RCTA
Revista Colombiana de Tecnologías de Avanzada
UNIPAMPLONA

Digital Object Identifier: 10.24054/rcta.v1i43.2778

Model Driven Engineering tool aimed at the generation of Smart Contracts for the Ethereum Blockchain platform.

Herramienta Model Driven Engineering destinada a la generación de Contratos Inteligentes para la plataforma de Blockchain Ethereum

MSc. Edgar Dulce [1,2,3], PhD. Julio Hurtado[2], MSc. Eduard Mantilla [1], MSc. Yenny Nuñez[1], PhD. José García-Alonso[3]

[1] **Universidad Nacional Abierta y a Distancia UNAD,** *Grupo de investigación DAVINCI, Bogotá, Colombia.*
[2] **Universidad del Cauca,** *Grupo de investigación IDIS, Popayán, Cauca, Colombia.*
[3] **Universidad de Extremadura,** *Grupo de investigación QUERCUS, Cáceres, Extremadura, España.*

Correspondence: *edgar.dulce@unad.edu.co*

Received: *November 8, 2023.* ***Accepted:*** *January 10, 2024.* ***Published:*** *February 21, 2024.*

How to Cite: *E. R. Dulce Villarreal, J. A. Hurtado Alegría, E. A. Mantilla Torres, Y. S. Núñez Álvarez, and J. M. García Alonso, "Model Driven Engineering tool aimed at the generation of Smart Contracts for the Ethereum Blockchain platform", RCTA, vol. 1, no. 43, pp. 1–8, Feb. 2024.*
Recovered from https://ojs.unipamplona.edu.co/index.php/rcta/article/view/2778

Copyright 2024 Colombian Journal of Advanced Technologies.
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Abstract: Blockchain technology is growing at a rapid pace in different environments. Smart contracts (SC) are immutable decentralized programs for Blockchain platforms that enforce, monitor and execute agreements, without the intervention of a trusted third party. But, due to their specificities, their development is a complicated process, as there are architectural concerns of each platform, which developers must understand. In this paper, we present a Model Driven Engineering tool intended for the generation of SC for the Ethereum Blockchain platform, for the Solidity programming language. This tool is composed of a Platform Specific Metamodel and a Model to Text Transformation, which allow generating the source code of the SCs. Also, we present a proof of concept where we generate and implement a metamodel, a model and deploy SC in a healthcare environment. The results are satisfactory in terms of the syntax of the generated SCs.

Keywords: Blockchain, Ethereum, MDE, Smart Contract, Solidity.

Resumen: La tecnología Blockchain está creciendo a un ritmo acelerado en diferentes entornos. Los contratos inteligentes (SC) son programas descentralizados inmutables para plataformas Blockchain que hacen cumplir, monitorear y ejecutar acuerdos, sin la intervención de un tercero de confianza. Pero, debido a sus especificidades, su desarrollo es un proceso complicado, ya que existen restricciones arquitectónicas de cada plataforma, que los desarrolladores deben comprender. En este trabajo, presentamos una herramienta Model Driven Engineering destinada a la generación de SC para la plataforma de Blockchain Ethereum, para el lenguaje de programación Solidity. Esta herramienta está compuesta de un Metamodelo Especifico de la Plataforma y una Transformación de Modelo a Texto, que permiten generar el código fuente de los SC. También, presentamos una prueba de concepto donde generamos e implementamos un metamodelo, un modelo y desplegamos SC en un entorno sanitario. Los resultados son satisfactorios en cuanto a la sintaxis de los SC generados.

Palabras clave: Blockchain, Contrato Inteligente, Ethereum, MDE, Solidity.

