

iContractML: A Domain-Specific Language for Modeling and Deploying Smart Contracts onto Multiple Blockchain Platforms

Mohammad Hamdaqa
mhamdaqa@ru.is
Reykjavik University
Reykjavik, Iceland

Lucas Alberto Pineda Metz
lucase18@ru.is
Reykjavik University
Reykjavik, Iceland

Ilham Qasse
ilham20@ru.is
Reykjavik University
Reykjavik, Iceland

ABSTRACT

Smart contracts are immutable digital programs deployed onto blockchain platforms to codify agreements. They enable blockchain technology to play a vital role in many fields, such as finance, health care, and energy. An important aspect of modeling and deploying smart contracts is to define the business process and rules that govern the agreements under which the corresponding actions are executed. Unfortunately, these models use a mix of technical and business-centric terminologies that are different based on the underlying blockchain platform that the smart contract is targeting. To address this issue, in this paper, we followed a feature-oriented domain analysis approach to identify the commonalities and variations between three of the common blockchain platforms that are used to deploy smart contracts; namely IBM Hyperledger Composer, Azure Blockchain Workbench, and Ethereum. Accordingly, we propose a reference model for smart contracts. The reference model is then realized as a modeling framework that enables developers to model and generate the structural code required to deploy a smart contract onto multiple blockchain platforms. The coverage of the proposed reference model was shown through mapping the concepts of the reference models to its corresponding constructs within each blockchain platform. Moreover, we provide three use cases to show how the proposed framework can empower developers to generate the structural code of smart contracts for the target platform through model transformation.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages.**

KEYWORDS

Smart Contracts, Blockchain, Model-Driven Engineering, Domain Specific Language, Ethereum, Hyperledger Composer, Azure Blockchain

1 INTRODUCTION

A smart contract is a decentralized software application that is deployed on the Blockchain, hence it inherits the blockchain's property of immutability [4]. Smart contracts are intended to replace

contractual agreements with self-executing and verifiable software code. Most smart contracts are simple programs that define a set of rules that govern the contractual agreement process between the contractual parties. Despite being simple, the development of smart contracts is challenging. This is due to the complexity and heterogeneity of the underlying platforms that are used to create and deploy the smart contracts [25]. As we will see in Section 2, different blockchain platforms use distinct terminologies and mandate the contract models to be specified according to the syntax defined by the platform.

To relieve the developers from dealing with this platform-specific complexity and enable them to focus on the business process rather than the syntax details of each blockchain platform, in this paper, we define a blockchain platform-independent modeling language for smart contracts, we call it iContractML.

The paper follows a feature-oriented domain analysis approach [15] to study the variations between the models used to specify smart contracts according to the specification of three widely used blockchain platforms (i.e., IBM Hyperledger Composer, Azure Blockchain Workbench, and Ethereum). We then apply model-driven engineering to build a modeling language and framework to enable the developers to model smart contracts and automatically generate the artifacts required to deploy the smart contract onto the target blockchain platform.

The main contributions of this paper are:

- (1) A reference model for smart contracts. The model was defined based on a comprehensive domain analysis of multiple smart contract platforms.
- (2) iContractML: A Domain-Specific Modeling Language for smart contracts modeling and development. Different from existing languages, iContractML supports multiple platforms and focuses on the smart contract structural components that are required for deployment.
- (3) An evaluation of the abstract and concrete syntax of iContractML was performed by mapping the concepts introduced by iContractML to its corresponding constructs on the target platforms to evaluate the fit of the proposed meta-model as well as the semiotic clarity of the concrete syntax. Moreover, the correctness of the generated deployment artifacts was evaluated using three use cases.

The rest of this paper is organized as follows: Section 2 presents a motivation example. Section 3 proposes a reference model for smart contracts based on a comprehensive domain analysis for smart contract platforms. Section 4 shows how iContractML is implemented. Section 5 evaluates the iContractML syntax and generated models. The related work is covered in Section 6. Finally, a conclusion and future research directions are presented in Section 7.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAM '20, October 19–20, 2020, Virtual Event, Canada

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8140-6/20/10...\$15.00

<https://doi.org/10.1145/3419804.3421454>

2 MOTIVATION EXAMPLE

What distinguishes a smart contract from a normal application is that the code of a smart contract is deployed onto a blockchain platform. This tight relationship between the smart contract and the blockchain introduces architectural and platform-specific constraints that developers are required to understand to build smart contract applications. Moreover, the platform heterogeneity that is manifested in the multiple blockchain platforms that a developer can target to deploy their code adds another layer of complexity. Particularly, because these platforms require different types of deployment models and artifacts and do not follow a specific standard or unified terminologies for specifying these models.

To shed light on some of these differences, let us consider the example in Table 1. The example shows part of the code of a medical insurance smart contract. We divided the code into blocks to make it easier to understand and compare the code segments. The two columns show the exact code logic specified according to the syntax of two major blockchain platforms. Those are Hyperledger Composer to the left, and Ethereum (written in Solidity) to the right in the table. The code specifies how to define a medical record, how to create a contractual party (i.e., the patient in this example), how to express the logic to change the patient information, and how to set the access rules and conditions on who can update the record. As shown in the table, while both contracts follow a similar structure, the syntax as well as the terminologies used by those two blockchain platforms varies significantly. For example, while Hyperledger uses a well-defined structure and uses business terms such as *asset* to define the contract state, Ethereum uses state variables to specify the same information. Similarly, while the contract method is referred to as a *transaction* in Hyperledger the Ethereum contract refers to it as *function*.

The mismatch between the artifacts that are required by different platforms adds to the complexity of building smart contracts. To address this mismatch, there is a need for a unified reference model for smart contracts to untangle the smart contract constructs from the underlying platform specifications, and to enable more efficient communication between the different stakeholders of the smart contracts. Moreover, there is a need for modeling frameworks to enable the developers to model and deploy smart contracts independently of the target blockchain platform.

