIM *Purchase* smart contract: State machine diagram.

to *msg.value* and *msg.sender* global Solidity variables in the PSM state machine diagram (Fig. 16).

The analysis of the generated and original Purchase smart contract code has revealed that the original has fewer lines of code (nSLOC) and a lower complexity score calculated by the VSCode Solidity Metrics plugin (Table 3). The AST Element (Block, Modifier Definition, Modifier Invocation, Expression Statement) count for the generated smart contract differs from the original because of the modifier usage: the modifiers *onlyBuyer* and *condition*, used in the original smart contract, are not present in the generated code. These modifiers were not generated since the algorithm only creates the modifiers for recurring guard Constraints, and the *onlyBuyer* and *condition* modifiers are used only once in the original smart contract code. It should also be mentioned that the *atState* modifier was generated instead of the *inState* modifier, which was present in the original smart contract; still, both modifiers serve the same purpose and differ in name and error message only. The AST metric VariableDeclaration varies between the original and generated versions because a *state* variable declaration is included in the generated constructor code. Nevertheless, both smart contracts are similar, reaching approx. 94% similarity score calculated by SmartEmbed, and approx. 99% cosine similarity.

The most notable differences are observed in the smart contract execution costs (Table 4): the *constructor+deployment* gas costs are reduced by approximately 18%, and the *abort()* and *confirmPurchase()* function execution costs are reduced by 46% and 36% accordingly. These differences can be attributed to the use of smart contract modifiers because our approach does not create a modifier unless the specific condition can be applied at least twice, thus leading to a generated

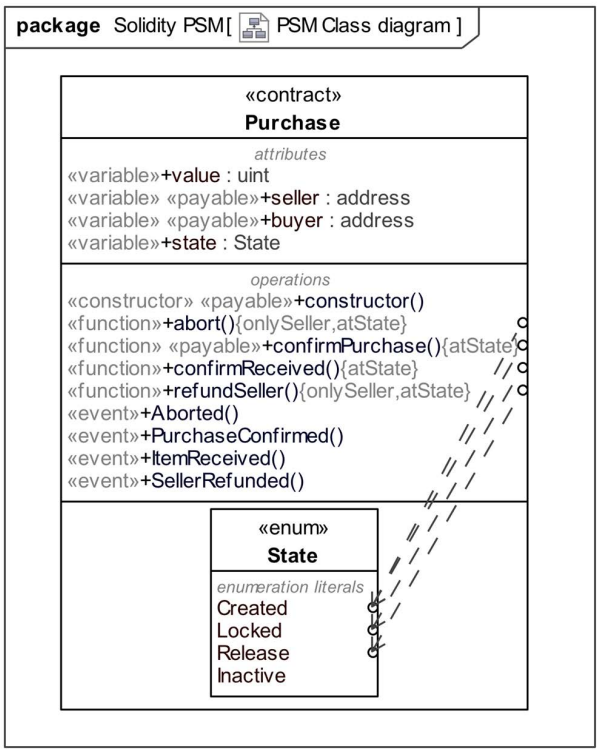

FIGURE 15. Solidity PSM *Purchase* smart contract: Class diagram.

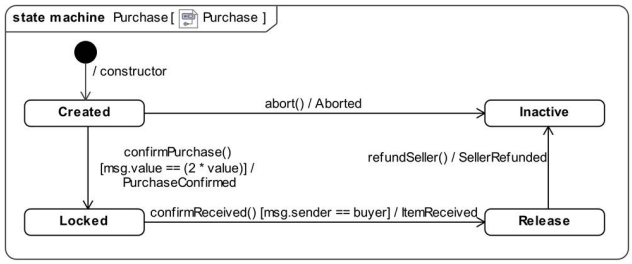

FIGURE 16. Solidity PSM *Purchase* smart contract: State machine diagram.

smart contract having lower complexity. The execution of the generated *abort()* function differs from the original in terms of the sequence of modifier invocation, and the generated *confirmPurchase()* function does not employ the *condition* modifier; instead, the conditional statement is included in the function body.