1. INTRODUCTION

Smart contracts (SCs) are immutable decentralized programs deployed on Blockchain (BC) platforms to enforce, monitor and execute agreements, without the intervention of a trusted third party. An SC allows inserting business logic into transactions and sharing them in an interoperable way with other BC [1]. The term SC was coined by lawyer and computer scientist Nick Szabo in 1996 [1]. With the use of robust cryptographic protocols, Szabo recognized the possibility of writing software that resembled contractual clauses, which would be binding on the parties and reduce their chances of non-compliance. While this was a novel idea in the 1990s, the technology needed for its proper development was lacking. It was only in 2008 when the development of BC technology provided the necessary platform and ecosystem for SCs [2]. SCs enable BC to play a vital role in many fields, such as finance and healthcare.

Most SCs are simple programs that define a set of rules governing the contractual agreement process between the parties. Despite being simple, SC development is challenging. This is due to the complexity and heterogeneity of the underlying platforms used to create and implement SCs [3].

In this paper, a detailed analysis of the official Ethereum documentation [4]. Then applying the Model Driven Engineering (MDE) methodology to create a metamodel, a model according to this metamodel and the necessary transformation to generate the source code of the SCs. All of the above, with some artifacts required in the Solidity programming language[1], which is the official language for Ethereum[2]. Ethereum is currently one of the most widely used BC platforms [5].

The rest of the paper is organized as follows: Section 2, provides an analysis of the presented problem. Next, section 3 discusses the methodology. Next, in Section 4 we present our platform-specific model for SC generation for Ethereum BC platforms, describe the entire MDE ecosystem required to improve its interoperability, and describe the metamodel generated for the Ethereum platform, for the Solidity programming language. We present a model-to-text transformation (m2t), in section 5, along with the program created in Acceleo[3] that performs the process. In section 6 we present a proof of concept, in which we develop a SC for patient management in a healthcare environment, in this process, we show the implementation, compilation and deployment of a SC. At the end of the paper, we present Conclusions and References.

2. ANALYSIS OF THE PRESENTED PROBLEM

What distinguishes an SC from a normal application is that its code is implemented on a BC platform. This close relationship between SC and BC introduces architectural and platform-specific constraints that developers must understand to create SC applications [6]. In addition, the platform heterogeneity that manifests itself in the multiple BC platforms that a developer can target to implement their code adds another layer of complexity. Particularly, because these platforms require different types of implementation models and artifacts and do not follow specific standard or unified terminologies to specify these models [7]. An important aspect of SC modeling and implementation is to define the message exchange process and the rules governing the agreements under which the corresponding actions are executed [7].

Also, as with most technologies, there are potential security threats, vulnerabilities, and other issues associated with SCs. Writing safe and secure SCs can be extremely difficult due to various business logics, as well as platform vulnerabilities and limitations [8]. The problems encountered in SCs are classified depending on the consensus mechanism used, the quality of the contract source code, lack of standard programming languages, among others [9]. Moreover, one of the biggest challenges in implementing a SC is how to unify the contract execution environments and programming language, since various BC systems may adopt different execution platforms and scripting languages of SCs [10]. By allowing Turing-

[1] https://docs.soliditylang.org/en/v0.8.21

[2] https://ethereum.org/en

[3] https://wiki.eclipse.org/Acceleo/

complete programming languages to implement SCs, recent BC, such as Ethereum, can reduce the needs for trusted intermediaries, arbitration, and execution costs. However, subtle bugs in SCs have led to huge economic losses, such as DAO attack[4], attacks on wallets with multiple parity signatures, and integer overflow attacks [11].

In relation to the above, we can say that SCs are a relatively new technology and are in a growth phase. Thus, greater abstraction and automation are key to mastering the complexity inherent in the process of building SCs, and the models are intended to obtain all the advantages that were once achieved with programming languages: a reduction of the semantic leap between the way in which developers think about solutions and the way in which they must express them, which results in less effort in the task of programming and therefore in greater productivity, more understandable programs and less costly maintenance [12].

3. METHODOLOGY

The MDE methodology is composed of the following principles [12]:

- A model represents totally or partially a part of a software system;
- These models are represented with domain specific languages (DSL) also called "modeling languages";
- A metamodel is used to formally represent a DSL;
- Automation is usually achieved through the translation of models to code by model transformations.