In the next section, we explain our approach towards building and defining a unified reference model for smart contracts.

3 A REFERENCE MODEL FOR SMART CONTRACTS

This paper adopts a feature-oriented domain analysis approach to investigate the variations between the models used to define smart contracts. We start by domain analysis of three platforms that are commonly used to deploy smart contracts (i.e., IBM Hyperledger Composer, Azure Blockchain Workbench, and Ethereum). Accordingly, we define a unified reference model for smart contracts.

3.1 Blockchain Domain Analysis

To study the domain of the blockchain specific applications, we performed a bottom-up analysis [21], in which, we examined the artifacts used in the development and deployment of smart contracts

onto three widely used platforms; namely, Hyperledger Composer, Azure Blockchain Workbench, and Ethereum. These platforms have been selected based on their popularity and because they cover the different types of blockchain platforms, which are permissioned, blockchain as a service, and public platforms respectively [22]. Based on examining the artifacts, we created abstractions and meta-models to describe the smart contracts in each platform. The corresponding metamodels are referenced in the project repository¹.

3.1.1 Azure Blockchain Workbench Domain Analysis. Blockchain Workbench is a cloud service that is designed to create and deploy blockchain applications on Microsoft Azure Blockchain [3]. In the Azure Blockchain Workbench, the user presents their smart contract as two files. The first is a Javascript file (.json) that has the configuration and structure of the contract. This file contains the representation of the options that the user can use at the time of creating, submitting, and using a smart contract. The second file consists of the functions' code that will be executed once the conditions of the smart contract are satisfied. In this paper, we will focus on the .json file, since it represents the contract structure and interface. Based on the domain analysis for the Microsoft Azure, we have created the abstract model for the platform, which consists of the following classes:

- **json:** The main class for the structural file which contains the main constructs of a smart contract.
- **Application:** Outlines the information to register an application to the Azure Blockchain, that will handle the smart contracts and the users that can participate.
- **Workflow:** A state machine to control the flow of the business logic of the smart contract. It contains states and functions.
- **Identifier:** Represents a collection of information used to describe the Workflow, Construct, and Function.
- **Constructor:** Used to define the input parameters, through the application, necessary to instantiate a smart contract.
- **Function:** Used to define the function that is going to appear on the Solidity file (.sol) and executed in the workflow.
- **State:** A unique stage in the workflow. An example would be the state Request that indicates that the smart contract is requesting information from a user.
- **Transitions:** Actions that can be performed at a certain state that change the current state to another one.
- **Application role:** A contractual party or a participant.

3.1.2 Ethereum Blockchain Domain Analysis. Ethereum blockchain is an open source, public blockchain platform that introduced the concept of smart contract in 2015 [24]. Smart contracts in Ethereum can be written with different programming languages such as Solidity and Vyper. In this paper, we have analyzed the smart contracts written in Solidity. The description of the smart contracts in Solidity is done on the .sol file. The constructs of a smart contract on Solidity is mainly found on the Contract class, which is what we focus on in this study. The following fields represent the generic structure of the contract class:

¹https://drive.google.com/open?id=1rkagnH5x6vbcyL865X5sIM-siLNN4_N8

Table 1: Difference in the Syntax of Smart Contracts Between Hyperledger Composer and Ethereum

Comment	Hyperledger Composer	Ethereum
Define the contract state for the medical record field	Defined as an asset : <pre>asset MedicalRecord identified by recordId { o String recordId -> Patient owner }</pre>	Defined as a State Variable: <pre>bytes32 recordId; Patient private owner;</pre>
Define a patient as a contractual party	Defined as participant: <pre>participant Patient identified by patientId { o String patientId o String name o Integer age o Gender gender }</pre>	Defined as State variable or struct: <pre>struct Patient { bytes32 name; uint age; Gender gender; }</pre>
The method that can be used to execute the contract functionality	Defined as Transaction: <pre>transaction setPatientName{ o String name }</pre>	Defined as Function: <pre>function setPatientName(bytes32 name) public{ patient.name = name; }</pre>
A condition that specifies who can access the contract method	Defined as rule in access policy file : <pre>rule PatientSeelUpdateOwnMedicalRecord { description: "Patient can read and update their own record only" participant(t): "org.medical.Patient" operation: ALL resource(y): "org.medical.record" condition: (v.owner.getIdentifier() == t.getIdentifier()) action: ALLOW }</pre>	Defined as a Modifier: <pre>address public recordOwner; modifier onlyOwner() { require(msg.sender == recordOwner); -; }</pre>

- State or State Variables: Variables which their values are permanently stored in contract storage.
- Function: Executable units of code within a contract.
- Function Modifier: A decorator to define precondition constraints on a function. E.g. they can automatically check a condition before executing the function.
- Event: They are used to create a transaction log.
- StructType: The class is used to define new types, which is composed of state variables.
- EnumType: A way to create a user-defined type.

- Transactions: The mechanism by which participants interact with the assets.
- Relationship: A type of definition declare by the Asset and/or Transaction to represent that a Participant owns the asset or that is capable of submitting that transaction.
- Attribute: represent the variable instantiated on the different classes.
- Concepts: Used to define a new possible type for resources (variables) created inside the Asset, Participant, and Transaction classes.
- Enumtype: Are one way to create a user-defined type.

3.1.3 Hyperledger Composer Blockchain Domain Analysis. Hyperledger Composer is an open-source development tool that helps developers to create and manage blockchain business network applications that are deployed on the Hyperledger Fabric blockchain [6]. The application deployment package that is used by the Hyperledger Composer is called Business Network Archive (bna). The bna file is a composed of a Model File (.cto), Script File (.js), Access Control (.acl) and a Query File (.qry).