D. STATEMACHINE SMART CONTRACT CODE GENERATION

The last example is the *StateMachine* smart contract provided along with common patterns in Solidity documentation [46], which illustrates the usage of common state machine pattern. The original example not only includes the *atState* modifier, but also introduces a way to circumvent the unsupported automatic timed function execution by employing the *timedTransitions* modifier.

Like in previous examples, the smart contract was specified as Blockchain PIM, encompassing Class diagram (Fig. 17) and State machine diagram (Fig. 18), where the naming

TABLE 3. Comparison of *Purchase* smart contract code metrics.

Source Units	Generated	Original
nSLOC	48	62
Complexity Score	40	42
AST Element	Generated	Original
Block	7	9
EmitStatement	4	4
EnumDefinition	1	1
EnumValue	4	4
EventDefinition	4	4
ExpressionStatement	18	19
FunctionCall	15	15
FunctionDefinition	5	5
ModifierDefinition	2	4
ModifierInvocation	6	8
StateVariableDeclaration	4	4
VariableDeclaration	5	6
Similarity		
SmartEmbed Similarity Score	0.944	
Cosine Similarity	0.996	

TABLE 4. Comparison of *Purchase* smart contracts code execution costs (gas).

Smart Contract Function	Generated	Original
constructor+deployment	598342	727904
abort()	25123	47023
confirmPurchase()	30028	47133
confirmReceived()	29998	29998
refundSeller()	32252	32252

of diagram elements was based on the naming provided in the Solidity documentation. In addition, some changes were made to the original smart contract to simplify the testing process: the Time expressions in the Time Event specifications were changed to *2 minutes* and *3 minutes* accordingly, instead of *10 days* and *2 days*, as provided in the original *StateMachine* smart contract code. So, the resulting PIM encompasses five functions (*bid()*, *reveal()*, *g*(), *h*(), *i*()) that are used in the PIM State Machine as Call Event Operations, and the previously discussed TimeEvents.

The results of the *StateMachine* PIM to PSM transformation are presented in the Solidity PSM Class diagram (Fig. 19) and the State Machine diagram (Fig. 20). The PSM Class diagram (Fig. 19) differs from the PIM Class diagram (Fig. 17), as it includes newly created *state* variable and *State* enum listing all five states (*AcceptingBlindedBids*, *RevealBids*, *AnotherStage*, *AreWeDoneYet*, *Finished*). Furthermore, a generated *atState* modifier is applied to PSM functions (Fig. 19), based on PIM State Machine Transitions. Likewise, according to the Time Events usage, a *timedTransitions* modifier is applied to all functions, and an additional smart contract Property *creationTime* is included in PSM Class.

The main differences between the PIM State Machine (Fig. 18) and the PSM State Machine (Fig. 20) are related to the usage of UML Time Event: PIM Time Events are represented as guard Constraints in the PSM State Machine, specifically the Time Event *after(2 mins)* (Fig. 18) was transformed into a guard Constraint *[block.timestamp >= creationTime + 2 minutes]* (Fig. 20), and the Time Event *after(3 minutes)* was transformed into another guard Constraint *[block.timestamp >= creationTime + 5 minutes]*

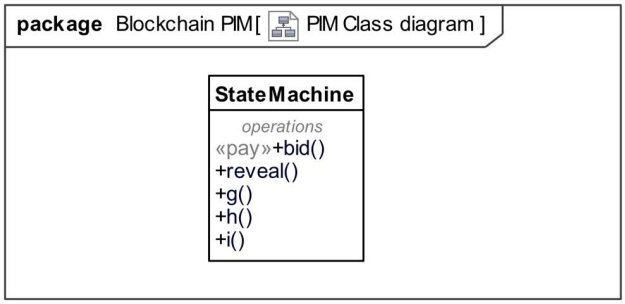

FIGURE 17. Blockchain PIM *StateMachine* smart contract: Class diagram.