An increase in the level of abstraction must be accompanied by an increase in the level of automation to be truly effective. In the case of MDE this is achieved by automatically generating code from the models created, either directly through m2t transformations, which is very complicated when one does not have well-defined mature models, or indirectly by defining intermediate models generated with model-to-model (m2m) transformations that facilitate the conversion of high-level abstraction models into the final code [13].

4. A PLATFORM SPECIFIC MODEL (PSM) FOR SC GENERATION FOR ETHEREUM BC PLATFORMS.

Considering the above and to relieve developers from dealing with this platform-specific complexity of BC platforms, and allow them to focus on the business process, rather than the syntax details of each BC platform, in this paper we start with one of the important points of the whole set of tools needed to achieve complete interoperability of the entire BC ecosystem (Fig. 1). In the work we presented and fully described in [6], we proposed a MDE experiment based on a 4-level architecture [12], which is summarized in Fig. 1. In this opportunity, we developed a metamodel (MM-A in Fig. 1), from which different PSMs can be created. Also, we create an m2t transformation in the Acceleo tool, which will be in charge of generating the source code of a SC for the Solidity.

4.1 Analysis

In this section, we analyze and describe the components present in Fig. 1.

At the core of our tool, there is a specific metamodel for the Ethereum BC platform (Fig. 2). For its construction, eCore was used as the metamodeling language, which is part of the Eclipse Modeling Framework EMF metamodeling architecture [14].

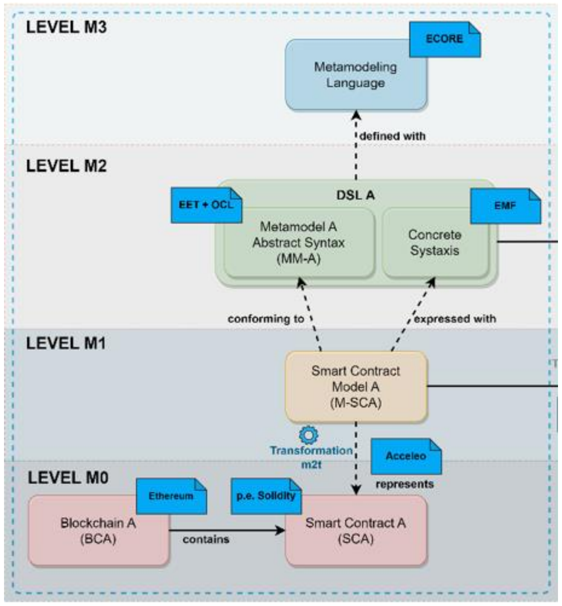


Fig. 1. *Description of the MDE process for the proposed experiment.* ***Source****: Based on the 4-level architecture [13].*

[4] https://blog.chain.link/reentrancy-attacks-and-the-dao-hack

For the construction of the metamodel, we followed the interactive and iterative approach proposed in [15], which allows the specification of model fragments by domain experts.

These fragments can be annotated with descriptions of the intent or requirements of particular elements. A metamodel is automatically induced, which can be interactively refactored and then compiled into an implementation metamodel for different platforms and purposes. In our case for the Ethereum BC platform. Also, we have reviewed other contributions to go adding the components present in the metamodel, these are presented in section 6.

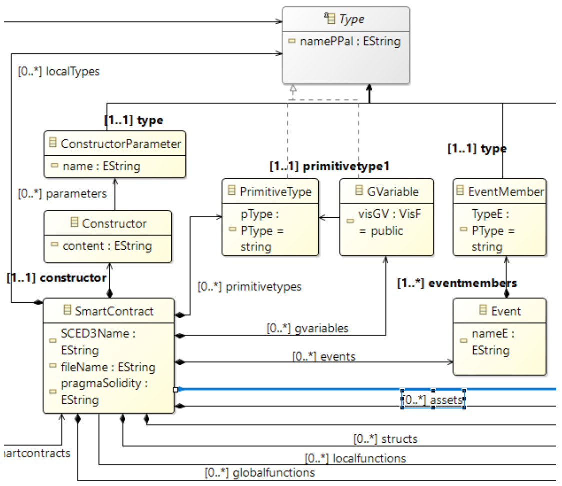


Fig. 2. *SCED3 - eCore metamodel for Ethereum, for Solidity programming language.* ***Source****: own elaboration.*