For this study, we are interested in the .cto file, which maintains the structure and main constructs of the smart contract on the Hyperledger Composer. Based on the domain analysis for the Hyperledger Composer, we have created the abstract model for the platform. The core classes of the abstract model are:

- Cto: Represents the .cto file, which contains the namespace of the network where the smart contracts are going to be published.
- Assets: Tangible or intangible goods, services, or property, that are stored in registries. Assets can represent almost anything in a business network.
- Participants: Members of a business network. They may own assets and submit transactions.

3.2 A Unified Reference Model for Smart Contracts

Based on the domain analysis for the targeted blockchain platforms, we were able to define the reference model in Figure 1.

In a nutshell, the modeling space of a smart contract defines a business process (*SContract*) that manages the relationships between a number of contractual parties (*Participants*). The contract defines those parties and gives them privileges that are stated as (*Conditions*) of who, how, and when a predefined process (*Transaction*) is executed in order to change the state of thing of intrinsic value to the participants (*Asset*). The state of this intrinsic value is stored on the Blockchain. It can represent a tangible or intangible good, service, or property, such as money, vehicles, real estate, or a liability.

We created a unified and abstract view of smart contracts represented in the Reference Model (UML metamodel) in Figure 1. This reference model defines the abstract syntax (metamodel) of the iContractML language. It was defined based on the variability analysis explained earlier. Hence the metamodel contains the elements required for a specifying a platform-independent model for a smart

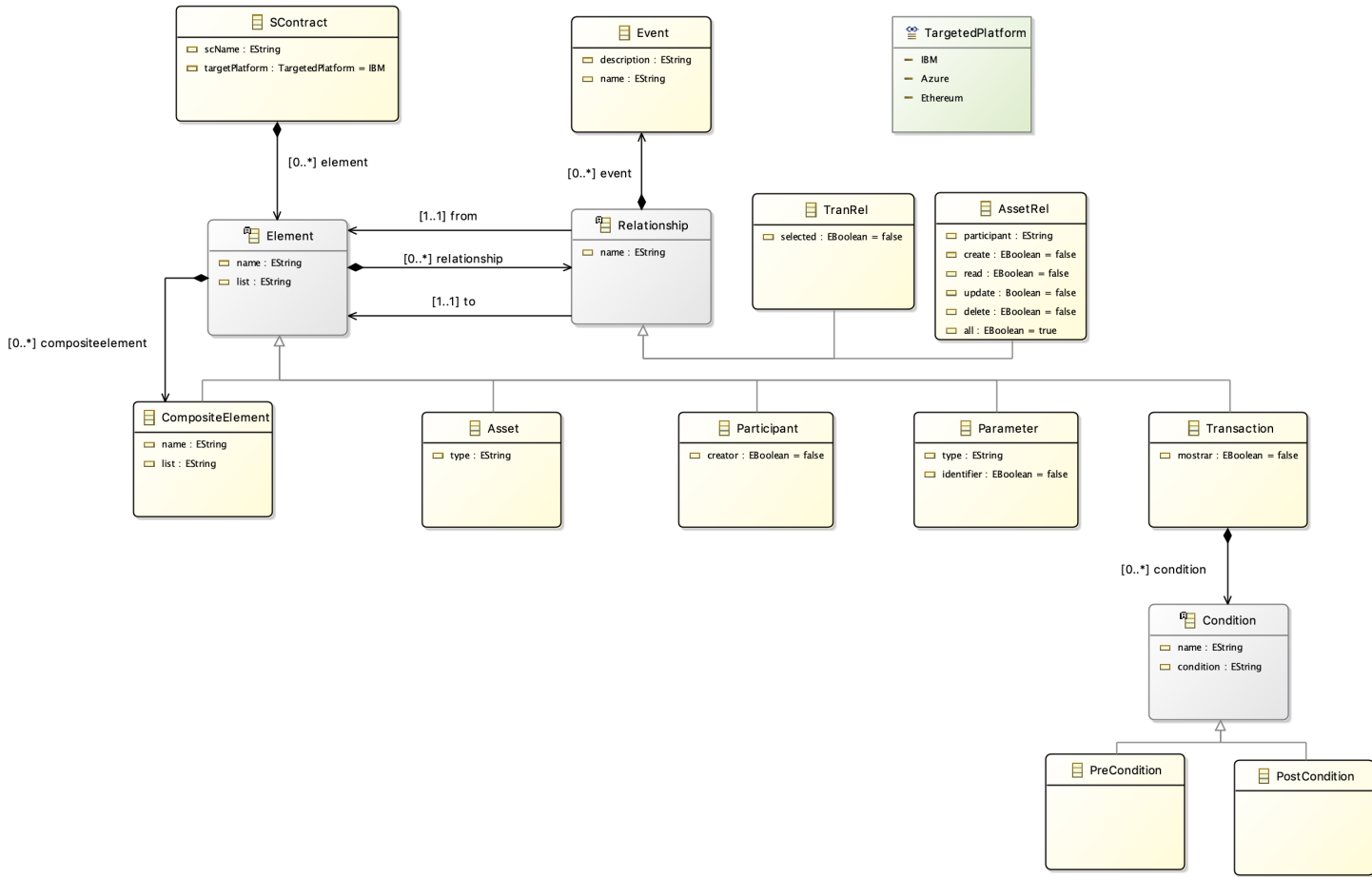


Figure 1: A Reference Model for Smart Contract

contract that can be used to generate the model artifacts based on the target blockchain. The following are the main components of the smart contract reference model:

- **SContract:** The main class that presents the smart contract, which contains everything that can be imposed onto a contract. When writing a smart contract one would have to define a name for it and the targeted platform (Hyperledger Composer, Microsoft Azure, Ethereum).
- **Participant:** This class is used to describe a contractual party. We can distinguish three generic types of participants; those are the contract owner who creates the contract, the beneficiary of the contract (a.k.a., the receiver) who would normally get affected by a change in the contract state (*Asset*), and the third-party observer who is not directly affected by the contract status. A smart contract must have an owner, who also defines the failover behavior of the contract. Moreover, the contract should at least define the relationship between two contractual parties (participants). The *creator* attribute in the participant is set to true or false to indicate if the participant is the owner (can instantiate the smart contract or not).