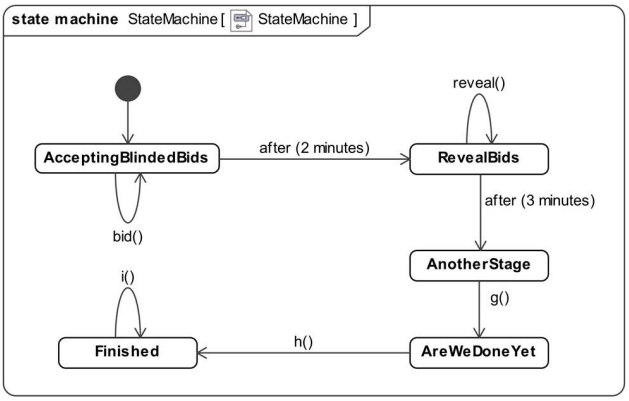

FIGURE 18. Blockchain PIM *StateMachine* smart contract: State machine diagram.

which added up three minutes to the previous Time Expression. The transition logic based on the transformed Time Events and the Time Expressions, is recorded in the *timedTransitions* modifier applied to functions of the PSM smart contract Class (Fig. 19).

The results of a comparison of the generated *StateMachine* smart contract code and the original (Table 5) demonstrate that these smart contracts differ in terms of nSLOC by 17 lines and Complexity Score by 7 points. The difference stems from the reduction of one function and one modifier usage: the original *StateMachine* code includes the *transitionNext* modifier and the *nextState()* function, which are not present in the generated smart contract code. Our approach does not support this implementation variation because the *transitionNext* modifier and *nextState*() function usage are applicable in a relatively small number of cases. The same result can be achieved by using *state* variable declarations in the appropriate function. This is also reflected in the decrease of the AST Element (Block, Expression Statement, Function Definition, Modifier Definition, Modifier Invocation) count in the generated smart contract. The most notable difference is the Function Call count since the *nextState()* function in the original smart contract code is used multiple times. Still, the calculated SmartEmbed Similarity score and Cosine Similarity show that the generated and the original smart contracts are similar, reaching a similarity of 82% and 95%, respectively.

Although the original and generated smart contracts differ in several aspects according to the code metrics, this

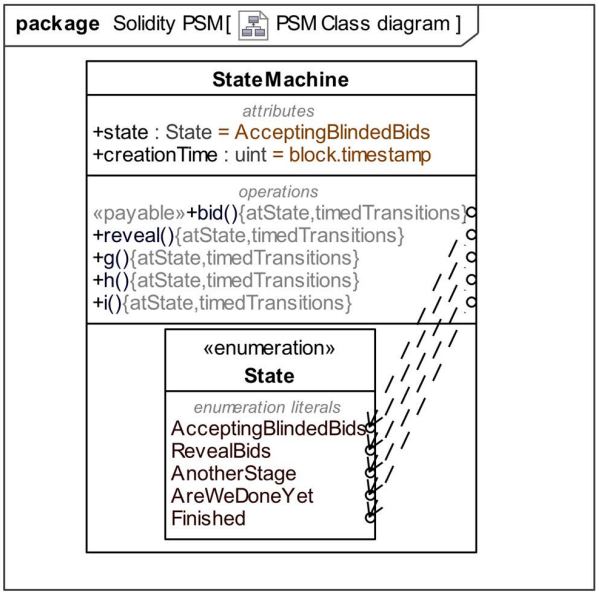

FIGURE 19. Solidity PSM *StateMachine* smart contract: Class diagram.

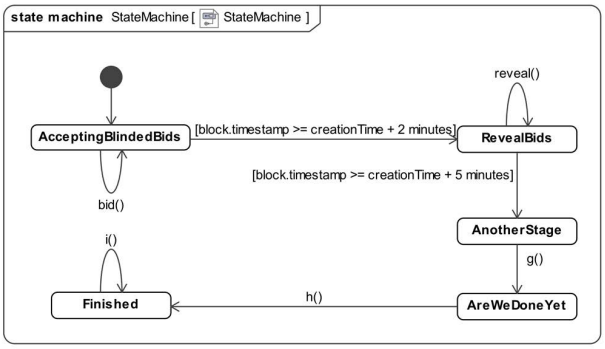

FIGURE 20. Solidity PSM *StateMachine* smart contract: State machine diagram.