Since our metamodel is platform-dependent (PSM), it enables the creation of models to improve interoperability between models that are generated in different BCs. From a model we can generate a SC with enough richness and it gives us the possibility that this model can be used with other models to form a more abstract infrastructure and thus facilitate the interoperability of SCs coming from different platforms.

All the code of a SC in Solidity is generated in a file with the extension *.sol*. The main class of our metamodel is SmartContract, from which the other classes that represent the structure of a contract in Solidity are derived.

The following is a description of each of the classes of the proposed metamodel:

- Reposotory: is a superclass in which several SCs are stored, as it is handled in other programming languages, for example, Java.
- Constructor and Constructor Parameter: Represents the constructor of the SC or owner of the contract, together with the specific parameters of this owner.
- User: Within an SC, there can be two other user types (user and thirdparty), for other participants and a third party, such as a notary. These are handled by the TUsers enumeration.
- Primitive Type: Used for primitive types of the language, handled by the PType enumeration. Some examples are: string, int, money or bool.
- GVariable: To manage Global Variables, its visibility is handled by the VisF enumeration, it can be: public, private, internal and external.
- Asset: Represents the assets that can be managed within a SC. It is represented by raw data that persist inside a SC and are stored inside a BC. This asset represents a value that can be tangible or intangible and its value is updated through Functions or Events.
- Mapping: It is a type of reference like arrays and structs, it allows referencing two or more types of data and managing them through a name.
- Event and Event Member: To manage events occurring in the logic of an SC. When an event is emitted, the arguments are stored in transaction logs in the BC and are accessible using the SC address.
- Struct: It is used to manage the construction of data structures, which are composed of other types of data (e.g., a patient type structure may be composed of a patient's ID, Name and Address).
- Instance Struct: It is used to instantiate structures created with the Struct class.
- Function, Local Function and GlobalFunction: Manages functions, which have the same behavior as in other programming languages, e.g., Java. In addition, they are used to create the necessary setters and getters.
- Function Parameter and Return: Used to manage the parameters of a function and the value it returns respectively.
- Type: It is an abstract class used to represent in class hierarchy the different types of data in solidity.

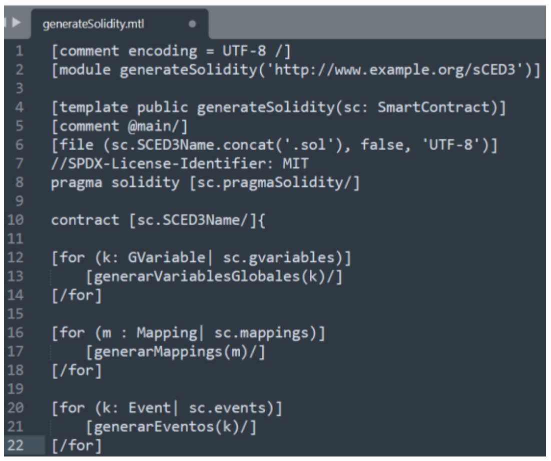
5. MODEL TO TEXT TRANSFORMATION (m2t) FOR SC ETHEREUM

As previously stated, by means of a set of transformations we can go from a high level of abstraction, to a very concrete level, in this case for SC generation. In this paper, one of the objectives is to create a m2t transformation, to generate the source code of a SC for the Ethereum platform, which is described below.

For the reason of not extending too much in the explanation, in Fig. 3, an extract of the source code of the generateSolidity.mtl file, created in Acceleo, is presented. This file is intended to generate the SC source code for the Ethereum platform, for the Solidity programming language. The complete source code can be found in the GitHub repository[5].