- **Transaction:** A class that outlines the public available functions in a smart contract that can change the contract status. A *Transaction* is accessible only for participants with access privilege and can be called from outside the blockchain. This access privilege is specified using a *TranRel* relationship that connect a *Transaction* to a *Participant*. The *Transaction* is one of the fundamental elements that define the contract semantic. In our modeling language, we follow a design by contract approach, when specifying a transaction, where each transaction is defined by the function itself in addition to post and pre-conditions. While some platforms (e.g., Azure) refer to this concept as a function others (e.g., Hyperledger composer) refer to it as a transaction. The different terminologies reflect the position from which someone refers to the concept. From a blockchain perspective, it is a transaction performed by a participant and sent to the blockchain. From a contract perspective, it is a public function that can be accessed from outside the blockchain through a transaction.
- **Asset:** This class represents a plain data object that persists the state of the contract and stored as a registry within a block on the blockchain. As mentioned earlier, an asset has

an intrinsic value that can be tangible or intangible and only updated through *Transactions*.

- **Relationship:** An abstract class that represents the relationships that different elements can have. They can contain Events with their own *description* and a name (*eveName*). These relationships can come in two types: The first one is *TranRel*, which is a relationship between Participant and Transaction. The second one is *AssetRel*, which is a relationship between Transaction and Asset.
- **Condition:** This concept represents a constraint to be satisfied before (Pre) or after (Post) the execution of a transaction. Preconditions are the required conditions to be met before executing the transaction. Postconditions are needed to be met after executing the transaction.
- **Event:** Specify a transaction log statement. An *Event* is optional as it is not supported by all platforms.

Other helper elements includes the *Element* and *Parameter* components. Those classes were added to ensure the genericity and extensibility of the metamodel. The *Element* is an abstract class used to represent everything that a contract can maintain. It is also possible for some of the elements to contain other kinds of elements (*CompositeElement*). On the other hand, the *Parameter* is a class that represents a user-defined type to be included in elements.

It is important to notice that the metamodel was defined to be generic enough so that it can be extended in the future as needed by adding more elements or relationships to support the syntactic and semantic rules of different platforms. For example, both the model *Element* and *Relationship* are defined as abstract classes so that other relationships and elements can be added if needed. Moreover, in our metamodel, the semantics are enforced through validation constraints. Those constraints can be relaxed or strengthened if needed. For example, while the abstract syntax of an *Element* allows connecting it with any other elements, using the validation rules, we constraint this relationship so that a participant cannot be directly connected to other participants. A participant can be connected only to a transaction using *TranRel* relationship. Similarly, a transaction can only be connected to an asset through an *AssRel* relationship.

To show the validity of the reference model, we mapped each concept in the reference model to its corresponding construct in the targeted platforms meta-models, as illustrated in Table 2. This mapping will be used in the quantitative evaluation section. The mappings in Table 2 is based on the core concepts. The mapping also shows the different artifacts (files) where the smart contract concepts are specified.

4 IContractML IMPLEMENTATION

The smart contract reference was specified in eCore then realized as visual Domain-Specific Modeling Language (DSML) for smart contract modeling. The eCore presents the abstract syntax and it is similar to Figure 1. In this section, we provide an overview of the modeling framework, we explain the concrete syntax, validation rules and the transformation templates that are used to support the automatic generation of the smart contract deployment artifacts (configurations) as well as the code skeleton for the smart contracts structural code.

4.1 The Concrete Syntax of iContractML

We used the OBEO Designer² to implement the concrete syntax of the model editor of iContractML. Figure 2 shows a screenshot of the modeling editor that shows (i) the model canvas (the red box), where the user can drag and drop components, (ii) the palette view (the green box) with all the elements that can be used to model a smart contract, and (iii) the properties view (the blue box), where the user can specify the attributes of the selected elements.

The screenshot also shows a sample model of a smart contract for a Digital Certificate. On the properties view, one can see the name as specified by the user and the targeted platform selected (IBM Hyperledger in this case).

The Digital Certificate model consists of one *asset* that is the Certificate, two *participants*; an Issuer and a Verifier, and three transactions that when executed can update or modify the state of the Certificate *asset*; those transactions are the Create Certificate, Accept Certificate, and Reject Certificate.

Each of the transactions is connected through a transaction relationship (solid line) to a participant and through an asset relationship (dashed lines) to the asset. The transaction relationship indicates who can execute the transaction. In the properties view of the transaction, the developer can further list all the pre- or post-conditions of the transaction. On the other hand, the asset relationship can be further annotated with events. In this example, there is an empty list on the top of the dashed line, which indicates that no events will be triggered and logged when the transaction is executed.

Notice that while only the names of the model components are shown on the model canvas, once an element is selected, all other attributes that can be added, deleted, or updated will be displayed on the properties view of that element. For example, using the properties view, the developer can add or delete attributes from the attribute list of the *asset*, and change their data types.