TABLE 5. Comparison of *StateMachine* smart contract code metrics.

Source Units	Generated	Original
nSLOC	29	46
Complexity Score	29	36
AST Element	Generated	Original
Block	7	9
EnumDefinition	1	1
EnumValue	5	5
ExpressionStatement	7	8
FunctionCall	1	6
FunctionDefinition	5	6
ModifierDefinition	2	3
ModifierInvocation	10	12
StateVariableDeclaration	2	2
VariableDeclaration	3	3
Similarity		
SmartEmbed Similarity Score	0.827	
Cosine Similarity	0.951	

difference is not reflected in the execution costs of the smart contract (Table 6). The generated code execution costs are lower in terms of *constructor+deployment* (this smart contract does not have a constructor, so only deployment costs are included) and *g*() and *h*() function execution. Still, the reduction is marginal, at most 4% for the deployment cost and less than 3% for the *g*() and *h*() functions.

TABLE 6. Comparison of *StateMachine* smart contracts code execution costs.

Smart Contract Function	Generated	Original
constructor+deployment	596266	622427
bid()	26167	26167
reveal()	26213	26213
g()	29608	30414
h()	27015	27418
i()	23949	23949

V. CONCLUSION

The development of smart contracts is a complex task, especially since the smart contract code deployed to the blockchain cannot be further modified. Therefore, smart contract development strategies based on the reuse of existing smart contract code are quite prevalent in current development practices, although they require much effort from the developer to understand and modify the reused code samples. On the other hand, modeling approaches are commonly employed for their key advantages, such as visualization and alleviation of comprehension of the modeled solutions. Moreover, Model Driven Architecture provides guidelines for shifting development focus from the source code onto the higher level of abstraction, i.e., models.

Our proposed approach is based on MDA principles and uses UML models not only to increase the comprehension and reusability of smart contracts but also to produce smart contract implementation code. We demonstrate how the principles of MDA, the UML Class and State machine diagrams, and our proposed transformation algorithms can be successfully applied for the development of Solidity smart contracts for the Ethereum blockchain.

In order to demonstrate the main capabilities of our approach, we have decided to use readily available smart contracts code from Solidity documentation examples for developing the Blockchain PIM, transforming it to Solidity PSM, and then generating Solidity smart contract code. The original smart contract code (from the Solidity documentation) and the generated smart contract code were compared in terms of similarity. Since it is difficult to unambiguously evaluate the similarity of two smart contracts, we used several metrics and techniques: Visual Studio Code Solidity code metrics, similarity scores calculated using the SmartEmbed tool, and gas costs for executing the deployed smart contract. The comparison results indicate that our approach can be successfully used for modeling smart contract structure and behavior using UML Class and State Machine diagrams and generating smart contract code.

However, a threat to the validity of the evaluation of our approach should be noted: different developers could produce a variety of models for the same solution, and the results of code generation depend on the contents of Blockchain PIM. However, if Blockchain PIM is developed by strictly adhering to our approach and using the exact types of UML diagrams, the specified models and the generated code should be similar in essence. To minimize the impact of the modeling tendencies or preferences on the evaluation of the generated code, we chose to evaluate our approach by using three different smart contract examples from the official Solidity documentation. The results of the comparison of code metrics for original and generated smart contracts tend to be quite similar, reaching more than 90% of the cosine similarity for all pairs of smart contracts.

Current plans for future work have two directions. The first direction is based on an important advantage of adhering to MDA principles, as it allows expanding the approach to other technologies at the PSM level, e.g., in addition to Ethereum Solidity PSM, the Hyperledger Go PSM could be developed for enabling the developer to choose the preferred platform after the main design decisions were made at the PIM level. Another important direction for future work is to develop an extensible curated library of specified Solidity code samples as OpaqueBehaviors, which could be reused in various smart contracts as an extension to the developed models.