As can be seen in Fig. 3, in line 2, we make use of our SCED3 metamodel explained in the previous section. In line 4, the main template is defined and a variable called *sc* is created, with which we access the main class of the SCED3 metamodel called SmartContract, and with which we access other elements of the metamodel.

In line 5, the file containing the SC is created. In this case, accessing through the variable *sc* to the parameter SCED3Name, which contains the name of the contract. With the concat('.sol') function, the extension .sol is concatenated to the file (remember that .sol is the extension for solidity files). In line 8, the version of solidity that we are going to use is defined, this is obtained from the pragmaSolidity parameter of the SmartContract class.


generateSolidity.mtl

```
1   [comment encoding = UTF-8 /]
2   [module generateSolidity('http://www.example.org/sCED3')]
3
4   [template public generateSolidity(sc: SmartContract)]
5   [comment @main/]
6   [file (sc.SCED3Name.concat('.sol'), false, 'UTF-8')]
7   //SPDX-License-Identifier: MIT
8   pragma solidity [sc.pragmaSolidity/]
9
10  contract [sc.SCED3Name/]{
11
12  [for (k: GVariable| sc.gvariables)]
13      [generarVariablesGlobales(k)/]
14  [/for]
15
16  [for (m : Mapping| sc.mappings)]
17      [generarMappings(m)/]
18  [/for]
19
20  [for (k: Event| sc.events)]
21      [generarEventos(k)/]
22  [/for]
```

Fig. 3. *GenerateSolidity.mtl program created in Acceleo, for the m2t transformation.*
Source*: own elaboration.*

In line 10, the word contract and its name indicate the beginning of the SC. Lines 12, 16 and 20 contain 3 for cycles in charge of calling the functions generateGlobalVariables, generateMappings and generateEvents, in charge of generating Global Variables, Mappings and Events of a contract.

6. VALIDATION THROUGH A PROOF OF CONCEPT

This section describes the main steps and tools used for the creation, implementation and deployment of a SC.

6.1. Environment

As an environment for the creation, implementation and validation of SCs, we will take the healthcare environment, directly in the patient registration process for a medical center that supports its information systems using BC technology.
We will start from the assumption that each patient has the following attributes:

- IDPatient: identification of the patient.
- namePatient: name of a patient.
- agePatient: age of a patient.

Then, to improve its administration, it is necessary to have these attributes in a structure called patient. Likewise, with the example contract, it will be possible to perform functions such as: Register patients and consult patients.

6.2. Tools used

The following tools are used for each phase of the process:

- Metamodel and model construction: with the eCore metamodeling language, included in Eclipse Modelling Framework (EMF).
- Transformation m2t: Acceleo is used, which is a code generator that implements the m2t specification, supports functions of a high-quality code generator IDE: simple syntax, efficient code generation, advanced tools, among others [12].
- Implementation of SCs: Remix, which is a web IDE, is used to write, test and debug SCs in Solidity.

6.3. Model creation

Having already created the metamodel described in section 4, EMF gives us the possibility to create instances of the metamodel (EMF calls them dynamic instances), which will later be transformed into the source code of an SC. These instances

[5] https://github.com/edgardulce77/MDETool-EthereumSoliditySC.git

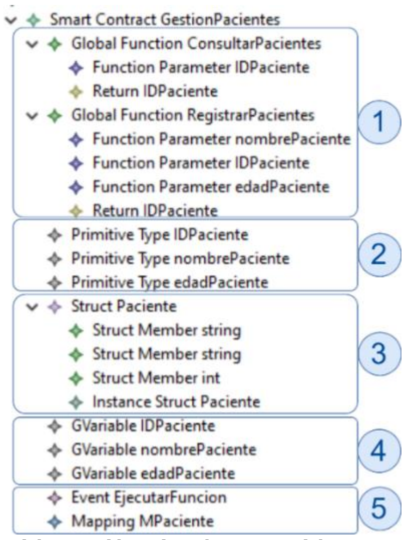
follow the XMI standard (XML Metadata Interchange or XML Metadata Interchange)[6]. In Fig. 3, a dynamic instance called "SmartContract" can be seen, which conforms to the metamodel proposed in Fig. 2, and which is described below:

