ISSN: 1692-7257 - Volumen 1 – Número 43 - 2024

RCTA
Revista Colombiana de Tecnologías de Avanzada
UNIPAMPLONA

Digital Object Identifier: 10.24054/rcta.v1i43.2778

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

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

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.*

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

Recibido: *8 noviembre 2023.* ***Aceptado:*** *10 enero 2024.* ***Publicado:*** *21 febrero 2024.*

Cómo citar: *E. R. Dulce Villarreal, J. A. Hurtado Alegría, E. A. Mantilla Torres, Y. S. Núñez Álvarez, y J. M. García Alonso, «Herramienta Model Driven Engineering destinada a la generación de Contratos Inteligentes para la plataforma de Blockchain Ethereum», RCTA, vol. 1, n.º 43, pp. 1–8, feb. 2024.*
Recuperado de https://ojs.unipamplona.edu.co/index.php/rcta/article/view/2778

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.

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.

1. INTRODUCCIÓN

Los contratos inteligentes SC (Smart Contract por sus siglas en inglés) son programas descentralizados inmutables desplegados en plataformas Blockchain (BC) para hacer cumplir, monitorear y ejecutar acuerdos, sin la intervención de un tercero de confianza. Un SC permite insertar lógica de negocio en las transacciones y compartirlas de una manera interoperable con otras BC [1]. El termino SC fue acuñado por el abogado y científico computacional Nick Szabo en 1996 [1]. Con el uso de protocolos criptográficos robustos, Szabo reconoció la posibilidad de escribir software que se asemejara a cláusulas contractuales, que fueran vinculantes para las partes y que redujeran sus posibilidades de incumplimiento. Si bien para los años noventa se trataba de una idea novedosa, no se contaba con la tecnología necesaria para su adecuado desarrollo. Fue sólo en el 2008 cuando el desarrollo de la tecnología BC ofreció la plataforma y el ecosistema necesarios para los SC [2]. Los SC permiten que BC desempeñe un papel vital en muchos campos, como las finanzas y la atención médica.

La mayoría de los SC son programas simples que definen un conjunto de reglas que rigen el proceso de acuerdo contractual entre las partes. A pesar de ser simples, el desarrollo de SC es un desafío. Esto se debe a la complejidad y heterogeneidad de las plataformas subyacentes que se utilizan para crear e implementar los SC [3].

En este documento se realiza un análisis detallado de la documentación oficial de Ethereum [4]. Luego aplicando la metodología de Ingeniería Dirigida Por Modelos MDE (Model Driven Engineering) para crear un metamodelo, un modelo acorde a este metamodelo y la transformación necesaria para generar el código fuente de los SC. Todo lo anterior, con algunos artefactos requeridos en el lenguaje de programación Solidity[1], el cual es el lenguaje oficial para Ethereum[2]. En la actualidad Ethereum es una de las plataformas de BC más utilizadas [5].

El resto del documento está organizado de la siguiente forma: La sección 2, proporciona un análisis de la problemática presentada. A continuación, en la sección 3 se aborda la metodología. Seguidamente, en la sección 4 presentamos nuestro modelo específico de la plataforma para la generación de SC para plataformas BC Ethereum, describimos todo el ecosistema MDE requerido para mejorar su interoperabilidad y describimos el metamodelo generado para la plataforma Ethereum, para el lenguaje de programación Solidity. Presentamos una transformación modelo a texto (m2t), en la sección 5, junto con el programa creado en Acceleo[3] que realiza el proceso. En la sección 6 presentamos una prueba de concepto, en la cual desarrollamos un SC para la gestión de pacientes en un entorno sanitario, en este proceso, mostramos la implementación, compilación y despliegue de un SC. Al final del documento, presentamos las Conclusiones y Referencias.