4.2 Validation Rules

In addition to the concrete syntax, the iContractML modeling framework implements a number of valuation rules to assure the validity of the model instances created using the modeling language. Some examples of the validation rules include:

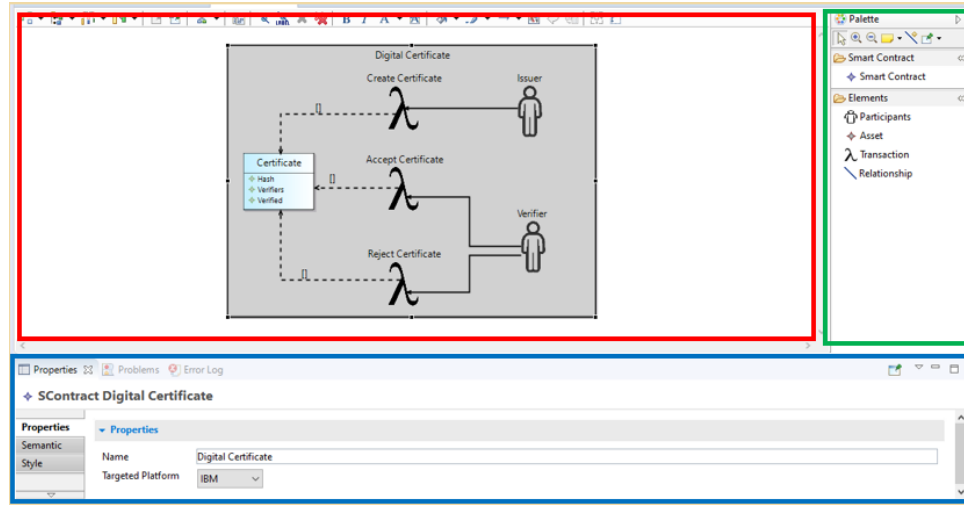
- an asset must have Type property.
- an asset must show the name and the type of any extra parameters.
- a relationship type must be valid and relate to an existing object (participant, asset, or transaction).
- a contract has one and only one owner
- a participant can be connected only to a transaction using *TranRel* relationship.
- a participant cannot be directly connected to other participants.
- a transaction can only be connected to an asset through an *AssRel* relationship.

The validation rules were implemented using the Acceleo Query Language (AQL). AQL is a language similar to Object Constraint

²www.obeodesigner.com

Table 2: Mapping the Reference Model Concepts to the Blockchain Platforms' Constructs

Concept	Azure Blockchain Workbench	Hyperledger Composer	Ethereum
Participants	ApplicationRoles construct defines participant with "Name" and "Description" (defined in .json file)	Participant construct identified with unique name and parameters to define its characteristics (defined in .cto file)	Participants are defined as a State Variables or Struct Type (defined in .sol file)
Assets	Workflow construct defines an asset (defined in .json file)	Asset concept identified with unique name (defined in .cto file)	Asset is defined as a State Variables (defined in .sol file)
Transactions	Function construct that work as the logic of transactions when creating or changing the smart contract (defined in .json file, while logic is in .sol file)	Transaction construct which identified with a unique name (defined in .cto file, while logic is in .js file)	Function construct which has unique name, return type and parameters (defined in .sol file)
Conditions	Transitions construct outline the actions available for the user of the smart contract to use (defined in .json file)	The conditions are defined in access policy file (.acl) which is not covered in .cto model	Modifier construct can be used to have condition before executing transaction (defined in .sol file)
Relationship(TranRel)	does not have relationship between transaction and participant	Relationship construct can define relation between transaction and participant (defined in .cto file)	Modifier construct can be used to define which user can execute function (defined in .sol file)
Relationship(AssetRel)	Workflow construct which has Function attribute to establish a relation between function and asset (defined in .json file)	Relationship construct can define relation between transaction and asset (defined in .cto file)	The relation between asset and transaction is realized by statements in the function construct (defined in .sol file)

**Figure 2: The iContractML Model Editor.**

Id*: Type: Label:

Label Expression:

Help Expression:

Is Enabled Expression:

Value Expression:

Candidates Expression:

Candidate Display Expression:

Number Of Columns:

Figure 3: Example of a Validation Rule Implementation (Asset Type Rule)

Language (OCL). Figure 3 shows an example of implementing the constraint "an asset must have a type value".

4.3 Transformation Template

To generate the deployment configurations and the code structure according to the target blockchain platform, we implemented

transformation templates for each of the corresponding blockchain platforms using the Accelele³ template language.

The transformation template takes the instance model generated using the iContractML model editor and applies a set of template transformations to generate a target model that conforms to the syntax of the target platform.

The code snippet in Table 3 shows a sample transformation template code to transform the model into code for the Ethereum platform. The transformation template code for all the blockchain platforms can be found in the project repository⁴.



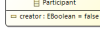



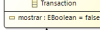

5 EVALUATION

In this paper, the evaluation strategy is to exploit three different use cases to assess the proposed DSML and evaluate if the generated smart contracts artifacts (i.e., configuration files) will be enough to deploy it on the targeted blockchain platform.

³<https://www.eclipse.org/accelele/>

⁴https://drive.google.com/open?id=1rkagnH5x6vbcyL865X5sIM-siLNN4_N8

Table 3: Transformation Template Table

Concept	Abstract Syntax	Concrete Syntax	Transformation Code Snippet (Ethereum)
Smart Contract / File			<pre>if (con.targetPlatform.toString() != 'Ethereum') [file (con.scName.replaceAll(' ', '_') + '.txt', false, 'UTF-8')] Contract [con.scName.replaceAll(' ', '_')]</pre>
Participants			<pre>for (par : Participant con.eAllContents("Participant")->asOrderedSet()) address public [par.name.replaceAll(' ', '_').toLowerCaseFirst()] Address [/for]</pre>
Asset			<pre>for (as : Asset con.eAllContents("Asset")->asOrderedSet()) for (par : Parameter as.eAllContents("Parameter")->asOrderedSet()) [par.type + ' public ' + par.name.replaceAll(' ', '_')] [/for]</pre>
Transaction			<pre>for (func : Transaction con.eAllContents("Transaction")) function [func.name/]() {for (tranrel : TranRel con.eAllContents("TranRel")) for (precond : Pre func.eAllContents("Pre")) require // [precond.name/] --- [precond.condition/] [/for]</pre>

5.1 Use Cases

Using iContractML, we have created models and generated the deployment artifacts for the following three use cases. The models and generated code, in addition, to sample videos are made available on the temporary project repository⁴ and will be published on GitHub by the time of publication. The models are available for artifact evaluation.