For illustrative purposes, and to understand the usefulness of the metamodel, we will summarize the creation of some elements:

1. SmartContract Patient Management: This is the base element of the metamodel and from which the other elements are derived. It is composed of the SC name and the Solidity version, on which the SC will be compiled. The 5 elements shown in Fig. 4 are derived from it. Global Function ConsultPatients: It is a global function that will allow querying a given patient by its ID. Within this function a parameter has been defined called.
2. PrimitiveType: These are primitive data types, in this case they are: NamePatient, String type, IDPatient string type and agePatient int type.

- Struct Patient: It is a data structure to manage patients within the SC, it is composed of three Struct Members, one for each of the parameters of the structure: PatientName (string), PatientID (string) and PatientAge (int).

3. GVariable: these are the global variables used to identify the patients.
4. Event: manages events within a BC, such as the registration of a patient.
5. Mapping: Mapping that relates the ID and name of a patient.



```
Smart Contract GestionPacientes
  Global Function ConsultarPacientes
    Function Parameter IDPaciente
    Return IDPaciente
  Global Function RegistrarPacientes
    Function Parameter nombrePaciente
    Function Parameter IDPaciente
    Function Parameter edadPaciente
    Return IDPaciente
  Primitive Type IDPaciente
  Primitive Type nombrePaciente
  Primitive Type edadPaciente
  Struct Paciente
    Struct Member string
    Struct Member string
    Struct Member int
    Instance Struct Paciente
  GVariable IDPaciente
  GVariable nombrePaciente
  GVariable edadPaciente
  Event EjecutarFuncion
  Mapping MPaciente
```

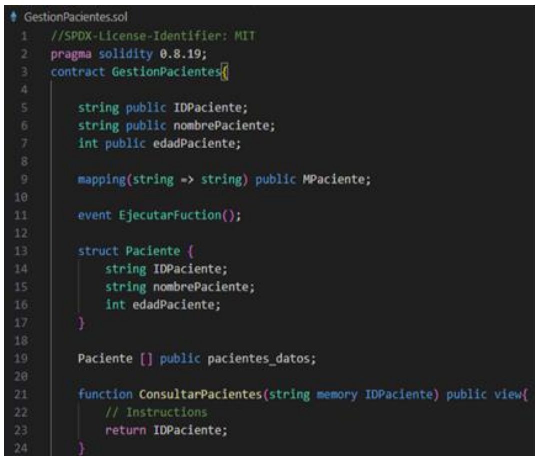
Fig. 4. *Model created based on the metamodel presented in Fig. 2.* ***Source****: own elaboration.*

6.4. Smart contracts generated

Now, having in mind the metamodel of Fig. 2 and the model of Fig. 4, with the help of the program generateSolidity.mtl created in Acceleo (Fig. 3), we run the m2t transformation, to generate the source code of the contract called GestionPacientes.sol. In Fig. 5, a part of the generated source code can be seen.

In Fig. 5, which is described below, several of the elements represented in the model of Fig. 3 can be seen:

- In line 2, the version of Solidity, in this case version 0.8.2, is seen.
- In line 4, the start of the SC called GestionPacientes.
- Between lines 5 to 7, 3 global variables IDPatient, namePatient and agePatient are created.
- In line 9, mapping called MPatiente, which relates two string fields.
- In line 11, an example of an event called executeFunction, which requires a string parameter.
- Between lines 13 to 17, a struct called Patient is created, with the fields of each of the patients.
- In line 19, the struct Patient is instantiated.
- Between lines 21 to 24, you can see the function ConsultPatients, its type and the parameter it returns (IDPatient).



```
GestionPacientes.sol
1   //SPDX-License-Identifier: MIT
2   pragma solidity 0.8.19;
3   contract GestionPacientes{
4   
5       string public IDPaciente;
6       string public nombrePaciente;
7       int public edadPaciente;
8   
9       mapping(string => string) public MPaciente;
10  
11      event EjecutarFuction();
12  
13      struct Paciente {
14          string IDPaciente;
15          string nombrePaciente;
16          int edadPaciente;
17      }
18  
19      Paciente [] public pacientes_datos;
20  
21      function ConsultarPacientes(string memory IDPaciente) public view{
22          // Instructions
23          return IDPaciente;
24      }
```

Fig. 5. *Source code of the GestionPacientes.sol contract.* ***Source****: own elaboration.*

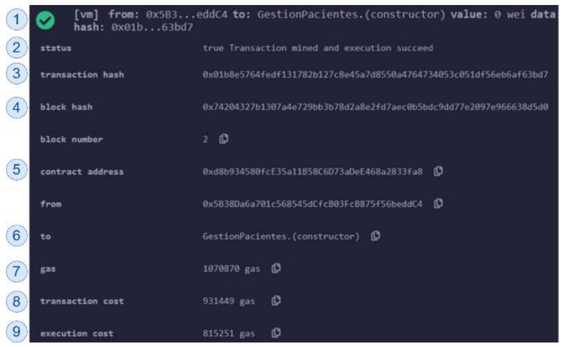
[6] http://www.omg.org/spec/XMI/

6.5. Validation of the generated contracts

The implementation of the generated contract was carried out using the Remix tool, dedicated to the development, compilation, deployment and testing of SCs programmed in Solidity. In Fig. 6, the results of this process can be seen.

Some results of the deployment are explained below:

- The check mark icon, tells us that the deployment was done correctly, in the same line the SC name is identified.
- Status: indicates that the contract was successfully mined and executed.
- Transaction Hash: is the hash of the transaction, to check its value.
- Block Hash: is the hash of the block in which the transaction was executed.
- Contract Address: is the 32-byte address of the SC.
- To: refers to the contract name, in this case GestionPacientes.
- Gas: is the cost of deploying the SC in the network.
- Transaction cost: is the cost of the transaction to deploy the contract.
- Execution cost: SC execution cost.



```
1  ✓  [vm] from: 0x5B3...eddC4 to: GestionPacientes.(constructor) value: 0 wei data
       hash: 0x01b...63bd7
2  status             true Transaction mined and execution succeed
3  transaction hash   0x01b8e5764fedf131782b127c8e45a7d8550a4764734053c051df56eb6af63bd7
4  block hash         0x74204327b1307a4e729bb3b78d2a8e2fd7aec0b5bdc9dd77e2097e966638d5d0
   block number       2
5  contract address   0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
   from               0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
6  to                 GestionPacientes.(constructor)
7  gas                1070870 gas
8  transaction cost   931449 gas
9  execution cost     815251 gas
```

Fig. 6. *Results of the deployment of the SC GestionPacientes.sol.* ***Source****: own elaboration.*

6.6. Analysis of Implementation Results

The deployment of the contract was satisfactory. The results indicate that our metamodel is able to generate many of the elements required in SCs for Ethereum BC platforms, preserving the syntax of the solidity programming language. Although in this study we only present the results with a proof of concept, to show that our metamodel is able to generate SC, and on these generate valid elements and artifacts, some additional evaluations are still missing, such as: evaluation of the quality of our metamodel and generated models, either by experts in the area or using some methodologies, such as MQuaRE tool, exposed by [17], which offers a set of artifacts to perform the evaluation of metamodels and also of the generated source code.