2. ANALISIS DE LA PROBLEMÁTICA PRESENTADA

Lo que distingue a un SC de una aplicación normal es que su código se implementa en una plataforma BC. Esta estrecha relación entre el SC y BC introduce restricciones arquitectónicas y específicas de la plataforma que los desarrolladores deben comprender para crear aplicaciones de SC [6]. Además, la heterogeneidad de la plataforma que se manifiesta en las múltiples plataformas BC a las que un desarrollador puede apuntar para implementar su código, agrega otra capa de complejidad. Particularmente, porque estas plataformas requieren diferentes tipos de modelos y artefactos de implementación y no siguen un estándar específico o terminologías unificadas para especificar estos modelos [7]. Un aspecto importante del modelado y la implementación de SC, es definir el proceso de intercambio de mensajes y las reglas que rigen los acuerdos bajo los cuales se ejecutan las acciones correspondientes [7].

[1] https://docs.soliditylang.org/en/v0.8.21
[2] https://ethereum.org/en
[3] https://wiki.eclipse.org/Acceleo/

Asimismo, como ocurre con la mayoría de las tecnologías, existen posibles amenazas a la seguridad, vulnerabilidades y otros problemas asociados con los SC. La redacción de SC seguros y protegidos puede ser extremadamente difícil debido a diversas lógicas comerciales, así como a las vulnerabilidades y limitaciones de la plataforma [8]. Los problemas encontrados en los SC se clasifican dependiendo del mecanismo de consenso utilizado, la calidad del código fuente del contrato, falta de lenguajes de programación estándar, entre otros [9]. Además, uno de los mayores desafíos en la implementación de un SC es cómo unificar los entornos de ejecución del contrato y el lenguaje de programación, ya que varios sistemas BC pueden adoptar diferentes plataformas de ejecución y lenguajes de escritura de los SC [10]. Al permitir que los lenguajes de programación Turing-completos implementen SC, las BC recientes, como Ethereum, pueden reducir las necesidades de intermediarios de confianza, arbitrajes y costes de ejecución. Sin embargo, errores sutiles en los SC han provocado enormes pérdidas económicas, como por ejemplo el ataque DAO[4], los ataques a monederos con múltiples firmas de paridad y los ataques de desbordamiento de enteros [11].

En relación con lo anterior, podemos decir que los SC son una tecnología relativamente nueva y se encuentra en fase de crecimiento. Así pues, una mayor abstracción y automatización son claves para dominar la complejidad inherente al proceso de construcción de los SC, y con los modelos se pretende obtener todas las ventajas que en su momento se consiguieron con los lenguajes de programación: una reducción del salto semántico entre la forma en la que los desarrolladores piensan las soluciones y la forma en la que deben expresarlas, la cual redunda en un menor esfuerzo en la tarea de programar y por tanto en una mayor productividad, en programas más comprensibles y en un mantenimiento menos costoso [12].

3. METODOLOGÍA

La metodología MDE, está compuesta de los siguientes principios [12]:

- Un modelo representa total o parcialmente una parte de un sistema software;
- Estos modelos son representados con lenguajes específicos del dominio (DSL) también denominados “lenguajes de modelado”;
- Un metamodelo es empleado para representar formalmente un DSL;
- La automatización es normalmente conseguida a través de la traducción de los modelos a código mediante transformaciones de modelos.

Un aumento en el nivel de abstracción debe ir acompañado de un aumento del nivel de automatización para que realmente sea efectivo. En el caso del MDE esto se consigue mediante la generación automática de código a partir de los modelos creados, ya sea directamente a través de transformaciones m2t, lo cual es muy complicado cuando no se tienen modelos maduros bien definidos, o de forma indirecta mediante la definición de modelos intermedios generados con transformaciones modelo a modelo (m2m) que facilitan la conversión de modelos de alto nivel de abstracción en el código final [13].