REFERENCES

[1] G. A. Pierro, R. Tonelli, and M. Marchesi, "An organized repository of ethereum smart contracts' source codes and metrics," *Future Internet*, vol. 12, no. 11, p. 197, 2020, doi: 10.3390/fi12110197.
[2] F. Casino, T. K. Dasaklis, and C. Patsakis, "A systematic literature review of blockchain-based applications: Current status, classification and open issues," *Telematics Informat.*, vol. 36, pp. 55–81, Mar. 2019, doi: 10.1016/j.tele.2018.11.006.
[3] D. Berdik, S. Otoum, N. Schmidt, D. Porter, and Y. Jararweh, "A survey on blockchain for information systems management and security," *Inf. Process. Manage.*, vol. 58, no. 1, Jan. 2021, Art. no. 102397, doi: 10.1016/j.ipm.2020.102397.
[4] M. Kondo, G. A. Oliva, Z. M. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: A case study on verified contracts from the ethereum blockchain platform," *Empirical Softw. Eng.*, vol. 25, no. 6, pp. 4617–4675, Nov. 2020, doi: 10.1007/s10664-020-09852-5.
[5] Ethereum. *Ethereum Improvement Proposals*. Accessed: Mar. 20, 2022. [Online]. Available: https://eips.ethereum.org/erc
[6] OpenZeppelin. *OpenZeppelin contracts*. Accessed: Mar. 20, 2022. [Online]. Available: https://openzeppelin.com/contracts/
[7] E. Domènguez, B. Pérez, L. Rubio, and M. A. Zapata, "A systematic review of code generation proposals from state machine specifications," *Inf. Softw. Technol.*, vol. 54, no. 10, pp. 1045–1066, Oct. 2012, doi: 10.1016/j.infsof.2012.04.008.
[8] A. Hsain and N. L. S. Mbarki, "Ethereum's smart contracts construction and development using model driven engineering technologies: A review," *Proc. Comput. Sci.*, vol. 184, pp. 785–790, Oct. 2021, doi: 10.1016/j.procs.2021.03.097.
[9] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Comput. Lang., Syst. Struct.*, vol. 52, pp. 43–62, Jun. 2018, doi: 10.1016/j.cl.2017.11.003.
[10] Object Management Group. (Jun. 18, 2014). *Model Driven Architecture (MDA) MDA Guide Review 2.0*. Accessed: Mar. 20, 2022. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01
[11] G. Sebastián, J. A. Gallud, and R. Tesoriero, "Code generation using model driven architecture: A systematic mapping study," *J. Comput. Lang.*, vol. 56, Feb. 2020, Art. no. 100935, doi: 10.1016/j.cola.2019.100935.
[12] Object Management Group. (Dec. 2017). *UML 2.5.1 Specification*. Accessed: Mar. 20, 2022. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/About-UML/
[13] N. Addison. *Unified Modeling Language (UML) Class Diagram Generator for Solidity Contracts*. Accessed: Mar. 20, 2022. [Online]. Available: https://github.com/naddison36/sol2uml
[14] M. Wáhrer and U. Zdun, "Design patterns for smart contracts in the Ethereum ecosystem," in *Proc. Things/GreenCom/CPSCom/SmartData/Blockchain/CIT*, Halifax, BC, Canada, 2018, pp. 1–9, doi: 10.1109/ICBC48266.2020.9169399.
[15] F. Volland. *State Machine|Solidity Patterns*. Accessed: Mar. 20, 2022. [Online]. Available: https://fravoll.github.io/solidity-patterns/state_machine.html

[16] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervas. Mobile Comput.*, vol. 67, Sep. 2020, Art. no. 101227, doi: 10.1016/j.pmcj.2020.101227.

[17] S. Alqahtani, X. He, R. Gamble, and M. Papa, "Formal verification of functional requirements for smart contract compositions in supply chain management systems," in *Proc. HICSS*, 2020, pp. 1–5, doi: 10.24251/HICSS.2020.650.