7. CONCLUSIONS

This paper presents a tool built following the MDE methodology for the generation of Smart Contracts on the Ethereum BC platform, for the Solidity programming language. This tool is composed of a metamodel, which is an abstraction of the main elements of the Solidity language, that allows modeling the main artifacts of the language and thus generating smart contracts. Likewise, the tool presents a Model-to-Text transformation (m2t) for the generation of the source code of the smart contracts, which was built in the Acceleo tool. Also, a proof of concept was performed regarding patient management in a healthcare environment, in which a model was created according to the metamodel presented and by means of the m2t transformation, the source code of the contract for patient registration and consultation was generated. In this test, the contract was implemented, deployed and compiled in a controlled environment, in which the satisfactory results are shown. However, in the whole process described, further studies and validations are needed to confirm the effectiveness and efficiency of the tool in other contexts, as well as to perform constant monitoring of the metamodel to ensure its relevance and adequacy to new demands and update of the Ethereum platform.

ACKNOWLEDGEMENT

We are deeply grateful to the DAVINCI, IDIS and QUERCUS research groups for all their support and accompaniment throughout the research process.

REFERENCES

[1] B. Aldughayfiq and S. Sampalli, “Digital Health in Physicians’ and Pharmacists’ Office: A Comparative Study of e-Prescription Systems’ Architecture and Digital Security in Eight Countries,” *OMICS*, vol. 25, no. 2, pp. 102–122, 2021, doi: 10.1089/omi.2020.0085.

[2] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Decentralized Business Review, p. 21260, 2008.

[3] W. Zou et al., "Smart contract development: Challenges and opportunities," *ITSE*, vol. 47, no. 10, pp. 2084–2106, 2019.

[4] P. Wackerow, "Documentación De Desarrollo De Ethereum," Aug. 2022.

[5] G. A. Oliva, et al., "An exploratory study of smart contracts in the Ethereum blockchain platform," *ESE*, vol. 25, no. 3, pp. 1864–1904, 2020, doi: 10.1007/s10664-019-09796-5.

[6] E. R. D. Villarreal, et al., "Blockchain for Healthcare Management Systems: A Survey on Interoperability and Security," *IEEE Access*, vol. 11, pp. 5629–5652, Jan. 2023, doi: 10.1109/ACCESS.2023.3236505.

[7] M. Hamdaqa, et al., "IcontractML: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms,", *SAM 2020*, 2020, pp. 34–44. doi: 10.1145/3419804.3421454.

[8] I. Qasse, et al., "IContractBot: A Chatbot for Smart Contracts' Specification and Code Generation,", *BotSE 2021*, 2021, pp. 35–38. doi: 10.1109/BotSE52550.2021.00015.

[9] D. Macrinici, et al., "Smart contract applications within blockchain technology: A systematic mapping study," *TIS*, vol. 35, no. 8, pp. 2337–2354, 2018, doi: 10.1016/j.tele.2018.10.004.

[10] H. Jin, X, et al. "Towards a novel architecture for enabling interoperability amongst multiple blockchains,", *ICDCS*, 2018, pp. 1203–1211.

[11] W. Nam and H. Kil, "Formal Verification of Blockchain Smart Contracts via ATL Model Checking," *IEEE Access*, vol. PP, p. 1, Aug. 2022, doi: 10.1109/ACCESS.2022.3143145.

[12] M. Brambilla, et al., Model-Driven Software Engineering in Practice: 2E, Milán, 2017.

[13] J. García, et al., "Desarrollo de Software Dirigido por Modelos Conceptos, Métodos y Herramientas", Madrid, 2013.

[14] F. Budinsky, Eclipse modeling framework: a developer's guide. AWP, 2004.

[15] N. Sanchez, et al., (05, 2022) *Blockchain smart contract meta-modeling*. Disponible: https://digital.cic.gba.gob.ar/handle/11746/11403.

[16] M. Hamdaqa, et al., "iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," *IST*, vol. 144, p. 106762, Apr. 2022, doi: 10.1016/J.INFSOF.2021.106762.

[17] G. C. Velasco, et al., "Evaluation of a High-Level Metamodel for Developing Smart Contracts on the Ethereum Virtual Machine," in AWB, 2023, pp. 29–42.