4. UN MODELO ESPECÍFICO DE PLATAFORMA (PSM) PARA LA GENERACIÓN DE SC PARA PLATAFORMAS BC ETHEREUM

Teniendo en cuenta lo anterior y para aliviar a los desarrolladores de lidiar con esta complejidad específica las plataformas BC, y permitirles enfocarse en el proceso comercial, en lugar de los detalles de sintaxis de cada plataforma de BC, en este documento iniciamos con uno de los puntos importantes de todo el conjunto de herramientas necesarias para lograr una interoperabilidad completa de todo el ecosistema de BC (Fig. 1). En el trabajo que presentamos y describimos completamente en [6], propusimos un experimento MDE basado en una arquitectura de 4 niveles [12], el cual se resume en la Fig. 1. En esta oportunidad, desarrollamos un metamodelo (MM-A en la Fig. 1), del cual se pueden crear diferentes PSM. Asimismo, creamos una transformación m2t (modelo a texto) en la herramienta Acceleo, la cual se encargará de generar el código fuente de un SC para el lenguaje de programación Solidity.

4.1 Análisis

En este apartado, se analizan y describen los componentes presentes en la Fig. 1.

En el centro de nuestra herramienta, se encuentra un metamodelo específico para la plataforma de BC Ethereum (Fig. 2). Para su construcción, se utilizó

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

eCore como lenguaje de metamodelado, el cual es parte de la arquitectura de metamodelado de Eclipse Modeling Framework EMF [14].

Para la construcción del metamodelo, seguimos el enfoque interactivo e iterativo propuesto en [15], este, permite la especificación de fragmentos de modelos por parte de expertos en el dominio.

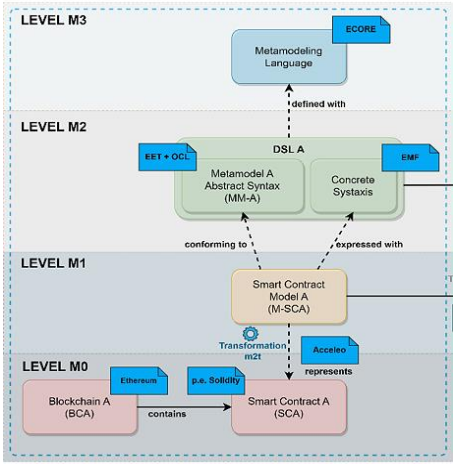

Fig. 1. *Descripción del proceso MDE para el experimento propuesto.* ***Fuente:*** *Basado en la arquitectura de 4 niveles [13].*

Estos fragmentos pueden anotarse con descripciones sobre la intención o las necesidades de determinados elementos. Se induce automáticamente un metamodelo, que puede refactorizarse de forma interactiva y, a continuación, compilarse en un metamodelo de implementación para diferentes plataformas y propósitos. En nuestro caso para la plataforma de BC Ethereum. También, hemos revisado otras contribuciones para ir agregando los componentes presentes en el metamodelo, estos se presentan en la sección 6.

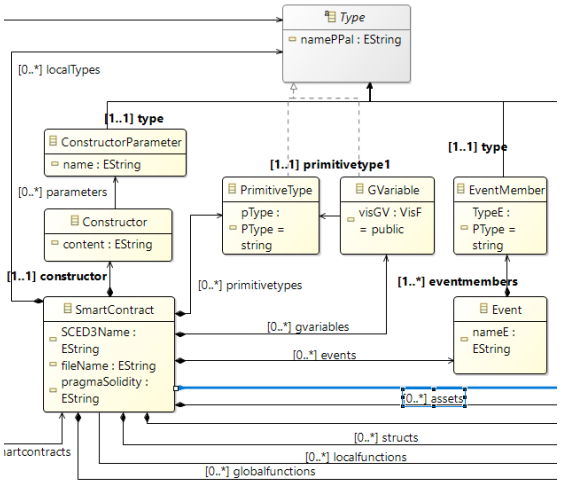

Fig. 2. *SCED3 - Metamodelo eCore para Ethereum, para leguaje de programación Solidity.* ***Fuente****: elaboración propia.*

Ya que nuestro metamodelo es dependiente de la plataforma (PSM), viabiliza la creación de modelos para mejorar la interoperabilidad entre los mismos, que son generados en diferentes BC. De un modelo podemos generar un SC con la suficiente riqueza y nos da la posibilidad de que este modelo, entre en juego con otros modelos, para conformar una infraestructura más abstracta y así facilitar la interoperabilidad de los SC provenientes de diferentes plataformas.