[18] N. Sanchez-Gomez, J. Torres-Valderrama, and M. J. Escalona, "Model-based software design and testing in blockchain smart contracts: A systematic literature review," *IEEE Access*, vol. 8, pp. 164556–164569, 2020, doi: 10.1109/ACCESS.2020.3021502.

[19] D. Suvorov and V. Ulyantsev, "Smart contract design meets state machine synthesis: Case studies," 2019, *arXiv:1906.02906*.

[20] P. Garamvölgyi, I. Kocsis, B. Gehl, and A. Klenik, "Towards model-driven engineering of smart contracts for cyber-physical systems," in *Proc. DSNW*, Luxembourg City, Luxembourg, 2018, pp. 134–139, doi: 10.1109/DSN-W.2018.00052.

[21] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," *Financial Cryptogr. Data Secur.*, vol. 11598, pp. 446–465, Oct. 2019, doi: 10.1007/978-3-030-32101-7_27.

[22] E. V. Sunitha and P. Samuel, 2019, "Automatic code generation from UML state chart diagrams," *IEEE Access*, vol. 7, pp. 8591–8608, 2019, doi: 10.1109/ACCESS.2018.2890791.

[23] M. Jurgelaitis and V. Drungilas, "Modelling principles for blockchain-based implementation of business or scientific processes," in *Proc. IVUS*, Kaunas, Lithuania, 2019, pp. 43–47. [Online]. Available: http://ceur-ws.org/Vol-2470/p13.pdf

[24] M. Jurgelaitis and V. Drungilas, "Smart contract code generation from platform specific model for Hyperledger go," in *Proc. Adv. Intell. Syst. Comput.*, vol. 1368, 2021, pp. 63–73, doi: 10.1007/978-3-030-72654-6_7.

[25] The Eclipse Foundation. *ATL*. Accessed: Mar. 20, 2022. [Online]. Available: https://www.eclipse.org/atl/

[26] Obeo. *Acceleo*. Accessed: Mar. 20, 2022. [Online]. Available: https://www.eclipse.org/acceleo/

[27] K. Hu, J. Zhu, Y. Ding, X. Bai, and J. Huang, "Smart contract engineering," *Electronics*, vol. 9, no. 12, p. 2042, 2020, doi: 10.3390/electronics9122042.

[28] V. A. D. Sousa, C. Burnay, and M. Snoeck, "B-MERODE: A model-driven engineering and artifact-centric approach to generate blockchain-based information systems," in *Advanced Information Systems Engineering* (Lecture Notes in Computer Science), Cham, Switzerland: Springer, 2020, pp. 117–133, doi: 10.1007/978-3-030-49435-3_8.

[29] L. Marchesi, M. Marchesi, and R. Tonelli, "ABCDE—Agile block chain DApp engineering," *Blockchain, Res. Appl.*, vol. 1, nos. 1–2, 2020, Art. no. 100002, doi: 10.1016/j.bcra.2020.100002.

[30] Q. Lu, A. B. Tran, I. Weber, H. O'Connor, P. Rimba, X. Xu, M. Staples, L. Zhu, and R. Jeffery, "Integrated model-driven engineering of blockchain applications for business processes and asset management," *Softw., Pract. Exp.*, vol. 51, no. 5, pp. 1059–1079, May 2021, doi: 10.1002/spe.2931.

[31] H. Syahputra and H. Weigand, "The development of smart contracts for heterogeneous blockchains," in *Proc. I-ESA*, Berlin, Germany, 2019, pp. 229–238, doi: 10.1007/978-3-030-13693-2_19.

[32] G. A. Pierro, "Smart-graph: Graphical representations for smart contract on the Ethereum blockchain," in *Proc. SANER*, Honolulu, HI, USA, 2021, pp. 708–714, doi: 10.1109/SANER50967.2021.00090.

[33] T. Gorski and J. Bednarski, "Applying model-driven engineering to distributed ledger deployment," *IEEE Access*, vol. 8, pp. 118245–118261, 2020, doi: 10.1109/ACCESS.2020.3005519.