- (1) **Grading reward system:** A smart contract used to reward a student if he/she gets a grade above or equal to the desired grade. In this case, we have three participants: The parents, the student, and the teachers. The smart contract is created by the parents to reward their child. The teacher has the action of reflecting the grade obtained in a test (by the student) on to the smart contract. The student has the withdrawing action of the reward amount once the condition is satisfied. For this example, we reflected the following concepts: Three participants, three transactions, two notification events, and one asset with two parameters.
- (2) **Digital certificate:** A smart contract that represents an original certificate given by an educational institution that signifies the approval at the end of a course studied by a student. The student has access to the blockchain to check it's own "diploma"/"certificate". For the smart contract to be accepted on the blockchain it has to pass through an approval system by a set of verifiers. In this case, we have two different participants. The first being the *issuer* of the certificate and the second is a list of Verifiers that can either accept or reject the created certificate. For this use case, we reflected the following concepts: Two participants, three transactions, and an asset with three parameters.
- (3) **Doctors treatment bill:** On this particular example we have two smart contracts. The first smart contract is created by a doctor "sent" to the insurer of its patient for it to pay the bill of the provided treatment. The second smart contract is created by the insurer and consists of bill payment for the patient, which is a monthly payment given to the user for the services provided by the insurer. For this example, we have the following concepts: Two smart contracts, two participants per contract, two transactions per contract based on their respecting participants, and one asset per contract.

After defining and creating the graphical representation of our use cases, we execute the transformation code to generate the smart contract code of each of the three platforms for every use case. Moreover, we have deployed and run these codes on their corresponding blockchain platform, without any modification. As a result, all the deployed codes did not show any syntax or structure error.

A sample of the generated smart contract codes for the second use case in different platforms is shown in Figure 4. Due to the page limit, the graphical representation and the generated smart contract code of the different use cases are omitted from the paper, however, they are fully demonstrated in the project repository⁴ including videos that shows a live demo. The files of the graphical representation for the use cases are ready for artifact evaluation and can be found in the folder "Use Cases".

5.2 Quantitative Analysis of the Abstract Syntax

To evaluate the fit of the reference model, to the targeted blockchain platforms, we have done quantitative analysis based on the language evaluation framework introduced by Guizzardi et al. [11]. The framework introduces four properties: lucidity, soundness, lacticity, and completeness. The evaluation is done on the mapping between the core concepts of the reference model and the constructs in the targeted platforms meta-models, as illustrated in Table 2. In the mapping, we have considered the core concepts of the reference model and excluded the abstract concept like Parameters, Elements, and Events.

We define each property for a reference model M and a blockchain platform P . The model M contains a set of model concepts $m \in M$. A platform P contains a set of platform constructs $p \in P$.

The four metrics are measured as follow:

- **Lucidity:** a model concept m is lucid iff it represents a maximum of one platform construct p . The lucidity of model M with respect to a platform P is measured by the percentage of model concepts m that are lucid. A low lucidity value indicates that the reference model has construct overload, which makes the reference ambiguous and has the potential of making errors.
- **Soundness:** a model concept m is sound iff it represents a minimum of one platform construct p . The soundness of

Ethereum	Microsoft Azure	Hyperledger Composer
<pre>pragma solidity >=0.4.22 <0.7.0; contract Digital_Certificate{ struct Verifier{ bytes32 name; } bytes32 public Hash; Verifier[] public Verifiers; bool public Verified; address public issuerAddress; address public verifierAddress; modifier onlyIssuer{ require(msg.sender == issuerAddress); } _;</pre>	<pre>{ "ApplicationName": "DigitalCertificate", "DisplayName": "Digital Certificate", "Description": "...", "ApplicationRoles": [{ "Name": "Issuer", "Description": "..." }, { "Name": "Verifier", "Description": "..." }], "Workflows": [{</pre>	<pre>namespace Digital_Certificate asset Certificate identified by Hash{ o String Hash o Verifier[] Verifiers o Boolean Verified } participant Issuer identified by name{ o String name } participant Verifier identified by name{ o String name } transaction Create_Certificate{ --> Issuer issuer --> Certificate certificate }</pre>

Figure 4: Sample of the Generated Smart Contract Code for the Digital Certificate Use Case

Table 4: iContractML Abstract Syntax Quantitative Analysis

Metrics	Azure Blockchain	Ethereum	Hyperledger Composer	All Platforms
Lucidity	6/6 (100%)	6/6 (100%)	6/6 (100%)	18/18 (100%)
Soundness	5/6 (83.3%)	6/6 (100%)	5/6 (83.3%)	16/18 (88.9%)
Laconicity	3/4 (75%)	2/4 (50%)	3/4 (75%)	8/12 (66.67%)
Completeness	4/4 (100%)	4/4 (100%)	4/4 (100%)	12/12 (100%)

model M with respect to a platform P is measured by the percentage of model concepts m that are sound. A low value of soundness shows that the reference model has construct excess, and indicates that the reference model is complicated with unnecessary model concepts that do not represent any platform construct.



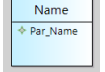


- **Laconicity:** a platform construct p is laconic iff it represents a maximum of one model concept m . The laconicity of platform P with respect to a model M is measured by the percentage of platform construct p that is laconic. A low level of laconicity indicates the reference model has construct redundancy, where one construct platform can be represented with multiple model concepts.
- **Completeness:** a platform construct p is complete iff it represents a minimum of one model concept m . The completeness of platform P with respect to a model M is measured by the percentage of platform construct p that is complete. A low value of completeness indicates that the mapping between the reference model and the targeted platform constructs is not complete.