Todo el código de un SC en Solidity se genera en un archivo con la extensión .sol. La clase principal de nuestro metamodelo es SmartContract, del cual se derivan las demás clases que representan la estructura de un contrato en Solidity.
A Continuación, se describen cada una de las clases del metamodelo propuesto:

- Reposotory: es una superclase en la cual se almacenan varios SC, como se maneja en otros lenguajes de programación, por ejemplo, Java.
- Constructor y ConstructorParameter: Representa al constructor del SC o dueño del contrato, junto con los parámetros específicos de este dueño.
- User: Dentro de un SC, puede haber otros dos tipos de usuario (user y thirdparty), para otros participantes y un tercero, como por ejemplo un notario. Estos son manejados mediante la enumeración TUsers.
- PrimitiveType: Se utiliza para tipos primitivos del lenguaje, manejados mediante la enumeración PType. Algunos ejemplos son: string, int, money o bool.
- GVariable: Para gestionar las Variables Globales, su visibilidad se maneja mediante la enumeración VisF, puede ser: public, private, internal y external.
- Asset: Representa los activos que se pueden manejar dentro de un SC. Se representa mediante datos sin formato que persisten dentro de un SC y se almacenan dentro de una BC. Este activo representa un valor que puede ser tangible o intangible y su valor se actualiza mediante Funciones o Eventos.
- Mapping: Es un tipo de referencia como arrays y structs, permite referenciar dos o más tipos de datos y gestionarlos mediante un nombre.
- Event y EventMember: Para gestionar los eventos ocurridos en la lógica de un SC. Cuando se emite un evento, se almacenan los argumentos en registros de transacciones en la BC y son accesibles utilizando la dirección del SC.
- Struct: Se utiliza para manejar la construcción de estructuras de datos, las cuales se componen de

otros tipos de datos (p.e. una estructura tipo paciente, puede estar compuesta de un ID, Nombre y Dirección de un paciente).

- InstanceStruct: Es para instanciar estructuras creadas con la clase Struct.
- Function, LocalFunction y GlobalFunction: Gestiona las funciones, las cuales tienen el mismo comportamiento que en otros lenguajes de programación, p.e. en Java. Además, sirven para crear los setters y getters necesarios.
- FunctionParameter y Return: Se utiliza para manejar los parámetros de una función y el valor que retorna respectivamente.
- Type: Es una clase abstracta que se utiliza para representar en jerarquía de clases los diferentes tipos de datos en solidity.

5. TRANSFORMACIÓN MODELO A TEXTO (m2t) PARA SC ETHEREUM

Como se dijo anteriormente, mediante un conjunto de transformaciones podemos ir de un nivel alto de abstracción, a un nivel muy concreto, en este caso para la generación de SC. En este documento, uno de los objetivos es crear una transformación m2t, para generar el código fuente de un SC para la plataforma Ethereum, la cual se describe a continuación.

Por el motivo de no extendernos demasiado en la explicación, en la Fig. 3, se presenta un extracto del código fuente del archivo generateSolidity.mtl, creado en Acceleo. Este archivo tiene como finalidad generar el código fuente del SC para la plataforma Ethereum, para el lenguaje de programación Solidity. El código fuente se puede encontrar completo en el repositorio de GitHub[5].