[34] M. Dumas, I. Weber, and A. Ponomarev, "CATERPILLAR: A business process execution engine on the Ethereum blockchain," 2019, *arXiv:1808.03517v3*.

[35] M. Skotnica, J. Klicpera, and R. Pergl, "Towards model-driven smart contract systems—Code generation and improving expressivity of smart contract modeling," in *Proc. EEWC*, Bolzano, Italy, 2020, pp. 1–15. [Online]. Available: http://ceur-ws.org/Vol-2825/paper1.pdf

[36] M. Wáhrer and U. Zdun, "Domain specific language for smart contract development," in *Proc. ICBC*, Toronto, ON, Canada, 2020, pp. 1–9, doi: 10.1109/ICBC48266.2020.9169399.

[37] M. Hamdaqa, L. A. P. Metz, and I. Qasse, "IContractML: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," in *Proc. SAM*, Montreal, QC, Canada, 2020, pp. 33–34, doi: 10.1145/3419804.3421454.

[38] V. Dwivedi and A. Norta, "Auto-generation of smart contracts from a domain-specific XML-based language," in *Intelligent Data Engineering and Analytic*. Singapore: Springer, 2021, doi: 10.13140/RG.2.2.34511.61609.

[39] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Auto-generation of smart contracts from domain-specific ontologies and semantic rules," in *Proc. iThings/GreenCom/CPSCom/SmartData/Blockchain/CIT*, Halifax, BC, Canada, 2018, pp. 963–970, doi: 10.1109/Cybermatics_2018.2018.00183.

[40] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *Int. J. Softw. Tools Technol. Transf.* vol. 20, no. 1, pp. 35–55, Feb. 2018, doi: 10.1007/s10009-016-0442-1.

[41] N. Zupan, P. Kasinathan, J. Cuellar, and M. Sauer, "Secure smart contract generation based on Petri nets," in *Blockchain Technology for Industry 4.0*. Singapore: Springer, 2020, pp. 73–98, doi: 10.1007/978-981-15-1137-0_4.

[42] K. Boogaard, "A model-driven approach to smart contract development," M.S. thesis, Dept. Inf. Comput. Sci., Utrecht Univ., Utrecht, The Netherlands, 2018.

[43] Ethereum. (2021). *Solidity Documentation*. Accessed: Mar. 20, 2022. [Online]. Available: https://docs.soliditylang.org/

[44] Ethereum. *Solidity by Example|Simple Open Auction*. Accessed: Mar. 20, 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/solidity-by-example.html#simple-open-auction

[45] Ethereum. *Solidity by Example|Safe Remote Purchase*. Accessed: Mar. 20, 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/solidity-by-example.html#safe-remote-purchase

[46] Ethereum. *Common Patterns/State Machine*. Accessed: Mar. 20, 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine

[47] ConsenSys Software. *VSCode Solidity Metrics*. Accessed: Mar. 20, 2022. [Online]. Available: https://github.com/ConsenSys/vscode-solidity-metrics/

[48] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2874–2891, Dec. 2021, doi: 10.1109/TSE.2020.2971482.

MANTAS JURGELAITIS received the B.Sc. and M.Sc. degrees in computer science from the Kaunas University of Technology, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree with the Faculty of Informatics. His research interests include UML modeling, MDA, model transformations, code generation, software, and blockchain-based systems engineering.

LINA ČEPONIENĖ received the B.Sc., M.Sc., and Ph.D. degrees from the Kaunas University of Technology, in 2000, 2002, and 2006, respectively. She is currently an Associate Professor and the Head of the Department of Information Systems, Faculty of Informatics, Kaunas University of Technology. Her research interests include UML modeling, MDA, model transformations, code generation, and blockchain-based systems engineering.

RITA BUTKIENĖ received the B.Sc., M.Sc., and Ph.D. degrees from the Kaunas University of Technology, in 1993, 1995, and 2002, respectively. She is currently an Associate Professor at the Kaunas University of Technology, where she is also the Dean of the Faculty of Informatics. Her research interests include information system engineering, ontologies, semantic technologies, databases, distributed ledger, information extraction, and retrieval.