Table 4 demonstrates the result of the four metrics measurement.

5.3 Quantitative Analysis of the Concrete Syntax

We adopt the framework proposed by Moody et al. [19] and Guizzardi et al. [10] to measure the semiotic clarity of the implemented

Table 5: Reference Model and Graphical Tool Symbols Mapping

Model	Graphical Symbol
Participant	
Transaction	
Asset	
Relationship(TranRel)	
Relationship(AssetRel)	
Condition	-

graphical tool (iContractML Concrete Syntax), shown in Table 5. The semiotic clarity is composed of symbol overload, redundancy, excess, deficit. The metrics are measured as follow:

- **Symbol redundancy:** a graphical symbol is redundant if a model concept m is represented by more than one graphical symbol. The Symbol redundancy is measured by the percentage of redundant symbols. High symbol redundancy indicates that the graphical tool is not clear, as there are multiple symbol choices for the same concept, which will confuse the user.
- **Symbol overload:** a graphical symbol is overloaded if it represents more than one model concept m . The Symbol overload is measured by the percentage of overloaded symbols. High symbol overload indicates that the graphical tool is ambiguous and has the possibility of misinterpretation.

Table 6: iContractML Concrete Syntax Quantitative Analysis

Metrics	Metric Result
Symbol redundancy	0%
Symbol overload	0%
Symbol excess	0%
Symbol deficit	1/6 (16.67%)

- Symbol excess: a graphical symbol is excessive if it does not represent any model concept m . The Symbol excess is measured by the percentage of overloaded symbols. High symbol excess shows that the graphical tool is complicated.
- Symbol deficit: a model concept m is deficit if it is not represented by any graphical symbol. The Symbol deficit is measured by the percentage of deficit model concept m . A high symbol deficit indicates that the graphical tool is incomplete.

Table 6 illustrates the result of the evaluation of the iContractML concrete syntax.

5.4 Discussion

The results of the quantitative analysis for iContractML show that in terms of granularity the reference model is 100% lucid and 66.67% laconic. This lucidity value indicates that the reference model does not have any ambiguous concepts. On the other hand, the laconicity value shows that there are redundant concepts that correspond to the same platform construct like Asset and Relationship (AssetRel) concepts that correspond to Workflow construct in Azure Blockchain Workbench. Although these concepts are corresponding to the same construct, they are targeting different attributes in the platform construct. In terms of coverage, the results show that the iContractML is 88.9% sound and 100% complete. This indicates that the model is complete and covers the core platform constructs.

The quantitative analysis results in Table 6 demonstrate that the graphical tool does not have any redundant, excess, and ambiguous symbols. There is only one missing graphical symbol for the condition concept which resulted in a 16.67% symbol deficit. The condition concept is represented in the graphical tool as a text box when defining a transaction concept.

While we evaluated iContractML using a number of use cases to show the ability of the language to generate valid models and artifacts, a usability study and qualitative analysis is still missing and will be considered in future work. Moreover, the current language supports generating the structural part of smart contracts. In the future, we are planning to extend the framework to support modeling the smart contract behavior and generate the target code accordingly.

6 RELATED WORK

In this section, we have done a comprehensive study on the related work of creating DSLs to develop smart contracts. Marlow [13] introduced a DSL that uses Haskell to represent the smart contracts which are deployed on the Cardano platform. The proposed solution also comes with a graphical tool call Meadow created based of Blockly.

Pyramid Scheme [1] tool targets specifically the Ethereum Virtual Machine (EVM). It's a plain text tool with an abstract representation of the components of an EVM written on Racket. This tool parses the plain text, compiles it, and generates a deployable EVM bytecode.

Attributes Deontic AI Conditions (ADICO)[8] is a DSL created with Scala that maps domain-specific constructs of the smart contracts to simpler concepts. Moreover, the tool creates smart contracts for EVMs.

Barclays' Smart Contract Template [23] is a full-fledged program that allows the user to create, save, delete smart contracts (if not in the blockchain). The tool works based on the language CLACK [5] and the smart contracts are deployed on the platform Corda.

FSolidM [16, 17] is a DSL for designing secure Ethereum smart contracts. It's a graphical used to create a state machine representation of the smart contract.

Combinatorial library [14] is a DSL that uses a syntactical library implemented on Haskell to represent the structure of smart contracts. Even though it does not translate directly to any smart contract platform, it has a good insight on how to represent and generalize the smart contracts.

Another approach uses UML statechart [9] to represent smart contracts that are going to be utilized as a way of controlling the usage and implementation of Cyber-Physical Systems (CPS) elements. Every part of the UML statechart is map to a Solidity piece of code that allows the generation of code from the UML diagram into a full functioning smart contract that can be published into Ethereum.

VeriSolid [18] is a DSL tool created base on WebGML, to use as a means of graphical representation for the user and the aforementioned FSolidM to represent the smart contract. The main objective of this DSL is to check the security and correctness of the created smart contracts. The smart contract generated by the tool can be deployed into the Ethereum platform.

Findel [2] is a declarative financial DSL that maintains its syntax to generate Findel contracts. The idea of the Findel contracts is that one can deploy them onto any platform desired by the user, given the appropriate programming required to function.

DAML [7] is a language used to create smart contracts between multi-party business processes. The contracts generated using DAML can be deployed on to the Digital Asset Ledger. The main idea of the language is that can be used by lawyers alongside developers to digitalize and automate legal contracts. Ergo project [20] is a domain-specific textual language that targets to generate the structural and behavioral code of legal smart contracts and can run on different blockchain platforms such as Corda and Hyperledger Fabric.

SPESC [12] is a specification language for smart contracts, that can represent the general structure of a smart contract to create collaborative design.