Como se puede ver en la Fig. 3, en la línea 2, se hace uso de nuestro metamodelo SCED3 explicado en la sección anterior. En la línea 4, se define el template principal y se crea una variable llamada sc, con la cual accedemos a la clase principal del metamodelo SCED3 llamada SmartContract, y con la cual se acceden a otros elementos del metamodelo.

En la línea 5, se crea el archivo que contiene el SC. En este caso, accediendo mediante la variable sc al parámetro SCED3Name, que contiene el nombre del contrato. Con la función concat('.sol') se concatena la extensión .sol al archivo (recordemos que .sol es la extensión para archivos solidity). En la línea 8, se define la versión de solidity que vamos a utilizar, esta se obtiene del parámetro pragmaSolidity de la clase SmartContract.

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. *Programa generateSolidity.mtl creado en Acceleo, para la transformación m2t.*
Fuente*: elaboración propia.*

En la línea 10, la palabra contract y su nombre, indican el inicio del SC. Las líneas 12, 16 y 20, contienen 3 ciclos for encargados de llamar a las funciones generateGlobalVariables, generateMappings y generateEventos, encargas de generar Variables Globales, Mappings y Eventos de un contrato.

6. VALIDACIÓN MEDIANTE UNA PRUEBA DE CONCEPTO

En este apartado, se describen las principales etapas y herramientas utilizadas para la creación, implementación y despliegue de un SC.

6.1. Entorno

Como entorno para la creación, implementación y validación de los SC, tomaremos el entorno de la salud, directamente en el proceso de registro de pacientes para un centro médico que soporte sus sistemas de información mediante la tecnología BC. Partiremos del supuesto de que cada paciente tiene los siguientes atributos:

- IDPaciente: identificación del paciente.
- nombrePaciente: nombre de un paciente.
- edadPaciente: edad de un paciente.

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

Entonces, para mejorar su administración, se hace necesario tener estos atributos en una estructura denominada paciente. Asimismo, con el contrato de ejemplo, se podrán realizar funciones como: Registrar pacientes y consultar pacientes.

6.2. Herramientas utilizadas

Para cada fase del proceso se utilizan las siguientes herramientas:

- Construcción del metamodelo y modelo: con el lenguaje de metamodelado eCore, incluido en Eclipse Modelling Framework (EMF).
- Transformación m2t: se utiliza Acceleo, el cual es un generador de código que implementa la especificación m2t, soporta funciones de un IDE generador de código de alta calidad: sintaxis simple, generación de código eficiente, herramientas avanzadas, entre otras [12].
- Implementación de los SC: se utiliza Remix, el cual es un IDE web, que se utiliza para escribir, probar y depurar SC en Solidity.

6.3. Creación del modelo

Teniendo ya creado el metamodelo descrito en la sección 4, EMF nos da la posibilidad de crear instancias del metamodelo (EMF los denomina instancias dinámicas), que posteriormente serán transformadas al código fuente de un SC. Estas instancias siguen el estándar XMI (Intercambio de metadatos XML o XML Metadata Interchange por sus siglas en ingles)[6]. En la Fig. 3, se puede ver una instancia dinámica llamada “SmartContract”, que es conforme al metamodelo propuesto en la Fig. 2, y el cual se describe a continuación:

Por motivos ilustrativos, y para comprender la utilidad del metamodelo, resumiremos la creación de algunos elementos:

1. SmartContract GestionPacientes: Es el elemento base del metamodelo y del cual se desprenden los demás elementos. Está compuesto por el nombre del SC y la versión de Solidity, sobre la cual se va a compilar el SC. De este se desprenden los 5 elementos indicados en la Fig. 4. Global Function ConsultarPacientes: Es una función global que permitirá consultar a un determinado paciente por su ID. Dentro de esta función se ha definido un parámetro llamado.
2. PrimitiveType: Son tipos de datos primitivos, en este caso son: NombrePaciente, tipo String, IDPaciente tipo string y edadPaciente tipo int.
3. Struct Paciente: Es una estructura de datos para gestionar a los pacientes dentro del SC, está compuesta de tres Struct Member, uno para cada uno de los parámetros de la estructura: nnombrePaciente (string), IDPaciente (string) e edadPaciente (int).
4. GVariable: son las variables globales utilizadas para identificar a los pacientes.
5. Event: gestiona eventos dentro de una BC, tales como el registro de un paciente.
   Mapping: Mapping que relaciona el ID y nombre de un paciente.

