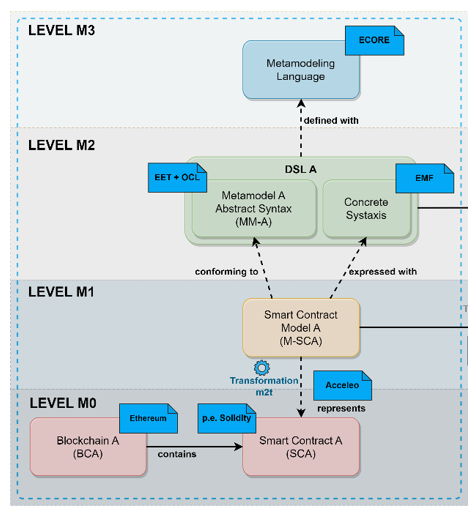
**A PLATFORM SPECIFIC MODEL (PSM) FOR SC GENERATION FOR ETHEREUM BC PLATFORMS.**

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 [1], we proposed a MDE experiment based on a 4-level architecture, 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.

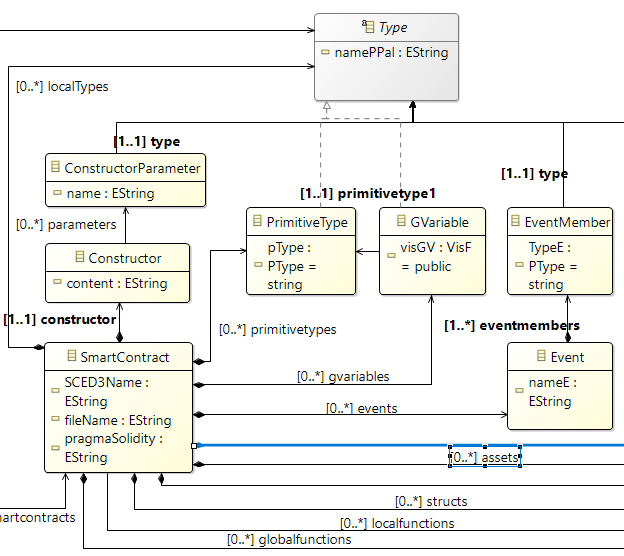
At the core of our tool, there is a specific metamodel for the Ethereum BC platform (Fig. 2). For its construction, eCore was used as the metamodeling language, which is part of the Eclipse Modeling Framework EMF metamodeling architecture [2].



***Fig. 1.*** *Description of the MDE process for the proposed experiment.* ***Source****:* ***Source****: own elaboration.*

For the construction of the metamodel, we followed the interactive and iterative approach proposed in [3], 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.

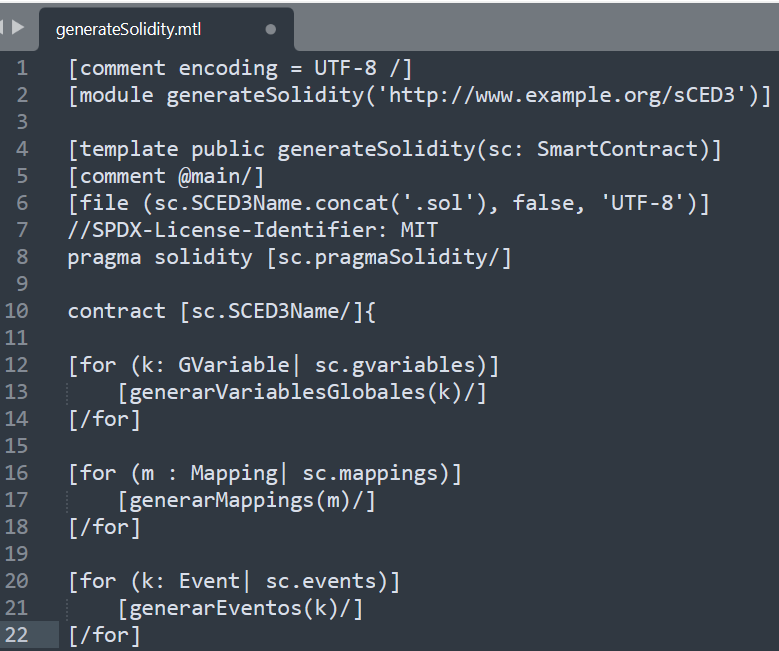
**5.** **MODEL TO TEXT TRANSFORMATION (m2t) FOR SC ETHEREUM**

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[[1]](#footnote-1).

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.



***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 [4].
* Implementation of SCs: Remix, which is a web IDE, is used to write, test and debug SCs in Solidity.

**6.3. Model creation**

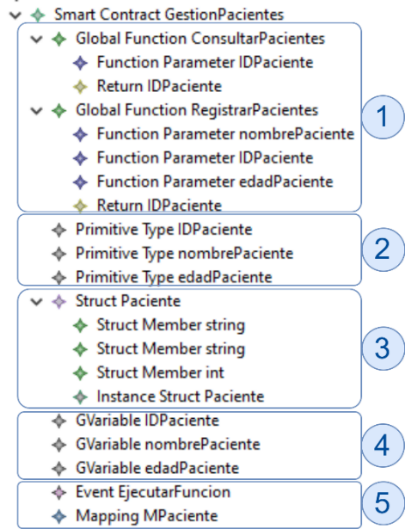
Having already created the metamodel described in section 4, EMF gives us the possibility to create instances of the metamodel (EMF calls them dynamic instances), which will later be transformed into the source code of an SC. These instances follow the XMI standard (XML Metadata Interchange or XML Metadata Interchange)[[2]](#footnote-2). 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).

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



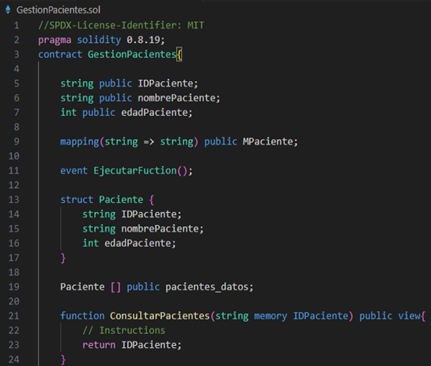
**Fig. 4.** Model created based on the metamodel presented in Fig. 2. **Source**: own elaboration.

**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).



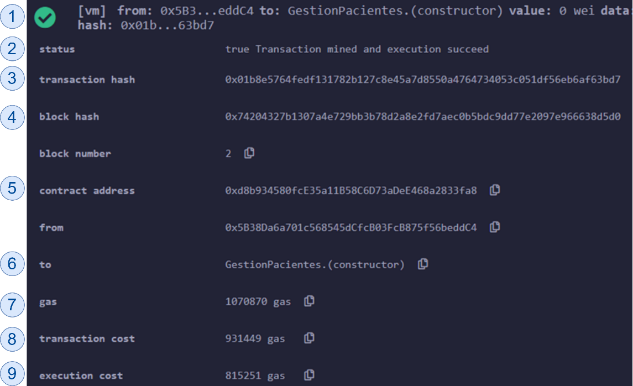
***Fig. 5.*** *Source code of the GestionPacientes.sol contract.* ***Source****: own elaboration.*

**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.



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

**REFERENCES**

[1] 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.

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

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

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

1. https://github.com/edgardulce77/MDETool-EthereumSoliditySC.git [↑](#footnote-ref-1)
2. http://www.omg.org/spec/XMI/ [↑](#footnote-ref-2)