Table 7 summarize the information above and shows the important elements when thinking about designing a Model-Driven Engineering (MDE) tool.

Few existing DSLs such as FSolidM and Ergo can generate both the structural and the behavioral code of the smart contract, which is an advantage compared to iContractML that is currently limited to the structural code. However, these DSLs are either targeting

Table 7: DSL Table Comparison

DSL	Graphical / Textual Representation	Code Generation	Targeted Language	Targeted Platform
Marlow	Graphical	Yes	Solidity or Plutus	Cardano Blockchain
Pyramid Scheme	Textual	Yes	Solidity	EVM
ADICO	Textual	Yes	Solidity	EVM
Barclay's SCT	Graphical	-	-	Corda
FSolidM	Graphical	Yes	Solidity	Ethereum
Combinatorial Library	Textual	No	Does not have	Does not have
UML State-chart	Graphical	Yes	Solidity	Ethereum
VeriSolid	Graphical	Yes	Solidity	Ethereum
Findel	Textual	No	Does not have	Does not have
DAML	Textual	No	Does not have	Digital Asset Ledger
Ergo	Textual	Yes	Multi*	Multi*
SPESC	Textual	No	Does not have	Does not have
iContractML	Graphical	Yes	Multi*	Multi*

one blockchain platform or only supporting textual representation. On the other hand iContractML which is a graphical tool that is independent of the blockchain platform and supports multiple programming languages for different blockchain platforms. Moreover, iContractML is not limited to a certain use case (such as Findel) and can be used in different use cases as shown in Section 5.1.

7 CONCLUSION

In this paper, we introduced iContractML: a DSML for modeling and deploying smart contracts onto multiple blockchains platforms. iContractML was realized based on comprehensive domain analysis for three smart contract development platforms, through which we defined a reference model for smart contracts.

iContractML provides a model editor and code generator that utilizes template-based transformation to support model-once-deploy-anywhere approach. The abstract syntax of iContractML was evaluated through mapping the metamodel concepts to different platform constructs then evaluating the lucidity, soundness, laconicity, and completeness of the metamodel. Moreover, the concrete syntax was evaluated by measuring its semiotic clarity.

To show the correctness and validity of the generated deployment model files, we used iContractML to model three use cases and generated the target deployment artifacts for three different platforms. The artifacts were accepted by all of the targeted platforms for each of the three different use cases, indicating that the deployment models generated are syntactically correct.

Even though our tool can generate the configuration files, these are not enough for a full deployment of a functional smart contract. This is because, currently, iContractML is only able to specify the structure and not the behavior of a smart contract. However, iContractML is still an advancement over existing DSMLs for smart contracts in the literature that targets one platform at a time.

Immediate future directions consist of conducting an empirical study to evaluate the usability of iContractML. Moreover, we are

planning to extend the language to support modeling the behavior of smart contracts in order to support a multi-platform full deployment experience.

REFERENCES

- [1] Gaurav Agrawal. 2018. DSLs for Ethereum Contracts. Retrieved May 2, 2020 from <https://medium.com/coinmonks/dsls-for-ethereum-contracts-380136177abd>
- [2] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure derivative contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 453–467.
- [3] Azure blockchain. 2016. Retrieved May 2, 2020 from <https://azure.microsoft.com/en-us/solutions/blockchain/>
- [4] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).
- [5] Christopher D Clack, Vikram A Bakshi, and Lee Braine. 2016. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771* (2016).
- [6] Hyperledger Composer. 2018. Hyperledger Composer Documentation. *Linux Foundation* (2018).
- [7] DAML SDK Documentation. 2019. Retrieved May 2, 2020 from <https://docs.daml.com/index.html>
- [8] Christopher K Frantz and Mariusz Nowostawski. 2016. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 210–215.
- [9] Péter Garamvölgyi, Imre Kocsis, Benjámín Gehl, and Attila Klenik. 2018. Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 134–139.
- [10] Giancarlo Guizzardi. 2013. Ontology-based evaluation and design of visual conceptual modeling languages. In *Domain engineering*. Springer, 317–347.
- [11] Giancarlo Guizzardi, Luis Ferreira Pires, and Marten Van Sinderen. 2005. An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 691–705.
- [12] Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. 2018. Spesc: A specification language for smart contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 132–137.
- [13] IOHK. 2018. *Marlowe: Financial contracts on blockchain*. Retrieved May 2, 2020 from <https://iohk.io/blog/marlowe-financial-contracts-on-blockchain/>
- [14] S Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering. *ACM SIG-PLAN Notices* 35, 9 (2000), 280–292.
- [15] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
- [16] Anastasia Mavridou and Aron Laszka. 2018. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*. Springer, 523–540.
- [17] Anastasia Mavridou and Aron Laszka. 2018. Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 270–277.
- [18] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-design smart contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 446–465.
- [19] Daniel Moody and Jos van Hilleghersberg. 2008. Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In *International Conference on Software Language Engineering*. Springer, 16–34.
- [20] Ergo Project. [n.d.]. Retrieved August 2, 2020 from <https://accordproject.org/projects/ergo/>
- [21] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. 2012. Bottom-up meta-modelling: An interactive approach. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 3–19.
- [22] Zeshun Shi, Huan Zhou, Yang Hu, Surbiryala Jayachander, Cees de Laat, and Zhiming Zhao. 2019. Operating Permissioned Blockchain in Clouds: A Performance Study of Hyperledger Sawtooth. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPD)*. IEEE, 50–57.
- [23] International Business Times. 2016. Barclays' Smart Contract Templates stars in first ever public demo of R3's Corda platform. Retrieved May 2, 2020 from <https://www.ibtimes.co.uk/barclays-smart-contract-templates-heralds-first-ever-public-demo-r3s-corda-platform-1555329>
- [24] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
- [25] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* (2019).