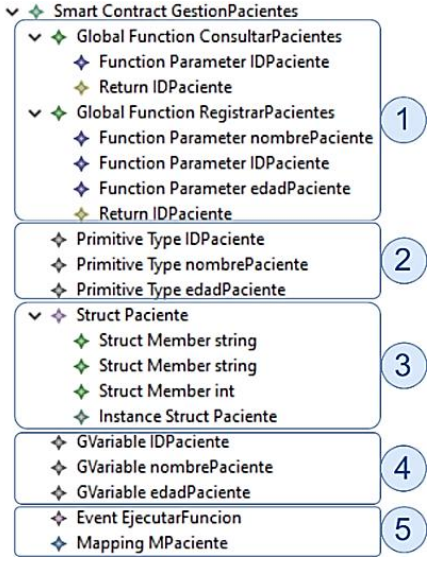

Fig. 4. *Modelo creado a partir del metamodelo presentado en la Fig. 2.*
Fuente*: elaboración propia.*

6.4. Contratos inteligentes generados

Ahora, teniendo presentes el metamodelo de la Fig. 2 y el modelo de la Fig. 4, con ayuda del programa generateSolidity.mtl creado en Acceleo (Fig. 3), ejecutamos la transformación m2t, para generar el código fuente del contrato llamado GestionPacientes.sol. En la Fig. 5, se puede ver una parte del código fuente generado.

En la Fig. 5, la cual se describe a continuación, se pueden ver varios de los elementos representados en el modelo de la Fig. 3:

- En la línea 2, se observa la versión de Solidity, en este caso la versión 0.8.2.

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

- En la línea 4, el inicio del SC llamado GestionPacientes.
- Entre las líneas 5 a 7, se crean 3 variables globales IDpaciente, nombrePaciente y edadPaciente.
- En la línea 9, mapping llamado MPaciente, que relaciona dos campos tipo string.
- En la línea 11, un ejemplo de evento llamado ejecutarFuncion, y el cual requiere un parámetro tipo string.
- Entre las líneas 13 a 17, se crea una struct llamada Paciente, con los campos de cada uno de los pacientes.
- En la línea 19, se instancia la struct Paciente.
- Entre las líneas 21 a 24, se puede ver la función ConsultarPacientes, su tipo y el parámetro que retorna (IDPaciente).

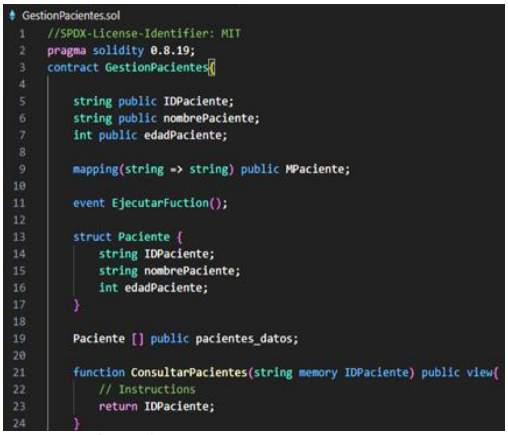

Fig. 5. *Código fuente del contrato GestionPacientes.sol.*
Fuente*: elaboración propia.*

6.5. Validación de los contratos generados

La implementación del contrato generado se realizó mediante la herramienta Remix[7], dedicada netamente al desarrollo, compilación, despliegue y prueba de SC programados en Solidity. En la Fig. 6, se pueden ver los resultados de todo este proceso.

A continuación, se explican algunos resultados del despliegue:

- El ícono de marca de chequeo, nos indica que el despliegue se hizo correctamente, en la misma línea se identifican el nombre del SC.
- Status: indica que el contrato fue minado y ejecutado satisfactoriamente.
- Transaction Hash: es el hash de la transacción, para comprobar su valor.
- Block Hash: es el hash del bloque en el cual se ejecutó la transacción.
- Contract Address: es la dirección de 32 bytes del SC.
- To: hace referencia al nombre del contrato, en este caso GestionPacientes.
- Gas: es el costo del despliegue del SC en la red.
- Transaction cost: es el costo de la transacción de desplegar el contrato.
- Execution cost: costo de ejecución del SC.

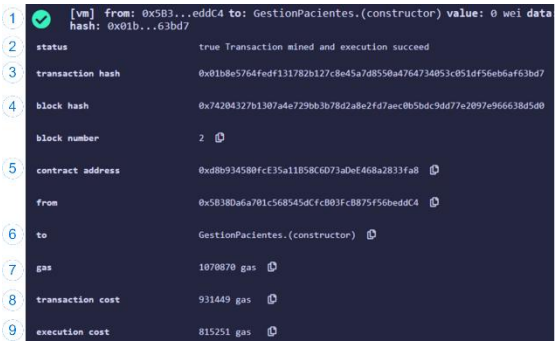

Fig. 6. *Resultados del despliegue del SC GestionPacientes.sol.*
Fuente*: elaboración propia.*

6.6. Análisis de los Resultados de la implementación

El despliegue del contrato fue satisfactorio. Los resultados indican que nuestro metamodelo es capaz de generar muchos de los elementos requeridos en los SC para plataformas de BC Ethereum, conservando la sintaxis propia del lenguaje de programación solidity. Si bien en este estudio solamente se presentan los resultados con una prueba de concepto, para mostrar que nuestro metamodelo es capaz de generar contratos inteligentes, y sobre estos generar elementos y artefactos válidos, aún faltan algunas evaluaciones adicionales, tales como: evaluación de la calidad de nuestro metamodelo y modelos generados, ya sea por parte de expertos en el área o utilizando algunas metodologías, como por ejemplo MQuaRE tool, expuesta por [17], que ofrece un conjunto de artefactos para realizar la evaluación de metamodelos y también del código fuente generado.

7. CONCLUSIONES

En este artículo se presenta una herramienta construida siguiendo la metodología MDE para la generación de contratos inteligentes en la plataforma de BC Ethereum, para el lenguaje de programación Solidity. Esta herramienta está

[7] https://remix.ethereum.org/

compuesta de un metamodelo, el cual es una abstracción de los elementos principales del lenguaje Solidity, que permite modelar los artefactos principales del lenguaje y así generar los contratos inteligentes. Asimismo, la herramienta presenta una transformación Modelo a Texto (m2t) para la generación del código fuente de los contratos inteligentes, la cual fue construida en la herramienta Acceleo. También, se realizó una prueba de concepto referente a la gestión de pacientes en un entorno sanitario, en la cual se creó un modelo conforme al metamodelo presentado y mediante la transformación m2t, se generó el código fuente del contrato para el registro y consulta de pacientes. En esta prueba, el contrato fue implementado, desplegado y compilado en un entorno controlado, en el cual se muestran los resultados satisfactorios. Sin embargo, en todo el proceso descrito, se necesitan más estudios y validaciones para confirmar la eficacia y eficiencia de la herramienta en otros contextos, así como realizar un seguimiento constante del metamodelo para garantizar su pertinencia y adecuación a las nuevas demandas y actualización de la plataforma Ethereum.

RECONOCIMIENTO

Agradecemos profundamente a los grupos de investigación DAVINCI, IDIS y QUERCUS, por todo el apoyo y acompañamiento en todo el proceso investigativo.

REFERENCIAS

[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.